2025 - 2026

Cours d'informatique

Rappels sur les algorithmes de tri

On dispose d'une liste $L = [L[0], L[1], ..., L[n-1]]$ de longueur n, dont les coefficients sont des entiers ou des flottants.

On souhaite écrire un algorithme permettant de trier les éléments de cette liste, par exemple dans l'ordre croissant. Il existe évidemment de nombreuses façons de procéder ; le but de ce cours est de décrire quelques-unes de ces méthodes, et de comparer leur efficacité. Ce cours sera l'occasion d'utiliser la notion d'algorithme récursif, que l'on creusera ultérieurement. On se penchera dans un second temps sur les preuves de corrections et les estimations de complexité temporelle des algorithmes de tris présentés.

I. Description de trois premiers algorithmes de tri

1. Le tri par sélection

Le principe

On crée une liste T, vide. On cherche le plus petit élément de L. On le place dans la liste T, et on le supprime de la liste L. Puis on cherche le nouveau plus petit élément de L, que l'on ajoute à la liste T et que l'on supprime de la liste L. On répète l'opération jusqu'à ce que la liste L soit vide : T contient alors tous les éléments de L, rangés dans l'ordre croissant.

Raffinement : le tri par sélection en place

Créer une liste, lui ajouter des éléments, et surtout supprimer un élément d'une liste, sont des opérations coûteuses (à chaque suppression, la liste L est recrée). On peut éviter cet inconvénient en utilisant la variante suivante du tri par sélection :

On cherche le plus petit élément de L, et on l'échange avec $L[0]$.

Puis on cherche le plus petit élément de $[L[1], ..., L[n-1]]$, que l'on échange avec $L[1]$.

On cherche ensuite le plus petit élément de $[L[2], ..., L[n-1]]$, on l'échange avec $L[2]$, et ainsi de suite…

Remarque

Il pourra être pratique d'écrire et d'utiliser un algorithme annexe, recherchant la place du plus petit élément d'une liste.

2. Le tri à bulles

On parcourt la liste L. Pour $i \in [0, n-2]$, si $L[i]$ est supérieur à $L[i+1]$, on échange ces deux éléments.

On effectue cette opération de parcours-échanges jusqu'à ce que la liste soit entièrement triée.

Possibilité 1

Remarquons qu'à l'issue du premier parcours de la liste, le plus grand élément de L sera nécessairement à la bonne place (la dernière). Après deux parcours, on est sûr que les deux plus grands éléments de L seront bien placés, et ainsi de suite… en fin de compte, après $n-1$ parcours de la liste, les $n-1$ plus grands éléments de L sont bien placés, le plus petit élément de L l'est donc aussi : la liste L est triée.

Possibilité 2

On peut également recommencer le parcours jusqu'à ce que, lors de celui-ci, aucun échange ne soit plus constaté : c'est qu'alors la liste est triée.

3. Le tri par insertion

Le principe

On crée une liste T, contenant uniquement $L[0]$. On y insère $L[1]$, en le plaçant correctement par rapport à $L[0]$. On ajoute ensuite $L[2]$, mis à la bonne place par rapport à $L[0]$ et $L[1]$, et ainsi de suite…

Remarque 1

Là encore, on peut améliorer l'algorithme en adoptant une méthode de tri par insertion en place :

Si $L[1] < L[0]$, on échange ces deux éléments, la sous-liste $[L[0], L[1]]$ est alors triée. Ensuite, si $L[2] < L[1]$, on échange ces deux éléments, et, si nécessaire, on fait également l'échange avec $L[0]$, de sorte que la sous-liste $[L[0], L[1], L[2]]$ soit correctement triée. De manière générale, la sous-liste $[L[0],..., L[i-1]]$ étant triée, on fait en sorte, à l'aide d'échanges successifs, que la sous-liste $[L[0],..., L[i-1], L[i]]$ le soit.

Remarque 2

Quelle que soit la version du tri par insertion utilisée, on voit qu'une fonction annexe serait utile :

Etant donnée une liste $T = [T[0], ..., T[n-1]]$ telle que la sous-liste $[T[0], ..., T[n-2]]$ soit triée, cette fonction fera en sorte qu'à l'aide d'échanges successifs du dernier élément avec les autres, la liste T soit entièrement triée.

II Implémentations en Python

1. Le tri par sélection

```
#première possibilité : sélection en place
def Tri_Selection1(L):
    for i in range(len(L)-1):
        mini=i
        for j in range(i+1,len(L)):
            if L[j]<L[mini]:
                mini=j
        L[i],L[mini]=L[mini],L[i]

#deuxième possibilité : sélection externe
def Tri_Selection2(L):
    T=[]
    n=len(L)
    for i in range(len(L)-1):
        mini=0
        for j in range(2,len(L)):
            if L[j]<L[mini]:
                mini=j
        T.append(L[mini])
        del(L[mini])
    return T+L
```

2. Le tri à bulles

```
#Version courte
def Tri_Bulle1(L):
    T=copy.copy(L)
    for i in range(len(T)):
        for j in range(len(L)-1):
            if T[j]>T[j+1]:
                L[i],L[i+1]=L[i+1],L[i]
    return T

#Plus compliqué, mais moins d'opérations inutiles
def Tri_Bulle2(L):
    fin=False
    while (not fin):
        fin=True
        for i in range(len(L)-1):
            if L[i]>L[i+1]:
                L[i],L[i+1]=L[i+1],L[i]
                fin=False
```

3. Le tri par insertion

```
# Inserer a pour fonction d'insérer T[i] a sa place dans T[0...i-1]
def Inserer(T,i) :
    if i>0 :
        if T[i-1] > T[i] :
            echange2(T,i,i-1)
            Inserer(T,i-1)

def Tri_Insertion(T) :
    for i in range(1,len(T)):
        Inserer(T,i)
```

III Diviser pour mieux régner : le tri fusion et le tri rapide

Les descriptions ci-dessous sont empruntées à N. Bailly (ESIEE Amiens)

1. Le tri fusion

a. Description

Le tri fusion est construit suivant la stratégie "diviser pour régner", en anglais "divide and conquer". Le principe de base de celle-ci est que, pour résoudre un problème complexe, il est souvent plus facile de le diviser en plusieurs sous-problèmes. Une fois ceux-ci résolus, il n'y a plus qu'à combiner les différentes solutions pour résoudre le problème global. La méthode "diviser pour régner" est tout à fait applicable au problème de tri : plutôt que de trier le tableau complet, il est préférable de trier deux sous tableaux de taille égale, puis de fusionner les résultats.

L'algorithme proposé ici est récursif. En effet, les deux sous tableaux seront eux même triés à l'aide de l'algorithme de tri fusion. Un tableau ne comportant que 0 ou un seul élément sera considéré comme trié : ce sont les cas terminaux. Etapes de l'algorithme :

- Division de l'ensemble de valeurs en deux parties égales
- Tri de chacun des deux ensembles
- Fusion des deux ensembles.

b. Une implémentation en Python

```
def fusion(L1,L2):
    T=[]
    i,j=0,0
    while i<len(L1) and j<len(L2):
        if L1[i]<L2[j]:
            T.append(L1[i])
            i+=1
        else:
            T.append(L2[j])
            j+=1
    if i<len(L1):
        return T+L1[i:]
    else:
        return T+L2[j:]

def trifusion(L):
    if len(L)<=1:
        return L
    else:
        return fusion(trifusion(L[:len(L)//2]),trifusion(L[len(L)//2:]))
```

2. Le tri rapide

a. Description

L'algorithme de tri rapide, "quick sort" en anglais, est un algorithme de type dichotomique. Son principe consiste à séparer l'ensemble des éléments en deux parties. La différence par rapport au tri fusion est que la séparation des différentes valeurs ne s'effectue pas n'importe comment. Pour effectuer la séparation, une valeur pivot est choisie. Les valeurs sont réparties en deux ensembles suivant qu'elles sont plus grandes ou plus petites que le pivot. Ensuite, les deux ensembles sont triés séparément, suivant la même méthode. L'algorithme, tout comme le tri fusion, est récursif, mais cette fois, il n'est pas nécessaire de fusionner les deux ensembles : le résultat du tri est égal au tri de l'ensemble dont les valeurs sont inférieures au pivot concaténé à l'ensemble des valeurs supérieures au pivot, ce dernier étant pris en sandwich entre les deux ensembles.

Le choix du pivot est un problème central de cet algorithme. L'idéal serait de pouvoir répartir les deux ensembles en deux parties de taille à peu près égales. Cependant, la recherche d'un pivot qui permettrait une partition parfaite de l'ensemble en deux parties égales aurait un coût trop important. C'est pour cela que le pivot est généralement choisi de façon arbitraire parmi les valeurs de l'ensemble. Dans la pratique, on décidera par exemple que le pivot est le premier élément de l'ensemble à fractionner.

b. Une implémentation en Python

```
def trirapide(L):
    if L == []:
        return []
    else:
        n = len(L)-1 #on va balayer la liste L et répartir les valeurs
        L1 = []
        L2 = []
        for k in range(1,n+1):
            if L[k]<=L[0]:
                L1.append(L[k]) #L1 reçoit les éléments plus petits
            else:
                L2.append(L[k]) #L2 reçoit les éléments plus grands
        L = trirapide(L1)+[L[0]]+trirapide(L2)
    return L
```

IV Analyse des algorithmes de tri

L est toujours une liste de flottants de longueur n , dont on veut trier les éléments dans l'ordre croissant.
Dans ce paragraphe, on revient rapidement sur le principe et l'implémentation en Python des différents tris présentés, on apporte des preuves de correction de ces implémentations, et on établit des estimations de leur complexité temporelle.

1. Le tri par sélection

Le plus naturel

Principe

On recherche le plus petit élément de L , que l'on met à la place d'indice 0 .

puis on recherche le plus petit élément de L[1 :], que l'on met à la place d'indice 1 .

on recherche ensuite le plus petit élément de L[2 :], et on le met à la place d'indice 2 …

et ainsi de suite jusqu'à ce que l'on ait épuisé toute la liste.

Implémentation en Python

```
1 def TriSelec(L):
2     for i in range(len(L)):
3         ch = i
4         for k in range(i+1,len(L)):
5             if L[k]<L[ch] :
6                 ch = k
7         L[ch],L[i] = L[i],L[ch]
8     return L
```

Preuve de correction

Invariant de boucle 1 : $P(k)$: après l'itération d'indice k de la boucle for de la ligne 4, ch est l'indice où l'on trouve l'élément maximal de $L[i:k+1]$.

Invariant de boucle 2 : $H(i)$: après i itérations de la boucle for de la ligne 2, $L[0:i]$ est trié dans l'ordre croissant.

Complexité

Prenons pour opérations élémentaires les comparaisons entre les éléments du tableau. Le tri par sélection effectue $\binom{n}{2} = \frac{n(n-1)}{2}$ comparaisons ; sa complexité est donc quadratique (ie. en $O(n^2)$).

2. Le tri à bulles

Le plus simple

Principe

On parcourt la liste en échangeant chaque élément avec son successeur s'ils ne sont pas rangés dans le bon ordre.

On réitère cette opération n fois, la liste est alors triée dans l'ordre croissant.

Implémentation en Python

```
def tribulles(L):
    for i in range(len(L)):
        for j in range(len(L)-1):
            if L[j]>L[j+1]:
                L[j],L[j+1]=L[j+1],L[j]
    return L
```

Preuve de correction

Invariant de boucle : $P(i)$: après l'itération d'indice i de la boucle for de la ligne 2, les i plus grands éléments de L sont rangés à la bonne place.

Complexité

Si l'on prend pour opérations élémentaires les comparaisons entre les éléments du tableau, le tri à bulles proposé effectue n^2 comparaisons ; sa complexité est donc elle aussi quadratique (en $O(n^2)$).

---

3. Le tri par insertion

Le plus méthodique

Principe

On part de $[L[0]]$, liste composée du seul élément $L[0]$. Dans cette liste, on insère successivement $L[1]$, $L[2]$, … , $L[n-1]$, en plaçant à chaque fois l'élément inséré à la bonne place par rapport aux éléments déjà triés.

Implémentation en Python

```
1  # Inserer a pour fonction d'insérer L[i] à sa place dans L[0...i-1]
2  def Inserer(L,i) :
3      fin=False
4      j=i
5      while not fin :
6          if j>0 :
7              if L[j-1] > L[j] :
8                  L[j],L[j-1]=L[j-1],L[j]
9                  j=j-1
10             else:
11                 fin=True
12         else:
13             fin=True
14
15 def Tri_Insertion(L) :
16     for i in range(1,len(L)):
17         Inserer(L,i)
18     return L
19
```

Preuve de terminaison

La question ne se pose que pour la fonction Inserer(L,i). j est un entier et, à chaque itération de la boucle while : soit fin prend la valeur True, et alors la boucle se termine ; soit j diminue de 1 ; supposons la boucle non terminée après i itérations. Alors fin n'a jamais pris la valeur True, j a donc diminué i fois de 1, et vaut donc 0 à l'issue de cette i – ème itération. fin prend donc la valeur True à l'itération suivante, et la boucle se termine.

Preuve de correction

Invariant de boucle : après i itérations, la sous-liste $L[:i+1]$ est triée dans l'ordre croissant.

- Après 0 itérations, $L[:1]$ ne contient que l'élément $L[0]$, donc est triée.
- Soit $i \in \{0, ..., n-2\}$. On suppose qu'après i itérations, la sous-liste $L[:i+1]$ est triée. On appelle alors Inserer(L,i+1), et :

  Dans la boucle while, soit $L[i+1] \geq L[i]$, et alors cette boucle se termine, et $L[:i+2]$ est triée. Soit $L[i+1] < L[i]$, et alors (puisque $L[i] = \max(L[0], ..., L[i])$ était déjà acquis), c'est que

$L[i] = \max(L[0], ..., L[i+1])$. Ces deux valeurs sont alors échangés, le plus grand élément de la sous-liste $L[:i+2]$ est donc à la bonne place, et $L[:i]$ est toujours triée. j prend la valeur i, donc, pour les itérations suivantes, tout se passe comme si l'on avait appelé Inserer(L,i) : par hypothèse de récurrence, la sous-liste $L[:i+1]$ est triée à l'issue de cette boucle while, et donc $L[:i+2]$ l'est aussi.

- Après $n-1$ itérations, la liste L est donc entièrement triée.

Complexité

On prend toujours pour opérations élémentaires les comparaisons.

La boucle for est de toutes façons itérée $n-1$ fois. Lors de son itération i, la boucle for est itérée entre 1 et $i+1$ fois, et, lors de chaque itération de cette boucle while, on effectue une ou deux comparaisons.

- La complexité dans le meilleur des cas est donc $C_m(n) = \sum_{i=1}^{n-1} 1 = n-1 = O(n)$, elle est donc linéaire.
- La complexité dans le pire des cas est de l'ordre de $C_p(n) = \sum_{i=1}^{n-1} (2i+1) = (n+1)(n-1) = O(n^2)$,

cette complexité dans le pire des cas est donc quadratique.

---

4. Le tri fusion

Le plus rapide

Principe

On coupe la liste L en deux parties de longueurs (à peu près) égales ; on trie chacune d'entre elles, et il est rapide ensuite de les réunir en une liste ordonnée. On adopte le même principe pour trier les deux demies-listes : on les coupe en deux moitiés que l'on trie puis réunit, et ainsi de suite par récursivité, en coupant en deux jusqu'à parvenir à des sous-listes de 0 ou 1 élément, donc déjà triées.

Implémentation en Python

```python
def Fusion(L1,L2): # L1 et L2 sont 2 listes déjà triées par ordre croissant
    i=0 # Fusion renvoie la liste L1+L2 triée par ordre croissant
    j=0
    res = []
    while i<len(L1) and j<len(L2) :
        if L1[i]<L2[j] :
            res.append(L1[i])
            i+=1
        else :
            res.append(L2[j])
            j+=1
    return(res + L1[i:] + L2[j:])

def TriFusion(L):
    if (len(L)<=1) :
        return(L)
    d = len(L)//2
    L1 = L[:d] # prend les éléments de 0 à d-1
    L2 = L[d:] # prend les éléments de d à la fin
    return(Fusion(TriFusion(L1),TriFusion(L2)))
```

Preuve de correction et de terminaison

- *Correction et terminaison de la fonction Fusion*

  On suppose les listes L_1 et L_2 déjà triées.

  On définit la propriété $P(n)$ par : « si la somme des longueurs de L_1 et L_2 vaut n, Fusion(L1,L2) se termine et renvoie une liste triée contenant tous les éléments de L_1 et L_2 ».

- *Correction et terminaison de la fonction Fusion*

On prouve la propriété $H(n)$: « pour une liste L de longueur n, TriFusion(L) se termine et renvoie la liste L triée ».

Calcul intuitif de la complexité

n éléments							
$n/2$ éléments				$n/2$ éléments			
$n/4$ éléments		$n/4$ éléments		$n/4$ éléments		$n/4$ éléments	

etc. , ce qui forme une tour à $\lfloor \log_2(n) \rfloor + 1$ étages nécessitant chacun n comparaisons, d'où une complexité en $\boxed{O(n \ln n)}$.

Calcul formel de la complexité

Evaluons déjà le coût, en terme de nombre de comparaisons, de la fusion de deux listes de longueur 2^{k-1}. On effectue une comparaison à chaque itération de la boucle while, et ce jusqu'à ce que l'une des deux listes soit vidée, donc ce nombre de comparaisons est compris entre 2^{k-1} et 2^k. En notant $C_p(n)$ la complexité (toujours en nombre de comparaisons) dans le pire des cas du tri fusion d'une liste de longueur n, on a donc :

$C_p(2^k) = 2\,C_p(2^{k-1}) + 2^k$ (coût de la fusion + coût du tri des deux demies-listes). On en déduit que $\dfrac{C_p(2^k)}{2^k} = \dfrac{C_p(2^{k-1})}{2^{k-1}} + 1$: la suite $\left(\dfrac{C_p(2^k)}{2^k}\right)$ est arithmétique, et, comme $\dfrac{C_p(1)}{1} = 0$, on en déduit que $\dfrac{C_p(2^k)}{2^k} = k$, soit $C_p(2^k) = k\,2^k$.

Pour n entier quelconque, on a $2^k \le n < 2^{k+1} \Leftrightarrow k = \left\lfloor \dfrac{\ln n}{\ln 2} \right\rfloor$, d'où :

$C_p\left(2^{\left\lfloor \frac{\ln n}{\ln 2} \right\rfloor}\right) \le C_p(n) \le C_p\left(2^{\left\lfloor \frac{\ln n}{\ln 2} \right\rfloor + 1}\right)$, ce qui donne $\left\lfloor \dfrac{\ln n}{\ln 2} \right\rfloor 2^{\left\lfloor \frac{\ln n}{\ln 2} \right\rfloor} \le C_p(n) \le \left(\left\lfloor \dfrac{\ln n}{\ln 2} \right\rfloor + 1\right) 2^{\left\lfloor \frac{\ln n}{\ln 2} \right\rfloor + 1}$.

Comme $\left\lfloor \dfrac{\ln n}{\ln 2} \right\rfloor \sim \dfrac{\ln n}{\ln 2}$ et $2^{\left\lfloor \frac{\ln n}{\ln 2} \right\rfloor} = O\left(2^{\frac{\ln n}{\ln 2}}\right) = O\left(e^{\frac{\ln n}{\ln 2} \ln 2}\right) = O(n)$, on obtient finalement :

$C(n) = O(n \ln n)$.

On montre de la même façon que la complexité dans le meilleur des cas est elle aussi en $O(n \ln n)$.

---

5. Le tri rapide (quicksort)

Le meilleur rapport qualité/prix

Principe

La méthode consiste à choisir 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 tableau :

- on peut prendre pour pivot (par exemple) le premier élément du tableau.
- On place tous les éléments inférieurs au pivot dans un sous-tableau qu'on appellera ici Petits, tous les éléments supérieurs au pivot dans un autre sous-tableau qu'on appellera Grands.
- On place le pivot à la fin des éléments de Petits, puis on place tous les éléments de Grands à sa suite.

Implémentation en Python

```
1  def TriRapide(L):
2      if L == [] :
3          return []
4      pivot = L[0]
5      Petits = []
6      Grands = []
7      for x in L[1:] :
8          if x<pivot :
9              Petits.append(x)
10         else : Grands.append(x)
11     return TriRapide(Petits) + [pivot] + TriRapide(Grands)
```

Preuve de correction et de terminaison

On peut choisir pour invariant :

$P(n)$: « pour une liste de longueur n, Trirapide(L) se termine, et renvoie la liste L triée dans l'ordre croissant ».

Complexité

On évalue à nouveau la complexité en nombre de comparaisons.

- Le pire des cas est obtenu en choisissant les pivots tels qu'à chaque séparation, l'un des deux tableaux obtenus soit de taille 0 (ce qui arrive lorsque la liste est déjà triée). Alors la complexité $C_p(n)$ du tri rapide d'un tableau de taille n dans le pire des cas vérifie : $C_p(n) = C_p(n-1) + n - 1$, d'où

$$C_p(n) = \sum_{k=1}^{n-1}(k-1) = \frac{n(n-1)}{2} = O(n) :$$

  la complexité dans le pire des cas est quadratique.

- Dans le meilleur des cas, les pivots séparent le tableau en deux parties de longueur égale. Pour un tableau de taille 2^k, la complexité dans le meilleur des cas vérifie donc :

$$C_m(2^k) = C_m(2^{k-1}) + C_m(2^{k-1} - 1) + 2^k - 1,$$

  d'où : $C_m(2^k) \leq 2\,C_m(2^{k-1}) + 2^k$.

  On s'est ramené à la relation de récurrence étudiée pour le tri fusion, on en déduit de manière analogue que :

$$\boxed{C_m(2^k) = O(n \ln n)}.$$

Remarques

- On peut montrer que la complexité en moyenne est également en $O(n \ln n)$.
- Il existe des variantes améliorant ce tri. Par exemple, au lieu de prendre pour pivot le premier élément, on peut utiliser des méthodes de recherche d'un pivot efficace. On peut également éviter la coûteuse concaténation écrite ligne 11, en effectuant des échanges : cf. par exemple la version proposée ci-dessous.

Un exemple de tri rapide en place :

```
def trirapide(L):
    """trirapide(L): tri rapide (quicksort) de la liste L"""
    def trirap(L, g, d):
        pivot = L[(g+d)//2]
        i = g
        j = d
        while True:
            while L[i]<pivot:
                i+=1
            while L[j]>pivot:
                j-=1
            if i>j:
                break
            if i<j:
                L[i], L[j] = L[j], L[i]
            i+=1
            j-=1
        if g<j:
            trirap(L,g,j)
        if i<d:
            trirap(L,i,d)

    g=0
    d=len(L)-1
    trirap(L,g,d)
```