Chapitre 4

Analyse d'algorithmes

Option informatique MPSI1 & MPSI2

Nous allons étudier dans ce chapitre les méthodes qui vont permettre d'analyser les programmes écrits et de les valider avant même leur exécution par un ordinateur.
Nous introduirons des algorithmes élémentaires pour trier des collections de valeurs et en donnerons leur analyse.

I Analyse des algorithmes

Lorsque l'on écrit un programme il est nécessaire que celui-ci soit correct.
Le premier outil qui vient à l'esprit est de le tester avec des entrées pour lesquelles on sait, à l'avance, le résultat attendu et de vérifier qu'il est bien obtenu. Cependant cela comporte des incertitudes.

- Si le programme ne passe pas les tests, on sait que quelque chose n'a pas fonctionné mais il est difficile de savoir ce qui pose problème. Est-ce l'algorithme, le système, le compilateur du langage . . . ?
- Si le programme passe les tests on ne sait qu'il fonctionne que dans les cas où on n'a pas vraiment besoin de lui (on a déjà la réponse). Qu'en est-il des nombreux cas non testés ?

Il existe un moyen plus sûr qui ne concerne que l'algorithme : c'est une analyse a-priori de celui-ci. Cette analyse est un travail théorique antérieur à l'exécution sur une machine. Cette démarche est très exigeante mais elle permet d'éviter de nombreux problèmes. C'est cette étude que nous allons ébaucher.
Nous allons analyser un programme en trois étapes.

1. Le programme fournit-il un résultat ? C'est le problème de la **terminaison**
2. Le programme fournit-il le bon résultat ? On doit en donner la **preuve**.
3. Le programme fournit-il le bon résultat dans un temps raisonnable ?
   On en étudie la **complexité**.

Ces trois question sont de plus en plus précises et seront souvent étudiées dans cet ordre. Cependant il y aura souvent des ressemblances entre la première et la troisième question : on pourrait reformuler le problème de la terminaison sous la forme

"*Le programme fournit-il un résultat dans un temps fini ?*"

Un temps fini peut être considéré comme la définition la plus élémentaire d'un temps raisonnable.

I.1 Exemples

Dans cette étude on va appliquer les méthodes aux fonctions suivantes

```
let somme_tableau tab =
  let n = Array.length tab in
  let som = ref 0 in
  for i = 0 to (n-1) do som := !som + tab.(i) done;
  !som;;
```

```
let estPuissance2 n =
  let p = ref n in
  while !p mod 2 = 0 do p := !p/2 done;
  !p = 1;;
```

```
let rec appartient x liste =
  match liste with
  |[] -> false
  |t::q when t = x -> true
  |t::q -> appartient x q;;
```

I.2 Terminaison

Le programme fournit-il un résultat ? C'est-à-dire finit-il ?
On emploie aussi le verbe **terminer** sous une forme intransitive et on parle de terminaison.
Il y a des limites à cette interrogation.

- On peut montrer qu'il n'existe pas d'algorithme permettant de répondre à cette question. En termes techniques le problème de l'arrêt est indécidable.
- Certains programmes n'ont pas besoin de terminer : on peut penser à certains jeux, à la console python, à l'interface windows, mac ou linux

Une réponse simple serait "*on le lance et on voit bien s'il s'arrête*"
Bien que facile à mettre en œuvre cette méthode n'est pas satisfaisante.

- Il se peut que le programme boucle indéfiniment pour certaines valeurs, pas toutes.
  Un test positif ne prouve pas que le programme est correct.
- Il arrive aussi que le temps de calcul soit plus long que le temps d'attente supportable.
  Un test négatif ne prouve pas que le programme est incorrect.

Il faut donc essayer de prouver la terminaison par une étude a-priori.

Un programme est composé d'une suite d'instructions ; elles sont séparées par un point-virgule dans OCaml. Une instruction peut être

- une instruction simple faisant appel à des fonctions de bases ou à des fonctions écrites par l'utilisateur,
- une instruction conditionnelle qui fait appel
  - à une évaluation simple à résultat booléen
  - une ou deux suites d'instructions selon qu'il y a ou non une clause **`else`**
- une suite d'instructions répétée par une boucle **`for`**,
- une suite d'instructions répétée par une boucle **`while`**.

Les instructions de base terminent si les fonctions qu'elles appellent terminent.
La suite d'instructions d'une boucle **`for`** est répétée une nombre de fois prévu à l'avance ; la boucle termine à condition que les instructions du corps terminent.
On remarquera que cette dernière propriété est mise en défaut si l'indice de la boucle est modifié dans les évaluations : **c'est une écriture à proscrire absolument !**

Dans un algorithme on doit donc prouver la terminaison des fonctions appelées, ce qui demande une étude particulière dans le cas des fonctions récursives, et celle des boucles **`while`**.

On peut remarquer que la preuve de terminaison peut être impossible.
La suite de Syracuse de terme initial a est la suite définie par

$$u_0 = a \quad u_{n+1} = \begin{cases} u_n/2 \text{ si } u_n \text{ est pair} \\ 3u_n + 1 \text{ si } u_n \text{ est pair} \end{cases} \text{ pour tout } n \geqslant 1$$

```
let rec syracuse a =
  if a = 1
  then 0
  else if a mod 2 = 0
       then 1 + (syracuse a/2)
       else 1 + (syracuse (3*a+1));;
```

On ne sait pas prouver que ce programme termine pour tout entier a ; ce serait un résultat mathématique (espéré) si on arrivait à le prouver.

Terminaison des boucles `while`

Pour prouver que les tests de sortie d'une boucle **`while`** finissent par être réalisés on peut utiliser un résultat mathématique simple :
toute suite strictement décroissante d'entiers tend vers moins l'infini
sous la forme :
toute toute suite strictement décroissante d'entiers positifs est finie.
Cela induit la définition suivante

Définition 1 : Variant de boucle

Un variant de boucle est une valeur entière dépendant des variables du programme

- qui est positive quand la condition de la boucle est réalisée
- et qui décroît strictement à chaque passage dans la boucle.

Si une boucle admet un variant alors elle termine.

En effet s'il existe un variant de boucle alors il ne peut exister qu'un nombre fini de répétitions [1] donc la boucle termine.

Terminaison des fonctions récursives

Pour prouver la terminaison d'une fonction récursive on emploie la notion d'un variant en introduisant une mesure entière de la taille des paramètres. On prouve alors que la fonction termine par récurrence sur cette mesure.
La mesure d'une donnée peut être :

- la valeur de n (ou le nombre de bit dans sa représentation en base 2) dans le cas d'une variable entière,
- le nombre d'éléments des listes, tableaux, fichiers,
- la longueur d'un mot,
- le degré d'un polynômes,
- le nombre de lignes (ou de colonnes ou leur produit) dans le cas d'une matrice,
- ...

1. On retrouve le fait que les boucles **`for i = a to b`** terminent toujours car $b - i$ est un variant de boucle, **si i n'est pas modifié dans la boucle**.

Terminaison des exemples

1. `somme_tableau` fait appel à des fonctions simples, on admet qu'elles terminent, et à une boucle `for`. Ainsi le programme termine.
2. Dans le cas de la fonction `estPuissance2` la valeur de `!p` est un variant.
3. Dans la fonction `appartient`, on note n la longueur de la liste.
   (a) Pour $n = 0$, le programme termine directement.
   (b) On suppose que la fonction termine quand on l'appelle avec une liste de taille n.
   On considère une liste de taille $n + 1$: son premier terme est t.
   Si t est la valeur recherchée, le programme termine.
   Sinon on appelle la fonction avec le reste de la liste, de taille n, donc termine.
   (c) Ainsi la fonction termine dans tous les cas.

I.3 Preuve

Le programme fournit-il le bon résultat ?
Il existe des outils théoriques (la logique de Hoare) qui permettent une formalisation de l'analyse de ce problème. C'est une étape au delà de ce cours d'introduction mais elle est indispensable quand on veut prouver des programmes complexes.

Dans le cadre de ce cours nous allons donner des outils plus simples. Dans le cas d'une suite d'instructions il faut prouver que chaque instruction fait ce qui est attendu. Dans le cas d'instructions simples, il suffit de vérifier la concordance de l'écriture avec le projet.

Invariants des boucles `for`

Dans la fonction `somme_tableau` on voit facilement que la valeur de `som` au début de la boucle pour l'indice i est la somme des termes du tableau pour les indices 0 à $i - 1$.

1. Pour $i = 0$ la somme est vide, on lui associé la valeur 0.
2. Si on suppose que la propriété est vraie au début d'un passage de boucle pour i on ajoute `tab.(i)` à `som` donc la propriété est vraie au passage suivant, quand l'indice vaut $i + 1$.
3. Lors du passage de la dernière boucle la propriété est vraie en sortie pour $i = n$ ce qui prouve que la fonction renvoie la somme des termes.

On a donné un **invariant** pour la boucle.

Définition 2 : Invariant de boucle `for`

Un invariant d'une boucle `for i = a to b` est une propriété $P(i)$ dépendant des variables et de l'indice i de la boucle telle que

- $P(a)$ est vraie avant le premier passage de la boucle,
- si $P(i)$ est vraie avec $i \leqslant b$ alors $P(i+1)$ est vérifiée après l'exécution des instructions de la boucle pour l'indice i,
- $P(b + 1)$ prouve le résultat attendu.

Si un boucle admet un invariant alors elle est prouvée.

On prouve par récurrence que $P(i)$ est vraie pour tout i entre a et $b+1$ d'où la preuve du résultat attendu.

Invariants des boucles `while`

On peut généraliser la définition pour les boucles `while`.

Définition 3 : Invariant de boucle `while`

Un invariant d'une boucle est une propriété P dépendant des variables telle que

- P est vraie à l'initialisation
- si P est vérifiée au début de la suite des instructions de la boucle alors elle est vraie au début de l'itération suivante
- P et la condition de fin prouvent le résultat attendu.

La principale difficulté sera souvent de trouver l'invariant : en fait rechercher un invariant peut être une étape utile lors de la recherche d'un algorithme.

Dans le cas de la fonction `estPuissance2` un invariant possible est

$$\mathcal{P} : \text{il existe un entier positif } k \text{ tel que } n = 2^k.p$$

1. $\mathcal{P}$ est valide à l'initialisation : $n = p = 2^0.p$.
2. Si $\mathcal{P}$ est vérifiée, $n = 2^k.p$ et si la condition de la boucle est vérifiée alors on divise p par 2, $p' = \frac{p}{2}$ et on a bien $n = 2^{k+1}.p'$.
3. Si $\mathcal{P}$ est vérifiée, $n = 2^k.p$ et si la condition de la boucle ne l'est pas alors p n'est pas divisible par 2 donc p vaut 1 si et seulement si n est une puissance de 2, ce qui est le résultat attendu.

Fonctions récursives

Dans le cas des fonctions récursives l'invariant consiste à prouver l'algorithme par récurrence sur la taille de l'entrée.

Dans la fonction `appartient`, on note n la longueur de la liste.

1. Pour $n = 0$, la fonction renvoie `false`, ce qui est le bon résultat car x ne peut pas appartenir à une liste vide.
2. On suppose que la fonction renvoie le bon résultat quand on l'appelle avec une liste de taille n. Pour une liste de taille $n + 1$:
   - si son premier terme est x elle doit renvoyer `true`, ce qui est le cas,
   - sinon x appartient à la liste si et seulement s'il appartient au reste, ce qui assuré avec l'hypothèse de récurrence.
3. Ainsi la fonction renvoie le bon résultat dans tous les cas.

I.4 Complexité

Le programme fournit-il le bon résultat dans un temps raisonnable ?
Le but ici n'est pas de prévoir le temps exact que va demander la résolution d'un problème mais d'avoir une idée de l'accroissement du temps quand la taille des données d'entrée augmente.
Pour cela on peut compter le temps utilisé en nombre de cycles du processeur.
Cependant il est souvent difficile de descendre à un niveau aussi bas d'instructions : on compte plutôt le nombre d'opérations "élémentaires" dans le langage choisi.
Même cela sera souvent inutilement compliqué. Le plus souvent on choisit une instruction élémentaire signifiante et on compte le nombre d'exécutions de cette instruction. Ce pourra être le nombre de comparaisons, d'affectation, de produits, ...
Ce que l'on cherche c'est prévoir comment évolue le temps d'exécution lors que les données d'entrée ont une mesure qui augmente.
Parfois l'entrée sera caractérisée par plusieurs mesures (lorsqu'il y a plusieurs listes en paramètres, dans le cas d'une matrice, ...).

On aboutit à une fonction de complexité dont la variable est une mesure de la donnée d'entrée. Cependant la complexité peut varier selon les différentes entrées possible d'une même taille. Nous choisirons souvent de calculer la complexité maximale [2].
On cherche donc une fonction $C(n)$ telle que, pour toutes les données d'entrée de taille n, le nombre d'instruction effectuées est majoré par $C(n)$.

1. Pour la fonction `somme_tableau` le nombre d'additions est la taille du tableau : on peut écrire $C(n) = n$.
2. Pour la fonction `estPuissance2` le nombre de divisions est l'exposant k de 2 dans la décomposition en facteurs premiers de n : on a $2^k \leqslant n$ donc $C(n) \leqslant \log_2(n)$.
3. Pour la fonction `appartient` on montre par récurrence sur la longueur n de la liste que le nombre de comparaisons vérifie $C(n) \leqslant n$.

Cette fonction $C(n)$ est en général inutilement précise.
La question qu'on se pose est, on y revient, l'évolution avec la taille n.

Que devient le temps de calcul si on multiplie la taille des données par 2 ?

On utilise la notation de Landau, $C(n) = \mathcal{O}\big(g(n)\big)$ qui signifie qu'il existe une constante K telle que $C(n) \leqslant Kg(n)$ pour tout n (ou pour tout $n \geqslant n_0$).
Le plus souvent g sera une fonction très simple de la forme $g(n) = n^\alpha$ ou $g(n) = a^n$.

- On parle de complexité linéaire quand la complexité est un $\mathcal{O}(n)$,
- On parle de complexité quadratique quand elle est un $\mathcal{O}(n^2)$,
- On parle de complexité polynomiale quand elle est un $\mathcal{O}(n^p)$,
- On parle de complexité quasi-constante quand elle est un $\mathcal{O}\big(\log_2(n)\big)$,
- On parle de complexité quasi-linéaire quand elle est un $\mathcal{O}\big(n\log_2(n)\big)$.
- On parle de complexité exponentielle quand elle est un $\mathcal{O}(a^n)$,

2. Mais on peut aussi déterminer la complexité en moyenne.

II Tris simples de tableaux

II.1 Qu'est-ce que trier ?

On ne trie que des collections d'éléments comparables : il existe une relation d'ordre total sur les éléments. Les cas simples sont ceux de listes d'entiers ou de flottants mais il arrivera souvent que les éléments soient moins directement comparables. On supposera donnée une fonction `plusGrand` telle que `plusGrand x y` renvoie `true` si et seulement si $x > y$.
Le cas simple est `let plusGrand x y = x > y;;`
Dans le cas de tableaux il y a deux résultats possibles :

- on peut modifier le tableau *en place*, c'est-à-dire en permutant les éléments du tableau
- on peut renvoyer un nouveau tableau sans modifier le tableau initial.

Dans cette partie on n'écrira que des tris de tableaux en place.

Un outil important dans le cas d'un tri en place est la permutation de deux valeurs dans un tableau.


```
let echanger t i j =
  let temp = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- temp;;
```

II.2 Tri par sélection

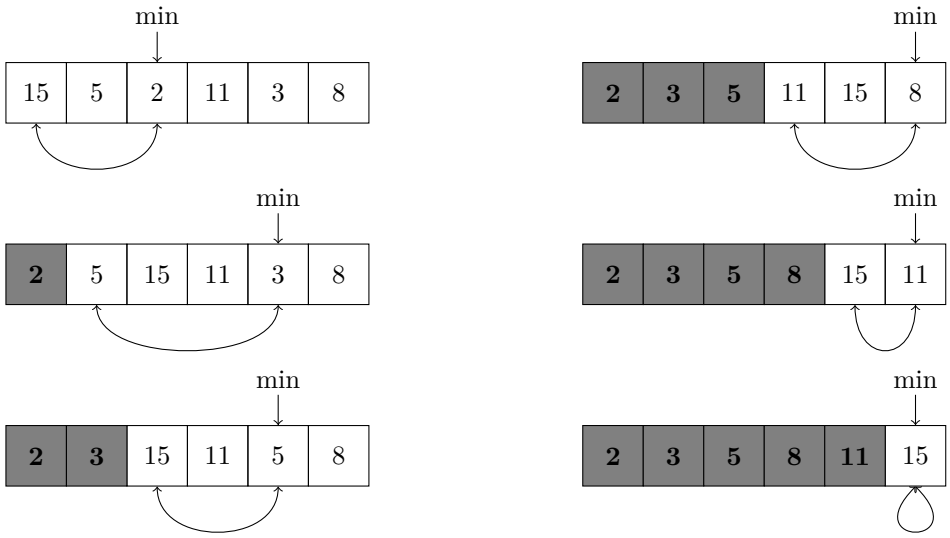
Plutôt que proposer un algorithme sorti de nulle part nous allons essayer de déterminer un moyen d'arriver au résultat en partant d'un invariant de boucle.
On part d'une liste non triée, aucun élément n'est à à sa place, on veut arriver à une liste triée, chaque élément est à sa place. Il semble raisonnable de se fixer comme invariant d'avoir les i premiers éléments (les i plus petits) à leur place c'est-à-dire aux indices 0 à $i-1$, les autres éléments étant à une position quelconque entre les indices i et $n-1$. On veut donc écrire une fonction

```
let trier t =
  let n = Array.length t in
  for i = 0 to (n-1) do
    (* Les i plus petits éléments sont à leur place *)
    instructions
    (* Les i+1 plus petits éléments sont à leur place *) done;;
```

- Pour $i = 0$, la propriété est triviale : aucun élément n'est assuré d'être bien placé.
- Si la propriété est vraie à la fin de la boucle pour $i = n-1$ alors n éléments (c'est-à-dire tous) sont à leur place, la liste est triée.
- Il reste donc à écrire les instruction qui permettent de passer de i à $i+1$ et on aura écrit un algorithme tout en le prouvant.

On part donc d'un tableau commençant par les i plus petits éléments dans l'ordre et on veut parvenir à $i+1$ éléments bien placés. Il suffit simplement de placer le $i+1$-ième à sa place, à l'indice i. Cet élément est le plus petit parmi les $n-i$ encore à placer, c'est donc le minimum des éléments situés entre les indices i et $n-1$ et on doit le placer à l'indice i.

```
let trier t =
  let n = Array.length t in
  for i = 0 to (n-1) do
    (* Les i plus petits éléments sont à leur place *)
    (* calculer j, indice du minimum entre i et n-1 *)
    echanger t i j
    (* Les i+1 plus petits éléments sont à leur place *) done;;
```



On remarque que la dernière opération est inutile : si $n-1$ éléments sont bien placés, le dernier est aussi bien placé.
Il reste donc uniquement à déterminer j, on le fait dans une fonction externe

```
let indiceMinxEntre t a b =
  (* Entrées : un tableau et deux indices
    Sortie : indice d'un minimum pour lae section entre a et b *)
  let i_min = ref a in
  let t_min = ref t.(a) in
  for i = (a+1) to b do
    if plusGrand t_min t.(i)
    then (i_min := i; t_min := t.(i)) done
 !i_max;;
```

Tri par sélection

```
let trier t =
  let n = Array.length t in
  for i = 0 to (n-2) do
    let j = indiceMinEntre t i (n-1) in
    echanger t i j
```

Analyse du tri par sélection d'un tableau

Terminaison Il n'y a que des boucles `for` : l'algorithme termine.

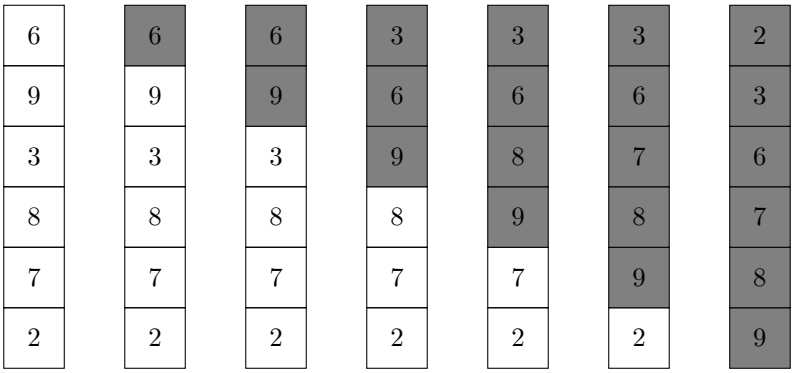
Preuve On l'a construit à partir d'un invariant de boucle ce qui donne la preuve, à condition que `indiceMinEntre` soit prouvée. Dans cette fonction un invariant est
$\mathcal{P}(i)$: pour tout j compris entre a et $i-1$, `t.(j)` $\geqslant$ `t_min` et `t_min` $=$ `t.(!i_min)`.

Complexité $C(n)$ est le nombre d'appels à `plusGrand` pour le tri d'un tableau de taille n.

`indiceMinEntre t a b` effectue $b-a$ comparaisons donc
$C(n) = \sum_{i=0}^{n-2}(n-i-i) = \sum_{k=1}^{n-1} k = \frac{n(n-1)}{2}$: la complexité est quadratique.

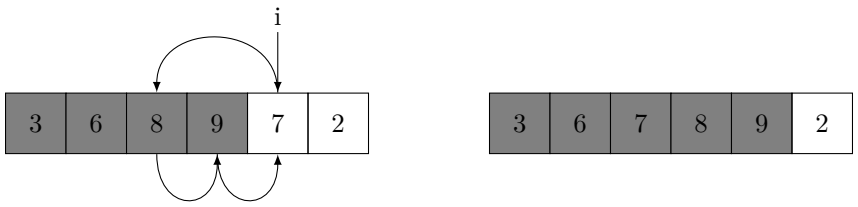
II.3 Tri par insertion

La méthode précédente programmait le tri en construisant pas-à-pas le tableau trié.
On peut aussi partir du tableau à trier et ajouter terme-à-terme les éléments à trier. On aboutit alors à un autre invariant : les i premiers termes sont triés.




```
let trier t =
  let n = Array.length t in
  for i = 0 to (n-1) do
    (* Les i premiers termes de la liste sont triés *)
    placer t i
    (* Les i+1 premiers termes de la liste sont triés *) done;;
```

La fonction `placer` doit donc être une fonction qui reçoit un tableau dont les i premiers éléments sont triés et qui insère l'élément d'indice i de manière à déplacer les éléments plus grands que `t.(i)` et placés avant i vers la droite pour pouvoir placer la valeur de `t.(i)`.




```
let placer t i=
  let j = ref (i-1) in
  while !j >= 0 && plusGrand t.(!j) t.(!j+1) do
    echanger t !j (!j+1);
    decr j done;;
```

Analyse de `placer`

Terminaison `!j` est un entier qui doit être positif et qui décroît strictement dans la boucle, c'est donc un variant et la boucle **`while`** termine.

Preuve Un invariant est formé de l'ensemble $\mathcal{P}(k)$ de propriétés suivant où k est la valeur de `!j`

1. `-1 <= k < i`

2. Les éléments d'indices 0 à i forment une permutation des i premiers éléments du tableau initial et les autres sont inchangés.
3. `t.(0) <= t.(1)<= ... <= t.(k)`
4. `t.(k+1)<= t.(k+2)<= ... <= t.(i)`
5. `t.(k) <= t.(k+2)` pour `0 <= k < i -1`

Exercice 1
Prouver que ces propriétés forment bien un invariant.

Complexité `placer t i` fait au plus i comparaisons.

Analyse du tri par insertion d'un tableau

Terminaison La terminaison de `placer`, la boucle `for` du tri termine.

Preuve On a construit la fonction à partir d'un invariant de boucle ce qui donne la preuve car on a prouvé que `placer` permet le passage de l'invariant.

Cmplexité Si $C(n)$ est le nombre maximum d'appels à `plusGrand` pour le tri d'un tableau de taille n on a $C(n) \leqslant \sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$: la complexité est quadratique.

Exercice 2
Quelle est le nombre de comparaisons dans le cas d'une liste déjà triée et dans le cas d'une liste triée dans l'ordre inverse ?

Exercice 3
Dans la fonction `placer` on fait 3 affectations lors de chaque échange. Or un des éléments qui intervient dans l'échange est toujours le même.
Proposer une fonction qui demande moins d'affectations.

Complexité moyenne du tri par insertion

On suppose qu'à chaque appel de `placer t i`, les positions de `t.(i)` après le placement sont équiprobables. La complexité moyenne de `placer t i` est définie par $\overline{C_p}(i) = \frac{1}{i+1}\sum_{k=0}^{i} n_i(k)$ où $n_i(k)$ est le nombre de comparaisons effectuées dans `placer t i` lorsque la valeur de `t.(i)` finit par être placée en `t.(k)`.

Exercice 4
Calculer $\overline{C_p}(i)$. En déduire la nombre moyen de comparaisons dans le tri.

III Tris simples de listes

Dans le cas de liste, nous allons écrire des algorithmes récursifs, comme le suggère le caractère récursif du type de données.

III.1 Tri par insertion

Le principe le plus simple consiste à trier récursivement la queue de la liste puis à créer une liste triée en adjoignant la tête.

```
let rec tri liste =
  match liste with
  |[] -> []
  |t::q -> inserer t (tri q);;
```

Le cas terminal est celui d'une liste vide dont le tri fournit une liste vide.
Dans le cas général on prouve qua fonction est correcte par récurrence ; il faut donc écrire une fonction `inserer t l` qui donnera une liste triée contenant les éléments de `l` avec `t` en plus.

```
let rec insertion x liste =
  match liste with
  |[] -> [x]
  |t::q when plusGrand t x -> x::liste
  |t::q -> t::(insertion x q);;
```

Analyse de `insertion`

Terminaison L'appel récursif porte sur une liste de taille diminuée de 1 et la fonction termine pour les listes de taille 0. On en déduit par récurrence sur la taille que l'application termine pour toute liste.

Preuve On prouve, toujours par récurrence sur la taille, que si `liste` est triée, alors `insertion x liste` est triée.

- Si `liste` est vide, la fonction renvoie `[x]`qui est triée.
- Si `x` est inférieur à la tête de la liste alors `x::liste` est triée.
- On note `liste = t::q`. `q` est une partie d'une liste triée donc est triée d'où, par hypothèse de récurrence, `insertion x q` est triée. Si `t` est strictement inférieur à `x`, il est donc inférieur à tous les éléments de `insertion x q` donc `t::(insertion x q)` est triée. Ainsi le résultat est toujours trié.

Complexité Au pire on doit comparer `x` à tous éléments de `liste`, dans le cas où `x` majore tous les éléments.

Analyse du tri par insertion d'une liste

Terminaison L'appel récursif porte sur une liste de taille diminuée de 1 et la fonction termine pour les listes de taille 0. On en déduit par récurrence sur la taille que l'application termine pour toute liste.

Preuve La preuve de `insertion` permet de prouver par récurrence que le résultat de `tri` est toujours triée.

Complexité Pour une liste de taille n on appelle `insertion` pour une liste de taille $n-1$ puis $n-2$ jusqu'à 0 donc la complexité est majorée par $\sum_{i=0}^{n-1} i = \frac{n(n-1)}{2}$.

La complexité est quadratique.

III.2 Tri par sélection

Dans le tri par insertion, le travail se fait après avoir trié récursivement la queue de la liste, on doit insérer la tête dans la liste triée. On peut inverser l'ordre difficulté en isolant le terme minimal et le reste puis en ajoutant simplement le minimum à la liste triée obtenu récursivement.

```
let rec tri liste =
  match liste with
  |[] -> []
  |t::q -> let mini, reste = separerMin liste in
           min :: (tri reste);;
```

Exercice 5
Après en avoir précisément défini le résultat attendu, écrire une fonction `sepererMin`.

Exercice 6
Analyser (terminaison, preuve, complexité) la fonction `sepererMin`.

Exercice 7
Analyser (terminaison, preuve, complexité) la fonction de tri.

IV Conclusion

Il est recommandé d'utiliser plutôt le tri par insertion. en effet, sa complexité est variable tandis que celle du tri par sélection est toujours le maximum, $\frac{n(n-1)}{2}$ qui correspond à toutes comparaisons possibles de 2 éléments parmi n qu'il est possible de faire.
On peut expérimenter que, dans le cas de tableaux ou de listes "presque" triés, le tri par insertion peut avoir une complexité linéaire, ce qui est optimal dans le cas de la création d'un ensemble de n éléments. Dans le cas de tris plus rapide (on en verra quelques uns) il arrivera que le tri ne se fera qu'incomplètement et que les derniers rangements se fasse avec un tri par insertion.

Solutions

Solution de l'exercice 1

Initialisation Au départ on `k = i-1` avec $0 \leqslant i < n$ donc 1. est vraie,

On n'a rien fait donc 2. est valide avec la permutation égale à l'identité,

3. exprime le tri initial des i premiers éléments,

4. et 5. sont sans objet. Ainsi $\mathcal{P}(i-1)$ est valide

Passage de l'invariant On suppose que $\mathcal{P}(k)$ est valide, avec $k \geqslant 0$ et `t.(k) > t.(k+1)`.

On veut prouver $\mathcal{P}(k-1)$ après les deux instructions de la boucle.

1. On a $0 \leqslant k < i$ donc $-1 \leqslant k-1 < i-1 < i$ donc 1. est valide.
2. On a permuté les valeurs de 2 indices inférieurs à i, 2. est préservé.
3. Les éléments d'indices 0 à $k-1$ sont inchangés donc la condition 3. de $\mathcal{P}(k)$ fournit `t.(0) <= t.(1)<= ... <= t.(k-1)`, la condition 3. de $\mathcal{P}(k-1)$ est vérifiée.
4. On avait `t.(k) > t.(k+1)` avant l'échange donc on a `t.(k) < t.(k+1)` après.
   La condition 5. de $\mathcal{P}(k)$ donne, après l'échange, `t.(k) <= t.(k+2)`
   Comme on avait, d'après la condition 4. de de $\mathcal{P}(k)$, `t.(k+2) <= ... <= t.(i)` avec des valeurs inchangées pour ces indices. Ainsi la condition 4. de $\mathcal{P}(k-1)$ est vérifiée car on a `t.((k-1)+1)<= t.((k-1)+2)<= ... <= t.(i)` :

Conclusion Si on a $k = -1$, la propriété 4. fournit la croissances des $i+1$ premiers éléments.

Si on a $k \geqslant 0$ et `t.(k) <= t.(k+1)`, les propriété 3. et 4. fournissent la croissances des $i+1$ premiers éléments.

Solution de l'exercice 2

Pour une liste triée `placer t i` demande une comparaison pour $i > 0$ et n'en effectue aucune si $i = 0$ donc on effectue $n-1$ comparaisons.

Dans le cas d'une liste inversement triée, le nombre de comparaisons est maximal, $\frac{n(n-1)}{2}$.

Solution de l'exercice 3

Il suffit de copier les grands éléments à leur droite et, à la fin, de placer l'élément à insérer à sa place. Il ne faut pas oublier de mettre cet élément dans une variable.

```
let placer t i=
  let j = ref (i-1) in
  let pivot = t.(i) in
  while !j >= 0 && plusGrand t.(!j) !pivot do
    t.(!j+1) <- t.(!j);
    decr j done;
  t.(!j+1) <- pivot;
```

Solution de l'exercice 4

Pour arriver à k on doit échanger `t.(j)` et `t.(j+1)` pour `!j` prenant les valeurs $i-1, j-2$ jusqu'à k : à chaque fois on a fait une comparaison. Si $k > 0$ on doit faire une dernière comparaison pour finir la boucle.

Ainsi $n_i(k) = i-k+1$ pour $k > 0$ et $n_i(0) = i$ d'où

$\overline{C_p}(i) = \dfrac{i + \sum_{k=1}^{i}(i-k+1)}{i+1} = \dfrac{i}{2} + 1 - \dfrac{1}{i+1}$. Ainsi la complexité moyenne est quadratique car

$$\sum_{i=0}^{n-1}\left(\frac{i}{2} + 1 - \frac{1}{i+1}\right) = \frac{n^2}{4} + \frac{3n}{4} - \sum_{k=1}^{n}\frac{1}{k} \sim \frac{n^2}{4}$$

Solution de l'exercice 5

`separerMin liste` doit envoyer un élément de la liste inférieur à tous les éléments de la liste et une liste contenant tous les autres éléments.

```
let rec separerMin liste =
  match liste with
  |[] -> failwith "Pas de minimum dans une liste vide"
  |[x] -> x, []
  |t::q -> let mini, reste = separerMin q in
           if  plusGrand t mini
           then mini, t::reste
           else t, q;;
```

Solution de l'exercice 6

Terminaison L'appel récursif porte sur une liste de taille diminuée de 1 et la fonction termine pour les listes de taille 0. On en déduit par récurrence sur la taille que l'application termine pour toute liste.

Preuve On prouve, toujours par récurrence sur la taille, la spécification est respectée.

- Si `liste` ne contient qu'un élément, il est bien le minimum et il ne reste rien.
- On note `liste = t::q` et `mini, reste = separerMin q`.
  D'après l'hypothèse de récurrence, `mini` minore tous les éléments de `q`.
  Si on a `mini < t` alors `mini` minore tous les éléments de `liste` et les autres éléments sont les autres éléments de `q`, qui forment `reste` d'après l'hypothèse de récurrence plus `t` donc `mini, t::reste` répond bien aux spécifications.
  Si on a `t <= mini` alors `t` minore tous les éléments de `q` ainsi que `t` et les éléments restants forment `q` donc `t, q` répond bien aux spécifications.

Complexité On montre par récurrence qu'au pire on doit effectue $n-1$ comparaisons dans le traitement d'une liste de taille n.

Solution de l'exercice 7

Terminaison L'appel récursif porte sur une liste de taille diminuée de 1 et la fonction termine pour les listes de taille 0. On en déduit par récurrence sur la taille que l'application termine pour toute liste.

Preuve Les spécification de `separerMin` permettent de prouver, toujours par récurrence sur la taille, que le tri renvoie une liste triée contenant tous les éléments de la liste passée en paramètre.

Complexité Pour une liste de taille n on appelle `separerMin` pour une liste de taille n puis $n-1$ jusqu'à 1 donc la complexité est majorée par $\sum_{i=}^{n}(i-1) = \frac{n(n-1)}{2}$.