TP : Traitement parallèle

Algorithme de tri rapide

Realise par : JOUAL Anouar
SADIQI Omar

Introduction

La classification des algorithmes de tri est très importante, car elle permet de choisir l'algorithme le plus adapté au problème traité, tout en tenant compte des contraintes imposées par celui-ci. Les principales caractéristiques qui permettent de différencier les algorithmes de tri, outre leur principe de fonctionnement, sont la complexité temporelle, la complexité spatiale et le caractère stable.

- La complexité temporelle (en moyenne ou dans le pire des cas) : mesure le nombre d'opérations élémentaires effectuées pour trier une collection d'éléments. C'est un critère majeur pour comparer les algorithmes de tri, puisque c'est une estimation directe du temps d'exécution de l'algorithme. Dans le cas des algorithmes de tri par comparaison, la complexité en temps est le plus souvent assimilable au nombre de comparaisons effectuées, la comparaison et l'échange éventuel de deux valeurs s'effectuant en temps constant.

- La complexité spatiale (en moyenne ou dans le pire des cas) : représente, quant à elle, la quantité de mémoire dont va avoir besoin l'algorithme pour s'exécuter. Celle-ci peut dépendre, comme le temps d'exécution, de la taille de l'entrée. Il est fréquent que les complexités spatiales en moyenne et dans le pire des cas soient identiques. C'est souvent implicitement le cas lorsqu'une complexité est donnée sans indication supplémentaire.

La complexité moyenne du tri rapide pour n éléments est proportionnelle à $n \log n$, ce qui est optimal pour un tri par comparaison, mais la complexité dans le pire des cas est quadratique. Malgré ce désavantage théorique, c'est en pratique un des tris les plus rapides, et donc un des plus utilisés. Le pire cas est en effet peu probable lorsque l'algorithme est correctement mis en œuvre et il est possible de s'en prémunir définitivement avec la variante Introsort.

Le tri rapide ne peut cependant pas tirer avantage du fait que l'entrée est déjà presque triée. Dans ce cas particulier, il est plus avantageux d'utiliser le tri par insertion ou l'algorithme Smoothsort.

L’algorithme de tri :

S'il s'agit de trier une liste simple « Liste », il n'est pas utile d'utiliser autre chose que Liste.sort() de Python qui est très très efficace.

Mais il arrive des cas où l'on veut trier selon des critères particuliers, et là, on a besoin d'une fonction de tri performante.

Voilà une fonction de tri basée sur le tri rapide de Hoare (quicksort) . Ce tri est effectivement très rapide.

La méthode consiste à placer un élément du tableau (appelé pivot) à sa place définitive, en permutant tous les éléments de telle sorte que tous ceux qui sont inférieurs au pivot soient à sa gauche et que tous ceux qui sont supérieurs au pivot soient à sa droite.

Cette opération s'appelle le partitionnement. Pour chacun des sous-tableaux, on définit un nouveau pivot et on répète l'opération de partitionnement. Ce processus est répété récursivement, jusqu'à ce que l'ensemble des éléments soit trié.

Concrètement, pour partitionner un sous-tableau :

- on place le pivot à la fin (arbitrairement), en l'échangeant avec le dernier élément du sous-tableau ;
- on place tous les éléments inférieurs au pivot en début du sous-tableau ;
- on place le pivot à la fin des éléments déplacés.

Résultats d’exécution de l’algorithme :

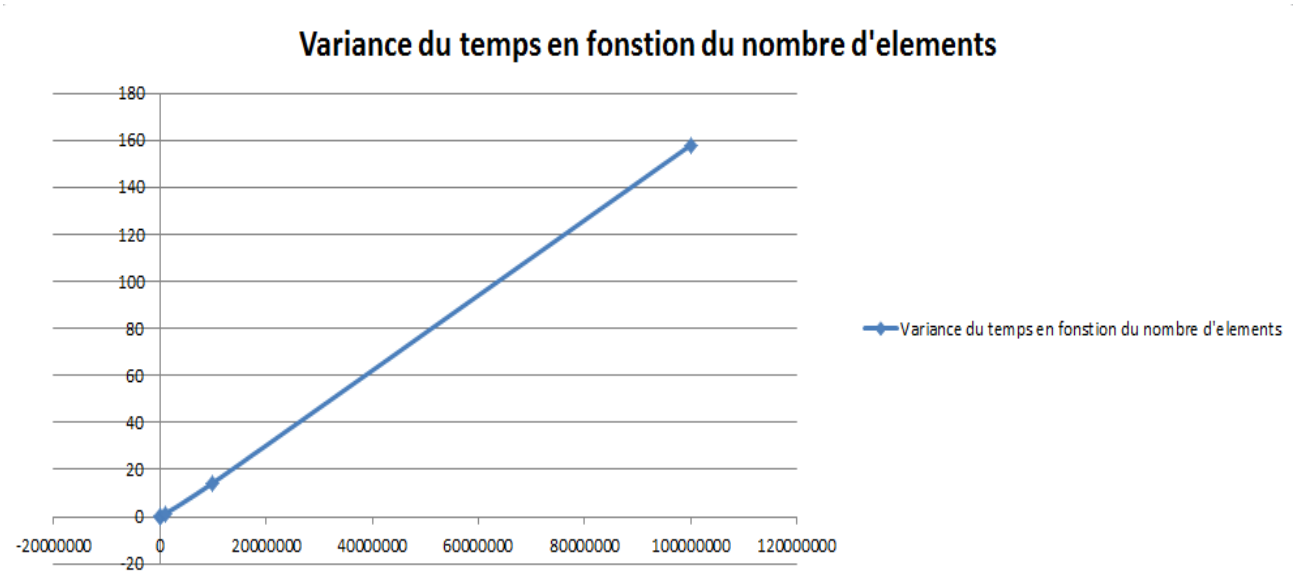
- ***Efficacité du tri:***

Pour des listes de 1000 éléments au hasard il est seulement un peu meilleur (-20%) que le tri par insertion avec recherche par dichotomie. Par contre, pour des listes plus importantes, il est plus rapide.

- **Tableau du resultats :**

Nombre d’éléments du liste	**1000**	**10000**	**100000**	**1000000**	**10000000**
Temps d’exécution	**0.010766**	**0.112045**	**1.287335**	**14.416152**	**157.981986**

- **Graphique de variance de temps d’execution en fonction du nombre d’éléments :**



Description du travail :

- Langage utilisé : python
- Logiciel de travail : pycharm
- Algorithme : tri par pivot
- Processeur : CPU 3.40 GHz

Une mise en œuvre naïve du tri rapide utilise un espace mémoire proportionnel à la taille du tableau dans le pire cas. Il est possible de limiter la quantité de mémoire à $\Theta(\log n)$ dans tous les cas en faisant le premier appel récursif sur la liste la plus petite. Le second appel récursif, situé à la fin de la procédure, est récursif terminal.

Conclusion :

Beaucoup d'algorithmes existent, mais certains sont bien plus utilisés que d'autres en pratique. Le tri par insertion est souvent plébiscité pour des données de petite taille, tandis que des algorithmes asymptotiquement efficaces, comme le tri fusion, le tri par tas ou quicksort, seront utilisés pour des données de plus grande taille.

Il existe des implémentations finement optimisées, qui sont souvent des algorithmes hybrides. Timsort utilise ainsi à la fois les méthodes de tri fusion et de tri par insertion, et est utilisé entre autres par Android, Javaet Python ; Introsort, qui combine quicksort et tri par tas, est utilisé dans certaines implémentations du tri C++.

La comparaison empirique d'algorithmes n'est pas aisée dans la mesure ou beaucoup de paramètres entrent en compte : taille de données, ordre des données, matériel utilisé, taille de la mémoire vive, etc. Par exemple, les essais effectués sur des données tirées aléatoirement ne représentent pas forcément très fidèlement les comportements obtenus avec des données réelles.

Annexe : « code du programme »

```
import random
import time

def Function_quicksort(Liste):
    def quicksort(Liste, a, b):
        pivot = Liste[(a+b)//2]
        i = a
        j = b
        while True:
            while Liste[i]<pivot:
                i+=1
            while Liste[j]>pivot:
                j-=1
            if i>j:
                break
            if i<j:
                Liste[i], Liste[j] = Liste[j], Liste[i]
            i+=1
            j-=1
        if a<j:
            quicksort(Liste,a,j)
        if i<b:
            quicksort(Liste,i,b)
    a=0
    b=len(Liste)-1
    quicksort(Liste,a,b)

drp=10000000
Liste=[]

for i in range(0,drp):
    Liste.append(random.randint(1,1000))

tps=time.clock()
Function_quicksort(Liste)

for i in range(1,drp):
    if Liste[i]<Liste[i-1]:
        print(" erreur  ",i, Liste[i])
for i in range(1,drp):
    print (Liste[i])

print ("temps d'execution = ",time.clock()-tps)
```