Programmation Dynamique

Dr. Mahfoud Houari

mahfoud.houari@gmail.com

Université Abou-Bakr Belkaïd - Tlemcen

2022/2023

Programmation Dynamique

(*Dynamic Programming*)

Exemple introductif – *Suite de Fibonacci*

Méthode récursive :

```
int Fibo(int N){
    if (N == 0 || N == 1)
        return 1 ;
    else
        return Fibo(N - 1) + Fibo(N - 2) ;
}
```

La méthode est assez simple et intuitive, mais est-elle praticable ?

BCTC : $2 = O(1)$

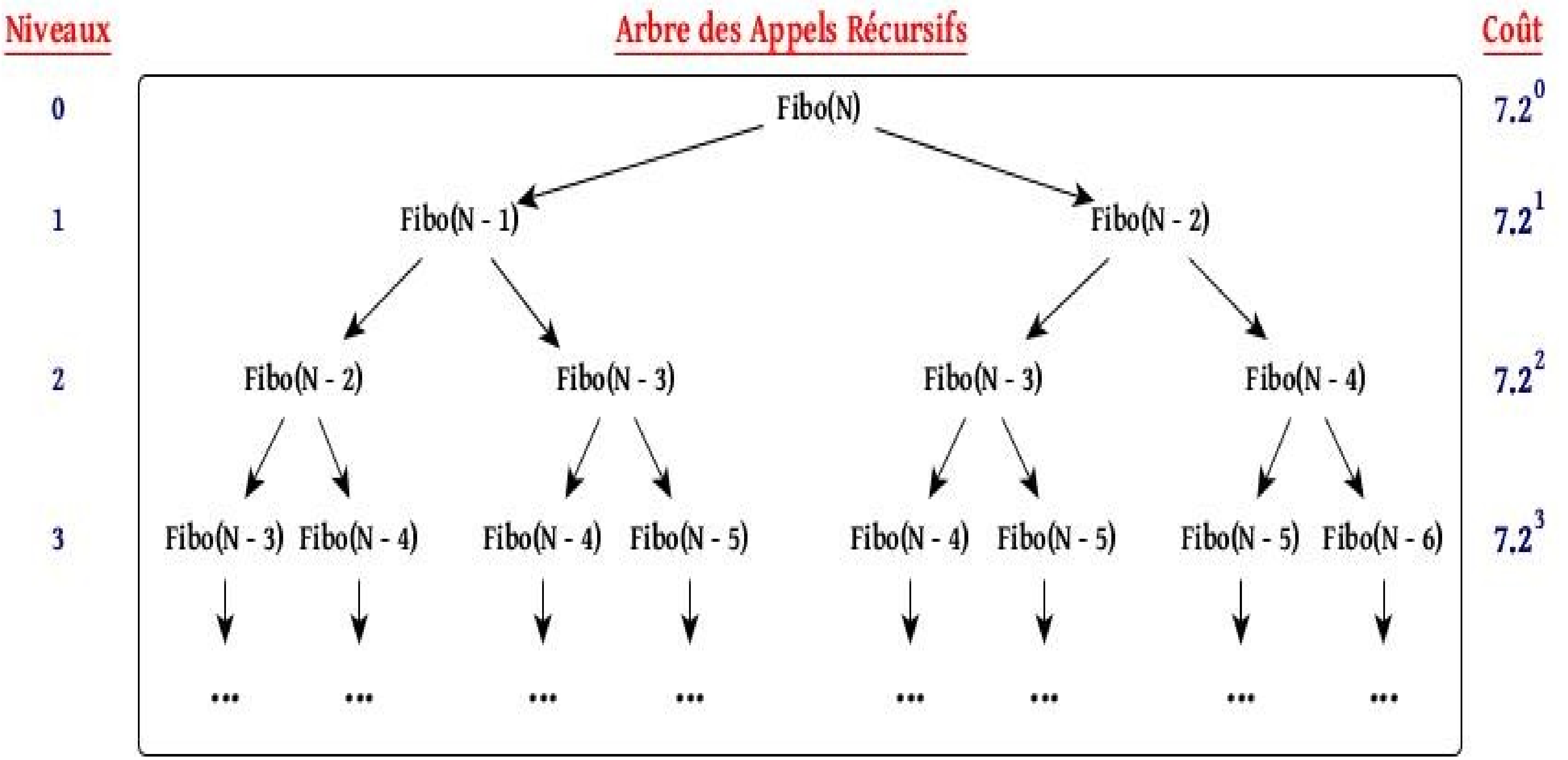
WCTC : $7 + C_{N-1} + C_{N-2} = 7 + (7 + C_{N-2} + C_{N-3}) + (7 + C_{N-3} + C_{N-4}) = ?$

BCSC :

WCSC :

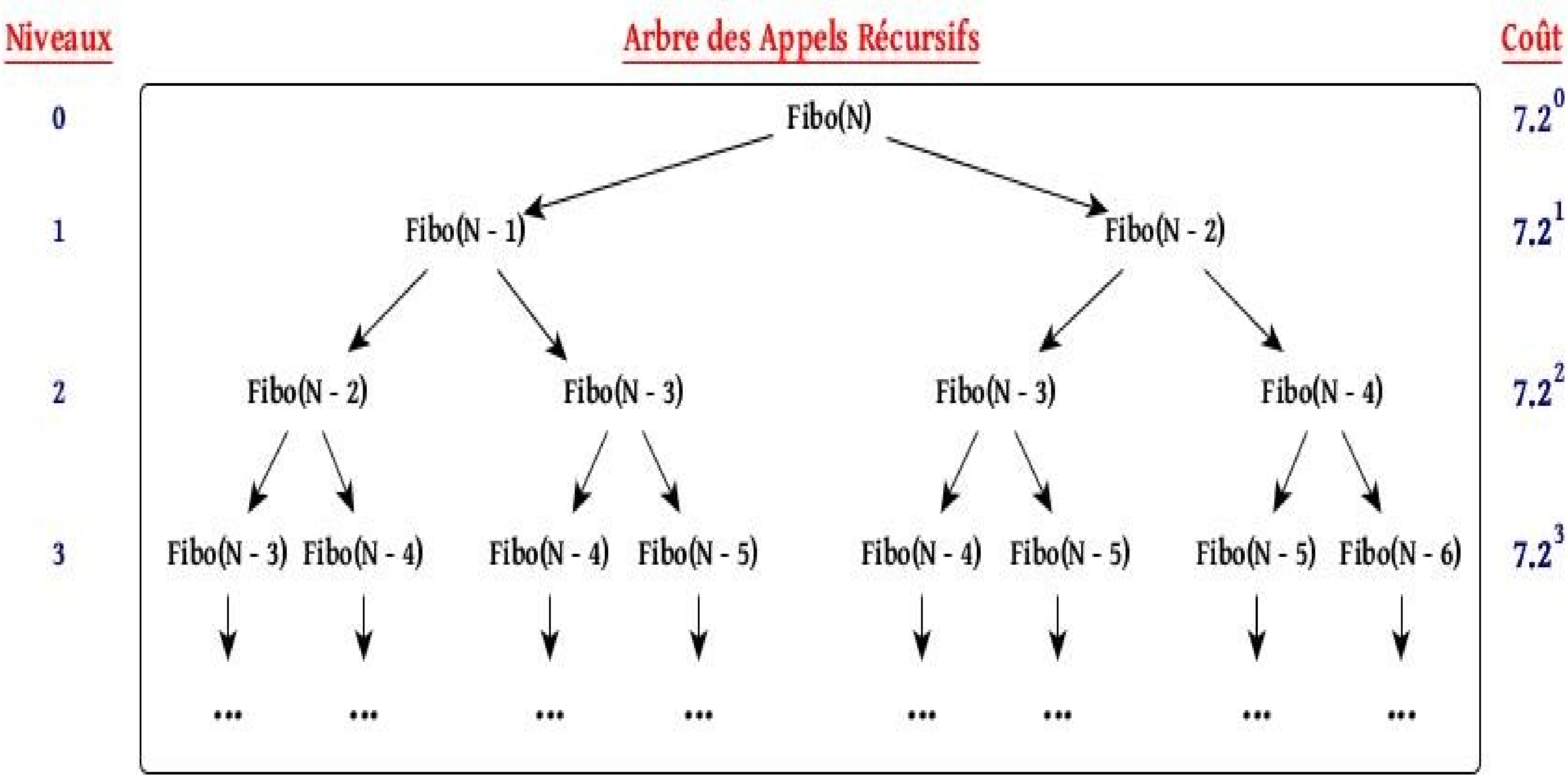
? ?

Exemple introductif – *Suite de Fibonacci*



Remarque : *Le nombre de niveaux est égal à* ***N - 1.***

Exemple introductif – *Suite de Fibonacci*



Coût total : $7(2^0 + \ldots + 2^{N-1}) = 7(1 - 2^N)/(1 - 2) \Rightarrow \boldsymbol{O(2^N)}$.

Exemple introductif – *Suite de Fibonacci*

Méthode récursive :

```
int Fibo(int N){
    if (N == 0 || N == 1)
        return 1 ;
    else
        return Fibo(N - 1) + Fibo(N - 2) ;
}
```

La méthode est assez simple et intuitive, mais est-elle praticable ?

BCTC : **2 = O(1)**

WCTC : **7.2^N = O(2^N)** ⇒ La méthode est *impraticable*

BCSC : **1 = O(1)**

WCSC : **1 case mémoire par niveau ⇒ O(N)**

Exemple introductif – *Suite de Fibonacci*

Méthode récursive :

```
int Fibo(int N){
    if (N == 0 || N == 1)
        return 1 ;
    else
        return Fibo(N - 1) + Fibo(N - 2) ;
}
```

⇒ *L'**inefficacité** de la méthode est due au **calcul répétitif** des sous-problèmes.*

⇒ ***Comment éviter cela ?***

⇒ ***Enregistrer le résultat de chaque appel récursif***

Exemple introductif – *Suite de Fibonacci*

Méthode Dynamique Ascendante :

```
int Fibo(int N){
    if(N == 0 || N == 1)                                      O(1)
        return 1 ;                                            O(1)
    int [] resultats = new int[N + 1] ;                       O(N)
    resultats[0] = 1 ;                                        O(1)
    resultats[1] = 1 ;                                        O(1)
    for (int i = 2 ; i ≤ N ; i++){                            O(N)
        resultats[i] = resultats[i - 1] + resultats[i - 2] ;  O(N)
    }
    return resultats[N] ;                                     O(1)
}
```

BCTC :	**O(1)**	**BCSC :**	**O(1)**
WCTC :	**O(N)**	**WCSC :**	**O(N)**

Principe de la Programmation Dynamique :

1. Un problème de taille ***N*** nécessite la résolution de sous-problèmes de taille inférieure à ***N*** (*Pas forcément une division*).

2. S'applique pour des méthodes récursives qui présentent l'aspect de ***chevauchement de sous-problèmes***.

3. L'aspect de ***mémorisation*** permet de calculer et enregistrer une *seule fois* la solution optimale de chaque sous-problème.

4. La solution globale est construite, d'une façon *ascendante* ou *descendante* en combinant les sous-solutions enregistrées.

5. Accroître la complexité spatiale pour réduire la complexité temporelle.

Exemple introductif – *Suite de Fibonacci*

Méthode Dynamique Descendante :

```
class Main {
        HashMap<Integer, Integer> Memo ;
        public static void main(String[] args){
                Memo = new HashMap<Integer, Integer>();
                Memo.put(0, 1); Memo.put(1, 1) ;
                int N = 15 ;
                System.out.println(Fibo(N)) ;
        }
        public static int Fibo(int N){
                ......
        }
}
```

Exemple introductif – *Suite de Fibonacci*

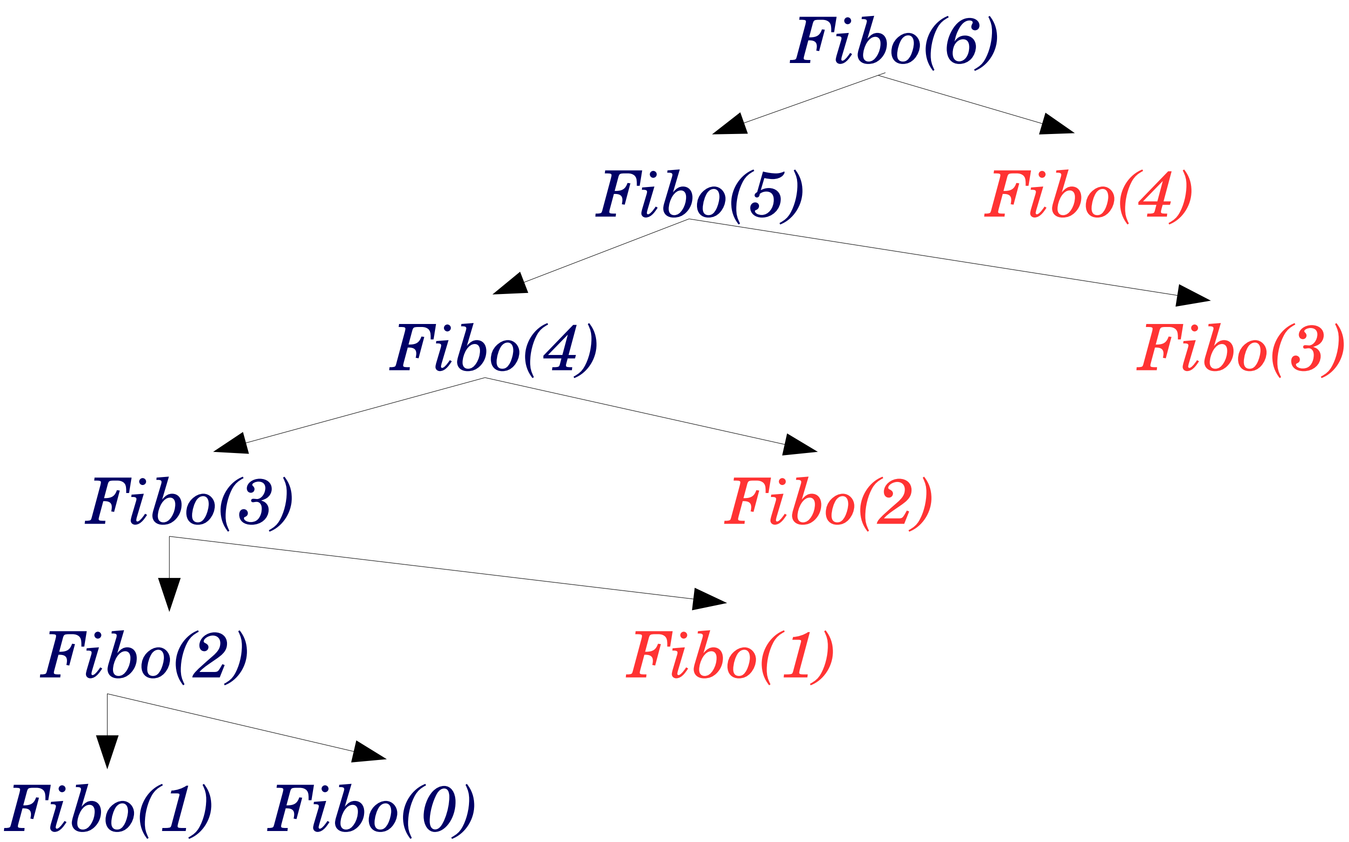
Méthode Dynamique Descendante :

```
public static int Fibo(int N){
    if (Memo.containsKey(N))    //Le sous-problème de taille N est déjà résolu
        return Memo.get(N) ;
    else {
        int R = Fibo(N-1) + Fibo(N-2); //Calculer la solution du sous-problème
        Memo.put(N, R) ;
        return R ;
    }
}
```

BCTC :	O(1)	**BCSC :**	O(?)
WCTC :	O(?)	**WCSC :**	O(?)

Exemple introductif – *Suite de Fibonacci*

Exemple de déroulement de la méthode descendante:



<u>Coût temporel</u>

O(*1*)

O(*1*) + O(*1*)

O(*1*) + O(*1*)

O(*1*) + O(*1*)

O(*1*) + O(*1*)

O(*1*) + O(*1*)

Exemple introductif – *Suite de Fibonacci*

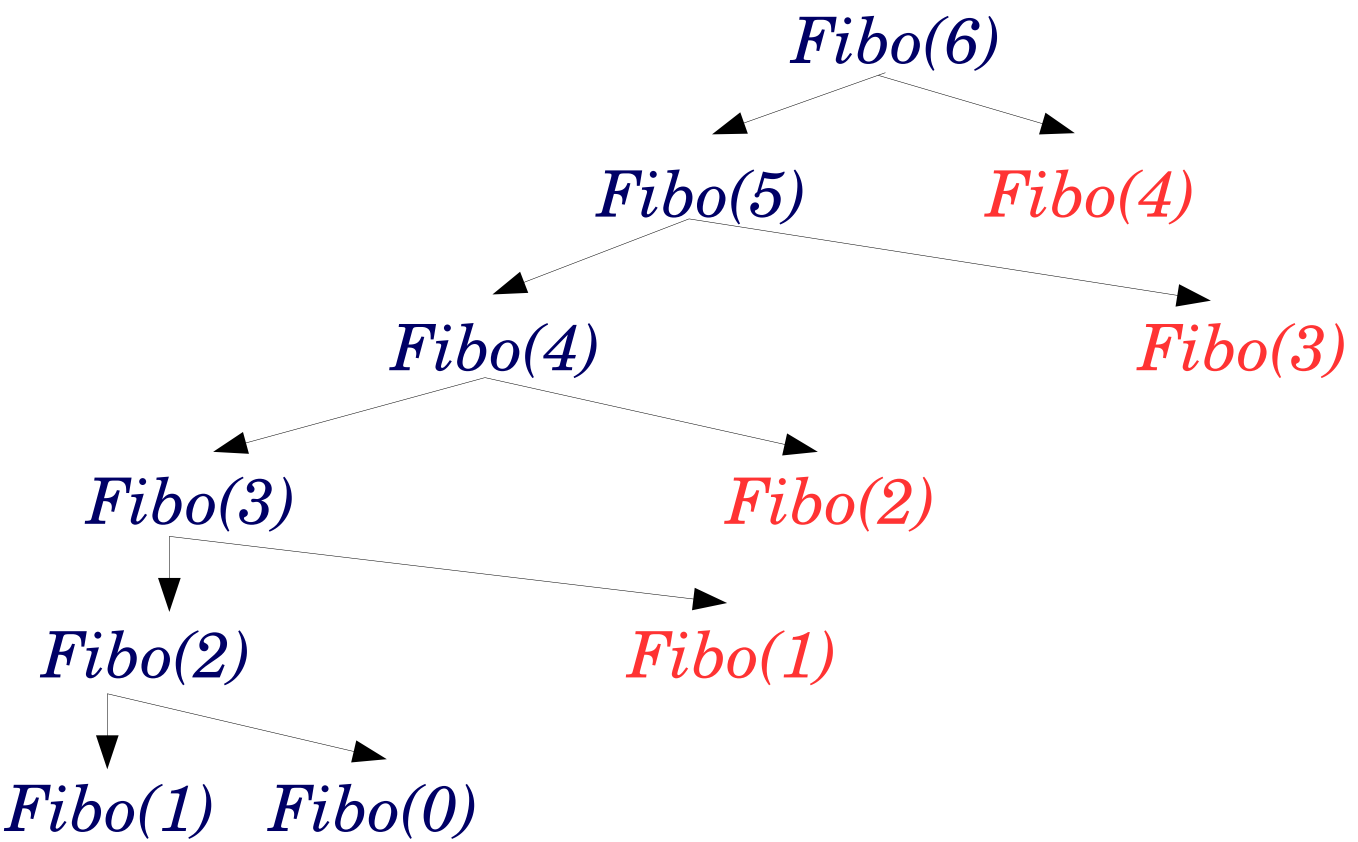
Méthode Dynamique Descendante :

```
public static int Fibo(int N){
    if (Memo.containsKey(N))    //Le sous-problème de taille N est déjà résolu
        return Memo.get(N) ;
    else {
        int R = Fibo(N-1) + Fibo(N-2); //Calculer la solution du sous-problème
        Memo.put(N, R) ;
        return R ;
    }
}
```

BCTC :	O(1)	**BCSC :**	O(?)
WCTC :	O(*N*)	**WCSC :**	O(?)

Exemple introductif – *Suite de Fibonacci*

Exemple de déroulement de la méthode descendante:



Coût spatial

O(*1*)

O(*1*)

O(*1*)

O(*1*)

O(*1*)

O(*1*)

Exemple introductif – *Suite de Fibonacci*

Méthode Dynamique Descendante :

```
public static int Fibo(int N){
    if (Memo.containsKey(N))    //Le sous-problème de taille N est déjà résolu
        return Memo.get(N) ;
    else {
        int R = Fibo(N-1) + Fibo(N-2); //Calculer la solution du sous-problème
        Memo.put(N, R) ;
        return R ;
    }
}
```

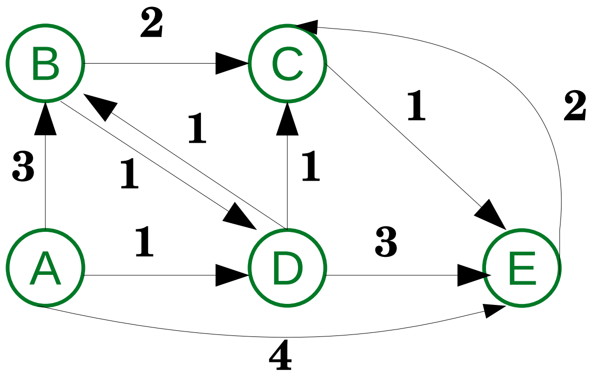
BCTC :	O(1)	**BCSC :**	O(1)
WCTC :	O(*N*)	**WCSC :**	O(N) + O(N)

Critères d'applicabilité de la PRD :

1) Chevauchement de sous-problèmes :

Les mêmes sous-problèmes sont calculés plusieurs fois.

Exemple : Problème de ***Plus Court Chemin***



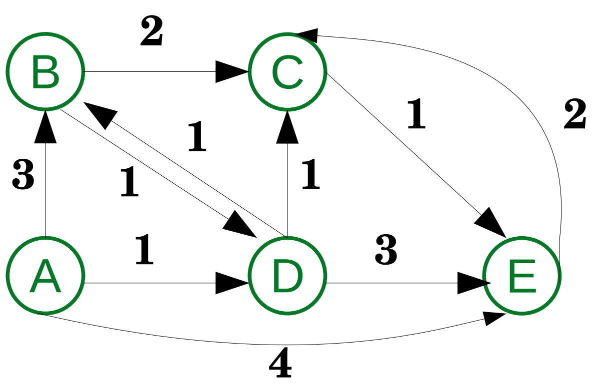
Y-aura t-il un chevauchement en calculant le chemin PCC(*A*,*E*) ?

Critères d'applicabilité de la PRD :

1) Chevauchement de sous-problèmes :

Les mêmes sous-problèmes sont calculés plusieurs fois.

Exemple : Problème de ***Plus Court Chemin***



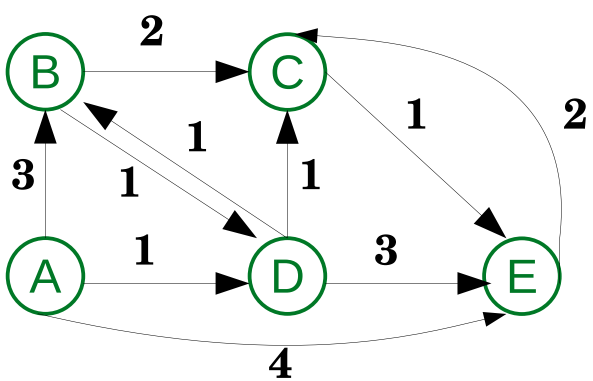
PCC(*A*,*E*) *invoque* ***deux fois*** *le calcul du* PCC(*C*,*E*).

Critères d'applicabilité de la PRD :

2) Sous-structure optimale :

La solution optimale du problème initiale comprend des sous-solutions optimales pour ses sous-problèmes.

Exemple : Problème de ***Plus Court Chemin***



PCC(***A***, ***E***) = ***A*** → ***D*** → ***C*** → ***E***.

D'où :

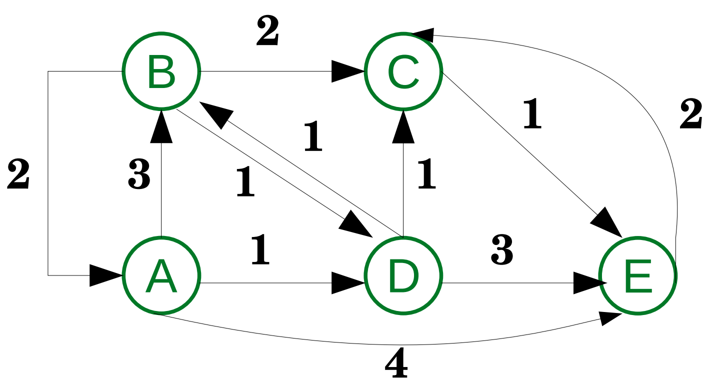
PCC(***A***, ***C***) = ***A*** → ***D*** → ***C***.
PCC(***D***, ***E***) = ***D*** → ***C*** → ***E***.
...

⇒ *Les deux critères sont valides.*

⇒ *D'où, le problème **PCC** peut être résolu avec la programmation dynamique.*

Critères d'applicabilité de la PRD :

Exemple : Problème de ***Plus Long Chemin*** (sans répétition de nœuds)



PLC(***A***, ***E***) = ***A → B → D → E (7)***

Peut-on déduire que

PLC(***D***, ***E***) = ***D → E (3)*** ? ***NON***
⇒ PLC(***D***, ***E***) = ***D → B → C → E (4)***

PLC(***B***, ***D***) = ***B → D (1)*** ? ***NON***
⇒ PLC(***B***, ***D***) = ***B → A → D*** ***(3)***

⇒ *Pas de sous-structure optimale.*

⇒ *D'où, le problème **PLC** ne peut pas être résolu avec la programmation dynamique.*

Application de la PRD pour le problème du

« *Rendu de monnaie* »

Version 1

Problème du « *Rendu de monnaie* » – Version 1

Description :

Un distributeur de boissons accepte les pièces de ***5***, ***10***, ***20***, ***50***, et ***100 DA***. Étant donné une monnaie ***M*** à rendre, le but est de trouver le ***nombre minimum*** de pièces à rendre dont la valeur totale est égale à ***M***.

Exemple :

Pour **M = 155** DA, la solution optimale consiste à rendre **3** pièces : **100** + **50** + **5**.

⇒ ***La PRD est-elle applicable pour ce problème ?***

Problème du « *Rendu de monnaie* » – Version 1

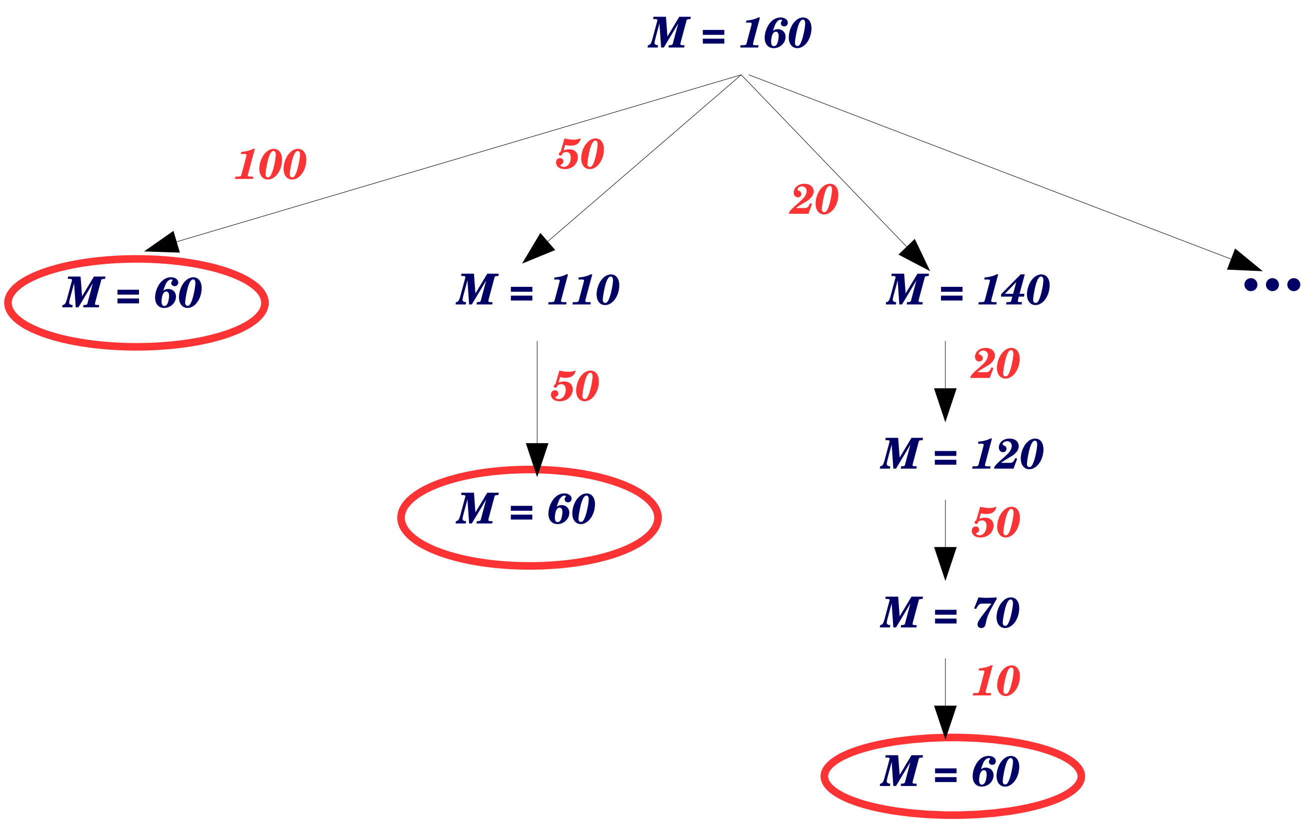
⇒ ***Formulation de la résolution exacte***

$$\mathbf{Sol}(M) = \mathbf{Min}\ (1 + \mathbf{Sol}(M - P[i]))$$

pour toute pièce $P[i]$ dont la valeur est $\leq M$

Problème du « *Rendu de monnaie* » – Version 1

⇒ *Chevauchement de sous-problèmes :*



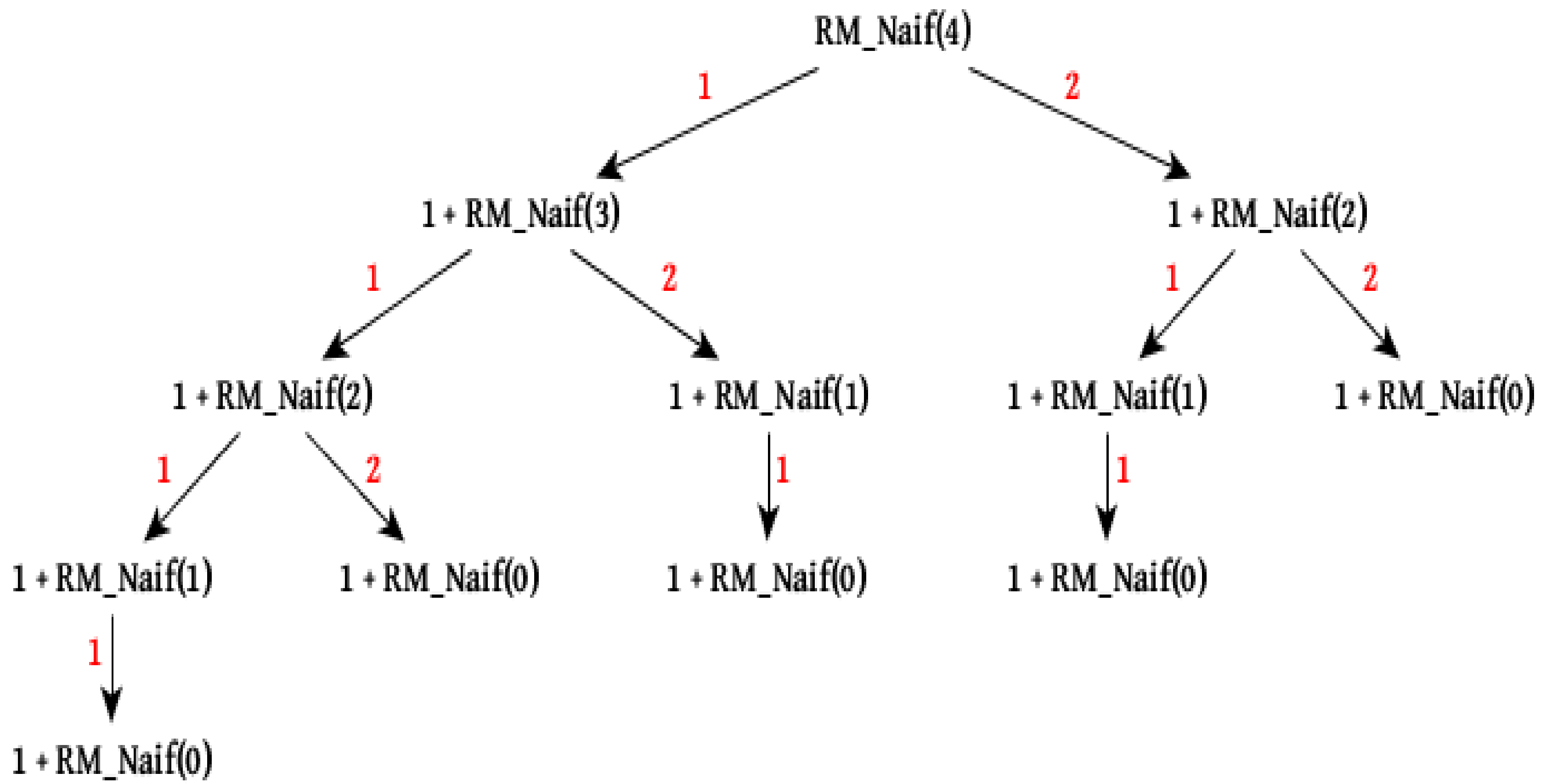
Problème du « *Rendu de monnaie* » – Version 1

Solution exhaustive naïve :

```
int RM_Naif(int M, int[] P){
    if(M == 0) return 0 ;
    int sol = 100000 ;
    for (int i = 0 ; i < P.length ; i++){
        if(P[i] <= M){
            sol = Math.min(sol , 1 + RM_Naif(M - P[i], P)) ;
        }
    }
    return sol ;
}
```

Problème du « *Rendu de monnaie* » – Version 1

Solution exhaustive naïve – *AARs pour M=4*



Sol(4) **= Min(1 +** ***Sol(3)*****, 1+** ***Sol(2)*****)**

Problème du « *Rendu de monnaie* » – Version 1

Solution exhaustive naïve :

```
int RM_Naif(int M, int[] P){
	if(M == 0) return 0 ;
	int sol = 100000 ;
	for (int i = 0 ; i < P.length ; i++){
		if(P[i] <= M){
			sol = Math.min(sol , 1 + RM_Naif(M - P[i], P)) ;
		}
	}
	return sol ;
}
```

Exercice : En fonction de ***M*** et ***P***, vérifier que ***RM_Naif*** est *exponentielle en temps* et *polynomiale en espace*.

Problème du « *Rendu de monnaie* » – Version 1

Solution dynamique descendante:

```
class Main {
        HashMap<Integer, Integer> Memo ;
        public static void main(String[] args){
                Memo = new HashMap<Integer, Integer>();
                Memo.put(0, 0);
                int M = 35 ;
                System.out.println(RM_PRD(M)) ;
        }
        public static int RM_PRD(int M, int[] P){

                . . .
        }
}
```

Problème du « *Rendu de monnaie* » – Version 1

Solution dynamique descendante:

```
public static int RM_PRD(int M, int[] P){
    if (memo.containsKey(M))
        return memo.get(M) ;
    int sol = 100000 ;
    for (int i = 0 ; i < P.length ; i++){
        if(P[i] <= M){
            sol = Math.min(sol , 1 + RM_PRD(M - P[i], P)) ;
        }
    }
    memo.put(M, sol) ;
    return sol ;
}
```

Solution naïve

Problème du « *Rendu de monnaie* » – Version 1

Solution dynamique descendante:

```
public static int RM_PRD(int M, int[] P){
    if (memo.containsKey(M))
        return memo.get(M) ;
    int sol = 100000 ;
    for (int i = 0 ; i < P.length ; i++){
        if(P[i] <= M){
            sol = Math.min(sol , 1 + RM_PRD(M - P[i], P)) ;
        }
    }
    memo.put(M, sol) ;
    return sol ;
}
```

Solution naïve

WCTC : O(????)
WCSC : O(????)

Question :

La PRD assure t-elle toujours une complexité polynomiale ?

Réponse :

1. Si le nombre de sous-problèmes est polynomial et chacun nécessite un traitement polynomial alors la méthode dynamique est polynomiale.

2. Si le nombre de sous-problèmes n'est pas polynomial alors la PRD ne garantit pas une complexité polynomiale.