Projet Confiture

Algorithmique II

LU3IN003

Inès BENAMER BELKACEM 21204927
Thibaut MARCQ 21202966

Semestre 5 - L3

Sommaire

Algorithme I

Question 1.a

Avec les valeurs définies dans la section précédente, on obtient :

$$\text{m(s, i)} = \text{m(748, 8)} = 9$$

Question 1.b

Soit $P(s, i)$: le nombre minimum de bocaux nécessaires pour remplir une quantité s de confiture en utilisant uniquement les bocaux de taille $V[1], \ldots, V[i]$ est donné par $m(s, i)$.

Base : Pour tout $i \in \{1, \ldots, k\}$, lorsque $s = 0$, on a $m(0, i) = 0$ selon la définition (il est possible de réaliser la capacité totale 0 avec 0 bocal).

Induction : Montrons $P(s, i)$ pour $s > 0$ et $i \in \{1, \ldots, k\}$. Supposons que $P(s, i)$ est vraie, montrons que $P(s + 1, i)$ l'est aussi.

On veut montrer que :

$$m(s, i) = \min\{m(s, i - 1), m(s - V[i], i) + 1\}.$$

Il y a deux options pour remplir exactement une quantité s en utilisant les bocaux :

— **Ne pas utiliser le bocal de taille $V[i]$:**
On cherche la solution optimale en utilisant les bocaux $V[1]$ à $V[i-1]$, d'où l'appel $m(s, i - 1)$.

— **Utiliser un bocal de taille $V[i]$:**
On remplit un bocal de taille $V[i]$, ce qui réduit la quantité de confiture restante à $s - V[i]$. On continue ensuite de chercher la solution pour la quantité restante tout en pouvant réutiliser le bocal $V[i]$. Cela donne $m(s-V[i], i)$, et donc $m(s-V[i], i)+1$ puisque un bocal a été utilisé lors de cet appel.

On prend ensuite le minimum entre ces deux options pour minimiser le nombre de bocaux utilisés.

Conclusion : Pour tout $i \in \{1, \ldots, k\}$, $P(s, i)$ est vraie. Ainsi, $m(s, i)$ renvoie bien la quantité minimale de bocaux $V[1], \ldots, V[i]$ nécessaires pour répartir une quantité s de confiture.

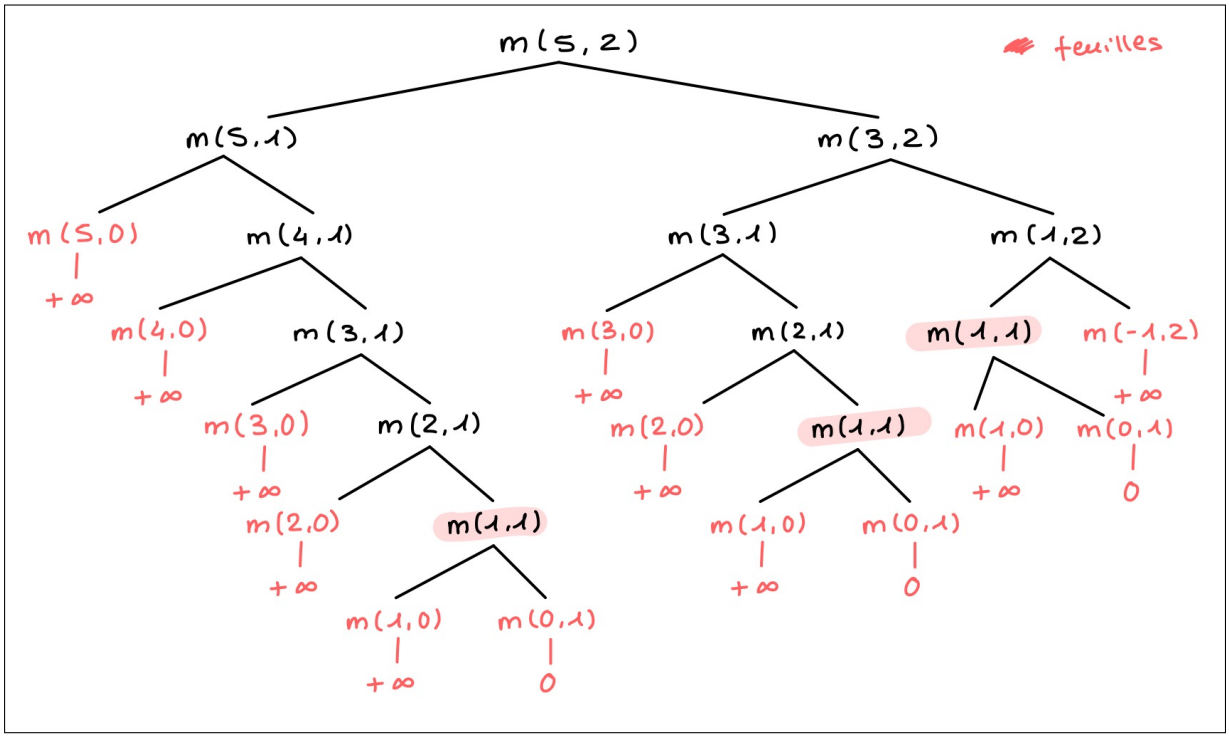
Question 2

Calcul de m(S) en pseudo-code :

Fonction `minBocaux`(s, i, V) **:**
  si $s == 0$ **alors**
    retourner 0;
  fin
  sinon si $i == 0$ ***ou*** $s < 0$ **alors**
    retourner $+\infty$;
  fin
  $SansBocal \leftarrow$ `minBocaux`(s, $i-1$, V);
  $AvecBocal \leftarrow$ `minBocaux`($s - V[i]$, i, V) $+1$;
  retourner $\min(SansBocal, AvecBocal)$;
fin

Question 3

Arbre des appels pour $S = 5$, $k = 2$, $V[1] = 1$ et $V[2] = 2$:



Question 4

Dans l'arbre précédent, $m(1,1)$ est calculé 3 fois pour $S = 5$. Pour une valeur de S impair, $m(1,1)$ est calculé $\lceil \frac{S}{2} \rceil$ fois.

$m(1,1)$ est présent :

— 3 fois dans $m(5)$

— 5 fois dans $m(9)$

— 6 fois dans $m(11)$

— 365 fois dans $m(729)$

Algorithme II

Question 5.a

Avec la formule de récurrence, on observe que les valeurs $m(q - V[i], i)$ et $m(s, i-1)$ doivent être calculées avant de pouvoir calculer $m(s, i)$. Ainsi, on commence par remplir les cases correspondant aux cas de base (feuilles de l'arbre), puis on remonte petit à petit dans les calculs pour obtenir les valeurs de $m(s, i)$ (c'est-à-dire, on remonte dans les nœuds de l'arbre). Enfin, on pourra remplir la case $T[s][i]$ correspondant à l'appel initial.

Question 5.b

Pseudo-code de l'algorithme :

```
Fonction AlgoOptimise(s, k, V) :
    M ← initialiserTab(s + 1, k + 1)
    pour i de 0 à k faire
        M[0][i] ← 0;
    fin
    pour x de 1 à s faire
        pour y de 1 à k faire
            SansBocal ← M[x][y − 1];
            AvecBocal ← +∞;
            si x ≥ V[y] alors
                AvecBocal ← M[x − V[y]][y] + 1;
            fin
            M[x][y] ← min(SansBocal, AvecBocal);
        fin
    fin
    retourner M[s][k];
fin
```

$initialiseTab(s+1, k+1)$ initialise un tableau de taille $(s+1) \times (k+1)$ avec toutes les cases à $+\infty$

Question 5.c

Complexité temporelle :
L'algorithme effectue deux boucles. La boucle extérieure parcourt les quantités x de 1 à s. La boucle intérieure parcourt les types de bocaux i de 1 à k. À chaque itération, on effectue seulement une comparaison et des changements de valeurs de variables. La complexité temporelle est donc en $\mathcal{O}(s \times k)$. C'est une complexité polynomiale en fonction de la quantité s et du nombre de types de bocaux k.

Complexité spatiale :
On utilise un tableau de taille $(s+1) \times (k+1)$ pour stocker des entiers. Ainsi, la complexité spatiale de l'algorithme est en $\mathcal{O}(s \times k)$. C'est aussi une complexité polynomiale en fonction de la quantité s et du nombre de types de bocaux k.

Question 6.a

Pseudo-code de l'algorithme :

```
Fonction AlgoOptimise2(s, k, V) :
    m ← initialiserTab(s + 1, k + 1, (+∞, [0] × len(V)));
    pour i de 0 à k faire
        m[0][i] ← (0, [0] × len(V));
    fin
    pour i de 1 à s faire
        pour j de 1 à k faire
            sb_nb, sb_tab ← m[i][j − 1];
            ab_nb, ab_tab ← (+∞, [0] × len(V));
            si i ≥ V[j − 1] alors
                tmp, ab_tab ← m[i − V[j − 1]][j];
                ab_tab ← copie(ab_tab);
                ab_tab[j − 1] ← ab_tab[j − 1] + 1;
                ab_nb ← tmp + 1;
            fin
            si sb_nb ≤ ab_nb alors
                m[i][j] ← (sb_nb, sb_tab);
            fin
            sinon
                m[i][j] ← (ab_nb, ab_tab);
            fin
        fin
    fin
    retourner m[s][k];
fin
```

Soit $len(V)$ la taille du tableau V représentant les volumes des bocaux. La fonction $initialiserTab(s+1, k+1)$ initialise un tableau de taille $(s+1) \times (k+1)$ avec toutes les cases contenant des tuples $(+\infty, [0] \times len(V))$.

Le terme $[0] \times len(V)$ représente un tableau $[0, ..., 0]$ de taille $len(V)$. Chaque élément de ce tableau est initialisé à zéro, correspondant au nombre de bocaux utilisés pour chaque type de volume.

Cet algorithme reprend la structure de l'algorithme précédent. Néanmoins, il implémente un tuple d'entier et de tableau à la place d'un simple tableau dans l'algorithme précédent. Il implémente donc des opérations supplémentaires pour choisir la case à incrémenter du tableau.

L'algorithme comporte toujours un tableau de $(s+1) \times (k+1)$ cases. Chaque case comporte maintenant une valeur et un tableau de k cases. Chaque case a donc une complexité spatiale en $\mathcal{O}(k)$. La complexité spatiale totale de l'algorithme est donc en :

$$\mathcal{O}((s+1) \times (k+1) \times k) = \mathcal{O}(s \times k^2).$$

Question 6.b

Pseudo-code de l'algorithme backward :

```
Fonction AlgoBackward(s, k, V) :
    M ← CalculerMatrice(s, k, V);
    utilises ← [0] × len(V);
    i ← s;
    j ← k;
    tant que i > 0 et j ≥ 0 faire
        si M[i][j] == M[i − V[j]][j] + 1 alors
            utilises[j] ← utilises[j] + 1;
            i ← i − V[j];
        fin
        sinon
            j ← j − 1;
        fin
    fin
    retourner utilises;
fin
```

La fonction `CalculerMatrice` correspond à la fonction de la question 5.b. Elle renvoie le tableau M plutôt que de renvoyer la valeur $M[s][k]$.

Question 7

Dans AlgoBackward, on commence par faire appel à la fonction `CalculerMatrice`. Comme vu précédemment, cet appel se fait en $\mathcal{O}(s \times k)$.

On initialise ensuite le tableau `utilises`, ce qui se fait en $\mathcal{O}(k+1) \in$O(k).

Pour la boucle `tant que`, on itère jusqu'à ce que l'un des indices atteigne 0. Le pire des cas est donc si, pour arriver au cas de base, chaque indice décrémente d'au plus 1 de manière alternée à chaque appel. Dans ce cas, pour obtenir `i=0` ou `j=0`, on aura effectué $\mathcal{O}(s+k)$ itérations.

A chaque tour de boucle, on réalise uniquement des opérations d'accès a un tableau, de comparaison, et d'affectation, toutes en $\Theta(1)$. La complexité d'un tour de boucle est donc $\Theta(1)$.

L'ensemble du bloc tant que est donc de complexité :

$$\mathcal{O}(s+k) \times \Theta(1) = \mathcal{O}(s+k).$$

La complexité globale est donc en :

$$(\mathcal{O}(s \times k) + \mathcal{O}(k) + \mathcal{O}(s+k)) \in \mathcal{O}(s \times k).$$

Ainsi, on peut penser que cet algorithme est de complexité polynomiale.

Pourtant, il faut aussi considérer la taille s de l'entrée.
Soit $t = \log_2(s)$ la taille de l'entrée, alors $s = 2^t$. On a donc une complexité de l'algorithme en $\mathcal{O}(2^t \times k)$.
La complexité est donc pseudo-polynomiale.

Algorithme III

Question 8

Pseudo-code de l'algorithme :

```
Fonction SubAlgoGlouton(s, k, V) :
    res ← 0;
    pour i de k à 1 en décrémentant faire
        tant que s ≥ V[i] faire
            s ← s − V[i];
            res ← res + 1;
        fin
    fin
    retourner res;
fin
```

Cette fonction est de complexité temporelle O(k×s).

Question 9

Il existe bien des systèmes de capacités non glouton-compatibles.
Par exemple, avec $V = [1, 3, 4, 5]$, si on souhaite distribuer une quantité $S = 7$:

- L'algorithme glouton va utiliser un pot de capacité 5 puis 2 pots de capacité 1, soit 3 pots.
- La solution optimale va utiliser un pot de capacité 4 et un pot de capacité 3, soit 2 pots.

Dans ce système de capacités, l'algorithme glouton ne fournit donc pas la solution optimale

Question 10

On se place dans le cas où on a une quantité S à répartir dans les 2 premiers pots du tableau.
Soit $d = V[2]$.

La solution optimale utilisera un pot de quantité d tant que la quantité restante est supérieure à d, donc autant de pots d que possible. C'est la solution optimale car le seul autre pot disponible est le pot 1 de capacité $V[1] = 1$. On préférera toujours utiliser un pot de capacité d à d pots de capacité 1. En particulier, il utilisera $S//d$ pots de capacité d et $S\%d$ pots de capacité 1.

L'algorithme glouton fera la même chose car il privilégiera toujours le plus grand pot jusqu'à ce que ça ne soit plus possible.
Donc pour $k = 2$ (avec $V[1] = 1$), le système de capacité est glouton compatible.

Question 11

L'algorithme de test est principalement composé de deux boucles ainsi que d'une condition, appelant `AlgoGlouton`.

Nombre d'itération de la première boucle (pour S de $(V[3]+2)$ à $(V[k-1]+V[k]-1)$) :

$$nbIter = (V[k-1] + V[k] - 1 - (V[3] + 2)) + 1 = V[k-1] + V[k] - V[3] - 2$$

C'est donc en $\mathcal{O}(V[k-1] + V[k] - V[3] + 2) \in \mathcal{O}(V[k-1] + V[k]) \in \mathcal{O}(V[k])$

La première boucle est donc en $\mathcal{O}(V[k])$.

Nombre d'itération de la deuxième boucle (pour j de 1 à k) : k tours de boucle, donc en O(k)
La condition au coeur des boucles fait appel à `AlgoGlouton` deux fois. Cette fonction a une complexité en $\mathcal{O}(k)$ (question 8). On a donc une complexité en $\mathcal{O}(k+k) = \mathcal{O}(2k) = \mathcal{O}(k)$.

Ainsi, la complexité totale de l'algorithme est en :

$$\mathcal{O}(V[k] \times k \times k) = \mathcal{O}(V[k] \times k^2)$$

Mise en oeuvre

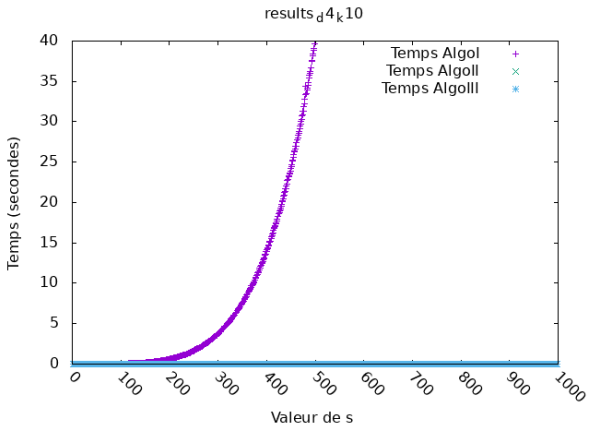
Question 12

Par la suite, nommons :

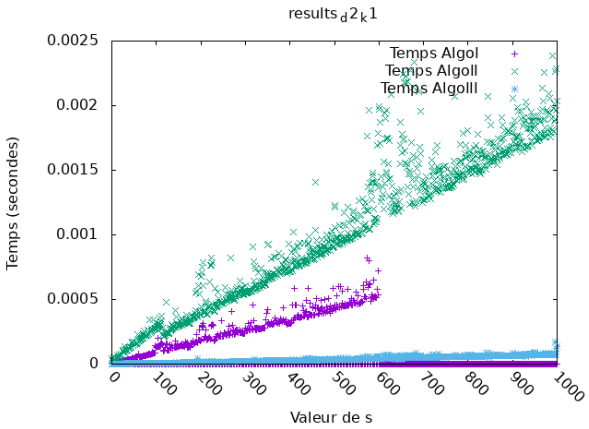
- **Algorithme I** : l'algorithme défini par la relation de récurrence (Partie 1).
- **Algorithme II** : l'algorithme utilisant la mémoïsation (Partie 2).
- **Algorithme III** : l'algorithme glouton (Partie 3).

Nous avons réalisé plusieurs exécutions des algorithmes précédents en faisant varier la variable d, le nombre de cases du tableau de bocaux k, et bien sûr, la quantité S de confiture.
On obtient ainsi plusieurs résultats, tous disponibles en annexe. Les graphiques affichent toutes les valeurs ou seulement celles des algorithmes II et III. Nous nous intéressons ici à quelques cas particuliers pour décrire la complexité des algorithmes.

À partir des deux graphiques suivants, on observe que l'Algorithme I possède une complexité exponentielle. Les durées de réalisation de ses appels augmentent drastiquement plus la valeur de k est grande. En revanche, lorsque k=1, les durées de réalisation ne croient pas de la même façon. Cela confirme donc bien sa complexité en $\mathcal{O}(2^k \times s)$.

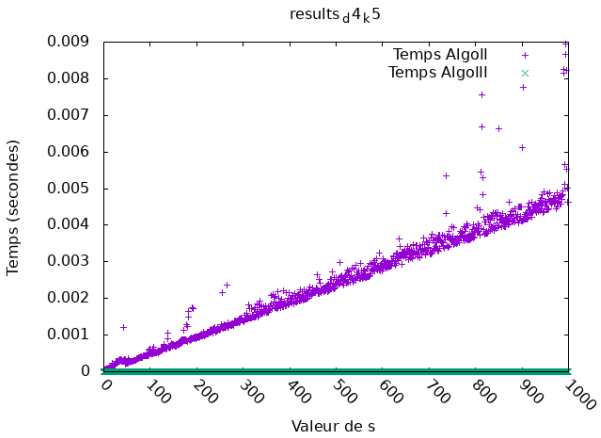


Résultats pour $d = 4$ et $k = 10$



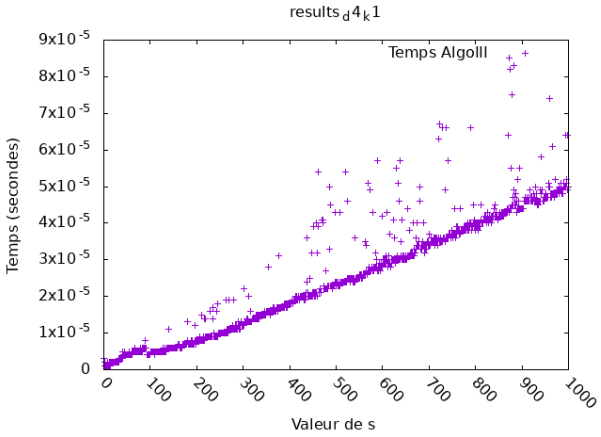
Résultats pour $d = 2$ et $k = 1$

On peut aussi observer que l'Algorithme II suit une progression linéaire. Les temps augment en fonction de la taille de S. Cela confirme la complexité pseudo-polynomiale trouvée à la question 7. On peut d'ailleurs observer que dans les cas où k=1, l'Algorithme II est moins performant que l'Algorithme I, celui-ci n'ayant pas une complexité prenant en compte la taille de l'entrée (graphique précédent).



Résultats pour $d = 4$ et $k = 5$

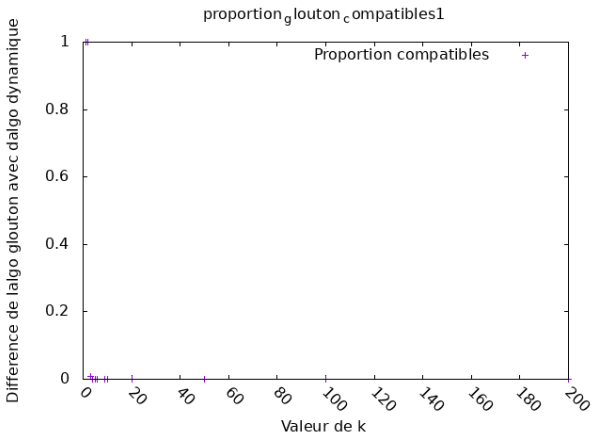
Enfin, sur tous les graphiques, on observe que l'Algorithme III possède des temps d'exécution restant constants. Pourtant nous avions déduit à la question 8 une complexité pseudo-polynomiale en $O(k \times 2^t)$. Les résultats sur les graphiques s'expliquent par le fait que les pires cas sont intimement liés à la valeur de k choisie. Avec une valeur de k suffisament grande, il y a peu de chances que les boucles while soient à chaque fois exécutées s fois. Un des cas où cette possibilité se produit est celui où k=1. On observe alors des durées d'exécution croissant en fonction de la taille s (graphique dessous).



Résultats pour $d = 4$ et $k = 1$

Question 13

Nous avons testé pour cette question la proportion de systèmes gloutons-compatibles sur un grand ensemble de valeurs. Pour chaque valeur de k donnée, nous calculons la proportion de systèmes glouton-compatibles par rapport à tous les systèmes générés.



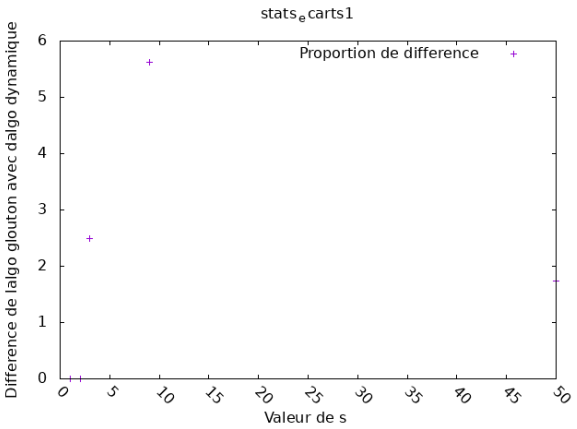
Proportions de compatibilité de l'algo glouton en fonction de k

On constate que mise à part pour une taille de tableau capacités k=1 ou k=2 (pour lesquels a déjà été montrée la validité), la proportion de systèmes glouton-compatibles est quasiment nulle sur les croix violettes. Cela s'explique en effet. Pour qu'un système soit glouton-compatible, il faut qu'il donne la solution optimale peu importe la quantité S donnée en entrée. L'algorithme glouton prend au maximum les plus grandes, mais une solution optimale ne garantit pas une telle répartition des capacités sur un système généré aléatoirement. On peut remarquer qu'en augmentant significativement le nombre de systèmes crées pour un k donné, la proportion de systèmes gloutons-compatibles augmente, mais il n'est pas nécessaire d'en discuter pour notre application. Les systèmes glouton-compatibles sont donc un cas particulier de systèmes, et ne fournissent pas la solution exacte l'écrasante majorité du temps.

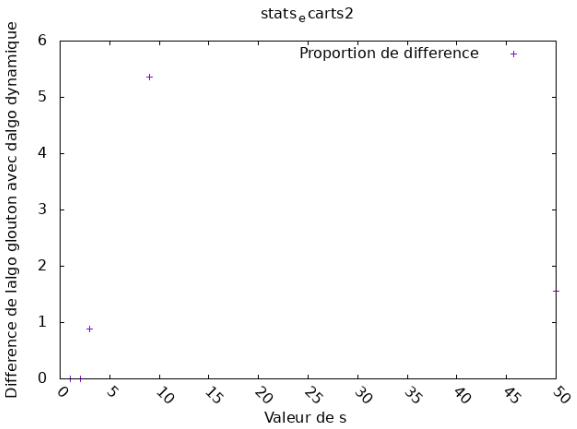
Il devient alors pertinent d'étudier la différence de résultat obtenue entre l'algorithme de programmation dynamique et l'algorithme glouton, afin de déterminer "à quel point l'algorithme glouton se trompe".

Question 14

Pour cette question, nous avons testé la différence de valeurs obtenues pour différentes instances de système, sur des tableaux de capacités de tailles différentes.



ex1 : Différence de résultat en fonction de *k



ex2 : Différence de résultat en fonction de *k

Ces figurent nous montrent le pourcentage de différence entre les deux algorithmes $AlgorithmeII$ et $AlgorithmeIII$ sur leur valeur de retour en fonction de la taille k du tableau d'entrée. Elles nous indiquent à quel point $AlgorithmeIII$ renvoie une valeur éloignée de $AlgorithmeII$. On constate que cette proportion a une croissance ralentie et atteint un plateau lorsque la valeur de k augmente, et tend vers 6% de différence. Sur d'autres jeux de tests nous avons pu obtenir une différence de 3% en moyenne. Nous avons donc une idée de l'erreur que nous donne l'algorithme glouton. On peut effectuer laisser à l'utilisateur le choix de décider si cette marge convient à son utilisation ou non. On peut également effectuer les calculs avec l'algorithme glouton plus rapide, puis estimer le résultat réel avec une marge d'erreur donnée.