1. Principe

La recherche dichotomique utilise un ensemble déjà trié avec un ordre défini pour diviser l'étendue de recherche à chaque fois par deux à chaque étape.

Exemple : trouver un nom dans un annuaire téléphonique qui consiste en une liste triée alphabétiquement. On peut s'y prendre de plusieurs façons différentes. En voici deux :

- **Recherche linéaire** : parcourir les pages dans l'ordre (alphabétique) jusqu'à trouver le nom cherché.
- **Recherche dichotomique** : ouvrir l'annuaire au milieu, si le nom qui s'y trouve est plus loin alphabétiquement que le nom cherché, regarder avant, sinon, regarder après. Refaire l'opération qui consiste à couper les demi annuaires (puis les quarts d'annuaires, puis les huitièmes d'annuaires, etc.) jusqu'à trouver le nom cherché.

Typiquement en python on applique la recherche dichotomique sur une liste triée.

a. Algorithme général de recherche dichotomique :

Liste triée L
Valeur recherchée val
Tant que intervalle de recherche >= 1 **alors**
    Si valeur associée à l'indice milieu est égale à la valeur recherchée **alors** retourner indice
    Sinon
        Si valeur recherchée inférieure à la valeur associée à l'indice milieu**alors**
            Changer intervalle de recherche pour l'intervalle entre le début de l'intervalle précédent et la valeur de l'indice -1
        Sinon
            Changer intervalle de recherche pour l'intervalle entre la fin de l'intervalle précédent et la valeur de l'indice + 1
Fin de tant que
retourner Vide ou – 1

b. Une traduction de cet algorithme peut être :

2. Notions de complexité

a. Définition

Pour mesurer la performance d'un algorithme on définit un calcul de sa complexité. Puis on la compare avec celles d'autres algorithme qui effectuent la même chose. Il existe deux types de complexité :

complexité spatiale : permet de quantifier l'utilisation de
complexité temporelle: permet de quantifier la vitesse

Pour la complexité temporelle qui compare différents algorithmes, les règles de ce calcul doivent être indépendantes :

- du langage de programmation utilisé ;
- du processeur de l'ordinateur sur lequel sera exécuté le code ;
- de l'éventuel compilateur ou interpréteur employé.

Par soucis de simplicité, on fera l'hypothèse que toutes les opérations élémentaires sont à égalité de coût, soit 1 « unité » de temps.

Exemple : a = b * 3 : 1 multiplication + 1 affectation = 2 « unités »

La complexité en temps d'un algorithme sera exprimée par une fonction, notée T (pour Time), qui dépend :

- du nombre n de données à traiter.
- de la donnée en elle-même, de la façon dont sont réparties les différentes valeurs qui la constituent.

On distingue deux formes de complexité en temps :

- la complexité dans le meilleur des cas : c'est la situation la plus favorable,
  par exemple : recherche d'un élément situé à la première position d'une liste
- la complexité dans le pire des cas : c'est la situation la plus défavorable,
  par exemple : recherche d'un élément dans une liste alors qu'il n'y figure pas

On calculera le plus souvent la complexité dans le pire des cas, car elle est la plus pertinente. Il vaut mieux en effet toujours envisager le pire.

b. Ordre de grandeur

Pour comparer des algorithmes, il n'est pas nécessaire d'utiliser la fonction T, mais seulement l'ordre de grandeur asymptotique, noté O (« grand O »).

$T(n)$ est en $O(f(n))$ s'il existe un seuil n_0 à partir duquel la fonction T est toujours dominée par la fonction f, à une constante multiplicative fixée c près.

Pour les matheux : une fonction $T(n)$ est en $O(f(n))$ (« *en grand O de f(n)*«) si :

$$\exists n_0 \in \mathbb{N}, \exists\, c \in \mathbb{R}, \forall n \in \mathbb{R},\ n \geq n_0 \Rightarrow |T(n)| \leq c \times |f(n)|$$

Autrement dit :

Exemples :

$$T_1(n)=7$$
$$T_2(n)=22n+5$$
$$T_3(n)=2n^2+1n+3$$
$$T_4(n)=2+5\times (n-1)\ 5$$

Comparons pour une liste de taille n dans le pire des cas.

La recherche naïve :

```
def recherche_naive(t, valeur):
    """
    recherche naïve
    """
    compteur = 0
    for i in range(len(t)):
        compteur += 1
        if t[i] == valeur :
            return i
    print(f"compteur naïve : {compteur}")
    return None
```

ici on parcourt les n éléments de la liste et on effectue à chaque fois une comparaison. Dans le pire des cas on aura parcouru les n éléments de la liste. La complexité est donc de n (comparaisons).

Soit une complexité

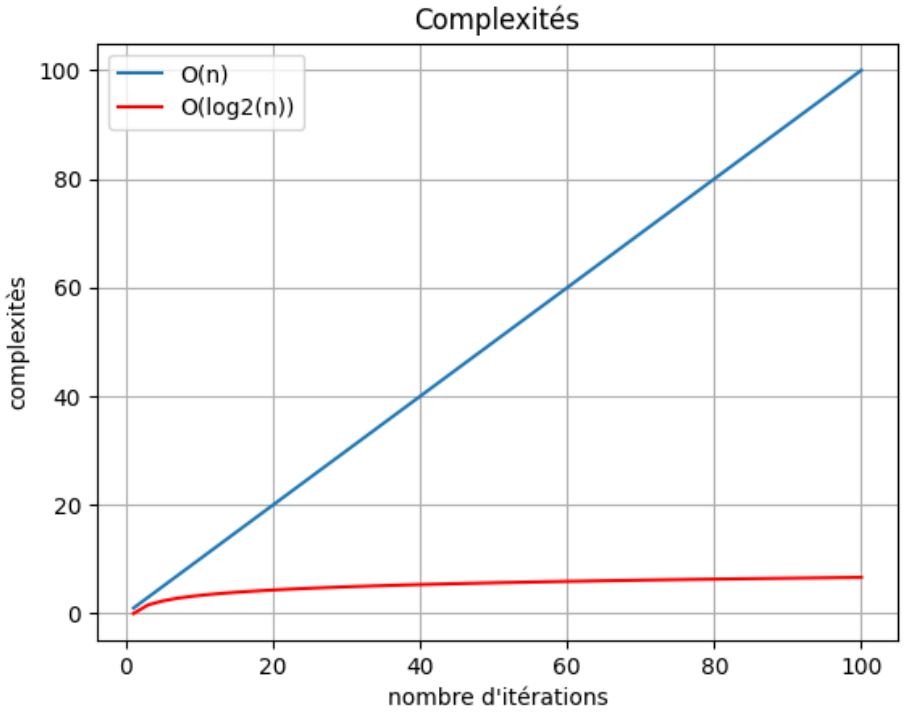
La recherche dichotomique :

A chaque étape on divise la liste en 2. Dans le pire des cas on cherche un entier i tel que :

$\frac{n}{2^i} = 1$ Ce qui revient à obtenir un tableau de taille 1

En transformant cette équation par la fonction logarithmique :

- Soit une complexité en



Exercice Trouver le nombre de comparaisons pour une liste de :

Liste de n éléments	Nombre de comparaisons par calcul	Nombre de comparaisons par logiciel
100		
1000		
10000		

c. Preuve d'un algorithme

Définition : Réaliser la preuve d'un algorithme, c'est :

- Prouver qu'il se termine : On parle de .
- Prouver qu'il fait bien ce que l'on attend de lui : On parle

Terminaison : Il s'agit de vérifier ici que les calculs effectués par l'algorithme s'arrêtent bien. En particulier, lors de l'utilisation d'une boucle conditionnelle par l'algorithme, il est primordial de vérifier que l'on sort bien de cette boucle. On parle aussi de .

Ici **le variant est largeur de l'intervalle soit de la quantité fin - début**

La condition de boucle étant début <= fin , cela correspond exactement à ce que notre variant soit positif ou nul.

Montrons maintenant que le variant décroît strictement lors de l'exécution du corps de la boucle.

On commence par définir milieu = (début + fin) // 2.

En particulier, on a alors début <= milieu<= fin.

Ensuite, trois cas sont possibles.

▷ si liste[milieu] == val, on sort directement de la boucle à l'aide d'un return. La terminaison est assurée.

▷ si liste[milieu] > val, on modifie la valeur de fin. En appelant fin2 cette nouvelle valeur, on a :

fin2 - début < milieu - début <= fin - début car fin2 = milieu - 1 < milieu.

Ainsi, le variant a strictement décru.

▷ sinon, on modifie début et on a de même :

fin -debut2 <fin - milieu <= fin - début

De même, le variant a strictement décru.

Ayant réussi à exhiber un variant pour notre boucle, nous avons prouvé qu'elle termine bien.

Correction : Il s'agît ici de prouver que l'algorithme fait bien ce qu'on lui demande. Pour cela, on va chercher un . C'est une propriété P qui est **vérifiée avant l'entrée dans la boucle, qui est vérifiée à chaque itération de la boucle** et qui amène au **résultat escompté à la sortie de la boucle**.

Ici **l'invariant est** : S val est présente dans liste , c'est nécessairement à un indice compris entre début et fin (inclus).

<u>Cas où val est dans la liste :</u>

On suppose donc qu'en entrée de boucle, Inv est vérifié. Après avoir défini milieu, trois cas sont examinés :

1. si tab[milieu] == val, on sort de la boucle prématurément à l'aide de l'instruction return milieu, donc ce cas ne nous intéresse pas dans le cadre de la preuve d'invariant de boucle.

2. si liste[milieu] > val, et si val est présente dans le tableau, alors comme celui-ci est ordonné de façon croissante, cela implique que val ne peut être présent à l'indice milieu, ni après. On en déduit d'après Invariant que si val se trouve dans liste c'est nécessairement à un indice compris entre deb et milieu - 1. Ainsi, après l'affectation fin = milieu - 1, Inv est encore vérifié.

3. sinon, on a liste[milieu] < val et, de même, cela implique que l'on ne peut trouver val dans le tableau à un indice inférieur ou égal à milieu. Ainsi, en supposant Invariant vrai au départ et en effectuant l'affectation deb = milieu + 1, l'invariant reste vérifié.

Cas où val n'est pas dans la liste

Voyons maintenant comme cela nous permet de prouver la correction de cette fonction lorsque le résultat est None (-1) . Avant d'entrer dans la boucle, on a début= 0 et fin = milieu - 1, et donc si val est présente dans tab, c'est nécessairement à un indice compris entre début et fin. Ainsi, Invariant est vérifié en entrant dans la boucle.

Puisqu'il s'agit d'un invariant de cette boucle, il est encore vérifié en sortant de la boucle. Mais à ce moment, on sait que :

▷ Invariant : si val est présente dans liste à une position pos, alors on a début <= pos<= fin.

▷ Sortie de boucle : la condition de boucle début<= fin n'est plus vérifiée, on a donc fin<début.

En combinant les deux, si val est présente dans tab à une position pos, alors début <= pos <= fin, ce qui implique que début <= fin, qui est incompatible avec début < fin.

Ainsi, en sortie de boucle, val ne peut être présente dans la liste (car aucun indice n'est possible) et le résultat renvoyé None est donc correct.