Projet - Algorithme avancée

Ibtissem Bouzidi

Contents

1 Introduction

Le tri est un élément de base sur lequel reposent de nombreux autres algorithmes. C'est lié à plusieurs idées passionnantes que vous verrez tout au long de votre carrière en programmation.Comprendre comment les algorithmes de tri en Python fonctionnent en arrière-plan ainsi leur complexité .

2 Les algorithmes de Tri

2.1 Tri par sélection

Le tri par sélection (ou tri par extraction) est un algorithme de tri par comparaison.

2.1.1 Algorithme

Algorithm 1 Tri_sélection(T,n)

```
Result: Tableau
Var i,j, min, aux : entier;
 T : tableau de taille n
 for i ← 1 to (n − 1) do
    min ← i
     for j ← 1 to n do
        if T[j] < T[min] then
            min ← j
        end
    end
    aux ← T[min]
     T[min] ← T[i]
     T[i] ← aux
end
```

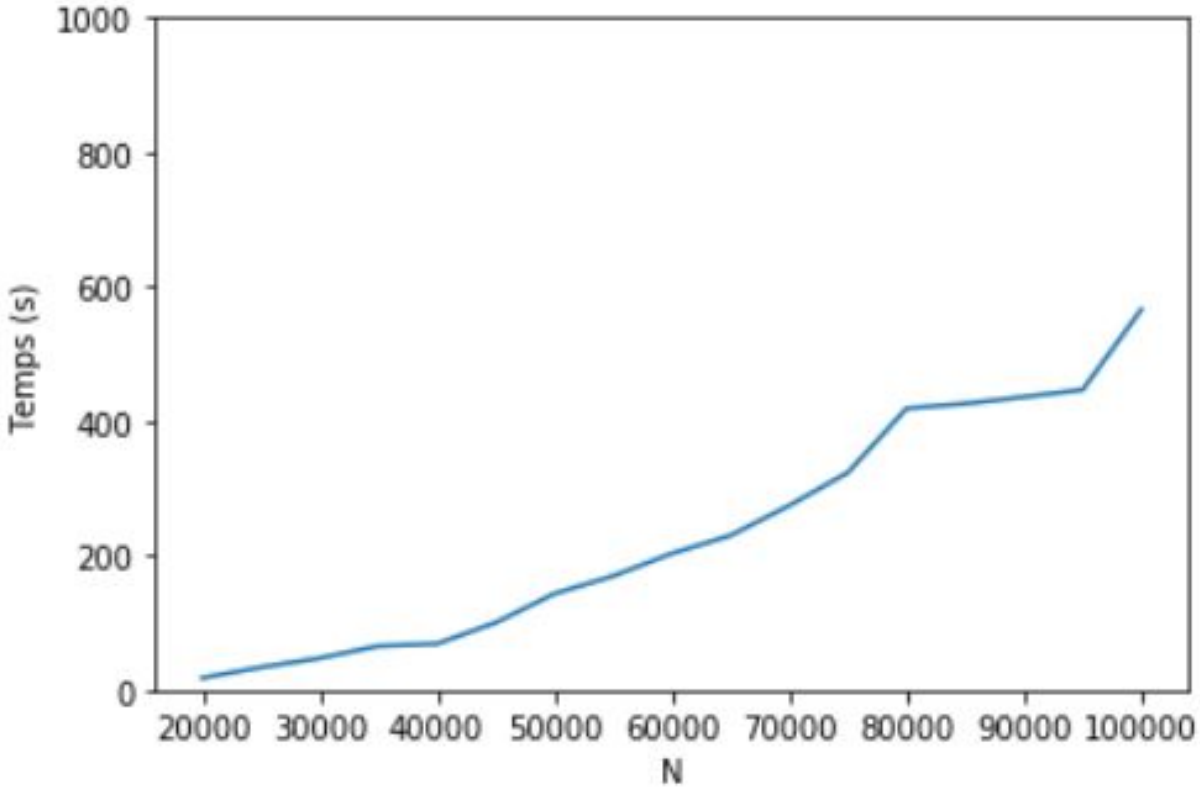
2.1.2 La complexité asymptotique théorique

Ici, nous avons une boucle imbriquée; la boucle externe parcourt tous les éléments, puis la boucle interne fait de même . Si la taille du tableau $n = 100$, il y aura 10000 exécutions . Au fur et à mesure que le nombre de n éléments augmente, le nombre d'étapes augmentera de façon exponentielle. C'est l'un des temps d'exécution les plus lents. Chaque boucle est un temps $O(n)$, donc $O(n) * O(n) = O(n)$.

2.1.3 Le tableau des mesures du temps

N	Temps d'execution
20 000	17.909039(s)
25 000	33.599156(s)
30 000	47.284122(s)
35 000	65.369982(s)
40 000	68.685982(s)
45 000	100.221794(s)
50 000	143.180081(s)
55 000	169.538738(s)
60 000	203.027320(s)
65 000	229.771123(s)
70 000	274.378331(s)
75 000	323.430547(s)
80 000	418.855480(s)
85 000	7,1(min)
90 000	7,25(min)
95 000	7,433(min)
100 000	9,433(min)

2.1.4 La courbe



la courbe correspondant à tri sélection

2.2 Trià bulle :

Il consiste à comparer répétitivement les éléments consécutifs d'un tableau, et à les permuter lorsqu'ils sont mal triés.

2.2.1 Algorithme:

Algorithm 2 Tri_bulle(T,n)

```
Result: Tableau
Var i,j, min, aux : entier;
 T : tableau de taille n
 for i ← (n − 1) to 1) do
 |  for j ← 0 to (i − 1) do
 |  |  if T[j + 1] < T[j] then
 |  |  |  aux ← T[min]
 |  |  |  T[min] ← T[i]
 |  |  |  T[i] ← aux
 |  |  |
 |  |  end
 |  end
end
```

2.2.2 La complexité asymptotique théorique

L'algorithme du TB se compose de deux boucle imbriquées dans lesquelles l'algorithme effectue $n-1$ comparaisons, puis $n-2$ comparaisons, et ainsi de suite jusqu'à ce que la comparaison finale soit effectuée. Cela revient à un total de $(n-1)+(n-2)+(n-3)+\ldots+2+1=\frac{n(n-1)}{2}$ comparaisons, qui peuvent également s'écrire n^2-n .

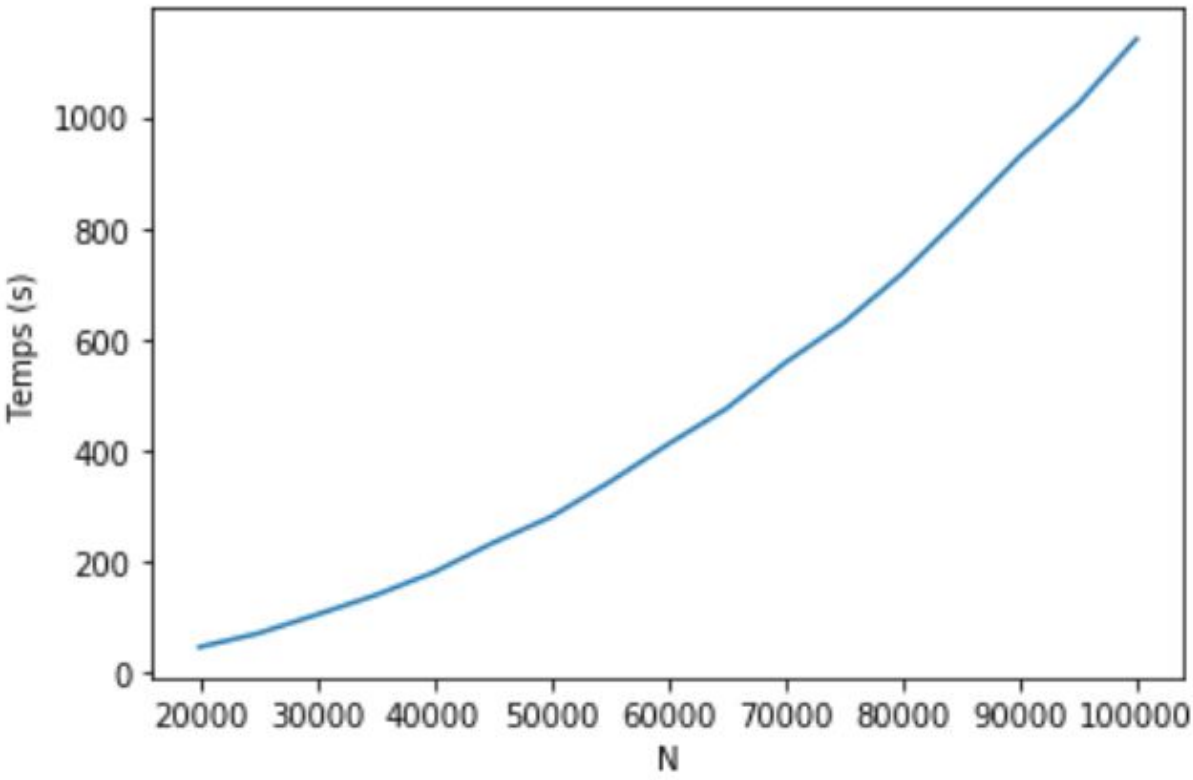
Vous avez appris plus tôt que Big O se concentre sur la croissance du runtime par rapport à la taille de l'entrée. Cela signifie que, pour transformer l'équation ci-dessus en complexité Big O de l'algorithme, vous devez supprimer les constantes car elles ne changent pas avec la taille d'entrée.

Cela simplifie la notation en n^2-n . Puisque n^2 croît beaucoup plus vite que n , ce dernier terme peut également être supprimé, laissant un tri à bulles avec une complexité moyenne et dans le pire des cas de $O(n^2)$.

2.2.3 Le tableau des mesures du temps

N	Temps d'execution
20 000	44.631405(s)
25 000	69.208176(s)
30 000	102.895946(s)
35 000	137.835043(s)
40 000	179.231089(s)
45 000	231.807946(s)
50 000	279.019998(s)
55 000	341.743010(s)
60 000	409.864490(s)
65 000	474.470823(s)
70 000	556.899191(s)
75 000	628.916660(s)
80 000	717.820203(s)
85 000	13,66(min)
90 000	15,46(min)
95 000	17,03(min)
100 000	19(min)

2.2.4 La courbe



2.3 Tri par insertion :

l'algorithme de tri par insertion est simple à implémenter et à comprendre. Mais contrairement au tri à bulles, il construit la liste triée un élément à la fois en comparant chaque élément avec le reste de la liste et en l'insérant dans sa position correcte.

2.3.1 Algorithme:

Algorithm 3 Tri_insertion(T,n)

Result: Tableau
Var i,j, x : entier;
T : tableau de taille n
for $i \leftarrow 1$ **to** $(n-1)$ **do**
  $x \leftarrow T[i+1]$
  $j \leftarrow i$
  while $(T[i] > x)$ *and* $(j > 0)$ **do**
    $T[i+1] \leftarrow T[j]$
    $j \leftarrow j-1$
  end
  $T[j+1] \leftarrow x$
end

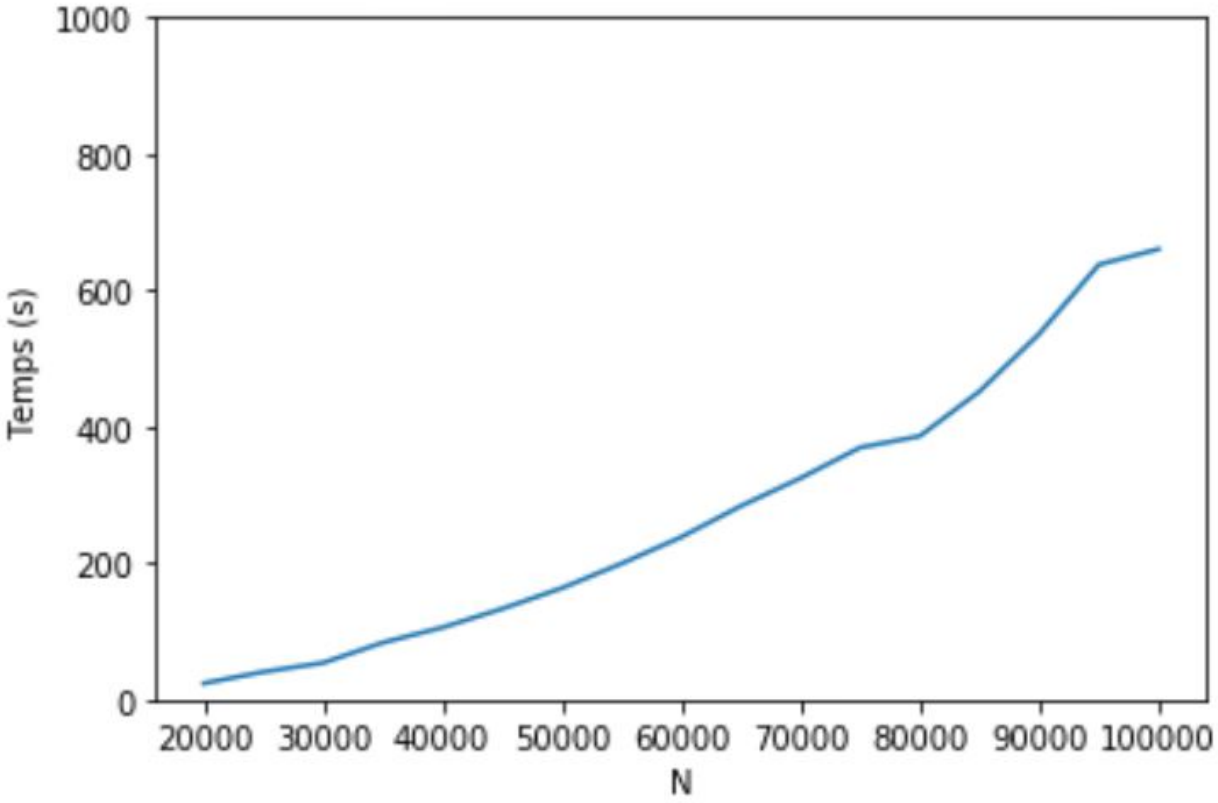
2.3.2 La complexité asymptotique théorique

l'algorithme de TI comporte quelques boucles imbriquées qui parcourent la liste. La boucle interne est assez efficace car elle ne parcourt la liste que jusqu'à ce qu'elle trouve la position correcte d'un élément. Cela dit, l'algorithme a toujours une complexité d'exécution $O(n^2)$ dans le cas pir et moyen.

2.3.3 Le tableau des mesures du temps

N	Temps d'execution
20 000	23.698377(s)
25 000	40.473397(s)
30 000	53.757612(s)
35 000	83.397519(s)
40 000	105.601291(s)
45 000	132.987960(s)
50 000	163.324201(s)
55 000	199.301080(s)
60 000	238.092566(s)
65 000	283.657871(s)
70 000	324.566172(s)
75 000	369.424136(s)
80 000	386.227379(s)
85 000	7,51(min)
90 000	8,93(min)
95 000	10,61(min)
100 000	11(min)

2.3.4 La courbe



2.4 Tri fusion

Le tri par fusion est un algorithme de tri très efficace. Il est basé sur l' approche diviser-pour-conquérir , une technique algorithmique puissante utilisée pour résoudre des problèmes complexes.

2.4.1 Algorithme

type :
Séquence= Structure
element : tableau [MAX] déntier
taille : entier
fin Structure

Algorithm 4 fusionner(S1,S2 : sequance var S : sequance)

Result: Tableau
Var i,j,k : entier;
$i \leftarrow 1$
$j \leftarrow 1$
$S.taille \leftarrow S1.taille + S2.taille$
repeat
  if $S1.element[i] < S2.element[j]$ **then**
    $S.element[k] \leftarrow S1.element[i]$
    $i \leftarrow i + 1$
    $k \leftarrow k + 1$
  else
    $S.element[k] \leftarrow S2.element[j]$
    $j \leftarrow j + 1$
    $k \leftarrow k + 1$
  end
until $(i > S1.taille)$ *ou* $(j > S2.tille)$;
while $(i < S1.taille)$ **do**
  $S.element[k] \leftarrow S1.element[i]$
  $i \leftarrow i + 1$
  $k \leftarrow k + 1$
end
while $(j < S2.taille)$ **do**
  $S.element[k] \leftarrow S2.element[j]$
  $j \leftarrow j + 1$
  $k \leftarrow k + 1$
end

Algorithm 5 diviser(S:sequece,var : S1, S2 :sequence)

Result: entier
Var i,j: entier;
for $i \leftarrow 1$ **to** $(S.taille\ div\ 2)$ **do**
  $S1.element[i] \leftarrow S.element[i]$
end
$S1.taille \leftarrow S.taille$ div 2
for $i \leftarrow (S.taille\ div\ 2) + 1$ **to** $(S.taille)$ **do**
  $S2.element[i] \leftarrow S.element[i]$
end
$S2.taille \leftarrow (S.taille - S1.taille)$

Algorithm 6 tri_fision(S : sequence)

```
Var S1 , S2 : sequence
if S.taille > 1 then
    diviser(S, S1, S2)
    tri_fision(S1)
    tri_fision(S2)
    fusionner(S2)
end
```

2.4.2 La complexité asymptotique théorique

Pour analyser la complexité du tri par fusion, vous pouvez examiner ses deux étapes séparément:

1. fusionner() a un temps d'exécution linéaire. Il reçoit deux séquence dont la longueur combinée est au plus n (la taille du séquence d'entrée d'origine), et il combine les deux séquence en regardant chaque élément au plus une fois. Cela conduit à une complexité d'exécution de $O(n)$

2. divise() le tableau d'entrée de manière récursive et appelle fusionner()chaque moitié. Étant donné que le tableau est divisé par deux jusqu'à ce qu'un seul élément reste, le nombre total d'opérations de réduction de moitié effectuées par cette fonction est de $log_2 n$. Puisqu'il fusionner()est appelé pour chaque moitié, nous obtenons un temps d'exécution total de $O(nlog_2 n)$.
   Il est intéressant de noter que $O(nlog_2 n)$ est la meilleure exécution possible dans le pire des cas pouvant être obtenue par un algorithme de tri.

2.4.2.1 Relation de récurrence Pour simplifier les choses, supposons que n est une puissance de 2, soit $n = 2^k$ pour certains k

- Le temps d'exécution d'un algorithme récursif peut être analysé à l'aide d'un **relation de récurrence**. Chaque étape de " division " produit deux sous-problèmes de taille $n/2$.

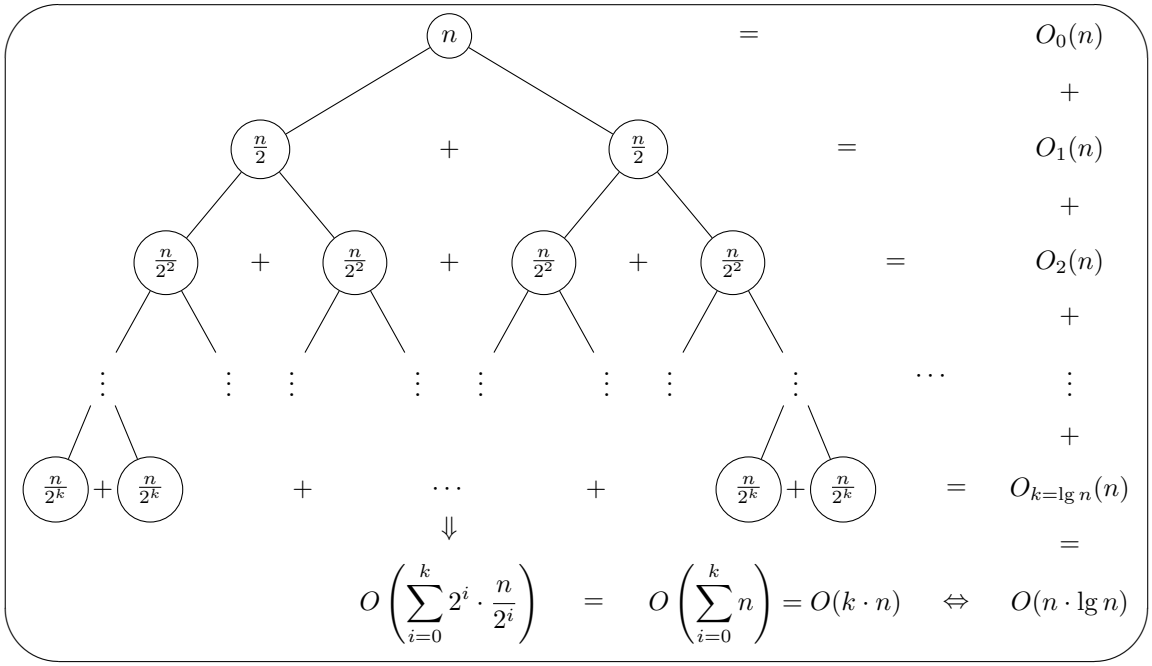
- Soit $T(n)$ le temps d'exécution le plus défavorable du tri de fusion sur un tableau de n éléments. Nous avons:

$$\begin{aligned} T(n) &= c_1 + T(n/2) + T(n/2) + c_2 n \\ &= 2T(n/2) + (c_1 + c_2 n) \end{aligned}$$

- Simplifie, $T(n) = 2T(n/2) + \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{If } n = 1 \\ T(\lceil \frac{n}{2} \rceil) + T(\lfloor \frac{n}{2} \rfloor) + \Theta(n) & \text{If } n > 1 \end{cases}$$

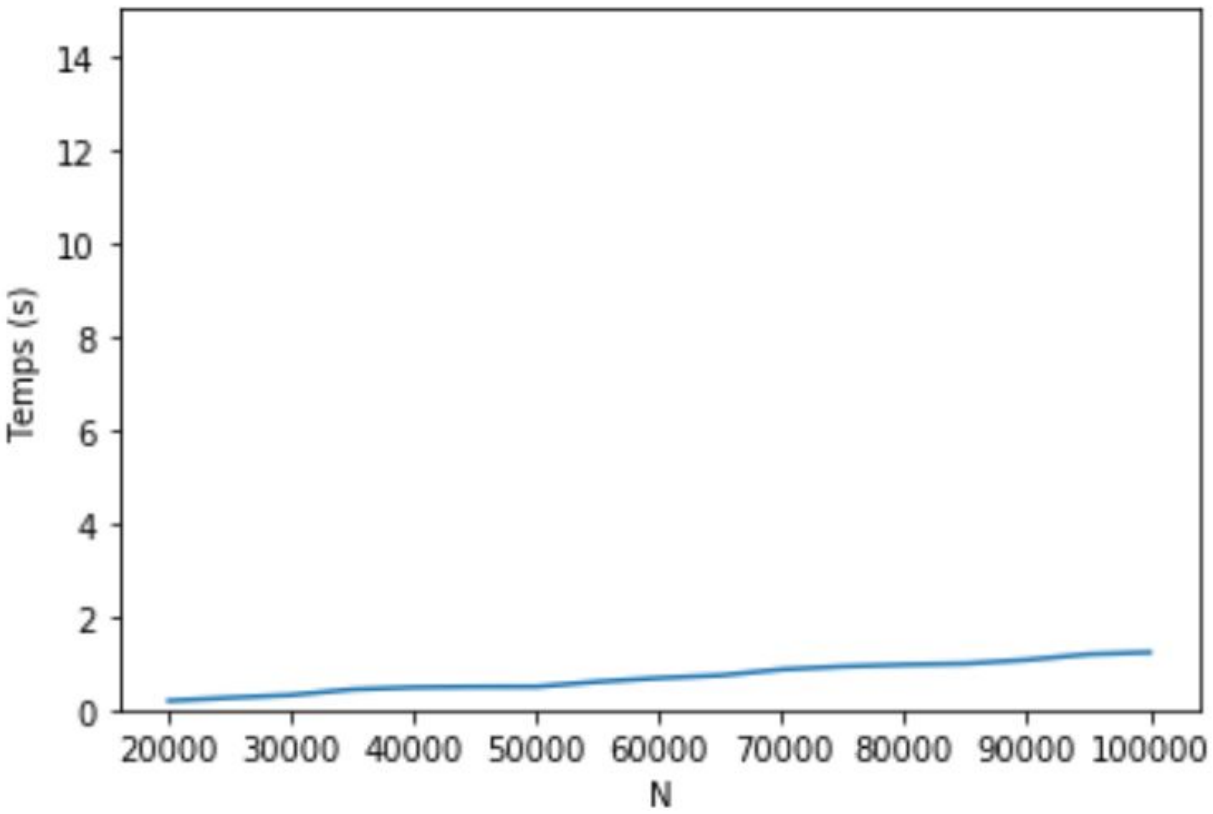
2.4.2.2 Rresoudre la relation de récurrence



2.4.3 Le tableau des mesures du temps

N	Temps d'execution
20 000	0.189236(s)
25 000	0.254567(s)
30 000	0.312633(s)
35 000	0.433292(s)
40 000	0.476262(s)
45 000	0.489499(s)
50 000	0.492604(s)
55 000	0.596565(s)
60 000	0.675480(s)
65 000	0.732747(s)
70 000	0.859293(s)
75 000	0.923387(s)
80 000	0.962447(s)
85 000	0.985198(s)
90 000	1.064705(s)
95 000	1.185699(s)
100 000	1.226289(s)

2.4.4 La courbe



la courbe correspondant à tri fusion

2.5 Tri rapide

Le Tri rapide algorithme applique le principe de diviser pour régner, de diviser le tableau d'entrée en deux petits tableau. L'algorithme trie ensuite les deux tableau de manière récursive jusqu'à ce que la liste résultante soit complètement triée.

2.5.1 Algorithme

Algorithm 7 partitionner(T,deb,fin)

Result: entier
Var i,j, pivot, aux : entier;
$i \leftarrow fin$
$pivot \leftarrow (deb + fin)/2$
for $j \leftarrow deb$ **to** fin **do**
  if $T[j] \leq pivot$ **then**
    $i \leftarrow i + 1$
    $aux \leftarrow T[i]$
    $T[i] \leftarrow T[j]$
    $T[j] \leftarrow aux$
  end
end

Algorithm 8 tri_rapide(T,deb,fin)

Var pivot : entier;
if $deb < fin$ **then**
  $pivot \leftarrow partitionner(T, deb, fin)$
  tri_rapide(T,deb,pivot-1)
  tri_rapide(T,pivot+1,fin)
end

2.5.2 La complexité asymptotique théorique

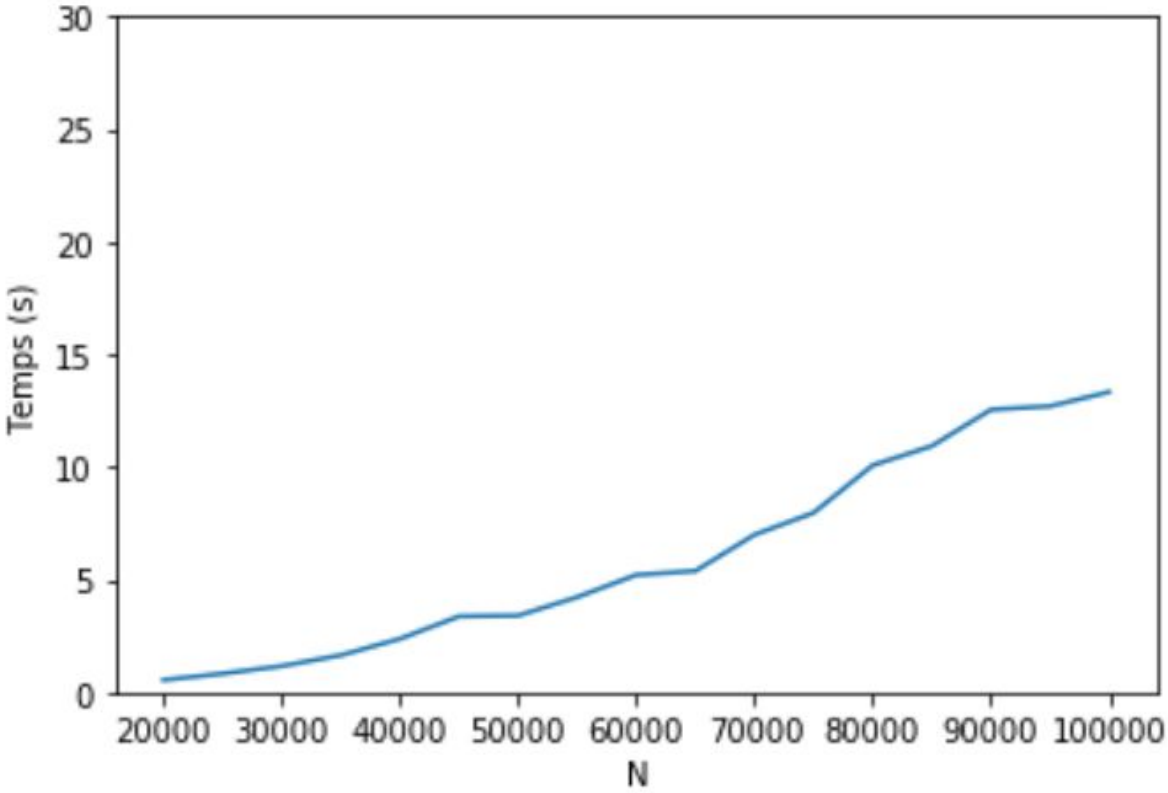
En utilisant la valeur médiane comme le pivot, vous retrouvez avec un runtime final de $O(n) + O(nlog_2(n))$. Vous pouvez simplifier cela à $O(nlog_2(n))$ car la partie logarithmique croît beaucoup plus rapidement que la partie linéaire.

1. La façon dont la sélection de pivot affecte le temps d'exécution de l'algorithme.
2. Le meilleur scénario $O(n)$ se produit lorsque la valeur sélectionnée est proche de la médiane du tableau
3. un scénario $O(n^2)$ se produit lorsque le pivot est la valeur la plus petite ou la plus grande du tableau.

2.5.3 Le tableau des mesures du temps

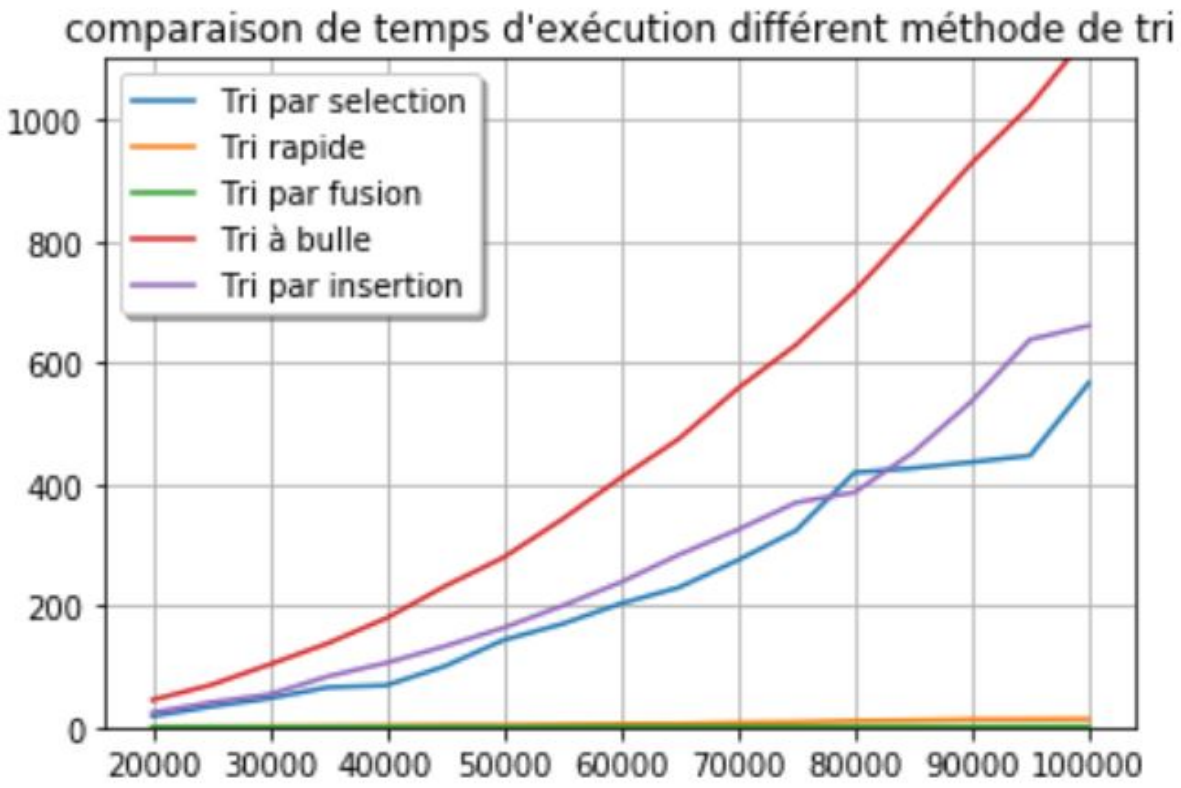
N	Temps d'execution
20 000	0.546236(s)
25 000	0.823267(s)
30 000	1.159791(s)
35 000	1.640299(s)
40 000	2.368704(s)
45 000	3.367129(s)
50 000	3.401085(s)
55 000	4.221853(s)
60 000	5.208756(s)
65 000	5.384215(s)
70 000	6.992965(s)
75 000	7.967626(s)
80 000	10.077397(s)
85 000	10.932843(s)
90 000	12.546186(s)
95 000	12.701415(s)
100 000	13.333375(s)

2.5.4 La courbe



la courbe correspondant à tri rapide

2.6 Comparaison



la courbe correspondant à tri rapide

N	Tri par selection	Tri par insertion	Tri à bulle	Tri par fusion	Tri rapide
85 000	7,1(min)	7,51(min)	13,66(min)	0.985(s)	10.932(s)
100 000	9,43(min)	11(min)	19(min)	1.226(s)	13.333(s)

Performance du PC — **ASUS**

- Processeur: Intel® Core(TM) i7-7500U, 7ème génération (2,70 GHz, up to 3.50 GHz)
- Mémoire RAM: 8 Go
- Disque Dur: 1To - Carte Graphique: NVIDIA

3 Conclusion

Le tri est un outil essentiel. Avec la connaissance des différents algorithmes de tri en Python et la manière de maximiser leur potentiel, vous êtes prêt à mettre en œuvre des applications et des programmes plus rapides et plus efficaces

Dans ce projet, vous avez appris:

- omment fonctionne Python avec les tri()
- Qu'est-ce que la **notation Big O** et comment l'utiliser pour comparer l'efficacité de différents algorithmes
- Comment mesurer le **temps réel** passé à exécuter votre code
- Comment implémenter cinq algorithmes de tri différents en Python
- Quels sont les **avantages et les inconvénients** de l'utilisation de différents algorithmes
- Vous avez également découvert différentes techniques telles que la écursivité , la**division** et la **diviser pour régner** et la randomisation . Ce sont des éléments de base fondamentaux pour résoudre une longue liste d'algorithmes différents.

Merci pour la lecture!