Avez-vous déjà cherché un mot dans un dictionnaire ou cherché votre nom dans une liste alphabétique ? Quelle est la meilleure méthode pour faire ce type de recherche ?

On considérera dans les exemples ci-dessous que l'on **recherche une valeur *N* dans une liste d'entiers triés dans l'ordre croissant**, mais les méthodes décrites peuvent être appliquées à tout type de données ordonnées.

Partie I : Méthode séquentielle (par comparaisons)

La méthode séquentielle consiste à comparer successivement le nombre *N* recherché à chaque valeur du une liste, jusqu'à ce qu'il soit trouvé, afin de renvoyer l'indice (donc la position) de *N* dans le une liste. Cette méthode est utilisable pour des listes ordonnées, mais aussi pour des listes non-ordonnées.

On considère, par exemple, la liste d'entiers suivante :

1	2	3	7	9	11	13	15	17

I.1. Décrire la suite d'étapes dans la recherche du nombre *N* = 13, puis dans le cas de *N* = 18 en utilisant les termes « élément » et « indice ».

I.2. Dans le pire des cas (que vous définirez), combien d'étapes sont nécessaires pour trouver une valeur dans une liste de longueur 60 000 ?

I.3. Comment peut-on alors qualifier la dépendance de la complexité (et donc du temps d'exécution) de cet algorithme par rapport à la taille de la liste ?

I.4. Conclure sur l'inconvénient principal de la méthode séquentielle.

Partie II : Méthode dichotomique

Lorsque l'on cherche à retrouver un élément dans une liste ordonnée, le choix d'un algorithme efficace devient une nécessité, et l'algorithme de recherche séquentielle ne l'est pas forcément. Y a-t-il une meilleure méthode ?

A. Jeu du « plus petit/plus grand »

Avec un partenaire, faite une partie du jeu « plus petit/plus grand » :
L'un de vous (le maitre du jeu) pense à un nombre entier entre 0 et 100, que l'autre doit le deviner en faisant des propositions. Le maitre du jeu répond par « plus petit » ou « plus grand », jusqu'à ce que le nombre soit trouvé.
Pendant le jeu, réfléchissez à la méthode la plus efficace pour trouver le nombre, c'est-à-dire en faisant le moins de proposition possible

Imaginez maintenant que le joueur ne veut plus jouer (rabat joie !). Le maitre du jeu décide alors de faire jouer son ordinateur (c'est-à-dire que c'est l'ordinateur qui va chercher le nombre auquel pense le maitre du jeu).

II.1. Proposer une ébauche de pseudo-code (juste les grandes lignes, la méthode) qu'il faudrait fournir à l'ordinateur pour qu'il puisse jouer au jeu.

Appeler le professeur pour proposer et valider votre méthode

B. La dichotomie

Au cours d'une recherche par dichotomie (« tomia », couper, « dikha », en deux), on compare l'élément recherché à l'élément médian (au milieu) de la liste. Soit on tombe sur le résultat, soit on divise par deux la taille de l'ensemble en déterminant si l'élément recherché est avant ou après l'élément médian.

II.2. On considère la liste d'entiers vue dans la partie I. Décrire la suite d'étapes dans la recherche du nombre *N* = 13, puis dans le cas de *N* = 18 en utilisant les termes « élément » et « indice ».

II.3. En vous aidant de la description donnée en introduction, compléter le pseudo-code de l'algorithme de la recherche par dichotomie ci-dessous.

Algorithme de recherche par dichotomie

```
Entrée : N un entier, L une liste d'entiers
i = 0
j = longueur de L
l'intervalle de recherche est [i,j-1]
TANT QUE i ..... à j
        k = indice médian de l'intervalle de recherche
        SI l'élément médian .....
                on sort de la boucle en faisant .....
        SINON SI N est avant l'élément médian
                j = .....
        SINON
                i = .....
        FIN SI
FIN TANT QUE
SI l'élément restant est N
        RENVOYER k
FIN SI
```

II.4. Déterminer l'expression de l'indice *k* en fonction de *i* et *j*.

II.5. Montrer que j-i est un variant de boucle. Que peut-on en conclure sur la boucle de l'algorithme de dichotomie ?

II.6. Dans le pire des cas (défini précédemment), combien d'étapes sont nécessaires pour trouver une valeur dans une liste de longueur 60 000 ?

II.7. Comparer la complexité de cet algorithme par rapport à l'algorithme de recherche séquentielle et conclure.

Partie III : A vos ordinateurs !

A. Méthode séquentielle

III.1. Écrire une fonction de recherche séquentielle, nommée *Comparaison*, qui prend en arguments l'élément *N* à chercher et la liste ordonnée *Liste* de taille *Taille*, et qui renvoie le rang de *N* (si existant).
Si *N* n'existe pas dans la liste, renvoyer la valeur -*Taille* pour rang (utile pour les fonctions suivantes).
Utiliser une boucle WHILE pour le code (pour faciliter l'étude de la complexité).

III.2. Modifier la fonction *Comparaison* pour qu'elle renvoie aussi la complexité (le nombre d'affectation et comparaisons ayant été nécessaire) de la recherche.

Appeler le professeur pour vérifier votre fonction

Nous allons maintenant étudier la complexité de cet algorithme de recherche, à l'aide du code.

III.3. Ecrire une fonction *ComplexComp* prenant en argument la taille *Taille* de la liste. Cette fonction doit :

- Générer par compréhension une liste de taille *Taille* des entiers naturels
- Pour 100 occurrences (à faire 100 fois) :
  - Générer aléatoirement dans le domaine [0, 2**Taille*] un entier *N* à chercher

    ```
    from random import randint
    N = randint(0,2*Taille)
    ```
  - Chercher *N* dans *Liste* par la méthode séquentielle pour obtenir le nombre d'étapes nécessaires pour la recherche.
- Renvoyer à la fin le nombre moyen d'étapes pour 100 occurrences (donc la complexité moyenne de l'algorithme pour une taille de liste donnée).

III.4. Appeler la fonction GraphComp fournis pour obtenir le graphique de la complexité de l'algorithme en fonction de la taille. Confirmer l'hypothèse que la complexité de l'algorithme de recherche séquentielle est linéaire.

B. La dichotomie

III.5. A partir du pseudo-code précédent, écrire une fonction Python de recherche par dichotomie, nommée *Dichotomie*, qui prend en arguments l'élément *N* à chercher et la liste ordonnée *Liste* de taille *Taille*, et qui renvoie le rang de *N* (si existant). Si *N* n'existe pas dans la liste, renvoyer -*Taille* en rang.
Utiliser une boucle WHILE pour le code (pour faciliter l'étude de la complexité).

III.6. Modifier la fonction *Dichotomie* qu'elle renvoie aussi la complexité (le nombre d'affectation et comparaisons ayant été nécessaire) de la recherche.
Attention : trouver l'élément médian d'une liste constitue aussi une étape

Nous allons maintenant étudier la complexité de cet algorithme de recherche, à l'aide du code.

III.7. Ecrire une fonction *ComplexDicho* prenant en argument la taille *Taille* de la liste. Cette fonction doit :

- Générer par compréhension une liste de taille *Taille* des entiers naturels
- Pour 100 occurrences (à faire 100 fois) :
  - Générer aléatoirement dans le domaine [0, 2**Taille*] un entier *N* à chercher

```
from random import randint
N = randint(0,2*Taille)
```

  - Chercher *N* dans *Liste* par la méthode dichotomique pour obtenir le nombre d'étapes nécessaires pour la recherche.
- Renvoyer à la fin le nombre moyen d'étapes pour 100 occurrences (donc la complexité moyenne de l'algorithme pour une taille de liste donnée).

III.8. Appeler la fonction GraphDicho fournis le pour obtenir le graphique de la complexité de l'algorithme en fonction de la taille. Montrer que la complexité de l'algorithme de recherche par dichotomie est plus efficace que l'algorithme de recherche séquentielle.

Pour aller un peu plus loin…

Etude de la trichotomie

Dans la trichotomie, au lieu de diviser la liste en deux dans la recherche de *N*, on la divise en 3 et on détermine dans quel tiers se trouve *N*.
Proposer un pseudo-code pour cette méthode de recherche.
Faire l'étude de la complexité de cet algorithme (le nombre d'étape) pour une liste de longueur 60 000 et comparer avec la méthode dichotomique.

Ce qu'il faut retenir

❖ Recherche séquentielle

La méthode séquentielle consiste à comparer successivement le nombre *N* recherché à chaque valeur de la liste, jusqu'à ce qu'il soit trouvé, afin de renvoyer l'indice (donc la position) de *N* dans la liste.

Algorithme de recherche séquentielle

```
Entrées : Liste, une liste de donnée de taille Taille ; N, un entier
Compteur = 0
TANT QUE Compteur ne dépasse pas la taille de Liste et élément de Liste n'est pas N
    Ajouter 1 à Compteur
FIN TANT QUE
SI Compteur est inférieur à la taille de Liste
    RENVOYER Rang
```

La complexité de l'algorithme de recherche séquentielle est linéaire ($\propto$ *Taille*)

❖ Recherche par dichotomie

Dans la méthode par dichotomie, on compare l'élément recherché à l'élément médian (au milieu) de la liste. Soit on tombe sur le résultat, soit on divise par deux la taille de l'ensemble en déterminant si l'élément recherché est avant ou après l'élément médian.

Algorithme de recherche par dichotomie

```
Entrée : N un entier, L une liste d'entiers
i = 0
j = longueur de L
l'intervalle de recherche est [i,j-1]
TANT QUE i inférieur à j
    k = indice médian de l'intervalle de recherche
    SI l'élément médian est N
        on sort de la boucle en faisant i = k et j = k
    SINON SI N est avant l'élément médian
        j = k
    SINON
        i = k+1
    FIN SI
FIN TANT QUE
SI l'élément restant est N
    RENVOYER k
FIN SI
```

L'algorithme de recherche par dichotomie est bien plus efficace que l'algorithme de recherche séquentielle : sa complexité est logarithmique ($\propto \log(Taille) < Taille$)