Analyse des algorithmes: une introduction

La question abordée dans ce chapitre est la suivante:

Comment choisir parmi les différentes approches pour résoudre un problème?

Exemples: Liste chaînée ou tableau?
algorithme de tri par insertion de tri
rapide?
…, etc

Pour comparer des solutions, plusieurs points peuvent être pris en considération

- Exactitude des programmes (démontrer que le résultat de l'implantation est celui escompté)
- Simplicité des programmes
- Convergence et stabilité des programmes (il est souhaitable que nos solutions convergent vers la solution exacte; la perturbation des données ne chamboule pas d'une manière drastique la solution obtenue)
- Efficacité des programmes (il est souhaitable que nos solutions ne soient pas lentes, ne prennent pas de l'espace mémoire considérable)

- Le point que nous allons développer dans ce chapitre est celui de l’efficacité des algorithmes.

- **Définition:** Un algorithme est un ensemble d'instructions permettant de transformer un ensemble de données en un ensemble de résultats et ce, en un nombre fini étapes.

- Pour atteindre cet objectif, un algorithme utilise deux ressources d'une machine: **le temps et l'espace mémoire.**

- **Définition 1:** la complexité temporelle d'un algorithme est le temps mis par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.

- **Définition 2**: la complexité spatiale d'un algorithme est l'espace utilisé par ce dernier pour transformer les données du problème considéré en un ensemble de résultats.

Comparaison de solutions

Pour comparer des solutions entre-elles, deux méthodes peuvent être utilisées:

- Méthode empirique
- Méthode mathématique

Cette comparaison se fera, en ce qui nous concerne, relativement à deux ressources critiques: **temps, espace mémoire,…**

Dans ce qui suit, nous allons nous concentrer beaucoup plus sur le temps d'exécution

Facteurs affectant le temps d'exécution:

1. machine,
2. langage,
3. programmeur,
4. compilateur,
5. algorithme et structure de données.

Le temps d'exécution dépend de la longueur de l'entrée.

Ce temps est une fonction **T**(*n*) où *n* est la longueur des données d'entrée.

Exemple 1: x=3; la longueur des données dans ce cas est limite à une seule variable.

Exemple 2:

```
sum = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=n; j++)
    sum++;
```

En revanche, dans ce cas, elle est fonction du paramètre n
}

- La longueur des données d’entrée, définissant le problème considéré, est définie comme étant l’espace qu’elle occupe en mémoire.
- Cet espace est logarithmique de la valeur considérée. Par exemple, le nombre N occupe log N bits d’espace mémoire.

Pire cas, meilleur cas et cas moyen

Toutes les entrées d'une taille donnée ne nécessitent pas nécessairement le même temps d'exécution.

Exemple:

soit à rechercher un élément C dans un tableau de n éléments triés dans un ordre croissant.

Deux solutions s'offrent à nous:

1. Recherche séquentielle dans un tableau de taille n.
   Commencer au début du tableau et considérer chaque élément jusqu'à ce que l'élément cherché soit trouvé ou declaré inexistant.

2. Recherche dichotomique: tient compte du fait que les éléments du tableau sont déjà triés. Information ignorée par l'algorithme de la recherche séquentielle.

- Ces deux algorithmes sont présentés comme suit:

```
int  recherche(int *tab, int C){
   int i;
   i = 0;
  while (i<n && tab[i] != C )
            i = i+1;
  if (i == n)
     return(0);
  else return(i);
}   /* fin de la fonction */
```

```
int  recherche(int *tab, int C){
   int sup, inf, milieu;
   bool trouve;
   inf = 0; sup = n-1; trouve = false;
  while (sup >=inf && !trouve) {
          milieu = (inf + sup) / 2;
          if (C == tab[milieu])
              trouve = true;
         else if (C < tab[milieu])
                    sup = milieu -1;
               else inf = milieu + 1;
  if (!trouve)
     return(-1);
  return(milieu)
}   /* fin de la fonction */
```

La méthode empirique

- Elle consiste à coder et exécuter deux (ou plus) algorithmes sur une batterie de données générées d'une manière aléatoire
- À chaque exécution, le temps d'exécution de chacun des algorithmes est mesuré.
- Ensuite, une étude statistique est entreprise pour choisir le meilleur d'entre-eux à la lumière des résultats obtenus.

Problème!

Ces résultats dépendent

- la machine utilisée;
- jeu d'instructions utilisées
- l'habileté du programmeur
- jeu de données générées
- compilateur choisi
- l'environnement dans lequel est exécuté les deux algorithmes (partagés ou non)
- …. etc.

Méthode mathématique

- Pour pallier à ces problèmes, une notion de complexité plus simple mais efficace a été proposée par les informaticiens.
- Ainsi, pour mesurer cette complexité, la méthode mathématique consiste non pas à la mesurer en secondes, mais à faire le décompte des instructions de base exécutées par ces deux algorithmes.

- Cette manière de procéder est justifiée par le fait que la complexité d'un algorithme est en grande partie induite par l'exécution des instructions qui le composent.

Cependant, pour avoir une idée plus précise de la performance d'un algorithme, il convient de signaler que la méthode expérimentale et mathématique sont en fait complémentaires.

Comment choisir entre plusieurs solutions?

1. **Décompte des instructions**

- Reconsidérons la solution 1 (recherche séquentielle) et faisons le décompte des instructions. Limitons-nous aux instructions suivantes:

- Affectation notée par e
- Test noté par t
- Addition notée par a

- Il est clair que ce décompte dépend non seulement de la valeur C mais de celles des éléments du tableau.

- Par conséquent, il y a lieu de distinguer trois mesures de complexité:

- 1. dans le meilleur cas
- 2. dans le pire cas
- 3. dans la cas moyen

- **Meilleur cas:** notée par $t_{min}(n)$ représentant la complexité de l'algorithme dans le meilleur des cas en fonction du paramètre n (ici le nombre d'éléments dans le tableau).

- **Pire cas**: notée par $t_{max}(n)$ représentant la complexité de l'algorithme dans le cas le plus défavorable en fonction du paramètre n (ici le nombre d'éléments dans le tableau).

- **Cas Moyen**: notée par $t_{moy}(n)$ représentant la complexité de l'algorithme dans le cas moyen en fonction du paramètre n (ici le nombre d'éléments dans le tableau). C'est-à-dire la moyenne de toutes les complexités, t(i), pouvant apparaître pour tout ensemble de données de taille n (t(i) représente donc la complexité de l'algorithme dans le cas où C se trouve en position i du tableau). Dans le cas où l'on connaît la probabilité P_i de réalisation de la valeur t(i), alors par définition, on a:

$$t_{moy}(n) = p_1 t(1) + \ldots + p_n t(n)$$

- Il est clair que pour certains algorithmes, il n’y a pas lieu de distinguer entre ces trois mesures de complexité. Cela n’a pas vraiment de sens.

Meilleur cas pour la recherche séquentielle:

Le cas favorable se présente quand la valeur C se trouve au début du tableau

$t_{min}(n) = e + 3t$ (une seule affectation et 3 test: deux dans la boucle et un autre à l'extérieur de la boucle)

Pire cas: Le cas défavorable se présente quand la valeur C ne se trouve pas du tout dans le tableau. Dans ce cas, l'algorithme aura à examiner, en vain, tous les éléments.

$t_{max}(n) = 1e + n(2t+1e+ 1a)+ 1t + 1t$

$= (n+1)e + na + (2n+2)t$

Cas moyen: Comme les complexités favorable et défavorable sont respectivement (e + 3t) et (n+1)e + na + (2n+3)t, la complexité dans le cas moyen va se situer entre ces deux valeurs. Son calcul se fait comme suit:

On suppose que la probabilité de présence de C dans le tableau A est de ½. De plus, dans le cas où cet élément C existe dans le tableau, on suppose que sa probabilité de présence dans l'une des positions de ce tableau est de 1/n.

Si C est dans la position i du tableau, la complexité t(i) de l'algorithme est:

$$t(i) = (i+1)e + ia + (2i+2)t$$

Par conséquent, dans le cas où C existe, la complexité
moyenne de notre algorithme est :

$$X_{moy}(n) = \frac{1}{n} \sum_{i=0}^{n} (i+1)e + ia + (2i+2)t$$

$$X_{moy}(n) = \frac{1}{2}(3n+1)e + \frac{1}{2}(n+1)a + (n+4)t$$

Par analogie, si l'élément C n'existe pas dans le tableau, la complexité de notre algorithme est $t_{max}(n)$.

Par conséquent:

$$T_{moy}(n) = \frac{1}{2}\left(X_{moy}(n) + t_{\max}(n)\right)$$

$$= \frac{5}{4}(n+1)e + \frac{1}{2}(3n+1)a + \frac{1}{2}(3n+7)t$$

Complexité asymptotique

- Le décompte d'instructions peut s'avérer fastidieux à effectuer si on tient compte d'autres instructions telles que: accès à un tableau, E/S, opérations logiques, appels de fonctions,.. etc.
- De plus, même en se limitant à une seule opération, dans certains cas, ce décompte peut engendrer des expressions que seule une approximation peut conduire à une solution.
- Par ailleurs, même si les opérations élémentaires ont des temps d'exécution constants sur une machine donnée, ils sont différents d'une machine à une autre.

Par conséquent:

- Pour ne retenir que les caractéristiques essentielles d'une complexité, et rendre ainsi son calcul simple (mais indicatif!), il est légitime d'ignorer toute constante pouvant apparaître lors du décompte du nombre de fois qu'une instruction est exécutée.

- Le résultat obtenu à l'aide de ces simplifications représente **la complexité asymptotique** de l'algorithme considéré.

Ainsi, si

$t_{max}(n) = (n+1)e + (n-1)a + (2n+1)t$,

alors on dira que la complexité de cet algorithme est tout simplement en n. On a éliminé tout constante, et on a supposé aussi que les opérations d'affectation, de test et d'addition ont des temps constants.

Définition: La complexité asymptotique d'un algorithme décrit le comportement de celui-ci quand la taille n des données du problème traité devient de plus en plus grande, plutôt qu'une mesure exacte du temps d'exécution.

- Une notation mathématique, permettant de représenter cette façon de procéder, est décrite dans ce qui suit:

Notation grand-O

Définition: Soit **T**(*n*) une fonction non négative. **T**(*n*) est en O(*f*(*n*)) s'il existe deux constante positives *c* et n_0 telles que:
T(*n*) <= *cf*(*n*) pour tout *n* >= n_0.

Utilité: Le temps d'exécution est borné

Signification: Pour toutes les grandes entrées (i.e., *n* >= n_0), on est assurés que l'algorithme ne prend pas plus de *cf*(*n*) étapes.
Þ Borne supérieure.

Grand-O: Exemples

Exemple 1: Initialiser un tableau d'entiers

for (int i=0; i<n; i++) Tab[i]=0;

Il y a n itérations

Chaque itération nécessite un temps <= c,
où c est une constante (accès au tableau + une affectation).

Le temps est donc T(n) <= cn

Donc T(n) = O(n)

Grand-O: Exemples

Exemple 2: T(n) = $c_1 n^2 + c_2 n$.
$c_1 n^2 + c_2 n <= c_1 n^2 + c_2 n^2 = (c_1 + c_2) n^2$
pour tout $n >= 1$.
T(n) <= cn^2 où $c = c_1 + c_2$ et $n_0 = 1$.

Donc, **T**(n) est en O(n^2).

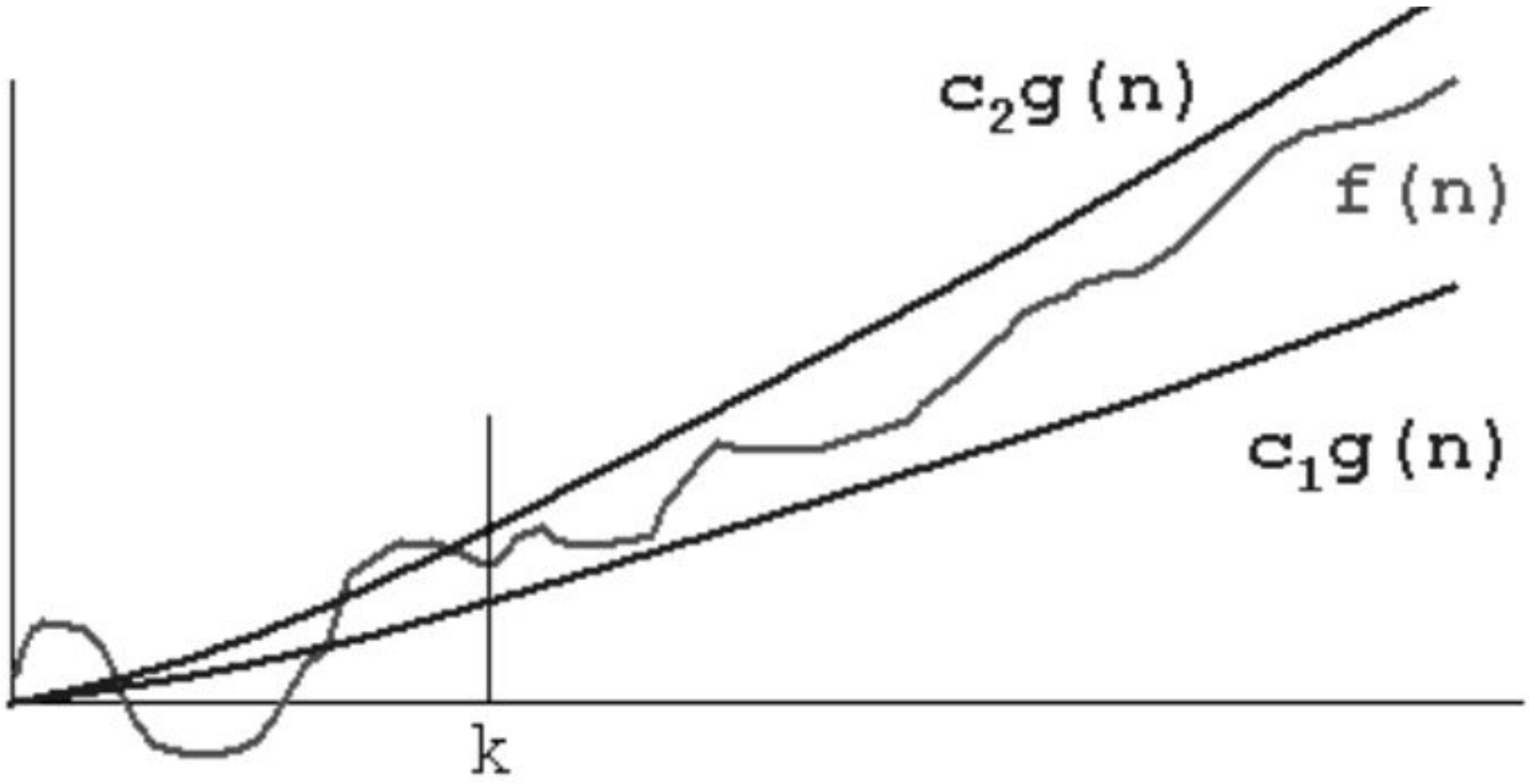
Exemple 3: T(n) = c. On écrit T(n) = O(1).

Grand-Omega

Définition: Soit T(N), une fonction non négative. On a $\mathbf{T}(n) = \Omega(g(n))$ s'il existe deux constantes positives c et n_0 telles que $\mathbf{T}(n) >= cg(n)$ pour $n > n_0$.

Signification: Pour de grandes entrées, l'exécution de l'algorithme nécessite au moins $cg(n)$ étapes.

Donc ça nous donne une borne inférieure.


$c_2g(n)$
$f(n)$
$c_1g(n)$
k

Grand-Omega: Exemple

$\mathbf{T}(n) = c_1 n^2 + c_2 n$.

$c_1 n^2 + c_2 n >= c_1 n^2$ pour tout $n > 1$.

$\mathbf{T}(n) >= cn^2$ pour $c = c_1$ et $n_0 = 1$.

Ainsi, $\mathbf{T}(n)$ est en $\Omega(n^2)$ par définition.

Noter qu'on veut la plus grande borne inférieure.

La notation grand-Theta

Lorsque le grand-O et le grand-omega d'une fonction coïncident, on utilise alors la notation grand-theta.

Définition: Le temps d'exécution d'un algorithme est dans $\Theta(h(n))$ s'il est à la fois en $O(h(n))$ et $\Omega(h(n))$.

Exemple

Q(n)

Q(n^2)

Q(n^3)

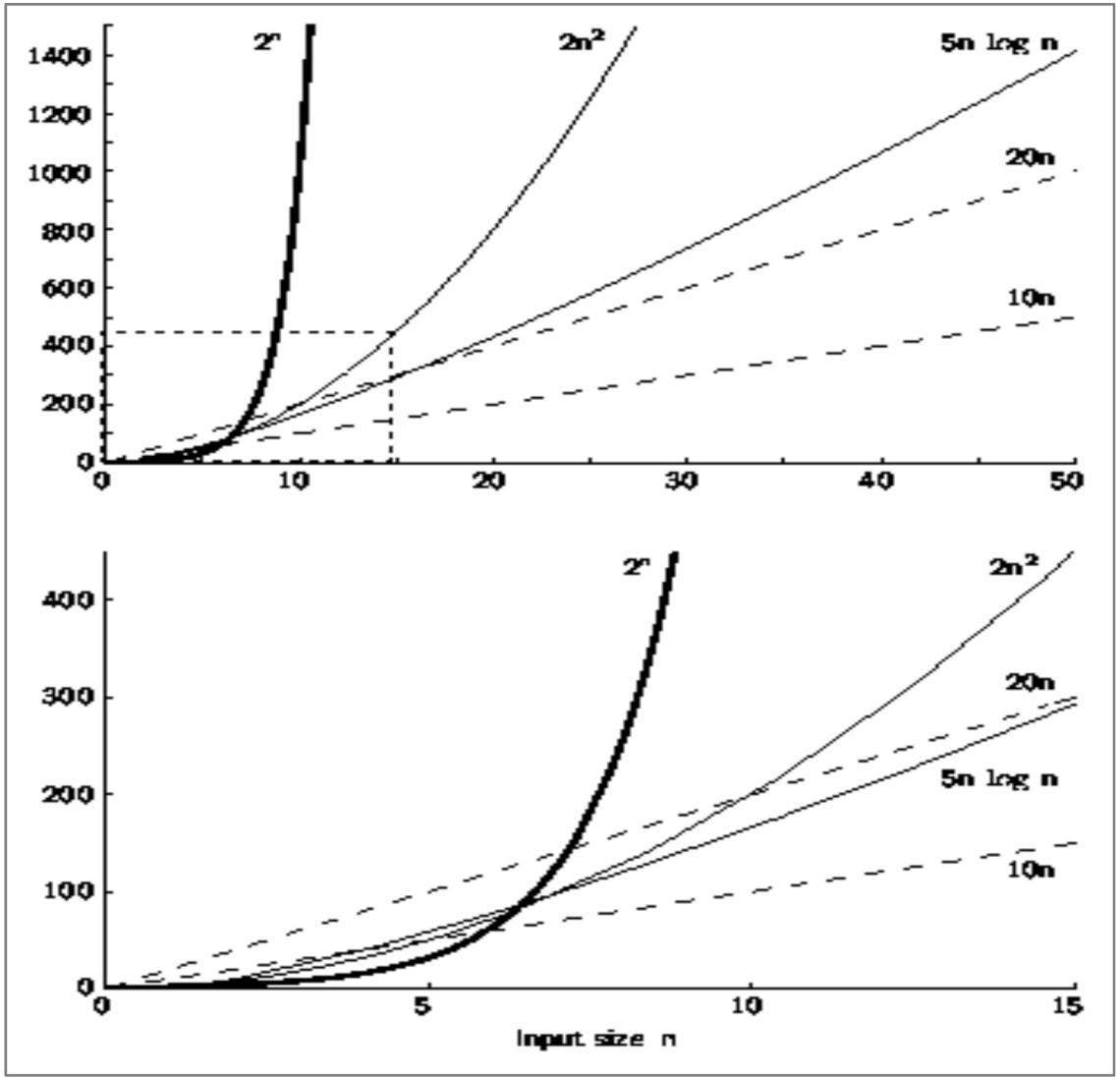
Q(2^n)

Q(lg n)

	$n = 10$	$n = 1000$	$n = 100000$	
n	10	1000	10^5	secondes
$10n$	100	10^4	10^6	
$100n$	1000	10^5	10^7	
n^2	100	10^6	10^{10}	heures
$10n^2$	1000	10^7	10^{11}	
$100n^2$	10^4	10^8	10^{12}	
n^3	1000	10^9	10^{15}	années
$10n^3$	10^4	10^{10}	10^{16}	
$100n^3$	10^5	10^{11}	10^{17}	
2^n	1024	$> 10^{301}$	∞	
$10 \cdot 2^n$	$> 10^4$	$> 10^{302}$	∞	
$100 \cdot 2^n$	$> 10^5$	$> 10^{303}$	∞	
$\lg n$	3.3	9.9	16.6	
$10 \lg n$	33.2	99.6	166.1	
$100 \lg n$	332.2	996.5	1661.0	

$$O(\lg n) \in O(n) \in O(n^2) \in O(n^3) \in O(2^n)$$

Taux de croissance



Remarques

"Le meilleur cas pour mon algorithme est *n* = 1 car c'est le plus rapide." FAUX!

On utilise la notation grand-O parce qu'on s'intéresse au comportement de l'algorithme lorsque *n* augmente et non dans le pire cas.

Meilleur cas: on considère toutes les entrées de longueur n.

Notes

Ne pas confondre le pire cas avec la notation asymptotique.

La borne supérieure réfère au taux de croissance ou à l'ordre de grandeur.

Le pire cas réfère à l'entrée produisant le plus long temps d'exécution parmi toutes les entrées d'une longueur donnée.

Règles de simplification 1

Si

$$f(n) = O(g(n))$$

et

$$g(n) = O(h(n)),$$

alors

$$f(n) = O(h(n)).$$

Règles de simplification 2

Si

$$f(n) = O(kg(n))$$

où $k > 0$ est une constante,

alors

$$f(n) = O(g(n)).$$

Règles de simplification 3

Si

$$f_1(n) = O(g_1(n))$$

et

$$f_2(n) = O(g_2(n)),$$

alors

$$(f_1 + f_2)(n) = O(\max(g_1(n), g_2(n)))$$

Règles de simplification 4

Si

$$f_1(n) = O(g_1(n))$$

et

$$f_2(n) = O(g_2(n))$$

alors

$$f_1(n) f_2(n) = O(g_1(n)\ g_2(n))$$

Quelques règles pour calculer la complexité d'un algorithme

- **Règle 1:** la complexité d'un semble d'instructions est la somme des complexités de chacune d'elles.
- **Règle 2:** Les opérations élémentaires telle que l'affectation, test, accès à un tableau, opérations logiques et arithmétiques, lecture ou écriture d'une variable simple … etc, sont en $O(1)$ (ou en $Q(1)$).

- **Règle 3:** Instruction if: maximum entre le then et le else

switch: maximum parmi les différents cas

Règle 4: Instructions de répétition

1. la complexité de la boucle for est calculée par la complexité du corps de cette boucle multipliée par le nombre de fois qu'elle est répétée.

2. En règle générale, pour déterminer la complexité d'une boucle while, il faudra avant tout déterminer le nombre de fois que cette boucle est répétée, ensuite le multiplier par la complexité du corps de cette boucle.

Règle 5: Procédure et fonction: leur complexité est déterminée par celui de leur corps. L'appel à une fonction est supposé prendre un temps constant en O(1) (ou en Q(1))

Notons qu'on fait la distinction entre les fonctions récursive et celles qui ne le sont pas:

- Dans le cas de la récursivité, le temps de calcul est exprimé comme une relation de récurrence.

Exemples

Exemple 1: a = b;

Temps constant: Q(1).

Exemple 2:

```
somme = 0;
for (i=1; i<=n; i++)
  somme += n;
```

Temps: Q(n)

Exemple 3:

```
somme = 0;
for (j=1; j<=n; j++)
  for (i=1; i<=n; i++)
    somme++;
for (k=0; k<n; k++)
  A[k] = k;
```

Temps: $Q(1) + Q(n^2) + Q(n) = Q(n^2)$

Exemple 4:

```
somme = 0;
for (i=1; i<=n; i++)
  for (j=1; j<=i; j++)
    somme++;
```

Temps: $Q(1) + O(n^2) = O(n^2)$

Exemple 5:

```
somme = 0;
for (k=1; k<=n; k*=2)
  for (j=1; j<=n; j++)
    somme++;
```

Temps: Q(nlog n) pourquoi?

Efficacité des algorithmes

- **Définition:** Un algorithme est dit efficace si sa complexité (temporelle) asymptotique est dans O(P(n)) où P(n) est un polynôme et n la taille des données du problème considéré.
- **Définition:** On dit qu'un algorithme A est meilleur qu'un algorithme B si et seulement si:

$$t_A(n) \in O\big(t_B(n)\big) mais\ t_B(n) \notin t_A(n)$$

Où $t_A(n)$ et $t_B(n)$ sont les complexités des algorithmes A et B, respectivement.

Meilleur algorithme ou ordinateur?

	$\mathbf{n = 10}$	$\mathbf{n = 1000}$	$\mathbf{n = 100000}$	
$\mathbf{n}$	10	1000	10^5	secondes
$\mathbf{10n}$	100	10^4	10^6	
$\mathbf{100n}$	1000	10^5	10^7	
$\mathbf{n^2}$	100	10^6	10^{10}	heures
$\mathbf{10n^2}$	1000	10^7	10^{11}	
$\mathbf{100n^2}$	10^4	10^8	10^{12}	
$\mathbf{n^3}$	1000	10^9	10^{15}	années
$\mathbf{10n^3}$	10^4	10^{10}	10^{16}	
$\mathbf{100n^3}$	10^5	10^{11}	10^{17}	
$\mathbf{2^n}$	1024	$> 10^{301}$	∞	
$\mathbf{10 \cdot 2^n}$	$> 10^4$	$> 10^{302}$	∞	
$\mathbf{100 \cdot 2^n}$	$> 10^5$	$> 10^{303}$	∞	
$\lg \mathbf{n}$	3.3	9.9	16.6	
$\mathbf{10} \lg \mathbf{n}$	33.2	99.6	166.1	
$\mathbf{100} \lg \mathbf{n}$	332.2	996.5	1661.0	

On suppose que l'ordinateur utilisé peut effectuer 10^6 opérations à la seconde

Robustesse de la notation O, Q et W

Algorithmes	Complexité	**Taille max. Résolue par les machines actuelles**	**Taille max. Résolue par les machines 100 fois plus rapides**
A1	log n	T1	$Z1 = (T1)^{100}$
A2	n	T2	Z2 =100T2
A3	n log n	T3	Z3 =100T3
A4	n^2	T4	Z4 = 10T4
A5	2^n	T5	Z5 = T5+log 100
A6	n!	T6	Z6 = T6 + log 100/ logT6 −1

Remarque

- Les relations entre les complexités Ti et Zi, données dans le tableau précédent, peuvent être obtenues en résolvant l'équation suivante:

- 100 f(Ti) = f(Zi)
- Où f(.) représente la complexité de l'algorithme considéré.

Exemple.

- Pour l'algorithme A6 (n!), nous avons à résoudre l'équation suivante:

$$100\ (T6)! = (Z6)!$$

Pour les grandes valeurs de n, nous avons la formule suivante (de Stirling)

$$n! = \left(\frac{n}{e}\right)^n$$

Par conséquent, on obtient ce qui suit:

$$100\left(\frac{T6}{e}\right)^{T6} = \left(\frac{Z6}{e}\right)^{Z6}$$

En introduisant la fonction log, on obtient:

$$\log 100 + T6(\log T6 - 1) = Z6(\log Z6 - 1)$$

En posant Z6 = T6 + e, en approximant log (T6+ e) par log T6, pour de très petites valeurs de e, on obtient:

$$Z6 = T6 + \frac{\log 100}{\log T6 - 1}$$

Exemples d’analyse d’algorithmes non récursifs

Exemple1. Produit de deux matrices

```
void multiplier(int *A[][p], int *B[][m], int *C[][m],
                        int n, int m, int p){
for (i = 0; i<n; i++)
    for (j=0; j<m; j++){
        S = 0;
         for(k =0; k<p; k++)
              S = S + A[i][k]*B[k][j];
          C[i][j] = S;
     } /* fin de la boucle sur j */
} /* fin de la fonction */
```

Analyse: le corps de la boucle sur k est en O(1) car ne contenant qu'un nombre constant d'opérations élémentaires. Comme cette boucle est itérée p fois, sa complexité est alors en O(p). La boucle sur j est itérée m fois. Sa complexité est donc en m.O(p) = O(mp). La boucle sur i est répétée n fois. Par conséquent, la complexité de tout l'algorithme est en O(nmp).

Note: Il est clair qu'il n'y pas lieu de distinguer les différentes complexités: dans tous les cas, nous aurons à effectuer ce nombre d'opérations.

2. **Impression des chiffres composant un nombre**

Le problème consiste à déterminer les chiffres composant un nombre donné. Par exemple, le nombre 123 est composé des chiffres 1, 2 et 3.

Pour les trouver, on procède par des divisions successives par 10. A chaque fois, le reste de la division génère un chiffre. Ce processus est répété tant que le quotient de la division courante est différent de zéro.

- Par exemple, pour 123, on le divise par 10, on obtient le quotient de 12 et un reste de 3 (premier chiffre trouvé); ensuite, on divise 12 par 10, et on obtient un reste de 2 (deuxième chiffre trouvé) et un quotient de 1. Ensuite, on divise 1 par 10; on obtient un reste de 1 (troisième chiffre trouvé) et un quotient de zéro. Et on arrête là ce processus.

L’algorithme pourrait être comme suit:

```
void  divisionchiffre(int n){
   int quotient, reste;
  quotient = n / 10;
  while (quotient >= 10){
      reste = n % 10;
      printf(‘%d ’, reste);
      n = quotient;
      quotient = n / 10;
    }
    reste = n % 10;
   printf(‘%d ’, reste);
}/* fin de la fonction
```

Analyse: Comme le corps de la boucle ne contient qu'un nombre constant d'instructions élémentaires, sa complexité est en O(1). Le problème consiste à trouver combien de fois la boucle while est répétée. Une fois cette information connue, la complexité de tout l'algorithme est facile à dériver. Déterminons donc ce nombre. Soit k l'itération k. Nous avons ce qui suit:

itération k	1	2	3	....	k
valeur de n	n/10	n/100	n/1000	...	$n/10^k$

Donc, à l'itération k, la valeur courante de n est de $n/10^k$

Or, d'après l'algorithme, ce processus va s'arrêter dès que

$$n/10^k < 10$$

Autrement dit, dès que

$$n = n/10^{k+1}$$

En passant par le log,

$$k + 1 = \log n$$

Autrement dit, le nombre d'itérations effectuées est

$$k = O(\log n)$$

Par conséquent, la complexité de l'algorithme ci-dessus est en $O(\log n)$.

3. PGCD de deux nombres

```
int PGCD(int A, int B){
        int  reste;
        reste = A % B;
         while (reste !== 0)
           {
               A = B;
               B = reste;
               reste = A % B;
           }
          return(B);

} /* fin de la fonction */
```

Analyse: Encore une fois, le gros problème consiste à déterminer le nombre de fois que la boucle while est répétée. Il est clair que dans ce cas, il y a lieu normalement de distinguer les trois complexités. En ce qui nous concerne, nous allons nous limiter à celle du pire cas. Pour ce qui est de celle du meilleur cas, elle est facile à déterminer; mais, en revanche, celle du cas moyen, elle est plus compliquée et nécessite beaucoup d'outils mathématique qui sont en dehors de ce cours.

Pour ce faire, procédons comme suit pour la complexité dans le pire cas:

Analyse PGCD suite

Avant de procéder, nous avons besoin du résultat suivant:

Proposition : Si reste = n % m alors reste < n/2
Preuve: Par définition, nous avons:

reste = n –q.m; q >=1

reste <= n –m (1)

On sait aussi que reste <= m -1 (2)

En additionnant (1) avec (2), on obtient:

2 reste < n – 1

donc: reste < n / 2 CQFD

PGCD Suite

Durant les itérations de la boucle while, l'algorithme génère, à travers la variable reste, la suite de nombre de nombre {r0, r1, r2, r3 , ... }, représentant les valeurs que prennent les variable n et m, où

$$r_0 = n ; r_1 = m ;$$

$$r_{j+1} = r_{j-1} \bmod r_j ; \; j \geq 2$$

De la proposition précédente, on peut déduire

$$r_{j+1} < r_{j-1}/2$$

Par induction sur *j*, on obtient l'une des deux relations suivantes, selon la parité de l'indice *j*

PGCD suite

$r_j < r_0 / 2^{j/2}$ si j est pair

$r_j < r_0 / 2^{(j-1)/2}$ si j est impair

Dans les deux cas, la relation suivantes est vérifiée:

$r_j < \max(n,m) / 2^{j/2}$

Dès que $r_j < 1$, la boucle while se termine,
c'est-à-dire dès que:

$$2^{j/2} = \max(n,m)$$

Par conséquent, le nombre de fois que la boucle while est répétée est égal à

$$2\log(\max(n,m)) = O(\log \max(n,m)).$$

Comme le corps de cette boucle est en O(1), alors la complexité de tout l'algorithme est aussi en

$$O(\log \max(n,m))$$

4. Recherche d'un élément dans un tableau trié

```
int  recherche(int *tab, int C){
   int sup, inf, milieu;
   bool trouve;
   inf = 0; sup = n; trouve = false;
  while (sup >=inf && !trouve) {
          milieu = (inf + sup) / 2;
          if (C == tab[milieu])
              trouve = true;
         else if (C < tab[milieu])
                    sup = milieu -1;
               else inf = milieu + 1;
  if (!trouve)
     return(0);
  return(milieu)
}   /* fin de la fonction */
```

Analyse: comme nous l'avons déjà mentionné précédemment, il y a lieu de distinguer entre les trois différentes complexités.

Meilleur cas: Il n'est pas difficile de voir que le cas favorable se présente quand la valeur recherchée C est au milieu du tableau. Autrement dit, la boucle while ne sera itérée qu'une seule fois. Dans ce cas, l'algorithme aura effectué un nombre constant d'opérations; c'est-à-dire en O(1).

- **Pire cas:** Ce cas se présente quand l'élément C n'existe pas. Dans ce cas, la boucle while sera itérée jusqu'à ce que la variable sup < inf. Le problème est de savoir combien d'itérations sont nécessaires pour que cette condition soit vérifiée. Pour le savoir, il suffit de constater, qu'après chaque itération, l'ensemble de recherche est divisé par deux. Au départ, cet intervalle est égal à sup (= n-1) – inf (= 0) + 1 = n.

Itération	intervalle de recherche
0	n
1	n/2
2	n/4
3	n/8
.....	.....
k	$n/ 2^k$

On arrêtera les itérations de la boucle while dès que la condition suivante est vérifiée

$$n/ 2^k = 1 \text{ donc } k = O(\log n)$$

Autrement dit, la complexité de cet algorithme dans le pire cas est en O(log n).

2. Exemples d’analyse d’algorithmes récursifs

- **Définition:** une fonction est récursive si elle fait appel à elle-même d'une manière directe ou indirecte
- La récursivité est une technique de programmation très utile qui permet de trouver des solutions d'une grande élégance à un certain nombre de problèmes.
- **Attention!**

lorsqu'elle mal utilisée, cette subtilité informatique peut créer un code totalement inefficace.

Propriétés d’une récursivité

1. La récursivité (appels de la fonction à elle-même) doit s’arrêter à un moment donné (test d’arrêt). Autrement, l’exécution va continuer indéfiniment

```
void exemple()
{
   cout << "La recursion\n";
   exemple();
}
```

2. Un processus de réduction: à chaque appel, on doit se rapprocher de la condition d'arrêt.

Exemple

```
int mystere (int n, int y){
   if (n == 0) return y;
   else return (mystere (n +1,y));
}
```

- Pour n > 0, la condition d'arrêt ne pourra pas être atteinte.

Tours de hanoi

```
void  hanoi(int n, int i, int j, int  k){
   /*Affiche les messages pour déplacer n
    disques
     de la tige i vers la tige k en utilisant la tige j
    */
   if  (n > 0)
     {
        hanoi(n-1, i, k, j)
        printf ('Déplacer %d vers %d', i,k);
        hanoi(n-1, j, i, k)
      }
} /* fin de la fonction */
```

Analyse de Hanoi

Pour déterminer la complexité de cette fonction, nous allons déterminer combien de fois elle fait appel à elle-même. Une fois ce nombre connu, il est alors facile de déterminer sa complexité. En effet, dans le corps de cette fonction, il a y a:

- Un test
- Deux appels à elle même
- Deux soustractions
- Une opération de sortie

En tout, pour chaque exécution de cette fonction, il y a 6 opérations élémentaires qui sont exécutées.

Hanoi suite

Soit t(n) la complexité de la fonction hanoi(n,i,j,k). Il n'est pas difficile de voir, quelque que soit les trois derniers paramètres, t(n-1) va représenter la complexité de hanoi(n-1, -,-,-).

Par ailleurs, la relation entre t(n) et t(n-1) est comme suit:

$t(n) = t(n-1) + t(n-1) + 6$, si $n > 0$

$t(0) = 1$ (un seul test)

Autrement écrit, nous avons:

$t(n) = 2\,t(n-1) + 6$, si $n > 0$

$t(0) = 1$ (un seul test)

Hanoi suite

Pour résoudre cette équation (de récurrence), on procède comme suit:

$t(n) = 2\,t(n-1) + 6$
$2\,t(n-1) = 4\,t(n-2) + 2.6$
$4t(n-2) = 8\,t(n-3) + 4.6$
.....
$2^{(n-1)}\,t(1) = 2^n\,t(0) + 2^{(n-1)}.6$

En additionnant membre à membre, on obtient:

$t(n) = 2^n\,t(0) + 6(1+2+4+ \ldots + 2^{(n-1)}$
$= 2^n + 6.\,(2^{n-1} - 1)$
$= O(2^n).$

4. Nombres de Fibonacci

```
int Fibonacci(int n){
  int temp;
  if (n==0)
      temp = 0;
  else if (n==1)
            temp = 1;
        else temp = Fibonacci(n-1) + Fibonacci(n-
   2);
  return (temp);
}
```

Soit t(n) la complexité de la fonction Fibonacci(n). Il n'est pas difficile de voir que t(n-1) va représenter la complexité de Fibonacci(n-1) et t(n-2) celle de Fibonacci(n-2).

Par ailleurs, la relation entre t(n), t(n-1) et t(n-2) est comme suit:

$t(n) = t(n-1) + t(n-2) + 8$, si $n > 1$
$t(0) = 1$ (un seul test)
$t(1) = 2$ (2 tests)

Pour résoudre cette équation (aux différences), on va procéder comme suit:

Soit G(x) = Sum_{n=0}^{infini} t(n)x^n

Il est facile de voir:
Sum_{n>1} t(n)x^n = sum_{n>1} t(n-1)x^n +
sum_{n>1}t(n-2)x^n

Pour faire ressortir G(x), on fait comme suit:

Sum_{n>1} t(n)x^n = sum_{n=0}^{infini} t(n)x^n -
t(0)x^0
– t(1)x^1
= G(x) – t(1) –t(0)
Sum_{n>1} t(n-1)x^n = x sum_{n>1}^{infini} t(n-
1)x^(n-1)
= x sum_{n>0}^{infini} t(n)x^(n)
= x sum_{n=0}^{infini} t(n)x^n
–t(0)x^0
= x(G(x) – t(0))

Sum_{n>1} t(n-2)x^n = x^2 sum_{n>1}^{infini} t(n-1)x^(n-2)

= x^2 sum_{n=0}^{infini} t(n)x^(n)

= x^2G(x)

Par conséquent, on obtient:

G(x) – t(1) – t(0) = xG(x) – x – x^2G(x)
G(x)(x^2 – x -1) = x – 3
G(x) = (x-3)/(x^2 – x -1) = (x-3)/(x-a)(x-b)
Où a = (1+racine(5))/2
b = (1-racine(5))/2

On peut aussi mettre

$G(x) = f/(x-a) + g/(x-b)$

On obtient

$a = (1/(\text{racine}(5))$

$b = -(1/(\text{racine}(5))$

$G(x) = 1/(\text{racine}(5))(1/(x-a) – 1/(x-b)$

Rappels de mathématiques:

1/(x-a) = \sum_{n=0}^{infini} (a^nx^n)

et

1/(x-b) = \sum_{n=0}^{infini} (b^nx^n);

Par conséquent:

1/(x-a) - 1/(x-b) = \sum_{n=0}^{infini} (a^n-b^nx^n)

Par conséquent, on obtient:

G(x)= 1/(racine(5))(sum_{n=0}^{infini} (a^n-b^n)x^n) (rel1)

Et nous avons aussi:

G(x) = \sum_{n=0}^{infini} t(n)x^n (rel2)

Par identification entre (rel1) et (rel2), on obtient:

t(n) = 1/(racine(5)(a^n –b^n)

= O(a^n) = O(((1+racine(5))/2)^n)

- Exercice: Montrer d’une manière plus simple que la complexité T(n) de la fonction Fibonacci est en O(2^n).

Éléments de complexité amortie

Complexité amortie

- **Définition**

Dans l'analyse amortie, le temps mis pour effectuer une suite d'opérations est pris comme la moyenne arithmétique sur l'ensemble de ces opérations (ne pas confondre avec l'espérance mathématique) . Cela guarantie une borne supérieure sur le temps moyen de chaque opération dans le pire cas.

Complexité amortie

- La motivation de cette démarche est que la complexité dans le pire cas donne généralement une borne pessimiste car elle ne considère que l'opération la plus coûteuse sur l'ensemble des opérations restantes.
- La méthode amortie consiste à répartir les côuts sur toutes opérations de telle manière que chacune d'elle aura un coût unique.

- En d’autres termes,

L’analyse par rapport à la complexité amortie garantie la performance moyenne de chaque opération dans le plus mauvais cas.

L'analyse amortie est différente de l’analyse en moyenne. En effet,

- dans l'analyse en moyenne, on cherche à exploiter le fait que le pire cas est peu probable en faisant des hypothèses sur la distribution des entrées ou sur les choix aléatoires effectués dans l'algorithme ;
- dans l'analyse amortie, on cherche à exploiter le fait que le pire cas de l'algorithme ne peut pas se produire plusieurs fois consécutivement, ou de manière trop rapprochée, quelle que soit l'entrée.

Complexité amortie 3

- Pensez comme si, lors de vos achats, vous aviez dépensé 10$ pour l'article 1, 20$ pour l'article 2 et 3 dollars pour l'article 3. Vous pourriez dire que le coût de chaque article est de (20+10+3)3 = 11$.
- On voit bien que le coût pour certains articles est sous-sestimé, aors qu'il est surestimé pour d'autres. Ne pas confondre la complexité amortie avec la complexité en moyenne (qui, elle, fait des suppositions probabilistes). Dans le cas amorti, aucune distribution probabiliste n'est requise.
- Cette mesure représente en quelque sorte la complexité en moyenne du cas défavorable.

Complexité amortie 4

- L’approche consiste à affecter un coût artificiel à chaque opération dans la séquence d’opérations. Ce coût artificiel est appelé coût amorti d’une opération.

- Le coût amorti d’une opération est un artifice de calcul qui souvent n’a aucune relation avec le coût réel: souvent, ce coût peut être n’importe quoi. La seule propriété requise par un coût amorti est que le coût réel total de toute la séquence d’opérations doit être borné par le coût total amorti de cette séquence d’opérations.

- Autrement dit, pour les besoins de l’analyse, il sera suffisant d’utiliser le coût amorti au lieu de l’actuel coût de l’opération.

Complexité amortie 5

- L'analyse amortie consiste à estimer une borne supérieure sur le coût total T(n) requis par une séquence de n opérations.
- Quelques opérations, parmi cette séquence, peuvent avoir une coût énorme et d'autres, un coût moindre. L'algorithme génère un coût T(n) pour les n opérations.
- Le coût amorti pour chaque opération est T(n)/n.

Complexité amortie 6

Il existe 3 méthodes pour déterminer la complexité amortie d'un algorithme. La différence réside dans la manière dont le coût est assigné aux différentes opérations.

1. Méthode par aggrégation
2. Méthode comptable
3. Méthode du potentiel

Dans ce qui suit, nous allons illustrer chacune de ces 3 méthodes sur l'exemple du compteur binaire.

Le problème du compteur binaire

Étant donné un tableau de n bits, le problème consiste, à partir des bits 00000…0, à compter le nombre de fois que les bits changent de valeur (de 0 à 1 et de 1 à 0), à chaque fois que la valeur 1 est ajoutée. On suppose que cette opération est répétée n fois. L'algorithme est alors comme suit:

```
INCREMENT(A)

1.      i=0;
2.      while i < length(A) and A[i]=1 do
3.         A[i]=0;
4.         i=i+1;
5.      if  i < length(A) then
6.         A[i] = 1
```

Compteur binaire d'ordre 4

Counter value	COUNTER	Bits flipped (work T(n))
0	0 0 0 **0**	0
1	0 0 **0 1**	1
2	0 0 1 **0**	3
3	0 **0 1 1**	4
4	0 1 0 **0**	7
5	0 1 **0 1**	8
6	0 1 1 **0**	10
7	0 **1 1 1**	11
8	1 0 0 **0**	15

Analyse amortie

- Une analyse naïve montre que'une séquence de n opérations génère un coût de O(n^2). En fait, l'algorithme INCREMENT, dans le pire cas, a une complexité de O(n). S'il est répété n fois, alors la complexité est en O(n^2). En effet, ce ne sont pas tous les bits qui sont changés à chaque itération.

1. Analyse par aggrégation

- Combien de fois A[0] a t-il changé
- Réponse: à chaque fois
- Combien de fois A[1] a t-il changé
- Réponse: chaque 2 fois.
- Combien de fois A[2] a t-il changé
- Réponse: chaque 4 fois.
- ...etc.
- Le coût total pour changer A[0] est n,celui de A[1] est floor(n/2), celui de A[2] est floor(n/4), etc.

- Par conséquent, le coût total amorti est
- T(n) = n + n2 +n/4+…+1
  <= n\sum{i>=0} 1/2^I
  <= 2n

Le coût amorti d’une opération est
t(n) = T(n)/n = 2 = O(1)

2. Analyse comptable

- Dans cette méthode, on assigne des coûts différents aux opérations: quelqu'unes seront surchargées et d'autres sous-chargées. La quantité dont nous chargeons une opération est appelée le coût amorti.
- Quand une opération dépasse son coût réel, la différence est affectée à un autre objet, de la structure de données comme un crédit.

Suite 1

- Ce crédit est ensuite utilisé plus tard pour payer les opérations de complexité amortie moindre que leur coût réel.
- Le coût amorti d'une opération peut donc être vu comme étant réparti entre le coût réel et le crédit qui est soit déposé soit utilisé. Cela est différent de la méthode précédente dans la mesure où les opérations ont toutes le même coût.
- Pour satisfaire la propriété fondamentale d'un coût amorti, il y a lieu d'avoir le coût total amorti comme une borne supérieure du coût réel. Par conséquent, on doit faire attention à ce que le crédit total doit toujours être positif.

Suite 2

- Dans le cas du problème du compteur binaire, assignons un côut amorti de 2unités pour initiliaser un bit à 1 (le 2 vient du fait qu'un bit est soit mis à 1, soit mis à 0). Quand un bit est intiliasé, on utilise 1unité (sur les 2unités) pour payer l'initiliasation proprement dite, et nous plaçons l'autre unité sur ce bit comme crédit.
- En tout temps, tout bit 1 possède 1unité de crédit sur lui. Donc, pas besoin de charger les bits quand ils passent à 0.

Suite 3

- Le coût amorti de l'algorithme peut maintenant être déterminé comme suit:
- Le coût de la réinitialisation de bits dans la boucle while est payé par les unités sur les bits initialisés. Au plus un seul bit est initialisé à chaque itération de l'algorithme, voir ligne 6, et par conséquent le coût amorti par opération est au plus de 2 unités. Le nombre de 1 dans le compteur n'est jamais **<0,** et donc la quantité de crédit est toujours >=0. Par conséquent, pour n opérations de l'algorithme, le coût total amorti est en O(n), majorant le coût total réel.

3. Méthode du potentiel

Au lieu de représenter le travail prépayé comme un crédit stocké avec des objets spécifiques dans la structure de données, la méthode du potentiel de l'analyse amortie représente ce travail comme un potentiel qui peut être libéré pour payer des opérations futures. Ce potentiel est associé à toute la structure de données, au lieu d'une opération comme c'est le cas de la méthode comptable.

Principe de la méthode

- On commence avec une structure de données D_0 sur laquelle n opérations sont effectuées. Pour chaque i =1,2,..,n, on pose ci le cout réel de la ième opération et Di la structure de données qui résulte de l'application de la ième opération sur la structure de données D_{i-1}. La fonction potentielle Ψ qui associe chaque Di à un nombre $\Psi(D_i)$

Suite 1

- Le coût amorti d_i de la ième opération, par rapport à Ψ, est défini comme suit:

d_i = c_i +\psy(D_i) - \psy(D_i-1)

Le coût amorti est le coût réel plus l'augmentation du potentiel dû à l'opération

Le coût total amorti est donc:

\sum_{i=1}^n d_i = \sum_{i=1}^n +\psy(D_n)-\psy(D_0)

Suite 2

- Si nous pouvions définir une fonction\psy telle que \psy(D_n)>= \psy(D_0), alors le coût total amorti est un majorant sur le coût total réel.
- En pratique, on supose toujours

\psy)D_0) = 0.

- Définissons la fonction potentielle comme
- $\Phi(D_i)$ = bi = nombre de 1 dans la structure de données.

Déterminons le coût amorti de l'algorithme INCREMENT

Supposons que la ième exécution de INCREMENT a changé t_i bits à 0. Alors le coût total de cette opération est au plus t_i +1, car en plus de cette initialisation, elle peut aussi mettre un 1(le plus à gauche

- Le nombre de 1 dans le tableau, après la ième opération de INCREMENT est

bi <= b_{i-1}-t_i +1

La différence des potentiels est alors

\psy(D_i) - \psy(D_{i-1} <= b_{i-1}-t_i +1

- b_{i-1}

<= 1-t_i

Le coût amorti est donc

d_i = c_i + \psy(D_i) - \psy(D_{i-1}

<= (t_i + 1)+ (1-t_i)

= 2 = O(1)

Comme le compteur commence à 0, alors nous avons bien

$$\Phi(D_i) \geq \Phi(D_0) = 0.$$

Le coût total amorti est bien un majorant sur le coût total réel. Le coût total amorti est bien n*O(1) = O(n).

Quelques Références

1. D. Rebaïne (2000): une introduction à l'analyse des algorithmes, ENAG Édition.

2. Cormen *et al.* (1990): Algorithms, MacGraw Hill.

3. J.M. Lina (2004): Analyse amortie des algorithmes, ETS, Montréal.

4. Data structures, Algorithmes and applications,
Sartaj Sahni, Silicon Press, 1999.