AP06 ALGORITHMES DE RECHERCHE

I. Algorithmes de recherche naïfs

Exercice 1 : Recherche d'un élément dans une liste

Ecrire une fonction `recherche_naive(elt, liste)` qui renvoie `True` si l'élément `elt` est présent dans la liste `liste`, et `False` sinon.

Code Python

```
def recherche_naive(elt, liste):
    ''' Renvoie True si elt est dans liste et False sinon'''









    #
assert recherche_naive(3, [1, 2, 3, 4, 5]) == True
assert recherche_naive(6, [1, 2, 3, 4, 5]) == False
```

Définition 1 : Notion de complexité d'un algorithme

La **complexité** d'un algorithme est une mesure de la quantité de ressources (temps, mémoire, etc.) que l'algorithme utilise en fonction de la taille de l'entrée. La complexité permet de comparer les algorithmes entre eux et de choisir le plus efficace pour résoudre un problème donné.
Nous allons nous intéresser à la complexité temporelle, (temps d'exécution de l'algorithme en fonction de la taille de l'entrée) exprimée en utilisant la notation **O** (grand O).
Pour étudier la complexité d'un algorithme on se place toujours dans le **pire des cas**, c'est à dire que l'on suppose que l'algorithme doit parcourir tous les éléments de l'entrée pour trouver la solution.

Complexité de la recherche naïve

- Dans le pire des cas, l'algorithme de recherche naïve doit parcourir tous les éléments de la liste pour vérifier si l'élément recherché est présent ou non. (Elément absent ou présent à la fin de la liste)
- Si la liste possède n éléments, alors l'algorithme de recherche naïve doit effectuer n comparaisons. On dira que la complexité est de l'ordre de O(n).
- Si la liste contient 10 éléments, l'algorithme doit effectuer 10 comparaisons dans le pire des cas.
- De manière générale, si la taille de la liste double, le temps d'exécution de l'algorithme de recherche naïve double également. La complexité de l'algorithme de recherche naïve est linéaire.

Définition 2 : Notion de terminaison d'un algorithme

Un algorithme est dit **terminant** s'il s'arrête après un nombre fini d'étapes pour toute entrée donnée.
Un algorithme se termine de façon certaine si :

- Il ne contient pas de boucle infinie.
- S'il fait appel à une boucle infinie (WHILE), il doit y avoir une condition d'arrêt qui est vérifiée à un moment donné.

La condition d'arrêt est appelée **variant de boucle**.

Définition 3 : Le variant de boucle

Un **variant de boucle** est une quantité qui est modifiée à chaque itération d'une boucle et qui permet de garantir la terminaison de l'algorithme.
Pour qu'un variant de boucle soit efficace, il doit être :

- **positif ou nul** pour que la boucle se poursuive ;
- **décroissant** à chaque itération pour que la boucle se rapproche de sa condition d'arrêt.

Exercice 2 : Variant de boucle

Code Python

```python
def recherche_naive(elt, liste):
    ''' Renvoie True si elt est dans liste et False sinon'''
    i = 0
    while i < len(liste):
        if liste[i] == elt:
            return True
        i += 1
    return False

assert recherche_naive(3, [1, 2, 3, 4, 5]) == True
assert recherche_naive(6, [1, 2, 3, 4, 5]) == False
```

1. Identifier le variant de boucle de l'algorithme de recherche naïve.
2. Montrer que ce variant de boucle est positif ou nul au début de la boucle, et qu'il est décroissant à chaque itération de la boucle.

Définition 4 : Notion de correction d'un algorithme

Un algorithme est dit **correct** s'il renvoie la bonne solution pour toute entrée donnée.
Pour prouver la correction d'un algorithme, on peut utiliser l'invariant de boucle. Un **invariant de boucle** est une propriété qui est vérifiée avant et après chaque itération d'une boucle. Si l'invariant de boucle est vérifié, alors l'algorithme est correct.

Exercice 3 : Correction d'un algorithme

Code Python

```python
def recherche_naive(elt, liste):
    ''' Renvoie True si elt est dans liste et False sinon'''
    i = 0
    while i < len(liste):
        if liste[i] == elt:
            return True
        i += 1
    return False

assert recherche_naive(3, [1, 2, 3, 4, 5]) == True
assert recherche_naive(6, [1, 2, 3, 4, 5]) == False
```

1. Identifier le variant de boucle de l'algorithme de recherche naïve.
2. Montrer que ce variant de boucle est positif ou nul au début de la boucle, et qu'il est décroissant à chaque itération de la boucle. Conclure.
3. Identifier un invariant de boucle de l'algorithme de recherche naïve.
4. Montrer que cet invariant de boucle est vérifié avant et après chaque itération de la boucle. Conclure.

Exercice 4 : division euclidienne

Le principe de l'algorithme de division euclidienne est de soustraire le diviseur du dividende jusqu'à ce que le reste soit inférieur au diviseur. Le nombre de soustractions effectuées correspond au quotient de la division euclidienne, et le reste correspond au reste de la division euclidienne.
Si `a` est le dividende et `b` est le diviseur, alors l'algorithme de division euclidienne peut être écrit de la manière suivante :

1. Initialiser le quotient `q` à 0 et le reste `r` à `a`.
2. Tant que le reste `r` est supérieur ou égal au diviseur `b`, faire :
   - Soustraire le diviseur `b` du reste `r`.
   - Incrémenter le quotient `q` de 1.
3. Retourner le quotient `q` et le reste `r`.

Code Python

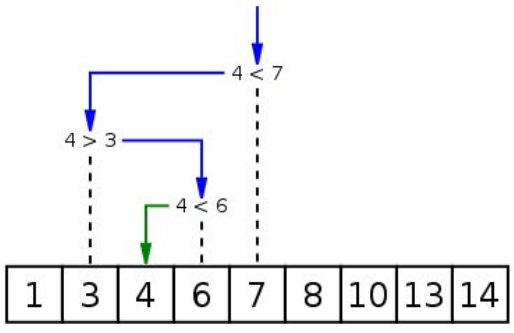
```
def division_euclidienne(a, b):
    ''' Renvoie quotient et reste de la division euclidienne de a par b '''




assert division_euclidienne(10, 3) == (3, 1)
assert division_euclidienne(10, 4) == (2, 2)
```

1. Identifier le variant de boucle de l'algorithme de division euclidienne.
2. Montrer que ce variant de boucle est positif ou nul au début de la boucle, et qu'il est décroissant à chaque itération de la boucle. Conclure.
3. Identifier un invariant de boucle de l'algorithme de division euclidienne.
4. Montrer que cet invariant de boucle est vérifié avant et après chaque itération de la boucle.
5. Conclure que l'algorithme de division euclidienne est correct.

II. Algorithme de recherche dans une liste triée

Définition 1 : Diviser pour régner

En informatique, la technique de **diviser pour régner** (*divide and conquer*) est une stratégie algorithmique qui consiste à diviser un problème en sous-problèmes plus petits, à résoudre ces sous-problèmes de manière récursive, puis à combiner les solutions pour obtenir la solution du problème initial.



- on compare la valeur du milieu avec la valeur cherchée. $4 < 7$ donc :
- on conserve la moitié gauche du tableau : `[1,3,4]`
- on compare la valeur du milieu avec la valeur cherchée. $4 > 3$ donc :
- on conserve la moitié droite du tableau : `[4]`
- on compare la valeur du milieu avec la valeur cherchée. La valeur est trouvée.

Exercice 5 : Algorithme de recherche dans une liste triée

```python
def recherche_dichotomique(val, L):
    i_deb = 0
    i_fin = len(L) - 1
    while ........................:
        i_cen = ...................
        if ........................:
            ........................
        elif ......................:
            ........................
        else:
            ........................
    return False
```

i_deb	i_fin	while	i_cen	IF #6	IF #8

1. Compléter l'algorithme de recherche dichotomique.
2. Compléter la trace de l'algorithme de recherche dichotomique pour la valeur 4 et la liste `[1,3,4,6,7,8,10,13,14]`.
3. Identifier le variant de boucle de l'algorithme de recherche dichotomique.
4. Montrer que ce variant de boucle est positif ou nul au début de la boucle, et qu'il est décroissant à chaque itération de la boucle. Conclure.
5. Identifier un invariant de boucle de l'algorithme de recherche dichotomique.
6. Montrer que cet invariant de boucle est vérifié avant et après chaque itération de la boucle. Conclure que l'algorithme de recherche dichotomique est correct.
7. Comparer les complexités des deux algorithmes de recherche.

Définition 2 : Le logarithme

Le **logarithme** est un opérateur mathématique qui a entre autre comme propriété :

$$\log_2(2^k) = k \times \log_2(2) = k \qquad \text{et } \log_2(2) = 1$$

Dans notre exemple, lorsque la taille de la liste double, alors le nombre de boucles augmente seulement de 1.

Supposons que la liste comporte n éléments et que n est tel qu'il existe un $k \in \mathbb{N}$ tel que $n = 2^k$.

Etant donné que le nombre de boucles augmente de 1 quand le nombre d'éléments est multiplié par deux alors k est le nombre de boucles nécessaires pour obtenir une liste d'un seul élément.

n	valeur de k	nombre de boucles
1	0	0
2	1	1
4	2	2
8	3	3
16	4	4
...	...	...
n	k	k
2n	k+1	k+1

Pour extraire k en fonction de n, on utilise une fonction nommée `logarithme` et notée `log` :

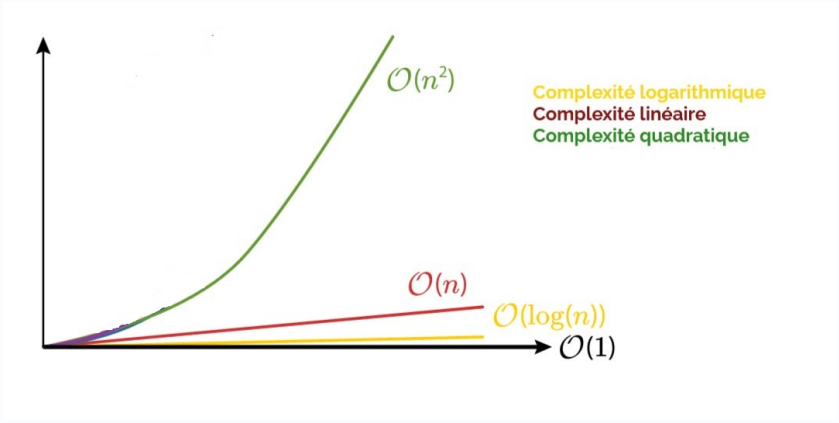
$$n = 2^k \iff \log_2(n) = \log_2(2^k) = k\log_2(2) = k$$

Ainsi $k = \log_2(n)$
k étant le nombre de boucles nécessaires pour obtenir une liste d'un seul élément, alors la complexité est de l'ordre de $O(\log(n))$.

Définition 3 : Complexité logarithmique

Dans l'exemple précédent, on dira que cet algorithme possède une complexité **logarithmique** notée $O(\log(n))$

Définition 4 : Comparaison des complexités



Propriété 1 : Détermination de complexité

Pour déterminer la complexité d'un algorithme, on peut utiliser les propriétés suivantes :

- Si un algorithme n'effectue pas de boucle, alors sa complexité est de l'ordre de $O(1)$.
- Si un algorithme effectue une boucle FOR, alors sa complexité est de l'ordre de $O(n)$.
- Si un algorithme effectue k boucles FOR imbriquées, alors sa complexité est de l'ordre de $O(n^k)$.
- Si un algorithme effectue une boucle WHILE et à chaque itération la taille du problème est divisée par 2, alors sa complexité est de l'ordre de $O(\log(n))$.

III. Algorithme de recherche d'extrémum

Exercice 6 : Recherche d'extrémum

1. Ecrire la fonction `recherche_naive(L)` qui renvoie la valeur max de la liste `L` à l'aide d'une boucle FOR.
2. Déterminer la complexité de cet algorithme.

Exercice 7 : Recherche d'extrémum avec une boucle WHILE

On considère maintenant l'algorithme de recherche de maximum suivant :

Code Python

```
def recherche_max(L):
    ''' Renvoie la valeur max de la liste '''
    maxi = L[-1]
    while len(L) > 0:
        valeur = L.pop()
        if valeur > maxi:
            maxi = valeur
    return maxi
```

1. Effectuer la trace de cet algorithme pour la liste `[4, 8, 7, 9, 4]`
2. Identifier le variant de boucle de cet algorithme .....
3. Montrer que ce variant de boucle est positif ou nul au début de la boucle, et qu'il est décroissant à chaque itération de la boucle. Conclure.
4. Identifier un invariant de boucle de cet algorithme .....
5. Montrer que cet invariant de boucle est vérifié avant et après chaque itération de la boucle. Conclure.
6. Quelle est la complexité de cet algorithme ? .....