Partie 1

Exercice 1

Déterminer la complexité des fonctions suivantes en terme de nombre d'additions et de soustractions. On donnera d'abord la valeur exacte T(n) puis l'ordre de grandeur O(n).

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


def machin(n):
  res=truc(n)
  for i in range(0,n):
    res -=1
  return res

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

def fonctionFinale(n):
  res =[]
  for i in range(0,n):
    res.append(chose(n))
  return res
```

Partie 2

Exercice 2 : Recherche dans un jeu de cartes

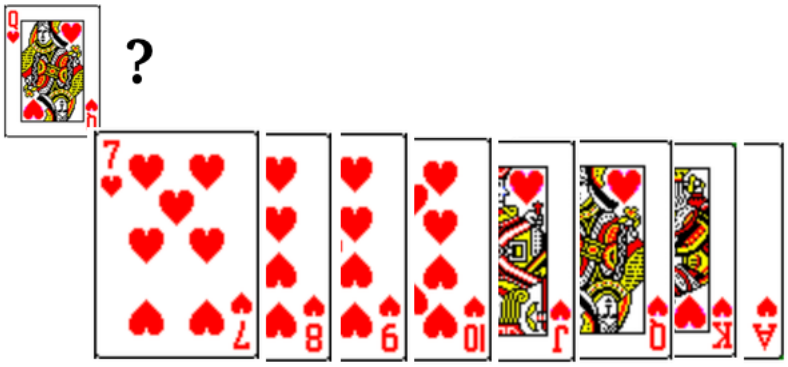

FIGURE 1 – cartes triées

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.

Partie 3

Exercice 3 : 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.

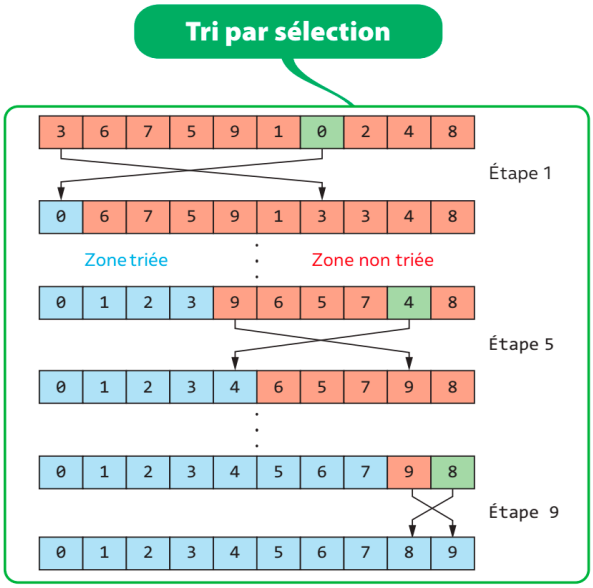

FIGURE 2 – illustration du tri par selection

```
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.

Partie 4

COURS : Evaluer la qualité d'un algorithme

Un même algorithme peut avoir plusieurs *implémentations* pour la même *spécification*. Pourtant, la qualité d'un algorithme varie beaucoup selon cette implémentation. On cherche alors des critères de mesure de cette qualité :

4.1 Complexité spatiale

C'est une mesure sur les données qui réflète la quantité d'informations contenues : le nombre et le contenu des variables.

4.2 Complexité temporelle

4.2.1 Definition

C'est une estimation du temps d'execution d'un programme, independamment de la machine. En pratique, cela correspond au nombre d'opérations effectuées par le programme. Ce nombre d'opérations dépendant de la taille n des données en entrée, on évalue une fonction **g(n)**.

4.2.2 Exemples

Soient 2 implémentations du même algorithme :

```
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. Sa complexité est égale à n :

$$g(n) = n$$

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

```
1 def multiplie2(b,n):
2   L=[]
3   for i in range(n):
4     y = b * i
5       L.append(y)
6     return L
```

On pourrait penser que la complexité de `multiplie2` est $g(n) = 2 \times n$. 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 alors pour cette 2^e fonction :

$$g(n) = n$$

.

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

```
1 def multiplie3(b,n):
2   L=[]
3   i = n - 1
4   while i >=0 :
5     y = b * i
6     L.append(y)
7     i -= 1
```

```
8     return L
```

Pour déterminer le nombre d'itérations de la boucle, on définit un **variant de boucle**. C'est une quantité qui **décroit** à chaque itération. C'est lui qui va amener la boucle à se terminer : Trouver un variant de boucle démontre que la boucle **termine**.

On repère alors sa valeur initiale, sa valeur finale, ainsi que le nombre d'itérations de la boucle.

Exemple : Avec `multiplie3`, le variant de boucle, c'est `i`. Sa valeur passe de n - 1 à 0. Le nombre d'instructions élementaires dans la boucle est de 3.

La complexité serait alors $g(n) = 3 \times n$, mais on prendra $g(n) = n$

4.2.3 Notation de Landau

Le plus souvent, il s'agira d'estimer la complexité dans le pire des cas, exprimée sous la forme : $O(g(n))$.

Pour l'exemple précédent, la complexité est notée **O(n)** en notation de Landau.

Lorsqu'il est possible de déterminer une fonction asymptotique de la complexité, la notation devient Θ(g).

Partie 5

Règles pour estimer la complexité O(g(n))

5.1 Règles

- Poser n = "la taille des paramètres".
- Ne compter que les **instructions essentielles** à partir d'une unité de mesure : T(n)
- Pour chacune des boucles du programme, repérer le **variant de boucle** et calculer le nombre d'itérations : combien de fois on passe dans la boucle.
- Puis calculer une estimation d'une borne supérieure f(n) de T(n) :
  - Si T(n) contient une somme de termes, conserver uniquement le plus divergent.
  - g(n) est alors estimé à partir de T(n) : Ne pas considérer les multiplicateurs C : si T(n) = C.f(n), alors g(n) = f(n). Par exemple, si T(n) = 3*n, prendre g(n) = n.
  - Sauf précision contraire, la complexité demandée est la complexité au pire en temps.

5.2 Instructions élémentaires T(n)

T(n) est la somme des unités de mesure, comptée pour chaque instruction.

La première étape est d'identifier les séquences dans un algorithme. Si votre algorithme est composé des séquences :

$$I_1 I_2 I_3 ...$$

Alors :

$$T(n) = T_{I_1}(n) + T_{I_2}(n) + T_{I_3}(n) + ...$$

Dans la fonction `multiplie1`, il y a 3 séquences I1, I2, I3 :

```
1 # sequence I1
2   L=[]
```

```
1 # sequence I2
2   for i in range(n):
3     y = b * i
```

```
4       L.append(y)
```

```
1   # sequence I3
2       return L
```

Les séquences I1 et I3 contiennent chacune une seule instruction élémentaires. Cependant, I2 est une séquence complexe. De ce fait :

$$T(Somme) = 2 + T(I_2)$$

Une unité de mesure peut-être :

- une addition
- une soustraction
- une multiplication, une division, Mod (%), Div
- une opération arithmétique simple (sans appel de fonctions)
- une comparaison, les opérations booléennes (et,ou,non)
- une affectation
- des opérations de lectures et écritures simples

En pratique, on considèrera qu'il n'y a pas de différence entre les 3 opérations suivantes (à moins que l'énoncé donne une consigne différente) :

- a =b ;
- a=b*c ;
- a=a+b*c ;

5.3 Boucles

Le calcul de la complexité ne doit pas dépendre du type de boucle, et donc du type d'algorithme. On ne considèrera pas, sauf mention contraire :

- l'initialisation de la variable utilisée comme variant de boucle
- l'incrémentation de cette variable
- la comparaison avec la valeur d'arrêt

Sinon, il faudra compter de la manière suivante :

$$T(n) = 1 + \Sigma_{j=1,\ldots,n}(2 + T_j(I))$$

5.4 Instructions conditionnelles

5.4.1 conditionnel simple

```
1   Si Condition Alors :
2       I ;
```

Dans le cas défavorable, où l'expression conditionnelle revoie `True`, le bloc d'instruction I est exécuté. On a alors :

$$T(n) = Tcondition(n) + T_I(n)$$

- T(n) représente le nombre total d'instructions
- Tcondition(n) représente le nombre d'instructions nécessaire pour tester la condition (qui peut-eˆtre 1 s'il s'agit par exemple d'une simple comparaison entre deux expressions arithmétiques).
- TI représente le nombre d'instructions dans I.

5.4.2 Conditionnel avec alternative

```
Si Condition Alors :
    I1 ;
Sinon :
    I2;
```

Dans le cas défavorable, on comptera :

$$T(n) = Tcondition(n) + max(T_{I_1}(n), T_{I_2}(n))$$

Partie 6

Principales classes de complexité

Ces complexités sont classées par temps d'execution croissant de l'agorithme correspondant.

complexité	classe
Θ(1)	temps constant
Θ(log n)	logarithmique en base 2 : log2(n)
Θ(n)	linéaire
Θ(n*log n)	quasi linéaire
Θ(n2)	quadratique, polynômial
Θ(n3)	cubique, polynômial
Θ(2n)	exponentiel (problème très difficiles)

Partie 7

Méthodes employée pour les resolutions d'exercices

7.1 somme d'une suite arithmetique

Soit la somme S :

$$S = \Sigma_{i=1}^{n} i$$

alors on démontre que :

$$S = \frac{n \times (n+1)}{2}$$

7.2 fonction log

Parfois, on a à evaluer le nombre de divisions par 2 qu'il faut appliquer à un nombre N pour amener sa valeur à 0. Cette nombre de division est directement egal à la fonction $log(N)$.

Par exemple : dans la recherche dichotomique.

Partie 8

Application à l'algorithme de recherche

8.1 Recherche linéaire

```
def recherche(X,L):
    j = 0
    n = len(L)
    while j<n and X!=L[j]:
        j += 1
    if j==n : return -1
    return j
```

D'après les règles énoncées : * Le variant de boucle, c'est j * L'estimation se fait dans le pire des cas : j doit atteindre 0. * Il n'y a qu'une seule instruction significative à prendre en considération : la comparaison de X avec les éléments de liste `X!=L[j]`.

8.2 Recherche dichotomique

- La dimension des données sera prise comme egale à `len(mots)`. Appelons cette valeur n.
- Le variant de boucle, c'est $droite - gauche$, qui vaut au départ n, et 0 à la fin de la boucle, si la valeur n'a pas été trouvée (pire des cas).
- Les instructions essentielles de la boucle, ce sont les comparaisons
  - `mots[milieu] == X`
  - et `mots[milieu] > X`

Pour simplifier le raisonnement, disons qu'il n'y a qu'une seule instruction essentielle par itération.

A la fin de la première itération, le variant de boucle vaut n/2.

On peut alors exprimer le nombre d'opérations T(n) pour cet algorithme comme égal à :

$$T(n) = 1 + T(n/2)$$

On aura, avec le même raisonnement,

$$T(n/2) = 1 + T(n/4)$$

Et ainsi de suite jusqu'à ce que n//2, et le variant de boucle, soient egaux à 0.

On a alors $T(n) = 1 + 1 + ...$ un nombre de fois égal au nombre de divisions par 2 de n, nécessaires pour amener n à 0. Cette valeur est egale à $log_2(n)$.

La complexité est alors **O(log(n))**.