Code de partage avec Capytale : 06c5-1633205

Exercice 1 - optimiser la recherche de valeurs dans une liste triée

On commencera par taper les commandes `L=[rd.randint(1,201) for k in range(0,100)]` puis `L.sort()`

1. Que font ces deux commandes ?

   La première crée une liste de 100 nombres entiers compris entre 0 et 200, et la deuxième trie la liste par ordre croissant.

2. Ecrire une fonction qui prend en entrée un entier n et qui renvoie 0 si n n'est pas dans la liste et un indice auquel il se trouve sinon.

   En première approche, on peut simplement chercher à examiner tous les nombres de la liste en testant s'ils sont égaux au nombre cherché (avec `L` définie au préalable).

```
def cherche_valeur(L,n) :
    rang=-1
    for i in range(0,100)
    :
        if L[i]==n:
            rang=i
    return rang
```

   On peut optimiser en utilisant une boucle `while` pour s'arrêter dès que la valeur cherchée a été trouvée.

```
def cherche_valeur2(L,n) :
    i=0
    indice =-1 #réponse par défaut
    while i<99 and L[i]!=n :
        i=i+1
    if L[i]==n:
        indice =i
    return indice
```

3. Proposer un algorithme dichotomique qui donne le même résultat.

   C'est le même principe que la dichotomie avec les fonctions (pour rechercher le zéro d'une fonction). A chaque étape, on divise par 2 la taille de notre « cible » et donc ici au bout de 6 étapes au plus, on aura divisé par 64 et il ne reste plus que 2 entiers à tester.
   On continue à diviser l'intervalle tant qu'il est plus grand que 1 strictement, donc à la fin de la boucle `min` et `max` contiennent deux entiers consécutifs qu'il faut tester.

```
def dicho(L,n) :
    min=0
    max=100
    while max-min>1:
        m=int(np.floor((max+min)/2)) #
    floor pour avoir un nombre entier puis
    int car floor renvoie un réel
        if L[m]>n:
            max=m-1
        else :
            min=m
    if L[min]==n :
        return min
    elif L[max]==n :
        return max
    else :
        return -1
```

4. Comparer la vitesse des deux méthodes en les répétant un grand nombre de fois.

   Avec la boucle `for`, on réalise à chaque fois n opérations (où n est la taille de la liste, ici 100). Avec la boucle `while`, on passe en moyenne à $\frac{n}{2}$ opérations quand la valeur est trouvée (ici 50) et avec la dichotomie, on descend à $\ln(n)$ (ici une division par 10). Mais avec une liste d'un million d'éléments, le gain d'efficacité est plus spectaculaire, puisqu'on réalise avec l'algorithme dichomotique moins de 20 opérations (comparé à un million pour le premier algorithme).

Exercice 2 - tri d'une liste

1. Retour sur le TP 9 : calculer la « vitesse » du tri par « sélection » (sélection du minimum de manière itérative).

   Pour une liste de taille n la vitesse de ce tri est de l'ordre de $\frac{n(n+1)}{2}$
   en effet, on parcourt au début une liste de taille n pour trouver le minimum d'où n « opérations » puis on recommence avec une liste de taille $n-1$...

2. Proposer une autre méthode de tri (indication : on parle de « tri par insertion ».

   C'est la méthode que l'on utilise naturellement pour trier les cartes que l'on a dans la main : on prend les cartes une par une, par exemple en partant de la gauche et on les insère à leur place, c'est-à-dire qu'on s'arrete dès que l'on « bute » sur une carte plus petite dans le tas trié (si on range par ordre croissant).
   C'est donc ce que l'on fait ici en partant de la deuxième (celle d'indice 1 donc, car on considère que la première est déjà triée). Et on est obligé d'appliquer l'insertion à toutes les cartes à partir de la deuxième (d'où la boucle `for`)

```
for i in range (1,len(L)):
  j=i # on crée l'indice "glissant"
    while L[j]<L[j-1] and j>0:
        a=L[j-1]
        L[j-1]=L[j]
        L[j]=a
        j=j-1 # on met à jour l'indice où est notre nombre ou "carte"
print(L)
```

   ce programme fait donc la même chose que `sort()` ou le tri par recherche de minimum, mais nous n'avons pas amélioré l'efficacité car dans le « pire des cas » (une liste triée par ordre décroissant par exemple), le programme fait 1 puis 2, puis ..., puis $n-1$ opérations et on retrouve donc le même ordre de grandeur $\frac{(n-1)n}{2}$