Partie 1

Exercices

1.1 Exercice 0 :

1.1.1 suites arithmétiques

Une suite arithmétique (u_n) de raison `r` peut être définie par la formule de récurence :

$$u_{n+1} = u_n + r$$

Le tableau suivant montre 2 listes calculées à partir de suites arithmétiques.

B10 =B9+4

	A	B	C	D
1	0	2		
2	1	6		
3	2	10		
4	3	14		
5	4	18		
6	5	22		
7	6	26		
8	7	30		
9	8	34		
10	9	38		
11		200		
12				

FIGURE 1 – tableau Excel

1. Proposer une formule générale pour le calcul de la somme des termes d'une suite arithmétique.
2. Calculer la somme des termes de la colonne de gauche
3. Calculer la somme des termes de la colonne de droite
4. Soit `u(n)` la fonction python qui calcule et retourne la valeur de `u` au rang `n`. Ecrire une fonction `somme_arithmetique` en python qui calcule la somme des `n` premiers termes de la suite `u(n)` et retourne la valeur calculée. Cette fonction utilisera une boucle bornée, et non la formule proposée au 1.

1.1.2 Fonction logarithmique

Le logarithme en base a d'un nombre x strictement positif peut être calculé à partir de :

$$log_a(x) = \frac{log(x)}{log(a)}$$

En informatique, on uitlise préferentiellement la fonction logarithmique en base 2.

1 Calculer les logarithmes en base 2 de 8, 16, 32, 64. Que remarquez-vous ? 2. Combien de fois successives faut-il diviser 8 par 2 pour arriver à 1 ? 3. Même question pour 16 puis 32. Conclure : Exprimer d'une autre manière ce que signifie le logarithme en base 2 d'un nombre.

1.2 Exercice 1

```
1 def multiplie1(b,n):
2   L=[]
3   for i in range(n):
4     L.append(b*i)
5   return L
```

Le programme execute n fois la ligne 4. Le nombre d'opérations significatives effectuée est $T(n) = n$: On aura pour `multiplie1`une classe de complexité algorithmique

$$O(n)$$

.

Si on ajoute des lignes dans la boucle `for`, pour faire par exemple :

```
def multiplie2(b,n):
  L=[]
  for i in range(n):
    y = b * i
      L.append(y)
    return L
```

Le nombre d'opérations significatives effectuée par `multiplie2` est $T(n) = 2 \times n$.

C'est deux fois plus que `multiplie1`. Or cette différence ne vient que d'une différence des **details d'implémentation** du même algorithme, et ne doit pas être considérée pour le calcul de la complexité.

On aura aussi pour `multiplie2` une classe de compléxité algorithmique

$$O(n)$$

.

Enfin, ce même algorithme peut être implémenté avec une boucle non bornée :

```
def multiplie3(b,n):
  L=[]
  i = n - 1
  while i >=0 :
    y = b * i
    L.append(y)
    i -= 1
  return L
```

1. Pour les fonctions `multiplie1` et `multiplie2` : Enoncer un ensemble de règles pour déterminer T(n).
2. Pour les fonctions `multiplie1` et `multiplie2` : Enoncer une règle pour évaluer la complexité algorithmique $O(g(n))$ à partir de T(n).
3. Déterminer T(n) pour `multiplie3`. Vérifier que l'on obtient aussi une classe de complexité $O(n)$ pour `multiplie3`.

1.3 Exercice 2

```python
def truc(n):
  res=0
  for i in range(0,n):
    res = res + 1
  return res

def chose(n):
  res=0
  for i in range(n):
    res = res + truc(i)
  return res
```

1. Déterminer T(n) et $O(g(n))$ pour la fonction `truc`
2. Idem pour la fonction `chose`

1.4 Exercice 3

On donne l'une des fonctions de l'interface de programmation du `serpent` dans le jeu *Snake* :

```python
def supprime_queue(S):
    # decale toutes les valeurs de la liste S vers la gauche:
    # copie toutes les valeurs S[i+1] dans S[i]
    # puis supprime le dernier element
    for i in range(len(S)-1):
        S[i] = S[i+1]
    S.pop()
```

1. Determiner la fonction T(n) qui exprime le nombre d'opérations effectuées par la fonction pour une liste de taille `n`
2. Déterminer la classe de complexité algorithmique **O(g(n))** de cette fonction.

1.5 Exercice 4 : Recherche dans un jeu de cartes

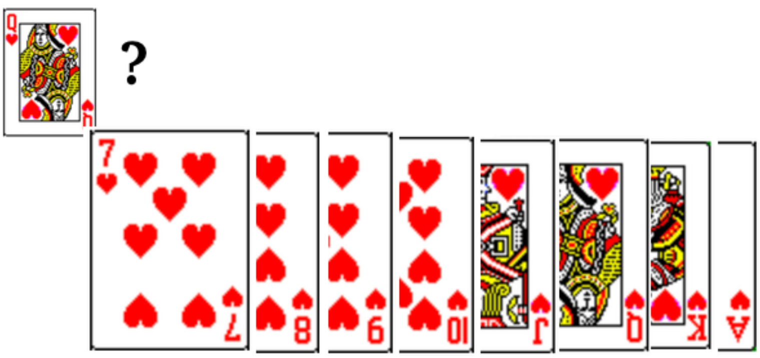

FIGURE 2 – cartes triées

1. Ecrire une liste L représentant le jeu de cartes de l'image. La carte qui a pour valeur 7 sera représentée par l'entier 1, puis celle de valeur 8 aura la valeur 2, etc ... jusqu'à l'As qui vaut 8.

2. Expliquer avec une méthode de votre choix comment l'algorithme de recherche réduit cette liste jusqu'à trouver la carte de la Dame de Coeur. Comparer ainsi l'efficacité des 2 algorithmes, celui de recherche sequentielle et celui de recherche dichotomique.

1.6 Exercice 5 : Tri par selection

Principe :

On recherche le plus petit élément et on le met à sa place (en l'échangeant avec le premier). On recherche le second plus petit et on le met à sa place, etc.

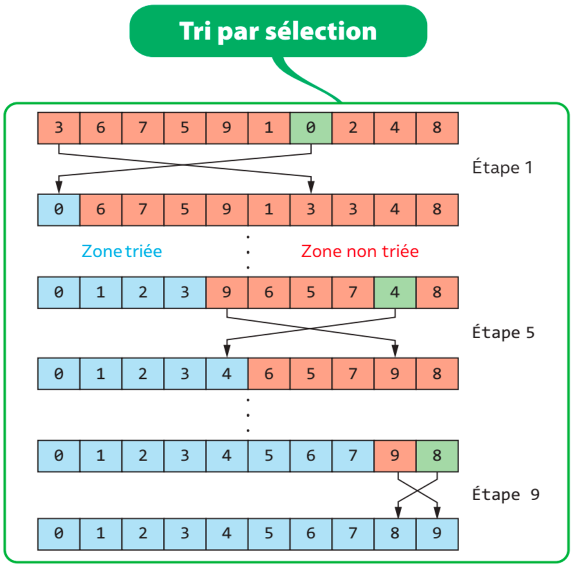

FIGURE 3 – illustration du tri par selection

```python
def tri_selection(L):
  n = len(L)
  for i in range(0, n - 1):
    #recherche le plus petit élément de i à la fin
    mini = i
    for j in range(i + 1, n):
        if L[j] < L[mini]:
          mini = j
    #échanger les cases i et mini
    tmp = L[i]
    L[i] = L[mini]
    L[mini] = tmp
```

1. Dans un tableau : Représenter la liste L entre les étapes 1 et 5. Indiquer à chaque fois le nombre d'opérations effectuées.

2. Supposons maintenant que les éléments de la liste L à trier sont rangés en sens inverse. Cela va-t-il augmenter ou diminuer le nombre d'opérations effectuées pour trier cette liste ?

3. Calculer la complexité dans le pire des cas **O(g(n))** de cet algorithme.