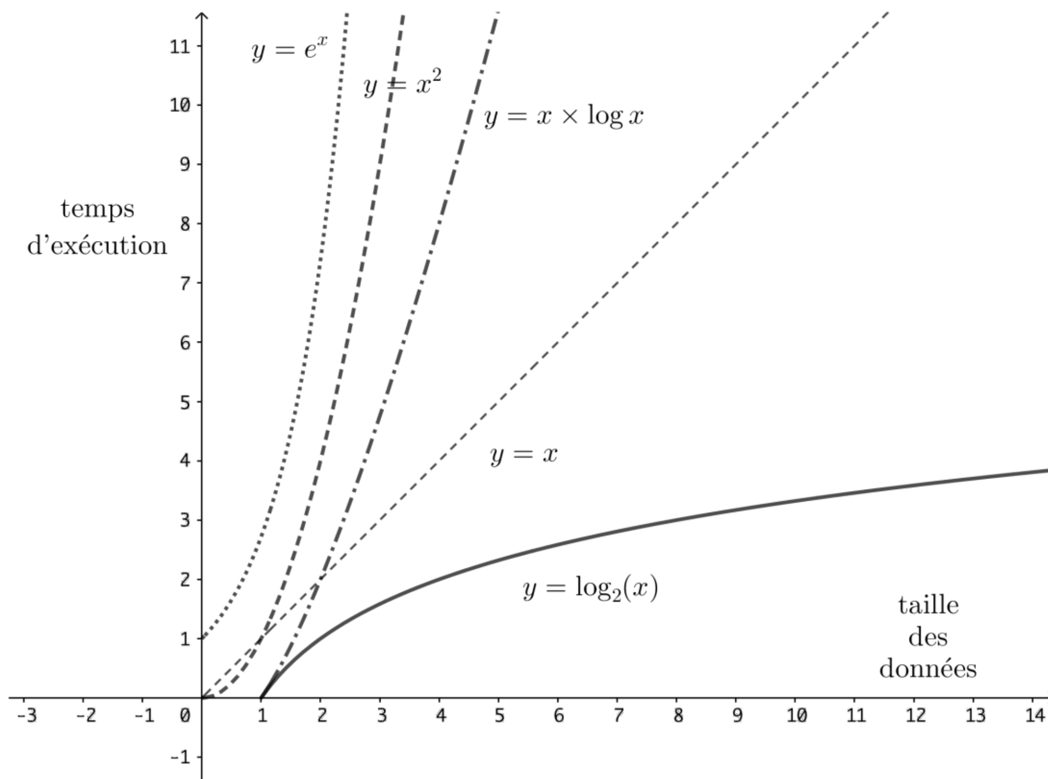
Révisions – Algorithmes

I – Les différentes complexités

Au niveau lycée, on s'intéresse presque uniquement à la complexité des algorithmes en temps. Il existe aussi une complexité en espace mémoire. La complexité en temps est notée $O(fonction)$, 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 augmente 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(nlogn) : 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 $O(e^n)$: algorithmes très lents



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

1. Ecrire 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'éléments 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 - Algorithmes de tri 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 """
1     for i in range(1 , len(tab)):
2         a_inserer = tab[i]
3         j = i
4         while j > 0 and tab[j - 1] > a_inserer:
5             tab[j] = tab[j - 1]
6             j = j - 1
7         tab[j] = a_inserer
8     return tab
```

	i	a_inserer[i]	j	test1	test2	tableau
L1						
L2						
L3						
L4						

(b) Coût (*complexité temporelle*)

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

1. Rappeler le principe de l'algorithme

2. Écrire un programme Python réalisant cette recherche.

   Appeler `rechercheParDichotomie` la fonction qui effectue une recherche dichotomique d'un élément dans un tableau trié et renvoi l'indice de l'élément recherché ou `None` si cet élément n'est pas dans le tableau.

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

V – les algorithmes gloutons

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 ?