Complexité des algorithmes
Tris (première approche)

Introduction : notations o, $\mathcal{O}$, Θ, Ω.
On travaille uniquement avec des suites positives.
Définitions
$u_n = o(v_n)$ si et seulement si $\frac{u_n}{v_n} \xrightarrow[n\to\infty]{} 0$.
$u_n = \mathcal{O}(v_n)$ si et seulement si $\exists n_0,\ \exists M > 0,\ \forall n \geqslant n_0,\ u_n \leqslant M v_n$.
$u_n = \Theta(v_n) \iff u_n = \mathcal{O}(v_n)$ et $v_n = \mathcal{O}(u_n)$.
$u_n = \Omega(v_n) \iff v_n = \mathcal{O}(u_n)$.

1 Taille des données

Un algorithme prend comme paramètre des données appartenant à un ensemble $\mathcal{D}$.
On notera souvent $|d|$ la taille d'une donnée d.
Pour mesurer la complexité d'un algorithme, on cherche à quantifier les données par leur taille, qui sera généralement un entier naturel.

Taille d'un entier : $\mathcal{O}(1)$ ou le nombre de bits qui le représentent (i.e $\lfloor \log_2 n \rfloor + 1$).
Taille d'un booléen ou d'un flottant : Constante ou $\mathcal{O}(1)$.
Taille d'un vecteur : sa longueur ou plus généralement la somme des tailles de ses éléments.
Taille d'une liste : idem.
Taille d'une chaîne de caractères : sa longueur.

2 Complexité temporelle

2.1 Introduction

Une préoccupation fondamentale de l'informatique est de mesurer l'efficacité d'un programme en terme de vitesse d'exécution.
La complexité temporelle d'un programme est une application $t : \mathcal{D} \to \mathbb{R}^+$ telle que $t(d)$ mesure le temps d'exécution du programme pour la donnée d. Pour simplifier, on quantifie les données par leur taille.

Exemples :

Recherche d'un élément dans une liste. On teste chaque élément de la liste. On effectue au plus n comparaisons (n étant la longueur de la liste), mais pour deux listes de même longueur, le nombre de tests n'est pas toujours le même. On dit que la complexité est linéaire, ou que c'est un $\mathcal{O}(n)$.

Recherche d'un élément dans un tableau trié. Par dichotomie, on effectue au plus de l'ordre de $\lfloor \log_2 n \rfloor + 1$ tests.

Maximum dans une liste. On est obligé de balayer toute la liste. On effectue toujours n comparaisons, quelle que soit la liste de longueur n.

2.2 Complexité dans le pire des cas

Soit $\mathcal{D}_n$ l'ensemble des données de taille n. La complexité dans le pire des cas est l'application $\mathbb{N} \to \mathbb{N}$ définie par $T(n) = \sup_{d \in \mathcal{D}_n} t(d)$.

On peut définir de même la complexité dans le meilleur des cas : $T(n) = \inf_{d \in \mathcal{D}_n} t(d)$.

Exemple : pour la recherche dans une liste de longueur n, la complexité est n dans le pire des cas et 1 dans le meilleur des cas.

2.3 Complexité en moyenne

On suppose que les données sont distribuées selon une certaine loi de probabilité p. La complexité en moyenne est l'espérance de la complexité des données de même taille : $T(n) = E(C_n)$ où C_n est la complexité restreinte à l'ensemble $\mathcal{D}_n$ des données de taille n.
Si $\mathcal{D}_n$ est fini, $T(n) = \sum_{d \in \mathcal{D}_n} p_n(d)\, t(d)$ où $p_n(d)$ est la probabilité d'apparition de la donnée d dans $\mathcal{D}_n$.

En général, les données de l'ensemble $\mathcal{D}_n$ sont équiprobables. Si $\mathcal{D}_n$ est fini, $T(n) = \dfrac{1}{\operatorname{card} \mathcal{D}_n} \sum_{d \in \mathcal{D}_n} t(d)$.

Cette complexité n'est pas toujours facile à évaluer, mais constitue une mesure très pertinente de l'efficacité d'un algorithme.

2.4 Remarques

La complexité d'un problème $\mathcal{P}$ est la borne inférieure des complexités des algorithmes résolvant $\mathcal{P}$.

Classification des types de complexité en fonction de la taille n des données :

$\mathcal{O}(1)$: constante.
$\mathcal{O}(\log n)$: logarithmique.
$\mathcal{O}(n)$: linéaire.
$\mathcal{O}(n \log n)$: quasi linéaire.
$\mathcal{O}(n^2)$: quadratique.
$\mathcal{O}(n^\alpha)$ avec $\alpha > 0$: polynomiale.
$\mathcal{O}(a^n)$ avec $a > 1$: exponentielle.

Le point de vue des informaticiens est qu'une complexité polynomiale (c'est-à-dire $\mathcal{O}(n^\alpha)$ pour un certain $\alpha > 0$) est jugée raisonnable, alors qu'une complexité exponentielle ($\Theta(a^n)$ avec $a > 1$) ou pire ne l'est pas.
Certains problèmes ont une complexité non polynomiale qu'on ne peut pas améliorer. On peut citer par exemple les tours de Hanoï ($2^n - 1$ déplacements), le problème du sac à dos, le problème du voyageur de commerce, la recherche d'un parcours hamiltonien d'un graphe (passer par tous les sommets une et une seule fois). Pour d'autres problèmes, certains algorithmes sont exponentiels et d'autres polynomiaux.
Par exemple, le test de primalité d'un entier est de complexité polynomiale (par rapport au nombre de bits) : ce résultat a été démontré par l'algorithme AKS publié en 2002 (complexité en $\mathcal{O}((\log n)^{12})$).

Tableau comparatif
On considère qu'une instruction prend $10^{-9} s$.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
10	10^{-8}	3.10^{-8}	10^{-7}	10^{-5}	5.10^{-8}	10^{-6}	3.10^{-3}
100	10^{-7}	6.10^{-7}	10^{-5}	10^{-3}	4.10^{8}	10^{21}	∞
1000	10^{-6}	10^{-5}	10^{-3}	1	10^{167}	∞	∞
10^6	10^{-3}	2.10^{-2}	10^3	10^9	∞	∞	∞
10^9	1	30	10^9	10^{18}	∞	∞	∞

3 Complexité spatiale

On s'intéresse ici à l'occupation mémoire requise lors de l'exécution d'un programme.
Pour un programme non récursif, il s'agit de la somme des tailles de toutes les variables locales utilisées.
Pour un programme récursif, il faut rajouter la place mémoire occupée par la pile des appels récursifs.
Par exemple, pour l'exponentielle récursive naïve, on peut dépasser la taille de la pile, d'où l'intérêt de l'exponentielle rapide.

Important : dans une programmation récursive agissant sur des tableaux, on doit éviter à tout prix de recopier le tableau dans l'appel récursif. La fonction principale ne sera généralement pas récursive, mais fera appel à une fonction auxiliaire récursive agissant sur des entiers, les appels récursifs portant sur les indices du tableau.

Exemple. Programme de calcul de la suite de Fibonacci : $u_{n+1} = u_n + u_{n-1}$, $u_0 = 0$, $u_1 = 1$.
L'algorithme consiste en une boucle de 1 à n. Si on ne garde que les deux dernières valeurs, la complexité spatiale est $\mathcal{O}(1)$. Si on garde toutes les valeurs précédemment calculées, elle devient $\Theta(n)$.

Avec l'augmentation continue des capacités mémoires des composants utilisés (clés USB, disques durs, CD et DVD), la complexité spatiale revêt moins d'importance que la complexité temporelle.

4 Analyse d'un algorithme comportant un seul appel récursif

4.1 Description

L'algorithme se présente sous la forme suivante : $\mathcal{A}(d) = \begin{cases} I_1 & \text{si } d \in B \\ \begin{cases} I_2, \\ \mathcal{A}(\Phi(d)), \\ I_3 \end{cases} & \text{si } d \notin B \end{cases}$

où B est l'ensemble des cas de base, Φ est la fonction récursive, vérifiant pour tout $|d| = n$, $|\Phi(d)| = \varphi(n)$ avec $\varphi(n) < n$ pour $d \notin B$ pour assurer la terminaison de l'algorithme.

Alors $T(n) = \begin{cases} t(I_1) \text{ si } d \in B \text{ pour tout } d \text{ tel que } |d| = n, \\ t(I_2) + t(I_3) + T(\varphi(n)) \text{ sinon.} \end{cases}$

Cette équation est appelée équation de complexité de $\mathcal{A}$.

Remarque : si on a $|\Phi(d)| \leqslant \varphi(n)$ lorsque $|d| = n$, on obtient une inéquation de complexité $T(n) \leqslant t(I_2) + t(I_3) + T(\varphi(n))$. On peut résoudre le cas limite correspondant à l'égalité.

4.2 L'appel récursif porte sur une donnée de taille $n-1$

$T(n) = T(n-1) + f(n)$. On suppose $T(0) = 0$. La solution de cette récurrence est $\boxed{T(n) = \sum_{k=1}^{n} f(k)}$.

Lemme. *Soient (u_n) et (v_n) deux suites $\geqslant 1$. On pose $U_n = \sum_{k=1}^{n} u_k$ et $V_k = \sum_{k=1}^{n} v_k$.*
Si $u_n = \mathcal{O}(v_n)$, alors $U_n = \mathcal{O}(V_n)$.
Si $u_n = \Theta(v_n)$, alors $U_n = \Theta(V_n)$.

Proposition.
- *Si $f(n) = \mathcal{O}(1)$, alors $T(n) = \mathcal{O}(n)$.*
- *Si $f(n) = \mathcal{O}(n)$, alors $T(n) = \mathcal{O}(n^2)$.*
- *Si $f(n) = \mathcal{O}(n^\alpha)$, alors $T(n) = \mathcal{O}(n^{\alpha+1})$.*

Mêmes résultats avec des Θ.

preuve. $\int_{k-1}^{k} x^\alpha \, dx \leqslant k^\alpha \leqslant \int_{k}^{k+1} x^\alpha \, dx$, donc $\int_0^n x^\alpha \, dx \leqslant \sum_{k=1}^{n} k^\alpha \leqslant \int_1^{n+1} x^\alpha \, dx$.
On en déduit que $\sum_{k=1}^{n} k^\alpha \underset{n\to\infty}{\sim} \frac{n^{\alpha+1}}{\alpha+1}$. •

Exemples.

- **Maximum d'une liste :**

```
let rec maxi = function
    [a] -> a
  | t::q -> let m = maxi q in if t > m then t else m ;;
```

  En comptant les tests, l'équation de complexité s'écrit $T(n) = 1 + T(n-1)$, d'où $T(n) = n$ (en moyenne, dans le pire ou dans le meilleur des cas).

- **Recherche dans une liste :**

```
let rec recherche x = function
     [] -> false
 | t::q -> (x=t) || recherche x q ;;
```

  Dans le pire des cas, $T(n) = 1 + T(n-1)$, d'où $T(n) = n$. La complexité en moyenne n'a pas de sens, tout dépend de l'étendue des valeurs possibles.

- **Position dans une liste :**

```
let rec position x l = match l with
  [] -> failwith "non présent"
  | t::q when t = x -> 0
  | t::q -> 1 + position x q ;;
```

On note n la longueur de la liste passée en argument.
La complexité est égale à n dans le pire des cas, 1 dans le meilleur des cas.
Si on se limite aux permutations des entiers de 0 à $n-1$, la complexité en moyenne de la position d'un tel entier est égale à $\frac{n}{2}$.

- **Miroir d'une liste :**

```
let rec miroir = fun
 [] -> []
 | t::q -> miroir q @ [t] ;;
```

$T(n) = T(n-1) + n - 1$, d'où $T(n) = \Theta(n^2)$.

Amélioration :

```
let miroir l =
 let rec miroir_aux l m = match l with
  | [] -> m
  | t::q -> miroir_aux q (t::m)
 in miroir_aux l [] ;;
```

La fonction `miroir_aux` est récursive terminale. `miroir_aux l m` déverse les éléments de l dans m. Le résultat est donc l renversée concaténée avec m. Le coût de `miroir_aux l m` est le nombre d'ajouts d'éléments, c'est-à-dire la longueur de l. Le coût de `miroir` est donc linéaire.

5 Le ou les appels récursifs portent sur des données de taille $\frac{n}{2}$

$T(n) = a\,T(\frac{n}{2}) + f(n)$. On convient que $T(0) = T(1) = 0$. Généralement, $a = 1$ ou 2.
Pour la résolution, on se limite au cas où n est une puissance de 2.
On pose $u_k = T(2^k)$. Alors $u_k = a\,u_{k-1} + f(2^k)$.

$\frac{u_j}{a^j} = \frac{u_{j-1}}{a^{j-1}} + \frac{f(2^j)}{a^j}$, d'où en sommant de $j = 1$ à k : $\boxed{u_k = a^k \sum_{j=1}^{k} \frac{f(2^j)}{a^j}}$.

Proposition. *On suppose $f(n) = \mathcal{O}(n^\beta)$.*
Si $a = 2^\beta$, alors $T(n) = \mathcal{O}(\log_2 n\, n^\beta)$.
Si $a > 2^\beta$, alors $T(n) = \mathcal{O}(n^{\log_2 a})$.
Si $a < 2^\beta$, alors $T(n) = \mathcal{O}(n^\beta)$.
Mêmes résultats avec des Θ.

preuve. On se limite au cas où n est une puissance de 2 : $n = 2^k$.
$u_k = T(2^k) = \mathcal{O}(a^k \sum_{j=1}^{k} \frac{2^{j\beta}}{a^j})$. On pose $b = \frac{2^\beta}{a}$.
Si $b > 1$, $\sum_{j=1}^{k} b^j = \mathcal{O}(b^k)$ donc $u_k = \mathcal{O}(2^{k\beta})$, d'où $T(n) = \mathcal{O}(n^\beta)$.
Si $b < 1$, $\sum_{j=1}^{k} b^j = \mathcal{O}(1)$, donc $u_k = \mathcal{O}(a^k)$, d'où $T(n) = \mathcal{O}(n^{\log_2 a})$.
Si $b = 1$, $\sum_{j=1}^{k} b^j = k$, donc $u_k = \mathcal{O}(ka^k)$, d'où $T(n) = \mathcal{O}\log_2 n\, n^\beta)$. •

6 L'algorithme comporte plusieurs appels récursifs

6.1 Estimation de la complexité

Un algorithme récursif comportant deux appels récursifs à des données de taille $n-1$ ou $n-1$ et $n-2$ sera de complexité exponentielle.
Exemple pour un appel à $n-1$ et un appel à $n-2$: $T(n) = T(n-1) + T(n-2) + f(n)$.
Faire l'arbre des appels récursifs.
On effectue de nombreux appels récursifs inutiles. Sans optimisation du programme, la complexité est exponentielle.
Exemple du calcul de la suite de Fibonacci : $u_n = u_{n-1} + u_{n-2}$. $u_0 = a$, $u_1 = b$. Le programme suivant (Fibonacci modulo p pour éviter les dépassements de capacité) est proscrit :

```
let rec fib n p = match n with
    0 -> 0
  | 1 -> 1
  | n -> (fib (n-1) p + fib (n-2) p) mod p ;;
```

Le calcul de `fib 50 p` ne termine pas en un temps raisonnable, alors qu'il ne s'agit que de 50 additions. La complexité est de l'ordre de la valeur calculée, c'est-à-dire géométrique de raison le nombre d'or : $(\frac{1+\sqrt{5}}{2})^n$.

6.2 Amélioration pour Fibonacci

6.2.1 Transformation en algorithme itératif

```
let fibo a b n =
  let u = ref a and uu = ref b in
  for k = 2 to n do uu := !u + !uu; u := !uu - !u done;
  !uu ;;
```

La complexité temporelle est linéaire, la complexité spatiale constante.

6.2.2 Transformation en algorithme récursif linéaire à deux variables

Pour calculer u_n à partir de $u_0 = a$ et $u_1 = b$, il suffit de calculer u_{n-1} avec $u_0 = b$ et $u_1 = a + b$.

```
let fibo n p =
 let rec fib_aux n a b = match n with
    0 -> a  | 1 -> b
  | n -> fib_aux (n-1) b ((a+b) mod p)
 in fib_aux n 0 1 ;;
```

Correction : on montre par récurrence sur n que `fib_aux n a b` calcule u_n sachant que $u_0 = a$, $u_1 = b$ et pour tout $n \geqslant 2$, $u_n = u_{n-1} + u_{n-2} \mod p$.
La complexité temporelle est linéaire, la complexité spatiale également.

Conclusion : On essaiera dans la mesure du possible de transformer un tel algorithme en algorithme itératif ou récursif avec un seul appel récursif. Cela pourra souvent se faire en augmentant le nombre de variables.

7 Algorithmes de tri quadratiques

7.1 Généralités

On dispose d'un tableau ou d'une liste dont les valeurs appartiennent à un ensemble totalement ordonné (des entiers, des flottants, des chaînes de caractère, etc) et on souhaite les trier, par exemple dans l'ordre croissant.

Dans le cas d'un tableau, type mutable, deux options sont possibles : ou bien trier en place, c'est-à-dire modifier les valeurs du tableau t passé en paramètre de façon à ce qu'à la fin du programme t soit trié, ou bien laisser t inchangé et créer un nouveau tableau contenant les mêmes valeurs que t mais qui soit trié.

Dans le cas d'une liste, type non mutable, la liste initiale ne change pas, donc le programme devra renvoyer une liste triée contenant les mêmes valeurs que la liste initiale.

Les trois algorithmes présentés ci-dessous sont de complexité quadratique, des algorithmes plus performants seront vus dans le chapitre suivant.

7.2 Tri bulle

7.2.1 Algorithme pour un tableau

On parcourt le tableau en échangeant $t[i]$ et $t[i+1]$ si $t[i] > t[i+1]$. Après un passage, l'élément maximum se trouve en dernière position. On recommence sur le tableau privé du dernier élément et ainsi de suite jusqu'à ce que le tableau soit trié, ce qui se produit nécessairement avant le $(n-1)$-ième passage.
Si aucun échange n'a été effectué lors d'un passage, le tableau est trié (on introduira donc une sentinelle pour le détecter et ne pas faire de passage inutile).
Exemple : trier $v = [3; 4; 1; 2]$.

7.2.2 Analyse

Un passage dans un tableau de taille k nécessite $k-1$ tests.
On démontre par récurrence sur p qu'au bout de p passages, le k-ième passage s'effectuant sur un tableau de taille $n-k+1$, les p plus grands éléments du tableau se trouvent bien placés.
L'algorithme se termine donc au pire au bout de $n-1$ passages (car le premier élément est nécessairement à la bonne place), donc nécessite $\sum_{k=1}^{n-1}(n-k) = \frac{n(n-1)}{2} = \Theta(n^2)$ tests. L'algorithme est quadratique.
Dans le pire des cas, il est en $\Theta(n^2)$ quand le tableau est trié dans l'ordre décroissant.

7.2.3 Codage pour un tableau

```
let un_passage (v,b) =
 let n = Array.length v in
 b := true;
 for k = 0 to n-2 do
  if v.(k) > v.(k+1) then
         (let aux = v.(k) in v.(k) <- v.(k+1); v.(k+1) <- aux; b:= false)
 done
;;
let tri_bulle_tableau v =
 let b = ref false in     (* b est la sentinelle *)
 while not !b do un_passage (v,b) done; v ;;
```

7.2.4 Codage pour une liste

```
let rec passage l booleen = match l with
   []  -> [], booleen
 | [a] -> [a], booleen
 | a::b::q ->
    if a <= b then let (l1, reponse) = passage (b::q) booleen in (a::l1, reponse)
              else let (l1, reponse) = passage (a::q) booleen in (b::l1, false)
;;
let rec tri_bulle l =
 let (s, verdict) = passage l true in
 if verdict then s else tri_bulle s
;;
```

7.3 Tri sélection

On extrait le minimum de la séquence, on le place en tête, et on poursuit avec le reste de la séquence jusqu'à la fin.

7.3.1 Analyse

L'extraction du minimum de k éléments nécessite $k-1$ comparaisons, donc le nombre de tests est égal à $\sum_{k=2}^{n}(k-1) = \frac{n(n-1)}{2} = \Theta(n^2)$. L'algorithme est quadratique, et effectue toujours le même nombre de comparaisons.

7.3.2 Codage pour un vecteur

Pour chaque indice $i \leqslant n-1$, on détermine l'indice k tel que $t[k] = \min_{i \leqslant j \leqslant n-1} t[j]$ puis on échange les cases d'indices i et k.

```
let tri_selection t =
 let n = Array.length t in
 for i = 0 to n-2 do
  let k = ref i in
  for j = i+1 to n-1 do if t.(j) < t.(!k) then k := j done;
  let temp = t.(i) in t.(i) <- t.(!k); t.(!k) <- temp
 done;
 t ;;
```

Exemple : `t=[|1;6;0;3|]`.
Après $i = 0$, `t` devient `[|0;6;1;3|]`.
Après $i = 1$, `t` devient `[|0;1;6;3|]`.
Après $i = 2$, `t` devient `[|0;1;3;6|]`.

7.3.3 Codage pour une liste

La fonction `min_reste` renvoie le minimum d'une liste et la liste privée de cet élément.
La fonction `tri_selection` fait appel à la fonction `min_reste` avant de s'appeler récursivement sur la liste privée du minimum.

```
let rec min_reste = function
   [] -> failwith "liste vide"
| [a]  -> a, []
| a::l -> let (b,s) = min_reste l in if a < b then (a, b::s) else (b, a::s)
;;
let rec tri_selection = function
  [] -> []
 | l -> let (x,t) = min_reste l in x::(tri_selection t) ;;
```

7.4 Tri insertion

A chaque étape, on insère un nouvel élément dans une séquence déjà triée. On peut le décrire comme le tri des joueurs de cartes.

7.4.1 Codage pour une liste

La fonction `insere : 'a -> 'a list -> 'a list` prend en argument un élément x et une liste triée l et renvoie une liste triée obtenu en insérant au bon endroit x dans l.
La fonction `tri_insertion : 'a list -> 'a list` réalise le tri en insérant la tête de la liste passée en argument dans la queue triée récursivement.

```
let rec insere x l = match l with
     [] -> [x]
 | b::q -> if x <= b then x::l else b::(insere x q) ;;
let rec tri_insertion = function
     [] -> []
 | a::l -> insere a (tri_insertion l) ;;
```

La terminaison et la correction des deux fonctions se montrent par récurrence sur la longueur de la liste.

7.4.2 Codage pour un vecteur

Algorithme : pour chaque indice i de 1 à $n-1$, on insère $t[i]$ dans la séquence triée $t[0], \ldots, t[i-1]$. L'insertion s'effectue en partant de l'indice $i-1$ et en plaçant l'élément à sa place après avoir décalé d'un cran vers la droite les éléments supérieurs à $t[i]$.
Invariant de boucle : après passage de i dans la boucle, le tableau $t[0..i]$ est trié.

```
let tri_insertion t =
 let n = Array.length t in
 for i = 1 to n-1 do
  let x = t.(i) and j = ref (i-1) in
  while !j >= 0 && t.(!j) > x do
   t.(!j +1) <- t.(!j); decr j
  done;
  t.(!j +1) <- x
 done;
 t ;;
```

Analyse : pour chaque indice i entre 1 et $n-1$, l'insertion de $t[i]$ demande au plus i comparaisons, donc la complexité dans le pire des cas est $\sum_{i=1}^{n-1} i = \Theta(n^2)$. L'algorithme est quadratique, mais n'effectue pas toujours le même nombre de comparaisons.