Algorithmes Gloutons

Greedy Algorithms

Dr. Mahfoud Houari

mahfoud.houari@gmail.com

Université Abou-Bakr Belkaïd - Tlemcen

2022/2023

Exemple Introductif 1:

Problème de *Partition Équitable*

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

Exemple 1:

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

Exemple 1:

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

Exemple 1:

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

Exemple 1:

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

Exemple 1:

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

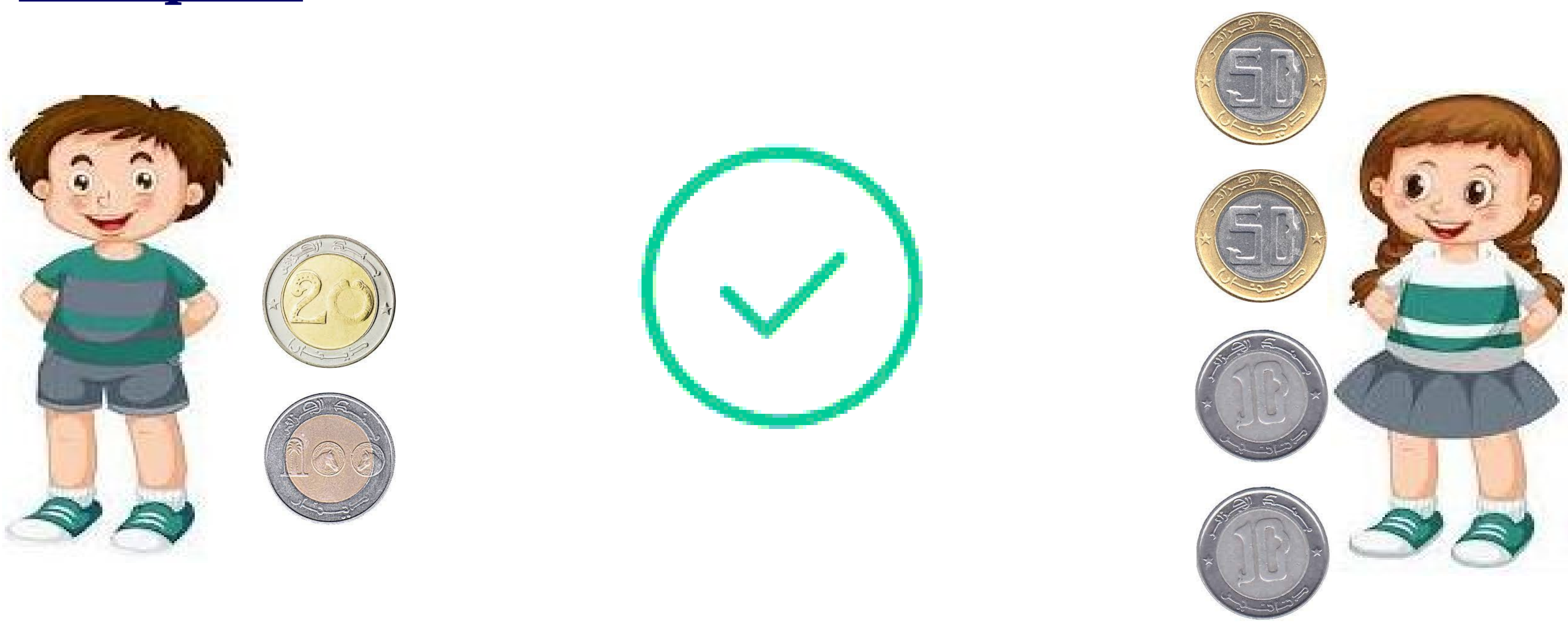
Exemple 1:

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

Exemple 1:



Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

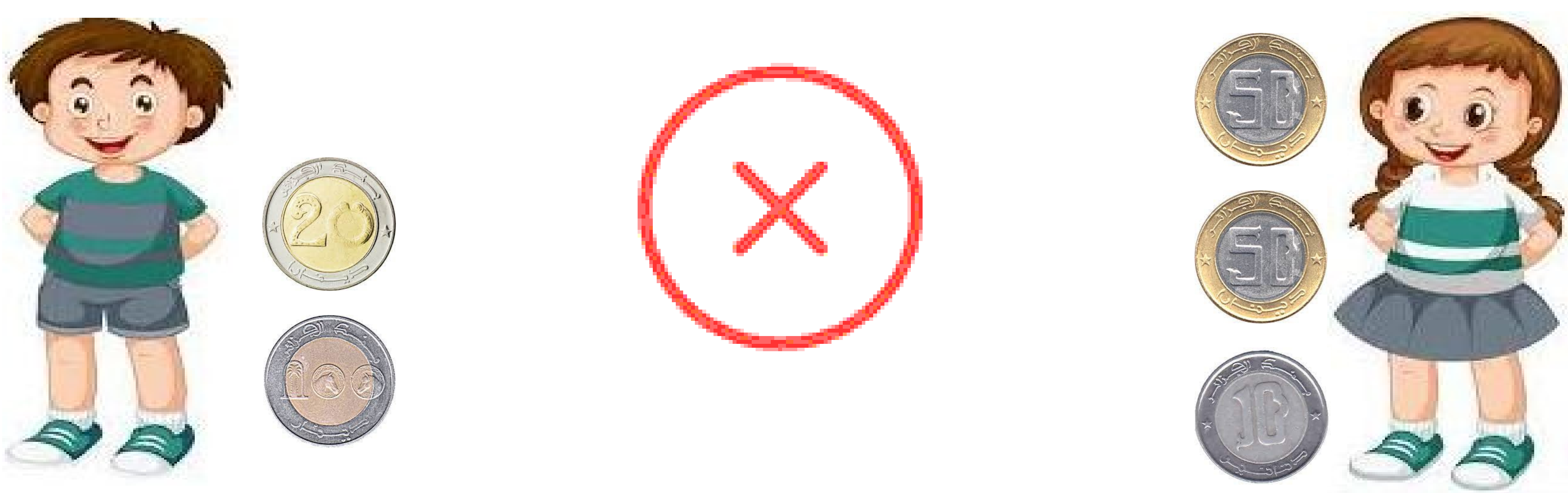
Exemple 2:

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

Exemple 2:



Est-ce le critère qui est mal conçu ?

Problème de « *Partition Équitable* »

Description :

Soit donné un ensemble ***S*** de pièces de monnaie, le but est de tester si ***S*** peut être partitionné en deux sous-ensembles ***S1*** et ***S2*** telle que la somme des pièces du ***S1*** et égale à celle du ***S2***.

Exemple 2:

NON : c'est l'instance qui n'a pas de solution

Exemple Introductif 2:

Problème du « *Rendu de monnaie* »

Version 1

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

Description :

Un distributeur de boisson 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**.

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

Solution exhaustive:

```
int RM_Naife(int M, int[] P){
    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 ;
}
```

WCTC : O($|P|^{M}$)
WCSC : O($|M|$)

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

Solution dynamique descendante:

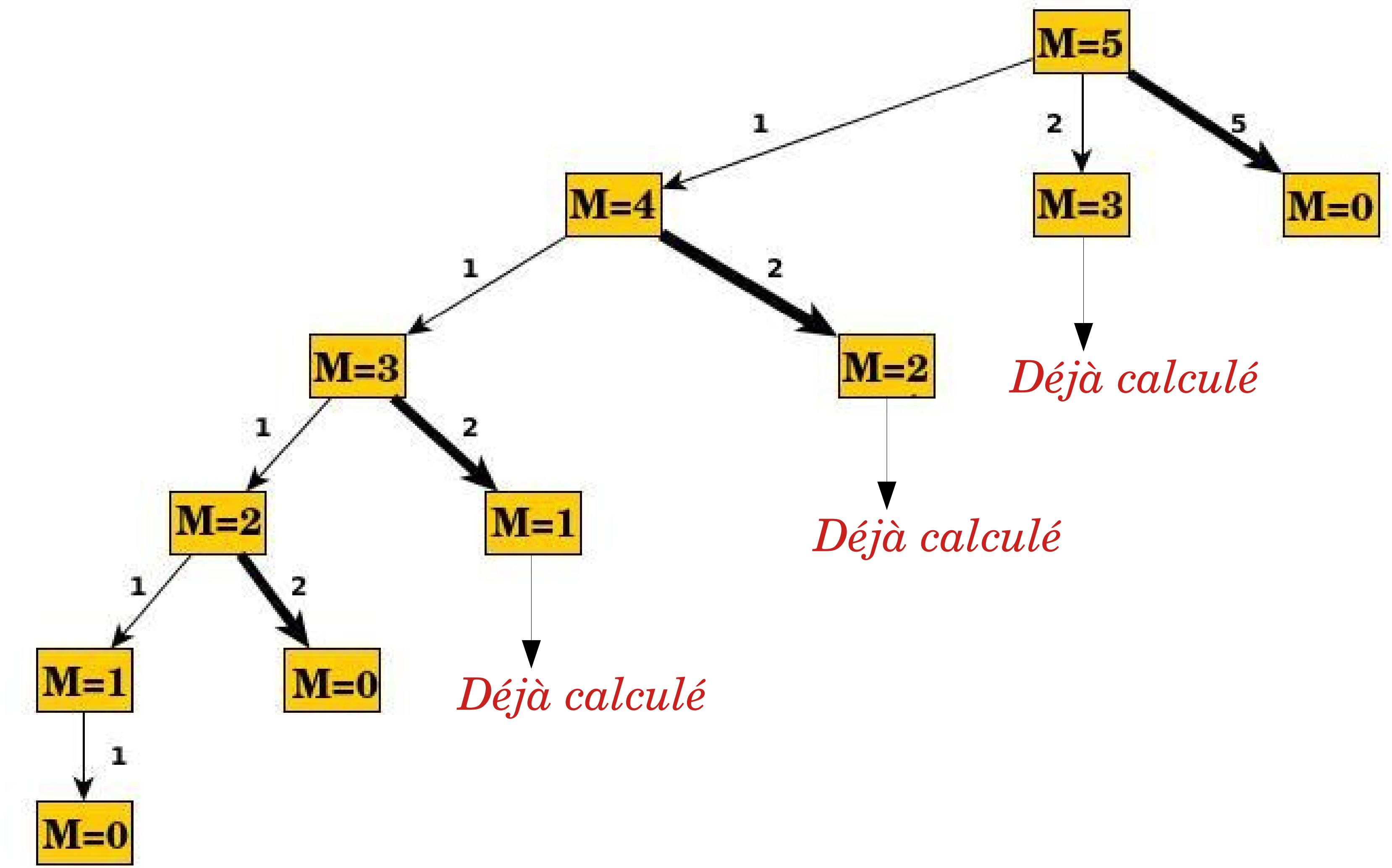
```
HashMap<Integer, Integer> memo = new HashMap<Integer, Integer>();
memo.put(0, 0) ;

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 ;
}
```

WCTC : $O(M.|P|) = O(M)$
WCSC : $O(M)$

Peut-on améliorer cette solution ?

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

Arbre des Appels Récursifs (M=5) :



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

L'amélioration consiste à ***sélectionner*** la plus grande pièce à chaque itération que de considérer toutes les pièces.

Critère de sélection glouton (choix glouton)

Approche *gloutonne*

- Définir un ***choix glouton*** selon la nature du problème.
- Choisir des sous-solutions ***localement optimales*** pour arriver à une solution ***globalement optimale***.
- Analyser une ***seule branche*** qui mène à la solution optimale.
- (+) Plus efficace par rapport à la PRD.
- (-) Choix glouton souvent difficile à prouver
- (-) L'exactitude de la solution dépend du choix glouton :

Choix optimal $\Rightarrow$ ***Solution exacte***

Choix non optimal $\Rightarrow$ ***Solution approchée***

Solution gloutonne :

Algorithme glouton itératif (*Version simple*):

```
P = trieDecroissant(P);

int RM_Glouton(int M, int[] P){
    int Nb = 0 ;
    for (int i = 0 ; i < P.length ; i++){
        if(P[i] <= M){
            Nb += M / P[i] ; M = M % P[i] ;
        }
    }
    return Nb ;
}
```

WCTC : O(????).

WCSC : O(????).

Solution gloutonne :

Algorithme glouton itératif (*Version simple*):

```
P = trieDecroissant(P);

int RM_Glouton(int M, int[] P){
    int Nb = 0 ;
    for (int i = 0 ; i < P.length ; i++){
        if(P[i] <= M){
            Nb += M / P[i] ; M = M % P[i] ;
        }
    }
    return Nb ;
}
```

WCTC : $O(|P|.\log_2(|P|) + |P|)$.

WCSC : $O(|P|) + O(1)$.

Solution gloutonne :

Algorithme glouton itératif (*Version simple*):

```
P = trieDecroissant(P);

int RM_Glouton(int M, int[] P){
    int Nb = 0 ;
    for (int i = 0 ; i < P.length ; i++){
        if(P[i] <= M){
            Nb += M / P[i] ; M = M % P[i] ;
        }
    }
    return Nb ;
}
```

Complexités statiques

WCTC : $O(|P|.\log_2(|P|) + |P|)$.

WCSC : $O(|P|) + O(1)$.

Solution gloutonne :

Algorithme glouton itératif (*Version simple*):

```
P = trieDecroissant(P);

int RM_Glouton(int M, int[] P){
    int Nb = 0 ;
    for (int i = 0 ; i < P.length ; i++){
        if(P[i] <= M){
            Nb += M / P[i] ; M = M % P[i] ;
        }
    }
    return Nb ;
}
```

WCTC : O(|*P*|).

WCSC : O(*1*).

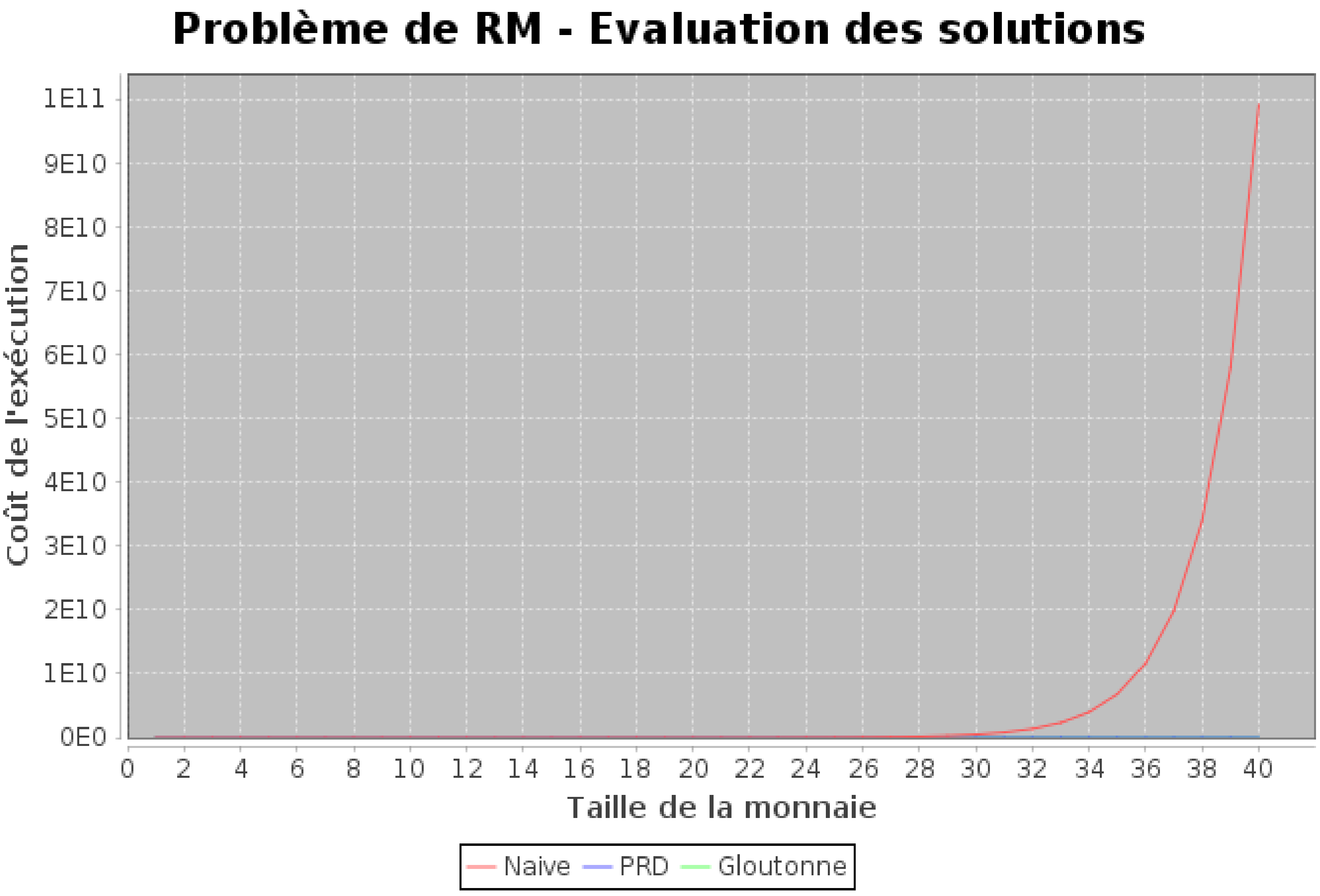
Solution gloutonne – *Récapitulatif*

- L'approche dynamique est exécutée en ***O(M)***. Donc, plus que la valeur de ***M*** est grande plus que le temps augmente.

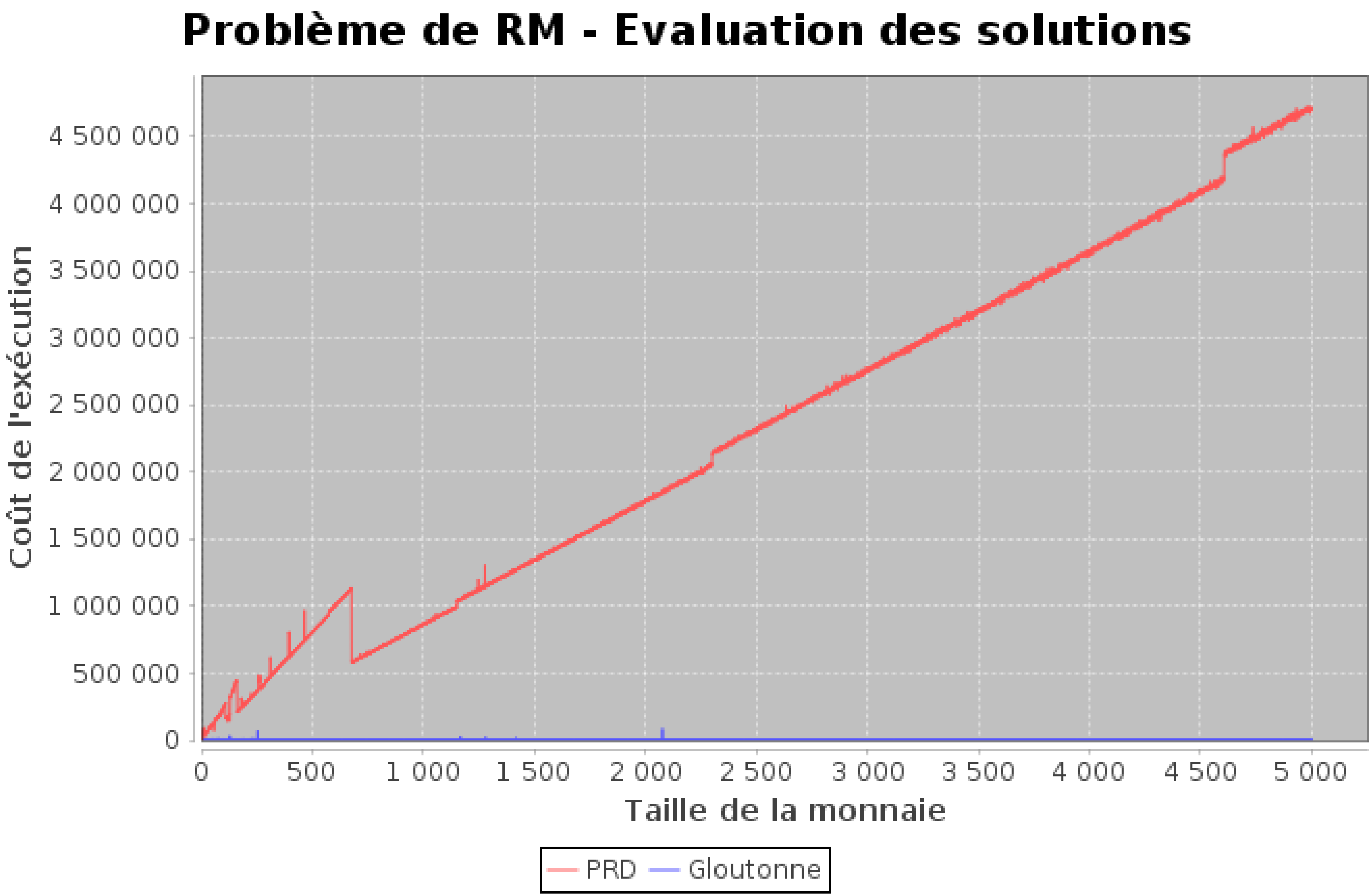
- L'approche gloutonne est exécutée en ***O(P)***. Vu que le système monétaire ne change pas, donc le temps est constant quelque soit la valeur de ***M***.

- L'approche dynamique nécessite une complexité spatiale bornée par ***O(M)*** alors que celle de sa version gloutonne est bornée par ***O(1)***.

Rendu de monnaie – Comparatif de solutions



Rendu de monnaie – Comparatif de solutions



Solution gloutonne

Exercice :

Modifiez la méthode ***RM_Glouton*** pour qu'elle retourne le gain maximal et aussi bien les pièces à retourner pour assurer ce gain.

Solution gloutonne :

Algorithme glouton itératif (*Version complète*):

```
P = trieDecroissant(P);
HashMap<Integer, Integer> RM_Glouton(int M, int[] P){

    HashMap<Integer, Integer> Sol = new HashMap<Integer, Integer> ;

    for (int i = 0 ; i < P.length ; i++){

        if(P[i] <= M){

            Sol.put(P[i], M / P[i]);

            M = M % P[i] ;

        }

    }
    return Sol ;
}
```

WCTC : O(|P|).

WCSC : O(|P|).

Solution gloutonne

Le choix glouton considéré est-il optimal pour tous les systèmes monétaires ?

Solution gloutonne

Le choix glouton considéré est-il optimal pour tous les systèmes monétaires ?

Réponse : ***Il est optimal uniquement pour le système*** [1, 2, 5, 10, 20, 100, 200]**.**

Contre-exemple : ***Essayons avec le système*** [200,100,50,20] ***pour*** *M = 110***.**

⇒ **Solution optimale : 1.50 + 3.20**

⇒ **Solution trouvée avec le critère : 1.100**

Solution gloutonne

Le choix glouton considéré serait-il optimal si on considère la disponibilité des pièces ?

Contre-exemple:

- L'ensemble de pièces de monnaie **{20, 10, 5, 2, 1}**.
- Les disponibilités des pièces sont **{ 1, 2, 1, 4, 0}**.
- **M = 21**.

⇒ **Solution optimale :** **1**.10 + **1.5** + **3.2** = **5** pièces.
⇒ **Solution gloutonne :** **1.20** = **1** pièce ⇒ *Solution incorrecte*.

Peut-on faire du Backtracking pour améliorer ce choix ?

Conception d'une solution gloutonne pour le problème d'*ordonnancement de tâches*

Problème d'*ordonnancement de tâches*

- Soit une salle de conférence qui peut être allouée à la fois à une et une seule tâche (cours, workshop, réunion,...).

- Soit un ensemble de tâches $t_1, t_2, \ldots, t_n$ ayant chacune une date de début d_i et une date de fin f_i ($f_i > d_i$).

Objectif : Allouer la salle à un nombre maximum de tâches.

Exemple :

t_i	1	2	3	4	5	6	7	8	9	10	11
d_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Quelques solutions optimales : (t_1, t_4, t_8, t_{11}) **ou** (t_2, t_4, t_9, t_{11}).

Solutions non optimales : (t_1, t_6, t_{11}), (t_3, t_7, t_{11}), (t_3, t_9, t_{11}).

Problème d'*ordonnancement de tâches*

Quelques choix gloutons possibles :

1. La tâche qui commence le plus tôt.

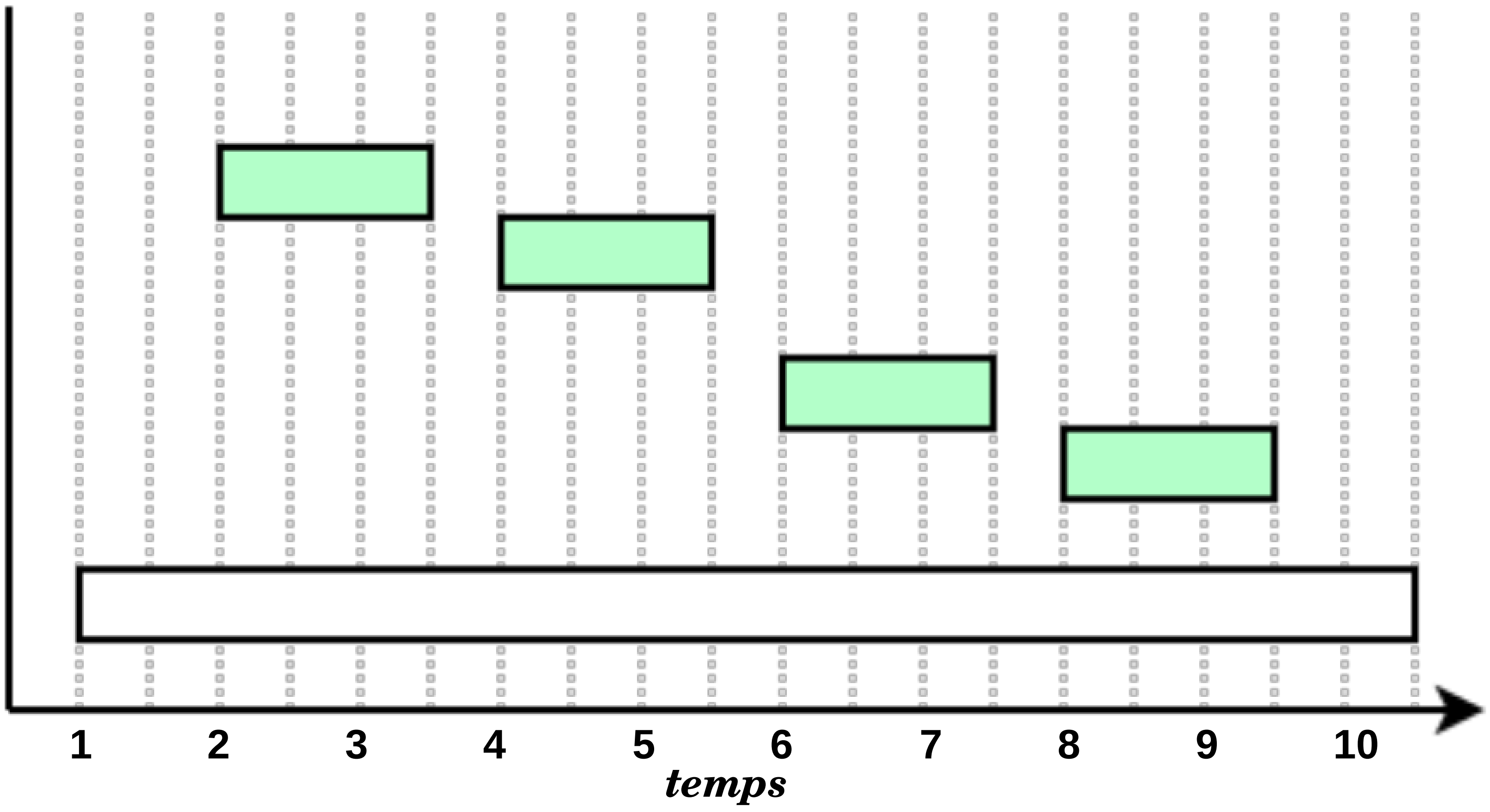
2. La tâche qui occupe le moins de temps (f_i - d_i est plus petit).

3. La tâche qui termine le plus tôt.

4. La tâche qui commence le plus tard (***à vous de le traiter***).

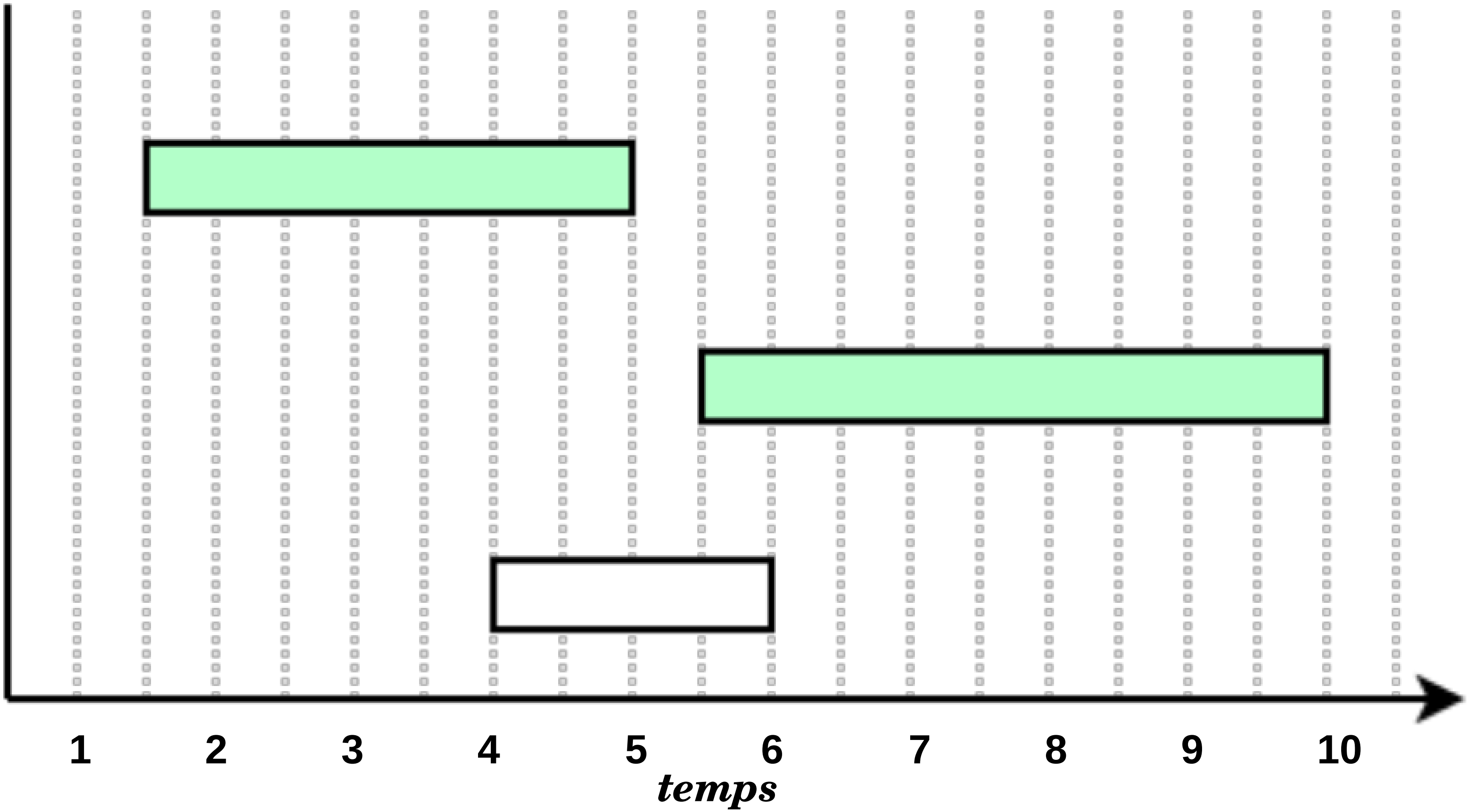
Problème d'*ordonnancement de tâches*

1. La tâche qui commence le plus tôt : (*non optimal*)



Problème d'*ordonnancement de tâches*

2. La tâche qui occupe le moins de temps : (*non optimal*)



Problème d'*ordonnancement de tâches*

3. La tâche qui termine le plus tôt : (*critère optimal*)

Il n'existe aucun contre-exemple pour ce critère

Problème d'*ordonnancement de tâches*

3. La tâche qui termine le plus tôt : (*critère optimal*)

Il n'existe aucun contre-exemple pour ce critère

Exemple d'application:

Pré-traitement : trier les tâches selon leurs dates de fin.

t_i	1	2	3	4	5	6	7	8	9	10	11
d_i	1	3	0	5	3	5	6	8	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Solution gloutonne : (t_1, t_4, t_8, t_{11}).

Solution gloutonne – Selon le 3ème choix

Algorithme glouton itératif:

```
//Définir la classe Tache
//trier en ordre croissant le tableau T selon les dates de fin

ArrayList<Tache> OT_Glouton(Tache[] T){
        ArrayList<Tache> Sol = new ArrayList<Tache>();
        int derniere_fin = 0;

        for(int i = 0 ; i < T.length ; i++){
              if(T[i].debut >= derniere_fin){
                    Sol.add(T[i]);
                    derniere_fin = T[i].fin;
              }
        }

        return Sol;
}
```

WCTC : **O(?) + O(?).**

WCSC : **O(?) + O(?).**

Solution gloutonne – Selon le 3ème choix

Algorithme glouton itératif:

```
//Définir la classe Tache
//trier en ordre croissant le tableau T selon les dates de fin

ArrayList<Tache> OT_Glouton(Tache[] T){
        ArrayList<Tache> Sol = new ArrayList<Tache>();
        int derniere_fin = 0;

        for(int i = 0 ; i < T.length ; i++){
                if(T[i].debut >= derniere_fin){
                        Sol.add(T[i]);
                        derniere_fin = T[i].fin;
                }
        }

        return Sol;
}
```

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

WCSC : $O(|T|)$.

Problème de « *Partition Équitable* »

Exercice :

1. Proposez une méthode gloutonne pour le problème de ***partition équitable***.
2. Calculez la *WCTC* et *WCSC* de votre méthode.