Chapitre 1
Complexité des algorithmes

I Complexité

I.1 Qu'est-ce qu'un algorithme ?

La notion d'algorithme est une notion difficile : il n'est pas si évident que cela de la définir. L'approche théorique de la notion d'algorithme passe par le concept de "machine de Turing", modèle simplifié, idéalisation mathématique, d'un ordinateur "réel". On appelle alors algorithme tout programme tournant sur une machine de Turing. Une fonction f est alors dite calculable s'il existe un algorithme (c'est-à-dire une machine de Turing) qui soit capable de calculer $f(x)$ pour tout x raisonnable. Cette idéalisation de l'ordinateur, possède une certaine universalité. D'autres approches de la notion d'algorithme conduisent à la même classe de fonctions calculables.

Nous nous contenterons évidemment dans ce cours d'une approche plus concrète : nous appellerons algorithme toute fonction écrite dans un langage de programmation tel que Caml. Signalons quelques problèmes posés par notre définition :

- La dépendance du langage utilisé. Cette dépendance n'est qu'apparente. Si l'on peut écrire un algorithme calculant une fonction f en, mettons, Pascal, on pourra aussi en écrire un en Caml. En revanche, certains algorithmes peuvent être plus performants dans un langage que dans un autre langage. Problème intéressant, mais au-delà de la portée de ce cours. Un algorithme sera dorénavant une fonction Caml.
- Le codage des objets manipulés par un algorithme peut avoir de l'importance. On supposera toujours que l'on a affaire à des codages raisonnables. Il est clair que coder un entier en base 1 ou en base 2 conduira sans doute à des comportements différents des algorithmes : en base 2, 20 s'écrit 10100, et en base 1, il s'écrit 11111111111111111111. Que pensez vous de l'écriture de 1000000 ?
- On ne peut pas coder des ensembles infinis d'objets, ni même certains objets qui sont "infinis" par essence : par exemple, les réels non décimaux posent un problème. Ainsi, on ne peut pas coder l'ensemble des nombres réels, mais seulement un sous-ensemble fini de celui-ci. Les algorithmes de calcul numérique doivent tenir compte de ce fait.

I.2 Complexité

Naïvement, la complexité d'un algorithme est le "nombre d'opérations" nécessaires à l'algorithme pour effectuer son travail. Cette complexité donne une idée du temps nécessaire à l'algorithme pour s'exécuter. La notion de complexité n'a réellement de sens que dans le cadre des machines de Turing. Aussi, nous contenterons-nous d'évaluer le nombre d'opérations d'un certain type judicieusement choisi nécessaires à l'exécution d'un algorithme. Ainsi :

- Dans un algorithme de tri comparatif, une opération significative sera sans doute une comparaison.
- Dans un algorithme de multiplication ou d'addition d'entiers longs, ce sera peut-être une opération sur les bits.
- Dans une multiplication de polynômes à coefficients réels, ce sera sûrement une somme ou un produit de réels.
- Dans le cas d'une fonction récursive, il pourra être intéressant de calculer le nombre d'appel récursifs.
- Dans un algorithme déplaçant beaucoup de données, le nombre d'opérations d'affectation sera certainement révélateur du temps de calcul.

Lors de l'étude d'un algorithme, il n'est pas toujours évident de dégager quelles sont les opérations importantes. L'expérimentation est parfois utile.

I.3 Taille des données

La complexité d'un algorithme calculant $f(x)$ dépend bien entendu de la donnée x fournie au départ à cet algorithme. Cette dépendance peut parfois être très complexe. Cela dit, elle est parfois fonction uniquement de la *taille* de x, que l'on définit comme le nombre de bits nécessaires au codage de x.

Cette taille dépend donc du codage choisi pour les données. Mais tous les codages raisonnables fourniront en général des tailles comparables. Ainsi, par exemple, le codage d'un entier N en base 2 demande k fois plus de bits qu'un codage en base 10, mais le facteur k est indépendant de N.

Que vaut la constante k ci-dessus ?

Ainsi, en travaillant à constante multiplicative près, on pourra s'affranchir de discuter trop sur le codage des données. Ainsi, lorsque nous dirons qu'un objet X est de taille 1, cela signifiera que tous les objets du même type que X ont une taille bornée. A titre indicatif :

- Un entier Caml, un réel Caml sont des données de taille 1.
- Un entier naturel "long" n est une donnée de taille $\log n$.
- Un tableau de n entiers est de taille n (mais un tableau de n entiers longs $[x_1, \cdots, x_n]$ serait de taille $\sum_{i=1}^{n} \log x_i$, taille pouvant être bornée par exemple par $n \max_i \log x_i$).
- Une matrice $m \times n$ d'entiers est de taille $m.n$
- Un polynôme de degré N à coefficients entiers est de taille N.
- Un graphe G possédant M sommets et N arêtes est de taille $M + N$ si l'on choisit de le coder par un tableau étiqueté par les sommets de G, dont les éléments sont des listes formées des sommets adjacents au sommet courant. Mais il sera de taille M^2 si on choisit de coder G par une matrice A de taille $M \times M$, dont le coefficient $a_{i,j}$ vaut 1 s'il y a une arête du sommet i vers le sommet j, et 0 sinon.

I.4 Différents types de complexité

La complexité d'un algorithme est fonction de cet algorithme et des paramètres fournis à celui-ci. Souvent, un calcul exact de la complexité s'avère très difficile, voire impossible. On s'oriente en général vers les calculs suivants :

- La complexité en pire cas consiste à calculer la quantité $\mathcal{C}(N)$ égale au maximum des complexités pour toutes les données de taille N.
- La complexité en moyenne consiste à associer aux données X de taille N une probabilité, et à considérer la moyenne $\overline{\mathcal{C}(N)}$ de la complexité de l'algorithme sur toutes ces données.

Chacune de ces deux complexités est importante. La complexité en pire cas est rassurante. Elle nous affirme que jamais l'algorithme ne se comportera de façon imprévue. La complexité en moyenne nous dit que certains algorithmes, même s'ils s'avèrent peu intéressants sur certaines données, peuvent se comporter de façon tout à fait correcte en général (c'est le cas du tri rapide étudié un peu plus loin).

I.5 En conclusion

Étudier un algorithme c'est :

- Le décrire, généralement au moyen d'une fonction Caml.
- Prouver sa terminaison et sa correction. L'algorithme doit effectivement calculer ce à quoi on s'attend, et ceci se prouve mathématiquement.
- Étudier sa complexité. Les calculs de complexités sont parfois effectués de façon exacte, et parfois en ordre de grandeur. Un résultat du type $\mathcal{C}(n) = O(n \log n)$ sera souvent beaucoup plus intéressant que $\mathcal{C}(n) = (n + \frac{3}{2}) \log \frac{3n^2+7n-1}{3n+4}$ (mais pas toujours).

Nous allons, dans la suite du chapitre, étudier quelques algorithmes choisis au hasard (enfin presque), afin de mettre en pratique ce qui vient d'être raconté.

II Exponentiation dichotomique

La méthode d'exponentiation dichotomique repose sur le principe que pour calculer x^n, il n'est pas nécessaire d'effectuer n multiplications de x par lui même, étant donné que x^n est en gros égal à $(x^2)^{n/2}$, avec une correction éventuelle dans le cas où n est un entier impair. L'algorithme peut être décrit en deux versions, l'une récursive et l'autre itérative.

II.1 Algorithme récursif

II.1.1 L'Algorithme

```
let rec pow x n =
  if n = 0 then 1
  else
    let y = pow ( x * x ) ( n / 2 ) in
    if n mod 2 = 0 then y
    else x * y;;
```

L'algorithme ci-dessus fonctionne lorsque x est entier. Réécrire la fonction ci-dessus pour qu'elle accepte des paramètres de type quelconque, et une multiplication également quelconque.

II.1.2 Terminaison et preuve

On fait une démonstration par récurrence sur n.

- Si $n = 0$, c'est clair. L'algorithme termine et renvoie effectivement x^0.
- Donnons nous $n > 0$. Supposons que cela fonctionne pour tout entier strictement inférieur à n. Un appel à `pow` va s'aiguiller dans la partie `else`. Mais comme $n/2 < n$, l'appel récursif à `pow` se termine et y vaut donc $(x^2)^{n/2}$ si n est pair, et $(x^2)^{(n-1)/2}$ si n est impair. D'où la preuve.

II.1.3 Nombre de multiplications

On va compter le nombre de multiplications d'entiers effectuées par l'algorithme. La division par 2 et le modulo 2 peuvent être réalisés par des décalages binaires. On ne les compte pas. On démontre par récurrence (forte) sur $n > 0$ que

$$T(n) \leq c \lg n + d$$

où c et d sont des réels strictement positifs bien choisis. On remarque au préalable que l'algorithme, hormis l'appel récursif, effectue au maximum 2 multiplications (en fait, 0, 1 ou 2 selon que n est nul, pair ou impair).

- Pour $n = 1$ ça fonctionne pour tout c réel positif et tout $d \geq 2$.
- Si c'est vrai pour tout $k < n$, on a alors

$$\begin{aligned} T(n) &\leq 2 + T(\lfloor n/2 \rfloor) \\ &\leq 2 + c \lg n/2 + d \\ &= c \lg n + (d + 2 - c) \end{aligned}$$

On voit que, par exemple, $d = 2$ et $c = 2$ conviennent.

II.2 Algorithme itératif

II.2.1 L'Algorithme

On considère l'algorithme suivant :

```
let pow x n =
   let z = ref 1
   and m = ref n
   and y = ref x in
   while !m <> 0 do
      if !m mod 2 = 1 then z := !z * !y ;
      m := !m / 2 ;
      y := !y * !y
   done;
   !z ;;
```

A chaque itération, m est remplacé par la partie entière de $\frac{m}{2}$. Il s'ensuit par récurrence forte que l'algorithme termine toujours. Nous allons prouver que cet algorithme renvoie effectivement x^n, puis évaluer le nombre de multiplications.

II.2.2 Preuve de l'algorithme

On va montrer que la quantité zy^m est invariante dans la boucle.
Soient z, y, m les valeurs des variables à l'entrée d'une itération, et z', y', m' leurs valeurs à la sortie de cette même itération. Deux cas se présentent :

- $m = 2p$ est pair, non nul. Alors $z' = z$, $y' = y^2$ et $m' = p$. On en tire $z'y'^{m'} = z(y^2)^p = zy^m$.
- $m = 2p+1$ est impair. Alors $z' = zy$, $y' = y^2$ et $m' = p$. On en tire $z'y'^{m'} = zy(y^2)^p = zy^{2p+1} = zy^m$.

En entrée de boucle, on a $zy^m = 1.x^n = x^n$. En sortie de boucle, on a $zy^m = zy^0 = z$. Par l'invariance, il vient en sortie de boucle $z = x^n$.

II.2.3 Nombre de multiplications

Notons $T(n)$ le nombre de multiplications effectuées par l'algorithme. La complexité de `pow` est alors majoré par 2 fois le nombre d'itérations de la boucle `while` (la division de m par 2 n'est pas comptée, elle peut être effectuée à l'aide d'un décalage de bits). Cette quantité peut être calculée exactement en fonction de n.

On écrit $n = \sum_{j=0}^{k} a_j 2^j$ où les a_j valent 0 ou 1, et a_k est égal à 1. Ceci est en toute rigueur valable lorsque $n \neq 0$. Sinon, la boucle n'est pas exécutée.

On montre ensuite par récurrence qu'à l'issue de l'étape p, m vaut

$$\sum_{j=p+1}^{k} a_j 2^{j-p-1}.$$

Ainsi m est non nul pour tout $p \leq k-1$.

A l'étape k, m devient donc nul. La boucle est exécutée k fois, et à chaque itération on effectue 1 ou 2 multiplications.

On effectue donc entre k et $2k$ multiplications, et on remarque de plus que l'entier k vérifie très clairement $2^k \leq n < 2^{k+1}$. La boucle est donc parcourue k fois avec $k = \lfloor \lg n \rfloor$ et le nombre de multiplications est ainsi $T(n) = O(\lg n)$.

Calculer le nombre exact de multiplications effectuées en fonction de n et du nombre $\gamma(n)$ de chiffres non nuls dans l'écriture de n en base 2 (le nombre de bits non nuls de l'entier n).

III Tri rapide

Le tri rapide est une méthode de tri de tableaux intéressante à plus d'un point :

- Comme son nom l'indique, c'est un tri efficace.
- Le tri rapide permet de trier un tableau sans faire appel à des ressources de mémoire supplémentaires : le tableau est trié sur place.
- Les idées mises en oeuvre dans l'analyse de l'algorithme sont très générales. On retrouve parfois la même analyse pour des algorithmes n'ayant *a priori* aucun rapport avec le tri rapide.
- Le tri rapide est au programme de l'option info, et vous devez être capables de réécrire l'algorithme à tout instant.

Le principe du tri est le suivant :

- Soit x le premier élément du tableau. On place dans la partie gauche du tableau les éléments inférieurs ou égaux à x, et dans la partie droite le reste des éléments, tout en positionnant x à la charnière entre les deux parties.
- On trie récursivement les deux parties, étant entendu qu'un tableau à 1 élément est considéré comme déjà trié.

III.1 L'Algorithme

L'algorithme de tri se scinde en une fonction de partition, et une fonction de tri proprement dite. La fonction de partition partitionne le sous-tableau du tableau t formé des éléments dont les indices sont compris au sens large entre p et r. Elle renvoie un entier j désignant la limite entre les deux "moitiés" du sous-tableau partitionné.

```
let permuter t i j =
   let tmp = t.(i) in
   t.(i) <- t.(j);
   t.(j) <- tmp;;

let partition t p r =
   let x = t.(r)
   and i = ref (p - 1) in
   for j = p to r - 1 do
      if t.(j) <= x then (
         i := !i + 1;
         permuter t !i j;
      )
   done;
   permuter t (!i + 1) r;
   !i + 1;;

let rec quicksort t p r =
   if p < r then
      let q = partition t p r in
      quicksort t p ( q - 1 ) ;
      quicksort t ( q + 1 ) r ;;

let tri t = quicksort t 0 ( Array.length t - 1 ) ;;
```

III.2 Analyse de la fonction de partition

La preuve de la correction de la fonction de partition est délicate. Nous la laissons de côté. Cette fonction effectue $r-p$ comparaisons d'éléments du tableau (et au plus $r-p+1$ échanges d'éléments de tableau).

III.3 Nombre de comparaisons

III.3.1 Le cas moyen

Le tri rapide d'un tableau de taille n commence par partitionner le tableau en prenant comme pivot son premier élément (celui d'indice 0), puis trie récursivement deux sous-tableaux. Les tailles respectives de ces sous-tableaux ont pour valeurs possibles $(0, n-1)$, $(1, n-2)$,... ,$(n-2, 1)$, $(n-1, 0)$. En faisant l'hypothèse que ces tailles sont équiprobables, on en déduit que le nombre de comparaisons $C(n)$ effectuées par le tri rapide pour trier un tableau de taille n est en moyenne :

$$\begin{aligned} C(n) &= n-1+\frac{1}{n}\sum_{k=0}^{n-1}(C(k)+C(n-1-k)) \\ &= n-1+\frac{2}{n}\sum_{k=0}^{n-1}C(k) \end{aligned}$$

avec de plus $C(0)=0$, $C_1=0$. Il reste à étudier cette récurrence.
Pour $n \geq 2$ on a

$$nC_n-(n-1)C_{n-1}=2(n-1)+2C_{n-1}$$

d'où

$$nC_n=2(n-1)+(n+1)C_{n-1}.$$

Posons $D_n=\frac{C_n}{n+1}$. Il vient pour $n \geq 2$,

$$D_n=2\frac{n-1}{n(n+1)}+D_{n-1}$$

d'où

$$D_n=D_1+2\sum_{k=2}^{n}\frac{k-1}{k(k+1)}.$$

On a

$$\frac{k-1}{k(k+1)}=-\frac{1}{k}+\frac{2}{k+1}.$$

On en tire, puisque $D_1=0$, que

$$D_n=\frac{2}{n+1}+\sum_{k=1}^{n}\frac{1}{k}-2.$$

Cette quantité étant équivalente à $\ln n$, on a donc

$$C_n \sim n\ln n.$$

III.3.2 Le pire cas du quicksort

Nous allons montrer que le pire cas survient lorsque le tableau de départ, de taille n, est déjà trié. Cette démonstration se fait en deux étapes. On prouve tout d'abord que le quicksort demande $O(n^2)$ comparaisons sur un tableau trié de n éléments. On montre ensuite que ce $O(n^2)$ est maximal, au sens où le nombre de comparaisons du tri rapide sur

un tableau de taille n est inférieur ou égal à cn^2 où c est une constante convenablement choisie.

- Prenons un tableau trié. Supposons pour simplifier que les éléments du tableau sont tous distincts. La fonction de partition ne modifie rien dans le tableau. Mais elle effectue au moins n comparaisons. Puis, le tri rapide est appelé sur deux sous-tableaux, l'un de taille 0 et l'autre de taille $n-1$. Ainsi, $C(n) \geq n + C(n-1)$. D'où :

$$C(n) \geq \frac{n(n-1)}{2} = O(n^2)$$

- Montrons maintenant que, pour un tableau quelconque, on a $T(n) \leq cn^2$ pour une constante c convenable. On a tout d'abord pour tout $n \geq 1$

$$T(n) \leq n + \max_{0 \leq k \leq n-1} (T(k) + T(n-1-k))$$

Pour $n = 0$, n'importe quel réel c convient. Supposons trouvé un réel positif c convenant pour tout entier k strictement inférieur à l'entier $n > 0$. On a alors

$$T(n) \leq n + c \max_{0 \leq k \leq n-1} (k^2 + (n-1-k)^2)$$

On vérifie facilement que la fonction $\varphi : [0, n-1] \to R$ définie par $\varphi(x) = x^2 + (n-1-x)^2$ atteint son maximum aux bornes de son intervalle de définition, ce maximum étant égal à $(n-1)^2$. On en déduit que

$$T(n) \leq n + c(n-1)^2$$

On vérifie alors que, pourvu que $c \geq$, on aura bien $T(n) \leq cn^2$. Ainsi, $T(n) \leq n^2$.

En conclusion, le tri rapide possède en moyenne des qualités indéniables, mais risque de s'avérer mauvais dans certains cas. On peut montrer que ces cas sont rares d'un point de vue probabiliste (l'écart type du nombre de comparaisons est négligeable devant sa moyenne). De plus, il existe des méthodes simples pour éviter les cas pathologiques. Il suffit par exemple d'échanger le premier élément du tableau à trier (le pivot) avec un élément au hasard pour être "presque sûr" de tomber sur le cas moyen. Toutes ces affirmations demanderaient bien entendu d'être prouvées soigneusement.

IV Multiplication rapide de matrices

Prenons deux matrices carrées $n \times n$ A et B. Le calcul naïf du produit $C = AB$ nécessite celui des n^2 coefficients de la matrice C. Pour chacun d'entre eux, il faut effectuer n multiplications et $n-1$ additions, ce qui donne en fin de compte un algorithme en $O(n^3)$.

Il est possible de faire mieux, lorsque la taille des matrices est une puissance de 2 (dans le cas général, l'algorithme qui suit peut être adapté).

- Soient, pour commencer, deux matrices carrées 2×2 A et B définies par

$$A = \begin{pmatrix} a & c \\ b & d \end{pmatrix} \qquad B = \begin{pmatrix} a' & c' \\ b' & d' \end{pmatrix}$$

On calcule les quantités suivantes :

$$\begin{cases} m_1 = (b+d-a)(d'-c'+a') \\ m_2 = aa' \\ m_3 = cb' \\ m_4 = (a-b)(d'-c') \\ m_5 = (b+d)(c'-a') \\ m_6 = (c-b+a-d)d' \\ m_7 = d(a'+d'-b'-c') \end{cases}$$

On a alors

$$AB = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

Il est ainsi possible de multiplier deux matrices 2×2 en utilisant 7 multiplications et 24 additions.

Montrer que 15 additions sont suffisantes.

- Voyons en quoi ceci est une amélioration de l'algorithme naïf. Le calcul ci-dessus peut s'appliquer à des multiplications par blocs. Pour multiplier deux matrices $2n \times 2n$, on les coupe chacune en 4 blocs $n \times n$ et on utilise les formules ci-dessus, récursivement bien sûr. Il est enfin clair que deux matrices 1×1 se multiplient avec 1 multiplication scalaire et 0 addition scalaire. Notons $M(n)$ et $A(n)$ respecivement le nombre de multiplications et d'additions requis pour multiplier deux matrices carrées de taille 2^n. On a $M(0) = 1$ et $A(0) = 0$. De plus, on a les relations de récurrence

$$M(n) = 7M(n-1)$$

$$A(n) = 7A(n-1) + 15.2^{2(n-1)}$$

La relation sur M est claire. En ce qui concerne $A(n)$, On a d'une part les additions cachées dans l'appel récursif, et d'autre part les sommes de matrices de taille $2^n/2 = 2^{n-1}$ qui sont effectuées classiquement et demandent donc pour chacune $(2^{n-1})^2$ additions scalaires.

On obtient pour tout n positif

$$M(n) = 7^n.$$

Un calcul en cascade fournit

$$A(n) = 5.7^n - 5.4^n.$$

N'oublions pas que ces calculs concernent des matrices de taille 2^n. Pour un produit de matrices de taille n, on obtient, en remplaçant n par $\lg n$ dans les expressions ci-dessus, un nombre de multiplications scalaires égal à $n^{\lg 7}$ et un nombre d'additions scalaires égal à $5.n^{\lg 7} - 5.n^2$. Sachant que $\lg 7 \simeq 2.8$, nous avons donc un algorithme dont le nombre d'opérations est négligeable par rapport à l'algorithme naïf.

Revenons un instant sur le titre du paragraphe. La multiplication rapide de matrices est-elle rapide ? Nous allons comparer brièvement le nombre total d'opérations (addition, multiplication) effectuées d'une part par l'algorithme naïf, d'autre part par l'algorithme que nous venons d'étudier. On a approximativement :

- $6.n^{\lg 7}$ opérations pour l'algorithme rapide.
- $2.n^3$ opérations pour l'algorithme naïf.

On peut considérer que l'algorithme rapide est vraiment rapide lorsque $6.n^{\lg 7} \leq 2n^3$, c'est-à-dire lorsque $n^{3-\lg 7} \geq 3$ ce qui donne pour n une valeur supérieure à 300. L'algorithme rapide n'a donc d'intérêt que pour des grosses matrices. Signalons en plus les multiples appels récursifs, consommateurs de temps et de mémoire. En fait, une implémentation réelle de la multiplication rapide des matrices devrait être particulièrement soignée et optimisée pour avoir un quelconque intérêt. Dans la pratique, on met en oeuvre un algorithme mixte, où l'algorithme naïf prend le relais lorsque la taille des matrices devient inférieure à une certaine taille.

Refaire le calcul en en attribuant un poids 2 aux multiplications.

V Tri par paquets

On désire trier un tableau a de n réels compris entre 0 (au sens large) et 1 (au sens strict), notés a_0 ,a_1, ..., a_{n-1}. On place ces réels dans un tableau b de n listes,$b_0, b_1, ..., b_{n-1}$ de sorte que si $\frac{i}{n} \leq x \leq \frac{i+1}{n}$, alors $x \in b_i$. En plus condensé, $\forall i \in [0, n-1]\, a_i \in b_{\lfloor n a_i \rfloor}$. On trie ensuite chacune des listes b_j par un tri "naïf", de type tri par insertion par exemple. On concatène ensuite les listes triées. Le tableau a est alors trié.

V.1 L'Algorithme

```
let tri_paquets a =
   let n = Array.length a in
   let b = Array.make n [] in
   for i=0 to n - 1 do
      let j = int_of_float(a.(i) *.  (float_of_int n)) in
      b.(j) <- a.(i)::  b.(j)
   done;
   for j = 0 to n - 1 do
      b.(j) <- tri_insertion b.(j)
   done;
   let k = ref 0
   and i = ref 0 in
   while !i <= n - 1 do
      match b.(!k) with
      | [] -> k := !k+1
      | x::l -> a.(!i) <- x ; b.(!k) <- l ; i:= !i+1
   done;;
```

Prouver cet algorithme.

V.2 Complexité en moyenne

Nous allons évaluer le nombre moyen de comparaisons effectuées par l'algorithme.

Placer les a_i dans le tableau b demande n affectations. La concaténation des b_i nécessite $4n$ affectations. On peut essayer de se convaincre, en effet, que le reste des opérations demande un temps linéaire en n.

Trier la liste b_i nécessite $O({n_i}^2)$ comparaisons où n_i est le nombre d'éléments de b_i. Appelons T_i le nombre moyen de comparaisons requises pour trier b_i. L'entier k étant pris entre 0 et n, la probabilité que n_i soit égal à k vaut $\binom{n}{k} p^k (1-p)^{n-k}$ avec $p = 1/n$. La variable aléatoire n_i suit en fait une loi binômiale. On a ainsi

$$T_i = O(\sum_{k=0}^{n} (k^2 \binom{n}{k} p^k (1-p)^{n-k})$$

Il reste à calculer la somme ci-dessus. Sans utiliser de notions probabilistes de type "variance" ($T_i = E(n_i^2) = V(n_i) + E(n_i)^2$), introduisons

$$\varphi(x) = (px + 1 - p)^n = \sum_{k=0}^{n} (x^k \binom{n}{k} p^k (1-p)^{n-k})$$

On vérifie sans peine que

$$x(x\varphi'(x))' = \sum_{k=0}^{n} (k^2 x^k \binom{n}{k} p^k (1-p)^{n-k})$$

En utilisant l'galité ci-dessus et l'expression factorisée de φ, puis en faisant $x = 1$, on en tire $T_i = np[1 + (n-1)p]$. Remplaçant p par $1/n$, on obtient en fin de compte $T_i = 2 - \frac{1}{n}$.

La complexité moyenne du tri des b_i, $i = 0..n$ est donc $T = \sum_{i=0}^{n-1} T_i = 2n - 1$. Le tri par paquets est donc linéaire en moyenne, à condition de faire des hypothèses de régularité de la distribution probabiliste des éléments du tableau.

VI Récurrences classiques

Les algorithmes précédents ont mis en évidence des suites récurrentes, dont certaines se doivent d'être connues

VI.1 Suites T(n)=T(n-1)+a

De telles suites interviennent fréquemment dans certains algorithmes, comme par exemple tri par sélection ou tri rapide en pire cas. Le calcul de T_n en fonction de n est immédiat et on obtient $T_n = O(n^2)$.

VI.2 Suites diviser pour régner. Cas particulier

On suppose dans tout ce qui suit que a et α sont réels, avec $a > 0$ et $\alpha \geq 0$.

Ces suites interviennent typiquement dans des algorithmes du type “diviser pour régner” comme le tri par fusion, l'exponentiation rapides, les diverses multiplications rapides (Karatsuba, Schonhage-Strassen), etc.

La partie entière pourrait être remplacée sans problème par une fonction “plafond” (pour $x \in \mathbf{R}$ le plafond de x est l'unique entier n, noté $\lceil x \rceil$, vérifiant $n - 1 < x \leq n$). On pourrait éventuellement envisager une combinaison des deux, avec des récurrences du genre $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + ...$

Dans ce paragraphe, nous allons résoudre ces récurrences dans le cas particulier où $n = 2^p$ est une puissance de 2. Définissons $t_p = T(2^p)$. On a alors la relation de récurrence

$$\forall p \geq 1 \qquad t_p = at_{p-1} + b2^{\alpha p}$$

Écrivons ces égalités en cascade, sommons pour éliminer les t intermédiaires. On obtient

$$t_p = a^p t_0 + b(2^\alpha a^{p-1} + 2^{2\alpha} a^{p-2} + ... + 2^{p\alpha} a^0)$$

On distingue trois cas, selon que a est égal à 2^α, strictement inférieur, ou strictement supérieur.

VI.2.1 Cas $a = 2^\alpha$

On a alors $t_p = 2^{\alpha p}(t_0 + b2^\alpha p) = O(p2^{\alpha p})$. En revenant à n, on a donc

$$T(n) = O(n^\alpha \lg n).$$

On rencontre ce cas lors de l'exponentiation dichotomique ($\alpha = 0$, $a = 1$) ou dans le tri par fusion ($\alpha = 1$, $a = 2$).

VI.2.2 Cas $a < 2^\alpha$

L'identité remarquable se transforme en

$$t_p = a^p t_0 + b2^\alpha \frac{2^{\alpha p} - a^p}{2^\alpha - a}$$

Ainsi, $t_p = O(2^{\alpha p})$, d'où

$$T(n) = O(n^\alpha).$$

Intuitivement, c'est le terme en n^α de la récurrence qui est prépondérant sur l'appel récurrent au rang $n/2$.

VI.2.3 Cas $a > 2^\alpha$

Le calcul fait ci-dessus pour la valeur de t_p reste valable, mais cette fois, on a $t_p = O(a^p)$, d'où

$$T(n) = O(n^{\lg a}).$$

Ce genre de situation se rencontre, par exemple, dans le cas de la multiplication rapide des matrices ($a = 7$, $\alpha = 2$) ou des polynômes.

Cette fois-ci, l'appel récurrent prend un temps prépondérant sur le “terme correcteur”.

VI.3 Suites diviser pour régner. Cas général

On ne fait plus d'hypothèse sur n dans ce paragraphe. Nous allons montrer que les résultats précédents restent toujours valables dans le cas général. Pour alléger un peu nos calculs introduisons la notation suivante :

Étant donnés p entiers u_i, $i = 0..p$, où les u_i valent 0 ou 1, et u_p est égal à 1, on note $[u_0, u_1, ..., u_p] = \sum_{i=0}^{p} u_i 2^i$. On convient également que $[0] = 0$. Tout entier naturel n s'écrit alors de façon unique $n = [u_0, ..., u_p]$ avec les conventions ci-dessus pour les u_i. L'entier p, en particulier, est entièrement caractérisé par n. D'ailleurs, pour n non nul, on a $2^p \leq n < 2^{p+1}$ ce qui implique que $p = \lfloor \lg n \rfloor$.

Un autre intérêt de cette notation est que si $n = [u_0, ..., u_p]$, alors $\lfloor n/2 \rfloor = [u_1, ..., u_p]$. Cette notation est en revanche peu adaptée lorsque l'on s'intéresse à des “plafonds” au lieu de parties entières.

Revenons à notre récurrence. Soit $n = [u_0, ..., u_p]$ un entier non nul. On a alors

$$T(n) = T([u_0, ..., u_p]) = aT(\lfloor n/2 \rfloor) + bn^\alpha = aT([u_1, ..., u_p]) + [u_0, ..., u_p]^\alpha$$

ou encore

$$T([u_0, ..., u_p]) = aT([u_1, ..., u_p]) + b2^{\alpha \lg [u_0, ..., u_p]}$$

On a ainsi la double inégalité

$$aT([u_1, ..., u_p] + b2^{\alpha p} \leq T([u_0, ..., u_p] < aT([u_1, ..., u_p] + b2^{\alpha(p+1)}$$

Soient t_p et t'_p les suites définies par récurrence par $t_0 = t'_0 = T(1)$ et, pour tout $p > 0$, $t_p = at_{p-1} + b2^{\alpha p}$ et $t'_p = at'_{p-1} + b2^\alpha 2^{\alpha p}$. On a $t_p \leq T([u_0, ..., u_p]) < t'_p$. Or, les suites t_p et t'_p sont du type de celles étudiées dans le cas particulier du paragraphe précédent (n puissance de 2) et leur comportement dépend de la position de a par rapport à 2^α. Dans le cas où, par exemple, $a > 2^\alpha$, on a

$$t_p = a^p t_0 + b2^\alpha \frac{2^{\alpha p} - a^p}{2^\alpha - a}$$

$$t'_p = a^p t'_0 + b2^{2\alpha} \frac{2^{\alpha p} - a^p}{2^\alpha - a}$$

On constate que $T(n)$ est coincé entre deux suites, équivalentes toutes deux à un multiple constant de a^p. On a donc $T(n) = O(a^p) = O(n^{\lg a})$. Les deux autres cas se traitent de façon identique.

Il est bon de remarquer que l'on a également $n^{\lg a} = O(T(n))$. Le rapport $\frac{T(n)}{n^{\lg a}}$ est à la fois majoré, et minoré par un réel strictement positif lorsque n tend vers l'infini. On note parfois $T(n) = \Theta(n^{\lg a})$.

VII Conclusion

Ce chapitre ne donne qu'un aperçu de ce que l'on peut faire en analyse d'algorithmes. Retenir de tout cela qu'il est possible de prouver les algorithmes. Évaluer leur complexité peut être instructif : l'analyse d'un algorithme permet en effet de cerner ses faiblesses, et parfois de les corriger.

Exercices

1/ La recherche du plus petit élément d'un tableau de taille n nécessite $n-1$ comparaisons. Déterminer un algorithme permettant de trouver le plus petit et le plus grand élément d'un tableau de taille n avec $3\lceil \frac{n}{2} \rceil$ comparaisons.
Indication : comparer les éléments du tableau deux par deux.

2/ La recherche d'un élément dans un tableau trié peut être effectuée avec $O(\lg n)$ comparaisons. Ecrire l'algorithme correspondant et prouver cette assertion.

3/ On considère l'algorithme suivant, renvoyant le plus grand élément d'un tableau :

```
let maximum t =
let n = Array.length t
and m = ref t.(0) in
for i = 0 to n - 1 do
    if t.(i) > !m then
(*)     m := t.(i)
done ;
!m;;
```

On souhaite déterminer le nombre moyen de fois que la ligne (*) est exécutée. On suppose que le tableau t est une permutation aléatoire de n nombres distincts.

(a) Soit x un nombre choisi parmi $i+1$ nombres distincts. Quelle est la probabilité p que x soit le plus grand de ces nombres ?

(b) Soit i un entier entre 0 et $n-1$. Quelle est la probabilité que la ligne (*) soit exécutée lors de la ième itération ?

(c) Soit s_i la variable aléatoire valant 0 si la ligne (*) n'est pas exécutée à la ième itération et 1 sinon. Montrer que la moyenne de s_i (son espérance mathématique) est égale à $\frac{1}{i+1}$.

(d) Soit s la variable aléatoire "nombre de fois que la ligne (*) est exécutée". Exprimer s à l'aide des s_i, en déduire la moyenne de s, et motrer que l'affectation (*) est réalisée en moyenne $\ln n$ fois sur un tableau de taille n.

4/ Cet exercice n'est pas censé être résolu de façon rigoureuse. On suppose donné un tableau tel que, lors du tri de ce dernier par quicksort, la fonction de partition coupe systématiquement dans une proportion de $\frac{99}{100}$ pour $\frac{1}{100}$ (au lieu du classique moitié-moitié). Montrer que ce n'est pas grave, et que le tableau sera quand même trié avec $O(n \lg n)$ comparaisons.

5/ On considère l'algorithme suivant (Karatsuba), permettant de multiplier deux polynômes P et Q de degré strictement inférieur à n, où n est une puissance de 2. Un polynôme est modélisé par le tableau de ses coefficients.

- ⋆ Si n=1, c'est facile.
- ⋆ Sinon, on écrit $P = P_1 + X^{n/2}P_2$ et $Q = Q_1 + X^{n/2}Q_2$, avec P_1, Q_1, P_2, Q_2 de degré strictement inférieur à $n/2$. Puis, on calcule récursivement les quantités $A = (P_1 + P_2)(Q_1 + Q_2)$, $B = P_1Q_1$ et $C = P_2Q_2$.

(a) Exprimer PQ à l'aide des quantités A, B et C.

(b) Écrire un algorithme récursif permettant le calcul de PQ. On supposera écrite une fonction `somme` permettant d'additionner deux polynômes de degré inférieur ou égal à n avec $n+1$ additions scalaires.

(c) Déterminer le nombre d'additions scalaires et de multiplications de coefficients effectuées par l'algorithme.
Peut-on qualifier cet algorithme de "multiplication rapide" de polynômes ?

6/ Permutations aléatoires. On considère l'algorithme suivant :

```
let random_vect n =
let t = Array.make n 0 in
for i = 0 to n - 1 do
    t.(i) <- i + 1
done ;
for i = 0 to n - 1 do
    let a = Random.int (i + 1)
    and b = t.(i) in
    t.(i) <- t.(a) ;
    t.(a) <- b
done ;
t ;;
```

Démontrer que cet algorithme fournit toutes les permutations de l'ensemble $\{1, 2, ..., n\}$ en $O(n)$ opérations élémentaires.

7/ Modifier la fonction de partition du tri rapide pour que, pour une liste L et un entier x, `partition x L` renvoie un triplet (L_1, L_2, q) où L_1 est la liste des éléments de L inférieurs ou égaux à x, L_2 est le reste des éléments de L, et q est le nombre d'éléments de L_1. On considère la fonction suivante :

```
let rec selection L i =
match L with
| [] -> failwith "selection"
| (x::L') -> let (L1,L2,q) = partition L' x in
    if i < q then selection L1 i
    else if i = q then x
    else selection L2 (i - q - 1) ;;
```

(a) Montrer que `selection L i` renvoie le $i$ème élément de la liste L dans l'ordre croissant.

(b) Déterminer l'ordre de grandeur du nombre moyen de comparaisons nécessaires pour le calcul de `selection L i`. On s'inspirera de l'étude faite en cours pour le tri rapide.

(c) Trouver le nombre de comparaisons dans le pire cas.
La valeur moyenne vous paraît-elle intéressante ? On aurait pu plus simplement trier la liste puis prendre le $i$ème élément de la liste triée. Aurait-on eu raison ?

8/ Soit $P = a_0 + a_1 X + ... + a_n X^n$ un polynôme à coefficients réels. Soit t un réel. Remarquer que $P = a_0 + X(a_1 + ... + a_n X^{n-1})$ et en déduire un algorithme de calcul de $P(t)$ ne nécessitant que n additions et $n + 1$ multiplications.

9/ L'algorithme de tri ci-dessous est appelé "tri par insertion".
Pour trier le tableau $t = [|a_0; ...; a_{n-1}|]$, on effectue pour chaque i entre 1 et $n - 1$ l'opération suivante : si $a_{i-1} > a_i$ on échange a_i et a_{i-1}. On recommence avec a_{i-1} et a_{i-2}, et ainsi de suite jusqu'à ce que l'élément envisagé soit supérieur au précédent.

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

```
    done
done ;;
```

(a) Montrer que l'on trie ainsi le tableau.

(b) Déterminer le nombre moyen de comparaisons effectuées par le tri par insertion.

(c) Déterminer le nombre maximal de comparaisons effectuées par ce même tri.

(d) Déterminer le nombre minimal de comparaisons effectuées par ce même tri.

10/ On considère la variante du tri rapide suivante :

On se donne un entier M. Si le nombre d'éléments de la liste L est strictement inférieur à M, on trie L à l'aide d'un tri par insertion (qui demandera $\frac{n(n-1)}{4}$ comparaisons en moyenne). Sinon, on partitionne et on rappelle récursivement l'algorithme. Le nombre moyen C_n de comparaisons nécessaires au tri d'une liste de taille n est alors donné par la récurrence

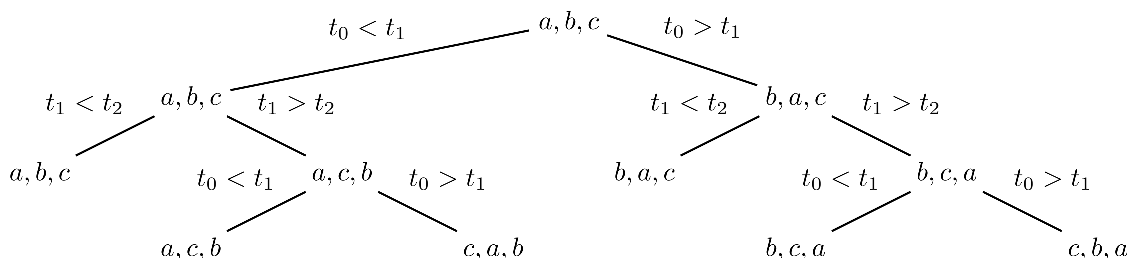
$$\begin{cases} C_n = n + 1 + \dfrac{1}{n} \displaystyle\sum_{k=0}^{n-1} (C_k + C_{n-1-k}) & \text{si} \quad n \geq M \\ C_n = \dfrac{n(n-1)}{4} & \text{si} \quad n < M \end{cases}$$

(a) Résoudre exactement cette récurrence.

(b) En supprimant les termes négligeables devant n dans l'expression de C_n, trouver une fonction $f(M)$ vérifiant $C_n = 2n \ln n + f(M)n + o(n)$.

(c) Déterminer numériquement le minimum de f.

11/ On dit qu'un algorithme de tri de tableau est *comparatif* lorsque cet algorithme effectue des comparaisons entre éléments du tableau pour effectuer le tri. À chaque algorithme de tri comparatif est associé un arbre binaire, résumant sur tous les échantillons de tableaux à trier la suite des comparaisons à effectuer. A titre d'exemple, voici l'abre associé au tri par insertion d'un tableau $t = [\![a, b, c]\!]$ contenant trois éléments distincts.



(a) Dresser l'arbre associé au tri rapide d'un tableau de taille 3.

(b) On appelle hauteur d'un arbre la distance maximale de la racine de l'arbre à ses feuilles. Combien un arbre binaire de hauteur h a-t-il au maximum de feuilles ? Combien l'arbre associé au tri d'un tableau de taille n doit-il avoir de feuilles ? Quelle est, en fonction de n, la hauteur minimale d'un tel arbre? En conclure un résultat concernant le nombre de comparaisons en pire cas pour les tris comparatifs.