Rapport de Projet

Confitures

LU3IN003

Algorithmique

Licence d'informatique L3

Sorbonne Université

Samy NEHLIL – Amel BELDJILALI

Année universitaire 2021-2022

I. Partie théorique:

A. Algorithme 01:

Soit m(s) le nombre minimum de bocaux pour une quantité S de confiture et un tableau de capacités V.
On définit les problèmes intermédiaires suivants:
Etant donné un entier s et un entier i$\in \{1,\ldots, k\}$, on note m(s,i) le nombre minimum de bocaux nécessaire pour une quantité totale s en ne choisissant des bocaux que dans le système de capacités V[1], V[2], … ,V[i].
Initialisation de la récursion:

- m(0, i) = 0 ∀i ∈ {1, . . . , k} : il est possible de réaliser la capacité totale 0 avec 0 bocal.
- m(s, 0) = +∞ ∀s ≥ 1 : il n'est pas possible de réaliser la capacité s ≥ 1 sans utiliser au moins un bocal.
- m(s, i) = +∞ ∀i ∈ {1, . . . , k} ∀s < 0 : il n'est pas possible de réaliser une capacité négative.

Réponse 01:

a) La valeur de m(s) en fonction des valeurs m(s, i) définies dans la section précédente est: **m(s) = min {m(s, i) \ i ∈ {1, . . . , k}}.**

b) Montrons la relation de récurrence suivante pour tout i ∈ {1, . . . , k}:

$$m(s,i) = \begin{cases} 0 \text{ si } s = 0 \\ \min\{m(s, i-1), m(s - V[i], i) + 1\} \text{ sinon} \end{cases}$$

Prouvons que P est vérifiée pour tout i ∈ {1, . . . , k} par récurrence :

- Cas de base: i = 1
  - Si s = 0: m(0, 1) = 0 d'après les hypothèses
  - Si s < 0: m(s, 1) = min{m(s, 0), m(s − 1, 1) + 1} = min{+∞, +∞} = +∞ (vérifiée).
  - Si s > 0: m(s, 1) = min{m(s, 0), m(s − 1, 1) + 1} = min{+∞, m(s, 1)} = m(s, 1) (vérifiée)
- Hypothèse de récurrence: P(i) est vérifiée pour un i ∈ {1, . . . , k}.
- Induction:
  - Si s = 0: m(0, i + 1) = 0 d'après les hypothèses
  - Si s < 0: m(s, i + 1) = min{m(s, i), m(s − V[i + 1], i + 1) + 1} = min{+∞, +∞} = +∞ (vérifiée).
  - Si s > 0: m(s, i + 1) = min{m(s, i), m(s − V[i + 1], i + 1) + 1}?
    - Si S - V[i + 1] > 0: Alors on peut utiliser au moins un bocal de capacité V[i + 1], d'où m(s, i + 1) = m(s - V[i + 1], i + 1) + 1
    - Si S - V[i + 1] > 0: Alors le bocal i + 1 a une capacité plus grande que la quantité S dans m(s, i + 1) = m(s, i)

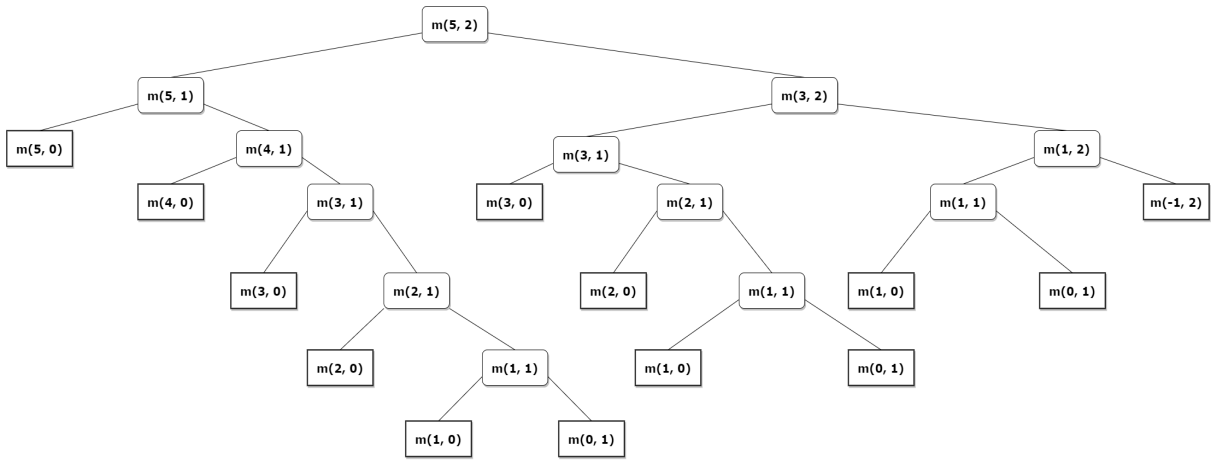
Réponse 02:

Pseudo-algorithme basé sur la relation de récurrence précédente et calculant m(s):

```
Algorithme AlgoRec(s,i,v):
 if s=0:          //si s==0 alors retourner 0
        return 0
 if i=0:          //si i==0 alors retourner l'infini
        return infini
 if s<0:          //si s<0 alors retourner l'infini
        return infini
 else:            //sinon appliquer la relation de récurrence
        return min(AlgoRec(s,i-1,v), AlgoRec(s-v[i],i,v)+1)
```

Réponse 03:

L'arbre des appels récursifs de l'algorithme pour S = 5 et k = 2 avec V[1] = 1 et V[2] = 2:



Réponse 04:

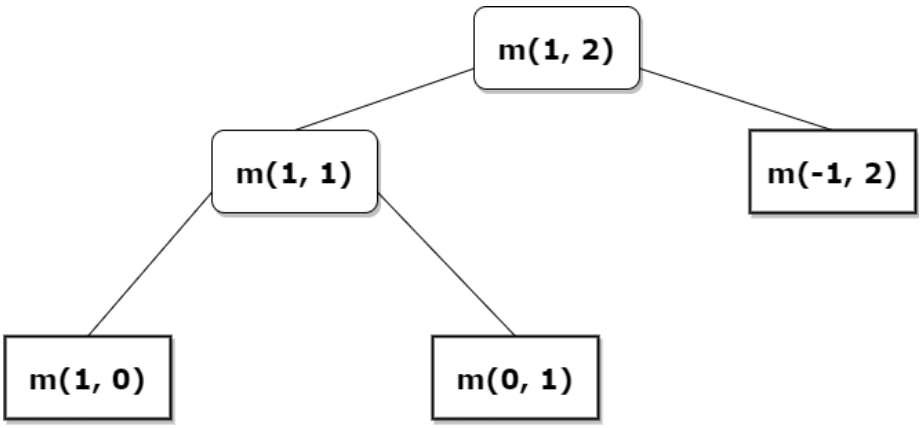
Dans l'arbre des appels récursifs représenté dans la question précédente, la valeur m(1, 1) est calculée trois (3) fois.

Plus généralement, si k = 2 avec V [1] = 1 et V [2] = 2 et pour S impair (On pose S = 2p + 1, $p \in N$), la valeur m(1, 1) est calculée $\left[\frac{S}{2}\right]$ fois.

<u>Preuve:</u> Prouvons que la valeur m(1, 1) est calculée $\left[\frac{S}{2}\right]$ fois par récurrence sur p:

Soit: Q(p): "La valeur m(1, 1) est calculée $\left[\frac{S}{2}\right]$ fois pour S = 2p + 1, $\forall\ p \in N$"

- Cas de base: p = 0
  p = 0 ⇒ S = 1, on a donc l'arbre des appels récursifs suivant:



m(1, 1) est calculée une seule (1) fois donc Q(0) est vérifiée (car $\left[\frac{1}{2}\right] = 1$).

- Hypothèse de récurrence: Q(p) est vérifiée pour un $p \in N$
- Induction: Prouvons que Q(p+1) est vérifiée:

  On a: m(2(p+1)+1, 2) = m(2p+3, 2)

  = min$\{m(2p + 3, 1), m(2p + 1, 2)\}$

  (selon la formule de récurrence)

  = min$\left\{2p + 3, \left[\frac{2p+1}{2}\right] + 1\right\}$

  (selon l'hypothèse de récurrence)

  = $\left[\frac{2p+1}{2}\right] + 1$

  = $\left[\frac{2p+1}{2}\right] + \frac{2}{2}$ …….(1)

  = $\left[\frac{2p+3}{2}\right]$ ..... (2)

  Donc Q(p) est vérifiée pour tout p$\in N$.

Remarque:

1. $\left[\frac{S}{2}\right]$ représente la partie entière supérieure de la division de S par 2
2. Pour passer de l'étape (1) à (2), on a utilisé la propriété suivante:

   Si m = $[n]$ alors n-1 < m $\leq$ n

B. Algorithme 02:

Améliorons l'algorithme précédent en ne calculant pas la même valeur m(s, i) plusieurs fois:

Réponse 05:

a) L'ordre de remplissage des cases du tableau M selon la formule de récurrence est le suivant:

```
Pour i allant de 0 à k:
        Pour j allant de 0 à S:
                M[i, j] ← m(i, j)
        FPOUR
FPOUR
```

C'est-à-dire par colonne.

b) L'algorithme itératif:

```
AlgoOptimisé(S, i, V):
        Pour j allant de 0 à S:
                Pour k allant de 0 à i:
                        M[j, k] ← + ∞
                FPOUR
        FPOUR
        Pour j allant de 0 à i:
                Pour k allant de 0 à S:
                        Si k-V[j]<0 alors
                                m2 ← + ∞
                        Sinon
                                m2 ← M[k-V[j],j]
                        M[k,j] ←  min(M[k,j-1],m2+1)
        FPOUR
```

c) Analyse de la complexité temporelle et spatiale de l'algorithme:

   i) Complexité temporelle: O(S.k)

Car il nous faut S.k itérations pour remplir la matrice M.

   ii) Complexité spatiale: O(S.k)

Car on a (S+1).(k+1) cases dans la matrice M.

Réponse 06:

a) Les modifications à effectuer dans l'algorithme précédent:

```
AlgoOptimisé(S, i, V):
        Pour j allant de 0 à S:
                Pour k allant de 0 à i:
                        M[j, k] ← + ∞
                FPOUR
        FPOUR
        Pour j allant de 0 à i:
                Pour k allant de 0 à S:
                        Si k-V[j]<0 alors
                                m2 ← + ∞
                        Sinon
                                m2 ← M[k-V[j],j]
                        M[k,j] ←  min(M[k,j-1],m2+1)
        FPOUR
```

Complexité temporelle : O(k.s.k)

Complexité Spatiale : O(k.s.k)

b) Le pseudo-code de l'algorithme retour:

```
AlgorithmeRetour ( ) :
//retourner A
 A = [ ]
//initialisation
 Pour t = 1 .. k faire:v
      A[t] <- 0
 Tant que (Vrai) faire:
          Si(S-V[i]>=0) ET (M[S,i] == M[S-V[i], i]+1) alors :
                  A[ i ] = A[ i ] + 1
                  s = s - V [ i ]
          Sinon:
                  i = i - 1
                  Si (i < 0):
                     break
 return A
```

Complexité temporelle : O(k.s)

Complexité Spatiale : O(k.s + k)

Réponse 07:

C. Algorithme 03:

Cas particulier et algorithme glouton:

Réponse 08:

Ecriture d'un algorithme glouton pour résoudre le problème des bocaux de confiture:

```
Algorithme AlgoGlouton(S: int, k, V):
    indice_capacites = k
    nb = 0

    while indice_capacites >= 0:
        rempli = S div V[indice_capacites]
        nb += rempli
        encore_a_remplir = S modulo V[indice_capacites]
        S = encore_a_remplir
        indice_capacites = indice_capacites  - 1

    return nb
```

Un système de capacité est dit glouton-compatible si, pour ce système de capacité, l'algorithme glouton produit la solution optimale quelle que soit la quantité totale S.

Réponse 09:

Montrons qu'il existe des systèmes de capacités qui ne sont pas glouton-compatibles :
Par contre exemple,

On vérifie que le système de capacité V = [1,3,4] n'est pas canonique :

Pour S = 6, l'algorithme glouton propose d'utiliser 3 bocaux, (1, 0, 2), alors qu'il est possible de n'en utiliser que 2 : (0, 2, 0).
En effet, la méthode gloutonne va d'abord choisir le bocal de plus haute valeur en dessous du rendu, soit celui de capacité 4. Il restera ensuite 2 capacités à utiliser, la seule possibilité étant d'utiliser 2 bocaux de capacité 1.

On aura donc 3 bocaux au lieu de 2, la solution n'est pas optimale.

Réponse 10:

Montrer que tout système de capacité V avec k = 2 est glouton-compatible (on rappelle qu'on a toujours V [1] = 1).

Soit S une capacité

soit v = (1,c1) le système à capacités

On a que le principe de l'algorithme glouton pour la résolution du problème de bocaux à remplir est basé sur la division euclidienne des entiers naturels.

En effet, pour c = (1,c1), le résultat produit par l'algorithme glouton est bien le résultat de la division euclidienne de S par c.

Soit k = (q,r) le résultat de cette division, on a donc :

$$S = q*c1 + r \text{ avec } r < c1$$

l'algorithme glouton retourne le résultat suivant : q + r bocaux à utiliser qui est optimal (car propriété de la division euclidienne.

II. Mise en oeuvre:

A. Implémentation

Entrées-Sorties

Trois programmes

trois programmes correspondant aux trois algorithmes I, II et III

Programme I

```python
#Algorithme 1 récursif
import sys
"""
Algorithme récursif pour le calcul du nombre minimum de bocaux à remplir
par la quantité s de confiture en utilisant i cases du système de capacités v
"""
def AlgoRec(s,i,v):
  if s==0:              #si s==0 alors retourner 0
    return 0
  if i==0:              #si i==0 alors retourner l'infini
    return sys.maxsize
  if s<0:               #si s<0 alors retourner l'infini
    return sys.maxsize
  else:                 #sinon appliquer la relation de récurrence
    return min(AlgoRec(s,i-1,v), AlgoRec(s-v[i-1],i,v)+1)
```

Programme II

```python
[17] #Algorithme 2 : utilisation de la matrice M
     """
     algorithme iteratif AlgoDyn (en pseudo-code) qui determine le
     nombre minimum de bocaux necessaires pour une quantite de confiture S
     """
     import numpy as np
     import sys
     def AlgoDyn(S,i, V):
       #m = math.inf
       M = np.zeros((S+1,i+1),int)
       #initialisation de la matrice M avec des valeurs infinies
       for k in range(S+1):
         M[k,0]= sys.maxsize
       for j in range(1,i+1):
         for k in range(1,S+1):
           try:
             if k-V[j-1]<0:
               m2 = sys.maxsize
             else:
               m2 = M[k-V[j-1],j]
             M[k,j]= min(M[k,j-1],m2+1)
           except e:
             print(e)
       return M[S,i]
```

Programme III

```
#ALGORITHME 3 : méthode gloutonne
"""
Détermine les bocaux (et leur nombre) à choisir pour remplir la quantité
passée en argument.
Utilise la division euclidienne et le modulo de façon à pouvoir
traiter tous les cas.
"""
def AlgoGlouton(S: int,k, V):
    indice_capacites = k-1
    a = 0
    while indice_capacites >= 0:
        rendu = S // V[indice_capacites]
        a += rendu
        encore_a_rendre = S % V[indice_capacites]
        S = encore_a_rendre
        indice_capacites -= 1
    return a
```

Tests de fonctionnement

Pour les tests de fonctionnement des trois programmes, nous avons choisi le jeu d'essai suivant :

```
#Test AlgoRec(s,i,v)
v = [1,2,5,10,20,50,100,200]
for s in range(5,100,22):
  print("S = ",s)
  for k in range(1,9):
    print("le nombre de bocaux minimum pour remplir {} dg de confiture en utilisant {}
capacités du système v est : {}".format(s,k,AlgoRec(s,k,v)))
```

L'exécution de ce programme de test sur les trois algorithmes donne exactement les mêmes résultats (ceux attendus).

B. Analyse de complexité expérimentale

Mesure de complexité temporelle:

Pour mesurer la complexité temporelle des trois algorithmes I, II et III, nous avons choisi d'utiliser la métrique de temps d'exécution, cette dernière est obtenue dans les deux points d'entrée et de sortie de chaque algorithme.

Représentation de la complexité temporelle:

Quant à la représentation des graphes représentant les complexités temporelles des algorithmes, nous avons opté pour la bibliothèque de python matplotlib qui offre plusieurs fonctions.

Programmes et fonctions utilisés:

Fonction SystemExpo(d,k) :

//cette fonction crée un système Expo de capacité V de la forme

V [1] = 1 **V [2] = d** **V [3] =** d^2 **: : :** **V [k] =** d^{k-1}

Fonction plot_complexity(d,k):

//Cette fonction varie la capacité S et calcule pour chacune le nombre de bocaux minimum dans le système SystemExpo(d,k) selon l'algorithme choisi, puis affiche le graphe de complexité temporelle

Pour générer des tests avec différentes valeurs pour k, d, s, on exécute la fonction suivante:

```
def varier(smax,kmax, numAlgo):
  for d in range(2,5):
    for k in range(2,kmax):
      plot_complexity(d,k, smax,numAlgo)
```

Résultats de l'analyse expérimentale:

Algorithme I:

l'algorithme récursif a une complexité temporelle théorique de :

Comme indiqué sur le cahier des charges, nous nous sommes arrêtés dès que la complexité temporelle de l'algorithme a dépassé 1 minute, le résultat est donc le suivant:

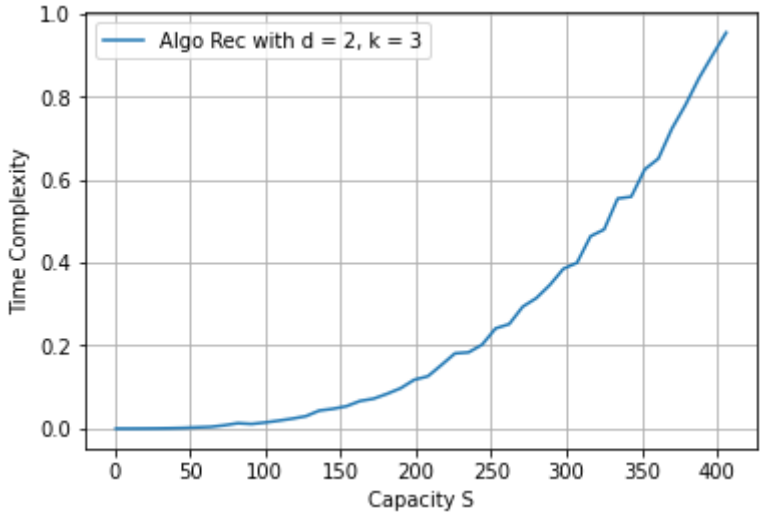
<u>Résultat:</u>

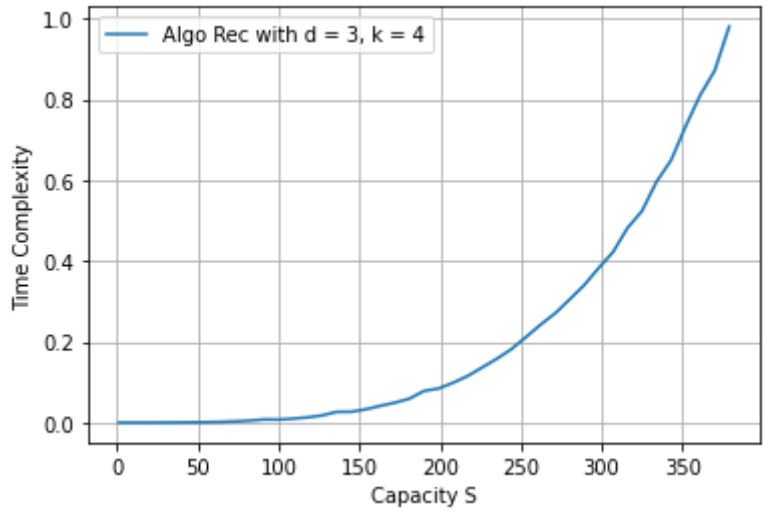
Dans ce qui suit les résultats trouvés pour :

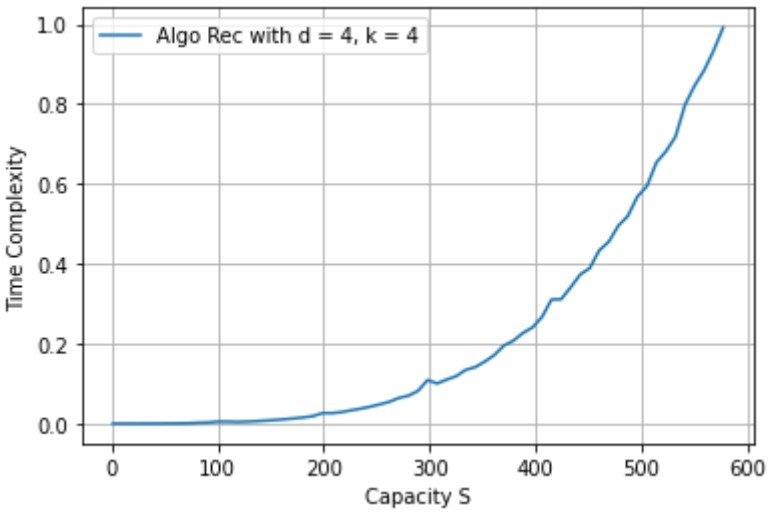
d = 2,3,4 ;

k = 2 .. 10 ;

S = 2 .. 600 (avec un pas de 9)







Observation: On remarque que la courbe représentant la complexité temporelle de la fonction récursive évolue de façon exponentielle, plus S grandit et plus le temps d'exécution est plus grand.
Critique: Le résultat ci-dessus est un résultat attendu et justifié, car le temps d'exécution de la fonction récursive croît rapidement, en effet, ceci est dû au fait que des mêmes valeurs sont calculées plusieurs fois, ce qui augmente considérablement le temps d'exécution de l'algorithme.

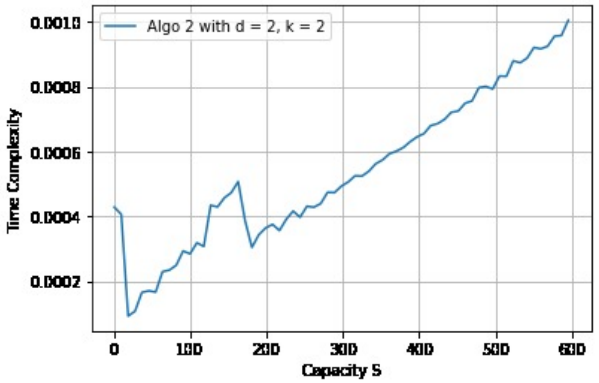
Algorithme II:
l'algorithme dynamique a une complexité temporelle théorique de : O(k.s)
Résultat:

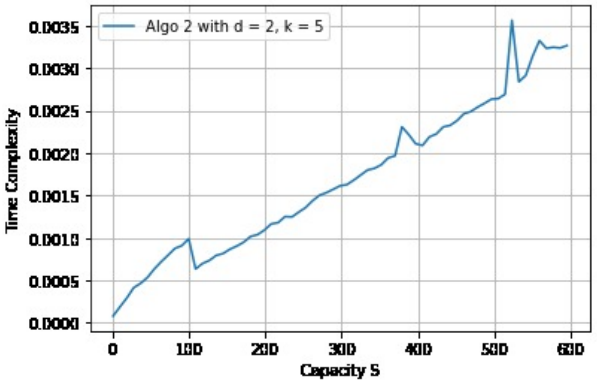
Dans ce qui suit les résultats trouvés pour :

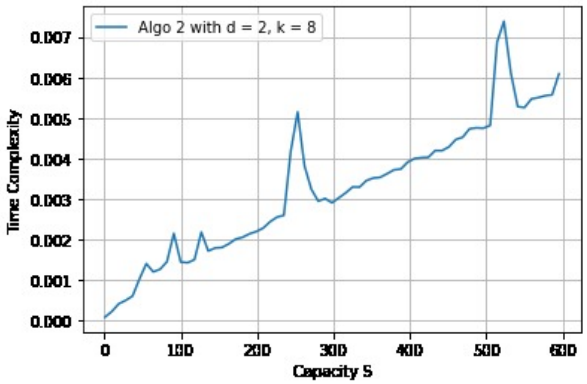
pour d = 2,3,4 :

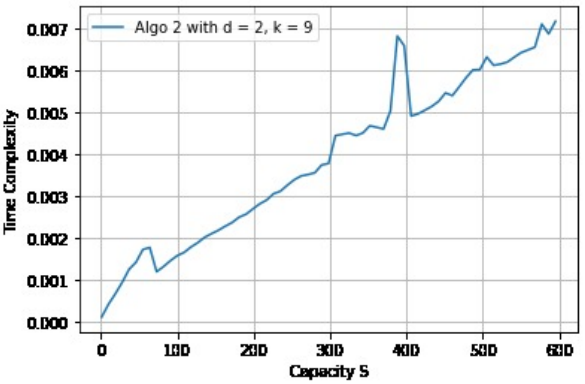
pour k = 2 .. 10 ;

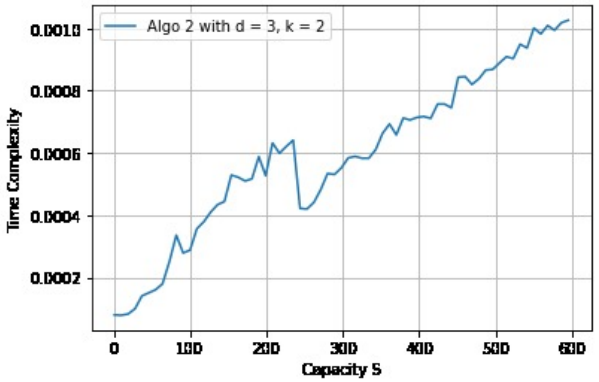
pour S = 2 .. 600 (avec un pas de 9);

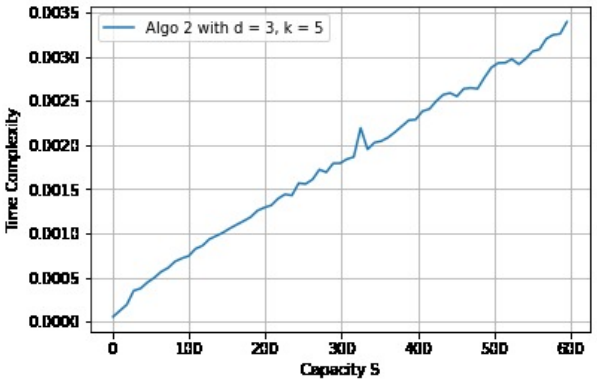


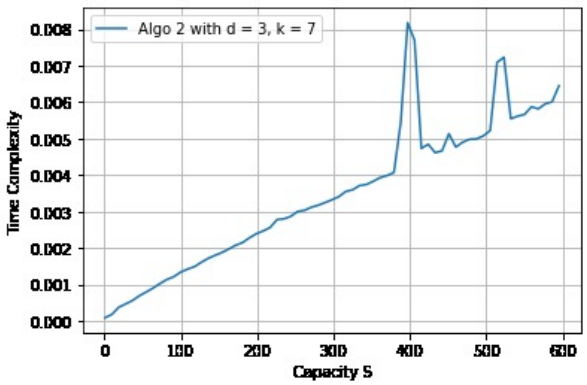


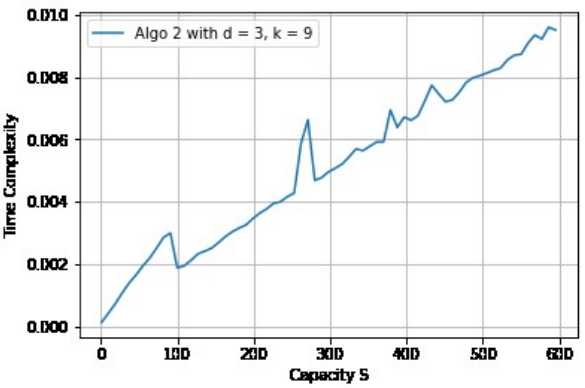


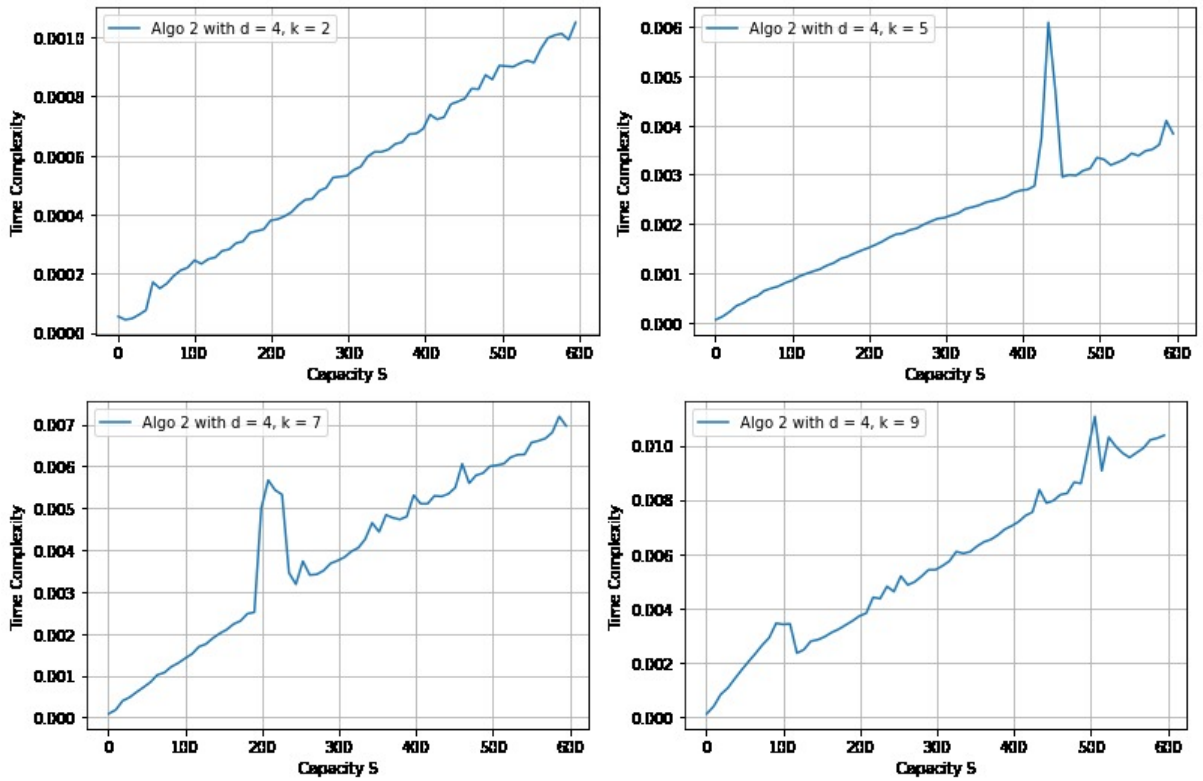










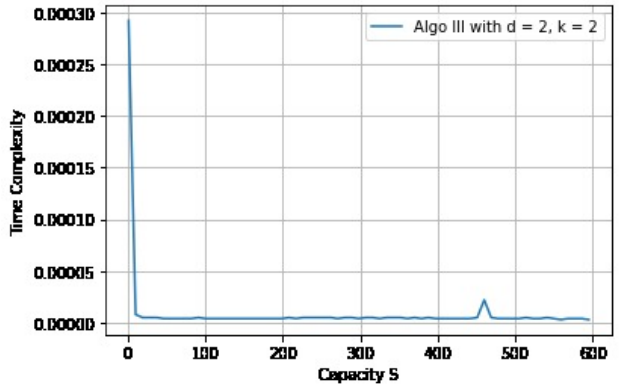


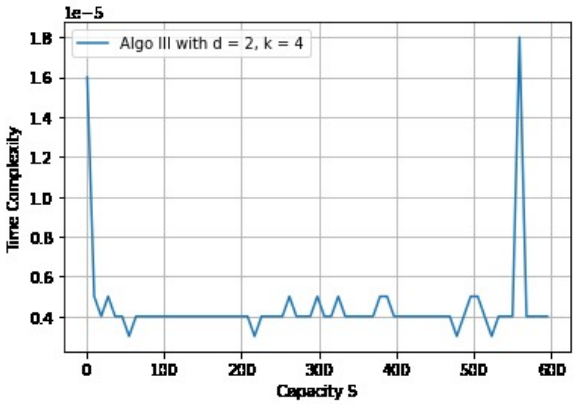
Observation: On remarque sur l'ensemble des résultats obtenus que la courbe représentant la complexité temporelle de la fonction dynamique évolue de façon relativement linéaire, plus S grandit et plus le temps d'exécution grandit.
On remarque aussi sur les quatres courbes que plus on augmente k plus le temps d'exécution croit pour les mêmes valeurs de d et de S.
Critique: Le résultat ci-dessus est un résultat justifié, car le temps d'exécution de la fonction dynamique est linéaire en fonction de k et de S qui représentent la taille de la matrice M utilisée.

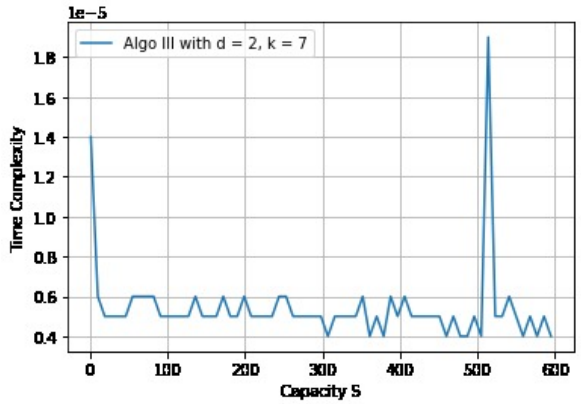
Algorithme III:
l'algorithme glouton a une complexité temporelle théorique de : O(k) // k étant une constante
Résultat:
Dans ce qui suit les résultats trouvés pour :

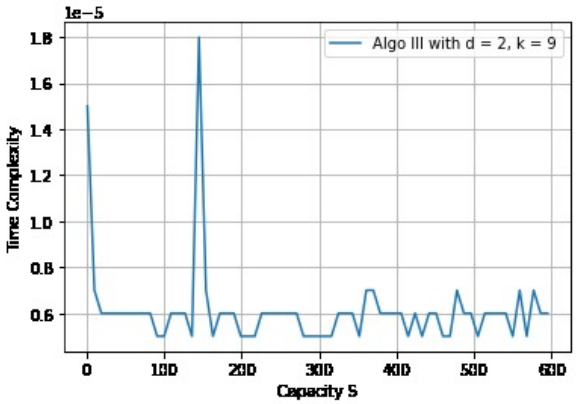
pour d = 2,3,4 :
  pour k = 2 .. 9;
    pour S = 2 .. 600 (avec un pas de 9);

d = 2

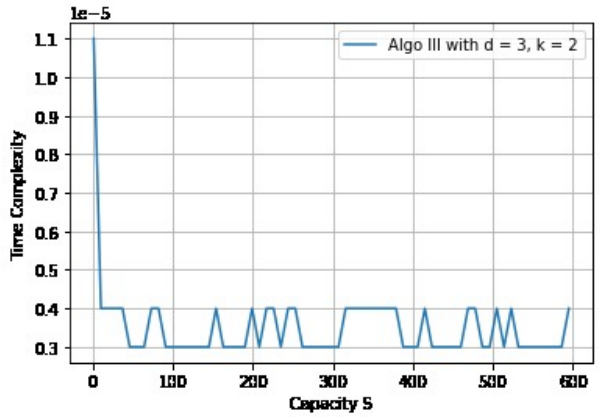

Algo III with d = 2, k = 2
Time Complexity
Capacity S

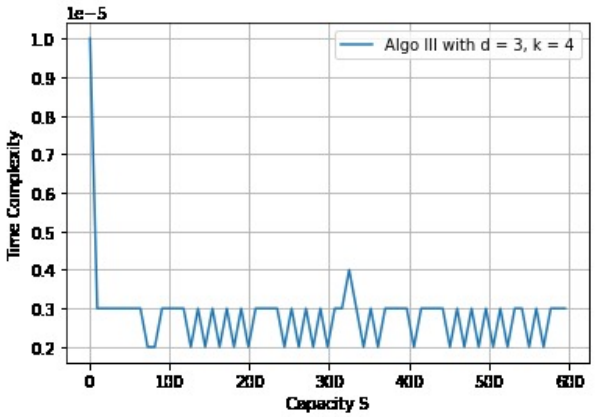

1e−5
Algo III with d = 2, k = 4
Time Complexity
Capacity S

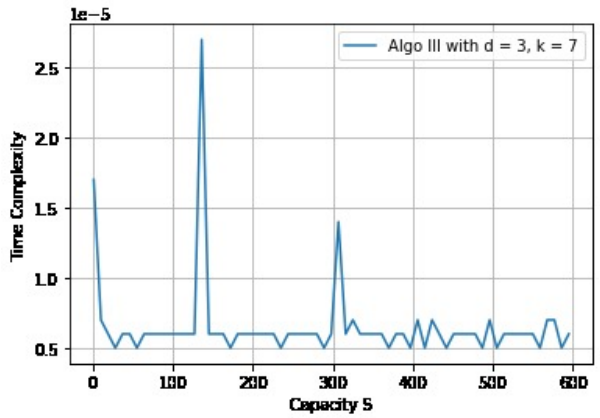

1e−5
Algo III with d = 2, k = 7
Time Complexity
Capacity S

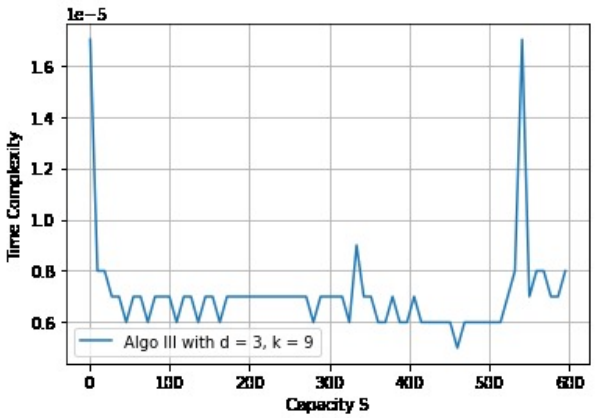

1e−5
Algo III with d = 2, k = 9
Time Complexity
Capacity S

d = 3:

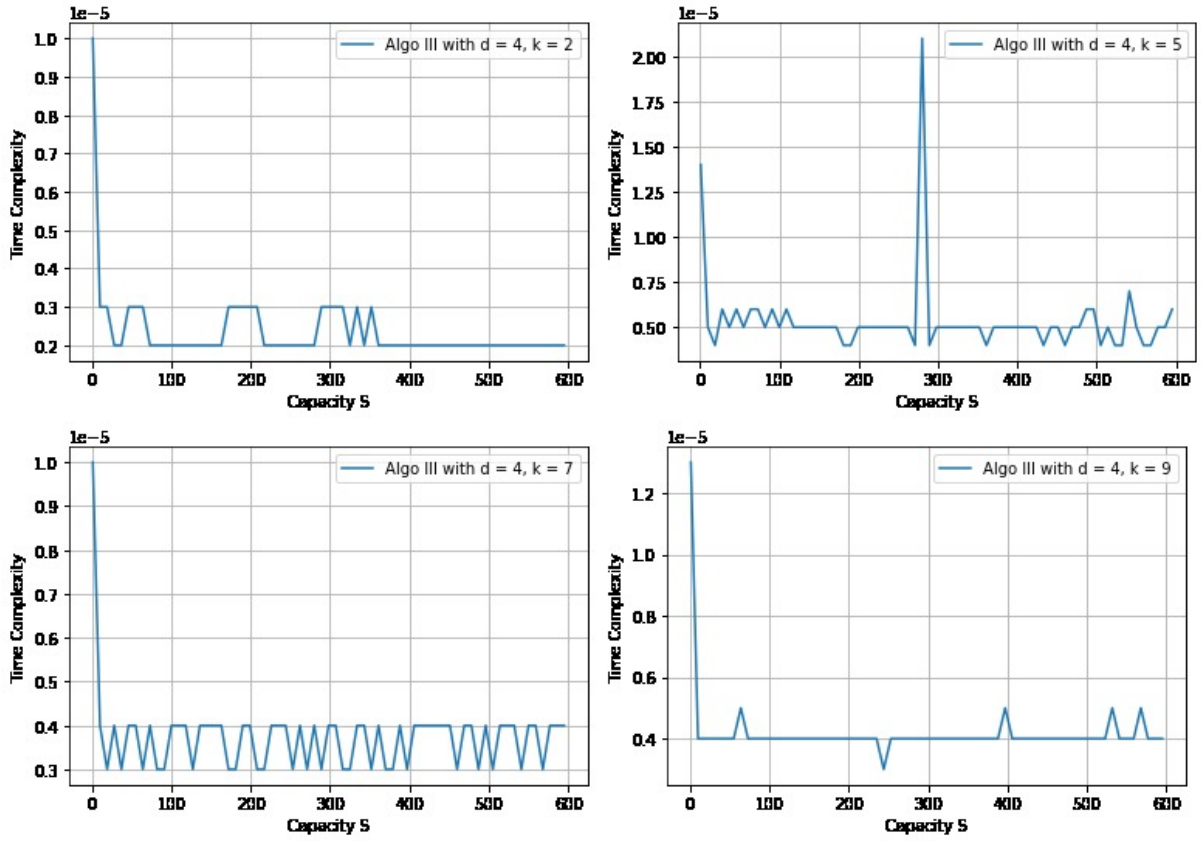

1e−5
Algo III with d = 3, k = 2
Time Complexity
Capacity S


1e−5
Algo III with d = 3, k = 4
Time Complexity
Capacity S


1e−5
Algo III with d = 3, k = 7
Time Complexity
Capacity S


1e−5
Algo III with d = 3, k = 9
Time Complexity
Capacity S

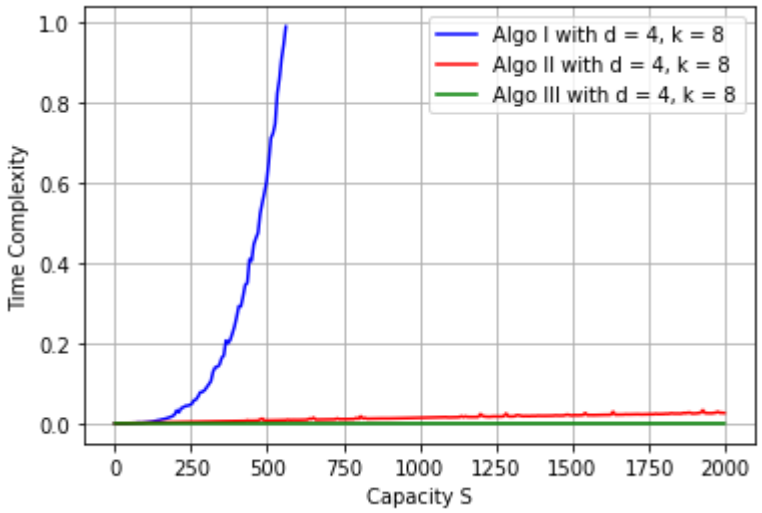
d = 4:



Observation: On remarque sur l'ensemble des résultats obtenus que la courbe représentant la complexité temporelle de la fonction gloutonne évolue de façon relativement constante, plus S grandit le temps d'exécution est au-dessous d'un certain seuil.
Critique: Le résultat ci-dessus est un résultat justifié, car le temps d'exécution de la fonction gloutonne est constant quelque soit la valeur de la capacité S pour un k fixe.

Performances des algorithmes:
l'algorithme glouton est beaucoup plus performant que les deux autres algorithmes pour la métrique considérée (temps d'exécution programme). L'algorithme dynamique performe moins bien que l'algorithme glouton, mais mieux que l'algorithme récursif qui lui a un temps d'exécution exponentiel.

Dans la figure ci-dessous, on trouve la courbe représentant les trois algorithmes, avec comme paramètres : d=2, k=8, S variant entre 2 et 1000 avec un pas de 7.



C. Utilisation de l'algorithme glouton

Génération de systèmes de capacités

Pour générer des systèmes de capacités, nous avons utilisé le code suivant :

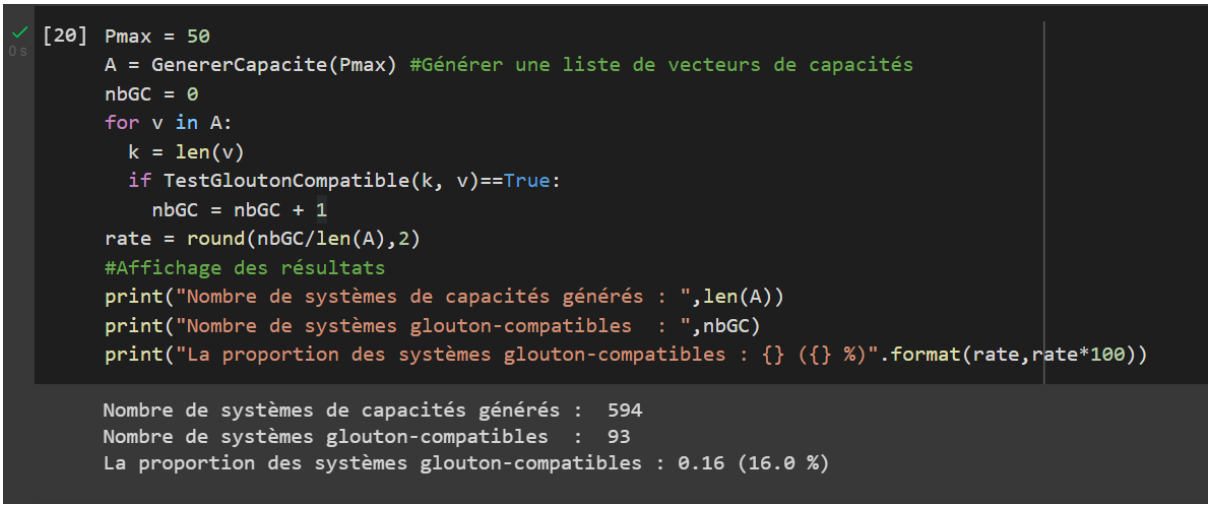
```
from numpy.random import randint
"""
Génération de systèmes de capacités
Cet algorithme génère aléatoirement des vecteurs de capacités
en utilisant la fonction randint de la bibliothèque numpy.random
Cet algorithme retourne une liste A de vecteurs de capacités
"""
def GenererCapacite(Pmax):
 A = []
 V = []
 for k in range(2,5):
  for j in range(1,30):
   V = randint(2, Pmax, k)
   V = list( dict.fromkeys(V) )
   V.sort()
   V = [1] + V
   if len(V)>2:
    A.append(V)
 return A
```

Question 13

Le code utilisé pour calculer la proportion de systèmes glouton-compatibles parmi l'ensemble des systèmes possibles générés est le suivant:

```
Pmax = 50
A = GenererCapacite(Pmax) #Générer une liste de vecteurs de capacités
nbGC = 0
for v in A:
  k = len(v)
  if TestGloutonCompatible(k, v)==True:
    nbGC = nbGC + 1
rate = round(nbGC/len(A),2)
#Affichage des résultats
print("Nombre de systèmes de capacités générés : ",len(A))
print("Nombre de systèmes glouton-compatibles  : ",nbGC)
print("La proportion des systèmes glouton-compatibles : {} ({} %)".format(rate,rate*100))
```

Un exemple d'exécution du code ci-dessus est dans la figure suivante:



Question 14

Statistiques sur les systèmes détectés non glouton-compatibles :

Pour élaborer ces statistiques, nous avons exécuté la fonction ci-dessus avec comme valeurs en entrée :

Pmax = 50, f = 10.

- Nombre total des systèmes de capacité examinés : 593
- Nombre de systèmes glouton-compatibles : 74
- Nombre de systèmes non glouton-compatibles : 519
- La proportion des systèmes glouton-compatibles : 0.12 (12 %)
- La valeur moyenne de l'ensemble des valeurs maximums des écarts calculés : 11,91
- La valeur moyenne de l'ensemble des valeurs moyennes des écarts calculés : 3,89
- La valeur médiane de l'ensemble des valeurs maximums des écarts calculés : 9
- La valeur médiane de l'ensemble des valeurs moyennes des écarts calculés : 3.08

Pmax = 100, f = 8.

- Nombre total des systèmes de capacité examinés : 297
- Nombre de systèmes glouton-compatibles : 31
- Nombre de systèmes non glouton-compatibles : 266
- La proportion des systèmes glouton-compatibles : 0.10 (10 %)
- La valeur moyenne de l'ensemble des valeurs maximums des écarts calculés : 24,86
- La valeur moyenne de l'ensemble des valeurs moyennes des écarts calculés : 8,24
- La valeur médiane de l'ensemble des valeurs maximums des écarts calculés : 20
- La valeur médiane de l'ensemble des valeurs moyennes des écarts calculés : 7.11

Pmax = 200, f = 8.

- Nombre total des systèmes de capacité examinés : 297
- Nombre de systèmes glouton-compatibles : 18
- Nombre de systèmes non glouton-compatibles : 279

- La proportion des systèmes glouton-compatibles : 0.6 (6 %)
- La valeur moyenne de l'ensemble des valeurs maximums des écarts calculés : 53,32
- La valeur moyenne de l'ensemble des valeurs moyennes des écarts calculés : 18,19
- La valeur médiane de l'ensemble des valeurs maximums des écarts calculés : 39
- La valeur médiane de l'ensemble des valeurs moyennes des écarts calculés : 14,39