CC3 – Algorithmique avancé

<u>1 heures</u>, **<u>documents non-autorisés.</u>** Le barème est donné à titre indicatif.

<u>Vous pouvez écrire vos algorithme dans votre langage de programmation préféré ou en pseudo code.</u>

Exercice 1. *Programmation dynamique : problème du rendu de monnaie (13 points)*

On considère la situation suivante. Vous êtes un marchand et vous devez rendre une certaine quantité d'argent exprimée en euros. Pour ça, vous disposez de pièces[1] de différentes valeurs, toutes en quantité illimitée mais vous voulez utiliser le moins de pièces possibles.
Par exemple, si vous avez des pièces de 1,2 et 5 euros et que vous voulez rendre 13 euros, vous pourriez rendre **13 pièces** de 1 euros, mais ce ne serait pas optimal. Vous pouvez en effet, ne rendre que **4 pièces** : deux de 5 euros, une de 2 euros et une de 1 euro.

Formellement, le problème d'optimisation auquel on s'intéresse est le suivant :
- Nom : Problème du rendu de monnaie.
- Entrée : un entier $k \geq 0$ encodé en binaire et un tableau d'entiers v de taille n où $v[i] > 0$ est la valeur de la i-ème pièce.
- Solutions admissibles : un tableau t d'entiers de taille n tel que $t[i]$ est le nombre de fois où on rend la i-ème pièce et tel que

$$\sum_{1 \leq i \leq n} t[i].v[i] = k.$$

- Fonction objectif (à minimiser) : nombre de pièces rendues. En d'autres termes :

$$c(t) = \sum_{1 \leq i \leq n} t[i].$$

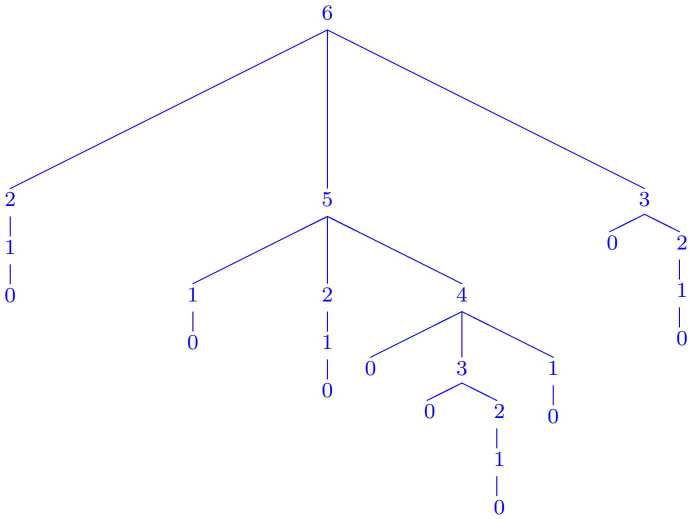
1. Écrire un algorithme récursif naïf $nombre\_pieces(k, v)$ qui retourne le nombre de pièces optimal du problème du rendu de monnaie. *(Vous pouvez supposez que v contient toujours la valeur 1 et qu'il existe donc toujours une solution.)*
☞

```python
def nombre_pieces(k,v):
        if k == 0:
                return 0
        meilleur = float("inf")
        for i in range(len(v)):
                if v[i] <= k:
                        meilleur = min(meilleur,nombre_pieces(k-v[i],v))
        return meilleur+1

print(nombre_pieces(6,[4,1,3]))
```

2. Dessiner l'arbre des appels récursif de votre algorithme sur l'entrée $k = 6, v = [4, 1, 3]$. Quel est la solution optimal et son nombre de pièces pour cette entrée ?
☞

1. On considérera que les billets sont des pièces ici.



La solution optimal est $t = [0, 0, 2]$ avec $c(t) = 2$. Autrement dit, on utilise seulement 2 pièces qui sont toute les deux de valeur 3.

3. Donner la complexité en temps de votre algorithme au pire des cas en fonction de n et k.

☞ En étant très violent : $O(n^k)$. À chaque récursion on a au pire n choix (une pour chaque valeur de pièce) et chaque pièce retire au moins 1 euros du rendu.

4. Le problème du rendu de monnaie en général est NP-dure. Cela-dit, si on suppose que k est polynomial par rapport à n, alors il existe des algorithmes efficaces. En utilisant les principes de la programmation dynamique, proposez un tel algorithme.

☞

Très similaire à la fonction naïve mais on mémorise les résultats intermédiaire pour ne pas les recalculer plusieurs fois le même sous-problème et on calcule de manière ascendante et pas descendante.

```
def nombre_pieces2(k,v):
        r = [float('inf')]*(k+1)
        r[0] = 0
        for j in range(1,k+1):
                meilleur = float("inf")
                for i in range(len(v)):
                        if v[i] <= j:
                                meilleur = min(meilleur,r[j-v[i]])
                r[j] = meilleur + 1
        return r[-1]
```

5. Quel est la complexité en temps et en espace de votre algorithme en fonction de k et n?

☞ En espace $O(k)$ la taille du tableau. En temps $O(kn)$ car calculer chaque case du tableau nécessite n opérations.

6. Réécrivez votre dernier algorithme pour qu'il vous renvoi la solution optimale plutôt que son nombre de pièces. (C'est-à-dire, au lieu de renvoyer $c(t)$, il doit maintenant renvoyer t.)

☞ On peut mémoriser dans un tableau l de taille $k + 1$ quelle pièce on prend à chaque étape pour arriver à la solution optimale. Cela nous permet de reconstituer le tableau t.

```
def nombre_pieces3(k,v):
        l = [None]*(k+1)
        r = [float('inf')]*(k+1)
        r[0] = 0
        for j in range(1,k+1):
                meilleur = float("inf")
                for i in range(len(v)):
                        if v[i] <= j and r[j-v[i]] < meilleur:
                                meilleur = r[j-v[i]]
                                l[j] = i
                r[j] = meilleur + 1
        t = [0]*len(v)
        j = k
```

```
        while j != 0:
                t[l[j]] += 1
                j -= v[l[j]]
        return t
```

Exercice 2. *Algorithme d'approximation : problème du rendu de monnaie (7 points)*

Maintenant, nous allons nous intéresser au même problème du rendu de monnaie mais en arrêtant de considérer que k est polynomialement borné par n. Dans ce cas le problème exacte devient NP-dure.

Vous (qui êtes encore un marchand) adoptez la stratégie suivante :

(a) Vous prenez la plus grosse pièce à votre disposition dont la valeur est plus petite que la somme qu'il vous reste à rendre à vous la donnez au client.

(b) Vous répétez cette opération jusqu'à ce qu'il ne reste plus d'argent à donner au client.

Encore une fois, on suppose que vous avez des pièces de 1 euros et que vous n'êtes donc jamais bloqué.

1. En utilisant cette stratégie gloutonne, combien de pièces de chaque type allez vous rendre sur l'entrée $k = 1\ 000\ 016$ et $v = [10, 8, 1]$? Est-ce que c'est une solution optimale ?

☞ Vous allez rendre 100007 pièces :

— 100 001 pièces de 10 euros de valeur total 1 000 010 (il vous reste 6 à rendre)

— 6 pièces de 1 euro (il vous reste 0 à rendre).

Ce n'est pas optimal car on pourrait rendre seulement 100002 pièces : 100 000 pièces de 10 euros et deux pièces de 8 euros.

2. Écrire un algorithme polynomial qui donne le nombre de pièces rendues avec cette stratégie gloutonne. Vous pouvez supposer que v est déjà trié par ordre décroissant.

Attention : comme k est codé en binaire, si votre algorithme tourne en $O(k)$, ce n'est pas polynomial. Par exemple, si $v = [1]$, et que votre algorithme répète une opération $k = k - 1$ jusqu'à ce que k soit égale à 0, votre algorithme prend un temps exponentiel en fonction de l'entrée.

☞ L'astuce est d'utiliser des divisions entières et modulos pour savoir combien de fois chaque pièce apparaît.

```
def nombre_pieces_glouton(k,v):
        v = list(reversed(sorted(v))) #pas necessaire
        nb = 0
        for i in range(len(v)):
                nb += k // v[i]
                k = k % v[i]
                print(v[i],v,nb,k)
        return nb
```

3. Est-ce qu'il existe un $\delta > 1$ tel que l'algorithme glouton est une δ-approximation ?

☞ Prenons $m = 4\delta$, $k = 2m$ et $v = [m + 1, m, 1]$. L'algorithme glouton renvoie $t = [1, 0, m - 1]$ qui utilise m pièces au lieu de la solution optimale $t^* = [0, 2, 0]$ qui utilise 2 pièces. Donc la solution gloutonne utilise $m/2 = 2\delta$ fois plus de pièce que la solution optimale. Donc, pour tout δ, on peut trouver une instance où le nombre de pièces rendues est strictement supérieur à δ fois le nombre de pièces de la solution optimale. Donc, l'algorithme n'est une δ-approximation pour aucun δ.

Pour information : Les ensemble de pièces v tel que l'algorithme glouton est optimal sont appelés des *systèmes de monnaie canoniques*. Les quasi-totalité des systèmes ayant cours dans le monde sont canoniques. Par exemple le système des euros $v = [50, 20, 10, 5, 2, 1]$ est canonique. Il existe d'ailleurs des algorithme très efficaces pour vérifier qu'un système de monnaie est canonique.