Programmation Dynamique (suite)

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

2022/2023

Applicabilité de la PRD pour le problème du

« *Rendu de monnaie* »

Version 2

Problème du « *Rendu de monnaie* » – Version 2

Description :

Le problème est caractérisé par la monnaie à rendre ***M***, le système monétaire ***P***, et la disponibilité des pièces ***D***, tel que ***D*[i]** = ***k*** signifie que la pièce ***P*[i]** est présente dans le système avec ***k*** occurrences.

Exemple. Pour :

M = 110 DA,

P = [5, 10, 20, 50, 100]

D = [0, 0, 4, 4, 1]

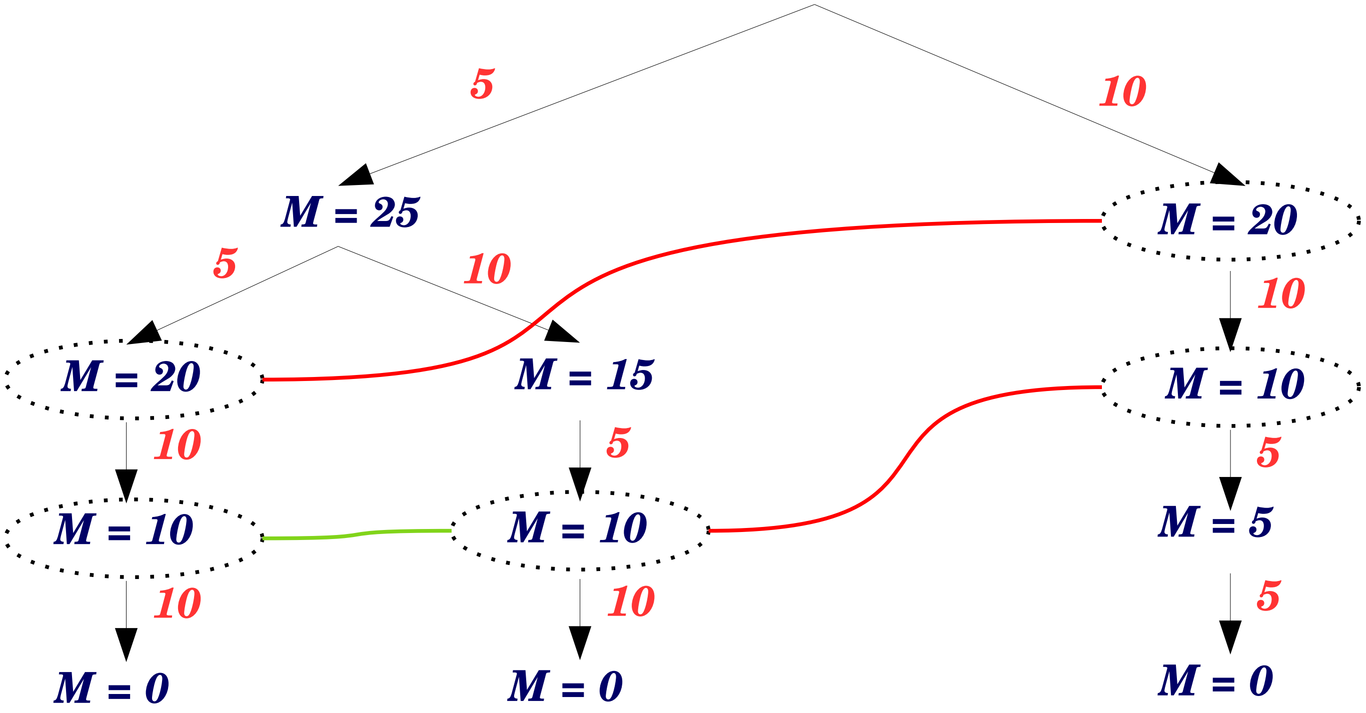
La solution optimale consiste à rendre **4** pièces : 1.**50** + 3.**20**.

⇒ ***Existe t-il un chevauchement de sous-problèmes ?***

Problème du « *Rendu de monnaie* » – Version 2

⇒ *Chevauchement de sous-problèmes :*

M = *30*, *P*=[5,10,20], *D*=[2, 2, 0]



⇒ Il existe bien un chevauchement entre les appels récursifs ayant la même disponibilité.

Problème du « *Rendu de monnaie* » – Version 2

Solution naïve :

```
int RM_Naif(int M, int[] P, int[] D){
	if(M == 0)	return 0;
	int sol = 100000;
	boolean blocage = true;
	for (int i = 0; i < P.length; i++){
		if(P[i] <= M && D[i] > 0){
			D[i]--;
			int solCourante = RM_Naif(M - P[i], P, D);
			if(solCourante != -1) {
				blocage = false;
				sol = Math.min(sol , 1 + solCourante);
			}
		}
	}
	return blocage? -1 : sol;
}
```

Problème du « *Rendu de monnaie* » – Version 2

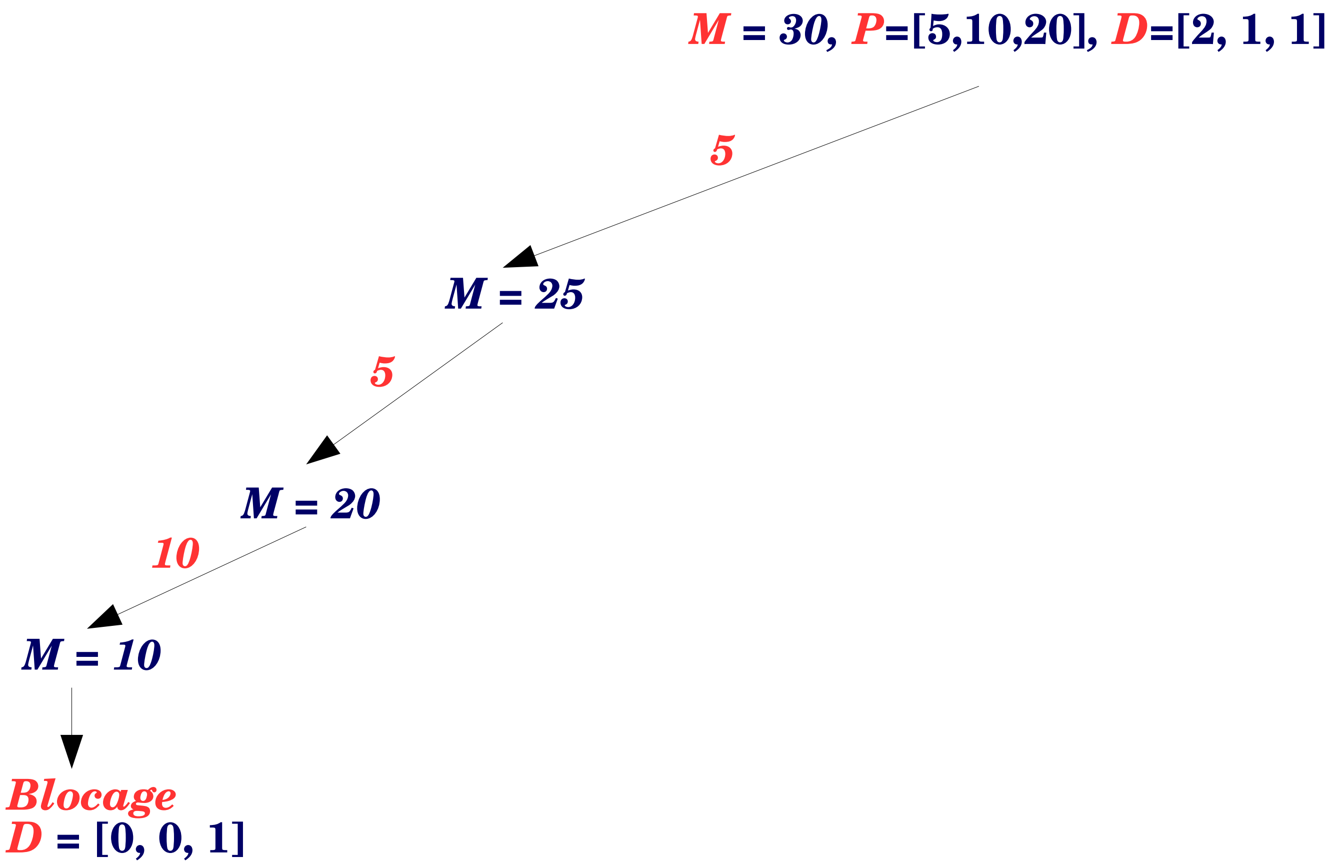
Solution naïve :

```
int RM_Naif(int M, int[] P, int[] D){
    if(M == 0)    return 0;
    int sol = 100000;
    boolean blocage = true;
    for (int i = 0; i < P.length; i++){
        if(P[i] <= M && D[i] > 0){
            D[i]--;
            int solCourante = RM_Naif(M - P[i], P, D);
            if(solCourante != -1) {
                blocage = false;
                sol = Math.min(sol , 1 + solCourante);
            }
        }
    }
    return blocage? -1 : sol;
}
```

Fausse solution

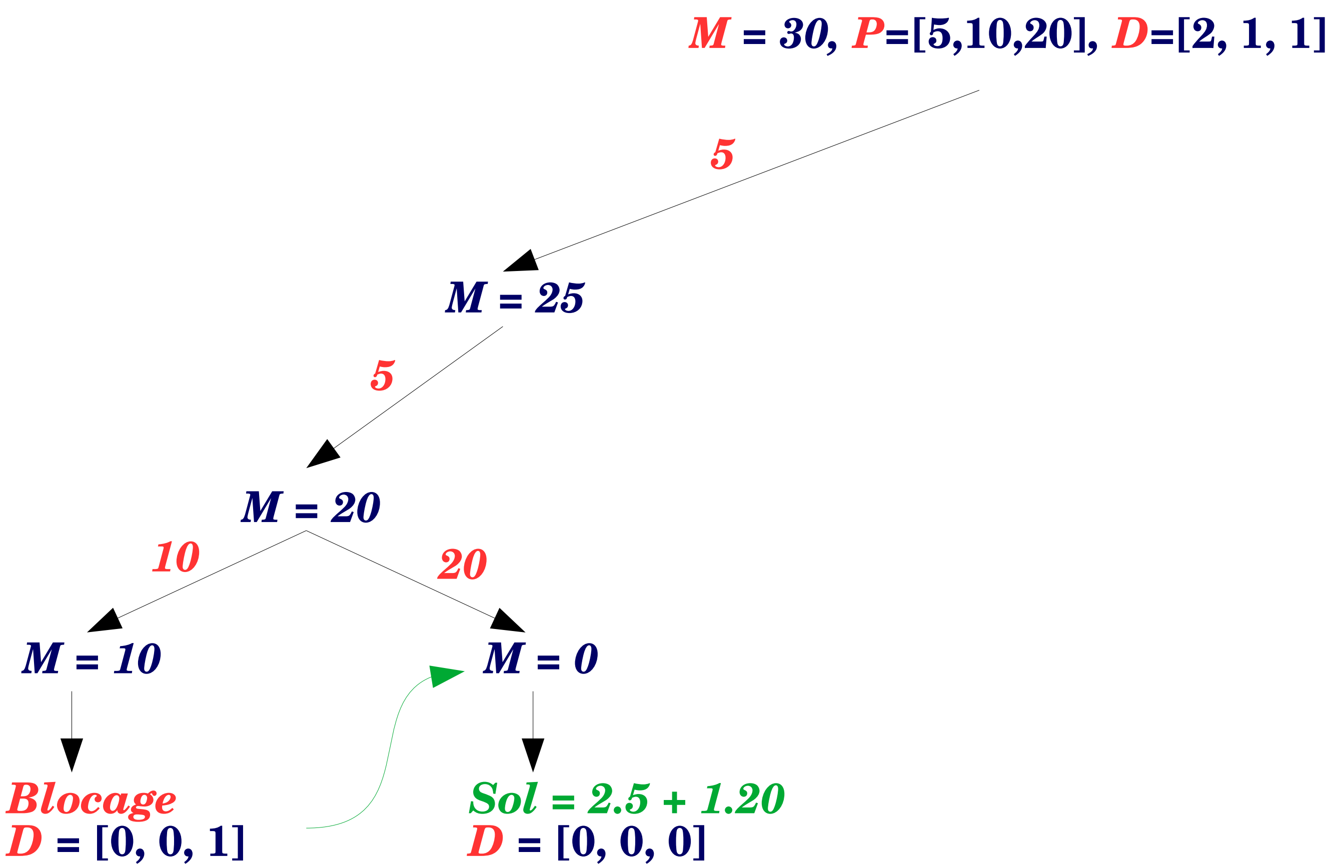
Problème du « *Rendu de monnaie* » – Version 2

Pourquoi ne doit-on pas changer le D original ?



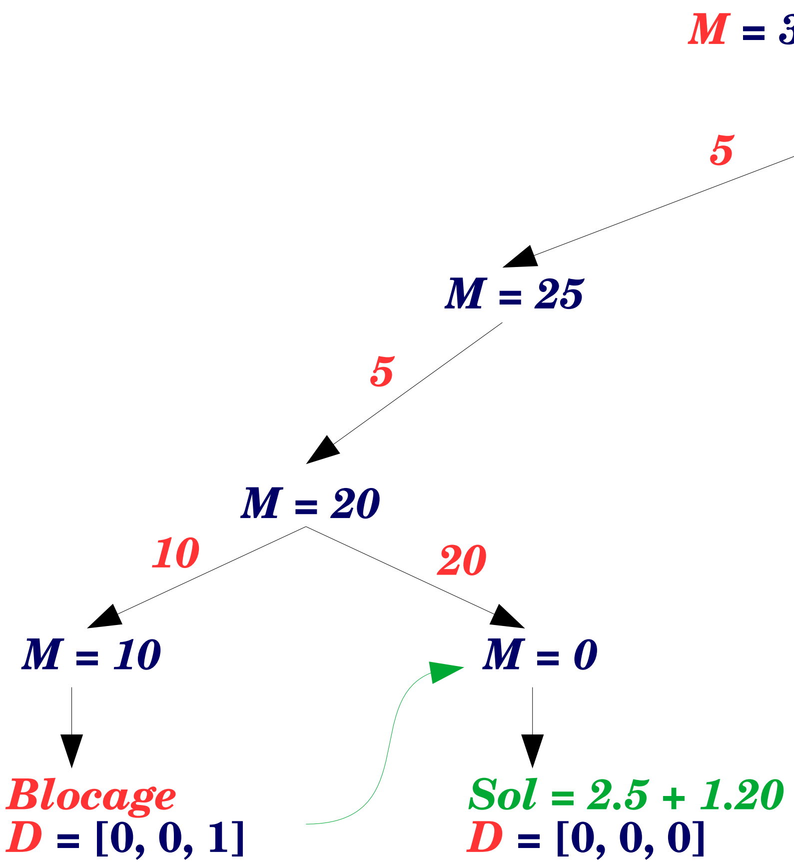
Problème du « *Rendu de monnaie* » – Version 2

Pourquoi ne doit-on pas changer le D original ?



Problème du « *Rendu de monnaie* » – Version 2

Pourquoi ne doit-on pas changer le D original ?



⇒ Solution fausse retournée : 3 pièces

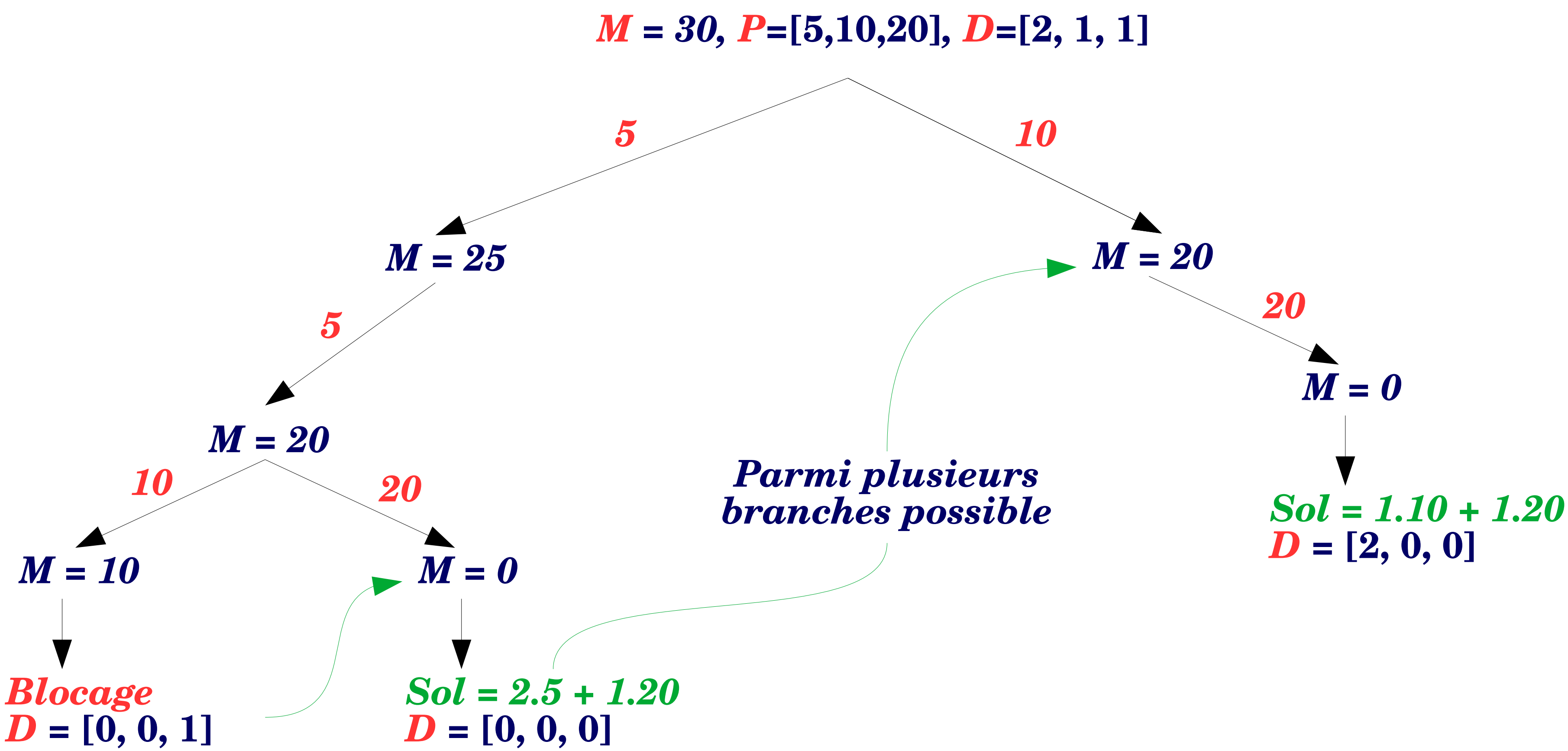
Problème du « *Rendu de monnaie* » – Version 2

Solution naïve :

```
int RM_Naif(int M, int[] P, int[] D){
	if(M == 0)	return 0;
	int sol = 100000;
	boolean blocage = true;
	for (int i = 0; i < P.length; i++){
		if(P[i] <= M && D[i] > 0){
			D[i]--;
			int solCourante = RM_Naif(M - P[i], P, D);
			D[i]++;
			if(solCourante != -1) {
				blocage = false;
				sol = Math.min(sol , 1 + solCourante);
			}
		}
	}
	return blocage? -1 : sol;
}
```

Problème du « *Rendu de monnaie* » – Version 2

Pourquoi ne doit-on pas changer le D original ?



⇒ Solution correcte retournée : 2 pièces

Problème du « *Rendu de monnaie* » – Version 2

Solution dynamique descendante:

```
class Main {
	HashMap<Integer, Integer> Memo ;
	public static void main(String[] args){
		Memo = new HashMap<String, Integer>();
		int[] P = {5, 10, 20} ;
		int[] D = {2, 1, 1} ;
		int M = 30 ;
		System.out.println(RM_PRD(M, P, D)) ;
	}
	public static int RM_PRD(int M, int[] P, int[] D){

		. . .
	}
}
```

Problème du « *Rendu de monnaie* » – Version 2

Solution dynamique descendante:

```
int RM_PRD(int M, int[] P, int[] D){
	String cle = Arrays.toString(D) ;
	if(Memo.containsKey(cle)) return Memo.get(cle) ;
	int sol = 100000, solCourante;
	boolean blocage = true;
	for (int i = 0; i < P.length; i++){
		if(P[i] <= M && D[i] > 0){

			D[i]--; solCourante = RM_PRD(M - P[i], P, D); D[i]++;
			if(solCourante != -1) {
				blocage = false;
				sol = Math.min(sol , 1 + solCourante);
			}
		}
	}
	if(blocage) return -1;
	Memo.put(cle, sol) ; return sol ;
}
```

Solution naïve

Question :

La version dynamique assure t-elle une complexité polynomiale ?

Réponse :

- Pour l'exemple donné auparavant, D=[2,1,1], la méthode ***RM_PRD*** enregistre au maximum 3.2.2 cas possibles de disponibilités (c-a-d. Sous-problèmes différents).

- Supposons que toutes les pièces sont disponibles avec un nombre d'occurrences $\boldsymbol{k}$ au début. Alors, il y aura $(\boldsymbol{k}+1)^{|\boldsymbol{D}|}$ sous-problèmes différents à prendre en charge par la méthode ***RM_PRD***.

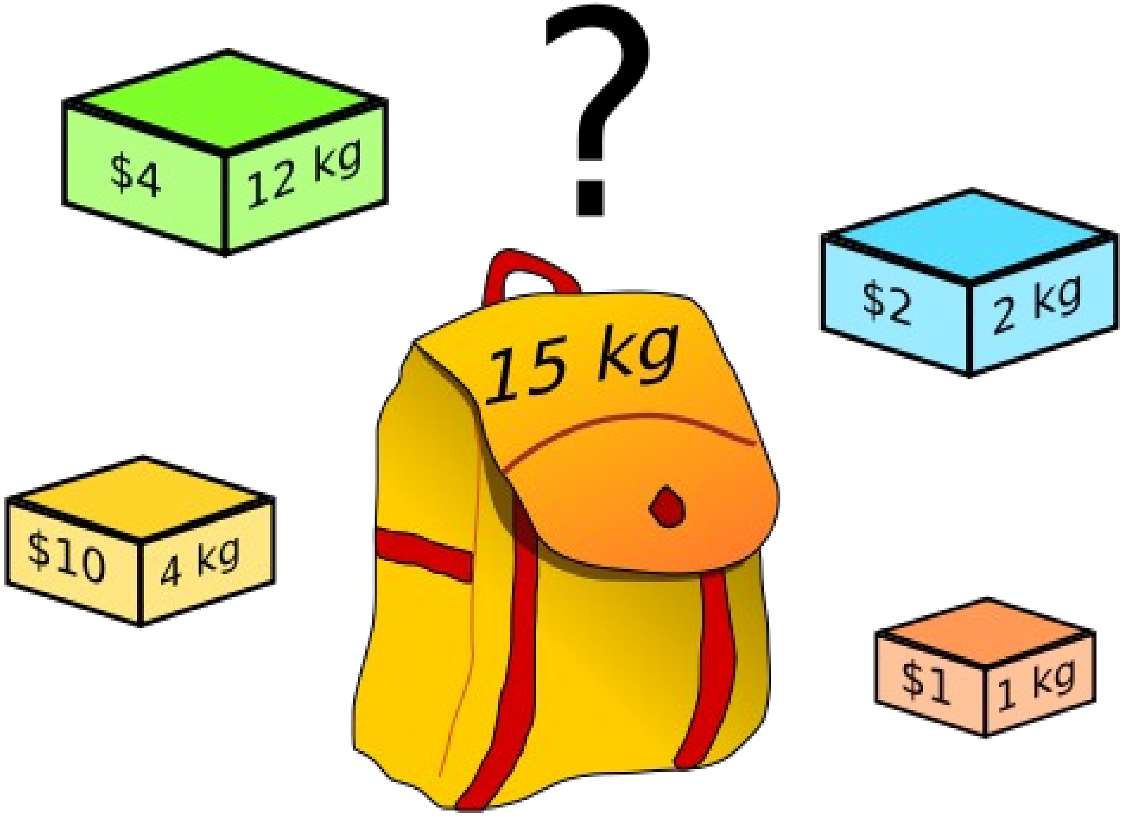
- Le nombre de sous-problèmes n'est pas polynomial, d'où, la méthode n'a pas de complexité polynomiale.

- Le problème de ***Rendu de Monnaie*** avec disponibilité n'a pas de solution polynomiale.

Applicabilité de la PRD pour le problème du

« *Sac à dos 0/1* »

Problème du *Sac à dos* – Description



Objectif :

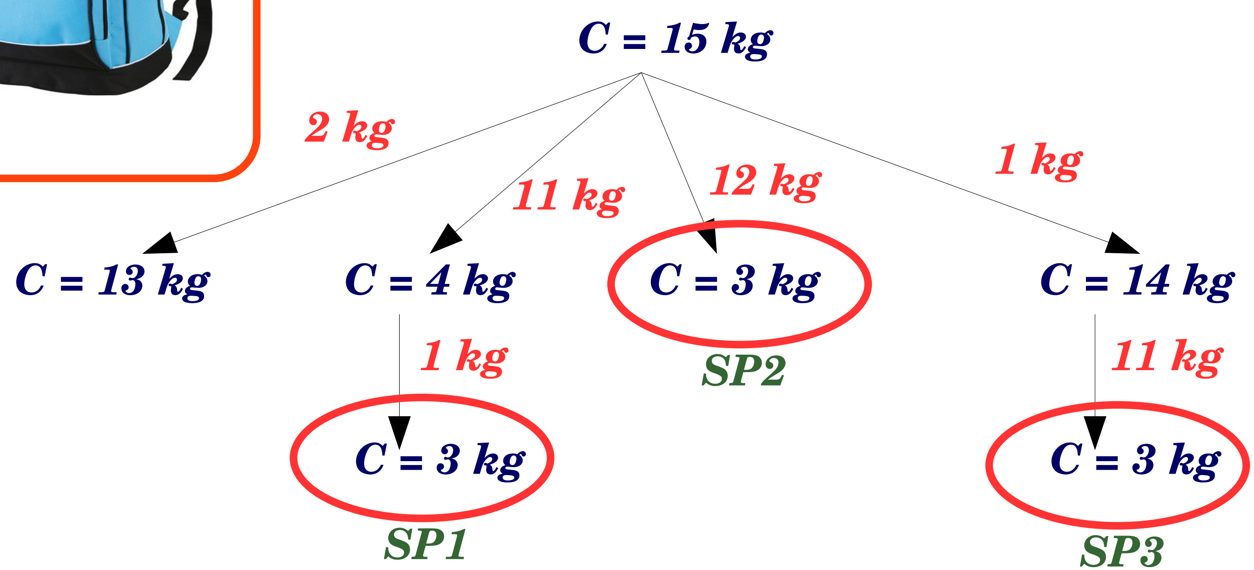
Déterminer les objets à prendre pour maximiser le gain total tout en respectant le poids maximal.

Problème du « *Sac à Dos* » – Version 0/1

⇒ *Chevauchement de sous-problèmes :*

Objet 1	2 kg	2 DA
Objet 2	11 kg	6 DA
Objet 3	12 kg	4 DA
Objet 4	1 kg	5 DA

C = 15 kg



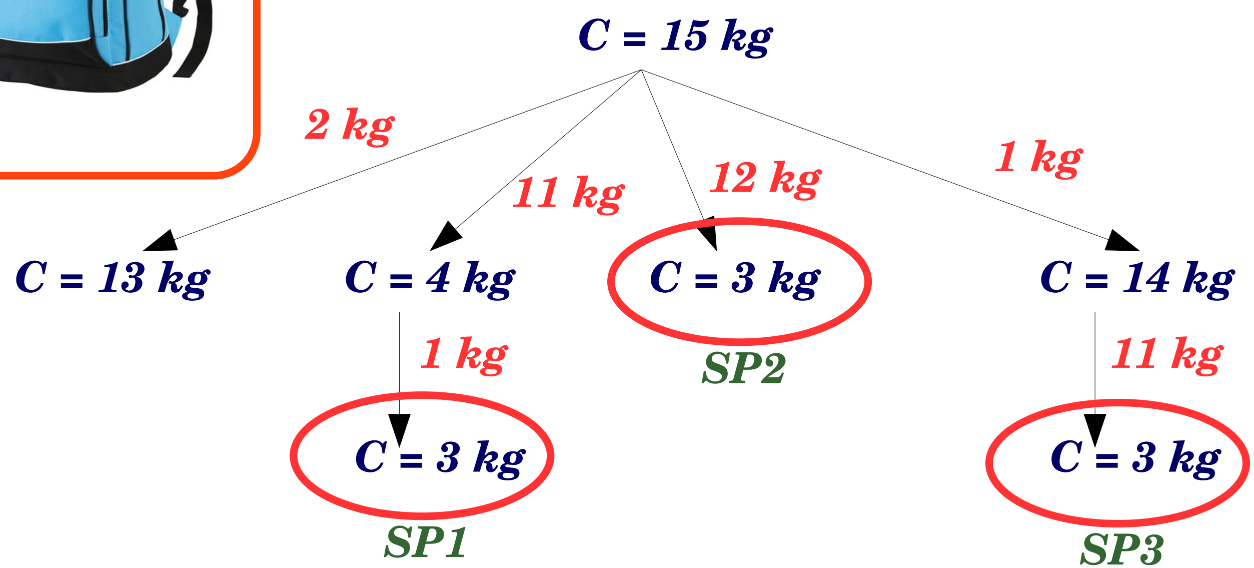
S'agit t-il d'un chevauchement ?

Problème du « *Sac à Dos* » – Version 0/1

⇒ *Chevauchement de sous-problèmes :*

Objet 1	2 kg	2 DA
Objet 2	11 kg	6 DA
Objet 3	12 kg	4 DA
Objet 4	1 kg	5 DA

C = 15 kg



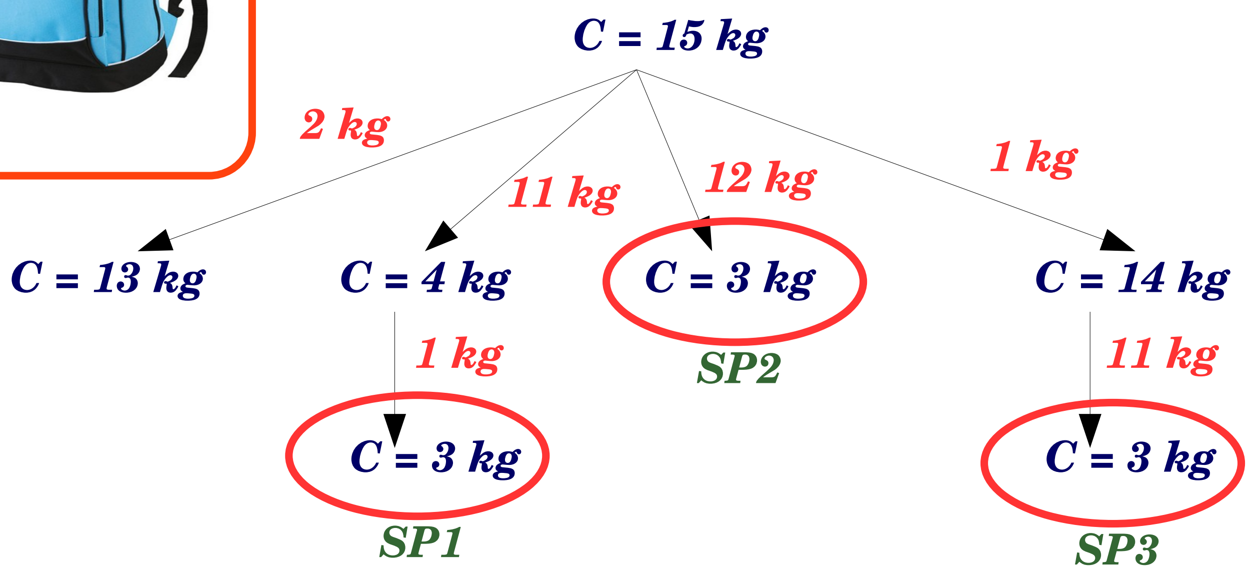
***NON !** Uniquement SP1 et SP3 sont équivalents*

Problème du « *Sac à Dos* » – Version 0/1

⇒ *Chevauchement de sous-problèmes :*

Objet 1	2 kg	2 DA
Objet 2	11 kg	6 DA
Objet 3	12 kg	4 DA
Objet 4	1 kg	5 DA

C = 15 kg



Remarque : ***Un sous-problème est caractérisé par les objets choisis.***

Problème du *Sac à dos* – Solution naïve

Méthode naïve :

```
int SAD_Naif(int C, int[] P, int[] G, boolean[] Choisis){
    int R = 0 ;
    for (int i = 0 ; i < P.length ; i++){
        if(P[i] ≤ C && ! Choisis[i]){
            Choisis[i] = true ;
            R = Math.max(R , G[i] + SAD_Naif(C - P[i], P, G, Choisis)) ;
            Choisis[i] = false ;
        }
    }
    return R ;
}
```

Problème du *Sac à dos* – Solution avec la PRD

Méthode Dynamique :

```
int SAD_PRD(int C, int[] P, int[] G, boolean[] Choisis){
    String cle = Arrays.toString(Choisis) ;
    if(Memo.containsKey(cle))  return Memo.get(cle) ;
    int R = 0 ;
    for (int i = 0 ; i < P.length ; i++){
        if(P[i]  ≤  C && ! Choisis[i]){
            Choisis[i] = true ;
            R = Math.max(R, G[i] + SAD_PRD(C - P[i], P, G, Choisis)) ;
            Choisis[i] = false ;
        }
    }
    Memo.put(cle, R) ;
    return R ;
}
```

Question :

Combien de sous-problèmes différents peut-il y avoir pour le *Sac à Dos 0/1* ?