

Exercices

Exercice ITC3.1 : Calcul d'une somme [*]

On propose l'algorithme suivant où n est un entier :

```

1 def somme(n):
2     s=0
3     for i in range(n+1):
4         if i%2==0:
5             s=s+i
6     return s

```

1. Que calcule cet algorithme ?
2. Quelle est sa complexité en nombre d'additions, le paramètre de complexité étant n ?
3. À l'aide de vos connaissances de maths, proposez un algorithme en $O(1)$ qui fait la même chose.

Exercice ITC3.2 : Nombres premiers

Un entier naturel est premier s'il est supérieur ou égal à 2, et n'est divisible que par 1 et par lui-même. En pratique, il suffit, pour prouver qu'un nombre est premier, de vérifier qu'il n'est divisible par aucun entier compris entre 2 et $\sqrt{n}$ inclus.

1. La fonction suivante teste si un entier n est premier :

```

1 import math
2 def est_premier(n:int) -> bool:
3     i=2
4     while i<=math.sqrt(n):
5         if n%i==0:
6             return False
7         i=i+1
8     return True

```

Déterminez la complexité de cette fonction dans le pire des cas en fonction de n , en nombre de divisions (le modulo est en fait une division) puis en nombre de calculs de racine carrée.

2. Le calcul de la racine carrée de n est long ; proposez une modification du programme pour qu'elle ne soit calculée qu'une seule fois.
3. On appelle nombres premiers jumeaux deux nombres premiers qui sont séparés de 2. Par exemple 3 et 5, ou 11 et 13.

La fonction suivante renvoie la liste de tous les nombres premiers jumeaux inférieurs ou égaux à un entier n donné.

```

1 def liste_premiers_jumeaux(n:int)->list:
2     L_jumeaux=[]
3     for i in range(2,n-1):
4         if est_premier(i) and est_premier(i+2):
5             L_jumeaux.append([i,i+2]) # on stocke le couple (i,i+2)
6     return L_jumeaux

```

Déterminer sa complexité en nombre de divisions + nombre de `append`, dans le pire des cas, en fonction de n .

4. On propose une autre fonction :

```

1 def liste_premiers_jumeaux2(n:int)->list:
2     L_premiers=[]
3     for i in range(2,n+1):
4         if est_premier(i):

```

```

5 |         L_premiers.append(i) # on stocke i
6 |     L_jumeaux=[]
7 |     for j in range(len(L_premiers)):
8 |         if L_premiers[j+1]==L_premiers[j]+2:
9 |             L_jumeaux.append([j,j+2])
10 |     return L_jumeaux

```

Il y a une erreur dans la seconde boucle ; corrigez-la.

Quelle est la complexité de cette fonction ? Quel gain apporte-t-elle par rapport à la précédente ?

Exercice ITC3.3 : Recherche de doublets dans une liste d'entier **[**]**

On dispose d'une liste L d'entiers compris entre 0 et $N_{max} = 100$ (inclus). On souhaite écrire un algorithme qui détermine si cette liste contient un doublon (c'est-à-dire s'il existe un entier qui apparaît au moins deux fois) : si c'est le cas, il renvoie cet entier, sinon il renvoie -1.

1. Premier algorithme : recherche naïve.

L'algorithme le plus simple consiste à parcourir deux fois la liste dans une double boucle de manière à comparer toutes les paires de termes de la liste :

```

1 | n ← longueur de L
2 | pour i allant de 0 à n-1 faire
3 |     pour j allant de 0 à n-1 faire
4 |         si i différent de j et  $L[i]=L[j]$  alors
5 |             retourner  $L[i]$ 
6 |         fin si
7 |     fin pour
8 | fin pour
9 | retourner -1

```

Déterminez la complexité de cet algorithme en nombre de comparaisons ($!=$ et $==$) par rapport au paramètre n .

2. Comment peut-on améliorer cet algorithme pour faire deux fois moins de comparaisons ? Est-ce que cela change la complexité asymptotique ?
3. Second algorithme : avec mémorisation.

La mémorisation consiste à utiliser un peu plus de mémoire pour stocker des résultats précédents de manière à gagner du temps pour la suite de l'algorithme. Ici, comme on sait que les termes de la liste sont compris entre 0 et N_{max} , on va créer un tableau qui se souvient des entiers qu'on a déjà rencontrés.

```

1 | n ← longueur de L
2 | memo ← liste contenant (Nmax+1) zéros
3 | pour i allant de 0 à n-1 faire
4 |     si memo[L[i]]=1 alors
5 |         retourner  $L[i]$ 
6 |     sinon
7 |         memo[L[i]] ← 1
8 |     fin si
9 | fin pour
10 | retourner -1

```

Déterminez la complexité de cet algorithme en nombre de comparaisons.

4. Comparez les deux algorithmes en terme de complexité et en terme de mémoire.

Exercice ITC3.4 : Multiplication de deux entiers **[**]**

On souhaite écrire un algorithme qui calcule le produit de deux entiers a et b positifs avec $a > b$. Le paramètre de complexité est b .

1. On propose un premier algorithme simple :

```

1 def produit1(a,b):
2     p=0
3     q=b
4     while q>0:
5         q=q-1
6         p=p+a
7     return p

```

Déterminez la complexité de cet algorithme en nombre d'opérations élémentaires (addition, soustraction).

2. On propose un autre algorithme plus compliqué :

```

1 def produit2(a,b):
2     p=0
3     puissance=a
4     q=b
5     while q>0:
6         if q%2==1:
7             p=p+puissance
8             puissance=puissance*2
9             q=q//2
10    return p

```

Remarque : on peut s'étonner de trouver une multiplication et une division dans un algorithme fait pour calculer un produit ; en fait, les opérations de multiplication et de division **par 2** sont des opérations élémentaires en langage binaire, correspondant simplement à décaler d'un bit le nombre (équivalent à multiplier ou diviser par 10 en base 10).

- Donnez une majoration simple de la valeur de q après n tours de boucle.
- Déduisez-en que la boucle est parcourue au maximum $\log_2(b)$ fois.
- Déduisez-en la complexité de cet algorithme en nombre d'opérations élémentaires (addition, multiplication/division par 2).

Exercice ITC3.5 : Tri à bulles [**]

On dispose d'une liste L de nombres ; on veut la trier dans l'ordre croissant. On propose l'algorithme suivant :

```

1 n ← longueur de L
2 pour i allant de 1 à n-1 faire
3     j ← i
4     x ← L[i]
5     tant que j>0 et L[j-1]>x faire
6         L[j] ← L[j-1]
7         j ← j-1
8     fin tq
9     L[j] ← x
10 fin pour

```

- Déterminez la complexité de cet algorithme en nombre de comparaisons et en nombre d'affectations, le paramètre de complexité étant n :
 - dans le pire des cas
 - dans le meilleur des cas
- Notons $S(n)$ le nombre moyen de comparaisons dans le cas général.
 Considérons une liste de $n + 1$ termes. L'algorithme commence par trier les n premiers termes, puis il fait remonter le $n + 1$ -ième terme. Combien de comparaisons va-t-il faire pour cela (donner un intervalle) ? Combien en moyenne ? En déduire une relation de récurrence entre $S(n + 1)$ et $S(n)$, puis déterminer la complexité moyenne de cet algorithme.

Exercice ITC3.6 : Recherche du maximum d'une liste [***]

On dispose d'une liste L de nombre réels, de longueur n .

1. Écrivez un algorithme (en Python ou en pseudo-code) qui recherche et renvoie le maximum de cette liste.
2. Quelle est sa complexité en nombre d'affectations dans le meilleur des cas ?
3. Quelle est sa complexité en nombre d'affectations dans le pire des cas ?
4. Notons $S(n)$ le nombre moyen d'affectations effectuée par cet algorithme pour une liste de longueur n .

On considère une liste de longueur n . Quelles valeurs peut avoir l'indice i de l'élément maximal ? Quelle est la probabilité qu'il ait une valeur particulière ?

En décomposant la recherche du maximum de la liste en 3 étapes : les termes de 0 à $i - 1$, le terme d'indice i , puis le reste, établir une relation entre $S(n)$ et les $S(i < n)$.

5. En déduire que S vérifie la relation de récurrence $S(n) = S(n - 1) + \frac{1}{n}$.

Déduisez-en que la complexité moyenne de cet algorithme en nombre d'affectations est en $O(\log(n))$.

Corrections des exercices

Corrigé de l'exercice ITC3.1 : Calcul d'une somme [*]

1. Il calcule la somme des entiers pairs inférieurs ou égaux à n .
2. La boucle est faite $n+2$ fois et une fois sur deux il y a une addition, donc le nombre d'additions est $\frac{n+2}{2}$ (arrondi) donc la complexité est en $O(n)$.
3. Notons $n = 2p$ si n est pair et $n = 2p + 1$ s'il est impair (c'est-à-dire $p = n//2$ - division entière). Alors on a la somme des entiers pairs de 2 à $2p$, c'est-à-dire $\sum_{i=1}^p 2i = 2\frac{p(p+1)}{2} = p(p+1)$. On peut donc proposer :

```
1 def somme(n):  
2     p=n//2  
3     return p*(p+1)
```

qui exécute 3 opérations, donc en $O(1)$.

Corrigé de l'exercice ITC3.2 : Nombres premiers

1. Dans le pire des cas, le nombre est premier et la boucle s'exécute $\sqrt{n} - 1$ fois, avec une division à chaque fois, donc la complexité est en $O(n^{1/2})$ pour les divisions. Il en est de même pour les calculs de racine.
2. On peut améliorer cela en ajoutant `racine=math.sqrt(n)` avant la boucle, puis `while i<=racine:`
3. La boucle est effectuée environ n fois, et à chaque fois un appelle 2 fois la fonction précédente, plus un `append`, d'où un nombre d'opérations de l'ordre de $2.n.n^{1/2}$ donc la complexité est en $O(n^{3/2})$.
4. La première boucle est effectuée n fois et appelle une fois le test de primalité plus un `append`, donc le nombre de division est de l'ordre de $n^{3/2}$.
La seconde boucle doit être `for j in range(len(L_premiers)-1):`; elle est effectuée au pire n fois et effectue un `append`.
La complexité complète est donc encore en $O(n^{3/2})$, on a gagné cependant un facteur 2 au pris d'une utilisation plus grande de la mémoire.

Corrigé de l'exercice ITC3.3 : Recherche de doublets dans une liste d'entier [**]

1. Dans la boucle intérieure (en j), on effectue 2 comparaisons. Cette boucle est effectuée n fois ce qui fait $2n$ comparaisons. On prend alors en compte la boucle en i qui effectue n fois les $2n$ comparaisons précédentes, donc au final on effectue $n \times 2n = 2n^2$ comparaisons. La complexité est donc en $O(n^2)$.
2. En remplaçant `j allant de 0 à n-1` par `j allant de i+1 à n-1` on effectue deux fois moins de comparaisons, mais la complexité reste en $O(n^2)$.
3. Ici on a une seule boucle dans laquelle est effectuée 1 comparaison et 1 affectation. La complexité sera alors en $O(n)$.
4. Le second algorithmes est plus rapide pour n grand, mais prend plus de mémoire.

Corrigé de l'exercice ITC3.4 : Multiplication de deux entiers [**]

1. La boucle est parcourue b fois, et contient deux opérations à chaque fois, donc la complexité est en $O(b)$.
2.
 - (a) q subit une division entière par 2 à chaque tour de boucle, donc il est divisé par au moins 2, donc au bout de n tours de boucle, il aura été divisé par au moins 2^n , ce qui implique $q \leq \frac{b}{2^n}$.
 - (b) La boucle s'arrête lorsque $q \leq 1$ (car q est entier) donc elle s'arrête si $\frac{b}{2^n} \leq 1$ soit $n \geq \log_2(b)$.
 - (c) À chaque tour de boucle il y a 3 opérations élémentaires ce qui donne une complexité en $O(\log(n))$.

Corrigé de l'exercice ITC3.5 : Tri à bulles [**]

1. La grande boucle s'effectue n fois; la boucle intérieure s'effectue :

- dans le meilleur des cas, aucune fois, ce qui donne 2 comparaisons et 3 affectations, d'où une complexité en $O(n)$ pour les deux opérations
 - dans le pire des cas, i fois, ce qui donne $2i$ comparaisons et $2i + 3$ affectations, d'où une complexité en $\sum_{i=0}^{n-1} 2i = n(n+1) = O(n^2)$ (en comparaison, et aussi en affectations car $n(n+1) + 3n = O(n^2)$).
2. Pour faire remonter le $n + 1$ -ième terme, on va effectuer entre 2 et $2n$ comparaisons, soit en moyenne $n + 1$ comparaisons si le nouveau terme est réparti de façon aléatoire par rapport aux autres. On a donc $S(n+1) = S(n) + n + 1$ avec $S(0) = 0$ ce qui donne $S(n) = \frac{n(n+1)}{2}$ donc la complexité moyenne est en $O(n)$ avec cependant un préfacteur moitié de la complexité dans le pire des cas.

Corrigé de l'exercice [ITC3.6](#) : Recherche du maximum d'une liste [***]

```

1.
1 def maximum(L):
2     max=L[0]
3     for i in range(1,len(L)):
4         if L[i]>max:
5             max=L[i]
6     return max

```

2. Dans le meilleur des cas, le maximum est le premier terme, donc 1 seule affectation est faite, donc la complexité est en $O(1)$.
3. Dans le pire des cas, la suite est strictement croissante, et il y a une affectation à chaque tour de boucle, soit n affectations au total, donc la complexité est en $O(n)$.
4. Le maximum peut être en tout i compris entre 0 et $n - 1$ avec une équiprobabilité $\frac{1}{n}$.
Si le maximum est en i , alors l'algorithme effectue en moyenne $S(i - 1)$ affectations jusqu'à $i - 1$, puis une affectation à i , puis plus rien, soit $S(i - 1) + 1$ affectations. En tout, on aura donc $S(n) = \sum_{i=0}^{n-1} \frac{1}{n} (S(i - 1) + 1)$
5. $S(n - 1) = \sum_{i=0}^{n-2} \frac{1}{n-1} (S(i - 1) + 1)$ donc on voit que $nS(n) = (n - 1)S(n - 1) + S(n - 1) + 1$ soit $nS(n) = nS(n - 1) + 1$ donc $S(n) = S(n - 1) + \frac{1}{n}$.
Comme $S(1) = 1$ on en déduit que $S(n) = \sum_{i=1}^n \frac{1}{i}$ qui est une suite dominée par $\ln(n + 1)$ (car $S(n) < \sum_{i=1}^n \int_i^{i+1} \frac{1}{x} dx = \int_1^{n+1} \frac{1}{x} dx = \ln(n + 1) - \ln(1)$) donc par $\log(n)$ (car tous les logarithmes sont du même ordre à un facteur près).