Travaux pratiques 8

Tris simples

Informatique tronc commun MPSI et PCSI

Dans ce T.P. nous allons mettre en place des méthodes de tri d'une liste de nombres.
Le tri est une opérations très importante en informatique, on en verra quelques applications, c'est en particulier une étape indispensable pour pouvoir utiliser l'algorithme de recherche par dichotomie.

I Qu'est-ce que trier ?

La question peut sembler évidente : on part d'une liste et on veut une liste triée, en général par ordre croissant. Cependant il y a une ambiguïté sur ce que l'on veut : souhaite-t-on que la liste initiale soit triée, c'est-à-dire modifiée en place ? Ou souhaite-ton obtenir une nouvelle liste, triée, en conservant la liste initiale ?
Dans ce T.P. nous allons principalement trier en place : la fonction de tri recevra une liste en paramètre et ne renverra rien (pas de `return` mais la liste sera modifiée.

```
L = [15, 5, 2, 11, 3, 8]
trier(L)

> L
[2, 3, 5, 8, 11, 15]
```

Dans ce cadre, un fonction importante est celle qui permet de permuter deux valeurs dans une liste.

Ici aussi, cette fonction modifie une liste en place, sans renvoyer de résultat, on dit aussi que c'est une procédure.

Exercice 1 - Permutation
Écrire une fonction `echanger(L, i, j)` qui échange les valeurs de `L[i]` et de `L[j]`.
On pourra commencer par expliquer pourquoi l'écriture suivante n'est pas bonne.

```
def echanger(L, i, j):
    L[i] = L[j]
    L[j] = L[i]
```

II Comment trier ?

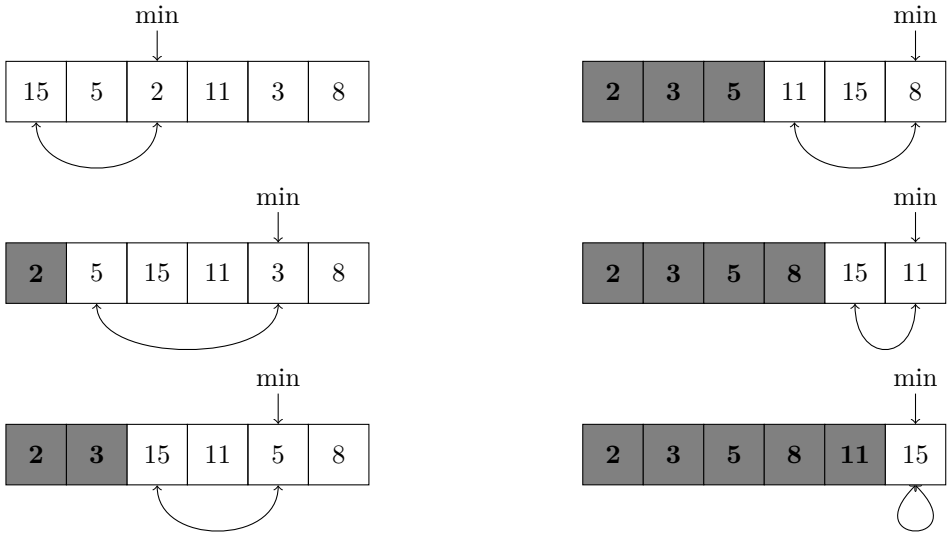
Plutôt que proposer un algorithme sorti de nulle part nous allons essayer de déterminer un moyen d'arriver au résultat par une prise en compte des spécifications du problème.
L'idée sera de progresser pas-à-pas dans la résolution.
On part d'une liste non triée, aucun élément n'est à à sa place, on veut arriver à une liste triée, chaque élément est à sa place. Il semble raisonnable d'essayer de mettre un par un les éléments à leur place. On veut donc écrire une fonction

```
def trier(L):
    n = len(L)
    for i in range(n):
        # Les i plus petits éléments sont à leur place
        faire quelque chose
        # Les (i+1) plus petits éléments sont à leur place
```

On voit apparaître une propriété qui dépend de i, *"Les i plus petits éléments sont à leur place"*, qui doit être vraie au départ de chaque passage de la boucle, on peut la noter $\mathcal{P}(i)$.
De plus cette propriété est vraie au départ : demander que 0 élément soit à leur place n'est pas contraignant et, si elle est vraie à la fin de la boucle `for`, c'est que la liste est triée : les n éléments sont à leur place.
Une propriété avec ces caractéristiques est un **invariant de boucle**. C'est un outil important pour écrire des algorithmes corrects mais il est très souvent implicite. Si on parvient à énoncer précisément un invariant et à démontrer que les instructions que l'on écrit permettent de déduire $\mathcal{P}(i+1)$ de $\mathcal{P}(i)$, on a fait la preuve que notre algorithme fait bien ce qu'il est censé faire.

Il reste maintenant à savoir que faire dans la boucle, on est guidé par l'invariant.
On suppose qu'on part de `L = [15, 5, 2, 11, 3, 8]`.
On admet, qu'après 2 étapes, on soit parvenu à `L = [2, 3, 15, 11, 5, 8]` : $\mathcal{P}(i)$ est valide pour $i = 2$ car on a bien les deux plus petits éléments, 2 et 3, à leur place .
Pour arriver à valider $\mathcal{P}(i+1)$, c'est-à-dire $\mathcal{P}(3)$, on veut placer le suivant, 5 à sa position, c'est-à-dire à l'indice 2. Dans ce but

- on cherche l'indice, j, d'un élément minimal entre les positions 2 ($2 = i$) et la fin
- on échange les valeurs de `L[i]` et de `L[j]`



On remarque que la dernière opération est inutile : si $n-1$ éléments sont bien placés, le dernier est aussi bien placé.

Exercice 2 - Indice d'un minimum partiel
Écrire une fonction `indiceMinEntre(L, i, j)` qui renvoie un indice k en lequel la liste `L` admet une valeur minimale parmi les valeurs `L[i]`, `L[i+1]`, ..., `L[j]` : `L[k] <= L[r]` pour tout $r \in \{i, i+1, \ldots, j\}$.

On pourra supposer, sans avoir besoin de le vérifier, qu'on a $i \leqslant j$.
`indiceMinEntre(L, 2, 5)` doit renvoyer 4 pour `L = [2, 3, 15, 11, 5, 8]`.

Exercice 3 - Tri par sélection
En déduire un fonction `trier(L)`.

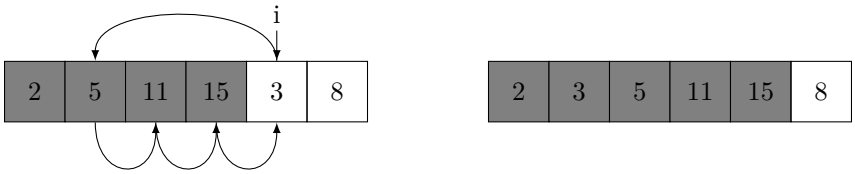
Exercice 4 - Complexité
Déterminer le nombre de comparaisons et le nombre d'échanges effectués par la fonction.

III Une autre méthode

La méthode précédente programmait le tri en construisant pas-à-pas le tableau trié.
On peut aussi partir du tableau à trier et ajouter terme-à-terme les éléments à trier. On aboutit alors à un autre invariant : les i premiers termes sont triés.

```
def trier(L):
    n = len(L)
    for i in range(n):
        # Les i premiers termes de la liste sont triés
        faire quelque chose
        # Les (i+1) premiers termes de la liste sont triés
```

On représente ci-après l'étape correspondant à $i = 4$. Les 4 premiers éléments, d'indices 0, 1, 2 et 3 sont triés et les suivants sont restés à leur place. On met en place l'élément d'indice i pour avoir $i+1$ éléments triés.



On remarquera que l'ordre des permutations est important ; dans l'exemple ci-dessus, on permute les indices 4 et 3 puis 3 et 2 puis 2 et 1. On ne permutera pas 1 et 0 car la valeur en 1, (3) est supérieure à la valeur en 0 (2).
On remarque que, pour $i = 0$, il n'y a rien à faire.

Exercice 5 - Mise en place
Écrire une fonction `mettreEnPlace(L, i)` qui reçoit une liste `L` et un indice i tels que **les i premiers éléments de `L` sont triés** et qui met en place l'élément d'indice i de telle manière que les $i + 1$ premiers éléments de `L` sont triés sans modifier les suivants.

Exercice 6 - Tri par insertion
En déduire un fonction `trier(L)`.

Exercice 7 - Complexité
Déterminer le nombre maximal de comparaisons et le nombre d'échanges effectués par la fonction. Quel est le nombre minimal ?

IV Tri de couples

Le plus souvent les objets à trier sont plus compliqués que de simples nombres.
Nous allons donner quelques applications pour lesquelles les objets sont des couples (ou des tuples quelconques) de nombres que l'on veut trier selon la première composante. On souhaitera que le tri soit stable, c'est-à-dire que, en cas d'égalité, les termes avec les mêmes premières composantes sont dans le même ordre à la fin.
`[(4, 7), (3, 8), (4, 2), (8, 3)]` devient `[(3, 8), (4, 7), (4, 2), (8, 3)]`.
On s'assurera que c'est bien le cas avec les fonctions à écrire.

Exercice 8 - Tri de couples
Modifier la fonction de tri de votre choix pour qu'elle trie des listes de couples selon la première composante.

On peut aussi trier selon l'ordre lexicographique $(a, b) < (c, d)$ si $a < c$ ou $a = c$ et $b < d$.

Exercice 9 - Tri de couples
Modifier la fonction de tri de votre choix pour qu'elle trie des listes de couples selon l'ordre lexicographique.

V Applications

Dans les exemples suivants on demandera de résoudre un problème dans le cas général et dans le cas de listes triées, en demandant un algorithme plus rapide dans le cas de listes triées.
On connait déjà cette situation avec la recherche dans une liste : si la liste n'est pas triée, la recherche se fait en examinant chaque élément, on doit faire n comparaisons pour constater qu'un élément n'appartient pas à une liste. Si la liste est triée le nombre de comparaisons est de l'ordre de $\log_2(n)$ avec la recherche par dichotomie.
Bien entendu le coût du tri est supérieur au gain si on ne fait qu'une recherche mais, le plus souvent, on fera de nombreuses recherches qui justifieront la complexité du tri.
De plus, dans certains cas, les tris plus efficaces qui seront étudiés plus tard rendent l'opération de tri rentable immédiatement.

V.1 Somme fixée

L'objectif est de déterminer s'il existe deux éléments dans une liste dont la somme vaut un nombre fixé.
On supposera que les éléments de la liste sont distincts.
Par exemple, pour la liste `L = [15, 5, 2, 11, 3, 8]`, on peut obtenir 16 avec $5+11$ mais on ne peut pas obtenir 21.

Exercice 10
Écrire une fonction `sommeDe2(liste, k)` qui renvoie la liste des couples d'indices `(i, j)` tels que `liste[i] + liste[j] = k` et `i < j`. La liste pourra être vide.

`sommeDe2(L, 13)` peut renvoyer `[(1, 5), (2, 3)]`.
Le nombre d'additions et de comparaisons sera de l'ordre de $\frac{n^2}{2}$ si n est la longueur de la liste.

Exercice 11
On suppose maintenant que la liste est triée.
Écrire une fonction `sommeDe2(liste, k)` qui renvoie la liste des couples d'indices `(i, j)` tels que `liste[i] + liste[j] = k` en faisant moins d'opérations.

V.2 Jointure

On considère deux listes de couples `L1` et `L1`.
Leur **jointure** selon la première composante est l'ensemble des triplets (a, b_1, b_2) tels que (a, b_1) est un élément de `L1` et (a, b_2) est un élément de `L1`.
Par exemple la jointure de `[(7, 6), (2, 8), (4, 5)]` et de `[(3, 9), (2, 11), (7, 5)]` peut être la liste `[(7, 6, 5), (2, 8, 11)]`, l'ordre des éléments dans la liste n'est pas imposé.

Exercice 12
Écrire une fonction `jointure(liste1, liste2)` qui calcule la jointure des deux listes passées en paramètre. Combien de comparaisons sont effectuées en fonctions des tailles n_1 et n_2 des listes ?

On suppose maintenant que les deux listes sont triées selon leur première composante.
Dans un premier temps on suppose que les premières composantes sont distinctes deux à deux dans chaque liste.

Exercice 13
Écrire une fonction `jointure(liste1, liste2)` qui calcule la jointure en effectuant au plus $2(n_1 + n_2)$ comparaisons.

Exercice 14
Peut-on obtenir un résultat semblable en ne supposant plus que les premières composantes sont distinctes deux à deux dans chaque liste ?

V.3 Intervalles

Dans cette partie les couples représentent un intervalle fermé : (a, b) représente $[a; b]$ et on impose $a \leqslant b$. À partir d'un ensemble d'intervalle on veut construire une représentation de l'union sous forme d'une union d'intervalles disjoints deux à deux. Par exemple, si la liste est `[(4, 12), (33, 41), (6, 8), (7, 16), (25, 38), (18, 21)]`, l'union des intervalles correspondants correspondants est $[4; 16] \cup [18; 21] \cup [25; 41]$.
On doit donc obtenir `[(4, 16), (18, 21), (25, 41)]`.
L'algorithme, dans le cas général, est assez difficile à écrire, par contre si on commence par trier les couples selon la première composante, on peut trouver un algorithme qui effectue un nombre de comparaisons majoré par la taille de la liste.

Exercice 15
Écrire une fonction `union(liste)` en supposant triée la liste de couple selon la première composante.

Solutions

Solution de l'exercice 1 - Permutation
Dans la fonction proposée la valeur de `L[i]` est perdue, `L[j]` est conservé.

```
def echanger(L, i, j):
    temporaire = L[i]
    L[i] = L[j]
    L[j] = temporaire
```

Python permet une écriture rapide mais parfois dangereuse, par exemple dans les matrices `numpy`.

```
def echanger(L, i, j):
    L[i], L[j] = L[j], L[i]
```

Solution de l'exercice 2 - Indice d'un minimum partiel

```
def indiceMinEntre(L, i, j):
    imin = i
    Lmin = L[i]
    for k in range(i+1, j+1): #i a déjà été vu
        if L[k] < Lmin:
            Lmin = L[k]
            imin = k
    return imin
```

Solution de l'exercice 3 - Tri par sélection

```
def trier(L)
    n = len(L)
    for i in range(n-1):
        j = indiceMinEntre(L, i, n-1)
        echanger(L, i, j)
```

Solution de l'exercice 4 - Complexité
Il y a $(n-1)+(n-2)+\cdots+1=\frac{n(n-1)}{2}$ comparaisons et $n-1$ échanges.

Solution de l'exercice 5 - Mise en place

```
def mettreEnPlace(L, i):
    while i > 0 and L[i-1] > L[i]:
        echanger(L, i, i-1)
        i = i - 1
```

On peut éviter les 3 affectations de `echanger` :

```
def mettreEnPlace(L, i):
    a = L[i]
    while i > 0 and L[i-1] > a:
        L[i] = L[i-1]
        i = i - 1
    L[i] = a
```

Solution de l'exercice 6 - Tri par insertion

```
def trier(L)
    n = len(L)
    for i in range(1, n):
        mettreEnPlace(L, i)
```

Solution de l'exercice 7 - Complexité

Au pire, on compare le terme avec tous ses prédécesseurs. On fait alors $(n-1)+(n-2)+\cdots+1 = \frac{n(n-1)}{2}$ comparaisons et autant d'échanges échanges.
Si la liste est déjà triée on ne fait que $n-1$ comparaisons et aucune permutation.

Solution de l'exercice 8 - Tri de couples

Ce sont les fonctions auxiliaires qui sont modifiées.

```
def indiceMinEntrePrem(L, i, j):
    imin = i
    Lmin = L[i][0]
    for k in range(i+1, j+1): #i a déjà été vu
        if L[k][0] < Lmin:
            Lmin = L[k]
            imin = k
    return imin
```

```
def trierPrem(L)
    n = len(L)
    for i in range(n-1):
        j = indiceMinEntreCouple(L, i, n-1)
        echanger(L, i, j)
```

```
def mettreEnPlacePrem(L, i):
    while i > 0 and L[i-1][0] > L[i][0]:
        echanger(L, i, i-1)
        i = i - 1
```

```
def trierPrem(L)
    n = len(L)
    for i in range(1, n):
        mettreEnPlacePrem(L, i)
```

Solution de l'exercice 9 - Tri de couples

On peut définir une fonction de comparaison :

```
def compare(c1, c2):
    (a1, b1) = c1
    (a2, b2) = c2
    return a1 < a2 or (a1 = a2 and b1 < b2)
```

```
def indiceMinEntreLexi(L, i, j):
    imin = i
    Lmin = L[i]
    for k in range(i+1, j+1): #i a déjà été vu
        if compare(L[k], Lmin) :
            Lmin = L[k]
            imin = k
    return imin
```

```
def trierLexi(L)
    n = len(L)
    for i in range(n-1):
        j = indiceMinEntreCouple(L, i, n-1)
        echanger(L, i, j)
```

```
def mettreEnPlaceLexi(L, i):
    while i > 0 and compare(L[i], L|i-1]):
        echanger(L, i, i-1)
        i = i - 1
```

```
def trierLexi(L)
    n = len(L)
    for i in range(1, n):
        mettreEnPlacePrem(L, i)
```

Solution de l'exercice 10

```
def sommeDe2(liste, k):
    reponse = []
    n = len(liste)
    for i in range(n):
        for j in range(i+1, n):
            if liste[i] + liste[j] == k:
                reponse.append((i, j))
    return reponse
```

Solution de l'exercice 11

```
def sommeDe2(liste, k):
    reponse = []
    a = 0
    b =  len(liste) -1
    while a < b:
        s = liste[a] + liste[b]
        if s == k:
            reponse.append((a, b))
            a = a + 1
            b = b - 1
        elif s < k:
            a = a + 1
        else:
            b = b - 1
    return reponse
```

$b - a$ diminue de 1 au moins à chaque étape et on fait 2 comparaisons et une addition au plus à chaque étape : il y a donc au plus $n - 1$ sommes et $2n - 2$ comparaisons.

Solution de l'exercice 12

```
def jointure(liste1, liste2):
    reponse = []
    n1 = len(liste1)
    n2 = len(liste2)
    for i in range(n1):
        for j in range(n2):
            a1, b1 = liste1[i]
            a2, b2 = liste2[j]
            if a1 == a2:
                reponse.append((a1, b1, b2))
    return reponse
```

On fait $n_1 n_2$ comparaisons

Solution de l'exercice 13

```
def jointure(liste1, liste2):
    reponse = []
    n1 = len(liste1)
    n2 = len(liste2)
    i1 = 0
    i2 = 0
    while i1 < n1 and i2 < n2:
        a1, b1 = liste1[i1]
        a2, b2 = liste2[i2]
        if a1 == a2:
            reponse.append((a1, b1, b2))
            i1 = i1 + 1
            i2 = i2 + 1
        elif a1 < a2:
            i1 = i1 + 1
        else:
            i2 = i2 + 1
    return reponse
```

Solution de l'exercice 14

```
def jointure(liste1, liste2):
    reponse = []
    n1 = len(liste1)
    n2 = len(liste2)
    i1 = 0
    i2 = 0
    while i1 < n1 and i2 < n2:
        a1, b1 = liste1[i1]
        a2, b2 = liste2[i2]
        if a1 == a2:
            listeb1 = [b1]
            listeb2 = [b2]
            i1 = i1 + 1
            i2 = i2 + 1
            while i1 < n1 and liste1[i1][0] == a1:
                listeb1.append(liste1[i1][1])
                i1 = i1 + 1
            while i2 < n2 and liste1[i2][0] == a1:
                listeb2.append(liste1[i2][1])
                i2 = i2 + 1
            for y1 in listeb1:
                for y2 in listeb2:
                    reponse.append((a1, y1, y2))
        elif a1 < a2:
            i1 = i1 + 1
        else:
            i2 = i2 + 1
    return reponse
```

Solution de l'exercice 15

```
def union(liste):
    reponse = []
    n = len(liste)
    i = 0
    while i < n:
        a, b = liste[i]
        i = i + 1
        while i < n and liste[i][0] <= b:
            b = max(b, liste[i][1])
            i = i +1
        reponse.append((a, b))
    return reponse
```