Cours: Algorithmes et Structures des Données

Chapitre 10: La complexité des algorithmes

Réalisé par:
Dr. Sakka Rouis Taoufik

1

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 peut avoir plusieurs algorithmes : A_1, A_2, A_3, …, 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 10 : La Complexité des 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 10 : La Complexité des 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

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 algorithmique 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 et meilleur qu'un algorithme b et s'il est optimal ou s'il ne doit pas être utilisé. . .

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.

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 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 $tri_1(t)$ et $tri_2(t)$ 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.

III. Notion de croissance asymptotique

Nous intéressons aux fonctions de profil $N \rightarrow N$ ou $N \rightarrow R+$.

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**.

III. Notion de croissance asymptotique

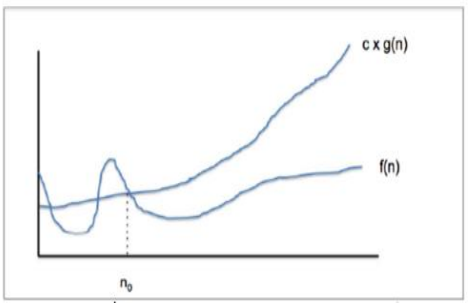
Définition 1 (Notation O omicron)

Soit g(n) une fonction de $N \rightarrow R^+$ 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'a 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:
$n^2 \in O(2^n+2n)$
$20n^2 \in O(3n^3+2n+1)$
$20n^2 \in O(20n^2+50n+5)$
$20n^2 \in O(19n^2)$
$n^2 \notin O(200n)$



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 n0 = 2

3) Si $f(n)=2n^2+3$ alors f(n)= **O**(n^2),
➔ prendre par exemple c=3 et n0 = 2

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

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

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

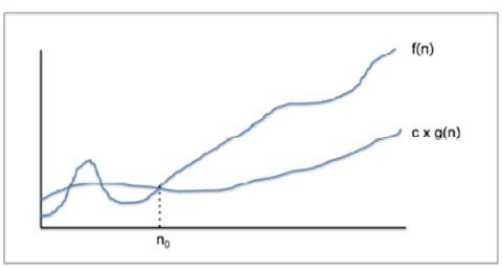
III. Notion de croissance asymptotique

Définition 2 (Notation Ω oméga)

Soit g(n) une fonction de $N \to R^+$ 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'a partir 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.



III. Notion de croissance asymptotique

Exercice sur la notation Ω oméga

Prouver que :

1) Si f(n)=4 alors f(n)= **Ω**(1),

2) Si f(n)=4n+2 alors f(n)= **Ω**(n)

3) Si f(n)=4n²+1 alors f(n)= **Ω**(n)

III. Notion de croissance asymptotique

Définition 3 (Notation *θ* théta)

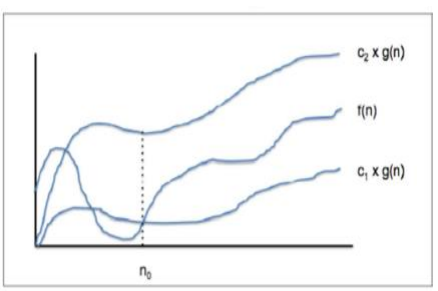
Soit g(n) une fonction de $N \rightarrow R^+$ On définit:

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

➔ g est équivalent (ou comparable) asymptotiquement à f

- A partir d'un certain rang n0
  la fonction f(n) peut être bornée inférieurement par c1 ***g(n);**
  la fonction f(n) peut être bornée supérieurement par c2* ***g(n);***

Exemples
$n^2 \notin \theta(2^n+2n)$
$20\ n^2 \notin \theta(3n^3+2n+1)$
$20\ n^2 \in \theta\ (20n^2+50n+5)$
$20\ n^2 \in \theta(19n^2)$
$n^2 \notin \theta(200n)$



III. Notion de croissance asymptotique

Exercice sur la notation θ théta: Prouver que :

1) Si f(n)=4 alors f(n) ∈ θ(1),

2) Si f(n)=4n+2 alors f(n) ∈ θ(n),

3) Si f(n)=4n²+1 alors f(n) ∈ θ(n²)

IV. Règles de calcul pour les algorithmes itératifs

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)**

IV. Règles de calcul pour les algorithmes itératifs

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 (T1+T2)=max(O(T1); O(T2))

Traitement 1 T1(n)
Traitement 2 T2(n)

IV. Règles de calcul pour les algorithmes itératifs

- **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

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

IV. Règles de calcul pour les algorithmes itératifs

➢ 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)= nombre d'itérations × [$O(T_{cond})$ + $O(T_{trait})$]

IV. Règles de calcul pour les algorithmes itératifs

Rq: Pour la complexité temporelle: On peut simplifier ces règles en comptant le nombre de répétitions de l'opération fondamentale (ou élémentaire).

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)

IV. Règles de calcul pour les algorithmes itératifs

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)

21

IV. Règles de calcul pour les algorithmes itératifs

L'opération élémentaire (T[i]==info) se répète :

- 1 fois : si info==T[0] : meilleur des cas (ou cas optimiste ou minimum) → la complexité temporelle est O(1)

- n/2 fois: si info==T[(n-1) /2] : cas moyen (ou cas réaliste) → la complexité temporelle est O(n/2)➔ O(n)

- n fois : si info==T[n-1] : pire des cas (ou cas pessimiste ou maximal) → la complexité temporelle est O(n)

22

V. Règles de calcul pour les algorithmes récursifs

- Méthode 1: par substitution

Étape 1 : relation de récurrence
Déterminer la relation de récurrence en fonction des termes d'ordre inférieur.

Étape 2 : relation en substituant
Développer la relation en substituant progressivement les termes d'ordre inférieur dans l'équation.

Étape 3 : démonstration
Prouver le résultat obtenu par récurrence.

$$T(n) = \begin{cases} a \text{ si } n = 1 \\ b + T(n-1) \text{ sinon} \end{cases}$$

$$\begin{aligned} T(n) &= b + T(n-1) \\ T(n-1) &= b + T(n-2) \\ T(n-2) &= b + T(n-3) \\ &\dots \\ T(2) &= b + T(1) \end{aligned}$$

➔ $T(n) = (n-1)b + a$

- Pour n = 1, T(n) = a, ce qui est V
- On suppose que le résultat est vrai jusqu'à l'ordre n – 1, on montre que c'est V pour n.

$$\begin{aligned} T(n) &= (n-1)\,b + a \\ &= nb + (b-a) \end{aligned}$$

a et b deux const ➔ **O(n)**

V. Règles de calcul pour les algorithmes récursifs

- Méthode 1: par substitution

```
unsigned facto (unsigned n) {
      if (n == 1|| n==0 )
            return 1 ;
      else
            return n*facto (n-1) ;
}
```

$$T(n) = \begin{cases} 1 & \text{si } n = 0 \text{ ou } n = 1 \\ T(n-1) + 2 & \text{si } n > 1 \end{cases}$$

// 1 seul appel récursif +
//opér de multiplication + return

$$\begin{aligned} T(n) &= T(n-1) + 2 \\ T(n-1) &= T(n-2) + 2 \\ T(n-2) &= T(n-3) + 2 \\ &\vdots \\ T(1) &= 1 \end{aligned}$$

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

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

- ❖ La fonction facto effectue un seul appel récursif et un nombre constant d'opérations à chaque appel.
- ❖ Sa relation de récurrence est T(n)=T(n−1)+2,
- ➔ Complexité temporelle linéaire O(n)

V. Règles de calcul pour les algorithmes récursifs

- Méthode 2: par Expansion (ou de déroulement)

— utilisée pour les récurrences à pas constant.

Uniquement quand la relation de récurrence est de la forme

$$T(n) = aT(n-b) + O(n^c \log^d n)$$

- a, b des constantes positives $a, b > 0$
- c, d des constantes positives $c, d \geq 0$

$$T(n) \in \begin{cases} O(n^c \log^d n) & 0 < a < 1 \\ O(n^{c+1} \log^d n) & a = 1 \\ O(a^{n/b}) & a > 1 \end{cases}$$

Exemple 1: T(n) = 2T(n - 1) + 1
a = 2; b = 1; c = 0; d = 0
a > 1; d = 0 ➔ cas 3
T(n) = $O(2^n)$

Exemple 2: T(n) = T(n- 1) + n
a = 1; b = 1; c = 1; d = 0
a = 1 ➔ cas 2
T(n) = $O(n^2)$

25

V. Règles de calcul pour les algorithmes récursifs

- Méthode 3: Master Theorem (1989)

Uniquement quand la relation de récurrence est de la forme
$T(n) = aT(\frac{n}{b}) + f(n)$

- 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 $b^c > a$: $T(n) \in O(f(n))$

26

V. Règles de calcul pour les algorithmes récursifs

- Méthode 3: Master Theorem (1989)

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

a = 4; b = 2; c = 1
$b^c = 2^1 = 2$ < a ➔ cas 1
$T(n) = O(n^2)$

Exemple 2: T(n) = 3T(n/2)+ n
a = 3; b = 2; c = 1
$b^c = 2^1 = 2$ < a ➔ cas 1
$T(n) = O(n^{\log_2(3)})$

Exemple 3: T(n) = 2T(n/2)+ n
a = 2; b = 2; c = 1
$b^c = 2^1 = 2$ = a ➔ cas 2
$T(n) = O(n\log_2(n))$

Exemple 4: T(n) = 2T(n/2)+ n^2
a = 2; b = 2; c = 2
$b^c = 2^2 = 4$ > a ➔ cas 3
$T(n) = O(n^2)$

27

V. Règles de calcul pour les algorithmes récursifs

- Méthode 4: Akra-Bazzi Theorem (1998)

Généralisation de la méthode Master pour les relations de récurrence de la 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 $\sum_{i=1}^{k} b_i^x = 1$

28

V. Règles de calcul pour les algorithmes récursifs

- Méthode 4: Akra-Bazzi Theorem (1998)

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

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

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

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

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

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

29

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}

30

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 $T_{min} = T_{max} = T_{moy} = (n^2)$ où n est la taille du tableau à trier.

VII. Classes de complexité

- **O(1) : complexité constante,** pas d'augmentation du temps d'exécution quand le paramètre croit
- **O(log(n)) : complexité logarithmique**, augmentation très faible du temps d'exécution quand le paramètre croit. *Exemple : algorithmes qui décomposent un problème en un ensemble de problèmes plus petits (dichotomie).*
- **O(n) : complexité linéaire**, augmentation linéaire du temps d'exécution quand le paramètre croit (si le paramètre double, le temps double). *Exemple : algorithmes qui parcourent séquentiellement des structures linéaires.*
- **O(nlog(n)) : complexité quasi-linéaire**, augmentation un peu supérieure à O(n). *Exemple : algorithmes qui décomposent un problème en d'autres plus simples, traités indépendamment et qui combinent les solutions partielles pour calculer la solution générale.*

VII. Classes de complexité

- **$O(n^2)$: complexité quadratique,** quand le paramètre double, le temps d'exécution est multiplié par 4. Cette complexité est acceptable uniquement pour des données de petite taille. *Exemple : algorithmes avec deux boucles imbriquées.*
- **$O(n^i)$: complexité polynomiale,** quand le paramètre double, le temps d'exécution est multiplié par 2^i. Un algorithme utilisant i boucles imbriquées est polynomial. *Exemple : algorithme utilisant i boucles imbriquées.*
- **$O(i^n)$: complexité exponentielle,** quand le paramètre double, le temps d'exécution est élevé à la puissance 2.
- **O(n!) : complexité factorielle,** asymptotiquement équivalente à n^n. Un algorithme de complexité factorielle ne sert à rien.

Relations asymptotiques entre les complexités

VIII. Exercises d'application

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 = (premier + 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 [premier..M-1]

x > T[M] : l'élément x, s'il existe, se trouve dans l'intervalle [M+1..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.

VIII. Exercises d'application

1) Réalisation

VIII. Exercises d'application

2) Complexité

-Complexité en temps :

On va comptabiliser l'opération de comparaison?

On distingue les cas suivant :

-Cas minimum ou optimiste: une seul 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 x n'appartient pas à nom. 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 initiale est divisé en deux sous problème disjoints et de taille plus ou moins égale.

VIII. Exercises d'application

2) Complexité

Supposant que N est le nombre d'éléments

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

➔$C_N=C_{N/2}+1$

avec $C_1=1$

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

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

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

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

…

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

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

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

37

VIII. Exercise d'application

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

38

VIII. Exercises d'application

Exercice 2:

Quelle est la complexité de la méthode suivante ?

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


VIII. Exercises d'application

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]

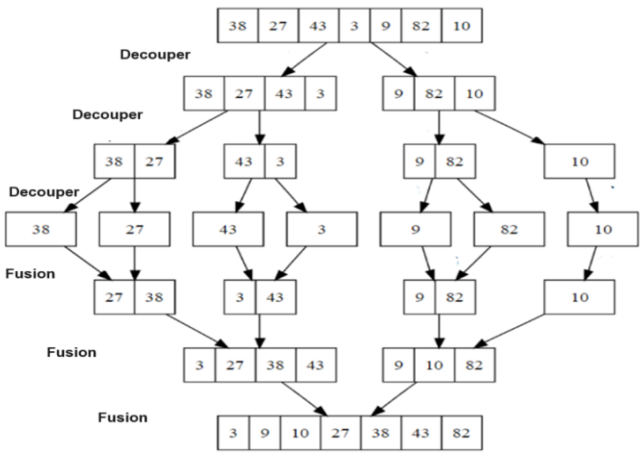
- Trier les deux sous-tableaux T[1, .. n/2] et T[n/2 +1,..n] (récursivement, ou on ne fait rien s'ils sont de taille 1)

-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.

VIII. Exercises d'application

Illustration de l'algorithme de **trie par fusion** est :



VIII. Exercises d'application

1) 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) ;
        }
}
void tri (int T [ ], int n){
        TriFusion( T, , n-1) ;
}
```

VIII. Exercises d'application

```
void fusion (int tab [ ], int g, int m, int d) {
        int n1 = m - g + 1, n2 = d - m;
        int i, j, k;
         int T1 [n1],  T2 [n2] ;
        for (i = 0; i < n1; i++)
                T1[i] = tab[g + i];
        for (j = 0; j < n2; j++)
                T2[j] = tab[m + 1 + j];
 /* maintient trois pointeurs, un pour chacun des deux tableaux et un pour
    maintenir l'index actuel du tableau trié final     */
         i = 0;       j = 0;       k = g;
        while (i < n1 && j < n2)
            if (T1[i] <= T2[j])
                tab[k++] = T1[i++];
           else
                tab[k++] = T2[j++];
    /* tab[k++]= (T1[i] <= T2[j]) ? T1[i++]:T2[j++]; */
 // Copiez tous les éléments restants du tableau non vide
        while (i < n1)          tab[k++] = T1[i++];
        while (j < n2)          tab[k++] = T2[j++];
}
```


VIII. Exercises d'application

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.

➔ $C_N = 2 * C_{N/2} + N$

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

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

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

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

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

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

$= 2^2 * (2*C_{2^{n-3}} + 2^{n-2}) + 2 * 2^n$

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

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

…

$= 2^n * C_{2^{n-n}} + n * 2^n = 0 + (\log_2 N) * N$

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