Diviser pour Régner

Dr. Mahfoud Houari
mahfoud.houari@gmail.com
Université Abou-Bakr Belkaïd - Tlemcen

2022/2023

Avant de commencer

1. On sait c'est quoi un algorithme efficace *MAIS comment le concevoir* ?

2. Une conception naïve mène souvent à des résultats non satisfaisants (en termes de temps et/ou de mémoire).

3. L'utilisation d'un *paradigme de conception d'algorithmes* peut assurer un certain degré d'efficacité.

4. Les fameux paradigmes de conception :

- *a) Recherche exhaustive*
- *b) Diviser pour Régner*
- *c) Programmation Dynamique*
- *d) Algorithmes Gloutons*

5. Quel**s** paradigme**s** pour quel problème ?

Paradigme « *Diviser pour Régner* »

(*Divide and Conquer*)

Principe :

1. Utilisé par la majorité des algorithmes récursifs.

2. Il est basé sur trois étapes :

a) **Diviser :** le problème initial en sous-problèmes de tailles plus petites.

b) **Résoudre :** les sous-problèmes *directement* ou *récursivement*.

c) **Combiner :** les solutions des sous-problèmes pour produire la solution du problème initial.

3. Certains problèmes ne demandent pas de combinaisons.

4. Le nombre de sous-problèmes est arbitraire.

5. Des techniques de parallélisme peuvent être appliquées pour résoudre les sous-problèmes indépendants.

Exemple 1 : La « Recherche dichotomique »

➡ **Chercher la valeur** ***-2*** **:**

-3	1	2	4	5	6	7	8	9

-2 < 5

-3	1	2	4

-2 < 1

-3

-2 ≠ -3 ➡ **Élément** ***-2*** **n'existe pas**

Remarque : Pas de combinaisons de sous-solutions.

Exemple 1 : La « Recherche dichotomique »

Calculer la WCTC et WCSC de la recherche dichotomique ?

```
public boolean Dicho(int[]  T, int E, int D, int F) {
    if(D > F)                                   1
        return false ;                          1
    int M = (D + F)/2 ;                         4
    if(T[M] == E)                               2
        return true ;                           1
    else if(T[M] > E)                           2
        return Dicho(T, E, D, M - 1) ;          2 + C_|T|/2
    else
        return Dicho(T, E, M + 1, F)            2 + C_|T|/2
}
```

WCTC : $11 + C_{|T|/2} = 11 + 11 + C_{|T|/4} = \ldots\ldots = 11.\log_2(|T|)$

Exemple 1 : La « Recherche dichotomique »

Calculer la WCTC et WCSC de la recherche dichotomique ?

```
public boolean Dicho(int[]  T, int E, int D, int F) {          4
    if(D > F)
        return false ;
    int M = (D + F)/2 ;                                         1
    if(T[M] == E)
        return true ;
    else if(T[M] > E)
        return Dicho(T, E, D, M - 1) ;                          C_|T|/2
    else
        return Dicho(T, E, M + 1, F)                            C_|T|/2
}
```

WCSC : $5 + C_{|T|/2} = 5 + 5 + C_{|T|/4} = \ldots\ldots = 5.\log_2(|T|)$.

Exemple 1 : La « Recherche dichotomique »

Calculer la WCTC et WCSC de la recherche dichotomique ?

```
public boolean Dicho(int[]  T, int E, int D, int F) {
    if(D > F)
        return false ;
    int M = (D + F)/2 ;
    if(T[M] == E)
        return true ;
    else if(T[M] > E)
        return Dicho(T, E, D, M - 1) ;
    else
        return Dicho(T, E, M + 1, F)
}
```

⇨ **<u>WCTC :</u> O(log$_2$(|T|)).**

⇨ **<u>WCSC :</u> O(log$_2$(|T|)).**

Réflexion autour de la recherche :

Étant donné que $\mathbf{log_3}(|T|) < \mathbf{log_2}(|T|)$ pourquoi pas opter plutôt pour une recherche trichotomique ce qui consiste à diviser le tableau en trois parties au lieu de deux.

La complexité d'une recherche trichotomique est bornée par $\mathbf{O}(\mathbf{log_3}(|T|))$ et donc légèrement meilleure par rapport à sa voisine dichotomique.

Donc, peut-on faire la conclusion suivante ?

Plus que la fraction de division augmente
plus que l'efficacité de la recherche est meilleure

En pratique non, plus que la fraction de division augmente, plus que le nombre de calculs par appel récursif augmente, et donc on s'approche de $\mathbf{O}(|T|)$.

Exemple 2 : La « Hauteur d'un Arbre »

Calculer la WCTC et WCSC de la méthode suivante en supposant que l'arbre binaire est équilibré et composé de *N nœuds* ?

```
public int hauteur(Arbre A) {
    if ((A == null) || (A.fg == null && A.fd == null))
        return 0 ;
    else
        return 1 + Math.max(hauteur(A.fd), hauteur(A.fg)) ;
}
```

WCTC : O(?????).

WCSC : O(?????).

Remarque : Toutes les étapes du paradigme sont présentes.

Exemple 2 : La « Hauteur d'un Arbre »

Calculer la WCTC et WCSC de la méthode suivante en supposant que l'arbre binaire est équilibré et composé de *N nœuds* ?

```
public int hauteur(Arbre A) {
    if ((A == null) || (A.fg == null && A.fd == null))
        return 0 ;
    else
        return 1 + Math.max(hauteur(A.fd), hauteur(A.fg)) ;
}
```

WCTC : $13 + C_{fd} + C_{fg} = 13+(13+C_{fd.fd}+C_{fg.fd}) + (13+C_{fd.fg}+C_{fg.fg}) = 13.N$

***Attention :* les feuilles nécessitent moins de 13 instructions, mais 13.N reste une borne acceptable.**

Exemple 2 : La « Hauteur d'un Arbre »

Calculer la WCTC et WCSC de la méthode suivante en supposant que l'arbre binaire est équilibré et composé de *N nœuds* ?

```
public int hauteur(Arbre A) {
    if ((A == null) || (A.fg == null && A.fd == null))
        return 0 ;
    else
        return 1 + Math.max(hauteur(A.fd), hauteur(A.fg)) ;
}
```

WCSC : $3 + C_{fd} + C_{fg}$ = **Est-ce Juste ?**

Exemple 2 : La « Hauteur d'un Arbre »

Calculer la WCTC et WCSC de la méthode suivante en supposant que l'arbre binaire est équilibré et composé de *N nœuds* ?

```
public int hauteur(Arbre A) {
    if ((A == null) || (A.fg == null && A.fd == null))
        return 0 ;
    else
        return 1 + Math.max(hauteur(A.fd), hauteur(A.fg)) ;
}
```

WCSC : $3 + C_{fd} + C_{fg}$ = **Est-ce Juste ?** **NON**

Attention : ***les deux appels récursifs sont consécutifs et donc on ne considère que le coût d'un seul appel vu que l'espace mémoire est libéré après chaque appel.***

Exemple 2 : La « Hauteur d'un Arbre »

Calculer la WCTC et WCSC de la méthode suivante en supposant que l'arbre binaire est équilibré et composé de *N nœuds* ?

```
public int hauteur(Arbre A) {
    if ((A == null) || (A.fg == null && A.fd == null))
        return 0 ;
    else
        return 1 + Math.max(hauteur(A.fd), hauteur(A.fg)) ;
}
```

WCSC : $3 + C_{fd} = 3 + 3 + C_{fd.fd} = \ldots\ldots = 3.\log_2(N)$

Exemple 2 : La « Hauteur d'un Arbre »

Calculer la WCTC et WCSC de la méthode suivante en supposant que l'arbre binaire est équilibré et composé de *N nœuds* ?

```
public int hauteur(Arbre A) {
    if ((A == null) || (A.fg == null && A.fd == null))
        return 0 ;
    else
        return 1 + Math.max(hauteur(A.fd), hauteur(A.fg)) ;
}
```

➡ **WCTC : O(N).**

➡ **WCSC : O($\log_2$(N)).**

Exemple 2 : La « Hauteur d'un Arbre »

Calculer la WCTC et WCSC de la méthode suivante en supposant que l'arbre binaire est équilibré et composé de N *nœuds* ?

```
public int hauteur(Arbre A) {
    if ((A == null) || (A.fg() == null && A.fd() == null))
        return 0 ;
    else {
        int hg = hauteur(A.fg) ;
        int hd = hauteur(A.fd) ;
        return 1 + Math.max(hg, hg) ;
    }
}
```

WCSC : **5.log2(N) ???**

Exemple 2 : La « Hauteur d'un Arbre »

Calculer la WCTC et WCSC de la méthode suivante en supposant que l'arbre binaire est équilibré et composé de *N nœuds* ?

```
public int hauteur(Arbre A) {
    if ((A == null) || (A.fg() == null && A.fd() == null))
        return 0 ;
    else {
        int hg = hauteur(A.fg) ;
        int hd = hauteur(A.fd) ;
        return 1 + Math.max(hg, hg) ;
    }
}
```

WCSC : ~~5~~ .log2(N) ??? NON

Exemple 2 : La « Hauteur d'un Arbre »

Calculer la WCTC et WCSC de la méthode suivante en supposant que l'arbre binaire est équilibré et composé de *N nœuds* ?

```
public int hauteur(Arbre A) {
    if ((A == null) || (A.fg() == null && A.fd() == null))
        return 0 ;
    else {
        int hg = hauteur(A.fg) ;
        int hd = hauteur(A.fd) ;
        return 1 + Math.max(hg, hg) ;
    }
}
```

WCSC : $3.\log2(N) + 2 \Rightarrow O(\log_2(N))$

Exemple 3 : « *Chercher un élément dans un tableau non trié* »

Algorithme naïf:

```
public boolean Existe(int[] T , int E) {
    for(int i = 0 ; i < T.length ; i ++ ){
        if(T[i] == E) return true ;
    }
    return false ;
}
```

WCTC = O(|*T*|). WCSC = O(*1*).

Exemple 3 : « *Chercher un élément dans un tableau non trié* »

Algorithme DpR:

```
public boolean ExisteDpR(int[] T, int E, int D, int F){
    if(D > F )
        return false ;
    else {
        int M = (D + F)/2 ;
        if(T[M] == E)
            return true ;
        else
            return ExisteDpR(T, E, D, M-1) ||
                            ExisteDpR(T, E, M+1, F) ;
    }
}
```

WCTC $= O(|T|)$. **WCSC** $= O(\log_2 |T|)$.

Exemple 3 : « *Chercher un élément dans un tableau non trié* »

Conclusions :

- Ce n'est pas tout le temps possible d'améliorer un algorithme naïf en utilisant la stratégie *Diviser Pour Régner*.

- L'algorithme *DpR* assure au pire des cas la même efficacité en revanche il permettra d'appliquer du parallélisme ce qui peut le rendre efficace.

Exemple 4 : *Problème de tri – Tri naïf*

Algorithme naïf:

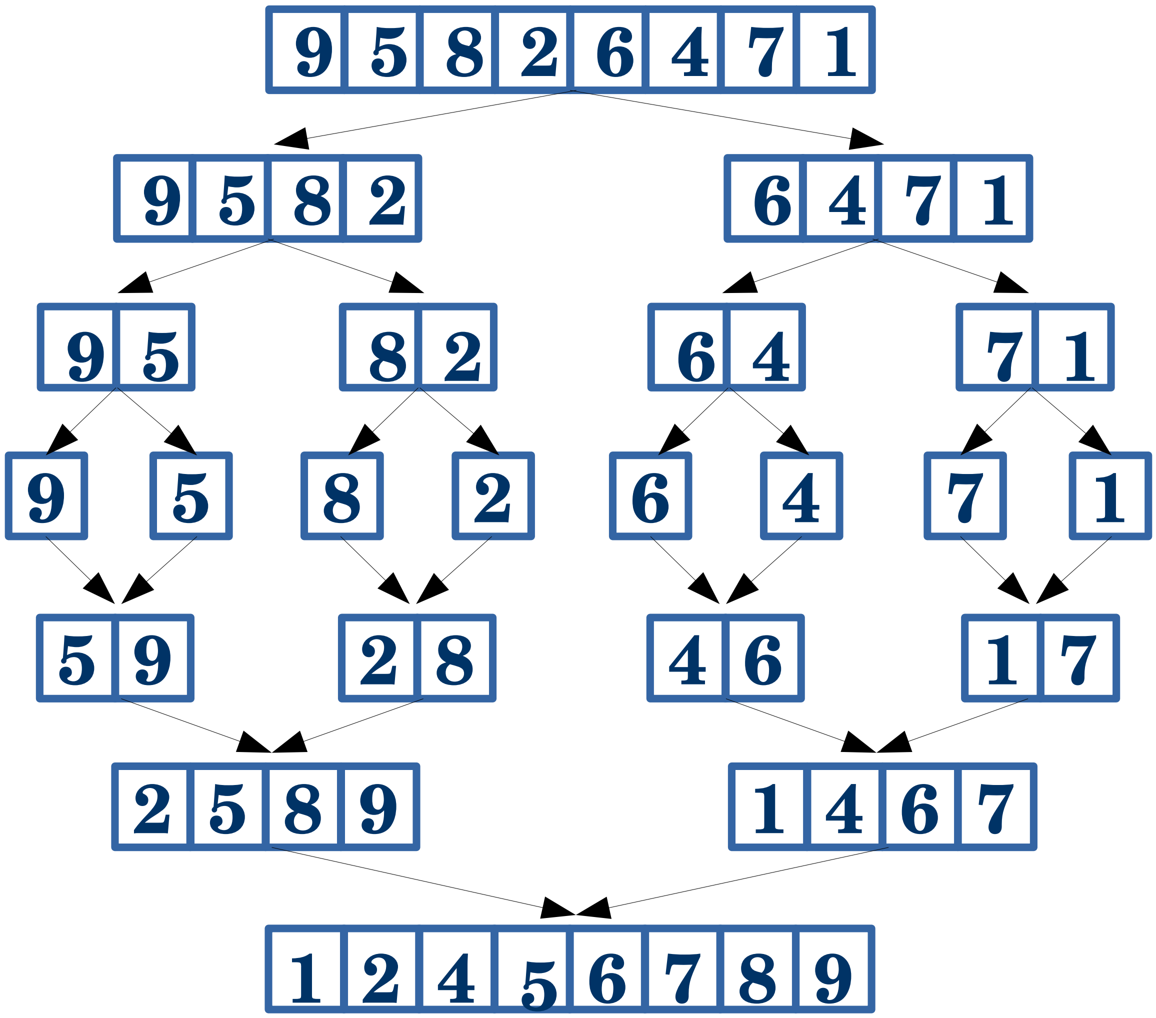
```
public int[] triNaif (int[] T){
    int tmp ;
    for(int i=0 ; i < T.length - 1; i++ ){
        for(int j=i+1 ; j < T.length ; j++ ){
            if(T[j] < T[i]){
                tmp = T[i] ; T[i] = T[j] ; T[j] = tmp ;
            }
        }
    }
    return T ;
}
```

- **Trouver la WCTC et WCSC de cette méthode ?**

  WCTC = $O(|T|^2)$ et WCSC = $O(1)$

- **Peut-on faire mieux ?**

Exemple 4 : *Problème de tri – Tri par Fusion*



Remarque : Les trois étapes du paradigme sont toutes présentes.

Exemple 4 : *Problème de tri – Tri par Fusion*

Algorithme Diviser-pour-Régner:

```
public int[] triFusion(int[] T){
    if( T.length <= 1 )
            return T;
    else {
      int [] TG = new int [T.length / 2];
      int [] TD = new int [T.length - T.length / 2];
      int i = 0 ,  j = 0 ;
      while (i < T.length/2) {
            TG[i] = T[i] ; i++
      }
      while (i < T.length) {
            TD[j] = T[i] ; i++ ; j++;
      }
      return fusionner(triFusion(TG), triFusion(TD));
    }
}
```

Exemple 4 : *Problème de tri – Tri par Fusion*

Algorithme Diviser-pour-Régner:

```
public int[] fusionner(int[] T1, int[] T2){
    int [] F  = new int[T1.length + T2.length] ;
    int i = 0, i1 = 0, i2 = 0 ;
    while (i < F.length){
            if( i1 < T1.length && i2 < T2.length){
                if(T1[i1] < T2[i2])     {
                    F[i] = T1[i1] ; i++ ; i1++ ;
                }
                else {
                    F[i] = T2[i2] ; i++ ; i2++ ;
                }
            } else if( i1 < T1.length){
                F[i] = T1[i1] ; i++ ; i1++ ;
            }  else {
                F[i] = T2[i2] ; i++ ; i2++ ;
            }
    }
    return F ;
}
```

Exercice : **Vérifier que la WCTC de la fonction *fusionner* est bornée par O(|T1| + |T2|).**

Exemple 4 : *Problème de tri – Tri par Fusion*

Analyse de la WCTC :

public int[] *triFusion*(int[] *T*){	
if(*T*.*length* <= 1)	**2**
return *T*;	**1**
else {	
int [] *TG* = new int [*T*.*length* / 2];	**5 + |T|/2**
int [] *TD* = new int [*T*.*length* - *T*.*length* / 2];	**8 + |T|/2**
int ***i*** = 0 , ***j*** = 0 ;	**4**
while (***i*** < *T*.*length*/2) {	**3.|T|/2 + 3**
TG[***i***] = *T*[***i***] ; ***i***++	**5.|T|/2**
}	
while (***i*** < *T*.*length*) {	**|T| + 4**
TD[***j***] = *T*[***i***] ; ***i***++ ; ***j***++;	**7.|T|/2 + 7**
}	
return *fusionner*(*triFusion*(*TG*), *triFusion*(*TD*));	**1 + O(|T|)**
}	
}	

Exemple 4 : *Problème de tri – Tri par Fusion*

Analyse de la WCTC :

```
public int[] triFusion(int[] T){
    if( T.length <= 1 )                                  O(1)
            return T;                                    O(1)
    else {
      int [] TG = new int [T.length / 2];                O(|T|)
      int [] TD = new int [T.length - T.length / 2];     O(|T|)
      int i = 0 , j = 0 ;                                O(1)
      while (i < T.length/2) {                           O(|T|)
            TG[i] = T[i] ; i++                           O(|T|)
      }
      while (i < T.length) {                             O(|T|)
            TD[j] = T[i] ; i++ ; j++;                    O(|T|)
      }
      return fusionner(triFusion(TG), triFusion(TD));    O(|T|)
    }
}
```

Exemple 4 : *Problème de tri – Tri par Fusion*

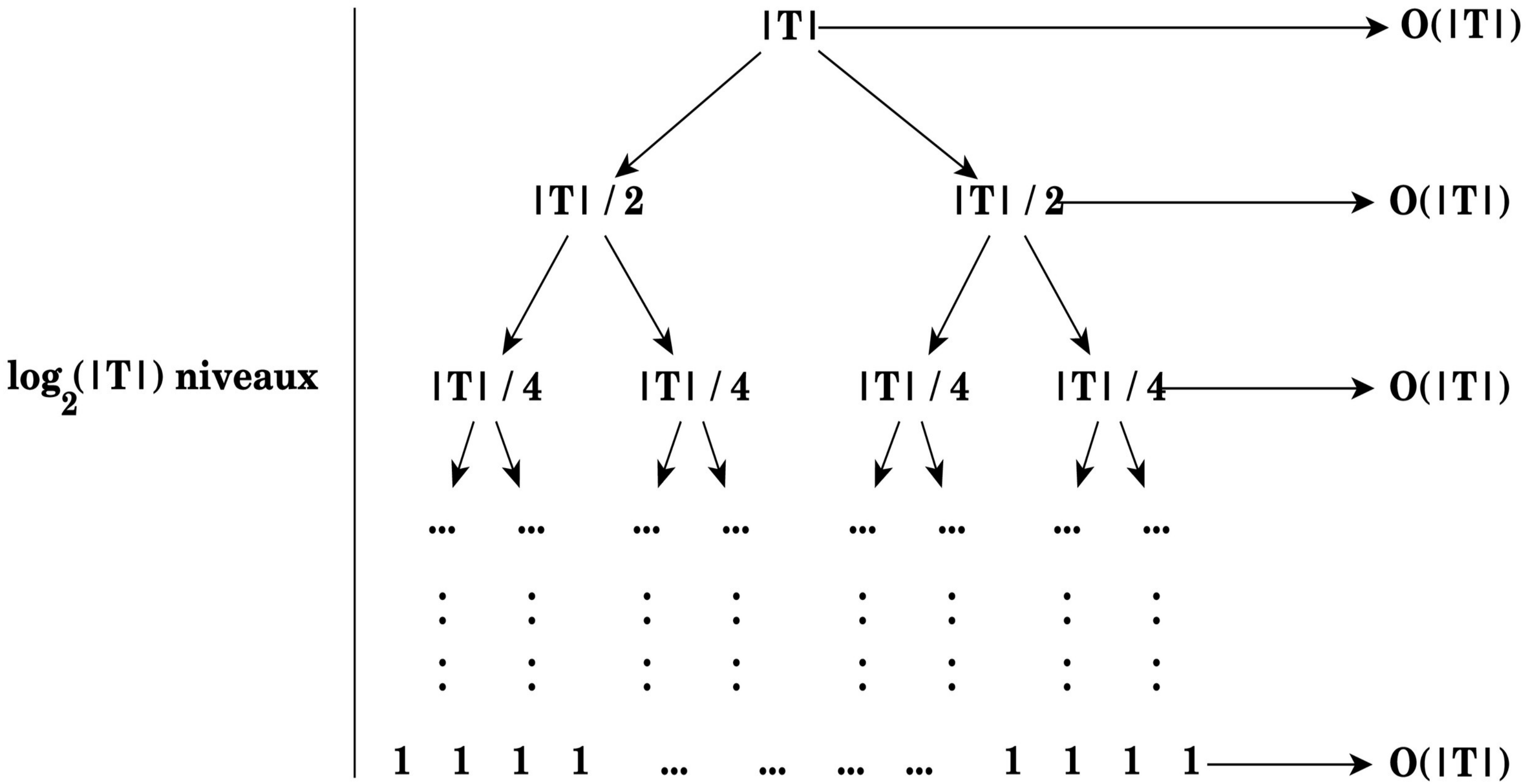
Analyse de la WCTC :

```
public int[] triFusion(int[] T){
    if( T.length <= 1 )                                    O(1)
            return T;                                      O(1)
    else {
      int [] TG = new int [T.length / 2];                  O(|T|)
      int [] TD = new int [T.length - T.length / 2];       O(|T|)
      int i = 0 , j = 0 ;                                  O(1)
      while (i < T.length/2) {                             O(|T|)
            TG[i] = T[i] ; i++                             O(|T|)
      }
      while (i < T.length) {                               O(|T|)
            TD[j] = T[i] ; i++ ; j++;                      O(|T|)
      }
      return fusionner(triFusion(TG), triFusion(TD));      O(|T|)
    }
}
```

D'où, chaque appel récursif avec un tableau de taille $|T|$ nécessite un coût borné par $O(|T|)$.

Exemple 4 : *Problème de tri – Tri par Fusion*

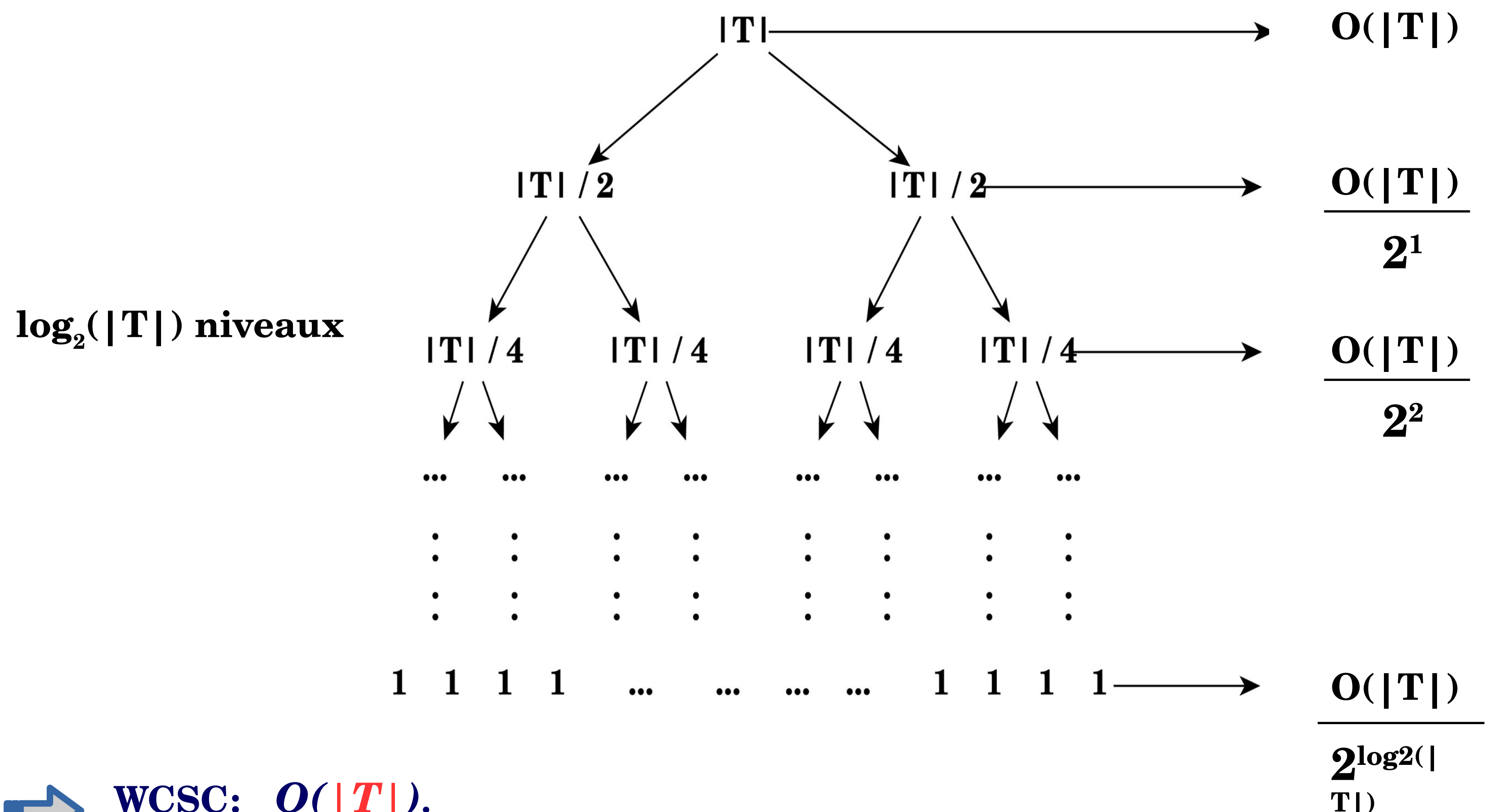
Analyse de la WCTC :



WCTC: $O(|T|.log_2(|T|))$.

Exemple 4 : *Problème de tri – Tri par Fusion*

Analyse de la WCSC :



⇨ **WCSC:** ***O(|T|).***

Exemple 4 : *Problème de tri – Tri par Fusion*

Analyse de la WCSC :

Peut-on réduire la WCSC du Tri par Fusion à :

- $O(1)$?
- $O(\log_2(|T|))$?

Exemple 5 : *Recherche dans une matrice carrée*

⇒ Une méthode Diviser-pour-Régner (*Version 1*) :

Méthode DpR :

```
boolean ExisteDpR(int [][] A, int N, int E){
      if(N == 1) return A[0][0] ==  E ;
      int[][] A1 = new int[N/2][N/2] ;
      int[][] A2 = new int[N/2][N/2] ;
      int[][] A3 = new int[N/2][N/2] ;
      int[][] A4 = new int[N/2][N/2] ;
      for(int i=0 ; i< N/2 ; i++){for (int j=0 ; j<N/2 ; j++){   A1[i][j] = A[i][j] ; }}
      for(int i=0 ; i< N/2 ; i++){for (int j=N/2 ; j<N ; j++){  A2[i][j] = A[i][j] ; }}
      for(int i=N/2 ; i< N ; i++){for (int j=0 ; j<N/2 ; j++){  A3[i][j] = A[i][j] ; }}
      for(int i=N/2 ; i< N ; i++){for (int j=N/2 ; j<N ; j++){ A4[i][j] = A[i][j] ; }}

      return  ExisteDpR(A1, N/2, E) || ExisteDpR(A2, N/2, E) ||
              ExisteDpR(A3, N/2, E) || ExisteDpR(A4, N/2, E) ;
}
```

WCTC: $O(N^2 . \log_2(N))$

Exemple 5 : *Recherche dans une matrice carrée*

⇒ **Une méthode Diviser-pour-Régner (*Version 2*) :**

Méthode DpR :

```
boolean Existe(int [][] A, int E, int DL, int FL, int DC, int FC){
      if(DL > FL || DC > FC) return false;

      if(DL == FL && DC == FC) return A[DL][DC] == E ;

      return  Existe(A, E, DL, (FL – DL)/2, DC, (FC – DC)/2)            //A1
              ||  Existe(A, E, DL, (FL- DL)/2, (FC-DC)/2+1, FC)          //A2
              ||  Existe(A, E, (FL-DL)/2+1, FL, DC, (FC-DC)/2)           //A3
              ||  Existe(A, E, (FL-DL)/2+1, FL, (FC-DC)/2+1, FC)         //A4
}
```

WCTC: $O(4^{\log_2(N)}) = O(N^2)$.