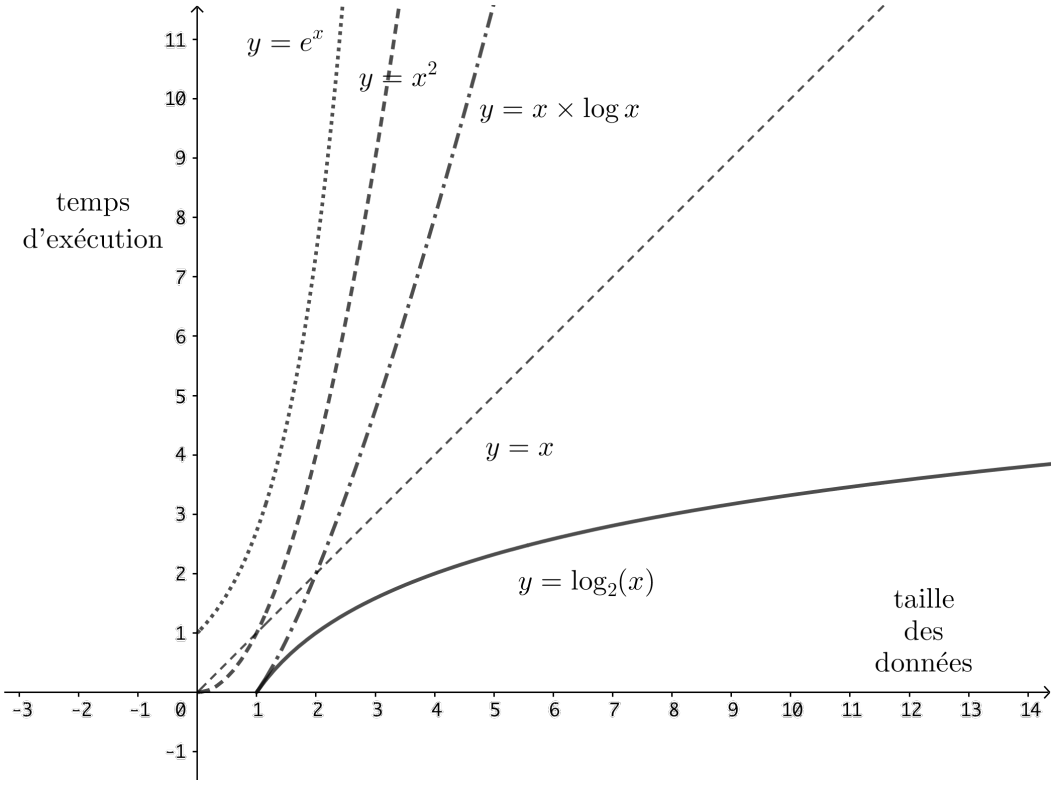
Les Algorithmes Du Programme De Première

I. Les différentes complexités

Au niveau lycée, on s'intéresse presque uniquement à la complexité des algorithmes en temps. Il existe aussi un e complexité en espace mémoire. La complexité en temps est notée $O(\textit{fonction})$, où *fonction* est réduite « au plus simple », en prenant uniquement la partie à la croissance la plus rapide, et en supprimant éventuellement les constantes multiplicatives.
Les complexités les plus fréquentes sont :

- Complexité constante $O(1)$: le temps d'exécution est constant quelle que soit la taille des données.
- Complexité logarithmique $O(\log n)$ ou $O(\ln n)$: schématiquement, si on multiplie la taille des données par 2 , alors on augment le nombre d'opérations de 1
- Complexité linéaire $O(n)$: si on augmente la taille des données de n, on augmente le temps de traitement de n
- Complexité log-linéaire ou quasi-linéaire $O(n\log n)$: augmente un peu plus vite que linéaire, mais beaucoup moins que quadratique
- Complexité quadratique $O(n^2)$: si on multiplie la taille des données par 2 (respectivement 3), on multiplie le temps par 4 (respectivement 9).
- Complexité cubique $O(n^3)$: si on multiplie la taille des données par 2 (respectivement 3), on multiplie le temps par 8 (respectivement 27).
- Complexité exponentielle $O(2^n)$ ou $O(\mathrm{e}^n)$: algorithmes très lents



© Hélène Carles – Lycée Agropolis – Montpellier, et Frédéric Mandon – Lycée jean-Jaurès – Saint-Clément-de-Rivière

Il existe d'autres ordres de grandeur. Dans la pratique un algorithme de complexité quadratique est déjà assez lent, mais il est fréquent que l'on ne dispose pas d'algorithmes rapides pour de nombreux problèmes.

II. Parcours séquentiel d'un tableau

Reprendre et compléter chez vous les paragraphes suivants à partir du notebook http://www.maths-info-lycee.fr/notebooks/tnsi_00_epreuve_pratique_revisions.ipynb

1. Écrire les algorithmes suivants en langage naturel
   (a) `rechercheElement(tab,element)` renvoie `True` si `element` appartient au tableau, `False` sinon (`tab` est un tableau de longueur n d'élements de type quelconque)

   (b) `recherchemax (tab)` renvoie le maximum des valeurs du tableau (`tab` est un tableau d'entiers de longueur n)

   (c) `calculSomme(tab)` renvoie la somme des entiers contenus dans le tableau de longueur n

2. Donner la complexité en temps de chacun des algorithmes.

III. Les algorithmes de tri par insertion et par sélection

1. **Par insertion**
   (a) En vous aidant du programme Python ci-dessous, expliquer le principe de l'algorithme de tri par insertion.

   Vous pourrez compléter le tableau (papier) ci-après, donnant toutes les étapes intermédiaires, en prenant comme exemple `t = [13 , 45 , 20]`

```
def tri_insertion(tab):
    """
    Donnée : prend un tableau tab de longueur n
    Résultat : renvoie le tableau trié par ordre croissant
    """
    for i in range(1 , len(tab)):                    #L1
        a_inserer = tab[i]                           #L2
        j = i                                        #L3
        while j > 0 and tab[j - 1] > a_inserer:      #L4
            tab[j] = tab[j - 1]                      #L5
            j = j - 1                                #L6
        tab[j] = a_inserer                           #L7
    return tab
```


© Hélène Carles – Lycée Agropolis – Montpellier, et Frédéric Mandon – Lycée jean-Jaurès – Saint-Clément-de-Rivière

Ligne du programme	`i`	`a_inserer[i]`	`j`	`test1`	`test2`	`tableau`
L0 (état initial du tableau)						
L1 (`for`)						
L2 (`a_inserer = tab[i]`)						
L3 (`j = i`)						
L4 (`while`)						
L5						

(b) Coût (*complexité temporelle*) et correction de l'algorithme

i. **Coût**

Dans le pire des cas (le tableau est rangé dans l'ordre décroissant), comptons le nombre de comparaisons.

Pour $i = 1$, le « tant que » est exécuté 1 fois

Pour $i = 2$, le « tant que » est exécuté 2 fois

Etc. jusqu'à $i = n-1$, où le « tant que » est exécuté $n-1$ fois.

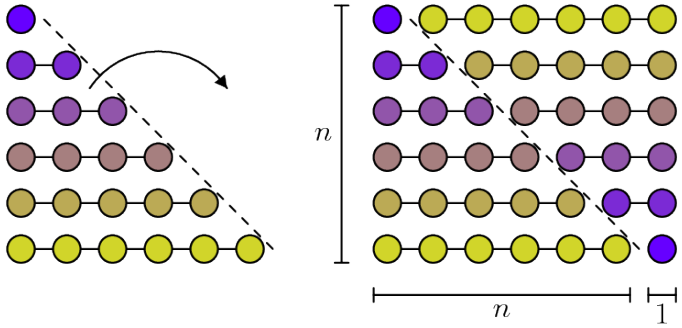
donc (1+2+...n-1) fois (deux tests et deux affectations)+ 3(n-1) affectations

Dans le cas où nous avons un tableau à trier qui contient n éléments, nous aurons :

$1+2+\ldots+\left(n-3\right)+\left(n-2\right)+\left(n-1\right)=\frac{n\left(n-1\right)}{2}=\frac{1}{2}n^2-\frac{1}{2}n$ comparaisons : la complexité est en $O\left(n^2\right)$, quadratique.

Pour expliquer le calcul de la somme, un dessin suffit :

© Hélène Carles – Lycée Agropolis – Montpellier, et Frédéric Mandon – Lycée jean-Jaurès – Saint-Clément-de-Rivière



On peut préciser davantage, mais cela ne changera par le résultat final.

Dans les « tant que » on fait deux tests et deux affectations, soit $4\times\frac{n(n-1)}{2}=2n(n-1)$ opérations, et on dans le « pour » il y a trois affectations, soit $3(n-1)$ opérations.

D'où au total $2n(n-1)+3(n-1)=2n^2+n-3$ opérations effectuées. Comme on ne prend que le terme de plus degré, sans tenir compte des constantes, on arrive au même résultat : complexité en $O(n^2)$.

ii. **Correction** (on examine la terminaison et correction partielle)
Terminaison de l'algorithme (l'algorithme se termine) *Compléter :*

Correction partielle (l'algorithme fait ce qu'il est supposé faire) **:** à faire sur le modèle de l'algorithme par sélection.

2. **Par sélection**

Le principe :
On dispose d'une liste de n éléments.
On cherche le plus petit
On le place en 1ère position en échangeant les deux éléments
On recommence avec les éléments restants.

(a) Compléter ci-dessous l'algorithme écrit en langage naturel :

```
pour i allant de 0 à n - 1 :                #L1
   indMin ← i                               #L2
   pour j allant de …… à n - 1 :             #L3
      si tableau[j] … tableau[indMin]:      #L4
         … ← …                              #L5
   échanger …                               #L6
```

(b) Écrire un programme Python (tests unitaires compris) réalisant ce tri.
Appeler `tri_selection` la fonction.

(c) Compléter le tableau ci-dessous donnant l'exécution pas à pas

© Hélène Carles – Lycée Agropolis – Montpellier, et Frédéric Mandon – Lycée jean-Jaurès – Saint-Clément-de-Rivière

Ligne du programme	i	iMin	j	test	tableau
L1	0				

(d) Coût et correction de l'algorithme

i. **Coût**

On parcourt les n cases du tableau pour la recherche de l'indice du plus petit élément et on fait un échange.

Puis on parcourt $n - 1$ cases du tableau pour la recherche de l'indice du plus petit élément et on fait un échange, etc.

On parcourt la dernière case et on fait un échange.

Donc au total nous aurons :

$(n-1)+(n-2)+(n-3)+\ldots+2+1=\frac{n(n-1)}{2}=\frac{1}{2}n^2-\frac{1}{2}n$ comparaisons. Les n échanges que l'on rajoute sont de coût négligeable.

La complexité est en $O(n^2)$, quadratique.

ii. **Terminaison de l'algorithme**

On a deux boucles imbriquées de longueur finie donc l'algorithme a une fin.

© Hélène Carles – Lycée Agropolis – Montpellier, et Frédéric Mandon – Lycée jean-Jaurès – Saint-Clément-de-Rivière

iii. **Correction partielle de l'algorithme**

Invariant de boucle :

Juste avant l'itération *i*, le tableau est trié (par ordre croissant) entre les indices ……

De plus tous les éléments restants sont supérieurs ou égaux à tab[*i* - 1].

Élements déjà triés	Éléments plus grands non triés
— — — — — — —	— — — — — — — — — — —

En sortie de la boucle : l'algorithme a ajouté en fin des éléments triés le plus petit de ceux restants, sachant que tous les éléments restants sont plus grands que les éléments déjà triés. On aura donc les éléments d'indice de 0 à *i* du tableau sont triés et les autres plus grands. Autrement dit, l'invariant de boucle est vrai en entrée et en sortie de boucle

En conclusion, on a démontré la correction de l'algorithme.

IV. Recherche dichotomique d'un élément dans une liste triée

1. Rappeler le principe de l'algorithme

2. Écrire un programme Python réalisant ce tri.

   Appeler `recherche_dichotomique` la fonction, ajouter les tests unitaires que vous trouverez utile de faire.

3. Complexité et terminaison
   a. Au pire, on va trouver l'élément lorsque l'intervalle de recherche est de longueur 0, c'est-à-dire au bout de *d* subdivisions, avec ……...

   C'est-à-dire $d = \log_2 n$.

   Ce qui nous donne un coût en performance en $O(\log n)$

   b. Pour démontrer la terminaison, nous allons utiliser un **variant de boucle.**

   Le variant de boucle *longueur* = ... décroît strictement à chaque itération.

   En effet à chaque étape on divise par deux, *longueur* devient $(nouvelle)$ *longueur* = ...

   Le variant *longueur* diminue strictement lors de chaque étape. Par ailleurs c'est un entier, donc il finira par atteindre 0 (si *valeur* n'est pas trouvée avant) : le programme se termine

© Hélène Carles – Lycée Agropolis – Montpellier, et Frédéric Mandon – Lycée jean-Jaurès – Saint-Clément-de-Rivière

V. les algorithmes gloutons (revus dans le TD sur la programmation objet)

1. Le principe

   Un algorithme glouton fait à chaque étape un choix localement optimal dans le but d'obtenir à la fin un optimum global, sans jamais remettre en question un choix déjà fait.
2. Donner les étapes d'un algorithme glouton standard, éventuellement en utilisant un exemple

3. Rappeler l'utilisation de la fonction Python `sorted`. On peut aussi donner un exemple

4. Optimalité des algorithmes gloutons

   On dispose d'une clé USB pouvant stocker 128 Mo et des fichiers de taille 60 Mo, 50 Mo, 100 Mo et 10 Mo. On souhaite utiliser au maximum l'espace disponible sur la clé.
   Un algorithme glouton ayant trié les fichiers par taille décroissante choisira 100Mo puis 10Mo.
   L'algorithme est-il optimal ?

VI. Algorithme des k plus proches voisins (k-NN)

Reprendre le notebook Méthode k-nn sur la page http://www.maths-info-lycee.fr/nsi_1ere.html, avec les fichiers de données.
On dispose d'un ensemble de n points qui sont « classifiés », le problème consiste à classifier un nouveau point M.
Le principe

On détermine les k plus proches points de M
On détermine la classe majoritaire (en cas d'égalité n'importe laquelle)
On l'attribue à M.

Écrire un programme Python réalisant ceci avec

Entrées : un couple de coordonnées M, une liste contenant les coordonnées et classe des points du jeu de données et *k* entier non nul.
Résultat : la classe du point M

1. Calculer toutes les distances de M avec le jeu de données
2. Classer pas distances croissantes
3. Retenir à partir du jeu de données les *k* plus proches voisins
4. Prendre les valeurs retenues (dans l'étape 3) :
   compter le nombre d'occurrences de chaque classe
5. Attribuer à M la classe la plus fréquente

© Hélène Carles – Lycée Agropolis – Montpellier, et Frédéric Mandon – Lycée jean-Jaurès – Saint-Clément-de-Rivière