Devoir n°2

3 heures

Disque dur à deux têtes

On attachera une grande importance à la concision, à la clarté, et à la précision de la rédaction. Le temps d'exécution $T(f)$ d'une fonction f est le nombre d'opérations élémentaires (addition, soustraction, multiplication, division, affectation, etc.) nécessaire au calcul de f. Lorsque ce temps d'exécution dépend d'un paramètre n, il sera noté $T_n(f)$. On dit que la fonction f s'exécute : en temps $\mathcal{O}(n\alpha)$, s'il existe $K > 0$ tel que pour tout n, $T_n(f) \leqslant Kn\alpha$.

Ce problème étudie des stratégies de déplacement des têtes d'un disque dur afin de minimiser le temps moyen d'attente entre deux requêtes au disque dur (de lecture ou d'écriture). Dans ce problème, le disque dur est représenté par une demi-droite $[0, +\infty)$ et possède deux têtes de lecture/écriture. Chacune des têtes peut aller indifféremment à n'importe quelle position sur le disque pour y lire ou écrire une donnée. Les deux têtes peuvent être au même endroit ou encore se croiser. On ne s'intéresse qu'aux temps de déplacement des têtes et non aux temps de lecture/écriture. Les deux têtes ont la même vitesse de déplacement. Le temps de déplacement d'une tête est supposé égal à la distance qu'elle parcourt.
Une requête r est un entier positif ou nul représentant l'emplacement du disque auquel l'une des deux têtes doit se rendre. Initialement les deux têtes sont chacune à la position 0. Le disque dur est muni d'une mémoire (appelée cache) qui permet d'enregistrer n requêtes ($n > 0$) avant de les traiter. À chaque bloc de n requêtes présentes dans le cache, le contrôleur du disque dur doit alors satisfaire ce bloc de requêtes, dans leur ordre d'arrivée, en minimisant le déplacement total des deux têtes. L'ordre importe puisqu'une opération d'écriture peut précéder une autre opération de lecture ou d'écriture. Il faut donc déterminer pour chacune des n requêtes le numéro de la tête à déplacer de manière à minimiser la somme totale des temps de tous les déplacements.

Partie A – Coût d'une séquence de déplacements

Un bloc de n requêtes est représenté par une suite de n entiers positifs ou nuls $[r_1, r_2, \ldots, r_n]$ rangés dans un tableau r de taille n. Une séquence de déplacements $[d_1, d_2, \ldots, d_n]$ est une suite de n entiers, 1 ou 2, rangés dans un tableau d indiquant à l'étape i qui de la première tête ($d_i = 1$) ou de la deuxième tête ($d_i = 2$) doit se déplacer à la position r_i ($1 \leqslant i \leqslant n$). Le coût d'une séquence de déplacements est la somme totale des distances parcourues par chacune des têtes.

Ainsi pour le bloc de requêtes $[5, 2, 4]$ le coût de la séquence de déplacements $[1, 1, 2]$ est $5 + 3 + 4 = 12$, alors que le coût de $[1, 2, 1]$ vaut $5 + 2 + 1 = 8$.

1. Pour le bloc de requête $[7, 3, 8]$, quel est le coût de déplacements $[1, 1, 1]$? de déplacements $[2, 1, 2]$?
2. Écrire une fonction coutDe(r, d) qui calcule le coût d'une séquence de déplacements d pour le bloc de requêtes r. On donne le squelette à compléter et commenter :

```
def coutDe(r, d):
    n = len(r) # le nombre de déplacement à effectuer
    cout = 0  # initialisation du coût à 0
    p1 = 0    # position de la tête 1
    p2 = 0    # position de la tête 2
    for i in range(n): # i de 0 à n-1
        if d[i] == 1:  # si tête 1
            cout += abs(r[i]-p1) # le coût du déplacement en plus
            p1 = r[i]  # nouvelle position tête 1
        else:          # sinon tête 2
            cout += abs(r[i]-p2) # le coût du déplacement en plus
            p2 = r[i]  # nouvelle position tête 2
    return cout        # on retourne le coût total cherché
```

3. Combien de séquences de déplacements satisfont un bloc de requêtes r donné ?

Le coût optimal d'une suite de requêtes r est le plus petit coût des séquences de déplacements satisfaisant le bloc de requêtes r.

4. Montrer qu'il existe toujours une séquence de déplacements de coût optimal (minimal).

   Montrer aussi qu'il existe toujours une séquence de déplacements de coût optimal qui commence par 1, c'est-à-dire commençant par déplacer la première tête.

5. Est-il raisonnable d'envisager une méthode de programmation par force brute ? Est-ce que la symétrie de la question précédente apporte une réduction notable ?

6. Pourquoi peut-on représenter toutes les possibilité de déplacements des têtes pour une requête de taille n à l'aide des nombres de 0 à $2^n - 1$ représentés en notation binaire ?
   On rappelle que la fonction python bin prend un entier comme argument et envoie une chaine de la forme par exemple si $n = 10$, '0b1010'. On supposera bin à coût constant.
   Expliquer alors le fonctionnement de l'algorithme en force brute suivant en commentant chaque ligne et donner sa complexité temporelle.

```
def couMinForce(r):
    n = len(r)
    cmin = float('inf')
    for k in range(2**n):
        b = bin(k)[2:]    # ?
        p = len(b)
        b = '0'*(n-p)+b   # ?
        c = coutDe(r, b)
        if c < cmin :
            cmin = c
    return cmin
```

Partie B – Coût optimal pour deux requêtes

Dans cette partie, le cache est de taille 2 ($n = 2$). Il n'y a donc que deux requêtes r_1 et r_2. **Par convention, la première tête sera toujours celle qui bouge sur la première requête.**

7. Donner une séquence de déplacements de coût minimal pour chacun des deux blocs de requêtes $[10, 3]$ et $[3, 10]$.

8. Écrire une fonction coutOpt2(r1 , r2) qui retourne un tableau d, de longueur 2, donnant une séquence de déplacements de coût optimal.

Partie C – Coût optimal pour trois requêtes

Dans cette partie, le cache est de taille 3 ($n = 3$). Il y a donc trois requêtes r_1 , r_2 et r_3.
Par convention, la première tête sera toujours celle qui bouge sur la première requête.

9. On suppose que la fonction de la question précédente a été étendue au cas de trois re- quêtes en appliquant la même règle de décision à la troisième requête qu'à la deuxième requête, c'est à dire que l'on chosit la tête la plus proche (minimisation locale).
   L'appliquer en justifiant sur l'exemple $[20, 9, 1]$. Comment nomme-t-on en générale une telle approche ?

10. Énumérer toutes les stratégies possibles sur l'exemple de la question précédente. En déduire que l'approche de la question précédente ne fournit pas la solution de coût minimal.

11. Écrire une fonction coutOpt3(r1 , r2 , r3) qui retourne un tableau d donnant une séquence de déplacements de coût optimal.

Partie D – Méthode gloutonne pour n requêtes

12. On implémente la méthode gloutonne pour un nombre quelconque de requêtes. On convient de commencer par déplacer la tête 1. Complèter et commenter le code suivant :

```
def glouton(r):
    n = len(?)    # ?
    p2 = 0        # ?
    p1 = ?
    cout = ?
```

```
    for i in range(1, ?):
        c1 = abs(r[i]-p1)  # ?
        c2 = abs(r[i]-p2)  # ?
        if c1 < c2 :
            p1 = ?         # ?
            cout += ?
        else:
            p2 = ?         # ?
            cout += ?
    return cout
```

13. Quelle est la complexité selon la longueur de la requête de la méthode gloutonne ? Commenter la pertinence de la méthode gloutonne.

Partie E – Coût optimal pour n requêtes

Dans cette partie, on calcule le coût minimal sans pour autant trouver une séquence de déplacements donnant ce coût, et celà en utilisant une méthode dynamique, avec le principe des sous-structures optimales.
Par commodité, chacune des deux têtes peut effectuer indifféremment le premier déplacement.
On pose $r_0 = 0$ pour coder la position initiale des têtes. À un instant donné, la configuration des têtes du disque dur est représentée par une paire (i, j) codant le numéro des deux dernières requêtes respectivement satisfaites par chacune des deux têtes : la première tête a satisfait en dernier la i ième requête et la deuxième tête la j ième requête. Par convention, la configuration initiale est $(0, 0)$. La dynamique sera basée sur ces notations.

À chaque requête r_k , on associe la matrice $(n + 1) \times (n + 1)$ représentée par le tableau d'entiers à deux dimensions cout_k. L'élément cout_k [i][j] est égal au coût optimal pour atteindre la configuration (i, j), après avoir satisfait la k ième requête. On pose cout_k[i][j]$= +\infty$ si cette configuration n'est pas accessible.

14. La dynamique est basée sur les propriétés suivantes :
    (1) cout_0[0][0]=0 et cout_0[i][j]=$+\infty$ pour tout $i \neq 0$ ou $j \neq 0$;
    (2) cout_k[i][k] est le minimum des $|r_k - r_j|+\text{cout}_{k-1}$[i][j] pour $0 \leqslant j \leqslant n$;
    (3) cout_k[k][j] = cout_k[j][k] ;
    (4) cout_k[i][j]=$+\infty$ si $i \neq k$ et $j \neq k$.

    Justifier les propriétés (1), (3) et (4).

15. Justifier la propriété (2).

16. On considère la requête $[20, 9, 1]$. Déterminer les matrices cout_k pour $k = 0, 1, 2, 3$. On obtiendra

$$\text{cout}_3 = \begin{pmatrix} +\infty & +\infty & +\infty & 39 \\ +\infty & +\infty & +\infty & 37 \\ +\infty & +\infty & +\infty & 32 \\ 39 & 37 & 32 & +\infty \end{pmatrix}$$

17. Expliquer de manière générale comment calculer le coût optimal d'une suite de requêtes $[r_1, r_2, \ldots, r_n]$ à l'aide du tableau correspondant cout_n.

18. Écrire une procédure mettreAJour(cout, r, k) qui met à jour le tableau cout en fonction de la nouvelle requête r_k , de sorte que si cout contenait les valeurs du tableau cout_{k-1}, alors, après la mise à jour, cout contient les valeurs du tableau cout_k. Pour $+\infty$, on utilisera float('inf'). On donne le squelette à compléter et commenter :

```
def mettreAJour(cout, r, k):
    n = len(r)
    rr = [0] + r                              # ?
    mat = [cout[i][:] for i in range(n+1)] # ?

    for i in range(n+1):
        for j in range(i, n+1):
            if i != k and j != k:  # ?
                cout[i][j] = ?
                cout[j][i] = ?
            else:
                c = min([? for j in range(n+1)])  # ?
                cout[i][k] = ?
```

```
                    cout[k][i] = ?
```

19. En déduire une fonction coutOpt(r) permettant de trouver le coût minimal du bloc de n requêtes r. Donner le temps d'exécution de coutOpt(r) par rapport à n. On donne un squellete à compléter et commenter :

```
def coutOpt(r):
    n = len(r)
    cout = [[float('inf')]*(n+1) for _ in range(n+1)] # ?
    cout[0][0] = ?
    for k in range(1, ?):
        mettreAJour( ? )
    return ?
```

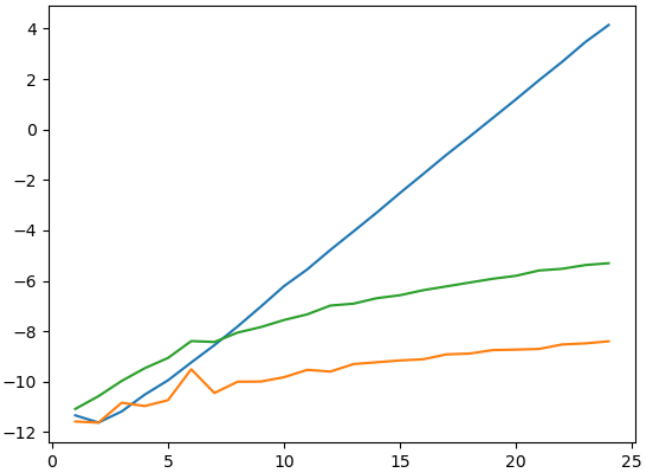
Donner la complexité de coutOpt en fonction de n la longueur de la requête r.

20. Que peut-on remarquer sur les matrices successives (cf l'exemple) ?
Expliquez pourquoi il semble qu'il soit nécessaire de ne retenir finalement qu'une ligne de la matrice.

21. Écrire une nouvelle fonction coutOpt2(r) qui calcule le coût minimal du bloc de n requêtes r en n'utilisant qu'un tableau cout à une dimension de taille $n+1$. Évaluer son nouveau temps d'exécution.

Indication : On remarquera (faire un dessin) que pour parvenir à la configuration (i,k), avec $i < k-1$, nécessairement on doit venir de la configuration $(i,k-1)$, en revanche pour la configuration $(k–1,k)$ on peut provenir de n'importe quelle configuration $(k–1,j)$.

22. On trace les courbes des logarithmes des temps d'exécution (ordonnées) des trois algorithmes d'optimisation du coût de déplacement des têtes selon la taille n (abscisses) de la requête r. Indiquer la corespondance entre les courbes et les algorithmes coutForce, coutOpt et coutOpt2 et justifier.
Comment devrait-on placer la courbe de l'algorithme glouton ?



1. Pour le bloc de requête $[7, 3, 8]$, le coût de la séquence de déplacements $[1, 1, 1]$ est de $7 + 4 + 5 = 16$; le coût de la séquence de déplacements $[2, 1, 2]$ est de $7 + 3 + 1 = 11$.

2. On donne le code complété :

```
def coutDe(r, d):
    n = len(r) # le nombre de déplacement à effectuer
    cout = 0  # initialisation du coût à 0
    p1 = 0    # position de la tête 1
    p2 = 0    # position de la tête 2
    for i in range(n): # i de 0 à n-1
        if d[i] == 1:  # si tête 1
            cout += abs(r[i]-p1) # le coût du déplacement en plus
            p1 = r[i]  # nouvelle position tête 1
        else:          # sinon tête 2
            cout += abs(r[i]-p2) # le coût du déplacement en plus
            p2 = r[i]  # nouvelle position tête 2
    return cout        # on retourne le coût total cherché
```

3. Pour chaque requêtes, nous avons 2 possiblités de choix de têtes. Ainsi pour n déplacements, nous avons 2^n choix, qui est donc le nombre de séquences de déplacements possibles associés à un bloc de requêtes r de longeur n.

4. Si on considère l'ensemble des coûts des 2^n possibilités, nous obtenons une partie non vide (finie, mais pas utilie pour un minimum, mais cela le serait pour un maximum) de $\mathbb{N}$ qui admet donc un plus petit élément. Il existe donc une (ou plusizeurs) séquences de de déplacements qui minimise le coût.

   Les roles des têtes peut être échangé puisque elles sont positionnées initialement en 0 toutes les deux. Ainsi il y a une parfaite symétrie des séquences de déplacements associées à leur coût. Il y a donc une séquence de de déplacements commençant par la tête 1 qui minimise le coût , (et sa symétrique qui commence par la tête 2).

5. La force brute nécessite de créer les 2^n séquences de déplacements, calculer leur coût tout en déterminant la valeur minimale. Nous avons un algorithme de complexité au moins exponentielle. Cela n'est pas raisonnable lorsque n grand. La symétrie divise la complexité par 2, certes, mais cela ne change en rien la complexité exponentielle (un an est mieux que 2 ans, mais si on vaut un résultat en quelques secondes ...).

6. Il y a 2^n possibilités de déplacements des têtes, et ainsi avec n bits, on peut représenter les entiers de 0 à $2^n - 1$, donc les 2^n possibilités. La représentation est alors naturelle, les 0 et 1 représentants par exemples les têtes 1 et 2 respectivement. Par exemple, avec $n = 3$, 011 représente la séquence de déplacements $[1, 2, 2]$ et 101 la séquence de déplacements $[2, 1, 2]$.

7. On donne l'algorithme en force brute, utilisant la notation binaire :

```
def couMinForce(r):
    n = len(r)
    cmin = float('inf')   # initialisation de cmin
    for k in range(2**n): # k de 0 à 2^n-1
        b = bin(k)[2:]    # il faut enlever les 2 premiers caractères 0b
        p = len(b)
        b = '0'*(n-p)+b   # on complète par des zéros devant, taille n
        c = coutDe(r, b)  # on calcule le coût
        if c < cmin :     # nouveau minimum
            cmin = c
    return cmin
```

8. Pour $[10, 3]$, on a soit $[1, 1]$ de coût $10 + 7 = 17$ soit $[1, 2]$ de coût $10 + 3 = 13$. On a ainsi $[1, 2]$ de coût minimum.

   Pour $[3, 10]$, on a soit $[1, 1]$ de coût $3 + 7 = 10$ soit $[1, 2]$ de coût $3 + 10 = 13$. On a ainsi $[1, 1]$ de coût minimum.

9. Par exemple :

```python
def coutOpt2(r1, r2):
    d = [1] # on commence par la tête 1
    if r2 <= abs(r1-r2): # c'est la tête 2 la plus proche
        d.append(2)      # on l'utilise
    else:
        d.append(1)
    return d
```

10. Sur l'exemple $[20, 9, 1]$, nous commençons par la tête 1 ;
ensuite, on choisit la tête 2 qui se déplace de 9, tandis que la tête 1 devrait se déplacer de 11 ;
ensuite, on déplace la tête la plus proche, qui est la 2.
La séquance selon cette méthode est $[1, 2, 2]$, de coût $20 + 9 + 8 = 37$.
Il s'agit d'une méthode dite gloutonne.

11. Il y a 4 stratégies :
$[1, 1, 1]$ de coût $20 + 11 + 8 = 39$
$[1, 1, 2]$ de coût $20 + 11 + 1 = 32$
$[1, 2, 1]$ de coût $20 + 9 + 19 = 48$
$[1, 2, 2]$ de coût $20 + 9 + 8 = 37$.
La méthode gloutonne n'est donc pas optimale.

12. Sans se fatiguer, par exemple :

```python
def coutOp3(r1, r2, r3):
    r = [r1, r2, r3]
    a = coutDe(r, [1,1,1])
    b = coutDe(r, [1,2,1])
    c = coutDe(r, [1,1,2])
    d = coutDe(r, [1,2,2])
    return min(a,b,c,d)
```

13. La propriété (1) est l'étape d'initialisation. Les deux têtes sont à l'origine, sans requête effectuée.
La propriété (3) est liée à la symétrie par rapport aux deux têtes vue en question 4.
La propriété (4) signifie qu'une seule tête a bougé lors de l'exécution de la dernière requête, et donc les configurations accesibles doivent contenir k : les autres ne sont pas accessibles.

14. Pour la propriété principale (2), cela est plus délicat bien sûr. C'est en fait le coeur du sujet.
Il faire appel à la notion de **sous-stucture optimale**, valide ici, c'est à dire que si une requête a été satisfaite de manière optimale jusqu'à l'ordre k, la sous-requête jusqu'à l'ordre $k-1$ l'est aussi forcément (sinon . . .).
Mais, en plus, il faut tenir compte de la dernière étape, et effectuer le minimum des différentes possibilités selon cette dernière étape, d'où la formule

$$\text{cout}_k\text{[i][k]} = \min_{0 \leqslant j \leqslant n} |r_k - r_j| + \text{cout}_{k-1}\text{[i][j]}$$

avec cout_k[i][k] le coût minimal après avoir satisfait la k requête que la tête 1 a effectué la requête i en dernier et la tête 2 la requête k en dernier ;
avec cout_{k-1}[i][j] le coût minimal après avoir satisfait la $k-1$ requête que la tête 1 a effectué i en dernier et la tête 2 j en dernier ;
avec $|r_k - r_j|$ étant le coût du déplacement de r_j vers r_k puisque c'est la tête 2 qui se déplace en dernier pour satisfaire la requête k, et que avant, la tête 2 était en position r_j, avec j un certain indice entre 0 et n (mais on a en fait $j \leqslant k-1$).

15. On considère la requête $[20, 9, 1]$. La requête augmentée est $[0, 20, 9, 1]$. La matrice est initialisée à

$$\begin{pmatrix} 0 & +\infty & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty \end{pmatrix}$$

Puis, pour la requête $k = 1$, pour $i \neq 1$ t $j \neq 1$, le coefficient est $+\infty$, soit déjà en tenant compte de la symétrie

$$\begin{pmatrix} +\infty & a & +\infty & +\infty \\ a & b & c & d \\ +\infty & c & +\infty & +\infty \\ +\infty & d & +\infty & +\infty \end{pmatrix}$$

Pour finir, en tenant compte de la formule (2) : minimum des $|r_1 - r_j| + \text{cout}_{k-1}[i][j]$, $0 \leqslant j \leqslant 3$:

$$\begin{pmatrix} +\infty & 20 & +\infty & +\infty \\ 20 & +\infty & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty \\ +\infty & +\infty & +\infty & +\infty \end{pmatrix}$$

Puis pour $k = 2$, minimum des $|r_2 - r_j| + \text{cout}_{k-1}[i][j]$, $0 \leqslant j \leqslant 3$:

$$\begin{pmatrix} +\infty & +\infty & 31 & +\infty \\ +\infty & +\infty & 29 & +\infty \\ 31 & 29 & +\infty & +\infty \\ +\infty & +\infty & & +\infty \end{pmatrix}$$

Puis pour $k = 3$, minimum des $|r_3 - r_j| + \text{cout}_{k-1}[i][j]$, $0 \leqslant j \leqslant 3$:

$$\begin{pmatrix} +\infty & +\infty & +\infty & 39 \\ +\infty & +\infty & +\infty & 37 \\ +\infty & +\infty & +\infty & 32 \\ 39 & 37 & 32 & +\infty \end{pmatrix}$$

Le coût minimum est alors de 32.

16. Il s'agit alors d''effectuer la dernière étape, déterminer le minimum des cout[i][j], pour i de 0 à n, c'est à dire après avoir effectué la dernière requête bien sûr, donc sur le tableau cout_n. On remarque sur l'exemple qu'en fait, on peut effectuer le minimum uniquement sur la dernière colonne.

17. On complète le code :

```
def mettreAJour(cout, r, k):
    n = len(r)
    rr = [0] + r  # r0 = 0
    mat = [cout[i][:] for i in range(n+1)] # copie pour ne pas écraser les
                                           # données lors de la mise à jour
    for i in range(n+1):
        for j in range(i, n+1):            # on utilise la symétrie
            if i != k and j != k:
                cout[i][j] = float('inf')  # (4)
                cout[j][i] = float('inf')  # (4)
            else:
                c = min([abs(rr[k]-rr[j])+mat[i][j] for j in range(n+1)]) # (2)
                cout[i][k] = c  # (2)
                cout[k][i] = c  # la même
```

La complexité est celle d'une double boucle $\mathcal{O}(n^2)$, avec un corps interne de complexité $\mathcal{O}(n)$ pour le calcul du minimum, soit une complexité globale en $\mathcal{O}(n^3)$.

18. En itérant la mise à jour :

```
def coutOpt(r):
    n = len(r)
    cout = [[float('inf')]*(n+1) for _ in range(n+1)] # tableau initialisé
    cout[0][0] = 0 # sauf en 0,0
    for k in range(1,n+1): # n mises à jour à effectuer
        mettreAJour(cout, r, k)  # mise à jour
    return min([cout[i][n] for i in range(n)])
```

On initialise le tableau avec $+\infty$ partout et on retourne le minimum du tableau (dernière colonne).
Nous avons une complexité en $\mathcal{O}(n^4)$, avec la question précédente puisque l'on effectue n mise à jour.

19. On remarque sur les tableau de la question 16 qu'une seule ligne (ou colonne) intervient finalement. Il doit donc être possible d'effectuer un algorithme nécessitant un tableau d'une seule dimension. Il faudra bien sûr bien étudier techniquement la mise à jour.

20. Une optimisation qui utilise la remarque (il faut faire un schéma pour comprendre la dynamique affinée ici)

```python
def coutOptPlus(r):
    n = len(r)
    rr = [0] + r
    cout = [float('inf')]*(n+1)
    cout[0] = 0
    for k in range(1, n+1):
        # on utilise la remarque
        # attention, on commence par mettre à jour la position k-1
        cout[k-1] = min([abs(rr[k]-rr[j]) + cout[j] for j in range(n+1)])
        # ensuite on peut modifier les autres
        for i in range(0,n+1):
            if i != k-1 :
                cout[i] = abs(rr[k]-rr[k-1]) + cout[i]
    return cout
```

Nous obtenons une complexité en $\mathcal{O}(n^2)$, le corps de la boucle en k étant en $\mathcal{O}(n)$, k allant de 1 à n. Nous gagnons deux ordres de grandeur.
Pour la complexité en mémoire, nous gagnons un ordre de grandeur.

21. Comme $\ln(n2^n) = n\ln(2) + \ln(n) \sim n\ln(2)$, la courbe en bleue est la courbe associée à la méthode en force brute.
On a $\ln(n^4) = 4\ln(n)$ et $\ln(n^2) = 2\ln(n)$,, et ainsi la courbe verte est la courbe associée à la méthode coutOpt et la courbe orange est la courbe asspciée à coutOpt2.
L'algorithme glouton est en $\mathcal{O}(n)$, soit $\ln(n)$ pour le graphique, il faudrait placer la courbe associée en dessous de la courbe orange, avec une même allure logarithmique.