Informatique - DM 2 Corrigé

1. (a) Pour une longueur de 4 mètres, il y a 5 manières de couper :
   — $4 = 4$ qui donne un revenu de 6 €
   — $4 = 1 + 3$ qui donne un revenu de $1 + 5 = 6$ €
   — $4 = 1 + 1 + 2$ qui donne un revenu de $1 + 1 + 3 = 5$ €
   — $4 = 1 + 1 + 1 + 1$ qui donne un revenu de $1 + 1 + 1 + 1 = 4$ €
   — $4 = 2 + 2$ qui donne un revenu de $3 + 3 = 6$ €
   Le meilleur revenu est 6 € qui est obtenu par les coupes 4 (sans couper) ou 1 + 3 ou 2 + 2.

   (b) Pour un câble de 5 mètres, il y a 7 manières de couper et le meilleur revenu est de 8 € qui est obtenu pour une seule coupe : 2 + 3.

2. (a) Écrire une fonction `coupe(n)` qui renvoie la liste des schémas de coupe.

   On crée la liste de manière récursive en initialisant avec `[[]]` puis à chaque étape on rajoute `True` et `False` à chaque élément de la liste pour former la liste suivante. Après $n-1$ itérations, on a les 2^{n-1} schémas.

```python
def coupe(n):

    s = [[]]

    for k in range(n-1):
        L = []
        for A in s:
            L.append(A + [True])
            L.append(A + [False])
        s = L

    return s
```

   (b) Pour $n = 1$ la fonction doit renvoyer `[[]]` et pour $n = 2$ on doit obtenir `[[True], [False]]`. On le vérifie et on obtient `coupe(4)` :

```
>>> coupe(1)
[[]]
>>> coupe(2)
[[True], [False]]
>>> coupe(4)
[[True, True, True], [True, True, False], [True, False, True], [True,
False, False], [False, True, True], [False, True, False], [False,
False, True], [False, False, False]]
```

   (c) Écrire une fonction `revenu(s)` qui renvoie le revenu de la vente des morceaux correspondant à un schéma de coupe `s`.

   On parcourt le schéma. Tant qu'il n'y a pas de coupe à effectuer on incrémente une variable correspondant à la longueur du morceau courant. On ajoute son prix au total lors d'une coupe en ré-initialisant la longueur.

```
def revenu(s):

    r = 0        # revenu courant
    l = 1        # longueur du morceau courant

    for coupe in s:
        if coupe:
            r += prix[l]
            l = 1
        else:
            l += 1
    r += prix[l]  # ne pas oublier le prix du dernier morceau

    return r
```

`revenu([False, True, False, False])` donne : 8

(d) En déduire une fonction `revenu_max1(n)` qui renvoie un couple donnant le revenu maximal et un schéma correspondant.

```
def revenu_max1(n):

    L = coupe(n)
    r = 0    #revenu max courant

    for s in L:
        rev = revenu(s)
        if rev > r:
            r = rev
            c = s    #la coupe

    return (r, c)
```

Donner `revenu_max1(8)`.

```
>>> revenu_max1(8)
(13, [False, True, False, False, True, False, False])
```

(e) i.) Notons $C_1(n)$ la complexité de la fonction `coupe(n)`.

La fonction est constituée de deux boucles `for` imbriquées. La première est effectuée $n-1$ fois pour i allant de 0 à $n-2$. À chaque itération, la longueur de la liste `s` est multipliée par 2 ainsi `len(s)` $= 2^i$. La boucle imbriquée est effectuée `len(s)` fois et le coût de chaque itération est une constante, son coût est donc un $O(2^i)$. Or on sait que $\sum_{i=0}^{n-2} 2^i = 2^{n-1} - 1$ d'où il vient :

$$\boxed{C_1(n) = O(2^n).}$$

ii.) Notons $C_2(m)$ la complexité de la fonction `revenu(s)` avec $m =$ `len(s)`.

La boucle est effectuée m fois et dans le pire des cas, deux instructions sont effectuées à chaque itération, ainsi :

$$\boxed{C_2(m) = O(m).}$$

iii.) Notons $C(n)$ la complexité de la fonction `revenu_max1(n)` en fonction de n.
La boucle est effectuée `len(L)`$=2^{n-1}$ fois. À chaque itération, il y a au pire 3 instructions d'affectation et une comparaison exécutées auxquelles il faut ajouter l'appel à `revenu(s)` où `len(s)`$= n-1$. On a donc, en tenant compte des instructions qui ne sont pas dans la boucle (affectations et appel de `coupe(n)`) :
$C(n) = C_1(n) + 2^{n-1}(C_2(n-1) + O(1)) + 1 = O(2^n) + 2^{n-1}O(n-1) = O(n2^n)$
On a donc une très mauvaise complexité : $\boxed{C(n) = O(n2^n).}$

(f) On peut utiliser le code suivant :

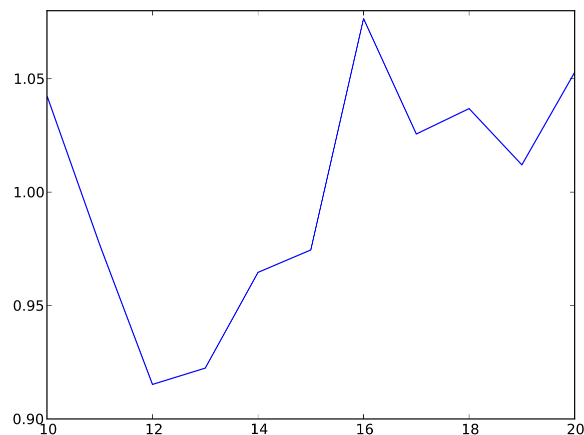
```
from time import time

def temps_moy(p):
    T = [0 for k in range(10, 21)]
    for i in range(p):
        for k in range(10, 21):
            t0 = time()
            revenu_max1(k)
            T[k-10] += time()-t0
    for k in range(11):
        T[k] /= p
    return T

T = temps_moy(5)  # les temps (moyens sur 5 executions)
T_q = [T[k-10]/(k*2**k) for k in range(10,21)]
# Les temps quotientes par k2^k
m = sum(T_q)/len(T_q) # la moyenne de l'ensemble des rapports
T_q_n = [T_q[k]/m for k in range(11)]  # Le tableau  normalise

from matplotlib import pyplot as plt
x = list(range(10, 21))
y = list(T_q_n)
plt.plot(x, y, 'b')
plt.show()
```

avec lequel on obtient, après quelques instants d'attente, une figure du type suivant :



Ici, les valeurs s'écartent peu de l'unité ; la complexité chronométrée est en assez bon accord avec le résultat théorique obtenu précédemment.

Remarque : si on remplace `temps_moy(5)` par `temps_moy(100)`, par exemple, on obtient un résultat plus conforme à la théorie car en « moyennant » davantage on réduit l'écart-type.

3. On considère un câble de longueur n.
   Pour une coupe quelconque, on note i la longueur du premier morceau [1]. La longueur totale des autres morceaux est donc de $n-i$, ainsi le revenu maximal pour une telle coupe est donc $r = p_i + r_{n-i}$.
   On en déduit que le revenu maximal pour toutes les coupes possibles (quel que soit la longueur du premier morceau) est donné par :

$$\boxed{r_n = \max_{1 \leqslant i \leqslant n} p_i + r_{n-i}}$$

4. **Un meilleur algorithme**

   (a) Écrire une fonction `r_max(R)` prenant en entrée la liste `R` des revenus maximaux pour les câbles de longueur strictement inférieure à n et renvoie le revenu maximal pour un câble de longueur n.

```
def r_max(R):

    n = len(R)
    r_maxi = 0

    for i in range(1, n + 1):
        r = prix[i] + R[n - i]
        if r > r_maxi:
            r_maxi = r

    return r_maxi
```

   (b) En déduire une fonction `revenu_max2(n)` qui renvoie le revenu maximal pour un câble de longueur n.

```
def revenu_max2(n):

    R = [0]

    for k in range(n):
        R.append(r_max(R))

    return R[-1]
```

   On obtient alors :

```
>>> revenu_max2(8)
13
```

   (c) On note $C'_1(n)$ la complexité de la fonction `r_max(R)` avec $n =$ `len(R)`.
   On a une unique boucle qui est exécutée n fois. On a donc :

$$\boxed{C'_1(n) = O(n)\ .}$$

   Notons maintenant $C'(n)$ la complexité de `revenu_max2(n)` en fonction de n.
   On a une unique boucle dans laquelle on appelle `r_max(R)`. À chaque itération, la longueur de la liste `s` est incrémentée d'une unité ainsi `len(s)` $= k + 1$. On a donc :

1. Remarquer que pour $i = n$ le raisonnement reste valide vu que $r_0 = 0$.

$$C'(n) = O(1) + \sum_{k=0}^{n-1} \big(O(1) + C_1'(k)\big). \quad \text{Or } \sum_{k=0}^{n-1} k = \frac{n(n-1)}{2}.$$

On a donc $\boxed{C'(n) = O(n^2).}$

5. (a) Écrire une fonction `r_max_coupe(R)` qui renvoie un couple donnant le revenu maximal et la longueur du premier morceau à couper.

```
def r_max_coupe(R):

    n = len(R)
    r_maxi = 0

    for i in range(1, n + 1):
        r = prix[i] + R[n - i]
        if r > r_maxi:
            r_maxi = r
            l = i

    return (r_maxi, l)
```

(b) En déduire une fonction `revenu_max_coupe(n)` qui renvoie le revenu maximal pour un câble de longueur n ainsi qu'une liste de longueurs correspondant à une solution maximale.

```
def revenu_max_coupe(n):

    R = [0]     # liste des revenus maximaux
    S = [0]     # liste des longueurs des premieres sections

    for k in range(n):
        (r, l) = r_max_coupe(R)
        R.append(r)
        S.append(l)

    L = []          # longueurs de coupes optimales
    while n != 0:
        L.append(S[n])    # on stocke la meilleure premiere coupe
        n = n - S[n]        # on actualise la nouvelle longueur a decouper

    return (R[-1], L)
```

On obtient par exemple :

```
>>> revenu_max_coupe(8)
(13, [2, 3, 3])
```