DS d'informatique n° 4 - Corrigé

Problème I : 1. Sur un pixel, il y a $256^3 = 2^{24} \approx 16.10^6$ possibilités (avec $2^10 \approx 10^3$). Un pixel est codé par 3 octets.

2. Le pixel en bas à gauche est `Img[h-1][0]`.
3. La taille en octets est $3840 \times 2160 \times 3 = 24883200$. Donc environ 24.9 Mo.
4.
```
def dim(Img):
        h = len(Img)
        w = len(Img[0])
        return w , h
```
5.
```
def nouvelle(w,h):
        Img = []
        for i in range(h):
                ligne = []
                for j in range(w):
                        ligne.append([255,255,255])
                Img.append(ligne)
        return Img
```
6.
```
def NbreCouleurs(Img):
        w,h = dim(Img)
        l = []
        for i in range(h):
                for j in range(w):
                        pix = Img[i][j]
                        est_dedans = False
                        for p in l:
                                if p == pix:
                                        est_dedans = True
                                        break
                        if not(est_dedans):
                                l.append(pix)
        return len(l)
```
7.
```
def Negatif(Img):
        w,h = dim(Img)
        neg = nouvelle(w,h)
        for i in range(h):
                for j in range(w):
                        pix = Img[i][j]
                        for k in range(3):
                                neg[i][j][k] = 255-pix[k]
        return neg
```

8. La complexité temporelle de `NbreCouleurs` est quadratique $O(N^2)$ car on vérifie l'appartenance d'un pixel à la listes des précédents. La taille de la liste `l` est au plus N (si tous les pixels sont différents) donc la complexité spatiale est linéaire $O(N)$.
   Pour `Negatif` les complexités spatiale et temporelle sont linéaires $O(N)$ car on parcours et recrée une image entière.

Problème II : 1. (a)

```
def facto(n):
        f = 1
        for k in range(n):
                f *= (k+1)
        return f
```

(b)

```
def binom1(n,k):
        return facto(n)//facto(k)//facto(n-k)
```

(c) La complexité de `facto` est linéaire en temps. Donc `binom1` est $O(n)+O(k)+O(n-k) = O(n)$ linéaire également.

2. (a)

```
def binom2(n,k):
        if k==0:
                return 1
        else:
                return binom2(n-1,k-1)*n//k
```

(b) La complexité en temps de cette fonction récursive est linéaire.

3. (a)

```
def ligne(n):
        if n == 0:
                return [1]
        l = [0]+ligne(n-1)+[0]
        return [l[k-1]+l[k] for k in range(n+1)]
```

(b)

```
def binom3(n,k):
        return ligne(n)[k]
```

La complexité de cette fonction est celle de `ligne`. La complexité spatiale et temporelle est quadratique $O(n^2)$ car il faut créer tout le triangle de Pascal et le conserver en mémoire.

4. En terme de complexité les deux premières méthodes semblent les plus efficace. Pourtant la seconde fait des calculs sur des nombres plus petit donc sur moins de 'bits'. Donc la seconde méthode est la meilleur. En effet les résultats des factorielles sont très grand et stockés sur des multi-entiers.

Problème III : 1. (a)

```
def vers_liste(nb):
        l,a = [],nb
        while a>10:
                l = [a%10]+l #append a l'envers
                a = a//10
        return [a]+l
```

(b)

```
def valide10(isbn10):
        l=vers_liste(isbn10)
        s=0
        for k in range(10):
                s += l[k]*(10-k)
        return (s%11==0)
```

(c)

```
def controle(livre):
        s=0
        for k in range(9):
                s += livre[k]*(10-k)
        return (-s)%11
```

2. (a) Avec l'ancienne norme 10^9 livres peuvent être encoder.
Avec la nouvelle norme 10^{12} livres peuvent être encoder.

(b)

```
def valide13(isbn13):
        l = vers_liste(isbn13)
        s=0
        for k in range(13):
                if k%2==0:
                        s += l[k]
                else:
                        s += 3*l[k]
        return (s%10 == 0)
def controle13(livre):
        s=0
        for k in range(12):
                if k%2==0: s += livre[k]
                else: s += 3*livre[k]
        return (-s)%10
```

(c)

```
def valide(isbn):
        if len(isbn)==10: return valide10(isbn)
        else: return valide13(isbn)
```

Problème IV : 1. (a)

fct1	l	r	x
	[1,5,7,8]	0	1
	[1,5,7,8]	1	5
	[1,5,7,8]	2	7
	[1,5,6,7,8]	2	7

fct2	l	i
	[1,5,7,8,8]	3
	[1,5,7,7,8]	2
	[1,5,6,7,8]	2

(b) Les fonctions insèrent dans une liste triée un nouvelle élément de sorte qu'elle soit encore triée.

(c) L'instruction `l=l+[a]` crée une copie intégrale de la liste et lui ajoute 'a'. Puis remplace l par cette nouvelle valeur. La complexité en temps et en mémoire est linéaire.

L'instruction `l.append(a)` ajoute directement 'a' à la liste. Sa complexité en temps et en mémoire est constante.

(d) Il y a une boucle simple pour détecter le rang d'insertion de complexité (au pire) linéaire en temps et constant en mémoire. L'instruction `l[:r]+[a]+l[r:]` recopie toute la liste. Elle est linéaire en temps et en mémoire dans tous les cas. Donc la complexité totale est $O(N)$ en temps et mémoire.

2. (a) L'instruction `i = i-1` montre que la suite des valeurs est strictement décroissante. Donc la condition `i>0` sera attente au pire des cas en N étapes.

Le nombre `i` est le rang d'insertion. Le pire des cas est l'élément est insérer au début i.e. 'a' est plus petit que la liste : $O(N)$. Le meilleur des cas est l'élément est plus grand que la liste : $O(1)$.

(b) Au fil de la boucle, si la liste est triée à l'étape 'i' alors la liste est triée à l'étape 'i-1'. En effet si $l_0 \leq ...l_{i-2} \leq l_{i-1} \leq l_i \leq l_i \leq l_{i+1}...$ alors $l_0 \leq ...l_{i-2} \leq l_{i-1} \leq l_{i-1} \leq l_i \leq l_{i+1}....$

A la fin, on a bien $l_{i-1} \leq a < l_i$ car la boucle s'arrête. Donc la liste devient $l_0 \leq ...l_{i-2} \leq l_{i-1} \leq a \leq l_i \leq l_{i+1}...$ avec le 'a' ajouter à sa place. Le programme est valide.

3. (a)

```
def quicksort(l,a=0,b=len(l)-1):
        if a<b:
                pivot = l[b]
                k = a-1 \# rang d'insertion du pivot
                for i in range(a,b):
                        if l[i]<pivot:
                                k += 1
                                l[k],l[i] = l[i],l[k]
                l[b],l[k+1] = l[k+1],l[b]
                quicksort(l,a,k)
                quicksort(l,k+2,b)
```

(b) Le code tri à chaque fois une liste déjà triée. Sa complexité est au moins $O(n\log(n))$ au lieu d'une insertion en $O(n)$.

(c) On peut extraire avec `file.pop()` et insérer avec `fct2(file,a)`.

4. On crée une file de priorité. Les éléments sont des tuples ('Nom du devoir', 'Date de rendu', 'durée'). On crée un score = 'Date de rendu'- 'durée'. Le plus petit étant

prioritaire. On classe la file de priorité suivant ce score. On extrait le devoir le plus prioritaire et on insère les autres devoirs en fonction de leurs priorité.