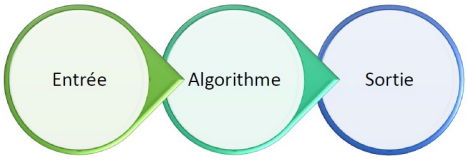
Le mot « algorithme » vient du nom du mathématicien Al-Khwârizmî1 (latinisé au Moyen Âge en Algoritmi), qui, au ixe siècle écrivit le premier ouvrage systématique donnant des solutions aux équations linéaires et quadratiques.

Objectif :

- Découverte de quelques algorithmes tel que le parcours séquentiel d'un tableau
- Notion de coût et de complexité.

Algorithme c'est quoi ?

- Un algorithme est une procédure pas à pas de résolution d'un problème s'exécutant en un temps fini
- La plupart des algorithmes transforment des données d'entrée en des données de sortie



Recette de cuisine :

> Source:N Nisse

1. la séquence d'opérations est non ambigüe / systématique (l'ordre des opérations, la définition de chaque opération... sont parfaitement définis) 2. si l'entrée est du bon type, la sortie doit être valide (celle attendue).L'algorithme est alors correct.

Ex: si vous avez des oeufs (quels qu'ils soient), du lait (en bonne quantité)... vous voulez une île flottante, pas une bouillabaisse... Au contraire, si vous n'avez que du poisson, n'espérez pas une île flottante...

Exemple : Démarrage d'une voiture.

1. prendre la clé ;
2. ouvrir la voiture ;
3. s'asseoir et régler siège et rétroviseurs ;
4. insérer et tourner la clé ;
5. baisser le frein à main ;
6. bidouiller les pédales...
7. si l'ordre est modifié, ça peut mal se passer...si l'entrée n'est pas du bon type (ex: un vélo au lieu d'une voiture), vous n'obtiendrez pas le résultat espéré...

> Source:N Nisse

Cas concret : Recherche d'une occurence dans un tableau

- Prenons l'exemple d'un algorithme qui prend en entrée un tableau t d'entiers et un entier x, et qui "répond" par "oui" ou par "non" à la question "x est-il présent dans le tableau t ?". Dans ce cas, la "valeur de sortie" sera "oui" ou "non".

Entrée [ ]:

```
VARIABLE
t : tableau d'entiers
x : nombre entier
tr : booléen (VRAI ou FAUX)
i : nombre entier
DEBUT
tr ← FAUX
i ← 0
tant que i<=longueur(t) et que tr==FAUX:
  si t[i]==x:
    tr ← VRAI
  fin si
  i ← i+1
fin tant que
renvoyer la valeur de tr
```

- La première chose à faire quand on étudie un algorithme, c'est de le "faire tourner à la main" : on "exécute" l'algorithme en utilisant uniquement une feuille et un crayon.

Faite tourner "à la main" l'algorithme pour

- t=[5,8,15,23]
- pour x=15 et pour x=12.

Programmer cet algorithme et tester votre programme

Entrée [21]:

```
#Solution
t=[5,8,15,23]
x=23
tr=False
i=0
while i<= len(t)-1 and tr==False: #attention indice il faut mettre len(t)-1
    if t[i]==x:
        tr=True

    i+=1
print (tr)
```

```
True
```

Exercices :

Testez ces cinq fonctions avec python tutor pour différentes valeurs de v et t.

[http://www.pythontutor.com/visualize.html#mode=edit (http://www.pythontutor.com/visualize.html#mode=edit)](http://www.pythontutor.com/visualize.html#mode=edit)

Fonction random avec python

- import random
- t= [random.randrange(0,21) for i in range (10)]

Entrée [22]:

```
#Fonction 1
def fonct1(v,t):
    trouvee=False
    for i in range (len(t)):
        if t[i]==v:
            trouvee=True
```

Entrée [23]:

```
#Fonction 2
def fonct2(v,t):
    i=0
    while t[i]!=v:
        i+=1
```

Entrée [33]:

```
#Fonction 3
def fonct3(v,t):
    i=0
    while i<len(t) and t[i] != v:
        i=i+1
    return i<len(t)
```

Entrée [31]:

```
#Fonction 4
def fonct4(v,t):
    i=0
    while i < len(t):
        if t[i]==v:break
        i=i+1
    return i<len(t)
```

Entrée [32]:

```python
#Fonction 5
def fonct5(v,t):
    for i in range (len(t)):
        if t[i]==v :
            return True
```

Entrée [99]:

```python
# fonction perf_counter qui permet de déterminer le temps d'execution
import random
from time import*
debut=perf_counter()
t = [random.randrange(0, 1000) for i in range(200000)]
fonct5 (12,t)
fin = perf_counter()
```

```
temps d'execution (en ms) : 455.5897000027471
```

Entrée [123]:

```python
# test des différentes fonctions
import random
t= [random.randrange(0,21) for i in range (20)]
print (t)
```

```
[18, 9, 16, 4, 6, 9, 8, 1, 6, 11, 0, 7, 1, 20, 5, 8, 16, 1, 5, 12]
```

Out[123]: True

Notion de Complexité d'un algorithme :

La notion de complexité d'un algorithme va rendre compte de l'efficacité de cet algorithme. Pour un même problème, par exemple trié un tableau, il existe plusieurs algorithmes, certains algorithmes sont plus efficaces que d'autres (par exemple un algorithme A mettra moins de temps qu'un algorithme B pour résoudre exactement le même problème, sur la même machine).

Il existe **2 types de complexité : une complexité en temps et une complexité en mémoire.** Nous nous intéresserons ici uniquement à la complexité en temps. **La complexité en temps est directement liée au nombre d'opérations élémentaires** qui doivent être exécutées afin de résoudre un problème donné. L'évaluation de ce nombre d'opérations élémentaires n'est pas chose facile, on rencontre souvent des cas litigieux.

Prenons pour exemple notre algorithme "x est-il présent dans le tableau t ?".**

Il y a 2 cas à traiter :

1. L'entier recherché est bien présent dans le tableau, il se trouve à la position d'index j
2. L'entier recherché n'est pas présent dans le tableau

Cas 1: L'entier est bien dans le tableau.

Entrée [ ]:

```
VARIABLE
t : tableau d'entiers
x : nombre entier
tr : booléen (VRAI ou FAUX)
i : nombre entier
DEBUT
1 fois   tr ← FAUX
1 fois   i ← 1
j+1 fois tant que i<=longueur(t) et que tr==FAUX:
j fois      si t[i]==x:
1 fois       tr ← VRAI
           fin si
j fois     i ← i+1
        fin tant que
1 fois  renvoyer la valeur de tr
```

Au total nous avons : 1 + 1 + j + 1 + j + 1 + j + 1 = 3j + 5 opérations élémentaires

Cas 2: L'entier est bien dans le tableau.

Entrée [ ]:

```
VARIABLE
t : tableau d'entiers
x : nombre entier
tr : booléen (VRAI ou FAUX)
i : nombre entier
DEBUT
1 fois   tr ← FAUX
1 fois   i ← 1
n+1 fois tant que i<=longueur(t) et que tr==FAUX:
n fois      si t[i]==x:
0 fois       tr ← VRAI
           fin si
n fois     i ← i+1
        fin tant que
1 fois  renvoyer la valeur de tr
```

Au total nous avons : 1 + 1 + n + 1 + n + 0 + n + 1 = 3n + 4 opérations élémentaires

Calcul de la complexité de notre exemple.

- Comme dans la plupart des cas n > j, on effectue plus d'opérations élémentaires quand le nombre recherché n'est pas dans le tableau.
- **On parlera de complexité dans le "pire" des cas.**

Pour notre exemple le nombre total d'opérations élémentaires est de 3n + 4

- On s'interresse toujours a des tableaux de grande tailles car plus les tableaux sont grands et plus les différences entre les algorithmes seront flagrantes.
- Pour comparer des algorithmes, nous allons donc uniquement nous intéresser à ce que l'on appelle **"l'ordre de grandeur asymptotique".** La définition précise de cet "ordre de grandeur asymptotique" est trop complexe pour être abordé ici. Vous devez juste savoir que cet "ordre de grandeur asymptotique" concerne les cas où l'on prend n très très grand. On note cet "ordre de grandeur asymptotique" avec un O majuscule. Pour le cas qui nous intéresse, nous aurons :
- **3n+4 = O(n)**
- La relation ci-dessus signifie que "3n+4 est dominée asymptotiquement par **n**"
- nous utiliserons systématiquement cette notation O pour exprimer la complexité des algorithmes : au final on dira donc que la complexité de notre algorithme "x est-il présent dans le tableau t ?" est **O(n)**.

Exercice:

À faire vous-même 2

- Écrivez un algorithme permettant de trouver le plus grand entier présent dans un tableau. Vous ferez "tourner à la main" votre algorithme en utilisant le tableau t = [3,5,1,8,4,2]. Vous déterminerez ensuite la complexité de votre algorithme.

Entrée [125]:

```
# A faire
```

Entrée [129]:

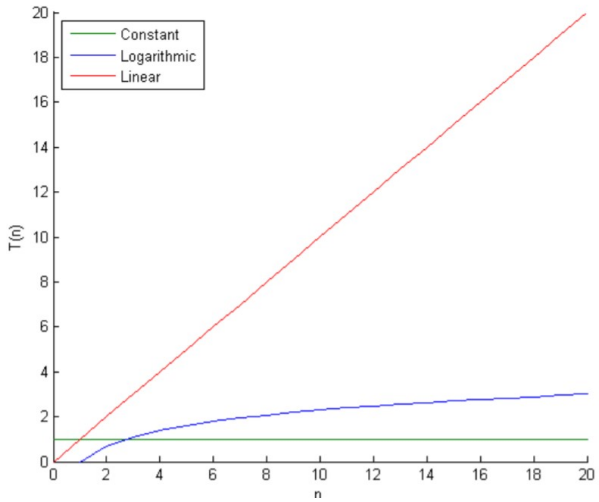
```
t = [3, 5, 1, 8, 4, 2]
```

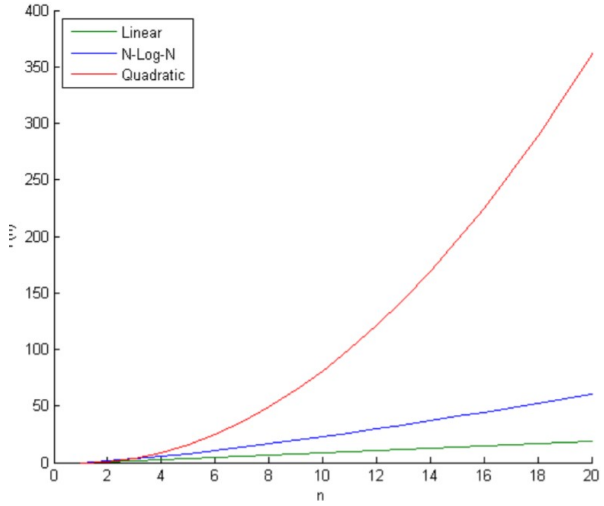
...

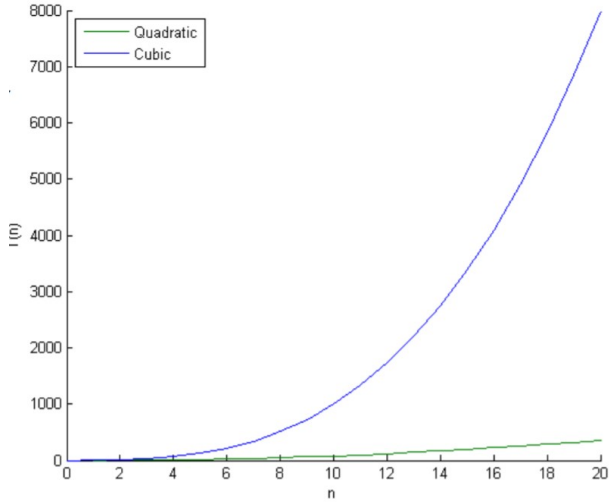
Les fonctions importantes

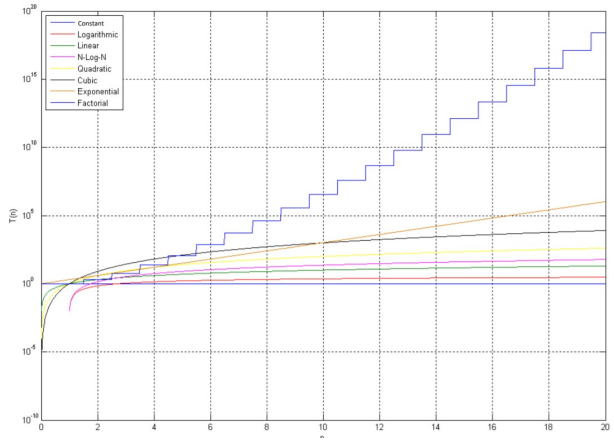
Huit fonctions sont souvent utilisées en analyse algorithmique

- Constante ≈ 1
- Logarithmique ≈ log n
- Linéaire ≈ n
- N-log-N ≈ n log n
- Quadratique ≈ n^2
- Cubique ≈ n^3
- Exponentielle ≈ 2^n
- Factorielle ≈ n!


Constant
Logarithmic
Linear
T(n)
n


Linear
N-Log-N
Quadratic
n


Quadratic
Cubic
T(n)
n


Constant
Logarithmic
Linear
N-Log-N
Quadratic
Cubic
Exponential
Factorial
T(n)

Entrée [ ]: