

Université de Monastir

Cours: Conception et Analyse d'Algorithmes

Chapitre 2: Analyse d'algorithmes

Réalisé par:

Dr. Sakka Rouis Taoufik

1

Chapitre 2 : Analyse d'algorithmes

I. Introduction

La classe des algorithmes qui satisfont une spécification donnée, si elle n'est pas vide, contient en général une infinité d'éléments.

En effet, un problème ayant une spécification donnée peut avoir plusieurs algorithmes : $A_1, A_2, A_3, \dots, A_n$.

Par exemple, on cite : problème de tri et le problème de recherche dans une table.

Questions posées :

- Sur quelle base peut-on comparer ces algorithmes deux à deux ?
- Quels critères est-il pertinent d'utiliser dans la pratique ?₂

Chapitre 2 : Analyse d'algorithmes

I. Introduction

Typiquement, on distingue des **critères statiques** (c'est-à-dire indépendants de l'exécution de l'algorithme) et des critères dynamiques. En ce qui concerne les critères statiques, nous pouvons citer :

-le temps de développement d'une solution (d'un algorithme) à partir d'une spécification : (la lisibilité du programme ; sa maintenabilité ; la qualité de son ergonomie ; sa sécurité d'utilisation ; sa longueur ; etc)

➔ Les quatre premiers points ne doivent être négligés, mais ils sont **trop subjectifs**. Ils dépendent autant du programmeur et /ou du lecteur que de l'algorithme lui-même pour permettre une comparaison **impartiale**.

3

Chapitre 2 : Analyse d'algorithmes

I. Introduction

L'objectif de sécurité est, du point de vue des outils qui permettent d'évaluer l'algorithme, très lié à **la notion de correction** du programme par rapport à sa spécification.

Illustration : Démontrer qu'une propriété est satisfaite pour le programme comme par exemple le programme ne modifie que des fichiers locaux.

Le critère longueur est à la base de la notion de complexité de **Kolmogorov**.

En ce qui concerne les **critères dynamiques**, ils portent soit sur la place mémoire (**la complexité spatiale**) utilisée pour une exécution soit sur le temps d'exécution (**la complexité temporelle**).

4

Chapitre 2 : Analyse d'algorithmes

I. Introduction

La complexité algorithmique permet de mesurer les performances d'un algorithme et de le comparer avec d'autres algorithmes réalisant les même fonctionnalités.

La complexité d'un algorithme consiste en l'étude de la quantité de ressources (de **temps** ou **espace**) nécessaire à l'exécution de cet algorithme

La complexité algorithmique est un concept fondamental pour tout informaticien, elle permet de déterminer si un algorithme **a** est meilleur qu'un algorithme **b** et s'il est optimal ou s'il ne doit pas être utilisé. . .

5

Chapitre 2 : Analyse d'algorithmes

II. Complexité spatiale versus complexité temporelle

Plusieurs arguments militent en faveur du critère temporel plutôt que du critère spatial :

- pour occuper de la place il faut consommer du temps : l'inverse n'est pas vrai

- la place occupée à l'issue de l'exécution, en supposant que l'exécution termine, et est à nouveau disponible. La mémoire est une ressource recyclable.

- la complexité temporelle établit une ligne de partage consensuelle entre les solutions praticables et les autres. Penser aux algorithmes de complexité exponentielle et supérieurs.

6

Chapitre 2 : Analyse d'algorithmes

II. Complexité spatiale versus complexité temporelle

Un Type de données peut être mis en œuvre (ou implémentés) par plusieurs Structures de Données : SD_1 , SD_2, \dots, SD_n .

La comparaison est en général impossible pour deux structures de données relatives à un type de données, puisqu'elle devrait porter sur plusieurs opérations.

➔ L'idéal serait de concevoir une mise en œuvre qui soit la meilleure pour toutes les opérations.

Autre idée : définir une structure d'ordre sur les opérations. Ceci va induire une forme originale d'analyse de complexité : **l'analyse amortie**

Chapitre 2 : Analyse d'algorithmes

III. Notion de croissance asymptotique

Comparaison de fonctions

Considérons le problème du tri d'un tableau sans doublon. Ce problème est spécifié informellement de la manière suivante :

«la fonction $\text{tri}(t)$ délivre un tableau représentant la permutation triée par ordre croissant des valeurs de t .» Supposons que nous cherchions à comparer la complexité de deux fonctions déterministes $\text{tri}_1(t)$ et $\text{tri}_2(t)$ qui satisfont cette spécification.

Supposons par ailleurs que la complexité de la fonction tri_1 (respectivement tri_2) soit donnée par la fonction $f_1(n)$ (respectivement $f_2(n)$), n étant la taille de t .

Chapitre 2 : Analyse d'algorithmes

III. Notion de croissance asymptotique

Nous intéressons aux fonctions de profil $N \rightarrow N$ ou $N \rightarrow \mathbb{R}^+$.

Ce choix est justifié par le fait que la complexité d'un algorithme est une fonction à valeurs entières ou réelles non négatives définie sur une grandeur scalaire discrète, typiquement **la taille de la donnée**.

Le caractère total de ces fonctions tient au fait qu'en général toutes **les tailles sont significatives**.

9

Chapitre 2 : Analyse d'algorithmes

III. Notion de croissance asymptotique

Définition 1 (Notation O omicron)

Soit $g(n)$ une fonction de $N \rightarrow \mathbb{R}^+$ On définit:

$$O(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \geq 0, \forall n \geq n_0, f(n) \leq c g(n)\}$$

→ Cela signifie qu'à partir certain rang, la fonction f est majorée par une constante fois la fonction g . Il s'agit donc d'une situation de **domination** de la fonction f par la fonction g .

Exemples:

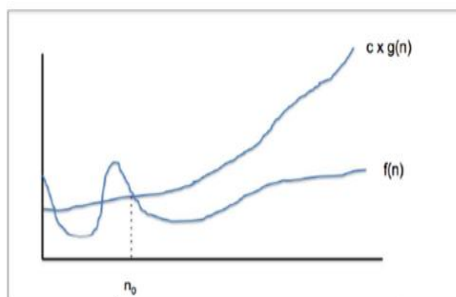
$$n^2 \in O(2^n + 2n)$$

$$20n^2 \in O(3n^3 + 2n + 1)$$

$$20n^2 \in O(20n^2 + 50n + 5)$$

$$20n^2 \in O(19n^2)$$

$$n^2 \notin O(200n)$$



10

Chapitre 2 : Analyse d'algorithmes

III. Notion de croissance asymptotique

Exercices sur la notation O omicron

Exercice 1: Prouver que :

- 1) Si $f(n)=4$ alors $f(n)= O(1)$,
→ prendre par exemple $c=5$ et $n_0 = 1$
- 2) Si $f(n)=3n+2$ alors $f(n)= O(n)$,
→ prendre par exemple $c=4$ et $n_0 = 2$
- 3) Si $f(n)=2n^2+3$ alors $f(n)= O(n^2)$,
→ prendre par exemple $c=3$ et $n_0 = 2$

11

Chapitre 2 : Analyse d'algorithmes

III. Notion de croissance asymptotique

Exercices sur la notation O omicron

Exercice 2:

Pour chacune des fonctions $T_i(n)$ suivantes déterminer sa complexité asymptotique dans la notation O.

Exemple

$$0) T_0(n) = 3n \quad \in O(n)$$

$$1) T_1(n) = 6n^3 + 10n^2 + 5n + 2$$

$$2) T_2(n) = 3 \log n + 4$$

$$3) T_3(n) = 2^n + 6n^2 + 7n$$

$$4) T_4(n) = 7k + 2$$

$$5) T_5(n) = 4 \log n + n$$

$$6) T_6(n) = 6n^3 + 10n^2 + 5n + 2$$

12

Chapitre 2 : Analyse d'algorithmes

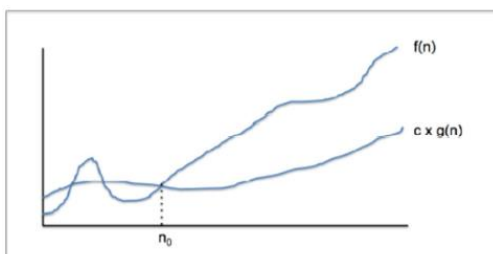
III. Notion de croissance asymptotique

Définition 2 (Notation Ω oméga)

Soit $g(n)$ une fonction de $\mathbb{N} \rightarrow \mathbb{R}^+$ On définit

$$\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 \geq 0, \forall n \geq n_0, \quad c g(n) \leq f(n)\}$$

→ Cela signifie qu'à partir d'un certain rang, la fonction f est minorée par une constante fois la fonction g . Il s'agit donc d'une situation de **domination** de la fonction g par la fonction f .



13

Chapitre 2 : Analyse d'algorithmes

III. Notion de croissance asymptotique

Exercice sur la notation Ω oméga

Prouver que :

- 1) Si $f(n)=4$ alors $f(n)= \Omega(1)$,
- 2) Si $f(n)=4n+2$ alors $f(n)= \Omega(n)$
- 3) Si $f(n)=4n^2+1$ alors $f(n)= \Omega(n^2)$

14

Chapitre 2 : Analyse d'algorithmes

III. Notion de croissance asymptotique

Définition 3 (Notation θ théta)

Soit $g(n)$ une fonction de $\mathbb{N} \rightarrow \mathbb{R}^+$ On définit:

$$\Theta(g(n)) = \{f(n) \mid \exists c_1 > 0, \exists c_2 > 0, \exists n_0 \geq 0, \forall n \geq n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)\}$$

→ g est équivalent (ou comparable) asymptotiquement à f

- A partir d'un certain rang n_0

la fonction $f(n)$ peut être bornée inférieurement par $c_1 * g(n)$;

la fonction $f(n)$ peut être bornée supérieurement par $c_2 * g(n)$;

Exemples

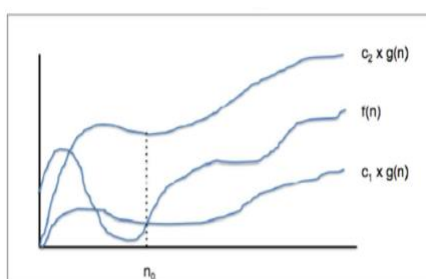
$$n^2 \notin \theta(2^n + 2n)$$

$$20 n^2 \notin \theta(3n^3 + 2n + 1)$$

$$20 n^2 \in \theta(20n^2 + 50n + 5)$$

$$20 n^2 \in \theta(19n^2)$$

$$n^2 \notin \theta(200n)$$



15

Chapitre 2 : Analyse d'algorithmes

III. Notion de croissance asymptotique

Exercice sur la notation θ théta: Prouver que :

1) Si $f(n)=4$ alors $f(n) \in \theta(1)$,

2) Si $f(n)=4n+2$ alors $f(n) \in \theta(n)$,

3) Si $f(n)=4n^2+1$ alors $f(n) \in \theta(n^2)$

16

Chapitre 2 : Analyse d'algorithmes

IV. Règles de calcul de la complexité en O

Pour calculer la complexité grand **O** d'un algorithme il faut compter le nombre **d'opérations de base** qui effectue comme :

- Opération arithmétique ou logique
 - Opération d'affectation
 - Vérification d'une condition
 - Opération d'entrée/Sortie
- ➔ La complexité de chaque opération de base est **constante** ou **O(1)**

17

Chapitre 2 : Analyse d'algorithmes

IV. Règles de calcul de la complexité en O

Complexité temporelle d'une suite d'opérations élémentaires

- **Cas d'une instruction simple** (écriture, lecture, affectation):

Le temps d'exécution de chaque instruction simple est **O(1)**

- **Cas d'une suite d'instruction simple:**

Le temps d'exécution d'une séquence d'instructions est déterminée par la règle de la somme. C'est donc le temps de la séquence qui a le plus grand temps d'exécution:

$$O(T) = O(T_1 + T_2) = \max(O(T_1); O(T_2))$$

Traitement 1 $T_1(n)$

Traitement 2 $T_2(n)$

18

Chapitre 2 : Analyse d'algorithmes

IV. Règles de calcul de la complexité en O

➤ Cas d'un traitement conditionnel

Le temps d'exécution d'une instruction **Si** est le temps d'exécution des instructions exécutées sous condition, + le temps pour évaluer la condition. Pour une alternative, on se place dans le cas le plus défavorable.

Exemple:

Si (condition) **Alors**

Traitement 1

Sinon

Traitement 2

Fin Si

$$O(T) = O(T_{\text{cond}}) + \max [O(T_{\text{trait 1}}), O(T_{\text{trait 2}})]$$

19

Chapitre 2 : Analyse d'algorithmes

IV. Règles de calcul de la complexité en O

➤ Cas d'un traitement itératif

Le temps d'exécution d'une boucle est la somme du temps pour évaluer le corps et du temps pour évaluer la condition. Souvent ce temps est le produit du nombre d'itérations de la boucle par le plus grand temps possible pour une exécution du corps,

```
while (cond ){
    /* trait */
```

```
}
```

```
do{
    /* trait */
```

```
}while (cond);
```

```
For ( ; cond; ){
    /* trait */
```

```
}
```

$$O(T) = \text{nombre d'itérations} \times [O(T_{\text{cond}}) + O(T_{\text{trait}})]$$

20

Chapitre 2 : Analyse d'algorithmes

IV. Règles de calcul de la complexité en O

Quelle est l'opération fondamentale ou élémentaire dans cette fonction?

```
int somme(int T[], int n)
{
    int i, S=0;
    for (i=0 ; i<n ; i++) /* n itérations */
        S=S+T[i];        /* opération fondamentale ou élémentaire */
    return S;
}
```

La complexité Temporelle de l'exemple ci-dessus est : $O(N)$
 → Temps Linéaire

La complexité spatiale = $O(n + 3)$ (n cases de T + i + S + n) = $O(N)$

21

Chapitre 2 : Analyse d'algorithmes

IV. Règles de calcul de la complexité en O

Question: Quelle est l'opération fondamentale ou élémentaire dans cette fonction?

```
int recherche (int T[], int n , int info)
{
    int i=0;
    do
    {
        if (T[i]==info) /* opération fondamentale ou élémentaire */
            return 1;
        else
            i=i+1;
    }while(i<n);
    return 0;
}
```

La complexité spatiale = $O(n + 3)$ = $O(N)$

22

Chapitre 2 : Analyse d'algorithmes

IV. Règles de calcul de la complexité en O

Pour la complexité temporelle: On compte le nombre de répétition de l'opération fondamentale ou élémentaire ($T[i] == \text{info}$) ?

- 1 fois : si $\text{info} == T[0]$: meilleur des cas (ou cas optimiste ou minimum) → la complexité temporelle est $O(1)$
- n fois : si $\text{info} == T[n-1]$: pire des cas (ou cas pessimiste ou maximal) → la complexité temporelle est $O(n)$
- $n/2$ fois: si $\text{info} == T[(n-1)/2]$: cas moyen (ou cas réaliste) → la complexité temporelle est $O(n/2) \rightarrow O(n)$

23

Chapitre 2 : Analyse d'algorithmes

V. De la relation de récurrence vers la complexité

■ Méthode 1

Uniquement quand la relation de récurrence est de la forme

$$T(n) = aT(n-b) + f(n)$$

- a, b des constantes positives $a, b > 0$
- $f(n) \in O(n^c \log_2^d(n))$ avec $c, d \geq 0$

Deux cas :

- 1 $a = 1$: $T(n) \in O(nf(n))$
- 2 $a > 1$ et $d = 0$: $T(n) \in O(a^{\frac{n}{b}} f(n))$

Exemple 1: $T(n) = 2T(n-1) + 1$

$a = 2$; $b = 1$; $c = 0$; $d = 0$

$a > 1$; $d = 0 \rightarrow$ cas 2

$T(n) = O(2^n)$

Exemple 2: $T(n) = T(n-1) + n$

$a = 1$; $b = 1$; $c = 1$; $d = 0$

$a = 1 \rightarrow$ cas 1

$T(n) = O(n^2)$

24

Chapitre 2 : Analyse d'algorithmes

V. De la relation de récurrence vers la complexité

■ Méthode 2: Méthode Master (1989)

Uniquement quand la relation de récurrence est de la forme

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- a une constante $a \geq 1$
- b une constante $b > 1$
- $f(n)$ une fonction asymptotiquement positive

Trois cas :

- ① $f(n) \in O(n^c)$ avec $b^c < a$: $T(n) \in O(n^{\log_b(a)})$
- ② $f(n) \in O(n^c \log_2^d(n))$ avec $d \geq 0$ une constante et $b^c = a$:
 $T(n) \in O(n^c \log_2^{d+1}(n))$
- ③ $f(n) \in O(n^c)$ avec $b^c > a$: $T(n) \in O(f(n))$

25

Chapitre 2 : Analyse d'algorithmes

V. De la relation de récurrence vers la complexité

■ Méthode 2: Méthode Master (1989)

Exemple 1: $T(n) = 4T(n/2) + n$

$$a = 4; b = 2; c = 1$$

$$b^c = 2^1 = 2 < a \rightarrow \text{cas 1}$$

$$T(n) = O(n^2)$$

Exemple 3: $T(n) = 2T(n/2) + n$

$$a = 2; b = 2; c = 1$$

$$b^c = 2^1 = 2 = a \rightarrow \text{cas 2}$$

$$T(n) = O(n \log_2(n))$$

Exemple 2: $T(n) = 3T(n/2) + n$

$$a = 3; b = 2; c = 1$$

$$b^c = 2^1 = 2 < a \rightarrow \text{cas 1}$$

$$T(n) = O(n^{\log_2(3)})$$

Exemple 4: $T(n) = 2T(n/2) + n^2$

$$a = 2; b = 2; c = 2$$

$$b^c = 2^2 = 4 > a \rightarrow \text{cas 3}$$

$$T(n) = O(n^2)$$

26

Chapitre 2 : Analyse d'algorithmes

V. De la relation de récurrence vers la complexité

■ Méthode 3: Méthode Akra-Bazzi (1998)

Généralisation de la méthode Master pour les relations de récurrence de la

$$\text{forme } T(n) = \sum_{i=1}^k T(b_i * n) + f(n)$$

- b_i une constante positive $b_i < 1$
- k une constante $k \geq 1$
- $f(n) \in O(n^c \log_2^d(n))$ avec $c, d \geq 0$

On pose $e = \sum_{i=1}^k b_i^c$

Trois cas :

- 1 $e < 1 : T(n) \in O(n^c \log_2^d(n))$
- 2 $e = 1 : T(n) \in O(n^c \log_2^{d+1}(n))$
- 3 $e > 1 : T(n) \in O(n^x)$ avec x l'unique solution de $\sum_{i=1}^k b_i^x = 1$

27

Chapitre 2 : Analyse d'algorithmes

V. De la relation de récurrence vers la complexité

■ Méthode 3: Méthode Akra-Bazzi (1998)

Exemple 1: $T(n) = 4T(n/2) + n$

$$k = 4, b_i = \frac{1}{2}, c = 1, d = 0$$

$$e = \sum_{i=1}^k b_i^c = \sum_{i=1}^4 \frac{1}{2} = 2 > 1 \Rightarrow \text{cas 3}$$

$$\sum_{i=1}^4 \left(\frac{1}{2}\right)^x = 1 \iff \left(\frac{1}{2}\right)^x = \frac{1}{4}, x = 2$$

$$T(n) = O(n^2)$$

Exemple 2: $T(n) = T(n/2) + T(n/4) + n^2$

$$k = 2, b_1 = \frac{1}{2}, b_2 = \frac{1}{4}, c = 2, d = 0$$

$$e = \sum_{i=1}^k b_i^c = \left(\frac{1}{2}\right)^2 + \left(\frac{1}{4}\right)^2 = \frac{5}{16} < 1 \Rightarrow \text{cas 1}$$

$$T(n) = O(n^2)$$

28

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

On distingue trois cas :

-cas optimiste ou favorable ou meilleur T_{\min} : une donnée qui nécessite le temps minimum d'exécution

-cas pessimiste ou le plus défavorable ou pire T_{\max} : une donnée qui nécessite le temps maximum d'exécution

-cas moyen T_{moy} : une donnée dont le temps d'exécution est quelque part entre T_{\min} et T_{\max}

29

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

Remarque

Certains algorithmes sont insensibles à la donnée initiale. Par exemple, le tri par sélection ne dépend pas de la configuration initiale du tableau à trier. Ainsi pour cet algorithme $T_{\min} = T_{\max} = T_{\text{moy}} = (n^2)$ où n est la taille du tableau à trier.

30

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

Exercice 1: problème de recherche dichotomique

La technique de recherche dichotomique n'est applicable que si le tableau est déjà trié (par exemple dans l'ordre croissant). Le but de recherche dichotomique est de diviser l'intervalle de recherche par 2 à chaque itération. Pour cela, on procède de la façon suivante :

Soient premier et dernier les extrémités gauche et droite de l'intervalle dans lequel on cherche la valeur x , on calcule M , l'indice de l'élément médian :

$$M = (\text{premier} + \text{dernier}) \div 2$$

Il y a 3 cas possibles :

$x = T[M]$: l'élément de valeur x est trouvé, la recherche est terminée

$x < T[M]$: l'élément x , s'il existe, se trouve dans l'intervalle $[\text{premier}..M-1]$

$x > T[M]$: l'élément x , s'il existe, se trouve dans l'intervalle $[M+1..\text{dernier}]$

La recherche dichotomique consiste à itérer ce processus jusqu'à ce que l'on trouve x ou que l'intervalle de recherche soit vide.

31

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

1) Réalisation

32

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

2) Calcul de la complexité

Relation de complexité est :

$T(N) = T(N/2) + 1 \rightarrow$ Méthode Master

$a = 1; b = 2; c = 0$

$b^c = 2^0 = 1 = a \rightarrow$ cas 2

$T(n) = O(n^0 \log_2^1 n) = O(\log_2 n)$

33

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

3) Analyse de la complexité

-Complexité en temps :

On va comptabiliser l'opération de comparaison?

On distingue les cas suivant :

-Cas minimum ou optimiste: une seule comparaison, ceci traduit que x coïncide avec $T[0+(N-1)/2]$.

-Cas maximum ou pessimiste : (dans le pire des cas) un tel cas traduit que x n'appartient pas à T . À chaque itération, on part d'un problème de taille N et moyennant une comparaison, on tombe sur un problème de taille $N/2$. L'algorithme de recherche dichotomique applique le principe « diviser pour résoudre » ou encore « diviser pour régner » : le problème initial est divisé en deux sous problèmes disjoints et de taille plus ou moins égale.

34

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

3) Analyse de la complexité

Supposant que N est le nombre d'éléments

On note C_N : le nombre de fois où $X==T[n]$ est effectuée.

$$\rightarrow C_N = C_{N/2} + 1$$

avec $C_1 = 1$

On pose $N=2^n$ ou $n=\log_2 N \rightarrow C_N = C_{2^n}$

$$C_{2^n} = C_{2^{n-1}} + 1$$

$$= C_{2^{n-2}} + 1 + 1$$

$$= C_{2^{n-3}} + 3$$

...

$$= C_1 + n = 1 + n = 1 + \log_2 N$$

Ainsi, cet algorithme est $O(\log_2 N)$

-Cas moyen : entre 1 et $\log_2 N$

35

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

3) Analyse de la complexité

Remarque : le gain apporté par l'application de l'algorithme de recherche dichotomique sur un tableau **trié** peut être illustré par l'exemple suivant :

On souhaite effectuer une recherche sur un tableau trié T de taille $N=10000$.

Si on applique

l'algorithme de recherche séquentielle la complexité dans le cas moyen est 5000

l'algorithme de recherche dichotomique, la complexité au pire des cas est

$$O(\log_2 10000) \approx 14$$

36

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

Exercice 2:

Quelle est la complexité de la méthode suivante ?

```
int puissance (int n, int a) {
    int aux = n ;
    int p = a ;
    int resultat=1 ;
    while ( aux != 0 ) {
        if (aux % 2 == 1) {
            resultat = resultat * p ;
        }
        aux=aux/2 ;
        p = p * p ;
    }
    return resultat ;
}
```

37

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

Exercice 3: problème de tri fusion

Le principe de **trie par fusion** est :

- Découper le tableau $T[1, .. n]$ à trier en deux sous-tableaux $T[1, .. n/2]$ et $T[n/2 + 1, .. n]$
- Trier les deux sous-tableaux $T[1, .. n/2]$ et $T[n/2 + 1, .. n]$ (récursivement, ou on ne fait rien s'ils sont de taille 1)
- Fusionner les deux sous-tableaux triés $T[1, .. n/2]$ et $T[n/2 + 1, .. n]$ de sorte à ce que le tableau final soit trié.

- 1) Proposer une implémentation récursive pour cette technique
- 2) Déterminer sa complexité.

38

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

2) Réalisation

```
#include <stdio.h>

void fusion (int T [ ], int g, int m, int d); /* → O(n) */

void TriFusion(int T [ ], int g, int d){
    int m;
    if (g<d){
        m=(g+d)/2;
        TriFusion(T, g, m) ;
        TriFusion(T, m+1, d) ;
        Fusion(T, g, m, d) ;
    }
}
```

39

Chapitre 2 : Analyse d'algorithmes

VI. Analyse classique de la complexité des fonctions

2) Complexité au pire des cas

Supposant que N est le nombre d'éléments

On note C_N : le nombre de fois où l'instruction de base est effectuée.

$$\rightarrow C_N = 2 * C_{N/2} + N$$

avec $C_1=0$ (1 seul élément donc rien a faire)

On pose $N=2^n$ ou $n=\log_2 N \rightarrow C_N = C_{2^n}$

$$\begin{aligned} C_{2^n} &= 2 * C_{2^{n-1}} + 2^n \\ &= 2 * (2 * C_{2^{n-2}} + 2^{n-1}) + 2^n \\ &= 2^2 * C_{2^{n-2}} + 2^n + 2^n \\ &= 2^2 * C_{2^{n-2}} + 2 * 2^n \\ &= 2^2 * (2 * C_{2^{n-3}} + 2^{n-2}) + 2 * 2^n \\ &= 2^3 * C_{2^{n-3}} + 2^n + 2 * 2^n \\ &= 2^3 * C_{2^{n-3}} + 3 * 2^n \\ &\dots \\ &= 2^n * C_{2^{n-n}} + n * 2^n = 0 + (\log_2 N) * N \end{aligned}$$

Ainsi, cet algorithme est $O(N \log_2 N)$

40

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

Dans une analyse amortie, le temps requis pour effectuer une suite (ou séquence) d'opérations sur une structure de données est une moyenne sur l'ensemble des opérations effectuées.

L'analyse amortie permet de montrer que **le coût moyen** d'une opération est faible si l'on établit sa moyenne sur une suite d'opérations, même si l'opération prise individuellement est coûteuse.

41

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

➤ L'analyse amortie diffère de l'analyse du cas moyen au sens où on ne fait pas appel aux probabilités : une analyse amortie garantit les performances moyennes de chaque opération dans le cas **le plus défavorable**.

➤ L'analyse amortie permet de déterminer de façon plus fine la complexité en pire cas (ou dans le cas le plus défavorable) d'**une séquence d'opérations**, notamment dans le cas où les opérations n'ont pas un coût réel forcément constant.

On dispose de trois méthodes standards pour l'analyse amortie :

- La méthode de l'agrégat détermine un majorant pour le temps d'exécution de toute la séquence en pire cas.
- La méthode comptable affecte des coûts différents à chaque opération considérée et permet de mettre de côté du temps pour des opérations ultérieures plus coûteuses.
- La méthode du potentiel attribue un crédit global à toute la structure de données. Un tel crédit peut être dépensé ultérieurement dans d'autres opérations.

42

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

1/ Méthode de l'agrégat

Dans la méthode de l'agrégat, on montre que pour tout n , une suite de n opérations prend le temps $T(n)$ dans **le cas le plus défavorable**. Dans le cas le plus défavorable le coût moyen ou coût amorti, par opération est donc $T(n)/n$. Notez que le coût amorti s'applique à chaque opération, même quand il existe plusieurs types d'opérations dans la séquence.

Exemple : Opérations de pile

Les trois opérations fondamentales de pile sont :

- empiler(x) empile l'objet x sur la pile
- depiler() dépile le sommet de la pile
- pile_vide() voir si la pile est vide ou non

Le coût total d'une séquence de n opérations empiler et depiler vaut donc n , et le temps d'exécution réel pour n opérations est donc $O(n)$.

43

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

1/ Méthode de l'agrégat (suite)

On propose la nouvelle opération multidep qui retire les k premiers objets de la pile. Voici le pseudo-code de cette opération.

```
multidep( k)
    tant que (Non pile_vide( ) et k !=0 ) faire
        depiler( )
        k :=k-1
    finTQ
```

Quel est le temps d'exécution de multidep(k) sur une pile de s objets ?

→ $\min(s,k)$ c'est une fonction linéaire.

44

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

1/ Méthode de l'agrégat (suite)

Coût d'une séquence de n opérations

Le coût d'une séquence de n opérations empiler, depiler et multidep sur une pile initialement vide est analysé comme suit :

-Le coût dans le cas le plus défavorable d'une opération multidep dans la séquence est $O(n)$ puisque la taille de la pile est au plus égale à n.

-Le coût dans le cas le plus défavorable d'une opération de pile qlq est donc $O(n)$.

-Une suite de n opérations coûte $O(n^2)$ car on pourrait avoir $O(n)$ opérations multidep coûtant chacune $O(n)$.

-Bien que cette analyse soit correcte, le résultat $O(n^2)$, obtenu en considérant le coût le plus défavorable de chaque opération individuelle n'est pas assez fin.

45

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

1/ Méthode de l'agrégat (suite)

➤ Grâce à la méthode d'analyse par agrégat, on peut obtenir un meilleur majorant, qui prend en compte globalement la séquence n opérations. En fait, bien qu'une seule opération multidep soit potentiellement coûteuse, une suite de n opérations empiler, depiler et multidep sur une pile initialement vide **peut coûter au plus $O(n)$ (et non pas $O(n^2)$)**.

➤ Pourquoi ?

-Chaque objet peut être dépilé au plus une fois pour chaque empilement de ce même objet. Donc, le nombre de fois que depiler peut être appelée sur une pile non vide, y compris les appels effectués à l'intérieur de multidep vaut **au plus le nombre empiler**, qui lui-même **vaut au plus n**.

➔ Pour une valeur qlq de n, n'importe quelle suite de n opérations empiler, depiler et multidep prendra donc au total **$O(n)$** .

➔ Le coût moyen d'une opération est $O(n)/n=O(1)$.

➔ Dans cet exemple, chacune de trois opérations de pile a donc un **coût amorti $O(1)$** .

46

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

2/ Méthode comptable

➤ La méthode comptable consiste à affecter à **chaque opération** un **coût amorti** ;

➤ Si le **coût amorti** d'une opération est plus grand que son **coût réel**, alors le supplément est appelé **crédit** ;

➤ Lorsqu'on choisit les coûts amortis, on doit s'assurer que **toute suite de n opérations** donne **toujours** un coût amorti **supérieur** au coût réel :

$$\sum_{i=1}^n \hat{c}_i \geq \sum_{i=1}^n c_i, \quad \text{Avec } C_i \text{ est le coût réel et } \hat{C}_i \text{ est le coût amorti de la } i \text{ ème opération ;}$$

➤ Autrement dit, le crédit total doit être positif en tout temps

$$\sum_{i=1}^n \hat{c}_i - \sum_{i=1}^n c_i \geq 0$$

47

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

2/ Méthode comptable (suite)

Pour le cas de la SD Pile

➤ Les coûts réels des opérations sur une pile sont :

- Empiler (x) → $O(1)$
- Depiler () → $O(1)$
- multidep (k) → $O(\min(k, s))$ avec **s** est le nb d'objets de la pile et **k** est le nb d'objets qu'on souhaite dépiler

➔ On peut choisir comme coûts amortis les valeurs suivantes :

- Empiler (x) → $O(2)$
- Depiler () → $O(0)$
- multidep (k) → $O(0)$

➤ **Clairement**, le crédit total est toujours positif, puisqu'il est égal au nombre d'éléments dans la pile à chaque instant.

48

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

3/ Méthode du potentiel

- C'est la méthode la plus utilisée en pratique ;
- Une fonction de potentiel Φ est une fonction qui affecte à une **structure de données** un **nombre réel** ;
- Analogue au concept d'énergie potentielle ;
- Soient D_0, D_1, \dots une suite de **structures de données** obtenues par application successive d'opérations ;
- Le coût amorti \hat{c}_i de la i -ème opération par rapport à Φ est défini par:

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$

49

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

3/ Méthode du potentiel (suite)

- Le coût amorti total d'une suite de n opérations est

$$\begin{aligned} \sum_{i=1}^n \hat{c}_i &= \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0). \end{aligned}$$

- Si on choisit Φ de sorte que $\Phi(D_n) \geq \Phi(D_0)$, alors le coût amorti total est une borne supérieure du coût réel total ;

➔ En pratique, on a souvent $\Phi(D_0) = 0$ et il suffit de montrer que $\Phi(D_i) \geq 0$ pour tout i .

50

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

3/ Méthode du potentiel (suite)

Pour le cas de la SD Pile

➤ La fonction potentiel la plus simple est :

$\Phi(D_i)$ = nombre d'éléments dans la pile D_i .

➔ Clairement,

$\Phi(D_0) = 0$ quand on commence avec une pile vide et

$\Phi(D_i) \geq 0$ pour tout i .

➔ Il ne nous reste qu'à calculer le coût amorti de chaque opération.

51

Chapitre 2: Analyse d'algorithmes

VI. Analyse amortie

3/ Méthode du potentiel (suite)

Pour le cas de la SD Pile

Si la i ème opération est Empiler, alors le coût amorti est

$$\begin{aligned}\hat{C}_i &= C_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= 1 + (\Phi(D_{i-1}) + 1) - \Phi(D_{i-1}) \\ &= 1 + (|D_{i-1}| + 1) - |D_{i-1}| \\ &= 1 + 1 \\ &= 2\end{aligned}$$

Si la i ème opération est multidep (qui inclut Depiler quand $k = 1$), alors le coût amorti est

$$\begin{aligned}\hat{C}_i &= C_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &= k + (\Phi(D_{i-1}) - k) - \Phi(D_{i-1}) \\ &= k + (|D_{i-1}| - k) - |D_{i-1}| \\ &= k - k \\ &= 0\end{aligned}$$

52