1 Recherche séquentielle

July 26, 2024

[1]:
```
def search(li:list, val)->int:
    """
    Retourne l'indice de val dans li si val est dans li, et None sinon.
    """
    for i in range(len(li)):
        if li[i] == val:
            return i
    return None
```

[2]:
```
print(search([3, 2, 6, 1, 7], 1))
```

```
3
```

[3]:
```
print(search([3, 2, 6, 1, 7], 4))
```

```
None
```

[4]:
```
def search2(li:list, val)->[int]:
    """
    Retourne la liste des indices de toutes les occurrences de val dans li.
    """
    ind = []
    for i in range(len(li)):
        if li[i] == val:
            ind.append(i)
    return ind
```

[5]:
```
print(search([3, 2, 3, 3, 7], 3))
```

```
0
```

La fonction `search2` à l'avantage d'être cohérente au niveau du type de la valeur retournée : quoi qu'il arrive, on retourne le type `list`, et une liste vide code donc l'absence de `val` dans `li`.

En revanche elle nécessite de parcourir toute la liste à chaque fois, là où la première fonction search s'arrêtait dès que `val` était trouvée.

La fonction search2 parcourt toute la liste li, qui contient n élément. Son coût est donc une fonction affine de n : $C(n) = an + b$. La fonction search parcourt toute la liste uniquement dans le pire cas possible (où `val` est à la fin de `li`, mais en moyenne elle coûte $\frac{n}{2}$, donc son coût est également affine).

Remarque : le facteur constant a correspond aux instructions qu'on répète n fois (celles qui sont à l'intérieur de la boucle, comme le test d'égalité par exemple), tandis que le facteur constant b correspond aux instructions qu'on n'exécute qu'une fois (comme l'instruction return par exemple).

[6]:
```
def maxi(li:[float])->float:
    """
    Retourne la maximum de la liste de nombres li.
    """
    M = li[0]
    for i in range(len(li)):
        if li[i] > M:
            M = li[i]
    return M
```

[7]:
```
def maxi(li:[float])->float:
    """
    Retourne la maximum de la liste de nombres li.
    """
    M = li[0]
    for el in li:
        if el > M:
            M = el
    return M
```

[8]:
```
print(maxi([3, 2, 6, 1, 7]))
```

```
7
```

[9]:
```
def mini(li:[float])->float:
    """
    Retourne la minimum de la liste de nombres li.
    """
    m = li[0]
    for el in li:
        if el < m:
            m = el
    return m
```

[10]:
```
print(mini([3, 2, 6, 1, 7]))
```

```
1
```

Les fonctions `maxi` et `mini` parcourent toute la liste `li`, leur coût est donc une fonction affine de n.

[11]:
```
def imax(li:list)->int:
    """
    Retourne l'indice du maximum de li.
    """
    iM = 0
```

```
    M = li[0]
    for i in range(len(li)):
        if li[i] > M:
            M = li[i]
            iM = i
    return iM
```

[12]:
```
def imax(li:list)->int:
    """
    Retourne l'indice du maximum de li.
    """
    iM = 0
    for i in range(len(li)):
        if li[i] > li[iM]:
            iM = i
    return iM
```

[13]:
```
print(imax([3, 2, 6, 7, 1]))
```

```
3
```

Même chose pour `imin` en écrivant `<` à la place de `>` pour le test de comparaison entre `li[i]` et `M`.

Les deux fonctions précédentes ont un coût affine en n (un parcours entier de la liste).

[14]:
```
def imax(li:list)->int:
    """
    Retourne l'indice du maximum de li.
    """
    return search(li, maxi(li))
```

[15]:
```
print(imax([3, 2, 6, 7, 1]))
```

```
3
```

Cette dernière version de `imax` a aussi un coût affine en n, mais elle nécessite deux parcours de la `li` (un pour trouver la valeur du max, et un pour retourner son indice).

[16]:
```
def iMax(li:[float])->(int, float):
    """
    Retourne l'indice du maximum de li et sa valeur.
    """
    iM = 0
    M = li[0]
    for i in range(len(li)):
        if li[i] > M:
            M = li[i]
            iM = i
    return iM, M
```

[18]:
```
print(iMax([3, 2, 6, 7, 1]))
```

```
(3, 7)
```

La fonction `iMax` a également un coût affine en n (un parcours complet).

[19]:
```
def count(li:list)->dict:
    """
    Retourne le dictionnaire qui associe à chaque valeur de li son
    nombre d'occurences dans li.
    """
    d = {}
    for el in li:
        if el in d:
            d[el] += 1
        else:
            d[el] = 1
    return d
```

[20]:
```
print(count([2, 1, 3, 1, 1, 2, 3, 1]))
```

```
{2: 2, 1: 4, 3: 2}
```

Le mot-clé `in` est équivalent à l'emploi d'une fonction de recherche séquentielle :

[21]:
```
def dans(li:list, val):
    """
    Retourne True si val est dans li, et False sinon.
    """
    for el in li:
        if el == val:
            return True
    return False
```

[22]:
```
def count(li:list)->dict:
    """
    Retourne le dictionnaire qui associe à chaque valeur de li son
    nombre d'occurences dans li.
    """
    d = {}
    for el in li:
        if dans(d, el):
            d[el] += 1
        else:
            d[el] = 1
    return d
```

[23]:
```
print(count([2, 1, 3, 1, 1, 2, 3, 1]))
```

```
{2: 2, 1: 4, 3: 2}
```

[26]:
```
def maxs(li:[float])->(float, float):
    """
    Retourne le premier et le second maximum de li.
    """
    iM1, M1 = iMax(li)
    li2 = li[:iM1] + li[iM1+1:]
    M2 = maxi(li2)
    return M1, M2
```

[27]:
```
print(maxs([3, 2, 5, 7, 1, 9]))
```

```
(9, 7)
```

[30]:
```
def maxs(li:[float])->(float, float):
    """
    Retourne le premier et le second maximum de li.
    """
    if li[0] > li[1]:
        M1, M2 = li[0], li[1]
    else:
        M1, M2 = li[1], li[0]

    for i in range(len(li)):
        if li[i] > M1:
            M2 = M1
            M1 = li[i]
        elif li[i] > M2:
            M2 = li[i]

    return M1, M2
```

[31]:
```
print(maxs([3, 2, 5, 7, 1, 9]))
```

```
(9, 7)
```

[ ]:
```

```