Complexité d'un algorithme.

Durée d'exécution d'un programme.

I Complexité d'un algorithme.

Pour traiter un même problème, il existe souvent plusieurs algorithmes. Quand on doit choisir, l'un des critères est celui du ***temps d'exécution***, qu'on appelle aussi parfois ***coût*** de l'algorithme. Cette dénomination n'est pas usurpée car ce temps conditionne les ressources utilisées (machine sur laquelle on exécutera l'algorithme, consommation électrique…).
Pour un logiciel interactif, par exemple, un temps de réponse court est un élément essentiel du confort de l'utilisateur. De même, certains programmes industriels doivent être utilisés un grand nombre de fois dans un délai très court et même un programme qui n'est exécuté qu'une seule fois, par exemple un programme de simulation écrit pour tester une hypothèse de recherche, est inutilisable s'il demande des mois ou des années de calcul.

Par exemple, si l'on cherche à afficher la liste des diviseurs d'un nombre entier *n*, on peut écrire l'algorithme suivant :

Combien d'opérations ont été effectuées ?
Combien d'affichages ont été effectuées ?
Modifier l'algorithme précédent pour obtenir cette information.

Voici un théorème de mathématiques :

Théorème : soit $n = p \times q$ avec $p \geq \sqrt{n}$ alors q est un diviseur de n inférieur ou égal $\sqrt{n}$.

Il suffit donc de chercher chaque diviseur *q* inférieur ou égal à $\sqrt{n}$ et de calculer $p = \frac{n}{q} \in \mathbb{N}$ pour obtenir tous les diviseurs.
En utilisant ce théorème, proposer un nouvel algorithme calculant les diviseurs d'un entier n .

!!!! Dans le cas où *n* est un carré parfait, on prend soin de ne pas afficher sa racine carrée deux fois.

Combien d'opérations ont été effectuées ?
Combien d'affichages ont été effectuées ?

Ce deuxième algorithme ne fait plus que $\sqrt{n}$ itérations, qui effectuent chacune un calcul de reste, une ou deux comparaisons, et zéro, un ou deux affichage(s). Au total, il coûte donc

- un calcul de racine carrée
- $\sqrt{n}$ calculs de reste
- Entre 0 et $2\sqrt{n}$ affichages

En général, on cherche à déterminer comment le temps d'exécution d'un algorithme varie en fonction d'un paramètre qu'on appelle la ***taille*** du problème. Le temps de recherche des diviseurs d'un entier *n* dépend de *n*, qu'on pourra donc naturellement prendre comme taille du problème. Comme on l'a vu, selon l'algorithme, ce temps peut être proportionnel à *n* ou à $\sqrt{n}$.

L'évaluation du temps mis par un algorithme pour s'exécuter est un domaine de recherche à part entière, car elle se révèle parfois très difficile.

II Durée d'exécution d'un programme.

Dans le module ***time***, il existe une fonction nommée ***clock*** : elle renvoie la durée de temps en secondes (dont la signification exacte dépend du système d'exploitation) et qui permet par soustraction de mesurer les durées d'exécution.

```
from time import clock
def durée(fonction,n):
    """renvoie la duree d execution de fonction"""
    debut=clock()
    fonction(n)
    fin=clock
    return(fin-debut)
```

```
0.17871675356099104 secondes pour executer la fonction diviseur si n= 100
0.18052910136435685 secondes pour executer la fonction diviseur_bis si n= 100
```

```
1.1924845978064127 secondes pour executer la fonction diviseur si n= 1000000
0.8664632774604684 secondes pour executer la fonction diviseur_bis si n= 1000000
>>>
```

Autre exemple : on cherche à créer la liste des valeurs de la fonction sinus pour les n premiers entiers avec n assez grand. Le programme suivant permet de comparer (en terme de temps d'exécution) trois manières différentes de créer une liste :

```python
from time import clock
from math import sin

def f1(n):
    l=[]
    for i in range(n):
        l+=[sin(i)]
    return(l)

def f2(n):
    l=[]
    for i in range(n):
        l.append(sin(i))
    return(l)

def f3(n):
    return[sin(i) for i in range (n)]

def duree(fonction,n=1000000):
    """renvoie la duree d execution de fonction"""
    debut=clock()
    fonction(n)
    fin=clock()
    return(fin-debut)

print('utilisation de +=',round(duree(f1),5),'s')
print('utilisation de append',round(duree(f2),5),'s')
print('utilisation de list-comprehension',round(duree(f3),5),'s')
```

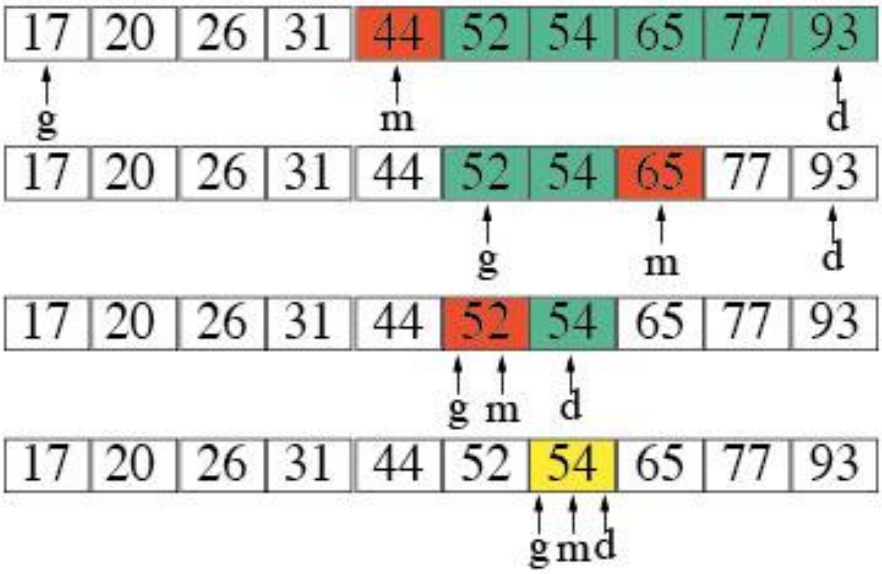
Voici deux résultats :

```
>>>
utilisation de += 2.31521 s
utilisation de append 1.94402 s
utilisation de list-comprehension 1.50485 s
>>>
utilisation de += 2.34961 s
utilisation de append 1.9183 s
utilisation de list-comprehension 1.4707 s
>>>
```

Qu'en conclure ?

Exercice : on veut déterminer si un élément x appartient à une liste L déjà triée.
1°) Ecrire un premier algorithme :

2°) En vous inspirant du schéma ci-dessous, écrire un algorithme, le plus « efficace » possible, pour réaliser cette recherche.



Recherche dichotomique

Si la liste sur laquelle est appliquée la recherche d'un élément est triée, le temps de recherche peut être singulièrement diminué en recourant à une méthode permettant d'éliminer la moitié des éléments de la liste à chaque nouvelle itération. Cet algorithme, appelé **recherche dichotomique**, consiste à comparer l'élément cherché avec l'élément central de la liste. Si l'élément cherché lui est inférieur, il doit se trouver dans la première moitié de la liste; si l'élément cherché lui est supérieur, il doit se trouver dans la deuxième moitié de la liste. En appliquant récursivement cette technique à la moitié choisie, l'algorithme aboutira soit à la position de l'élément cherché, soit à aucune position auquel cas l'élément n'appartient pas à la liste, comme l'illustre la figure ci-dessous:

Voici le programme Python correspondant :