Recherche rapide

1. Introduction

Considérons une liste d'éléments triés, par exemple une liste de nombres entiers comme celle-ci, contenant 10 nombres générés aléatoirement entre 0 et 100 puis triés :

```
[2, 13, 14, 18, 21, 26, 44, 49, 77, 93]
```

On souhaite faire une recherche dans cette liste :

- ▷ rechercher un nombre dans la liste ;
- ▷ rechercher tous les nombres contenus dans un intervalle donné.

Plus généralement, on peut être amené à faire des recherches dans une liste d'objets comportants plusieurs attributs. Les recherches dans les bases de données sont de ce type. Dans un autre domaine, l'algorithme de *lancer de rayon* nécessite de rechercher rapidement des objets localisés dans une certaine région de l'espace.

Pour effectuer une recherche d'un élément particulier dans une liste de nombres entiers, la méthode de recherche linéaire consiste à parcourir la liste jusqu'à rencontrer l'élément cherché ou bien jusqu'à la fin si l'élément cherché n'est pas dans la liste. Cette méthode est en moyenne de complexité temporelle $O(N)$. Il en est de même de la recherche des éléments contenus dans un intervalle donné, à condition que la liste soit triée au préalable. Nous allons étudier une méthode de recherche plus efficace.

2. Arbre binaire d'intervalles

2.a. Principe

Nous définissons la valeur médiane d'une liste comme la valeur d'indice int(N/2) de la liste triée, où N est la taille de la liste.

La recherche rapide repose sur le principe d'une dichotomie récursive par la valeur médiane. On pourrait effectuer cette dichotomie récursive à chaque recherche, mais il est plus efficace de la faire au préalable en introduisant une nouvelle *structure de données*, appelée *arbre binaire*.

Considérons la valeur médiane de la liste donnée en exemple (26), et divisons la liste L en deux sous-listes L_1 et L_2, la première contenant tous les éléments strictement inférieurs à la valeur médiane, la seconde contenant tous les éléments supérieurs ou égaux à la valeur médiane. Nous obtenons ainsi :

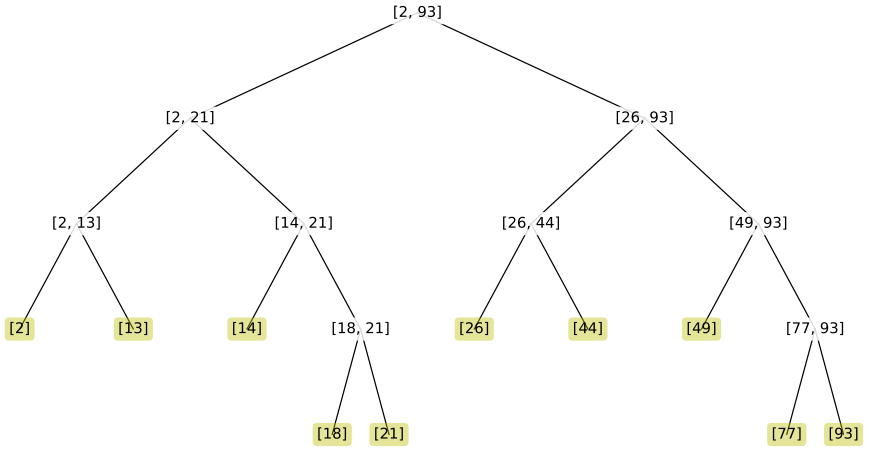
```
L1=[2,13,14,18,21], L2=[26,44,49,93]
```

La première liste contient tous les éléments de l'intervalle `[2,21]`, la seconde contient ceux de l'intervalle `[26,93]`. Continuons récursivement cette division sur les deux sous-listes. Par exemple, la liste `L1` est divisée autour de la valeur médiane (14) en

```
L11=[2,13], L12=[14,18,21]
```

La division de `L11` donne les deux éléments 2 et 13. La division de `L12` donne l'élément 14 et la liste `L122=[18,21]`, dont la division donne les éléments 18 et 21. Le processus récursif de division s'arrête lorsqu'on parvient à une liste à un élément (ou éventuellement quelques éléments comme nous le verrons plus loin).

Le résultat de ce processus de division est stocké dans un arbre binaire, appelé *arbre binaire d'intervalles*. Voici une représentation de l'arbre associé à la liste prise en exemple :



Chaque nœud est caractérisé par un intervalle, car le sous-arbre descendant du nœud contient tous les éléments contenus dans cet intervalle. Les feuilles, c'est-à-dire les nœuds sans descendant, contiennent les éléments. Dans un arbre binaire, chaque nœud (autre qu'une feuille) a deux enfants : un nœud de gauche et un nœud de droite. La racine est le nœud situé tout en haut ; il est bien sûr associé à l'intervalle contenant tous les éléments de la liste. L'arbre descendant d'un nœud est aussi appelé le sous-arbre de ce nœud. Pour chaque nœud, la valeur médiane utilisée pour la division est la borne inférieure de l'intervalle de son nœud enfant de droite.

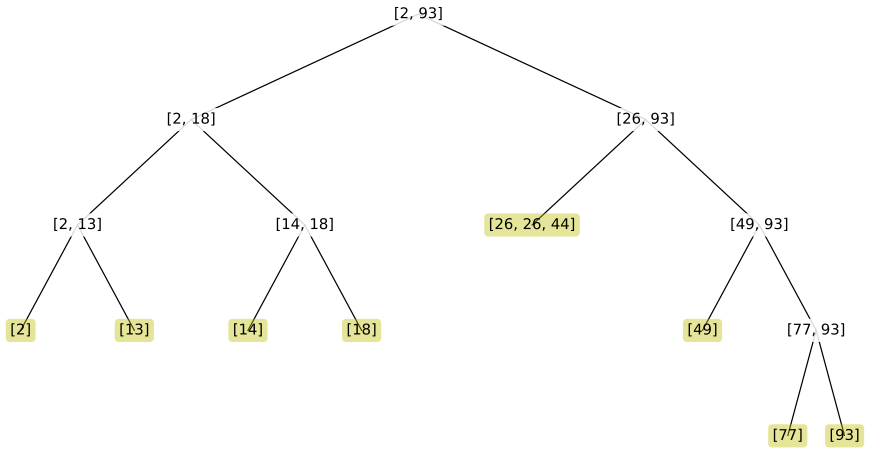
La profondeur de l'arbre est le nombre maximal d'arêtes qu'il faut parcourir en partant de la racine pour parvenir à une feuille. Pour cet exemple, la profondeur est 4. Afin de trouver comment la profondeur de l'arbre dépend du nombre N d'éléments, considérons le cas où N est une puissance de deux, c'est-à-dire $N = 2^p$. Les N feuilles descendent de 2^{p-1} nœuds, lesquels descendent de 2^{p-2} nœuds, etc. Le nombre de niveaux jusqu'à la racine est donc p. Sachant que $p = ln(N)/ln(2)$, la profondeur de l'arbre est proportionnelle à $ln(N)$.

Une fois l'arbre associé à une liste construit, nous pouvons faire des recherches en partant de la racine. Considérons la recherche d'un élément donné, par exemple le nombre 44. La comparaison de ce nombre avec la valeur médiane de la liste (26) nous oriente vers le nœud enfant de droite (intervalle [26,93]). Arrivé à ce nœud, la comparaison avec la valeur médiane (49) nous oriente vers le nœud enfant de gauche. Finalement, la valeur médiane associée à ce dernier nœud (44) nous oriente vers son enfant de droite. Celui-ci se résume à un élément, qui est comparé à celui que l'on cherche.

Ce type d'algorithme de parcours d'un arbre binaire a une complexité proportionnelle à la profondeur de l'arbre, c'est-à-dire $O(ln(N))$, ce qui en fait une méthode beaucoup plus efficace que la recherche linéaire. Par exemple pour N=1000, le rapport $ln(N)/N$ vaut moins de 10^{-2}.

La liste étant déjà triée, il est bien sur possible d'accomplir la recherche avec dichotomie par la médiane sans avoir construit un arbre. Nous verrons cependant que la généralisation à un espace de dimension quelconque nécessite la construction préalable d'un arbre.

Remarquons que si certains éléments de la liste sont égaux, une feuille peut contenir plusieurs éléments. Voici par exemple l'arbre obtenu lorsqu'on remplace l'élément 21 par 26.



La liste [26,26,44] ne peut pas être divisée car sa valeur médiane est 26.

2.b. Construction de l'arbre d'intervalles

L'implémentation d'un arbre en Python peut se faire en définissant une classe pour un nœud :

```
class Noeud:
    def __init__(self):
        self.x_median = 0
        self.liste_x = [] # réservé au débogage
        self.x_min = 0
        self.x_max = 0
        self.noeud_gauche = None
        self.noeud_droite = None
```

Les données associées à un nœud sont : la valeur médiane utilisée pour la division, la liste des éléments associés au nœud, les bornes de l'intervalle. Il n'est pas vraiment nécessaire de stocker dans chaque nœud la liste des éléments mais nous la mettons pour le déboguage. De même, les bornes de l'intervalle ne sont pas indispensables car on peut facilement les obtenir en parcourant l'arbre. Seul le stockage de la valeur médiane est indispensable.

Un nœud (non feuille) contient aussi les deux nœuds enfants. Par défaut, il n'y a pas de nœuds enfants (objet `None`). Remarquons qu'un nœud a soit deux enfants soit aucun. S'il n'en a pas, il s'agit d'une feuille contenant un ou plusieurs éléments dans `liste_x`.

La construction de l'arbre se fait en utilisant la récursivité. Pour cela, nous écrivons une fonction qui génère le sous-arbre d'un nœud donné, fonction qui prend aussi pour argument la liste associée.

```
def generation_arbre(noeud,liste_x):
    n = len(liste_x)
    noeud.liste_x = liste_x
    noeud.x_min = liste_x[0]
    noeud.x_max = liste_x[n-1]
    if n==1:
        return
    else:
        m = int(n/2)
        x_median = liste_x[m]
        noeud.x_median = x_median
        i = m-1
        while i>=0 and liste_x[i]==x_median:
            i -= 1
        i+=1
        if i>0:
            liste_gauche = liste_x[0:i]
            noeud.noeud_gauche = Noeud()
            generation_arbre(noeud.noeud_gauche,liste_gauche)
            liste_droite = liste_x[i:n]
            noeud.noeud_droite = Noeud()
            generation_arbre(noeud.noeud_droite,liste_droite)
        else:
            return
```

Les quatre premières lignes mémorisent dans le nœud la liste et les bornes de l'intervalle. Si la liste ne contient qu'un élément, c'est que le nœud est une feuille et il faut donc stopper la récursion. Dans le cas contraire, la valeur médiane est obtenue puis on recherche d'éventuels éléments égaux à la médiane mais situés avant dans la liste. Si la liste est effectivement divisée (`i>0`) alors la liste de gauche est extraite et un enfant nœud de gauche est créé (avec des valeurs par défaut). La fonction `generation_arbre` est alors appelée récursivement pour remplir ce nœud et le sous-arbre associé. On procède de même pour le nœud enfant de droite. Si la liste n'est pas divisée, c'est parce-qu'elle contient des éléments égaux, et la récursion s'arrête avec une liste comportant plus d'un élément.

2.c. Travaux pratiques

Télécharger le script arbre-1d.py

Écrire une fonction `rechercher_valeur(noeud,valeur)` qui recherche récursivement une valeur dans un nœud. Cette fonction doit renvoyer `True` si la valeur est trouvée parmi les éléments, `False` dans le cas contraire.

Générer une liste de nombres entiers avec la fonction `random.randrange(a,b)`. Trier cette liste avec la fonction `sort`, qui s'applique à une liste `L` par la syntaxe : `L.sort()`. Générer l'arbre associé à cette liste et tester la fonction `rechercher_valeur`.

Écrire une fonction `obtenir_arbre(noeud,pile)` qui place dans une pile (initialement vide) tous les éléments contenus dans le sous-arbre associé à un nœud. La fonction doit aller chercher les éléments dans les feuilles de l'arbre et non pas dans les listes stockées dans les nœuds (qui ne sont utilisées que pour le débogage).

Concevoir un test permettant de savoir si deux intervalles fermés ont une intersection non vide.

Concevoir un algorithme récursif de parcours de l'arbre permettant de rechercher tous les éléments contenus dans un intervalle $[a, b]$ donné. Écrire une fonction `rechercher_dans_intervalle(noeud,a,b,pile)` qui effectue cette recherche et place les éléments trouvés dans une pile.

2.d. Solution

```
def rechercher_valeur(noeud,valeur):
    if noeud.noeud_gauche!=None and noeud.noeud_droite!=None:
        if valeur < noeud.x_median:
            return rechercher_valeur(noeud.noeud_gauche,valeur)
        else:
            return rechercher_valeur(noeud.noeud_droite,valeur)
    else:
        for x in noeud.liste_x:
            if valeur==x:
                return True
        return False

def obtenir_arbre(noeud,pile):
    if noeud.noeud_gauche==None and noeud.noeud_droite==None:
        for x in noeud.liste_x:
            pile.append(x)
    else:
        obtenir_arbre(noeud.noeud_gauche,pile)
        obtenir_arbre(noeud.noeud_droite,pile)

def rechercher_dans_intervalle(noeud,a,b,pile):
    if noeud==None:
        return
    if noeud.x_min >= a and noeud.x_max <= b:
        obtenir_arbre(noeud,pile)
    else:
        if not((a<noeud.x_min and b<noeud.x_min) or (a>noeud.x_max and b>noeud.x_max)
```

```
rechercher_dans_intervalle(noeud.noeud_gauche,a,b,pile)
rechercher_dans_intervalle(noeud.noeud_droite,a,b,pile)
```

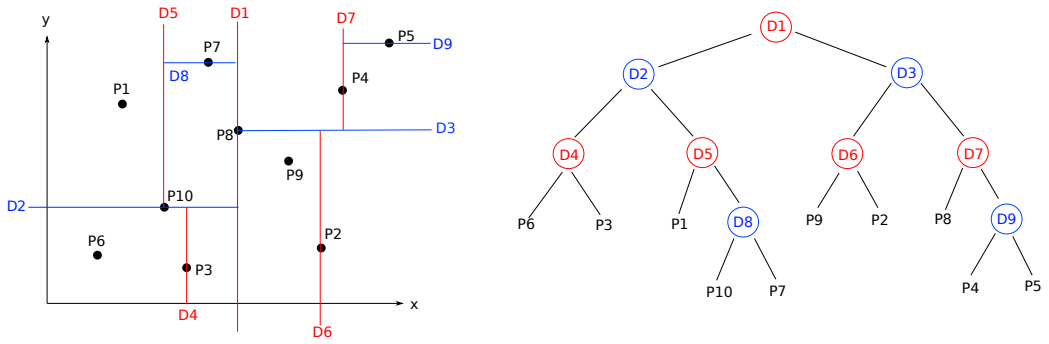
3. Recherche dans un espace de dimension K

3.a. Principe

Une recherche rapide dans un espace de dimension K peut être accomplie en construisant un arbre k-d (k dimensional tree). Considérons le cas K=2 avec par exemple des points placés dans le plan, chaque point ayant deux coordonnées (x, y) réelles. Il s'agira par exemple de rechercher tous les points situés dans un rectangle donné.

L'arbre k-d est une généralisation de l'arbre de recherche décrit précédemment pour un espace de dimension 1. Sa construction consiste à séparer les points alternativement autour de la médiane des valeurs de x puis de la médiane des valeurs de y. Autrement dit, l'axe choisi pour effectuer la séparation dépend de manière cyclique de la profondeur atteinte dans l'arbre.

La figure suivante montre un exemple avec 10 points :



On a représenté chaque nœud par la droite (ou demi-droite ou segment) de séparation qui lui est associée, qui sert à générer ses deux nœuds enfants. Les nœuds réalisant une division dans la direction X sont représentés en rouge. Les nœuds réalisant une division dans la direction Y sont en bleu. Chaque nœud correspond à un domaine du plan. Par exemple le nœud D2 correspond au demi-plan situé à gauche de la droite D1 : tous les points de son sous-arbre appartiennent à ce domaine.

La séparation dans la direction X se fait avec la valeur médiane des coordonnées des points. Nous utilisons la même convention que pour le cas 1-d : la valeur médiane est l'élément d'indice int(N/2) de la liste des points rangés par valeur croissante de X et les éléments dont l'abscisse est égale à la valeur médiane sont placés à droite.

La généralisation à un arbre de dimension K est immédiate : l'arbre est construit en alternant de manière cyclique la séparation selon les K dimensions. Une application de l'arbre k-d est le partitionnement de l'espace (K=3) pour localiser rapidement des objets appartenant à un certain domaine.

3.b. Obtention de la valeur médiane selon un axe

L'obtention de la valeur médiane (soit de x soit de y) est plus délicate que dans le cas 1-d car il ne suffit plus de trier la liste au départ. Heureusement, il existe un algorithme qui permet d'obtenir la médiane d'une liste en temps linéaire (complexité temporelle

O(N)). Une fois la médiane obtenue, il est aisé de faire la séparation de la liste en temps linéaire. Contrairement au cas 1-d où l'obtention de la médiane d'une liste déjà triée est de complexité O(1), la complexité O(N) rend impérative la construction préalable d'un arbre de recherche, qui permet de se dispenser de la recherche de médiane au moment de la recherche, l'objectif étant bien sûr d'effectuer un grand nombre de recherches sur un arbre donné.

L'algorithme de sélection de la médiane (ou plus généralement d'un élément de rang donné dans une liste) repose sur le même principe que l'algorithme de tri rapide par partitionnement. Le partitionnement consiste à choisir un pivot dans la liste, par exemple le dernier élément, et à réarranger la liste pour que tous les éléments inférieurs ou égaux au pivot soient situés avant le pivot et tous les éléments supérieurs au pivot soient situés après le pivot.

La fonction suivante effectue le partitionnement d'une sous-liste d'une liste donnée (entre les indices `debut` et `fin`). La fonction renvoie l'indice de la position finale du pivot dans la sous-liste traitée. La liste traitée est en fait une liste de points, chacun étant une liste de k nombres. On fournit aussi l'axe sur lequel se fait la recherche (0 pour x, 1 pour y, etc.)

```
def partition(liste,axe,debut,fin):
    pivot = liste[fin][axe]
    i = debut
    for j in range(debut,fin):
        if liste[j][axe]<=pivot:
            liste[i],liste[j] = liste[j],liste[i]
            i += 1
    liste[i],liste[fin] = liste[fin],liste[i]
    return i
```

Les deux sous-listes implicitement obtenues sont `L1=liste[debut:debut+i]` et `L2=liste[debut+2:fin]`, le pivot étant l'élément `liste[debut+i]`.

L'algorithme de sélection d'un élément de rang r donné est récursif. Soit k la longueur de L_1, c'est-à-dire la position du pivot dans la sous liste traitée, relativement au premier élément de cette sous-liste. Il y a trois cas à considérer :

- ▷ Si $k = r$, le pivot est l'élément de rang r cherché.
- ▷ Si $k > r$, l'élément cherché se trouve dans la sous-liste L_1, au rang r.
- ▷ Si $k < r$, l'élément cherché se trouve dans la sous-liste L_2 au rang $r - k$.

Dans les deux derniers cas, la fonction de sélection est appelée récursivement, respectivement sur les listes L_1 et L_2.

Voici la fonction de sélection :

```
def selection(liste,axe,debut,fin,rang):
    if debut==fin:
        return liste[debut][axe]
    i = partition(liste,axe,debut,fin)
    k = i-debut+1
    if rang==k:
```

```
        return liste[i][axe]
    elif rang < k:
        return selection(liste,axe,debut,i-1,rang)
    else:
        return selection(liste,axe,i+1,fin,rang-k)
```

et la fonction d'appel pour obtenir la valeur médiane :

```
def mediane(liste,axe):
    return selection(liste.copy(),axe,0,len(liste)-1,int(len(liste)/2)+1)
```

Pour tester cette fonction, on génère une liste de points du plan aléatoirement :

```
import random
N=10
K=2
points = []
for i in range(N):
    p = []
    for axe in range(K):
        p.append(random.random())
    points.append(p)

mediane_x = mediane(points,0)


print(mediane_x)
--> 0.5964783357384704
```

Nous avons aussi besoin d'une fonction qui génère deux sous-listes, la première contenant tous les éléments dont l'abscisse sur l'axe est strictement inférieure à la médiane, la seconde contenant ceux dont l'abscisse est supérieure ou égale à la médiane. Cela se fait en parcourant linéairement la liste :

```
def separation_mediane(liste,axe):
    m = mediane(liste,axe)
    N = len(liste)
    K = len(liste[0])
    L1 = []
    L2 = []
    for i in range(N):
        if liste[i][axe] < m:
            L1.append(liste[i])
        else:
            L2.append(liste[i])
    return (m,L1,L2)
(m,L1,L2) = separation_mediane(points,0)
```

```
print(numpy.array(L1))
--> array([[ 0.58821768,  0.65875766],
       [ 0.59203442,  0.63751161],
       [ 0.27319816,  0.58534503],
       [ 0.44497768,  0.62637128],
       [ 0.43733262,  0.94277302]])

print(numpy.array(L2))
--> array([[ 0.68740957,  0.77541864],
       [ 0.7135312 ,  0.12135793],
       [ 0.88632483,  0.60759237],
       [ 0.59647834,  0.46372686],
       [ 0.61878617,  0.43242965]])
```

3.c. Génération de l'arbre k-d

On définit tout d'abord une classe pour les nœuds de l'arbre. Celle-ci est similaire à celle déjà définie pour le cas 1-d, mais nous mémorisons dans chaque nœud l'axe utilisé pour la génération de ses deux enfants.

```
class NoeudKd:
    def __init__(self):
        self.axe = 0
        self.valeur_mediane = 0
        self.liste_points = [] # réservé au débogage
        self.noeud_gauche = None
        self.noeud_droite = None
```

Nous avons enlevé les valeurs minimale et maximale car elle ne sont plus immédiatement accessibles (la liste n'est pas triée). La seule information qu'il importe vraiment de stocker est la valeur médiane, car son obtention est coûteuse en temps de calcul.

3.d. Travaux pratiques

Télécharger le script arbre-kd.py

Écrire la fonction `generation_arbre-kd(noeud,liste_points,K,profondeur=0)`, permettant de générer l'arbre récursivement. La profondeur, initialement nulle, doit être incrémentée d'une unité à chaque appel récursif. L'axe de la division est le reste de la division entière de la profondeur par K (`profondeur%K`).

On se place dans le cas particulier K=2. Écrire une fonction `recherche_domaine(noeud,xmin,ymin,` qui recherche les points situés dans le domaine défini par $(x > x_{min}, y > y_{min})$. La fonction doit ajouter les points trouvés dans `pile`.

3.e. Solution

```
def generation_arbre_kd(noeud,liste_points,K,profondeur=0):
    noeud.liste_points = liste_points
    if len(liste_points)==1:
        return
    else:
```

```
axe = profondeur%K
(m,L1,L2) = separation_mediane(liste_points,axe)
noeud.valeur_mediane = m
noeud.axe = axe
if len(L1)!=0:
    noeud.noeud_gauche = NoeudKd()
    generation_arbre_kd(noeud.noeud_gauche,L1,K,profondeur+1)
if len(L2)!=0:
    noeud.noeud_droite = NoeudKd()
    generation_arbre_kd(noeud.noeud_droite,L2,K,profondeur+1)
```