Lycée Louis-Le-Grand, Paris
MPSI 4 – Informatique commune
N. Carré, A. Troesch

2017/2018

TP n° 7 : Quelques algorithmes de tri

Pour tout ce TP, on se créera trois tableaux, l'un de longueur 10 (pour les tests de validité), les deux autres de longueur 1000 et 10000 (pour les tests d'efficacité), remplis de nombres aléatoires compris entre 0 *et* 10000*, et on déterminera le temps d'exécution des algorithmes de tri sur ces 2 tableaux, de sorte à comparer les qualités des différents algorithmes.*

Correction de l'exercice 1 – On fait d'abord une recherche du minimum de tableau. On parcourt le tableau à partir de l'indice a, en repérant au fur et à mesure le minimum provisoire. On pourrait envisager de ne repérer que l'indice, mais cela impose d'accéder plus souvent à des éléments du tableau. Même si cet accès est en temps constant, il est plus lent que l'accès à une variable numérique. L'expérience montre un gain d'environ 30% lorsqu'on stacke aussi la valeur du minimum.

```
def chercheMin(tab,a):
    """ recherche du minimum du tableau au delà  du rang a """
    i = a
    m = tab[a]
    for j in range(a+1, len(tab)):
        if tab[j] < m:
            m = tab[j]
            i = j
    return i
```

On peut alors écrire le tri selection :

```
def selection(tab):
    """ tri par selection """
    for k in range(len(tab)-1):
        i = chercheMin(tab,k)
        tab[k],tab[i] = tab[i],tab[k]
    return tab
```

La recherche du minimum se fait en temps linéaire (on fait un parcours du tableau), et on répète environ n fois cette recherche (même si les éléments parmi lesquels on fait cette recherche sont de moins en moins nombreux, cela nous fait parcourir la moitié d'un carré). Ainsi, on a une complexité en $\Theta(n^2)$. Ceci s'illustre très bien sur les essais numériques :

```
import random
import time

tableau1 = [random.randint(0,10000) for i in range(10)]
tableau2 = [random.randint(0,10000) for i in range(1000)]
tableau3 = [random.randint(0,10000) for i in range(10000)]

def test(tab):
    """ test de durée d'exécution """
    debut = time.time()
    selection(tab)
    fin = time.time()
    return fin - debut
```

```
print(tableau1)
print(selection(tableau1))
print(test(tableau2))
print(test(tableau3))
```

Les réponses obtenues sont :

```
[7973, 546, 8794, 223, 8682, 9231, 7233, 8101, 1416, 2712]
[223, 546, 1416, 2712, 7233, 7973, 8101, 8682, 8794, 9231]
0.05588245391845703
5.233356952667236
```

Les deux premières lignes forment le test de validité sur un tableau de taille 10, les deux dernières lignes donnent le temps de réponse pour des tableaux de taille 1000 et 10000. On observe un facteur 100 dans les temps de réponse, pour un facteur 10 sur la taille des entrées, ce qui illustre bien le caractère quadratique de l'algorithme.

Correction de l'exercice 2 – On parcourt le tableau en triant les éléments au fur et à mesure : pour chaque nouvel élément lu, on va l'insérer à sa place dans la partie déjà triée (les éléments qui précendent. Pour ce faire, on le compare aux précédents (par ordre décroissant), en faisant l'échange tant qu'il n'est pas à sa place.

```
def insertion(T):
    """ tri par insertion """
    for i in range(1,len(T)):
        j = i-1
        while j >= 0 and T[j] > T[j+1]:
            T[j+1],T[j] = T[j],T[j+1]
            j -= 1
    return T
```

Les tests de validité et de rapidité donnent (avec la même définition des tableaux, et on changeant juste le nom d'appel de la fonction `selection` en `insertion` dans la fonction `test`) :
On effectue des tests de validité et de rapidité :

```
import random
import time

tableau1 = [random.randint(0,10000) for i in range(10)]
tableau2 = [random.randint(0,10000) for i in range(1000)]
tableau3 = [random.randint(0,10000) for i in range(10000)]


def test(tab):
    """ test de durée d'exécution """
    debut = time.time()
    insertion(tab)
    fin = time.time()
    return fin - debut

print(tableau1)
print(insertion2(tableau1))
print(test(tableau2))
print(test(tableau3))
```

On obtient les résultats suivants :

```
[2076, 7238, 7202, 9504, 8909, 7977, 686, 27, 8484, 299]
[27, 299, 686, 2076, 7202, 7238, 7977, 8484, 8909, 9504]
0.12979388236999512
13.093523263931274
```

Le facteur 100 sur les temps de réponse pour des données différant d'un facteur 10 traduit le caractère quadratique de l'algorithme : à chaque nouvel élément, on est amené à parcourir tous les éléments précédents pour positionner le nouvel élément ; chaque étape s'éffectue en coût constant. Dans le pire des cas (tableau initial rangé en sens inverse) on parcourt un demi-carré. Dans le meilleur des cas (tableau trié), on ne parcourt qu'une diagonale (on ne revient pas en arrière). Ainsi, le complexité est en $O(n^2)$ et $\Omega(n)$, mais pas en $\Theta(n^2)$.

On peut se dire qu'une recherche dichotomique de l'emplacement de l'insertion peut accélérer l'algorithme. C'est vrai, mais la complexité reste en $O(n^2)$, du fait de l'insertion, qui restera prépondérante devant la recherche. En utilisant les facilités de Python, on obtient :

```
def insertion2(T):
    """ tri par insertion avec recherche dichotomique"""
    for i in range(1,len(T)):
        x = T[i]
        if T[0] >= x:
            T.insert(0,T.pop(i))
        elif x < T[i-1]:
            k = 0
            l = i-1
            while l-k > 1:
                m = (k + l) // 2
                if T[m] < x:
                    k = m
                else:
                    l = m
            T.insert(k+1,T.pop(i))
    return T
```

On obtient des résultats spectaculaires (on change juste le nom d'appel dans la fonction `test`) :

```
[1489, 5434, 5458, 9331, 5038, 4909, 5787, 7795, 6131, 9380]
[1489, 4909, 5038, 5434, 5458, 5787, 6131, 7795, 9331, 9380]
0.004994630813598633
0.1226797103881836
```

La différence notable provient de l'utilisation de la méthode optimisée `insert`. Pour une comparaison plus pertinente, on fait une troisième version dans laquelle on effectue l'insertion manuellement :

```
def insertion3(T):
    """ tri par insertion avec recherche dichotomique"""
    for i in range(1,len(T)):
        x = T[i]
        if T[0] >= x:
            k = -1
        elif x < T[i-1]:
            k = 0
            l = i-1
            while l-k > 1:
```

```
            m = (k + l) // 2
            if T[m] < x:
                k = m
            else:
                l = m
    else:
        k = i-1
    for j in range (0,i-k-1):
            T[i-j]=T[i-j-1]
    T[k+1] = x
return T
```

Cette fois, les tests donnent :

```
[467, 4286, 3442, 5811, 4794, 8571, 3745, 7413, 9727, 4364]
[467, 3442, 3745, 4286, 4364, 4794, 5811, 7413, 8571, 9727]
0.06376409530639648
6.281818628311157
```

On gagne donc à peu près un facteur 2 de la sorte.

Correction de l'exercice 3 –

1. On réalise le réanrrangement du tableau entre les indices a et b inclus :

```
def rearrangement(T,a,b):
    """ réarrange le tableau T entre les indices a et b inclus,
    de sorte à choisir un pivot, et positinner les éléments inférieurs
    au pivot à gauche du pivot et ceux qui sont supérieurs à droite.
    On retourne la position du pivot (le tableau initial étant modifier
    en place)"""
    piv = random.randint(a,b) # choix aléatoire du pivot
    p = T[piv]
    T[piv], T[b]= T[b], p # positionnement du pivot en fin
    # A tout moment le début de tranche contient les éléments < pivot
    # la suite les éléments >= pivot
    # et les derniers (à  part le pivot) ceux qu'on n'a pas encore
    # reclassés
    j = a # indice suivant la fin de la première tranche
    for i in range(a,b):
        if T[i] < p:
            T[i], T[j] = T[j],T[i]
            j += 1
        # pour positionner dans la première partie, on échange avec le
        # dernier de la deuxième partie, et on déplace notre séparation
        # Dans le cas inverse, l'élément est déjà  bien positionné
    T[j], T[b] = T[b], T[j]
        # on replace le pivot en l'échangeant avec le
        # premier élément de la seconde partie.
    return j
```

2. On réalise alors une fonction récursive pour le tri. Le principe de construction d'un algorithme récursif est le suivant : on suppose que notre fonction réalisant l'objectif voulu est valide pour tous les objets de taille

strictement inférieure à n et on s'arrange, à l'aide de cela (c'est-à-dire en s'autorisant à utiliser ladite fonction sur des objets de taille $< n$), pour réaliser l'objectif pour des objets de taille n. Il s'agit donc d'une construction par récurrence (forte). Il ne faut donc pas oublier d'initialiser, c'est-à-dire d'arrêter la récursivité lorsque les objets deviennent petits (ici, pour des tableaux de taille 0 et 1, sur lesquels il n'y a plus rien à faire).

L'idée ici est de commencer par un réarrangement. On trie ensuite la partie du tableau située avant le pivot (de taille strictement inférieure à la taille intiale, donc on peut utiliser une récursivité), et de même pour la partie située après le pivot. Vu le traitement initial, et en supposant la fonction valide pour les objets de taille plus petite, cela effectue bien un tri du tableau.

Pour pouvoir appliquer l'algorithme récursivement, il est nécessaire de passer en paramètre les indices de début et fin de traitement (puisqu'on l'appliquera à un morceau du tableau initial)

```
def triRapideCoeur(T,a,b):
    if b-a > 0:   # permet d'initialiser la récursivité
        pivot = rearrangement(T,a,b)
        triRapideCoeur(T,a,pivot-1)
        triRapideCoeur(T,pivot+1,b)
```

Pour se dispenser de passer a et b en paramètres en utilisation normale (consistant à trier tout le tableau), on définit ensuite :

```
def triRapide(T):
    triRapideCoeur(T,0,len(T)-1)
```

On teste la validité et le temps de réponse.

```
import random
import time

tableau1 = [random.randint(0,10000) for i in range(10)]
tableau2 = [random.randint(0,10000) for i in range(1000)]
tableau3 = [random.randint(0,10000) for i in range(10000)]

def test(tab, methode):
    """ test de durée d'exécution """
    debut = time.time()
    methode(tab)
    fin = time.time()
    return fin - debut

print(tableau1)
triRapide(tableau1)
print(tableau1)
print(test(tableau2,triRapide))
print(test(tableau3,triRapide))
```

On obtient par exemple :

```
[5456, 7761, 3989, 660, 297, 5847, 1788, 1036, 2028, 5539]
[297, 660, 1036, 1788, 2028, 3989, 5456, 5539, 5847, 7761]
0.00601649284362793
0.07390689849853516
```

Le rapport légèrement supérieur à 1 entre les deux temps d'exécution illustre la complexité quasi-linéaire en moyenne du tri rapide (c'est à dire en $n\ln(n)$). Nous ne justifions pas cette complexité en moyenne.

3. On adapte le tri par insertion d'un exercice précédent, en utilisant une recherche dichotomique (mais sans l'utilisation de la méthode `insert` afin de ne pas trop fausser les estimations de complexité). On doit ici réaliser un tri par insertion partiel (entre deux indices) :

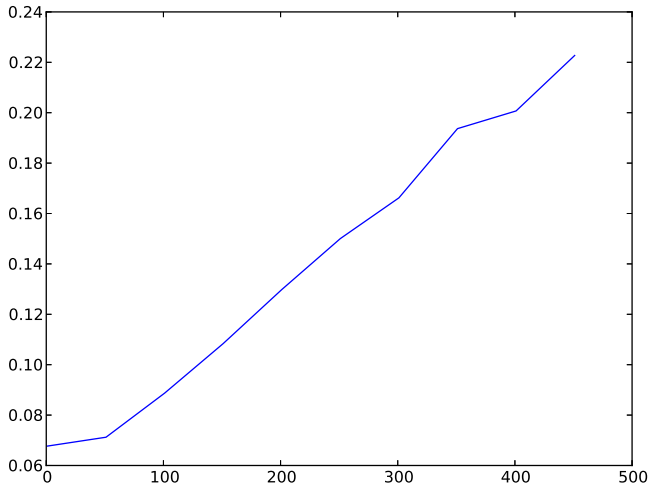
```
def insertion(T,a,b):
  """ tri par insertion avec recherche dichotomique entre a et b"""
  for i in range(a+1,b+1):
      x = T[i]
      if T[a] >= x:
          k = a-1
      elif x < T[i-1]:
          k = a
          l = i-1
          while l-k > 1:
              m = (k + l) // 2
              if T[m] < x:
                  k = m
              else:
                  l = m
      else:
          k = i-1
      for j in range (0,i-k-1):
              T[i-j]=T[i-j-1]
      T[k+1] = x
  return T
```

On rajoute alors un seuil dans les différentes fonctions :

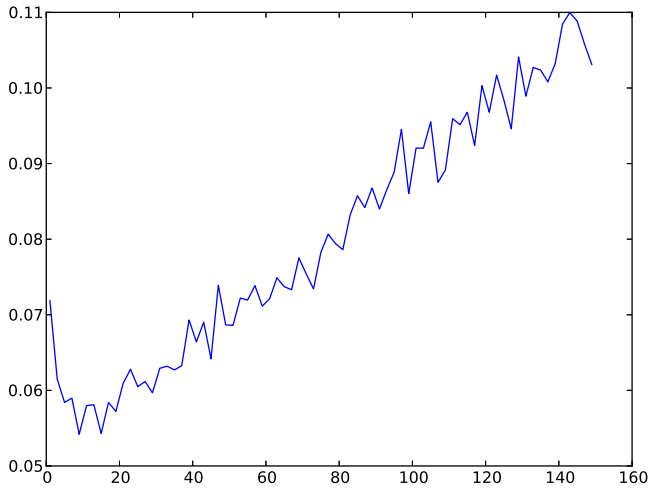
```
def triRapideCoeur(T,a,b,s0):
    if b-a > s0:   # permet d'initialiser la récursivité
        pivot = rearrangement(T,a,b)
        triRapideCoeur(T,a,pivot-1,s0)
        triRapideCoeur(T,pivot+1,b,s0)
    else:
        insertion(T,a,b)


def triRapide(T,s):
    triRapideCoeur(T,0,len(T)-1,s)
```

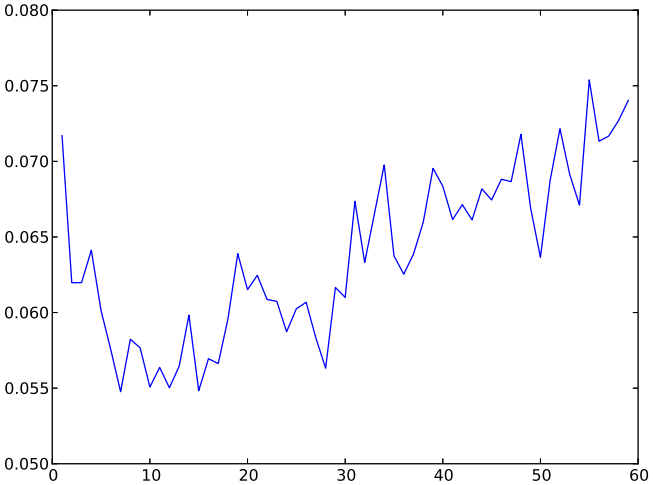
Voici le graphe obtenu en 1 et 500 :



on affine entre 1 et 150, par pas de 2 :



Puis entre 1 et 60 par pas de 1 :



Apparemment, le seuil optimal est autour de 10 ou 15.

4. On fait une petite variante en lançant un tripar insertion global (arrivé au seuil n_0, on laisse tel quel dans un premier temps). Pour que cette méthode soit efficace, on ne faut pas faire une recherche par dichotomie, car il

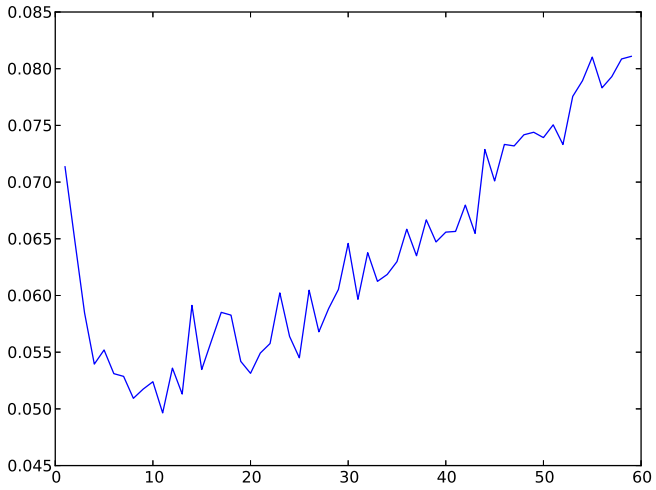
faut pouvoir exploiter le fait que le tri par insertion classique est meilleur sur les tableaux presque triés. Mais ceci n'est vrai que pour la méthode usuelle, consistant à comparer l'élément à trier avec les précédents jusqu'à ce qu'il soit bien positionné. En effet, si le tableau est presque trié, en moyenne, on n'aura pas à remonter beaucoup chacun des éléments, et ceci de façon bornée par le seuil (ce qui nous assure une traitement linéaire).

```
def triRapideCoeur2(T,a,b,s0):
    if b-a > s0:   # permet d'initialiser la récursivité
        pivot = rearrangement(T,a,b)
        triRapideCoeur2(T,a,pivot-1,s0)
        triRapideCoeur2(T,pivot+1,b,s0)


def triRapide2(T,s):
    triRapideCoeur2(T,0,len(T)-1,s)
    insertion2(T)

def insertion2(T):
    """ tri par insertion """
    for i in range(1,len(T)):
        j = i-1
        while j >= 0 and T[j] > T[j+1]:
            T[j+1],T[j] = T[j],T[j+1]
            j -= 1
    return T
```

Le graphe obtenu pour des seuils allant jusqu'à 60 est cette fois :



Le seuil optimal se situe également vers 10. On constate aussi que l'ordre de grandeur du temps de réponse est le même.

Correction de l'exercice 4 –

1. On effectue la fusion en parcourant simutanément les deux tableaux et en selectionnant des 2 tableaux la plus petite valeur : pour fusionner deux paquets de cartes, on les place face visible vers le haut (au haut du paquet on a les plus petites valeurs), côte à côte, et on selectionne à chaque fois la plus petite des 2 cartes visibles au haut des 2 paquets. Cela donne :

```
def fusion(T1,T2,T,a):
```

```
    k = a
    while len(T1) != 0:
        if len(T2) != 0:
            if T1[0] < T2[0]:
                T[k] = T1.pop(0)
                k += 1
            else:
                T[k] = T2.pop(0)
                k += 1
        else:
            T[k] = T1.pop(0)
            k += 1
```

Ici, on suppose qu'on fusionne deux tableaux T_1 et T_2 en donnant le résultat dans un tableau T à partir de l'indice a (en remplaçant les données présentes dans T.

On obtient alors l'algorithme récursif de tri fusion, consistant à couper le tableau en sa moitié, trier récursivement les 2 moitiés, puis fusionner ces deux tris : pour cela on extrait les deux sous-tableaux, et on les fusionne dans le tableau initial. Pour pouvoir opérer la récursivité, on décrit cela entre deux indices a et b dans un premier temps.

```
def triFusionCoeur(T,a,b):
    if b-a >= 1:
        c = (a+b) // 2
        triFusionCoeur(T,a,c)
        triFusionCoeur(T,c+1,b)
        fusion(T[a:c+1],T[c+1:b+1],T,a)
    return T

def triFusion(T):
    triFusionCoeur(T,0,len(T)-1)
```

On fait des tests de validité et de durée :

```
import random
import time
import matplotlib.pyplot as plt

tableau1 = [random.randint(0,10000) for i in range(10)]
tableau2 = [random.randint(0,10000) for i in range(1000)]
tableau3 = [random.randint(0,10000) for i in range(10000)]

def test(tab):
    """ test de durée d'exécution """
    debut = time.time()
    triFusion(tab)
    fin = time.time()
    return fin - debut

print(tableau1)
triFusion(tableau1)
print(tableau1)
print(test(tableau2))
print(test(tableau3))
```

On obtient les résultats suivants :

```
2627, 9505, 4332, 5115, 90, 3552, 7282, 30, 8865, 7194]
[30, 90, 2627, 3552, 4332, 5115, 7194, 7282, 8865, 9505]
0.008220911026000977
0.11953186988830566
```

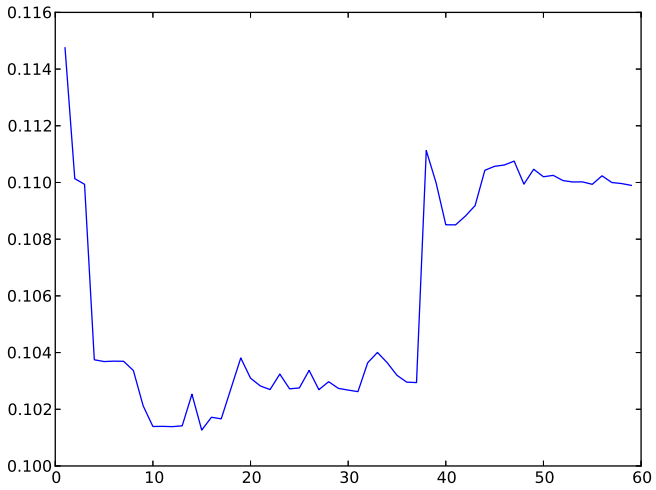
Le rapport un peu supérieur à 10 pour un rapport de taille de 10 illustre bien la complexité en $n\ln(n)$ (c'est un Θ ici ; cela peut se justifier avec les règles usuelles concernant les algorithmes de type diviser pour régner).

2. On implémente le tri comme dans un exercice précédent, entre deux valeurs a et b incluses. On obtient alors :

```
def triFusionSeuilCoeur(T,a,b,s):
    if b-a > s:
        c = (a+b) // 2
        triFusionSeuilCoeur(T,a,c,s)
        triFusionSeuilCoeur(T,c+1,b,s)
        fusion(T[a:c+1],T[c+1:b+1],T,a)
    else:
        insertion(T,a,b)
    return T

def triFusionSeuil(T,s):
    triFusionSeuilCoeur(T,0,len(T)-1,s)
```

Comme dans un exercice précédent, on peut obtenir le graphe de la durée d'exécution en fonction du seuil :



Le seuil optimal est assez flou, entre 10 et 20.

Il existe bien d'autres algorithmes de tri, plus ou moins efficaces, et plus ou moins adaptés à certaines situations. Ainsi, avec des hypothèses supplémentaires, on peut descendre en dessous du seuil $n\ln(n)$ pour la complexité. Par exemple, si on sait que les valeurs à trier sont toutes des valeurs contenues dans un ensemble fini connu, il n'est pas dur de trouver un algorithme de tri linéaire (tri par dénombrement). Comment feriez-vous ?