CH.1 COMPLEXITÉ

- 1.1 Les ordres de grandeur
- 1.2 Les récurrences linéaires
- 1.3 Des exemples

1.1 Les ordres de grandeur

Chaque problème peut être résolu de différentes manières par des algorithmes différents.
Chaque algorithme peut être implémenté de façon différente.
La performance de l'implémentation d'un algorithme est mesurée par différents paramètres.
Le paramètre principal est l'estimation du *temps d'exécution* en fonction de la taille des données.
Un autre paramètre est la *taille* de la mémoire nécessaire en fonction de la taille des données.
On donne ce résultat sous forme *asymptotique*, lorsque la taille tend vers l'infini.
À taille égale des données, le temps ou la place peuvent être différents suivant la donnée particulière. On peut donc s'intéresser au résultat minimum, au maximum ou à la performance en moyenne.

L'implémentation physique (processeur, mémoire, compilateur) fait qu'on ne peut connaître les performances qu'à un facteur constant près.
On cherche en général à trouver un majorant de la performance, d'où l'usage de la notation O.
Les ordres de grandeur qu'on trouve habituellement sont

$O(1)$	ordre *constant*
$O(\log n)$	ordre *logarithmique*

Polynomial :

$O(n)$	ordre *linéaire*
$O(n \log n)$	ordre *n log n*
$O(n^2)$	ordre *quadratique*
$O(n^3)$	ordre *cubique etc*.

Exponentiel :

$O(a^n)$, où $a > 1$

L'analyse en moyenne est généralement plus difficile que l'analyse dans le pire des cas.

L'ordre de la performance en place est au maximum l'ordre de la performance en temps (pour accéder à k valeurs, il faut un temps proportionnel à k.)

Le calcul de la performance ne dispense pas d'expérimenter. La valeur calculée est une *limite*, dans le pire des cas intégrant de plus une *constante*.
Deux méthodes de même complexité théorique peuvent aboutir à des résultats pratiques très différents. L'une peut être bien meilleure en moyenne, ou avoir une constante beaucoup plus petite.
Dans l'intervalle d'usage de l'algorithme, une méthode de complexité plus élevée peut être plus efficace qu'une méthode *a priori* meilleure.

1.2 Les récurrences linéaires

L'outil mathématique nécessaire pour conclure la plupart des analyses d'algorithme est la résolution de récurrences linéaire à coefficients constants, avec second membre.

L'ordre des récurrences est souvent 1.

On résout en considérant d'abord le cas homogène (sans second membre).

Si la récurrence est $A(n) = C\,A(n-1)$, la solution est $A(n) = K\,C^n$, où K est une constante (qui vaut $A(0)$).

Si la récurrence est d'ordre 2, de la forme $A(n) = C\,A(n-1) + D\,A(n-2)$, la solution est $A(n) = K\,r_1^n + L\,r_2^n$, où K et L sont des constantes et où r_1 et r_2 sont les racines (distinctes) du polynôme $X^2 - C\,X - D$. Si le polynôme a une racine double r, la solution est $A(n) = K\,r^n + L\,nr^n$.

Au cas où la récurrence est d'ordre plus élevé, on utilise les formules convenables.

Les seconds membres qu'on traitera sont soit des polynômes, soit des exponentielles. Si le second membre est un polynôme, on essaie un polynôme de degré un de plus. Si c'est une exponentielle a^n, on essaie un multiple de la même exponentielle si elle n'est pas déjà solution de la récurrence homogène et un multiple de na^n sinon. On ajoute à cette solution particulière la solution générale de la récurrence homogène.

Exemples :

$A(n) = A(n-1) + n$ donne $A(n) = A(0) + 1 + ... + n$ et, d'après les formules, $A(n) = (n^2 + n)/2 + A(0)$, d'où la formule pour la somme des n premiers entiers.

En particulier, $A(n) = O(n^2)$.

$A(n) = 2\,A(n-1) + 2n$ a comme solutions $A(n) = A(0)\,2^n - 2n - 4$.
Ici, si $A(0)$ n'est pas nul, alors $A(n) = O(2^n)$.

$A(n) = 2\,A(n-1) + 2^n$ a comme solutions $A(n) = A(0)\,2^n + n\,2^n$.
Ici, on trouve $A(n) = O(n\,2^n)$.

Exemple avec récurrence d'ordre 2 : voir l'exemple des nombres de Fibonacci plus loin.

1.3 Des exemples

Calcul de factorielle : $n! = 1 \times 2 \times \ldots \times n$

Méthode itérative :

```
unsigned long fact(int n)
{
   unsigned long f;
   int i;
   if((n==0)||(n==1)) return 1;
   f=1;
   for(i=2;i<=n;i++) f=f*i;
   return f;
}
```

On a une boucle, parcourue $n-1$ fois, contenant chaque fois une multiplication.
Complexité en temps de $O(n)$ et en place constante.

Méthode récursive :

```
unsigned long fact(int n)
{
   if(n==0) return 1;
   return n*fact(n-1);
}
```

Le temps satisfait une relation de récurrence $T(n)=T(n-1)+C$, qui a comme solution $T(0)+Cn$, donc temps $O(n)$.
La place est *aussi* en $O(n)$.

Dans le cas de la factorielle, on ne peut pas dépasser $n = 13$...
La place n'a pas beaucoup d'importance.

Autre exemple : la suite de Fibonacci $F(n)$.
$F(0) = 0$, $F(1) = 1$, $F(n) = F(n-1) + F(n-2)$

Premières valeurs : 0 1 1 2 3 5 8 13 21 34 55 89 144 233
Valeur exacte $F(n) = \sqrt{5}^{-1}(\varphi^n - \varphi'^n)$ où $\varphi = (1+\sqrt{5})/2$ et $\varphi' = (1-\sqrt{5})/2$.
En particulier, $F(n) = O(\varphi^n)$ croît exponentiellement.

Nous n'allons pas utiliser cette formule pour le calcul de $F(n)$.

Les diverses fonctions vont illustrer les ordres de complexité.

Méthode récursive brutale :

```
unsigned long FIBONACCI1(int n)
{
   if(n==0) return 0;
   if(n==1) return 1;
   return (FIBONACCI1(n-1)+FIBONACCI1(n-2));
}
```

Dans ce cas, le temps vérifie $T(n) = T(n-1) + T(n-2) + C$. La solution générale est de la forme $T(n) = k\,\varphi^n + k'\,\varphi'^n - C = O(\varphi^n)$. Le temps est d'ordre exponentiel. La place est, elle en $O(n)$. (Voir les appels actifs.)

On constate assez vite que $T(n)/T(n-1) \cong 1{,}6$ comme prévu.

La fonction est concise, très proche de la définition, mais la performance est très mauvaise.

Méthode itérative dans un tableau :

```
unsigned long FIBONACCI2(int n)
{
   unsigned long fibo[n+1];
   int i;
   if(n==0) return 0;
   if(n==1) return 1;
   fibo[0]=0; fibo[1]=1;
   for(i=2;i<=n;i++) fibo[i]=fibo[i-1]+fibo[i-2];
   return fibo[n];
}
```

La boucle est parcourue $n-1$fois. Chaque fois il y est fait un nombre fini d'additions. Le temps est donc $T(n) = O(n)$. La place requise est également $O(n)$, à cause du tableau alloué.

On est assez proche de la formule définissant $F(n)$.

Méthode itérative glissante :

```
unsigned long FIBONACCI3(int n)
{
   unsigned long tempo,fi_1,fi_2;
   int i;
   if(n==0) return 0;
   if(n==1) return 1;
   fi_1=1; fi_2=0;
   for(i=2;i<=n;i++)
   {
      tempo=fi_1+fi_2;
      fi_2=fi_1;
      fi_1=tempo;
   }
   return tempo;
}
```

La boucle est parcourue $n-1$fois. Chaque fois il y est fait un nombre fini d'additions et d'affectations. Le temps est donc $T(n) = O(n)$.
La place requise est $O(1)$.

Méthode par exponentiation intelligente.

L'algorithme est fondé sur 2 remarques :
- on peut calculer a^n en effectuant un nombre de multiplications en $O(\log n)$;
- on peut calculer $F(n)$ (comme le résultat de toute récurrence linéaire à coefficients constant) par un calcul de puissance d'une matrice.

Pour la première, on pose $n = 2m + r$, où $r = 0$ ou 1 selon que n est pair ou impair. On a donc $a^n = a^{2m+r} = (a^m)^2$ ou $(a^m)^2 a$ selon que n est pair ou impair. Le nombre de multiplications nécessaires vérifie $M(n) = M(n/2) + 1$ ou 2 selon la parité. En itérant (par exemple) on trouve $M(n) \leq M(n/4) + 4 \leq M(n/2^k) + 2k$. Si $k = 1 + \log n$, dans cette dernière somme il ne reste que $2k$. Donc $M(n) = O(\log n)$.

L'autre remarque est l'écriture matricielle de la récurrence linéaire.

Si l'on pose $F_n = \begin{pmatrix} F(n-1) \\ F(n) \end{pmatrix}$ et $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$,

la récurrence devient $F_n = A\,F_{n-1}$. Cela donne, en itérant, $F_n = A^{n-1}F_1$.

On peut calculer A^{n-1} en temps logarithmique. On trouve $F(n)$ à la deuxième ligne et deuxième colonne de la matrice.

Ceci donne le code suivant pour le calcul de l'exponentielle :

```
{
   unsigned long A1[2][2], B[2][2], C[2][2];
   unsigned long element;
   int i,j,k,l;
   for(i=0;i<2;i++) for(j=0;j<2;j++) A1[i][j]=A[i][j]; /* recopie A dans A1 */
   B[0][0]=B[1][1]=1; B[0][1]=B[1][0]=0; /* matrice identite Id */
   if(n==0) for(i=0;i<2;i++) for(j=0;j<2;j++)
   {
      A[i][j]=B[i][j]; /* A^0=Id */
      return;
   }
   for(l=n;l>0;l=l/2)
   {
      if(l%2==1)
      {
         for(i=0;i<2;i++) for(j=0;j<2;j++) /* Calcul de C=B*A1 */
         {
            element=0;
            for(k=0;k<2;k++) element=element+B[i][k]*A1[k][j];
            C[i][j]=element;
         }
         for(i=0;i<2;i++) for(j=0;j<2;j++) B[i][j]=C[i][j]; /* B=C */
      }
      for(i=0;i<2;i++) for(j=0;j<2;j++) /* Calcul de C=A1^2 */
      {
         element=0;
         for(k=0;k<2;k++) element=element+A1[i][k]*A1[k][j];
         C[i][j]=element;
      }
      for(i=0;i<2;i++) for(j=0;j<2;j++) A1[i][j]=C[i][j]; /* A1=C */
   }
   for(i=0;i<2;i++) for(j=0;j<2;j++) A[i][j]=B[i][j]; /* A=B */
}
```


Ce qui donne le code suivant pour le calcul de F_n :

```
unsigned long FIBONACCI4(int n)
{
   unsigned long A[2][2];
   if(n==0) return 0;
   A[0][0]=0; A[0][1]=A[1][0]=A[1][1]=1;
   EXPONENT(A,n-1);
   return A[1][1];
}
```

Théoriquement, c'est la meilleure méthode, en $O(\log n)$. La place est constante.Mais cela n'est vrai qu'à une constante multiplicative près et asymptotiquement.

La croissance rapide des nombres de Fibonacci fait empêche de les calculer avec le type `unsigned long` dès que n dépasse 48.

Jusqu'à cette valeur, ce dernier algorithme est moins bon que les algorithmes linéaires (voir expériences).