Dichotomie

Dans le Programme Officiel :

Contenus	Capacités attendues	Commentaires
Recherche dichotomique dans un tableau trié	Montrer la terminaison de la recherche dichotomique à l'aide d'un variant de boucle.	Des assertions peuvent être utilisées. La preuve de la correction peut être présentée par le professeur.

Début de séance : Pour aborder le problème de la recherche efficace d'un élément au sein d'un ensemble, un exemple pertinent peut être la recherche d'un contact dans le moteur de recherche de contacts de Facebook : Il y a environ 2,5 milliards d'utilisateurs, et pourtant la recherche est instantanée. Il y a donc clairement un travail d'optimisation de l'algorithme de recherche. La suite de ce cours ne prétend pas répondre à cette question, mais présentera une méthode simple et efficace de recherche.

Activité d'introduction

Jeu du *"devine un nombre entre 1 et 100"*

Si je choisis un nombre entre 1 et 100, quelle est la stratégie optimale pour deviner ce nombre le plus vite possible ?
(à chaque étape, une indication (trop grand, trop petit) permet d'affiner la proposition suivante)

Réponse attendue : la meilleure stratégie est de *couper en deux* à chaque fois l'intervalle d'étude. On démarre de 50, puis 75 ou 25, etc.

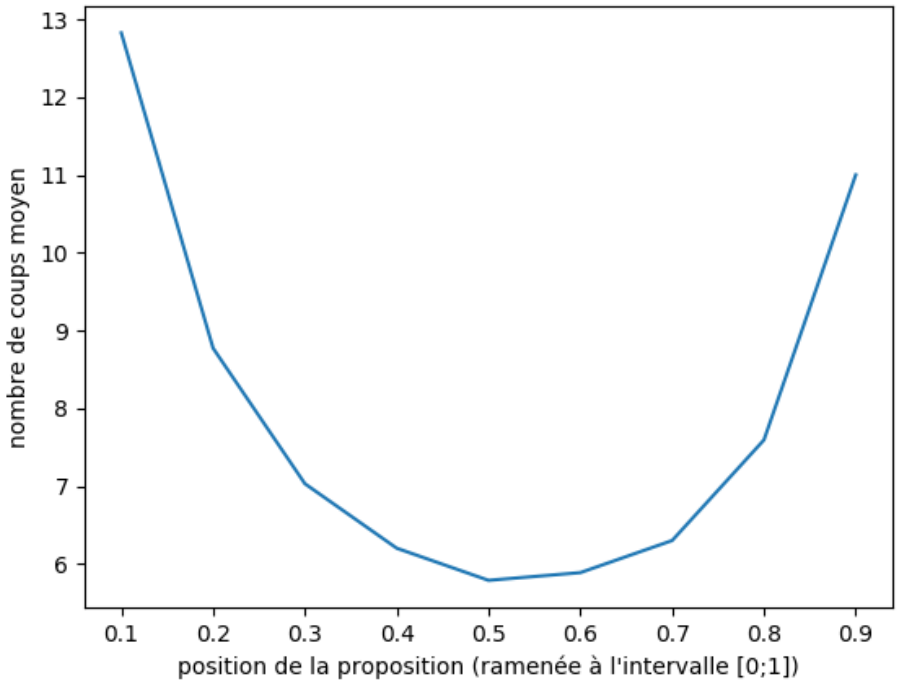
Il convient toute fois de remettre en question (si ce n'est pas fait par les élèves) cette méthode qui paraît *naturellement* optimale : si je propose 90 comme nombre de départ, j'ai certes moins de chance que le nombre soit entre 90 et 100, mais s'il l'est, j'ai gagné un gros avantage car mon nouvel intervalle est très réduit.

On peut alors présenter (ou rappeler, suivant l'avancée des élèves dans le programme de Mathématiques de Première) la notion d'**espérance probabiliste**.

Un exemple signifiant pour les élèves est toujours : "On lance un dé, s'il tombe sur le 6 vous recevez 2 euros, sinon vous me donnez 1 euro. Voulez-vous jouer ?" Leur réponse est non, mais si on passe leur gain de 2 euros à 100 euros, la réponse devient oui. Les élèves dépassent alors la simple observation de la probabilité et arrivent naturellement à la notion d'espérance.

Retour sur le jeu du choix du nombre

Le graphique ci-dessous représente le nombre de coups moyens (sur 10 000 parties simulées)



Interprétations et remarques

- si le choix se porte *toujours* sur le nombre situé à la moitié de l'intervalle (0.5), le nombre de coups moyen avant la victoire (sur 10 000 parties) est environ 6.
- si le choix se porte *toujours* sur le nombre situé à 90 % de l'intervalle (0.9), le nombre de coups moyen avant la victoire (sur 10 000 parties) est environ 11.
- l'asymétrie de la courbe (qui devrait être symétrique) est due aux arrondis par défaut dans le cas de nombres non entiers.

Conclusion générale de l'activité d'introduction

La stratégie optimale est de diviser en deux à chaque étape l'intervalle d'étude. On appelle cela une méthode par **dichotomie**, du grec ancien διχοτομία, dikhotomia (« division en deux parties »)

Algorithmes de recherche d'une valeur dans une liste triée

Préambule : la méthode que nous allons utiliser implique que les valeurs ont été triées auparavant.

Exemple à l'oral, illustrant la nécessité d'avoir une liste ordonnée : "pouvez-vous deviner la couleur que j'ai choisie dans la liste suivante ?"

In [1]:
```
coul = ["bleu", "jaune", "rouge", "vert", "violet", "marron"]
```

hormis le test de toutes les valeurs, aucune méthode efficace n'est possible.

Dans toute la suite, nous rechercherons un élément dans une liste d'entiers **triée** dans l'ordre croissant.
Considérons donc la liste L suivante :

In [12]:
```
L = [2, 3, 6, 7, 11, 14, 18, 19, 24]
```

L'objectif est de définir un algorithme de recherche efficace d'une valeur arbitraire présente dans cette liste.

Méthode 1 : recherche par balayage

C'est la méthode la plus intuitive : on essaie toutes les valeurs (par exemple, dans l'ordre croissant) jusqu'à trouver la bonne.

Exercice 1

Écrire un code permettant d'afficher l'indice de la valeur `14` dans la liste `L = [2, 3, 6, 7, 11, 14, 18, 19, 24]` .

correction possible :

In [1]:
```
L = [2, 3, 6, 7, 11, 14, 18, 19, 24]
for k in range(7):
    if L[k] == 14 :
        print(k)
```

```
5
```

Exercice 2

Écrire une fonction `trouve(L, p)` qui renvoie l'indice d'une valeur `p` dans une liste `L` . Si la valeur `p` n'est pas trouvée, on renverra `None` .

correction possible :

In [1]:
```
def trouve(L, p) :
    for k in range(len(L)) :
        if L[k] == p :
            return k
    return None
```

Complexité de la méthode

Le nombre (maximal) d'opérations nécessaires est proportionnel à la taille de la liste à étudier. Si on appelle n la longueur de la liste, on dit que cet algorithme est **d'ordre** n. On rencontrera parfois la notation $O(n)$.

Questions :

- La méthode utilisée nécessitait-elle que la liste soit triée ? (réponse : non)
- Est-on sûr que cet algorithme s'arrête ? (réponse : oui, car il fait au maximum n tours de boucle)

Méthode 2 : recherche dichotomique

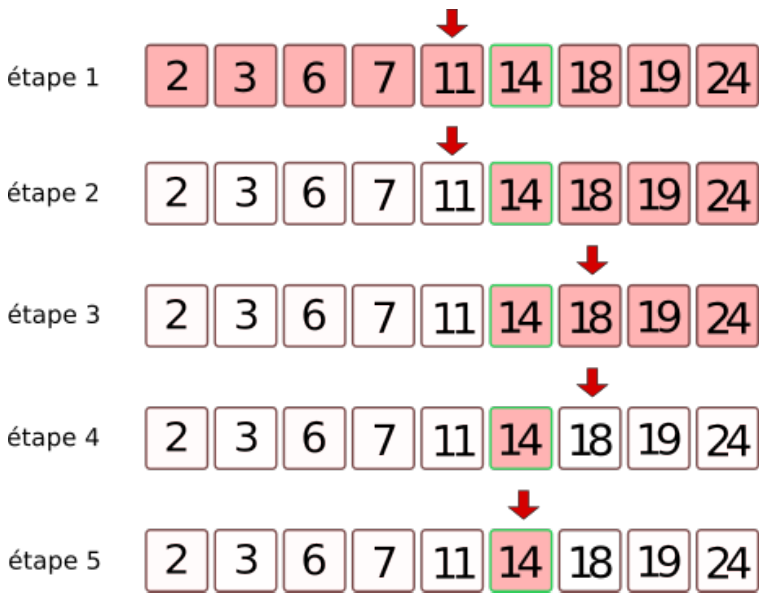
Comment appliquer la méthode vue dans l'activité d'introduction ? (discussion avec les élèves pour élaboration de l'algorithme)

Exemple d'algorithme pouvant être proposé par un élève :

- on se place *au milieu* de la liste.
- on regarde si on est inférieur ou supérieur à la valeur cherchée.
- on ne garde que la bonne moitié de la liste qui nous intéresse, et on recommence jusqu'à trouver la bonne valeur.

Illustration

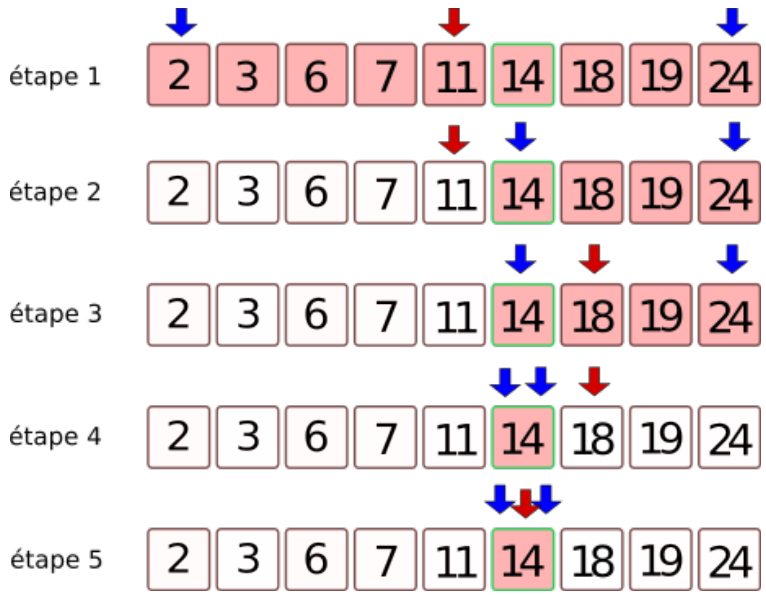
Recherchons la valeur 14 dans notre liste L .



- étape 1 : toute la liste est à traiter. On se place sur l'élément central. Son indice est la partie entière de la moitié de la longueur de la liste. Ici il y a 9 éléments, donc on se place sur le 4ème, qui est 11.
- étape 2 : on compare 11 à la valeur cherchée (14). Il faut donc garder tout ce qui est supérieur à 11.
- étape 3 : on se place au milieu de la liste des valeurs qu'il reste à traiter. Ici il y a 4 valeurs, donc il n'y a pas de valeur centrale. On va donc se positionner sur la 2ème valeur, qui est 18.
- étape 4 : on compare la valeur 18 à la valeur cherchée : 14. Elle est supérieure, donc on garde ce qui est à gauche. Il n'y a plus qu'une valeur.
- étape 5 : on se place sur la valeur 14 et on compare avec 14. La valeur est trouvée.

Programmation de la méthode de dichotomie

Nous allons travailler avec deux variables `indice_debut` et `indice_fin` qui vont délimiter la liste à étudier. Ces indices sont représentés en bleu sur la figure ci-dessous. La valeur de l' `indice_central` (représenté en rouge) sera égale à `(indice_debut + indice_fin) // 2`



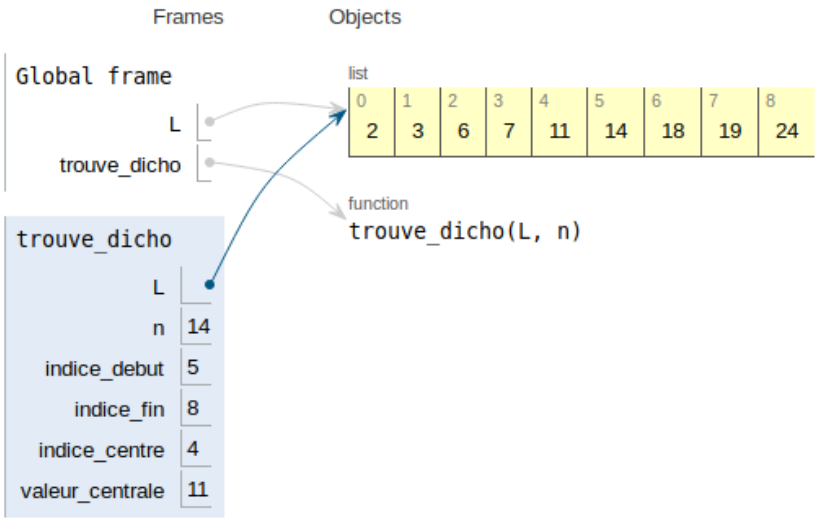
Le programme s'arrête lorsque la valeur cherchée a été trouvée, ou lorsque `indice_fin` devint inférieur à `indice_debut` .

In [2]:
```
def trouve_dicho(L, n) :
    indice_debut = 0
    indice_fin = len(L) - 1
    while indice_debut <= indice_fin :
        indice_centre = (indice_debut + indice_fin) // 2
        valeur_centrale = L[indice_centre]
        if valeur_centrale == n :
            return indice_centre
        if valeur_centrale < n :
            indice_debut = indice_centre + 1
        else :
            indice_fin = indice_centre - 1
    return None
```

In [24]:
```
print(trouve_dicho(L,14))
print(trouve_dicho(L,2))
print(trouve_dicho(L,24))
print(trouve_dicho(L,1976))
```

```
5
0
8
None
```

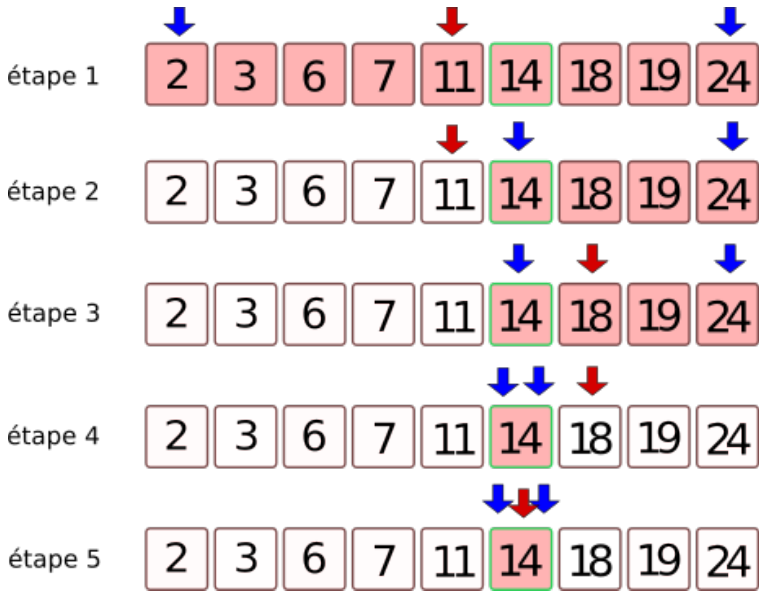
Une visualisation de l'évolution des variables `indice_debut` et `indice_fin` est disponible sur le site pythontutor via ce lien (http://pythontutor.com/visualize.html#code=L%20%3D%20%5B2,%203,%206,%207,%2011,%2014,%2018,%2019,%2024%5D%0A%0Adef%20trouve_dicho%28L,%20n%29%20%3A%0A%20%20%20%20indice_debut%20%3D%200%0A%20%20%20%20indice_fin%20%3D%20len%28L%29%20-%201%0A%20%20%20%20while%20indice_debut%20%3C%3D%20indice_fin%20%3A%0A%20%20%20%20%20%20%20%20indice_centre%20%3D%20%28indice_debut%20%2B%20indice_fin%29%20//%202%0A%20%20%20%20%20%20%20%20valeur_centrale%20%3D%20L%5Bindice_centre%5D%0A%20%20%20%20%20%20%20%20if%20valeur_centrale%20%3D%3D%20n%20%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20return%20indice_centre%0A%20%20%20%20%20%20%20%20if%20valeur_centrale%20%3C%20n%20%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20indice_debut%20%3D%20indice_centre%20%2B%201%0A%20%20%20%20%20%20%20%20else%20%3A%0A%20%20%20%20%20%20%20%20%20%20%20%20indice_fin%20%3D%20indice_centre%20-%201%0A%20%20%20%20return%20None%0A%0Aprint%28trouve_dicho%28L,14%29%29&cumulative=false&curInstr=0&heapPrimitives=nevernest&mode=display&origin=opt-frontend.js&py=3&rawInputLstJSON=%5B%5D&textReferences=false).



Terminaison de l'algorithme

Est-on sûr que l'algorithme va se terminer ?
La boucle `while` qui est utilisée doit nous inciter à la prudence (voir cours (https://github.com/glassus/nsi/blob/master/Premiere/Theme01_Bases_de_Python/02_Boucle_while/boucles_while.ipynb) sur la boucle While).
Il y a en effet le risque de rentrer dans une boucle infinie.
Pourquoi n'est-ce pas le cas ?

Aide : observer la position des deux flèches bleues lors de l'exécution de l'algorithme



La condition de la boucle `while` est `indice_debut <= indice_fin` , qui pourrait aussi s'écrire `indice_fin >= indice_debut` .
Au démarrage de la boucle, on a :

In [25]:
```
    indice_debut = 0
    indice_fin = len(L) - 1
```

Ceci qui nous assure donc de bien rentrer dans la boucle.

Ensuite, à chaque étape, les deux variables `indice_debut` et `indice_fin` vont se **rapprocher** jusqu'à ce que le programme rencontre un `return` ou bien jusqu'à ce que `indice_fin` devienne inférieur à `indice_debut` .

Ceci nous assure donc que le programme va bien se terminer.

Variant de boucle
On dit que la valeur `indice_fin - indice_debut` représente le **variant de boucle** de cet algorithme. Ce variant est un nombre entier, d'abord strictement positif, puis qui va décroître jusqu'à la valeur 0.

Complexité de l'algorithme

Combien d'étapes (au maximum) sont-elles nécessaires pour arriver à la fin de l'algorithme ?
Imaginons que la liste initiale possède 8 valeurs. Après une étape, il ne reste que 4 valeurs à traiter. Puis 2 valeurs. Puis une seule valeur.
Il y a donc 4 étapes avant de trouver la valeur cherchée.

Exercice :

1. Remplissez le tableau ci-dessous :

taille de la liste	1	2	4	8	16	32	64	128	256
nombre d'étapes	_	_	_	4	_	_	_	_	_

1. Pouvez-vous deviner le nombre d'étapes nécessaires pour une liste de 4096 termes ?
2. Pour une liste de 2^n termes, quel est le nombre d'étapes ?

Conclusion : C'est le nombre de puissances de 2 que contient le nombre N de termes de la liste qui est déterminant dans la complexité de l'algorithme. Ce nombre s'appelle le *logarithme de base 2* et se note $\log_2(N)$. On dit que l'algorithme de dichotomie a une **vitesse logarithmique**. On rencontrera parfois la notation $O(\log_2(n))$.

Expériences et comparaison des vitesses d'exécution

Avec une liste contenant 100 000 valeurs

In [1]:
```
# ce code permet de transformer le contenu du fichier input_centmille.txt en
une liste L de 100 000 valeurs.
# il est rappelé ici car très utile aux élèves pour résoudre les défis posés
sur le site :
# https://callicode.fr/pydefis/listedefis , sur lequel les élèves sont fréqu
emment appelés à aller.

f = open("input_centmille.txt",'r')
l = f.readlines()
L = []
for k in l :
    L.append(int(k[:-1]))
```

Mesurons le temps nécessaire pour trouver l'indice de la dernière valeur de la liste (qui est 299474) avec la méthode de balayage (méthode 1) :

In [12]:
```
%timeit trouve(L, 299474)
```

```
4.46 ms ± 85.1 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

Mesurons le temps nécessaire pour trouver l'indice de la dernière valeur de la liste (qui est 299474) avec la méthode par dichotomie (méthode 2) :

In [14]:
```
%timeit trouve_dicho(L, 299474)
```

```
2.62 µs ± 33.2 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Comparaison des deux méthodes : l'algorithme dichotomique est bien plus rapide que l'algorithme de balayage (la différence d'ordre de grandeur est de 10^3, qui correspond bien à l'ordre de grandeur de $\frac{n}{\log(n)}$ lorsque n vaut 10^5).

Avec une liste contenant 1 000 000 valeurs (soit 10 fois plus que la liste précédente)

In [17]:
```
# ce code permet de transformer le contenu du fichier million.txt en une lis
te L de 1 000 000 valeurs.
f = open("input_million.txt",'r')
l = f.readlines()
L = []
for k in l :
    L.append(int(k[:-1]))
```

Mesurons le temps nécessaire pour trouver l'indice de la dernière valeur de la liste (qui est 2999306) avec la méthode de balayage (méthode 1) :

In [18]:
```
%timeit trouve(L, 2999306)
```

```
46.5 ms ± 1.04 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Mesurons le temps nécessaire pour trouver l'indice de la dernière valeur de la liste (qui est 2999306) avec la méthode par dichotomie (méthode 2) :

In [19]:
```
%timeit trouve_dicho(L, 2999306)
```

```
3.02 µs ± 48.7 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Comparaison des deux méthodes : l'algorithme dichotomique est toujours bien plus rapide que l'algorithme de balayage (la différence d'ordre de grandeur est de 10^4, qui correspond bien à l'ordre de grandeur de $\frac{n}{\log(n)}$ lorsque n vaut 10^6).

Influence de la taille de la liste sur la vitesse de chaque méthode :

- méthode 1: la recherche dans une liste 10 fois plus grand prend environ 10 fois plus de temps : la vitesse de l'algorithme est bien proportionnelle à la taille n de la liste. $\frac{10^6}{10^5} = 10$
- méthode 2: la recherche dans une liste 10 fois plus grand prend environ 1,2 fois plus de temps : la vitesse de l'algorithme est bien proportionnelle au **logarithme** de la taille n de la liste. $\frac{\log(1000000)}{\log(100000)} \approx 1,2$

Remarque : Il ne faut toutefois pas oublier que la méthode dichotomique, bien plus rapide, nécessite que la liste ait été auparavant triée. Ce qui rajoute du temps de calcul ! (*cf cours sur les tris en fin d'année*)