|  |  |  |
| --- | --- | --- |
| Numérique et Sciences Informatiques | | |
| 1h30 | **Coût des algorithmes** |  |
| **Objectif** : estimer le coût d'un algorithme | | |
| **Matériel**: feuille de papier et Python | | |

**Le coût algorithmique**

* Ouvrez **programme1.py**, testez le **tri par insertion** avec différentes tailles de L : 1000, 5000, 10 000 éléments. Que constatez-vous concernant les durées d'exécution ?
* D'après-vous, peut-on espérer obtenir une réponse avec la taille de L : 1 000 000 000 ?

On se rend compte que l’algorithme prend du temps pour se terminer et donc pour renvoyer la liste triée.

Dans le cas d'un algorithme permettant de trouver des mots de passe en force brut (= test de toutes les combinaisons possibles), on pourra compter en millions d'années pour avoir le résultat.

Il est possible d’estimer le temps nécessaire pour qu’un algorithme se **termine**, pour cela nous allons calculer son **coût**.

On appelle **coût** le nombre d’instructions élémentaires exécutées dans un algorithme.

À partir du moment où l’on connaît le nombre d’instructions exécutées, il suffit de connaître le temps que le processeur mettra pour effectuer une instruction et de multiplier ce temps par le nombre d’instructions.

Par exemple, dans l’algorithme suivant :

note ← demander à l'utilisateur de rentrer une note

**Si** note>=15 **faire**

afficher "Bravo"

**Fin si**

Nous avons :

* Une affectation.
* Une comparaison.
* Un affichage.

Le coût est donc de 3 car nous effectuons au maximum 3 opérations élémentaires.

Si une opération nécessite 1ns (nanoseconde) pour être exécutée alors ce programme sera exécuté en 3ns.

Prenons maintenant le cas d’un algorithme contenant une boucle :

L=[5,1,6,9,8,7,4,6,4,5,7,8,2,5,4,6,7]

**pour** i allant de 0 à taille de L - 1 **faire**

afficher la valeur L[i]

**fin pour**

* Quel est le coût de cet algorithme ?
* Quel est le coût de cet algorithme pour un tableau qui contient **n** éléments ?
* Quel est le coût de l'algorithme suivant :

**fonction** affichage(L):

**pour** i allant de 0 à taille de L - 1 **faire**

afficher la valeur L[i]

**fin pour**

L=[5,1,6,9,8,7,4,6,4,5,7,8,2,5,4,6,7]

affichage(L)

affichage(L)

affichage(L)

* Quel est le coût de l'algorithme suivant :

L=[5,1,6,9,8,7,4,6,4,5,7,8,2,5,4,6,7]

**Pour** i allant de 0 à taille de L - 1 **faire**

**Pour** j allant de 0 à taille de L - 1 **faire**

afficher la valeur L[i]\*L[j]

Nous allons maintenant comparer l’évolution du **coût** de différents algorithmes en fonction de la taille des données en entrée.

Pour cela nous utilisons l'outil Python **matplot** qui permet de tracer des courbes.

* Ouvrez **programme2.py**, exécutez-le puis associez les fonctions test1 et test2 aux courbes.
* Modifiez la taille de la liste (300 par exemple) avec des valeurs plus grandes, que constatez-vous pour la fonction test1 ?
* Ouvrez **programme3.py**, exécutez-le (taille = 15 puis 300) puis associez les fonctions test1, test2, test3, test4 et test5 aux courbes.

Pour simplifier les choses nous allons simplement indiquer l’ordre de grandeur du coût (=complexité), pour cela nous utiliserons la notation de **Landeau O** :

* **O(1)** pour les algorithmes ayants un coût constant (qui ne dépend pas de la taille des données en entrée).

C'est le cas de la fonction test1 : O(3) → O(1).

* **O(n)** pour les algorithmes ayants un coût linéaire.

C'est le cas des fonctions test2 : O(n), test3 : O(n+2) → O(n) et test5 : O(3n) → O(n).

Par exemple : l'algorithme de parcours d'une liste.

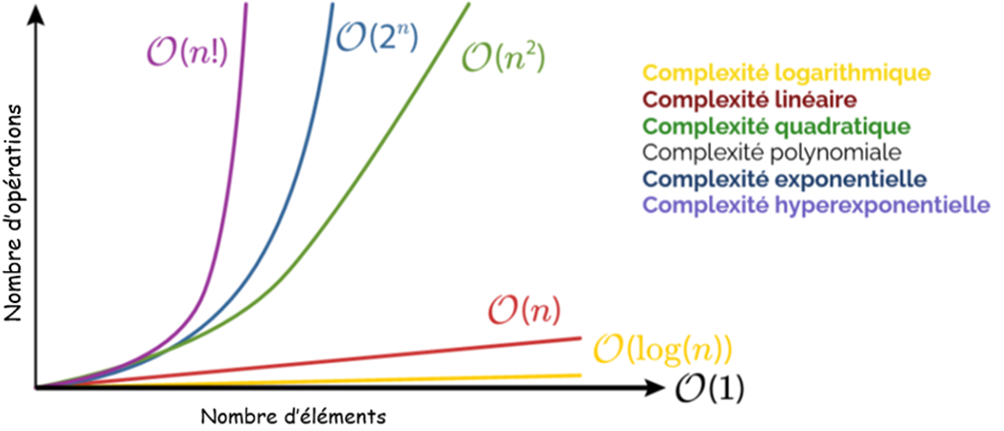
* **O(n2)** pour les algorithmes ayants un coût quadratique.

C'est le cas de la fonction test4.

Par exemple : les algorithmes de tri par sélection et par insertion.

* **O(log(n))** pour les algorithmes ayants un coût logarithmique.

par exemple : l'algorithme de recherche dichotomique : O(log2(n)) → O(log(n)).



L'algorithme DeepBlue, qui a battu Kasparov, testait tous les coups possibles et anticipait les coups à venir : coût exponentiel.

L'algorithme de tri quicksort à un coût logarithmique moyen en O(n.log(n)) mais un coût quadratique au pire en O(n²).

Rappel : l'algorithme de tri par sélection a un coût quadratique dans tous les cas en O(n²).

Rappel : l'algorithme de tri par insertion a un coût quadratique au pire cas en O(n²) mais un coût linéaire dans le meilleur cas en O(n).

Prenons maintenant l’exemple de l'algorithme qui recherche l'indice d'une valeur dans une liste quelconque.

L = [5,1,6,9,8,7,4,6,4,5,7,8,2,5,4,6,7]

val ← demander à l'utilisateur la valeur recherchée dans L

i ← 0

**Tant que** val≠L[i] **et** i<len(L)) **faire**

i ← i+1

**Fin tant que**

afficher : i "est l'indice de val"

* Quel est le coût de cet algorithme dans le meilleur des cas ?
* Quel est le coût de cet algorithme dans le pire des cas ?

On constate que le coût d'un algorithme dans le meilleur des cas peut être différent de son coût dans le pire des cas.

Connaitre les coûts des algorithmes dans le meilleur ou dans le pire des cas peut avoir un intérêt suivant les données à traitées.

Exemple pour l'algorithme de **tri par insertion** :

* Coût linéaire **O(n) dans le meilleur des cas** : lorsque la liste est déjà triée l'algorithme effectuera seulement n comparaisons.
* Coût quadratique **O(n2) dans le pire des cas** : lorsque la liste est triée à l'envers l'algorithme effectuera n x n comparaisons (n comparaisons pour savoir si un élément est mal placé et n autres comparaisons pour savoir où placer l'élément mal placé).
* Coût moyen : lorsque la liste est déjà en parti triée, dans ce cas l'algorithme du tri par insertion sera moins coûteux que celui par tri par sélection qui aura toujours un coût de O(n2).

**Conclusion**

Connaitre le **coût temporel** (= **complexité)** des algorithmes permet de faire des choix (ex : choix d’un algorithme de tri).

Il existe également la notion de **coût spatial** d'un algorithme pour évaluer l’espace nécessaire en mémoire pour le fonctionnement de celui-ci.

Pour améliorer le **coût temporel** de certains algorithmes on peut utiliser des supercalculateurs (simulations nucléaires, prévisions météorologiques, calcul des récompenses bitcoin ...).

Supercalculateur BULL conçu et fabriqué à Angers :

