6.1 Correction

Un algorithme décrit un ensemble d'opérations à effectuer pour résoudre un problème donné. C'est généralement un objet complexe à plusieurs titres. D'une part, il faut prévoir une suite d'opérations dont la réalisation va s'étaler dans le temps, et dont chacune dépendra de celles réalisées avant. D'autre part, un algorithme est destiné à s'appliquer à une variété d'entrées possibles, et doit être adapté aux spécificités de chacune d'entre elles. En outre, cette complexité augmente à mesure que l'on s'attaque à des problèmes plus complexes, ou même simplement à mesure que l'on apporte des solutions plus efficaces mais plus élaborées à des problèmes simples. La démonstration de la *correction* d'un algorithme, c'est-à-dire la vérification que l'algorithme résoud bien totalement le problème donné, devient alors une phase à part entière de son analyse.

6.1.1 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 algorithmique décrit précisément à la fois l'ensemble des entrées auxquelles un algorithme doit pouvoir être appliqué et ce que l'algorithme doit produire.

Définition 6.1 – spécification d'un problème algorithmique

La spécification d'un problème comporte deux parties :

- la description des entrées admissibles, et
- la description du résultat ou des effets attendus.

On décrit les entrées admissibles par des contraintes appelées *préconditions*. Par extension, la *spécification* d'un algorithme est la spécification du problème résolu par cet algorithme.

La spécification d'un algorithme peut être vue comme un contrat concernant le fonctionnement de l'algorithme : si on lui fournit des entrées validant les préconditions, alors l'algorithme doit produire un résultat conforme à la spécification. À ce titre, la spécification d'un problème ou d'un algorithme sert plusieurs objectifs, en fonction du point de vue.

- Lors de la conception ou de la programmation d'un algorithme, la spécification dit précisément ce qui doit être réalisé.
- Lors de l'utilisation d'un algorithme ou d'un programme dans un algorithme tiers, la spécification décrit les résultats auxquels l'utilisateur peut s'attendre.

- Lors du test d'un programme, la spécification donne les critères permettant de se prononcer sur la réussite ou l'échec d'un test.
- Lors de la justification de la correction d'un algorithme, la spécification énonce ce qui doit être démontré.

Pour les fonctions les plus simples, la spécification peut se résumer à une équation mathématique décrivant ce qui doit être calculé, en précisant éventuellement le domaine admissible pour les différents éléments.

Exemple 6.1 – exponentiation

Considérons le calcul de la n-ème puissance d'un nombre a.

$$a^n = \underbrace{a \times \ldots \times a}_{n \text{ fois}}$$

Ce problème a pour entrées deux nombres a et n, et on peut préciser sa spécification en deux points :

- on impose comme précondition que n soit un entier positif ou nul,
- le résultat doit être le nombre a^n.

Mathématiquement la notion de puissance existerait également avec des contraintes moins fortes sur n. Notre précondition définit le périmètre auquel on s'intéresse ici.

Dans des cas plus riches en revanche une spécification peut prévoir différentes issues, en fonction d'une propriété de haut niveau sur les relations qu'entretiennent les entrées.

Exemple 6.2 – recherche dans un tableau

On prend en entrée un tableau a de longueur n, et on y cherche une occurrence d'un certain élément v. Ce problème ressemble à celui étudié en introduction, mais sans contraintes sur la forme du tableau a. On peut lui donner la spécification suivante.

- Précondition sur le tableau a : il doit avoir la longueur n.
- Le résultat peut prendre deux formes, s'adaptant à la présence ou à l'absence de v dans a :
  - si la valeur v est présente dans a, alors le résultat r doit être un indice tel que $a[r] = v$,
  - si la valeur v n'est pas présente dans a, alors le résultat doit être -1.

À noter dans le précédent exemple : rien dans sa précondition n'interdit qu'un élément apparaisse deux fois. Dans le cas où l'élément v cherché apparaît plusieurs fois dans le tableau a, notre spécification demande que le résultat soit l'un quelconque des indices des occurrences de v. On pourrait énoncer des variantes imposant dans ce cas le choix d'un indice précis, par exemple le premier. Ce serait là cependant une nouvelle spécification, d'un problème différent. De même, renforcer la précondition pour restreindre les tableaux admissibles, par exemple en se limitant aux tableaux dont les éléments sont rangés en ordre croissant, définirait un problème différent, résolu par des algorithmes différents. En l'occurrence, on retrouverait le cadre de la recherche dichotomique vue en introduction.

Type des entrées et préconditions

Dans la description des entrées d'un algorithme on distingue traditionnellement deux aspects :

- la nature générale des données, pouvant correspondre à la notion de type dans un langage comme C ou OCaml,
- les propriétés supplémentaires qu'elles doivent vérifier, c'est-à-dire les préconditions.

Ainsi, le type des données n'est généralement pas considéré comme faisant partie des préconditions. Voici deux exemples simples de types d'entrées, avec différentes préconditions qui peuvent y être ajoutées.

- Les nombres entiers, pour lesquels on peut ajouter des préconditions notamment sur les plages de valeurs acceptables. Par exemple : « être positif ou nul » ou « être un indice valide d'un certain tableau ».
- Les tableaux de nombres, pour lesquels on peut ajouter des préconditions aussi bien sur la forme du tableau lui-même que sur les valeurs qu'il contient. Par exemple : « ne pas être vide », « être trié en ordre croissant » ou « ne pas contenir de doublons ».

L'action attendue d'un programme ne se limite pas toujours à renvoyer un résultat. Certains programmes visent ainsi à *modifier* une structure de données passée en entrée. La spécification donne alors une description précise du nouvel état attendu pour cette structure, en fonction de l'état d'origine.

Exemple 6.3 – tri en place d'un tableau

On prend en entrée un tableau a et on souhaite le trier : après le tri, le tableau doit contenir les mêmes éléments qu'avant, mais en ordre croissant. Dans cette spécification, l'aspect « être trié » s'applique à l'état du tableau après modification. L'aspect « contenir les mêmes éléments » est en revanche une relation entre l'état initial du tableau et l'état modifié. Pour énoncer préci-

sément la spécification du problème, il faut donc distinguer ces deux états. On note ici a l'état initial du tableau, et a' son état après modification. La spécification est alors la suivante.

- L'état a' obtenu après modification est tel que :
  - a' est trié : pour tous indices valides $i < j$ on a $a'[i] \leqslant a'[j]$,
  - a' est une permutation de a : il existe une permutation σ des indices du tableau telle que pour tout indice valide i on a $a'[i] = a[\sigma(i)]$.
- L'état initial a du tableau donné en entrée est arbitraire (il n'y a pas de précondition).

Préconditions

Les préconditions décrivent les contraintes que doivent vérifier les données prises en entrée par un algorithme résolvant le problème. Dit autrement, les préconditions définissent les entrées valides, et délimitent ainsi les contours du problème que l'on cherche à résoudre. Pour reprendre la métaphore du contrat, un algorithme n'est tenu de fournir les résultats attendus que lorsque ses entrées respectent les préconditions. Dans tous les autres cas, son comportement peut être arbitraire. Selon le point de vue, les préconditions jouent donc un rôle différent.

- Lors de la conception d'un algorithme, on peut tenir les préconditions pour acquises. On cherche à résoudre le problème uniquement pour les entrées valides. Le reste est hors sujet.
- Lors de la définition d'un ensemble de tests, les préconditions délimitent également les entrées qu'il est légitime de tester : pour une entrée ne validant pas les préconditions, la spécification ne dit rien du comportement attendu.
- Lors du raisonnement sur un algorithme, les préconditions deviennent des hypothèses. On suppose qu'elles sont valides et on peut s'en servir pour déduire d'autres faits.
- Lors de l'utilisation d'un algorithme, il faut s'assurer qu'on ne lui fournit que des entrées valides. Faute de cela, le résultat pourra ne pas être celui attendu.
- Lors de l'écriture d'un programme, on *peut* prévoir d'interrompre l'exécution et produire une erreur lorsque les préconditions ne sont pas réalisées. Dans ce livre on utilisera parfois `assert` à cet effet au début d'une fonction, en C aussi bien qu'en OCaml. On peut aussi ne rien faire à ce propos, et laisser le programme faire n'importe quoi lorsque les entrées sont invalides.

Ainsi, dans la conception d'un algorithme de recherche dans un tableau trié, toute considération sur les tableaux non triés est hors sujet. Si un utilisateur d'un tel algorithme fournit en entrée un tableau non trié, il a toutes les chances de recevoir en retour un résultat incorrect, mais il en sera le seul fautif. En l'espèce, il ne serait pas raisonnable pour le programmeur de vérifier que le tableau est trié : cette seule vérification coûterait autant que la recherche la plus naïve !

Programme 6.2 – exponentiation rapide en OCaml

```
let rec power a n =
  if n = 0 then 1
  else let b = power a (n/2) in
       if n mod 2 = 0 then b * b
       else a * b * b
```

6.1.2 Preuves de correction par récurrence

On a vu en introduction de ce chapitre que la justification de la correction d'un algorithme peut nécessiter un suivi fin de l'évolution de chacune des variables. Cependant, les choses peuvent devenir considérablement plus simples dans le cas d'un algorithme qui ne modifie aucune variable ni aucune structure de données. On peut alors ramener le raisonnement sur les algorithmes à un raisonnement équationnel similaire à celui qui se présenterait dans un calcul mathématique ordinaire.

Programmes récursifs et équations. Cette situation s'applique notamment à de nombreux programmes OCaml, puisque dans ce langage les variables sont immuables, de même que certaines structures de base comme les listes. Considérons le problème de l'exponentiation : écrire une fonction `power` qui prend en entrée deux entiers a et n avec n positif ou nul, et qui renvoie a^n. Nous allons en considérer deux solutions écrites en OCaml.

Une version naïve s'écrirait comme suit.

```
let rec power a n =
  if n = 0 then 1
  else a * power a (n-1)
```

On peut résumer l'action de cette fonction par ces deux équations.

$$\begin{cases} \texttt{power}\ a\ 0 &= 1 \\ \texttt{power}\ a\ n &= a \times (\texttt{power}\ a\ (n-1)) \qquad \text{si } n > 0 \end{cases}$$

Ces deux équations sont au plus près de la définition de a^n.

Le programme 6.2 propose une version plus subtile. Comme la précédente, cette nouvelle fonction isole le cas a^0. Après, les choses changent : la fonction utilise des formules différentes dans le cas où n est pair, c'est-à-dire où il existe un k tel que

$n = 2k$, et dans le cas où n est impair, c'est-à-dire où il existe un k tel que $n = 2k + 1$.

$$\begin{cases} \texttt{power}\ a\ 0 &= 1 \\ \texttt{power}\ a\ (2k) &= (\texttt{power}\ a\ k)^2 & \text{si } k > 0 \\ \texttt{power}\ a\ (2k+1) &= a \times (\texttt{power}\ a\ k)^2 \end{cases}$$

Ces trois équations reflètent le calcul effectué par cette nouvelle version de `power`, et peuvent servir à raisonner sur cet algorithme récursif. Notez que ces équations font abstraction de la variable intermédiaire `b` du programme pour donner directement des formules simplifiées, qui sont suffisantes pour raisonner sur le résultat produit.

Principe de raisonnement par récurrence. Le principe de raisonnement par récurrence permet d'établir qu'une certaine propriété P mentionnant un entier n est valide quelle que soit la valeur prise par n. Le principe consiste à montrer que dès que P est valide pour certains entiers, alors elle l'est encore nécessairement pour les suivants. Ainsi, si la propriété est vraie pour 0, on peut ensuite en déduire de proche en proche qu'elle est vraie pour n'importe quel entier.

Dans le développement suivant, on note $P(n)$ pour rendre visible le fait qu'une propriété P dépend de n. On peut alors noter $P(0)$, $P(5)$ ou $P(n+1)$ pour parler de la propriété P appliquée à des entiers particuliers.

Théorème 6.1 – principe de récurrence simple

Soit $P(n)$ une propriété dépendant d'un entier naturel n. Si

1. $P(0)$ est valide, et
2. pour tout n, la validité de $P(n)$ implique la validité de $P(n+1)$,

alors $P(n)$ est valide pour tout entier $n \in \mathbb{N}$.

Démonstration. Supposons que P vérifie bien les deux points mais qu'il existe au moins un $n \in \mathbb{N}$ invalidant P. Soit n_0 le plus petit des entiers n tels que $P(n)$ n'est pas vraie. Cet entier n_0 ne peut pas être 0, car cela contredirait le premier point. Alors on a bien $n_0 - 1 \in \mathbb{N}$ et par minimalité de n_0 la propriété $P(n_0 - 1)$ est vraie. Mais alors du deuxième point on déduit que $P(n_0)$ est vraie : contradiction. □

L'utilisation de ce principe de récurrence demande d'abord de définir une propriété cible P. On a ensuite deux étapes.

1. *Cas de base :* on vérifie la validité de $P(0)$.
2. *Hérédité :* on montre que pour tout $n \in \mathbb{N}$ tel que $P(n)$ est valide, $P(n+1)$ est encore valide. Dans ce cadre, l'hypothèse selon laquelle $P(n)$ est valide est appelée *hypothèse de récurrence.*

Le principe de récurrence simple permet alors directement de déduire que la propriété P est valide pour tout $n \in \mathbb{N}$.

Preuve de correction par récurrence simple. On peut appliquer ce principe pour démontrer la correction d'une fonction récursive sur les entiers comme la version naïve de `power`.

Exemple 6.4 – correction de l'exponentiation naïve

On note $P(n)$ la propriété « pour tout a, `power` $a\ n = a^n$ ». Démontrons par récurrence que $P(n)$ est valide pour tout entier n.

- Cas de base. On veut vérifier que $P(0)$ est valide, c'est-à-dire que `power` $a\ 0 = a^0$. C'est immédiat puisque par définition `power` $a\ 0 = 1 = a^0$.
- Hérédité. On considère un n pour lequel $P(n)$ est valide, et on veut justifier que $P(n+1)$ est encore valide. On peut le faire avec le calcul suivant.

$$\begin{aligned} \texttt{power}\ a\ (n+1) &= a \times (\texttt{power}\ a\ n) && \text{car } n+1 \geqslant 1 \\ &= a \times a^n && \text{par hypothèse de récurrence} \\ &= a^{n+1} \end{aligned}$$

On en déduit que $P(n)$ est valide pour tout entier $n \in \mathbb{N}$. Autrement dit, notre fonction d'exponentiation naïve est correcte.

Principe de récurrence forte. Le principe de récurrence simple est adapté pour raisonner sur un programme comme la première version de `power`, où l'appel récursif éventuellement utilisé par `power` $a\ n$ concerne le prédécesseur direct de n. Dans la deuxième version cependant, l'appel récursif `power` $a\ \frac{n}{2}$ concerne un entier potentiellement plus lointain : on s'écarte du cas d'hérédité simple. Une variante du principe de récurrence nous donne plus de souplesse.

Théorème 6.2 – principe de récurrence forte

Soit $P(n)$ une propriété dépendant d'un entier naturel n. Si

1. pour tout n, la validité conjointe des $P(k)$ pour tous les naturels $k < n$ implique la validité de $P(n)$,

alors $P(n)$ est valide pour tout entier $n \in \mathbb{N}$.

Démonstration. Notons $P'(n)$ la propriété « $P(\ell)$ est valide pour tout $\ell \leqslant n$ », et démontrons qu'elle est vraie pour tout $n \in \mathbb{N}$, par une récurrence simple.

- Cas de base : on veut montrer que $P'(0)$ est valide, c'est-à-dire que $P(\ell)$ est valide pour tout $\ell \leqslant 0$, ce qui se résume à montrer que $P(0)$ est valide. On applique l'hypothèse que nous avons sur P : la validité conjointe des $P(k)$ pour $k < 0$ implique la validité de $P(0)$. Or il n'existe pas de $k < 0$, donc toutes les $P(k)$ pour $k < 0$ sont bien vraies, et $P(0)$ également.
- Hérédité : on suppose $P'(n)$ vraie pour un n donné et on veut en déduire $P'(n+1)$. Autrement dit, on suppose $P(\ell)$ valide pour tout $\ell \leqslant n$ et on cherche à en déduire que $P(\ell)$ est valide pour tout $\ell \leqslant n+1$. Comme par hypothèse de récurrence, $P(\ell)$ est déjà valide pour tout $\ell \leqslant n$, il ne reste qu'à montrer la validité de $P(n+1)$. Or, par hypothèse de récurrence à nouveau, tous les $P(k)$ pour $k < n+1$ sont valides. Donc par hypothèse sur P la propriété $P(n+1)$ est bien valide.

Finalement, par principe de récurrence simple la propriété $P'(n)$ est vraie pour tout $n \in \mathbb{N}$. On en déduit que $P(n)$ elle-même est vraie pour tout $n \in \mathbb{N}$. □

Comme dans le cas de la récurrence simple, l'utilisation du principe de récurrence forte commence par l'identification d'une propriété cible P. On a ensuite un unique critère à vérifier.

1. Pour tout n tel que $P(k)$ est valide pour tous les $k < n$, on montre que $P(n)$ est valide.

Toutes les hypothèses de validité de $P(k)$ pour les $k < n$ sont des *hypothèses de récurrence*. On peut être surpris par l'absence apparente d'un cas de base à ce principe de récurrence. Cette absence n'est cependant qu'une illusion : le critère unique de la récurrence appliqué au rang 0, demande de justifier que $P(0)$ est vraie en utilisant comme hypothèses de récurrence la validité de tous les $P(k)$ pour $k < 0$. Comme il n'existe pas de $k < 0$ nous n'avons en fait aucune hypothèse de récurrence, et la situation devient donc exactement celle du cas de base de la récurrence simple.

Preuve de correction par récurrence forte. Le principe de récurrence forte est plus souple dans son utilisation que celui de récurrence simple : alors que dans le cas héréditaire de la récurrence simple on dispose d'une unique hypothèse de récurrence, concernant le rang précédent, on a avec la récurrence forte des hypothèses de récurrence pour *tous* les rangs précédents. L'objectif ne pourra qu'en être plus facile à atteindre.

Grâce à ce principe de récurrence renforcé, nous allons pouvoir démontrer la correction de notre deuxième version de l'exponentiation.

Exemple 6.5 – correction de l'exponentiation rapide

Posons $P(n)$ la propriété « pour tout a, `power a n` $= a^n$ », et démontrons-la par récurrence forte.

Considérons donc un $n \in \mathbb{N}$ tel que l'équation $\texttt{power}\ a\ k = a^k$ soit vraie pour tous les $k < n$ et raisonnons par cas sur n.

- Si $n = 0$, alors $P(n)$ est bien vraie. En effet, quel que soit a on a bien $\texttt{power}\ a\ 0 = 1 = a^0$.
- Sinon, raisonnons par cas sur la parité de n.
  - Si n est pair, c'est-à-dire s'il existe k tel que $n = 2k$, alors $\texttt{power}\ a\ n$ renvoie une valeur égale à $(\texttt{power}\ a\ k)^2$. Or, avec $n \neq 0$ on a $k < n$: la propriété $P(k)$ est vraie et nous donne l'équation $\texttt{power}\ a\ k = a^k$. Donc

$$\begin{aligned} \texttt{power}\ a\ n &= (\texttt{power}\ a\ k)^2 \\ &= (a^k)^2 && \text{par hyp. de récurrence} \\ &= a^{2k} \\ &= a^n \end{aligned}$$

    et $P(n)$ est bien vérifiée.
  - À l'inverse, si n est impair, c'est-à-dire s'il existe k tel que $n = 2k+1$, alors $\texttt{power}\ a\ n$ renvoie une valeur égale à $a \times (\texttt{power}\ a\ k)^2$, avec à nouveau la propriété P vraie pour k puisque $k < n$. On peut donc conclure avec le calcul

$$\begin{aligned} \texttt{power}\ a\ n &= a \times (\texttt{power}\ a\ k)^2 \\ &= a \times (a^k)^2 && \text{par hyp. de récurrence} \\ &= a \times a^{2k} \\ &= a^{2k+1} \\ &= a^n \end{aligned}$$

    et $P(n)$ est bien vérifiée.

Par principe de récurrence forte, notre propriété P est bien vraie pour tout entier $n \in \mathbb{N}$. Autrement dit, l'appel de fonction $\texttt{power}\ a\ n$ calcule bien a^n pour tout entier positif n.

Correction d'algorithmes récursifs manipulant des listes. Au-delà des algorithmes numériques, le mode de raisonnement équationnel s'applique naturellement aux algorithmes récursifs manipulant des structures de données immuables comme les listes OCaml. Ainsi, les fonctions `insert` et `insertion_sort` du programme 6.3

définissent les équations suivantes.

Programme 6.3 – tri par insertion en OCaml

```
let rec insert x l = match l with
  | []                -> [x]
  | y :: _ when x <= y -> x :: l
  | y :: l'           -> y :: insert x l'

let rec insertion_sort = function
  | []     -> []
  | x :: l -> insert x (insertion_sort l)
```

$$\texttt{insertion\_sort}\ l \;=\; \begin{cases} \texttt{[]} & \text{si } l = \texttt{[]} \\ \texttt{insert}\ x\ (\texttt{insertion\_sort}\ l') & \text{si } l = x::l' \end{cases}$$

$$\texttt{insert}\ x\ l \;=\; \begin{cases} [x] & \text{si } l = \texttt{[]} \\ x::l & \text{si } l = y::l' \text{ et } x \leqslant y \\ y::(\texttt{insert}\ x\ l') & \text{si } l = y::l' \text{ et } x > y \end{cases}$$

La fonction `insertion_sort` trie récursivement une liste non vide en triant d'abord sa queue, puis en insérant l'élément de tête à sa place légitime dans la queue triée [1].

Pour toute liste OCaml l, notons $l^\dagger$ la liste formée des mêmes éléments rangés par ordre croissant. Montrer la correction de la fonction `insertion_sort` revient à démontrer l'équation suivante.

$$\texttt{insertion\_sort}\ l = l^\dagger$$

Cependant, les principes de récurrence que nous avons utilisés pour raisonner sur les algorithmes d'exponentiation ne s'appliquent qu'à des entiers. Pour les utiliser dans ce cadre, il faut donc raisonner sur la taille des listes manipulées (à la section 6.4 nous verrons un nouveau principe de récurrence permettant un raisonnement plus direct).

1. Cet fonction de tri n'est pertinente que pour des listes de petite taille. Nous en verrons plus dans la suite de ce chapitre.

Exemple 6.6 – correction de l'insertion dans une liste triée

Démontrons que la fonction `insert` insère un élément à sa place dans une liste triée. On note $P(n)$ la propriété « pour toute liste l triée de longueur n et tout élément x, on a `insert` x $l = (x::l)^\dagger$ ». Montrons par récurrence simple que $P(n)$ est vraie pour tout $n \in \mathbb{N}$.

- Cas de base : la seule liste de longueur zéro est `[]`, pour laquelle on a `insert` x `[]` $= [x]$. Or $[x]$ est bien une permutation triée de $x::[]$.
- Hérédité : supposons $P(n)$ vraie et considérons une liste l de longueur $n+1$. Cette liste a nécessairement la forme $y::l'$ avec l' de longueur n. On veut montrer que `insert` x $(y::l')$ est une permutation triée de $x::y::l'$. On a deux cas selon la comparaison de x et y.
  - Si $x \leqslant y$ alors `insert` x $(y::l') = x::y::l'$ et cette liste est bien une permutation triée d'elle-même.
  - Si $x > y$ alors `insert` x $(y::l') = y::$(`insert` x l'). Par hypothèse de récurrence `insert` x l' est une permutation triée de $x::l'$, donc $y::$(`insert` x l') est une permutation de $y::x::l'$ et donc de $x::y::l'$. En outre, la liste $y::l'$ était supposée triée : pour tout $z \in l$ on a $y \leqslant z$. Donc $y::$(`insert` x l') est triée.

Exemple 6.7 – correction du tri par insertion d'une liste

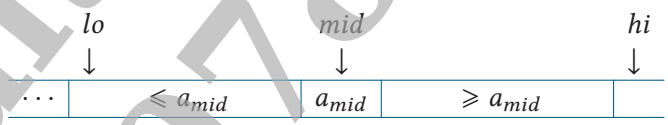
Démontrons que `insertion_sort` trie une liste. On note $P(n)$ la propriété « pour toute liste l de taille n, `insertion_sort` $l = l^\dagger$ » et on raisonne par récurrence.

- Cas de base : la seule liste de longueur zéro est `[]`, pour laquelle on a `insertion_sort` `[]` $=$ `[]`. Or `[]` est bien une permutation triée d'elle-même.
- Hérédité : supposons que `insertion_sort` trie toute liste de longueur n, et considérons une liste l de longueur $n+1$. Nécessairement, l est de la forme $x::l'$ avec l' de longueur n. Par définition de `insertion_sort` on a `insertion_sort` $(x::l')$ $=$ `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 la fonction `insert` s'applique bien. Par spécification de `insert`, `insert` x (`insertion_sort` l') est une permutation triée de $x::$(`insertion_sort` l'), et donc une permutation triée de $x::l'$.

Programme 6.4 – recherche dichotomique récursive

```
let rec binary_search_rec v a lo hi =
  (* on recherche v dans a[lo..hi[ *)
  if hi <= lo then raise Not_found;
  let mid = lo + (hi - lo) / 2 in
  if v < a.(mid) then binary_search_rec v a lo mid
  else if v > a.(mid) then binary_search_rec v a (mid + 1) hi
  else mid
```

Correction d'algorithmes récursifs manipulant des tableaux. Le mode de raisonnement équationnel s'applique à tout algorithme récursif ne modifiant aucune variable ou structure de données. Cela vaut y compris lorsque l'algorithme manipule des structures mutables, tant que celles-ci ne sont que consultées, et non modifiées. Le programme 6.4 donne le code OCaml d'une fonction qui cherche une occurrence de l'élément v dans le tableau a entre les indices lo (inclus) et hi (exclu). Précondition : le tableau a est supposé trié en ordre croissant. Cette fonction est une version récursive de la recherche dichotomique. Elle consulte l'élément à un indice mid correspondant au centre de l'intervalle de recherche (arrondi vers la gauche). À moins que cet élément soit déjà celui cherché, la recherche se poursuit alors récursivement dans la moitié gauche ou droite, selon que la valeur cherchée v est plus petite ou plus grande que l'élément médian $a[mid]$.



On peut démontrer la correction de cet algorithme par récurrence (forte) sur la longueur de l'intervalle de recherche.

Exemple 6.8 – correction de la recherche dichotomique récursive

Démontrons donc que pour tout tableau trié a, toute valeur v cherchée et tous indices lo et hi définissant un intervalle valide de a :

- si v apparaît dans $a\,[lo, hi[$ alors `binary_search_rec` v a lo hi renvoie un indice $i \in [lo, hi[$ tel que $a[i] = v$, et
- si v n'apparaît pas dans $a\,[lo, hi[$ alors `binary_search_rec` v a lo hi déclenche l'exception `Not_found`,

par récurrence forte sur la longueur $hi - lo$ de l'intervalle de recherche. On raisonne d'abord par cas sur la longueur de l'intervalle.

- Si $hi \leqslant lo$, alors `binary_search_rec` v a lo hi déclenche l'exception `Not_found`. C'est bien conforme à l'issue attendue, puisque l'intervalle $a\,[lo, hi[$ est vide et ne peut donc contenir v.
- Sinon $lo < hi$, et l'indice $mid = lo + \lfloor \frac{hi-lo}{2} \rfloor$ est tel que $lo \leqslant mid < hi$. On en déduit en particulier que mid est un indice valide du tableau a (sûreté), et on raisonne par cas sur la comparaison entre $a[mid]$ et v.
  - Si $v = a[mid]$ alors `binary_search_rec` v a lo hi renvoie mid. C'est correct car v apparaît bien dans le tableau et mid est justement l'indice d'une occurrence de v.
  - Si $v < a[mid]$ alors `binary_search_rec` v a lo hi renvoie le résultat de `binary_search_rec` v a lo mid. La longueur $mid - lo$ de l'intervalle de recherche couvert par l'appel récursif est bien strictement inférieure à $hi - lo$: par hypothèse de récurrence le résultat de cet appel sera correct. On conclut par cas sur la présence de v dans $a\,[lo, hi[$.
    - Si v n'apparaît pas dans $a\,[lo, hi[$, alors en particulier v n'apparaît pas non plus dans $a\,[lo, mid[$: par hypothèse de récurrence `binary_search_rec` v a lo mid déclenche l'exception `Not_found`, ce qui est bien l'issue attendue pour `binary_search_rec` v a lo hi.
    - Cas où v apparaît dans $a\,[lo, hi[$. Comme a est trié et $v < a[mid]$ on déduit que pour tout indice $i \in [mid, hi[$ on a $v < a[i]$. Ainsi, toutes les occurrences de v sont nécessairement dans $a\,[lo, mid[$, et l'appel récursif `binary_search_rec` v a lo mid sur cet intervalle de recherche trouvera bien l'élément v.
  - Le cas $v > a[mid]$ est symétrique.

Finalement, pour chercher la valeur v dans l'intégralité d'un tableau a supposé trié, il suffit d'appeler `binary_search_rec` avec l'intervalle de recherche couvrant le tableau entier.

```
let binary_search v a =
  binary_search_rec v a 0 (Array.length a)
```

Sûreté

Certaines opérations élémentaires des algorithmes et des programmes sont potentiellement dangereuses : employées avec les mauvais paramètres, elles entraînent l'interruption immédiate du programme, voire des comportements indéterminés. Par exemple :

- la division n'est possible que pour des diviseurs non nuls,
- la racine carrée n'existe que pour des nombres positifs ou nuls,
- l'accès à un tableau n'est légitime que pour des indices valides.

Chaque fois qu'une telle opération apparaît, on doit s'assurer qu'elle est bien utilisée de manière valide. On appelle cette propriété la *sûreté* d'un algorithme.
Ainsi, en présence d'une opération `3 / x` ou `3 % x` on doit garantir que `x` ne peut valoir `0`. De même, en présence d'un accès `a[i]` à un tableau il faut s'assurer que l'indice `i` ne peut pas être négatif, ni supérieur à l'indice maximal du tableau `a`. Ce souci transparaît notamment dans la justification de la correction de la recherche dichotomique : on y a vérifié que l'indice `mid` était valide, grâce à l'encadrement $lo \leqslant mid < hi$ et l'hypothèse que tous les indices de l'intervalle $[lo, hi[$ étaient bien des indices valides du tableau a.

Effets de bords qui n'empêchent pas le raisonnement équationnel

Certains programmes produisent des effets de bord qui n'invalident pas l'hypothèse fonctionnelle, c'est-à-dire qui n'ont pas d'influence sur la valeur des résultats produits. On peut citer un programme qui enregistrerait à la volée des données de diagnostic indépendantes du résultat produit, comme le nombre d'occurrences de certaines opérations. Nous verrons également plus loin (section 6.3.8 et section 9.4.2) des programmes enregistrant certains résultats intermédiaires pour pouvoir les réutiliser plus tard. Dans une telle situation, on a un effet de bord qui modifie possiblement le temps nécessaire à la production du résultat, mais pas la valeur du résultat. Dans toutes ces situations, le raisonnement équationnel reste possible.

6.1.3 Invariants de boucle

Le raisonnement équationnel sur les résultats des algorithmes et des programmes est pratique, mais n'est possible que sous une hypothèse : les variables et structures de données utilisées doivent être *immuables*. Lorsque ce n'est pas le cas, c'est-à-dire par exemple lorsque l'on s'autorise à modifier le contenu d'une variable ou d'un tableau, il faut intégrer au raisonnement un suivi de l'évolution des variables et des données.

Le schéma d'une analyse de correction devient donc :

- supposer qu'*avant exécution* les préconditions sont valides,
- caractériser l'évolution des variables du programme,
- sur la base des valeurs des variables et données obtenues *après exécution*, vérifier que le résultat ou l'effet est conforme à ce qui était attendu.

Programme 6.5 – exponentiation rapide

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

Suivi de l'évolution des variables d'un programme. Considérons l'évolution des deux variables a et b au cours de l'exécution de la séquence d'instructions suivante.

```
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 est modifiée et contient la valeur $n_a - n_b$. Après la deuxième, b est modifiée à son tour et contient $n_b + (n_a - n_b) = n_a$. Après la dernière, a est modifiée à nouveau et 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 de chaque variable après chaque instruction.

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 fonctionne lorsque l'on sait précisément quelles opérations sont réalisées et dans quel ordre. L'exemple d'échange des valeurs de deux variables a cette propriété, puisque cet algorithme ne contient ni branchements, ni opérations conditionnelles, ni boucles. Cette situation est rare !

Considérons un programme un peu plus complexe : le programme 6.5 donne une version itérative de l'algorithme d'exponentiation rapide. On peut suivre l'évolution

de ses variables dans une exécution particulière en choisissant des valeurs de départ. Ainsi, en fournissant les paramètres $a = 2$ et $n = 5$ on obtient le déroulé suivant.

Instruction	r	a	n
`r = 1`	1	2	5
`r *= a`	2	2	5
`a = a * a`	2	4	5
`n = n / 2`	2	4	2
`a = a * a`	2	16	2
`n = n / 2`	2	16	1
`r *= a`	32	16	1
`a = a * a`	32	256	1
`n = n / 2`	32	256	0
`return r`	32		

La boucle **`while`** formant le cœur de l'algorithme divise naturellement le processus : chaque tour de boucle est une étape du calcul, et ces étapes sont un bon niveau de granularité pour résumer l'évolution des différentes variables. En prenant le nouvel exemple $a = 2$ et $n = 11$, on aurait ainsi le développement suivant.

Étape	r	a	n
Init.	1	2	11
Tour 1	2	4	5
Tour 2	8	16	2
Tour 3	8	256	1
Tour 4	2048	65536	0
Fin	2048		

Sans préciser la valeur fournie pour n, il est en revanche impossible d'établir un tel tableau. Les instructions réalisées dépendent de n, et en particulier de son écriture binaire : le nombre de tours de boucle est lié au nombre de chiffres, et la réalisation ou non du branchement à chaque tour dépend des valeurs des bits consécutifs.

À la place, on cherche à déterminer des relations entre les valeurs des différentes variables, qui restent vraies dans toutes les exécutions possibles.

Invariant de boucle. Pour raisonner en toute généralité sur un algorithme complexe, on cherche à établir des *invariants de boucle* : pour chaque boucle de l'algorithme étudié on cherche des propriétés à propos des variables, qui sont valides du début à la fin de l'exécution de la boucle (ces propriétés sont donc « *invariablement* valides »).

Définition 6.2 – invariant de boucle

Étant donnés un algorithme et une boucle de cet algorithme, un *invariant de boucle* est une propriété qui, quelles que soient les entrées valides fournies à l'algorithme :

- est valide avant le premier tour de boucle,
- est préservée par chaque tour de boucle.

Dans cet énoncé, la notion de « être préservée » est définie ainsi : si la propriété est valide au commencement d'un tour de boucle, alors elle l'est encore à la fin de ce tour. On considère qu'une entrée de l'algorithme est « valide » si elle en vérifie les préconditions.

Un invariant peut faire référence aux variables et données manipulées par l'algorithme. Le plus souvent, les invariants intéressants font notamment référence aux variables et données modifiées par la boucle elle-même : leur raison d'être est la caractérisation de ces modifications. Dans sa forme la plus simple, un invariant peut par exemple être une relation arithmétique entre les variables du programme.

Exemple 6.9 – exponentiation naïve

Voici une version itérative de l'algorithme d'exponentiation naïf : on calcule a^n en multipliant n fois par a une variable `r` initialisée à 1.

```
int power_n(int a, int n) {
  int r = 1;
  for (int i=0; i < n; i++) {
    r *= a;
  }
  return r;
}
```

On peut résumer la progression de cet algorithme par une unique formule, liant l'évolution des variables `r` et `i` au fil des tours de boucle successifs : à chaque étape du calcul on a

$$r = a^i$$

Cette formule est un invariant de la boucle **`for`** de `power_n`. On peut en effet vérifier que :

- juste avant le premier tour de boucle, on a bien $r = 1 = a^0 = a^i$, et
- si cette propriété est vraie au début d'un tour de boucle alors elle l'est encore à la fin, puisque durant un tour r est multipliée par a et i est incrémentée de 1.

L'écriture précise d'une justification de la préservation de l'invariant demande d'être précautionneux : certaines variables étant modifiées entre le début et la fin d'un tour donné, on gagne à se donner des notations différenciant les valeurs du début et celles de la fin.

Exemple 6.10 – préservation de l'invariant de l'exponentiation naïve

On note r et i les valeurs de `r` et `i` au début d'un tour de boucle, et r' et i' leurs valeurs à la fin de ce même tour. Pour démontrer la préservation de l'invariant on suppose donc $r = a^i$ et on vérifie que $r' = a^{i'}$. Une fois ce contexte posé, la justification de la préservation redevient un simple calcul.

$$r' = a \times r = a \times a^i = a^{i+1} = a^{i'}$$

Les invariants d'une boucle étant des propriétés préservées par les tours de boucle successifs, ils nous donnent des informations qui seront encore valides au terme de l'exécution de la boucle.

Propriété 6.1 – invariant de boucle correct

Étant donnés un algorithme, une boucle de cet algorithme et un invariant de cette boucle, pour toute entrée valide de l'algorithme l'invariant est vérifié à la fin de l'exécution de la boucle.

Notez que cet énoncé n'est utile que si la boucle a effectivement une fin ! La notion d'invariant reste cependant indépendante de la terminaison. Même si une boucle ne termine jamais, ses invariants restent vérifiés de manière permanente.

Exemple 6.11 – conclusion sur l'exponentiation naïve

Une fois qu'on a établi que l'équation $r = a^i$ est un invariant de la boucle `for` de `power_n`, on en déduit que la formule est encore vraie à la fin de l'exécution de la boucle, c'est-à-dire lorsque $i = n$. La valeur de r à cet instant est donc a^n, et la valeur renvoyée par la fonction est bien correcte.

Remarquez que l'invariant d'une boucle peut être *temporairement* invalide pendant l'exécution d'un tour de boucle, les variables n'étant pas toutes mises à jour en même temps. Ce qui compte est que l'invariant soit vrai à nouveau à la fin du tour. Cela suffit à garantir qu'il soit encore vrai au début du tour suivant, puis à la fin du tour suivant, et ainsi de suite jusqu'à la fin des tours. Ce phénomène est visible par exemple dans le cas de la fonction `power_b` d'exponentiation rapide (programme 6.5).

Exemple 6.12 – exponentiation rapide

Reprenons la fonction power_b (programme 6.5). Notons a_0 et n_0 les valeurs initialies des deux arguments a et n. Notons également a, n et r les valeurs des trois variables du programme à un instant donné. Alors la formule

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

est un invariant de la boucle. Vérifions ce point.

- Avant le premier tour on a $r = 1$, $a = a_0$ et $n = n_0$, et l'équation $r \times a^n = a_0^{n_0}$ est immédiate.
- Supposons la formule $r \times a^n = a_0^{n_0}$ vraie au début d'un tour de boucle et notons a', n' et r' les valeurs des trois variables telles que mises à jour à la fin du tour. On a alors deux cas à considérer selon la parité de n.

Décomp. n	Mises à jour	Calcul $r' \times a'^{n'}$
$n = 2k$	$\begin{cases} a' = a^2 \\ n' = k \\ r' = r \end{cases}$	$\begin{aligned} r \times (a^2)^k &= r \times a^{2k} \\ &= r \times a^n \end{aligned}$
$n = 2k+1$	$\begin{cases} a' = a^2 \\ n' = k \\ r' = a \times r \end{cases}$	$\begin{aligned} a \times r \times (a^2)^k &= r \times a^{2k+1} \\ &= r \times a^n \end{aligned}$

Dans tous les cas $r' \times a'^{n'} = a_0^{n_0}$ et l'invariant reste valide à la fin du tour de boucle.

Il n'y a pas de restriction imposant que chaque boucle ne dispose que d'un unique invariant : toutes les propriétés effectivement préservées par la boucle peuvent être ajoutées à la liste de ses invariants. En outre, les invariants d'une boucle ne se limitent pas à de simples équations arithmétiques liant les différentes variables du programme. On peut utiliser comme invariants des propriétés arbitrairement complexes. En étudiant la recherche dichotomique en introduction de ce chapitre, nous avons ainsi répertorié trois propriétés invariantes, dont deux comportaient une quantification sur les valeurs d'un fragment de tableau.

Exercice
43 p.308
44 p.308

Exemple 6.13 – invariants de la recherche dichotomique

La boucle de la fonction `binary_search` (programme 6.1) admet plusieurs invariants intéressants, impliquant notamment que les variables `lo` et `hi` découpent le tableau en trois intervalles, et que la valeur v cherchée ne peut pas se trouver en dehors de l'intervalle `[lo, hi[`. Voici les trois énoncés.

1. $0 \leqslant lo \leqslant hi \leqslant n$,
2. pour tout indice $i \in [0, lo[$ on a $a[i] < v$,
3. pour tout indice $i \in [hi, n[$ on a $v < a[i]$.

Notez qu'il n'y a pas une unique manière d'énoncer les invariants d'une boucle. Ici on aurait pu remplacer les deux derniers invariants par l'unique propriété « pour tout indice i tel que $a[i] = v$, on a $lo \leqslant i < hi$ ». Cette dernière propriété est un peu moins précise, mais elle reste suffisante pour justifier la correction de l'algorithme !

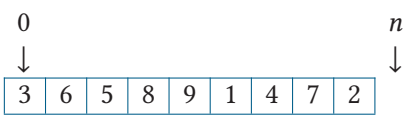
6.1.4 Cas d'étude : correction d'algorithmes de tri

Nous allons détailler dans cette section trois algorithmes de tri en place d'un tableau. Leur objectif commun est de réarranger les éléments d'un tableau donné en argument de sorte à les trier par ordre croissant.

6.1.4.1 Tri par insertion

Lorsqu'un algorithme contient plusieurs boucles, imbriquées ou non, chacune dispose de ses propres invariants et les uns peuvent servir à la justification des autres. Ce cas de figure, très courant, apparaît par exemple dans l'algorithme de *tri par insertion*.

Présentation de l'algorithme. L'algorithme de tri par insertion réarrange les éléments d'un tableau de sorte à les trier par ordre croissant. Il procède en réarrangeant d'abord les deux premiers éléments, puis en y intégrant le troisième, puis le quatrième, et ainsi de suite jusqu'à avoir traité tous les éléments. Le programme 6.6 en donne une réalisation en C. Si on part du tableau

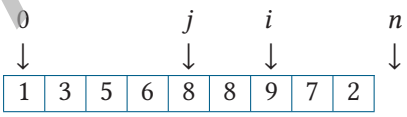


Programme 6.6 – tri par insertion

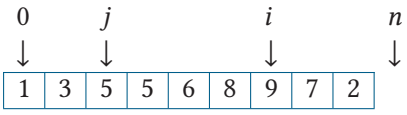
```
void insertion_sort(int a[], int n) {
  for (int i = 1; i < n; i++) {
    // invariant: a[0..i[ est trié
    int v = a[i];
    int j = i;
    while (j > 0 && a[j-1] > v) {
      a[j] = a[j-1];
      j--;
    }
    a[j] = v;
  }
}
```

on obtient donc après avoir trié les six premiers éléments la configuration suivante.

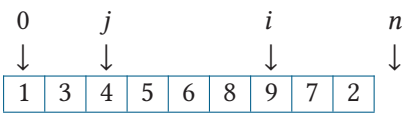
On passe d'une étape à la suivante en *insérant* le prochain élément à la bonne place dans le préfixe déjà trié. Ce faisant, on décale au besoin les éléments qui doivent se trouver à sa droite. En l'occurrence, le prochain élément à insérer est 4. La fonction le mémorise, puis décale d'un cran tous les éléments du préfixe qui sont plus grands que 4, en partant de la droite. On copie donc d'abord vers la droite les valeurs 9, puis 8,



puis 6, et enfin 5.



Ne reste alors plus qu'à enregistrer 4 dans la case libérée par le décalage de 5 et on obtient un nouveau segment initial trié, un peu plus long que le précédent.



Invariants. Le tri pas insertion utilisant deux boucles imbriquées, nous allons avoir des invariants pour la boucle externe **for** *et* pour la boucle interne **while**. Détaillons cette analyse en partant de l'extérieur.

Le principe de la boucle externe **for** est le suivant : le segment à gauche de l'indice i est trié. On en extrait un invariant :

1. pour tous indices $k_1, k_2 \in [0, i[$ tels que $k_1 < k_2$ on a $a[k_1] \leqslant a[k_2]$.

Cette première propriété est vraie au début du premier tour de boucle : la variable i vaut 1 et le segment $[0, 1[$ de longueur 1 est évidemment trié.

Pour montrer que la propriété est préservée, il faut détailler le corps de cette première boucle, et en particulier analyser la boucle interne **while**. Cette boucle interne décale d'une case vers la droite tous les éléments du segment $[0, i[$ qui sont plus grands que v, afin que l'on puisse ensuite insérer v dans la case libérée juste devant le sous-segment décalé. On a, à chaque étape de cette boucle interne, deux segments intéressants $[0, j[$ et $]j, i]$, avec un indice j tel que $j \in [0, i]$. Ils sont tous deux triés, tous les éléments du premier sont inférieurs à tous les éléments du second, et v est également inférieur à tous les éléments du second. On résume ces faits par trois invariants :

2. l'indice j est tel que $0 \leqslant j \leqslant i$,
3. le segment $[0, i]$ est trié, si l'on ignore l'indice j,
4. pour tout indice $k \in]j, i]$ on a $v < a[k]$.

Préservation des invariants de la boucle interne. On veut maintenant montrer que les propriétés 2, 3 et 4 sont bien des invariants de la boucle interne **while**. Cette boucle est à l'intérieur de la boucle externe **for** : chaque exécution de la boucle **while** se fait donc à l'intérieur d'un tour de la boucle **for**. Considérons donc un tour arbitraire de la boucle externe **for**, pour une certaine valeur i, et supposons que l'invariant 1 est vérifié au début de ce tour. Sous cette hypothèse, on étudie les propriétés 2, 3 et 4.

- La propriété 2 est vraie au début de la boucle interne car j est initialisée à i.
- La propriété 3 est vraie au début de la boucle interne car, quand $j = i$ la propriété 3 est équivalente à la propriété 1, qu'on a supposée vraie.
- La propriété 4 est vraie au début de la boucle interne car, quand $j = i$ on a $]j, i] = \emptyset$.

On montre alors que les trois propriétés 2, 3 et 4 sont préservées par chaque tour de la boucle interne. On considère un tour de la boucle interne au début duquel 2, 3 et 4 sont vérifiées. On note j' et a' la valeur de j et l'état de a à la fin du tour.

- Au début du tour $0 < j \leqslant i$ et pendant le tour j décroît de 1. Donc à la fin $0 \leqslant j' \leqslant i$.

- À la fin du tour, pour tout $k \in [0, j'[, a'[k] = a[k]$ et pour tout $k \in]j'+1, i], a'[k] = a[k]$. En outre $a'[j'+1] = a[j']$. Soient $k_1, k_2 \in [0, i]$ tels que $k_1 < k_2$, $k_1 \neq j'$ et $k_2 \neq j'$.
  - ◇ Si $k_1 \neq j'+1$ et $k_2 \neq j'+1$ alors $a'[k_1] = a[k_1] \leqslant a[k_2] = a'[k_2]$.
  - ◇ Si $k_1 = j'+1$ alors $a'[k_1] = a[j-1] \leqslant a[k_2] = a'[k_2]$.
  - ◇ Si $k_2 = j'+1$ alors $a'[k_1] = a[k_1] \leqslant a[j-1] = a'[k_2]$.

  Dans tous les cas j' et a' vérifient la propriété 3.
- À la fin du tour, pour tout $k \in]j'+1, i] =]j, i]$ on a $a'[k] = a[k]$ et donc $v < a'[k]$. En outre, $v < a[j-1] = a'[j] = a'[j'+1]$. Donc la propriété 4 est bien préservée.

Conclusion de la preuve de correction. On a maintenant la garantie que les propriétés 2, 3 et 4 sont des invariants de la boucle interne, et sont en particulier vérifiées à la fin de l'exécution de cette boucle. En outre, à la fin de cette boucle on a soit $j = 0$ soit $a'[j-1] \leqslant v$. Avec les propriétés 3 et 4, on déduit que le segment $[0, i]$ du tableau modifié a' est bien trié. On en déduit que la propriété 1 a bien été préservée, et est un invariant de la boucle externe. On en déduit encore que la propriété 1 est encore vraie après le dernier tour de la boucle **for**, et donc que le tableau final est bien trié.

Cependant, la preuve n'est pas finie ! Nous avons justifié que le tableau obtenu à la fin était bien trié, mais il faut également assurer que ce tableau final est une permutation du tableau d'origine. Pour cela, on étend les invariants 1 et 3 avec les précisions suivantes, où l'on note a_0 l'état d'origine du tableau et a son état courant :

1'. le segment $[0, i[$ du tableau a est une permutation du segment $[0, i[$ du tableau a_0,

3'. les segments $[0, j[$ et $]j, i]$ du tableau a sont une permutation du segment $[0, i[$ du tableau a_0.

On conclut en vérifiant que ces invariants sont bien préservés par les boucles du programme, qui ne font bien que déplacer les éléments de a sans jamais en perdre.

6.1.4.2 Tri rapide

Les boucles et la récursion *ne sont pas* deux techniques antagonistes. Il est tout à fait possible d'utiliser les deux à l'intérieur d'une même fonction. On peut observer ceci par exemple avec l'algorithme de *tri rapide*, pour l'analyse duquel nous allons devoir combiner les techniques des sections précédentes.

Spécifications, invariants et commentaires

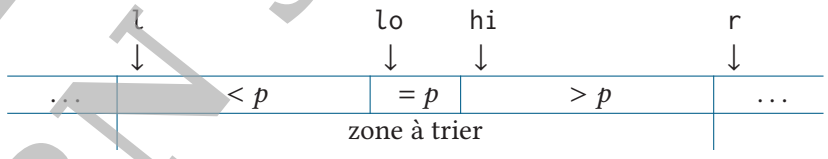
On présente dans ce chapitre les notions de spécification et d'invariant en tant qu'outils de raisonnement sur les algorithmes et les programmes. Cependant ces concepts ont une utilité bien plus large, et sont avant tout un excellent moyen d'expliquer ou de comprendre une fonction ou un algorithme. En résumé :

- la spécification d'une fonction décrit précisément la manière dont celle-ci doit être utilisée, et les résultats produits,
- les invariants d'une boucle éclairent le rôle joué par les différentes variables, et la manière dont l'algorithme fonctionne.

Ainsi, spécifications et invariants expliquent des éléments clés d'un programme qui ne sont pas forcément transparents à la lecture du code. À ce titre, ils font partie des commentaires les plus intéressants que l'on puisse inclure dans le code d'un programme ! Remarquez d'ailleurs que notre programme 6.6 contient bien un commentaire de cette nature.

Présentation de l'algorithme. Le cœur de l'algorithme de tri rapide est le suivant : après avoir choisi un élément « pivot » on trie séparément les éléments inférieurs au pivot et les éléments supérieurs au pivot. Dans le code C du programme 6.7 la fonction principale `quickrec` trie le segment `[l, r[` du tableau `a`, c'est-à-dire entre les indices `l` (inclus) et `r` (exclu).

Le pivot est `a[l]`, le premier élément du segment à trier. La première étape consiste à réarranger les éléments en trois groupes : à gauche les éléments plus petits que le pivot, à droite les éléments plus grands, et au milieu le pivot et les éventuels autres éléments qui lui seraient égaux.



L'algorithme conclut alors en triant récursivement, et surtout séparément, les groupes gauche et droit.

Le réarrangement en trois groupes est opéré par la boucle **`for`**. Durant l'exécution de cette boucle les trois groupes se forment progressivement avec les éléments d'un quatrième groupe : celui des éléments non encore répartis. Ce quatrième

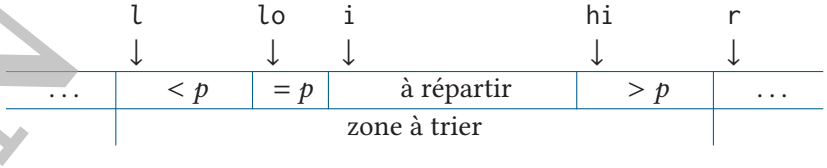
Programme 6.7 – tri rapide d'un tableau

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

// trie uniquement a[l..r[
void quickrec(int a[], int l, int r) {
  if (r - l <= 1) return;
  int p = a[l], lo = l, hi = r;
  for (int i = l+1; i < hi; ) {
    if      (a[i] == p)  { i++; }
    else if (a[i] <  p)  { swap(a, i++, lo++); }
    else  /* a[i] >  p */ { swap(a, i, --hi); }
  }
  quickrec(a, l, lo);
  quickrec(a, hi, r);
}

void quicksort(int a[], int n) {
  knuth_shuffle(a, n);
  quickrec(a, 0, n);
}
```

groupe est situé dans le tableau entre le groupe des éléments égaux au pivot et celui des éléments plus grands que le pivot.



La boucle progresse en consultant le premier élément de la zone à répartir, et en l'intervertissant au besoin avec un autre élément pour le placer dans le bon groupe. Le segment [i, hi[des éléments à répartir diminue à chaque tour, soit par incrément de i soit par décrément de hi.

Spécification. La fonction `quickrec` trie en place le segment $[l, r[$ du tableau a. Sa spécification adapte celle déjà vue pour le tri en place d'un tableau, en tenant compte de l'intervalle cible.

- Précondition : les indices l et r définissent un intervalle valide du tableau a. C'est-à-dire : $0 \leqslant l \leqslant r \leqslant n$.
- L'état a' du tableau après action de `quickrec`(a, l, r) est tel que :
  - le segment $[l, r[$ de a' est trié,
  - le segment $[l, r[$ de a' est une permutation du segment $[l, r[$ de a.

Ossature de la preuve de correction. La fonction `quickrec` trie un segment $[l, r[$ d'un tableau a, à l'aide d'appels récursifs sur des segments de tableau plus petits. On justifie sa correction par une récurrence forte sur la longueur du segment trié.

Considérons un appel `quickrec`(a, l, r). Supposons que `quickrec` est correcte sur tout segment de longueur strictement inférieure à $r - l$, et raisonnons par cas sur la longueur $r - l$ du segment à trier.

- Si la longueur $r-l$ est 0 ou 1 la fonction s'arrête immédiatement, et le segment ciblé est bien déjà trié.
- Si $r - l > 1$ la fonction suit les étapes suivantes.
  1. La boucle **`while`** permute les éléments du segment $[l, r[$. Supposons que la permutation obtenue en sortie de cette boucle a les propriétés suivantes :
     - tous les éléments du segment $[l, lo[$ sont strictement inférieurs à $a[lo]$,
     - tous les éléments du segment $[lo, hi[$ sont égaux à $a[lo]$,
     - tous les éléments du segment $[hi, r[$ sont strictement supérieurs à $a[lo]$,
     - les indices lo et hi vérifient $l \leqslant lo < hi \leqslant r$.

     On justifiera ces propriétés dans l'analyse de la boucle elle-même.
  2. Trier récursivement le segment $[l, lo[$. Comme $lo < hi \leqslant r$, le segment $[l, lo[$ est un segment valide de a et sa longueur $lo - l$ est strictement inférieure à $r-l$. Par hypothèse de récurrence, cet appel trie correctement le segment $[l, lo[$.
  3. Trier récursivement le segment $[hi, r[$. De même que ci-dessus il s'agit d'un segment valide et strictement plus court que $[l, r[$: par hypothèse de récurrence, il trie correctement le segment $[hi, r[$.

Après ces trois étapes, on obtient bien une permutation du segment $[l, r[$ du tableau d'origine. Il ne reste qu'à justifier que cette permutation est triée. Soient donc $i, j \in [l, r[$ tels que $i < j$. On veut montrer que $a[i] \leqslant a[j]$. Raisonnons par cas sur les segments auxquels appartiennent les indices i et j.

- Si $i, j \in [l, lo[$ ou $i, j \in [hi, r[$ on a bien $\texttt{a[}i\texttt{]} \leqslant \texttt{a[}j\texttt{]}$, car on a déjà justifié que chacun de ces deux segments était trié.
- Dans tous les autres cas, on fait une comparaison intermédiaire avec $\texttt{a[}lo\texttt{]}$.
  - Si $i, j \in [lo, hi[$ alors $\texttt{a[}i\texttt{]} = \texttt{a[}lo\texttt{]} = \texttt{a[}j\texttt{]}$.
  - Si $i \in [l, lo[$ et $j \in [lo, hi[$ alors $\texttt{a[}i\texttt{]} < \texttt{a[}lo\texttt{]} = \texttt{a[}j\texttt{]}$.
  - Si $i \in [lo, hi[$ et $j \in [hi, r[$ alors $\texttt{a[}i\texttt{]} = \texttt{a[}lo\texttt{]} < \texttt{a[}j\texttt{]}$.
  - Si $i \in [l, lo[$ et $j \in [hi, r[$ alors $\texttt{a[}i\texttt{]} < \texttt{a[}lo\texttt{]} < \texttt{a[}j\texttt{]}$.

Par principe de récurrence forte, la fonction `quickrec` trie donc bien correctement le segment cible $[l, r[$.

Analyse de la boucle de tripartition. On complète l'analyse de la fonction `quickrec` de notre tri rapide en fournissant des invariants pour la boucle **`while`**. Cette boucle permute les éléments du segment $[l, r[$ pour les réorganiser en trois groupes en fonction d'un élément *pivot* :

- dans le segment $[l, lo[$, les éléments plus petits que le pivot,
- dans le segment $[lo, hi[$, les éléments égaux au pivot,
- dans le segment $[hi, r[$, les éléments plus grands que le pivot.

Le pivot est l'élément présent à l'origine à l'indice l. Au cours de l'exécution de la boucle, le segment $[l, r[$ est découpé en quatre partie : trois segments avec des éléments déjà répartis en fonction de leur relation au pivot, et un segment d'éléments qui n'ont pas encore été considérés.

l ↓	lo ↓	i ↓	hi ↓	r ↓
$\cdots$	$<$ pivot	$=$ pivot	non répartis	$>$ pivot	$\cdots$

Ce schéma contient un certain nombre de sous-entendus, notamment le fait que les indices délimitant les segments apparaissent dans le bon ordre et le fait que le segment $[lo, i[$ n'est pas vide, puisqu'il contient au moins un exemplaire du pivot p à l'indice lo. Avec ces précisions, on exprime les invariants ainsi :

1. $l \leqslant lo < i \leqslant hi \leqslant r$,
2. pour tout $k \in [l, lo[$ on a $a[k] < p$,

3. pour tout $k \in [lo, i[$ on a $a[k] = p$,
4. pour tout $k \in [hi, r[$ on a $a[k] > p$.

À cela, il faut ajouter une indication selon laquelle la boucle ne fait bien que réarranger les éléments qui étaient présents à l'origine dans le segment $[l, r[$ du tableau `a`. Pour exprimer cela, notons a_0 l'état d'origine du tableau `a` et a son état courant :

5. le segment $[l, r[$ du tableau a est une permutation du segment $[l, r[$ du tableau a_0.

Justifions que ces cinq propriétés sont bien des invariants de la boucle **`while`**.

Initialisation. On se place dans le cadre d'un appel `quickrec`(a, l, r). On suppose que les préconditions soient vérifiées, c'est-à-dire que $[l, r[$ est bien un segment valide du tableau a, et que la boucle **`while`** s'apprête à être exécutée, c'est-à-dire que le programme n'a pas été interrompu après le test $r - l \leqslant 1$. On a donc $l + 1 < r$. À l'initialisation, on a $lo = l$, $hi = r$ et $i = l+1$, ce qui suffit à justifier la propriété 1. Les segments $[l, lo[$ et $[hi, r[$ sont vides, ce qui trivialise les propriétés 2 et 4. Enfin, le segment $[lo, i[= [l, l + 1[$ contient l'unique indice l et on a bien $p = a[l]$ puisque p a été initialisé ainsi. En outre, le tableau est toujours dans l'état origine a_0, qui est bien une permutation triviale de lui-même.

Préservation. Considérons qu'au début d'un tour de boucle, le tableau `a` et les variables `p`, `l`, `r`, `lo`, `hi` et `i` ont des valeurs a, p, l, r, lo, hi et i validant les cinq propriétés. Notons a', lo', hi' et i' les valeurs telles que modifiées par le tour de boucle (les variables `p`, `l` et `r` ne sont pas modifiées et chacune conserve sa valeur). On raisonne par cas sur la comparaison entre $a[i]$ et p.

- Si $a[i] = p$ alors $a' = a$ et
$$\begin{cases} lo' &= lo \\ i' &= i + 1 \\ hi' &= hi \end{cases}$$
Le tableau `a` et les variables `lo` et `hi` n'ayant pas été modifiés les propriétés 2, 4 et 5 sont immédiatement préservées. Comme on avait $i < hi$ on a bien $i' = i + 1 \leqslant hi$ et la propriété 1 est également préservée. Enfin, si on prend $k \in [lo, i'[= [lo, i + 1[$ on a deux possibilités :
  - soit $k \in [lo, i[$, et par hypothèse 3 on a $a[k] = p$,
  - soit $k = i$, et on est justement dans le cas où $a[i] = p$.
- Si $a[i] < p$ alors
$$\begin{cases} lo' &= lo + 1 \\ i' &= i + 1 \\ hi' &= hi \end{cases} \quad \text{et} \quad \begin{cases} a'[i] &= a[lo] \\ a'[lo] &= a[i] \\ a'[k] &= a[k] \end{cases} \text{ pour } k \neq i \text{ et } k \neq lo$$

La propriété 1 est préservée, puisque `lo` et `i` ont été incrémentées conjointement et qu'on avait $i < hi$. La propriété 5 est préservée puisque la modification de `a` est une simple permutation de deux éléments du segment $[l, r[$. La propriété 4 est préservée puisque `hi` n'a pas été modifiée, ni le segment $[hi, r[$ du tableau `a`. Pour la propriété 2, prenons $k \in [l, lo'[= [l, lo + 1[$. On a deux possibilités :

- ◇ soit $k \in [l, lo[$, et alors $a'[k] = a[k] < p$,
- ◇ soit $k = lo$, et alors $a'[k] = a'[lo] = a[i] < p$.

Pour la propriété 3, prenons $k \in [lo', i'[= [lo + 1, i + 1[$. On a deux possibilités :

- ◇ soit $k \in [lo + 1, i[$, et alors $a'[k] = a[k] = p$,
- ◇ soit $k = i$, et alors $a'[k] = a'[i] = a[lo] = p$.

◆ Si $a[i] > p$ alors

$$\begin{cases} lo' &= lo \\ i' &= i \\ hi' &= hi - 1 \end{cases} \quad \text{et} \quad \begin{cases} a'[i] &= a[hi - 1] \\ a'[hi - 1] &= a[i] \\ a'[k] &= a[k] \quad \text{pour } k \neq i \text{ et } k \neq hi - 1 \end{cases}$$

La propriété 1 est préservée, puisque seule `hi` a été décrémentée et qu'on avait $i < hi$. La propriété 5 est préservée puisque la modification de `a` est une simple permutation de deux éléments du segment $[l, r[$. Les propriétés 2 et 3 sont préservées puisque `lo` et `i` n'ont pas été modifiées, ni les segments $[l, lo[$ et $[lo, i[$ du tableau `a`. Pour la propriété 5, prenons $k \in [hi', r[= [hi - 1, r[$. On a deux possibilités :

- ◇ soit $k \in [hi, r[$, et alors $a'[k] = a[k] > p$,
- ◇ soit $k = hi - 1$, et alors $a'[k] = a'[hi - 1] = a[i] > p$.

Finalement, dans tous les cas les cinq propriétés sont préservées : il s'agit bien d'invariants. On en déduit qu'à la fin de l'exécution de la boucle ces cinq propriétés sont toujours vérifiées. En combinant avec la condition d'arrêt $i \geqslant hi$ de la boucle, qui donne en particulier $i = hi$, on conclut bien que :

- ◆ les indices lo et hi vérifient $l \leqslant lo < hi \leqslant r$,
- ◆ tous les éléments du segment $[l, lo[$ sont strictement inférieurs à $a[lo]$,
- ◆ tous les éléments du segment $[lo, hi[$ sont égaux à $a[lo]$,
- ◆ tous les éléments du segment $[hi, r[$ sont strictement supérieurs à $a[lo]$.

Cela justifie l'hypothèse qui avait été faite dans l'analyse récursive de `quickrec` et conclut la démonstration de correction de cette fonction : `quickrec` trie le segment de tableau `a` $[l, r[$.

Conclusion. On peut maintenant que conclure que le tri rapide lui-même, réalisé par la fonction `quicksort`, trie correctement le tableau passé en paramètre, puisqu'il applique `quickrec` sur l'intégralité du tableau. Notez que `quicksort` réalise au préalable un mélange du tableau avec la fonction `knuth_shuffle`, détaillée dans un exercice. Ce mélange ne fait que permuter les éléments de manière aléatoire et ne perturbe donc pas la correction du tri. Quant à l'intérêt de mélanger un tableau avant de le trier... rendez-vous à la section 6.3.6.

Exercice 26 p.155

Que trier ?

Dans cette section, nous trions des tableaux contenant des données les plus simples possibles : des entiers. On se concentre ainsi sur le cœur des algorithmes. Tous les algorithmes vus ici sont cependant adaptables à des données plus riches. On pourrait par exemple imaginer le cas d'un tableau contenant des structures de la forme suivante, combinant une clé entière utilisée pour le tri, et des données à proprement parler.

```
struct Data {
  int key;
  ??? contents;
};
```

Les comparaisons d'éléments seraient alors à faire via le champ `key`. Dans une telle situation, la gestion des éléments « égaux » que nous avons vu apparaître dans la boucle de tripartition prend tout son sens : on peut se retrouver avec plusieurs structures partageant une même clé, mais contenant des données différentes. Il est alors important de bien traiter chacune indépendamment des autres.

Notez que lorsque plusieurs éléments partagent une même clé, ils apparaissent côte-à-côte dans le tableau trié mais peuvent apparaître dans un ordre arbitraire (et notre tripartition est effectivement susceptible de les mélanger). Les algorithmes de tri qui préservent l'ordre relatif des éléments partageant une même clé sont appelés des *tris stables.*

Exercice 46 p.309

6.1.4.3 Tri fusion

En programmation, on recommande souvent de découper le problème à résoudre en plusieurs tâches bien délimitées et indépendantes. Ceci facilite le développement : chaque tâche est alors résolue par une fonction dédiée, que l'on peut espérer écrire indépendamment des autres. Ces bienfaits de la modularité se retrouvent jusque dans l'analyse des programmes.

Lorsqu'une fonction principale utilise des fonctions auxiliaires, chaque fonction peut être analysée séparément des autres. Pour démontrer la correction de la fonction principale, nous n'avons besoin que de la spécification des fonctions auxiliaires, pas de leur détail ! Voyons cela avec l'algorithme de *tri par fusion.*

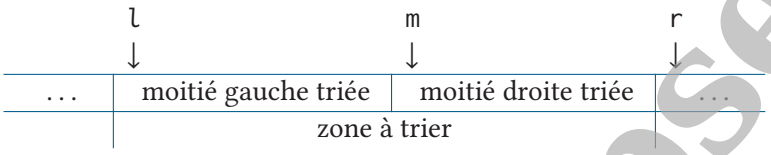
Programme 6.8 – tri fusion récursif de tableaux

```
void merge(int a1[], int a2[], int l, int m, int r) {
  int i = l, j = m;
  for (int k = l; k < r; k++)
    if (i < m && (j == r || a1[i] <= a1[j]))
      a2[k] = a1[i++];
    else
      a2[k] = a1[j++];
}

void mergesortrec(int a[], int tmp[], int l, int r) {
  if (r - l <= 1) return;
  int m = l + (r - l) / 2;
  mergesortrec(a, tmp, l, m);
  mergesortrec(a, tmp, m, r);
  if (a[m-1] <= a[m]) return; // optimisation
  for (int i = l; i < r; i++) tmp[i] = a[i];
  merge(tmp, a, l, m, r);
}

void mergesort(int a[], int n) {
  int *tmp = calloc(n, sizeof(int));
  mergesortrec(a, tmp, 0, n);
  free(tmp);
}
```

Présentation de l'algorithme. L'algorithme de tri par fusion réarrange les éléments d'un tableau de sorte à les trier par ordre croissant. Il procède en triant d'abord séparément la moitié gauche et la moitié droite du tableau, avant d'entrelacer les deux demi-tableaux triés. Dans le code C du programme 6.8, comme dans le cas précédent du tri rapide, on a une fonction principale `mergesortrec` qui trie un segment `[l, r[` du tableau `a`. Pour trier chaque moitié, on applique récursivement l'algorithme. On obtient un tableau formé de deux moitiés indépendantes, toutes deux triées.



La fusion des deux moitiés triées en un unique segment trié ne peut pas être réalisée de manière satisfaisante en ne faisant que des permutations d'éléments à l'intérieur du tableau `a` (ce qu'on avait par exemple avec la boucle de répartition du tri rapide). À la place, on copie intégralement les deux moitiés triées dans un tableau auxiliaire `tmp`, puis on réécrit directement dans `a` le résultat de leur fusion. Cet entrelacement est confié à la fonction dédiée `merge`.

Dans le code du programme 6.8, pour éviter d'allouer un nouveau tableau auxiliaire à chaque appel récursif, on crée un tableau `tmp` une fois pour toutes dans la fonction `mergesort`, puis on passe ce tableau en paramètre à `mergesortrec`. Notez que la fonction `mergesortrec` ne dépend aucunement du contenu d'origine de `tmp` : elle ne fait que l'utiliser comme un espace annexe où elle écrit elle-même le contenu dont elle a besoin pour réaliser ensuite une fusion.

Spécification de la fonction auxiliaire. La fonction `merge` prend en paramètres un tableau d'origine a_1 et un tableau de destination a_2, avec nécessairement a_1 différent de a_2, ainsi que trois indices l, m et r avec $l \leqslant m \leqslant r$ et tels que $[l, r[$ est un intervalle valide dans les deux tableaux. La fonction suppose en outre que les deux segments $a_1\,[l, m[$ et $a_1\,[m, r[$ sont triés. Elle place alors la fusion triée des segments $[l, m[$ et $[m, r[$ du tableau a_1 dans le segment $[l, r[$ du tableau a_2.

Exercice 45 p.309

On peut étudier cette fonction pour elle-même, énoncer et justifier les invariants de sa boucle **`for`**, et finalement démontrer sa correction. Cependant en ce qui concerne l'étude du tri par fusion, le détail de cette analyse importe peu. Pour raisonner sur la correction de `mergesortrec` ou de toute autre fonction utilisant `merge` il suffit de connaître la spécification de `merge`, puisque celle-ci décrit précisément ce qui peut être attendu d'elle.

Correction de la fonction principale. La fonction principale `mergesortrec` utilise des appels récursifs, sur des segments de taille moitié. On peut opter pour un raisonnement par récurrence forte.

Considérons donc un entier $n \in \mathbb{N}$, et supposons que `mergesortrec` trie correctement tout segment de tableau de longueur $k < n$. On veut justifier qu'elle trie correctement de même les segments de longueur n. On raisonne par cas sur n.

- Les cas $n = 0$ et $n = 1$ sont immédiats : un segment de longueur 0 ou 1 est déjà trié et `mergesortrec` a raison de ne rien faire dans ce cas.

- Si n vaut au moins 2, on a deux appels récursifs sur des segments gauche et droit de longueurs respectives $\lfloor \frac{n}{2} \rfloor$ et $\lceil \frac{n}{2} \rceil$, toutes deux strictement inférieures à n. Par hypothèse de récurrence ces deux appels trient correctement leurs cibles respectives. Alors :
  - soit le dernier élément de la moitié gauche est déjà inférieur au premier élément de la moitié droite, et la fonction peut s'arrêter puisque le tableau est bien trié,
  - soit l'ensemble du segment `[l, r[` est placé dans le tableau auxiliaire `tmp` et on conclut avec un appel à la fonction `merge`.

  Dans le cas de l'appel à `merge` les préconditions de cette fonction sont bien remplies : d'une part `a` et `tmp` sont deux tableaux distincts, et d'autre part les indices `l`, `m` et `r` sont des indices valides et tels que $\texttt{l} \leqslant \texttt{m} \leqslant \texttt{r}$. On en déduit donc immédiatement que l'action de `merge` place dans le segment `[l, r[` le résultat de la fusion triée de des segments `[l, m[` et `[m, r[` de `tmp`, qui sont eux-mêmes des copies des segments correpondants de `a`.

  La fonction `mergesortrec` a donc bien trié le segment `[l, r[` du tableau `a`.

6.2 Terminaison

On s'attend la plupart du temps à ce qu'un programme, après avoir calculé un certain temps, renvoie un résultat ou achève la tâche qui lui avait été confiée, c'est-à-dire que son exécution *se termine*. Certains éléments en revanche ouvrent la possibilité pour l'exécution d'un programme de se poursuivre indéfiniment, sans jamais terminer. C'est le cas par exemple de la boucle **`while`** ou des appels récursifs de fonctions.

```
int i=1;
while (true) {
  printf("%ikm à pied, ça use les souliers\n", i++);
}
```

```
let rec chanson n =
  Printf.printf "%ikm à pied, ça use les souliers\n" n;
  chanson (n+1)
in chanson 1
```

Lors de l'utilisation de boucles ou de fonctions récursives dans un algorithme dont l'exécution est supposée terminer, cet arrêt mérite donc une justification.