Groupe 3

LU3IN003 - 2024
Projet d'algorithmique

Yanis Saadi Dit Saada
Ines Harraoui

Sommaire

I. Partie théorique

- Algorithme I

Question 1

a) Si $V[i] \leq S$, *alors*, $m(S) = 1 + m(s - V[i], i-1)$, sinon $m(S) = m(s, i-1)$

b) *m(s, i)* : représente le nombre minimum de bocaux nécessaires.

1. **Cas de base** :
   - Si s=0, alors $m(0,i)=0$ (aucun bocal nécessaire)
   - Si s<0, alors $m(s,i)=+\infty$ (impossible)
2. **Récurrence pour s>0** :
   - Option 1 : Aucun bocal de type *V[i]* ne peut être utilisé, donc $m(s,i)=m(s,i-1)$
   - Option 2 : On peut utiliser un bocal de type *V[i]*, donc $m(s,i)=1+m(s-V[i],i)$

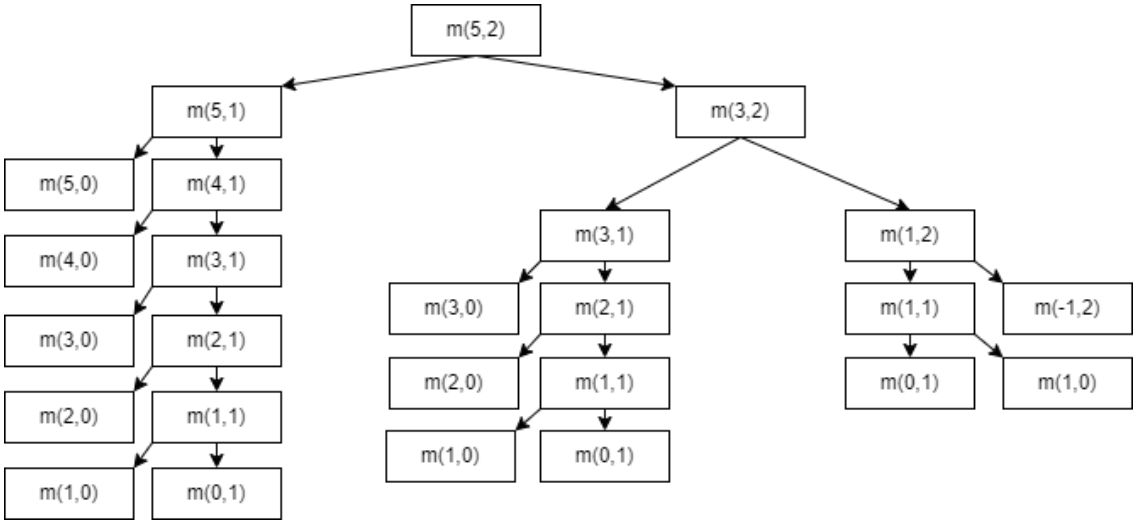
Si $V[i] \leq S$, *m(s,i)* vaut le minimum des deux options : $m(s,i) = \min(m(s,i-1), 1+m(s-V[i],i))$.

3. **Conclusion** : *m(s,i)* représente bien le nombre minimum de bocaux nécessaires pour ranger la quantité exacte, S.

Question 2

Algorithme *m***(***s* : entier **,** *i* : entier **,** *V* : tableau d'entiers**)** : entier
  Si *s* == 0 **alors**
    Retourner 0
  Fin
  Sinon si *s* < 0 ***alors***
    Retourner +∞
  Sinon si *i* == 0 **alors**
    Si *s* % *V*[0] == 0 **alors**
      Retourner *m*(*s* - *V*[*i*],*i*,*V*) + 1
    Sinon
      Retourner +∞
  Sinon
    SansBocal ← *m*(*s*, *i* - 1, *V*)
    AvecBocal ← *m*(*s* - *V[i]*, *i*, *V*) + 1
    Retourner min**(***SansBocal***,** *AvecBocal***)**

Question 3



Question 4

Dans l'arbre ci-dessus, *m(1,1)* apparaît 3 fois.
On remarque que pour *S = 3, m(1,1)* apparaît 2 fois et pour *S = 1, m(1,1)* apparaît qu'une fois.
On en déduit la relation suivante : pour *S* = 1, *m(1,1)* n'apparaît qu'une fois et pour tout *S* impair, *m(1,1)* apparaît $\lfloor S/2 \rfloor$ fois.

- Algorithme II

Question 5

a) Ordre de remplissage des cases du tableau *M*, de taille $(S+1) \times K$:

- On initialise $M[0][i]$ à 0.
- On initialise $M[s][0]$ à $+\infty$, pour $s > 0$.
- Pour chaque valeur de *s*, de 0 à *S*, et pour chaque *i*, de 1 à *K*, chaque case $M[s][i]$, dépend de $M[s][i-1]$ (*SansBocal*) et de $M[s-V[i]][i]$ (*AvecBocal*).

b) Algorithme qui détermine le nombre minimum de bocaux nécessaires pour une quantité *S* :

Algorithme AlgoOptimise(*S* : entier, *K* : entier, *V* : tableau d'entiers**)** : matrice

```
M : tableau de taille (S+1) × K, initialisé avec +∞
Pour s de 0 à S faire
    Pour i de 0 à K-1 faire
        Si i == 0 alors
            Si s%V[0] == 0 alors
                M[s][i] ← s
            Sinon
                M[s][i] ← +∞
        Sinon si s == 0 alors
            M[s][i] ← 0
        Sinon si V[i] > s alors
            M[s][i] ← M[s][i-1]
        Sinon
            M[s][i] ← min(M[s][i-1], M[s-V[i]][i] + 1)
        Fin si
    Fin pour
Fin pour
Retourner M
```

---

c) Complexité temporelle : $O(K \times S)$
Complexité spatiale : $O(K \times S)$

Question 6

a) Chaque case $M[s][i]$ contiendra un tableau A de taille K représentant les bocaux utilisés pour chaque capacité.
Si s=0, $M[0][i]$=[0,0,…,0] car aucun bocal n'est nécessaire
Si i=0, $M[s][0]$=[0,0,…,0] car il est impossible de remplir s sans utiliser de bocaux
Si un bocal $V[i]$ n'est pas utilisé, on copie la valeur de $M[s][i-1]$.
Sinon, on incrémente $A2[i]$ avec $A2$, une copie de A pour ne pas changer le tableau de $M[s][i-1]$.

Algorithme *AlgoOptimisé2*(*S* : entier, *K* : entier, *V* : tableau d'entiers) : tableau d'entiers

```
A ← [0, 0, 0, 0, 0, 0, 0, 0]
T ← [+∞,+∞,+∞,+∞,+∞,+∞,+∞,+∞]
M : tableau de taille (S+1) × K, initialisé avec +∞
Pour s allant de 0 à S faire :
    Pour i allant de 0 à K−1 faire :
        Si i=0 alors :
            M[0][i] ← A
        Si s=0 alors :
            M[s][0] ← A
        Si V[i]>s alors :
            M[s][i] ← M[s][i−1]
        Sinon :
            c ← min(somme(M[s][i−1]),somme(M[s−V[i]][i])+1)
            Si c = somme(M[s][i−1]) alors :
                M[s][i] ← M[s][i−1]
            Sinon :
                A2 ← copie de M[s−V[i]][i]
                A2[i] ← A2[i] +1
                M[s][i] ← A2
            Fin si
        Fin si
    Fin pour
Fin pour
Retourner M[S][K−1]
```

---

b)

Algorithme *backward*(*S*,*K*,*V*) : tableau

```
M ← AlgoOptimise(S, K, V)
A ← tableau de taille K, initialisé à 0
Tant que S > 0 et i ≥ 0 faire
    Si  i > 0 et M[S][i] = M[S][i-1] faire
        i ← i – 1
    Sinon
        A[i] ← A[i] + 1
        S ← S - V[i]
  Fin tant que
Fin pour
Retourner A
```

Question 7

Non, la complexité n’est pas polynomiale. Elle est pseudo-polynomiale.

Un algorithme est pseudo-polynomial si sa complexité est polynomiale par rapport à la valeur numérique des entrées, et non seulement au nombre de bits nécessaires pour les représenter.

La complexité de l'algorithme *AlgoOptimise* dépend fortement de S ; lorsque S augmente, le temps d'exécution de la boucle s allant de 0 à S devient plus long, car cette boucle effectue S itérations. À chaque itération, une autre boucle est exécutée, qui est majorée par K.

Avant de déterminer la complexité exacte, il est important de préciser que S est un entier représenté en binaire, et le nombre de bits nécessaires pour cette représentation est $log_2(S)$, que nous notons t. Si $t = log_2(S)$, alors $S = 2^t$, ce qui signifie que la complexité de *AlgoOptimise* est en réalité $O(2^t \times K)$. Cela montre que la complexité est exponentielle par rapport à la taille de l'entrée en bits, et donc l'algorithme est pseudo-polynomial.

La complexité de *backward* est dominée par l'appel à *AlgoOptimise*. Ainsi, même avec l'ajout de la boucle **Tant que**, la complexité reste $O(S \times K)$ ou $O(2^t \times K)$. L'algorithme *backward* ne modifie pas la complexité pseudo-polynomiale globale, mais il ajoute un temps d’exécution supplémentaire qui reste négligeable.

- Algorithme III

Question 8

Algorithme *AlgoGlouton***(***S* : entier, *K* : entier, *V :* tableau d’entiers) : tableau d’entiers

```
A ← tableau de taille K, initialisé à 0
i ← K−1
Tant que i ≥ 0 faire :
    Tant que S≥V[i] faire :
        A[i] ← A[i] +1
        S ← S −V[i]
    Fin tant que
    i ← i−1
Fin tant que
Retourner A
```

Complexité temporelle : $O(S \times K)$

Question 9

Il existe des systèmes de capacités qui ne sont pas glouton-compatibles car en utilisant la méthode qui privilégie toujours le bocal de plus grande capacité, on peut parfois obtenir une solution non optimale.

Le problème survient lorsque choisir le bocal de plus grande capacité à une étape conduit à utiliser plus de bocaux au total que si on avait choisi un bocal plus petit.

Considérons l'exemple V=[1,2,4,5,6,7,29,30] et S=3. Dans ce cas, l'algorithme glouton choisirait 30, ce qui laisse S=3. Ensuite, pour combler S=3, il faut utiliser un bocal de capacité 2 et un bocal de capacité 1, soit un total de 3 bocaux.

Or, en choisissant d'abord 29, il reste S=4, qui peut être couvert par un bocal de capacité 4. Ainsi, on utilise seulement 2 bocaux, ce qui est optimal.

Question 10

Le plus petit bocal $V[1]$ permet toujours d'ajuster parfaitement la quantité restante, car tout entier S peut être exprimé comme une somme de $V[1]$.

Si $S \geq V[2]$, l'algorithme glouton sélectionnera d'abord autant de bocaux de capacité $V[2]$ que possible. Pour la quantité restante, on complétera avec des bocaux de capacité $V[1]$. Cette solution est optimale car $V[1]$ est la seule autre option possible.
Ainsi, tout système de capacités de taille $2 \geq k$ est glouton-compatible, quelle que soit la quantité S recherchée.

Question 11

AlgoGlouton est en $O(S \times K)$ car il contient une boucle allant de K-1 à 0 imbriquant une boucle qui, dans le pire cas, effectue S passages.

Dans *TestGloutonCompatible*, on effectue une boucle tel que S prend une valeur de $V[3]+2$ à $V[k-1]+V[k]-1$. Or, S est majoré par la somme des valeurs dans V qu'on notera n. Ensuite, on effectue une autre boucle qui est majorée par K et qui fera deux appels à *AlgoGlouton* à chaque passage. Donc, la complexité de notre deuxième boucle sera en $O(K \times 2 \times (K \times S))$. On aura donc une complexité de *TestGloutonCompatible* qui sera en $O(S \times K \times 2 \times (K \times S) = O(K^2 \times S^2)$.

Puisque l'on a majoré S par n, *la* somme des valeurs dans V, on obtient $O(K^2 \times n^2)$ qui représente bien une complexité polynomiale.

Maintenant que nous avons étudier de manière théorique notre problème passons a l'expérimentation.

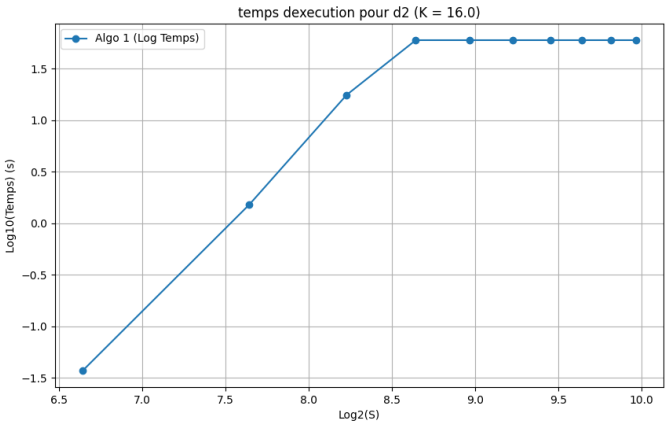
II. Mise en œuvre

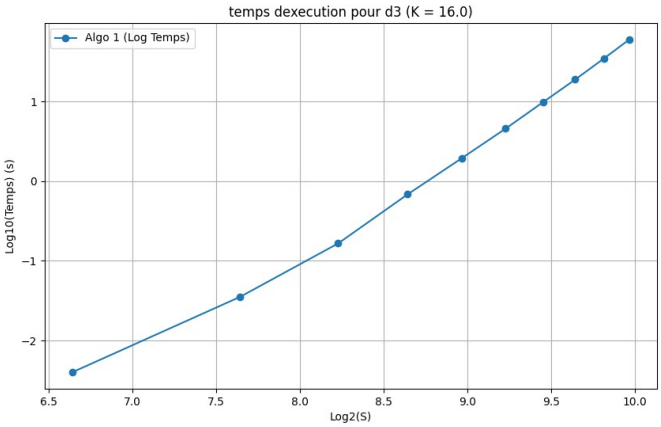
- Analyse de complexité expérimentale

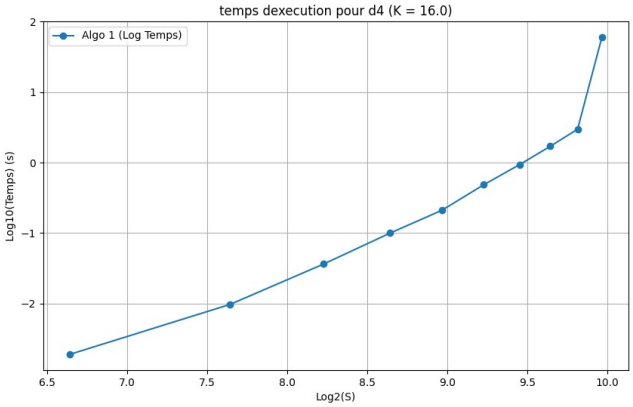
Question 12

Pour notre analyse de complexité expérimentale, nous avons créé un programme *eval*() pouvant générer des systèmes de taille pouvant aller jusqu'à K grâce à *Systeme_Expo* et qui calcule pour une quantité s valant d'un départ fixé jusqu'à une quantité S.
Les temps d'exécution sont ensuite sauvegardés dans des fichiers d'évaluation avec un fichier pour chaque valeur de d, et pour chaque K. Analysons chaque algorithme.

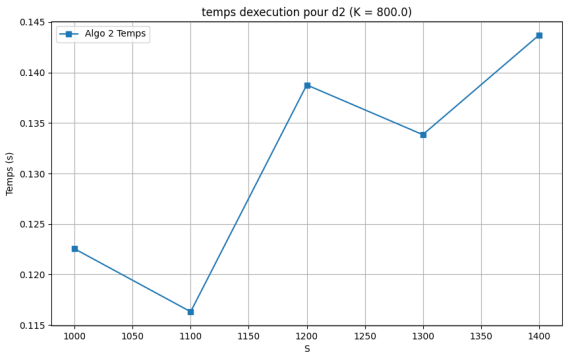
Le temps d'exécution d'Algo I est extrêmement lent en raison des nombreux calculs qui sont répétés, rendant cet algorithme inefficace. On observe dans les graphes ci-dessous que les courbes sont assez similaires les unes des autres et croissent toutes exponentiellement. Remarquons que le premier graphe se stabilise à une valeur proche de 1,5 car son temps d'exécution est majoré à 60 secondes au vu de son inefficacité aberrante et de ses multiples erreurs de récursion apparaissant à partir d'une certaine valeur de *S* trop grande.

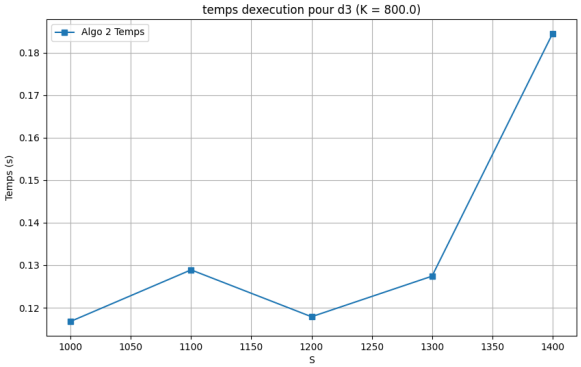


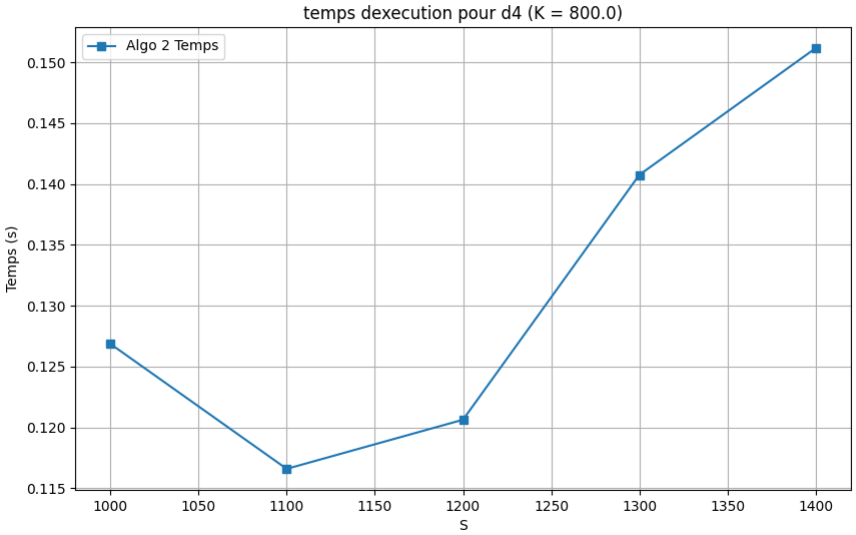




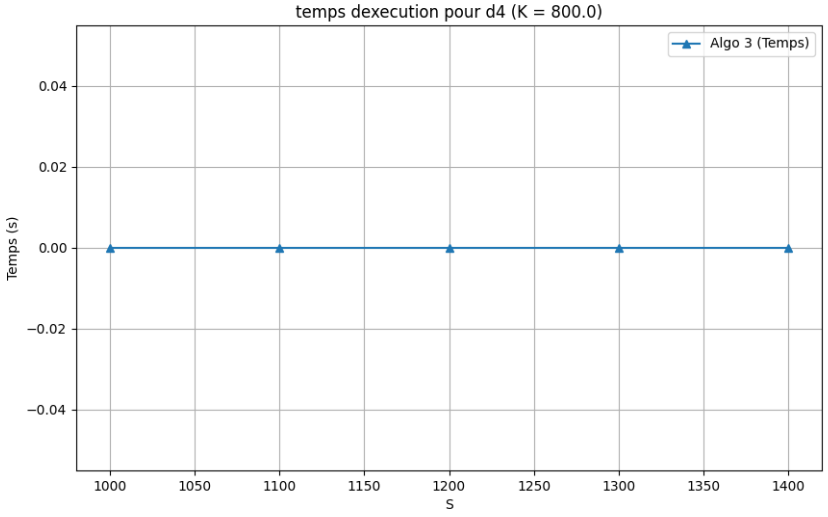
Lors de l'étude théorique de notre Algorithme II, nous avons conclu que sa complexité était pseudo-polynomiale. En analysant les graphes ci-dessous, on peut appuyer ce propos car chaque courbe croît par rapport à la valeur de *S* qui est le facteur dominant de notre temps d'exécution.



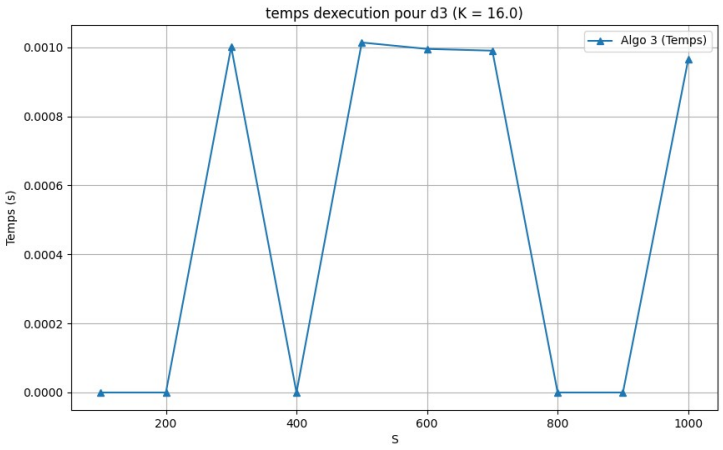




Finalement, l'algorithme III est celui qui est le plus atypique à analyser. En effet, il est extrêmement rapide donc son temps d'exécution est quasiment nul. Lors de l'analyse théorique, nous avons démontrer que notre algorithme avait une complexité polynomiale. Cependant, notre analyse expérimentale nous montre qu'il est aussi très bon pour des très grandes instances comme on peut le voir dans le graphe ci-dessous :



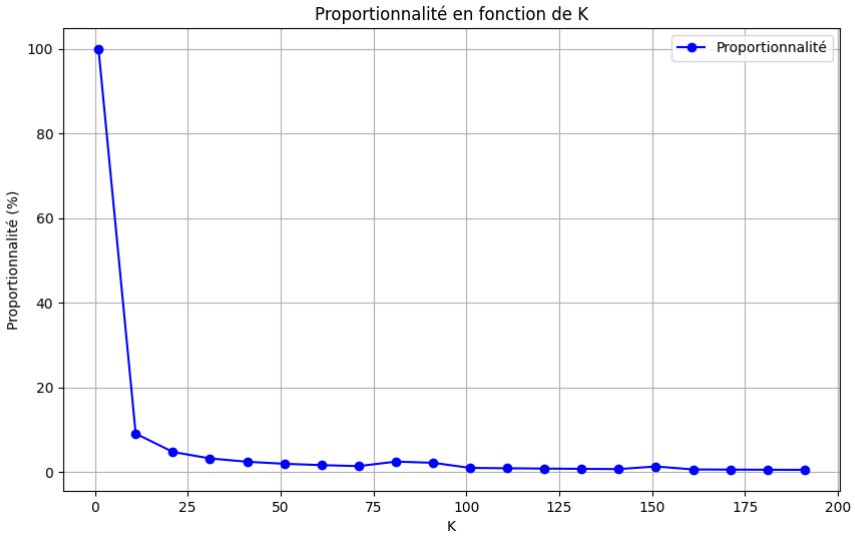
On mentionne le fait qu'il est aussi efficace dans des petites instances même s'il existe de très petites fluctuations.



Nous savons à présent comment fonctionnent nos algorithmes, que nous avons effectué des analyses sur ceux-ci, qu'en est-il de la solution générale à notre problème ?
Nous allons bien évidemment ignorer le premier algorithme et comparer les deux autres pour savoir lequel utiliser et pourquoi.

Question 13

On remarque que, graphiquement, plus *k* augmente, plus la proportion de systèmes glouton-compatibles diminue.

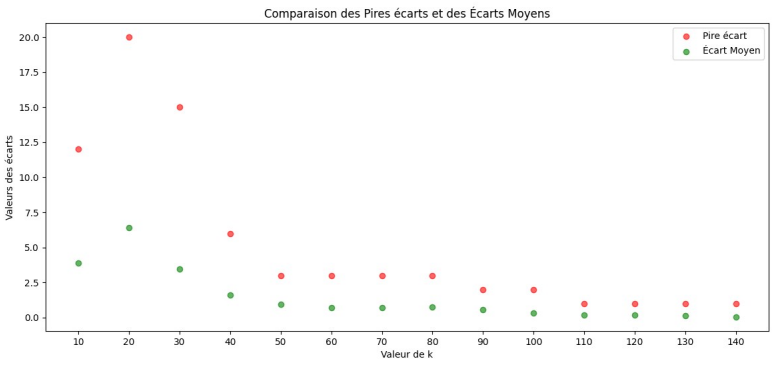


En utilisant l'algorithme *TestGloutonCompatible*, on a réalisé un programme *resolution*(*f*,*pmax*) qui calcule la proportion de systèmes glouton-compatibles parmi un ensemble de systèmes de taille *k* avec *k* allant de 1 à *K* croissant.
On remarque que plus *k* augmente, plus le nombre de systèmes non glouton-compatible diminue. En effet, lorsqu'on a $k \leq 2$, on possède tout le temps un système glouton compatible. Cependant, pour $k > 2$, on atteint une proportion de systèmes glouton-compatible quasiment nul.

L'algorithme glouton étant le plus rapide cela nous pose un grand inconvénient sur la résolution du problème.

Question 14

Pour les systèmes non glouton-compatibles, l'écart entre la solution gloutonne et optimale peut être important, mais cela dépend des valeurs des capacités. En effet, lorsque la capacité *k* augmente, on déduit du graphe ci-dessous que l'écart moyen et le pire écart diminue drastiquement. Par conséquent, même si l'algorithme glouton n'est pas optimal, il reste néanmoins très proche de cette solution optimale.



III. Conclusion

Ce projet avait pour objectif de fournir une solution efficace, tant en termes de rapidité d'exécution que de précision, pour le problème étudié. Pour ce faire, nous avons analysé trois algorithmes.

Le premier algorithme, bien qu'intuitif, a été rapidement écarté en raison de sa lenteur, particulièrement pour les grandes instances. Le deuxième algorithme (en programmation dynamique) s'est révélé fiable, offrant systématiquement la solution optimale. Cependant, son temps d'exécution devient prohibitif lorsque la quantité S ou le nombre de types de bocaux K augmente. Le troisième algorithme (glouton) s'est montré extrêmement rapide, mais ne garantit pas toujours une solution optimale, sauf pour les systèmes dits glouton-compatibles.

Pour évaluer la pertinence de l'algorithme glouton en pratique, nous avons mesuré la proportion de systèmes glouton-compatibles et comparé les écarts entre les solutions optimales et gloutonnes. Les résultats montrent que la proportion de systèmes glouton-compatibles diminue à mesure que K augmente. Toutefois, pour les systèmes non glouton-compatibles, les écarts entre les solutions gloutonnes et optimales se réduisent lorsque K devient grand.

Pour les petites instances, il est recommandé d'utiliser l'algorithme glouton lorsque le système est glouton-compatible et l'algorithme en programmation dynamique sinon. Pour les grandes instances, l'algorithme glouton est généralement préférable, car il offre une solution quasi-optimale en un temps très réduit. Bien que l'algorithme en programmation dynamique fournisse des solutions exactes, son temps d'exécution est souvent trop élevé pour être pratique dans ces cas.