

COMPLEXITÉ ALGORITHMIQUE

Exemple : Fonctions qui prend en argument un entier n et qui renvoie la somme des entiers de 1 à n :

Algorithme n°1

```
Fonction somme1(Entier n)
  Entier somme=0
  Entier i

  Pour i de 1 à n faire
    somme = somme + i
  Fin_Pour

  Renvoyer somme
```

Si n augmente, le
nombre d'affectation
(ici 1 seule)
n'augmente pas

Si n augmente, le
nombre d'affectations
augmente d'autant

Algorithme n°2

```
Fonction somme2(Entier n)
  Entier somme=0

  somme =  $n*(n+1)/2$ 
  Fin_Pour

  Renvoyer somme
```

COMPLEXITÉ ALGORITHMIQUE

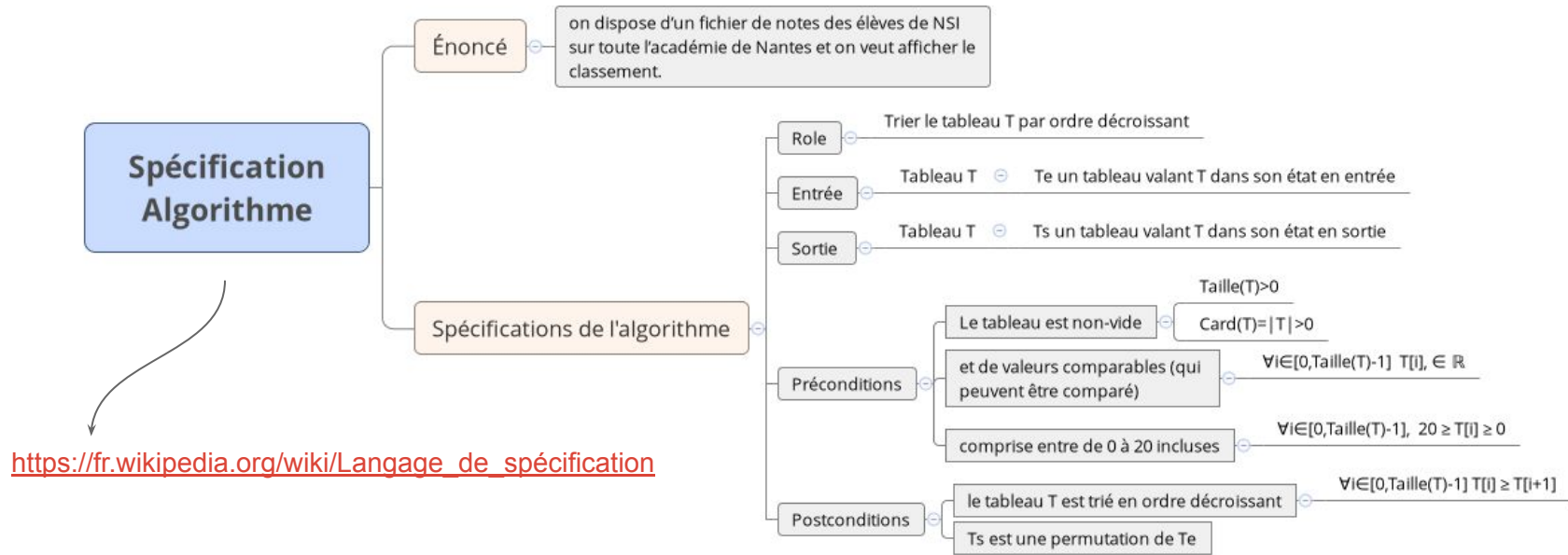
La notion de complexité en algorithmique est liée à la mesure des performances a priori des algorithmes :

- Aucun rapport donc avec sa difficulté (à concevoir ou à comprendre)
- Même s'il y aura parfois un lien ... inversement proportionnel (les solutions simples, naïves, seront souvent peu performantes)
- Complexité spatiale (mémoire) vs Complexité temporelle (instructions élémentaires : comparaison / affectation)

Complexité \neq Difficulté !

SPÉCIFICATION D'UN ALGORITHME

Version contextuelle : On dispose d'un fichier de notes des élèves de NSI sur toute l'académie de Nantes et on veut afficher le classement.



Décrire le quoi, pas le comment.

OUTILS MATHÉMATIQUES

- Somme d'une constante indépendante : $\sum_{j=a}^b \text{constante} = (b-a+1) \cdot \text{constante}$
- Somme des n premiers entiers : $1+2+3+\dots+(n-1)+n = \sum_{i=1}^n i = \frac{n \cdot (n+1)}{2}$
- Somme des n-1 premiers entiers : $0+1+2+3+\dots+(n-1) = \frac{n \cdot (n+1)}{2} - n = \frac{n(n-1)}{2}$
- Somme de puissances : $\sum_{k=0}^{n-1} a^k = \frac{a^n - 1}{a - 1}$
- Somme des puissances de 2 : $\sum_{k=0}^{n-1} 2^k = \sum_{k=1}^n (2^{(k-1)}) = \frac{2^n - 1}{2 - 1} = 2^n - 1 = 1 + 2 + 4 + 8 + \dots + 2^{(n-1)}$
- Logarithme base a : $y(x) = a^x \Leftrightarrow x(y) = \log_a(y)$ avec $\log_a(y) = \frac{\log_b(y)}{\log_b(a)}$

IMAGE : CONVERSION COULEURS -> NIVEAUX DE GRIS

```
from PIL import Image
img=Image.open("phare1.jpg")
largeur,hauteur=img.size

for y in range(hauteur):
    for x in range(largeur):
        r,v,b=img.getpixel((x,y))
        moyenne=(r+v+b)/3
        r = moyenne
        v = moyenne
        b = moyenne
        img.putpixel((x,y),(r,v,b))
img.show()
```

→ Taille des données en entrée

Taille de l'image noté "n", valant hauteur x largeur en pixels

→ Pire cas / Meilleur cas

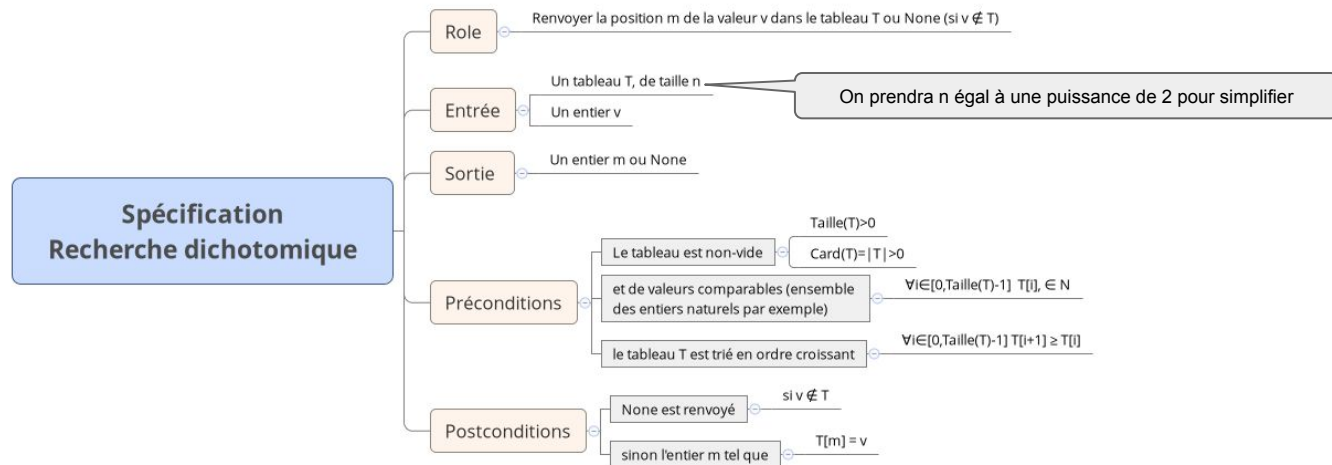
Il n'y a pas de pire / meilleur cas, il est nécessaire de parcourir tous les pixels l'image

$$O_{affectation}(n) = \sum_{y=0}^{hauteur-1} \left(\sum_{x=0}^{largeur-1} 6 \right) = \sum_{y=0}^{hauteur-1} (6 \cdot (largeur-1-0+1))$$

$$O_{affectation}(n) = 6 \cdot largeur \cdot \sum_{y=0}^{hauteur-1} 1 = 6 \cdot largeur \times hauteur = 6 \cdot n$$

- Il n'y avait pas besoin de calculs savants pour appréhender la **complexité linéaire** (ordre n) de cet algorithme. Il faut parcourir les "n" pixels !!!
- Les bornes de la boucle interne (for x ...) sont indépendantes de y (la boucle externe)

RECHERCHE DICHOTOMIQUE (BINARY SEARCH)



→ Taille des données en entrée

Taille de la liste noté " n "

→ Pire cas / Meilleur cas

L'élément recherché n'est pas dans la liste

→ Meilleur cas

L'élément recherché est au centre de la liste

Approche naïve :

Parcourir la totalité de la liste T (complexité en n)

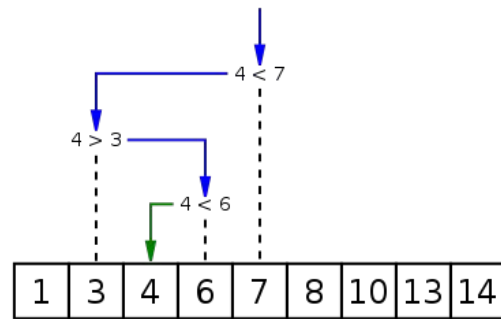
OU MIEUX profiter du fait que le tableau soit trié

RECHERCHE DICHOTOMIQUE (BINARY SEARCH)

```

fonction recherche_dichotomique( element, tableau ) entier
    bas ← 0
    haut ← longueur(tableau)-1
    trouve ← Faux
    tant que bas <= haut et trouve = Faux
        milieu ← (bas + haut) // 2
        si element = tableau[milieu] alors
            trouve ← Vrai
        sinon
            si element > tableau[milieu] alors
                bas ← milieu+1
            sinon
                haut ← milieu-1
            fin si
        fin si
    fin tant que
    si trouve=vrai alors
        indice ← milieu
    sinon
        indice ← -1
    fin si
    retourner indice

```



https://upload.wikimedia.org/wikipedia/commons/f/f7/Binary_search_into_array.png

- **Meilleur cas** : l'élément recherché est au milieu de la liste (Exemple : cas où la valeur 7 est recherchée).
- **Pire cas** : l'élément recherché n'est pas dans la liste (Exemple : cas où la valeur 9 est recherchée).

L'algorithme boucle k fois afin d'avoir bas>haut (soit l'inverse de bas <= haut) :

$$\frac{n}{2^k}=1 \rightarrow k=\log_2(n) \quad O(n)=\sum_0^k A=A.(k+1)=A.\log_2(n)+A$$

A le nombre d'opérations élémentaire de la boucle

TRI SÉLECTION / ORDRE CROISSANT

```
01 Début Tri Sélection
02   pour i de 1 jusqu'à n faire
03     pour j de i+1 jusqu'à n faire
04       si  $T[j] < T[i]$  alors
05          $temp = T[j]$ 
06          $T[j] = T[i]$ 
07          $T[i] = temp$ 
08       Fin si
09     Fin pour
10   Fin Pour
11 Fin Tri Sélection
```

Les indices de liste évoluent
de 1 à n (pseudo-code) !!!

```
T=[12,16,9,14,13,5]
```

```
print(T)
```

```
for i in range(len(T)):
    for j in range(i+1,len(T)):
        if  $T[j] < T[i]$  :
             $T[j], T[i] = T[i], T[j]$ 
    print(T)
```

→ Taille des données en entrée

Taille de la liste à trier noté "n"

→ Opérations élémentaires prises en compte

- Le nombre de comparaisons (ligne 04)
- Le nombre d'affectations (ligne 05, 06 et 07)

→ Pire cas / Meilleur cas

- Le nombre de comparaisons (ligne 04) est constant et fonction de la taille de la liste d'entrée, pas de meilleur / pire cas.

- Le nombre d'affectations (ligne 05, 06 et 07) est maximum si la liste d'entrée est triée en ordre décroissant, nul si elle est déjà triée en ordre croissant.

```
>>> %Run 'Tri_Sélection.py'
```

[12,	16,	9,	14,	13,	5]
[9,	16,	12,	14,	13,	5]
[5,	16,	12,	14,	13,	9]
[5,	12,	16,	14,	13,	9]
[5,	9,	16,	14,	13,	12]
[5,	9,	14,	16,	13,	12]
[5,	9,	13,	16,	14,	12]
[5,	9,	12,	16,	14,	13]
[5,	9,	12,	14,	16,	13]
[5,	9,	12,	13,	16,	14]
[5,	9,	12,	13,	14,	16]

TRI SÉLECTION / COMPLEXITÉ

01 Début Tri Sélection

02 pour i de 1 jusqu'à n faire

03 pour j de i+1 jusqu'à n faire

04 si $T[j] < T[i]$ alors

05 temp = Tab[j]

06 T[j] = Tab[i]

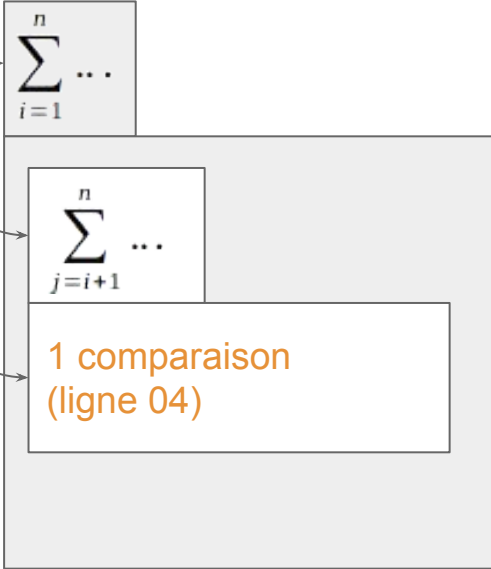
07 T[i] = temp

08 Fin si

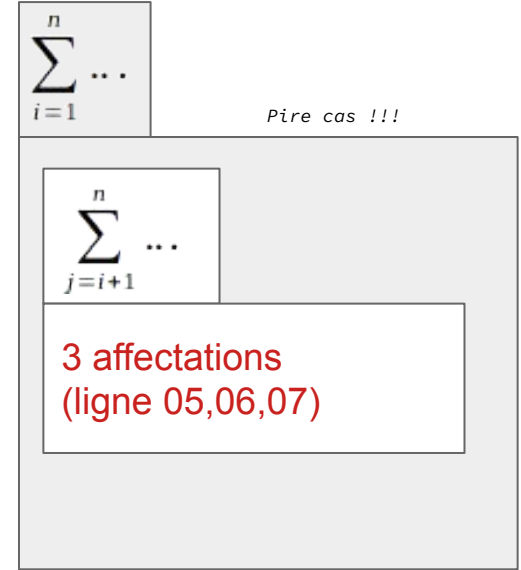
09 Fin pour

10 Fin Pour

11 Fin Tri Sélection



$$\sum_{i=1}^n \left(\sum_{j=i+1}^n 1 \right)$$



$$\sum_{i=1}^n \left(\sum_{j=i+1}^n 3 \right)$$

TRI SÉLECTION / COMPLEXITÉ (COMPARAISON)

$$O(n) = \sum_{i=1}^n \left(\sum_{j=i+1}^n 1 \right) = \sum_{i=1}^n (n - (i+1) + 1) = \sum_{i=1}^n (n - i)$$

$$O(n) = \sum_{i=1}^n (n - i) = \sum_{i=1}^n n - \sum_{i=1}^n i$$

$$O(n) = n \cdot (n - 1 + 1) - \frac{n \cdot (n + 1)}{2}$$

$$O(n) = \frac{1}{2} \cdot (n^2 - n)$$

complexité en comparaison

n	10	20	30	40	50	60	70	80	90	100
O(n)	45	190	435	780	1225	1770	2415	3160	4005	4950

La complexité en affectation est pour le pire cas, identique à la complexité en comparaison avec un facteur x3 (Les 2 boucles "pour" sont parcourus avec 3 affectations/boucle interne)

Tri_Sélection.py

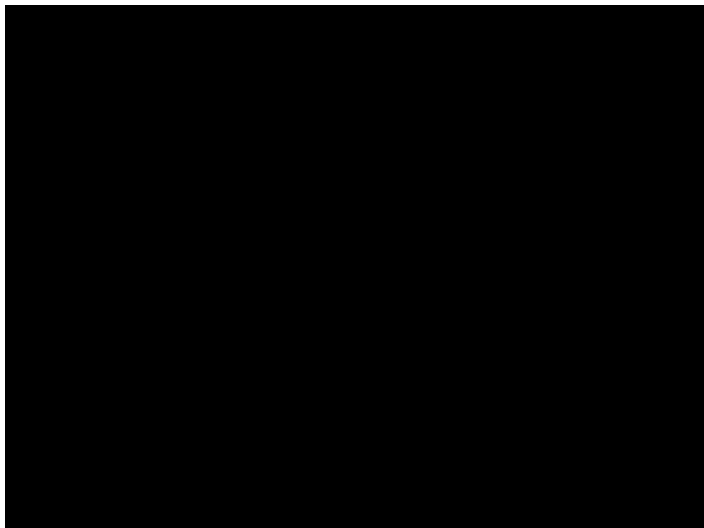
```
1 from random import *
2
3 for n in range(10,101,10):
4     T=[randint(1,100) for _ in range(n)]
5
6     nb_comparaison=0
7
8     for i in range(len(T)):
9         for j in range(i+1,len(T)):
10             nb_comparaison+=1
11             if T[j] < T[i]:
12                 T[j],T[i] = T[i], T[j]
13
14
15 print("n=",n," => nb_comparaison=",nb_comparaison)
16
17
```

Console

>>> %Run 'Tri_Sélection.py'

```
n= 10 => nb_comparaison= 45
n= 20 => nb_comparaison= 190
n= 30 => nb_comparaison= 435
n= 40 => nb_comparaison= 780
n= 50 => nb_comparaison= 1225
n= 60 => nb_comparaison= 1770
n= 70 => nb_comparaison= 2415
n= 80 => nb_comparaison= 3160
n= 90 => nb_comparaison= 4005
n= 100 => nb_comparaison= 4950
```

TRI INSERTION / ORDRE CROISSANT



- Le nombre d'affectation et le nombre de comparaison dépendent de la taille et de l'ordre initial des données.
- Pire cas : La liste est triée en ordre décroissant
- Meilleur cas : La liste est déjà triée dans l'ordre croissant

```
01  Début Tri insertion
02    pour i de 2 à n faire
03      j=i
04      memoire=Tab[i]
05      tant que j>1 et memoire<Tab[j-1]
06        Tampon=Tab[j-1]
07        Tab[j-1] = Tab[j]
08        Tab[j] = Tampon
09        j=j-1
10      fin_tant_que
11      Tab[j]= memoire
12    fin_pour
13  fin Tri Insertion
```

Meilleur cas :

Tab[i] < Tab[i-1] toujours faux, on s'épargne les 4 affectations du bloc while. Il n'y a qu'une comparaison par cycle de la boucle pour

Pire cas :

Tab[i] < Tab[i-1] toujours vrai, on effectue les 4 affectations du bloc while. La boucle while sera exécutée i fois (j>1)

Les indices de liste évoluent de 1 à n (pseudo-code) !!!

TRI INSERTION / ORDRE CROISSANT

```
01  Début Tri insertion
02      pour i de 2 à n faire
03          j=i
04          memoire=Tab[i]
05          tant que j>1 et memoire<Tab[j-1]
06              Tampon=Tab[j-1]
07              Tab[j-1] = Tab[j]
08              Tab[j] = Tampon
09              j=j-1
10          fin_tant_que
11          Tab[j]= memoire
12      fin_pour
13  fin Tri Insertion
```

Meilleur cas :

Tab[i] < Tab[i-1] toujours faux, on s'épargne les 4 affectations du bloc while. Il n'y a qu'une double comparaison par cycle de la boucle pour

Pire cas :

Tab[i] < Tab[i-1] toujours vrai, on effectue les 4 affectations du bloc while. La boucle while sera exécutée i fois (j>1) avec 2 comparaisons

$$O_{\text{comparaisonMC}}(n) = \sum_{i=2}^n 2 = 2 \cdot (n-2+1) = 2 \cdot n - 2$$

$$O_{\text{comparaisonMC}}(n) = 2 \cdot n - 2$$

$$O_{\text{comparaisonPC}}(n) = \sum_{i=2}^n \left(\sum_{j=i}^2 2 \right) = \sum_{i=2}^n \left(\sum_{j=2}^i 2 \right)$$

$$O_{\text{comparaisonPC}}(n) = \sum_{i=2}^n \left(\sum_{j=2}^i 2 \right) = \sum_{i=2}^n 2 \cdot (i-1)$$

$$O_{\text{comparaisonPC}}(n) = \sum_{i=2}^n 2 \cdot (i-1) = 2 \cdot \left(\sum_{i=2}^n i - \sum_{i=2}^n 1 \right)$$

$$O_{\text{comparaisonPC}}(n) = 2 \cdot \left(\frac{n^2}{2} + \frac{n}{2} - n \right)$$

$$O_{\text{comparaisonPC}}(n) = n^2 - n$$

Sans entrer dans les calculs, en affectation, la complexité est similaire, soit en linéaire (en n) dans le meilleur cas, et quadratique (en n²) dans le pire cas.

BILAN : TRI INSERTION / TRI SÉLECTION

Pire cas

Meilleur cas

Comparaison

n^2

n^2

Affectation

n^2

0

Tri par sélection

Pire cas

Meilleur cas

Comparaison

n^2

n

Affectation

n^2

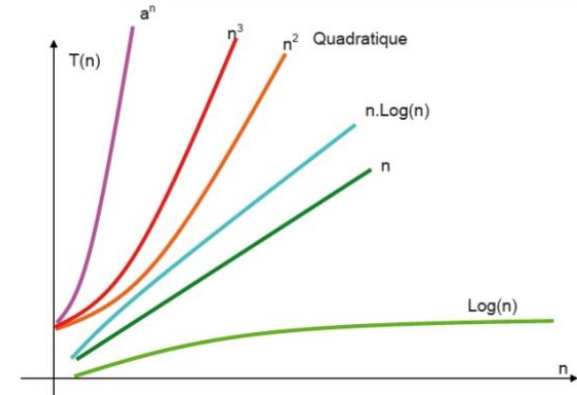
n

Tri par insertion

Tableau comparatif des tris procédant par comparaisons

Nom	Cas optimal	Cas moyen	Pire des cas	Complexité spatiale	Stable
Tri rapide	$n \log n$	$n \log n$	n^2	$\log n$ en moyenne, n dans le pire des cas ; variante de Sedgwick : $\log n$ dans le pire des cas	Non
Tri fusion	$n \log n$	$n \log n$	$n \log n$	n	Oui
Tri par tas	$n \log n$	$n \log n$	$n \log n$	1	Non
Tri par insertion	n	n^2	n^2	1	Oui
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	Non
Tri par sélection	n^2	n^2	n^2	1	Non
Timsort	n	$n \log n$	$n \log n$	n	Oui
Tri de Shell	n	$n \log^2 n$ ou $n^{3/2}$	$n \log^2 n$ pour la meilleure suite d'espacements connue	1	Non
Tri à bulles	n	n^2	n^2	1	Oui
Tri arborescent	$n \log n$	$n \log n$	$n \log n$ (arbre équilibré)	n	Oui
Smoothsort	n	$n \log n$	$n \log n$	1	Non
Tri cocktail	n	n^2	n^2	1	Oui
Tri à peigne	n	$n \log n$	n^2	1	Non
Tri pair-impair	n	n^2	n^2	1	Oui

https://fr.wikipedia.org/wiki/Algorithme_de_tri



- Le tri par sélection est globalement peu efficace.
- Le tri par insertion est pertinent si la liste est presque triée (cas optimal)

TRI : BORNE INFÉRIEURE / $N \cdot \log(N)$