

Devoir surveillé n°2 - Corrigé

Exercice 1 : 1. Dans cet exercice, qui consiste à calculer la puissance d'un nombre avec un nombre de multiplications optimisé, il est bien sûr **hors sujet d'utiliser la fonction puissance ** préprogrammée de Python !**

```

1 def puissance(a, n):
2     ''' calcule a^n par l'algorithme d'exponentiation rapide '''
3     if n == 0:
4         return 1
5     elif n % 2 == 0:
6         p = puissance(a, n // 2) #on fait UN SEUL appel récursif
7         return p * p #on utilise le résultat de cet appel récursif
8     else:
9         return a * puissance(a, n - 1)

```

2. Déroulement de l'appel `puissance(2,10)` :

- `puissance(2,10)` (10 pair) appelle `puissance(2,5)`
- `puissance(2,5)` (5 impair) appelle `puissance(2,4)`
- `puissance(2,4)` (4 pair) appelle `puissance(2,2)`
- `puissance(2,2)` (2 pair) appelle `puissance(2,1)`
- `puissance(2,1)` (1 impair) appelle `puissance(2,0)`
- `puissance(2,0)` renvoie 1 (cas de base : 0 multiplication)

Remontée des valeurs et calcul des multiplications :

- `puissance(2,1)` effectue $2 \times 1 = 2$ (1 multiplication)
- `puissance(2,2)` effectue $2 \times 2 = 4$ (1 multiplication : $p * p$)
- `puissance(2,4)` effectue $4 \times 4 = 16$ (1 multiplication : $p * p$)
- `puissance(2,5)` effectue $2 \times 16 = 32$ (1 multiplication)
- `puissance(2,10)` effectue $32 \times 32 = 1024$ (1 multiplication : $p * p$)

Au total, l'exécution effectue exactement 5 multiplications.

Exercice 2 : 1. Somme de deux matrices parcimonieuses :

```

1 def somme(M,N):
2     ''' ajoute deux matrices parcimonieuses de même taille codées sous forme de dictionnaire '''
3     S={'dim':M['dim']} #le dictionnaire où sera stockée la matrice somme
4     for clef in M:
5         if clef!='dim':
6             if clef in N:
7                 somme=M[clef]+N[clef]
8                 if somme!=0: #si le coefficient est nul on ne le stocke pas
9                     S[clef]=somme
10            else:
11                S[clef]=M[clef]
12    for clef in N:
13        if clef!='dim':
14            if clef not in M:
15                S[clef]=N[clef]
16    return S

```

2. Pour calculer la complexité, nous devons remarquer deux choses :

- la première boucle "for clef in M" va tourner $(c_1 + 1)$ fois (le nombre de clefs dans M), et la deuxième boucle va tourner $(c_2 + 1)$ fois
- le test "if clef in N" va se réaliser **en temps constant** (particularité des dictionnaires : la recherche d'une clef dans un dictionnaire se fait en temps constant), de même que le test "if clef not in M".

La complexité de cette fonction est : $C(c_1, c_2) = 1 + (1 + c_1)(1 + 1 + 2) + (1 + c_2)(1 + 1 + 1) = 4c_1 + 3c_2 + 8$, donc la complexité est en $O(c_1 + c_2)$.

Exercice 3 : 1. Première méthode

(a) Fonction `test_longueur` :

```
1 def test_longueur(L):
2     """ renvoie le nombre de 1 consécutifs en début de liste """
3     i = 0
4     while i < len(L) and L[i] == 1:
5         i = i + 1
6     return i
```

(b) Dans le pire des cas, la liste ne contient que des 1 (ou débute par une plage de n fois le chiffre 1). La boucle `while` effectue alors n itérations avec des opérations à temps constant $O(1)$.

La complexité dans le pire des cas est donc linéaire : $O(n)$.

(c) Fonction `max_longueur` avec slicing :

```
1 def max_longueur(L):
2     """ renvoie la longueur de la plus grande sous-liste de 1 """
3     maxi = 0
4     n = len(L)
5     for i in range(n):
6         if L[i] == 1:
7             valeur = test_longueur(L[i:]) #on appelle UNE SEULE fois la fonction test_longueur
8             if valeur > maxi:
9                 maxi = valeur
10    return maxi
```

(d) Dans le pire des cas (par exemple pour une liste composée uniquement de 1), pour chaque indice i allant de 0 à $n - 1$:

- L'opération de tranche `L[i:]` crée une nouvelle liste de longueur $n - i$, ce qui nécessite $O(n - i)$ opérations.
- L'appel à `test_longueur(L[i:])` parcourt cette sous-liste en $O(n - i)$ opérations.

Le nombre total d'opérations est donc proportionnel à $\sum_{i=0}^{n-1} (n - i) = \frac{n(n + 1)}{2}$.

La complexité dans le pire des cas est donc quadratique : $O(n^2)$.

2. Deuxième méthode

Fonction `max_longueurV2` de complexité linéaire :

```

1 def max_longueurV2(L):
2     ''' L est une liste composée de 0 et de 1 '''
3     compteur=0
4     maxi=0
5     for x in L:
6         if x==1:
7             compteur+=1
8         if x==0:
9             compteur=0
10        if compteur>maxi:
11            maxi=compteur
12    return maxi

```

Cette fonction effectue un unique parcours de la liste L , en effectuant des affectations et comparaisons basiques à chaque étape.

La complexité temporelle est bien linéaire : $\mathcal{O}(n)$.

Exercice 4 (*Bonus pour ceux qui auraient tout fini*) : .

3. Suite des valeurs de n lors du calcul de a^{1000} :

- 1000 → 500 (pair, 1 mult)
- 500 → 250 (pair, 1 mult)
- 250 → 125 (pair, 1 mult)
- 125 → 124 (impair, 1 mult)
- 124 → 62 (pair, 1 mult)
- 62 → 31 (pair, 1 mult)
- 31 → 30 (impair, 1 mult)
- 30 → 15 (pair, 1 mult)
- 15 → 14 (impair, 1 mult)
- 14 → 7 (pair, 1 mult)
- 7 → 6 (impair, 1 mult)
- 6 → 3 (pair, 1 mult)
- 3 → 2 (impair, 1 mult)
- 2 → 1 (pair, 1 mult)
- 1 → 0 (impair, 1 mult)
- 0 (cas de base, 0 mult)

Il faut exactement 15 multiplications pour calculer a^{1000} .

La méthode naïve nécessiterait $1000 - 1 = 999$ multiplications. On réalise ainsi un gain considérable.

4. Lorsque n est pair, il est divisé par 2. Lorsqu'il est impair, il subit une soustraction de 1 puis est immédiatement divisé par 2 à l'étape suivante. La profondeur de la récursion est donc bornée par $2\lfloor \log_2(n) \rfloor + 1$. Chaque appel récursif effectue un nombre constant d'opérations.

La complexité temporelle est logarithmique : $\mathcal{O}(\log n)$.

5. **Analyse du code proposé :**

- **Maladresse commise :** Au lieu de stocker le résultat de `puissance(a, n // 2)` dans une variable pour effectuer une seule multiplication, le code réalise deux appels récursifs distincts identiques.
- **Impact sur la complexité :** La relation de récurrence devient $T(n) = 2T(n/2) + \mathcal{O}(1)$ dans le cas pair. L'arbre des appels devient un arbre binaire complet de profondeur $\log_2 n$, comportant $2^{\log_2 n} = n$ feuilles.

La complexité redescend en $\mathcal{O}(n)$ (linéaire), annulant tout l'intérêt de l'exponentiation rapide.