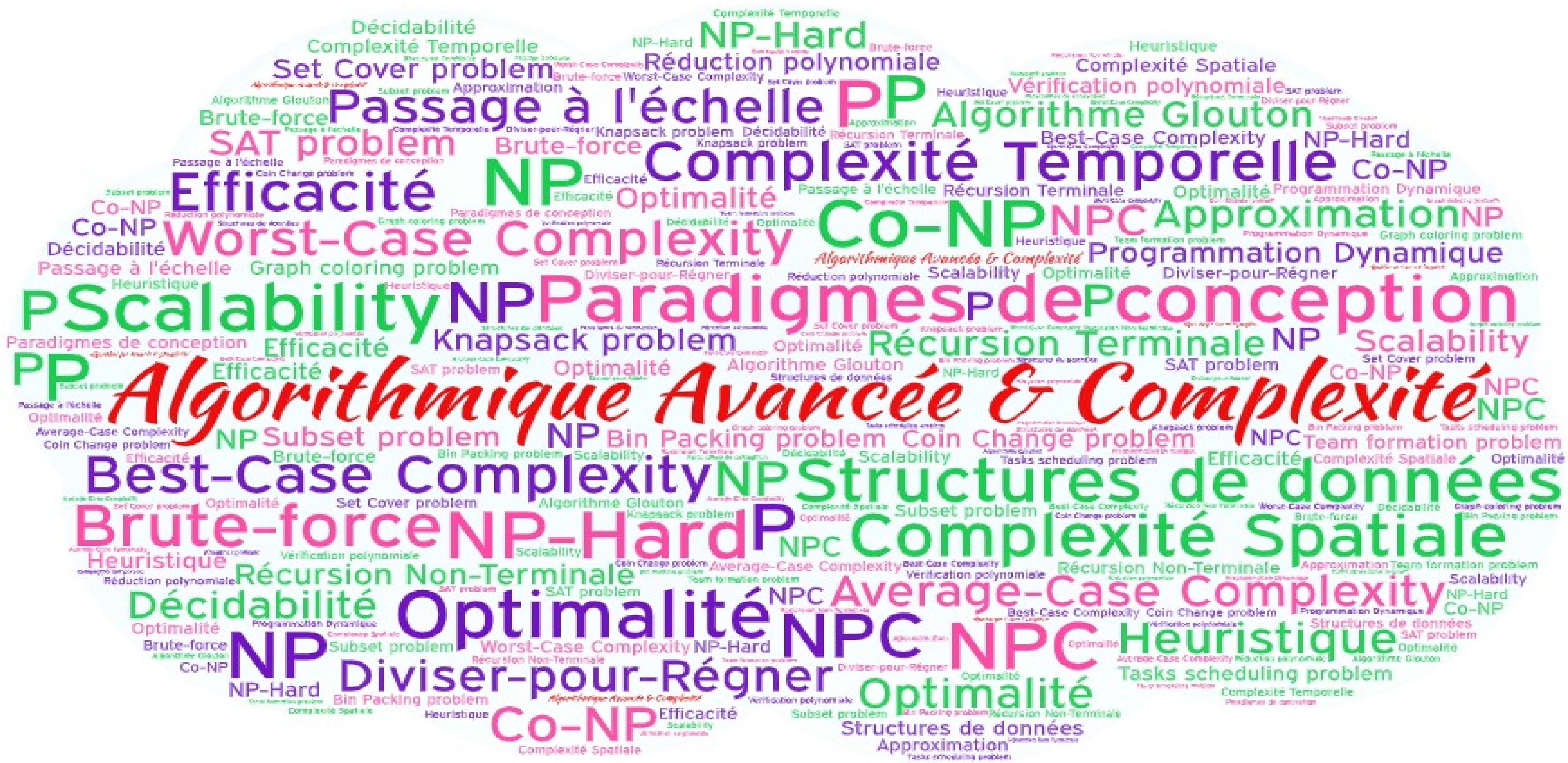

Diviser pour Régner
(Suite)

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

2022/2023

Exemple : *Union des Intervalles*

Étant donné un tableau ***T*** contenant des intervalles triés selon leur début, l'objectif est de trouver l'union de tous ces intervalles si elle existe. Par exemple :

Pour ***T***={**[1,5], [3,8], [7,13],[13,21]**}, le résultat doit être **[1, 21]**.
Pour ***T***={**[9,13],[13,21]**}, le résultat doit être **NULL**.

À faire :

1. Proposez une méthode itérative pour ce problème et calculez sa *WCTC* et sa *WCSC*.

2. Proposez une méthode *Diviser-pour-Régner* pour ce problème et calculez sa *WCTC* et sa *WCSC*.

Exemple : *Union des Intervalles*

Méthode naïve :

```
public static Intervalle UI_Naive(Intervalle[] T) {
    if(T.length == 0)
            return null;

    if(T.length == 1)
        return T[0];
    Intervalle resultat = new Intervalle(T[0].g, T[0].d);

    for(int i=1; i < T.length; i++) {
        if(T[i].unionWith(resultat) == null)
            return null;
        else
            resultat = T[i].unionWith(resultat);
    }
    return resultat;
}
```

WCTC : O(|T|)
WCSC : O(1)

Exemple : *Union des Intervalles*

Méthode DpR – *Mesure de la WCTC*

```
public static Intervalle UI_DpR(Intervalle[] T, int D, int F) {
      if(D > F)                                              1
         return null;                                        1
      if(D == F)                                             1
         return T[D];                                        2

      int M = (D + F)/2;                                     4

      Intervalle resG = UI_DpR(T, D, M);                     2 + C_G

      Intervalle resD = UI_DpR(T, M + 1, F);                 3 + C_D

      if(resG == null || resD == null)                       3
         return null;                                        1
      else
         return resG.unionWith(resD);                        1 + C_unionWith
   }
```

Exemple : *Union des Intervalles*

Méthode DpR – *Mesure de la WCTC*

```
public static Intervalle UI_DpR(Intervalle[] T, int D, int F) {
      if(D > F)
          return null;
      if(D == F)
          return T[D];

      int M = (D + F)/2;

      Intervalle resG = UI_DpR(T, D, M);

      Intervalle resD = UI_DpR(T, M + 1, F);

      if(resG == null || resD == null)
          return null;
      else
          return resG.unionWith(resD);
   }
```

$O(1)+C_G+C_D$

Exemple : *Union des Intervalles*

Méthode DpR – *Mesure de la WCTC*

```
public static Intervalle UI_DpR(Intervalle[] T, int D, int F) {
        if(D > F)
            return null;
        if(D == F)
            return T[D];

        int M = (D + F)/2;

        Intervalle resG = UI_DpR(T, D, M);

        Intervalle resD = UI_DpR(T, M + 1, F);

        if(resG == null || resD == null)
            return null;
        else
            return resG.unionWith(resD);
    }
```

WCTC : O(|*T*|)
WCSC : O(?????)

Exemple : *Union des Intervalles*

Méthode DpR – *Mesure de la WCSC*

```
public Intervalle UI_DpR(Intervalle[] T, int D, int F) {     3
      if(D > F)
         return null;
      if(D == F)
         return T[D];

      int M = (D + F)/2;                                     1

      Intervalle resG = UI_DpR(T, D, M);                     1+C_G

      Intervalle resD = UI_DpR(T, M + 1, F);                 1+C_D

      if(resG == null || resD == null)
         return null;
      else
         return resG.unionWith(resD);                        C_unionWith
   }
```

Exemple : *Union des Intervalles*

Méthode DpR – *Mesure de la WCSC*

```
public Intervalle UI_DpR(Intervalle[] T, int D, int F) {     3
      if(D > F)
          return null;
      if(D == F)
          return T[D];

      int M = (D + F)/2;                                        1

      Intervalle resG = UI_DpR(T, D, M);                        1+C_G

      Intervalle resD = UI_DpR(T, M + 1, F);                    1+C_D

      if(resG == null || resD == null)
          return null;
      else
          return resG.unionWith(resD);                          6
   }
```

Exemple : *Union des Intervalles*

Méthode DpR – *Mesure de la WCSC*

```java
public Intervalle UI_DpR(Intervalle[] T, int D, int F) {
        if(D > F)
            return null;
        if(D == F)
            return T[D];

        int M = (D + F)/2;

        Intervalle resG = UI_DpR(T, D, M);


        Intervalle resD = UI_DpR(T, M + 1, F);

        if(resG == null || resD == null)
            return null;
        else
            return resG.unionWith(resD);
    }
```

$O(1)+C_D + \cancel{C_G}$

Exemple : *Union des Intervalles*

Méthode DpR – *Mesure de la WCSC*

```
public static Intervalle UI_DpR(Intervalle[] T, int D, int F) {
        if(D > F)
            return null;
        if(D == F)
            return T[D];

        int M = (D + F)/2;

        Intervalle resG = UI_DpR(T, D, M);

        Intervalle resD = UI_DpR(T, M + 1, F);

        if(resG == null || resD == null)
            return null;
        else
            return resG.unionWith(resD);
    }
```

<u>WCTC :</u> $O(|T|)$
<u>WCSC :</u> $O(\log_2(|T|))$

Calculer la Complexité d'un Algorithme « Diviser pour Régner »

Calculer la Complexité d'un Algorithme *DpR* :

1. À travers l'Arbre des Appels Récursifs :

- Calculer le coût d'un seul appel récursif **en fonction des entrées**.
- Tracer un arbre montrant comment sont générés les appels récursifs.
- À chaque niveau de l'arbre, calculer le coût de tous les appels récursifs appartenant à ce niveau.
- Compter le nombre des niveaux de l'arbre (un ***log*** en cas de division).
- Le coût total de l'algorithme est la somme des coûts de tous les niveaux (*multiplication*, *suite géométrique*,….).

Calculer la Complexité d'un Algorithme *DpR* :

1. À travers l'Arbre des Appels Récursifs :

Exemple de la recherche dichotomique :

```
public boolean Rech_Dichotomique(int[] T, int E, int D, int F) {
    if (D > F)                                              1
        return false ;
    int M = D + (F – D)/2 ;                                 5
    if(T[M] == E)                                           2
        return true ;
    else if (T[M] > E)                                      2
        return Rech_Dichotomique(T, E, D, M -1);            2
    else return Rech_Dichotomique(T, E, M+1, F);            2
}
```

⇒ *Chaque appel nécessite un coût borné par* **O(1)**.

Calculer la Complexité d'un Algorithme *DpR* :

1. À travers l'Arbre des Appels Récursifs :

Exemple de la recherche dichotomique :

- À chaque niveau de l'arbre on effectue un nombre d'instructions qui ne dépend pas de la taille N du tableau, donc c'est $O(1)$.
- Le nombre des niveaux de l'arbre est égal à $log_2(N)$.
- Le coût total de l'algorithme : $\underbrace{O(1) + \ldots + O(1)}_{log_2(N)\ \textit{fois}} = O(log_2(N))$.

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Un algorithme DpR peut produire l'équation des récurrences suivante :

$$C(N) = a.C(N/b) + O(N^d) \quad \text{où} \quad a >= 1,\ b >= 2,\ d >= 0$$

- N : La taille du problème initial.
- a : Le nombre de sous-problèmes considérés à chaque récursion.
- b : La taille du problème est divisée sur b à chaque récursion.
- $O(N^d)$: Le coût de chaque récursion (division, combinaison,…) sans compter le coût des appels récursifs.

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Un algorithme DpR peut produire l'équation des récurrences suivante :

$$C(N) = a.C(N/b) + O(N^d) \quad \text{où} \quad a >= 1,\ b >= 2,\ d >= 0$$

Cette équation est résolue comme suit :

- $d > \log_b(a) \quad \Rightarrow \quad C(N) = O(N^d)$.
- $d = \log_b(a) \quad \Rightarrow \quad C(N) = O(N^d . \log_b(N))$.
- $d < \log_b(a) \quad \Rightarrow \quad C(N) = O(N^{\log_b(a)})$.

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 1 : ***La recherche dichotomique***

$$C(N) = 1.C(N/2) + O(N^0)$$

Donc on a :

$$d = 0 = \log_2(1) \Rightarrow C(N) = O(N^0 . \log_2(N))$$

$$= \boxed{O(\log_2(N))}$$

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 2 : ***Le Tri par Fusion***

$$C(N) = ?.C(N/?) + O(N^{?})$$

Calculer la Complexité d'un Algorithme *DpR* :

Tri par fusion:

```
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|)
    }
}
```

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 2 : ***Le Tri par Fusion***

$$C(N) = 2.C(N/2) + O(N^1)$$

Donc on a :

$$d = 1 = \log_2(2) \quad \Rightarrow \quad C(N) = \boxed{O(N^1 . \log_2(N))}$$

Calculer la Complexité d'un Algorithme *DpR* :

2. **À travers le théorème des récurrences :** *" Master Theorem "*

Exemple 3 : ***Recherche dans un arbre binaire A équilibré ayant N nœuds***

$$C(N) = ?.C(N/?) + O(N^{?})$$

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 3 : ***Recherche dans un arbre binaire A équilibré ayant N nœuds***

$$C(N) = ?.C(N/?) + O(N^{?})$$

```
public boolean existe(Arbre A, int E) {
      if (A == null)
            return false ;
      else if (A.valeur() == E)
            return true ;
      else
            return existe(A.fg(), E) || existe(A.fd(), E) ;
}
```

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 3 : ***Recherche dans un arbre binaire A équilibré ayant N nœuds***

$$C(N) = 2.C(N/2) + O(N^0)$$

```
public boolean existe(Arbre A, int E) {
      if (A == null)
            return false ;
      else if (A.valeur() == E)
            return true ;
      else
            return existe(A.fg(), E) || existe(A.fd(), E) ;
}
```

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 3 : ***Recherche dans un arbre binaire A équilibré ayant N nœuds***

$$C(N) = 2.C(N/2) + O(N^0)$$

Donc on a :

$$d = 0 < \log_2(2) = 1 \quad \Rightarrow \quad C(N) = O(N^{\log_2(2)})$$

$$= \boxed{O(N)}$$

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 3 : ***Recherche dans un arbre binaire A équilibré ayant N nœuds***

$$C(N) = 2.C(N/2) + O(N^0)$$

Donc on a :

$$d = 0 < \log_2(2) = 1 \quad \Rightarrow \quad C(N) = O(N^{\log_2(2)})$$

$$= \boxed{O(N)}$$

Que devient cette équation de récurrences si l'arbre n'est pas équilibré ?

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 4 : ***Recherche dans un ABR A équilibré de N nœuds***

$$C(N) = ?.C(N/?) + O(N^{?})$$

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 4 : ***Recherche dans un ABR A équilibré de N nœuds***

$$C(N) = ?.C(N/?) + O(N^?)$$

```
public boolean existe(Arbre A, int E) {
    if (A == null)
        return false ;
    else if (A.valeur() == E)
        return true ;
    else if (A.valeur() > E)
        return existe(A.fils_gauche(), E) ;
    else
        return existe(A.fils_droit(), E) ;
}
```

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 4 : ***Recherche dans un ABR A équilibré de N nœuds***

$$C(N) = 1.C(N/2) + O(N^0)$$

```
public boolean existe(Arbre A, int E) {
     if (A == null)
          return false ;
     else if (A.valeur() == E)
          return true ;
     else if (A.valeur() > E)
          return existe(A.fils_gauche(), E) ;
     else
          return existe(A.fils_droit(), E) ;
}
```

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 4 : ***Recherche dans un ABR A équilibré de N nœuds***

$$C(N) = 1.C(N/2) + O(N^0)$$

Donc on a :

$$d = 0 = \log_2(1) \quad \Rightarrow \quad C(N) = O(N^0 . \log_2(N))$$

$$= \boxed{O(\log_2(N))}$$

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 5 : ***Afficher l'inverse d'un tableau T de N éléments***

$$C(N) = ?.C(N/?) + O(N^0)$$

Calculer la Complexité d'un Algorithme *DpR* :

Méthode *DpR* pour afficher l'inverse d'un tableau:

```
public void AlgoDpR(int[] T){
    if( T.length == 1 )
            System.out.print( T[0] + "  ") ;
    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++} ;

            AlgoDpR(TD) ; AlgoDpR(TG) ;
    }
}
```

$a = 2$, $b = 2$,   et $d = 1$

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 5 : ***Afficher l'inverse d'un tableau T de N éléments***

$$C(N) = 2.C(N/2) + O(N^1)$$

Donc on a :

$$d = 1 = \log_2(2) \quad \Rightarrow \quad C(N) = \boxed{O(N^1 . \log_2(N))}$$

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 6 : ***Recherche dans une matrice carrée***

On veut tester si un élément ***E*** appartient à une matrice ***A*[*N*][*N*]**.

Supposons que ***N*** est pair pour faciliter le discours.

$\Rightarrow$ Une recherche naïve coûtera $\mathbf{O}(N^2)$ en temps et en espace.

$\Rightarrow$ Proposer une méthode *DpR* et calculer sa *WCTC*.

Calculer la Complexité d'un Algorithme *DpR* :

2. À travers le théorème des récurrences : *" Master Theorem "*

Exemple 6 : ***Recherche dans une matrice carrée***

```
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)
              ||  Existe(A, E, DL, (FL- DL)/2, (FC-DC)/2+1, FC)
              ||  Existe(A, E, (FL-DL)/2+1, FL, DC, (FC-DC)/2)
              ||  Existe(A, E, (FL-DL)/2+1, FL, (FC-DC)/2+1, FC)
}
```

$$a = 4, \qquad b = 2, \qquad \text{et} \quad d = 0$$

$$d < \log_2(4) \quad \Rightarrow C(N) = O(N^{\log_2(4)}) = O(N^2).$$

Calculer la Complexité d'un Algorithme *DpR* :

HOME-WORK n°01:

On dispose de deux grands entiers X et Y représentés par des tableaux d’entiers de taille N (multiple de 2), par exemple X={2, 3} et Y={5, 6} représentent les entiers *23* et *56* respectivement. Le calcul de $X.Y$ peut être réalisé selon le développement suivant :

- $X = X_g.10^{N/2} + X_g$ //X_g (X_d) est la partie gauche (droite) du tableau X.
- $Y = Y_g.10^{N/2} + Y_d$ //Y_g (Y_d) est la partie gauche (droite) du tableau Y.

D’où

$$X.Y = X_g .\ Y_g .\ 10^N + X_g .\ Y_d .10^{N/2} + X_d .\ Y_g .10^{N/2} + X_d .\ Y_d.$$

Calculer la Complexité d'un Algorithme *DpR* :

HOME-WORK n°01:

1. Proposez une méthode *Diviser-pour-Régner* pour implémenter le développement expliqué. La méthode peut avoir une signature comme suit :

   int *Multiplication*(int[] *X*, int[] *Y*,….) *//penser aux indices permettant de diviser*

   L'utilisation de boucles n'est pas autorisée.

2. Une méthode **int *Puissance*(int *N*)** doit être implémentée pour calculer $\mathbf{10^{N}}$.

3. Calculez la *WCTC* et *WCSC* de votre méthode.

Calculer la Complexité d'un Algorithme *DpR* :

HOME-WORK n°02:

Soit donné un tableau ***T*** composé d'entiers triés, et un entier ***K***, l'objectif est de tester s'il y a deux éléments dans ***T*** dont la somme est égale à ***K***. Si c'est le cas, les deux éléments doivent être retournés. **L'utilisation de boucles n'est pas autorisée**.

1. Proposez une méthode *Diviser-pour-Régner* pour résoudre ce problème. La méthode peut avoir une signature comme suit :

   Paire *Test*(int *X*, int *K*,….) *//penser aux indices permettant de diviser*

2. Calculez la *WCTC* et *WCSC* de votre méthode.
3. Généralisez votre solution pour tester s'il y a trois éléments dont la somme est égale à ***K***.
4. Calculez la *WCTC* et *WCSC* de cette généralisation.