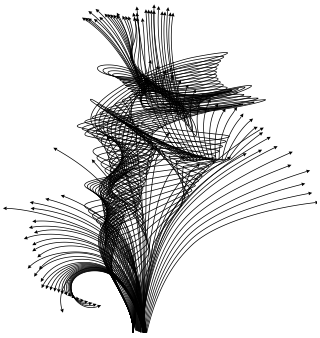
Complexité des algorithmes



En informatique dans le domaine du traitement automatique de données, la question de la performance des programmes est centrale. Si un problème donné est traité rapidement par un programme alors la solution trouvée est performante. Dans le cas contraire, se pose la question de trouver une solution plus efficace afin de raccourcir le temps de traitement.
De manière générale, le traitement d'un certain nombre de données requiert un temps d'exécution lié à ce volume de données.

1. Notion de complexité

Observation du lien entre temps de calcul et nombre d'opérations

Les instructions d'un programme s'exécutent à très grande vitesse, si bien que le résultat semble instantané. Cependant si le nombre d'instructions devient très grand, le temps de traitement peut devenir observable et handicapant.

Q1. Ecrire sur un IDE Python les programmes suivants mettant en œuvre une ou deux boucles for.

Programme 1	*Programme 2*	*Programme 3*

`a = 0`
`for _ in range(n):`
`  a = a + 1`
	`a = 0`
	`for _ in range(n):`
	`  a = a + 1`
	`for _ in range(n):`
	`  a = a + 1`
		`a = 0`
		`for _ in range(n):`
		`  for _ in range(n):`
		`  a = a + 1`

Q2. Exécuter ces programmes pour les valeurs de n suivantes et **estimer** à chaque fois le temps d'exécution (imperceptible, proche d'une seconde, proche de 2 secondes ...)

n	Programme 1	Programme 2	Programme 3
10			
1000			
10 000			

Q3. Justifier que le temps d'exécution de ces trois programmes est différent pour une valeur de n identique.

Amélioration des performances d'un algorithmes

Les deux fonctions effectuent le même même calcul mais de deux façons différentes. Elles prennent en paramètres deux tableaux de nombres et renvoient un nombre.

```
def somme_produits(t1, t2):
    somme = 0
    for a in t1:
        for b in t2:
            somme = somme + a * b
    return somme
```

```
def produit_sommes(t1, t2):
    somme1 = 0
    somme2 = 0
    for a in t1:
        somme1 = somme1 + a
    for b in t2:
        somme2 = somme2 + b
    return somme1 * somme2
```

Dans le cas de deux tableaux de trois éléments a, b, c tels que :

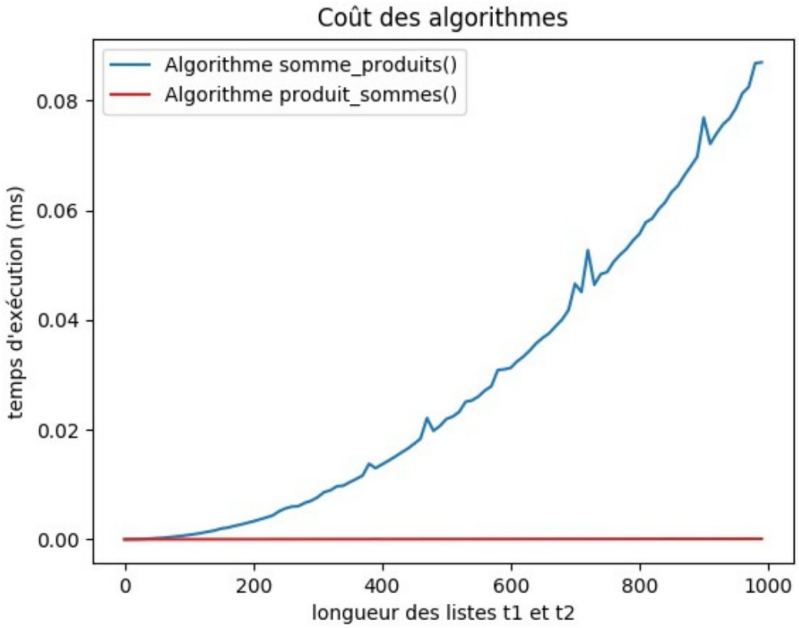
`t1 = [a1, b1, c1]` et `t2 = [a2, b2, c2]` ;

la fonction `somme_produits(t1, t2)` retournera alors la valeur :

```
a1*a2 + a1*b2 + a1*c2 + b1*a2 + b1*b2 + b1*c2 + c1*a2 + c1*b2 + c1*c2
```

Q4. Déterminer l'expression littérale renvoyée par la fonction produit_sommes(t1,t2). **Démontrer** en factorisant l'expression ci-dessus que les résultats renvoyés par les deux fonctions sont équivalentes.

L'évolution du coût en temps (temps de calcul) de ces deux algorithmes en fonction de la longueur des tableaux t1 et t2 est le suivant :

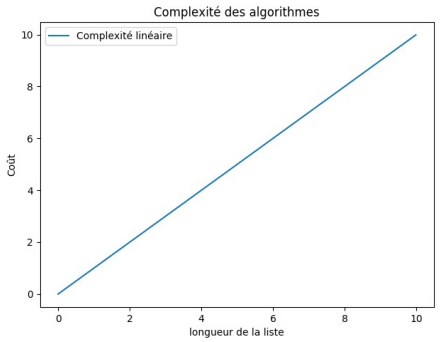


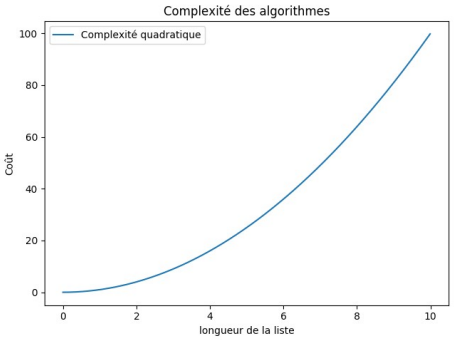
Q5. Indiquer l'algorithme le plus performant puis à partir des constations faîtes aux questions Q1 à Q3 **justifier** ces écarts de performances.

2.Complexité en temps

La complexité en temps d'un algorithme renseigne sur l'ordre de grandeur du nombre d'opérations élémentaires à exécuter.
Dans le cas d'opérations sur des tableaux, cette complexité dépendra du nombre d'éléments n de ce tableau. La complexité permet d'estimer à **la louche** l'évolution du coût en temps d'un algorithme lorsque n augmentera. Cette complexité s'exprime avec la notation $\mathcal{O}$
Les deux complexités les plus simples à identifier sont :

Perfor-mance	Type	Nota-tion	Structure algorithmique	Allure du coup

− ↓ +	**Complexité linéaire**	$\mathcal{O}(n)$	Une boucle for itérant le nombre d'éléments n de la liste :
			`for i in range(n) :`
			`  traitement`
				

	Complexité quadratique	$\mathcal{O}(n^2)$	Deux boucles for imbriquées itérant tout ou partie du nombre d'éléments n de la liste :
			`for i in range(n) :`
			`  traitement1`
			`  for j in range(n) :`
			`  traitement2`
			ou
			`for i in range(n) :`
			`  traitement1`
			`  for j in range(i) :`
			`  traitement2`
				

QCM d'application

1. On considère la fonction Python suivante, qui prend en argument une liste L et renvoie le maximum des éléments de la liste :

```
def rechercheMaximum(L):
    max = L[0]
    for i in range(len(L)):
        if L[i] > max:
            max = L[i]
    return max
```

On note *n* la taille de la liste. Quelle est la complexité en nombre d'opérations de l'algorithme ?

Réponses :

A- constante, c'est-à-dire ne dépend pas de *n*
B- linéaire, c'est-à-dire de l'ordre de *n*
C- quadratique, c'est-à-dire de l'ordre de n^2

2. Combien de temps mettra cet algorithme pour trouver la valeur maximale dans une liste deux fois plus longue (taille *2n*) ?

Réponses :

A- Le même temps que sur la liste de taille n si le maximum est dans la première moitié de la liste.

B- On a ajouté n valeurs, l'algorithme mettra donc n fois plus de temps que sur la liste de taille n.

C- Le temps sera simplement doublé par rapport au temps mis sur la liste de taille n.

D- On ne peut pas savoir, tout dépend de l'endroit où est le maximum.

3. Complexité des algorithmes de tri par insertion et de tri par sélection

Ces notions de complexité sont des critères de choix importants notamment lors de l'utilisation d'algorithmes de tri puisqu'ils peuvent être amenés à classer un très grand nombre de données.
Pour une liste donnée, un algorithme peut être meilleur qu'un autre mais avec une autre liste, ce peut être le contraire. Pour illustrer ce phénomène, nous allons comparer le coût (performance en temps) de l'algorithme de tri par insertion à celui de l'algorithme de tri par sélection dans le cas d'une liste aléatoire puis dans le cas d'une liste triée.

Performance des algorithmes de tri

L'évaluation des performances des différents algorithmes de tri peut facilement être testé en ligne sur :

http://lwh.free.fr/pages/algo/tri/comparaison_tri.html

Q6. Copier dans le tableau suivant la courbe de coût des algorithmes de tri par insertion et sélection dans les cas de données aléatoires et de données déjà triées. **En déduire** pour chaque cas la complexité de l'algorithme

	Tri par insertion		**Tri par sélection**	
Coût (cas de données aléatoires)				
Complexité (entourer la bonne réponse)	Complexité linéaire $\mathcal{O}(n)$	Complexité quadratique $\mathcal{O}(n^2)$	Complexité linéaire $\mathcal{O}(n)$	Complexité quadratique $\mathcal{O}(n^2)$
Coût (cas de données triées, meilleur des cas)				
Complexité (entourer la bonne réponse)	Complexité linéaire $\mathcal{O}(n)$	Complexité quadratique $\mathcal{O}(n^2)$	Complexité linéaire $\mathcal{O}(n)$	Complexité quadratique $\mathcal{O}(n^2)$

4.Justification des performances

Rappel des algorithmes de tri

Tri par insertion

```
def tri_insertion(t):
  '''trie le tableau t dans ordre crois'''
  for i in range(1, len(t)) :
    ind_val = i
    while ind_val > 0 and t[ind_val-1] >
val :
      t[ind_val] = t[ind_val-1]
      ind_val = ind_val - 1
    t[indice_val] = val
```

Tri par sélection

```
def tri_par_selection(t):
  '''trie le tableau t dans ordre
crois.'''
  for deb_nonTrie in range(len(t)) :
    ind_valMin =  deb_nonTrie
    for i in range(deb_nonTrie+ 1
,len(t)):
      if t[i] < t[ind_valMin] :
        ind_valMin = i
    echange(t,deb_nonTrie,ind_valMin)
```

Q7. A la lecture des algorithmes de tri par insertion et tri par sélection, **justifier** à partir du tableau page 3/5 que ces algorithmes sont au pire des cas de complexité quadratique $\mathcal{O}(n^2)$

Q8. A partir de l'animation http://lwh.free.fr/pages/algo/tri/tri_insertion.html , **justifier** que dans le cas de données déjà triées, la complexité du tri par insertion devienne linéaire.