

Université de Monastir

# Cours: Conception et Analyse d'Algorithmes

## Chapitre 1: La récursivité (Parie 3)

Réalisé par:

Dr. Sakka Rouis Taoufik

1

### Chapitre 1: La récursivité

#### I. Le paradigme diviser pour régner

Le paradigme **diviser pour régner** (ou diviser pour résoudre) est une technique algorithmique qui consiste à :

**Diviser** : découper un problème initial en sous-problèmes ;

**Régner** : résoudre les sous-problèmes (récursivement ou directement s'ils sont assez petits) ;

**Combiner** : calculer une solution au problème initial à partir des solutions des sous-problèmes.

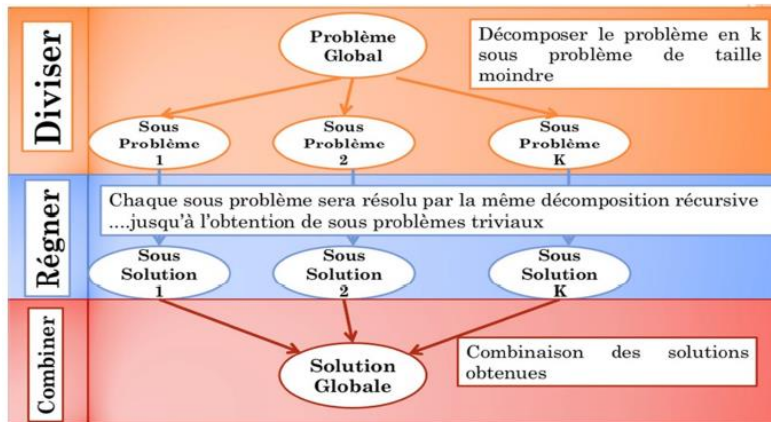
Cette technique fournit des algorithmes efficaces pour de nombreux problèmes.

**NB.** Les algorithmes basés sur le paradigme *diviser pour régner* sont souvent adaptés pour être exécutés sur des machines avec **plusieurs processeurs**.

2

## Chapitre 1: La récursivité

### I. Le paradigme diviser pour régner



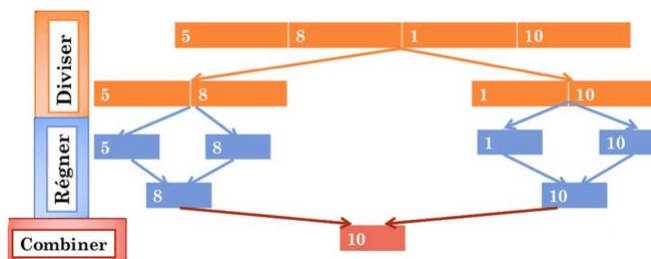
3

## Chapitre 1: La récursivité

### I. Le paradigme diviser pour régner

#### Exemple 1: Recherche le Maximum

Soit T un tableau à n éléments, écrire une fonction récursive permettant de rechercher l'indice du maximum dans T en utilisant le paradigme diviser pour régner!



4

## Chapitre 1: La récursivité

### I. Le paradigme diviser pour régner

#### Solution de l'exemple 1:

5

## Chapitre 1: La récursivité

### I. Le paradigme diviser pour régner

#### Exemple 2

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.

Proposer une implémentation récursive et une autre itérative pour cette méthode.

6

## Chapitre 1: La récursivité

### I. Le paradigme diviser pour régner

**Solution de l'exemple 2:**

7

## Chapitre 1: La récursivité

### II. Complexité de qq algo basés sur ce paradigme

**Cas de l'Algo de Rech Dichotomique.**

**-Complexité en temps :**

On va comptabiliser l'opération de comparaison de  $x$  avec  $T[m]$

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 au tableau. 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.

8

## Chapitre 1: La récursivité

### II. Complexité de qq algo basés sur ce paradigme

Supposant que  $N$  est le nombre d'éléments,  
On note  $C_N$  : le nombre de fois où la comparaison 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  $\log_2 N$

9

## Chapitre 1: La récursivité

### II. Complexité de qq algo basés sur ce paradigme

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

10

## Chapitre 1: La récursivité

### III. Exercice d'application

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.

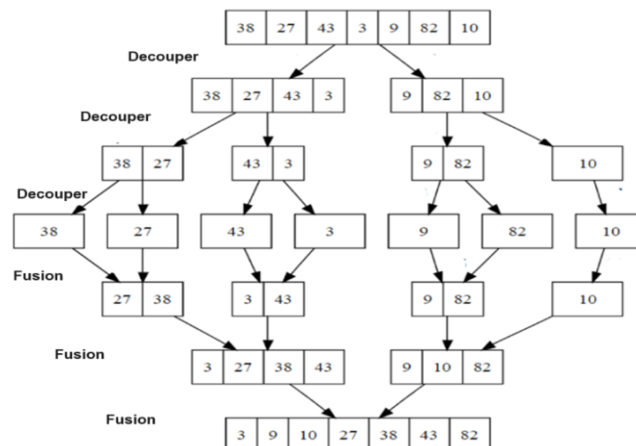
- 1) Proposer une implémentation récursive pour cette technique
- 2) Déterminer sa complexité.

11

## Chapitre 1: La récursivité

### III. Exercice d'application

Illustration de l'algo de trie par fusion



12

## Chapitre 1: La récursivité

### III. Exercice d'application

#### Solution

```
#include <stdio.h>

void fusion (int T [ ], int g, int m, int d);

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, 0, n-1) ;
}
```

13

## Chapitre 1: La récursivité

### III. Exercice d'application

#### Solution

```
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 compteurs, 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++];
}
```

14