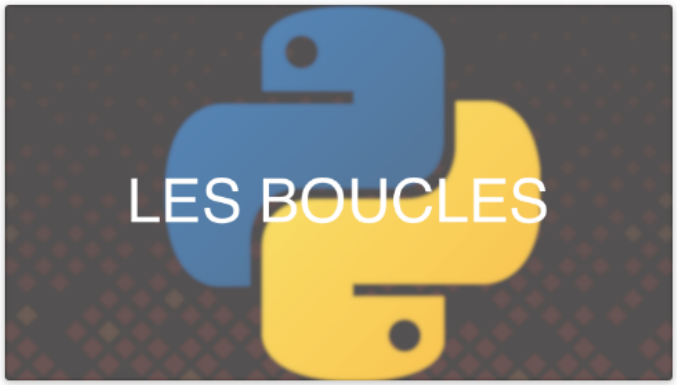
Une nouvelle recherche....

Après la **recherche séquentielle** vue en début de chapitre, puis deux **tris,** nous allons de nouveau chercher un élément dans une liste mais cette fois-ci, TRIÉE ….

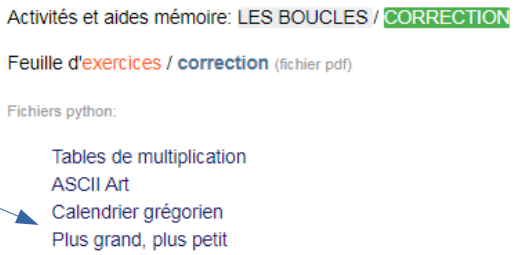


Rappelez vous

On revient sur le JEU du nombre entre 1 et 100

1°) Retrouvez le programme écrit en Python qui cache un nombre entier entre 1 et 100 et demande à l'utilisateur de le retrouver.

Sur le site fredpeuriere.com, il est ici



On comptait même le nombre de coups

Rappelez le nombre maximal de coups **7**

et expliquez comment il a été obtenu

Le nombre de candidats restants si le nombre n'est pas donné est au plus

100 → 50 →25 → 13 → 7 → 4 → 2 → 1 **7 est le nombre minimum tel que $2^7 > 100$**

Observez et faire fonctionner le fichier *Jeu dicho inversé* depuis le site: décrire son rôle

Ce programme donne pour chaque nombre de 1 à 100,le nombre de coups pour retrouver ce nombre dans la liste triée de 1 à 100 (pas 1) par dichotomie

3°) Nous allons nous intéresser toujours à une liste triée mais, désormais la liste n'est plus forcément une liste de nombres consécutifs.....

a) Comment repérer les éléments de cette liste et surtout comment repérer l'élément du milieu de la liste ? **Par leur position, donnée par leur indice.**

b) Faire fonctionner le même principe A LA MAIN afin de savoir si la valeur **v = 81** est dans la liste triée suivante : list = [2 , 7 , 19 , 26 , 45 , 70 , 81 , 85 , 86 , 97]

ind	0	1	2	3	4	5	6	7	8	9
list	2	7	19	26	45	70	81	85	86	97

1 – m=int((0+9)/2)=4 , list[4]=45 ≠ 81

comme list[m] < 81, alors deb = m+1

2 - m=int((5+9)/2)=7 , list[7]=85 ≠ 81

comme list[m] > 81, alors fin = m-1

3 - m=int((5+6)/2)=5 , list[5]=70 ≠ 81

comme list[m] < 81, alors deb = m+1

4 - m=int((6+6)/2)=6 , list[6]=81 = 81

on renvoie 6 (indice de 81 dans list)

c) Recommencez seul avec v= 77

1 – m=int((0+9)/2)=4 , list[4]=45 ≠ 77

comme list[m] < 77, alors deb = m+1

2 - m=int((5+9)/2)=7 , list[7]=85 ≠ 77

comme list[m] > 77, alors fin = m-1

3 - m=int((5+6)/2)=5 , list[5]=70 ≠ 77

comme list[m] < 81, alors deb = m+1

4 - m=int((6+6)/2)=6 , list[6]=81 ≠ 77

comme deb=fin, l'élément v n'est pas dans list : on renvoie FAUX

Fiche d'identité de l' algorithme de **RECHERCHE DICHOTOMIQUE**

Principe : diviser pour régner
La méthode diviser pour régner consiste en trois étapes :

DIVISER : découper le problème initial en sous problèmes ;

RÉGNER : résoudre les sous problèmes

COMBINER : calculer une solution au problème initial à partir des solutions des sous problèmes

Dans un premier temps, la recherche par dichotomie va en fait trouver LA POSITION d'un élément dans la liste triée : il faut comparer l'élément avec la valeur de la case au milieu du tableau ; si les valeurs sont égales, on renvoie la position, sinon on recommence dans la moitié du tableau pertinente**. S**i l'élément n'est pas dans le tableau, on renvoie Faux

Exemple : la recherche d'un mot dans le dictionnaire

Pseudo Code

v ← élément cherché
deb ← indice du premier élément de la liste
fin ← indice du dernier élément de la liste
m ← int((deb + fin)/2)
Tant que deb <= fin faire
    si v = liste[m] alors
        renvoyer m
    sinon si liste[m] >v alors
        fin ← m–1
    sinon
        deb ← m+1
    fin si
    m ← int((deb + fin)/2)
Fin Tant que
Renvoyer Faux

Python

```
1  def dicho_rang(v,list):          #recherche la présence de l'entier v dans la liste triée list
2      deb = 0                      #indice de debut de liste initialisé à 0
3      fin = len(list)-1            #indice de fin de liste initialisé au dernier indice de la liste
4      m=int((deb+fin)/2)           # m est l'indice de l'élément du "milieu"
5      while deb<=fin:              # condition d'arret : la liste est parcourue
6          if list[m] ==v:          # v est trouvé dans la liste
7              return m
8          elif list[m]>v:          # on va s'occuper de la liste inférieure
9              fin = m-1
10         else :                   # on s'occupe de la liste supérieure
11             deb = m+1
12         m=int((deb+fin)/2)
13     return False                 # on indique que l'élément n'a pas été trouvé
```

TERMINAISON de l'algorithme
Nous allons nous intéresser au fait que cet algorithme s'arrête....en effet, prouver ici que l'algorithme remplit bien son rôle (CORRECTION de l'algorithme) n'est pas trivial...
Pour prouver qu'un algorithme s'arrête, on choisit une variable et on vérifie que la suite formée par les valeurs de cette variable au cours des itérations converge

en un nombre fini d'étapes vers une valeur satisfaisant la condition d'arrêt. Cette variable s'appelle un **variant de boucle.**

Il s'agit de démontrer que l'algorithme se termine dans tous les cas et qu'il n'y a pas de boucle infinie.

Dans ce cas précis, on peut choisir la suite $(fin_i - deb_i)_{i\in\mathbb{N}}$ .La condition d'arrêt de la boucle TANT QUE utilisée dans l'algorithme est **Tant que deb<=fin** mais on n'entre pas toujours dans la boucle :

Soit fin_i et deb_i avec i l'indice qui compte la i[ème] itération de la boucle : deb_0 =0 et fin_0 = n.

On note alors $m_i = (deb_i + fin_i)/2$

Si nous sommes à la k[ème] itération, il y a trois grandes possibilités :

- $deb_k > fin_k$ ou $list[m_k] = v$, l'algorithme se termine, car la boucle n'est pas parcourue.
- $deb_k \leqslant fin_k$ et $v < list[m_k]$, on "entre" dans la boucle : $deb_{k+1} = deb_k$ et $fin_{k+1} = m_k - 1$: on a $fin_{k+1} - deb_{k+1} < fin_k - deb_k$
- $deb_k \leqslant fin_k$ et $v > list[m_k]$, on "entre" dans la boucle : $deb_{k+1} = m_k + 1$ et $fin_{k+1} = fin_k$. On a alors $fin_{k+1} - deb_{k+1} < fin_k - deb_k$

On a donc chaque fois, $fin_{k+1} - deb_{k+1} < fin_k - deb_k$: la suite $(fin_i - deb_i)_{i\in\mathbb{N}}$ est strictement décroissante. Il existe donc un entier p de $\mathbb{N}$ tel que :

- soit $deb_p > fin_p$, l'algorithme va se terminer, car on ne parcourt pas la boucle et l'algorithme renvoie FAUX.
- soit $v = list[m_p]$ avec $m_p = (deb_p + fin_p)/2$, l'algorithme va se terminer, car on parcourt la boucle et l'algorithme renvoie le rang m_p .

L'algorithme se termine donc toujours.

Complexité en temps :

Le logarithme en base 2

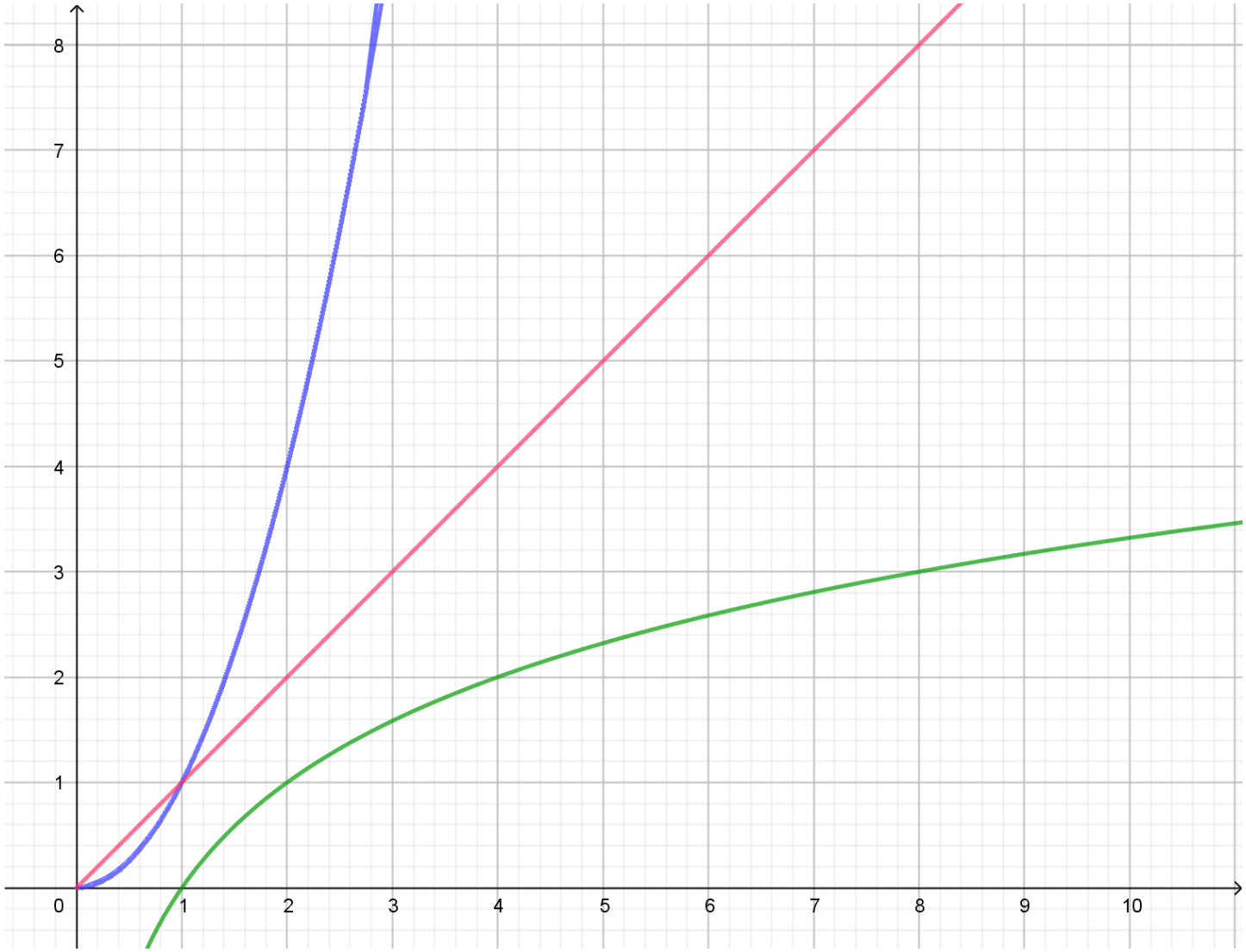
DEFINITION : On le note $\log_2$ **et pour** $x\in\mathbb{R}$, $\log_2(2^x) = x$

Lors d'une recherche dichotomique d'un élément v dans une liste triée « list » de longueur n, le nombre *f* de fois dans le cas le plus défavorable ($v \notin list$) où la liste sera coupée en deux vérifie $\frac{n}{2^f} = 1 \Leftrightarrow 2^f = n \Leftrightarrow f = log_2(2^n)$

La complexité en temps dans le pire des cas de l'algorithme de recherche dichotomique est $O(\log_2(n))$

L'algorithme de recherche dichotomique est donc plus efficace que l'algorithme de recherche séquentielle car pour tout x, $x > \log_2(x)$. Néanmoins, il est à noter que la recherche dichotomique utilise un tableau TRIE, ce qui peut aussi avoir un coût en temps...

$y = x$ (en rouge), $y = x^2$ (en bleu) et $y = \log_2(x)$ (en vert)



Exercices

Exercice 1 – *ANALYSER*

Combien de valeurs sont examinées lors d'un appel à la *dichotomie* sur la liste *[0,1,2,2,3,6,9,13,13,20]* pour trouver 7 ?

DICHOTOMIE

1 – m=int((0+9)/2)=4 , list[4]=3 ≠ 7 ***# 1ere valeur***

comme list[m] < 7, alors deb = m+1

2 - m=int((5+9)/2)=7 , list[7]=13 ≠ 7 ***#2ème valeur***
comme list[m] > 7, alors fin = m-1

3 - m=int((5+6)/2)=5 , list[5]=6 ≠ 7 ***#3ème valeur***
comme list[m] < 7, alors deb = m+1

4 - m=int((6+6)/2)=6 , list[6]=9 ≠ 7 ***#4ème valeur***
comme list[m] > 7, alors fin = m-1

5 - m=int((6+5)/2)=5 et deb>fin => on sort de la boucle et on renvoie r donc FAUX

On a examiné 4 valeurs.

SEQUENTIELLE

C'est le pire des cas (7 n'est pas dans la liste) : on doit donc examiner toutes les valeurs une par une : il faut tester 10 valeurs ! C'est plus du double de la dichotomie !!

Exercice 2 – *TRADUIRE ET DÉVELOPPER*

Écrire en Python une fonction *Repetition(n)* qui calcule et renvoie le plus petit entier f

tel que $2^f > n$.

Rappeler ce que représente f dans une recherche dichotomique dans une liste de taille n.

f représente le plus nombre de fois (donc entier) pour lequel la division par 2 de n donne un résultat inférieur à 1 : si on divise le nombre d'éléments de la liste par 2 de manière répétitive, f représente le nombre de divisions pour lequel est on certain de n'avoir plus qu'un seul choix.