Chapitre 2

Analyse des programmes

2.1 Bonnes pratiques

Peu importe le langage utilisé, il est important d'avoir de bonnes habitudes de programmation afin de faciliter la compréhension du code, son analyse, ou encore faciliter le débuggage.

- Même si la syntaxe du langage ne l'impose pas, aérer et indenter son code notamment pour voir apparaître clairement les structures imbriquées.
- Donner des noms les plus explicites possible aux variables afin de faciliter la compréhension du code.
- Insérer des commentaires, pas pour paraphraser le code, mais pour donner des informations supplémentaires facilitant la compréhension ou bien indiquant des éléments importants pour l'analyse des programmes (par exemple, une précondition, un invariant).
- Apprendre à comprendre les erreurs de compilation pour identifier plus efficament les sources d'erreurs à l'avenir. Bien prendre en compte les messages d'erreurs même s'il ne s'agit que d'alertes.
- Compiler très régulièrement : il est beaucoup plus simple de trouver la provenance d'une erreur quand on sait où chercher, il ne faut donc pas avoir fait trop de modifications depuis la dernière compilation.
- Réaliser des jeux de tests pertinents pour s'assurer du bon fonctionnement d'un programme. Notamment sur des valeurs particulières, pour s'assurer qu'il n'y a pas de problème lorsqu'on est dans les conditions limites d'utilisation (pas de division par 0 par exemple), mais aussi pour tester les performances du programme (sur de grands entiers par exemple).
- Pour se donner l'intuition d'une solution à un problème donné, il peut être très utile d'effectuer à la main (papier/crayon) la résolution sur un cas particulier. On identifiera ainsi plus clairement les différentes étapes de l'algorithme à mettre en œuvre et les éventuels cas de base.
- Dans la phase d'écriture d'un programme on peut utiliser des assertions (fonction `assert`) afin de vérifier que les préconditions sont toujours vérifées et obtenir des indications supplémentaires sur les parties du code générant d'éventuelles erreurs. Exemple d'erreur : l'exception `Divizion_by_zero` est levée, une assertion permet d'identifier qu'à la ligne 12 la variable `i` prend la valeur 0, on pourra donc corriger plus facilement la source de l'erreur.

2.2 Correction

Lorsqu'on souhaite démontrer que le résultat renvoyé par une fonction est correct, la réalisation d'un jeu de tests est bien entendu insuffisant.

Pour prouver la correction d'un algorithme on doit tout d'abord énoncer clairement sa ***spécification*** en précisant :

- les différentes entrées admissibles, pour lesquelles on peut imposer des ***préconditions***
- le résultat attendu après l'exécution de l'algorithme

Exemple 2.1. Pour la fonction `factorielle` définie dans l'exemple 1.20 page 12 la spécification est la suivante :

- `factorielle` prend en argument un entier n. La précondition est que n doit être positif ou nul.
- Le résultat attendu est $n!$.

La preuve de la correction s'effectue alors généralement en utilisant un raisonnement par récurrence, notamment lorsqu'on étudie des fonctions récursives, ou bien lorsque la présence d'une boucle permet de raisonner sur le nombre d'itérations.

Théorème 2.1 (**Principe de récurrence simple**)**.**

Soit $P(n)$ une propriété dépendant d'un entier naturel n. Si les deux conditions suivantes sont vérifiées :

- $P(0)$ est vérifiée ***(Initialisation)***
- lorsqu'on suppose, pour un certain entier n, que $P(n)$ est vérifiée (c'est l'***hypothèse de récurrence***), alors $P(n+1)$ est encore vérifiée ***(Hérédité)***

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

Exemple 2.2. Correction de la fonction `factorielle`
Montrons par récurrence sur n que pour tout $n \in \mathbb{N}$, la propriété $P(n)$: « `factorielle n` = n ! » est vérifée.

- Si $n = 0$: `factorielle n` $= 1 = 0!$. $P(0)$ est donc bien vérifée.
- Hérédité : On suppose que, pour un certain $n \in \mathbb{N}$, $P(n)$ est vérifée, montrons qu'alors $P(n+1)$ est encore vérifée.

$$\begin{aligned} \texttt{factorielle (n+1)} &= (n+1) \times (\texttt{factorielle n}) \\ &= (n+1) \times n! && \text{par hypothèse de récurrence} \\ &= (n+1)! \end{aligned}$$

Ainsi, par récurrence sur n, $P(n)$ est vérifée pour tout $n \in \mathbb{N}$.
La fonction `factorielle` est donc correcte puisqu'elle satisfait bien sa spécification.
Parfois cependant, on a besoin de davantage de souplesse quant au rang auquel l'hypothèse de récurrence doit être vérifée. Dans ce cas, on utilise plutôt une récurrence forte :

Théorème 2.2 (**Principe de récurrence forte**)**.**

Soit $P(n)$ une propriété dépendant d'un entier naturel n. Si les deux conditions suivantes sont vérifiées :

- $P(0)$ est vérifiée ***(Initialisation)***
- lorsqu'on suppose, pour un certain entier n, que $P(k)$ est vérifiée pour tous les entier k tels que $k < n$ alors $P(n)$ est encore vérifiée ***(Hérédité)***

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

Exemple 2.3. On définit la fonction Ocaml récursive `puiss_2 : int -> int * int` dont la spécification est la suivante :

- `puiss_2` prend en argument un entier naturel n non nul (précondition)
- Le résultat renvoyé est l'unique couple $(a, p) \in \mathbb{N} \times \mathbb{N}^\star$ tel que $n = a \times 2^p$ et a n'est pas divisible par 2

```
# let rec puiss_2 n =          (* n > 0 *)
        if n mod 2 <> 0 then
                n, 0
        else let a, p = puiss_2 (n/2) in
                a, (p+1)
;;
val puiss_2 : int -> int * int = <fun>
```

```
# puiss_2 44;;
- : int * int = (11, 2)
# puiss_2 56;;
- : int * int = (7, 3)
```

On note $P(n)$: « `puiss_2 n` est l'unique couple $(a,p) \in \mathbb{N}^\star \times \mathbb{N}$ tel que $n = a \times 2^p$ et 2 ne divise pas a ».
Montrons par récurrence forte sur n que $P(n)$ est vraie pour tout $n \in \mathbb{N}^\star$.

- Si $n = 1$: comme 1 n'est pas divisible par 2, `puiss_2 1 = (1, 0)`.
  Or on a bien $1 = 1 \times 2^0$ et 2 qui ne divise pas 1. La propriété $P(1)$ est donc bien vérifiée.
- Hérédité : On suppose que, pour un certain $n \in [\![2; +\infty[\![$, $P(k)$ est vérifée pour tout $k \in \mathbb{N}^\star$ tel que $k < n$. Montrons qu'alors $P(n)$ est encore vérifée.
  - Si n n'est pas divisible par 2, alors `puiss_2 n = (n, 0)`. Or on a bien $n = n \times 2^0$.
  - Sinon, en notant `(a, p) = puiss_2 (n/2)` on a `puiss_2 n = (a, p+1)`.

$$\begin{aligned} \text{Or,} \quad n &= \frac{n}{2} \times 2 \\ &= a \times 2^p \times 2 \qquad \text{où 2 ne divise pas } a \text{ par hypothèse de récurrence} \\ n &= a \times 2^{p+1} \end{aligned}$$

Ainsi, par récurrence forte sur n, $P(n)$ est vérifiée pour tout $n \in \mathbb{N}^\star$.
On en déduit la correction de la fonction `puiss_2 n`.

Dans le cas d'une boucle (`for` ou `while`) on utilise un invariant de boucle pour prouver la correction :

Définition 2.1. Un ***invariant de boucle*** est une propriété qui doit :

- être vérifiée avant l'exécution de la boucle
- rester vérifiée après chaque itération

En particulier, cette propriété sera encore vérifiée à la fin de l'exécution de la boucle et peut donc, si elle est bien choisie, permettre de prouver la correction de l'algorithme.

Exemple 2.4. On définit la fonction Ocaml `puissance_2 : int -> int * int` dont la spécification est la même que précédemment mais utilise cette fois une boucle `while` :

- `puissance_2` prend en argument un entier naturel n non nul (précondition)
- Le résultat renvoyé est l'unique couple $(a,p) \in \mathbb{N} \times \mathbb{N}^\star$ tel que $n = a \times 2^p$ et a n'est pas divisible par 2

```
# let puissance_2 n = let a, p = ref n, ref 0 in          (* n > 0 *)
      while !a mod 2 = 0 do        (* invariant : !a * 2^!p = n *)
            a := !a / 2;
            p := !p + 1
      done;
(!a, !p);;
val puissance_2 : int -> int * int = <fun>
# puissance_2 8;;
- : int * int = (1, 3)
# puissance_2 31;;
- : int * int = (31, 0)
```

Montrons que la propriété $a \times 2^p = n$ est invariante.

- À l'initialisation des variables, $a = n$ et $p = 0$. On a donc bien $a \times 2^p = n \times 2^0 = n$.
- Supposons la propriété vraie au début d'un tour de boucle : $a \times 2^p = n$.
  On note respectivement a' et p' les nouvelles valeurs des variables a et p à l'issue du tour de boucle.
  Montrons que la proriété $a' \times 2^{p'} = n$ est encore vérifiée.
  On a $a' = \frac{a}{2}$ et $p' = p + 1$ alors $a' \times 2^{p'} = \frac{a}{2} \times 2^{p+1} = a \times 2^p = n$ (par hypothèse).

Ainsi, *si le programme termine* (cela reste à démontrer !), l'invariant donné ci-dessus est toujours vérifié à la fin de l'exécution du programme. Ce qui prouve la correction du programme puisque le couple (a, p) renvoyé vérifie bien l'égalité $n = a \times 2^p$ et 2 ne divise pas a (c'est cette dernière propriété qui aura interrompu l'exécution de la boucle `while`).

2.3 Terminaison

Lorsqu'un algorithme est récursif ou fait appel à une boucle `while` il est nécessaire de prouver sa terminaison. En effet, dans le cas d'un algorithme récursif, il faut s'assurer qu'on finit bien par appeler un cas de base qui interrompra la succession d'appels récursifs. Dans le cas d'une boucle `while`, il faut s'assurer que la condition d'entrée en boucle finit par ne plus être vérifiée. Dans les deux cas, on utilise pour cela un variant :

Définition 2.2. Un ***variant*** est un paramètre défini en fonction des variables de l'algorithme qui :

- ne prend que des valeurs entières positives
- décroît strictement à chaque appel récursif ou itération d'une boucle (selon le cas)

Théorème 2.3. Tout algorithme possédant un variant termine.

Exemple 2.5. Revenons sur l'exemple 2.4 page 29. La variable a est un variant de boucle. En effet :

- à l'initialisation, $a = n \in \mathbb{N}^\star$
- à chaque itération, a prend la valeur $a' = \dfrac{a}{2}$ (condition d'entrée en boucle : 2 divise a), on a donc bien $a' \in \mathbb{N}^\star$ et $a' < a$.

On en déduit la terminaison de la fonction `puissance_2`.

Exemple 2.6. La fonction Ocaml récursive `mem` définie ci-dessous teste si un élément est présent dans une liste ou non.

```
# let rec mem elem = function
        | [] -> false
        | e::q when e = elem -> true
        | e::q -> mem elem q
;;
val mem : 'a -> 'a list -> bool = <fun>
# mem 4 [5; -2; 12; 6];;
- : bool = false
# mem "violon" ["piano"; "batterie"; "violon"; "trombone"];;
- : bool = true
```

La taille de la liste passée en argument est un variant puisque chaque nouvel appel récursif se fait sur la queue de la liste, qui a une donc une taille diminuée de 1. Cela justifie la terminaison de la fonction `mem`.

Remarque 2.1. Il n'est pas toujours évident de prouver la terminaison d'un algorithme. Par exemple, selon la conjecture de Syracuse, la fonction Ocaml définie ci-dessous termine pour toute entrée :

```
# let tps_de_vol n = let m = ref n and tps = ref 0 in
        while !m <> 1 do
                if !m mod 2 = 0 then
                        m := !m / 2          (* si !m est pair *)
                else
                        m := 3 * !m + 1;     (* si !m est impair *)
                tps := !tps + 1
        done;
!tps;;
val tps_de_vol : int -> int = <fun>
```

Mais cette conjecture n'a pour l'instant jamais été prouvée !

Remarque 2.2. On dit que la correction est ***partielle*** lorsque le résultat est correct, à la condition que l'algorithme s'arrête. On dit qu'elle est ***totale*** si de plus l'algorithme termine.

2.4 Tests

Nous avons vu que pour prouver la correction d'un algorithme on avait recours à un raisonnement par récurrence (éventuellement forte) ou à un invariant. Cependant il n'est pas toujours possible d'effectuer ce type de raisonnement et la recherche d'un invariant peu parfois s'avérer fastidieuse. Aussi on commencera généralement par effectuer un jeu de tests afin de vérifier la correction d'un programme. Il n'est pas attendu en MP2I de savoir générer automatiquement des jeux de tests, mais de savoir écrire un jeu de test pertinent :

- Lorsque le domaine d'entrée peut être partitionné, tester des entrées pour chacun des ensembles
- Tester aux valeurs limites

Exemple 2.7. Si un programme prend en entrée une valeur de type `'a list`, on pourra le tester sur des entrées de type `int list`, `string list`, `float list` ou encore `bool list`.
On pensera également à tester la liste vide `[]`.

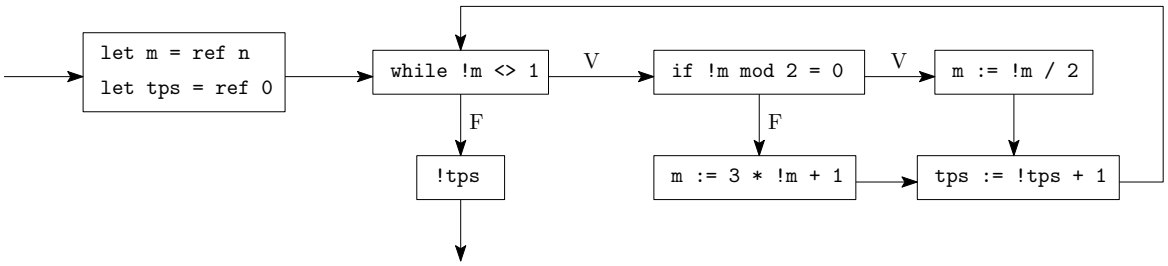
En complément, lorsqu'un programme contient des instructions conditionnelles on réalisera un test structurel en utilisant un graphe de flot de contrôle :

Définition 2.3. Le ***graphe de flot de contrôle*** d'un programme est un graphe orienté comportant :

- un sommet d'entrée - un sommet de sortie
- un sommet pour chaque bloc d'instructions élémentaires
- un sommet pour chaque condition et pour chaque boucle
- des arcs reliant les différents sommets - les arcs sortants d'une condition sont étiquetés par la valeur de la condition (V/F)

Une exécution, sur une entrée donnée, correspond alors à un ***chemin*** dans le graphe de flot de contrôle.

Exemple 2.8. Voici le graphe de flot de contrôle de la fonction `tps_de_vol` de la remarque 2.1 page 30.



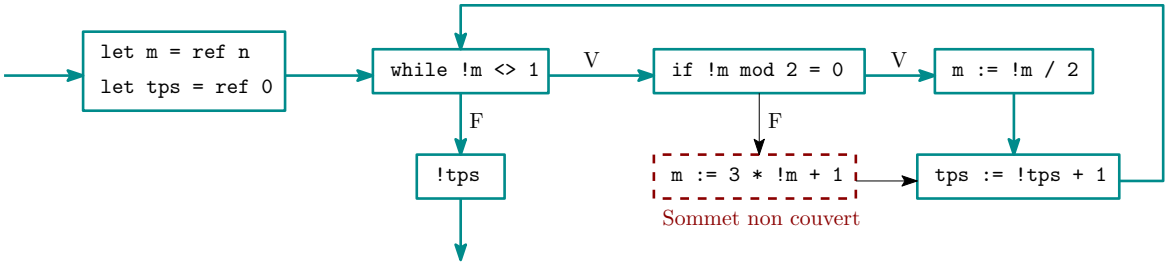
Définition 2.4. Dans un graphe de flot de contrôle, on dit qu'un chemin est ***faisable*** s'il existe une entrée pour laquelle l'exécution du programme correspond à ce chemin.

Remarque 2.3. L'existence de chemins infaisables est un problème ***indécidable*** (voir cours de spé).
Une fois déterminé le graphe de flot de controle, on peut réaliser un jeu de tests selon l'un ou l'autre des critères suivants (du plus faible au plus fort) :

- Couverture de toutes les instructions (i.e. de tous les sommets)
- Couverture de toutes les branches sur les chemins faisables (tous les arcs)
- Couverture de tous les chemins faisables

Remarque 2.4. Il peut y avoir une infinité de chemins faisables lorsque le graphe de flot de contrôle comporte des cycles. On pourra alors se contenter d'un jeu de tests satisfaisant un critère de couverture des chemins d'une longueur fixée ou bien choisir l'un des deux autres critères de converture (plus faibles).

Exemple 2.9. Dans le graphe de flot de contrôle de l'exemple précédent il y a une infinité de chemins faisables (par exemple, les puissances de 2 passées en entrées correspondent à des chemins faisables distincts). De plus, le seul test sur l'entrée `8` est insuffisant car ce test ne satisfait pas le critère de couverture des instructions puisque le chemin correspondant est le suivant :

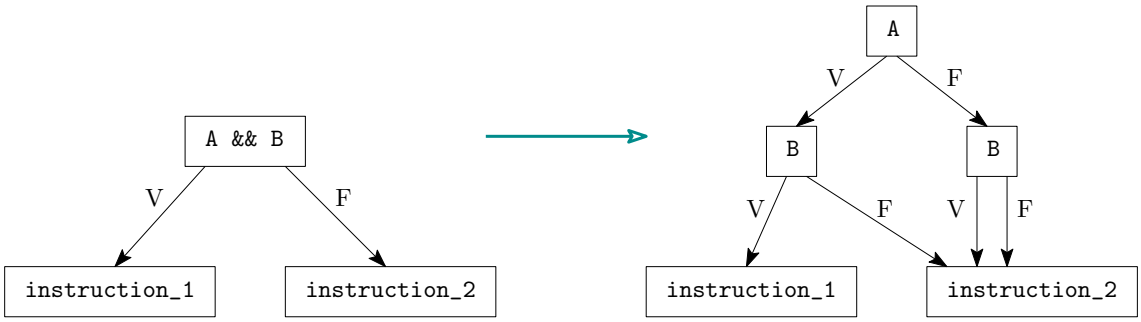


Enfin, lorsqu'une condition comporte des opérateurs booléens, on ne se contente pas de traiter l'expression comment globalement vraie ou fausse, on teste chaque cas possible.

Exemple 2.10. On donne la table de vérité de l'opérateur `&&` :

A	B	A && B
0	0	0
0	1	0
1	0	0
1	1	1

On fournira donc un jeu de tests testant de manière exhaustive la condition `A && B` :



2.5 Complexité

Maintenant qu'on sait justifier qu'un algorithme répond bien à un problème donné, il reste à évaluer ses performances. En effet, plusieurs algorithmes peuvent répondre à un même problème mais nécessiter un temps de calcul et un espace mémoire bien différents.

Définition 2.5. Pour un algorithme donné :

- sa ***complexité temporelle*** indique le temps de calcul nécessaire à son exécution
- sa ***complexité spatiale*** indique l'espace mémoire utilisé lors de son exécution

2.5.1 Complexité temporelle

Définition 2.6. La ***complexité temporelle*** d'un algortihme correspond au nombre d'opérations élémentaires réalisées à son exécution sur une entrée, exprimée en fonction de la taille de l'entrée.

Exemple 2.11. Pour un tableau ou une liste en entrée, on exprimera la complexité temporelle en fonction de la longueur du tableau ou de la liste.

Remarque 2.5. Parmi les différentes opérations élémentaires possibles on trouve par exemple :

- les additions, multiplications, etc.
- les comparaisons sur des types simples
- l'accès, en lecture ou en écriture, à un élément dans un tableau
- les affectations

On suppose que toutes ces opérations se font en temps constant.

Exemple 2.12. La recherche d'un maximum dans une liste non triée nécessite d'effectuer $n-1$ comparaisons pour une liste de taille n

Définition 2.7. La complexité algorithmique peut être évaluée de plusieurs façons :

- dans le pire cas : c'est la complexité maximale qu'on peut atteindre
- dans le meilleur cas : c'est la complexité minimale attendue
- en moyenne : il s'agit de la moyenne des complexités obtenues pour toutes les entrées possibles d'une taille donnée qu'on supposera équiprobables

Remarque 2.6. Dans la majorité des cas, c'est la complexité dans le pire des cas qu'on étudie. Si rien n'est précisé, c'est qu'on demande une complexité dans le pire des cas.

2.5.2 Classes de complexité

Le tableau suivant regroupe les différentes complexité rencontrées et leur dénomination, classées dans l'ordre croissant de complexité :

Complexité	Dite	Exemple
1	constante	opérations élémentaires
$\log n$	logarithmique	dichotomie
n	linéaire	recherche séquentielle, boucle
$n\log n$	linéarithmique	tri fusion, diviser pour régner
n^2	quadratique	tri insertion, boucles imbriquées
$a^n \quad (a>1)$	exponentielle	appels récursifs multiples

En pratique, on ne cherche pas particulièrement à connaître une expression exacte de la complexité mais simplement un ordre de grandeur de la ***complexité asymptotiqe***. Autrement dit, on cherche à déterminer, pour de grandes valeurs de n, un ordre de grandeur de la complexité.

Pour cela, on utilise les ***notations de Landau*** qui permettent de décrire le comportement asymptotique.

Définition 2.8. Soit $f, g : \mathbb{N} \longrightarrow \mathbb{N}$.

- On dit que $g(n)$ est un « grand O » de $f(n)$ et on note $\boxed{g(n) = \mathcal{O}(f(n))}$
  s'il existe un réel $k \in \mathbb{R}^+$ et un rang $n_0 \in \mathbb{N}$ tels que $g(n) \leq kf(n),\ \forall n \geq n_0$.
- On dit que $g(n)$ est un « Omega » de $f(n)$ et on note $\boxed{g(n) = \Omega(f(n))}$
  s'il existe un réel $k \in \mathbb{R}^+$ et un rang $n_0 \in \mathbb{N}$ tels que $g(n) \geq kf(n),\ \forall n \geq n_0$.
- On dit que $g(n)$ est un « Theta » de $f(n)$ et on note $\boxed{g(n) = \Theta(f(n))}$
  s'il existe deux réels $k, \ell \in \mathbb{R}^+$ et un rang $n_0 \in \mathbb{N}$ tels que $kf(n) \leq g(n) \leq \ell f(n),\ \forall n \geq n_0$.
- On dit que $g(n)$ est « équivalent » à $f(n)$ et on note $\boxed{g(n) \sim f(n)}$ si $\dfrac{g(n)}{f(n)} \underset{n\to+\infty}{\longrightarrow} 1$.

Remarque 2.7. Nous utiliserons principalement la notation $\mathcal{O}$ pour évaluer la complexité des algorithmes.

Exemple 2.13. Si $C(n) = 4n^2 + 5n + 2$ alors :

- $C(n) \sim 4n^2$: $C(n)$ est équivalent à n^2.
- $C(n) = \Theta\left(n^2\right)$: $C(n)$ est du même ordre de grandeur que n^2.
- $C(n) = \mathcal{O}\left(n^2\right)$: $C(n)$ est majoré par n^2, à un facteur constant près.
  Mais on a aussi $C(n) = \mathcal{O}\left(n^3\right)$, $C(n) = \mathcal{O}\left(n^8\right)$, ou même $C(n) = \mathcal{O}\left(2^n\right)$.
- $C(n) = \Omega\left(n^2\right)$: $C(n)$ est minoré par n^2, à un facteur constant près.
  Mais aussi $C(n) = \Omega(n)$ ou bien $C(n) = \Omega(1)$.

D'un point de vue de la complexité, on retiendra parmi toutes ces possibilités : $C(n) = \mathcal{O}\left(n^2\right)$.

2.5.3 Exemples de calcul de complexité temporelle

Exemple 2.14. Complexité d'une boucle : l'exponentiation naïve

```
# let pow a n = let res = ref 1 in
        for i = 1 to n do
                res := a * !res
        done;
        !res;;
val pow : int -> int -> int = <fun>
```

L'appel `pow a n` effectue exactement `n` tours de boucle. Au total, `n` multiplications sont donc effectuées. On en déduit une complexité $\Theta(n)$: la complexité est linéaire.

Exemple 2.15. Complexité avec des boucles emboîtées : le tri à bulles

```
# let tri_bulles tab = let taille = Array.length tab in
        (* trie en place le tableau tab *)
        for i = 0 to taille - 2 do
                for j = i + 1 to taille - 1 do
                        if tab.(j) < tab.(i) then
                                begin    (* echange des elements d'indice i et j *)
                                        let temp = tab.(i) in
                                        tab.(i) <- tab.(j);
                                        tab.(j) <- temp
                                end
                done
        done;;
val tri_bulles : 'a array -> unit = <fun>
# let t = [| 5; -2; 7; 1; 8; 4 |] in
        tri_bulles t;
        t;;
- : int array = [|-2; 1; 4; 5; 7; 8|]
```

- L'accès à la longueur du tableau s'effectue en temps constant.
- Le nombre d'opérations réalisées à chaque tour de la boucle interne est constant (une comparaison, deux accès en lecture, deux en écriture, une affectation).
- En notant n la longueur du tableau donné en entrée, le nombre d'itérations est égal à :

$$\sum_{i=0}^{n-2} n-1-i = \sum_{k=1}^{n-1} k = \frac{(n-1)n}{2}$$

- La complexité du tri à bulles est donc $\Theta\left(n^2\right)$: on dit que la complexité est quadratique.

Exemple 2.16. Complexité d'un algorithme dichotomique
Recherche d'un élément dans un tableau trié en utilisant une méthode « diviser pour régner ».

```
# exception Found of int;;
exception Found of int
# let cherche tab elem = let taille = Array.length tab in
        (* renvoie la premiere position de elem s'il est present dans tab, -1 sinon *)
        (* on suppose que tab est trie *)
        let i, j = ref 0, ref (taille - 1) in (* recherche entre les indices i et j *)
        try
                while !i <= !j do
                        let m = ( !i + !j ) / 2 in
                        (* division de la zone de recherche par 2 *)
                        if elem < tab.(m) then j := m - 1
                        else if elem > tab.(m) then i := m + 1
                        else raise (Found m)
                done;
                -1
        with Found m -> m;;
val cherche : 'a array -> 'a -> int = <fun>
# cherche [| 0; 2; 4; 6; 8; 10; 12 |] 4;;
- : int = 2
# cherche [| 0; 2; 4; 6; 8; 10; 12 |] 7;;
- : int = -1
```

- Dans le pire des cas, l'élément recherché n'est pas présent dans le tableau, aucune exception ne sera alors levée.
- On effectue moins d'une dizaine d'opérations lors de chaque itération.
- À chaque itération la longueur de l'intervalle de recherche est divisée par 2 :
  on passe d'une longueur $j-i$ à une longueur $\left\lfloor \frac{j-i}{2} \right\rfloor$ au maximum.
  Ainsi, si $2^k \leq n < 2^{k+1}$, alors $k+1$ tours de boucle seront effectués au maximum.
  Or, si $2^k \leq n < 2^{k+1}$, alors $k \leq \log n < k+1$ car la fonction log est strictement croissante sur $\mathbb{N}^\star$.
- On en déduit que la complexité de cet algorithme dichotomique est $\mathcal{O}(\log n)$

Remarque 2.8. Attention aux éventuels coûts cachés : même si l'algorithme dichotomique précédent est plus efficace qu'un algorithme de recherche séquentielle, il nécessite de travailler sur des tableaux triés.
Il ne serait cependant pas raisonnable de trier au préalable un tableau (coût au mieux en $\mathcal{O}(n\log n)$) car la complexité globale serait alors en $\mathcal{O}(n\log n)$!

Exemple 2.17. Complexité d'une fonction récursive

```
# let rec base a n = match n with
        (* renvoie la liste des n premieres puissances de a *)
        | 0 -> [1]
        | n -> (pow a n)::(base a (n-1))
;;
val base : int -> int -> int list = <fun>
# base 2 5;;
- : int list = [32; 16; 8; 4; 2; 1]
```

On a vu précédemment que l'appel `pow a n` effectuait exactement n multiplications, pour tout $n \in \mathbb{N}$.
En notant $C(n)$ le nombre de multiplications effectuées lors de l'appel `base a n` on a donc la relation de récurrence suivante : $\begin{cases} C(0) = 0 \\ C(n) = n + C(n-1) \quad , \forall n \in \mathbb{N}^\star \end{cases}$

On peut déterminer $C(n)$ grâce à une somme télescopique :

$$C(n) - \underbrace{C(0)}_{=0} = \sum_{k=1}^{n} C(k) - C(k-1)$$

$$= \sum_{k=1}^{n} k$$

$$C(n) = \frac{n(n+1)}{2}$$

On en déduit que la complexité de la fonction `base` est $\Theta\left(n^2\right)$.

Exemple 2.18. Complexité en moyenne
On considère la fonction suivante qui pour une permutations σ de $[\![0,n]\!]$ et un entier $i \in [\![0,n]\!]$ renvoie l'image de i par la permutation σ.

```
# let perm tab i =   (* 0 <= i < Array.length tab *)
       let j = ref 0 in
       while tab.(!j) <> i do
              j := !j + 1
       done;
!j;;
val perm : 'a array -> 'a -> int = <fun>
# perm [| 2; 5; 3; 1; 4; 0 |] 1;;
- : int = 3
```

Soit $n \in \mathbb{N}^*$ et un élément $i \in [\![0,n-1]\!]$. Parmi tous les tableaux de permutations de longueur n, il est équiprobable que l'image de i par une permutation soit n'importe quel valeur de $[\![0,n-1]\!]$.
Or le nombre d'itérations pour trouver que l'élément i est à la position j vaut j.
La complexité en moyenne (en nombre d'itérations) vaut donc : $C(n) = \frac{1}{n}\sum_{j=0}^{n-1} j = \frac{1}{n} \times \frac{(n-1)n}{2} = \frac{n-1}{2} = \Theta(n)$.
La complexité moyenne de la fonction `perm` est donc linéaire.

2.5.4 Complexité spatiale

Il peut parfois être intéressant de jouer sur la complexité spatiale d'un algorithme pour améliorer la complexité temporelle de celui-ci.

Exemple 2.19. Calcul des coefficients binomiaux
On peut écrire un premier algorithme permettant de calculer le coefficient binomial $\binom{n}{k}$,
en utilisant les relations suivantes :

$$\forall k,n \in \mathbb{N}^*,\ k<n, \qquad \binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

$$\forall n \in \mathbb{N}, \qquad \binom{n}{0} = \binom{n}{n} = 1$$

Le calcul des coefficients binomiaux est illustré par le triangle de Pascal :

n \ k	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

```
# let rec binom k n =
        if k = 0 || k = n then 1
        else binom (k-1) (n-1) + binom k (n-1);;
val binom : int -> int -> int = <fun>
# binom 2 4;;
- : int = 6
```

Cependant cette fonction a une complexité spatiale très importante liée à la pile d'appels récursifs.
De plus, elle a une complexité temporelle exponentielle.
En effet, on peut exprimer cette complexité grâce à la relation $C(n) = 2C(n-1)+1$, $\forall n \in \mathbb{N}^\star$, en montrant par récurrence sur n que $C(n) = 2^n - 1$, $\forall n \in \mathbb{N}$.
On peut considérablement améliorer cette complexité en calculant les coefficients binomiaux ligne par ligne et en stockant ces valeurs dans un tableau.
Afin de ne stocker qu'une ligne à la fois, on met la ligne à jour en partant de la droite.
On obtient la fonction suivante :

```
# let binome k n =
        let coeffs = Array.make (n+1) 0 in
        for j = 0 to n do
                coeffs.(j) <- 1;  (* j parmi j = 1 *)
                for i = j - 1 downto 1 do (* calcul de i parmi j *)
                        coeffs.(i) <- coeffs.(i-1) + coeffs.(i)
                done    (* il y a deja 1 a la position 0 *)
        done;
        coeffs.(k);;
val binome : int -> int -> int = <fun>
# binome 4 6;;
- : int = 15
```

Cette nouvelle fonction a une complexité spatiale linéaire (pas d'appels récursifs cette fois et on stocke un tableau de longueur n) et une complexité temporelle quadratique.
En effet, le nombre d'itérations vaut : $\sum_{j=0}^{n} j = \frac{n(n+1)}{2} = \Theta\left(n^2\right)$.