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 [`lo`, `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 [`lo`, `mid` [, et s'il est plus grand on poursuit dans la moitié droite [`mid` + 1, `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 $[\texttt{lo}, \texttt{hi}[$ 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 $[\texttt{lo}, \texttt{hi}[$ 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 $\texttt{hi} = \texttt{lo}$. 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 $\texttt{lo} < \texttt{hi}$. 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 ≥ `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 $\texttt{power}(a, n)$ doit être a^n.

La spécification du résultat se ramène à un unique prédicat : $\texttt{power}(a, n) = a^n$, 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 $\texttt{binarySearch}(x, t)$ 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.

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

- Spécification du résultat r de $\texttt{binarySearch}(x, t)$, 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 :

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

  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 $\texttt{stringSearch}(s, t)$ 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 $\texttt{tab}[0, \texttt{lo}[$ sont strictement inférieurs à la cible, et tous les éléments de $\texttt{tab}[\texttt{hi}, n[$ 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 :

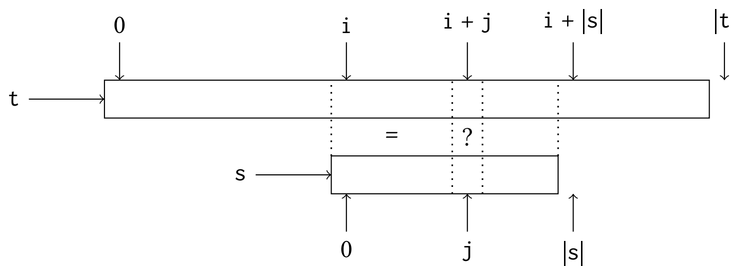
$$\begin{cases} 0 \leq \texttt{lo} \leq \texttt{hi} \leq n \\ \forall i \in [0, \texttt{lo}[\,, \texttt{tab}[i] < \texttt{x} \\ \forall i \in [\texttt{hi}, n[\,, \texttt{tab}[i] > \texttt{x} \end{cases}$$

(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`.

```java
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 :

`tab` $\longrightarrow$	−6	8	5	−6	−3	9	4	−8	7	5	7	6

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

`tab` $\longrightarrow$	−8	−6	−6	−3	4	5	5	6	7	7	8	9

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.

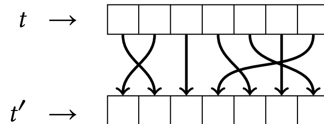
Pour exprimer que les éléments sont rangés en ordre croissant, on indique que tout élément $t'[j]$ situé à la droite d'un élément $t'[i]$ lui est supérieur ou égal.

	i	$<$	j
t' $\longrightarrow$	a	$\leq$	b

Formule associée, en notant n la taille du tableau :

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

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.

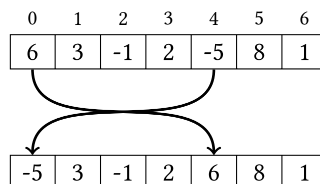
Avant de poursuivre. Sauriez-vous résoudre ce problème ?

2.2 Solution 1 : tri par sélection

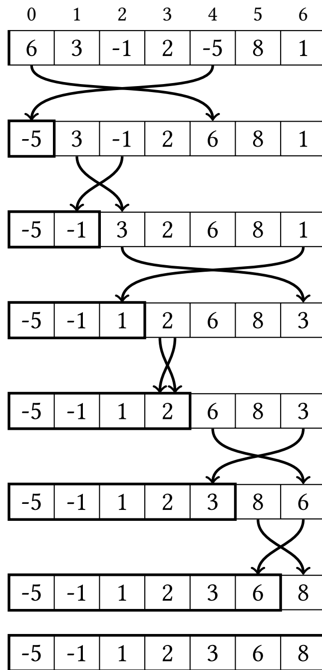
L'algorithme de ***tri par sélection*** procède ainsi :

- 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.



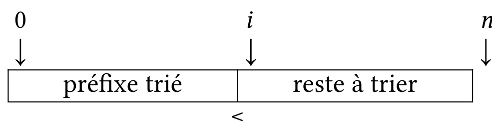
Code java. La fonction principale `selectionSort` fait appel à une fonction auxiliaire `swap` pour échanger deux éléments du tableau d'indices `i` et `j`, et une autre `indexMin` pour chercher l'indice d'un élément minimal dans un segment `tab`[`i`, n[.

```java
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 $[0, \mathtt{i}\,[$ est trié

$$\forall k_1, k_2 \in [0, \mathtt{i}\,[\,,\ k_1 < k_2 \Longrightarrow \mathtt{tab}[k_1] \le \mathtt{tab}[k_2]$$

— et les éléments du segment tab $[0, \mathtt{i}\,[$ sont plus petits que les éléments restants

$$\forall k_1 \in [0, \mathtt{i}\,[\,,\ \forall k_2 \in [\mathtt{i}, n\,[\,,\ \mathtt{tab}[k_1] \le \mathtt{tab}[k_2]$$

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 $[\mathtt{i}, n\,[$ (où on suppose que tab est un tableau de taille n).

— Précondition : le segment tab $[\mathtt{i}, n\,[$ n'est pas vide. Autrement dit : $\mathtt{i} < n$.

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

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

— l'indice jMin est dans l'intervalle $[\mathtt{i}, \mathtt{j}\,[$

$$\mathtt{i} \le \mathtt{jMin} < \mathtt{j}$$

— l'indice jMin est l'indice d'un élément minimal du segment tab $[\mathtt{i}, \mathtt{j}\,[$

$$\forall k \in [\mathtt{i}, \mathtt{j}\,[\,,\ \mathtt{tab}[\mathtt{jMin}] \le \mathtt{tab}[k]$$

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 :

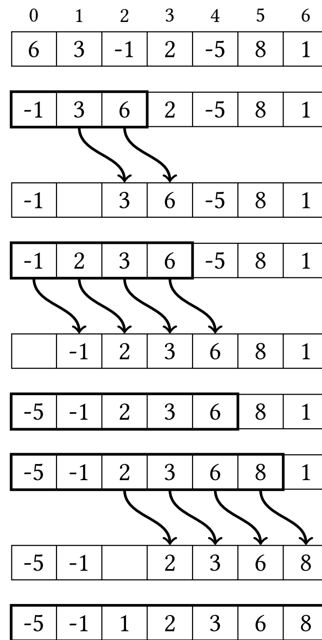
- trier en place le segment formé par le premier élément (rien à faire pour cette étape !),
- puis trier en place le segment formé par les deux premiers éléments,
- puis trier en place le segment formé par les trois premiers éléments,
- et ainsi de suite jusqu'à avoir trié l'ensemble.

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` $[0, \mathtt{i}\,[$ est trié.

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

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

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

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`]` ≥ `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`$[i]$ ≥ `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	$\Theta(1)$
$1\times2\times3\times4\times\ldots\times n$	$= \prod_{1\leq k\leq n} k$	$n!$	$\sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$	$\Theta(\sqrt{n}\left(\frac{n}{e}\right)^n)$
$\log(1)+\log(2)+\ldots+\log(n)$	$= \sum_{1\leq k\leq n} \log(k)$	$\log(n!)$	$\sim n\log(n)$	$\Theta(n\log(n))$

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

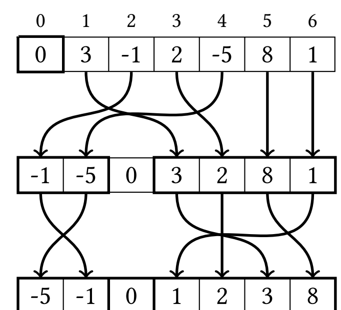
Les deux solutions précédentes au problème du tri ont pour point commun une complexité quadratique. On veut maintenant réaliser cette même tâche, mais plus rapidement. Remarquons le point suivant : pour les tris quadratiques que nous connaissons, trier un tableau de taille $\frac{n}{2}$ prend quatre 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

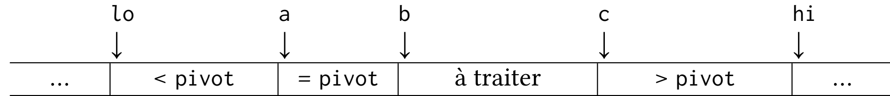
Code java. La fonction principale `quickSort` prend en paramètres un tableau `tab` et deux indices `lo` et `hi`, et trie le segment `tab`[`lo`, `hi`[. Pour cela, elle combine une boucle réorganisant le tableau autour d'un élément pivot, et des appels récursifs sur les deux sous-tableaux situés sous le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

```java
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

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

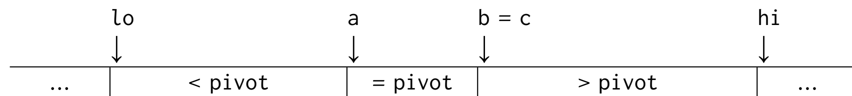
- le segment tab $[$a, b$[$ ne contient que des éléments égaux au pivot

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

- 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

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

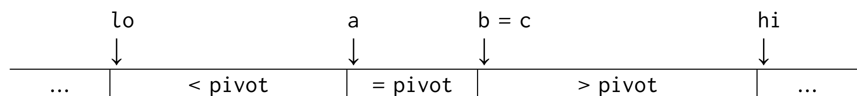
L'un de ces quatre segments est à coup sûr non vide : il s'agit de tab $[$a, b$[$, qui contient au moins une occurrence du pivot. On a donc la chaîne de comparaisons $\texttt{lo} \le \texttt{a} < \texttt{b} \le \texttt{c} \le \texttt{hi}$. À chaque tour de boucle, le segment des éléments à traiter est 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 $\texttt{tab}[i] \le \texttt{tab}[j]$ en raisonnant par cas sur les segments où se trouvent i et j.

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

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 $\texttt{fact}(n)$ suit l'une des deux formules suivantes.

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

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 $\texttt{power}(a,\ n)$ 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 :

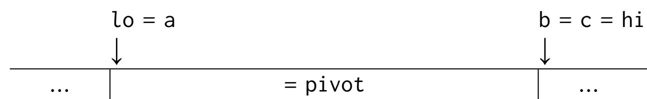
- Cas de base : $k = 0$. Alors $1 = 2^0 \leq n < 2^1 = 2$ et nécessairement $n = 1$. On calcule : $C(1) = 2 + C(0) = 2$. On a donc bien $0 + 1 \leq C(1) \leq 2(0+1)$.
- Hérédité. Soit k tel que pour tout n vérifiant $2^k \leq n < 2^{k+1}$ on a $k+1 \leq C(n) \leq 2(k+1)$. Soit n qui vérifie $2^{k+1} \leq n < 2^{k+2}$. 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 $\lfloor \log(n) \rfloor + 1 + \mathrm{w}(n)$, où $\mathrm{w}(n)$ 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 $\lceil \frac{n-1}{2} \rceil$ et $\lfloor \frac{n-1}{2} \rfloor$.

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 $n \geq 3$) :

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

On obtient :

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

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

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

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

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

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.*

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

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

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

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 $k \ge 0$	$\Theta(n^c(\log(n))^{k+1})$