Outils logiques et algorithmiques

Thibaut Balabonski @ Université Paris-Saclay
Édition 2024.

Dans ce cours, on étudie des algorithmes et des structures de données, ainsi que les techniques qui permettent de raisonner sur les algorithmes.

Du code et des preuves. Chaque semaine, nous étudierons au moins un nouvel algorithme, c'est-à-dire une méthode permettant de résoudre automatiquement un problème donné. On présentera tous les algorithmes étudiés sous la forme de programmes, écrits le plus souvent en java, et dans certains chapitres en caml. Il sera ainsi possible de les tester, de suivre pas à pas leur exécution, et de raisonner avec précision.

Nous verrons également chaque semaine des outils mathématiques qui permettent de décrire un problème à résoudre pour ensuite concevoir un algorithme associé, ou analyser un algorithme pour justifier que celui-ci résoud correctement le problème posé ou pour prédire son temps d'exécution.

Le cours mélange donc des aspects mathématiques et algorithmiques, et vise à la fois à vous faire découvrir des structures de données et algorithmes fondamentaux en informatique et à affûter vos capacités de raisonnement.

Plan. Le cours est séparé en trois parties, correspondant à trois grandes formes d'organisation des données sur lesquelles nous travaillerons.

- **I. Tableaux.** Techniques de base pour justifier le bon fonctionnement d'un algorithme et analyser son coût d'exécution. Algorithmes de recherche et de tri. Objectifs : savoir décrire précisément un problème à résoudre, suivre pas à pas l'exécution d'un algorithme, expliquer le fonctionnement d'un algorithme et justifier qu'il répond au problème posé, estimer le coût d'un algorithme.
- **II. Graphes.** Modélisation d'un problème à résoudre sous la forme d'un graphe. Relations binaires, relations d'ordre, relations d'équivalence. Algorithmes d'affectation de ressources, d'ordonnancement, d'exploration, de classification. Objectifs : abstraire un problème concret et le modéliser par une question sur des graphes, appliquer ou analyser un algorithme de graphes, déduire de la solution abstraite sur les graphes une solution concrète au problème d'origine, justifier qu'un algorithme termine, raisonner rigoureusement sur les propriétés structurelles d'un graphe.
- **III. Arbres.** Structures de données hiérarchiques ou arborescentes. Arbres de recherche, files de priorité, manipulation d'objets syntaxiques, récurrence structurelle. Objectifs : représenter et manipuler des structures de données intrinsèquement récursives, programmer des fonctions récursives, raisonner par récurrence.

Organisation pédagogique. Votre apprentissage nécessite trois choses.

1. Acquérir des informations brutes. C'est facile, il suffit d'avoir les bons documents sous la main, et de les lire. Ce poly a été écrit justement pour cela.
2. Comprendre ces nouvelles informations et les intégrer à l'ensemble de vos connaissances. C'est plus délicat, cela demande un peu de temps, et l'interaction avec les enseignants peut être une aide déterminante.
3. Vous entraîner à mettre vos nouvelles connaissances en pratique. C'est en particulier le rôle des séances de TD/TP, mais aussi du travail à la maison après les séances.

Nous allons mettre en place le rituel hebdomadaire suivant, pour tirer le meilleur parti des séances partagées avec les enseignants et pour que les cours « magistraux » se concentrent sur le point numéro 2. Chaque semaine, avant le cours, vous lirez un chapitre du poly et répondrez à un (tout petit) questionnaire en ligne. Toujours avant le cours, j'analyserai vos réponses au questionnaire pour adapter le contenu de la séance. Pendant le cours lui-même, je me concentrerai sur les questions et difficultés révélées par vos réponses au questionnaire. Après cela, vous devriez arriver en séance de TD prêts à passer directement à l'étape suivante : la pratique.

Vos objectifs de lecture hebdomadaire. Nous travaillerons chaque semaine sur un nouveau chapitre. La partie à lire sera systématiquement « l'ensemble des sections du chapitre dont le titre ne commence pas par "approfondissement" ». Le temps de référence à consacrer à cette lecture chaque semaine est environ une heure. Le premier objectif de cette lecture est de vous familiariser avec les concepts de la semaine, et d'intégrer leurs définitions, ainsi que de voir dans les grandes lignes le fonctionnement et les propriétés de l'algorithme étudié. Vous n'avez pas besoin de tout comprendre en détail à ce stade.

Notre premier contrat est le suivant : si vous avez effectivement lu les parties demandées avant la séance, alors vous pourrez suivre l'essentiel du cours sans être largués. Vous pouvez lire en plus les parties d'approfondissement ou non, en fonction de votre temps et de votre intérêt.

Le bon usage du questionnaire hebdomadaire. Le questionnaire accompagne votre lecture, et est inclus dans l'heure à passer avant chaque séance. Il est systématiquement composé de deux parties.

— Un petit nombre de questions à choix multiples, qui testent une compréhension basique des définitions du chapitre. Votre performance à ce QCM n'est pas notée : c'est pour vous l'occasion de vous assurer que vous avez correctement lu, et pour moi un moyen de détecter quels aspects ont été bien ou mal interprétés.

— Deux questions en texte libre, vous demandant d'expliquer deux aspects de votre choix du chapitre : un que vous jugez important ou intéressant, et un qui représente une difficulté. Il n'y a pas de « bonne réponse » à ces questions : ma principale mesure de qualité est que vous ayez rédigé quelques lignes (par exemple, deux phrases) avec vos mots à vous. Si vous hésitez entre plusieurs points à évoquer, vous pouvez les mentionner tous, ou en sélectionner un seul selon les critères de votre choix.

Important : vous pouvez vous servir du texte libre, et en particulier celui dédié aux difficultés, pour formuler des questions directes auxquelles je m'efforcerai de répondre pendant le cours. De manière générale, je me sers des réponses aux questionnaires pour adapter le contenu de chaque séance.

Notre deuxième contrat est le suivant : chaque semaine, une partie de la séance de cours sera dédiée aux principales difficultés et demandes exprimées via le questionnaire. Note : il faut répondre au plus tard la veille de la séance pour me permettre de faire cela. Le temps d'une séance ne permettant pas toujours de répondre à toutes les questions, je maintiendrai également à jour une foire aux questions avec quelques réponses additionnelles.

Participation et contrôle continu. Une partie de votre note de contrôle continu sera déterminée par le fait que vous participiez de manière honnête et régulière aux questionnaires hebdomadaires. Interprétation de « régulière » : répondre à 9 questionnaires sur 10 est parfait. Répondre à seulement la moitié n'est pas régulier. Interprétation de « honnête » : vos réponses libres contiennent effectivement une ou deux phrases, qui sont personnelles et en rapport avec le chapitre.

Bilan sur l'organisation. En vous astreignant à la lecture et au questionnaire hebdomadaires, vous vous assurez une bonne répartition du travail sur le semestre, une résolution précoce d'un certain nombre de difficultés qui pourraient sinon entraver la suite de votre semestre, et de manière générale vous profitez de séances de cours plus agréables à suivre et mieux adaptées à votre progression.

Table des matières

Outils logiques et algorithmiques

Thibaut Balabonski @ Université Paris-Saclay
Édition 2024.

Première partie

Tableaux

Cette partie est dédiée aux techniques de bases permettant de justifier le bon fonctionnement d'un algorithme et d'analyser le coût de son exécution. On prendra comme exemples des algorithmes manipulant des tableaux, qui seront systématiquement exprimés en langage Java.

1 Chercher

1.1 Panorama : recherche dichotomique

Problème : chercher un élément `x` dans un tableau `tab`. On veut renvoyer :

- un indice `i` tel que `tab[i]` contient `x`, si `x` apparaît dans le tableau,
- la valeur spéciale `-1`, si `x` n'apparaît pas dans le tableau.

Solution simple : recherche séquentielle. On énumère toutes les cases du tableau, on s'arrête si on trouve l'élément cherché, et on renvoie `-1` si on a parcouru tout le tableau sans trouver l'élément. En java :

```
static int sequentialSearch(int x, int[] tab) {
    for (int i=0; i < tab.length; i++) {
        if (tab[i] == x) return i;
    }
    return -1;
}
```

Cet algorithme est simple, et répond sans aucun doute au problème posé.

Solution alternative, lorsque l'on sait que les éléments du tableau sont triés en ordre croissant : recherche dichotomique. On définit un intervalle de recherche $[\texttt{lo}, \texttt{hi}[$ (indice `lo` inclus et indice `hi` exclu). L'intervalle couvre à l'origine tout le tableau et devient progressivement plus petit. On s'arrête lorsque l'on trouve l'élément ou lorsque l'intervalle de recherche devient vide. À chaque étape on considère la case `mid` du milieu de l'intervalle (arrondi vers le bas) : si l'élément cherché est plus petit on poursuit la recherche dans la moitié gauche $[\texttt{lo}, \texttt{mid}[$, et s'il est plus grand on poursuit dans la moitié droite $[\texttt{mid} + 1, \texttt{hi}[$. Dans tous les cas `mid` est exclu du nouvel intervalle. En java :

```
static int binarySearch(int x, int[] tab) {
    int lo = 0;
    int hi = tab.length;
    while (lo < hi) {
        int mid = lo + (hi-lo)/2;
        if (tab[mid] == x) return mid;
        if (x < tab[mid]) { hi = mid; }
        else              { lo = mid+1; }
    }
    return -1;
}
```

L'algorithme est nettement plus subtil. Est-on bien certain qu'il fonctionne ? En quoi est-il mieux que le précédent ?

Comparaison empirique. On peut tester les deux fonctions sur des tableaux aléatoires et constater qu'elles donnent les mêmes résultats : elles sont vraisemblablement correctes. Pour aller plus loin : comparaison des temps d'exécution sur des tableaux de différentes

tailles. Temps moyens pour 100 recherches aléatoires dans des tableaux d'une taille donnée, en micro-secondes :

Taille	`sequentialSearch`	`binarySearch`
10	0,9	1,0
20	1,2	1,1
50	2,2	1,4
100	4,8	1,8
1 000	38	2,5
10 000	352	2,9
1 000 000	21 000	7,1
100 000 000	2 100 000	18

Les performances des deux fonctions sont radicalement différentes. On doit pouvoir expliquer ce phénomène.

Analyse de complexité de `sequentialSearch`. Deux scénarios.

1. Si `x` est dans `tab`, on énumère tous les éléments jusqu'à la première occurrence de `x`. Si toutes les positions sont équiprobables pour cette occurrence on consulte en moyenne la moitié des éléments du tableau.
2. Si `x` n'est pas dans `tab`, on consulte tous les éléments du tableau.

Dans tous les cas on s'attend à consulter un nombre d'éléments proportionnel à la taille du tableau.

Analyse de complexité de `binarySearch`. La boucle **`while`** rend le comportement plus difficile à prédire.

Première étape : vérifier que l'algorithme progresse et finit toujours par s'arrêter. Argument : l'intervalle de recherche [`lo`, `hi` [devient strictement plus petit à chaque étape, tôt ou tard il devient donc vide et l'algorithme s'arrête.

Deuxième étape : mesurer l'évolution de [`lo`, `hi` [pour prédire le nombre d'étapes maximal avant arrêt. Observation : la longueur de l'intervalle est environ divisée par deux à chaque étape. Propriété plus précise : si $0 \leq \texttt{hi} - \texttt{lo} < 2^k$, alors l'algorithme s'arrête après k tours de boucle au maximum. Démonstration par récurrence sur k.

— Si $0 \leq \texttt{hi} - \texttt{lo} < 2^0 = 1$, alors `hi = lo`. La condition de la boucle est invalide, le programme s'arrête.

— Soit k tel que si $0 \leq \texttt{hi} - \texttt{lo} < 2^k$, alors l'algorithme s'arrête après k tours de boucle au maximum (hypothèse de récurrence).

Supposons $0 \leq \texttt{hi} - \texttt{lo} < 2^{k+1}$. Si $\texttt{hi} - \texttt{lo} < 2^k$ on conclut par hypothèse de récurrence. Sinon, en particulier `lo < hi`. On fait un premier tour de boucle et on calcule un indice `mid` tel que $0 \leq \texttt{mid} - \texttt{lo} < 2^k$ et $0 \leq \texttt{hi} - (\texttt{mid} + 1) < 2^k$. Puis :

— soit le programme renvoie `mid`, d'où arrêt,

— soit le programme poursuit après avoir modifié `hi` en `mid` : par hypothèse de récurrence le programme réalise au maximum k tours de boucle supplémentaires,

— soit le programme poursuit après avoir modifié `lo` en `mid + 1` : de même, par hypothèse de récurrence le programme réalise au maximum k tours de boucle supplémentaires.

Dans tous les cas : au maximum $k + 1$ tours de boucle au total.

Conclusion : pour tout tableau de taille strictement inférieure à 2^k, la recherche d'un élément utilise au maximum k tours de boucle. Autrement dit, la recherche dichotomique teste la présence ou l'absence d'un élément dans un tableau trié de taille N en consultant seulement $\log_2(N)$ éléments du tableau environ. Cela explique la différence de comportement observée avec la recherche séquentielle.

En supposant que vous êtes sûr de vous, comment convaincre un camarade sceptique ?

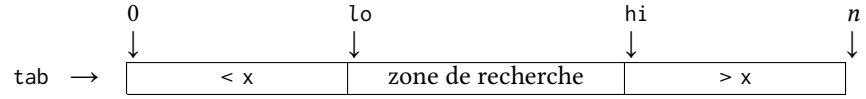
Correction de la recherche dichotomique. Justifions que la recherche dichotomique fonctionne à coup sûr, quand bien même elle ne consulte que très peu d'éléments du tableau.

Cas simple : lorsque l'on renvoie un indice de tableau avec la ligne

```
if (tab[mid] == x) return mid;
```

on vient bien de tester que la case d'indice `mid` du tableau `tab` contient précisément l'élément `x` cherché. La réponse est correcte à coup sûr.

Cas compliqué : lorsque l'on renvoie `-1` il faut justifier que l'on n'a pas pu rater l'élément cherché. Fait clé : l'élément cherché `x` ne peut jamais se trouver en dehors de l'intervalle de recherche [`lo`, `hi` [, car tous les éléments de `tab` [0, `lo` [sont strictement inférieurs à `x`, et tous les éléments de `tab` [`hi`, n [lui sont strictement supérieurs. Autrement dit : au cas où l'élément serait dans le tableau, il ne pourrait être qu'à l'intérieur de l'intervalle de recherche. Or, cet intervalle est vide lorsque l'on renvoie `-1` : l'élément ne peut pas s'y trouver. On peut résumer cet argument par le schéma suivant, qui détaille la signification de chaque variable, et les propriétés connues du tableau.



Reste à démontrer que le « fait clé » est bien toujours valide. Le procédé est similaire à celui d'une récurrence.

— À l'initialisation l'intervalle [`lo`, `hi` [couvre tous les indices du tableau : `x` ne peut assurément pas se trouver en dehors.

— Supposons qu'au début d'un tour de boucle, `lo` soit tel que tout élément `tab[`i`]` du tableau d'indice i < `lo` soit strictement inférieur à `x`, et que `hi` soit à l'inverse tel que tout élément `tab[`i`]` du tableau d'indice $i \geq$ `hi` soit strictement supérieur à `x`. On a trois cas possibles.

  1. Si le programme s'arrête en renvoyant `mid`, il n'y a rien à vérifier.
  2. Si `x < tab[mid]`, alors `hi` devient `mid`. Les mêmes éléments restent à gauche de `lo` : ils sont toujours strictement inférieurs à `x`. Les éléments à droite du nouveau `hi` sont les éléments à droite de `mid`. Le tableau étant trié, tout tel élément `tab[`i`]` vérifie `tab[mid]` ≤ `tab[`i`]`. Comme `x < tab[mid]`, ces éléments sont bien tous strictement supérieurs à `x`.
  3. Si `x > tab[mid]`, alors `lo` devient `mid + 1` et on conclut avec un raisonnement symétrique au précédent.

Ainsi, notre fait clé est vrai à l'initialisation, puis préservé par chaque nouveau tour de boucle : il reste vrai jusqu'à la fin de l'exécution du programme. En particulier, si la boucle s'arrête du fait de l'invalidation du test `lo < hi`, c'est-à-dire si on arrive à une situation où `lo ≥ hi`, alors les segments `tab` [0, `lo` [et `tab` [`hi`, n [couvrent tout le tableau, qui ne peut contenir `x`.

Preuve de sûreté de `binarySearch`. On a justifié que `binarySearch` :

— finit toujours par s'arrêter,

— consulte un nombre d'éléments au plus logarithmique en la taille du tableau,

— ne renvoie que des résultats corrects.

Il reste un angle mort dans cette analyse : le scénario où le programme s'interromprait à cause d'une erreur. En l'occurrence le programme manipule un tableau : on a un risque d'échec par une tentative d'accès en dehors des bornes du tableau.

Dans `binarySearch`, les seuls accès au tableau se font dans la boucle avec `tab[mid]` : il faut justifier que `mid` est toujours tel que 0 ≤ `mid` < `tab.length`. Pour cela on démontre d'une part que `lo` et `hi` sont toujours tels que 0 ≤ `lo` ≤ `hi` ≤ `tab.length`, et d'autre part que si `lo` < `hi`, alors le calcul définissant `mid` assure que `lo` ≤ `mid` < `hi`.

Avec cette dernière étape on garantit donc que notre programme `binarySearch` s'exécute toujours sans erreur et produit en un temps fini (et même en un temps très court) un résultat correct. Autrement dit, `binarySearch` est une solution garantie sûre et efficace au problème de la recherche d'un élément dans un tableau trié.

Dans ce cours nous allons découvrir de nombreux algorithmes ou structures de données répondant à des problèmes variés, ainsi que les outils qui permettent de raisonner sur ces algorithmes pour assurer qu'ils répondent bien au problème posé et évaluer leur efficacité.

1.2 Spécification d'un problème algorithmique

Un algorithme répond à un problème : étant données certaines entrées, produire un certain résultat ou effet. Avant même la conception d'un algorithme, il faut énoncer clairement le problème posé.

La spécification d'un problème comporte deux parties :

— description des contraintes que doivent vérifier les entrées (*préconditions*),

— description du résultat attendu.

Exemple pour l'exponentiation. On veut calculer la n-ème puissance d'un nombre a.

— Condition : n doit être un entier positif ou nul.

— Le résultat de `power`(a, n) doit être a^n.

La spécification du résultat se ramène à un unique prédicat : `power`$(a, n) = a^n$, de même que pour la précondition : $n \geq 0$ (s'il est déjà convenu qu'on ne manipule que des nombres entiers).

Exemple pour la recherche dans un tableau trié. On veut chercher un élément x dans un tableau t trié.

— Condition : t doit être trié en ordre croissant.

— Le résultat de `binarySearch`(x, t) doit être un entier i tel que $t[i] = x$ s'il en existe, et `-1` sinon,

Cette spécification est plus subtile. Voici son articulation logique explicitée.

— Condition : « être trié » est une propriété complexe. Sa définition contient une quantification sur les indices du tableau t, que l'on suppose de taille n.

$$\forall i, j \in [0, n[\,,\ i < j \implies t[i] \leq t[j]$$

— Spécification du résultat r de `binarySearch`(x, t), en notant n la longueur de t : on distingue deux cas, chacun impliquant encore une quantification sur les indices du tableau.

  — Si x est présent dans t, c'est-à-dire si $\exists i \in [0, n[\,,\ t[i] = x$, alors le résultat doit être un indice où x apparaît : $r \in [0, n[\wedge t[r] = x$.

  — Si x n'est pas présent dans t, c'est-à-dire si $\forall i \in [0, n[\,, t[i] \neq x$, alors le résultat doit vérifier $r =$ `-1`.

  Le tout résumé en une liste de (deux) formules :

$$\begin{cases} (\exists i \in [0, n[\,, t[i] = x) \implies r \in [0, n[\wedge t[r] = x \\ (\forall i \in [0, n[\,, t[i] \neq x) \implies r = \texttt{-1} \end{cases}$$

  Une telle liste exprime une *conjonction* : toutes les formules doivent être valides.

Exemple pour la recherche d'une séquence. On se donne un texte t, et on y cherche une séquence de lettres s.

— Condition : aucune, tous les textes et toutes les séquences cherchées sont a priori admissibles.

— Le résultat de `stringSearch`(s, t) doit être un entier i tel que le texte t contient une occurrence de la séquence s commençant au caractère d'indice i, s'il existe une telle occurrence, et `-1` sinon,

La spécification du résultat ressemble à celle vue pour la recherche dichotomique, mais fait un usage plus riche des quantificateurs puisque l'occurrence d'une séquence est déterminée par une suite de plusieurs lettres. Ainsi, « s est présente dans t » s'énonce « il existe un indice i à partir duquel tous les indices suivants correspondent aux lettres de s ». En notant n_t la longueur du texte t, et n_s la longueur de la séquence s :

$$\exists i \in [0, n_t - n_s]\,,\ \forall j \in [0, n_s[\,,\ s[j] = t[i + j]$$

Au contraire, « s est absente de t » s'énonce « quelque soit l'indice i de départ que l'on considère, on trouve au moins une des lettres suivantes qui diffère de la lettre correspondante de s ».

$$\forall i \in [0, n_t - n_s]\,,\ \exists j \in [0, n_s[\,,\ s[j] \neq t[i + j]$$

Préconditions. Elles décrivent les contraintes que doivent vérifier les données prises en entrée par un algorithme. Dit autrement, les préconditions définissent les entrées valides, et délimitent ainsi les contours du problème que l'on cherche à résoudre. Voici les manières dont il faut considérer les préconditions, selon le point de vue pris.

- Conception : on peut tenir les préconditions pour acquises. On cherche à résoudre le problème uniquement pour les entrées valides.
- Raisonnement : les préconditions deviennent des hypothèses. On suppose qu'elles sont valides et on peut en déduire d'autres choses.
- Utilisation : il faut s'assurer que les entrées que l'on fournit à un algorithme sont bien valides.
- Programmation : on *peut* interrompre le programme et produire un message d'erreur lorsque les préconditions ne sont pas réalisées, pour indiquer à l'utilisateur qu'il n'a pas suivi les règles. On peut aussi ne rien faire, et laisser le programme faire n'importe quoi lorsque les entrées sont invalides.

Ainsi dans la conception d'un algorithme de *recherche d'un élément dans un tableau trié*, toute considération sur les tableaux non triés est hors sujet : la recherche dans un tableau non trié est un *autre* problème. Si un utilisateur utilise `binarySearch` sur un tableau non trié, il a toutes les chances de recevoir en retour un résultat faux, mais c'est son problème. Le concepteur et le programmeur d'un algorithme ne sont pas responsables des utilisateurs qui ne lisent pas le mode d'emploi.

Note GL : le concepteur et le programmeur sont en revanche responsables du fait que le mode d'emploi soit simple et clair.

1.3 Invariants de boucles

Pour montrer qu'un algorithme est *correct*, c'est-à-dire résoud le problème posé :

- on suppose qu'avant exécution les préconditions sont valides,
- à l'aide de ces hypothèses on justifie qu'après exécution, le résultat correspond à la spécification.

Suivi de l'exécution d'un programme. Naturellement, le raisonnement suit l'exécution de l'algorithme et l'évolution progressive des différentes variables ou données. Par exemple, avec les trois instructions suivantes :

```
a = a-b
b = a+b
a = b-a
```

On note n_a la valeur initiale de la variable `a` et n_b la valeur initiale de la variable `b`. Après la première instruction, `a` contient $n_a - n_b$. Après la deuxième, `b` contient $(n_a - n_b) + n_b = n_a$. Après la dernière, `a` contient $n_a - (n_a - n_b) = n_b$. Finalement, les valeurs de `a` et `b` ont été échangées. On peut présenter ce suivi dans un tableau donnant le contenu des variables après chaque instruction.

	a	b
Initialement	n_a	n_b
`a = a-b`	$n_a - n_b$	n_b
`b = a+b`	$n_a - n_b$	n_a
`a = b-a`	n_b	n_a

Ce suivi précis pas-à-pas ne fonctionne pas pour les algorithmes plus complexes. Par exemple pour cette fonction d'exponentiation rapide :

```
static int power(int a, int n) {
    int r = 1;
    while (n > 0) {
        if (n % 2 == 1) r = r*a;
        a = a*a;
        n = n/2;
    }
    return r;
}
```

On a un nombre de tours dépendant de l'entrée, et une affectation conditionnelle qui, selon l'entrée, est effectuée à certains tours de boucle et pas à d'autres. On ne peut suivre l'exécution de cet algorithme que pour un n concret donné.

Invariant de boucle. Pour raisonner sur un tel algorithme on cherche à établir un *invariant de boucle*, c'est-à-dire une propriété logique à propos des variables, qui est valide du début à la fin de l'exécution (« *invariablement* valide »). Plus précisément, l'invariant doit :

— être vrai avant le premier tour de boucle,

— être préservé par chaque tour de boucle.

Exemple pour l'exponentiation rapide. On note a_0 et n_0 les valeurs initiales des deux arguments `a` et `n`, et a, n et r les valeurs des trois variables du programme à un instant donné. La formule

$$r \times a^n \quad = \quad a_0^{n_0}$$

est un invariant de la boucle. En effet :

— Avant le premier tour, $r = 1$, $a = a_0$ et $n = n_0$ et l'équation est immédiate.

— On suppose $r \times a^n = a_0^{n_0}$ vraie au début d'un tour de boucle et on note a', n' et r' les valeurs des variables telles que mises à jour à la fin du tour. Deux cas en fonction de la parité de n :

Décomp. n	a'	n'	r'	Calcul $r' \times a'^{n'}$
$n = 2k$	a^2	k	r	$r \times (a^2)^k = r \times a^{2k} = a^n$
$n = 2k+1$	a^2	k	$a \times r$	$r \times a \times (a^2)^k = r \times a^{2k+1} = a^n$

Dans tous les cas $r' \times a'^{n'} = a_0^{n_0}$: l'équation reste valide.

À noter : l'invariant peut être *temporairement* invalide pendant l'exécution d'un tour de boucle (les variables ne sont pas toutes mises à jour en même temps). Ce qui compte est que l'invariant soit vrai à nouveau à la fin du tour : il est alors également vrai au début du tour suivant, puis à la fin du suivant, et ainsi de suite jusqu'à la fin des tours.

Exemple : recherche dichotomique. Au début du cours, on a justifié la correction de la recherche dichotomique dans un tableau trié à l'aide d'un « fait clé » : l'élément `x` cherché ne peut pas se trouver en dehors de l'intervalle $[\texttt{lo}, \texttt{hi}[$, car tous les éléments de `tab` $[0, \texttt{lo}[$ sont strictement inférieurs à la cible, et tous les éléments de `tab` $[\texttt{hi}, n[$ lui sont strictement supérieurs. Ce « fait clé » est un invariant de la boucle **`while`**. On énonce également le fait que `lo` et `hi` définissent toujours intervalle du tableau et on obtient les invariants suivants :

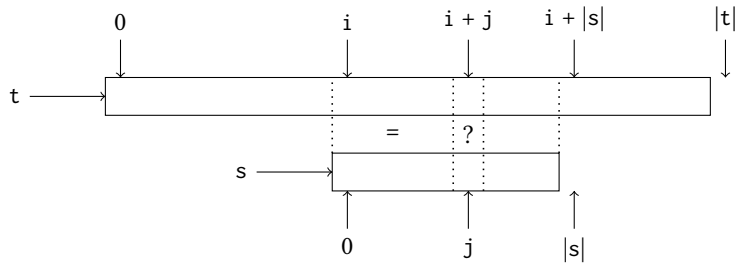
$$\begin{cases} 0 \le \texttt{lo} \le \texttt{hi} \le n \\ \forall i \in [0, \texttt{lo}[\,, \texttt{tab}[i] < \texttt{x} \\ \forall i \in [\texttt{hi}, n[\,, \texttt{tab}[i] > \texttt{x} \end{cases}$$

(on note n la taille du tableau `tab`). Ces propriétés sont vraies avant le premier tour de boucle, puis préservées par chaque tour : elles sont donc bien des « invariants » et restent vraies jusqu'à la fin de l'exécution. En particulier, quand la boucle s'arrête car `lo = hi` ces invariants assurent que l'élément `x` cherché n'apparaît pas dans le tableau : `-1` est bien le résultat attendu.

Exemple : recherche d'une séquence. On se donne le code java simple suivant, pour chercher une séquence `s` dans un texte `t`. Cet algorithme énumère toutes les positions de départ `i` possibles dans `t`, en omettant seulement celles qui ne laissent pas assez de place pour une occcurrence de la séquence `s`, puis on teste ensuite chaque caractère suivant la position `i`, en s'interrompant lorsque l'on observe une différence entre la séquence `s` et le segment observé de `t`.

```
static int stringSearch(String s, String t) {
    int ls = s.length();
    int lt = t.length();
    mainLoop:
    for (int i=0; i+ls <= lt; i++) {
        for (int j=0; j<ls; j++) {
            if (s.charAt(j) != t.charAt(i+j)) continue mainLoop;
        }
        return i;
    }
    return -1;
}
```

Le schéma suivant illustre ce descriptif : on a commencé à comparer la séquence s au segment t[i, i + |s|[, les j premiers caractères correspondaient et l'on s'intéresse maintenant au caractère d'indice j de s, c'est-à-dire au j + 1-ème.



L'algorithme parcourt ainsi tous les indices j de s tant qu'il n'observe pas de différence, puis recommence en incrémentant i jusqu'à avoir trouvé une occurrence complète de s. On détecte que l'on a trouvé une occurrence complète lorsque la boucle interne a mené l'indice j jusqu'à la longueur |s| de s.

Les invariants des boucles de notre programme `stringSearch` formalisent ce schéma.

— Invariant de la boucle interne : les segments s[0, j[et t[i, i + j[coïncident.

$$\forall k \in [0, \mathtt{j}[\,,\ \mathtt{s}[k] = \mathtt{t}[\mathtt{i} + k]$$

— Invariant de la boucle externe : on n'a trouvé aucune occurrence complète de s commençant avant l'indice i. Autrement dit, tout segment t[k, k + |s|[démarrant à un indice $k <$ i a au moins une différence avec la séquence s.

$$\forall k \in [0, \mathtt{i}[\,,\ \exists k' \in [0, |\mathtt{s}|[\,,\ \mathtt{t}[k + k'] \neq \mathtt{s}[k']$$

1.4 Approfondissement : boîte à outils logique

Objectif : répertorier des éléments de langage que l'on peut utiliser pour s'exprimer sans ambiguïté, pour permettre des descriptions précises et des argumentations claires. Les phrases construites avec ces éléments sont des *formules*, qui peuvent être vraies ou fausses en fonction de leur forme et/ou du contexte. Deux formules sont *équivalentes* si elles sont vraies dans les mêmes contextes.

Prédicats. Propriétés de base des objets dont on parle. Par exemple :

Prédicats	**Contexte**
$a = b$, $a \neq b$	a, b objets quelconques
$n_1 < n_2$, $n_1 \leq n_2$, $n_1 > n_2$, $n_1 \geq n_2$	n_1, n_2 nombres
$X \subseteq Y$	X, Y ensembles
$a \in X$, $a \notin X$	X ensemble, a élément

Connecteurs. Articulations avec lesquelles on combine deux prédicats ou formules.

Connecteur	**Prononciation**	**Notation**	**Formule vraie quand...**
Conjonction	A et B	$A \wedge B$	A, B toutes deux vraies
Disjonction	A ou B	$A \vee B$	au moins une parmi A, B vraie
Négation	non A	$\neg A$	A fausse
Implication	si A alors B	$A \Longrightarrow B$	B vraie au moins dans les contextes où A vraie

On a aussi des notations pour deux formules dégénérées :

Formule	**Notation**	**Formule vraie...**
Tautologie	$\top$	toujours
Contradiction	$\bot$	jamais

Relations d'équivalence entre formes logiques :

Principe	Équivalences	
Absorption	$A \wedge \bot \equiv \bot$	$A \vee \top \equiv \top$
Neutralité	$A \wedge \top \equiv A$	$A \vee \bot \equiv A$
Commutativité	$A \wedge B \equiv B \wedge A$	$A \vee B \equiv B \vee A$
Associativité	$A \wedge (B \wedge C) \equiv (A \wedge B) \wedge C$	$A \vee (B \vee C) \equiv (A \vee B) \vee C$
Distributivité	$A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$	$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$
Lois de de Morgan	$\neg(A \wedge B) \equiv (\neg A) \vee (\neg B)$	$\neg(A \vee B) \equiv (\neg A) \wedge (\neg B)$
Involutivité	$\neg\neg A \equiv A$	
Négation	$\neg A \equiv A \Longrightarrow \bot$	
Non-contradiction	$A \wedge \neg A \equiv \bot$	
Tiers exclu	$A \vee \neg A \equiv \top$	
Implication	$A \Longrightarrow B \equiv (\neg A) \vee B$	
Contraposition	$A \Longrightarrow B \equiv (\neg B) \Longrightarrow (\neg A)$	

Quantificateurs. Les phrases logiques font souvent référence à des objets indéterminés. Par exemple : $x \neq 0 \Longrightarrow x = y + 1$. On note $A(x, y)$ une telle formule A faisant référence à deux objets x et y pris dans un certain ensemble (ici : $\mathbb{N}$). Les quantificateurs indiquent quels objets concrets peuvent être désignés par les variables x et y. Par exemple, la formule $A(x, y)$ précédente est valable pour tout $x \in \mathbb{N}$, mais une fois la valeur de x choisie celle de y devient très fortement contrainte.

Quantification	Prononciation	Notation	Formule vraie quand...
Universelle	pour tout x on a	$\forall x \in E, A(x)$	A est vraie quelque soit l'objet $e \in E$ désigné par x
Existentielle	il existe x tel que	$\exists x \in E, A(x)$	A est vraie pour au moins un objet $e \in E$

Exemple : $\forall x \in \mathbb{N}, (x \neq 0 \Longrightarrow (\exists y \in \mathbb{N}, x = y + 1))$. On omet parfois l'ensemble E lorsqu'il est clair dans le contexte.

Relations d'équivalence entre formules avec quantificateurs :

Principe	Équivalences	
Indépendance	$\forall x, \forall y, A(x, y) \equiv \forall y, \forall x, A(x, y)$	$\exists x, \exists y, A(x, y) \equiv \exists y, \exists x, A(x, y)$
Lois de de Morgan	$\neg(\forall x, A(x)) \equiv \exists x, \neg A(x)$	$\neg(\exists x, A(x)) \equiv \forall x, \neg A(x)$

Note : dans la vie courante, on exprime les propriétés manipulées en français, et pas avec les notations logiques. Cependant, même en langue naturelle on se ramène aux articulations données par les connecteurs logiques, afin de s'exprimer et raisonner avec précision et clarté. Dans ce cours on alternera entre les deux langues.

Raisonnement. Pour *justifier* qu'un fait donné est vrai, on déduit sa véracité à l'aide de règles de raisonnement en partant de certains faits de base supposés vrais.

On a donc toujours dans ce contexte un ensemble de formules (appelées *hypothèses*) dont on suppose qu'elles sont vraies, et à partir desquelles on veut *déduire* qu'une certaine formule cible (la *conclusion*) est vraie également.

Chaque articulation logique est associée à des règles de déduction de base, indiquant notamment :

- comment justifier une conclusion présentant cette articulation (règle d'introduction)
- comment utiliser une hypothèse basée sur cette articulation (règle d'élimination)

On a en plus un certain nombre de grandes techniques : raisonnement par l'absurde, tiers exclu, contradiction, récurrence...

À noter : on ne cherche jamais à justifier que les hypothèses sont elles-mêmes vraies. On veut simplement justifier qu'elles ne peuvent être vraies sans que la conclusion ne le soit elle aussi. D'ailleurs, on verra ci-dessous que certaines techniques de raisonnement consistent au contraire à montrer que les hypothèses ne peuvent pas être vraies.

Comment justifier une formule cible :

Formule cible	Action nécessaire
$A \wedge B$	justifier les deux formules
$A \vee B$	justifier l'une des deux formules (au choix)
$\neg A$	supposer l'hypothèse A et obtenir une contradiction
$A \Longrightarrow B$	supposer l'hypothèse A et justifier B
$\top$	aucune action requise
$\bot$	justifier à la fois A et $\neg A$ (formule A au choix)
$\forall x \in E, A(x)$	justifier $A(x)$ sans rien supposer sur x (à part le fait que $x \in E$)
$\exists x \in E, A(x)$	trouver un $e \in E$ pour lequel on arrive à justifier $A(e)$

Comment utiliser une hypothèse :

Hypothèse	Action possible
$A \wedge B$	déduire A, déduire B (une au choix, ou les deux)
$A \vee B$	déduire C, si on peut justifier à la fois $A \Longrightarrow C$ *et* $B \Longrightarrow C$ (raisonnement par cas)
$\neg A$	déduire une contradiction, si on peut justifier A
$A \Longrightarrow B$	déduire B, si on peut justifier A
$\top$	aucune déduction possible
$\bot$	*ex falso quod libet* (on peut déduire tout ce qu'on veut)
$\forall x \in E, A(x)$	déduire $A(e)$ pour un $e \in E$ au choix (même partiellement indéterminé)
$\exists x \in E, A(x)$	introduire un y et l'hypothèse $A(y)$ (sans rien supposer d'autre sur $y \in E$)

Comment réfuter une formule :

Formule à réfuter	Action nécessaire
$A \wedge B$	réfuter l'une des deux formules (au choix)
$A \vee B$	réfuter les deux formules
$\neg A$	justifier A
$A \Longrightarrow B$	trouver un cas dans lequel A est vraie mais pas B
$\top$	impossible (à part *ex falso*)
$\bot$	aucune action requise
$\forall x \in E, A(x)$	trouver un $e \in E$ pour lequel on peut réfuter $A(e)$
$\exists x \in E, A(x)$	réfuter $A(x)$ sans rien supposer sur x (à part $x \in E$)

Techniques supplémentaires :

- *Ex falso.* L'hypothèse $\bot$ permet de justifier n'importe quelle conclusion. Autrement dit, un ensemble d'hypothèses permettant de déduire une contradiction permet de justifier n'importe quelle formule.
- *Raisonnement par l'absurde.* Pour justifier une conclusion A, on peut prendre comme hypothèse $\neg A$ et chercher une contradiction.
- *Tiers exclu.* Pour justifier une conclusion C, on peut raisonner par cas sur la disjonction $A \vee \neg A$ pour une formule A au choix. D'où : choisir une formule A puis :
  - sous l'hypothèse A, justifier C,
  - sous l'hypothèse $\neg A$, justifier C.

Raisonnement par récurrence. Principe additionnel, pour justifier qu'une formule $A(n)$ est vraie pour tous les entiers $n \in \mathbb{N}$. Deux actions nécessaires :

1. initialisation : justifier $A(0)$,
2. hérédité : prendre un $n \in \mathbb{N}$ arbitraire, supposer $A(n)$ et justifier $A(n+1)$.

On en déduit : $\forall n \in \mathbb{N}, A(n)$.

Variante : récurrence forte. Deux actions nécessaires :

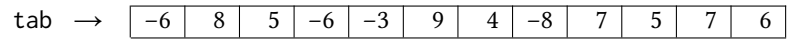
1. initialisation : justifier $A(0)$,
2. hérédité forte : pour un $n \in \mathbb{N}$ arbitraire non nul, supposer $A(k)$ pour tous les $k < n$, et justifier $A(n)$.

On déduit de même : $\forall n \in \mathbb{N}, A(n)$. Note : l'hérédité forte avec $n = 0$ correspond à l'initialisation.

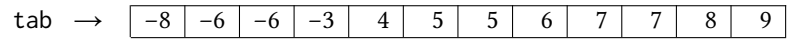
2 Trier

2.1 Problème : tri de tableau en place

On se donne un tableau contenant des entiers, et on souhaite réarranger ses éléments de sorte à ce qu'ils soient classés du plus petit au plus grand. Ce problème est le ***tri en place*** d'un tableau, où « en place » signifie que l'on travaille directement sur le tableau fourni et qu'on le modifie. Exemple d'entrée :



État attendu du tableau `tab` après tri de ses éléments :

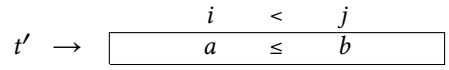


Spécification. La spécification du problème du tri en place comporte deux facettes :

- après le tri, les éléments du tableau doivent être rangés en ordre croissant,
- après le tri, le tableau doit contenir exactement les mêmes éléments qu'à l'origine (répétitions comprises).

On n'a en revanche aucune précondition : tous les tableaux d'éléments comparables doivent pouvoir être traités. Le tri en place modifie le tableau auquel on l'applique. Pour éviter les ambiguïtés, dans la suite on note t le tableau `tab` tel qu'il était avant application du tri, et t' ce même tableau une fois le tri effectué.

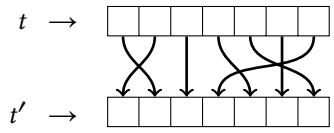
Pour exprimer que les éléments sont rangés en ordre croissant, on indique que tout élément $t'[j]$ situé à la droite d'un élément $t'[i]$ lui est supérieur ou égal.



Formule associée, en notant n la taille du tableau :

$$\forall i, j \in [0, n[\,,\ i < j \implies t'[i] \leq t'[j]$$

Pour exprimer que le tableau contient, après tri, les mêmes éléments qu'avant, on demande que le passage d'un tableau à l'autre soit obtenu par une permutation des cases.



On note n la taille du tableau `tab` (inchangée par le tri lui-même), et $\mathfrak{S}_n$ l'ensemble des ***permutations*** de l'intervalle $[0, n[$, c'est-à-dire des fonctions bijectives de l'intervalle $[0, n[$ vers lui-même. On obtient la formule :

$$\exists \sigma \in \mathfrak{S}_n,\ \forall i \in [0, n[\,,\ t'[\sigma(i)] = t[i]$$

Note : σ est une fonction qui représente les flèches du schéma précédent. Ainsi, l'image $\sigma(i)$ est l'indice de t' où arrive l'élément qui était à l'indice i dans t.

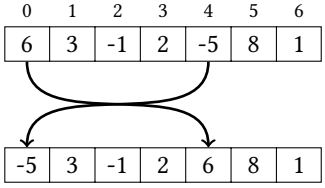
Avant de poursuivre. Sauriez-vous résoudre ce problème ?

2.2 Solution 1 : tri par sélection

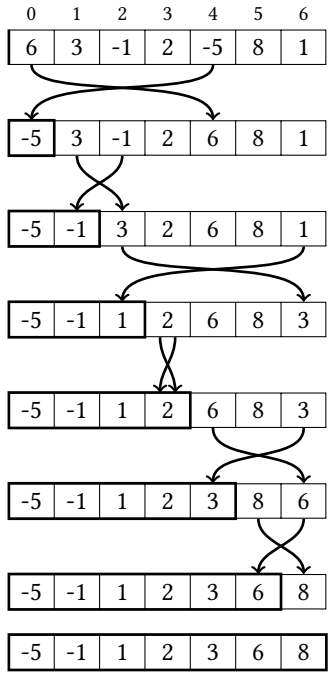
L'algorithme de ***tri par sélection*** procède ainsi :

- chercher le plus petit élément du tableau, et le mettre dans la première case ;
- puis, chercher le plus petit élément restant, et le mettre dans la deuxième case ;
- puis, chercher le plus petit élément restant, et le mettre dans la case suivante ;
- et ainsi de suite jusqu'à avoir traité l'ensemble.

Pour ne pas perdre d'éléments, chaque fois qu'il faut en déplacer un on effectue en réalité un ***échange*** des éléments contenus dans deux cases du tableau. Exemple de première étape, où l'on place le plus petit élément (−5, à l'indice 4) dans la première case (à l'indice 0).



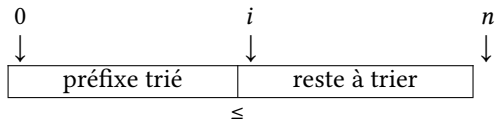
Exemple d'exécution. On part du tableau | 6 | 3 | -1 | 2 | -5 | 8 | 1 |. À chaque étape, la zone encadrée correspond à l'ensemble des cases pour lesquelles on a sélectionné une valeur. Notez que l'on s'épargne la sélection du dernier élément, qui est nécessairement déjà en place.



Code java. La fonction principale `selectionSort` fait appel à une fonction auxiliaire `swap` pour échanger deux éléments du tableau d'indices `i` et `j`, et une autre `indexMin` pour chercher l'indice d'un élément minimal dans un segment `tab`$[\texttt{i}, n[$.

```
static void swap(int[] tab, int i, int j) {
    int tmp = tab[i];
    tab[i] = tab[j];
    tab[j] = tmp;
}

static int indexMin(int[] tab, int i) {
    assert (i < tab.length);
    int jMin = i;
    for (int j = i+1; j < tab.length; j++) {
        if (tab[j] < tab[jMin])
            jMin = j;
    }
    return jMin;
}

static void selectionSort(int[] tab) {
    for (int i = 0; i < tab.length; i++) {
        int j = indexMin(tab, i);
        swap(tab, i, j);
    }
}
```

Invariants de l'algorithme. Après i étapes de cet algorithme, les i premières cases du tableau contiennent les i plus petits éléments, rangés par ordre croissant, et les cases suivantes contiennent les autres éléments, dans un ordre arbitraire.



Ces propriétés sont les ***invariants*** de l'algorithme, que l'on peut formaliser ainsi. Après `i` étapes de sélection :

— le segment `tab`$[0, $`i`$[$ est trié

$$\forall k_1, k_2 \in [0, \texttt{i}[\,,\ k_1 < k_2 \implies \texttt{tab}[k_1] \leq \texttt{tab}[k_2]$$

— et les éléments du segment `tab`$[0, $`i`$[$ sont plus petits que les éléments restants

$$\forall k_1 \in [0, \texttt{i}[\,,\ \forall k_2 \in [\texttt{i}, n[\,,\ \texttt{tab}[k_1] \leq \texttt{tab}[k_2]$$

À ces deux invariants s'ajoute celui énonçant qu'à chaque étape, le tableau est bien une permutation du tableau d'origine.

Spécification et invariants de la fonction auxiliaire. L'algorithme repose sur une fonction auxiliaire cherchant la position du minimum d'un segment de tableau. Spécifions la fonction `indexMin`(`tab`, `i`) de recherche du minimum de `tab`$[$`i`$, n[$ (où on suppose que `tab` est un tableau de taille n).

— Précondition : le segment `tab`$[$`i`$, n[$ n'est pas vide. Autrement dit : `i` $< n$.

— Le résultat r est l'indice d'un élément de `tab`$[$`i`$, n[$ minimal. Autrement dit : `i` $\leq r < n$ et $\forall k \in [$`i`$, n[\,,$ `tab`$[r] \leq$ `tab`$[k]$.

La fonction `indexMin` parcourt le segment `tab`$[$`i`$, n[$, et met à jour une variable `jMin` contenant l'indice du plus petit élément de la région `tab`$[$`i`$,$ `j`$[$ déjà parcourue. Invariants :

— l'indice `jMin` est dans l'intervalle $[$`i`$,$ `j`$[$

$$\texttt{i} \leq \texttt{jMin} < \texttt{j}$$

— l'indice `jMin` est l'indice d'un élément minimal du segment `tab`$[$`i`$,$ `j`$[$

$$\forall k \in [\texttt{i}, \texttt{j}[\,,\ \texttt{tab}[\texttt{jMin}] \leq \texttt{tab}[k]$$

2.3 Complexité : dénombrement d'opérations

Objectif de l'étude de la complexité : prédire le temps d'exécution ou la consommation mémoire d'un programme.

Expression de la complexité temporelle. Le temps d'exécution d'un programme est déterminé par :

— le temps nécessaire pour réaliser chaque opération de base,

— le nombre de fois que chaque opération est réalisée.

Opération de base : toute opération qu'on juge « atomique ». Par exemple : opérations arithmétiques, comparaisons, lecture ou écriture d'une case d'un tableau... Ces opérations de base n'ont pas toutes le même coût. Le plus souvent, compter les accès à la mémoire suffit à bien estimer le temps d'exécution, car cette opération est plutôt coûteuse. Dans le cas du tri en place d'un tableau, il s'agit des accès aux cases du tableau.

Le temps d'exécution d'un programme varie en fonction de ses entrées. Traditionnellement, on cherche à exprimer la complexité d'un algorithme en fonction de la taille de l'entrée. Pour un algorithme opérant sur des tableaux, on pourra par exemple exprimer une complexité $c(n)$ en fonction de la taille n du tableau pris en entrée.

Ordres de grandeur de complexité. En général, on ne s'intéresse pas à un décompte exact des opérations. On exprime plutôt un ***ordre de grandeur***, en se rapportant à quelques profils de référence. En voici quelques uns, exprimés en fonction d'une taille n pour les entrées.

Coût	Nom du profil	Cas typique	Évolution quand n double
1	constant	opération de base	pas d'évolution
$\log(n)$	logarithmique	dichotomie	ajout d'une constante
n	linéaire	boucle simple	multiplication par 2
$n\log(n)$	linéarithmique	diviser pour régner	multiplication par 2
n^2	quadratique	2 boucles imbriquées	multiplication par 4
n^3	cubique	3 boucles imbriquées	multiplication par 8
2^n	exponentiel	*backtracking*	carré

Si on considère une complexité $c(n) = 3n^2 + 5n + 17$, l'essentiel de la valeur de $c(n)$ est déterminée par le ***terme dominant*** $3n^2$, notamment lorsque l'on considère de grandes valeurs de n. On dit que $c(n)$ est ***équivalente*** à $3n^2$ et on note $c(n) \sim 3n^2$. En omettant la constante multiplicative 3, on peut également retenir que cette complexité est ***quadratique***, c'est-à-dire ***de l'ordre de*** n^2.

Les ***notations de Landau*** formalisent ces notions d'équivalence et d'ordre de grandeur. Ci-dessous, on considère que f et g sont des fonctions de $\mathbb{N}$ à valeurs positives.

Notation	Idée	Définition
$g(n) = \mathcal{O}(f(n))$	g majorée par f, à un facteur près	$\exists k \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \geq n_0,\ g(n) \leq kf(n)$
$g(n) = \Omega(f(n))$	g minorée par f, à un facteur près	$\exists k \in \mathbb{R}^+,\ \exists n_0 \in \mathbb{N},\ \forall n \geq n_0,\ kf(n) \leq g(n)$
$g(n) = \Theta(f(n))$	g de l'ordre de f (à un facteur près)	$g(n) = \mathcal{O}(f(n))$ *et* $g(n) = \Omega(f(n))$
$g(n) \sim f(n)$	g équivalente à f (précisément)	$\lim_{n\to\infty} \left(\frac{g(n)}{f(n)}\right) = 1$

Dans un calcul, $\mathcal{O}(f(n))$ désigne une fonction g arbitraire telle que $g(n) = \mathcal{O}(f(n))$. On dit de même « g est un $\mathcal{O}(f)$ » pour signifier $g(n) = \mathcal{O}(f(n))$. Les mêmes principes s'appliquent aux autres notations. Exemple : si on pose $c(n) = 3n^2 + 5n - 12$ on peut écrire :

En revanche, l'expression « g est au moins un $\mathcal{O}(f)$ » est une bêtise. Pourquoi ?

- $c(n) = \mathcal{O}(n^2)$, mais aussi a fortiori $c(n) = \mathcal{O}(n^3)$ ou même $c(n) = \mathcal{O}(2^n)$.
- $c(n) = \Omega(n^2)$, mais aussi a fortiori $c(n) = \Omega(n)$ ou même $c(n) = \Omega(1)$.
- $c(n) = \Theta(n^2)$.
- $c(n) \sim 3n^2$.

Dénombrement des opérations du tri par sélection. Dans les cas simples, pour une taille d'entrée donnée on peut calculer précisément le nombre d'opérations. Faisons-le pour le tri par sélection, en se concentrant sur le nombre de comparaisons de paires d'éléments du tableau, en fonction de la taille n du tableau `tab` donné en entrée.

- La fonction `indexMin` contient une boucle réalisant exactement une comparaison à chaque tour. La boucle réalise un tour pour chaque valeur de j dans l'intervalle $[i+1, n[$, soit $c_{\texttt{indexMin}}(i, n) = n - i - 1$ comparaisons au total.
- La fonction `selectionSort` ne fait pas elle-même de comparaison, mais appelle `indexMin` successivement pour toutes les valeurs de i dans l'intervalle $[0, n[$.

D'où nombre total de comparaisons :

$$c(n) = \sum_{0\leq i<n} c_{\texttt{indexMin}}(i, n) = \sum_{0\leq i<n} n - i - 1 = \sum_{0\leq i<n} i = \frac{n(n-1)}{2}$$

2.4 Solution 2 : tri insertion

L'algorithme de ***tri par insertion*** procède ainsi :

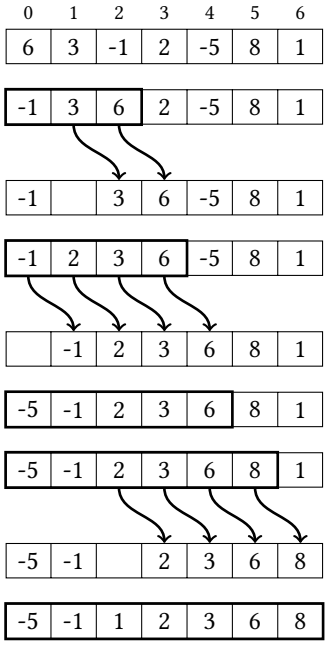
- trier en place le segment formé par le premier élément (rien à faire pour cette étape !),
- puis trier en place le segment formé par les deux premiers éléments,
- puis trier en place le segment formé par les trois premiers éléments,
- et ainsi de suite jusqu'à avoir trié l'ensemble.

Une fois un segment $t\,[0, i\,[$ trié, pour trier le segment $t\,[0, i]$ il suffit de :

- chercher la position j à laquelle devrait se trouver l'élément $t[i]$,
- décaler vers la droite tous les éléments de $t\,[j, i\,[$,
- puis insérer l'élément dans la case $t[j]$ maintenant libérée.

Exemple d'exécution. Partant du tableau | 6 | 3 | -1 | 2 | -5 | 8 | 1 |, on montre d'abord le tableau obtenu après tri des trois premiers éléments, puis on cherche à insérer l'élément 2 (situé à l'indice 3). On repère qu'il doit être inséré entre les éléments −1 et 3, c'est-à-dire à l'indice 1. On décale pour cela les éléments 3 et 6 d'une case vers la droite. On procède de même pour les éléments suivants. Notez que l'insertion de 8 (initialement à l'indice 5), lors

de la troisième étape représentée ici, ne nécessite aucun décalage puisque cet élément est plus grand que tous ceux situés à sa gauche.



Code java. La fonction principale `insertionSort` fait appel à une fonction auxiliaire `insert`, qui insère l'élément d'indice i du tableau `tab` au bon endroit dans le segment `tab`$[0, i]$, en décalant vers la droite les éléments qui doivent l'être.

```
static void insert(int[] tab, int i) {
    assert (i < tab.length);
    int v = tab[i];
    int j = i;
    while (j > 0 && tab[j-1] > v) {
        tab[j] = tab[j-1];
        j--;
    }
    tab[j] = v;
}

static void insertionSort(int[] tab) {
    for (int i=1; i < tab.length; i++) {
        insert(tab, i);
    }
}
```

Invariants de l'algorithme. Invariant principal : à l'étape `i`, le segment `tab`$[0, \texttt{i}[$ est trié.

$$\forall k_1, k_2 \in [0, \texttt{i}[\,,\ k_1 < k_2 \implies \texttt{tab}[k_1] \leq \texttt{tab}[k_2]$$

En outre, à toute étape le tableau est une permutation du tableau d'origine.

Dans la fonction `insert`, on décale d'un cran vers la droite tous les éléments du segment `tab`$[0, \texttt{i}[$ qui sont strictement supérieurs à `v`, en commençant par l'élément le plus à droite. Invariants de la fonction d'insertion : le segment `tab`$[0, \texttt{i}]$ est trié en ordre croissant et tous les éléments à droite de l'indice `j` (dans le segment `tab`$[0, \texttt{i}]$) sont strictement supérieurs à la valeur `v` à insérer.

$$\begin{cases} \forall k_1, k_2 \in [0, \texttt{i}]\,,\ k_1 < k_2 \implies \texttt{tab}[k_1] \leq \texttt{tab}[k_2] \\ \forall k \in [\texttt{j}+1, \texttt{i}]\,,\ \texttt{v} < \texttt{tab}[k] \end{cases}$$

En outre, à chaque étape, le tableau que l'on obtiendrait en insérant `v` à l'indice `j` est une permutation du tableau d'origine.

2.5 Complexité : meilleur cas, pire cas, moyenne

Différentes entrées de même taille peuvent donner des coûts d'exécution différents. C'est ce que l'on peut observer avec le tri par insertion. Considérons un tableau `tab` de taille n et dénombrons les opérations de comparaison.

- La fonction principale `insertionSort` réalise $n-1$ appels à la fonction auxiliaire `insert`, pour toutes les valeurs de i prises dans l'intervalle $[1, n[$.
- La fonction `insert` réalise :
  1. au minimum une comparaison, si `tab[`i`]` $\geq$ `tab[`$i-1$`]`,
  2. au maximum i comparaisons, si `tab[`i`]` < `tab[0]`,
  3. ou n'importe quel nombre intermédiaire.

Trois nuances de complexité. Pour tenir compte de cette variabilité on calcule trois complexités pour les entrées de taille n.

- ***Meilleur cas :*** complexité pour une entrée donnant un coût minimal. Indique le mieux qu'on puisse attendre, sur une entrée particulièrement favorable.
- ***Pire cas :*** complexité pour une entrée donnant un coût maximal. Indique un maximum, garanti jamais dépassé même sur les entrées les plus défavorables.
- ***Complexité moyenne*** sur toutes les entrées de taille n. Indique ce qu'on peut raisonnablement espérer pour une entrée prise au hasard.

Meilleur cas, pire cas et moyenne pour le tri par insertion. Pour calculer les complexités du tri par insertion, on se concentre sur les différentes complexités possibles de la partie dont la complexité peut effectivement varier, c'est-à-dire la fonction `insert`.

- Le cas minimum de `insert` est réalisé lorsque `tab[`i`]` $\geq$ `tab[`$i-1$`]`. Ce cas se produit à chaque appel à `insert` si `tab` est dès l'origine trié en ordre croissant. On a donc $n-1$ comparaisons au total dans le meilleur cas.
- Le cas maximum de `insert` est réalisé lorsque `tab[`i`]` < `tab[`j`]` pour tout $j \in [0, i[$. Ce cas se produit à chaque appel à `insert` si `tab` est à l'origine trié en ordre décroissant. On a donc $\sum_{1\leq i<n} i = \frac{n(n-1)}{2}$ comparaisons au total dans le cas le pire.
- Pour un appel à `insert` sur un tableau quelconque, toutes les complexités entre 1 et i sont équiprobables. Chaque appel à cette fonction réalise donc en moyenne $\frac{i}{2}$ comparaisons. On a ainsi $\sum_{1\leq i<n} \frac{i}{2} = \frac{n(n-1)}{4}$ comparaisons en moyenne pour un tri complet.

D'où meilleur cas $\sim n$, pire cas $\sim \frac{1}{2}n^2$ et en moyenne $\sim \frac{1}{4}n^2$ comparaisons.

2.6 Approfondissement : calculs de complexité

Identités utiles. Quelques identités utiles pour résoudre les sommes ou produits obtenus dans des calculs de complexité, avec ordres de grandeur.

Expression			Résultat	Équivalent	Ordre
$1+2+3+4+\ldots+n$	$=$	$\sum_{0\leq k\leq n} k$	$\frac{n(n+1)}{2}$	$\sim \frac{n^2}{2}$	$\Theta(n^2)$
$1+2+4+8+\ldots+2^n$	$=$	$\sum_{0\leq k\leq n} 2^k$	$2^{n+1}-1$	$\sim 2^{n+1}$	$\Theta(2^n)$
$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\ldots+\frac{1}{n}$	$=$	$\sum_{0\leq k\leq n} \frac{1}{k}$	H_n	$\sim \ln(n)$	$\Theta(\log(n))$
$1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\ldots+\frac{1}{2^n}$	$=$	$\sum_{0\leq k\leq n} \frac{1}{2^k}$	$2-\frac{1}{2^n}$	~ 2	$\Theta(1)$
$1\times 2\times 3\times 4\times\ldots\times n$	$=$	$\prod_{1\leq k\leq n} k$	$n!$	$\sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$	$\Theta(\sqrt{n}\left(\frac{n}{e}\right)^n)$
$\log(1)+\log(2)+\ldots+\log(n)$	$=$	$\sum_{1\leq k\leq n} \log(k)$	$\log(n!)$	$\sim n\log(n)$	$\Theta(n\log(n))$

Note : H_n s'appelle la *série harmonique*. Traditionnellement $\log_b$ est le logarithme en base b, et $\ln = \log_e$ est le logarithme naturel (népérien). Dans ce cours, en plus, on écrit simplement log sans précision de base pour $\log_2$.

Modèle des entrées pour le calcul en moyenne. Pour un ensemble fini de valeurs la notion de moyenne est simple : somme des valeurs divisée par le cardinal de l'ensemble. Ici le nombre de tableaux différents de taille n est infini. Cependant, pour étudier un algorithme de tri les valeurs exactes de chaque case d'un tableau n'ont pas d'importance : seules comptent les comparaisons deux à deux. Dans notre étude il n'y a pas de différence entre

2	0	1

et

19273	-374	2178

: seul compte le fait qu'on a d'abord le plus grand élément, puis le plus petit, puis le médian. En supposant que les tableaux ne contiennent pas de doublons, on a $n!$ configurations possibles pour un tableau de taille n, *et ces n configurations sont équiprobables.*

Pour les calculs de complexité moyenne on utilise ce modèle des *tableaux ordonnés aléatoirement sans répétition.* Les complexités moyennes obtenues restent valables avec une quantité modérée de répétitions. Si on veut pouvoir analyser un cas particulier d'application où on attend de nombreuses répétitions il faut adopter un autre modèle spécifique.

2.7 Approfondissement : tris en caml

Les tableaux existent également en caml. L'accès à la case d'indice `i` du tableau `tab` se note `tab.(i)`. L'affectation d'une nouvelle valeur se note `tab.(i) <- v`. Partant de cela, voici comment on aurait pu écrire les algorithmes de ce chapitre en caml.

Tri par sélection. On définit une variable *mutable* avec **`let`** `x = ref v` **`in`**. On accède à la valeur d'une variable mutable `x` avec `!x` et on la modifie avec `x := v'`. On sépare deux instructions avec un point-virgule ; Dans une boucle **`for`**, on donne l'indice de début et l'indice de fin (inclus), et on délimite le corps de la boucle par **`do`** et **`done`**.

```
let swap tab i j =
  let tmp = tab.(i) in
  tab.(i) <- tab.(j);
  tab.(j) <- tmp

let index_min tab i =
  assert (i < Array.length tab);
  let j_min = ref i in
  for j = i+1 to Array.length tab - 1 do
    if tab.(j) < tab.(!j_min) then
      j_min := j
  done;
  !j_min

let selection_sort tab =
  for i = 0 to Array.length tab - 1 do
    let j = index_min tab i in
    swap tab i j
  done
```

Tri insertion. Le corps d'une boucle **`while`**, comme celui d'une boucle **`for`**, est délimité par **`do`** et **`done`**.

```
let insert tab i =
  assert (i < Array.length tab);
  let v = tab.(i) in
  let j = ref i in
  while !j > 0 && tab.(!j-1) > v do
    tab.(!j) <- tab.(!j-1);
    decr j
  done;
  tab.(!j) <- v

let insertion_sort tab =
  for i = 0 to Array.length tab - 1 do
    insert tab i
  done
```

Essayez également d'écrire à nouveau ces algorithmes dans les autres langages que vous connaissez. Par exemple : python.

3 Accélérer

3.1 Problème : tri de tableau en place, plus rapidement

Les deux solutions précédentes au problème du tri ont pour point commun une complexité quadratique. On veut maintenant réaliser cette même tâche, mais plus rapidement. Remarquons le point suivant : pour les tris quadratiques que nous connaissons, trier un tableau de taille $\frac{n}{2}$ prend quatre fois moins de temps que trier un tableau de taille n. Ainsi, trier indépendamment l'une de l'autre deux moitiés d'un tableau de taille n revient à faire deux tris de tableaux de taille $\frac{n}{2}$, ce qui prend deux fois moins de temps que trier le tableau complet. Évidemment, on ne peut pas se contenter de cela : il faut encore faire en sorte que les deux moitiés triées puissent bien être combinées en un tableau globalement trié. Mais une partie du temps gagné sur les tris des deux moitiés peut être utilisée pour cela.

3.2 Algorithme : tri rapide

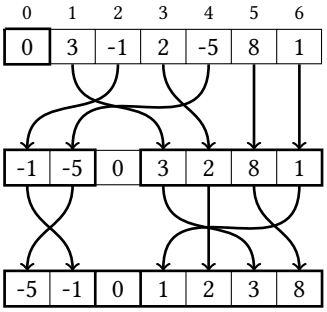
L'algorithme de ***tri rapide*** procède ainsi :

- placer dans une partie gauche du tableau les élément « petits » et dans une partie droite les éléments « grands »,
- trier indépendamment chacune des deux parties,
- réaliser le point précédent en utilisant à nouveau le même algorithme, jusqu'à n'avoir plus à trier que des tableaux si petits qu'ils n'ont plus à être découpés.

Le tri séparé des deux parties suffit à obtenir un ensemble trié, puisque l'on a pris soin à la première étape de ne mettre à droite que des éléments plus grands que ceux situés à gauche. Pour répartir les éléments du tableau en deux groupes, on les compare à un élément ***pivot*** pris dans le tableau :

- les éléments plus petits que le pivot sont déclarés « petits » et placés à gauche,
- les éléments plus grands que le pivot sont déclarés « grands » et placés à droite,
- le pivot lui-même est placé entre les deux groupes, et l'on peut même regrouper ainsi au « centre » toutes les occurrences de l'élément pivot s'il y en a plusieurs.

Le pivot peut être n'importe quel élément du tableau, par exemple le premier. Notez que les deux groupes à trier ensuite n'ont pas besoin d'inclure le pivot lui-même, puisque celui-ci est déjà à sa place définitive : il n'a que des éléments plus petits à sa gauche et que des éléments plus grands à sa droite.



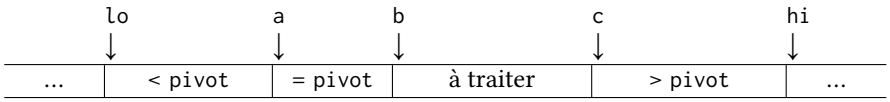
0	1	2	3	4	5	6
0	3	-1	2	-5	8	1

Code java. La fonction principale `quickSort` prend en paramètres un tableau `tab` et deux indices `lo` et `hi`, et trie le segment `tab`[`lo`, `hi`[. Pour cela, elle combine une boucle réorganisant le tableau autour d'un élément pivot, et des appels récursifs sur les deux sous-tableaux situés sous le pivot et au-dessus du pivot.

```
static void swap(int[] tab, int i, int j) {
    int tmp = tab[i];
    tab[i] = tab[j];
    tab[j] = tmp;
}

static void quickSort(int[] tab, int lo, int hi) {
    if (hi <= lo+1) return;
    int a=lo, b=lo+1, c=hi;
    int pivot = tab[lo];
    while (b < c) {
        if      (tab[b] < pivot) { swap(tab, b++, a++); }
        else if (tab[b] > pivot) { swap(tab, b, --c); }
        else /* tab[b] == pivot */ { b++; }
    }
    quickSort(tab, lo, a);
    quickSort(tab, c, hi);
}
```

```
static void quickSort(int[] tab) {
    quickSort(tab, 0, tab.length);
}
```

Invariants de la boucle de partition. Pendant l'opération de partition, le segment `tab[lo, hi[` est découpé en quatre parties :



— le segment `tab[lo, a[` ne contient que des éléments strictement inférieurs au pivot

$$\forall k \in [\texttt{lo}, \texttt{a}[\,,\ \texttt{tab}[k] < \texttt{pivot}$$

— le segment `tab[a, b[` ne contient que des éléments égaux au pivot

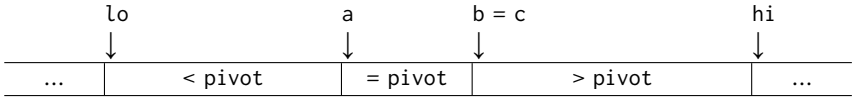
$$\forall k \in [\texttt{a}, \texttt{b}[\,,\ \texttt{tab}[k] = \texttt{pivot}$$

— le segment `tab[b, c[` contient des éléments non encore traités, qui peuvent être quelconques,

— le segment `tab[c, hi[` ne contient que des éléments strictement supérieurs au pivot

$$\forall k \in [\texttt{c}, \texttt{hi}[\,,\ \texttt{tab}[k] > \texttt{pivot}$$

L'un de ces quatre segments est à coup sûr non vide : il s'agit de `tab[a, b[`, qui contient au moins une occurrence du pivot. On a donc la chaîne de comparaisons $\texttt{lo} \leq \texttt{a} < \texttt{b} \leq \texttt{c} \leq \texttt{hi}$. À chaque tour de boucle, le segment des éléments à traiter est réduit d'une case au profit de l'un des trois autres. La boucle s'arrête lorsque `b = c`, c'est-à-dire lorsque le segment des éléments à traiter est vide. Le tableau a alors la forme suivante, où à coup sûr `a < c` (on a au moins un élément dans le segment des éléments égaux au pivot).

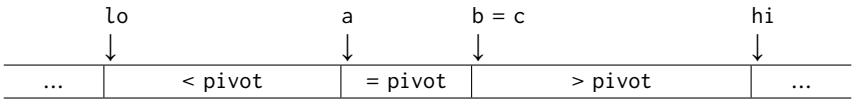


En outre, le segment de tableau obtenu après partition est bien une permutation du segment d'origine.

Technique de preuve : récurrence forte. Pour finir de justifier la correction du tri, et calculer sa complexité, il va nous falloir de nouvelles techniques pour gérer la récurrence. Le tri d'un tableau de taille n se ramène, après partition, au tri de deux tableaux de tailles strictement inférieures à n. On justifie alors que l'algorithme est correct à l'aide du principe de récurrence forte. Pour cela, on note $P(n)$ la propriété « `quickSort` trie correctement tout segment de tableau de longueur n », et on vérifie les conditions suivantes :

— $P(0)$: `quickSort` trie correctement tout segment de tableau de longueur 0. C'est immédiat car l'algorithme ne fait rien lorsque `lo = hi`, et un segment vide `tab[lo, lo[` est bien toujours trié.

— En fixant un $n \in \mathbb{N}$ et en supposant que $P(k)$ est vraie pour tout $k < n$, c'est-à-dire que `quickSort` trie correctement tout segment de tableau de longueur strictement inférieure à n, on cherche à démontrer que l'algorithme trie correctement un tableau de taille n. Les invariants de la boucle de partition nous assurent déjà que cette dernière réarrange le tableau sous la forme



avant d'appliquer récursivement l'algorithme aux segments `tab[lo, a[` et `tab[c, hi[`. Comme `a < c`, on sait que les longueurs `a − lo` et `hi − c` de ces deux segments sont strictement inférieures à n = `hi − lo`. Autrement dit, par hypothèse l'algorithme trie correctement ces deux segments. À la fin, on obtient donc bien une permutation du segment d'origine, dont on vérifie qu'elle est bien triée. Soient deux indices $i, j \in [\texttt{lo}, \texttt{hi}[$ tels que $i < j$. Vérifions que $\texttt{tab}[i] \leq \texttt{tab}[j]$ en raisonnant par cas sur les segments où se trouvent i et j.

- Si $i, j \in$ [lo, a [ou $i, j \in$ [c, hi [, on a bien tab[i] ≤ tab[j] car on a déjà justifié que ces deux segments étaient triés.
- Dans les autres cas, on fait une comparaison intermédiaire avec le pivot.
  - Si $i, j \in$ [a, b [, alors tab[i] = pivot = tab[j].
  - Si $i \in$ [lo, a [et $j \in$ [a, b [, alors tab[i] < pivot = tab[j].
  - Si $i \in$ [a, b [et $j \in$ [c, hi [, alors tab[i] = pivot < tab[j].
  - Si $i \in$ [lo, a [et $j \in$ [c, hi [, alors tab[i] < pivot < tab[j].

3.3 Complexité : équations récursives

Dans le cas d'algorithmes récursifs, la complexité peut elle-même être calculée par des équations récursives.

Exemple : factorielle. On veut calculer $n! = 1 \times 2 \times 3 \times \ldots \times n$. En java :

```
static int fact(int n) {
    if (n < 2) { return 1; }
    else       { return n * fact(n-1); }
}
```

Le nombre $C(n)$ de multiplications réalisées pour calculer `fact(`n`)` suit l'une des deux formules suivantes.

- pour $n < 2$, zéro,
- pour $n \geq 2$, une multiplication en plus du coût du calcul de `fact(`$n-1$`)`.

Autrement dit :

$$\begin{cases} C(0) &= 0 \\ C(1) &= 0 \\ C(n+1) &= 1 + C(n) \qquad \text{si } n \geq 1 \end{cases}$$

Exemple : exponentiation rapide. Principe de l'algorithme : on calcule rapidement de grandes puissances en remarquant que $a^{2m} = (a^m)^2$ et $a^{2m+1} = a \times (a^m)^2$. En java :

```
static int power(int a, int n) {
    if (n < 1) return 1;
    int b = power(a, n/2);
    if (n%2 == 0) { return b*b; }
    else          { return a*b*b; }
}
```

Pour une version alternative mais équivalente, on peut aussi remarquer que $a^{2m} = (a^2)^m$ et $a^{2m+1} = a \times (a^2)^m$.

Le nombre $C(n)$ de multiplications réalisées pour calculer `power(`a`,` n`)` vérifie les équations suivantes.

$$\begin{cases} C(0) &= 0 \\ C(2m) &= 1 + C(m) \qquad \text{si } m > 0 \\ C(2m+1) &= 2 + C(m) \end{cases}$$

Résolution des suites récursives simples. Soit $(u_n)_{n \geq n_0}$ une suite définie à partir du rang n_0. Pour tout $n \geq n_0$ on a :

$$u_n - u_{n_0} = \sum_{n_0 \leq k < n} (u_{k+1} - u_k)$$

(on qualifie cette équation de « télescopage » de la somme). En notant f la fonction telle que $u_{n+1} = u_n + f(n)$ pour tout $n \geq n_0$ on a donc :

$$u_n = u_{n_0} + \sum_{n_0 \leq k < n} f(k)$$

Cas particulier, la suite arithmétique : suite $(u_n)_{n \in \mathbb{N}}$ définie par

$$\begin{cases} u_0 &= b \\ u_{n+1} &= a + u_n \end{cases}$$

pour deux constantes a et b. Théorème : pour tout n, $u_n = an + b$.

Cas similaire, la suite géométrique : suite $(u_n)_{n\in\mathbb{N}}$ définie par

$$\begin{cases} u_0 &= b \\ u_{n+1} &= a \times u_n \end{cases}$$

pour deux constantes a et b. Théorème : pour tout n, $u_n = b \times a^n$.

Application à la factorielle et à l'exponentiation rapide. Les équations de complexité de la factorielle donnent une suite arithmétique avec $a = 0$ et $b = 1$, à partir du rang $n = 1$. D'où : pour tout $n \geq 1$, $C(n) = n - 1$.

Les équations de complexité de l'exponentiation rapide ne définissent pas une suite arithmétique ni une suite géométrique, puisqu'elles ne lient pas $C(n)$ et $C(n+1)$. En revanche on trouve une suite arithmétique en s'intéressant aux puissances de 2 :

$$\begin{cases} C(2^0) &= 2 \\ C(2^{k+1}) &= 1 + C(2^k) \end{cases}$$

D'où : pour tout $k \geq 0$, $C(2^k) = k + 2$.

Pour les autres nombres on obtient un encadrement : pour tous k et n, si $2^k \leq n < 2^{k+1}$ alors $k + 1 \leq C(n) \leq 2(k+1)$. Démonstration par récurrence sur k :

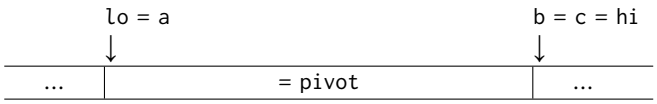
- Cas de base : $k = 0$. Alors $1 = 2^0 \leq n < 2^1 = 2$ et nécessairement $n = 1$. On calcule : $C(1) = 2 + C(0) = 2$. On a donc bien $0 + 1 \leq C(1) \leq 2(0+1)$.
- Hérédité. Soit k tel que pour tout n vérifiant $2^k \leq n < 2^{k+1}$ on a $k+1 \leq C(n) \leq 2(k+1)$. Soit n qui vérifie $2^{k+1} \leq n < 2^{k+2}$. On a $2^k \leq \lfloor \frac{n}{2} \rfloor < 2^{k+1}$, donc par hypothèse de récurrence $k + 1 \leq C(\lfloor \frac{n}{2} \rfloor) \leq 2(k+1)$. Or $C(n) = 1 + C(\lfloor \frac{n}{2} \rfloor)$ ou $C(n) = 2 + C(\lfloor \frac{n}{2} \rfloor)$. Donc $k + 2 \leq C(n) \leq 2(k+2)$.

On en déduit que pour tout $n \geq 0$, $\log(n) \leq C(n) \leq 2\log(n)$.

Exercice : plus précisément on a $\lfloor \log(n) \rfloor + 1 + w(n)$, où $w(n)$ est le nombre de 1 dans l'écriture binaire de n.

3.4 Complexité du tri rapide : cas extrêmes

Meilleur cas : partition dégénérée. Il existe une situation dans laquelle le tri rapide s'arrête particulièrement vite : lorsque tous les éléments sont égaux au pivot. L'étape de partition termine alors dans la situation



et les appels récursifs sur les segments vides `tab[lo, a[` et `tab[c, hi[` ne feront aucun travail supplémentaire.

Dans ce cas, le coût du tri se limite au coût de l'opération de partition : on a comparé `hi - lo - 1` paires d'éléments, et la complexité est linéaire.

Dans la suite de notre analyse, on éliminera ce cas en supposant que les n éléments du tableau à trier sont tous différents les uns des autres. En particulier, le pivot est présent en un seul exemplaire, et les $n-1$ autres éléments sont répartis entre les deux segments à trier récursivement.

Pire cas : partition déséquilibrée. Imaginons que le tableau à trier est *déjà* trié en ordre croissant (avec des éléments tous différents). En particulier, en choisissant le premier élément comme pivot, on trouve que les $n-1$ autres éléments lui sont strictement supérieurs, et doivent être placés du même côté. Le nombre $C_p(n)$ de comparaisons nécessaires au tri d'un tableau de cette forme ajoute :

- $n-1$ comparaisons pour comparer chaque autre élément au pivot,
- $C_p(n-1)$ comparaisons pour trier récursivement le segment des éléments supérieurs au pivot (ce segment étant également déjà trié, il suit la même formule).

Total : $C_p(n) = \sum_{1\leq k\leq n}(k-1) = \sum_{0\leq k'\leq n-1} k' = \frac{n(n-1)}{2}$. On conserve dans ce cas la complexité quadratique déjà observée pour le tri par sélection ou le tri par insertion.

Bilan : lorsque la partition du tableau en deux partie est très déséquilibrée, notre stratégie de découpage n'apporte rien.

Meilleur cas avec des éléments tous différents : partition équilibrée. À l'inverse, imaginons qu'à chaque étape, les $n-1$ éléments autres que le pivot soient répartis de manière équilibrée dans deux segments ayant des tailles aussi proches que possibles. Si $n-1$ est pair, les deux segments auraient ainsi la même taille $\frac{n-1}{2}$, et si $n-1$ est impair, l'un des deux contiendrait un élément de plus que l'autre. Le nombre $C_o(n)$ de comparaisons nécessaires pour trier un tableau de taille $n > 1$ dans ces conditions est donné par :

— les $n-1$ comparaisons du pivot avec chacun des autres éléments,

— les deux appels récursifs, sur des tableaux de tailles $\left\lceil \frac{n-1}{2} \right\rceil$ et $\left\lfloor \frac{n-1}{2} \right\rfloor$.

En particulier, les deux appels récursifs concernent des segments dont la taille est inférieure ou égale à $\frac{n}{2}$. On en déduit une borne supérieure valable lorsque $n > 1$.

$$C_o(n) \leq n + 2C_o(\frac{n}{2})$$

Réexprimons la formule dans le cas où n est une puissance de 2, c'est-à-dire où $n = 2^k$.

$$\begin{cases} C_o(2^0) & = & 0 \\ C_o(2^{k+1}) & \leq & 2C_o(2^k) + 2^{k+1} \end{cases}$$

Divisons enfin la deuxième équation par 2^{k+1}.

$$\frac{C_o(2^{k+1})}{2^{k+1}} \leq \frac{2C_o(2^k)}{2^{k+1}} + \frac{2^{k+1}}{2^{k+1}} = \frac{C_o(2^k)}{2^k} + 1$$

On y reconnaît une suite arithmétique, qui nous permet de conclure que, pour tout $k \in \mathbb{N}$, on a $\frac{C_o(2^k)}{2^k} \leq k$, et $C_o(2^k) \leq k2^k$. Autrement dit, si $n = 2^k$ on a $C_o(n) \leq n\log(n)$.

Finalement, si lors de l'exécution du tri rapide sur un tableau de taille n, chaque répartition des éléments d'un segment autour d'un pivot est *équilibrée*, le nombre de comparaisons effectué est de l'ordre de $n\log(n)$.

3.5 Approfondissement : complexité en moyenne du tri rapide.

Nous allons voir que la complexité du tri rapide, bien que susceptible de grandement varier en théorie, est en général excellente.

Notons $C(n)$ le nombre de paires d'éléments comparées en moyenne lors du tri rapide d'un tableau de taille n, dont on suppose que tous les éléments sont distincts (la présence de doublons ne fait qu'accélérer la résolution).

On a toujours $C(0) = C(1) = 0$. Pour $n \geq 2$, on a n cas possibles pour les tailles respectives des segments [lo, a[et [c, hi[: le segment [lo, a[peut avoir n'importe quelle longueur k comprise entre 0 et $n-1$ (bornes incluses), et l'autre segment a alors la longueur $n-1-k$. En outre, ces n cas sont équiprobables : le pivot peut se trouver à n'importe quelle position du segment [lo, hi[. Pour obtenir la complexité moyenne des deux appels récursifs il suffit donc de faire la moyenne de ces n cas. En comptant également les $n-1$ comparaisons nécessaires à la répartition préalable, on obtient :

$$\begin{aligned} C(n) &= n - 1 + \frac{1}{n}\left(\sum_{0\leq k<n} C(k) + C(n-1-k)\right) \\ &= n - 1 + \frac{2}{n}\sum_{0\leq k<n} C(k) \end{aligned}$$

Ici, $C(n)$ est exprimé en fonction de tous les $C(k)$ précédents. Pour exprimer $C(n)$ en fonction de $C(n-1)$ uniquement il faut, dans la somme, faire disparaître tous les éléments de $C(0)$ à $C(n-2)$. Pour cela on combine (pour $n \geq 3$) :

$$\begin{cases} nC(n) & = & n(n-1) + 2\displaystyle\sum_{0\leq k<n} C(k) \\ (n-1)C(n-1) & = & (n-1)(n-2) + 2\displaystyle\sum_{0\leq k<n-1} C(k) \end{cases}$$

On obtient :

$$nC(n) - (n+1)C(n-1) = 2(n-1)$$

On obtient une somme télescopique en divisant par $n(n+1)$:

$$\begin{aligned}\frac{C(n)}{n+1} - \frac{C(n-1)}{n} &= \frac{2(n-1)}{n(n+1)} \\ \frac{C(n)}{n+1} - \frac{C(2)}{3} &= \sum_{3\leq k\leq n}\left(\frac{C(k)}{k+1} - \frac{C(k-1)}{k}\right) = \sum_{3\leq k\leq n}\frac{2(k-1)}{k(k+1)} \\ \frac{C(n)}{n+1} &= \frac{1}{3} + 2\sum_{3\leq k\leq n}\frac{1}{k+1} - 2\sum_{3\leq k\leq n}\frac{1}{k(k+1)}\end{aligned}$$

On reconnaît dans cette expression la série harmonique et une série convergeant vers une constante. D'où finalement

$$C(n) \sim 2n\ln(n) \approx 1,39n\log(n)$$

Le nombre moyen de paires d'éléments comparées par le tri rapide est linéarithmique, avec une constante petite. Le cas idéal cité plus haut, dans lequel la partition du tableau en deux parties est toujours équilibrée, est en réalité représentatif du cas général !

Bilan sur le tri rapide : complexité excellente en général (linéarithmique en moyenne), mais mauvaise sur quelques cas particuliers (quadratique sur un tableau trié). Problème : dans les applications réelles le cas particulier du tableau presque trié n'est pas toujours aussi rare que dans le modèle aléatoire (imaginez un tableau qui avait déjà été trié, puis qu'on trie à nouveau après quelques modifications, ou un tableau construit à partir de plusieurs éléments triés). En pratique on gagne donc à ajouter de l'aléatoire dans cet algorithme, soit en choisissant le pivot au hasard, soit en *mélangeant* le tableau avant de le trier.

3.6 Approfondissement : *Master Theorem.*

Pour le « tri fusion » : voir TD. Son comportement est comparable au cas du tri rapide où la partition est équilibrée.

Le « théorème maître » donne directement l'ordre de grandeur du résultat pour des équations de forme similaire à celles du tri fusion, caractéristique des algorithmes de type « diviser pour régner ». On considère une équation récursive de la forme

$$C(n) = aC(\frac{n}{b}) + f(n)$$

où n est la taille du problème, a le nombre de sous-problèmes (entier non nul), $\frac{n}{b}$ la taille de chaque sous-problème (les sous-problèmes ont donc tous la même taille, à un arrondi arbitraire près), et $f(n)$ le coût propre à un appel donné (définition des sous-problèmes, combinaison des solutions, etc.). On suppose la fonction f croissante.

Exemples :

Algorithme	a	b	$f(n)$
Tri fusion	2	2 un arrondi inf. et un arrondi sup.	$\mathcal{O}(n)$ coût de la fusion
Recherche dichotomique	1	2 arrondi inf. ou sup. selon côté choisi	1 comparaison avec élément médian

L'ordre de $C(n)$ diffère selon que le coût des appels récursifs domine, est équilibré avec, ou est dominé par, le coût de gestion $f(n)$. On appelle $c_{\text{crit}} = \frac{\log(a)}{\log(b)} = \log_b(a)$ *l'exposant critique* qui permet de discriminer ces trois situations. On compare $f(n)$ à $n^{c_{\text{crit}}}$:

Cas	Complexité de $f(n)$	Condition	Ordre de $C(n)$
1.	$\mathcal{O}(n^c)$	$c < c_{\text{crit}}$	$\Theta(n^{c_{\text{crit}}})$
2.	$\Theta(n^c)$	$c = c_{\text{crit}}$	$\Theta(n^c\log(n))$
3.	$\Omega(n^c)$	$c > c_{\text{crit}}$	$\Theta(f(n))$

Les deux exemples du tri fusion et de la recherche dichotomique correspondent au cas 2. Ce cas 2 admet aussi une forme plus générale :

Cas	Complexité de $f(n)$	Condition	Ordre de $C(n)$
$2'$.	$\Theta(n^c(\log(n))^k)$	$c = c_{\text{crit}}$ et $k \geq 0$	$\Theta(n^c(\log(n))^{k+1})$

Outils logiques et algorithmiques

Thibaut Balabonski @ Université Paris-Saclay
Édition 2024.

Deuxième partie
Graphes

4 Gérer des conflits

4.1 Problème : allocation de ressources

Panique à l'université Saris-Paclay. Chaque filière a, chacune dans son coin, fixé son emploi du temps. Il y a chaque semaine des milliers de cours prévus, et il faut maintenant trouver une salle à chacun. Comme vous pouvez vous en douter, il y a nettement moins de salles disponibles que de cours à y loger quotidiennement : plusieurs devront certainement se succéder dans une même salle le même jour. Mais, évidemment, deux cours ne peuvent avoir lieu dans la même salle que si leurs horaires ne se recouvrent pas [1]. Nous avons donc :

- un ensemble de cours, dont certains peuvent se tenir dans une même salle (car leurs horaires sont disjoints) mais d'autres ne le peuvent pas (car leurs horaires se recouvrent),
- et un certain nombre de salles où loger nos cours.

L'objectif est d'affecter une salle à chaque cours en respectant ces contraintes.

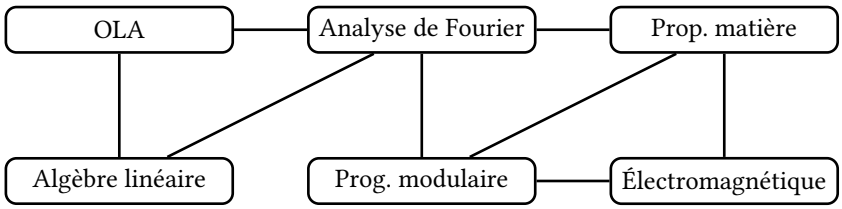
Échauffement. Voici un extrait des cours du lundi matin :

- Algèbre linéaire : 8h15–10h15
- Analyse de Fourier pour la physique : 8h15–10h45
- Outils logiques et algorithmiques : 8h45–10h15
- Transformations et propriétés de la matière : 10h30–12h00
- Programmation modulaire : 10h30–12h30
- Électromagnétique : 11h–12h45

De combien d'amphithéâtres avez-vous besoin au minimum pour organiser ces six cours ?

4.2 Modélisation : graphes non orientés

Points et traits. La structure centrale de notre problème est un ensemble d'éléments (ici : des cours), dont certains sont en relation l'un avec l'autre (ici : compatibles ou incompatibles). On peut résumer cette structure par un schéma dans lequel chaque cours occupe un point de l'espace, et où certains cours sont reliés par des traits, en fonction de leur relation. Voici notre exemple précédent, où on a dessiné un lien entre les cours qui sont incompatibles.



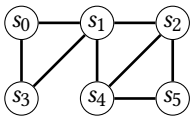
On appelle une telle structure un *graphe*.

1. Pour les besoins du scénario, on néglige les questions de capacité et de nature des salles : on suppose que toute salle peut accueillir n'importe quel cours.

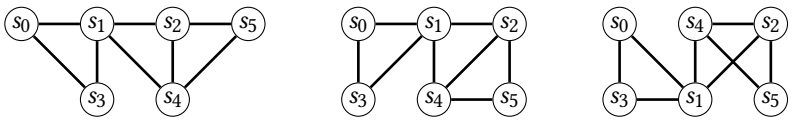
Définitions. Un ***graphe*** est formé par une paire (S, A) où :

- S est un ensemble d'éléments appelés ***sommets*** (ou *nœuds*),
- A est un ensemble d'éléments appelés ***arêtes*** (ou *arcs*, ou *flèches*),
- chaque arête $a \in A$ a deux extrémités $s, t \in S$.

On peut « dessiner » un graphe en représentant chaque sommet par un point du plan et chaque arête par un trait liant ses deux extrémités. Ci-dessous, un graphe à six sommets et huit arêtes. Il y a une arête entre les sommets s_0 et s_1, mais pas entre les sommets s_0 et s_4.

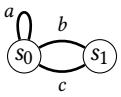


Dans un tel dessin, les positions exactes des différents sommets n'ont aucune importance, seules comptent les relations qu'entretiennent entre eux les sommets. Les dessins ci-dessous représentent tous le même graphe.



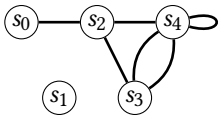
Deux sommets liés par une arête sont ***adjacents*** (ou *voisins*). Une arête a ayant un sommet s parmi ses extrémités est ***incidente*** à s.

On appelle ***boucle*** une arête dont les deux extrémités sont identiques (arête a ci-dessous). On parle d'***arêtes multiples*** (ou *parallèles*) lorsque plusieurs arêtes ont les mêmes deux extrémités (arêtes b et c ci-dessous).



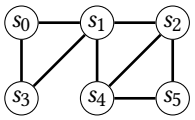
Un ***graphe simple*** est un graphe qui n'a ni boucles, ni arêtes multiples.

Le ***degré*** d'un sommet s d'un graphe simple, noté $\delta(s)$, est le nombre de voisins de ce sommet, équivalent au nombre d'arêtes ayant s pour extrémité. On généralise la notion aux graphes quelconques en comptant le nombre d'extrémités d'arêtes touchant s : des arêtes parallèles comptent chacune pour 1, et une boucle compte pour 2.



$$\begin{array}{llll} \delta(s_0) = 1 & & \delta(s_3) = 3 \\ \delta(s_1) = 0 & & \delta(s_4) = 5 \\ \delta(s_2) = 3 & & \end{array}$$

Problème du coloriage. Le problème du ***coloriage*** d'un graphe consiste à donner à chaque sommet une *couleur*, de sorte que les sommets adjacents aient toujours des couleurs différentes.



0	orange
1	violet
2	orange
3	turquoise
4	turquoise
5	violet

Notre problème d'allocation d'amphithéâtres peut se ramener au coloriage d'un graphe :

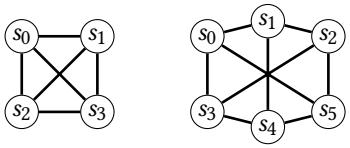
1. créer un graphe dont les sommets sont les cours auxquels affecter des salles, avec une arête entre deux sommets lorsque les cours correspondants sont incompatibles,
2. colorier le graphe en donnant des couleurs différentes aux sommets adjacents,
3. chaque couleur représente un amphithéâtre !

Remarquez que d'autres solutions sont possibles.

Si on reprend notre sélection de cours, son graphe, et le coloriage exemple précédent, il suffit donc de trois amphithéâtres :

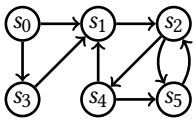
- un pour OLA et Propriétés de la matière,
- un pour Analyse de Fourier et Électromagnétisme,
- un pour Algèbre linéaire et Prog. modulaire.

Apparté : graphes planaires. Lorsque l'on peut dessiner un graphe de manière à ce que ses arêtes ne se croisent pas, on dit que ce graphe est ***planaire***. Le graphe dessiné ci-dessous à gauche est planaire : même si deux arêtes se croisent sur le dessin, il est possible d'éviter le croisement en plaçant différemment un sommet, ou en tordant une arête. Celui de droite en revanche, n'est pas planaire.



Un théorème célèbre, le *théorème des quatre couleurs*, affirme que quatre couleurs suffisent toujours pour colorier un graphe planaire.

Apparté : graphes orientés. Dans un ***graphe orienté***, chaque arête a une « direction ». On distingue ainsi pour chacune une extrémité de ***départ*** (ou *source*) et une extrémité d'***arrivée*** (ou *cible*). Dans le dessin d'un tel graphe, on représente une arête par une flèche allant du sommet de départ au sommet d'arrivée.



Pour un sommet s d'un tel graphe, le degré $\delta(s)$ se décompose en :

— un ***degré entrant*** : nombre d'arêtes dont s est l'arrivée,

— un ***degré sortant*** : nombre d'arêtes dont s est le départ.

Nous reviendrons sur la notion de graphe orienté au prochain chapitre.

4.3 Structure de données : graphe

Pour travailler avec des graphes simples non orientés, on utilisera principalement les opérations suivantes :

— énumérer les sommets,

— étant donné un sommet s, énumérer les voisins de s.

On peut par exemple imaginer le procédé suivant pour connaître le nombre total d'arêtes d'un graphe : pour chaque sommet s, compter les voisins de s, et à la fin diviser le total par 2.

Question : pourquoi divise-t-on la somme obtenue par 2 ?

Interface. Pour simplifier le code, on suppose toujours manipuler un graphe dont les sommets sont numérotés à partir de zéro. Ainsi, pour un graphe à n sommets, on désignera chaque sommet par un nombre pris dans l'intervalle $[0, n[$. Énumérer les sommets d'un graphe ou les voisins d'un sommet consiste donc à énumérer une liste d'entiers. Pour représenter un graphe, on veut donc essentiellement réaliser l'interface suivante, où `Iterable<Integer>` est le type des énumérations d'entiers pouvant être parcourues à l'aide d'une boucle *for each*.

```
interface Graphe {
    Iterable<Integer> sommets();
    Iterable<Integer> voisins(int s);
}
```

Notre fonction comptant le nombre d'arêtes d'un graphe réalisant cette interface s'écrirait ainsi.

```
static int nombreAretes(Graphe g) {
    int d = 0;
    for (int s : g.sommets()) {
        for (int v : g.voisins(s))
            d += 1;
    }
    return d/2;
}
```

Ces deux opérations d'énumération des sommets et des voisins permettent également facilement de réaliser des fonctions renvoyant le nombre de sommets d'un graphe, ou le degré d'un sommet. On supposera par la suite que ces méthodes existent bien.

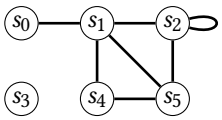
Faites-le.

Réalisation 1 : matrice d'adjacence. Un graphe G à n sommets $\{s_0, \dots, s_{n-1}\}$ et sans arêtes parallèles peut être représenté par une matrice M, carrée, d'ordre n, telle que :

— $M[i,j]$ = vrai s'il existe une arête entre s_i et s_j,

— $M[i,j]$ = faux sinon.

On appelle M la ***matrice d'adjacence*** de G. Remarque : dans les graphes *non orientés* que nous considérons ici, une arête entre s_i et s_j est aussi une arête entre s_j et s_i. Ainsi, chaque arête qui n'est pas une boucle donne deux cases à vrai dans la matrice, qui est symétrique.



	0	1	2	3	4	5
0		✓				
1	✓		✓		✓	✓
2		✓	✓			✓
3						
4		✓				✓
5		✓	✓		✓	

En java, on peut représenter une telle matrice par un simple tableau à deux dimensions. Voici le début du code. Pour créer un graphe, on se donne un constructeur prenant en paramètre le nombre de sommets et initialisant la matrice, et une méthode ajoutant un lien entre deux sommets.

```
class GrapheMat implements Graphe {
    public final int taille;
    private boolean[][] adj;

    public GrapheMat(int taille) {
        this.taille = taille;
        this.adj = new boolean[taille][taille];
    }

    public void ajouteArete(int s, int t) {
        adj[s][t] = true;
        adj[t][s] = true;
    }
```

Question : dans quel ordre sont renvoyés les voisins ici ?

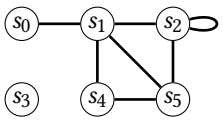
Pour énumérer les voisins d'un sommet s_i, on construit et on renvoie la liste des j tels que $M[i,j]$ vaut vrai.

```
    public Iterable<Integer> voisins(int s) {
        ArrayList<Integer> voisins = new ArrayList<>();
        for (int v=0; v<taille; v++) {
            if (adj[s][v]) voisins.add(v);
        }
        return voisins;
    }
```

Pour énumérer les sommets, c'est-à-dire les entiers de 0 à $n-1$, on pourrait de même construire une liste à la main et la renvoyer. Voici une variante dans laquelle on se passe de construire la liste. À la place, on concrétise la classe abstraite `AbstractList` en indiquant que l'élément d'indice i dans cette énumération est précisément i lui-même.

```
    public Iterable<Integer> sommets() {
        return new AbstractList<Integer>() {
            public Integer get(int index) { return index; }
            public int size() { return taille; }
        };
    }
}
```

Réalisation 2 : listes d'adjacence. Un graphe G à n sommets $\{s_0, \dots, s_{n-1}\}$ et sans arêtes parallèles peut également être représenté par un tableau A de taille n tel que $A[i]$ contient la liste des [numéros des] voisins du sommet s_i.



0	1			
1	0	2	4	5
2	1	2	5	
3				
4	1	5		
5	1	2	4	

Voici un code java pour cette seconde réalisation. Chaque liste de voisins est du type `ArrayList<Integer>`. Pour le tableau A, on utilise un nouveau `ArrayList` contenant ces listes de voisins [2].

```
class GrapheAdj implements Graphe {
    public final int taille;
    private ArrayList<ArrayList<Integer>> adj;

    public GrapheAdj(int taille) {
        this.taille = taille;
        this.adj = new ArrayList<>(taille);
        for (int s=0; s<taille; s++) adj.add(new ArrayList<>());
    }
    public void ajouteArete(int s, int t) {
        adj.get(s).add(t);
        adj.get(t).add(s);
    }
    public Iterable<Integer> voisins(int s) {
        return adj.get(s);
    }
    public Iterable<Integer> sommets() { /* identique à GrapheMat.sommets() */ }
}
```

Coût des graphes. Les deux propositions de réalisation des graphes ont des coûts différents. Si l'on considère un graphe à n sommets et k arêtes, voici les ordres de grandeur.

- Une matrice d'adjacence nécessite en mémoire un tableau de n^2 booléens. L'énumération des voisins d'un sommet a une complexité proportionnelle à n.
- Un tableau de listes d'adjacence nécessite en mémoire un tableau de n références, plus n listes contenant chacune n entiers au maximum. Plus précisément, ces listes réunies contiennent environ $2k$ éléments. L'énumération des voisins d'un sommet a une complexité proportionnelle au nombre de voisins.

Bilan : le plus souvent, la représentation par listes d'adjacence est plus efficace, autant pour l'utilisation de mémoire (car $n + k \leq n^2$) que pour le coût de l'énumération des voisins (car le nombre de voisins d'un sommet est toujours inférieur à n).

La représentation par matrice d'adjacence reste cependant à privilégier dans le cas d'un graphe *dense*, c'est-à-dire dans lequel la majorité des arêtes possibles sont présentes : la structure, plus simple, est alors plus efficace en pratique. Regardons précisément les constantes associées aux ordres de grandeur $\Theta(n^2)$ et $\Theta(n + k)$ pour la représentation en mémoire.

- Chaque booléen de la matrice d'adjacence occupe un octet, d'où un nombre d'octets $\sim n^2$ pour une matrice d'adjacence.
- Les listes d'adjacence contiennent des références vers des objets `Integer`. Chaque référence occupe huit octets, et chaque objet en java occupe en mémoire au moins 12 octets de plus que la place utilisée pour la donnée elle-même (pour `Integer`, contenant des entiers **`int`** de 4 octets, on a donc $12 + 4 = 16$ octets). En outre, chaque tableau redimensionnable `ArrayList` peut avoir une taille double du nombre d'éléments effectivement contenus. Le nombre d'octets pour un tableau de listes d'adjacence est donc compris entre $\sim 24(k + n)$ et $\sim 32(k + n)$.

Bilan : pour un graphe non orienté contenant un nombre d'arêtes proche du maximum $\sim n^2/2$, la matrice d'adjacence est plus compacte.

En outre, une matrice d'adjacence dense a un meilleur comportement vis-à-vis du cache, d'où un possible gain de vitesse.

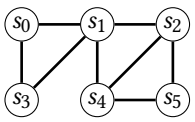
2. On pourrait vouloir utiliser un tableau primitif pour A au lieu d'un `ArrayList`, mais on ne le fait pas ici pour des raisons techniques liées au système de types de java : les tableaux primitifs ne peuvent pas, du moins en théorie, contenir des types génériques comme `ArrayList<T>`.

4.4 Algorithme : coloriage glouton

On va numéroter nos « couleurs » par des nombres entiers à partir de zéro. On considère donc l'ensemble infini de couleurs $\{c_0, c_1, c_2, \ldots\}$. Produire une coloration pour un graphe de sommets $\{s_0, \ldots, s_{n-1}\}$, c'est donc produire un tableau C de n entiers, tel que $C[i]$ est le numéro de la couleur affectée au sommet s_i.

On cherche à produire un coloriage utilisant un nombre de couleurs aussi petit que possible. Un algorithme de coloriage ***glouton*** consiste à considérer les sommets un à un, et à choisir pour chacun à son tour la couleur qui paraît la plus adaptée pour lui. On considérera comme « couleur la plus adaptée » pour un sommet s, la plus petite des couleurs qui n'a pas encore été utilisée pour un des voisins de s.

Reprenons notre exemple, et considérons les sommets dans l'ordre donné par leurs numéros.



Sommet	Couleurs voisines	Couleur choisie
s_0	aucune	c_0
s_1	$c_0(s_0)$	c_1
s_2	$c_1(s_1)$	c_0
s_3	$c_0(s_0)$, $c_1(s_1)$	c_2
s_4	$c_0(s_2)$, $c_1(s_1)$	c_2
s_5	$c_0(s_2)$, $c_2(s_4)$	c_1

En définissant c_0 comme orange, c_1 comme violet et c_2 comme turquoise, on obtient exactement le coloriage proposé plus tôt.

Code java. Dans le code, on initialise un tableau de couleurs avec le numéro `-1` pour les sommets pas encore coloriés. Lors du traitement d'un sommet, on enregistre les couleurs des sommets voisins dans un tableau de booléens. Il ne reste ensuite plus qu'à considérer chaque nombre entier à partir de `0` jusqu'à en trouver un qui n'a pas été coché dans ce tableau de booléens.

```
static int[] colorie(Graphe g) {
    // initialisation du tableau des couleurs
    int[] couleurs = new int[g.taille()];
    for (int s : g.sommets()) couleurs[s] = -1;
    // coloriage de chaque sommet
    for (int s : g.sommets()) {
        // énumération des couleurs des voisins
        int d = g.degre(s);
        boolean[] couleursVoisines = new boolean[d+1];
        for (int v : g.voisins(s)) {
            int c = couleurs[v];
            if (0 <= c && c <= d) couleursVoisines[c] = true;
        }
        // recherche de la première couleur disponible
        for (int c=0;; c++) {
            if (!couleursVoisines[c]) {
                couleurs[s] = c;
                break;
            }
        }
    }
    return couleurs;
}
```

Note : l'ensemble des couleurs voisines contient également la « fausse » couleur `-1` si l'un des voisins n'a pas encore été colorié. Cela n'a aucun impact sur le déroulement de l'algorithme.
Note : on suppose également que l'interface des graphes contient des méthodes `int taille()` et `int degre(int s)` renvoyant respectivement le nombre de sommets du graphe et le degré d'un sommet.
Remarque : dans ce code, on définit la taille du tableau de booléens des couleurs voisines en fonction du degré du sommet. Pourquoi ?

4.5 Approfondissement : analyse du coloriage glouton.

Évaluons la complexité de notre algorithme de coloriage, et la qualité du coloriage produit.

Complexité temporelle. La fonction `colorie` est la succession de deux boucles.

- La première boucle, initialisant le tableau `couleurs`, a une complexité proportionnelle au nombre n de sommets du graphe.
- La deuxième boucle, qui est la principale, réalise les opérations suivantes pour chaque sommet s.
  1. Énumération des voisins de s, pour remplir `couleursVoisines` : complexité proportionnelle au degré $\delta(s)$ de s.
  2. Recherche d'une couleur disponible : complexité encore proportionnelle à $\delta(s)$.

  D'où complexité totale de la deuxième boucle proportionnelle à la somme des degrés des sommets, c'est-à-dire proportionnelle au nombre k d'arêtes du graphe.

D'où complexité totale $\Theta(n + k)$.

Qualité du coloriage. L'algorithme glouton ne trouve pas systématiquement le plus petit nombre de couleurs possible pour le graphe auquel on l'applique. On peut cependant démontrer que le nombre χ de couleurs utilisées est borné par les degrés des sommets du graphe. Plus précisément, si on note $\Delta_{max} = \delta(s_{max})$ le degré d'un sommet s_{max} ayant le plus grand degré dans notre graphe, alors on a $\chi \leq \Delta_{max} + 1$.

Savez-vous trouver un contre-exemple ?

Démonstration. Considérons le choix d'une couleur pour un sommet s quelconque du graphe. Par définition, ce sommet a $\delta(s)$ voisins. Ces voisins utilisent donc au maximum $\delta(s)$ couleurs différentes. Ainsi, il existe au moins un nombre non utilisé dans l'intervalle $[0, \delta(s)]$: la couleur choisie pour s sera nécessairement dans cette intervalle, et en particulier inférieure ou égale à Δ_{max}. Finalement, l'algorithme glouton ne choisit que des couleurs appartenant à l'intervalle $[0, \Delta_{max}]$, et leur nombre est donc inférieur ou égal à $\Delta_{max} + 1$.

4.6 Approfondissement : relations binaires

Une *relation binaire* entre les éléments de deux ensembles A et B est un ensemble d'associations entre un élément de A et un élément de B, caractérisant des éléments qui sont « en relation » l'un avec l'autre. La nature de cette « relation » peut couvrir des situations extrêmement variées. Par exemple :

- similarité entre objets, comme l'égalité $=$,
- hiérarchie entre un tout et ses parties, comme l'appartenance $\in$,
- comparaison de grandeurs, comme la comparaison $\leq$,
- antécédents et images d'une fonction,
- dépendance entre deux événements,
- incompatibilité ou interférence entre deux faits,
- accessibilité d'un point d'arrivée depuis un point de départ...

Certains de ces exemples correspondent a des types de relations importantes en mathématiques, à savoir les relations fonctionnelles, les relations d'ordre et les relations d'équivalence, que nous aborderons progressivement. Certains aussi correspondent directement à des problèmes que l'on peut modéliser et résoudre à l'aide de graphes.

Relations binaires. Une ***relation binaire*** $\mathcal{R}$ entre deux ensembles A et B est un sous-ensemble du produit cartésien $A \times B$. Dans ce contexte, on note couramment $a\mathcal{R}b$ ou $\mathcal{R}(a, b)$ pour $(a, b) \in \mathcal{R}$. On parle de relation binaire ***homogène*** lorsque les ensembles A et B sont égaux.

Relations fonctionnelles. Une relation fonctionnelle est une relation binaire décrivant le lien entre les entrées et les sorties d'une fonction. Une telle relation $\mathcal{R}_f$ pour une fonction $f : A \longrightarrow B$ contient donc les paires (a, b) telles que $f(a) = b$.

La caractéristique principale d'une telle relation, qui définit le concept de fonction, est que la sortie $f(a)$ est uniquement déterminée par l'entrée a. Autrement dit, il ne peut pas y

avoir deux images associées au même antécédent. Une relation $\mathcal{R}_f \subseteq A \times B$ est ***fonctionnelle*** si pour tout $a \in A$ il existe au plus un $b \in B$ tel que $\mathcal{R}_f(a, b)$.

$$\forall a \in A,\ \forall b_1, b_2 \in B,\ a\mathcal{R}\,b_1 \wedge a\mathcal{R}\,b_2 \implies b_1 = b_2$$

Notez qu'avec cette définition, une fonction $f : A \longrightarrow B$ peut n'être que *partielle*, c'est-à-dire ne pas avoir de valeur $f(a)$ définie pour certaines entrées $a \in A$.
Une fonction $f : A \longrightarrow B$, définie par une relation fonctionnelle $\mathcal{R}_f \subseteq A \times B$, est :

- ***totale*** si tout élément de A a une image

  $$\forall a \in A,\ \exists b \in B,\ a\mathcal{R}_f\,b$$

- ***surjective*** si tout élément de B a au moins un antécédent

  $$\forall b \in B,\ \exists a \in A,\ a\mathcal{R}_f\,b$$

- ***injective*** si aucun élément de B n'a plus d'un antécédent

  $$\forall b \in B,\ \forall a_1, a_2 \in A,\ a_1\mathcal{R}_f\,b \wedge a_2\mathcal{R}_f\,b \implies a_1 = a_2$$

- ***bijective*** si elle est totale, surjective et injective.

Exemple : la fonction $f : \mathbb{N} \longrightarrow \mathbb{N}$ définie par $f(n) = n^2$ est donnée par la relation binaire $\mathcal{R}_f = \{(n, n^2) \mid n \in \mathbb{N}\}$. Cette fonction est totale et injective. La même fonction étendue au domaine $\mathbb{Z} \longrightarrow \mathbb{N}$ serait toujours totale mais plus injective, puisque $f(1) = f(-1) = 1$.

Relations binaires homogènes et graphes. Une relation binaire homogène $\mathcal{R} \subseteq E \times E$ est :

- ***symétrique*** si le fait pour deux éléments $e_1, e_2 \in E$ d'être en relation est indépendant de l'ordre dans lequel on les considère

  $$\forall e_1, e_2 \in E,\ e_1\mathcal{R}\,e_2 \implies e_2\mathcal{R}\,e_1$$

- ***réflexive*** si tout élément est en relation avec lui-même

  $$\forall e \in E, e\mathcal{R}\,e$$

- ***irréflexive*** si un élément n'est jamais en relation avec lui-même

  $$\forall e \in E, \neg e\mathcal{R}\,e$$

Que se passe-t-il si le graphe contient des boucles ? Et des arêtes parallèles ?

Un graphe simple (S, A) définit une relation binaire homogène $\mathcal{R}$ sur $S \times S$ par la condition « $s_1\mathcal{R}\,s_2$ s'il existe une arête $a \in A$ entre s_1 et s_2 ». Cette relation est *symétrique* et *irréflexive*. Réciproquement, toute relation binaire homogène $\mathcal{R}$ sur un ensemble E qui est symétrique et irréflexive définit un graphe simple ayant E pour ensemble de sommets et ayant une arête entre e_1 et e_2 si et seulement si $e_1\mathcal{R}\,e_2$.

5 Mettre de l'ordre

5.1 Problème : ordonnancement de tâches interdépendantes

L'université Saris-Paclay a encore besoin d'aide. L'enseignant d'un cours connu sous le nom de code « OLA » a fait la liste des outils et algorithmes qu'il allait présenter au cours du semestre, et s'est rendu compte que les dépendances entre ces différents points étaient sournoisement emmêlées. Il faut maintenant trouver un ordre dans lequel organiser les séances, de sorte que chacune ne dépende que de ce qui a déjà été vu avant.

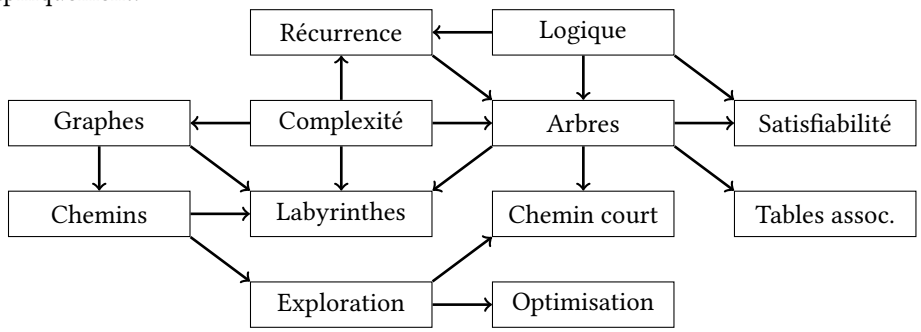
Voici un résumé de la liste, et des dépendances :

Sujet	**Dépendances** (doivent être traitées d'abord)
Graphes	Complexité
Chemins	Graphes
Récurrence	Logique, complexité
Complexité	
Labyrinthes	Graphes, arbres, chemins, complexité
Exploration	Chemins
Logique	
Arbres	Récurrence, complexité, logique
Plus court chemin	Exploration, arbres
Optimisation	Exploration
Satisfiabilité	Logique, arbres
Tables associatives	Arbres

Pouvez-vous trouver un ordre de présentation des sujets qui fasse en sorte qu'aucun ne soit abordé avant que toutes ses dépendances aient été elles-mêmes traitées ?

5.2 Modélisation : graphes orientés

On peut reprendre l'idée d'une modélisation du problème par un graphe : chaque sujet à traiter devient un sommet, et les arêtes matérialisent la relation de dépendance entre deux sujets. Différence par rapport aux graphes déjà vus : la relation de dépendance est ***orientée***. Lorsque l'on dit qu'un cours *A dépend* d'un cours *B*, cela signifie que *B* doit être programmé *avant A* : on fixe un *ordre* entre les éléments. On utilise des flèches pour représenter cela graphiquement.



Graphe orienté. La notion de ***graphe orienté*** permet d'exprimer ceci en donnant une *direction* à chaque arête. Au lieu de deux extrémités équivalentes, on a maintenant un sommet de ***départ*** (ou *source*) et un sommet d'***arrivée*** (ou *cible*). On dessine une telle arête sous la forme d'une flèche.

Pour un sommet s d'un tel graphe, le degré $\delta(s)$ se décompose en :

— un ***degré entrant*** : nombre d'arêtes dont s est l'arrivée,

— un ***degré sortant*** : nombre d'arêtes dont s est le départ.

Le degré sortant d'un sommet s correspond également au nombre d'éléments renvoyés par `voisins(`s`)`.

Chemins et cycles. Un ***chemin*** dans un graphe est une séquence C de sommets liés par des arêtes. C'est-à-dire : une séquence $C[0] \longrightarrow C[1] \longrightarrow C[2] \longrightarrow \ldots \longrightarrow C[k]$ où les $C[i]$ sont des sommets du graphe tels que pour tout $i \in [0, k[$, on a dans le graphe une arête *de* $C[i]$ *vers* $C[i+1]$. Dans un tel chemin, $C[0]$ est le ***départ***, et $C[k]$ l'***arrivée***. Le nombre k d'arêtes

Question : à quoi ressemble un chemin de longueur zéro ?

empruntées est la ***longueur*** du chemin. Note : dans un graphe orienté, les chemins doivent respecter l'orientation de chaque arête empruntée. Un ***cycle*** est un chemin dont le sommet d'arrivée est égal au sommet de départ. Un graphe ***acyclique*** est un graphe ne possédant aucun cycle.
Dans notre graphe des sujets, on a par exemple un chemin

Logique ⟶ Récurrence ⟶ Arbres ⟶ Labyrinthes

mais on n'a en revanche aucun cycle. En revanche, si l'arête entre « Complexité » et « Labyrinthes » était dans *l'autre sens*, c'est-à-dire de « Labyrinthes » vers « Complexité », alors on aurait un cycle.

Structure de données. Les structures de données que nous avons déjà vues pour les graphes sont tout à fait adaptées aux graphes orientés. Aussi bien dans une matrice d'adjacence que dans un tableau de listes d'adjacence, chaque arête non orientée entre deux sommets s et t est représentée par deux éléments : l'indication que t est un voisin de s, et celle que s est un voisin de t. Pour une arête orientée il suffit de ne garder qu'un de ces deux éléments, choisi selon la direction de l'arête. Les seules adaptations sont donc les suivantes.

— Dans l'interface `Graphe`, on précise la signification de la méthode `voisins` : un appel à `voisins(`s`)` énumère les sommets t pour lesquels il existe une arête orientée *de s vers t*.

— Dans la classe `GrapheMat`, on ajoute la méthode suivante, version simplifiée de `ajouteArete`.

```
public void ajouteAreteOrientee(int s, int t) { adj[s][t] = true; }
```

— Dans la classe `GrapheAdj`, on ajoute de même la méthode suivante.

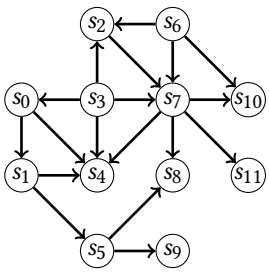
```
public void ajouteAreteOrientee(int s, int t) { adj.get(s).add(t); }
```

5.3 Algorithme : tri topologique

Le problème du ***tri topologique***, ou ***ordonnancement séquentiel***, d'un graphe orienté consiste à déterminer une séquence T contenant tous les sommets, dans un ordre tel que s'il existe une arête de s_i vers s_j, alors s_j apparaît après s_i dans T. Autrement dit, on cherche une permutation $\sigma \in \mathfrak{S}_n$ telle que s'il existe une arête de s_i vers s_j, alors $\sigma(i) < \sigma(j)$.

Principe de l'algorithme. Considérons un sommet s de degré entrant zéro, c'est-à-dire tel qu'il n'existe aucun t avec une arête $t \longrightarrow s$. Il n'y a donc aucun sommet qui doive nécessairement apparaître avant s : on peut choisir de sélectionner s en premier dans notre tri topologique. Une fois s sélectionné en premier, on peut sélectionner en deuxième tout autre sommet de degré entrant nul, *ou tout sommet dont la seule arête entrante vient de s*. Autrement dit, on peut sélectionner en deuxième tout sommet de degré entrant nul dans le graphe qu'on obtiendrait en supprimant s (et les arêtes incidentes à s).

On continue ainsi à sélectionner les sommets de degré entrant nul dans des graphes de plus en plus réduits, car on ne tient plus compte des sommets qui ont déjà été sélectionnés. Dans l'exemple ci-dessous, on appelle *sommets disponibles* ces sommets qui sont prêts à être sélectionnés, car il n'ont aucune arête entrante venant d'un sommet qui n'aurait pas déjà été sélectionné.



Étape	Disponibles	Sélectionné
1	s_3, s_6	s_6
2	s_3, s_2	s_3
3	s_0, s_2	s_2
4	s_0, s_7	s_0
5	s_1, s_7	s_1
6	s_5, s_7	s_5
7	s_7, s_9	s_7
8	$s_4, s_8, s_9, s_{10}, s_{11}$	s_{10}
9	s_4, s_8, s_9, s_{11}	s_{11}
10	s_4, s_8, s_9	s_8
11	s_4, s_9	s_9
12	s_4	s_4

Note : à chaque étape, le choix parmi les sommets disponibles est arbitraire. L'ordre donné ici en exemple correspond au plan réel du semestre.

Code java. Dans le code java suivant, on ne modifie pas le graphe lui-même pour supprimer les sommets sélectionnés (ce serait potentiellement coûteux). À la place, on crée un tableau `degreEntrant` qui renseigne le degré entrant de chaque sommet, et on décrémente les valeurs de ce tableau pour qu'elles ne comptent plus les sommets déjà sélectionnés. La boucle principale (boucle **`while`**) parcourt le tableau des sommets sélectionnés et réalise deux choses :

— pour chaque sommet s trouvé dans ce tableau, décrémenter le degré entrant des sommets v voisins de s,

— lorsque cette opération annule le degré entrant d'un sommet v, sélection de celui-ci.

On déclare que le tri topologique a réussi lorsque ce processus permet bien de sélectionner *tous* les sommets.

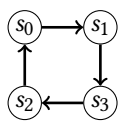
```
static int[] triTopologique(Graphe g) {
    int n = g.taille();
    // calcul du degré entrant de chaque sommet
    int[] degreEntrant = new int[n];
    for (int s : g.sommets())
        for (int v : g.voisins(s))
            degreEntrant[v] += 1;
    // initialisation avec les sommets de degré entrant nul
    int[] ordre = new int[n];
    int j = 0;
    for (int s : g.sommets())
        if (degreEntrant[s] == 0) ordre[j++] = s;
    // énumération des sommets sélectionnés
    int i = 0;
    while (i < j) {
        // degré entrant des voisins décroît, sélection si devient nul
        for (int v : g.voisins(ordre[i])) {
            degreEntrant[v] -= 1;
            if (degreEntrant[v] == 0) ordre[j++] = v;
        }
        i++;
    }
    // échec si on n'a pas sélectionné tous les sommets
    if (j < n) return null;
    return ordre;
}
```

Suivi de l'exécution de la boucle principale, sur notre graphe exemple.

		degreEntrant												
i	s	0	1	2	3	4	5	6	7	8	9	10	11	ordre $[0, j[$
		1	1	2	0	4	1	0	3	2	1	2	1	s_3, s_6
0	s_3	0	1	1	0	3	1	0	2	2	1	2	1	s_3, s_6, s_0
1	s_6	0	1	0	0	3	1	0	1	2	1	1	1	s_3, s_6, s_0, s_2
2	s_0	0	0	0	0	2	1	0	1	2	1	1	1	s_3, s_6, s_0, s_2, s_1
3	s_2	0	0	0	0	2	1	0	0	2	1	1	1	$s_3, s_6, s_0, s_2, s_1, s_7$
4	s_1	0	0	0	0	1	0	0	0	2	1	1	1	$s_3, s_6, s_0, s_2, s_1, s_7, s_5$
5	s_7	0	0	0	0	0	0	0	0	1	1	0	0	$s_3, s_6, s_0, s_2, s_1, s_7, s_5, s_4, s_{10}, s_{11}$
6	s_5	0	0	0	0	0	0	0	0	0	0	0	0	$s_3, s_6, s_0, s_2, s_1, s_7, s_5, s_4, s_{10}, s_{11}, s_8, s_9$

Note : les cinq dernières étapes ne font plus décroître les degrés, car les sommets restants s_4, s_{10}, s_{11}, s_8 et s_9 n'ont aucune arête sortante.

Cas où le tri topologique est impossible. Pour certains graphes, il n'est pas possible de trouver un tri topologique. C'est le cas notamment dès qu'un graphe contient un cycle.



5.4 Terminaison : technique du variant

La boucle principale de notre algorithme de tri topologique est une boucle **`while`**, comparant deux indices `i` et `j`. Pour que la boucle s'arrête, il faut que l'indice `i` devienne égal (ou supérieur) à l'indice `j`. Il n'est pas évident que cela arrive un jour :

— à chaque tour, `i` est incrémenté de 1,

— à chaque tour, `j` peut rester tel quel ou être incrémenté, de 1 ou plus.

Si l'indice `j` est non nul à l'origine, et est incrémenté au moins de 1 à chaque tour, il ne sera jamais rattrapé par l'indice `i` ! Nous avons besoin d'un argument plus élaboré que la simple croissance de `i` pour assurer que notre algorithme produit bien toujours un résultat en un temps fini.

En plus, si `j` devenait trop grand on échouerait à cause d'accès hors des limites du tableau `ordre`. On aimerait aussi éviter cela.

Argument intuitif : on s'attend à ce que chaque sommet du graphe ne soit ajouté qu'une fois à la liste `ordre`. La longueur de cette liste ne doit donc pas dépasser la taille n du graphe, et la boucle n'est pas censée effectuer plus de n étapes.

Argument de décroissance. Pour garantir qu'un algorithme avec une boucle s'arrête, on identifie un ***variant*** de la boucle, c'est-à-dire un nombre entier calculé en fonction des variables du programme et qui a les deux propriétés suivantes :

Ne pas confondre !
Invariant : *propriété* préservée
Variant : *quantité* décroissante

— il est positif ou nul,

— il décroît strictement à chaque tour.

Comme un nombre entier donné ne peut décroître qu'un nombre fini de fois avant de devenir négatif ou nul, on s'assure qu'une boucle dotée d'un variant ne peut pas « boucler infiniment ».

Pour notre tri topologique, on peut prendre comme variant le nombre suivant :

$$\texttt{j} - \texttt{i} + \sum_{0 \leq k < n} \texttt{degreEntrant}[k]$$

Vérifions les deux propriétés demandées.

— Ce nombre est bien positif pendant toute la durée de la boucle : d'une part la somme $\sum_{0 \leq k < n} \texttt{degreEntrant}[k]$ est une somme de nombres positifs ou nuls, et d'autre part $\texttt{j} - \texttt{i} \geq 0$ car la boucle teste elle-même `i < j`.

— D'un tour de boucle au suivant, la somme $\texttt{j} + \sum_{0 \leq k < n} \texttt{degreEntrant}[k]$ ne croît jamais, car chaque augmentation de `j` est associée à la baisse de l'un des $\texttt{degreEntrant}[k]$. À l'inverse, à chaque tour `i` augmente de 1. La valeur totale décroît donc au moins de 1.

Avec ce variant, on garantit la terminaison de la boucle sans justifier rigoureusement que chaque sommet est ajouté au plus une fois à `ordre`. On se contente ici de garantir que le nombre d'insertions dans `ordre` ne dépasse par le nombre d'arêtes du graphe. On n'a donc pas encore assuré que la taille n donnée au tableau `ordre` était suffisante.

Argument de décroissance, version plus précise. Pour comparer plus précisément `j` et n, on introduit une propriété liant `j` au nombre d'entrées inférieures ou égales à zéro dans `degreEntrant`.

$$\texttt{j} = \mathsf{card}(\{k \in [0, n[\mid \texttt{degreEntrant}[k] \leq 0\})$$

Cette propriété est préservée par la boucle **`for`** `(int v : g.voisins(ordre[i]))`. En effet, considérons un tour de cette boucle pour un sommet v.

— Si au début, $\texttt{degreEntrant}[v] > 1$, alors à la fin $\texttt{degreEntrant}[v] \geq 1$ et aucune des valeurs `j` et $\mathsf{card}(\{k \in [0, n[\mid \texttt{degreEntrant}[k] \leq 0\})$ n'a changé.

— Si au début, $\texttt{degreEntrant}[v] = 1$, alors à la fin $\texttt{degreEntrant}[v] = 0$ et les deux valeurs `j` et $\mathsf{card}(\{k \in [0, n[\mid \texttt{degreEntrant}[k] \leq 0\})$ ont augmenté de 1.

— Si au début, $\texttt{degreEntrant}[v] \leq 0$, alors à la fin $\texttt{degreEntrant}[v] \leq 0$ et aucune des valeurs `j` et $\mathsf{card}(\{k \in [0, n[\mid \texttt{degreEntrant}[k] \leq 0\})$ n'a changé (ce dernier cas ne doit pas arriver si `degreEntrant` a été correctement initialisé, mais il ne poserait pas de problème à cette propriété).

Pour assurer que notre propriété d'égalité est un invariant, il ne reste plus qu'à justifier qu'elle est vraie avant le début de la boucle. Pour cela, on remarque d'abord que l'égalité est également préservée par la boucle `while (i < j)`, puisque cette dernière n'a aucune action sur `j` ou `degreEntrant` autre que celles contenues dans la boucle `for`. Enfin, l'égalité est vraie avant le début de la boucle `while`, car la boucle précédente (`for (int s : g.sommets())`) initialise `j` précisément à la valeur demandée.

Notre invariant sur `j` en implique un autre : $\texttt{j} \leq n$. Comme en outre d'un bout à l'autre de l'algorithme $\texttt{i} \leq \texttt{j}$, on déduit un invariant sur `i` :

$$\texttt{i} \leq n$$

Avec cette propriété, on assure que la valeur $n - \texttt{i}$ est un variant de notre boucle `while` principale, et donc que celle-ci termine. On assure même que cette boucle termine après n tours au maximum.

5.5 Relations d'ordre

Une notion d'*ordre* est une manière de comparer et classer les éléments d'un ensemble.

Définition. Étant donné un ensemble E, une ***relation d'ordre*** sur E est une relation binaire homogène $\mathcal{R}$sur E qui est :

— réflexive : tout élément est comparable à lui-même

$$\forall e \in E,\ e \mathcal{R} e$$

— anti-symétrique : deux éléments distincts ne peuvent pas être comparables à la fois dans un sens et dans l'autre

$$\forall e_1, e_2 \in E,\ (e_1 \mathcal{R} e_2 \wedge e_2 \mathcal{R} e_1) \Longrightarrow e_1 = e_2$$

— transitive : la comparabilité se propage de proche en proche

$$\forall e_1, e_2, e_3 \in E,\ (e_1 \mathcal{R} e_2 \wedge e_2 \mathcal{R} e_3) \Longrightarrow e_1 \mathcal{R} e_3$$

Un ordre ***total*** est un ordre pour lequel tous deux éléments sont comparables (dans un sens ou dans l'autre).

$$\forall e_1, e_2 \in E,\ e_1 \mathcal{R} e_2 \vee e_2 \mathcal{R} e_1$$

Exemples

— Relation d'ordre usuelle $\leq$ sur un ensemble de nombres.

— Relation d'inclusion $\subseteq$ sur les parties d'un ensemble A.

— Relation de divisibilité $|$ sur les nombres entiers.

Ordre strict. Un ordre $\leq$ sur un ensemble E définit un ***ordre strict*** $<$ par la condition $e_1 < e_2 \iff (e_1 \leq e_2 \wedge e_1 \neq e_2)$. Cette relation est transitive, anti-symétrique et *ir*réflexive.

Plus petit élément, élément minimal. On considère un ensemble E et un ordre $\leq$ sur E. Étant donnés un sous-ensemble $A \subseteq E$ et un élément $x \in A$, on dit que :

— x est le ***plus petit élément*** de A si x est plus petit que tous les éléments de A

$$\forall a \in A,\ x \leq a$$

— x est le ***plus grand élément*** de A si x est plus grand que tous les éléments de A

$$\forall a \in A,\ a \leq x$$

Note : une partie A n'admet pas nécessairement de plus petit élément, mais dans le cas où un tel élément existe il est unique (de même pour le plus grand élément).

Étant donnés un sous-ensemble $A \subseteq E$ et un élément $x \in A$, on dit que :

— x est un élément ***minimal*** de A s'il n'existe pas dans A d'élément plus petit que x

$$\forall a \in A,\ a \leq x \Longrightarrow a = x$$

— x est un élément ***maximal*** de A s'il n'existe pas dans A d'élément plus grand que x

$$\forall a \in A,\ x \leq a \implies a = x$$

Note : minimal n'est pas la même chose que *plus petit*, et un élément minimal de A n'est pas nécessairement unique (de même pour maximal/plus grand).

Quelques propriétés.

— Le plus petit élément, s'il existe, est unique.

— Le plus petit élément, s'il existe, est l'unique élément minimal.

— Si l'ordre $\leq$ est total, les conditions « être le plus petit élément de A » et « être un élément minimal de A » deviennent équivalentes.

Majorants/minorants, bornes. On considère un ensemble E, un ordre $\leq$ sur E et une partie $A \subseteq E$.

— Un élément $x \in E$ est un ***minorant*** de A s'il est plus petit que tous les éléments de A

$$\forall a \in A,\ x \leq a$$

— Un élément $x \in E$ est un ***majorant*** de A s'il est plus grand que tous les éléments de A

$$\forall a \in A,\ a \leq x$$

— La ***borne inférieure*** de A est, s'il existe, le plus grand élément des minorants de A.

— La ***borne supérieure*** de A est, s'il existe, le plus petit élément des majorants de A.

Note : les majorants, minorants et bornes de A n'existent pas forcément, et dans le cas où ils existent n'appartiennent pas nécessairement à A.

5.6 Approfondissement : ordres bien fondés

La notion d'ordre permet de donner du sens à la notion de « progression » évoquée dans notre problème de justification de l'arrêt d'un algorithme : on considérera avoir progressé dès lors que l'on obtiendra quelque chose de *strictement plus petit* vis-à-vis de l'ordre choisi. En revanche, tous les ordres n'empêchent pas une telle progression de se poursuivre indéfiniment. Cette dernière propriété caractérise les ordres dits « bien fondés ».

Ordre bien fondé : définition. Un ordre $\leq$ sur un ensemble E est ***bien fondé*** s'il n'existe pas de suite infinie strictement décroissante pour $\leq$. Autrement dit, en notant $<$ l'ordre strict associé à $\leq$, il ne peut pas exister de suite $(x_k)_{k\in\mathbb{N}}$ telle que $\forall k \in \mathbb{N},\ x_{k+1} < x_k$.

Cette propriété traduit directement la notion d'arrêt cherchée.

Caractérisation alternative Un ordre $\leq$ sur un ensemble E est bien fondé si et seulement toute partie non vide de E admet un élément minimal. Autrement écrit :

$$\forall A \subseteq E,\ \ A \neq \emptyset \implies (\exists a \in A,\ \forall x \in A,\ x \leq a \implies x = a)$$

Preuve

— Supposons $(E, \leq)$ bien fondé. Soit A une partie non vide de E.
Raisonnement par l'absurde. Supposons que A n'admette pas d'élément minimal. Autrement dit, $\forall a \in A,\ \exists a' \in A,\ a' < a$. Comme A est non vide, il existe au moins un élément $a_0 \in A$. Comme $\leq$ est bien fondé, il n'existe pas de suite infinie strictement décroissante à partir de a_0. Soit $(a_k)_{k\in[0,N]}$ une suite strictement décroissante d'éléments de A à partir de a_0, qui soit la plus longue possible. Comme $a_N \in A$ et A n'admet pas d'élément minimal, il existe $a_{N+1} \in A$ avec $a_{N+1} < a_N$. Donc $(a_k)_{k\in[0,N+1]}$ est une suite strictement décroissante dans A strictement plus longue que la précédente. Contradiction, donc A doit admettre un élément minimal.

— Supposons que toute partie non vide A de E admette un élément minimal.
Raisonnement par l'absurde. Soit $(x_k)_{k\in\mathbb{N}}$ une suite infinie strictement décroissante dans E. On note A l'ensemble des valeurs de cette suite. Cet ensemble A est non vide (il contient par exemple x_0), et admet donc un élément minimal x_k. Or $x_{k+1} < x_k$ avec $x_{k+1} \in A$, ce qui contredit la minimalité de x_k. Donc il ne peut pas exister de suite infinie strictement décroissante dans E, et l'ordre $\leq$ est bien fondé.

Exemples

— Ordre usuel $\leq$ sur $\mathbb{N}$.

— Divisibilité $|$ sur $\mathbb{Z}$.

— Inclusion $\subseteq$ sur les parties d'un ensemble *fini*.

Contre-exemples

— Ordre usuel $\leq$ sur $\mathbb{Z}$: on peut descendre indéfiniment dans les négatifs.

— Ordre usuel $\leq$ sur $\mathbb{R}^+$: on peut s'approcher indéfiniment de 0 sans jamais l'atteindre.

— Inclusion $\subseteq$ sur les parties d'un ensemble infini : on peut définir une suite infinie d'ensembles qui ont toujours moins d'éléments mais restent infinis, comme la suite $([k, \infty[)_{k\in\mathbb{N}}$.

Récurrence bien fondée On considère un ensemble E, avec un ordre bien fondé $\leq$. En notant $<$ l'ordre strict associé à $\leq$ (par définition, $x < y$ si et seulement si $x \leq y \wedge x \neq y$), on a le nouveau principe de récurrence suivant.

Pour tout prédicat P sur les éléments de E, si

1. pour tout $e \in E$, $(\forall x \in E,\ x < e \implies P(x))$ implique $P(e)$,

alors pour tout élément $e \in E$ on a $P(e)$.

Autrement dit, si l'on peut déduire $P(e)$ dès lors que l'on suppose la propriété vraie pour tout les élément strictement inférieurs à e, et ceci pour chaque e, alors la propriété P est vraie pour tous les éléments de E. C'est la même idée que le principe de récurrence forte sur les entiers.

Justification du principe de récurrence bien fondée Supposons que pour tout $e \in E$, $(\forall x \in E,\ x < e \implies P(x))$ implique $P(e)$, notons A l'ensemble des éléments de $a \in E$ tels que $P(a)$ ne soit pas vraie et montrons que cet ensemble est vide en *raisonnant par l'absurde*.

Supposons cet ensemble A non vide. Comme $\leq$ est bien fondé et A non vide, il existe un élément minimal a de A. Soit $x \in E$ tel que $x < a$. Comme a est minimal dans A, nous savons que $x \notin A$, et donc que $P(x)$ est vraie. Ceci étant vrai pour pour tous les $x < a$, on déduit $P(a)$. Contradiction avec l'appartenance de a à A. Donc l'ensemble A doit être vide, et tous les éléments de E vérifient P.

5.7 Approfondissement : combinaison d'ordres

Comment jugeons-nous les propositions suivantes ?

— $(1, 2) < (3, 4)$?

— $(1, 3) < (2, 4)$?

— $(1, 5) < (2, 3)$?

— $(2, 3) < (1, 5)$?

De manière plus générale, étant donnés deux ensembles A et B, chacun avec un ordre (on pourra les noter respectivement $\leq_A$ et $\leq_B$), on cherche à ordonner les paires de $A \times B$.

Ordre produit cartésien L'***ordre produit*** sur $A \times B$ est défini par $(a_1, b_1) \leq (a_2, b_2)$ si et seulement si (a_1, b_1) est plus petit sur les deux composantes à la fois : $a_1 \leq_A a_2 \wedge b_1 \leq_B b_2$.

Par exemple :

— $(1, 2) \leq (3, 4)$

— $(1, 3) \leq (2, 4)$

— $(1, 5)$ et $(2, 3)$ sont incomparables

Note : l'ordre produit n'est donc pas total.

Si $\leq_A$ et $\leq_B$ sont deux ordres bien fondés, alors leur produit est lui aussi bien fondé.

Justification. Si on prend une suite infinie $(a_n, b_n)_{n\in\mathbb{N}}$ strictement décroissante pour l'ordre produit, alors on obtient deux suites infinies décroissantes $(a_n)_{n\in\mathbb{N}}$ et $(b_n)_{n\in\mathbb{N}}$ pour les ordres $\leq_A$ et $\leq_B$. On a plus précisément, pour tout $n \in \mathbb{N}$, d'une part $a_n \geq_A a_{n+1}$ et $b_n \geq_B b_{n+1}$, et d'autre part au moins l'une des deux conditions $a_n \neq a_{n+1}$ ou $b_n \neq b_{n+1}$. Ainsi, au moins l'une des deux suites (a_n) ou (b_n) décroît strictement infiniment souvent. Cela contredit l'hypothèse selon laquelle l'ordre correspondant ($\leq_A$ ou $\leq_B$) est bien fondé.

Ordre produit lexicographique Le ***produit lexicographique*** des deux ordres $\leq_A$ et $\leq_B$ consiste à comparer d'abord la première composante, puis à ne tenir compte de la deuxième composante qu'en cas d'égalité sur la première : on a $(a_1, b_1) \leq (a_2, b_2)$ si et seulement si $a_1 <_A a_2 \vee (a_1 = a_2 \wedge b_1 \leq_B b_2)$.

C'est selon ce principe que l'on compare deux mots dans le dictionnaire (l'ordre lexicographique est aussi appelé ***ordre du dictionnaire***).

- $(1, 2) \leq (3, 4)$
- $(1, 3) \leq (2, 4)$
- $(1, 5) \leq (2, 3)$

Note : l'ordre lexicographique est total.

Si $\leq_A$ et $\leq_B$ sont des ordres bien fondés, alors leur produit lexicographique est bien fondé également.

Justification. On va utiliser la caractérisation alternative des ordres bien fondés : toute partie non vide contient au moins un élément minimal. Soit donc $C \subseteq A \times B$ un ensemble non vide arbitraire de paires d'un élément de A et d'un élément de B. On note C_A l'ensemble des éléments de A apparaissant dans une paire de C. Formellement : $C_A = \{a \in A \mid \exists b \in B,\ (a, b) \in C\}$. Comme C n'est pas vide, C_A contient également au moins un élément. L'ordre $\leq_A$ étant bien fondé on en déduit qu'il existe un élément minimal a_0 pour $\leq_A$ dans C_A. Notons maintenant C_B l'ensemble des éléments de B apparaissant associés à a_0 dans l'ensemble C. Formellement : $C_B = \{b \in B \mid (a_0, b) \in C\}$. Cet ensemble C_B est à nouveau non vide, puisque $a_0 \in C_A$ et par définition de C_A il existe au moins un $b \in B$ tel que $(a_0, b) \in C$. L'ordre $\leq_B$ étant bien fondé on en déduit qu'il existe un élément minimal b_0 pour $\leq_B$ dans C_B. Il ne reste plus qu'à montrer que la paire (a_0, b_0) est un élément minimal de C pour l'ordre lexicographique. Soit donc $(a, b) \in C$, telle que $(a, b) \leq (a_0, b_0)$. Par définition de l'ordre lexicographique $\leq$, on a deux cas.

- Soit $a < a_0$, ce qui contredirait la minimalité de a_0 car $a \in C_A$.
- Soit $a = a_0$ et $b \leq_B b_0$. Alors $b \in C_B$, et par minimalité de b_0 on a donc $b = b_0$.

On a donc nécessairement $(a, b) = (a_0, b_0)$, et la paire (a_0, b_0) est bien minimale. Donc C admet un élément minimal, et ainsi l'ordre lexicographique est bien fondé.

5.8 Approfondissement : critère d'existence d'un tri topologique

On constate que la présence d'un cycle dans le graphe empêche tout tri topologique. Raisonnons par l'absurde : prenons un graphe contenant un cycle

$$t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \ldots \longrightarrow t_k \longrightarrow t_0$$

et supposons que ce graphe admet un tri topologique. Nécessairement, le tri topologique fait intervenir tous les sommets de ce cycle. Notons t_i celui qui apparaît en premier. En particulier, il apparaît *avant* t_{i-1}, ce qui contredit l'existence d'une arête $t_{i-1} \longrightarrow t_i$ (dans le cas où $i = 0$, on remplace dans ce raisonnement $i - 1$ par k).

Critère d'existence d'un tri topologique. Inversement, on peut démontrer le fait suivant.

Tout graphe sans cycle admet un tri topologique.

On commence par démontrer le lemme suivant : *tout graphe acyclique non vide admet au moins un sommet de degré entrant 0.*

Preuve par l'absurde. Soit G un graphe acyclique non vide tel que tous les sommets de G ont un degré entrant strictement positif. On va démontrer par récurrence que $\forall n \in \mathbb{N}$, le graphe G admet un chemin de longueur n :

- Cas de base : G étant non vide il contient un sommet s, et un chemin de longueur 0 de s à s.
- Itération : Soit $n \in \mathbb{N}$ tel que G admette un chemin de longueur n, et soit $t_0 \longrightarrow t_1 \longrightarrow \ldots \longrightarrow t_n$ un tel chemin de longueur n dans G. Le sommet de départ t_0 de ce chemin a un degré entrant non nul, il existe donc une arête $s \longrightarrow t_0$ ayant t_0 pour arrivée. On peut donc construire un chemin $s \longrightarrow t_0 \longrightarrow t_1 \longrightarrow \ldots \longrightarrow t_n$ de longueur $n + 1$ dans G.

Donc $\forall n \in \mathbb{N}$, le graphe G admet un chemin de longueur n. En particulier, en notant N le nombre de sommets de G, on sait que G admet un chemin de longueur $N+1$. Par le principe des tiroirs il existe un sommet s de G par lequel ce chemin passe deux fois. On extrait du chemin la séquence comprise entre les deux premières occurrences de s pour obtenir un chemin de s à s, c'est-à-dire un cycle. Contradiction avec l'hypothèse selon laquelle G est acyclique. Donc G admet nécessairement un sommet de degré entrant 0.

Retour au théorème : *tout graphe orienté acyclique admet un tri topologique.*

Preuve par récurrence sur le nombre de sommets du graphe. On note $P(n)$ la propriété : « tous les graphes acycliques à n sommets admettent un tri topologique ».

- Initialisation (preuve de $P(0)$) : un graphe vide est trié topologiquement par la séquence vide.
- Hérédité (preuve de $\forall n \in \mathbb{N},\ P(n) \Longrightarrow P(n+1)$). Soit n tel que $P(n)$ soit vraie, et soit G un graphe acyclique à $n+1$ sommets $\{s_0, \dots, s_n\}$. Par notre lemme, G admet un sommet s_i de degré entrant 0. Le sous-graphe G' obtenu en retirant de G ce sommet s_i et ses arêtes incidentes a n sommets, et est de plus acyclique (supposons qu'il existe un cycle dans G', alors ce cycle existerait également dans G ; or G est acyclique : contradiction). On peut donc appliquer l'hypothèse de récurrence à G' pour obtenir un tri topologique de G', c'est-à-dire une séquence $t_0, t_1, \dots, t_{n-1}$ des sommets $\{s_0, \dots, s_n\} \backslash \{s_i\}$ compatible avec l'orientation des arêtes. Alors la séquence $s_i, t_0, t_1, \dots, t_{n-1}$ obtenue en ajoutant le sommet s_i en tête de la précédente est un tri topologique du graphe G complet.

  En effet, soit une arête $s \longrightarrow t$ quelconque de G, montrons que s apparaît bien avant t dans la séquence $s_i, t_0, t_1, \dots, t_{n-1}$.

  - Si ni s ni t n'est égal à s_i, alors l'arête $s \longrightarrow t$ apparaît dans le sous-graphe G', et par hypothèse les sommets s et t apparaissent dans le bon ordre dans le tri topologique $t_0, t_1, \dots, t_{n-1}$ de G'.
  - Si $s = s_i$, alors $s = s_i$ apparait bien avant $t = t_j$, dans la séquence $s_i, t_0, t_1, \dots, t_{n-1}$.
  - Il n'est pas possible que $t = s_i$, car l'existence de l'arête $s \longrightarrow s_i$ contredirait l'hypothèse selon laquelle le sommet s_i a un degré entrant 0.

6 Trouver la voie

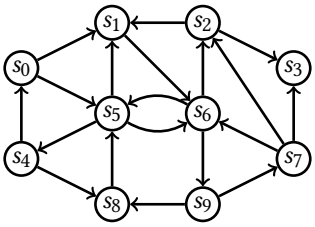
6.1 Problème : recherche de chemin

On se donne un graphe, un sommet de départ, et un sommet cible. Question : le sommet cible est-il accessible à partir du sommet de départ ? C'est-à-dire : existe-t-il un chemin allant du sommet de départ ou sommet cible, en suivant les arêtes du graphe ? Si oui, peut-on indiquer un tel chemin ? Et si plusieurs chemins sont possibles, peut-on privilégier les chemins les plus courts ? Dans ce chapitre, nous cherchons donc à *explorer* un graphe.

6.2 Algorithme : parcours en profondeur

Pour parcourir un graphe, on peut suivre une stratégie très simple : avancer dans une direction quelconque jusqu'à aboutir à un cul-de-sac, puis revenir sur ses pas jusqu'au dernier embranchement et choisir une nouvelle direction (c'est-à-dire une direction qui n'a pas encore été explorée).

On peut voir ceci comme une stratégie récursive : pour explorer un graphe à partir d'un sommet s, on considère tour à tour toutes les arêtes $s \longrightarrow t_i$, et on explore récursivement à partir de chacune des cibles t_i prises à tour de rôle. Attention cependant : il est possible lors de cette exploration récursive de retourner à un sommet à partir duquel l'exploration a déjà été faite, voire de retourner au point de départ. Dans ce cas, on ne veut pas refaire l'exploration déjà faite, et encore moins tourner en rond. On mémorise donc les sommets déjà vus, afin de ne pas les explorer à nouveau.



Exemple d'exploration. On prend pour point de départ le sommet s_6.

1. Explorer s_6, puis s_2, puis s_1.
2. Le seul successeur possible est s_6, déjà exploré, revenir au sommet précédent s_2 et choisir une autre voie. Explorer s_3.
3. Aucun successeur, revenir au précédent s_2. Aucune voie inexplorée restante, revenir au précédent s_6 et choisir une autre voie.
4. Explorer s_5 puis s_4 (note : le successeur s_1 de s_5 a déjà été exploré), puis s_0.
5. Les seuls successeurs possibles sont s_1 et s_5, déjà explorés, revenir au sommet précédent s_4 et choisir une autre voie. Explorer s_8.
6. Unique successeur s_5 déjà exploré, revenir au sommet précédent s_4. Aucune voie inexplorée restante, revenir encore au précédent s_5. Aucune voie inexplorée restante, revenir encore au précédent s_6 et choisir une autre voie. Explorer s_9, puis s_7.
7. Aucun successeur non exploré, revenir au précédent s_9. Aucune voie inexplorée restante, revenir au précédent puis s_6. Aucune voie inexplorée non plus, et pas non plus de précédent restant. Arrêt.

Code java. Pour marquer les sommets déjà explorés, on peut utiliser un tableau `vu` contenant un booléen par sommet du graphe, et tel que `vu[i]` vaut **`true`** si et seulement si s_i a effectivement été rencontré. Alors, avant tout appel récursif on peut vérifier si le sommet considéré n'a pas déjà été visité. La fonction principale `dfs` initialise ce tableau de booléens, puis appelle la fonction récursive `explore` réalisant notre stratégie d'exploration. À la fin le tableau `vu` indique, pour chaque numéro i, si le sommet s_i est accessible par un chemin à partir de la source `s`.

```
private static boolean[] vu;

private static void explore(Graphe g, int s) {
    vu[s] = true;
    for (int v : g.voisins(s)) {
        if (!vu[v]) explore(g, v);
    }
}

static boolean[] dfs(Graphe g, int s) {
    vu = new boolean[g.taille()];
    explore(g, s);
    return vu;
}
```

Voici le détail de l'exécution de `dfs` sur l'exemple précédent. On note `explore` k pour un appel récursif sur le sommet s_k, et `ignore` k lorsque le sommet s_k est examiné dans une énumération de voisins, mais ignoré car déjà marqué.

	Sommets vus									
Action	0	1	2	3	4	5	6	7	8	9
explore 6							✓			
+-- explore 2			✓				✓			
| +-- explore 1		✓	✓				✓			
| | +-- ignore 6		✓	✓				✓			
| +-- explore 3		✓	✓	✓			✓			
+-- explore 5		✓	✓	✓		✓	✓			
| +-- ignore 1		✓	✓	✓		✓	✓			
| +-- explore 4		✓	✓	✓	✓	✓	✓			
| | +-- explore 0	✓	✓	✓	✓	✓	✓	✓			
| | | +-- ignore 1	✓	✓	✓	✓	✓	✓	✓			
| | | +-- ignore 5	✓	✓	✓	✓	✓	✓	✓			
| | +-- explore 8	✓	✓	✓	✓	✓	✓	✓		✓	
| | | +-- ignore 5	✓	✓	✓	✓	✓	✓	✓		✓	
| +-- ignore 6	✓	✓	✓	✓	✓	✓	✓		✓	
+-- explore 9	✓	✓	✓	✓	✓	✓	✓		✓	✓
+-- explore 7	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
+-- ignore 2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
+-- ignore 3	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

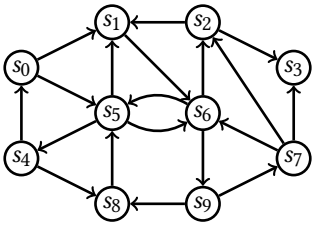
Ce mode d'exploration est appelé ***parcours en profondeur***, et est caractérisé comme suit : on s'avance aussi loin que possible sur un chemin donné, pour ne revenir sur nos pas qu'une fois un cul-de-sac atteint. Ceci est lié à l'emboîtement des appels récursifs : un appel donné à `explore(g, s)` ne se termine qu'une fois que toute l'exploration à partir de `s` est faite.

Question : que se passerait-il si l'instruction `vu[s] = true;` *était placée à la fin de l'exploration plutôt qu'au début ?*

6.3 Algorithme : parcours en largeur

Partant d'un sommet s donné, le parcours en profondeur consiste à choisir *une* voie et à la suivre jusqu'au bout avant de considérer les autres voies qui étaient possibles. Alternativement, on peut vouloir suivre toutes les voies en parallèle, en progressant en cercles concentriques autour de s : d'abord tous les voisins immédiats, puis les sommets accessibles depuis l'ensemble de ces voisins, puis les sommets accessibles depuis l'ensemble de ces suivants, etc. On parle ici de ***parcours en largeur***.

Exemple d'exploration. On part du sommet s_6.



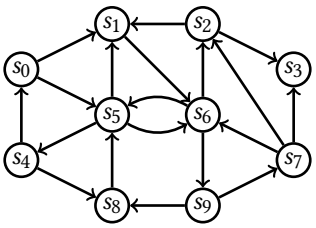
1. On regarde la source s_6. Ses voisins immédiats sont s_2, s_5 et s_9.
2. On regarde à tour de rôle s_2, s_5 et s_9, et on découvre les nouveaux sommets s_1, s_3, s_4, s_7 et s_8.
3. On regarde à tour de rôle les cinq précédents, et on découvre encore un nouveau sommet s_0.
4. Après s_0, on ne trouve plus de sommets non encore découverts : arrêt.

Code java. Il n'est plus question ici d'exploration récursive. L'analyse d'un sommet s du k-ème cercle consiste à observer ses voisins v, et à les enregistrer comme devant être analysés à l'étape $k+1$ (du moins, ceux des voisins qui n'ont pas déjà été vus). En pratique, il n'est pas nécessaire de matérialiser la transition entre les différents « cercles » : on stocke les sommets du k-ème et du $k+1$-ème cercle dans une même *file* de sommets en attente, et on traite un par un les sommets pris dans cette file. La discipline de file, dite *fifo* (« first in, first out »), assure que les premiers sommets analysés sont ceux qui ont été enregistrés en premier. Alors, tous les sommets du k-ème cercle seront analysés avant tous les sommets du $k+1$-ème cercle, eux-mêmes analysés avant tous les sommets du $k+2$-ème cercle, etc. À nouveau, la fonction renvoie le tableau de booléens identifiant les sommets accessibles depuis la source `s`.

Dans le code proposé ici, on remplace le tableau de booléens `vu` par un tableau d'entiers `dist`, tel que `dist[`s`]` vaut `-1` pour un sommet s non visité, et k, pour un sommet visité du k-ème cercle. On obtient donc une information plus précise : pas seulement d'accessibilité, mais de distance par rapport à la source.

```
static int[] bfs(Graphe g, int s) {
    int[] dist = new int[g.taille()];
    for (int t : g.sommets()) dist[t] = -1;
    Queue<Integer> enAttente = new ArrayDeque<>();
    dist[s] = 0;
    enAttente.add(s);
    while (!enAttente.isEmpty()) {
        int t = enAttente.remove();
        for (int v : g.voisins(t)) {
            if (dist[v] < 0) {
                dist[v] = dist[t] + 1;
                enAttente.add(v);
            }
        }
    }
    return dist;
}
```

Voici le détail de l'exécution de `bfs` sur l'exemple précédent. La file apparaît comme une ligne dans laquelle les éléments sont ajoutés par la droite et retirés par la gauche. Note : l'ordre dans lequel on considère les successeurs d'un sommet est a priori arbitraire.



										dist									
t	**voisins**	**enAttente**								0	1	2	3	4	5	6	7	8	9
		6								–	–	–	–	–	–	0	–	–	–
6	**2**, **5**, **9**		**2**	**5**	**9**					–	–	1	–	–	1	0	–	–	1
2	**1**, **3**			5	9	**1**	**3**			–	2	1	2	–	1	0	–	–	1
5	1, **4**, 6				9	1	3	**4**		–	2	1	2	2	1	0	–	–	1
9	**7**, **8**					1	3	4	**7**	**8**	–	2	1	2	2	1	0	2	2	1
1	6						3	4	7	8	–	2	1	2	2	1	0	2	2	1
3	∅							4	7	8	–	2	1	2	2	1	0	2	2	1
4	**0**, 8								7	8	**0**	3	2	1	2	2	1	0	2	2	1
7	6, 2, 3									3	0	3	2	1	2	2	1	0	2	2	1
8	5										0	3	2	1	2	2	1	0	2	2	1
0	1, 5											3	2	1	2	2	1	0	2	2	1

6.4 Comparaison des deux parcours

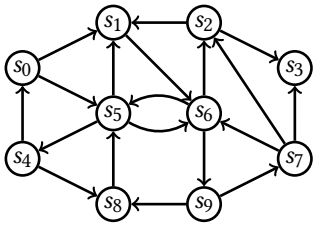
On peut ramener les parcours en profondeur et en largeur a une structure commune, mais l'un et l'autre n'explorent pas les sommets dans le même ordre.

Parcours en profondeur, sans récurrence. On peut obtenir une autre manière d'écrire un parcours en profondeur en prenant le code du parcours en largeur, et en remplaçant la file d'attente par une *pile*. La discipline de pile, dite *lifo* (« last in, first out »), fait que le premier sommet analysé est celui qui a été enregistré en dernier. Autrement dit, on poursuit d'abord dans la direction ouverte par le sommet courant, avant de revenir aux autres sommets du même cercle.

```
static boolean[] dfs(Graphe g, int s) {
    boolean[] vu = new boolean[g.taille()];
    Deque<Integer> enAttente = new ArrayDeque<>();
    vu[s] = true;
    enAttente.push(s);
    while (!enAttente.isEmpty()) {
        int t = enAttente.pop();
        for (int v : g.voisins(t)) {
            if (!vu[v]) {
                vu[v] = true;
                enAttente.push(v);
            }
        }
    }
    return vu;
}
```

Exemple d'exécution sur l'exemple précédent. La pile apparaît comme une ligne dans laquelle les éléments sont ajoutés et retirés par la droite. Note : l'ordre dans lequel on considère les successeurs d'un sommet est a priori arbitraire.

Question : que se passerait-il si on remplaçait les deux instructions `vu[s] = true;` *et* `vu[v] = true;` *par une unique instruction* `vu[t] = true;` *placée juste après* `s = enAttente.pop()` *?*

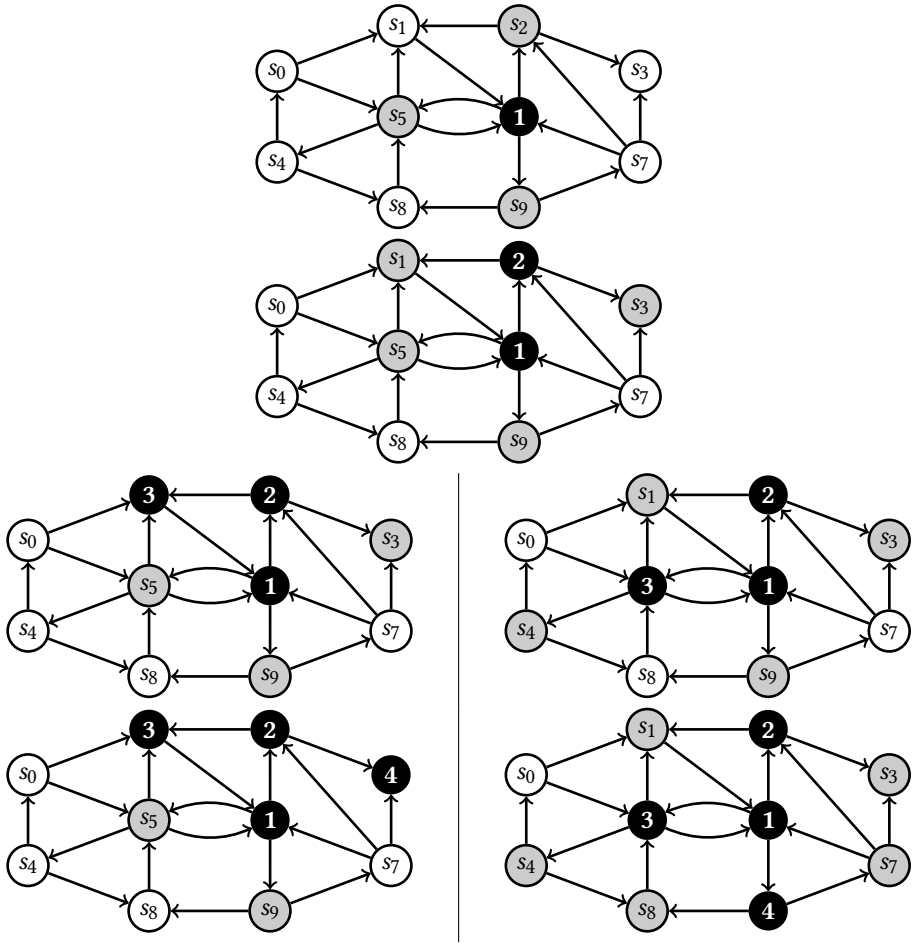


t	voisins	enAttente			
		6			
6	**9**, **5**, **2**	**9**	**5**	**2**	
2	**3**, **1**	9	5	**3**	**1**
1	6	9	5	3	
3	∅	9	5		
5	6, **4**, 1	9	4		
4	**8**, **0**	9	**8**	**0**	
0	5, 1	9	8		
8	5	9			
9	8, **7**	**7**			
7	6, 3, 2				

Note : dans la version récursive, une « pile » était bien présente. Mais où ?

Comparaison des deux parcours. Sur ces schémas, on représente en noir les sommets déjà explorés, en gris les sommets déjà vus mais pas encore explorés, et en blanc et les sommets pas encore vus. Les schémas de gauche correspondent à l'état du parcours en profondeur et ceux de droite à l'état du parcours en largeur, après 3 puis 4 sommets explorés. Les nombres en blanc dans les sommets noirs indiquent l'ordre dans lequel les sommets ont été explorés.

On peut observer que le parcours en profondeur (à gauche) visite en troisième et en quatrième deux sommets qui étaient accessibles depuis le deuxième sommet visité (mais pas depuis la source). À l'inverse, le parcours en largeur (à droite) explore en premier les trois voisins immédiats de la source.



6.5 Reconstruction de chemins.

Une petite modification de nos algorithmes de parcours permet de reconstruire un chemin de la source vers n'importe quel sommet accessible. On modifie pour cela le tableau de booléens `vu` en un tableau de sommets `pred` tel que :

- pour tout sommet s_i non vu, `pred[`i`]` vaut `-1`,
- pour tout sommet s_i vu, `pred[`i`]` contient le numéro du sommet s_j depuis lequel on a vu s_i,
- cas particulier pour la source s_{i_0} elle-même : on initialise `pred[`i_0`]` $= i_0$.

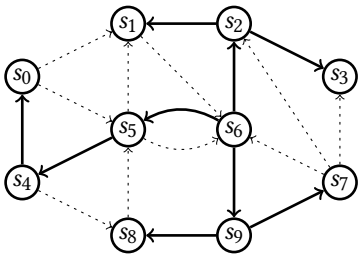
Ainsi, lorsque `pred[`i`]` est un entier positif $j \neq i$, on sait qu'on a une arête $s_j \longrightarrow s_i$, *et que le sommet s_j est lui-même accessible depuis la source.* En outre, seul le sommet source vérifie `pred[`i`]` $= i$. À l'inverse, `pred[`i`]` vaut `-1` exactement dans les cas où `vu[`i`]` aurait valu **`false`**. On ajoute pour finir une fonction qui, à partir de ce tableau des prédécesseurs, reconstruit le chemin complet. Cette adaptation s'applique aussi bien au parcours en profondeur qu'au parcours en largeur. Voici la version basée sur `bfs`.

```
public static int[] bfs(Graphe g, int s) {
    int[] pred = new int[g.taille()];
    for (int t : g.sommets()) pred[t] = -1;
    Queue<Integer> enAttente = new ArrayDeque<>();
    pred[s] = s;
    enAttente.add(s);
    while (!enAttente.isEmpty()) {
        int t = enAttente.remove();
        for (int v : g.voisins(t)) {
            if (pred[v] < 0) {
                pred[v] = t;
                enAttente.add(v);
            }
        }
    }
    return pred;
}

public static List<Integer> chemin(Graphe g, int s, int t) {
    int[] pred = bfs(g, s);
    LinkedList<Integer> chemin = new LinkedList<>();
    chemin.add(t);
    while (t != s) {
        t = pred[t];
        chemin.addFirst(t);
    }
    return chemin;
}
```

Dans le tableau de prédécesseurs fourni par un tel parcours, la source n'a pas de prédécesseur, les sommets du premier cercle ont pour prédécesseur la source, les sommets du deuxième cercle ont pour prédécesseur un sommet du premier cercle, etc. Pour notre graphe exemple, et une exploration en largeur des sommets dans l'ordre 6, 2, 5, 9, 1, 3, 4, 7, 8, 0, le tableau des prédécesseurs serait celui-ci.

0	1	2	3	4	5	6	7	8	9
4	2	6	2	5	6	6	9	9	6

On peut également visualiser ceci par une relation « prédécesseur », relation binaire sur les sommets du graphe qui ne retient que les arêtes qui ont réellement été utilisées lors du parcours.



6.6 Approfondissement : analyse du parcours en profondeur

On considère ici l'algorithme `dfs` donné page 40, c'est-à-dire l'exploration récursive.

Complexité. Considérons un graphe avec n sommets et k arêtes.

Première remarque : la fonction `explore` est appelée au plus une fois par sommet du graphe. En effet, elle n'est appelée que sur des sommets s vérifiant `vu`$[s]$ = `false`, et modifie cette valeur en `true` avant toute autre chose. Une fois la première instruction de l'appel réalisée elle ne peut donc plus être appelée une deuxième fois sur le même sommet.

Dans chaque appel à `explore` on a les opérations suivantes :

— modification de `vu` pour le sommet courant,

— énumération des voisins,

— consultation de `vu` pour chaque voisins,

— éventuels appels récursifs.

Le coût propre d'un appel, ne tenant compte que des opérations de l'appel lui-même et pas de ses sous-appels récursifs, est donc proportionnel au nombre de voisins du sommet (son degré).

L'ordre de grandeur maximum du coût total est donc donné par le nombre n de sommets (pour les appels récursifs) et le nombre k d'arêtes (pour le cumul des voisins énumérés dans chaque appel). D'où une complexité $O(n + k)$.

Le coût réel peut être inférieur dans le cas où une grande part des sommets *n'est pas* accessible depuis la source.

Spécification et correction. Spécification : l'algorithme `dfs` prend en entrée un graphe g, et un sommet s valide de g, et renvoie un tableau A tel que pour tout sommet t de G, on a $A[t]$ = `true` si et seulement si t est accessible depuis s par un chemin de g.

Montrons, par récurrence forte sur le nombre total d'appels récursifs déclenchés, qu'un appel `explore(`g`, `s`)` ne marque que des sommets accessibles depuis s.

— Cas de base : aucun appel récursif. Alors le seul sommet marqué est s lui-même, qui est bien accessible depuis s par le chemin vide.

— Cas héréditaire : les sommets marqués lors de l'appel `explore(`g`, `s`)` sont s lui-même, et les sommets marqués par les appels récursifs immédiats `explore(`g`, `v_i`)` effectués pour un certain nombre de voisins $\{v_1, \dots, v_k\}$ de s. Ces appels récursifs immédiats sont inclus dans l'appel principal : chacun déclenche un nombre total d'appels récursifs inférieur d'au moins 1 au total de l'appel principal, et on peut leur appliquer l'hypothèse de récurrence. Ainsi, chaque appel `explore(`g`, `v_i`)` ne marque que des sommets accessibles depuis v_i. Or, un sommet accessible depuis v_i est également accessible depuis s, puisqu'on a une arête $s \longrightarrow v_i$.

Montrons que, pour tout sommet t tel qu'il existe un chemin $s \longrightarrow \dots \longrightarrow t$, le sommet t est marqué après l'appel `explore(`g`, `s`)`. On le montre par récurrence sur la longueur du chemin.

— Cas de base, pour la longueur 0 : cela signifie que $t = s$, et ce sommet est bien marqué immédiatement.

— Pour un chemin de longueur $n+1$, on décompose en $s \longrightarrow \dots \longrightarrow u \longrightarrow t$ où le chemin de s à u est de longueur n. Par hypothèse de récurrence, u est bien marqué, ce qui signifie qu'un appel `explore(`g`, `u`)` a été déclenché. Lors de cet appel le sommet t, voisin de u, a été examiné. Alors soit t était déjà marqué, soit on a eu un appel `explore(`g`, `t`)` qui a bien marqué le sommet.

Note : pour être parfaitement rigoureux dans les hypothèses de cette preuve, on suppose que les sommets ne peuvent être marqués que *par la fonction* `explore`.

Avec ces deux points, on a bien montré que l'appel `explore(g, s)` réalisé dans la fonction `dfs` marque bien exactement les sommets accessibles depuis `s`.

6.7 Approfondissement : analyse du parcours en largeur

On considère ici l'algorithme `bfs` donné page 42, c'est-à-dire l'exploration avec une file *fifo*.

Complexité. Considérons un graphe avec n sommets et k arêtes.

Remarquons d'abord que chaque sommet ne peut être ajouté qu'une seule fois à la file `enAttente`. En effet, cet ajout est soumis à un test préalable de distance non définie, et la distance est justement définie au moment même où le sommet est ajouté, empêchant tout nouvel ajout du même sommet. Remarquons en passant que, de même, une distance définie dans le tableau `dist` n'est jamais modifiée par la suite.

Chaque tour de la boucle **`while`** traite un nouveau sommet de la file `enAttente`. On a donc au maximum n tours, pour chacun des n sommets. Le coût d'un tour de boucle donné est proportionnel au nombre de voisins du sommet considéré. Le coût total est donc $O(n + k)$, comme pour le parcours en profondeur.

Spécification et correction. Spécification : l'algorithme `bfs` prend en entrée un graphe g et un sommet s valide de g, et renvoie un tableau d'entiers D tel que pour tout sommet t de g, on a $D[t] \geq 0$ si et seulement si t est accessible depuis s et à distance $D[t]$ (c'est-à-dire que le plus petit nombre d'arêtes d'un chemin de s à t est $D[t]$), et $D[t] = -1$ sinon.

Notons d_t la distance de la source au sommet t, c'est-à-dire le plus petit nombre d'arêtes d'un chemin de s vers t, en posant $d_t = \infty$ lorsque t n'est pas atteignable. Pour un état donné de la file `enAttente`, notons d la distance `dist`$[t]$ renseignée pour le premier sommet de la file, si la file est non vide. La correction est obtenue à l'aide des invariants suivants pour la boucle **`while`** de l'algorithme.

1. La file `enAttente` est constituée :
   - d'abord d'une séquence de sommets à distance d,
   - puis d'une séquence de sommets à distance $d + 1$, qui sont exactement les sommets à distance $d + 1$ voisins des sommets à distance d absents de la première partie.
2. Tout sommet t à distance $d_t \leq d$ ou qui est présent dans la file `enAttente` est tel que `dist`$[t] = d_t$.
3. Tour sommet t à distance $d_t > d$ et qui n'est pas présent dans la file `enAttente` est tel que `dist`$[t] = -1$.

Ces propriétés sont bien valides avant le premier tour de boucle : la file `enAttente` contient alors exclusivement le sommet s, qui est l'unique sommet à distance zéro de lui-même.

Supposons les propriétés vraies au début d'un tour de boucle. L'algorithme considère alors le sommet t en tête de la file `enAttente`, qui par définition est à distance d de la source. On énumère ensuite chaque voisin v de t.

- Si v est tel que `dist`$[v] \geq 0$, alors l'algorithme ne fait rien.
- Sinon, par hypothèse on a $d_v > d$, et l'algorithme définit `dist`$[v] = d + 1$ et ajoute v à la file. Cette action est correcte, car on a bien un chemin $s \longrightarrow \ldots \longrightarrow t \longrightarrow v$ de longueur $d + 1$ de la source vers v.

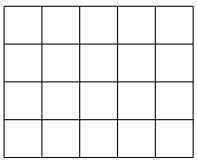
À la fin du tour, t a été retiré de la file, et tous les voisins v de t tels que $d_v = d + 1$ qui n'étaient pas déjà dans la file y ont été ajoutés. Remarque : si t était le dernier sommet de la file à distance d, alors la file est maintenant constituée exactement des sommets à distance $d + 1$.

L'algorithme s'arrête lorsque la file est vide. En notant d la distance du dernier sommet traité, cela signifie que le graphe ne contient aucun sommet à distance $d + 1$, et donc aucun sommet à une distance $> d$. Ainsi, tout sommet d à distance d_t finie est bien tel que `dist`$[t] = d_t$.

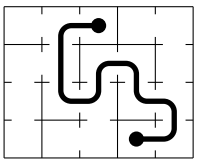
7 Se perdre

7.1 Problème : création d'un labyrinthe

Partons d'un terrain rectangulaire, formé d'une multitude de petites salles carrées séparées par des cloisons.

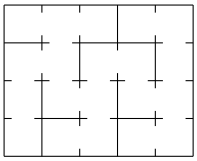


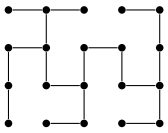
On souhaite ouvrir des portes dans certaines des cloisons, de sorte à former un *labyrinthe parfait* : on veut que quel que soit le choix d'une salle de départ et d'une salle d'arrivée il existe un unique itinéraire permettant d'aller de l'une à l'autre.



Note : quand on mentionne un « unique itinéraire » ici, on écarte implicitement les itinéraires qui contiendraient des rebroussements.

Un tel labyrinthe peut être vu comme un graphe, dont les sommets sont les salles, et les arêtes sont les portes entre deux salles. Ce graphe est non orienté.





Un « itinéraire » entre deux salles du labyrinthe est un chemin entre les sommets correspondants. L'existence hypothétique de plusieurs chemins entre deux sommets signalerait la présence d'un cycle dans le graphe. On peut donc reformuler notre objectif : partant d'un ensemble de sommets, créer un graphe *connexe* (tous les sommets peuvent être reliés deux à deux par des chemins) et *sans cycle*, en sélectionnant des arêtes parmi un ensemble d'arêtes autorisées (ici, une arête doit correspondre à une porte entre deux salles géographiquement adjacentes).

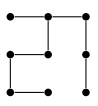
On propose de suivre la stratégie suivante : énumérer toutes les cloisons dans un ordre aléatoire, et pour chacune, y ouvrir une porte si cela ne fait pas apparaître de cycle. On pourrait imaginer une réalisation naïve de cette stratégie, en parcourant le graphe à la recherche d'un cycle à chaque étude d'une potentielle nouvelle arête : on aurait un parcours de coût linéaire en le nombre de salles, répété pour chaque arête potentielle. *Nous allons voir à la place une structure, qui permet de détecter en temps quasiment constant si l'ajout d'une arête crée un cycle.*

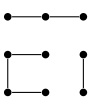
7.2 Composantes connexes d'un graphe

Connexité. Un graphe G non orienté est ***connexe*** si tous les sommets de G peuvent être reliés deux à deux par un chemin.

$$\forall s, s' \in G, \exists p \in \mathsf{chemins}(G), p : s \longrightarrow s'$$

Visuellement, un graphe connexe est un graphe « en un seul morceau » (à gauche), tandis qu'un graphe non connexe est un graphe comportant des parties isolées les unes des autres (à droite).





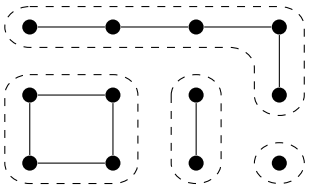
Ces « parties » sont formalisées par la notion de *composante connexe* que nous allons maintenant détailler.

Composante connexe. Dans un graphe G non orienté, une ***composante connexe*** est un sous-ensemble C *non vide* des sommets de G tel que :

1. tous les sommets de C sont connectés deux à deux par des chemins,
2. aucun sommet de C n'est connecté à un sommet hors de C.

Remarque : le critère 2 implique que les chemins du critère 1 utilisent exclusivement des sommets de C.

Visuellement, les composantes connexes sont les différents « morceaux » d'un graphe qui ne serait pas lui-même connexe.



Formellement, on définit une composante connexe de G comme un *sous-graphe connexe maximal* de G. Précisons le vocabulaire de cette définition.

— Un ***sous-graphe*** d'un graphe $G = (S, A)$ est un graphe $G' = (S', A')$ où S' est un sous-ensemble des sommets de G, et A' est l'ensemble des arêtes de G dont les extrémités sont dans S'.

$$\begin{cases} S' \subseteq S \\ A' = \{a \in A \mid \exists s, s' \in S', a : s \longrightarrow s'\} \end{cases}$$

Remarque : pour former un sous-graphe, on peut choisir un sous-ensemble $S' \subseteq S$ arbitraire. En revanche, une fois S' choisi, l'ensemble A' d'arêtes est fixé (*toutes* les arêtes de G liées aux sommets choisis).

— Dans cette définition, ***maximal*** est utilisé relativement à l'ordre d'inclusion sur les sommets : la clause de maximalité indique donc qu'un sous-graphe connexe de G contenant tous les sommets d'une composante C ne peut être que C elle-même. Autrement dit : si C est une composante connexe de G, alors un sous-graphe $C' \subseteq G$ dans lequel C est inclus au sens strict ne peut pas être connexe.

$$\begin{cases} \forall s, s' \in C, \exists p \in \mathsf{chemins}(C), p : s \longrightarrow s' & \text{(connexité)} \\ \forall C', C \subsetneq C' \Longrightarrow \exists s, s' \in C', \neg\exists p \in \mathsf{chemins}(C'), p : s \longrightarrow s' & \text{(maximalité)} \end{cases}$$

Les composantes connexes d'un graphe ont un certain nombre de propriétés utiles. En particulier, les composantes connexes d'un graphe G forment une ***partition*** de G :

— tout sommet de G appartient à *une et une seule* des composantes connexes de G.

Nous pourrons démontrer cette propriété avec un petit peu d'arsenal mathématique.

7.3 Équivalences

Une relation d'équivalence regroupe les objets d'un ensemble en paquets en fonction de caractéristiques communes. Une telle classification est associée à une relation binaire homogène, qui associe deux à deux les objets appartenant à un même paquet. On isole trois propriétés d'une telle relation reflétant l'appartenance de plusieurs objets à une même classe :

— tout élément appartient à son propre paquet (réflexivité),

— l'énoncé « deux éléments appartiennent au même paquet » ne dépend pas de l'ordre dans lequel on considère les éléments (symétrie),

— l'appartenance à un même paquet se propage de proche en proche (transitivité).

Ces trois point réunis caractérisent la notion d'équivalence.

Relation d'équivalence. Rappel : une relation binaire homogène $\mathcal{R} \subseteq E \times E$ est un ensemble de paires d'éléments « en relation » l'un avec l'autre. On a vu les cas particuliers suivants :

— la relation $\mathcal{R}$est ***réflexive*** lorsque tout élément est en relation avec lui-même

$$\forall e \in E, e\mathcal{R}e$$

— la relation $\mathcal{R}$ est ***symétrique*** lorsqu'elle ne distingue pas l'ordre de ses deux arguments

$$\forall e_1, e_2 \in E,\ e_1 \mathcal{R} e_2 \implies e_2 \mathcal{R} e_1$$

— la relation $\mathcal{R}$ est ***transitive*** lorsqu'elle se propage de proche en proche

$$\forall e_1, e_2, e_3 \in E,\ (e_1 \mathcal{R} e_2 \wedge e_2 \mathcal{R} e_3) \implies e_1 \mathcal{R} e_3$$

Une ***relation d'équivalence*** est une relation binaire homogène qui est à la fois réflexive, symétrique et transitive. Souvent, on utilisera le symbole $\approx$ plutôt que $\mathcal{R}$ pour une telle relation.
Exemples de relations d'équivalence :

— la relation « être égal à »,

— la relation « avoir la même taille que » sur des listes,

— la relation « être lié par un chemin à » dans un graphe non orienté.

Partant d'une relation d'équivalence, on peut reconstruire les « paquets » sous-jacents, sous la forme de *classes d'équivalence.*

Classes d'équivalence. On considère un ensemble E et une relation d'équivalence $\approx$ sur E. La ***classe d'équivalence*** d'un élément $e \in E$ pour $\approx$, notée $[e]$, est l'ensemble des éléments de E qui sont en relation avec e.

$$[e] \quad = \quad \{e' \in E \mid e \approx e'\}$$

Les classes d'équivalence de $\approx$ sont toutes les classes $[e]$ des élément $e \in E$. Tout élément e d'une classe d'équivalence C est appelé un ***représentant*** de cette classe. Nous allons montrer une propriété essentielle des classes : chaque élément $e \in E$ appartient à une et une seule classe d'équivalence de $\approx$.
Propriétés.

1. Pour tout $e \in E$ on a $e \in [e]$.
   Preuve : par réflexivité on a $e \approx e$, et donc par définition $e \in [e]$.
2. Deux éléments équivalents définissent la même classe.

   $$\forall e_1, e_2 \in E,\ e_1 \approx e_2 \implies [e_1] = [e_2]$$

   Preuve : soient e_1 et e_2 deux éléments de E tels que $e_1 \approx e_2$. Montrons que $[e_1] \subseteq [e_2]$.
   Soit $x \in [e_1]$. Par définition $e_1 \approx x$, d'où par symétrie $x \approx e_1$ et par transitivité $x \approx e_2$.
   Ainsi $e_2 \approx x$ par symétrie à nouveau, d'où par définition $x \in [e_2]$.
   On montrerait de même que $[e_2] \subseteq [e_1]$, donc $[e_1] = [e_2]$.
3. Deux classes d'équivalence sont soit disjointes, soit égales.

   $$\forall e_1, e_2 \in E, [e_1] \cap [e_2] = \emptyset \ \vee\ [e_1] = [e_2]$$

   Preuve : Soient e_1 et e_2 deux éléments de E.
   — *Si $[e_1] \cap [e_2] = \emptyset$, alors la conclusion est immédiate.*
   — *Sinon $[e_1] \cap [e_2] \neq \emptyset$, et il existe $e \in [e_1] \cap [e_2]$. Par définition de l'intersection $e \in [e_1]$ et $e \in [e_2]$, d'où par définition $e_1 \approx e$ et $e_2 \approx e$. Par symétrie on a donc $e \approx e_2$ et par transitivité $e_1 \approx e_2$. Alors la propriété 2 permet de conclure $[e_1] = [e_2]$.*

De ces propriétés, on déduit que l'ensemble $\mathcal{C} = \{C_1, C_2, \ldots\}$ des classes d'équivalence d'une relation d'équivalence $\approx$ sur E couvre tout E sans chevauchements.

$$E \subseteq C_1 \cup C_2 \cup \ldots \quad \wedge \quad \forall i, j, C_i \cap C_j = \emptyset$$

Ainsi, les classes d'équivalences de $\approx$ forment bien une ***partition*** de E : chaque élément $e \in E$ appartient à une et une seule classe d'équivalence de $\approx$.

Application aux composantes connexes d'un graphe. Les composantes connexes d'un graphe peuvent être définies comme les classes d'équivalence de la relation d'*accessibilité*. Étant donnés un graphe G et deux sommets s et s' de G, on dit que s' est accessible à partir de s, et on note $s \longrightarrow^* s'$, s'il existe un chemin dans G entre s et s'.

Une ressemblance avec une notation vue en PIL ne serait pas tout à fait fortuite.

On va d'abord vérifier que cette relation est une équivalence, puis analyser ses classes d'équivalence.

Dans un graphe non orienté, la relation $\longrightarrow^*$ est une relation d'équivalence.

Preuve :

- *Réflexivité.* Pour tout s on a bien un chemin de s à s (le chemin vide), et donc $s \longrightarrow^* s$.
- *Symétrie.* Soient s et s' tels qu'il existe un chemin de s à s' dans notre graphe G : $s = s_0 \longrightarrow s_1 \longrightarrow \ldots \longrightarrow s_{n-1} \longrightarrow s_n = s'$. Le graphe n'étant pas orienté, chaque arête peut être prise dans l'autre sens. On forme ainsi un chemin $s_n \longrightarrow s_{n-1} \longrightarrow \ldots \longrightarrow s_1 \longrightarrow s_0$ de s' vers s. Bilan : si $s \longrightarrow^* s'$ alors $s' \longrightarrow^* s$.
- *Transitivité.* Soient s_1, s_2 et s_3 tels qu'il existe un chemin de s_1 vers s_2 et un chemin de s_2 vers s_3. La concaténation de ces deux chemins forme un chemin allant de s_1 à s_3. Donc : si $s_1 \longrightarrow^* s_2$ et $s_2 \longrightarrow^* s_3$ alors $s_1 \longrightarrow^* s_3$.

La relation $\longrightarrow^*$ d'accessibilité étant une équivalence, elle définit des classes d'équivalence $[s]$ sur les sommets d'un graphe non orienté, que l'on peut maintenant analyser.

Dans un graphe non orienté G, les classes d'équivalence de la relation d'accessibilité sont précisément les composantes connexes de G.

Preuve : *toute classe $[s]$ est une composante connexe de G.*

- *Tous les sommets d'une classe $[s]$ sont connectés deux à deux par des chemins.*
  Soit $[s]$ une classe de $\longrightarrow^*$, et $s_1, s_2 \in [s]$ deux sommets de cette classe. Par définition de $[s]$ on a $s \longrightarrow^* s_1$ et $s \longrightarrow^* s_2$. Par symétrie on a $s_1 \longrightarrow^* s$, et par transitivité on déduit $s_1 \longrightarrow^* s_2$: on a un chemin de s_1 vers s_2.
- *Aucun sommet d'une classe $[s]$ n'est connecté à un élément n'appartenant pas à $[s]$.*
  Soit $[s]$ une classe de $\longrightarrow^*$ et $s_1 \in [s]$ un sommet de cette classe. Soit s_2 un sommet quelconque de G tel que $s_1 \longrightarrow^* s_2$. Alors par définition de $[s]$ on a $s \longrightarrow^* s_1$, et par transitivité on déduit $s \longrightarrow^* s_2$. Donc $s_2 \in [s]$. Bilan : un sommet s_2 accessible de puis s_1 est nécessairement dans $[s]$, et ainsi aucun sommet hors de $[s]$ ne peut être connecté à un sommet de $[s]$.

Preuve : *toute composante connexe C de G est la classe $[s]$ d'un certain sommet s.*
Soit C une composante connexe de G. Par définition, C n'est pas l'ensemble vide : il existe au moins un sommet $s \in C$.

- *La composante C est incluse dans la classe $[s]$.*
  Soit $s' \in C$ un sommet de la composante connexe C. Comme s et s' sont tous deux dans C, il existe un chemin $s \longrightarrow^* s'$, et donc $s' \in [s]$ par définition de $[s]$.
- *La classe $[s]$ est incluse dans la composante C.*
  Soit $s' \in [s]$. Par définition on a $s \longrightarrow^* s'$. Comme s est dans C, le sommet s' ne peut pas être en dehors de C.

On a donc bien $C = [s]$.

Application au problème du labyrinthe. Considérons un graphe G non orienté *sans cycles*, et deux sommets s et s' de G. Alors :

Ajouter l'arête $s \longrightarrow s'$ à G crée un cycle
si et seulement si
les sommets s et s' sont dans la même composante connexe.

Traduisons cela pour notre processus de génération de labyrinthe. On énumère toutes les cloisons dans un ordre aléatoire, et pour chacune :

- si elle sépare deux salles qui sont dans la même composante connexe (dans la même classe d'accessibilité), ne rien faire,
- si elle sépare deux salles qui sont dans des composantes connexes disjointes (dans des classes d'accessibilité différentes), créer une porte.

Pour compléter la construction, il ne nous manque plus qu'une structure ou un algorithme permettant de manipuler efficacement des classes d'équivalence de sommets.

7.4 Structure de données : *Union-Find*

La structure *Union-Find* (également appelée *disjoint sets*) permet de manipuler des parties disjointes d'un ensemble E, et en particulier des classes d'équivalence d'éléments de E. La structure fournit deux opérations principales :

— find(e) identifie la classe $[e]$ d'un élément $e \in E$,

— union(e_1, e_2) modifie la structure pour y fusionner les classes de e_1 et e_2.

L'opération find permet de déterminer si deux éléments e_1 et e_2 appartiennent à la même classe : il suffit de tester si find(e_1) = find(e_2). Dans le problème du labyrinthe, c'est ce test qui déterminera si deux salles sont déjà dans la même composante connexe du labyrinthe. L'opération union peut être utilisée pour bâtir une telle structure à partir d'un ensemble de paires d'éléments équivalents :

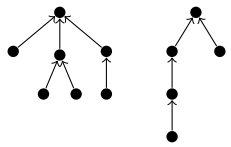
1. on part d'une structure initiale dans laquelle on considère que chaque élément e a une classe réduite à lui-même,
2. pour chaque paire (e_1, e_2) d'éléments équivalents, on fusionne les classes $[e_1]$ et $[e_2]$ à l'aide de l'opération union.

Modélisation avec des graphes. Notre structure d'*union-find* sera un graphe orienté ayant pour sommets les éléments de l'ensemble E, avec deux particularités de forme :

— chaque sommet a au plus une arête sortante,

— le graphe est acyclique.

Un tel graphe est composé de plusieurs blocs ayant des formes comme les suivantes, où tous les chemins convergent vers un élément racine.



Chaque bloc correspond à une classe, et l'élément « racine » d'un bloc peut servir à identifier le bloc (et donc une classe). On donne alors le comportement suivant à nos deux opérations :

— L'opération find(e) renvoie l'élément racine du bloc contenant e. Pour cela, il suffit de suivre les arêtes sortantes à partir de e jusqu'à arriver au bout du chemin.

— L'opération union(e_1, e_2) regroupe les deux blocs contenant e_1 et e_2. Pour cela, il suffit d'ajouter une arête entre les racines de ces deux blocs (à supposer que e_1 et e_2 ne soient pas déjà dans le même bloc).

Réalisation par un tableau. On n'a besoin que de deux informations par sommet :

1. le sommet est-il une racine ?
2. si le sommet n'est pas une racine, quel est le numéro de son unique successeur ?

On peut résumer ces deux informations dans un unique tableau `t` d'entiers, dans lequel

$$\begin{cases} t[i] = i & \text{si } s_i \text{ est une racine} \\ t[i] = j & \text{avec } i \neq j \text{ si } s_j \text{ est l'unique successeur de } s_i \end{cases}$$

Initialisation en java : on crée un tableau `t` dans lequel, pour tout i, $t[i] = i$.

```
class Uf {
  private int[] t;
  Uf(int n) {
    this.t = new int[n];
    for (int i=0; i<n; i++) { t[i] = i; }
  }
```

La méthode `find` doit trouver la racine du bloc de `e`. Pour cela elle s'appelle récursivement sur le successeur de `e`, jusqu'à arriver à un sommet qui est son propre successeur.

```
  int find(int e) {
    int s = t[e];
    if (s == e) { return e; }
    else        { return find(s); }
  }
```

La méthode `union` connecte les blocs de `e1` et de `e2` en désignant l'une des deux racines comme nouveau successeur de l'autre.

```
  void union(int e1, int e2) {
    int r1 = find(e1);
    int r2 = find(e2);
    if (r1 != r2) { t[r1] = r2; }
  }
}
```

Approfondissement : améliorations. Le coût d'utilisation de la structure *union-find* est essentiellement le coût de l'opération find, qui doit parcourir les successeurs d'un sommet jusqu'à trouver la racine du bloc. Le temps est donc proportionnel à la longueur du chemin à parcourir, qui peut lui-même être linéaire en le nombre de sommets. Pour maintenir des longueurs de chemins très courtes, on peut ajouter deux choses à ces algorithmes.

— *Union par rang :* au moment de relier les deux racines `r1` et `r2`, on essaie de mettre l'arête dans le sens qui générera les chemins les moins longs.

Pour réaliser cela, on associe à chaque sommet une information supplémentaire appelée « rang », qui majore la longueur des chemins de son bloc. Alors, dans l'opération union, au lieu de systématiquement faire de r2 le fils de r1, on prend la racine de plus petit rang pour en faire le fils de l'autre, et on met à jour le rang de la nouvelle racine si besoin. Ainsi on fait la fusion de sorte à minimiser la hauteur de l'arbre obtenu. Note : l'information de rang n'est utile que pour la racine de chaque bloc, on ne cherchera donc pas à la mettre à jour pour les autres.

```
class Uf {
  private int[] t;
  private int[] rang;
  Uf(int n) {
    this.t = new int[n]; for (int i=0; i<n; i++) { t[i] = i; }
    this.rang = new int[t]; for (int i=0; i<n; i++) { rang[i] = 0; }
  }
  void union(int e1, int e2) {
    int r1 = find(e1);
    int r2 = find(e2);
    if (r1 == r2) return;
    if (rang[r1] < rang[r2]) {
      t[r1] = r2;
    } else {
      t[r2] = r1;
      if (rang[r1] == rang[r2]) rang[r1]++;
    }
  }
}
```

— *Compression de chemins :* à chaque utilisation de `find`, on mémorise la racine trouvée pour ne plus jamais avoir besoin de parcourir à nouveau le même chemin.

On réalise cela en indiquant la racine trouvée comme nouveau successeur direct du sommet, et on le fait même pour chacun des sommets rencontrés sur la route. Ainsi le chemin de e à find(e) dans le graphe ne sera plus jamais parcouru à nouveau, car il a été remplacé par une unique arête.

```
  int find(int e) {
    int s = t[e];
    if (s == e) {
      return e;
    } else {
      int r = find(s);
      t[e] = r;
      return r;
    }
  }
```

Avec ces deux améliorations, la complexité diminue radicalement : chaque opération `find` a maintenant un temps quasiment constant.

7.5 Code final : génération du labyrinthe

Une classe simple pour un graphe non orienté. On conserve le principe précédent selon lequel les sommets sont numérotés.

```
class Graph {
  private final int size;
  private ArrayList<ArrayList<Integer>> adj;
  public Graph(int size) {
    this.size = size;
    this.adj = new ArrayList<>(size);
    for (int s=0; s<size; s++) { adj.add(new ArrayList<>()); }
  }
  public void addEdge(int s, int t) { adj.get(s).add(t); adj.get(t).add(s); }
  public int size() { return size; }
  public Iterable<Integer> succ(int s) { return adj.get(s); }
  public boolean hasEdge(int s, int t) {
      for (int v: succ(s)) { if (v == t) return true; }
      return false;
  }
}
```

Structure pour représenter les cloisons où l'on est susceptible de créer une porte. On donne : le numéro d'une case, et un booléen pour identifier la direction (si `true` : cloison verticale à l'est de la case, si `false` : cloison horizontale au sud).

```
static class Wall {
  int s;
  boolean v;
  Wall(int s, boolean v) { this.s = s; this.v = v; }
}
```

Si notre graphe correspond à une grille carrée de côté n, il aura n^2 sommets, et le sommet à la ligne i et colonne j aura le numéro $i * n + j$. Un voisin à l'est du sommet numéro k a donc le numéro $s + 1$, et un voisin au sud le numéro $s + n$.

Construction du labyrinthe : on initialise un graphe, dans lequel on ajoute des arêtes à mesure que l'on ouvre des portes dans les cloisons. En parallèle, on maintient une structure *union-find* pour savoir quelles salles sont déjà connectées par un chemin. Avant cette étape de construction, on génère l'ensemble des cloisons dans un tableau, et on mélange ce tableau. L'algorithme de mélange utilisé, bien que très simple, est connu pour générer des *mélanges parfaits* (toutes les permutations du tableau sont équiprobables).

```
static Graph mkLaby(int n) {
  Graph g = new Graph(n*n);
  Uf uf = new Uf(n*n);
  ArrayList<Wall> walls = new ArrayList<>(n*n);
  // Generate walls
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      if (i<n-1) { walls.add(new Wall(i*n+j, true)); }
      if (j<n-1) { walls.add(new Wall(i*n+j, false)); }
    }
  }
  // Randomize walls (Fisher-Yates algorithme, a.k.a. Knuth shuffle)
  Random rnd = new Random();
  int k = walls.size();
  for (int i=1; i<k; i++) { Collections.swap(walls, i, rnd.nextInt(i+1)); }
  // Select doors
  for (Wall w : walls) {
    int s = w.s;
    int t = w.s+(w.v?n:1);
    if (uf.find(s) != uf.find(t)) {
      uf.union(s, t);
      g.addEdge(s, t);
    }
  }
  return g;
}
```

Pour finir, une petite fonction pour affichier un labyrinthe en ASCII.

```
static void printLaby(int n, Graph g) {
  for (int j=0; j<n; j++) { System.out.print("####"); }
  System.out.println("##");
  for (int i=0; i<n; i++) {
    System.out.print("##");
    for (int j=0; j<n; j++) {
      System.out.print("␣␣");
      if (j<n-1 && g.hasEdge(i*n+j, i*n+j+1)) { System.out.print("␣␣"); }
      else { System.out.print("##"); }
    }
    System.out.println();
    System.out.print("##");
    for (int j=0; j<n; j++) {
      if (i<n-1 && g.hasEdge(i*n+j, (i+1)*n+j)) { System.out.print("␣␣"); }
      else { System.out.print("##"); }
      System.out.print("##");
    }
    System.out.println();
  }
}
```

Et le résultat !

```
##############################################################################
##  ##      ##  ##          ##      ##      ##  ##  ##  ##          ##  ##
##  ######  ##  ##  ##############  ######  ##  ##  ##  ##  ##########  ##
##  ##          ##              ##  ##          ##  ##          ##      ##
##  ##  ##  ##############  ######  ##  ######  ##  ##  ##############  ##
##      ##  ##  ##              ##  ##  ##  ##      ##                  ##
######  ######  ##########  ######  ##  ##  ##  ##  ######  ##########  ##
##  ##              ##  ##  ##      ##  ##      ##      ##  ##  ##      ##
##  ######  ##  ######  ##  ##  ##  ######  ##  ##########  ##  ######  ##
##          ##          ##  ##  ##          ##          ##  ##          ##
##########  ##  ##  ##  ##  ##  ##  ##########  ######################  ##
##          ##  ##  ##  ##  ##  ##      ##                  ##          ##
##################  ##  ##  ######  ##############  ##############  ######
##  ##              ##              ##      ##  ##                      ##
##  ##  ##############  ##################  ##  ######  ######  ##########
##  ##  ##          ##  ##      ##      ##  ##          ##              ##
##  ##  ##  ######  ##  ##  ######  ######  ######  ##########  ##########
##  ##  ##      ##  ##          ##          ##      ##                  ##
##  ######  ##  ######  ##  ##  ##  ##  ##############  ##################
##          ##  ##  ##  ##  ##  ##  ##              ##              ##  ##
##  ######  ######  ##########  ##  ##  ##  ##  ##  ######  ##########  ##
##  ##  ##  ##                      ##  ##  ##  ##  ##  ##  ##  ##      ##
##  ##  ######  ##############  ##########  ##  ######  ######  ##  ##  ##
##      ##      ##  ##  ##          ##  ##  ##          ##          ##  ##
######  ##  ##  ##  ##  ##############  ##  ######  ##############  ######
##      ##  ##  ##  ##  ##  ##              ##          ##      ##  ##  ##
######  ######  ##  ##  ##  ##########  ######  ##  ######  ##  ##  ##  ##
##      ##  ##          ##      ##      ##  ##  ##  ##          ##      ##
######  ##  ######  ##  ##  ##  ##  ######  ######  ##  ##############  ##
##                  ##      ##  ##          ##  ##          ##          ##
######  ##  ##########  ######  ##  ##  ######  ##  ##################  ##
##      ##  ##              ##  ##  ##  ##                  ##  ##  ##  ##
##  ######  ##  ##  ######################  ##  ######  ######  ##  ##  ##
##      ##  ##  ##  ##          ##  ##      ##      ##      ##  ##      ##
######  ##############  ##########  ##  ##  ##  ##############  ##  ######
##                  ##                  ##  ##  ##                      ##
##############################################################################
```

Outils logiques et algorithmiques

Thibaut Balabonski @ Université Paris-Saclay
Édition 2024.

Troisième partie

Arbres

8 T :: r :: i :: e :: r :: []

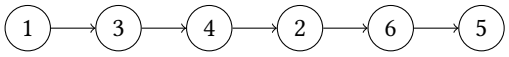
8.1 Problème : tri d'une liste chaînée

On revient à la question du tri déjà explorée sur les tableaux aux chapitres 2 et 3. Cependant, on s'intéresse maintenant à une autre forme de structure séquentielle : la liste chaînée. Cette structure est composée de *cellules*, dont chacune contient deux choses :

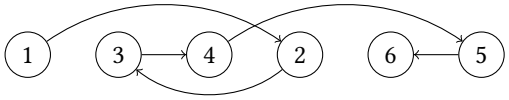
— un élément de la séquence, et

— un pointeur vers la cellule suivante.

Chaque cellule est donc un maillon d'une chaîne, et on peut parcourir la séquence d'éléments en partant de la première cellule et en suivant à chaque étape le pointeur vers la cellule suivante, jusqu'à arriver au bout de la chaîne (lorsqu'il n'y a plus de suivante).



Trier une liste chaînée, c'est produire une nouvelle configuration de ces pointeurs vers les cellules suivantes, de sorte à ce que suivre ces nouveaux pointeurs fasse parcourir les éléments en ordre croissant.



Trier une telle structure *en place* consisterait à conserver toutes les cellules et modifier leurs pointeurs. Cependant, les programmes basés sur des manipulations de pointeurs sont généralement complexes à mettre au point et à analyser, et un tel tri en place ne ferait pas exception [3]. On se restreint donc ici au cas des *listes immuables*, plus simples et souvent préférables. On voudra ainsi produire une *nouvelle liste*, composée de nouvelles cellules, comportant les mêmes éléments que la liste d'origine rangés par ordre croissant, *sans modifier la liste d'origine.*

8.2 Caractérisation récursive des listes chaînées

Une liste chaînée a une structure naturellement récursive, que l'on peut faire ressortir en reformulant la description ainsi :

Une liste non vide est une paire, formée par un élément et une liste.

Autrement dit, une liste est une structure de données pouvant prendre deux formes :

1. la liste vide, que l'on notera [],
2. une paire $e :: \ell$ formée par un élément e (la *tête*) et une liste ℓ (la *queue*).

On qualifie cette structure de *récursive* car une liste (non vide) contient elle-même une liste. En inversant le point de vue, à partir d'une liste ℓ donnée, on peut construire une liste plus grande en ajoutant un élément e en tête.

En appelant *longueur* d'une liste le nombre d'éléments qu'elle contient, on peut rendre encore plus précise la caractérisation précédente. Une liste est soit :

1. la liste vide [] (de longueur zéro),
2. une liste $e :: \ell$ de longueur $n > 0$, dont la queue ℓ est une liste de longueur $n - 1$.

3. Pour les curieux, l'une des sections d'approfondissement y est dédiée.

Notre liste exemple contenant la séquence d'éléments 1, 3, 4, 2, 6, 5 a une longueur de 6, et pourra être notée

$$1 :: (3 :: (4 :: (2 :: (6 :: (5 :: [])))))$$

Par convention, on donne aux opérations :: enchaînées un parenthésage implicite à droite, ce qui permet de simplifier la notation précédente en $1 :: 3 :: 4 :: 2 :: 6 :: 5 :: []$.

En java. Pour représenter une structure de liste chaînée immuable en java, il suffit de définir une unique classe comportant deux champs `head` et `tail`, respectivement pour la tête et la queue (et de déclarer ces deux champs immuables).

```
class List {
  final int  head;
  final List tail;

  List(int x, List l) {
    this.head = x;
    this.tail = l;
  }
}
```

On prend en plus comme convention que la liste vide est simplement représentée par le pointeur **`null`**, et on peut définir une liste contenant les éléments 1, 2, 3 par :

```
List l = new List(1, new List(2, new List(3, null)))
```

En caml. Les listes chaînées immuables sont un type de base en caml, utilisant les mêmes notations que ci-dessus, à savoir `[]` pour la liste vide et `::` pour l'ajout d'un élément en tête d'une liste. On peut donc définir la liste contenant les éléments 1, 2, 3 par :

```
let l = 1 :: (2 :: (3 :: []))
```

ou encore avec l'une des deux formes abrégées

```
let l = 1 :: 2 :: 3 :: []
let l = [1; 2; 3]
```

Si on voulait redéfinir manuellement un clone de ce type des listes chaînées, on utiliserait une définition de type algébrique, avec un constructeur `Nil` pour la liste vide, et un constructeur `Cons` prenant en paramètres un élément et une liste pour l'ajout d'un élément.

```
type list =
  | Nil
  | Cons of int * list
```

La liste avec les éléments 1, 2, 3 serait alors définie par :

```
let l = Cons(1, Cons(2, Cons(3, Nil)))
```

8.3 Fonctions récursives sur des listes chaînées

La caractérisation des listes en deux cas ([] ou $e :: \ell$) peut être utilisée pour définir des fonctions manipulant les listes. Dans les cas les plus simples, il suffit de deux équations pour définir une fonction f : une donnant le résultat $f([])$ pour la liste vide, et une exprimant le résultat $f(e :: \ell)$ pour une liste non vide. La deuxième équation pourra faire intervenir la valeur $f(\ell)$ prise par f sur la queue ℓ de la liste considérée.

Ainsi, une fonction longueur calculant la longueur d'une liste peut être définie par

$$\begin{cases} \mathsf{longueur}([]) & = & 0 \\ \mathsf{longueur}(e :: \ell) & = & 1 + \mathsf{longueur}(\ell) \end{cases}$$

De même, une fonction concat renvoyant la concaténation de deux listes ℓ_1 et ℓ_2, c'est-à-dire une liste formée en faisant se suivre, dans l'ordre, les éléments de ℓ_1 et les éléments de ℓ_2, peut être définie par

$$\begin{cases} \mathsf{concat}([], \ell_2) & = & \ell_2 \\ \mathsf{concat}(e :: \ell_1, \ell_2) & = & e :: \mathsf{concat}(\ell_1, \ell_2) \end{cases}$$

On pourrait alors observer les calculs détaillés suivants.

longueur(1 :: (2 :: (3 :: [])))	concat(1 :: (2 :: (3 :: [])), ℓ)
= 1 + longueur(2 :: (3 :: []))	= 1 :: concat(2 :: (3 :: []), ℓ)
= 2 + longueur(3 :: [])	= 1 :: (2 :: concat(3 :: [], ℓ))
= 3 + longueur([])	= 1 :: (2 :: (3 :: concat([], ℓ)))
= 3	= 1 :: (2 :: (3 :: ℓ))

En java. De telles équations donnent un modèle pour écrire un programme réalisant la fonction f. Le code repose sur un test permettant de différencier les deux formes possibles de la liste `l` donnée en argument : la liste est-elle vide, ou non ?

```
static int length(List l) {
  if (l == null) return 0;
  else           return 1 + length(l.tail);
}
static List concat(List l1, List l2) {
  if (l1 == null) return l2;
  else            return new List(l1.head, concat(l1.tail, l2));
}
```

À noter : le calcul de `length(l)` lorsque `l` n'est pas vide, repose sur un appel récursif sur la queue de la liste. Ce code, obtenu à partir d'équations récursives, est lui-même récursif ! Pour `concat`, on obtient de même une fonction récursive, qui travaille sur le premier des deux arguments. Dans cette deuxième fonction, le **`new`** `List` rend explicite la création d'une nouvelle cellule : la cellule de tête de `l1` est consultée, mais pas modifiée.

En caml. Des équations définissant une fonction sur une liste, on déduit aussi un code caml, centré sur une opération de filtrage énumérant les formes possibles de la liste.

```
let rec length l = match l with
  | []      -> 0
  | e :: l' -> 1 + length l'

let rec concat l1 l2 = match l1 with
  | []     -> l2
  | e :: l -> e :: concat l l2
```

Le code prend cette fois directement la forme d'une série d'équations. À noter : le *motif de filtrage* `e :: l'` ne fait pas que tester la forme « une liste dotée d'une tête et d'une queue », il attribue aussi des noms `e` et `l'` aux différents constituants (on peut utiliser à la place d'une telle lettre le symbole _ si on n'a pas besoin de nommer un constituant). On pourrait écrire également les deux versions allégées ci-dessous de la fonction `length` pour le même effet.

```
let rec length l = match l with
  | []     -> 0
  | _ :: l -> 1 + length l
```

```
let rec length = function
  | []     -> 0
  | _ :: l -> 1 + length l
```

Approfondissement : éviter les débordements de pile. Dans cet exemple, le code obtenu en traduisant directement les équations contient des appels récursifs inutilement coûteux. En effet, le simple fait d'appeler une fonction (récursive ou non) a un coût, qui vient s'ajouter au coût du travail réalisé par la fonction. Ce coût intrinsèque de l'appel étant modeste, nous n'en avons pas tenu compte dans les chapitres précédents. Cependant, la répétition due à la récurrence peut finir par le rendre visible. Appliquées à de grandes listes, les fonctions précédentes peuvent même provoquer une interruption du programme avec une erreur de « dépassement de pile » (*stack overflow*). Et ceci aussi bien en java qu'en caml. Des programmeurs avisés préféreraient certainement les solutions suivantes.

— En java, écrire la fonction `length` avec une boucle énumérant les cellules de la liste et incrémentant une variable `r`.

```
static int length(List l) {
  int r = 0;
  for (; l != null; l = l.tail) { r++; }
  return r;
}
```

— En caml, faire la récursion avec une fonction auxiliaire récursive terminale prenant deux paramètres : la longueur `r` du préfixe déjà parcouru, et la liste `l` restant à parcourir. Cette fonction auxiliaire est appelée en initialisant `r` à zéro, et `l` à la liste complète.

```
let length l =
  let rec loop r l = match l with
    | []     -> r
    | _ :: l -> loop (r+1) l
  in loop 0 l
```

L'appel récursif à `loop` est *terminal* : son résultat est directement transmis comme résultat de l'appel principal. Le compilateur caml reconnaît cette situation et l'optimise, de sorte à nous dispenser du coût de l'appel lui-même [4].

Le paramètre supplémentaire `r` donné à la fonction récursive terminale dans le code caml peut être comparé à la variable `r` qui est mise à jour dans le code java avec boucle. Ces deux versions améliorées ont d'ailleurs, à l'exécution, des comportements très similaires.

Nous verrons bientôt d'autres cas, dans lesquels même des programmeurs aguerris ne trouveraient rien à redire au code directement traduit des équations récursives.

8.4 Raisonnement récursif

En association avec le caractère récursif des listes chaînées et des fonctions les manipulant, il est possible de raisonner par récurrence sur les listes.

Principe de récurrence structurelle. Pour montrer qu'une certaine propriété $P(\ell)$ est valable pour toute liste ℓ, il suffit de s'assurer que la propriété est vraie pour la liste vide, et est préservée par ajout d'un élément en tête. C'est-à-dire qu'il suffit de justifier que :

— $P(\texttt{[]})$ est vraie (cas de base),

— pour toute liste ℓ et tout élément e, si $P(\ell)$ est valide alors $P(e :: \ell)$ est valide également (cas récursif).

On en déduit que la propriété P est valide pour toute liste que l'on puisse construire en ajoutant successivement des éléments en tête de la liste vide, c'est-à-dire pour toute liste.

Exemple. Supposons que l'on souhaite montrer la propriété suivante sur la concaténation de deux listes :

$$\forall \ell_1, \ell_2, \quad \texttt{longueur}(\texttt{concat}(\ell_1, \ell_2)) = \texttt{longueur}(\ell_1) + \texttt{longueur}(\ell_2)$$

Pour raisonner par récurrence structurelle sur ℓ_1, on s'intéresse à la propriété $P(\ell_1)$ définie par la formule

$$\forall \ell_2, \quad \texttt{longueur}(\texttt{concat}(\ell_1, \ell_2)) = \texttt{longueur}(\ell_1) + \texttt{longueur}(\ell_2)$$

Détaillons les deux cas de la récurrence structurelle sur ℓ_1.

— Cas de base, où ℓ_1 est la liste vide : il faut montrer $P(\texttt{[]})$, c'est-à-dire

$$\forall \ell_2, \quad \texttt{longueur}(\texttt{concat}(\texttt{[]}, \ell_2)) = \texttt{longueur}(\texttt{[]}) + \texttt{longueur}(\ell_2)$$

Comme souvent ce cas est facile. Ici il suffit de remarquer que, par définition de `concat` on a $\texttt{concat}(\texttt{[]}, \ell_2) = \ell_2$, et que par définition de `longueur` on a $\texttt{longueur}(\texttt{[]}) = 0$.

— Cas récursif, où on suppose que $P(\ell_1)$ est vraie et où on veut démontrer que $P(e :: \ell_1)$ est vraie. Pour une certaine liste ℓ_1, on suppose donc que

$$\forall \ell_2, \quad \texttt{longueur}(\texttt{concat}(\ell_1, \ell_2)) = \texttt{longueur}(\ell_1) + \texttt{longueur}(\ell_2)$$

et on veut montrer que

$$\forall \ell_2, \quad \texttt{longueur}(\texttt{concat}(e :: \ell_1, \ell_2)) = \texttt{longueur}(e :: \ell_1) + \texttt{longueur}(\ell_2)$$

4. Cette optimisation n'existe que dans peu de langages. En plus de caml, on peut citer C.

La première propriété est appelée *hypothèse de récurrence*. On peut détailler le calcul ainsi

	`longueur(concat(`$e :: \ell_1, \ell_2$`))`	
=	`longueur(`$e ::$ `(concat(`ℓ_1, ℓ_2`)))`	par déf. de `concat`
=	$1 +$ `longueur(concat(`ℓ_1, ℓ_2`))`	par déf. de `longueur`
=	$1 +$ (`longueur(`ℓ_1`)` + `longueur(`ℓ_2`)`)	par hyp. de récurrence
=	($1 +$ `longueur(`ℓ_1`)`) + `longueur(`ℓ_2`)`	
=	`longueur(`$e ::$ ℓ_1`)` + `longueur(`ℓ_2`)`	par déf. de `longueur`

On peut rapprocher le raisonnement précédent d'un raisonnement par récurrence classique sur les nombres entiers :

- dans le cas de base on considère la liste vide, qui est la seule liste de longueur 0, et
- dans le cas récursif on montre que la propriété est vraie pour une liste de longueur n+1, en partant de l'hypothèse qu'elle est vraie pour une liste de longueur n.

Deux preuves pour s'exercer : $\forall \ell,$ `concat(`ℓ`, [])` $= \ell$ et $\forall \ell_1, \ell_2, \ell_3,$ `concat(`ℓ_1`, concat(`ℓ_2, ℓ_3`))` = `concat(concat(`ℓ_1, ℓ_2`),` ℓ_3`)`.

8.5 Algorithme : tri insertion

Voici une réalisation en caml du tri par insertion de listes. Note : contrairement à ce qu'on a vu jusque-là sur les tableaux, mais comme annoncé au début du chapitre, ce tri n'est pas « en place », c'est-à-dire qu'il ne modifie pas la liste d'origine. À la place, on renvoie une nouvelle liste, contenant les éléments dans l'ordre. Il se trouve que ce fait, allié à la récursion, rend le raisonnement nettement plus simple.

```
let rec insert x l = match l with
  | []      -> [x]
  | y :: l' -> if x <= y then x :: l
                         else y :: insert x l'

let rec insertion_sort = function
  | []     -> []
  | x :: l -> insert x (insertion_sort l)
```

Spécifications de ces deux fonctions :

- `insertion_sort` renvoie une permutation triée de la liste donnée en argument.
- `insert` prend en paramètres un élément x et une liste triée ℓ, et renvoie une permutation triée de $x :: \ell$.

Savez-vous le faire avec des fonctions récursives terminales ? Serait-ce utile ici ?

Preuves de correction. On peut démontrer par récurrence que nos deux fonctions correspondent bien à ces spécifications. Pour `insert` :

- Cas de base : `insert(`x`, [])` $= [x]$, et $[x]$ est bien une permutation triée de $x :: []$.
- Cas récursif : on suppose que $y :: \ell$ est triée et on veut montrer que `insert(`$x, y :: \ell$`)` est une permutation triée de $x :: y :: \ell$. Deux cas en fonction de la comparaison de x et y.
  - Si $x \leq y$ alors `insert(`$x, y :: \ell$`)` $= x :: y :: \ell$ et cette liste est triée et est bien une permutation d'elle-même.
  - Si $x > y$ alors `insert(`$x, y :: \ell$`)` $= y ::$ `insert(`x, ℓ`)`. Par hypothèse de récurrence `insert(`x, ℓ`)` est une permutation triée de $x :: \ell$. La liste $y :: \ell$ était supposée triée : pour tout $z \in \ell$ on a donc $y \leq z$. Donc $y ::$ `insert(`x, ℓ`)` est triée. En outre, cette liste est une permutation de $y :: x :: \ell$, et donc également une permutation de $x :: y :: \ell$.

Pour `insertion_sort` :

- Cas de base : `insertion_sort([])` $= []$, résultat immédiat.
- Cas récursif : on veut montrer que `insertion_sort(`$x :: \ell$`)` est une permutation triée de $x :: \ell$. Le code donne `insertion_sort(`$x :: \ell$`)` $=$ `insert(`x`, insertion_sort(`ℓ`))`. Par hypothèse de récurrence `insertion_sort(`ℓ`)` est une permutation triée de ℓ. En particulier `insertion_sort(`ℓ`)` est triée et `insert` s'applique bien. Par spécification de `insert`, `insert(`x`, insertion_sort(`ℓ`))` est une permutation triée de $x ::$ `insertion_sort(`ℓ`)`, c'est-à-dire une permutation triée de $x :: \ell$.

Du côté des complexités, on pourrait calculer assez facilement que le tri insertion sur des listes chaînées, comme le tri insertion d'un tableau, est quadratique en la longueur de la séquence à trier.

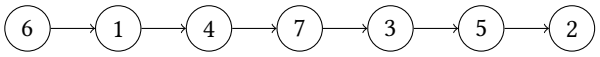
Arrivez-vous bien à le faire ?

En java. Pour information, voici une version java des fonctions précédentes. Les preuves de correction pour cette version seraient identiques à celles du paragraphe précédent.

```
static List insert(int x, List l) {
  if (l == null)        return new List(x, null);
  else if (l.head >= x) return new List(x, l);
  else                  return new List(l.head, insert(x, l.tail));
}

static List insertionSort(List l) {
  if (l == null) return null;
  else           return insert(l.head, insertionSort(l.tail));
}
```

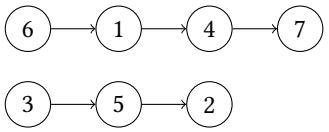
8.6 Algorithme : tri fusion

On peut trier une liste chaînée en un temps linéarithmique en la longueur de la liste. On présente pour cela l'algorithme de *tri fusion*[5].

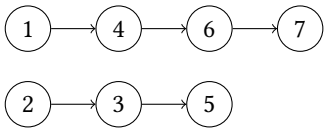


L'idée est la suivante :

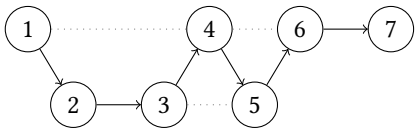
1. Séparer la liste en deux sous-listes de tailles égales (à un arrondi près si la liste à séparer a une taille impaire).



2. Trier chacune des deux sous-listes. Récursivement, bien sûr.



3. Reformer une unique liste triée à partir des deux sous-listes triées. C'est cette dernière étape qu'on appelle *fusion*.



En caml. On propose d'abord deux fonctions `take` et `chop`, qui respectivement, prennent les `n` premiers d'une liste ou ne conservent d'une liste que les éléments situés après les `n` premiers. Ces deux fonctions, appliquées avec un même `n` ciblant la moitié de la liste, permettent de la séparer en deux parties égales. Note : dans le code de la première fonction, `List.hd` et `List.tl` sont deux fonctions de la bibliothèque standard qui extraient respectivement la tête et la queue d'une liste.

```
let rec take n l =
  if n=0 then []
  else (List.hd l) :: (take (n-1) (List.tl l))

let rec chop n l =
  if n=0 then l
  else chop (n-1) (List.tl l)
```

5. Algorithme aperçu dans le TD3 dans le cas du tri en place d'un tableau.

La fonction principale `mergesort` découpe la liste donnée en argument en deux à l'aide de `take` et `chop`, trie les deux moitiés récursivement, puis les assemble à l'aide d'une autre fonction auxiliaire `merge`. À moins que la liste d'origine soit si petite qu'il soit certain qu'elle est déjà triée (c'est-à-dire : si elle ne contient que zéro ou un élément).

```
let rec mergesort l =
  let n = length l in
  if n < 2 then l
  else let k = (n+1)/2 in
       let l1, l2 = take k l, chop k l in
       merge (mergesort l1) (mergesort l2)
```

La fonction de fusion prend en paramètres deux listes `l1` et `l2` supposées triées, et produit une unique liste triée entrelaçant les éléments de `l1` et `l2`. Elle est également récursive : elle compare le premier élément de `l1` et le premier élément de `l2`, sélectionne le plus petit des deux, puis fusionne ce qui reste. Le processus s'arrête lorsque l'une des deux listes à fusionner est vide : il suffit alors de conserver l'autre intégralement.

Pour lesquelles de ces fonctions serait-il préférable d'utiliser une variante récursive terminale ? Sauriez-vous les écrire ?

```
let rec merge l1 l2 = match l1, l2 with
  | [], l | l, []    -> l
  | x1::tl1, x2::tl2 -> if x1 <= x2 then x1 :: (merge tl1  l2)
                                    else x2 :: (merge  l1 tl2)
```

En java. Voici une version java du tri fusion présenté ci-dessus en caml. Les deux vont s'analyser exactement de la même façon.

```
static List take(int n, List l) {
  if (n == 0) return null;
  else        return new List(l.head, take(n-1, l.tail));
}
static List chop(int n, List l) {
  if (n == 0) return l;
  else        return chop(n-1, l.tail);
}

static List merge(List l1, List l2) {
  if (l1 == null) return l2;
  if (l2 == null) return l1;
  if (l1.head <= l2.head) return new List(l1.head, merge(l1.tail, l2     ));
  else                    return new List(l2.head, merge(l1     , l2.tail));
}

static List mergeSort(List l) {
  int n = length(l);
  if (n < 2) return l;
  int k = (n+1)/2;
  List l1 = take(k, l), l2 = chop(k, l);
  return merge(mergeSort(l1), mergeSort(l2));
}
```

8.7 Approfondissement : analyse du tri fusion

Spécification. Précisons les spécification des quatre fonctions formant notre tri fusion.

- Un appel `chop n l` renvoie la liste formée en retirant à `l` ses `n` premiers éléments. Précondition : la liste `l` doit contenir au moins `n` éléments.
- Un appel `take n l` renvoie la liste formée avec les `n` premiers éléments de `l`, dans l'ordre. Précondition : la liste `l` doit contenir au moins `n` éléments.
- Un appel `merge l1 l2` renvoie une liste triée formée des éléments des listes `l1` et `l2`. Précondition : `l1` et `l2` doivent elle-mêmes être triées.
- Un appel `mergesort l` renvoie une liste triée formée des éléments de la liste `l`.

Terminaison. Nos quatre fonctions récursives terminent à coup sûr sur toute entrée. On le vérifie à l'aide de la technique du variant. Rappel : adaptée pour justifier la terminaison d'une fonction récursive f, la technique demande d'identifier une quantité positive dépendant des paramètres de f, qui décroît strictement à chaque appel récursif. On peut choisir les variants suivants :

- Pour `take` et `chop`, le paramètre `n` est lui-même un variant : c'est un entier supposé positif ou nul, et l'unique appel récursif présent dans le code de `take` (ou `chop`) utilise la valeur `n-1`.
- Pour `mergesort`, on prend comme variant la longueur n de la liste `l` donnée en paramètre. Les appels récursifs se font avec des listes de longueurs $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$, et uniquement dans le cas où $n \geq 2$: on a bien décroissance.
- Pour `merge`, on prend comme variant la somme des longueurs des deux listes `l1` et `l2`. Il s'agit bien d'un entier positif, car les longueurs sont elle-mêmes des entiers positifs. En outre, chaque appel récursif s'applique à : l'une des deux listes telle qu'elle a été donnée, et la queue de l'autre liste. On ne sait pas à l'avance quelle liste va décroître, mais on sait qu'une des deux le fait : la somme va bien décroître (en l'occurrence, de précisément 1).

Correction. Les variants identifiés dans l'analyse précédente nous disent également comment organiser notre raisonnement par récurrence, si l'on veut démontrer que nos fonctions récursives sont bien correctes.

- Pour `take` et `chop`, une récurrence simple sur `n` suffit.
- Pour `merge`, on peut faire une récurrence simple sur la somme des longueurs des deux listes.
- Pour `mergesort`, il faut faire une récurrence forte sur la longueur de la liste.

Complexité. Les fonctions `take` et `chop` ont manifestement une complexité proportionnelle à `n`. La fonction `merge` a une complexité linéaire en la somme des longueurs des deux listes passées en argument. Lors d'un appel `mergesort l` avec `l` de taille n, on a les opérations suivantes.

- Découpage de la liste `l` en deux moitiés avec `take` et `chop` : coût $\mathcal{O}(n)$.
- Deux appels récursifs sur des listes de tailles $\lceil n/2 \rceil$ et $\lfloor n/2 \rfloor$.
- Fusion des deux listes obtenues avec `merge`. La somme des longueurs de ces deux listes est n : le coût de cette étape est encore $\mathcal{O}(n)$.

La relation de récurrence pour la complexité $C(n)$ de `mergesort` sur les listes de longueur n a donc la forme

$$C(n) \quad = \quad 2C(\frac{n}{2}) + f(n)$$

où $f(n) = \mathcal{O}(n)$. On a déjà vu une telle équation dans le cas le plus favorable du tri rapide.

$$C(n) \quad = \quad \Theta(n \log(n))$$

Conclusion : le tri fusion de listes chaînées a une complexité linéarithmique.

8.8 Approfondissement : tri fusion, version récursive terminale (caml)

On améliore le tri fusion présenté en caml en s'assurant que les fonctions de découpage et de fusion sont récursives terminales. Au passage, on combine les fonctions `take` et `chop` en une seule fonction `split_at` renvoyant une paire de listes. Cette fonction utilise une fonction auxiliaire récursive terminale `split` telle que `split n l1 l2` place les `n` premiers éléments de `l1` en tête de `l2` (dans l'ordre inverse), et renvoie la paire formée par : ce qui reste de `l1`, et la version étendue de `l2`. Il suffit alors d'appliquer cette fonction auxiliaire en prenant pour `l1` la liste à découper et pour `l2` la liste vide.

```
let split_at n l =
  let rec split n l1 l2 = match n, l1 with
    | 0, _     -> l1, l2
    | n, x::tl -> split (n-1) tl (x :: l2)
  in split n l []
```

Remarque : en transférant les éléments en tête de `l1` vers `l2` on renverse leur séquence, mais cela ne perturbe pas le tri qui va suivre.

La nouvelle fonction de fusion est de même basée sur une fonction récursive terminale `merge l l1 l2` où : `l1` et `l2` sont ce qui reste à fusionner, et `l` est la séquence déjà fusionnée. À noter cependant : comme on n'ajoute les éléments qu'en tête, la séquence `l` est triée à l'envers. Ainsi, lorsque l'une des deux listes `l1` ou `l2` est épuisée et que l'on veut concaténer `l` et la liste restante, on le fait en renversant `l`. La fonction `rev_append` réalise cela [6].

```
let rec rev_append l1 l2 = match l1 with
  | []    -> l2
  | x::tl -> rev_append tl (x::l2)

let merge l1 l2 =
  let rec merge l l1 l2 = match l1, l2 with
    | [], l' | l', []  -> rev_append l l'
    | x1::tl1, x2::tl2 -> if x1 <= x2 then merge (x1::l) tl1  l2
                                      else merge (x2::l)  l1 tl2
  in merge [] l1 l2
```

La fonction `mergesort` ne change pas, si ce n'est pour s'adapter au remplacement de `take` et `chop` par l'unique fonction `split_at`. On ne peut pas rendre `mergesort` récursive terminale car le tri récursif des deux moitiés est suivi d'une opération de fusion. Cependant, on ne risque pas de dépassement de pile ici puisque le nombre d'appels emboîtés à `mergesort` est logarithmique en la taille de la liste.

```
let rec mergesort l =
  let n = length l in
  if n < 2 then l
  else let l1, l2 = split_at ((n+1)/2) l in
       merge (mergesort l1) (mergesort l2)
```

8.9 Approfondissement : tri fusion en place (java)

Listes mutables. Reprenons notre définition des listes en java, et autorisons cette fois la mutation du pointeur `tail` vers la cellule suivante.

```
class List {
  final int head;
  List tail;
  ...
}
```

Certaines des fonctions déjà écrites, comme `length`, n'ont pas vocation à changer. On peut en revanche imaginer une nouvelle spécification pour la fonction `concat` : au lieu de créer une nouvelle liste réunissant les éléments de ses deux paramètres `l1` et `l2`, on peut lui demander de *modifier* physiquement `l1` pour que le pointeur `tail` de sa dernière cellule soit maintenant dirigé vers `l2`.

```
static void concat(List l1, List l2) {
  assert (l1 != null);
  for(; l1.tail != null; l1 = l1.tail) {}
  l1.tail = l2;
}
```

Note : par sa spécification même, cette version n'a de sens que si `l1` contient bien au moins une cellule ! Pour une variante fonctionnant également avec `l1` vide, il faut encore une autre spécification. En voici une possible :

— si `l1` n'est pas vide, la fonction doit rediriger le dernier pointeur `tail` de `l1` vers `l2`,

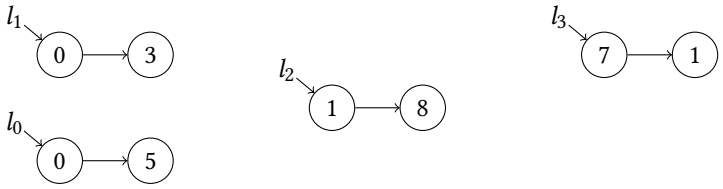
— et dans tous les cas, la fonction doit renvoyer la première cellule de la liste fusionnée.

Cette nouvelle spécification change même le type de la fonction, qui doit maintenant renvoyer (un pointeur vers) une cellule. Il s'agira en général de la première cellule de `l1`, sauf dans le cas où `l1` est vide, où on renvoie à la place la première cellule de `l2`.

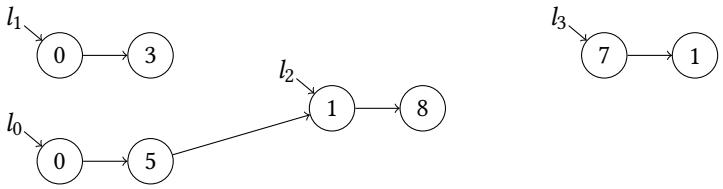
6. Fonction présente dans la bibliothèque standard, et analysée prochainement en TD.

```java
static List concat(List l1, List l2) {
  if (l1 == null) return l2;
  List c = l1;
  for(; c.tail != null; c = c.tail) {}
  c.tail = l2;
  return l1;
}
```

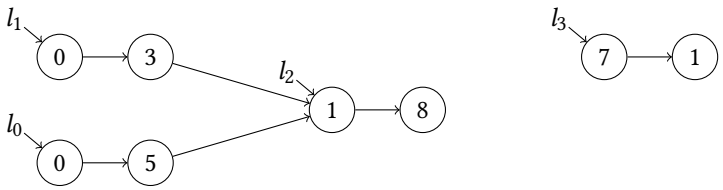
Pièges des listes mutables. Lorsqu'elles sont autorisées, les modifications de listes peuvent rapidement devenir acrobatiques, car elles introduisent des phénomènes de *partage*, ou *aliasing* : au fil du temps, plusieurs fragments de listes peuvent être reliés, et une modification d'une liste peut avoir des répercussions sur une autre. Par exemple, partons de quatre listes `l0`, `l1`, `l2` et `l3` contenant respectivement les séquences `{0, 5}`, `{0, 3}`, `{1, 8}` et `{7, 1}`.



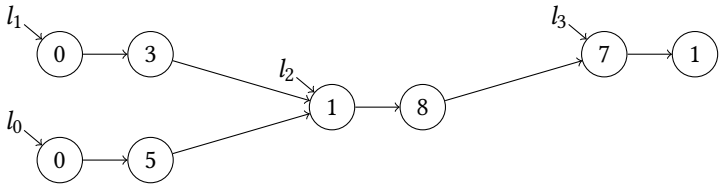
Avec la concaténation `concat(l0, l2)`, on connecte la dernière cellule de `l0` à la première cellule de `l2`. La liste `l0` contient maintenant la séquence `0, 5, 1, 8`.



Si on enchaîne avec la concaténation `concat(l1, l2)`, de manière similaire on lie les deux listes.



Si enfin on appelle `concat(l1, l3)`, la liste `l1` grandit encore pour contenir la séquence `0, 3, 1, 8, 7, 1`.



Cependant la liste `l2` a été modifiée également, même si elle n'était pas mentionnée dans l'opération : elle contient maintenant la séquence `1, 8, 7, 1`. En effet, cette liste était physiquement intégrée à `l1`, et a donc subi des effets des modifications appliquées à `l1`. De même, la liste `l0` contenait la liste `l2` et a également été affectée : elle contient maintenant la séquence `0, 5, 1, 8, 7, 1` que l'on n'aurait peut-être pas souhaité voir.

Ces phénomènes rendent les programmes manipulant des listes chaînées modifiables difficiles à mettre au point et à analyser.

Tri fusion en place, en java. On veut un tri fusion ne créant pas de nouvelles cellules, mais modifiant les pointeurs `tail` de la liste passée en paramètre. Dans un tel contexte, on peut déjà remplacer la paire de fonctions `take/chop` par une unique fonction `split` effectuant une coupure dans la liste donnée : la liste `l` après coupure n'est plus que la première partie de la liste d'origine, et on demande à `split` de renvoyer un pointeur vers la deuxième partie. Note : après action de `split`, la liste `l` est définitivement raccourcie.

```
static List split(int n, List l) {
  for(; n > 1; n--) l = l.tail;
  List res = l.tail;
  l.tail = null;
  return res;
}
```

La fonction `merge` fusionne deux listes `l1` et `l2` en modifiant les pointeurs `tail` des cellules. Encore plus manifestement que la précédente, cette opération est « destructrice », dans le sens où les listes d'origine sont perdues (leurs cellules sont recyclées pour former la liste fusionnée). La fonction `merge` renvoie un pointeur vers la première cellule de la liste fusionnée, qui sera la première cellule de l'une des deux anciennes listes `l1` ou `l2`. La boucle principale de la fonction maintient une cellule `last`, qui est la dernière cellule courante de la liste fusionnée, et donc la prochaine dont on va redéfinir le pointeur `tail`.

```
static List merge(List l1, List l2) {
  if (l1 == null) return l2;
  if (l2 == null) return l1;
  // Initialisation du résultat
  List res;
  if (l1.head <= l2.head) { res = l1; l1 = l1.tail; }
  else                    { res = l2; l2 = l2.tail; }
  // Dernière cellule du résultat partiel...
  List last = res;
  // ... que l'on met à jour tant que l1 et l2 ne sont pas épuisées.
  while (l1 != null && l2 != null) {
    if (l1.head <= l2.head) { last.tail = l1; l1 = l1.tail; }
    else                    { last.tail = l2; l2 = l2.tail; }
    last = last.tail;
  }
  // Lorsque l'une des listes est épuisée on concatène l'autre et on conclut.
  if (l1 == null) { last.tail = l2; }
  else            { last.tail = l1; }
  return res;
}
```

Il ne reste alors plus qu'à combiner ces deux premières fonctions pour réaliser le tri `mergeSort`. Comme avec la fonction `merge`, on renvoie un pointeur vers la première cellule de la liste produite.

```
static List mergeSort(List l) {
  int n = length(l);
  if (n < 2) return l;
  List l2 = split((n+1)/2, l);
  return merge(mergeSort(l), mergeSort(l2));
}
```

Petit quizz : après la définition et le tri suivant, vers quelle séquence pointe la variable `l` *?*

```
List l = new List(3, new List(2, new List(5, new List(1, new List(4, null)))));
List r = mergeSort(l);
```

9 Chercher ses clés

9.1 Problème : le mot le plus fréquent

Considérons un texte français d'une taille respectable. Par exemple, *Notre-Dame de Paris* de Victor Hugo. Question : quels en sont les dix mots les plus fréquents ?

Pour répondre simplement à cette question, on peut lire l'intégralité du texte et tenir à jour un lexique, dans lequel on note les mots rencontrés et le nombre d'apparitions de chacun. À la fin, ne reste plus qu'à parcourir ce lexique pour prélever les mots ayant le plus grand nombre d'occurrences. Cependant, pour que ceci soit viable nous avons besoin d'une structure de données :

- qui permet de stocker des mots, et d'associer une information à chaque mot stocké,
- dans laquelle on peut efficacement :
  - tester la présence ou l'absence d'un mot,
  - ajouter un nouveau mot,
  - obtenir ou mettre à jour l'information associée à un mot.

Une telle structure associant des *valeurs* (ici : les nombres d'occurrences) à des *clés* (ici : les mots) est appelée un *tableau associatif*. Nous allons voir l'une des manières efficaces de réaliser un tel tableau associatif, basée sur une structure d'*arbre binaire*, c'est-à-dire une structure de données récursive qui, à chaque étape, se scinde en deux branches.

Une autre structure que vous connaissez déjà pour réaliser cela est la table de hachage.

9.2 Structure : arbre binaire

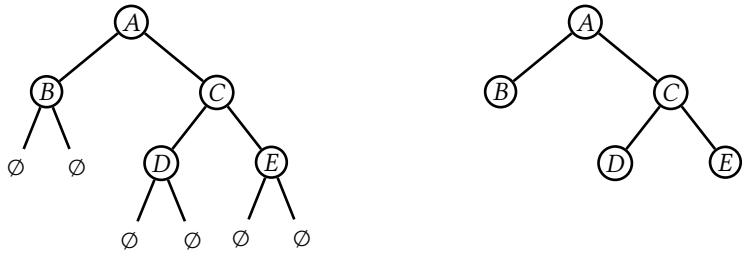
Caractérisation récursive. Un *arbre binaire* est une structure qui est :

- soit vide,
- soit formée d'un *nœud* principal appelé *racine* et de deux sous-structures appelées *sous-arbre gauche* et *sous-arbre droit*.

On a une structure récursive similaire à celle des listes chaînées. Différence : une liste chaînée non vide possède un pointeur vers une unique queue, tandis qu'un arbre binaire possède deux pointeurs vers deux sous-arbres (avec distinction d'un côté gauche et d'un côté droit).

Exemple. Voici un exemple d'arbre binaire, où l'arbre vide est représenté par ∅. Dans les dessins cependant, on ne fait en général pas figurer les sous-arbres vides. On utilisera donc couramment le schéma de droite. L'arbre de cet exemple a pour racine le nœud A, pour sous-arbre gauche l'arbre contenant uniquement le nœud B, et pour sous-arbre droit l'arbre contenant les nœuds C, D et E.

Oui, en info un arbre a une racine en haut et des branches qui s'étendent vers le bas.



Vocabulaire. Une *feuille* est un nœud dont les deux sous-arbres sont vides. Un *nœud interne* est un nœud qui n'est pas une feuille. Dans l'arbre ci-dessus, B, D et E sont des feuilles, et A et C sont des nœuds internes.

La *taille* d'un arbre binaire est le nombre de nœuds qu'il contient. La *profondeur* d'un nœud est le nombre de liens à suivre pour atteindre ce nœud en partant de la racine de l'arbre. La *hauteur* d'un arbre binaire non vide est la profondeur maximale de ses feuilles, plus un. La *hauteur* de l'arbre vide est zéro. Ainsi, l'arbre ci-dessus a une taille de 5. Le nœud A a la prondeur 0, le nœud B la prondeur 1 et le nœud D la prondeur 2. La hauteur de l'arbre est 3.

Dans un arbre binaire, le *fils gauche* (resp. *fils droit*) d'un nœud est la racine du sous-arbre gauche (resp. sous-arbre droit), si ce sous-arbre n'est pas vide. Chaque nœud en dehors de la racine est le fils (gauche ou droit) d'un unique nœud appelé son *parent*. Dans l'arbre ci-dessus, le nœud A a pour fils gauche le nœud B, et pour fils droit le nœud C. Le parent de D est C, et le parent de C est A.

Un arbre peut être vu comme un graphe, dont les sommets sont les nœuds de l'arbre et dont les arêtes sont les liens entre parents et fils. On pourra donc réemployer ici certaines notions vues sur les graphes. *Attention toutefois :* pour un arbre binaire, contrairement au cas des graphes habituels, le placement des nœuds est significatif : le sous-arbre gauche d'un nœud ne doit pas être confondu avec son sous-arbre droit. En particulier, les deux arbres binaires ci-dessous ne sont pas égaux : le nœud B est dans l'un des cas le fils gauche de A, et dans l'autre cas le fils droit.

Dans chacun de ces deux cas le nœud A a bien deux sous-arbres, mais l'un ou l'autre des deux sous-arbres est vide, et le nœud A n'a donc qu'un seul fils.

Dimensions. Voici quelques propriétés liant la taille et la hauteur d'un arbre binaire.

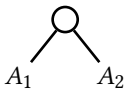
1. *Un arbre binaire a au plus 2^d nœuds à profondeur d.*
   Démonstration par récurrence sur d :
   - Cas $d = 0$: dans un arbre non vide, seule la racine a une profondeur de 0.
   - Cas héréditaire : on suppose que tout arbre binaire a au plus 2^d nœuds à profondeur d et on démontre que tout arbre binaire a au plus 2^{d+1} nœuds à profondeur $d+1$. Soit un arbre A. S'il est vide, il n'a aucun nœud à profondeur $d+1$. S'il est non vide, il possède deux sous-arbres A_1 et A_2. Les nœuds à profondeur $d+1$ de A sont à profondeur d dans A_1 ou dans A_2. Par hypothèse de récurrence il y en a au maximum 2^d dans A_1 et 2^d dans A_2. D'où en additionnant $2^d + 2^d = 2 \times 2^d = 2^{d+1}$ nœuds à profondeur $d+1$ dans A au total.
2. *Un arbre binaire de hauteur h a au plus $2^h - 1$ nœuds.*
   Démonstration : chaque nœud d'un arbre binaire de hauteur h a une profondeur $d < h$. Or il y a au plus 2^d nœuds à profondeur d. Le nombre total de nœuds est donc au plus $\sum_{0 \leq d < h} 2^d = 2^h - 1$.
3. *Un arbre binaire de hauteur h possède au moins h nœuds.*
   Démonstration : il faut au moins un nœud à chacune des profondeurs 0, 1, 2, jusqu'à $h-1$, soit au moins h nœuds au total.
4. *Un arbre binaire non vide avec n sommets a une hauteur h vérifiant $\lfloor log_2(n) \rfloor < h \leq n$.*
   Cette propriété est un corollaire des deux précédentes.

Bilan : lorsque les différents niveaux de profondeur d'un arbre sont « bien remplis », la hauteur est logarithmique en le nombre total de nœuds. Ce point sera la clé de l'efficacité des structures de données bâties sur des arbres [7].

9.3 Raisonnement par récurrence structurelle.

Plutôt que de se ramener à de la récurrence sur les entiers, on peut également tirer parti de la nature récursive des arbres binaires pour raisonner directement par récurrence sur la structure des arbres. Pour montrer qu'une certaine propriété P est valide pour tous les arbres binaires il suffit de vérifier que :

- P est valide pour l'arbre vide, et que
- dès que A_1 et A_2 sont deux arbres binaires pour lesquels P est valide, P reste valide pour l'arbre composé



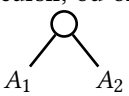
Exemple. Pour tout arbre A la hauteur h_A et la taille n_A vérifient $n_A < 2^{h_A}$.
On raisonne par récurrence structurelle. Comme dans un raisonnement par récurrence usuel, on a deux cas à traiter.

- Cas de base, pour l'arbre vide : il faut montrer que la hauteur $h_\emptyset$ et la taille $n_\emptyset$ de l'arbre vide vérifient $n_\emptyset < 2^{h_\emptyset}$. Or, par définition on a

  $$\begin{aligned} h_\emptyset &= 0 \\ n_\emptyset &= 0 \end{aligned}$$

  On a donc bien $n_\emptyset = 0 < 1 = 2^0 = 2^{h_\emptyset}$.

7. Plus d'information dans la section d'approfondissement sur les arbres équilibrés.

— Cas récursif, où on veut montrer que la propriété est valide pour un arbre A de la forme  en supposant qu'elle est bien valide pour les deux sous-arbres A_1 et A_2. Autrement dit, en notant h_1 et n_1 (resp. h_2 et n_2) la hauteur et la taille de A_1 (resp. A_2), on suppose que

$$\begin{aligned} n_1 &< 2^{h_1} \\ n_2 &< 2^{h_2} \end{aligned}$$

et on veut démontrer que

$$n_A < 2^{h_A}$$

Les deux suppositions sont appelées *hypothèses de récurrence.*
Détail du raisonnement :

$$\begin{aligned} n_A &= 1 + n_1 + n_2 && \text{par déf. de la taille} \\ &\leq 2^{h_1} + n_2 && \text{par hyp. de récurrence 1} \\ &< 2^{h_1} + 2^{h_2} && \text{par hyp. de récurrence 2} \\ &\leq 2^{\max(h_1,h_2)} + 2^{\max(h_1,h_2)} \\ &= 2^{1+\max(h_1,h_2)} \\ &= 2^{h_A} && \text{par déf. de la hauteur} \end{aligned}$$

9.4 Représentation des arbres binaires en caml

On peut définir un type de données caml pour représenter des arbres binaires en suivant la caractérisation récursive des arbres. Définissons donc un arbre comme étant :

— soit l'arbre vide, représenté par la constante `E`,
— soit un nœud, représenté par l'application `N(`l`, `r`)` du constructeur `N` à deux sous-arbres gauche et droit l et r.

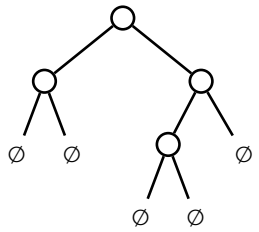
Déclaration d'un tel type en caml :

```
type tree =
  | E
  | N of tree * tree
```

Alors la définition

```
let t = N(N(E, E), N(N(E, E), E))
```

associe la variable `t` à l'arbre



On peut alors définir des fonctions manipulant les arbres binaires grâce aux mécanismes de filtrage. Par exemple pour calculer la taille d'un arbre, c'est-à-dire compter son nombre de nœuds, on prévoit un cas indiquant que l'arbre vide ne contient aucun nœud, et un cas indiquant que la taille d'un arbre non vide est obtenue en ajoutant un à la somme des tailles de ses deux sous-arbres.

Rappel : la ligne
`let rec size = function`
est ici équivalente à
`let rec size t = match t with`

```
let rec size = function
  | E       -> 0
  | N(l, r) -> 1 + size l + size r
```

Cette définition caml réalise les équations suivantes :

$$\begin{cases} \texttt{size}(\emptyset) &= 0 \\ \texttt{size}\left(\begin{array}{c} \circ \\ A_1 \quad A_2 \end{array}\right) &= 1 + \texttt{size}(A_1) + \texttt{size}(A_2) \end{cases}$$

On peut calculer la hauteur d'une manière similaire.

```
let rec height = function
  | E       -> 0
  | N(l, r) -> 1 + max (height l) (height r)
```

Arbres binaire annotés. On peut associer des informations à chaque sommet d'un arbre pour représenter des objets intrinsèquement arborescents ou organiser des structures de données. Selon la structure représentée on peut utiliser pour cela des stratégies variées :

— mettre des données dans chaque nœud, ou

— ne mettre des données que dans les feuilles, ou

— mettre des informations de natures différentes dans les feuilles et dans les nœuds internes.

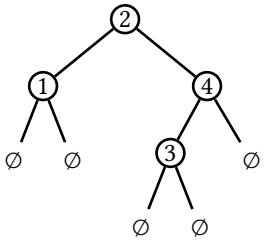
Considérons ici la version la plus simple : chaque nœud contient un nombre entier. On peut enrichir notre type caml pour ajouter ce troisième paramètre au constructeur `N` :

```
type tree =
  | E
  | N of tree * int * tree
```

La nouvelle définition

```
let t = N(N(E, 1, E), 2, N(N(E, 3, E), 4, E))
```

associe la variable `t` à l'arbre



où on a écrit dans chaque nœud la donnée associée.

Parcours d'arbre. La présence d'un élément x dans un arbre A peut être ramenée à deux équations simples :

— l'arbre vide ne contient aucun élément

$$x \notin \emptyset$$

— un élément x présent dans un arbre non vide A peut être à la racine, dans le sous-arbre gauche ou dans le sous-arbre droit

$$x \in \overset{v}{\underset{A_1 \quad A_2}{\diagup\diagdown}} \iff x = v \ \vee \ x \in A_1 \ \vee \ x \in A_2$$

Toujours en utilisant le filtrage, on peut directement traduire ces équations en du code caml :

```
let rec mem x t = match t with
  | E          -> false
  | N(l, v, r) -> x = v || mem x l || mem x r
```

Dans le cas d'un arbre vide, cette fonction renvoie `false`. Dans le cas d'un arbre non vide elle teste d'abord si l'élément cherché est à la racine, et dans le cas contraire elle poursuit la recherche dans le sous-arbre gauche, puis dans le sous-arbre droit.

Note : l'opérateur `||` est *paresseux*. Si sa condition de gauche est vraie, il renvoie `true` sans évaluer la condition de droite. La fonction précédente est donc équivalente à :

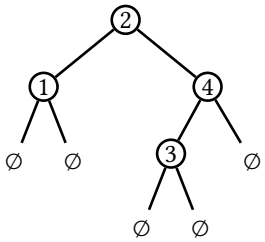
```
let rec mem x t = match t with
  | E          -> false
  | N(l, v, r) -> if x = v then true
                  else if mem x l then true
                  else mem x r
```

Autrement dit, cette fonction `mem` réalise un parcours en profondeur de l'arbre, et explore les branches de gauche à droite.

9.5 Représentation des arbres binaires en java

Comme on l'avait fait pour les listes, on peut représenter des arbres binaires à l'aide d'une unique classe représentant un nœud, et utiliser le pointeur nul pour l'arbre vide.

```
class N {
  final N l, r;
  final int v;
  N(N l, int v, N r) { this.l = l; this.v = v; this.r = r; }
  ...
```



On utilise alors la définition

```
N t = new N(new N(null, 1, null), 2, new N(new N(null, 3, null), 4, null));
```

pour définir l'arbre exemple rappelé dans la marge.

On peut ensuite ajouter à la classe des définitions de fonctions récursives réalisant les fonctions `size`, `height` et `mem` déjà vues.

```
  ...
  static int size(N t) {
    if (t == null) { return 0; }
    else           { return 1 + size(t.l) + size(t.r); }
  }
  static int height(N t) {
    if (t == null) { return 0; }
    else           { return 1 + Math.max(height(t.l), height(t.r)); }
  }
  static boolean mem(int x, N t) {
    if (t == null) { return false; }
    else           { return x == t.v || mem(x, t.l) || mem(x, t.r); }
  }
}
```

9.6 Structure : arbres binaires de recherche

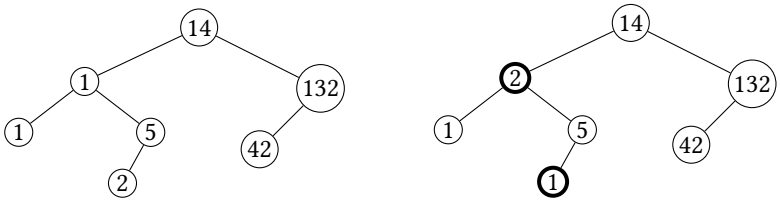
Dans un *arbre binaire de recherche*, on stocke une donnée dans chaque nœud d'une manière qui permet de localiser efficacement les données. Il s'agit de créer une structure d'arbre dans laquelle la recherche d'un élément peut être similaire à la recherche dichotomique :

- considérer un élément médian m,
- s'arrêter s'il s'agit de l'élément x cherché,
- sinon, chercher « à gauche » ou « à droite » selon que x est plus petit ou plus grand que m.

Avec une structure d'arbre binaire, on prend comme élément médian l'élément stocké dans la racine, et les directions « à gauche » et « à droite » consistent à poursuivre la recherche dans le sous-arbre gauche ou dans le sous-arbre droit.

L'invariant important des tableaux triés permettant la recherche dichotomique était : tous les éléments de la partie gauche sont inférieurs à l'élément médian, et tous les éléments de la partie droite sont supérieurs à l'élément médian. Ainsi par exemple, si on cherchait un élément plus petit que l'élément médian il était garanti qu'il ne se trouvait pas dans la partie droite, qui pouvait être ignorée dans la suite de la recherche. On peut transposer cette propriété dans le cadre des arbres.

Structure. Un *arbre binaire de recherche* (ABR) est un arbre binaire annoté dans lequel, pour tout nœud A portant un élément m, tous les éléments présents dans le sous-arbre gauche de A sont inférieurs à m et tous les éléments présents dans le sous-arbre droit de A sont supérieurs à m. L'arbre ci-dessous à gauche est un ABR. Celui de droite ne l'est pas : on y trouve une occurrence de 1 dans le sous-arbre droit d'un nœud portant la valeur 2.



Recherche et ajout. Dans de tels arbres, on peut transposer l'algorithme de recherche dichotomique. La condition d'arrêt selon laquelle on abandonne la recherche et on déclare que l'élément n'appartient pas à la collection lorsque la zone de recherche est vide correspond ici à tester la vacuité de l'arbre.

```
let rec mem x t = match t with
  | E                     -> false
  | N(_, m, _) when x = m  -> true
  | N(l, m, _) when x < m  -> mem x l
  | N(_, m, r) (* x > m *) -> mem x r
```

```
static boolean mem(int x, N t) {
  if (t == null)     { return false; }
  else if (x == t.v) { return true; }
  else if (x  < t.v) { return mem(x, t.l); }
  else /* x > t.v */ { return mem(x, t.r); }
}
```

Cet algorithme répond dans le pire cas en un temps proportionnel à la hauteur de l'arbre (logarithmique si l'arbre a une bonne forme, linéaire au pire).

Pour ajouter un élément dans un arbre binaire de recherche tout en préservant la propriété des ABR, il suffit de commencer par utiliser l'algorithme de recherche, puis :

— si l'élément a été trouvé, ne rien faire,

— sinon, on a atteint un sous-arbre vide, qu'on remplace par une feuille contenant l'élément à ajouter.

Dans cet algorithme, on a choisi de ne pas ajouter une deuxième copie d'un élément déjà présent.

```
let rec add x t = match t with
  | E                     -> N(E, x, E)
  | N(_, m, _) when x = m  -> t
  | N(l, m, r) when x < m  -> N(add x l, m, r)
  | N(l, m, r) (* x > m *) -> N(l, m, add x r)
```

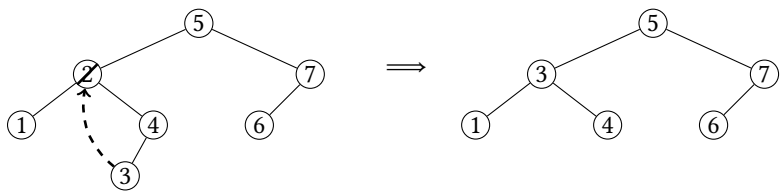
```
static N add(int x, N t) {
  if (t == null)     { return new N(null, x, null); }
  else if (x == t.v) { return t; }
  else if (x  < t.v) { return new N(add(x, t.l), t.v, t.r); }
  else /* x > t.v */ { return new N(t.l, t.v, add(x, t.r)); }
}
```

Approfondissement : retrait d'un élément. Pour retirer un élément d'un arbre binaire de recherche tout en préservant la propriété des ABR il faut commencer par aller chercher cet élément, puis on a différents cas de figure :

— Si l'élément à retirer est sur une feuille, il suffit de retirer la feuille.

— Si l'élément à retirer est sur un nœud ayant un seul fils (gauche ou droit), il suffit de retirer ce nœud et de le remplacer par son unique fils.

— Si l'élément à retirer est sur un nœud ayant deux fils, on le remplace par un nœud portant la plus petite valeur du sous-arbre droit (cette dernière étant alors retirée du sous-arbre droit).



En caml on introduit déjà une fonction `min` qui cherche le plus petit élément d'un arbre binaire de recherche (cet élément est « le plus à gauche »), et une fonction `remove_min` qui retire le plus petit élément d'un arbre binaire de recherche. L'une et l'autre ont comme précondition que l'arbre doit être non vide. On peut ensuite en déduire une fonction `remove` qui applique notre stratégie.

```
let rec min = function
  | E          -> assert false
  | N(E, v, _) -> v
  | N(l, _, _) -> min l

let rec remove_min = function
  | E          -> assert false
  | N(E, _, r) -> r
  | N(l, v, r) -> N(remove_min l, v, r)

let rec remove x t = match t with
  | E                     -> E
  | N(l, v, r) when x < v -> N(remove x l, v, r)
  | N(l, v, r) when x > v -> N(l, v, remove x r)
  | N(E, _, r) (* x=v *)  -> r
  | N(l, _, E) (* x=v *)  -> l
  | N(l, _, r) (* x=v *)  -> N(l, min r, remove_min r)
```

```
static int min(N t) {
  if (t == null)        { throw new IllegalArgumentException(); }
  else if (t.l == null) { return t.v; }
  else                  { return min(t.l); }
}
static N removeMin(N t) {
  if (t == null)        { throw new IllegalArgumentException(); }
  else if (t.l == null) { return t.r; }
  else                  { return new N(removeMin(t.l), t.v, t.r); }
}
static N remove(int x, N t) {
  if (t == null)    { return null; }
  else if (x < t.v) { return new N(remove(x, t.l), t.v, t.r); }
  else if (x > t.v) { return new N(t.l, t.v, remove(x, t.r)); }
  else /* x == v */
       if      (t.l == null) { return t.r; }
       else if (t.r == null) { return t.l; }
       else                  { return new N(t.l, min(t.r), removeMin(t.r)); }
}
```

9.7 Approfondissement : table associative

La structure d'arbre binaire de recherche peut directement être adaptée pour représenter une table associative. Il suffit de :

— faire porter deux éléments à chaque nœud : une clé et la valeur associée,

— ordonner les nœuds en fonction de leurs clés.

Si nos clés sont des chaînes de caractères, comme dans notre problème de recensement des mots d'un texte, on peut naturellement comparer deux clés à l'aide de l'ordre lexicographique (celui du dictionnaire). C'est le critère qui est utilisé dans les bibliothèque standard de caml et java (et de nombreux langages).

En caml. On crée un nouveau type d'arbre, où chaque nœud porte une chaîne de caractères (une clé) et un entier (une valeur).

```
type abr =
  | E
  | N of abr * string * int * abr
```

Les fonctions déjà vues s'adaptent naturellement. La recherche par exemple se fait à l'aide d'une clé, et répond à la question « cette clé est-elle présente dans la table ? ».

```
let rec mem k = function
  | E              -> false
  | N(l, k', _, r) -> if k < k' then mem k l
                      else if k > k' then mem k r
                      else true
```

Note : la fonction `mem` n'a jamais besoin de consulter les valeurs. On pourrait de même écrire une fonction `remove` retirant une clé d'un arbre (et la valeur associée), et une fonction `add` ajoutant une clé et une valeur un arbre.

En plus des fonctions déjà connues, on peut ajouter des fonctions manipulant les valeurs. Par exemple une fonction `find` renvoyant la valeur associé à une clé dans une table, avec la même structure que `mem`

Pourrait-on aussi écrire une fonction `mem_v`, qui indique si une valeur `v` est présente dans la table ? Avec quelle complexité ?

```
let rec find k = function
  | E              -> raise Not_found
  | N(l, k', v, r) -> if k < k' then find k l
                      else if k > k' then find k r
                      else v
```

ou encore une fonction `update` renvoyant une table mise à jour avec une nouvelle association clé/valeur (similaire à `add` mais modifiant une éventuelle valeur déjà présente).

```
let rec update k v = function
  | E               -> N (E, k, v, E)
  | N(l, k', v', r) -> if k < k' then N(update k v l, k', v', r)
                       else if k > k' then N(l, k', v', update k v r)
                       else N(l, k, v, r) (* on écrase la valeur précédente *)
```

9.8 Approfondissement : représentation alternative en java

Jusque-là, on a utilisé en java une classe unique pour les nœuds, et le pointeur nul pour l'arbre vide. En programmation objet, on peut alternativement utiliser plusieurs classes pour les différentes formes possibles d'une structure. On définit d'abord une interface commune, fixant le type général des arbres et déclarant les méthodes qu'on voudra pouvoir appliquer.

```
interface Tree {
  int size();
  int height();
  boolean mem(int x);
}
```

Puis on fournit une classe concrète pour chaque forme que peut prendre un arbre, avec définition de chacune des méthodes pour le cas correspondant. La classe concrète `E` va représenter l'arbre vide (on ne manipule donc plus de pointeurs nuls). Cette classe n'a pas besoin d'attributs et peut se contenter du constructeur par défaut. Elle contient en revanche des définitions pour chacune des méthodes, correspondant au cas `t == null` dans la version précédente.

```
class E implements Tree {
  public int size() { return 0; }
  public int height() { return 0; }
  public boolean mem(int x) { return false; }
}
```

La classe concrète `N` va représenter un nœud. On lui donne trois attributs représentant la valeur portée par le nœud et ses deux sous-arbres gauche et droit, et un constructeur adapté. Les définitions des différentes méthodes correspondent cette fois au cas récursif, et les appels récursifs sont remplacés par des appels des méthodes des sous-arbres.

```
class N implements Tree {
  public final Tree l, r;
  public final int v;
  public N(Tree l, int v, Tree r) { this.l = l; this.v = v; this.r = r; }

  public int size() { return 1 + l.size() + r.size(); }
  public int height() { return 1 + Math.max(l.height(), r.height()); }
  public boolean mem(int x) {
    if (x == v)      { return true; }
    else if (x < v)  { return l.mem(x); }
    else /* x > v */ { return r.mem(x); }
  }
}
```

Dans cette version, la définition de chaque fonction sur les arbres est donc répartie entre les différentes classes concrètes (vu dans l'autre sens : chaque classe est responsable, dans chaque fonction, de la partie qui la concerne). Pour ajouter une nouvelle fonction on va donc étendre l'interface, puis ajouter une définition à chaque classe. Pour ajouter une fonction `add` par exemple :

```
interface Tree {
  ...
  Tree add(int x);
}

class E extends Tree {
  ...
  public Tree add(int x) { return new N(new E(), x, new E()); }
}

class N extends Tree {
  ...
  public Tree add(int x) {
    if (x == v)      { return this; }
    else if (x < v)  { return new N(l.add(x), v, r); }
    else /* x > v */ { return new N(l, v, r.add(x)); }
  }
}
```

Avantage en revanche : on va gagner plus de souplesse pour définir des structures arborescentes plus riches, dans lesquelles les nœuds n'ont pas tous la même forme [8].

9.9 Approfondissement : arbres binaires de recherche équilibrés

Arbres binaires équilibrés. Un arbre binaire est *équilibré* si pour chaque nœud N de cet arbre les hauteurs des sous-arbres gauche et droit ne diffèrent pas de plus de 1.

Un arbre binaire équilibré de hauteur h a au moins $F_{h+2} - 1$ sommets, où $(F_k)_{k\in\mathbb{N}}$ est la suite de Fibonacci.

Rappel de la définition de la suite de Fibonacci.

$$\begin{cases} F_0 &= 0 \\ F_1 &= 1 \\ F_{n+2} &= F_{n+1} + F_n \end{cases}$$

Démonstration par récurrence forte sur h. On considère un arbre binaire équilibré A de hauteur h, et on suppose que tout arbre binaire équilibré de hauteur $k < h$ possède au moins $F_{k+2} - 1$ nœuds. Raisonnement par cas sur la forme de A.

- Si A est vide, on a $h = 0$ et 0 sommet, et on a bien $F_2 - 1 = 1 - 1 = 0$.
- Si A est formé par une racine et par deux sous-arbres A_1 et A_2, alors l'un des deux sous-arbres a la hauteur $h - 1$. Supposons qu'il s'agit de A_1 (l'autre cas sera symétrique). Comme $h - 1 < h$, par hypothèse de récurrence A_1 possède au moins $F_{(h-1)+2} - 1 = F_{h+1} - 1$ nœuds. Comme A est équilibré, la hauteur du deuxième sous-arbre A_2 diffère d'au plus 1 avec la hauteur $h-1$ de A_1. Donc la hauteur de A_2 est au moins $h-2$. Comme $h - 2 < h$, par hypothèse de récurrence A_2 a au moins $F_{(h-2)+2} - 1 = F_h - 1$ nœuds. Le nombre de nœuds total est donc au moins $1+(F_{h+1}-1)+(F_h-1) = F_{h+1}+F_h-1 = F_{h+2}-1$.

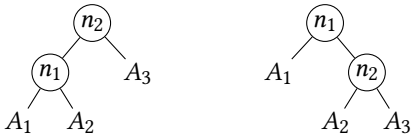
La suite de Fibonacci ayant une croissance exponentielle, on en déduit qu'un *arbre binaire équilibré a une hauteur logarithmique* en son nombre de nœuds.

AVL comme Adelson-Velski et Landis, les inventeurs.

Arbres AVL. Les arbres AVL sont des arbres binaires de recherche qui sont toujours équilibrés. Les opérations de recherche, d'insertion et de suppression d'un élément utilisent les mêmes algorithmes que ceux déjà vus pour les arbres de recherche, à une différence près : à chaque fois qu'un nœud est ajouté à ou retiré d'un sous-arbre, on vérifie l'équilibrage de ce sous-arbre. Si le nœud ajouté ou retiré amène un déséquilibre, alors on réarrange le sous-arbre concerné pour rétablir l'équilibre.

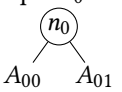
Le rééquilibrage est basé sur une opération de *rotation*, qui transforme l'arbre ci-dessous à gauche en celui à droite (et inversement).

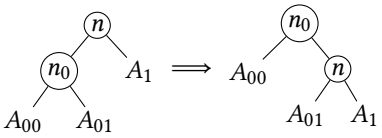
8. Vous en verrez des exemples : en TP, dans un chapitre ultérieur, mais aussi en PIL.

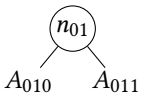


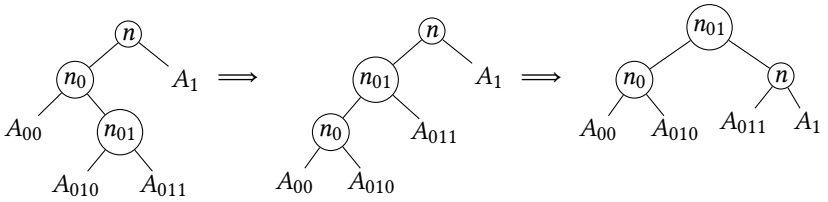
Une telle rotation, appliquée à un arbre binaire de recherche, préserve sa qualité d'arbre de recherche. En revanche, selon les profondeurs respectives des sous-arbres A_1, A_2 et A_3 une telle rotation peut modifier l'équilibre de l'arbre (en l'occurrence, on utilisera des rotations pour corriger un équilibre menacé).

Cas d'étude. On part d'un arbre équilibré et on ajoute un élément x à son sous-arbre gauche. On obtient un arbre  dans lequel on peut avoir un déséquilibre du côté gauche si l'ajout d'un élément a fait augmenter la hauteur du sous-arbre gauche. En notant h_0 et h_1 les hauteurs des deux sous-arbres A_0 et A_1 le risque est donc que $h_0 = h_1 + 2$.

Dans cette situation, au moins un des deux sous-arbres de A_0 =  a une hauteur de $h_1 + 1$. S'il s'agit du sous-arbre de gauche A_{00}, une rotation simple suffit à rétablir l'équilibre.



S'il s'agit du sous-arbre de droite A_{01} =  , on enchaîne deux rotations.



Réalisation en caml. On enrichit le type des arbres pour étiqueter chaque nœud avec sa hauteur.

```
type avl =
  | E
  | N of avl * int * avl * int
```

On obtient la hauteur d'un arbre non vide en lisant le champ correspondant.

```
let height = function
  | E            -> 0
  | N(_, _, _, h) -> h
```

Pour alléger la création des nœuds on utilise la fonction suivante, qui renseigne elle-même la hauteur.

```
let node l n r =
  N(l, n, r, 1 + max (height l) (height r))
```

L'ajout d'un élément se fait comme précédemment, à ceci près que chaque nouveau nœud est créé par une fonction auxiliaire qui fait le rééquilibrage si nécessaire. On utilise deux fonctions de rééquilibrage aux fonctionnements symétriques, pour les ajouts à gauche ou à droite.

```
let rec add x t = match t with
  | E                      -> node E x E
  | N(_, n, _, _) when x=n  -> t
  | N(l, n, r, _) when x<n  -> rebalance_left (add x l) n r
  | N(l, n, r, _) (* x>n *) -> rebalance_right l n (add x r)
```

La fonction chargée de corriger les déséquilibres à gauche est une traduction directe des rotations décrites plus haut.

```
let rebalance_left a0 n a1 =
  if height a0 > height a1 + 1 then
    match a0 with
    | N(a00, n0, a01, _) when height a00 >= height a01 ->
       node a00 n0 (node a01 n a1)
    | N(a00, n0, N(a010, n01, a011, _), _) ->
       node (node a00 n0 a010) n01 (node a011 n a1)
    | _ -> assert false
  else
    node a0 n a1
```

La fonction `rebalance_right` s'écrirait de manière symétrique, et ces deux fonctions pourraient de même être utilisées pour que la fonction `remove` préserve l'équilibre d'un AVL.

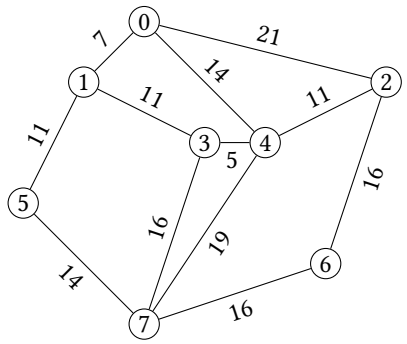
10 Trouver un raccourci

10.1 Problème : l'itinéraire le plus court

Imaginons une carte routière avec ses routes, ses villes et ses carrefours, et supposons que l'on cherche à déterminer un itinéraire le plus court possible entre une ville de départ et une ville d'arrivée.

On peut imaginer résumer notre carte par un graphe, dont les sommets seraient les villes et les arêtes les routes reliant deux villes. Nous avons déjà vu que les parcours en profondeur et en largeur permettaient de trouver des chemins depuis un sommet source vers les autres sommets accessibles d'un graphe, et également que le parcours en largeur trouvait systématiquement des chemins utilisant un nombre minimal d'arêtes. Cependant, l'itinéraire empruntant le nombre minimal de routes n'est pas nécessairement celui qui nous intéresse ici : toutes les routes n'ont pas la même longueur. Autrement dit : certaines arêtes de notre graphe représentent un chemin plus long que d'autres, et deux arêtes « courtes » peuvent être préférables à une arête « longue ».

On enrichit notre modèle de graphe pour ajouter à chaque arête un « coût », (appelé traditionnellement *poids*), sous la forme d'un nombre qui sera d'autant plus faible que l'arête sera intéressante. Ici, ce coût représentera directement la longueur du chemin représenté par l'arête. On cherche alors des *chemins les plus courts*, c'est-à-dire des chemins pour lesquels la somme des poids des arêtes est minimale.



Ces *graphes pondérés* ont une interface très similaire à l'interface déjà utilisée pour les graphes ordinaires : on y ajoute principalement une fonction `int weight(int s, int t)` donnant le poids de l'arête allant du sommet `s` au sommet `t` (en supposant qu'une telle arête existe bien). Pour le code java de ce chapitre, on utilisera ainsi l'interface suivante.

```
interface WGraph {
  int size();                    // nombre de sommets
  Iterable<Integer> succ(int s); // voisins de s
  int weight(int s, int t);      // poids de l'arête s --> t
}
```

On peut en imaginer une réalisation inspirée des listes d'adjacence, où la listes des voisins d'un sommet `s` est remplacée par une table de hachage associant chaque voisin `t` de `s` à sa distance.

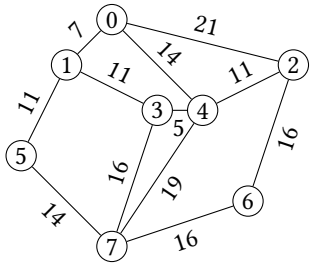
```
class WGraphAdj implements WGraph {
  private final int size;
  private ArrayList<HashMap<Integer, Integer>> adj;

  WGraphAdj(int size) {
    this.size = size;
    this.adj = new ArrayList<>(size);
    for (int s=0; s<size; s++) { adj.add(new HashMap<>()); }
  }
  void addEdge(int s, int t, int d) { adj.get(s).put(t, d); }

  public int size() { return this.size; }
  public Iterable<Integer> succ(int s) { return adj.get(s).keySet(); }
  public int weight(int s, int t) { return adj.get(s).get(t); }
}
```

10.2 Algorithme de Dijkstra

L'algorithme de Diskstra est une adaptation du parcours en largeur qui permet de trouver les chemins les plus courts dans un graphe pondéré [9]. Dans l'un et l'autre, on souhaite explorer les sommets par ordre croissant de « distance » à la source. Les deux diffèrent par la notion de distance considérée, et la structure utilisée pour stocker les sommets en attente.

— Dans le parcours en largeur, la distance est évaluée en nombre d'arêtes. On utilise une file FIFO, et on explore les sommets dans l'ordre de leur découverte.

— Dans l'algorithme de Dijkstra, la distance est évaluée par le poids des chemins. L'ordre de découverte des sommets ne correspond plus nécessairement aux distances croissantes à la source. À chaque nouvelle étape on choisit, parmi les sommets en attente, celui qui est à la plus courte distance de la source.



Voici un exemple d'exécution de l'algorithme. On explore le graphe donné dans la marge en prenant comme sommet s d'origine le sommet numéro 5. Le tableau montre à chaque ligne :

— le sommet `t` exploré,

— les sommets en attente, classés par ordre de distance croissante,

— un tableau `dist` tel que `dist[t]` est le poids du meilleur chemin connu vers `t`,

— un tableau `pred` tel que `pred[t]` précède `t` dans le meilleur chemin connu vers `t`,

— un tableau `visited` mémorisant les sommets déjà explorés.

		dist								pred								visited							
t	En attente	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
–	**5(0)**	–	–	–	–	–	**0**	–	–	–	–	–	–	–	**5**	–	–	–	–	–	–	–	–	–	–
5	**1(11)**, **7(14)**	–	**11**	–	–	–	0	–	**14**	–	**5**	–	–	–	5	–	**5**	–	–	–	–	–	✓	–	–
1	7(14), **0(18)**, **3(22)**	**18**	11	–	**22**	–	0	–	14	**1**	5	–	**1**	–	5	–	5	–	✓	–	–	–	✓	–	–
7	0(18), 3(22), **6(30)**, **4(33)**	18	11	–	22	**33**	0	**30**	14	1	5	–	1	**7**	5	**7**	5	–	✓	–	–	–	✓	–	✓
0	3(22), 6(30), 4(**32**), **2(39)**	18	11	**39**	22	**32**	0	30	14	1	5	**0**	1	**0**	5	7	5	✓	✓	–	–	–	✓	–	✓
3	4(**27**), 6(30), 2(39)	18	11	39	22	**27**	0	30	14	1	5	0	1	**3**	5	7	5	✓	✓	–	✓	–	✓	–	✓
4	6(30), 2(**38**)	18	11	**38**	22	27	0	30	14	1	5	**4**	1	3	5	7	5	✓	✓	–	✓	✓	✓	–	✓
6	2(38)	18	11	38	22	27	0	30	14	1	5	4	1	3	5	7	5	✓	✓	–	✓	✓	✓	✓	✓
2		18	11	38	22	27	0	30	14	1	5	4	1	3	5	7	5	✓	✓	✓	✓	✓	✓	✓	✓

À noter dans cet exemple d'exécution, un point qui diffère avec le parcours en largeur déjà connu : il est possible ici, à un moment où l'on connaît déjà des chemins vers un certain sommet `t`, de découvrir un nouveau chemin meilleur que ceux déjà connus. Dans ce cas, on met à jour les tableaux `dist` et `pred` pour ne mémoriser à chaque fois que le meilleur chemin connu. Dans l'exemple, cela apparaît par exemple avec le sommet s_4 : on découvre en premier un chemin de longueur 33 venant de s_7, puis un chemin de longueur 32 venant de s_0, et enfin un chemin de longueur 27 venant de s_3.
À la fin de l'exécution, en consultant `dist` et `pred` on apprend :

— que s_2 est à distance 38, en passant par s_4,

— que s_4 est à distance 27, en passant par s_3,

— que s_3 est à distance 22, en passant par s_1,

— que s_1 est à distance 11, directement depuis s_5.

Ainsi, la séquence 5, 1, 3, 4, 2 est le chemin le plus court de 5 vers 2 (de longueur totale 38).

Complexité ? Présenté ainsi, l'algorithme semble suffisamment similaire au parcours en largeur pour avoir la même complexité, linéaire en les nombres de sommets et d'arêtes du graphe. Cependant, un coût supplémentaire est caché à chaque étape dans la sélection du prochain sommet à explorer. On peut imaginer plusieurs scénarios.

1. Ranger les sommets à traiter par ordre croissant de distance dans un tableau ou une liste. La sélection du prochain sommet est immédiate, mais chaque insertion d'un nouveau sommet est linéaire (car il faut l'insérer à la bonne place).
2. Laisser les sommets à traiter dans le désordre, et les parcourir à la recherche du minimum à chaque nouvelle étape. L'insertion est efficace, mais la sélection est linéaire.
3. Trouver une nouvelle structure de données, dans laquelle l'insertion *et* la sélection seront toutes les deux efficaces !

Évidemment, nous allons regarder ici ce dernier scénario, avec la structure de *file de priorité*.

9. À condition que les poids des arêtes soient bien positifs ou nuls !

File de priorité. On fixe pour la file de priorité une interface `PQueue`, dont les principales méthodes sont : ajouter un nouvel élément à la file, en précisant son niveau de priorité, et retirer de la file l'élément le plus prioritaire. Ici, on représente le niveau de priorité par un entier positif d'autant plus petit que l'élément est prioritaire.

```
interface PQueue {
  boolean isEmpty();         // teste si la file est vide
  void insert(int x, int p); // insère l'élément x avec la priorité p
  int extractMin();          // retire et renvoie l'élément de priorité minimale
}
```

Voici une réalisation en java de l'algorithme de Dijkstra, avec une structure proche du parcours en largeur déjà vu, mais en utilisant une file de priorité à la place de la file FIFO. On utilise également les différents éléments déjà énumérés dans l'exemple d'exécution : `t` est le sommet en cours d'exploration, le tableau `visited` indique les sommets déjà explorés, les tableaux `dist` et `pred` donnent les informations des meilleurs chemins connus à un moment donné (on fixe `dist[u]` = `pred[u]` = -1 si on n'a pas encore découvert de chemin atteignant un sommet `u`). Lorsque l'on explore un sommet `t` on considère chacun de ses voisins. Si un voisin `v` n'avait encore jamais été découvert, *ou si le chemin menant à* `v` *en passant par* `t` *est meilleur que le meilleur chemin vers* `v` *déjà connu*, alors on renseigne les tableaux `dist` et `pred` avec les nouvelles informations et on ajoute `v` à la file de priorité. On donne comme « priorité » à `v` le poids du chemin vers `v` que l'on vient de découvrir. Ainsi, les sommets explorés par ordre de priorité croissante seront bien explorés par ordre de distance à la source.

```
static int[] dijkstra(WGraph g, int s) {
  // initialisation
  int n = g.size();
  boolean[] visited = new boolean[n];
  int[] dist = new int[n]; for (int t=0; t<n; t++) dist[t] = -1; // dist. source
  int[] pred = new int[n]; for (int t=0; t<n; t++) pred[t] = -1; // chemins

  // file de priorité
  PQueue pqueue = new ... ; // insérer ici la classe concrète utilisée

  // insertion de la source
  pqueue.insert(s, 0); dist[s] = 0; pred[s] = s;

  // exploration
  while (!pqueue.isEmpty()) {
    // on considère le sommet non visité à plus courte distance de la source
    int t = pqueue.extractMin();
    if (visited[t]) continue;
    visited[t] = true;
    for (int v: g.succ(t)) {
      // insertion d'un successeur dans la file si vu pour la première fois
      // ou si chemin plus court que le meilleur connu actuellement
      int d = dist[t] + g.weight(t, v);
      if (dist[v] < 0 || d < dist[v]) {
        pqueue.insert(v, d); dist[v] = d; pred[v] = t;
      }
    }
  }
  return pred;
}
```

À noter : notre interface de file de priorité ne propose pas de fonction permettant de « mettre à jour » la priorité d'un élément déjà présent. À la place, on ajoute l'élément à nouveau. Dans cet algorithme, il est donc possible qu'un même sommet `t` apparaisse plusieurs fois dans la file de priorité, avec plusieurs priorités correspondant chacune à la longueur de l'un des chemins de `s` à `t` découverts. Chacun ne sera cependant exploré qu'une fois, correspondant à la meilleure priorité : lorsque l'ordre de la file nous amène à une deuxième occurrence (moins prioritaire) le sommet a déjà été marqué dans le tableau `visited`.

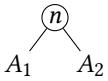
Les files de priorité avec possibilité de mise à jour existent ! Mais elles sont plus complexes que ce que nous allons voir ici.

Bilan. L'efficacité de l'algorithme de Dijkstra va dépendre de la structure utilisée pour la file de priorité. Dans la suite du chapitre, nous allons voir une nouvelle utilisation des arbres binaires permettant de réaliser une file de priorité efficace.

10.3 Structure : tas binaire

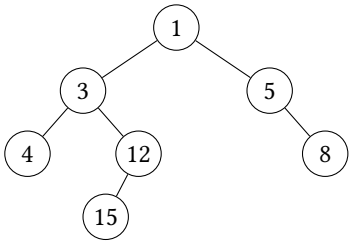
Un *tas binaire* est un arbre binaire dont chaque nœud contient un élément, et où l'élément porté par un nœud est inférieur ou égal aux éléments portés par ses fils. Autrement dit :

— l'arbre vide est un tas,

— un arbre composé de la forme 

est un tas si A_1 et A_2 sont tous deux des tas, et si n est inférieur ou égal aux racines de A_1 et A_2 (et si A_1 ou A_2 est vide, il n'y a rien à vérifier pour ce sous-arbre).

Propriété essentielle d'un tas : le plus petit élément est systématiquement à la racine.



Consulter l'élément minimal d'un tas est facile : c'est l'élément porté par la racine. On veut également des algorithmes efficaces pour retirer cet élément minimal (c'est-à-dire pour reformer un tas avec le reste), et pour ajouter un élément quelconque (à une position qui respecte la propriété d'ordre). On a plusieurs stratégies pour cela.

Première réalisation : *skew heap*. Un tas étant un arbre binaire, on peut reprendre directement l'une des défininitions déjà vues au chapitre précédent. Par exemple en caml.

```
type heap =
  | E
  | N of heap * int * heap
```

On a déjà dit que l'élément minimal d'un tas se trouvait à la racine. On l'obtient donc directement ainsi.

```
let min t = match t with
  | E          -> raise Not_found
  | N(_, x, _) -> x
```

Nous allons voir que l'ajout d'un élément quelconque ou le retrait de la racine peuvent tous deux être ramenés à une unique opération de fusion de deux tas. Supposons donc disposer d'une fonction merge qui prend deux tas t_1 et t_2 en paramètres, et qui renvoie un nouveau tas combinant les éléments de t_1 et t_2.

— Ajout d'un élément `x` à un tas `t`. Remarquons d'abord que l'arbre `N(E, x, E)` est un tas (quel que soit `x`). On peut donc réaliser l'ajout en fusionnant `t` avec le tas précédent contenant l'unique élément `x` :

$$\mathsf{add}(x, t) \quad = \quad \mathsf{merge}(\mathsf{N}(\mathsf{E}, x, \mathsf{E}), t)$$

Le code caml est immédiat.

```
let add x t =
  merge (N(E, x, E)) t
```

— Retirer la racine d'un tas revient à reformer un tas avec les éléments de ses deux sous-arbres, qui sont eux-même des tas. On peut prendre pour cela le résultat de la fusion de ces deux sous-arbres.

$$\mathsf{remove\_min}(\mathsf{N}(A_1, x, A_2)) \quad = \quad \mathsf{merge}(A_1, A_2)$$

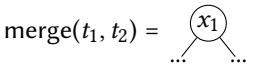
Le code caml s'obtient également directement.

```
let remove_min = function
  | E          -> raise Not_found
  | N(l, x, r) -> merge l r
```

Fusion de *skew heaps*. Ne reste plus qu'à savoir réaliser cette fusion. Si l'un ou l'autre des tas à fusionner est vide le résultat est immédiat : il suffit de prendre l'autre. Supposons alors avoir deux tas non vides.

On cherche à former un nouveau tas contenant tous les éléments de t_1 et de t_2. Ce tas à construire devra avoir son plus petit élément à la racine. Comme t_1 et t_2 sont tous deux des tas, on sait que x_1 est le plus petit élément de t_1, et x_2 le plus petit élément de t_2. Le plus petit élément de l'ensemble ne peut donc être que x_1 ou x_2. On compare donc x_1 et x_2.

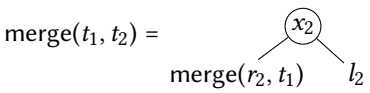
— Si $x_1 \leq x_2$, alors x_1 est le plus petit élément du tas fusionné, qui devra nécessairement avoir la forme



où les deux sous-arbres se partagent les éléments non encore utilisés, c'est-à-dire les deux sous-tas l_1, r_1 de t_1, et le tas t_2 entier. On sait que l_1, r_1 et t_2 sont trois tas et ne contiennent que des éléments supérieurs ou égaux à x_1 : chacun aurait le droit de constituer l'un des deux sous-arbres de notre résultat. Cependant, il n'y a que deux places pour trois : en supposant que l'on utilise directement l_1 ou r_1 ou t_2 comme sous-arbre droit, il faut fusionner les deux autres pour avoir un unique sous-arbre gauche. *N'importe quel choix de répartition formerait un tas correct.* En voici deux possibles :

Dans la version de gauche, on garde ce qui vient de t_1 d'un côté et ce qui vient de t_2 de l'autre. Dans la version de droite, on les entremêle (et on fait passer à gauche le sous-arbre droit r_1 de t_1 et à droite son sous-arbre gauche l_1). Dans les deux cas, la complexité de l'opération de fusion est proportionnelle à la hauteur des arbres considérés, et la fusion est donc efficace si les arbres sont équilibrés. Il se trouve que la solution de droite, en faisant tourner les sous-arbres à chaque opération, évite de tout le temps charger le même côté et permet ainsi de maintenir un bon équilibre *en moyenne*[10].

— Si à l'inverse $x_1 > x_2$, on peut faire une construction symétrique :



Résumé de tout ceci en quelques lignes de caml :

```
let rec merge t1 t2 = match t1, t2 with
  | E, t | t, E -> t
  | N(l1, x1, r1), N(l2, x2, r2) ->
     if x1 <= x2 then N (merge r1 t2, x1, l1)
                 else N (merge r2 t1, x2, l2)
```

Utilisation d'un tas pour l'algorithme de Dijkstra. On adapte directement la structure que l'on vient de voir, en stockant non pas un uniquement élément `int`, mais une paire formée d'un sommet et de sa priorité, et on remplace la comparaison `x1 <= x2` par une comparaison des priorités. En voici la transposition en java avec une classe `N` représentant un nœud d'arbre binaire (où la valeur est sockée dans un attribut `v`, et la priorité dans un attribut `p`). On réalise enfin l'interface `PQueue` avec une deuxième classe possédant pour unique attribut un (pointeur vers un) tel nœud, et qui modifie cet attribut à chaque fabrication d'un nouvel arbre.

10. Cette réalisation de la fusion ne produit *pas que* des arbres équilibrés. Une analyse fine permet cependant d'assurer que, sur la durée, les fusions successives appliquées à un même tas auront une complexité moyenne logarithmique : si certaines fusions s'avèrent ponctuellement très coûteuses, il est certain qu'elles seront compensées par de nombreuses fusions très peu coûteuses. On parle ici d'analyse de complexité *amortie*. C'est ce même concept qui est à l'œuvre lorsque l'on dit que l'ajout d'un élément à un tableau redimensionnable peut-être considéré comme une opération en temps constant, alors même que ponctuellement cet ajout peut nécessiter une copie de l'ensemble du tableau.

```
class N {
  final int v, p;
  final N l, r;
  N(N l, int v, int p, N r) {
    this.l = l; this.v = v; this.p = p; this.r = r;
  }

  static N add(int v, int p, N t) {
    return merge(new N(null, v, p, null), t);
  }
  static int min(N t) {
    if (t == null) { throw new IllegalArgumentException(); }
    else           { return t.v; }
  }
  static N removeMin(N t) {
    if (t == null) { throw new IllegalArgumentException(); }
    else           { return merge(t.l, t.r); }
  }
  static N merge(N t1, N t2) {
    if (t1 == null)        { return t2; }
    else if (t2 == null)   { return t1; }
    else if (t1.p <= t2.p) { return new N(merge(t1.r, t2), t1.v, t1.p, t1.l); }
    else                   { return new N(merge(t2.r, t1), t2.v, t2.p, t2.l); }
  }
}
```

```
class SkewQueue implements PQueue {
  private N heap;
  SkewQueue() { this.heap = null; }
  public boolean isEmpty() { return heap == null; }
  public void insert(int v, int p) { heap = N.add(v, p, heap); }
  public int extractMin() {
    int v = N.min(heap);
    heap = N.removeMin(heap);
    return v;
  }
}
```

Pour compléter notre code de l'algorithme de Dijkstra, il suffit maintenant de compléter la ligne manquante par

```
PQueue pqueue = new SkewQueue();
```

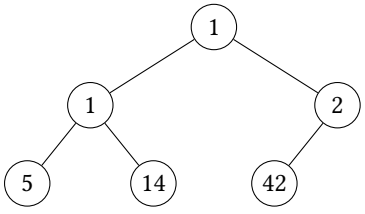
10.4 Approfondissement : tas binaire mutable

Dans certains cas, on peut représenter un arbre binaire par un tableau, où chaque case correspond à un nœud. L'idée est la suivante.

- La racine correspond à la case d'indice 0.
- Le fils gauche du nœud d'indice k est le nœud d'indice $2k + 1$.
- Le fils droit du nœud d'indice k est le nœud d'indice $2k + 2$.

Et donc le parent du nœud d'indice $k > 0$ a l'indice $\left\lfloor \frac{k-1}{2} \right\rfloor$.

Ainsi, l'arbre



est représenté par le tableau suivant.

0	1	2	3	4	5
1	1	2	5	14	42

On parle de représentation *implicite* de l'arbre binaire. *Attention cependant :* ceci n'est adapté que lorsque la forme de l'arbre s'y prête. En l'occurrence il faut un arbre bien équilibré et même dans l'idéal un arbre binaire *quasi-parfait*, c'est-à-dire un arbre dans lequel tous les niveaux de profondeur sont complètement remplis, sauf éventuellement le dernier niveau (dans lequel on demande que les nœuds soient serrés aussi à gauche que possible). En effet, dans le cas contraire le tableau contiendrait des « trous ».

On peut utiliser cette structure pour représenter un tas, dont le bon équilibre est contrôlé de manière stricte. Il s'agira cette fois d'un tas *mutable* : l'ajout ou l'extraction d'un élément modifie le tas en place. Aussi, pour pouvoir traiter l'algorithme de Dijkstra, on stocke dans le tableau des paires formées par une valeur et une priorité.

```
class BinaryQueue implements PQueue {
  // Classe représentant les paires (valeur, priorité)
  static class PrioPair {
    final int v, p;
    PrioPair(int v, int p) { this.v = v; this.p = p; }
  }

  // Support de la file de priorité, avec fonctions de comparaison et d'échange
  ArrayList<PrioPair> t;
  boolean prio(int i, int j) { return t.get(i).p < t.get(j).p; }
  void swap(int i, int j) { Collections.swap(t, i, j); }

  // Fonctions auxiliaires ciblant les voisins utiles
  int parent(int i) { return (i-1)/2; }
  int left(int i)   { return 2*i+1; }
  int right(int i)  { return 2*i+2; }

  BinaryQueue() { this.t = new ArrayList<>(); }
  public boolean isEmpty() { return t.size() == 0; }
```

On insère un nouvel élément en l'ajoutant à la fin du tableau, c'est-à-dire comme une nouvelle feuille. On fait ensuite remonter cet élément vers la racine jusqu'à ce qu'il atteigne une place légitime. Pour cela, la fonction auxiliaire `moveUp` permute l'élément ciblé avec son parent, tant qu'il est strictement plus prioritaire que son parent.

```
  void moveUp(int i) {
    if ( i > 0 && prio(i, parent(i)) ) {
      swap(i, parent(i));
      moveUp(parent(i));
    }
  }

  public void insert(int v, int p) {
    t.add(new PrioPair(v, p));
    moveUp(t.size() - 1);
  }
```

L'opération d'extraction est un peu plus complexe : on remplace l'élément retiré à la racine par l'élément de la dernière case, puis on le fait descendre dans l'arbre tant qu'au moins un de ses fils est plus prioritaire que lui. Point d'attention : lorsque l'on fait descendre un élément il faut l'échanger avec le plus petit de ses deux fils. Il faut aussi prévoir le cas, possible à la fin, où le nœud considéré a un seul fils, qui est alors nécessairement à gauche. La fonction auxiliaire `minChild` sélectionne le plus prioritaire des deux fils (ou l'unique fils le cas échéant), et la fonction auxiliaire `moveDown` réalise les échanges.

```
  int minChild(int i) {
    if ( right(i) < t.size() && prio(right(i), left(i)) ) { return right(i); }
    else                                                  { return left(i); }
  }
  void moveDown(int i) {
    if ( left(i) < t.size() && prio(minChild(i), i) ) {
      swap(i, minChild(i));
      moveDown(minChild(i));
    }
  }
```

```
  public int extractMin() {
    int r = t.get(0).v;
    swap(0, t.size()-1);
    t.remove(t.size()-1);
    moveDown(0);
    return r;
  }
}
```

10.5 Approfondissement : Dijkstra, en caml

On a vu que caml permettait de réaliser facilement une structure de tas. Voyons comment compléter la réalisation de l'algorithme de Dijkstra dans ce langage également.

Graphes pondérés. Pour commencer, fixons un type pour les graphes pondérés. On peut utiliser un tableau de listes d'adjacence comme déjà vu en java. Ici plus précisément, on représente une liste d'adjacence avec une liste primitive de caml, qui contient des paires (voisin, longueur de l'arête). La fonction `succ` renvoie directement cette liste de paires (on n'aura donc pas besoin ici de la fonction `weight` vue dans l'interface java).

```
type wgraph = (int * int) list array

let size g = Array.length g
let succ s g = g.(s)
```

File de priorité polymorphe. Comme déjà vu avec java, notre file de priorité devra contenir à la fois des sommets et leurs priorités. Voici une définition *polymorphe* (ou *générique*) de la structure de tas qui va permettre cela. Le *paramètre de type* `'a` y désigne un type de données quelconque, et le type `'a heap` est un tas dont les nœuds portent des valeurs de type `'a`.

```
type 'a heap =
  | E
  | N of 'a heap * 'a * 'a heap
```

La version que nous avions déjà vue correspondait à `int heap`, mais tout le code donné pour les tas fonctionne également avec un type `'a` quelconque. En particulier, on voudra maintenant que ce `'a` ne représente pas un entier seul, mais une paire `int * int` contenant une priorité et un numéro de sommet (en caml, contrairement à java, les paires sont des types primitifs).

Petite subtilité sur la comparaison < en caml : elle est elle-même polymorphe. Elle ne s'applique donc pas seulement aux nombres entiers, mais à tout type de données. Lors de la comparaison de structures de données à plusieurs champs elle applique l'ordre lexicographique. En particulier, pour comparer deux paires (p_1, v_1) et (p_2, v_2), elle compare d'abord p_1 et p_2, et ne compare ensuite v_1 et v_2 qu'en cas d'égalité sur la composante p. Ici, nous allons donc utiliser des paires dans lesquelles le premier élément est la priorité, et le deuxième élément est le numéro du sommet (et ainsi le code déjà donné est encore valide).

Ensuite, on définit une file de priorité comme un pointeur vers un tel tas. Note caml : un type `'b ref` est le type d'une *référence*, c'est-à-dire d'un pointeur, vers un élément de type `'b`. Ici, `'b` est le type d'un tas, c'est-à-dire `'a heap`. On crée un nouveau pointeur avec l'opérateur `ref`, on lit la valeur associée avec l'opérateur `!` de *déréférencement*, et on modifie la valeur avec l'opérateur `:=` d'*affectation*.

Note : les parenthèses dans `('a heap) ref` sont facultatives.

```
type 'a pqueue = ('a heap) ref

let create ()  = ref E
let is_empty q = !q = E
let insert e q = q := add e !q
let extract_min q =
  let e = min !q in
  q := remove_min !q;
  e
```

Dijkstra. Le code a la même structure que son équivalent en java, avec quelques différences mineures : on définit une fonction auxiliaire locale `upd` qui regroupe les opérations d'insertion d'un sommet dans la file de priorité, et de mise à jour de ses informations dans `dist` et `pred` (le code correspondant n'est donc écrit qu'une fois, mais utilisé à deux endroits), lors de l'extraction d'un sommet de la pile on récupère directement sa distance (en java on ne récupérait que le numéro du sommet, puis on consultait `dist`), et l'itération sur les voisins d'un sommets se fait avec la fonction d'itération `iter` (en java on avait une boucle *for each*).

```
let dijkstra s g =
  (* Initialisation *)
  let n = WGraph.size g in
  let visited = Array.make n false in
  let dist = Array.make n (-1) in
  let pred = Array.make n (-1) in
  let pqueue = PQueue.create () in

  (* Fonction auxiliaire pour mettre à jour les informations d'un sommet
     et l'ajouter à la file de priorité *)
  let upd v d t =
    PQueue.insert (d, v) pqueue;
    dist.(v) <- d;
    pred.(v) <- t
  in
  upd s 0 s;

  while not (PQueue.is_empty pqueue) do
    let dt, t = PQueue.extract_min pqueue in
    if (not visited.(t)) then
      (visited.(t) <- true;
       (* Itération sur l'ensemble des voisins, pour insérer dans la file
          et mettre à jour ceux qui doivent l'être *)
       List.iter (fun (v, dtv) ->
           if (dist.(v) < 0 || dt+dtv < dist.(v)) then upd v (dt+dtv) t)
       (WGraph.succ t g))
  done;

  dist, pred
```

11 Ne pas se casser la tête

Un club écossais très *select* a fixé les règles suivantes pour autoriser l'adhésion de nouveaux membres.

1. Les membres non écossais doivent porter des chaussettes rouges.
2. Un membre sans kilt ne peut pas porter de chaussettes rouges.
3. Les membres mariés ne doivent pas sortir se promener le dimanche.
4. Un membre sort se promener le dimanche si et seulement s'il est écossais.
5. Tout membre portant un kilt doit être écossais et marié.

À se demander s'il est réellement possible d'entrer dans ce cercle... Pouvez-vous déterminer si certaines situations permettent réellement l'adhésion, et si oui lesquelles ?

Modélisation par des formules. Commençons par recenser les différentes caractéristiques des candidats qui sont concernées par ces règles, et associons une lettre à chacune.

E : Être écossais $\quad$ C : Porter des chaussettes rouges $\quad$ K : Porter le kilt
M : Être marié $\quad$ S : Sortir se promener le dimanche

Les règles d'adhésion peuvent ensuite être modélisées par des formules logiques liant ces différentes caractéristiques.

1. $\neg E \Rightarrow C$
2. $\neg K \Rightarrow \neg C$
3. $M \Rightarrow \neg S$
4. $E \iff S$
5. $K \Rightarrow (E \wedge M)$

On en déduit une traduction logique de notre problème : déterminer les combinaisons de valeurs de vérité pour les cinq variables propositionnelles E, C, K, M et S rendant vraie la formule d'adhésion résumée ici.

$$(\neg E \Rightarrow C) \ \wedge \ (\neg K \Rightarrow \neg C) \ \wedge \ (M \Rightarrow \neg S) \ \wedge \ (E \iff S) \ \wedge \ (K \Rightarrow (E \wedge M))$$

Résolution automatique ? Vous avez déjà appris à construire, à la main, une table de vérité pour une telle formule, qui permet de résoudre ce problème. Ici, nous aurions un tableau de 32 lignes recensant toutes les combinaisons possibles de valeurs de vérité pour nos 5 variables propotionnelles, et pour chacune la valeur de vérité de la formule complète. Nous allons maintenant voir comment traiter ce problème d'une manière plus algorithmique, en travaillant directement sur la structure des formules. Et cette structure va être... *un arbre !*

11.1 Structure récursive des formules logiques

Jusqu'ici, nous avons décrit une formule logique comme « une combinaison de propositions liées par des connecteurs ». Regardons de plus près la structure d'un tel objet. Une *formule* de la logique propositionnelle est :

- soit l'une des formules atomiques $\top$ (tautologie) ou $\bot$ (contradiction),
- soit une variable propositionnelle X,
- soit la négation $\neg F$ d'une formule F,
- soit la combinaison $F_1 \wedge F_2$ (conjonction), ou $F_1 \vee F_2$ (disjonction), ou $F_1 \Rightarrow F_2$ (implication) de deux formules F_1 et F_2 à l'aide d'un connecteur binaire.

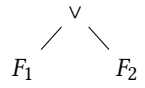
D'autres connecteurs, comme $\iff$, peuvent être encodés par des combinaisons des précédents.

On peut faire trois remarques à propos de cette description. D'une part, elle définit les formules logiques en décrivant comment on peut les *construire*. Une telle définition sous-entend deux choses :

- toute application de ces règles de construction produit une formule valide, et
- toute formule valide peut être obtenue à l'aide d'une telle construction.

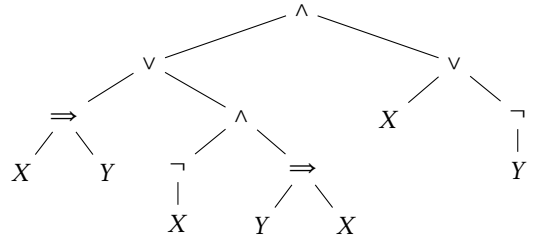
D'autre part, cette définition est récursive : les connecteurs $\neg$, $\wedge$, $\vee$ et $\Rightarrow$ construisent une formule propositionnelle *à partir de formules déjà construites*. Enfin, cette définition ressemble à une version enrichie de la définition d'un arbre binaire ! En conséquence de tout ceci, on pourra encore utiliser sur les formules les techniques de définition d'une fonction récursive par cas sur la forme d'une formule, et de raisonnement par récurrence structurelle.

Structure des formules, graphiquement. Pour faire ressortir plus clairement la structure arborescente d'une formule, on peut la représenter graphiquement. Dans le cas d'une formule composée, on place en haut le connecteur principal, et en-dessous de lui les sous-formules, reliées au connecteur, à la manière d'un arbre binaire.



Sur une formule plus riche, la connection se fait au niveau des connecteurs principaux des sous-formules, et certains nœuds ont un seul fils (car la négation est une opération *unaire*).

$$((X \implies Y) \vee (\neg X \wedge (Y \implies X))) \wedge (X \vee \neg Y)$$



Cette représentation graphique ressemble à nouveau à un graphe, avec une structure (un peu) plus variée qu'un arbre binaire. On y retrouve cependant encore la différence qu'on avait déjà évoquée entre graphes et arbres : le placement des nœuds est important. On le voit en particulier avec le connecteur d'implication : les deux formules suivantes n'ont pas la même signification !

Définition récursive de la taille d'une formule. La *taille* $|F|$ d'une formule F est, informellement, le nombre de symboles logiques qu'elle contient (variables propositionnelles incluses). On peut en donner une définition rigoureuse en fournissant une équation décrivant $|F|$ pour chaque forme possible de F. Comme lors de la définition de fonctions sur des listes chaînées ou des arbres binaires, nous avons des cas de base, pour les formules les plus simples, dans lesquels l'équation donne directement une valeur explicite,

$$\begin{aligned} |\top| &= 1 \\ |\bot| &= 1 \\ |X| &= 1 \end{aligned}$$

mais aussi des cas récursifs, pour les formules construites par combinaison de formules plus petites, dans lesquels l'équation fait intervenir la taille de ces sous-formules.

$$\begin{aligned} |\neg F| &= 1 + |F| \\ |F_1 \wedge F_2| &= 1 + |F_1| + |F_2| \\ |F_1 \vee F_2| &= 1 + |F_1| + |F_2| \\ |F_1 \implies F_2| &= 1 + |F_1| + |F_2| \end{aligned}$$

À noter : l'écriture d'une formule utilise des parenthèses pour éviter les ambiguïtés. Ces parenthèses ne font pas partie de la structure elle-même. On ne les a pas représentées dans l'arbre ci-dessus, et on ne les compte pas non plus dans la taille d'une formule. Par exemple :

$$\begin{aligned} |(\neg X \implies Y) \wedge \neg Y| &= 1 + |\neg X \implies Y| + |\neg Y| \\ &= 1 + (1 + |\neg X| + |Y|) + (1 + |Y|) \\ &= 1 + (1 + (1 + |X|) + 1) + (1 + 1) \\ &= 1 + (1 + (1 + 1) + 1) + 2 \\ &= 1 + 4 + 2 \\ &= 7 \end{aligned}$$

Variables d'une formule. Sur le même modèle, on peut donner une caractérisation récursive de l'ensemble $v(F)$ des variables présentes dans une formule F. En voici une dans laquelle on a regroupé les trois cas $F_1 \wedge F_2$, $F_1 \vee F_2$ et $F_1 \Longrightarrow F_2$ sous la forme commune $F_1 \circ F_2$, où le symbole $\circ$ représente un connecteur binaire quelconque.

$$\begin{array}{rcll} v(\top) & = & \emptyset & \\ v(\bot) & = & \emptyset & \\ v(X) & = & \{X\} & \\ v(\neg F) & = & v(F) & \\ v(F_1 \circ F_2) & = & v(F_1) \cup v(F_2) & \circ \in \{\wedge, \vee, \Longrightarrow\} \end{array}$$

Exemple :

$$\begin{array}{rcl} v((\neg X \Longrightarrow Y) \wedge \neg Y) & = & v(\neg X \Longrightarrow Y) \cup v(\neg Y) \\ & = & (v(\neg X) \cup v(Y)) \cup v(Y) \\ & = & (v(X) \cup v(Y)) \cup v(Y) \\ & = & \{X, Y\} \cup \{Y\} \\ & = & \{X, Y\} \end{array}$$

11.2 Représentation des formules en caml : types algébriques

Caml propose un mécanisme natif pour définir des structures de données construites ainsi récursivement : les types algébriques (que nous avons déjà utilisés pour les arbres binaires). On fournit pour cela un ensemble de *constructeurs*, correspondant chacun à l'une des formes possibles que peut prendre l'objet à construire. On peut noter par exemple :

- `True` pour la formule $\top$,
- `False` pour la formule $\bot$,
- `Not(f)` pour la négation de la formule f,
- `And(f1, f2)` pour la conjonction des formules f_1 et f_2,
- `Or(f1, f2)` pour la disjonction des formules f_1 et f_2,
- `Imp(f1, f2)` pour une implication de f_1 vers f_2.

Pour pouvoir différencier les variables, on donnera à chacune un numéro. Cela revient à considérer un ensemble $\{X_0, X_1, X_2, \ldots\}$ (infini, dénombrable) de variables propostionnelles. Il suffit alors de noter :

- `Var(k)` pour la variable X_k.

Avec une telle convention, la formule $\neg X_0 \vee (\neg X_1 \Longrightarrow X_0)$ sera écrite

```
Or(Not(Var(0)), Imp(Not(Var(1)), Var(0)))
```

ou encore de la manière suivante, où on a seulement ajouté un peu d'indentation pour rendre mieux visible la structure.

```
Or(Not(Var(0)),
   Imp(Not(Var(1)),
       Var(0)))
```

Définition d'un type algébrique. Pour introduire une telle structure en caml, on énumère les *constructeurs* que l'on souhaite définir. En l'occurrence : `True`, `False`, `Var`, `Neg`, `And`, `Or`, `Imp`. On précise également pour chacun les différents éléments auxquels il s'applique : rien pour `True` et `False`, un entier pour `Var`, une formule pour `Neg`, deux formules pour `And`, `Or`, `Imp`.

```
type fmla =            (* nom que l'on donne au type des formules *)
  | True               (* constructeur sans paramètre *)
  | False
  | Var of int         (* constructeur avec un paramètre entier *)
  | Not of fmla        (* constructeur s'appliquant à une formule *)
  | And of fmla * fmla (* constructeur s'appliquant à deux formules *)
  | Or  of fmla * fmla
  | Imp of fmla * fmla
```

L'aspect récursif de cette construction se manifeste par la mention du type `fmla` que l'on est en train de définir dans les paramètres de certains des constructeurs.

Définition d'une fonction par filtrage. On définit une fonction manipulant une telle structure à l'aide d'une opération de *filtrage*, notée **`match`**, qui permet d'énumérer les différentes formes possibles des formules, et de fournir un code adapté à chacune.

```
let rec size f = match f with
  | True        -> 1
  | False       -> 1
  | Var(_)      -> 1
  | Not(f)      -> 1 + size f
  | And(f1, f2) -> 1 + size f1 + size f2
  | Or (f1, f2) -> 1 + size f1 + size f2
  | Imp(f1, f2) -> 1 + size f1 + size f2
```

Factorisation de la définition, et simplifications de l'écriture. Dans les équations mathématique définissant la fonction v, on a jugé commode de factoriser les trois cas $F_1 \wedge F_2$, $F_1 \vee F_2$ et $F_1 \implies F_2$ en un unique cas $F_1 \circ F_2$, où $\circ$ peut représenter l'un ou l'autre des trois constructeurs. En effet, ces trois formes ont une structure similaire, et seront probablement régulièrement traitées de la même manière. On peut intégrer cette factorisation à notre définition du type algébrique. Pour cela, on remplace les trois constructions `And(f1, f2)`, `Or(f1, f2)` et `Imp(f1, f2)` par une unique construction `Bin(op, f1, f2)` regroupant les connecteurs binaires. Dans cette nouvelle construction, `f1` et `f2` sont toujours des formules, et `op` est une constante `And`, `Or` ou `Imp` désignant le connecteur.

```
type binop = And | Or | Imp
type fmla =
  | True
  | False
  | Var of int
  | Not of fmla
  | Bin of binop * fmla * fmla
```

Cette définition remplace la précédente (avec de nouvelles significations pour les symboles `And`, `Or` et `Imp`). On représente maintenant comme suit la formule $\neg X_0 \vee (\neg X_1 \implies X_0)$.

```
Bin(Or, Not(Var(0)), Bin(Imp, Not(Var(1)), Var(0)))
```

L'écriture d'une formule est légèrement plus lourd, mais le calcul de la taille est en revanche simplifié, puisqu'il regroupe les trois dernières lignes de la version précédente par

```
  | Bin(_, f1, f2) -> 1 + size f1 + size f2
```

Simplification d'écriture : les parenthèses ne sont pas nécessaires pour un constructeur prenant un unique paramètre, lorsque celui-ci est atomique. C'est le cas notamment pour `Var` lorsqu'on lui fournit une constante entière

```
Bin(Or, Not(Var 0), Bin(Imp, Not(Var 1), Var 0))
```

mais aussi pour `Not` ou `Var` utilisés dans un filtrage avec un identifiant ou un joker.

```
  | Not f -> 1 + size
  | Var _ -> 1
```

Simplification d'écriture : on peut grouper plusieurs cas de filtrage au même comportement.

```
  | True | False -> 1
```

En combinant tous ces éléments, voici la version finale de notre fonction de calcul de taille.

```
let rec size f = match f with
  | True | False | Var _  -> 1
  | Not f                -> 1 + size f
  | Bin (_, f1, f2)      -> 1 + size f1 + size f2
```

Autre exemple : fonction `varmax` donnant le numéro de la plus grande variable d'une formule.

```
let rec varmax f = match f with
  | True | False    -> -1    (* réponse -1 si aucune variable présente *)
  | Var i           -> i     (* i est supposé >= 0 *)
  | Not f           -> varmax f
  | Bin (_, f1, f2) -> max (varmax f1) (varmax f2)
```

11.3 Représentation des formules en java : abstraction et héritage

Dans un langage comme java, on peut facilement définir une structure de données récursive (on l'a vu avec les listes et les arbres aux chapitres précédents), mais on n'a pas de mécanisme dédié pour la définition et la manipulation d'un type algébrique [11]. Pour représenter une telle structure, une technique « orientée objet » traditionnelle consiste à : définir une interface ou une classe abstraite principale pour la structure de données, et une classe concrète implémentant cette interface pour chaque forme que peut prendre la structure.

Dans notre cas, on peut ainsi définir une interface `Fmla` couvrant l'ensemble des formules, dans laquelle on déclare les différentes méthodes que l'on voudra avoir.

```
interface Fmla {
  int size();
  int varmax();
}
```

Il reste ensuite à définir une classe concrète implémentant `Fmla` pour chaque connecteur logique ou forme atomique. Chacune de ces classes concrètes doit contenir :

— des attributs (immuables) pour chaque élément constituant une formule du type donné,

— un constructeur (sauf si le constructeur par défaut suffit),

— l'implémentation des méthodes `size` et `varmax` correspondant à la forme considérée.

Pour les formes atomiques $\top$ et $\bot$, il n'y a pas besoin d'attribut ni de constructeur spécifique, on indique simplement ce que doivent être la taille et la plus grande variable.

```
class True implements Fmla {
  public int size()   { return 1; }
  public int varmax() { return -1; }
}
```

Pour une variable, on introduit un attribut entier immuable désignant le numéro de la variable et un constructeur initialisant cet attribut (en plus des concrétisations des méthodes).

```
class Var implements Fmla {
  final int i;
  public Var(int i) { this.i = i; }
  public int size()   { return 1; }
  public int varmax() { return i; }
}
```

Pour une formule de la forme $\neg F$, on introduit un attribut pour la sous-formule F. Les définitions des méthodes `size` et `varmax` vont appeler les méthodes de la sous-formule.

```
class Not implements Fmla {
  final Fmla f;
  public Not(Fmla f) { this.f = f; }
  public int size()   { return 1 + f.size(); }
  public int varmax() { return f.varmax(); }
}
```

Pour une formule $F_1 \circ F_2$ avec un connecteur binaire $\circ$, on a deux attributs pour les sous-formules F_1 et F_2, et un troisième pour le connecteur lui-même. On définit l'ensemble des connecteurs utilisés à l'aide d'une énumération.

```
enum Binop { AND, OR, IMP }
class Bin implements Fmla {
  final Binop op;
  final Fmla f1, f2;
  public Bin(Binop op, Fmla f1, Fmla f2) {
    this.op = op; this.f1 = f1; this.f2 = f2;
  }
  public int size()   { return 1 + f1.size() + f2.size(); }
  public int varmax() { return Math.max(f1.varmax(), f2.varmax()); }     }
```

11. Depuis 2020, une grande part des nouveautés introduites par les mises à jour successives du langage java s'inscrivent dans un effort d'améliorer ceci, en intégrant à java certains des éléments que nous avons vus en caml. La version 21 de java (sept. 2023) en a stabilisé une bonne partie mais tout n'est pas encore terminé (certains aspects sont encore en cours de test, et d'autres restent à implémenter). Pour suivre l'évoluation de la situation, surveiller dans la documentation les mots-clés *sealed class*, *record class*, et *pattern matching*. `https://openjdk.org/jeps/0`
`https://docs.oracle.com/en/java/javase/21/language/java-language-changes.html`

Avec cette mise en place, chaque nouvelle fonction manipulant les formules se traduit par :

— une déclaration dans l'interface `Fmla`,

— une définition adaptée dans chaque classe concrète.

Ainsi, chaque classe concrète contient des définitions traduisant toutes les équations relatives à cette forme. Par exemple, `Bin.size` et `Bin.varmax` traduisent les deux équations

$$\begin{array}{rcl} |F_1 \circ F_2| & = & 1 + |F_1| + |F_2| \\ v(F_1 \circ F_2) & = & \max(v(F_1), v(F_2)) \end{array}$$

Avec cette famille de classes, note formule exemple $\neg X_0 \vee (\neg X_1 \implies X_0)$ peut être définie par

```
Fmla f = new Bin(Binop.OR,
                 new Not(new Var(0)),
                 new Bin(Binop.IMP, new Not(new Var(1)), new Var(0)))
```

11.4 Évaluation et transformation de formules

Sémantique booléenne. *Évaluer* une formule propositionnelle, c'est lui associer une valeur de vérité booléenne (vrai ou faux). Une telle évaluation se fait en fonction d'une *interprétation* des variables propositionnelles, c'est-à-dire d'une fonction associant une valeur de vérité à chaque variable. À des fins de programmation, et avec nos variables numérotées, on peut résumer une interprétation des variables par un tableau de booléens. D'où la fonction `eval` en caml, qui prend en paramètres une formule `f` (type `fmla`) et une interprétation `v` (type `bool array`), et qui renvoie une valeur de vérité (type `bool`).

```
let rec eval f v = match f with
  | True -> true
  | False -> false
  | Var i -> v.(i)    (* accès à l'élément d'indice i du tableau v *)
  | Not f -> not (eval f v)
  | Bin (And, f1, f2) -> eval f1 v && eval f2 v
  | Bin (Or,  f1, f2) -> eval f1 v || eval f2 v
  | Bin (Imp, f1, f2) -> not (eval f1 v) || eval f2 v
```

En java, cette nouvelle fonction est déclarée dans l'interface, et son code est réparti entre les différentes classes.

— Dans l'interface `Fmla`, on déclare :

```
boolean eval(boolean[] v);
```

— Dans la classe `True`, la méthode est constante :

```
public boolean eval(boolean[] v) { return true; }
```

— Dans la classe `False`, de même :

```
public boolean eval(boolean[] v) { return false; }
```

— Dans la classe `Var`, on consulte l'interprétation pour la variable numéro `this.i` :

```
public boolean eval(boolean[] v) { return v[i]; }
```

— Dans la classe `Not`, on évalue la sous-formule :

```
public boolean eval(boolean[] v) { return !f.eval(v); }
```

— Dans la classe `Bin`, on évalue les sous-formules et on consulte le connecteur.

```
public boolean eval(boolean[] v) {
  switch (op) {
    case AND: return f1.eval(v) && f2.eval(v);
    case OR:  return f1.eval(v) || f2.eval(v);
    case IMP: return !f1.eval(v) || f2.eval(v);
    default: throw new IllegalArgumentException();
  }
}
```

Il y a précisément deux exceptions. Lesquelles ?

Simplifications de formules. Dans chaque formule où apparaît l'une des formules atomiques $\top$ ou $\bot$, on peut faire des simplifications. Voici les équations que l'on peut appliquer :

$$\begin{array}{lll} F \wedge \bot \equiv \bot \wedge F \equiv \bot & F \wedge \top \equiv \top \wedge F \equiv F & \neg\top \equiv \bot \\ F \vee \top \equiv \top \vee F \equiv \top & F \vee \bot \equiv \bot \vee F \equiv F & \neg\bot \equiv \top \\ F \Longrightarrow \top \equiv \top & \top \Longrightarrow F \equiv F & \neg(\neg F) \equiv F \\ \bot \Longrightarrow F \equiv \top & F \Longrightarrow \bot \equiv \neg F & \end{array}$$

Exemple :

$$\begin{array}{ll} & (\bot \Longrightarrow Z) \wedge \neg((Y \vee \top) \wedge (X \Longrightarrow \bot)) \\ \equiv & \top \wedge \neg((Y \vee \top) \wedge (X \Longrightarrow \bot)) \\ \equiv & \top \wedge \neg(\top \wedge (X \Longrightarrow \bot)) \\ \equiv & \top \wedge \neg(\top \wedge \neg X) \\ \equiv & \top \wedge \neg(\neg X) \\ \equiv & \top \wedge X \\ \equiv & X \end{array}$$

Pour simplifier au maximum une formule à l'aide de ces équations, on peut combiner deux fonctions : une fonction `try_simp` qui essaie d'appliquer l'une des équations à la racine de la formule, et une fonction `simp` qui itère cette fonction `try_simp` sur tous les nœuds de la formule. En caml :

```
let try_simp f = match f with
  | Not True   -> False
  | Not False  -> True
  | Not(Not f) -> f
  | Bin(And, False, _) | Bin(And, _, False) -> False
  | Bin(And, True,  f) | Bin(And, f, True ) -> f
  | Bin(Or,  True,  _) | Bin(Or,  _, True ) -> True
  | Bin(Or,  False, f) | Bin(Or,  f, False) -> f
  | Bin(Imp, False, _) | Bin(Imp, _, True ) -> True
  | Bin(Imp, True,  f) | Bin(Imp, Not f, False) -> f
  | Bin(Imp, f, False) -> Not f
  | _ -> f

let rec simp f = match f with
  | True | False | Var _ -> f
  | Not f                -> try_simp (Not (simp f))
  | Bin(op, f1, f2)      -> try_simp (Bin(op, simp f1, simp f2))
```

Validité des simplifications. Une formule F' obtenue par simplification est *équivalente* à la formule F d'origine, ce qu'on note $F' \equiv F$ et qu'on définit par :

Pour toute interprétation v, on a $\mathsf{eval}(F', v) = \mathsf{eval}(F, v)$.

On peut le montrer en deux temps.

1. D'abord, on vérifie que chaque équation est bien correcte (il suffit de faire des tables de vérité pour chaque). Cela suffit à justifier que `try_simp` produit une formule équivalente à la formule prise en argument.

   Pour toute formule F, on a $\mathsf{try\_simp}(F) \equiv F$.

2. Ensuite, on montre que les simplifications récursives réalisées par `simp` sont encore valides.

   Pour toute formule F, on a $\mathsf{simp}(F) \equiv F$.

   On le démontre ci-dessous par *récurrence structurelle* sur F.

Raisonnement par récurrence structurelle sur les formules. Pour démontrer qu'une propriété $P(F)$ est vraie pour toute formule F, il suffit de vérifier que les règles de construction des formules ne permettent de construire que des formules vérifiant P, c'est-à-dire d'une part que la propriété P est vraie pour les formules atomiques, et que d'autre part toute

construction d'une nouvelle formule à l'aide d'un connecteur *préserve* la propriété P : si les sous-formules prises comme briques de base satisfont P, alors la nouvelle formule construite doit encore satisfaire P.

Ainsi, il suffit de vérifier que :

1. la propriété $P(\top)$ est vraie,
2. la propriété $P(\bot)$ est vraie,
3. la propriété $P(X)$ est vraie pour toute variable propositionnelle X,
4. la propriété $P(\neg F)$ est vraie pour toute formule F telle que $P(F)$ est vraie,
5. la propriété $P(F_1 \circ F_2)$ est vraie pour tout connecteur binaire $\circ$ et toutes formules F_1 et F_2 telles que $P(F_1)$ et $P(F_2)$ sont vraies,

pour s'assurer que la propriété P est vraie *pour toutes les formules*. Les trois premiers points sont des cas de base, et les deux derniers des cas récursifs. Dans les deux derniers points, les hypothèses $P(F)$, $P(F_1)$ et $P(F_2)$ que l'on suppose pour justifier $P(\neg F)$ ou $P(F_1 \circ F_2)$ sont les *hypothèses de récurrence*. Il s'agit du même principe de raisonnement par récurrence structurelle que nous avons déjà vu sur les listes et les arbres binaires.

Application à la simplification. On montre que pour toute formule F, on a $\mathsf{simp}(F) \equiv F$. On vérifie les cinq points du principe de récurrence.

1. Cas $\top$. Par définition $\mathsf{simp}(\mathsf{True}) = \mathsf{True}$, et on a donc bien $\mathsf{simp}(\mathsf{True}) \equiv \mathsf{True}$.
2. Cas $\bot$. De même, $\mathsf{simp}(\mathsf{False}) = \mathsf{False} \equiv \mathsf{False}$.
3. Cas X. De même, $\mathsf{simp}(\mathsf{Var}(i)) = \mathsf{Var}(i) \equiv \mathsf{Var}(i)$.
4. Cas $\neg F$, en supposant que $\mathsf{simp}(F) \equiv F$. *(on veut montrer simp*$(\neg F) \equiv \neg F$*)*
   On calcule :

$$\begin{array}{lcll} \mathsf{simp}(\mathsf{Not}(F)) & = & \mathsf{try\_simp}(\mathsf{Not}(\mathsf{simp}(F))) & \text{par déf. de simp} \\ & \equiv & \mathsf{Not}(\mathsf{simp}(F)) & \text{par propriété de try\_simp} \\ & \equiv & \mathsf{Not}(F) & \text{par hypothèse de récurrence} \end{array}$$

5. Cas $F_1 \circ F_2$, en supposant que $\mathsf{simp}(F_1) \equiv F_1$ et $\mathsf{simp}(F_2) \equiv F_2$.
   (on veut montrer simp$(F_1 \circ F_2) \equiv F_1 \circ F_2$*)*
   On calcule :

$$\begin{array}{lcll} & & \mathsf{simp}(\mathsf{Bin}(\circ, F_1, F_2)) & \\ & = & \mathsf{try\_simp}(\mathsf{Bin}(\circ, \mathsf{simp}(F_1), \mathsf{simp}(F_2))) & \text{par déf. de simp} \\ & \equiv & \mathsf{Bin}(\circ, \mathsf{simp}(F_1), \mathsf{simp}(F_2)) & \text{par propriété de try\_simp} \\ & \equiv & \mathsf{Bin}(\circ, F_1, \mathsf{simp}(F_2)) & \text{par hyp. de récurrence sur } F_1 \\ & \equiv & \mathsf{Bin}(\circ, F_1, F_2) & \text{par hyp. de récurrence sur } F_2 \end{array}$$

11.5 Approfondissement : algorithme de *backtracking*

Partant d'une formule F, on cherche une interprétation des variables propositionnelles de F qui rende la formule vraie. Une option pour cela est d'*explorer* l'ensemble des combinaisons de valeurs de vérité. Dans cette approche, on choisit une valeur de vérité arbitraire pour une première variable, puis pour une deuxième et ainsi de suite jusqu'à aboutir, soit à une solution (les valeurs choisies suffisent à rendre la formule vraie), soit à une impasse (les valeurs choisies suffisent à rendre la formule fausse). Lorsque l'on arrive à une impasse, on « revient sur nos pas » (« *backtrack* ») jusqu'au choix le plus récent, et on essaie à nouveau avec la valeur opposée.

Est-ce une exploration en largeur, ou en profondeur ?

Algorithme. L'algorithme d'exploration part d'une formule F et d'une interprétation v d'*une partie* des variables de F (on parle d'*interprétation partielle*) et cherche à compléter v d'une manière qui rende F vraie.

- Si v suffit à ce que F s'évalue à vrai, alors v est une solution, on peut s'arrêter.
- Si v suffit à ce que F s'évalue à faux, alors aucune manière de compléter v ne pourra donner une solution : on abandonne cette voie.
- Sinon, on choisit une variable X_i de F pas encore interprétée dans v, et récursivement :
  - on explore avec l'interprétation étendue $v, X_i \mapsto \text{vrai}$,
  - et en cas d'échec on explore avec l'interprétation opposée $v, X_i \mapsto \text{faux}$.

  Si ces deux tentatives échouent, on abandonne cette voie.

Pour évaluer la valeur d'une formule F avec une interprétation partielle v, on propose la technique suivante :

1. dans F, on remplace X par $\top$ si $v(X)$ = vrai et par $\bot$ si $v(X)$ = faux,
2. on simplifie la formule ainsi obtenue,
3. on a un résultat immédiat si la simplification donne $\top$ ou $\bot$.

Exemple. Considérons la formule

$$(X \Longrightarrow Y) \wedge (\neg X \vee (\neg Y \wedge \neg Z)) \wedge ((Y \Longrightarrow \neg Z) \vee X)$$

et partons de l'interprétation vide. On considère d'abord la variable X.

— On explore d'abord en fixant $v(X)$ = vrai. La formule simplifiée est

$$\begin{aligned}&(\top \Longrightarrow Y) \wedge (\neg\top \vee (\neg Y \wedge \neg Z)) \wedge ((Y \Longrightarrow \neg Z) \vee \top)\\ \equiv\ & Y \wedge (\bot \vee (\neg Y \wedge \neg Z)) \wedge \top\\ \equiv\ & Y \wedge (\neg Y \wedge \neg Z)\end{aligned}$$

On considère ensuite la variable Y.

— Avec $v(Y)$ = vrai, on échoue avec la formule $\top \wedge (\neg\top \wedge \neg Z) \equiv \bot$.

— Avec $v(Y)$ = faux, on échoue avec la formule $\bot \wedge (\neg\bot \wedge \neg Z) \equiv \bot$.

Aucune valeur de Y ne permet d'aboutir. Échec sur cette voie, on revient à X.

— On explore maintenant en fixant $v(X)$ = faux. La formule simplifiée est

$$\begin{aligned}&(\bot \Longrightarrow Y) \wedge (\neg\bot \vee (\neg Y \wedge \neg Z)) \wedge ((Y \Longrightarrow \neg Z) \vee \bot)\\ \equiv\ & \top \wedge (\top \vee (\neg Y \wedge \neg Z)) \wedge (Y \Longrightarrow \neg Z)\\ \equiv\ & Y \Longrightarrow \neg Z\end{aligned}$$

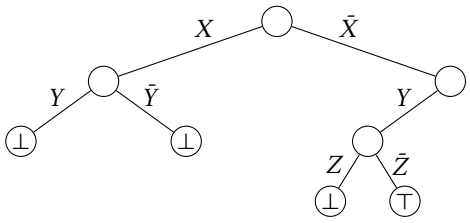
On considère ensuite la variable Y.

— Avec $v(Y)$ = vrai, on obtient la formule $\top \Longrightarrow \neg Z \equiv \neg Z$. On considère alors Z.

— Avec $v(Z)$ = vrai, on échoue avec la formule $\neg\top \equiv \bot$.

— Avec $v(Z)$ = faux, on obtient la formule $\neg\bot \equiv \top$: on a trouvé une solution !

Finalement, on a déterminé que l'interprétation $\{X \mapsto$ faux, $Y \mapsto$ vrai, $Z \mapsto$ faux$\}$ satisfait la formule. L'ensemble de cette exploration peut être résumé par l'arbre des différents choix testés.



En caml. On se donne une fonction auxiliaire `subst` pour remplacer une variable X_i par une formule F_s dans une formule F :

```
let rec subst i fs f = match f with
  | Var j when i = j -> fs
  | Var _            -> f
  | True | False     -> f
  | Not f            -> Not (subst i fs f)
  | Bin(op, f1, f2)  -> Bin(op, subst i fs f1, subst i fs f2)
```

On peut ensuite définir la fonction d'exploration récursive, qui prend en paramètres une formule `f` et une interprétation partielle `v`, et qui renvoie soit `Some v'` si on a trouvé une extension `v'` de `v` satisfaisant `f`, soit `None` s'il n'y a pas de solution. Chaque appel récursif est fait après remplacement de la variable par `True` ou `False`.

Sur le problème du club écossais, on trouve l'interprétation

$E \mapsto$ vrai
$C \mapsto$ faux
$K \mapsto$ faux
$M \mapsto$ faux
$S \mapsto$ vrai

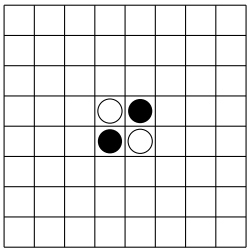
après avoir exploré 9 interprétations partielles.

```
let rec backtrack f v = match simplify f with
  | True  -> Some v
  | False -> None
  | f     -> let x = varmax f in
             let r = backtrack (subst x True f) ((x, true) :: v) in
             if Option.is_some r then r
             else backtrack (subst x False f) ((x, false) :: v)
let sat f = backtrack f []
```

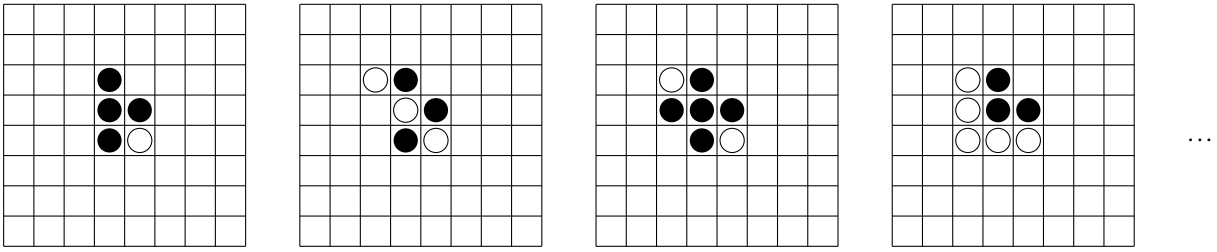
12 Se faire battre à tous les coups

12.1 Problème : algorithme gagnant pour Othello

Le jeu Othello, encore appelé Reversi, se joue sur un échiquier 8×8 avec des pions blancs d'un côté et noirs de l'autre. Initialement, quatre pions sont placés comme ceci au centre de l'échiquier :



Chaque joueur a une couleur et les noirs commencent. Un coup consiste à poser un pion de sa couleur sur une case libre de l'échiquier en réalisant au moins une « prise », c'est-à-dire une rangée de pions adverses (horizontale, verticale ou diagonale) encadrée par deux pions de sa couleur. Tous les pions de l'échiquier qui se retrouvent ainsi encadrés sont retournés et deviennent donc des pions du joueur qui vient de jouer. Voici un début de partie possible :



…

On note comment les blancs ont réalisé deux prises avec leur deuxième coup. Lorsqu'un joueur ne peut jouer, il passe son tour. La partie se termine lorsqu'aucun des deux joueurs ne peut jouer. Le joueur disposant alors du plus grand nombre de pions de sa couleur sur l'échiquier gagne la partie.

Dans ce chapitre, on cherche à définir un algorithme capable de nous battre à tous les coups au jeu d'Othello.

12.2 Graphe implicite et exploration

L'état courant d'une partie est défini par deux informations : la configuration des pions sur l'échiquier, et l'identification du joueur devant jouer le prochain coup. On peut alors voir le jeu comme un graphe dans lequel :

- chaque état possible de la partie correspond à un sommet,
- on a une arête de l'état s vers l'état t lorsque l'un des coups autorisés dans l'état s mène à l'état t.

Alors, chaque partie imaginable correspond à un chemin dans ce graphe, allant de l'état initial du jeu jusqu'à un état où la partie est déclarée terminée.

Contrairement à ce qu'on a fait jusque là, on ne construira pas explicitement un tel graphe en mémoire (sa taille serait prohibitive). On se contente à la place de fournir une structure de données décrivant un état du jeu, et une fonction calculant la liste des coups possibles. Voici la définition java de l'interface d'une telle structure.

```
public interface GameState {
  // caractéristiques d'un état
  char player();     // joueur dont c'est le tour
  boolean isFinal(); // partie terminée ?
  char outcome();    // si partie terminée, vainqueur ? (ou partie nulle ?)

  // les coups possibles, représentés par la liste des états où mènent ces coups
  List<GameState> moves();
}
```

12.3 Stratégies gagnantes

Pour jouer intelligemment à un tel jeu, on explore tout ou partie du graphe des coups possibles, pour déterminer les meilleurs coups. On cherche en particulier les *configurations gagnantes*, c'est-à-dire les états du jeu ayant l'une de ces trois propriétés :

— la partie est terminée, et le joueur gagne, ou

— c'est le tour du joueur, et l'un des coups mène à une configuration gagnante, ou

— c'est le tour de l'adversaire, et tous les coups mènent à une configuration gagnante.

Autrement dit, une configuration est gagnante pour le joueur si :

— tout coup de l'adversaire mène à une configuration telle que,

— il existe un coup du joueur menant à une configuration telle que,

— tout coup de l'adversaire mène à une configuration telle que,

— il existe un coup du joueur menant à une configuration telle que,

— ...

— la partie est terminée et le joueur gagne.

(on a pris ici comme point de départ une situation où c'est au tour de l'adversaire de jouer, il suffit d'en oublier la première ligne pour décrire la situation où c'est le tour du joueur). L'alternance entre les quantificateurs universels et existentiels traduit deux faits :

Le fait qu'il existe un chemin d'une configuration donnée à un état gagnant ne suffit pas à garantir la victoire, car on ne maîtrise pas les coups de l'adversaire !

— le joueur peut choisir son coup, il lui suffit donc qu'au moins un des coups possibles soit bon (à charge pour lui de déterminer lequel !),

— le joueur ne peut pas choisir les coups de son adversaires, il ne gagne donc à coup sûr que si aucun des coups que peut choisir l'adversaire ne prive le joueur de la victoire.

Une fonction très simple permet alors, partant d'un état du jeu `s` quelconque, d'explorer tous les états futurs possibles, pour déterminer si cet état `s` est gagnant pour le joueur `p`.

```
public static boolean winning(GameState s, char p) {
  if (s.isFinal()) {
    // Si la partie est terminée, le joueur gagne-t-il ?
    return (s.outcome() == p);
  }
  if (s.player() == p) {
    // Si c'est le tour du joueur, existe-t-il un coup gagnant ?
    for (GameState m : s.moves()) {
      if (winning(m, p)) return true;
    }
    return false;
  } else {
    // Si c'est le tour de l'adversaire, tous les coups sont-ils gagnants ?
    for (GameState m : s.moves()) {
      if (!winning(m, p)) return false;
    }
    return true;
  }
}
```

Problème : cette fonction n'est pas utilisable en pratique, à part pour les jeux les plus simples. En supposant qu'à chaque tour, le joueur ou l'adversaire a ne serait-ce que deux coups différents possibles, le nombre d'états à explorer est exponentiel en le nombre de coups joués (environ 60 pour Othello !) On peut bien sûr en économiser un peu en mémorisant les états déjà analysés, de la même manière qu'on marque les sommets d'un graphe rencontrés lors d'une exploration, mais cela ne sera jamais assez pour briser ce caractère exponentiel (le nombre d'états différents possibles à Othello est bien exponentiel en la taille du plateau).

12.4 Algorithme Min-Max : exploration à profondeur bornée

Pour un jeu comme Othello, il est impossible d'explorer jusqu'au bout l'ensemble des séquences de coups possibles jusqu'à la fin de la partie. On peut cependant déjà obtenir de très bon résultats en explorant seulement *quelques coups à l'avance*. L'exploration d'une séquence peut alors s'arrêter de deux manières différentes :

— si la séquence analysée va jusqu'à une partie terminée, on utilise comme précédemment la fonction `outcome` pour connaître l'éventuel vainqueur,

— si la séquence analysée s'arrête car on a déjà exploré le nombre de coups fixé, il nous faut *évaluer* à quel point l'état atteint est intéressant (ou au contraire catastrophique !) pour le joueur.

L'évaluation est *heuristique* : il ne s'agit pas d'un jugement absolu déterminant si l'état est gagnant ou non, mais seulement d'une indication que l'état *semble favorable*, ou au contraire défavorable, au joueur. Traditionnellement, l'évaluation prend la forme d'un score entier dans un intervalle fixé, par exemple l'intervalle $[-10000, 10000]$, où les nombres positifs sont favorables au joueur, les nombres négatifs défavorables, et où l'amplitude indique à quel point un état semble favorable ou défavorable.

À Othello, on pourra prendre les critères suivants pour évaluer un état :

— les jetons de la couleur du joueur dans les coins du plateau sont *très intéressants* (+9 par jeton),

— les jetons de la couleur du joueur sur les bords sont *intéressants* (+3 par jeton),

— les jetons de la couleur du joueur ailleurs sur le plateau sont *modérément intéressants* (+1 par jeton),

et pour chaque jeton adverse : les mêmes points en négatif. Si c'est le tour du joueur, on peut également compter quelques points supplémentaires pour chaque coup légal (avoir de nombreuses possibilités est généralement favorable). En voici une réalisation simple. Le plateau (`board`) est représenté par une chaîne de 64 caractères, avec : `'B'` pour un jeton noir, `'W'` pour un jeton blanc, `'.'` pour une case libre. Le joueur (`player`) est représenté par le caractère `'B'` ou `'W'`. La fonction `eval` calcule notre heuristique du point de vue d'un joueur `p`, et `other(p)` renvoie le joueur autre que `p`.

```
public class OthelloState implements GameState {
  final String board;
  final char player;

  public int eval(char p) {
    int value = (player == p)?moves().size():0;
    int[] weight = {
      9, 3, 3, 3, 3, 3, 3, 9,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      3, 1, 1, 1, 1, 1, 1, 3,
      9, 3, 3, 3, 3, 3, 3, 9
    };
    for (int i=0; i<64; i++) {
      if (board.charAt(i) == p) value += weight[i];
      else if (board.charAt(i) == other(p)) value -= weight[i];
    }
    return value;
  }

  public static char other(char p) {
    switch (p) {
      case 'W': return 'B';
      case 'B': return 'W';
      default: throw new IllegalArgumentException();
    }
  }
}
```

L'évaluation d'un état *avec une profondeur d*, c'est-à-dire en tenant compte de toutes les séquences possibles de d prochains coups, suit alors les principes suivants.

— Si l'état est celui d'une partie terminée, on lui attribue l'un des scores +10000 (partie gagnée), -10000 (partie perdue) ou 0 (partie nulle).

— À profondeur 0, on attribue à l'état le score heuristique donné par la fonction `eval`.

— À profondeur $d > 0$, on calcule les scores à profondeur $d - 1$ de tous les états suivants possibles, puis :

— si c'était le tour du joueur, on sélectionne le score le plus favorable (le joueur choisit son coup, on suppose qu'il le fait au mieux de ses capacités),

— si c'était le tour de l'adversaire, on sélectionne le score le moins favorable au joueur (l'adversaire, s'il joue bien, optimisera sa propre situation au détriment de celle du joueur).

Cet algorithme est appelé *Min-Max*, puisqu'il évalue un score en alternant des minimisations (des scores à l'issue des tours de l'adversaire) et des maximisations (des scores à l'issue des tours du joueur).

```
static int minmax(GameState state, int d, char p) {
  // Fin de l'exploration
  if (state.isFinal()) {
    char o = state.outcome();
    if (o == p) return 10000;
    else if (o == '.') return 0;
    else return -10000;
  }
  if (d == 0) {
    return state.eval(p);
  }
  // Examen des coups possibles
  int r = (state.player() == p)?-10000:10000;
  for (GameState m : state.moves()) {
    int value = minmax(m, d-1, p);
    if (state.player() == p) { // maximisation
      if (value > r) { r = value; }
    } else { // minisation
      if (value < r) { r = value; }
    }
  }
  return r;
}
```

Pour sélectionner un coup, il suffit alors d'évaluer tous les coups possibles à une profondeur fixée (par exemple : 4), puis prendre le plus favorable.

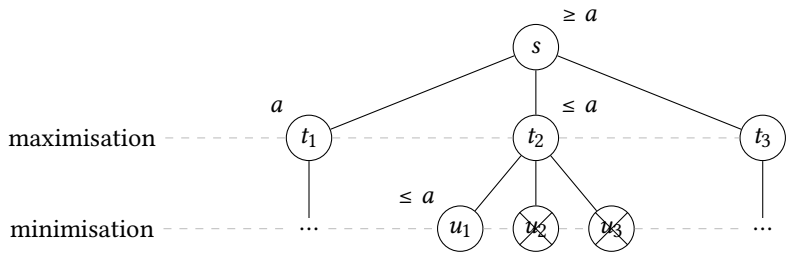
```
static GameState play(GameState state) {
  int bestScore = -10001;
  GameState bestMove = null;
  for (GameState m : state.moves()) {
    int score = minmax(m, DEPTH, state.player());
    if (score > bestScore) {
      bestScore = score;
      bestMove = m;
    }
  }
  assert (bestMove != null);
  return bestMove;
}
```

12.5 Algorithme Alpha/Beta : exploration optimisée

L'*élagage* α/β est une optimisation de l'algorithme Min-Max qui consiste à éviter l'exploration de certaines branches, dès lors l'on sait que ces branches ne peuvent pas modifier le score minimum (ou maximum) en cours de calcul pour une configuration donnée.

Le principe est le suivant. Supposons que l'on considère une position s du joueur, que l'on cherche à évaluer à profondeur d. On cherche donc la valeur maximale, parmi les évaluations à profondeur $d-1$ des configurations $t_1, t_2, ..., t_n$ correspondant aux différents coups possibles du joueur. Supposons que l'on a déterminé une certaine valeur a pour t_1, et que l'on regarde maintenant la configuration t_2. Rappel : dans cette configuration t_2, c'est à l'opposant de

jouer, on cherche donc la valeur minimale parmi les coups possibles u_1, ..., u_k de l'opposant.



Dès que l'on a pu déterminer une valeur r pour l'un des coups u_i de l'opposant, on sait que le minimum sera inférieur ou égal à r, et donc que la valeur calculée pour t_2 ne dépassera pas r. Autrement dit, si on trouve pour l'un des u_i une valeur inférieur ou égale à a, alors on sait que l'évaluation de t_2 sera inférieur ou égale à l'évaluation a obtenue pour t_1, et donc que le coup t_2 ne sera pas celui donnant la valeur maximale cherchée dans l'évaluation de s (et donc que l'on peut arrêter là l'évaluation de t_2).

Voici donc un nouveau code, qui est une version étendue de la fonction `minmax` prenant deux paramètres supplémentaires `a` et `b`, qui indiquent les seuils auxquels interrompre respectivement une minimisation ou une maximisation. Remarque : lorsque l'on met à jour l'un des seuils `a` ou `b`, c'est dans l'optique d'élaguer des « neveux », c'est-à-dire les successeurs d'un des coups alternatifs au coup que l'on vient d'analyser.

```
static int alphabeta(GameState state, int d, int a, int b, char p) {
  // Fin de l'exploration
  if (state.isFinal()) {
    if (state.outcome() == p) return 10000;
    if (state.outcome() == '.') return 0;
    return -10000;
  }
  if (d == 0) return state.eval(p);
  // Examen des coups possibles
  int r = (state.player() == p)?-10000:10000;
  for (GameState m : state.moves()) {
    int value = alphabeta(m, d-1, a, b, p);
    if (state.player() == p) { // maximisation
      if (value > r) r = value;
      if (r >= b)    return r; // seuil maximisation dépassé
      if (r > a)     a = r;    // mise à jour seuil minisation
    } else { // mininisation
      if (value < r) r = value;
      if (r <= a)    return r; // seuil minimisation dépassé
      if (r < b)     b = r;    // mise à jour seuil maximisation
    }
  }
  return r;
}

// initialisation de alpha et beta à des valeurs extrèmes
static int alphabeta(GameState state, int d, char p) {
  return alphabeta(state, d, -10000, 10000, p);
}

// fonction play similaire à la précédente
```

12.6 Mémoïsation : conserver en mémoire les évaluations déjà faites

Notre algorithme d'exploration, tel qu'il est écrit, n'empêche pas d'évaluer plusieurs fois une même configuration (et donc d'explorer plusieurs fois les coups qui suivent une configuration déjà vue). La profondeur d'exploration étant limitée par le paramètre d, cela ne peut pas d'engendrer de boucle infinie, mais dans certaines situations cela peut rendre les explorations plus longues que nécessaire.

Les algorithmes d'exploration de graphes du chapitre précédent, à l'inverse, marquaient l'ensemble des sommets déjà explorés pour ne pas les traiter à nouveau. Ce marquage utilisait un tableau de booléens, et chaque sommet était associé à une case par son numéro. Cette

solution n'est pas envisageable ici : les configurations possibles d'un jeu comme Othello sont bien trop nombreuses pour qu'on puisse imaginer leur donner un numéro à chacune, puis manipuler un tableau prévoyant une case pour chacune.

Pour obtenir un effet similaire, on va utiliser une structure de données annexe dans laquelle, pour chaque configuration explorée, on mémorise l'évaluation qui a été calculée. Ainsi, au moment d'évaluer une configuration, la procédure devient la suivante.

1. Regarder si la configuration a déjà été évaluée.
2. Si oui, renvoyer directement l'évaluation enregistrée, sans calculer à nouveau.
3. Si non, évaluer normalement, *et enregistrer le résultat obtenu.*

Une structure adaptée pour cela est la *table de hachage*. Pour pouvoir l'utiliser, il faut simplement s'assurer qu'une fonction de hachage est bien disponible pour le type de données utilisé comme clé (ici : une configuration du jeu). On peut par exemple ajouter à notre classe `OthelloState` les définitions suivantes pour assurer cela.

```
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null || this.getClass() != o.getClass()) return false;
  OthelloState that = (OthelloState) o;
  return this.player == that.player && this.board.equals(that.board);
}

public int hashCode() {
  return Objects.hash(board, player);
}
```

Il suffit alors d'ajouter trois éléments à notre algorithme alpha/beta :

— Avant de commencer une exploration, initialiser une table de hachage :

```
// déclaration
private static HashMap<GameState, Integer> memo;

// initialisation avant exploration
public static int alphabeta(GameState state, int d, char p) {
  memo = new HashMap<>();
  return alphabeta(state, d, -10000, 10000, p);
}
```

— Au début de la fonction récursive, vérifier si la configuration analysée a déjà été traitée :

```
static int alphabeta(GameState state, int d, int a, int b, char p) {
  if (memo.containsKey(state)) return memo.get(state);
  ...
```

— À la fin de cette même fonction, enregistrer le résultat obtenu avant de le renvoyer.

```
  ...
  memo.put(state, r);
  return r;
}
```

On peut même imaginer avoir une seule table de hachage, que l'on réutilise et enrichit à chaque exploration faite au cours d'une partie. Attention cependant : dans ce cas, une même configuration est susceptible d'être évaluée plusieurs fois, à différentes profondeurs, et il faut en tenir compte dans la gestion de la table.