Introduction à l'Algorithmique
ENSEIRB

Robert Cori

Année 2008-2009

Table des matières

Introduction

Ce cours qui est destiné à des élèves ingénieurs est une introduction à l'algorithmique, un concept au cœur de l'informatique. Il s'agit en illustrant par de nombreux d'exemples de montrer comment passer de la description informelle de la solution d'un problème à l'écriture d'un algorithme précis, contenant les détails des calculs et de leur enchaînement, permettant ainsi d'obtenir la solution effective de ce problème.

Cette étape est indispensable avant la réalisation d'un programme exécutable sur ordinateur donnant des résultats numériques.

Les problèmes que l'on considère ici relèvent des mathématiques discrètes : les objets sur lesquels on travaille sont des nombres entiers ou des ensembles finis, des mots ou *chaînes de caractères*, des arbres.

Dans ce cours, on se préoccupe beaucoup du temps mis pour effectuer un calcul, cela peut paraître étonnant alors que les processeurs sont de plus en plus rapides ; il faut toutefois noter que dans certains problèmes d'optimisation le nombre d'opérations à effectuer pour obtenir le résultat optimal est une fonction exponentielle de la taille des données ; le temps de calcul devient ainsi prohibitif dès que cette taille atteint quelques dizaines.

Un exemple simple est le suivant : on doit accomplir des tâches $T_1, T_2, \ldots, T_n$ le coût de la réalisation de l'ensemble de ces tâches dépend de l'ordre dans lequel elles sont accomplies ; pour déterminer le meilleur ordre il faut considérer tous ceux possibles et pour chacun d'entre eux déterminer son coût, or il y a $n!$ ordres possibles ce qui fait beaucoup : c'est un nombre qui croît de manière exponentielle avec n. Si la détermination du coût d'un ordre prend par exemple 10^{-8} secondes pour 20 tâches (ce qui est une performace à peine atteignable par les ordinateurs actuels , le temps de calcul pour évaleur tous les ordres sera de l'ordre de $(20!)10^{-8}$ secondes (soit de l' ordre de 2.10^{10} secondes), c'est à dire plusieurs milliers d'années. On remarque ainsi les limites de la puissance de l'informatique.

Les progrès de la vitesse des processeurs ne se traduisent en des gains substanciels d'efficacité des algorithmes, et donc en une amélioration tangible de la plus grande taille de problèmes solubles en quelques secondes, que pour les algorithmes de complexité polynomiale, comme l'illustre le tableau ci-dessous. On suppose que la vieille machine peut exécuter 10000 opérations en une seconde, et que la nouvelle machine est dix fois plus rapide. La première colonne donne la complexité de l'algorithme, la deuxième la plus grande taille de problème soluble en une seconde sur la vieille machine, la suivante donne la plus grande taille de problème soluble en une seconde sur la nouvelle machine. Il est clair que si l'algorithme utilisé est de complexité exponentielle, peu importe les progrès en architecture des machines : le problème restera de toute façon au-delà des capacités du programme.

$f(n)$	n	n'	Gain nouv machine	n'/n
$10n$	1000	10000	$n' = 10n$	10
$20n$	500	5000	$n' = 10n$	10
$5n \log n$	250	1842	$\sqrt{10}n \leq n' \leq 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
n^3	23	48	$n' = \sqrt[3]{10}n$	2.08
2^n	13	16	$n' = n + 3$	–

Sur ce tableau on voit aussi le seuil important franchi entre les algorithmes en $O(nlogn)$ et ceux en n^2 et un seuil encore plus important entre les algorithmes dont le temps est polynomial et ceux en temps exponentiel. Or il existe un grand nombre de problèmes pour lesquels on ne connaît pas d'algorithmes en temps polynomial permettant de les résoudre. C'est à la fois une question intrigante sur le plan conceptuel, et cruciale sur le plan pratique puisque de nombreux problèmes de cette famille se posent tous les jours dans les applications et que l'on espère pouvoir les résoudre effectivement.

CHAPITRE 1

La description des algorithmes élémentaires

Un algorithme est un processus qui à partir de certaines *données* donne un *résultat* qui est une fonction de ces données. Un algorithme est composé d'instructions élémentaires dont l'enchaînement permet d'obtenir le résultat. Des exemples simples de problèmes pour lesquels on cherche un algorithme sont les suivants :

- On dispose d'un annuaire du téléphone et on recherche le numéro d'un abonné à partir de son nom, de son prénom et de sa commune de résidence.
- la donnée est constituée de deux nombres entiers et on cherche leur plus grand diviseur commun.
- Dans un ouvrage de physique on cherche toutes les occurrences du mot "quantique".
- On se donne un ensemble de copies d'examen et on veut les trier par ordre alphabétique.

Une condition importante pour un algorithme est la condition de *terminaison*, un algorithme doit terminer son calcul en un temps fini. Un processus dont on ne sait s'il boucle indéfiniment n'est pas un algorithme.

1. Notation pour décrire les algorithmes

Nous avons choisi d'écrire les algorithmes avec une notation qui est très proche de celle du langage C, cela évite d'une part d'apprendre deux formalismes différents pour décrire un processus de calcul et d'autre part cela permet de passer très facilement de l'écriture d'un algorithme à sa réalisation sous forme de programme. Les différences entre notre notation et le langage C résident dans l'absence de déclaration des types des variables et des fonctions, un certain laxisme dans la gestion des passages de paramètres et dans l'écriture des fonctions d'affichage.

1.1. Variables, affectations. Un algorithme travaille sur un certain nombre de variables dont certaines contiennent les valeurs des données et d'autres devront contenir la *valeur* du résultat cherché à la fin de l'exécution de l'algorithme.

On donne des noms aux variables et on peut leur affecter une valeur. Une affectation s'écrit

```
x = E
```

Dans cette notation `x` est le nom d'une variable et `E` une expression calculable, cette expression peut contenir des noms de variables qui représentent les valeurs de celles-ci.

Une affectation est réalisée en deux temps

- Dans un premier temps l'expression qui se trouve à droite du signe `=` est évaluée, c'est à dire que les variables qui figurent dans cette expression sont remplacées par

leur valeur et les opérations qui figurent dans cette expression sont effectuées sur ces valeurs.
- Ensuite la valeur de la variable qui figure à gauche du signe = est modifiée, elle devient égale à ce qui a été trouvé lors de l'évaluation.

On utilisera le symbole ; pour terminer une instruction.

Un exemple d'une suite d'affectations est le suivant

```
x = 15;
y = 4;
r = x % y;
q = x / y;
```

Ici l'opération `%` désigne le reste de la division du nombre qui la précède par celui qui la suit, c'est l'opération arithmétique dite de *modulo*.

Noter que le nom d'une même variable peut figurer à gauche et à droite dans une affectation, cela signifie que la valeur de cette variable est mise à jour et que sa valeur actuelle sert à déterminer la nouvelle valeur qu'elle va prendre après l'affectation. Dans l'exemple suivant, la valeur de la variable `x` est multipliée par 2.

```
x = 2*x;
```

Une incrémentation d'une unité pour une variable peut s'écrire de façon concise à l'aide de l'opérateur `++`. L'instruction `i++ ;` ou de manière eáuivalente `++i ;` a pour effet d'augmenter de 1 la valeur de `i`.

1.2. Les instructions conditionnelles. Ces instructions permettent d'exécuter certaines opérations uniquement dans le cas où une condition est vérifiée et éventuellement d'exécuter d'autres opérations si elle ne l'est pas. On les écrit sous l'une des formes suivantes :

```
if (E) INSTRUCTION
if (E){ SUITE-INSTRUCTIONS}
if (E) INSTRUCTION1
    else INSTRUCTION2
```

ici `E` est une expression qui s'évalue à *vrai* ou $faux$, c'est ce que l'on appelle une expression booléenne . Pour les deux premieres formes d'instruction conditionnelle si `E` est évaluée à *vrai* l'instruction qui suit (ou la suite d'instructions entre accolades qui suit) est exécutée, sinon on passe à ce qui se trouve plus loin. Pour la troisième forme si `E` est évaluée à *vrai* c'est l'instruction `INSTRUCTION1` qui est exécutée, si elle est évaluée à $faux$ c'est l'instruction `INSTRUCTION2` qui est exécutée.

Une expression booléénne peut contenir des opérateurs de comparaisons entre des nombres comme par exemple :

`==`	égalité
`!=`	différent de
`<, <=`	inférieur, inférieur ou égal
`>, >=`	supérieur, supérieur ou égal

Ainsi `a == b` est évaluée à *vrai* si et seulement si les variables `a` et `b` ont la même valeur, `x >= 0` est évaluée à *vrai* si et seulement si la variable `x` a une valeur supérieure ou égale à 0.

On peut combiner des opérateurs de comparaison à l'aide des opérateurs ET, OU, NON que l'on décrit ci dessous.

– L'opérateur ET, noté `&&` est évalué à *vrai* si et seulement si les deux opérandes sont évalués à *vrai*, ce qui se résume par la table :

`&&`	*vrai*	*faux*
vrai	*vrai*	*faux*
faux	*faux*	*faux*

– L'opérateur : OU noté , `||` est évalué à *vrai* si au moins un des deux opérandes a pour valeur *vrai*.

`||`	*vrai*	*faux*
vrai	*vrai*	*vrai*
faux	*vrai*	*faux*

– et l'opérateur : NON noté `!` donne l'opposé de la valeur de son opérande.

`!`	
vrai	*faux*
`false`	*vrai*

Les deux conventions suivantes facilitent l'écriture de certains algorithmes :

– On n'évalue pas le deuxième opérande d'un `&&` si le premier donne la valeur `false`
```
if ((y != 0) && (x/y > 1))
```

– On n'évalue pas le deuxième opérande d'un `||` si le premier donne la valeur *vrai*
```
if ((y == 0) || (x/y > 1))
```

1.3. Les boucles de programme.

1.3.1. *Boucle* `while`.

– Cette boucle permet d'effectuer une instruction (ou une suite d'instructions entre accolades) tant qu'une expression booléenne est évaluée à *vrai*

– On réévalue l'expression à chaque tour de boucle jusqu'à ce que l'évaluation donne la valeur *faux*

Cette boucle s'écrit

```
while (EXP)
     INSTRUCTION

while (EXP)
     {SUITE-INSTRUCTIONS}
```

Noter que le corps de la boucle n'est jamais exécuté si la première évaluation de `EXP` donne la valeur $faux$.

1.3.2. *Boucle* `for`. Cette boucle permet de parcourir un ensemble de valeurs avec incrémentation à chaque tour de boucle elle est équivalent à une boucle `while`. Son utilisation permet de rendre les programmes plus clairs.

```
for(INSTR1; CONDITION; INSTR2)
       INSTRUCTION3
```

Cette instruction peut être remplacée par une boucle `while` de la façon suivante :

```
INSTR1
  while(CONDITION){
     INSTRUCTION3;
     INSTR2;
 }
```

1.4. Quelques conventions.

1.4.1. *Bloc d'instructions.* Une suite d'instructions séparées par des ; et qui est entourée d'accolades constitue un bloc et se comporte comme une seule instruction.

1.4.2. *Affichage.* On ne se préoccupe pas en algorithmique des détails d'affichage des résultats, on utilisera si nécessaire l'instruction `afficher(x, y, z, ...)` dont on supposera qu'elle affiche la valeur des variables `x, y, z, ....`

1.4.3. *Procèdures, fonctions.* Il est parfois plus clair de présenter un algorithme sous la forme d'un ensemble de fonctions, lesquelles peuvent être utilisées par d'autres fonctions. Une fonction pourra ainsi calculer une valeur à partir de paramètres et rendre un résultat à celle qui lui a fait appel. La valeur rendue par une fonction sera donnée après le mot `return` et l'appel d'une fonction sera fait en indiquant son nom et les valeurs des paramètres entre parenthèses.

2. Exemples simples

2.1. Le PGCD. L'algorithme de calcul du plus grand diviseur commun à deux nombres a et b est connu depuis l'antiquité, il est souvent appelé algorithme d'Euclide. Il part du principe que si b divise a alors ce pgcd est égal à b, sinon le reste de la division de a par b est aussi divisible par ce pgcd. On procède ainsi par étapes en remplaçant à chaque étape le couple a, b par b, r où r est le reste de la division de a par b. Le calcul se termine quand le reste est égal à 0. Noter que l'on n'a pas besoin de vérifier au départ que a est supérieur à b, en effet si tel n'est pas le cas, le reste de la divison de a par b sera a est la première étape consistera à remplacer a, b par b, a. Nous écrivons cet algorithme sous la forme d'une

fonction qui prend pour paramètres les nombres a et b, qui effectue le calcule et qui rend le résultat à une fonction qui l'appelle.

```
pgcd(a,b){
   while (b != 0) {
     r = a % b;
     a = b;
     b = r;
   }
return a;
}
```

2.2. Nombres en base 2. Tout nombre entier s'écrit de façon unique sous la forme d'une somme de puissances de 2 toutes différentes. Par exemple 2007 s'écrit

$$2007 = 1024 + 512 + 256 + 128 + 64 + 16 + 4 + 2 + 1$$

soit

$$2007 = 2^{10} + 2^9 + 2^8 + 2^7 + +2^6 + 2^4 + 2^2 + 2^1 + 2^0$$

On dit alors que l'écriture de 2007 en base 2 est 11111010111 plus généralement l'écriture de n en base 2 est égale à $c_k c_{k-1} \dots c_2 c_1 c_0$ si les nombres c_i sont des 0 ou des 1 et si :

$$n = \sum_{i=0}^{k} c_i 2^i$$

L'algorithme d'écriture de n en base 2 consiste à remarquer que si n est impair le dernier chiffre est un 1, et que si n est pair alors le dernier chiffre est un 0. Ensuite il faut répéter l'opération sur le quotient entier de n par 2.

Ainsi l'algorithme donné ci dessous affiche l'écriture de n en base 2 en commençant par le dernier chiffre et en procédant de droite à gauche :

```
while (a != 0){
    afficher(a%2);
    a = a/2;
}
```

Dans cette écriture le symbole / désigne la division entière de deux nombres, ainsi $2007/2 = 1003$.

2.3. Détenir un secret. Afin de pouvoir échanger des messages en toute sécurité Alice et Bob doivent connaître ensemble un nombre assez grand qui va servir au codage, un nombre dont ils seront les seuls à détenir la valeur. Mais comment faire pour déterminer sachant qu'ils sont éloignés l'un de l'autre et que leurs communications peuvent être interceptées ? Alors Alice qui a fait des études de cryptographie propose un algorithme, elle envoie à Bob deux nombres k et m qui peuvent être rendus publics (accessibles à tout le monde). Elle demande à Bob de choisir un nombre b très grand, de calculer $u = (k^b \mod m)$ et de lui envoyer cette valeur. il faut noter que la connaissance de u, k et m ne permet pas de déterminer b en temps raisonable, ceci même en disposant de moyens de

calcul gigantesques. De manière symétrique Alice chosit un nombre a très grand, calcule $v = (k^a \mod m)$ et envoie le résultat à Bob. Il est alors immédiat de vérifier que $(v^b \mod m)$ et $(u^a \mod m)$ sont égaux et que nul ne peut deviner cette valeur ; alors que Bob qui connait v et b peut déterminer celle-ci, de même pour Alice qui connaît u et a. Reste donc à trouver un moyen simple de calculer $u^a \mod m$ pour des nombres très grands. ceci sera vu en TDs.

3. Tableaux

3.1. Notation. Les tableaux sont introduits afin de pouvoir manipuler un très grand nombre de variables en utilisant un seul nom pour le tableau et un indice qui permet d'avoir accès à toutes les variables. En mathématiques, pour une suite on utilise un nom u et des indices : u_i, pour des raisons historiques en informatique on ne peut pas faire figurer l'indice en dessous du nom de la variable, on utilise ainsi la notation `u[i]`. Comme dans la plupart des langages de programmation (dont C), les indices partent systématiquement de 0, nous adopterons la même convention : les valeurs contenues dans un tableau `u` de taille `n` seront :

```
u[0], u[1], .... , u[n-1]
```

Dans cette notation `u[i]` se comporte comme une variable ordinaire, elle a une valeur qui peut être affectée et évaluée.

3.2. Manipulations de tableaux. On présente ainsi quelques algorithmes simples sur les tableaux :

– Somme des élements

```
sommeElements(t, n){
   s = t[0];
   for (i = 1; i < n; ++i)
       s = s + t[i];
   return s;
}
```

– Produit scalaire

```
 prodScal(u,  v, n){
   s = 0;
   for (i = 0; i < n; ++i)
       s = s + u[i] * v[i];
   return s;
}
```

– Somme de deux vecteurs

```
   somme (u, v, n){
      for ( i = 0; i < n; ++i)
          w[i] = u[i] + v[i];
      return w;
    }
```

Tableaux de tableaux (matrices)

Une matrice est un tableau composé de n lignes dont chacune contient m nombres, un exemple d'initialisation des valeurs d'une matrice :

```
for (i = 0; i < n; ++i)
    for (j = 0; j < m; ++j)
        mat[i][j] = i*j +1;
```

Exemple : produit d'une matrice par un vecteur

```
multiplie ( mat, v, n, m){
   for (i = 0; i < n; ++i){
       res[i] = 0;
       for (j = 0; j < m; ++j)
           res[i] = res[i] + mat[i][j]*v[j];
       }
   return res;
}
```

3.3. Tri.

3.3.1. *Recherche du plus petit élément.* La recherche du plus petit élément dans le tableau `tab` s'effectue en conservant le plus petit élément parmi ceux déjà examinés dans une variable `minT`. On initialise cette valeur par le premier élément du tableau, puis on examine les éléments du tableau les uns à la suite des autres et on modifie éventuellement la valeur du plus petit élément trouvé :

```
minT = tab[0];
for (j = 1; j < n; ++j)
        if (tab[j] < minT)
              minT = tab[j];
```

On peut aussi chercher l'indice du plus petit élément :

```
for (j = i+1; j < n; ++j)
        if (tab[jMin] > tab[j])
              jMin = j;
```

3.3.2. *Tri par sélection.* Le tri par selection consiste à chercher l'indice du plus petit élément, à echanger cet élément avec `tab[0]` puis à recommencer avec le tableau `tab[1]`, `...`, `tab[n-1]` et ainsi de suite. Cela donne :

```
for (int i = 0; i < n; ++i){
    int jMin = i;
    for (j = i+1; j < n; ++j)
       if (tab[jMin] > tab[j])
             jMin = j;
    int temp = tab[jMin};
```

```
        tab[jMin] = tab[i];
        tab[i] = temp;
    }
```

CHAPITRE 2

Récursivité

Les définitions par récurrence sont assez courantes en mathématiques. Prenons le cas de la suite de Fibonacci, définie par

$u_0 = u_1 = 1$
$u_n = u_{n-1} + u_{n-2}$ pour $n > 1$

On obtient donc la suite 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946,... Nous allons voir que ces définitions s'implémentent très simplement en informatique par les définitions récursives.

1. Fonctions récursives

1.1. Fonctions numériques. Pour calculer la suite de Fibonacci, une transcription littérale de la formule est la suivante :

```
fib (n) {

  if (n <= 1)
      return n;
  else
      return fib (n-1) + fib (n-2);
}
```

`fib` est une fonction qui utilise son propre nom dans la définition d'elle-même. Ainsi, si l'argument n est plus petit que 1, on retourne comme valeur 1. Sinon, le résultat est `fib`$(n-1)+$`fib`$(n-2)$. Il est donc possible en programmation, de définir de telles fonctions *récursives*. D'ailleurs, toute suite $\langle u_n \rangle$ définie par récurrence admet une écriture de cette manière, comme le confirment les exemples numériques suivants : factorielle et le triangle de Pascal.

```
fact(n)  {

  if (n <= 1)
      return 1;
  else
      return n * fact (n-1);
}


C(n, p)  {
```

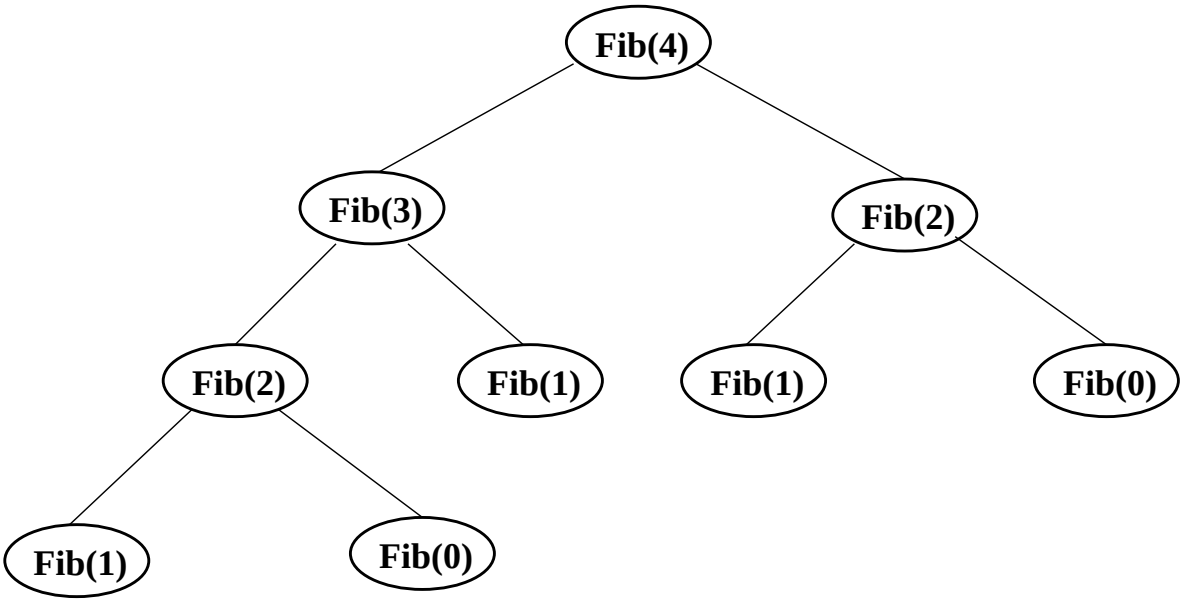


FIG. 1. Appels récursifs pour `fib`(4)

```
  if ((n == 0) || (p == n))
      return 1;
  else
      return C(n-1, p-1) + C(n-1, p);
}
```

On peut se demander comment l'ordinateur s'y prend pour faire le calcul des fonctions récursives. Nous allons essayer de le suivre sur le calcul de `fib`(4). Rappelons nous que les arguments sont transmis par valeur dans le cas présent, et donc qu'un appel de fonction consiste à évaluer l'argument, puis à se lancer dans l'exécution de la fonction avec la valeur de l'argument. Donc

```
fib(4) -> fib (3) + fib (2)
       -> (fib (2) + fib (1)) + fib (2)
       -> ((fib (1) + fib (1)) + fib (1)) + fib(2)
       -> ((1 + fib(1)) + fib (1)) + fib(2)
       -> ((1 + 1) + fib (1)) + fib(2)
       -> (2 + fib(1)) + fib(2)
       -> (2 + 1) + fib(2)
       -> 3 + fib(2)
       -> 3 + (fib (1) + fib (1))
       -> 3 + (1 + fib(1))
       -> 3 + (1 + 1)
       -> 3 + 2
       -> 5
```

Il y a donc un bon nombre d'appels successifs à la fonction `fib` (9 pour `fib`(4)). Comptons le nombre d'appels récursifs R_n pour cette fonction. Clairement $R_0 = R_1 = 1$, et $R_n = 1 + R_{n-1} + R_{n-2}$ pour $n > 1$. En posant $R'_n = R_n + 1$, on en déduit que $R'_n = R'_{n-1} + R'_{n-2}$ pour $n > 1$, $R'_1 = R'_0 = 2$. D'où $R'_n = 2 \times \texttt{fib}(n)$, et donc le nombre d'appels récursifs R_n vaut $2 \times \texttt{fib}(n) - 1$, c'est à dire 1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, 1973, 3193, 5167, 8361, 13529, 21891,... Le nombre d'appels récursifs est donc très élevé, d'autant plus qu'il existe une méthode itérative simple en calculant simultanément `fib`(n) et `fib`$(n-1)$. En effet, on a un calcul linéaire simple

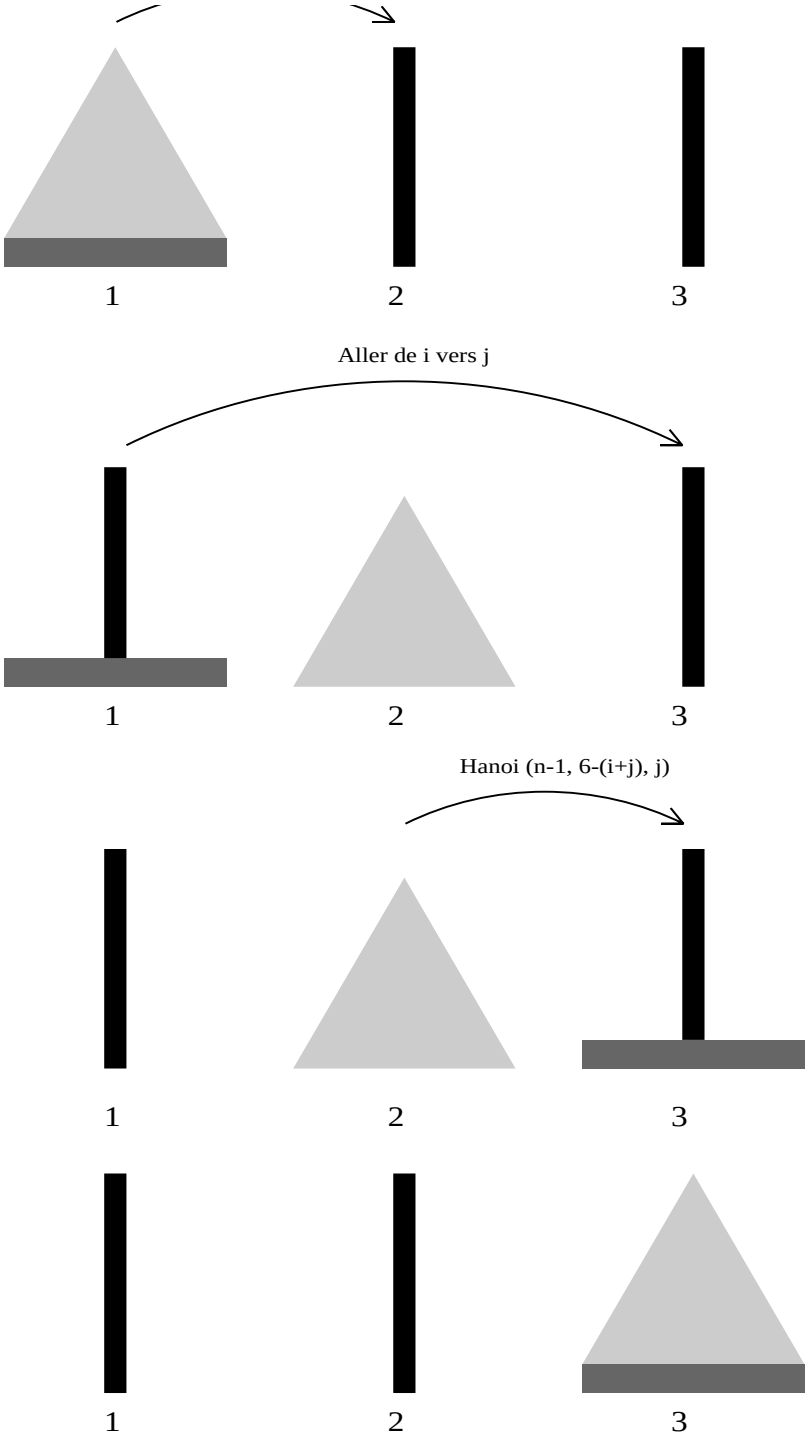
$$\begin{pmatrix} \texttt{fib}(n) \\ \texttt{fib}(n-1) \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0cd \end{pmatrix} \begin{pmatrix} \texttt{fib}(n-1) \\ \texttt{fib}(n-2) \end{pmatrix}$$

Ce qui donne le programme itératif suivant

```
fibIter(n) {
  u = 1; v = 1;
  for (i = 2; i <= n; ++i) {
      u0 = u; v0 = v;
      u = u0 + v0;
      v = u0;
  }
  return u;
}
```

Pour résumer, une bonne règle est de ne pas trop essayer de suivre dans les moindres détails les appels récursifs pour comprendre le sens d'une fonction récursive. Il vaut mieux en général comprendre synthétiquement la fonction. La fonction de Fibonacci est un cas particulier car son calcul récursif est particulièrement long. Mais ce n'est pas le cas en général. Non seulement, l'écriture récursive peut se révéler efficace, mais elle est toujours plus naturelle et donc plus esthétique. Elle ne fait que suivre les définitions mathématiques par récurrence. C'est une méthode de programmation très puissante.

2. Les tours de Hanoï



Dans les exemples vus plus haut on pouvait aussi bien écrire une fonction itérative qu'une fonction récursive, la seconde pouvait être préférable car plus élégante. Mais il y a des cas où l'écriture d'une fonction itérative est pratiquement impossible alors que la fonction récursive est très simple : l'illustration la plus classique de cette situation est le problème des tours de Hanoï.

On a 3 piquets en face de soi, numérotés 1, 2 et 3 de la gauche vers la droite, et n rondelles de tailles toutes différentes entourant le piquet 1, formant un cône avec la plus grosse en bas et la plus petite en haut. On veut amener toutes les rondelles du piquet 1 au piquet 3 en ne prenant qu'une seule rondelle à la fois, et en s'arrangeant pour qu'à tout moment il n'y ait jamais une rondelle sous une plus grosse. La légende dit que les bonzes passaient leur vie à Hanoï à résoudre ce problème pour $n = 64$, ce qui leur permettait d'attendre l'écroulement du temple de Brahma, et donc la fin du monde (cette légende fut inventée par le mathématicien français E. Lucas en 1883). Un raisonnement par récurrence permet de trouver la solution en quelques lignes. Si $n \leq 1$, le problème est trivial. Supposons maintenant le problème résolu pour $n-1$ rondelles pour aller du piquet i au piquet j. Alors, il y a une solution très facile pour transférer n rondelles de i en j :

1- on amène les $n-1$ rondelles du haut de i sur le troisième piquet $k = 6-i-j$,

2- on prend la grande rondelle en bas de i et on la met toute seule en j,

3- on amène les $n-1$ rondelles de k en j.

Ceci s'écrit

```
hanoi(n, i, j)  {

  if (n > 0) {
      hanoi (n-1, i, 6-(i+j));
      afficher(i," -> ", j);
      hanoi (n-1, 6-(i+j), j);
  }
}
```

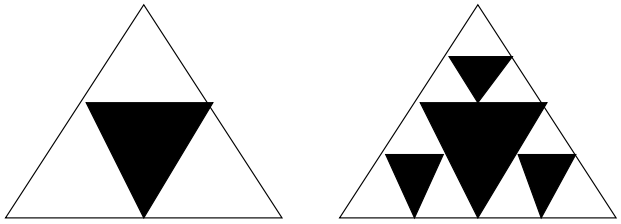
Ces quelques lignes de programme montrent bien comment en généralisant le problème, c'est-à-dire aller de tout piquet i à tout autre j, un programme récursif de quelques lignes peut résoudre un problème a priori compliqué. C'est la force de la récursion et du raisonnement par récurrence. Il y a bien d'autres exemples de programmation récursive, et la puissance de cette méthode de programmation a été étudiée dans la théorie dite de la *récursivité* qui s'est développée bien avant l'apparition de l'informatique (Kleene [**?**], Rogers [**?**]). Le mot récursivité n'a qu'un lointain rapport avec celui qui est employé ici, car il s'agissait d'établir une théorie abstraite de la calculabilité, c'est à dire de définir mathématiquement les objets qu'on sait calculer, et surtout ceux qu'on ne sait pas calculer. Mais l'idée initiale de la récursivité est très probablement à attribuer à Kleene (1935).

3. Dessins de courbes fractales

Considérons d'autres exemples de programmes récursifs. Des exemples imagés sont le cas de dessins graphiques et certaines fonctions fractales.

3.1. Figure de Sierpinski. Le figure de Sierpinski est souvent définie comme suit :
- La figure d'ordre 0, notée S_0 est un triangle équilatéral plein.
- La figure d'ordre 1 est obtenue en découpant dans S_0 le triangle dont les sommets sont les milieux des cotés du triangle S_o
- La figure de Sierpinski S_n d'ordre n est obtenue en découpant à l'intérieur de tous les triangles pleins de S_{n-1}, le triangle formé par les milieux de tous les cotés.

Par exemple les figures d'ordre 1 et 2 sont dessinés ci dessous



On se propose d'écrire un programme qui dessine la figure d'ordre n en ayant simplement pour primitive un ordre de dessin d'un segment de droite `moveto(a,)` déplace le crayon au point de coordonnées `a,b`, `lineto(c,d)` trace un segment du point courant au point de coordonnées `c,d` tout en déplaçant le crayon en ce point.

3.2. Autres fonctions fractales. Un premier exemple simple de fonction fractale est le flocon de von Koch qui est défini comme suit :

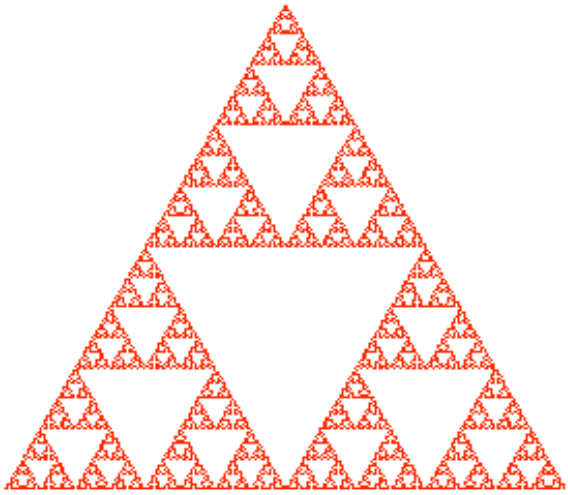
> Le flocon d'ordre 0 est un triangle équilatéral.
> Le flocon d'ordre 1 est ce même triangle dont les côtés sont découpés en trois et sur lequel s'appuie un autre triangle équilatéral au milieu.
> Le flocon d'ordre $n+1$ consiste à prendre le flocon d'ordre n en appliquant la même opération sur chacun de ses côtés.

Le résultat ressemble effectivement à un flocon de neige idéalisé. L'écriture du programme est laissé en exercice. On y arrive très simplement en utilisant les fonctions trigonométriques `sin` et `cos`.

Un autre exemple classique est la courbe du Dragon. La définition de cette courbe est la suivante : la courbe du Dragon d'ordre 1 est un segment entre deux points quelconques P et Q, la courbe du Dragon d'ordre n est la courbe du Dragon d'ordre $n-1$ entre P et R suivie de la même courbe d'ordre $n-1$ entre R et Q (à l'envers), où PRQ est le triangle isocèle rectangle en R, et R est à droite du vecteur PQ. Donc, si P et Q sont les points de coordonnées (x, y) et (z, t), les coordonnées (u, v) de R sont

$$u = (x + z)/2 + (t - y)/2$$
$$v = (y + t)/2 - (z - x)/2$$

La courbe se programme simplement par :



```
dragon(n, x, y, z, t) {

  if (n == 1) {
      moveTo (x, y);
      lineTo (z, t);
  } else {
      u = (x + z + t - y) / 2;
      v = (y + t - z + x) / 2;
      dragon (n-1, x, y, u, v);
      dragon (n-1, z, t, u, v);
  }
}
```

Si on calcule `dragon` $(20, 20, 20, 220, 220)$, on voit apparaître un petit dragon. Cette courbe est ce que l'on obtient en pliant 10 fois une feuille de papier, puis en la dépliant. Une autre remarque est que ce tracé lève le crayon, et que l'on préfère souvent ne pas lever le crayon pour la tracer. Pour ce faire, nous définissons une autre procédure `dragonBis` qui dessine la courbe à l'envers. La procédure `dragon` sera définie récursivement en fonction de `dragon`

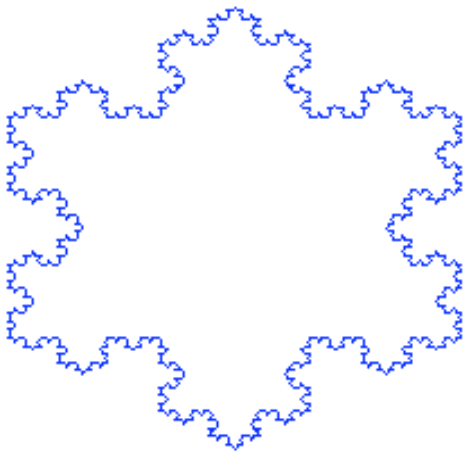


FIG. 2. Flocon de von Koch

et `dragonBis`. De même, `dragonBis` est définie récursivement en termes de `dragonBis` et `dragon`. On dit alors qu'il y a une *récursivité croisée*.

```
dragon (n, x, y, z,  t)  {

  if (n == 1) {
      moveTo (x, y);
      lineTo (z, t);
  } else {
      u = (x + z + t - y) / 2;
      v = (y + t - z + x) / 2;
      dragon (n-1, x, y, u, v);
      dragonBis (n-1, u, v, z, t);
  }
}

dragonBis(n, x, y, z, t)  {

  if (n == 1) {
      moveTo (x, y);
      lineTo (z, t);
```

FIG. 3. La courbe du Dragon

```
  } else {
      u = (x + z - t + y) / 2;
      v = (y + t + z - x) / 2;
      dragon (n-1, x, y, u, v);
      dragonBis (n-1, u, v, z, t);
  }

}
```

Il y a bien d'autres courbes fractales comme la courbe de Hilbert, courbe de Peano qui recouvre un carré, les fonctions de Mandelbrot. Ces courbes servent en imagerie pour faire des parcours "aléatoires" de surfaces, et donnent des fonds esthétiques à certaines images.

3.3. La fonction d'Ackermann. La suite de Fibonacci avait une croissance exponentielle. Il existe des fonctions récursives qui croissent encore plus rapidement. Le prototype est la fonction d'Ackermann. Au lieu de définir cette fonction mathématiquement, il est aussi simple d'en donner la définition récursive en Java

```
static ack(m, n) {

  if (m == 0)
      return n + 1;
  else
      if (n == 0)
          return ack (m - 1, 1);
      else
          return ack (m - 1, ack (m, n - 1));
}
```

On peut vérifier que

$$\begin{aligned}
&\texttt{ack}(0,n) = n+1\\
&\texttt{ack}(1,n) = n+2\\
&\texttt{ack}(2,n) \simeq 2*n\\
&\texttt{ack}(3,n) \simeq 2^n\\
&\texttt{ack}(4,n) \simeq 2^{\overset{\overset{\cdot^{\cdot^{\cdot}}}{2}}{\left.\vphantom{\overset{\cdot^{\cdot^{\cdot}}}{2}}\right\}n}}
\end{aligned}$$

Donc $\texttt{ack}(5,1) \simeq \texttt{ack}(4,4) \simeq 2^{65536} > 10^{80}$, c'est à dire le nombre d'atomes de l'univers.

3.4. Récursion imbriquée. La fonction d'Ackermann contient deux appels récursifs imbriqués, c'est ce qui la fait croître si rapidement. Un autre exemple est la fonction 91 de MacCarthy

```
f(n)  {

  if (n > 100)
      return n - 10;
  else
      return f(f(n+11));
}
```

Ainsi, le calcul de $\texttt{f}(96)$ donne $\texttt{f}(96) = \texttt{f}(\texttt{f}(107)) = \texttt{f}(97) = \cdots \texttt{f}(100) = \texttt{f}(\texttt{f}(111)) = \texttt{f}(101) = 91$. On peut montrer que cette fonction vaut 91 si $n \leq 100$ et $n - 10$ si $n > 100$. Cette fonction anecdotique, qui utilise la récursivité imbriquée, est intéressante car il n'est pas du tout évident qu'une telle définition donne toujours un résultat. Par exemple, la fonction de Morris suivante

```
 g(m, n) {

  if (m == 0)
      return 1;
  else
      return g(m - 1, g(m, n));
}
```

Que vaut alors $\mathtt{g}(1,0)$? En effet, on a $\mathtt{g}(1,0) = \mathtt{g}(0,\mathtt{g}(1,0))$. Il faut se souvenir que Java passe les arguments des fonctions par valeur. On calcule donc toujours la valeur de l'argument avant de trouver le résultat d'une fonction. Dans le cas présent, le calcul de `g(1,0)` doit recalculer `g(1, 0)`. Et le calcul ne termine pas.

CHAPITRE 3

Structures de données élémentaires

Dans ce chapitre, nous introduisons quelques structures utilisées de façon très intensive en programmation. Leur but est de gérer un ensemble fini d'éléments dont le nombre n'est pas fixé *a priori*. Les éléments de cet ensemble peuvent être de différentes sortes : nombres entiers ou réels, chaînes de caractères, ou des objets informatiques plus complexes comme les identificateurs de processus ou les expressions de formules en cours de calcul ... On ne s'intéressera pas aux éléments de l'ensemble en question mais aux opérations que l'on effectue sur cet ensemble, indépendamment de la nature de ses éléments. Ainsi les ensembles que l'on utilise en programmation, contrairement à ceux considérés en mathématiques qui sont fixés une fois pour toutes, sont des objets dynamiques. Le nombre de leurs éléments varie au cours de l'exécution du programme, puisqu'on peut y ajouter et supprimer des éléments en cours de traitement. Plus précisément les opérations que l'on s'autorise sur les ensembles sont les suivantes :

- *tester* si l'ensemble E est vide.
- *ajouter* l'élément x à l'ensemble E.
- *supprimer* un élément de l'ensemble E.

Cette gestion des ensembles doit, pour être efficace, répondre au mieux à deux critères parfois contradictoires : un minimum de place mémoire utilisée et un minimum d'instructions élémentaires pour réaliser une opération. La place mémoire utilisée devrait pour bien faire être très voisine du nombre d'éléments de l'ensemble E, multipliée par leur taille. En ce qui concerne la minimisation du nombre d'instructions élémentaires, on peut tester très simplement si un ensemble est vide et on peut réaliser l'opération d'ajout en quelques instructions, toutefois il est impossible de réaliser une suppression ou une recherche d'un élément quelconque dans un ensemble en utilisant un nombre d'opérations indépendant du cardinal de cet ensemble (à moins d'utiliser une structure demandant une très grande place en mémoire). Pour améliorer l'efficacité, on considère des structures de données dans lesquelles on restreint la portée des opérations de recherche et de suppression d'un élément en se limitant à la réalisation de ces opérations sur le dernier ou le premier élément de l'ensemble, ceci donne les structures de pile ou de file, nous verrons que malgré ces restrictions les structures en question ont de nombreuses applications.

1. Piles

La notion de pile intervient couramment en programmation, son rôle principal consiste à implémenter les appels de procédures. Nous n'entrerons pas dans ce sujet, plutôt technique, dans ce chapitre. Nous montrerons le fonctionnement d'une pile à l'aide d'exemples

choisis dans l'évaluation d'expressions Lisp.

On peut imaginer une pile comme une boîte dans laquelle on place des objets et de laquelle on les retire dans un ordre inverse de celui dans lequel on les a mis : les objets sont les uns sur les autres dans la boîte et on ne peut accéder qu'à l'objet situé au "sommet de la pile". De façon plus formelle, on se donne un ensemble E, l'ensemble des piles dont les éléments sont dans E est noté $Pil(E)$, la pile vide (qui ne contient aucun élément) est P_0, les opérations sur les piles sont *vide, ajouter, valeur, supprimer* comme sur les files. Cette fois, les relations satisfaites sont les suivantes (où P_0 dénote la pile vide)

$supprimer(ajouter(x, P)) = P$
$estVide(ajouter(x, P)) = faux$
$valeur(ajouter(x, P)) = x$
$estVide(P_0) = vrai$

A l'aide de ces relations, on peut exprimer toute expression sur les piles faisant intervenir les 4 opérations précédentes à l'aide de la seule opération $ajouter$ en partant de la pile P_0. Ainsi l'expression suivante concerne les piles sur l'ensemble des nombres entiers :

$$supprimer\ (ajouter\ (7,\ supprimer\ (ajouter\ (valeur\ (ajouter\ (5,\ ajouter\ (3, P_0)))),\ ajouter\ (9, P_0))))$$

Elle peut se simplifier en :

$ajouter(9, P_0)$

La réalisation des opérations sur les piles peut s'effectuer en utilisant un tableau qui contient les éléments et un indice qui indiquera la position du sommet de la pile. par référence.

```
estVide () {
  hauteur == 0;
}
ajouter(x){
  contenu[hauteur] = x;
  ++hauteur;
}

valeur() {
  if (!estVide (p))
  return contenu [hauteur - 1];
  else affiche ("pile vide");
 }

supprimer () {
```

```
    if (!estVide (p))
          --hauteur;
     else {affiche ("pile vide");
  }
}
```

Remarques. Chacune des opérations sur les piles demande un très petit nombre d'opérations élémentaires et ce nombre est indépendant du nombre d'éléments contenus dans la pile.

2. Evaluation des expressions arithmétiques préfixées

Dans cette section, on illustre l'utilisation des piles par un programme d'évaluation d'expressions arithmétiques écrites de façon particulière. Rappelons qu'expression arithmétique signifie dans le cadre de la programmation : expression faisant intervenir des nombres, des variables et des opérations arithmétiques (par exemple : $+ * / - \sqrt{\ }$). Dans ce qui suit, pour simplifier, nous nous limiterons aux opérations binaires $+$ et $*$ et aux nombres naturels. La généralisation à des opérations binaires supplémentaires comme la division et la soustraction est particulièrement simple, c'est un peu plus difficile de considérer aussi des opérations agissant sur un seul argument comme la racine carrée, cette généralisation est laissée à titre d'exercice au lecteur. Nous ne considérerons aussi que les entiers naturels en raison de la confusion qu'il pourrait y avoir entre le symbole de la soustraction et le signe moins.

Sur certaines machines à calculer de poche, les calculs s'effectuent en mettant le symbole d'opération après les nombres sur lesquels on effectue l'opération. On a alors une notation dite *postfixée*. Dans certains langages de programmation, c'est par exemple le cas de Lisp ou de Scheme, on écrit les expressions de façon *préfixée* c'est-à-dire que le symbole d'opération précède cette fois les deux opérandes, on définit ces expressions récursivement. Les expressions préfixées comprennent :

- des symboles parmi les 4 suivants : `+  *  (  )`
- des entiers naturels

Une *expression préfixée* est ou bien un nombre entier naturel ou bien est de l'une des deux formes :

`(+ e1 e2)` `(* e1 e2)`

où e_1 et e_2 sont des *expressions préfixées*.

Cette définition fait intervenir le nom de l'objet que l'on définit dans sa propre définition mais on peut montrer que cela ne pose pas de problème logique. En effet, on peut comparer cette définition à celle des nombres entiers : "tout entier naturel est ou bien l'entier 0 ou bien le successeur d'un entier naturel". On vérifie facilement que les suites de symboles suivantes sont des expressions préfixées.

```
47
(* 2 3)
(+ 12 8)
(+ (* 2 3) (+ 12 8))
```

```
(+ (*  (+ 35 36) (+ 5 7)) (* (* 2 3) (+ 95 96)))
```

Leur évaluation donne respectivement 47, 6, 20, 26 et 1998.

Pour représenter une expression préfixée, on utilise ici un tableau dont chaque élément représente une entité de l'expression. Ainsi les expressions ci-dessus seront représentées par des tableaux de tailles respectives 1, 5, 5, 13, 29. Les éléments du tableau sont des objets à trois champs, le premier indique la nature de l'entité : (symbole ou nombre), le second champ est rempli par la valeur de l'entité dans le cas où celle ci est un nombre, enfin le dernier champ est un caractère rempli dans les cas où l'entité est un symbole
On utilise les fonctions données ci-dessus pour les piles et on y ajoute les procédures suivantes :

```
   calculer (char a, int x, int y) {
      if (a == '+') return x + y;
      if (a == '*') return x * y;
}
```

La procédure d'évaluation consiste à empiler les résultats intermédiaires, la pile contiendra des opérateurs et des nombres, mais jamais deux nombres consécutivement. On examine successivement les entités de l'expression si l'entité est un opérateur ou si c'est un nombre et que le sommet de la pile est un opérateur, alors on empile cette entité. En revanche, si c'est un nombre et qu'en sommet de pile il y a aussi un nombre, on fait agir l'opérateur qui précède le sommet de pile sur les deux nombres et on répète l'opération sur le résultat trouvé.

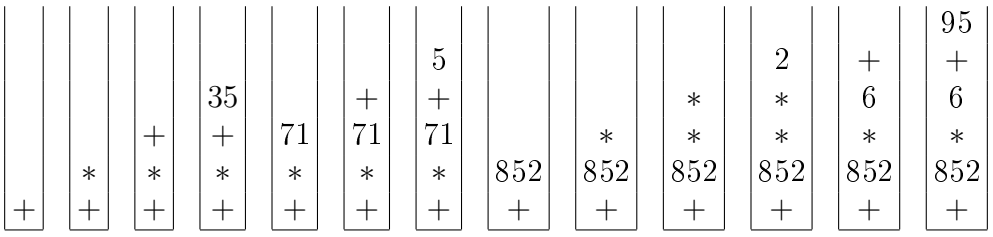


FIG. 1. Pile d'évaluation de l'expression dont le résultat est 1998

```
  inserer (x){
   while (!estVide() &&  estNombre(valeur())) {
      y = valeur();
      supprimer();
      op = valeur();
      supprimer();
      x = calculer (op, x, y);
  }
  ajouter(x);
}

  evaluer (u, n){
```

```
  for (int i = 0; i < n ; ++i) {
      if (!estNombre(u[i]) ) ajouter(u[i]);
      else  inserer (u[i]);
  }
  return valeur();
}
```

3. Files

Les files sont utilisées en programmation pour gérer des objets qui sont en attente d'un traitement ultérieur, par exemple des processus en attente d'une ressource du système, des sommets d'un graphe, des nombres entiers en cours d'examen de certaines de leur propriétés, etc ... Dans une file les éléments sont systématiquement ajoutés en queue et supprimés en tête, la valeur d'une file est par convention celle de l'élément de tête. En anglais, on parle de stratégie FIFO *First In First Out*, par opposition à la stratégie LIFO *Last In First Out* des piles. De façon plus formelle, on se donne un ensemble E, l'ensemble des files dont les éléments sont dans E est noté $Fil(E)$, la file vide (qui ne contient aucun élément) est F_0, les opérations sur les files sont *vide, ajouter, valeur, supprimer* :

- *estVide* est une application de $Fil(E)$ dans (*vrai, faux*), $estVide(F)$ est égal à *vrai* si et seulement si la file F est vide.
- *ajouter* est une application de $E \times Fil(E)$ dans $Fil(E)$, $ajouter(x, F)$ est la file obtenue à partir de la file F en insérant l'élément x à la fin de celle-ci.
- *valeur* est une application de $Fil(E) \setminus F_0$ dans E qui à une file F non vide associe l'élément se trouvant en tête.
- *supprimer* est une application de $Fil(E) \setminus F_0$ dans $Fil(E)$ qui associe à une file F non vide la file obtenue à partir de F en supprimant son premier élément.

Les opérations sur les files satisfont les relations suivantes

Pour $F \neq F_0$

$$supprimer(ajouter(x, F)) = ajouter(x, supprimer(F))$$
$$valeur(ajouter(x, F)) = valeur(F)$$

Pour toute file F

$$estVide(ajouter(x, F)) = faux$$

Pour la file F_0

$$supprimer(ajouter(x, F_0)) = F_0$$
$$valeur(ajouter(x, F_0)) = x$$
$$estVide(F_0) = vrai$$

Une première idée de réalisation sous forme de programmes des opérations sur les files est empruntée à une technique mise en oeuvre dans des lieux où des *clients* font la queue pour être *servis*, il s'agit par exemple de certains guichets de réservation dans les gares, de bureaux de certaines administrations, ou des étals de certains supermarchés. Chaque client qui se présente obtient un numéro et les clients sont ensuite appelés par les serveurs du guichet en fonction croissante de leur numéro d'arrivée. Pour gérer ce système deux nombres doivent être connus par les gestionnaires : le numéro

obtenu par le dernier client arrivé et le numéro du dernier client servi. On note ces deux nombres par *fin* et *début* respectivement et on gère le système de la façon suivante

- la file d'attente est vide si et seulement si $début = fin$,
- lorsqu'un nouveau client arrive on incrémente *fin* et on donne ce numéro au client,
- lorsque le serveur est libre et peut servir un autre client, si la file n'est pas vide, il incrémente *début* et appelle le possesseur de ce numéro.

```
    debut = 0; fin = 0;
    MaxF = ...

    estVide() {
    return  (debut == fin);
  }

    estPleine() {
    return  (fin == MaxF);
  }

  valeur () {
    if (estVide())
        affiche ("File Vide.");
    else return  contenu[debut];
  }


  ajouter (x) {
    if (estPleine())
        affiche ("File Pleine.");
    else{
       contenu[fin] = x;
       fin++;
    }
  }

  supprimer () {
    if (estVide())
        affiche ("File Vide.");
    else
        debut++;
  }
}
```

On peut aussi représenter toutes ces opérations optimisant la place prise par la file en utilisant la technique suivante : on réattribue le numéro 0 à un nouveau client lorsque l'on atteint un certain seuil pour la valeur de *fin*. On dit qu'on a un tableau (ou tampon) circulaire.

```
    debut = 0; fin = 0;
```

```
    pleine = false; vide = true;
    MaxF = ...

    estVide() {
    return  vide;
  }

    estPleine() {
    return  pleine;
  }

  valeur (File f) {
    if (estVide())
        affiche ("File Vide.");
    return  contenu[debut];
  }

   Successeur(int i) {
    return  (i+1) % MaxF;
  }

  ajouter (x) {
    if (estPleine())
        affiche ("File Pleine.");
    else{
       contenu[fin] = x;
       fin = Successeur(fin);
       vide = false;
       pleine = (fin == f.debut);
    }
  }

  supprimer () {
    if (estVide())
        affiche ("File Vide.");
    else {
       debut =  Successeur(debut);
       vide = (fin == debut);
      pleine = false;
  }
}
```

CHAPITRE 4

Arbres

1. Définition, Exemples

Nous avons déjà vu la notion de fonction récursive dans le chapitre 2. Considérons à présent son équivalent dans les structures de données : la notion d'arbre. Un arbre est soit un arbre atomique (une *feuille*), soit un *noeud* et une suite de sous-arbres. Graphiquement, un arbre est représenté comme suit

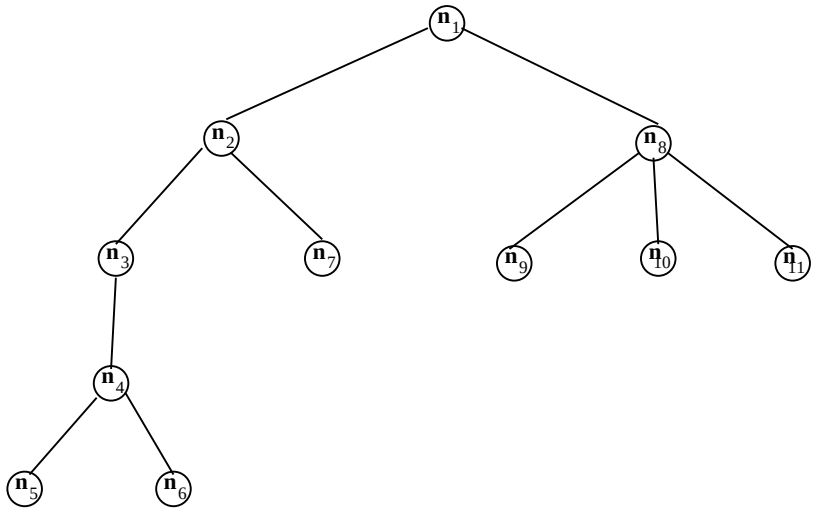


FIG. 1. Un exemple d'arbre

Le noeud n_1 est la *racine* de l'arbre, n_5, n_6, n_7, n_9, n_{10}, n_{11} sont les *feuilles*, n_1, n_2, n_3, n_4, n_8 les *noeuds internes*. Plus généralement, l'ensemble des *noeuds* est constitué des noeuds internes et des feuilles. Contrairement à la botanique, on dessine les arbres avec la racine en haut et les feuilles vers le bas en informatique. Il y a bien des définitions plus mathématiques des arbres, que nous éviterons ici. Si une branche relie un noeud n_i à un noeud n_j plus bas, on dira que n_i est un *ancêtre* de n_j. Une propriété fondamentale d'un arbre est qu'un noeud n'a qu'un seul père. Enfin, un noeud peut contenir une ou plusieurs valeurs, et on parlera alors d'*arbres étiquetés* et de la valeur (ou des valeurs) d'un noeud. Les *arbres binaires* sont des arbres tels que les noeuds ont au plus 2 fils. La *hauteur*, on dit aussi la *profondeur* d'un noeud est la longueur du chemin qui le joint à la racine, ainsi la racine est elle même de hauteur 0, ses fils de hauteur 1 et les autres noeuds de hauteur supérieure à 1.

Un exemple d'arbre très utilisé en informatique est la représentation des expressions arithmétiques et plus généralement des termes dans la programmation symbolique. Pour montrer l'aspect fondamental de la structure d'arbre, on peut tout de suite voir que les expressions arithmétiques calculées dans la section 2 se représentent simplement par des arbres comme dans la figure **??** pour `(* (+ 5 (* 2 3)) (+ (* 10 10) (* 9 9)))`. Cette représentation contient l'essence de la structure d'expression arithmétique et fait donc abstraction de toute notation préfixée ou postfixée.

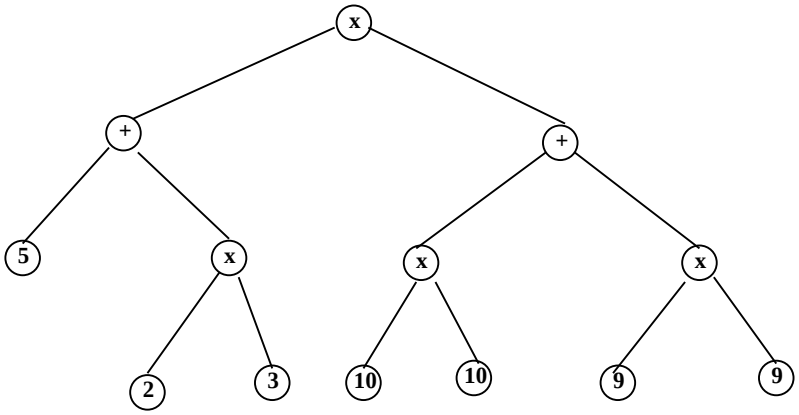


FIG. 2. Arbre d'une expression arithmétique

2. Tas

Un autre exemple de structure arborescente est la structure de tas (*heap*) utilisée pour représenter des files de priorité. Donnons d'abord une vision intuitive d'une file de priorité.

On suppose, comme au paragraphe 3, que des gens se présentent au guichet d'une banque avec un numéro écrit sur un bout de papier représentant leur degré de priorité. Plus ce nombre est petit, plus ils sont importants et doivent passer rapidement. Bien sûr, il n'y a qu'un seul guichet ouvert, et l'employé(e) de la banque doit traiter rapidement tous ses clients pour que tout le monde garde le sourire. La file des personnes en attente s'appelle une *file de priorité*. L'employé de banque doit donc savoir faire rapidement les 3 opérations suivantes : trouver un minimum dans la file de priorité, retirer cet élément de la file, savoir ajouter un nouvel élément à la file. Plusieurs solutions sont envisageables.

La première consiste à mettre la file dans un tableau et à trier la file de priorité dans l'ordre décroissant des priorités. Trouver un minimum et le retirer de la file est alors simple : il suffit de prendre l'élément le plus à droite, et de déplacer vers la gauche la borne droite de la file. Mais l'insertion consiste à faire une passe du tri par insertion pour mettre le nouvel élément à sa place, ce qui peut prendre un temps $O(n)$ où n est la longueur de la file.

Une autre méthode consiste à gérer la file comme une simple file du chapitre précédent, et à rechercher le minimum à chaque fois. L'insertion est rapide, mais la recherche du miniimum peut prendre un temps $O(n)$, de même que la suppression.

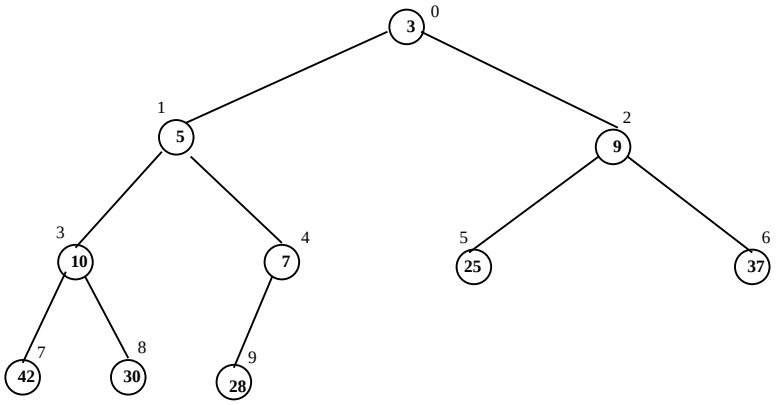


FIG. 3. Représentation en arbre d'un tas

i	0	1	2	3	4	5	6	7	8	9
$a[i]$	3	5	9	10	7	25	37	42	30	28

FIG. 4. Représentation en tableau d'un tas

Une méthode élégante consiste à gérer une structure d'ordre partiel grâce à un arbre. La file de n éléments est représentée par un arbre dont chaque noeud a deux fils ; chaque noeud contenant un élément de la file (comme illustré dans la figure 3). L'arbre vérifie deux propriétés importantes : d'une part la valeur de chaque noeud est inférieure ou égale à celle de ses fils, d'autre part l'arbre est quasi complet, propriété que nous allons décrire brièvement. Si l'on divise l'ensemble des noeuds en lignes suivant leur hauteur, on obtient en général dans un arbre binaire une ligne 0 composée simplement de la racine, puis une ligne 1 contenant au plus deux noeuds, et ainsi de suite (la ligne i contenant au plus 2^i noeuds). Dans un arbre quasi complet les lignes, exceptée peut être la dernière, contiennent toutes un nombre maximal de noeuds (soit 2^i). De plus les feuilles de la dernière ligne se trouvent toutes à gauche, ainsi tous les noeuds internes sont binaires, excepté le plus à droite de l'avant dernière ligne qui peut ne pas avoir de fils droit. Les feuilles sont toutes sur la dernière et éventuellement l'avant dernière ligne.

On peut numéroter cet arbre en largeur d'abord, c'est à dire dans l'ordre donné par les petits numéros figurant au dessus de la figure 3. Dans cette numérotation on vérifie que tout noeud i a son père en position $\lfloor (i-1)/2 \rfloor$, le fils gauche du noeud i est $2i+1$, le fils droit $2i++2$. Formellement, on peut dire qu'un tas est un tableau a contenant n entiers (ou des éléments d'un ensemble totalement ordonné) satisfaisant les conditions :

$$\begin{aligned} 2 \leq 2i \leq n \quad &\Rightarrow \quad a[2i] \geq a[i] \\ 3 \leq 2i+1 \leq n \quad &\Rightarrow \quad a[2i+1] \geq a[i] \end{aligned}$$

Ceci permet d'implémenter cet arbre dans un tableau `a` (voir figure 4) où le numéro de chaque noeud donne l'indice de l'élément du tableau contenant sa valeur.

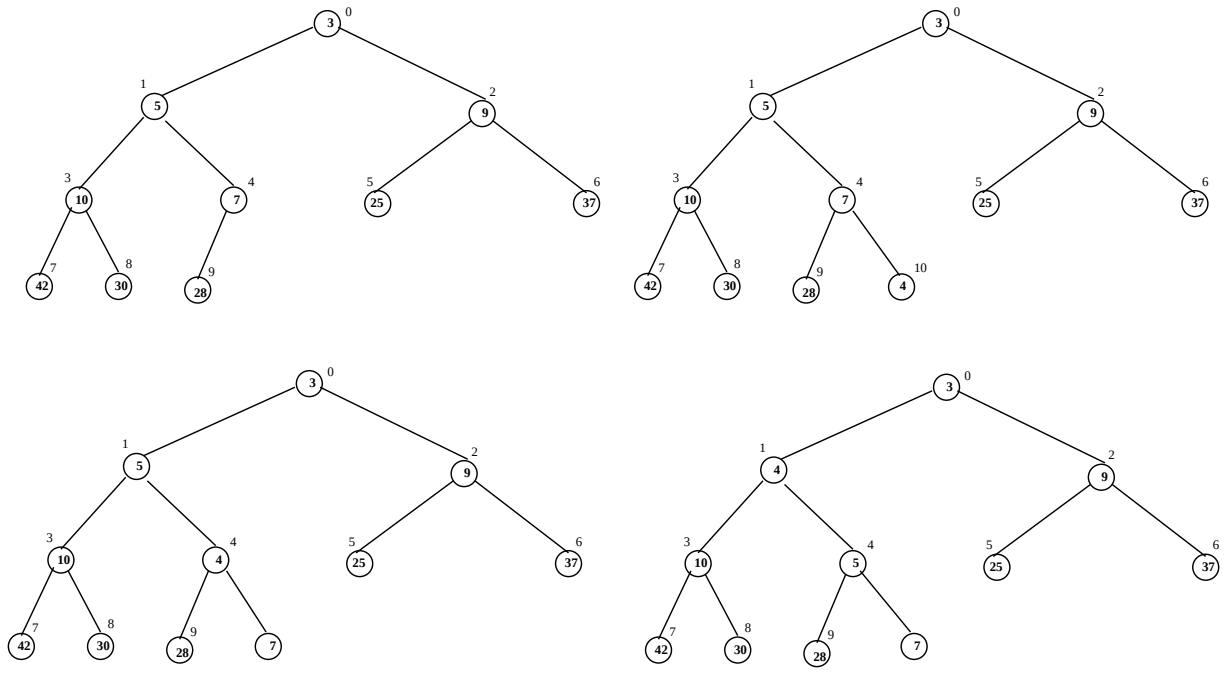


FIG. 5. Ajout dans un tas

L'ajout d'un nouvel élément v à la file consiste à incrémenter n puis à poser $a[n] = v$. Ceci ne représente plus un tas car la relation $a[(n-1)/2] \geq v$ n'est pas nécessairement satisfaite. Pour obtenir un tas, il faut échanger la valeur contenue au noeud n et celle contenue par son père, remonter au père et réitérer jusqu'à ce que la condition des tas soit vérifiée. Ceci se programme par une simple itération (cf. la figure 5).

```
ajouter (v)  {

  ++nTas;
  i = nTas - 1;
  while (i > 0 && a [(i - 1)/2] > v) {
      a[i] = a[(i - 1)/2];
      i = (i - 1)/2;
  }
  a[i] = v;
}
```

On vérifie que, si le tas a n éléments, le nombre d'opérations n'excédera pas la hauteur de l'arbre correspondant. Or la hauteur d'un arbre binaire complet de n noeuds est $\log n$. Donc `Ajouter` ne prend pas plus de $O(\log n)$ opérations.

On peut remarquer que l'opération de recherche du minimum est maintenant immédiate dans les tas. Elle prend un temps constant $O(1)$.

```
static int maximum ()  {
```

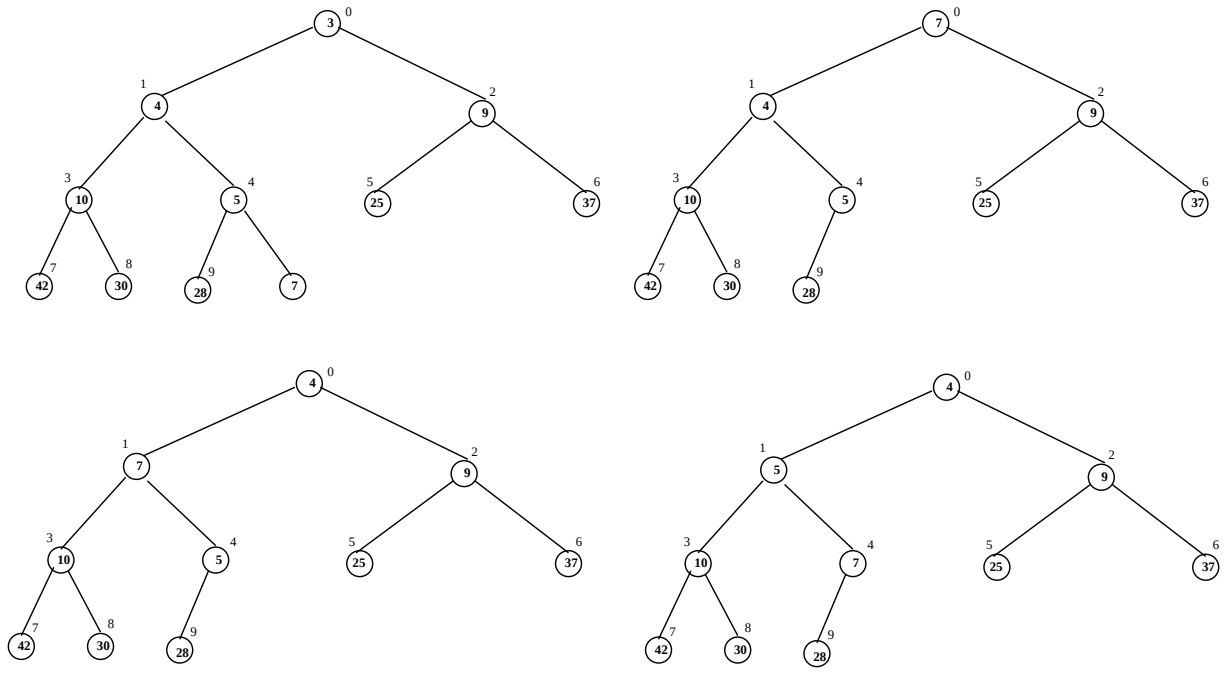


FIG. 6. Suppression dans un tas

```
  return a[0];
}
```

Considérons l'opération de suppression du premier élément de la file. Il faut alors retirer la racine de l'arbre représentant la file, ce qui donne deux arbres ! Le plus simple pour reformer un seul arbre est d'appliquer l'algorithme suivant : on met l'élément le plus à droite de la dernière ligne à la place de la racine, on compare sa valeur avec celle de ses fils, on échange cette valeur avec celle du vainqueur de ce tournoi, et on réitère cette opération jusqu'à ce que la condition des tas soit vérifiée. Bien sûr, il faut faire attention, quand un noeud n'a qu'un fils, et ne faire alors qu'un petit tournoi à deux. Le placement de la racine en bonne position est illustré dans la figure 6.
A nouveau, la suppression du premier élément de la file ne prend pas un temps supérieur à la hauteur de l'arbre représentant la file. Donc, pour une file de n éléments, la suppression prend $O(\log n)$ opérations. La représentation des files de priorités par des tas permet donc de faire les trois opérations demandées : ajout, retrait, chercher le plus grand en $\log n$ opérations. Ces opérations sur les tas permettent de faire le tri *HeapSort*. Ce tri peut être considéré comme alambiqué, mais il a la bonne propriété d'être toujours en temps $n \log n$ (comme le Tri fusion, cf page **??**).

HeapSort se divise en deux phases, la première consiste à construire un tas dans le tableau à trier, la seconde à répéter l'opération de prendre l'élément maximal, le retirer du tas en le mettant à droite du tableau. Il reste à comprendre comment on peut construire un tas à partir d' un tableau quelconque. Il y a une méthode peu efficace, mais systématique. On remarque d'abord que l'élément de gauche du tableau est à lui seul un tas. Puis on

ajoute à ce tas le deuxième élément avec la procédure `Ajouter` que nous venons de voir, puis le troisième, ... A la fin, on obtient bien un tas de N éléments dans le tableau `a` à trier. Le verbatimme est

```
heapSort ()  {

  nTas = 0;
  for (i = 0; i < N; ++i)
      ajouter (a[i]);
  for (i = N - 1; i >= 0; --i) {
      v = maximum();
      supprimer();
      a[i] = v;
   }
}
```

Si on fait un décompte grossier des opérations, on remarque qu'on ne fait pas plus de $N \log N$ opérations pour construire le tas, puisqu'il y a N appels à la procédure `Ajouter`. Une méthode plus efficace, que nous ne décrirons pas ici, qui peut être traitée à titre d'exercice, permet de construire le tas en $O(N)$ opérations. De même, dans la deuxième phase, on ne fait pas plus de $N \log N$ opérations pour défaire les tas, puisqu'on appelle N fois la procédure `Supprimer`. Au total, on fait $O(N \log N)$ opérations quelle que soit la distribution initiale du tableau `a`, comme dans le tri fusion. On peut néanmoins remarquer que la constante qui se trouve devant $N \log N$ est grande, car on appelle des procédures relativement complexes pour faire et défaire les tas. Ce tri a donc un intérêt théorique, mais il est en pratique bien moins bon que Quicksort.

3. Borne inférieure sur le tri

Il a été beaucoup question du tri. On peut se demander s'il est possible de trier un tableau de N éléments en moins de $N \log N$ opérations. Un résultat ancien de la théorie de l'information montre que c'est impossible si on n'utilise que des comparaisons.

En effet, il faut préciser le modèle de calcul que l'on considère. On peut représenter tous les tris que nous avons rencontrés par des arbres de décision. La figure 7 représente un tel arbre pour le tri par insertion sur un tableau de 3 éléments. Chaque noeud interne pose une question sur la comparaison entre 2 éléments. Le fils de gauche correspond à la réponse négative, le fils droit à l'affirmatif. Les feuilles représentent la permutation à effectuer pour obtenir le tableau trié.

THÉORÈME 4.1. *Le tri de N éléments, fondé uniquement sur les comparaisons des éléments deux à deux, fait au moins $O(N \log N)$ comparaisons.*

Démonstration. Tout arbre de décision pour trier N éléments a $N!$ feuilles représentant toutes les permutations possibles. Un arbre binaire de $N!$ feuilles a une hauteur de l'ordre de $\log N! \simeq N \log N$ par la formule de Stirling. □

COROLLAIRE 4.1. HeapSort *et le tri fusion sont optimaux (asymptotiquement).*

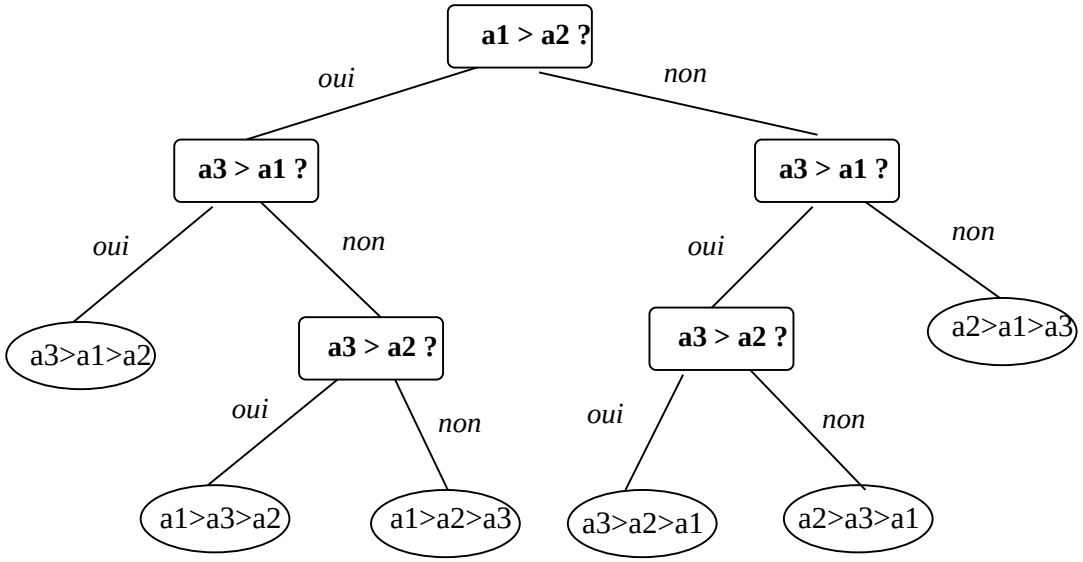


FIG. 7. Exemple d'arbre de décision pour le tri

En effet, ils accomplissent le nombre de comparaisons donné comme borne inférieure dans le théorème précédent. Mais, nous répétons qu'un tri comme Quicksort est aussi très bon en moyenne. Le modèle de calcul par comparaisons donne une borne inférieure, mais peut-on faire mieux dans un autre modèle ? La réponse est oui, si on dispose d'informations annexes comme les valeurs possibles des éléments a_i à trier. Par exemple, si les valeurs sont comprises dans l'intervalle $[1, k]$, on peut alors prendre un tableau b annexe de k éléments qui contiendra en b_j le nombre de a_i ayant la valeur j. En une passe sur a, on peut remplir le tableau k, puis générer le tableau a trié en une deuxième passe en ne tenant compte que de l'information rangée dans b. Ce tri prend $O(k + 2N)$ opérations, ce qui est très bon si k est petit.

CHAPITRE 5

Algorithmes gloutons

1. Principe de l'algorithme

De nombreux problèmes d'optimisation combinatoire se présentent sous la forme suivante :

> On se donne un ensemble E fini et à chaque élément e de E est affectée une valeur $v(e)$ qui est un réel positif ; on se donne de plus une condition C sur l'ensemble des parties de E. Le problème consiste à construire un sous-ensemble F de E tel que :
>
> $\sum_{e \in F} v(e)$ est maximal parmi les parties F satisfaisant C.

Dans plusieurs cas la condition C est fermée par sous-ensemble : si F satisfait C alors il en est de même pour tout sous-ensemble de F. Il est alors naturel de proposer un algorithme très simple consistant à initialiser F par $F = \emptyset$, puis à ajouter successivement des éléments à F suivant un certain critère (par exemple par valeurs décroissantes), jusqu'à obtenir un ensemble auquel on ne peut plus ajouter d'élément sans contredire C. On appelle *algorithme glouton* ce type d'algorithme, ce qualificatif décrit le fait qu'on se précipite sur l'élément le plus intéressant en premier sans se préoccuper des conséquences ultérieures de ce choix.

Cet algorithme donne très rapidement un résultat, en revanche ce résultat n'est pas toujours la solution optimale. Nous allons voir que ce n'est le cas que pour des familles de parties satisfaisant une condition particulière : on appelle ces familles *matroïdes*. Plusieurs ouvrages ont été consacrés, nous recommandons en particulier celui de D. Welsh [**?**] ; une bonne introduction au sujet est aussi donnée par le chapitre 17 du manuel [**2**].

Les questions relatives à l'affectation d'une ou plusieurs ressources à des utilisateurs (clients, processeurs, etc.) constituent une classe importante de problèmes qu'il est tentant de résoudre par un algorithme glouton. Nous allons montrer que dans certains cas très simples, l'algorithme glouton donne une solution optimale. Dans des cas plus complexes, l'algorithme donne une solution approchée, dont on se contente parfois, vu le temps de calcul prohibitif de la recherche de l'optimum exact. Après avoir traité ce cas simple nous considérons un problème classique sur les graphes : la recherche d'un arbre recouvrant de poids maximal (ou de manière équivalente minimal). La résolution de celui-ci par l'algorithme glouton amène à la structure de matroïde dont on donne quelques propriétés.

2. Exemples où l'algorithme glouton ne donne pas la bonne réponse

(1) **Sac a dos :** On a des objets $1, 2, 3, \ldots, n$ de poids $p_1, p_2, p_3, \ldots, p_n$ et on veut en mettre dans un sac de façon telle que le poids du sac soit inférieur ou égal à P. Il s'agit de trouver le poids maximum que l'on peut mettre dans le sac.

$$\mathcal{F} \quad = \{F \mid p(F) \; = \; \sum_{i \in F} p_i \; \leq \; P\}, \quad \text{et} \quad \forall i, \; 1 \leq i \leq n \;\; v(i) = p_i$$

(2) **Formation d'une liste de candidats :** On se propose de constituer une liste de personnes aussi large que possible à prendre parmi $x_1, x_2, \ldots, x_n$, en tenant compte d'incompabilités correspondant à des couples qui refusent energiquement de se retrouver ensemble sur la liste.

$$\mathcal{F} \quad = \{F \mid \forall x, y \in F, \; (x, y) \notin U\}, \quad \text{et} \quad \forall x \quad v(x) = 1$$

3. Un exemple où il donne la solution optimale

Le problème décrit précisément ci-dessous peut être résolu par l'algorithme glouton (mais, comme on le verra, l'algorithme glouton ne donne pas la solution optimale pour d'autres formulations du problème, pourtant proches de celle-ci). Il s'agit d'affecter une ressource unique, non partageable, successivement à un certain nombre d'utilisateurs qui en font la demande en précisant la période exacte pendant laquelle ils souhaitent en disposer.

On peut illustrer ceci en considérant la question de la location d'une voiture. Des clients formulent un ensemble de demandes de location et, pour chaque demande sont donnés le jour du début de la location et le jour de restitution du véhicule ; le but est d'affecter le véhicule de façon à satisfaire le maximum de clients (et non pas de maximiser la somme des durées de location). On vérifie que ce problème rentre dans le cadre général considéré plus haut. L'ensemble E est celui des demandes de location, pour chaque élément e de E, on note $d(e)$ la date du début de la location et $f(e) > d(e)$ la date de fin. La valeur $v(e)$ de tout élément e de E est égale à 1 et la contrainte à respecter pour le sous-ensemble F à construire est la suivante, (cette contrainte exprime que deux clients ne peuvent disposer en même temps du véhicule) :

$$\forall e_1, e_2, \quad e_1 \in F \text{ et } e_2 \in F \Rightarrow [d(e_1), f(e_1)] \cap [d(e_2), f(e_2)] = \emptyset.$$

L'algorithme glouton est décrit comme suit :

- *Etape 1 :*

  Classer les éléments de E par ordre de dates de fin croissantes. Les éléments de E forment alors une suite $e_1, e_2, \ldots, e_n$ telle que $f(e_1) \leq f(e_2) \leq \ldots \leq f(e_n)$
- *Etape 2 :*

  Initialiser $F := \emptyset$

  Pour i variant de 1 à n, ajouter la demande e_i à F si celle-ci ne chevauche pas la dernière demande appartenant à F.

THÉORÈME 5.1. *L'algorithme ci-dessus donne une solution optimale.*

Soit $F = \{x_1, x_2, \ldots, x_p\}$ la solution obtenue par l'algorithme glouton et soit $G = \{y_1, y_2, \ldots, y_q\}, q \geq p$ une solution optimale telle que le cardinal de $F \cap G$ soit maximum. Dans G et dans F nous supposons que les x_i et y_j sont classées par dates de fin croissantes. Nous allons montrer que $p = q$. Soit k le plus petit entier tel que $x_k \neq y_k$; ainsi $\forall i < k$, on a $x_i = y_i$. Par construction de F on a alors $f(y_k) \geq f(x_k)$. On peut alors remplacer G par $G' = \{y_1, y_2, \ldots y_{k-1}, x_k, y_{k+1}, y_q\}$ tout en satisfaisant à la contrainte de non chevauchement des demandes, ainsi G' est une solution optimale, de même cardinalité que G, ayant plus d'éléments en commun avec F que n'en avait G. En répétant cette opération suffisamment de fois on trouve un ensemble H de même cardinalité que G et qui contient F. L'ensemble H ne peut contenir plus d' éléments que F car ceux-ci auraient été ajoutés à F par l'algorithme glouton, ceci montre bien que $p = q$.

Remarques

1. Noter que le choix de classer les demandes par dates de fin croissantes est important. Si on les avait classées, par exemple, par dates de début croissantes, on n'aurait pas obtenu le résultat. On le voit sur l'exemple suivant avec trois demandes e_1, e_2, e_3 dont les dates de début et de fin sont données par le tableau suivant :

	e_1	e_2	e_3
d	2	3	5
f	8	4	8

Bien entendu, pour des raisons évidentes de symétrie, le classement par dates de début décroissantes donne aussi un résultat optimal.

2. On peut noter aussi que si le but est de maximiser la durée totale de location du véhicule l'algorithme décrit ci-dessus ne donne pas l'optimum. En particulier, il ne considérera pas comme prioritaire une demande de location de durée très importante. Une idée consiste à le modifier en classant les demandes par durées décroissantes et d'appliquer l'algorithme glouton, malheureusement cette technique ne donne pas non plus le bon résultat (il suffit de considérer une demande de location de 3 jours et deux demandes qui ne se chevauchent pas mais qui sont incompatibles avec la première chacune de durée égale à 2 jours).

3. S'il y a plus d'une ressource à affecter, par exemple deux voitures à louer, l'algorithme glouton consistant à classer les demandes suivant les dates de fin et à affecter la première ressource disponible, ne donne pas l'optimum, comme le montre l'exemple suivant :

	e_1	e_2	e_3	e_4
d	1	2	6	4
f	3	5	7	8

En effet les deux premières demandes sont satisfaites par des véhicules différents si on attribue le premier véhicule à la troisième demande, il n'est plus possible de satisfaire la quatrième.

3.1. Réalisation de taches sur un processeur séquentiel. Voici un exemple de matroïde issu d'une situation pratique. Des tâches $T_1, T_2, \ldots, T_k$, sont à réaliser sur un

processeur séquentiel, la réalisation de chacune de celles-ci prend une unité de temps et on donne pour chaque T_i un délai d_i à respecter (en supposant que l'on commence à l'instant 0, la tâche T_i doit être réalisée avant l'instant d_i). De plus une pénalité p_i est fixée pour le cas où la tache T_i ne serait pas réalisée dans les délais impartis (cette pénalité est fixe et ne dépend pas de l'instant de la réalisation). Il s'agit de déterminer l'ordre sur les tâches qui minimise la somme des pénalités. L'exemple suivant illustre le problème : il y a 7 tâches $T_1, \ldots, T_7$ dont les délais et pénalités sont données par le tableau suivant :

	T_1	T_2	T_3	T_4	T_5	T_6	T_7
d	4	3	2	2	6	3	5
p	70	80	90	50	30	40	20

Si on effectue les tâches dans l'ordre donné par T_1, T_2, T_3, T_4, T_5, T_6, T_7 les tâches T_3, T_4, T_6, T_7 ne sont pas effectuées dans les délais et la pénalité est de $90+50+40+20 = 200$. Pour l'ordre : T_3, T_4, T_2, T_5, T_7, T_1, T_6 seules les tâches T_1 et T_6 sont hors délais et la pénalité est 110, noter que l'optimum est atteint pour T_4, T_3, T_2, T_1, T_7, T_5, T_6 avec une pénalité de 40.

On considère la famille $\mathcal{F}$ de sous-ensembles F de $T = \{T_1, T_2, \ldots, T_k\}$ donnée par $F \in \mathcal{F}$ si les tâches contenues dans F peuvent être toutes réalisées dans les délais. On montre que cette famille satisfait les conditions sur les matroïdes. (Laissé en exercice, considérer les nombres $U(i, F)$ de tâches de délai inférieur ou égal à i). Ainsi l'algorithme glouton consistant à ajouter les tâches par ordre de pénalités décroissantes donne la solution optimale au problème posé.

CHAPITRE 6

Programmation dynamique

La m'ethode se rencontre dans des problèmes de nature différente on l'appelle pour des raisons historiques *programmation dynamique*, le terme programmation est employé ici dans le sens de méthode et non pas dans celui de réalisation d'un programme informatique.

La recherche de plus courts chemins peut être vue aussi comme la résolution d'un système d'équations linéaires dans un anneau particulier pour lequel cette résolution se limite au calcul de la puissance d'une matrice. Une introduction à cette théorie qui est largement développée par ailleurs est donnée.

La méthode de la programmation dynamique est expliquée dans ses grandes lignes au paragraphe suivant, elle est appliquée ensuite pour résoudre plusieurs problèmes d'optimisation dont l'un intervient dans le cadre de l'analyse du génome.

1. Présentation de la méthode

Les problèmes que l'on peut traiter à l'aide de cette méthode ont deux caractéristiques principales

- Une solution optimale à un problème de taille k contient dans un sens à préciser des solutions optimales à des problèmes de tailles plus petites que k.
- On peut construire une solution optimale à un problème de taille k en connaissant un petit nombre de solutions optimales à des problèmes de tailles plus petites que k.

Une méthode de résolution de problème par la programmation dynamique consiste à résoudre itérativement des problèmes de tailles de plus en plus grandes en conservant en mémoire les résultats obtenus pour les problèmes de petites tailles. En général une mémoire de l'ordre de n^2 pour un problème de taille n est nécessaire.

2. Un problème générique

L'exemple qui suit est un exemple typique de tel problème, on l'exprime souvent sous la forme suivante :

Un système est caractérisé par son *état* qui est un élément σ d'un ensemble fini Σ, on considère des suites $\sigma_0, \sigma_1, \ldots, \sigma_k$ d'états successifs du système, σ_i est l'état au temps $i = 1, \ldots, k$. Le système est commandé par des *décisions* à prendre représentées par des entiers $x_1, x_2, \ldots, x_k$: x_i est la décision prise au temps i. Le comportement du système est régi par k fonctions de transitions ϕ_i, une pour chaque valeur de i. La fonction de transition ϕ_i a pour donnée un état σ et une décision x, elle a pour résultat un état ; $\phi_i(\sigma, x)$ est l'état atteint au temps i si l'état au temps $i-1$ est σ et si la décision prise au temps i est x.

Une fonction de coût c_i pour chaque i compris entre 1 et k permet de mesurer les coûts des décisions prises. La fonction de coût a les mêmes données que la fonction de transition et a pour résultat un entier $c_i(\sigma, x)$ qui est le coût de la décision x au temps i si l'état du système est σ au temps $i-1$.

Le problème consiste à trouver une suite de transitions entre deux états donnés l'état initial et l'état final qui minimise la somme des coûts. Quelquefois l'état final n'est pas précisé, il faut alors obtenir celui qui donne le minimum.

Une autre version du problème consiste à considérer non pas des coûts c_i mais des gains g_i ; il faut alors rendre le gain maximum plutôt que de chercher un coût minimal.

Dans l'exemple de la recherche du chemin minimal la décision à prendre consiste à choisir un des arcs ou à rester sur place, l'état du système est le sommet où l'on se trouve dans le graphe. Les transitions possibles sont celles où l'on a choisi un arc ayant pour origine le sommet où l'on se trouve, le résultat de la transition consiste à passer au sommet extrémité de l'arc choisi, et le coût de la transition est égal à la longueur de l'arc.

2.1. Algorithme. L'algorithme de programmation dynamique s'exprime de la façon suivante : pour chaque couple d'états σ, τ on note $\delta_k(\sigma, \tau)$ le coût minimal d'une suite de transitions conduisant de σ à l'instant 0 à τ à l'instant k ; ce coût est infini si une telle suite n'existe pas. On a alors la formule de récurrence suivante qui permet de résoudre le problème de manière itérative :

$$\delta_k(\sigma, \tau) = \min_{\theta \in \Sigma} [\delta_{k-1}(\sigma, \theta) + c_k(\theta, x_k)]$$

Dans cette formule x_k satisfait $\phi_k(\theta, x_k) = \tau$.

On remarque qu'il suffit alors de conserver les valeurs de δ_{k-1} pour obtenir celles de δ_k, il n'est pas nécessaire de connaître les valeurs de δ_i pour $i < k-1$, la mémoire nécessaire est bien en $O(n^2)$ où n est le nombre d'états du système.

2.2. Le sac à dos. Le problème du sac à dos est connu comme étant NP-complet en général, toutefois dans le cas particulier où les poids des objets à emporter sont des entiers on peut obtenir un algorithme polynomial en fonction de la capacité totale du sac. Précisons le problème : on se donne des entiers $p_1, p_2, \ldots, p_k$ représentant les poids de certains objets et un nombre P représentant le poids maximal des objets que l'on peut emporter dans le sac ; pour chaque objet i ($i = 1, \ldots, k$), a_i est l'indice de satisfaction obtenu si on emporte i dans le sac. Le problème consiste à trouver le sous-ensemble d'objets qui maximise la somme des indices de satisfaction et tel que la somme des poids des objets qui le composent est inférieure ou égale à P.

Avec la terminologie du paragraphe précédent on considère des gains et non pas des coûts. L'état du système est la capacité qui reste dans le sac à dos, ainsi l'état initial est P, lorsque l'on prend l'objet i on passe d'une capacité p à la capacité $p - p_i$. Les différentes décisions sont représentées par une variable qui prend les valeurs 0 et 1, $x_i = 1$ si on met l'objet i dans le sac $x_i = 0$ sinon. Ainsi $\phi_i(p, x) = p - xp_i$ et $g_i(p, x) = a_i$. L'algorithme de

programmation dynamique consiste à gérer un tableau `benefice[p]` de taille P qui donne à chaque étape i le bénéfice maximum que l'on peut réaliser en prenant au plus i objets et en laissant une capacité résiduelle de p dans le sac. Les poids de chaque objet i sont donnés dans un tableau `poids[i]` et les nombres a_i dans un tableau `a[i]`.

```
Pour p = 1 jusqu'a  P faire benefice[p] = 0;
Pour i = 1 jusqu'a k faire {
         q = poids[i];
         Pour p = q jusqu'a P faire
            Si (benefice[p-q] < benefice[p] + a[i])
                 benefice[p-q] = benefice[p] + a[i];
        }
```

3. Traitement de séquences

La comparaison de deux séquences afin de déterminer leur similarité joue un rôle central en bio-informatique. On utilise souvent la programmation dynamique pour résoudre ces problèmes qui proviennent de l'analyse du génome humain et de la compréhension de sa structure primaire.

3.1. Sous-séquence communes. Un exemple simple consiste à déterminer la plus longue sous-séquence commune à deux séquences données. Précisons tout d'abord quelques définitions. Une *séquence* (ou un *mot*) est une suite finie de symboles (ou *lettres*) pris dans un ensemble fini (ou *alphabet*). Si $u = a_1 \cdots a_n$ est une séquence, où $a_1, \ldots, a_n$ sont des lettres, l'entier n est la *longueur* de u. Une séquence $v = b_1 \cdots b_m$ est une *sous-séquence* de $u = a_1 \cdots a_n$ s'il existe des entiers $i_1, \ldots, i_m$, $(1 \leq i_1 < \cdots i_m \leq n)$ tels que $a_{i_k} = b_k$ $(1 \leq k \leq m)$. Une séquence w est une *sous-séquence commune* aux séquences u et v si w est sous-séquence de u et de v. Une sous-séquence commune est dite *maximale* si sa longueur est maximale parmi toutes les sous-séquences communes.

On cherche d'abord à déterminer la longueur d'une sous-séquence commune maximale à $u = a_1 \cdots a_n$ et $v = b_1 \cdots b_m$. Pour cela, on note $L(i, j)$ la longueur d'une sous-séquence commune maximale aux mots $a_1 \cdots a_i$ et $b_1 \cdots b_j$, $(0 \leq j \leq m, 0 \leq i \leq n)$. On peut montrer que :

$$L(i,j) = \begin{cases} 1 + L(i-1, j-1) & \text{si } a_i = b_j \\ \max(L(i, j-1), L(i-1, j)) & \text{sinon.} \end{cases}$$

En effet, soit w une sous séquence de longueur maximale, commune à $a_1 \cdots a_{i-1}$ et à $b_1 \cdots b_{j-1}$ si $a_i = b_j$, wa_i est une sous-séquence commune maximale à $a_1 \cdots a_i$ et $b_1 \cdots b_j$. Si $a_i \neq b_j$ alors une sous-séquence commune à $a_1 \cdots a_i$ et $b_1 \cdots b_j$ est ou bien commune à $a_1 \cdots a_i$ et $b_1 \cdots b_{j-1}$ (si elle ne se termine pas par b_j) ; ou bien à $a_1 \cdots a_{i-1}$ et $b_1 \cdots b_j$, (si elle ne se termine par a_i). On obtient ainsi l'algorithme qui permet de déterminer la longueur d'une sous séquence commune maximale à $a_1 \cdots a_n$ et $b_1 \cdots b_m$

Il est assez facile de transformer l'algorithme pour retrouver une sous-séquence maximale commune au lieu de simplement calculer sa longueur. Pour cela, on met à jour un tableau `provient` qui indique lequel des trois cas a permis d'obtenir la longueur maximale.

Une fois ce calcul effectué il suffit de remonter à chaque étape de `i,j` vers `i-1, j-1`, vers `i, j-1` ou vers `i-1,j` en se servant de la valeur de `provient[i][j]`. On peut améliorer cette méthode et rendre la complexité plus faible grâce à des techniques astucieuses de programmation, c'est ce qui est fait pour la commande `diff` du système UNIX.

3.2. Alignement de séquences. Soit u et v deux mots écrits à l'aide de l'alphabet $\mathbb{A}$ (on prend souvent $\mathbb{A} = \{G, T, A, C\}$ en bioinformatique), on appelle alignement de u, v un couple de mots u', v' sur l'alphabet $\mathbb{A} \cup \{e\}$ (où $e \notin \mathbb{A}$ représente le caractère d'espacement), couple qui satisfait les conditions suivantes

– les deux mots u' et v' ont même longueur ($|u'| = |v'|$) ;
– la suppression de e dans u' et dans v' donne u et v respectivement ;
– les lettres e ne se retrouvent pas en même position dans u' et v'.

Un exemple d'alignement de $u = CGATTAG$ et $v = GATCGA$ est

$$\begin{array}{rcl} u' & = & C\ G\ A\ T\ T\ e\ A\ G \\ v' & = & e\ G\ A\ T\ G\ G\ A\ e \end{array}$$

Il y a bien entendu de très nombreux alignements possibles pour deux séquences, en fait on cherche ceux qui sont les plus intéressants dans le sens précis que l'on décrit ici :

On suppose donnée une fonction de similarité p entre les couples de lettres. La valeur de $p(a, b)$ est un nombre reél d'autant plus grand que les lettres sont considérées comme similaires d'un point de vue biologique, ainsi on prendra des valeurs négatives quand les lettres sont différentes, soit : $b \neq a$, $p(a, a) > 0 > p(a, b)$, de plus la symétrie impose $p(a, b) = p(b, a)$. D'autre part on se donne aussi un réel $q < 0$ qui exprime la similarité d'une lettre avec l'espacement, elle est indépendante de la lettre, ainsi pour tout a on a $p(a, e) = p(e, a) = q$.

Le score d'un alignement est alors donné par

$$\sum_{i=1}^{n} p(u'_i, v'_i)$$

où n est la longueur commune à u' et v'.

Par exemple, pour l'exemple précédent lorsque $p(a, b) = -1$ pour $a \neq b$, $p(a, a) = 5$ et $q = -2$ on trouve un score de 13.

La détermination du score le plus élevé d'un alignement s'effectue grâce à un algorithme de programmation dynamique basé sur la remarque suivante :

Le meilleur alignement de ua et vb s'obtient soit à partir de celui de u et v en alignant a et b, soit à partir de celui de ua et v en alignant b avec un espacement, soit enfin à partir de celui de u et vb en alignant a avec un espacement.

On construit ainsi pas à pas un tableau ms (pour meilleur score) à l'aide de la récurrence suivante :

Soit n la longueur de u et m celle de v, on note par $ms_{i,j}$ (pour $0 \leq i \leq n,\ 0 \leq j \leq m$) le score d'un meilleur alignement entre $u_1 u_2 \cdots u_i$ et $v_1 v_2 \cdots v_j$. Par convention on pose

$ms_{0,j} = qj,\ ms_{i,0} = qi$ correspondant à un aligenment dans lequel tous les caractères de l'une des séquences sont des espacements.

$$ms_{i,j} = \max \begin{cases} ms_{i-1,j-1} + p(u_i, v_j) \\ ms_{i-1,j} + q \\ ms_{i,j-1} + q \end{cases}$$

On note que le calcul de ms s'effectue en temps $O(nm)$ et semble nécessiter un espace de taille $O(nm)$, en fait on constate que l'on peut se limiter à conserver en mémoire une seule ligne de la matrice, la ligne suivante remplace alors la précédente lors du calcul.

Un dernier problème consiste à déterminer l'alignement optimal lui-même une fois calculé son score, c'est à dire connaisant le tableau ms, ceci est relativement simple si l'on connaît la totalité du tableau : on part de $ms_{m,n}$ et on remonte dans le tableau de $ms_{i,j}$ vers celui des $ms_{i-1,j-1}, ms_{i-1,j}$ ou $ms_{i,j-1}$ qui a déterminé sa valeur.

Dans le cas où l'on ne connaît que la dernière ligne du tableau, il faut utiliser une technique procédant du principe de *diviser pour regner* et qui pourra être vue en exercice.

Notons deux problèmes voisins de celui qui est considéré ici qui peuvent être traités par les mêmes techniques.

Le premier consiste étant données deux séquences u et v à trouver un facteur de chacune de celles-ci formant un couple qui présente un alignement de score maximal (rappelons qu'un facteur d'une séquence est une sous-séquence constituée de caractères consécutifs). Pour résoudre ce problème (on utilise ici le fait que $q < 0$ et $p(a, b) < 0$) si $a \neq b$ on applique l'algorithme ci-dessus en considérant $ms_{i,j}$ comme le meilleur score d'un alignement entre un suffixe de $u_1 u_2 \dots u_i$ et un suffixe de $v_1 v_2 \dots v_j$; ce suffixe pouvant être le mot vide, on voit que $m_{i,j}$ ne peut pas être négatif, ainsi 0 doit figurer comme quatrième possibilité dans la formule de récurrence donnant $m_{i,j}$.

Le second problème consiste à déterminer le meilleur alignement pour une fonction de score qui ne tient pas compte des espacements situés en début ou en fin de mot. L'algorithme pour ce problème ne diffère du précédent que dans l'initialisation des $m_{i,0}$ et $m_{j,0}$ qui doivent prendre la valeur 0. On retient alors la valeur maximale de la dernière ligne ou de la dernière colonne.

Bibliographie

[1] Beauquier Danièle, Berstel Jean, Chretienne Philippe *Eléments d'Algorithmique*, Masson, 1992.

[2] Cormen Thomas, Leiserson Charles, Rivest Ronald, *Introduction à d'Algorithmique*, MIT Press, 1990, traduction française chez Dunod.

[3] Knuth Donald, *The Art of Computer Programming, Vol 3 : Sorting and Searching*, Addison Wesley, 1973.

[4] Segwick Robert, *Algorithms in C*, Addison Wesley, 1998.

Index