COMPLEXITE ELEMENTAIRE

Table des matières

COMPLEXITE ELEMENTAIRE

1 Introduction

Les études de complexité sont destinées à apporter des informations sur la performance (coût) des algorithmes en termes de besoins en ressources, de temps d'exécution, informations qui pourront

- faire préférer un algorithme à un autre ou inciter à chercher un algorithme plus performant,
- faire préférer un système de représentation (codage) à un autre,
- inciter à changer d'objectif lorsque l'on ne prévoit pas de réponse en un temps acceptable ...

Les exemples sont illustrés par des programmes écrits en OCaml.

1.1 Identification des opérations coûteuses

Complexité temporelle (coût en temps d'exécution)

- coûts de calculs numériques (par exemple, en ordre décroissant : **, * et /, + et -),
- coûts de comparaison,
- coûts de parcours de structures (parcours de listes, de graphes),
- coût de recopie pour la création de données intermédiaires,
- coût d'accès (surtout disque),
- ...

Complexité spatiale (coût en ressources de stockage)

- redondance dans les structures de données, données inutiles (matrices creuses),
- duplication de structures de données,
- appels récursifs avec paramètres volumineux,
- ...

1.2 Coûts cachés ou coûts indirects

Certaines opérations ont un coût qui peut dépendre du langage de programmation, de l'environnement ou de la machine sur laquelle on exécute l'algorithme.

- Les ressources, provenant d'une bibliothèque, sont parfois anciennes et peu optimales,
- Les fonctions mathématiques usuelles, suivant la précision interne, suivant la façon dont elles sont réalisées, matériellement ou non, peuvent conduire à des temps d'exécution importants. Il faut savoir que le coût du calcul de $\sin, \log, \cdots$ est le même que celui d'une exponentielle (**).
- Les opérations sur listes, comme @ (concaténation) ou "rev" (renversement) ou "list-length" (longueur), peuvent avoir un coût important (selon l'implantation de la structure de liste).
- Les variables locales de duplication, utilisées de façon intermédiaire, les paramètres (fonctions récursives) peuvent conduire à la saturation de la capacité de stockage.

> **"Ecriture courte" ne signifie pas forcément "exécution rapide"** , l'utilisation de fonctions élaborées (sin, @, "rev", list-length ...) peut se révéler très coûteuse en temps d'exécution.

1.3 Stratégies de réduction des coûts

Un peu d'expérience ne nuit pas. On pourra

- faire un choix pertinent de structures de données, utiliser des données modifiables sur place ... .
- choisir des sous-algorithmes efficaces, adaptés aux structures de données.
- utiliser des données intermédiaires qui augmentent le coût spatial en améliorant le coût temporel (exemple récursivité double). **Eviter l'abus de paramètres dans les fonctions récursives**.
- Remplacer une programmation récursive par une programmation itérative : les algorithmes récursifs (à écriture courte) sont souvent de gros consommateurs de mémoire.
- ...

1.4 Evaluation des coûts

On cherche à évaluer le coût exact (si possible), ou un **coût maximal (dans le pire des cas)** ou encore un **coût moyen** (qui est bien souvent le coût le plus significatif). Un calcul exact est rarement possible et on se contente souvent de donner un **coût majoré** (sans exagérer), décrit par un ordre de grandeur.

Puisque le calcul de coût sert à comparer des algorithmes, on veillera particulièrement à bien identifier les éléments intervenant dans les calculs de coût, afin de comparer des résultats qui sont exprimés dans les mêmes termes (unités et paramètres).

1.4.1 Coût temporel

Unité de coût : on choisit l'opération la plus coûteuse, en ignorant les autres (avec précaution cependant, puisque les . . . petits ruisseaux peuvent faire de grandes rivières . . .).

Paramètre(s) de coût : cela peut être la taille d'une structure de données, un nombre d'itérations ou une association de plusieurs valeurs.

1.4.2 Coût spatial

On a souvent trop tendance à ne s'intéresser qu'aux coûts de calcul mais il ne faut pas négliger les coûts spatiaux, d'autant plus qu'ils peuvent induire des coûts temporels de gestion des données ou conduire à des blocages par saturation.
D'autre part, on peut être amené à utiliser une stratégie d'augmentation du coût spatial (coût à évaluer et à maitriser) pour réduire un coût temporel.

2 Notations pour la mesure de complexité.

On utilise les notations mathématiques usuelles : $\sim$, O , Θ pour exprimer un ordre de grandeur de la complexité en fonction d'un paramètre lorsque celui ci tends vers $+\infty$.

Définition 2.1. (Rappel de la définition de O, en $+\infty$)

Soient f et g deux applications de $\mathbb{N}^*$ vers $\mathbb{R}_+$.
On dit que f est un O de g (au voisinage de $+\infty$), ce que l'on note $f = O(g)$, si il existe une constante $A > 0$ telle que $\exists n_0 \in \mathbb{N} : \forall n \geqslant n_0 \; f(n) \leqslant A\, g(n)$.

Remarque. la constante A n'étant jamais exprimée dans une écriture $f = O(g)$, cette relation peut cacher des réalités différentes. Par exemple $10000\,n^3 = O(n^3)$ au même titre que $2\,n^3 = O(n^3)$.

Remarque. La relation O est réflexive, transitive et compatible avec la multiplication par des fonctions strictement positives.

Définition 2.2. (Rappel de la définition de Θ, en $+\infty$)

Soient f et g deux applications de $\mathbb{N}^*$ vers $\mathbb{R}_+$.
On dit que f est un Θ de g (au voisinage de $+\infty$), ce que l'on note $f = \Theta(g)$, si on a à la fois $f = O(g)$ et $g = O(f)$.

Remarque. La relation Θ est une relation d'équivalence (réflexive, symétrique et transitive), compatible avec la multiplication par des fonctions strictement positives.

Hiérarchie :
$$\begin{cases} f \sim g & \text{donne un encadrement effectif : } \dfrac{1}{2}\,g \leqslant f \leqslant \dfrac{3}{2}\,g. \\ f = \Theta(g) & \text{assure l'existence d'un encadrement, sans le donner.} \\ f = O(g) & \text{assure seulement l'existence d'une majoration, sans la donner.} \end{cases}$$

Classes usuelles de complexité (avec $a > 1$) :

$$\Theta(1) < \Theta(\log n) < \Theta(n) < \Theta(n\log n) < \Theta(n^2) < \cdots < \Theta(n^k) < \Theta(n^k \log^h n) < \Theta(n^{k+1}) \ll \Theta(a^n)$$
$$O(1) < O(\log n) < O(n) < O(n\log n) < O(n^2) < \cdots < O(n^k) < O(n^k \log^h n) < O(n^{k+1}) \ll O(a^n)$$

Remarques. On utilise le plus souvent les évaluations en O, moins précises mais plus faciles à calculer.

- Pour $h < k$, un $O(n^h \log^p n)$ est aussi un $O(n^k)$. Mais ce n'est pas vrai avec Θ.
- Un algorithme dont le coût est un $O(n^k)$ est dit **"de coût polynomial"**.
- Ces notions de complexité sont au voisinage de $+\infty$ et peuvent être trompeuses : pour $n < 100$, $n\log n$ qui est un $O(n\log n)$, est plus performant que $1000\log n$ qui est seulement un $O(\log n)$.

3 Formulation des coûts (coût maximal majoré en général)

Le coût C est une fonction d'un ou plusieurs paramètres de coût, exprimée en unité de coût, et qui peut être définie par un calcul direct (pour les algorithmes itératifs) ou à partir d'une relation de récurrence (cas des algorithmes récursifs).

Pour que les choses soient claires, il faut préciser

- la nature de l'évaluation (temporelle ou spatiale) ;
- l'unité de coût u (en général, l'opération la plus coûteuse) ;
- le(s) paramètre(s) de coût : $(i,j), n, p \times q \cdots$;
- le mode d'évaluation : coût maximal (au pire) (majoré) ou coût en moyenne (majoré) ;
- le mode de calcul (donner une relation de récurrence, justifier les majorations ...) ;

puis donner une évaluation du coût en fonction du (des) paramètre(s) à l'aide de

- O , à défaut de mieux (c'est ce que l'on fait le plus souvent, car c'est le plus facile) ;
- Θ , si possible ;
- $\sim$, éventuellement.

> Le plus souvent, on se ramène à des cas simples, calculables, par majoration (légère) de l'évaluation des coûts, pour obtenir des coûts majorés qui soient un $O(n^k \log^h n)$ ou un $O(a^n)$ (références usuelles). Cependant, une majoration abusive du calcul des coûts rend difficile les comparaisons ....

On s'intéresse ici plus particulièrement à l'évaluation du coût (maximal), temporel ou spatial, des algorithmes récursifs, où la fonction de coût est définie par une relation de récurrence.

4 Situations élémentaires

T représente un coût majoré, qui peut s'exprimer par une formulation exacte et dont on donne éventuellement un résultat exact. Le coût réel (majoré) **C** (dont on ne connaît pas forcément une formulation exacte) vérifiera $C(n) \leqslant T(n)$ d'où $C(n) = O(T(n))$, et la relation O étant transitive ....

Avec des constantes a, b positives, des fonctions f positives,

T(n) = a T(n-1) + b par exemple : ($a = 1$, itération), ($a = 2$, jeu des tours de Hanoï)

- si $a > 1$ alors $T(n) = O\left(a^n\right)$
- si $a = 1$ alors $T(n) = O\left(n\right)$
- si $a < 1$ alors $T(n) = O\left(n\right)$ $\quad (T(n) \leqslant T'(n)$ avec $T'(n) = 1 \times T'(n-1) + b)$

T(n) = T(n-1) + b n par exemple : le tri par insertion, avec recherche séquentielle

- $T(n) = O\left(n^2\right)$ $\quad$ (plus précisément, $T(n) \sim \dfrac{b}{2} n^2$)

T(n) = a T(n/2) + b par exemple : la recherche dichotomique

- si $a > 1$ alors $T(n) = O\left(n^{\log_2 a}\right)$ $\quad \left(= O(n^q) \quad \text{avec } q = \log_2 a > 0\right)$
- si $a = 1$ alors $T(n) = O\left(\log_2 n\right)$
- si $a < 1$ alors $T(n) = O\left(\log_2 n\right)$ $\quad (T(n) \leqslant T'(n)$ avec $T'(n) = 1 \times T'(n/2) + b)$

T(n) = 2 T(n/2) + f(n) par exemple : méthodes "diviser pour régner"

- si $f(n) = O(n^p)$ alors
  - si $p < 1$, alors $T(n) = O\left(n\right)$
  - si $p = 1$, alors $T(n) = O\left(n \log_2 n\right)$
  - si $p > 1$, alors $T(n) = O\left(n^p\right)$
- sinon ...

4.1 Preuves

- **T(n) = T(n-1) + b** On a alors $T(n) = \underbrace{b+b+\cdots+b}_{n} + cte$, soit $T(n) \sim nb$.

- **T(n) = a T(n-1) + b** On a $\dfrac{T(n)}{a^n} = \dfrac{T(n-1)}{a^{n-1}} + \dfrac{b}{a^n}$ et $\dfrac{T(n)}{a^n} = b\sum_{k=0}^{n}\dfrac{1}{a^k} + cte.$

  Avec $a > 1$, on a une série convergente et on en déduit que $T(n) = O(a^n)$.

- **T(n) = T(n-1) + b n**

  Alors $T(n) = b\times n + b\times(n-1) + \ldots + b\times 2 + b + cte = b\sum_{k=1}^{n} k + cte = b\dfrac{n(n+1)}{2} + cte \sim \dfrac{b}{2}n^2 = O(n^2).$

- **T(n) = a T(n/2) + b**

  On traite d'abord le cas où $n = 2^k$:

  En divisant par a^k, $\dfrac{T(2^k)}{a^k} = \dfrac{T(2^{k-1})}{a^{k-1}} + b\dfrac{1}{a^k}$ et on pose $x_k = \dfrac{T(2^k)}{a^k}$.

  La suite (x_k) vérifie $x_k = x_{k-1} + b\dfrac{1}{a^k}$ et on en déduit que $x_k = b\sum_{s=0}^{k}\left(\dfrac{1}{a}\right)^s + cte.$

  si $a > 1$, $x_k = b\dfrac{1-\frac{1}{a}^{k+1}}{1-\frac{1}{a}} + cte \sim \dfrac{b}{1-\frac{1}{a}}$ et $T(n) = T(2^k) = a^k x_k \sim a^k\dfrac{b}{1-\frac{1}{a}}$.

  Comme $a^k = a^{\log_2 n} = e^{\log_2 n \ln a} = e^{\ln n \log_2 a} = n^{\log_2 a}$, $T(n) \sim \dfrac{b}{1-\frac{1}{a}} n^{\log_2 a} = O\left(n^{\log_2 a}\right)$.

  si $a = 1$, $x_k = b(k+1) + cte \sim b\log_2 n$ et $T(n) = T(2^k) = x_k \sim b\log_2 n = O(\log_2 n)$.

  Puis pour n quelconque, on procède par encadrement :

  Avec k tel que $2^k \leqslant n < 2^{k+1}$, par croissance de T, on a $T(2^k) \leqslant T(n) \leqslant T(2^{k+1})$ et $T(n) = O\left(T(2^{k+1})\right)$, donc,

  si $a > 1$, on a $T(n) = O(n^{\log_2 a})$ et, si $a = 1$, on a $T(n) = O(\log_2 n)$.

- **T(n) = 2 T(n/2) + f(n)**, et, plus généralement, **T(n) = a T(n/2) + f(n)**, avec $f(n) = O(n^p)$.

  On définit q par $a = 2^q$ et, f étant un $O(n^p)$, quitte à majorer, on prends $f(n) = An^p$.

  On traite d'abord le cas où $n = 2^k$:

  En divisant par $a^k = 2^{kq}$, $\dfrac{T(2^k)}{2^{qk}} = \dfrac{T(2^{k-1})}{2^{q(k-1)}} + A\,2^{kp}\dfrac{1}{2^{kq}}$ et on pose $x_k = \dfrac{T(2^k)}{2^{qk}}$.

  La suite (x_k) vérifie $x_k = x_{k-1} + A\,2^{k(p-q)}$ et on en déduit que $x_k = A\sum_{s=0}^{k}\left(2^{p-q}\right)^s + cte$.

  si $p > q$, $x_k = A\dfrac{1-2^{(p-q)(k+1)}}{1-2^{p-q}} + cte \sim A\dfrac{2^{(p-q)(k+1)}}{2^{p-q}-1}$

  et $T(n) = T(2^k) = 2^{kq}x_k \sim 2^{kp}A\dfrac{2^{p-q}}{2^{p-q}-1} = O(2^{kp}) = O(n^p)$.

  si $p = q$, $x_k = A(k+1) + cte \sim Ak \sim A\log_2 n$

  et $T(n) = T(2^k) = a^k x_k = 2^{kp}x_k \sim n^k A\log_2 n = O(n^p\log_2 n)$.

  si $p < q$, $x_k = A\dfrac{1-\left(\frac{1}{2}\right)^{(q-p)(k+1)}}{1-\left(\frac{1}{2}\right)^{q-p}} \sim \dfrac{A}{1-\left(\frac{1}{2}\right)^{q-p}}$

  et $T(n) = T(2^k) = a^k x_k = 2^{kq}x_k \sim 2^{kq}\dfrac{A}{1-\left(\frac{1}{2}\right)^{q-p}} = O(2^{kq}) = O(n^q)$.

  Puis pour n quelconque, on procède par encadrement :

  Avec k tel que $2^k \leqslant n < 2^{k+1}$, par croissance de T, on a $T(2^k) \leqslant T(n) \leqslant T(2^{k+1})$ et $T(n) = O\left(T(2^{k+1})\right)$ d'où le résultat.

4.2 Exemples

4.2.1 T(n) = a T(n-1) + b

Somme des éléments d'un vecteur, d'une liste (d'entiers).

```
let vect_somme a =        (* somme des éléments du vecteur a *)
   let s = ref 0 in for i = 0 to (Array.length a)-1 do s := !s + a.(i) done; !s
and list_somme =          (* somme des éléments de la liste a (argument "fantôme") *)
   let rec aux s = function
     | [] -> s
     | h::q -> aux (s+h) q
   in aux 0
;;
(* val vect_somme : int array -> int = <fun>
   val list_somme : int list -> int = <fun> *)
```

Pour le calcul du coût temporel (ou complexité temporelle), on prend comme unité de coût, le coût d'une addition et comme paramètre de coût la taille n du vecteur ou de la liste. $C(n) = C(n-1)+1$ d'où $C(n) = O(n)$.

Tours de Hanoi

Enoncé : On dispose de n rondelles, de rayons différents, que l'on enfile sur trois piquets A B et C, de façon à ce qu'une rondelle ne soit jamais au dessus d'une autre rondelle de rayon inférieur.
Les rondelles étant initialement enfilées sur le piquet A, en faisant passer ces rondelles, une par une, d'un piquet à un autre, on cherche à transférer toutes les rondelles sur le piquet C.

Solution : On sait résoudre le problème pour 0 rondelles (et même pour 1 rondelle) et, pour déplacer une pile de $n \geqslant 1$ rondelles de A vers C, en utilisant B comme intermédiaire, il suffit de

- déplacer une pile de $n-1$ rondelles de A vers B en utilisant C comme intermédiaire (en final C sera vide, la rondelle de rayon maximal sera seule sur A)
- déplacer la rondelle de rayon maximal qui est restée sur A, vers C qui est vide (en final A sera vide, la rondelle de rayon maximal sera seule sur C)
- déplacer une pile de $n-1$ rondelles de B vers C en utilisant A comme intermédiaire

```
let hanoi a0 =   (* le contenu d'un piquet est une liste de rondelles *)
   let a = ref a0 and b = ref [] and c = ref [] in
   let rec shanoi depart intermediaire arrivee = function
      | 0  -> ()
      | n  ->  shanoi depart arrivee intermediaire (n-1);
               transfere depart arrivee;
               shanoi intermediaire depart arrivee (n-1);
   and    transfere p1 p2 = match !p1 with
      | []    -> ()
      | e::r  -> p1 := r; p2 := e::!p2;
                 imprimer ();                    (* imprimer les intermédiaires *)
   and    imprimer = function () -> imprime !a; imprime !b; imprime !c; print_newline ()
   and    imprime = function
      | []    -> print_string " --- "
      | e::r  -> print_int e; imprime r          (* donc rondelles de type int *)
   in
   shanoi a b c (List.length !a);
   (!a, !b, !c)
(* val hanoi : int list -> int list * int list * int list = <fun> *)
```

Exemple : `hanoi [1;2;3;4;5;6];; (* int list * int list * int list = [],[],[1;2;3;4;5;6] *)`

PREUVE : ... elle est donnée par la solution (récurrence).

Complexité temporelle : (exacte ici) (sans l'impression)
unité de coût : transfert d'une rondelle ; paramètre de coût : nombre n de rondelles.
On a $C(n) = 2 \times C(n-1)+1$ d'où $C(n) = O(2^n)$ et même $C(n) = 2^n - 1$.

4.2.2 T(n) = T(n-1) + b n

Tri par insertion d'une liste.

Principe : soient deux listes L_1 et L_2, L_1 étant déjà triée, on extrait le premier élément de L_2 pour l'insérer à la bonne place dans L_1.

```
let tri_insertion ordre =          (* fonction d'ordre comme paramètre *)
  let rec tri_insert l1 = function
    | []   -> l1
    | e::r -> tri_insert (insere ordre e l1) r
  and     insere ordre e = function
    | []   -> [e]
    | x::r -> if ordre e x then e::x::r else x::(insere ordre e r )
  in tri_insert []
;;
(* val tri_insertion : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun> *)
tri_insertion ( < ) [12;6;15;10;0;13;3;25;17;4];;
```

PREUVE : 1. de l'insertion ... 2. du tri, avec comme invariant "L_1 triée" ... .

Complexité temporelle (Unité : comparaison de deux éléments. Paramètre : taille de liste.)

- Coût maximal $D(k)$ de "insere" (insertion de e dans une liste triée à k éléments) :
  $D(0) = 0$; $D(k) = 1 + D(k-1)$ (dans le pire des cas on insère dans la queue de liste) d'où $D(k) = k$
- Coût maximal $C(n)$ de "tri_insert" (donc de "tri_insertion") pour une liste à n éléments :
  $C(0) = 0$; $C(n) = D(n) + C(n-1) = n + C(n-1)$ (à n fixé, on insère la tête dans une liste d'au plus n éléments (majoration large) et on trie le reste qui a $n-1$ éléments. D'où $C(n) = O(n^2)$

Remarques.

- On a majoré très largement : insertion dans le pire des cas et surtout insertion dans une liste de taille n alors que cette liste a une taille qui varie de 0 à n.
- On montre que la complexité moyenne du tri par insertion est aussi un $O(n^2)$, comme pour le coût maximal (mais avec une constante "cachée" plus faible).

Tri rapide d'une liste.

Principe : à partir de l'élément a de tête de liste, partager le reste de la liste en deux listes, une des plus petits que a, une des plus grands que a, que l'on trie suivant le même principe avant de les recoller.

```
let rec tri_rapide ordre = function
  | [] -> []
  | a::q -> let (u,v) = partage ordre a q
            in (tri_rapide ordre u) @ (a:: (tri_rapide ordre v ))
and    partage ordre a = function
  | [] -> ([],[])
  | e::q -> let (u,v) = partage ordre a q
            in if ordre e a then ((e::u), v) else (u,(e::v))
;;
(* val tri_rapide : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun>
   val partage : ('a -> 'a -> bool) -> 'a -> 'a list -> 'a list * 'a list =  <fun> *)
tri_rapide ( < ) [12;6;15;10;0;13;3;25;17;4];;
```

PREUVE : ... par induction.

Complexité temporelle (Unité : comparaison de deux éléments. Paramètre : taille de liste.)

- Le coût de concaténation @ de deux listes de taille k et $n-k$ est au pire égal à n accès et 0 comparaisons.
- Coût maximal $D(k)$ de "partage" d'une liste de k éléments : $D(k) = 1 + D(k-1)$ d'où $D(k) = k = O(k)$.
- Coût maximal $C(n)$ de "tri-rapide" (tri d'une liste à n éléments) :
  Le pire des cas est de tenter de trier une liste déjà triée. Le coût est donc maximal lorsque la liste est déjà triée, où l'on partage en une liste de longueur 0 et une liste de longueur $n-1$.
  $C(n) = 0 + D(n-1) + C(n-1)$ ou $C(n) = n - 1 + C(n-1)$ et $C(n) = O(n^2)$

Remarque. Ce tri a le même type de complexité maximale que le tri par insertion, cependant on montre (voir plus loin) que sa complexité moyenne est un $O(n \log n)$, ce qui le rend intéressant en pratique (pratiquement équivalents aux meilleurs).

4.2.3 T(n) = a T(n/2) + b

Exponentiation rapide, à exposants entiers.

Principe : avec $n = 2p + r$ où $r \in \{0,1\}$, on écrit $x^n = x^r * (x^p) * (x^p)$ et on calcule de même x^p

(le principe est valable dans un anneau, par exemple pour calculer la puissance $n^{\text{ième}}$ d'une matrice carrée).

```
let rec puissance x  = function     (* version récursive *)
   | 0 -> 1.0
   | n -> let u = puissance x (n/2)
          in if (n mod 2) = 0
          then u *. u else x *. u *. u
;;
(* puissance : float -> int -> float = <fun> *)
puissance 2. 10;;
```

Complexité temporelle : unité de coût : ***.** , paramètre de coût : n = exposant.
Le coût de la division entière par 2 étant négligé (il s'agit d'un simple décalage de la représentation binaire d'un entier), on a naturellement un coût maximal qui vérifie $C(n) = C(n/2) + 2$, d'où :

$C(n) = O(\log_2 n)$ **unités** : Puissance entière de float : l'unité est une multiplication de float. Puissance entière de matrices de float : l'unité est n^3 multiplications de float (améliorable avec un algorithme optimal de multiplication matricielle).

Complexité spatiale : unité de coût : float /matrice de float ; paramètre : n = exposant.
$S(n) = 1 + S(n/2)$, d'où : $S(n) = O(\log_2 n)$ **unités** (float /matrice de float).

Remarque Avec une programmation **itérative** la complexité spatiale sera constante !

Recherche dichotomique dans un vecteur trié, selon l'ordre du tri

Principe : Dans un vecteur **trié**, selon l'ordre "`ordre`", comparer l'élément cherché à l'élément médian et, en cas de différence, le chercher dans le demi-vecteur de gauche ou le demi-vecteur de droite.

```
let vect_cherche ordre v e =                          (* v trié selon ordre *)
  let i = ref 0 and j = ref (Array.length v -1) in
  while (!j - !i) > 0
  do if e = v.(!i) then j := !i
     else let m = (!i + !j)/2 in if ordre e v.(m) then j := m else i := m+1 done;
  v.(!i) = e
;;
(* val vect_cherche : 'a array -> ('a -> 'a -> bool) -> 'a -> bool = <fun> *)
vect_cherche (<=) [|0; 3; 4; 6; 10; 12; 13; 15; 17; 25|] 3;;
```

Complexité temporelle : unité de coût : comparaison, paramètre de coût : n = taille du vecteur.
Le coût d'accès à un élément de vecteur est constant et les coûts de calculs et d'affectation sur les entiers sont supposés négligeables devant le coût de comparaison d'un type 'a (supposé plus lourd qu'un type entier). Le coût maximal vérifie $C(n) = C(n/2) + 1$, d'où $C(n) = O(\log_2 n)$.

Remarque. Si on effectue le même type de recherche dans un vecteur non déjà trié, il faut chercher (au pire) successivement dans les deux moitiés, ce qui conduit à : $C(n) = 2C(n/2) + 1$, d'où $C(n) = O(n)$, qui est le temps de parcours maximal normal avec comparaison.

Tri par insertion d'un vecteur, avec recherche séquentielle/dichotomique dans un sous-vecteur trié.

Principe : Le tri, croissant, par insertion, d'un tableau t de n éléments consiste à rechercher itérativement la place d'insertion de l'élément n°i dans le sous-tableau précédent (indicé de 1 à $i-1$) supposé trié et à l'insérer (après décalage) à sa place. On commence à partir du deuxième élément et on termine quand tous les éléments ont été placés (le tri se fait sur place, dans le vecteur lui même).

Le coût du décalage (sans comparaison) étant négligé, si la recherche dans le sous-tableau est

- séquentielle (de coût $i-1$), le coût au pire du tri vecteur par insertion est un $O(n^2)$:
$$\sum_{i=2}^{n}(i-1) = \frac{(n-1)n}{2} = O(n^2)\,;$$
- dichotomique (de coût $\log_2(i-1)$), le coût au pire du tri vecteur par insertion est un $O(n \log_2 n)$:
$$\sum_{i=2}^{n}\log_2(i-1) \leqslant \int_1^n \log_2 x\,dx \leqslant n\log_2 n = O\big(n\log_2 n\big)\,.$$

4.2.4 T(n) = 2 T(n/2) + f(n) (ou T(n) = a T(n/2) + f(n))

Tri fusion d'une liste.

Principe : On divise la liste L, de taille n, à trier, en deux listes L_1 et L_2 de taille $\approx \frac{n}{2}$ que l'on trie selon le même principe avant de les fusionner dans l'ordre.

```
let rec tri_fusion ordre = function
  | []  -> []
  | [e] -> [e]
  | l   -> let rec divise = function
               | []       -> [], []
               | [e]      -> [e], []
               | a::b::r  -> let m1, m2 = divise r in a::m1, b::m2
           and     fusion x y = match x, y with
               | [], l -> l
               | l, [] -> l
               | (a1::r1 as l1), (a2::r2 as l2) -> if ordre a1 a2
                                                   then a1::(fusion r1 l2)
                                                   else a2::(fusion l1 r2)
           in let l1,l2 = divise l
           in fusion (tri_fusion ordre l1) (tri_fusion ordre l2)
;;
(* val tri_fusion : ('a -> 'a -> bool) -> 'a list -> 'a list = <fun> *)
tri_fusion ( > ) [1; 12; 2; 3; -1; 4; 14; 8; 10; 12; 21; 15; 7; 17; 11; 10];;
```

PREUVE : par induction ... .

Complexité temporelle (maximale) : unité de coût = comparaison ; paramètre : taille n de liste.

Division en deux listes : coût négligé.
Fusion avec comparaison de deux listes de taille $k_1+k_2=k$: $T(k)=1+T(k-1)=k$
On a donc $C(n)=C(n/2)+C(n-n/2)+T(n)$, soit $C(n)=2C(n/2)+n$, d'où $C(n)=O(n\log_2 n)$.

Tri rapide d'un vecteur, complexité en moyenne.

Principe : Partager le vecteur à trier (s'il a au moins deux éléments) en trois sous-vecteurs, le premier comprenant uniquement des éléments inférieurs ou égaux à un élément (l'élément pivot), le deuxième ne comportant qu'un seul élément (l'élément pivot) et le troisième contenant uniquement des éléments supérieurs ou égaux à l'élément pivot. Puis réappliquer récursivement le tri rapide sur les premier et troisième sous-vecteurs, l'élément pivot étant, lui, à sa place définitive (le tri se fait sur place, dans le vecteur lui même).

La partition d'un sous-vecteur compris entre les indices i et j, prend $t[i]$ comme élément pivot, déplace les éléments inférieurs ou égaux au pivot en début de sous-vecteur, les éléments supérieurs ou égaux au pivot en fin de sous-vecteur, positionne le pivot à sa place définitive et retourne l'indice de cette place.

Pour réaliser concrêtement la partition d'un sous-vecteur, on utilise deux parcours, l'un (*montant*) allant du début du sous-vecteur vers la fin, et l'autre (*descendant*) allant de la fin vers le début. Le parcours *montant* est suspendu quand un élément sera strictement supérieur au pivot ; le parcours *descendant* est suspendu quand un élément sera strictement inférieur au pivot. Les éléments seront alors échangés et les parcours reprennent jusqu'à ce qu'ils se rejoignent. La place du pivot est alors déterminée, le pivot y est mis, et l'indice de sa place est retourné.

```
let partition ordre t i j =
  let x = t.(i) and i1 = ref (i+1) and j1 = ref j in
  while !i1 <= !j1
  do while (!i1 <= !j1) && ( (ordre t.(!i1) x) || (t.(!i1) = x) )
     do t.(!i1-1) <- t.(!i1); incr i1 done;
     while (!j1 >= !i1) && ( (ordre x t.(!j1)) || (t.(!j1) = x) )
     do decr j1 done;
     if !i1 < !j1 then let v = t.(!i1) in t.(!i1) <- t.(!j1); t.(!j1) <- v ;
  done;
  t.(!j1) <- x;
  !j1
;;
(* val partition : ('a -> 'a -> bool) -> 'a array -> int -> int -> int = <fun> *)
```

```
let rec quicksort ordre t i j =      (* tri du vecteur t, de i à j, dans lui même *)
  if i >= j then ()
  else let k = partition ordre t i j in
       quicksort ordre t i (k-1);
       quicksort ordre t (k+1) j
;;
(* val quicksort : ('a -> 'a -> bool) -> 'a array -> int -> int -> unit = <fun> *)

let t = [|100; 7;5;1;12;3;24;25;8;10;2;14;7;2;9;4;17 |];;
quicksort (<) t 0 (Array.length t -1);;
t;;
```

PREUVE : 1. `partition` (par invariant) ... 2. `quicksort` (par induction) ... .

Complexité temporelle : unité de coût = comparaison ; paramètre : taille n du vecteur.

- Le coût de `(partition t i j)` est $j-i$.
- Le coût $C(n)$ de `(quicksort t 1 n)` vérifie $C(n)=n+C(k)+C(n-k)$, avec $C(1)=0$.
  - Au pire, on a $C(n)=n+C(n-1)$, ce qui donne un coût au pire en $O\left(n^2\right)$
  - En moyenne, on doit avoir $C(n)=n+2C(n/2)$, ce qui donne un coût moyen en $O\left(n\log_2 n\right)$.

Multiplication rapide des polynômes (Karatsuba, 1960).

Principe : Soient P et Q deux polynômes, n le plus grand de leurs degrés et $m=\left\lceil\frac{n}{2}\right\rceil$ (partie entière supérieure).

En écrivant $\begin{cases}P=X^m P_1+P_2, & d^oP_2<m\\ Q=X^m Q_1+Q_2, & d^oQ_2<m\end{cases}$ on a : $PQ=X^{2m}P_1Q_1+X^m(P_1Q_2+P_2Q_1)+P_2Q_2$

et cette formule donne un temps de calcul en $O(n^2)$ (4 produits de polynômes de degré $n/2$).

Karatsuba a proposé une amélioration qui conduit à un temps de calcul en $O(n^{\log_2 3})\approx O(n^{1.585})$:

En posant $\begin{cases}S_1=P_1Q_1,\\ S_2=P_2Q_2\\ S_3=(P_1+P_2)(Q_1+Q_2)\end{cases}$ on a $P_1Q_2+P_2Q_1=S_3-S_2-S_1$

et on peut écrire : $\boxed{PQ=X^{2m}S_1+X^m(S_3-S_2-S_1)+S_2}$

En calculant selon cette formule, on n'utilise que 3 produits de polynômes de degré $n/2$.

Complexité temporelle : $\begin{cases}\text{unité de coût : \textbf{multiplication} de deux coefficients (par exemple de type } \texttt{float}).\\ \text{paramètre de coût : } n=max(deg(P),deg(Q)) \text{ (pour une complexité au pire).}\end{cases}$

- Le coût de l'addition est négligeable devant le coût de la multiplication.
- Le calcul de P_1, P_2, Q_1, Q_2 a un coût négligeable (décalages et séparation) et ces polynômes sont de degré $n/2$.
- La multiplication par X^k a un coût négligeable (décalages, additions d'entiers, ...).
- $\begin{cases}\text{Calcul de } S_1=P_1Q_1 & : C(n/2)\\ \text{Calcul de } S_2=P_2Q_2 & : C(n/2)\\ \text{Calcul de } S_3=(P_1-P_2)(Q_1-Q_2) & : C(n/2)\end{cases}$

On a donc $C(n)=3C\left(\frac{n}{2}\right)$ au pire (formule $T(n)=aT(n/2)+b$), donc $\boxed{C(n)=O(n^{\log_2 3})=O(n^{1.58})}$

Remarque. Le calcul nécessite aussi au plus $4n$ additions de coefficients : $2\left(n+\frac{n}{2}+\frac{n}{4}+\cdots\right)\leqslant 4n$.

Complexité spatiale : à évaluer, cela dépendra du choix de la structure de données et de l'organisation des données.

Mise en œuvre complexe : il faut définir

- une structure de données et des règles de conduite
  - vecteur ou liste de coefficients, indicé par les degrés croissants
  - liste de `{coeff= ; degre= }` ordonnée par degrés décroissant (polynômes creux)
  - ...
- des fonctions usuelles : degré, addition, soustraction, produit par X^n, quotient et reste dans la divison par X^n, ...

A voir dans un autre exposé ...

5 Formulations de coûts plus générales

T(n) = a T(n-1) + f(n)

- si $a = 1$ alors
  - si $f(n) = O(n^p)$ alors $T(n) = O(n^{p+1})$
  - si $f(n) = O(b^n)$ avec $b > 1$ alors $T(n) = O(b^n)$
- si $a > 1$ alors
  - si $f(n) = O(n^\alpha \log^\beta n)$ alors $T(n) = O(a^n)$
  - si $f(n) = O(a^n)$ alors $T(n) = O(n\,a^n)$
  - si $f(n) = O(b^n)$ alors $T(n) = O((\max(a,b))^n)$
- si $a < 1$ alors $T(n) \leqslant T'(n)$ avec $T'(n) = T'(n-1) + f(n)$ (majoration qui ramène au cas $a = 1$)

T(n) = a T(n/2) + f(n), avec $f(n) = O(n^p)$ et $a = 2^q$ (ou $q = \log_2 a$).

- si $p < q$, alors $T(n) = O(n^q)$
- si $p = q$, alors $T(n) = O(n^q \log_2 n)$
- si $p > q$, alors $T(n) = O(n^p)$

Preuves :

- récurrence $T(n) = a\,T(n-1) + f(n)$:
  $\frac{T(n+1)}{a^{n+1}} = \frac{T(n)}{a^n} + \frac{f(n)}{a^{n+1}}$ d'où
  - si $a = 1$, $T(n) = T(1) + \sum_{k=1}^{n-1} f(k)$
    - si $f(n) = O(n^p)$ alors $\sum_{k=1}^{n-1} f(k) = O(n^{p+1})$
      puisque $\sum_{k=1}^{n-1} f(k) \leqslant \sum_{k=1}^{n-1} A\,k^p \leqslant \sum_{k=1}^{n-1} A\,n^p = A\,n^{p+1}$
    - si $f(n) = O(b^n)$ alors $\sum_{k=1}^{n-1} f(k) = O(b^n)$
      puisque $\sum_{k=1}^{n-1} f(k) \leqslant \sum_{k=1}^{n-1} A\,b^k \leqslant \sum_{k=0}^{n-1} A\,b^k = A\,\frac{1-b^n}{1-b}$
  - si $a > 1$, $\frac{T(n)}{a^n} = \frac{T(1)}{a} + \sum_{k=1}^{n-1} \frac{f(k)}{a^{k+1}}$
    - si la série converge alors $T(n) = O(a^n)$
      et la série converge en particulier pour $f(n) = O(n^\alpha \log^\beta n)$ ou $f(n) = O(b^n)$ avec $b < a$
    - si $f(n) = O(a^n)$ alors $T(n) = O(n\,a^n)$
    - si $f(n) = O(b^n)$ avec $b > a$, alors $T(n) = a^n\,O\left(\left(\frac{b}{a}\right)^n\right) = O(b^{n+1}) = O(b^n)$
  - si $a < 1$, on majore fortement, avec $a = 1$
- récurrence $T(n) = a\,T(n/2) + f(n)$:
  Déjà fait précédement : la démonstration de $T(n) = 2\,T(n/2) + f(n)$ a été faite dans le cas plus général où $T(n) = a\,T(n/2) + f(n)$.

6 Complexité en moyenne (exemples).

6.1 Complexité en moyenne du tri rapide de listes.

Le coût temporel maximal (dans le pire des cas) du tri rapide d'une liste de n éléments, est un $O(n^2)$ (voir ci-dessus).
Après partage en deux groupes, si le premier groupe est de taille k, on a, dans un tel cas, comme coût moyen de tri : $n-1+E(k)+E(n-1-k)$, avec $\begin{cases} n-1 \text{ coût de partage (parcours de la liste)} \\ E(k) \text{ coût moyen de tri d'une liste de } k \text{ éléments} \end{cases}$

Comme le premier groupe peut être de taille $0, 1, \cdots n-1$, de façon équiprobable pour des listes aléatoires,

le coût moyen $E(n)$ vérifie : $E(n) = \dfrac{1}{n}\displaystyle\sum_{k=0}^{n-1}[n-1+E(k)+E(n-1-k)]$, avec $E(0)=0$.

On peut écrire $E(n) = n-1+\dfrac{2}{n}\displaystyle\sum_{k=0}^{n-1}E(k)$, d'où $\begin{cases} nE(n) = n(n-1)+2\displaystyle\sum_{k=0}^{n-1}E(k) \\ (n+1)E(n+1) = (n+1)n+2\displaystyle\sum_{k=0}^{n}E(k) \end{cases}$

Par différence, on obtient $(n+1)E(n+1)-nE(n) = 2n+2E(n)$, d'où $(n+1)E(n+1) = (n+2)E(n)+2n$.

Ainsi $\dfrac{E(n+1)}{n+2} = \dfrac{E(n)}{n+1}+\dfrac{4}{n+2}-\dfrac{2}{n+1}$ et, par sommation télescopique, (et en revenant en n),

$$\begin{aligned} \frac{E(n)}{n+1} &= \frac{E(1)}{2}+4\sum_{k=1}^{n-1}\frac{1}{k+2}-2\sum_{k=1}^{n-1}\frac{2}{k+1} \\ &= \frac{E(1)}{2}+4\left(-\frac{3}{2}+\sum_{k=1}^{n+1}\frac{1}{k}\right)-2\left(-1-\frac{1}{n+1}+\sum_{k=1}^{n+1}\frac{1}{k}\right) = -4+\frac{2}{n+1}+2\sum_{k=1}^{n+1}\frac{1}{k} \end{aligned}$$

$$E(n) = -2n+2(n+1)\left(-1+\sum_{k=1}^{n+1}\frac{1}{k}\right) \underset{+\infty}{\sim} 2n\ln n$$

Le coût moyen, en temps, du tri rapide d'une liste de taille n est $E(n) \sim 2n\ln n = O(n\ln n)$

Remarque. La constante cachée par la notation O est faible.

6.2 Complexité moyenne de la recherche linéaire d'éléments finis.

On recherche un élément x dans un vecteur ou dans une liste, non triés, de n éléments, avec un coût temporel à évaluer selon $\begin{cases} \text{l'unité de coût temporel : comparaison de deux valeurs ;} \\ \text{le paramètre : } n. \end{cases}$.

On sait que le coût temporel maximal est n (dans le pire des cas x n'est pas présent et il faut cependant tester tous les éléments pour le savoir).

Evaluation du coût temporel moyen :

On suppose que **les éléments appartiennent à un ensemble de cardinal** p, par exemple $[\![1\cdots p]\!]$. Il y a alors p^n vecteurs ou listes possibles de n éléments.

- Cas spécial : x est absent ou se trouve en dernière position.
  Le coût de recherche est n et il y a $(p-1)^{n-1}p$ vecteurs ou listes qui présentent ce cas (les $n-1$ premiers éléments sont à prendre parmi $p-1$ éléments distincts de x et le dernier est quelconque.
- Cas intermédiaire : x se trouve en position k, avec $1 \leqslant k < n$.
  Le coût de recherche est égal à k et il y a $(p-1)^{k-1}p^{n-k}$ vecteurs ou listes qui présentent ce cas

Le coût moyen $E(n)$ vérifie donc : $E(n) = \dfrac{1}{p^n}\left[n(p-1)^{n-1}p+\displaystyle\sum_{k=1}^{n-1}k(p-1)^{k-1}p^{n-k}\right] = \cdots = p-\dfrac{(p-1)^n}{p^n-1}$.

On a donc $E(n) < p$ et $E(n) \xrightarrow[n\to+\infty]{} p$: le temps moyen de recherche est pratiquement constant

7 Exemples de réduction de la complexité ou de stratégies alternatives

On peut parfois réduire fortement la complexité temporelle ou spatiale, en utilisant des données intermédiaires, ou en changeant de concept. Parfois il faut aussi réduire ses prétentions ...

7.1 Suites de Fibonacci

Soit la suite de Fibonacci, définie par : $\begin{cases} F_0 = a \; ; \; F_1 = b \\ F_{n+2} = F_{n+1} + F_n \end{cases}$, pour $n \geqslant 0$.

1. Algorithme récursif, selon la définition brute (de complexité temporelle ou spatiale **exponentielle**) :

```
let fibonacci_1 a b n =
  let rec fibo = function
     | 0 -> a | 1 -> b
     | n -> fibo (n-1) + fibo (n-2)
  in fibo n;;
```

   - Complexité temporelle (en fonction du coût de l'addition "+").
     On a $\begin{cases} T(0) = 0 \; ; \; T(1) = 0 \\ T(n) = 1 + T(n-2) + T(n-1) \end{cases}$ et avec $u_n = T(n) + 1$, $\begin{cases} u_0 = 1 \; ; \; u_1 = 1 \\ u_{n+2} = u_{n+1} + u_n \end{cases}$, pour $n \geqslant 0$,
     qui est une suite (de Fibonacci !) définie par une relation de récurrence linéaire à coefficients constants.
     Avec $\alpha = \dfrac{1+\sqrt{5}}{2}$ et $\beta = \dfrac{1-\sqrt{5}}{2}$, on a : $u_n = \dfrac{\alpha^n - \beta^n}{\sqrt{5}} = O(\alpha^n)$, d'où $T(n) = O(\alpha^n)$ **additions**.
   - Complexité spatiale : unité de coût = taille d'un entier (paramètre).
     On a la même formule, donc cette complexité $S(n)$ vérifie aussi $S(n) = O(\alpha^n)$ **taille d'entier**.

2. Algorithme itératif, avec souvenir et glissement (de complexité temporelle linéaire) :

```
let fibonacci_2 a b n =
   let x  = ref (b-a) and y = ref a in
   for i = 1 to n do let z = !x in x := !y; y := z + !y done;
   !y;;
```

   - Complexité temporelle : unité : "+" (coût de "glissement" négligé). $C(n) \sim n = O(n)$ **additions**.
   - Complexité spatiale : $S(n) =$ cte **taille d'entier**.

3. Algorithme récursif, sous forme matricielle (équivalent en temps à la forme itérative) :

```
let fibonacci_3 a b n =
  let rec fibo_mat = function
     | 0  -> (a,b)
     | n -> let (u,v) = fibo_mat (n-1) in (v, u+v)
  in fst( fibo_mat n );;
```

   - Complexité temporelle : unité : "+". $C(n) = 1 + C(n-1)$ d'où $C(n) = n = O(n)$ **additions**.
   - Complexité spatiale : $S(n) = 2 + S(n-)$ d'où $S(n) = O(n)$ **taille d'entier**.

4. Algorithme récursif, de calcul rapide :
   Pour $a = 1$ et $b = 1$, on a la propriété bien connue de la suite de Fibonacci : $F_{n+p+1} = F_n F_{p+1} + F_{n-1} F_p$,
   et en posant $X_n = (F_n, F_{n+1})$, on a : $\begin{cases} X_{2n} = \left(F_{n+1}\left[F_{n+1} - 2F_n\right] + 2F_n^2 \; , \; F_n\left[2F_{n+1} - F_n\right] \right) \\ X_{2n+1} = \left(F_n\left[2F_{n+1} - F_n\right] \; , \; F_n^2 + F_{n+1}^2 \right) \end{cases}$
   Alors, par combinaison linéaire (espace vectoriel des suites de Fibonnacci),

```
let fibonacci_4 a b n =
  let rec fibo = function
    | 0 -> 1,1
    | n -> let (x, y) = fibo (n/2) in
           if n mod 2 = 0 then 2*x*(x-y)+y*y , x*(y+y-x) else x*(2*y-x) , x*x+y*y
  in match n with
    | 0 -> a | 1 -> b
    | _ -> let (x,y) = fibo (n-2) in a*x + b*y
;;
```

   That's sioux ! dont'it ?

   - Complexité temporelle : unité de coût : "*" (test, addition, division et multiplication par 2, étant négligeables devant "*") ; $C(n) = 3 + C(n/2)$ d'où $C(n) = 3\log_2 n = O(\log_2 n)$ **multiplications**.
   - Complexité spatiale : $S(n) = 2 + S(n/2)$ d'où $S(n) = 2\log_2 n = O(\log_2 n)$ **taille d'entier**.

7.2 Coupes maximales d'un vecteur

Etant donné un vecteur $\overrightarrow{V} = [|x_0, \ldots, x_n|]$ où les x_i sont des nombres, indicé de 0 à n, on cherche un sous-vecteur contigu $\overrightarrow{V}' = [|x_g, \ldots, x_d|]$ tel que $\sum_{i=g}^{d} x_i$ soit la plus grande somme de tous les sous-vecteurs contigus de $\overrightarrow{V}$.

1. Algorithme naïf.

On peut résoudre le problème à l'aide de trois boucles itératives imbriquées, indicées par g variant de 0 à n, d variant de g à n et i variant de g à d, d'où un algorithme de complexité temporelle $O(n^3)$ additions.

```
let maxSomme1 v =
  let n = Array.length v -1 and maxi = ref (v.(0)) and plage_maxi = ref (0,0) in
  for g = 0 to n
  do for d = g to n
     do let s = ref 0 in for i = g to d do s := !s + v.(i) done;
        if !s > !maxi then begin maxi := !s; plage_maxi := (g,d) end
     done
  done;
  !maxi, !plage_maxi;;
```

2. Amélioration de l'algorithme naïf. Dans l'algorithme naïf, les sommes $\sum_{i=g}^{j} x_i$ sont calculées plusieurs fois.

A l'aide d'une variable supplémentaire, on peut économiser l'exécution systématique de la boucle interne (i variant de g à d) et la remplacer par un calcul qui nécessite une seule addition.

On obtient un algorithme de complexité temporelle $O(n^2)$ additions.

```
let maxSomme2 v =
  let n = Array.length v -1 and maxi = ref (v.(0)) and plage_maxi = ref (0,0) in
  for g = 0 to n
  do let s = ref (0) in
     for d = g to n
     do s := !s + v.(d);
        if !s > !maxi then begin maxi := !s; plage_maxi := (g,d) end
     done
  done;
  !maxi, !plage_maxi;;
```

3. Méthode "diviser pour régner". Pour un sous-vecteur $[|x_g, \ldots, x_d|]$, avec $m = \left\lfloor \frac{g+d}{2} \right\rfloor$, on cherche

- le sous-vecteur de la forme $[|x_{g'}, \ldots, x_m]$ de somme maximale dans $\overrightarrow{V}_G = [|x_g, \ldots, x_m|]$,
- le sous-vecteur de la forme $[|x_{m+1}, \ldots, x_{d'}]$ de somme maximale dans $\overrightarrow{V}_D = [|x_{m+1}, \ldots, x_d|]$.

Un sous-vecteur de $[|x_g, \ldots, x_d|]$ qui réalise la somme maximale dans $[|x_g, \ldots, x_d|]$, est soit $[|x_{g'}, \ldots, x_{d'}|]$, soit dans V_G ou dans V_D.

La complexité temporelle $C(n)$ vérifie $C(n) = 2C(n/2) + n$, d'où un algorithme en $O(n \log n)$ additions.

```
let maxSomme3 v =
   let rec aux v d f =
     if d = f then v.(d),(d,f)
     else begin let s = ref 0 and m = (d+f)/2 in
                let mg = ref (v.(m)) and ig = ref m in      (* de d à m *)
                s := 0;
                for i = m downto d
                do s := !s + v.(i); if !s > !mg then begin mg := !s; ig := i end done;
                let md = ref (v.(m+1)) and id = ref m in    (* de m+1 à f *)
                s := 0;
                for i = m+1 to f
                do s := !s + v.(i); if !s > !md then begin md := !s; id := i end done;

                let mmg, (gg,dg) = aux v d m                    (* de d   à m *)
                and mmd, (gd,dd) = aux v (m+1) f                (* de m+1 à f *)
                in if (mmg > !mg + !md) && (mmg > mmd)
                   then mmg, (gg,dg)
                   else if (mmd > !mg + !md) && (mmd > mmg)
                        then mmd, (gd,dd)
                        else !mg + !md, (!ig, !id)
          end
     in aux v 0 (Array.length v -1);;
```

4. Méthode de récurrence simple.

Si on a déjà calculé le sous-vecteur $[|x_g,\dots,x_d|]$ de somme maximale S, situé le plus à droite possible, pour $[|x_1,\dots,x_{n-1}|]$, on en déduit, selon les valeurs de S, de $\sum_{i=g}^{n} x_i$ et de x_n, le sous-vecteur de somme maximale, situé le plus à droite possible, pour $[|x_1,\dots,x_n|]$.

(a) Dans une première étape, le calcul systématique de $\sum_{i=g}^{n} x_i$ conduit à un algorithme en $O(n^2)$ additions.

```
let maxSomme4a v =
  let rec aux v = function
    | 0 -> v.(0), (0,0)
    | n -> let ss, (g,d) = aux v (n-1) in
           let s1 = ref ss in for i = d+1 to n-1 do s1 := !s1 + v.(i) done;
           let s = !s1 + v.(n) in
           if s >= ss
           then if v.(n) >= s  then v.(n), (n,n) else s, (g,n)
           else if v.(n) >= ss then v.(n), (n,n) else ss, (g,d)
   in aux v (Array.length v -1)
;;
```

(b) Dans une seconde étape, grace à une variable auxiliaire, on ramène le calcul systématique de $\sum_{i=g}^{n} x_i$ à un calcul effectuant au plus une addition.
La complexité $C(n)$ vérifie $C(n) = C(n-1)+1$, et on a un algorithme en $O(n)$ additions.

```
let maxSomme4 v =
  let sg = ref (v.(0)) in
  let rec aux v = function
    | 0 -> v.(0), (0,0)
    | n -> let ss, (g,d) = aux v (n-1) in
           let s = !sg + v.(n) in
           if s > ss
           then if v.(n) > s
                then begin sg := v.(n);       v.(n), (n,n) end
                else begin sg := !sg + v.(n); s, (g,n)     end
           else if v.(n) > ss
                then begin sg := v.(n);       v.(n), (n,n) end
                else begin sg := !sg + v.(n); ss, (g,d)     end
   in aux v (Array.length v - 1)
;;
(* val maxSomme4 : int array -> int * (int * int) = <fun> *)
let v = [| 0; -2; 1; -3; 4; 5; 6; -5; -4; 1 ; 2; 4; 10; -1; 2; 3; 4; 1; -1; 7; 6 ; -2; -4|];;
maxSomme4 v;;  (* -> 44, (4, 20) *)
```

On est passé d'une complexité temporelle $O(n^3)$ à $O(n^2)$ puis à $O(n\log n)$ (et on aurait été tenté de s'arréter là !) pour finalement obtenir $O(n)$ (difficile de faire mieux !), le tout sans accroître la complexité spatiale.

7.3 Le problème du voyageur de commerce

Visiter une et une seule fois n villes, en minimisant la distance parcourue le long d'un réseau routier et en revenant au point de départ.
On ne connait pas d'algorithme donnant un tel circuit en temps polynomial, et la version décisionnelle (pour une distance D, existence d'un circuit plus court que D) est un problème NP-complet.

- Même sans chercher à minimiser la distance, on ne connaît pas de condition nécessaire et suffisante pour l'existence d'un circuit fermé passant une et une seule fois par chaque ville (il existe un certain nombre de conditions suffisantes).
- En admettant que l'on arrive à trouver au moins un circuit fermé passant une et une seule fois par chaque ville (ce qui est déjà un problème difficile, sauf cas particuliers), il y a peu de chance pour que celui-ci soit minimal ou même proche d'un circuit minimal.

On se limite à la recherche d'un chemin pour lequel la distance parcourue ne soit pas trop grande : A partir d'un circuit fermé initial (si on en a trouvé un !), on cherche à améliorer ce circuit avec des méthodes plutôt empiriques, issues de l'observation de phénomènes physiques, biologiques, etc ...

• Algorithme des fourmis. • Algorithme génétique. • Algorithme du recuit simultané. • etc ...

Cela peut donner des résultats spectaculaires, même avec un grand nombre de villes et de chemins possibles, dans un temps et avec une complexité spatiale acceptables.

7.4 Ce nombre est-il premier ?

L'entier $n =$ **24578956214702358742544692012511478923655540881789541785 67** est-il premier ?

Tout d'abord, une question fondamentale : **Y-a-t'il un intérêt à savoir si un tel nombre est premier ?** Si non, passer à la page suivante .... Si oui, pourquoi ?

En fait, de tels grands entiers premiers sont les éléments de base des méthodes de cryptographie et des processus de sécurité informatique (cartes bancaires par exemple).

Ces grands nombres, qui dépassent les capacités usuelles de représentation des nombres entiers dans les langages de programmation usuels, sont traités à l'aide de bibiothèques spécialisées (et d'ordinateurs performants !).
En Caml, pour représenter les grands entiers, on dispose de la bibliothèque spécifique : `big_int`

1. Recherche de facteurs de n par le **Crible d'Eratosthène** (276 – 194 av. JC) :

   Cette méthode, qui consiste à tester la divisibilité par tous les nombres jusq'à $\sqrt{n}$, n'aboutit pas en temps fini (suffisamment court) si n est très grand et effectivement premier ou même si n n'a que de très grands diviseurs ....

2. On se contente d'algorithmes de **"forte pseudo-primalité"** :

   (a) **Extrait de l'aide de Maple relative à "isprime(n)"** :
   It returns false if n is shown to be composite within one strong pseudo-primality test and one Lucas test and returns true otherwise. If isprime returns true, n is "very probably" prime - see Knuth "The art of computer programming", Vol 2, 2nd edition, Section 4.5.4, Algorithm P for a reference and H. Reisel, "Prime numbers and computer methods for factorization". No counter example is known and it has been conjectured that such a counter example must be hundreds of digits long.

   (b) Exemple : **Algorithme de Miller-Rabin** :

   Soit $n \in \mathbb{N}$, impair, et m un entier impair tel que $n-1 = 2^k m$.
   Soit a un entier (aléatoire) tel que $1 < a < n$.
   - Si $a^m \equiv 1 \mod n$ alors n est (peut-être) premier ;
   - sinon, s'il existe $i \in [\![0 \ldots k-1]\!]$ tel que $a^{2^i m} \equiv -1 \equiv n-1 \mod n$ alors n est (peut-être) premier ;
   - sinon n est (sûrement) factorisable.

   Mise en œuvre pratique :
   D'après le Théorème de raréfaction des nombres premiers, la probabilité d'erreur (réponse "n est premier" alors que n n'est pas premier) de cet algorithme est inférieure (largement) à $1/4$.

   Lorsque l'on "tire" e fois un nombre a, de façon aléatoire, $1 < a < n$, la probabilité pour que "n soit factorisable" , sachant que l'algorithme a répondu e fois "n est (peut-être) premier", est majorée par

   $$\frac{\log n - 2}{\log n - 2 + 4^e + 1}$$

   On prévoit donc d'effectuer au plus e (par exemple $e = 30$ tirages ou plus si n est très très grand).
   - Dès que l'algorithme répond "n est factorisable", on sait que n n'est pas premier.
   - Si l'algorithme a répondu e fois "n est premier" alors la probabilité pour que ce soit faux est extrêmement faible.

   Remarque. Eventuellent, on cumule ce test avec un autre test de (pseudo-)primalité.

   Eléments d'étude de complexité temporelle :

   - Le coût d'obtention de k et de m (fait une seule fois) est plus ou moins faible selon la structure de donnée utilisée pour représenter les grands entiers (quasi nul avec la représentation binaire).
   - Le calcul de $b = a^m \mod n$ a un coût en $O(\log_2 m)$ (exponentiation rapide modulo n).
   - Le calcul de $a^{2^i m} = b^{2i} \mod n$ se fait avec un coût faible, puisque, à chaque itération on peut calculer $b^{2^i} = \left(b^{2^{i-1}}\right)^2 \mod n$, avec un coût de calcul égal à celui d'un carré et d'une division ($\mod n$) sur le résultat de l'itération précédente.

7.5 Factorisation de (grands) entiers

Il s'agit d'un problème difficile, pour lequel on ne connaît pas de méthode générale réalisable en un temps fini (ou raisonnablement court).

1. Crible d'Eratosthène, 276 – 194 av. JC (pour mémoire, voir précédemment).
2. Méthode de factorisation de Fermat, ~ 1640 (lente pour des grands nombres).

   Soit N impair, se décomposant sous la forme $N = xy$ avec $1 < x \leqslant y$, (x et y inconnus, impairs).

   $$\text{Si on pose } \begin{cases} u = \dfrac{x+y}{2} \\ v = \dfrac{y-x}{2} \end{cases} \text{, on a } \begin{cases} x = u - v \\ y = u + v \end{cases} \quad \text{et } N = u^2 - v^2.$$

   On va donc chercher u et v entiers tels que $N = u^2 - v^2$ et on retournera le couple $(u-v, u+v)$:

   Algorithme brut :
   - Initialisation : $u \leftarrow \lfloor \sqrt{N} \rfloor$, $v \leftarrow 0$, $r \leftarrow u^2 - N$ (c.a.d. $r \leftarrow u^2 - v^2 - N$)
   - Tant que $r <> 0$ faire
     - si $r < 0$ alors $u \leftarrow u + 1$, sinon $v \leftarrow v + 1$
     - $r \leftarrow u^2 - v^2 - N$ (calcul un peu lourd)
   - Retourner $(u-v, u+v)$

   Algorithme amélioré : en utilisant $u' = 2u + 1$ et $v' = 2v + 1$,
   - Initialisation : $u' \leftarrow 2\lfloor \sqrt{N} \rfloor + 1$, $v' \leftarrow 1$, $r \leftarrow \lfloor \sqrt{N} \rfloor^2 - N$
   - Tant que $r <> 0$ faire
     - si $r < 0$ alors $r \leftarrow r + u'$ puis $u' \leftarrow u' + 2$
       sinon $r \leftarrow r - v'$ puis $v' \leftarrow v' + 2$ (pas de multiplication)
   - Retourner $\left(\dfrac{u'-v'}{2}, \dfrac{u'+v'-2}{2} \right)$

   Remarques.
   - $y = u + v$ sera le plus petit facteur de N supérieur ou égal à $\sqrt{N}$
   - $x = u - v$ sera le plus grand facteur de N inférieur ou égal à $\sqrt{N}$

   et on pourra continuer à chercher la décomposition complète de N, en factorisant de même x et y ...

   En pratique, l'algorithme de Fermat n'est utilisable que pour des nombres pas trop grands, de la forme $N = x \times y$ avec x et y de même ordre de grandeur (c'est à dire proches de $\sqrt{N}$).
   On aura intérêt à rechercher au préalable les "petits" facteurs premiers de N, dans une "base" de facteurs premiers connus, avant d'appliquer l'algorithme de Fermat au quotient.
3. Méthode "$\rho - 1$" (ou "ρ") de Pollard ~ 1975.

   Cette méthode (simple) de factorisation de l'entier N, suppose que N admet un facteur premier p (inconnu) tel que $p-1$ ne possède que de petits facteurs premiers.

   > Soit B une borne telle que pour tout facteur premier u de $p-1$ on ait $u \leqslant B$.
   > Soit g un nombre compris entre 2 et $p-2$, (par exemple $g = 2$ ou $g = 3$).
   > **Algorithme** :
   > - $a \leftarrow g$
   > - Pour $k = 2$ jusqu'à B, $a \leftarrow a^k \mod N$ (exponentielle rapide $\mod N$)
   > - Avec $d \leftarrow pgcd(a-1, N)$,
   >
   > si $1 < d < N$ alors (d est un facteur de N) sinon (échec ...)

   En cas de succès, on pourra continuer la décomposition de N en tentant la factorisation de d et de N/d.

   Cette méthode permet (**parfois**) de trouver des facteurs de très grands nombres (mais les concepteurs de grands nombres se méfient ... et les choisissent tels qu'ils résistent à cet algorithme.
   Remarque. Si on prend B trop grand, on perd de l'efficacité et pour $B = \sqrt{N}$, l'algorithme se sera pas plus efficace que le crible d'Eratosthène.

   La méthode "$p-1$" a été adaptée pour donner une méthode plus générale qui fait l'hypothèse beaucoup moins restrictive : "il existe un entier proche de p qui n'admet que des petits facteurs premiers" ... (méthode des "courbes elliptiques").

7.6 Réduction de la complexité spatiale de l'exponentiation matricielle rapide

L'algorithme récursif du calcul de x^n par exponentiation rapide a une complexité temporelle en $O(\log_2 n)$ multiplication du type de donnée et une complexité spatiale en $O(\log_2 n)$ structure de donnée.

Dans le cas de l'exponentiation de grandes matrices, la complexité spatiale peut être considérablement réduite en utilisant la version itérative :

1. Ressources de base du calcul matriciel :

```
let id_matrix n =
    let u = Array.make_matrix n n 0. in for i = 0 to n-1 do u.(i).(i) <- 1. done; u
;;
let mult_matrix m1 m2 =
   let p = Array.length m1 and q = Array.length m1.(0) and r = Array.length m2.(0) in
   let m = Array.make_matrix p r 0. in
   for i = 0 to p-1
   do for j = 0 to r-1
      do for k = 0 to q-1 do m.(i).(j) <- m.(i).(j) +. m1.(i).(k) *. m2.(k).(j) done
      done
   done;
   m
;;
let ( *? ) m1 m2 = mult_matrix m1 m2
;;
```

   Remarque. `mult_matrix` est de complexité temporelle $O(n^3)$ multiplications de float.

2. Version récursive de l'exponentiation matricielle rapide :

```
let rec exp_mat_rap x  = function
   | 0 -> id_matrix (Array.length x)
   | n -> let u = exp_mat_rap x (n/2) in
          if (n mod 2) = 0 then u *? u else x *? u *? u
;;
(* val exp_mat_rap : float array array -> int -> float array array = <fun> *)
```

   - Complexité temporelle en $O(\log_2 n)$ multiplications de matrice
   - Complexité spatiale en $O(\log_2 n)$ structure de matrice.

3. Version itérative de l'exponentiation matricielle rapide :

```
let exp_mat_rap_iter x n =
    let xn = ref (id_matrix (Array.length x)) and c = ref x and k = ref n in
    while !k > 0
    do if !k mod 2 <> 0 then xn := !xn *? !c;
       c := !c *? !c;
       k := !k/2;
    done;
    !xn
;;
```

   Algorithme qui est toujours de complexité temporelle en $O(\log_2 n)$ multiplications de matrice, mais qui est de **complexité spatiale constante**, en $O(1)$ structure de matrice (`xn` et `c` sont modifiés en place).

$< \mathscr{F}\mathscr{I}\mathscr{N} >$