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

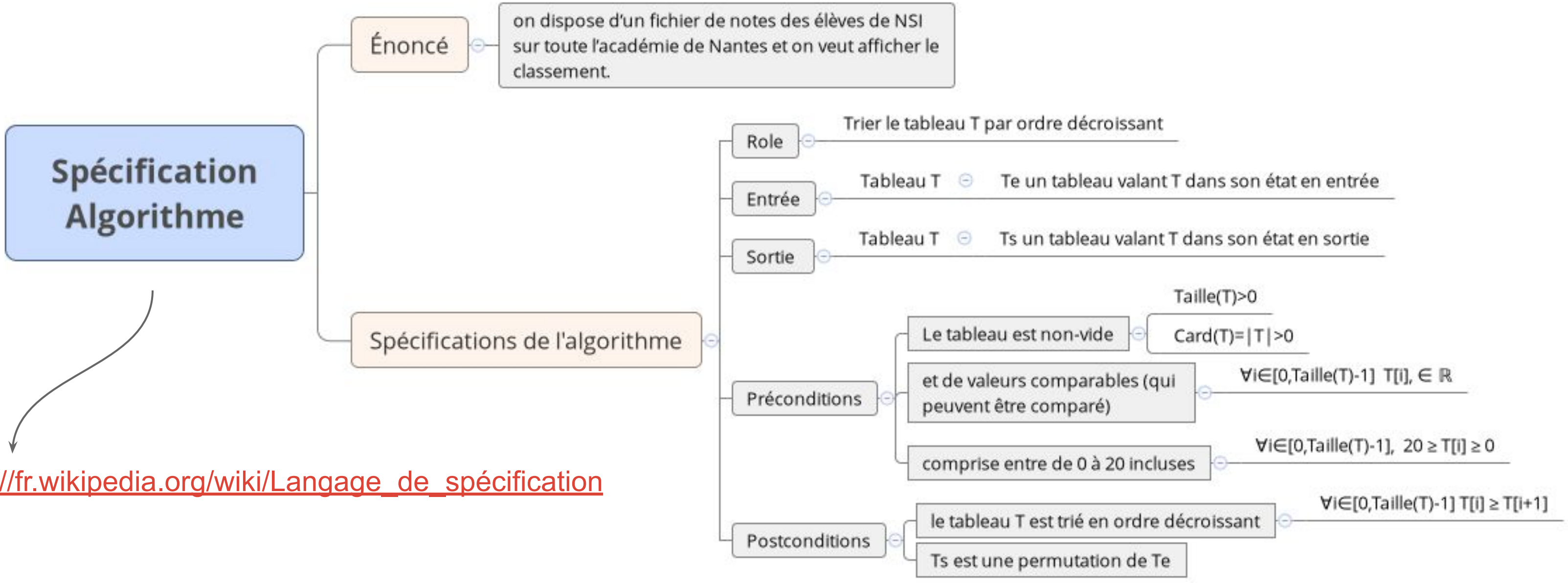
La notion de complexité en algorithmique est liée à la mesure des performances <u>a priori</u> des algorithmes :

- ➔ 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 ≠ 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.



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

Décrire le quoi, pas le comment.

Outils mathématiques

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

https://www.supinfo.com/cours/2ADS/chapitres/01-notion-complexite-algorithmique / http://villemin.gerard.free.fr/Wwwgvmm/Identite/SomPuiss.htm

Image : Conversion couleurs -> Niveaux de gris

```python
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))
        r = (r+v+b)//3
        v = (r+v+b)//3
        b = (r+v+b)//3
        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

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

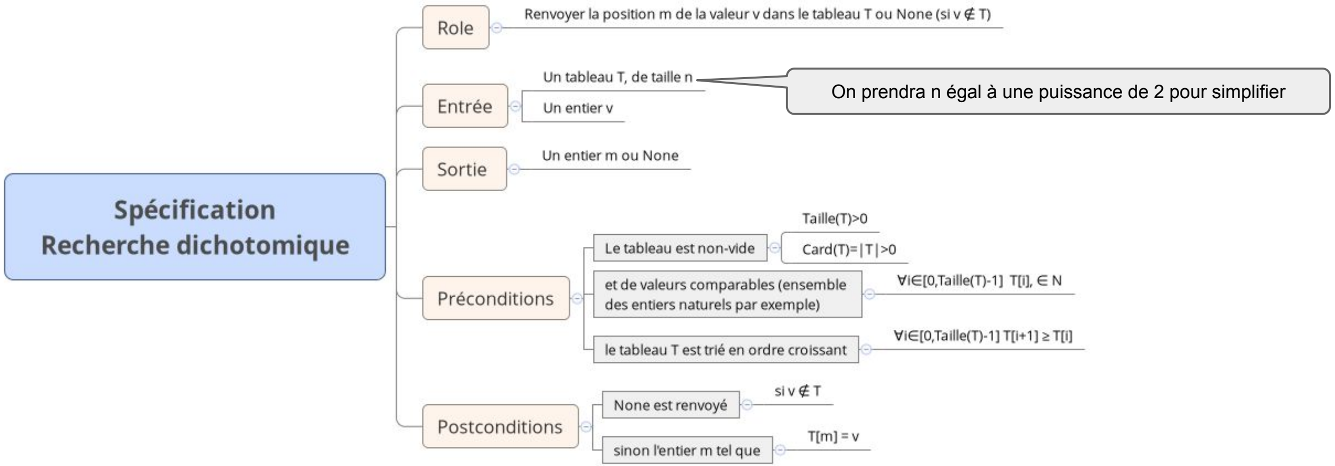
$$O_{affectation}(n) = 6.largeur. \sum_{y=0}^{hauteur-1} 1 = 6.largeur\ x\ hauteur = 6.n$$

➔ 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)

➔ On fait abstraction du coup de la fonction img.putpixel

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

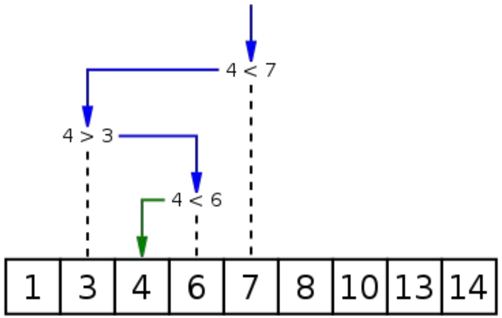
Approche naïve :

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) :

$$\frac{n}{2^k}=1 \rightarrow k=\log_2(n) \qquad O(n)=\sum_{0}^{k} A = A.(k+1) = A.\log_2(n)+A$$

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) !!!

```python
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)
```

```
>>> %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]
```

➔ 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 trié en ordre croissant.*

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
```

$$\sum_{i=1}^{n} \dots$$

$$\sum_{j=i+1}^{n} \dots$$

1 comparaison
(ligne 04)

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

$$\sum_{i=1}^{n} \dots$$

Pire cas !!!

$$\sum_{j=i+1}^{n} \dots$$

3 affectations
(ligne 05,06,07)

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

Tri sélection / Complexité (comparaison)

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

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

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

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

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

Tri_Sélection.py

```python
from random import *

for n in range (10,101,10):
    T=[randint(1,100) for _ in range(n)]

    nb_comparaison=0

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


    print("n=",n," => nb_comparaison=",nb_comparaison)
```

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
```

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 Insertion / Ordre croissant

```
Début Tri insertion
    pour i de 2 à n faire
        j=i
        memoire=Tab[i]
        tant que j>1 et memoire<Tab[j-1]
            Tampon=Tab[j-1]
            Tab[j-1] = Tab[j]
            Tab[j] = Tampon
            j=j-1
        fin_tant_que
        Tabl[j]= memoire
    fin_pour
fin Tri Insertion
```

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

Meilleur cas :

Tab[i] < Tab[i-1] toujours faux, on s'épargne les 4 affectations du bloc while. Il n'y a qu'une affectation 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)

- ➔ 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 trié en ordre décroissant
- ➔ Meilleur cas : La liste est déjà triée dans l'ordre croissant

Tri Insertion / Ordre croissant

```
Début Tri insertion
    pour i de 2 à n faire
        j=i
        memoire=Tab[i]
        tant que j>1 et memoire<Tab[j-1]
            Tampon=Tab[j-1]
            Tab[j-1] = Tab[j]
            Tab[j] = Tampon
            j=j-1
        fin_tant_que
        Tabl[j]= memoire
    fin_pour
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

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

$$\boxed{O_{comparaisonMC}(n)=2.n-2}$$

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

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

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

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

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

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

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	**n²**	**n²**
Affectation	**n²**	**0**

Tri par sélection

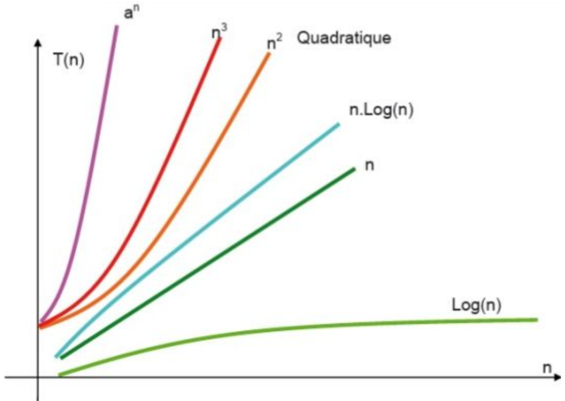
	Pire cas	Meilleur cas
Comparaison	**n²**	**n**
Affectation	**n²**	**n**

Tri par insertion

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

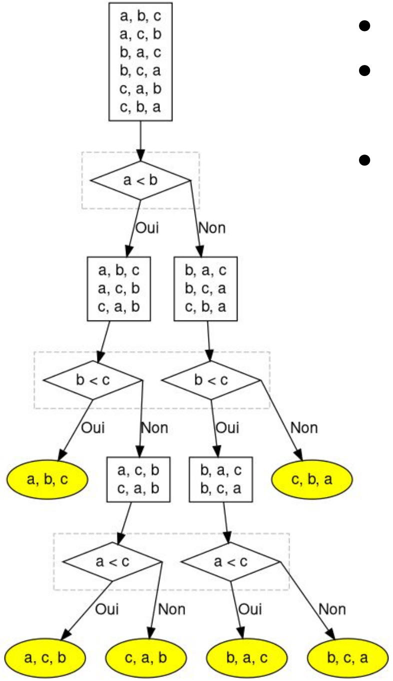
Tableau comparatif des tris procédant par comparaisons

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 Sedgewick : $\log n$ 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



- ➔ 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.log(n)



- Il y a n! possibilités de solutions
- A chaque comparaison, le nombre de solution est divisé par 2 (en moyenne),
- Quand le nombre de choix est égale à 1, l'algorithme termine en ayant effectué k comparaisons.

$\frac{n!}{2^k}=1$ => $2^k=n!$ => $k=\log_2(n!)$ théorème de Stirling $\log_2(n!)\simeq n\log_2(n)$ pour n → ∞

On retiendra : L'utilisation de tris basés sur des comparaisons ne peuvent pas avoir une complexité temporelle moyenne inférieure à **n.log$_2$(n)**

tri sans comparaison

https://haltode.fr/algo/tri/tri_denombrement.html