

Rapport Projet Final Advanced Data Structures and Algorithms

DU PELOUX Diane - GALBEZ Flavien

December 19, 2019



Nous avons réalisé l'ensemble du projet de conquête du Joker de la ville de Gotham, jusqu'au bout du monde. En passant par les algorithmes de Prim, Djikstra, la recherche dichotomique, et ..., nous avons répondu à toutes les problématiques que nous présenteront dans ce rapport, suivis des solutions. Nous allons structurer ce document en quatre grandes parties (représentant les quatre parties du projet) composées de 3 sous-parties chacune (représentant chaque question du sujet).

1 Connecting the people

Dans cette partie, nous avons pour but de créer un graphique reliant toutes les stations de Gotham city en indiquant, pour chaque lien tracé, la distance entre les deux stations reliées. Puis dans une seconde partie nous aurons à afficher le plus court chemin reliant toutes les stations sans relier deux noeuds déjà liés au chemin tracé.

1.1 Create and display a graph representing the Gotham City's railways

Pour créer un graphique représentant toutes les liaisons et les stations affichées sur la map ci-dessous (plus précisément les stations comprises à l'intérieur du carré tracé), nous avons dû écrire notre propre fichier "csv" (affiché ci-dessous). Celui-ci est composé de trois colonnes: la première comprend la station de départ, la seconde la station d'arrivée, et la troisième la distance séparant les deux stations. Nous précisons que les distances ont été estimées, la plus petite correspondant à 1, la plus grande à 4 et les autres déduites proportionnellement à ces distances.



chinadocks	stevensburg	2
chinadocks	stockely	1
chinadocks	wharlow	2
stevensburg	duelish	1
stevensburg	the gantry	4
stevensburg	merchants sq	3
wharlow	harlow	1
wharlow	sheal br	4
haysville	newstreet	1
haysville	murtagh	2
newstreet	privvy	3
standrews	the narrows	2
murtagh	the narrows	1
the narrows	wayne centra	1
the narrows	prosper st	1
prosper st	wayne centra	1

Nous avons ensuite lu ce csv (code ci-dessous) et établit une liste des stations chacune comprenant une liste de leurs stations reliées, associées elles-même à une distance qui leur est propre.

Indice	Type	Taille	Valeur
0	list	2	['e city park', [...], [...], [...], [...]]
1	list	2	['23 st', []]
2	list	2	['n city', []]
3	list	2	['north g', []]
4	list	2	['good st', []]
5	list	2	['merchant', []]
6	list	2	['standree', []]
7	list	2	['gainsly east', [...], [...], [...]]
8	list	2	['newman', [...], [...], [...], [...]]

Souhaitant afficher un graphique en dehors de la console, nous avons générée une image avec des icônes de stations de métro et des liens reliant chaques icônes aux icônes des stations originellement reliées. Les pondarations sont ajoutées au centre des liens grâce à la valeur de "centrepondération" et affichés dans la boucle "dessinpondération".

```
def fonction(dessin_stations,dessin_arbre,dessin_ponderations,melange):
    global liste_des_villes,s

    #Je crée une image de dimension lxl.
    l,L=5000,5000
    image = Image.new('RGB', (l, L), color="white")
    draw = ImageDraw.Draw(image)

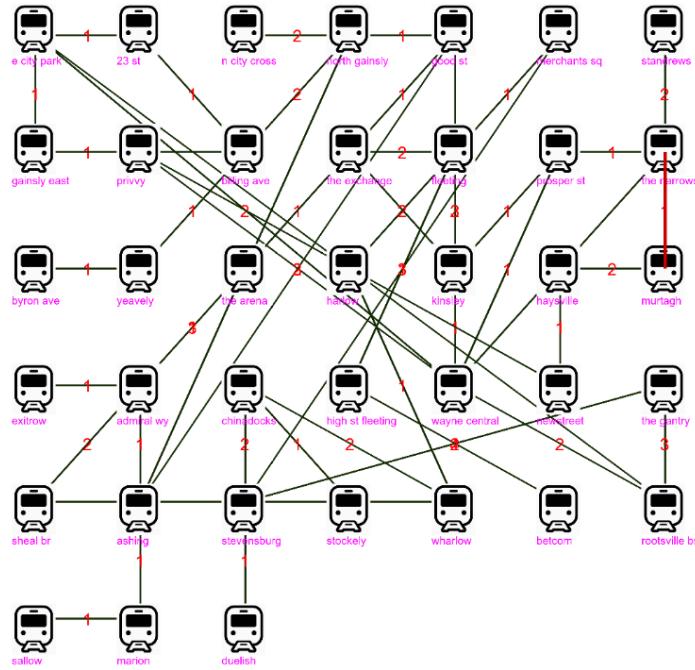
    #J'arrache les différentes stations avec leurs noms et le fais les co
    #par dessus les traits.
    subway = Image.open("subway.png")
    font = ImageFont.truetype("arial.ttf", 65)
    font2 = ImageFont.truetype("arial.ttf", 100)
    compteur = 0
    position = []
    for i in range(8):
        for j in range(7):
            if compteur <len(s):
                position += [[int((i+1)*l/9),int((j+1)*L/8)]]
    position = position[:len(s)]
```

Puis nous affichons les noms de stations aux icônes et nous affichons les liens correspondant grâce au code suivant:

```
#Ici j'affiche les stations si souhaité
for i in s:
    for j in i[1]:
        [b,a] = position[liste_des_villes.index(i[0])]
        [d,c] = position[liste_des_villes.index(j[0])]
        if dessin_arbre:
            draw.line((a,b,c,d), fill=10000,width=10)
            centre_ponderation = ((a+c)/2-30,(b+d)/2-50)
        if dessin_ponderations:
            draw.text(centre_ponderation,donner_distance(i[0],j[0]),fill=(255,0,0),font=font2)

#Ici j'affiche les liens si souhaité
if dessin_stations:
    compteur = 0
    for j in range(6):
        for i in range(7):
            if compteur <len(s):
                image.paste(subway,box=(int((i+1)*l/8-subway.size[0]/2),int((j+1)*L/7-subway.size[1]/2)))
                draw.text((int((i+1)*l/8-subway.size[0]/2),int((j+1)*L/7+150)),liste_des_villes[compteur],fill=(255,0,255),font=font)
            compteur += 1
```

Nous obtenons alors, le graphique suivant :



Pour l'obtenir il suffit de compiler "positionner.py" qui générera une image dans le dossier où le .py est situé et cela générera la solution dans le dossier "gif_avec_stations_avec_chemin".

1.2 Which kind of algorithm create a connected graph while minimizing the total amount of distance? Show the algorithm? What is its complexity?

"En théorie des graphes, étant donné un graphe non orienté connexe dont les arêtes sont pondérées, un arbre couvrant de poids minimal (ACM) de ce graphe est un arbre couvrant (sous-ensemble qui est un arbre et qui connecte tous les sommets ensemble) dont la somme des poids des arêtes est minimale (c'est-à-dire de poids est inférieur ou égal à celui de tous les autres arbres couvrants du graphe)." (Wikipédia)

Notre problème correspond complètement à ce cas, nous devons trouver une distance minimale de trajet reliant toutes les stations de la ville. Pour cela nous avons un graphique que nous allons parcourir avec l'algorithme de Prim qui nous permettra d'obtenir l'ACM. L'algorithme de Prim calcule un arbre couvrant minimal dans un graphe connexe valué et non orienté. La complexité temporelle de l'approche est $O(V^2)$ car nous utilisons une matrice de contiguïté. Celle-ci est composée de toutes les branches candidates possibles pour la prochaine décision de chemin à prendre.

Ci-dessous nous allons présenter étape par étape notre algorithme de Prim. Voici le corps de notre algorithme de Prime :

```
#Ici on peut lire l'algorithme de prime
choix_depart = choice(liste_des_villes)
arbre_branches = [choix_depart], []
while len(arbre) < len(liste_des_villes):
    candidats = chercher_les_noeuds_candidats(arbre, branches)
    branches += [selectionner_le_meilleur_candidat(candidats)]
    arbre = lister_les_noeuds_comptes(branches)
```

Nous pouvons observer la valeur choix de départ correspondant au noeud de départ et l'initialisation de nos deux listes, l'une comprenant les noeuds parcourus et l'autre les branches utilisées (noeuds + distance). Dans la boucle "while" nous avons trois appels de fonction qui sont les suivantes:

```

def chercher_les_noeuds_candidats(liste,selectionnes):
    global s
    candidats = []
    for i in liste:
        for j in s:
            if i == j[0]:
                for k in j[1]:
                    if not liste.count(k[0]):
                        candidats += [[i]+k]
    candidats = trier(candidats)
    filtre = []
    for i in candidats:
        if not selectionnes.count(i):
            filtre += [i]
    return filtre

def selectionner_le_meilleur_candidat(candidats):
    for i in range(len(candidats)):

        candidats[i][0],candidats[i][2] = candidats[i][2],candidats[i][0]
    candidats.sort()
    return [candidats[0][2],candidats[0][1],candidats[0][0]]

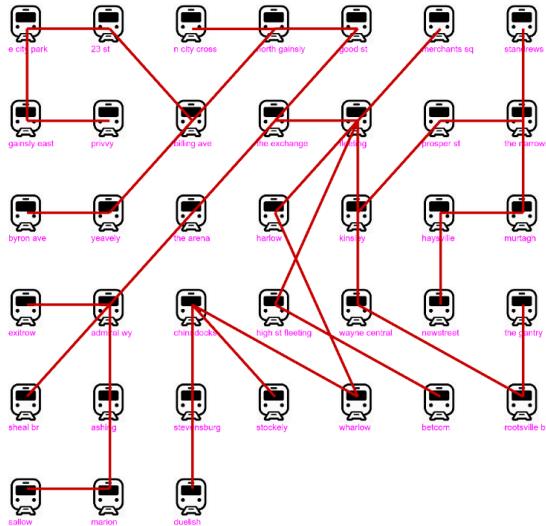
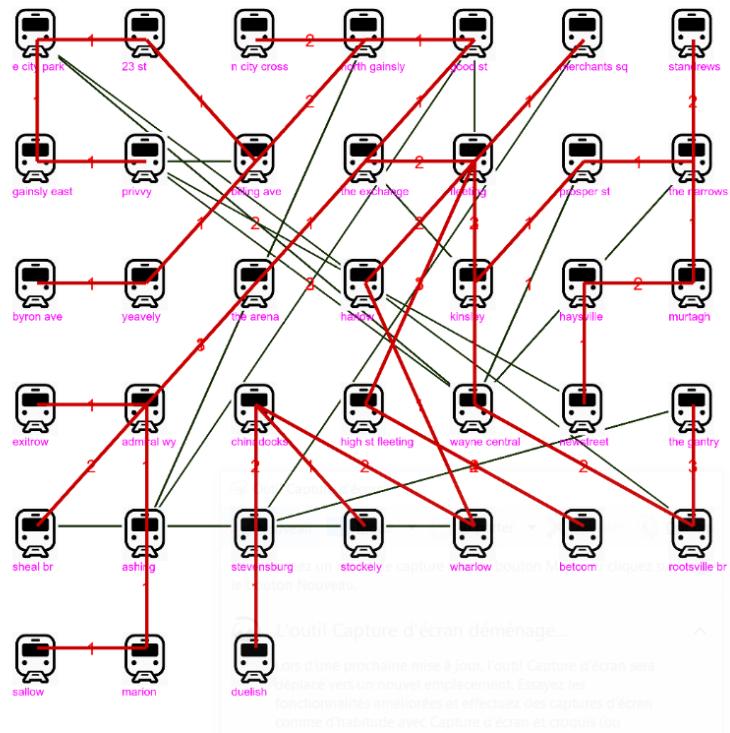
def lister_les_noeuds_comptes(liste):
    s = []
    for i in liste:
        if not s.count(i[0]):
            s += [i[0]]
        if not s.count(i[1]):
            s += [i[1]]
    return s

```

La première nous permet de chercher tous les noeuds qui pourront être concernés par le prochain chemin à tracer, la deuxième selectionne le chemin le plus intéressant en fonction de son poids, et la dernière renvoie les noeuds qui ont été parcourus. Ainsi dans la liste "branche" nous auront tous les chemins utilisés et dans "arbre" tous les noeuds parcourus. La boucle s'arrêtera quand la taille de l'arbre correspondra à la taille de la liste originale des stations.

1.3 Show and display a solution of the connected network

La solution obtenue est la suivante :



2 Spread the revolution

A notre disposition, nous avons une matrice des distances entre prétendues villes. Notre objectif, déterminer la distance la plus courte entre deux villes, est-ce un chemin direct ou une voie passant par une autre ville ? A nous de le découvrir !

2.1 Which kind of algorithm create a connected graph while minimizing the distance to a unique city? What is its complexity?

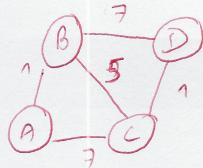
L'algorithme que nous allons utiliser pour créer un graphe connecté tout en minimisant la distance à une cité est l'algorithme de Djikstra. En théorie des graphes, l'algorithme de Dijkstra sert à résoudre le problème du plus court chemin. Il permet, par exemple, de déterminer un plus court chemin pour se rendre d'une ville à une autre connaissant le réseau routier d'une région. Plus précisément, il calcule des plus courts chemins à partir d'une source dans un graphe orienté pondéré par des réels positifs. On peut aussi l'utiliser pour calculer un plus court chemin entre un sommet de départ et un sommet d'arrivée. et algorithme est de complexité polynomiale. Plus précisément, pour n noeuds et a arcs, le temps est en $O((a+n)/\log n)$, voire en $O(a+n/\log n)$. Dans le cas du problème la compléxité serait de $O(100/\log 100)$.

2.2 Show and display a solution of the proposed algorithm.

Ci-dessous une explication détaillée avec schéma pour expliquer le code qui a été rendu (et aussi commenté).

Pour procéder à la recherche, je vais utiliser cette matrice :

$[None, None, None, None]$
A B C D



Je choisi de partir de A :



$[0, None, True], None, None]$

↑ ↑
"nous ne trouverons pas de chemin plus court pour aller en A, non à 100%"
pour aboutir au chemin qui donne cette distance, il faut avoir
fait le chemin qui a abouti jusqu'à None, puis de None à A.

A est le dernier élément pour lequel j'ai affirmé avoir trouvé le plus court chemin.
Je vais lancer une recherche partant de là :

- je pense aller en B. Venant de A j'aurai parcouru une distance de $0+1$
- je pense aller en C. Venant de A j'aurai parcouru une distance de $\frac{dist AB}{0+1}$
 $dist AC$

J'avais $[0, None, True], None, None]$

Je corrige ma proposition pour B : $(1, A, False)$ (voir *) à avant chemin,
c'est donc une meilleure proposition :

$[0, None, True], (1, A, False), (7, A, False), None]$

Maintenant que j'ai procédé à la recherche partant de A, je valide le plus court des chemins non déjà validé :

$[0, None, True], (1, A, True), (7, A, False), None]$

①

B est le dernier élément pour lequel j'ai affirmé avoir trouvé le plus court chemin. Je vais lancer une recherche partant de là :

- je peux aller en A. Mais A a déjà été validé, je ne trouverai pas plus court pour l'atteindre.

- je peux aller en C. Venant de B j'aurais parcouru une distance $1 + \cancel{5} = 8$
dist B-C

- je peux aller en D. Venant de B j'aurais parcouru une distance $1 + 7 = 8$

Je compare ma proposition pour C. $(\cancel{8}, B, \text{False})$ à celle déjà établie :
 $(7, A, \text{False})$

La nouvelle proposition est meilleure.

j'avais : $[(0, \text{None}, \text{True}), (1, A, \text{True}), (7, A, \text{False}), \text{None}]$

j'ai maintenant : $[(0, \text{None}, \text{True}), (1, A, \text{True}), (6, B, \text{False}), (\cancel{8}, B, \text{False})]$

Maintenant que j'ai procédé à une recherche, je valide le plus court des chemins non déjà validés

j'ai maintenant : $[(0, \text{None}, \text{True}), (1, A, \text{True}), (6, B, \text{True}), (8, B, \text{False})]$

je procède à la recherche partant de C, j'affirme $(7, C, \text{False})$ à comparaison à $(8, B, \text{False})$, le candidat gagne, on a :

$[(0, \text{None}, \text{True}), (1, A, \text{True}), (6, B, \text{True}), (7, C, \text{False})]$

Validation du plus court non True :

$[(0, \text{None}, \text{True}), (1, A, \text{True}), (6, B, \text{True}), (7, C, \text{True})]$ ②

```

def executer_dijkstra(d_ou_partir):
    global M
    #Initialisation de liste dijkstra qui me permettra d'explorer les chemins
    dijkstra = [None]*len(M)
    #distance jusqu'ici, 'ou étais-je avant d'être là, ce chemin est-il valide, est-ce le dernier sélectionné
    dijkstra[d_ou_partir] = [0,None,False]
    #distance jusqu'ici, 'ou étais-je avant d'être là, ce chemin est-il valide, est-ce le dernier sélectionné

    for i in range(len(M)-1):
        #for i in range(3):
            #quel est le chemin le plus court non déjà valide ?
            if i:
                indexe = 0
                recherche = []
                for i in range(len(dijkstra)):
                    if dijkstra[i] != None:
                        if not dijkstra[i][2]:
                            recherche += [[dijkstra[i][0],i]]
                recherche.sort()
                indexe = recherche[0][1]
            else:
                indexe = d_ou_partir
                dijkstra[indexe][2] = True
                distance = dijkstra[indexe][0]

            #A partir d'ici ou puis-je aller ?
            possibilites = M[indexe]
            new_dijkstra = []
            for i in range(len(possibilites)):
                if possibilites[i] != None: #Toujours le cas dans notre exercice
                    new_dijkstra += [[distance+possibilites[i],indexe,False]]
                else:
                    new_dijkstra += [None]

            #Remplacement chemins si de meilleurs sont trouvés voir création si nouvelle opportunité
            for i in range(len(dijkstra)):
                if dijkstra[i] != None:
                    if not dijkstra[i][2]: #Si on est pas déjà sur que ce soit le chemin min vers ce point Y
                        if new_dijkstra[i] != None: #que le point X de départ conduise vers Y
                            if i != indexe: #que Y ne soit pas X
                                if new_dijkstra[i][0] < dijkstra[i][0]:# et que le chemin de X vers Y est plus court que ce trouvés précédemment
                                    dijkstra[i] = new_dijkstra[i]
                else:
                    if new_dijkstra[i] != None:
                        if i != indexe:
                            dijkstra[i] = new_dijkstra[i]
    return dijkstra

```

Si nous lançons notre programme "Djikstra.py" nous obtiendrons un menu où l'on pourra choisir le point de départ et le point d'arriver, puis un affichage du meilleur chemin à emprunter. Voici un exemple de résultat :

```

D'où voulez-vous partir ?1

Où voulez-vous aller ?4
Pour aller de g1 vers g4 (d=7.7), il vaut mieux :
- aller de g1 vers g9(d=6.1)
- aller de g9 vers g4(d=1.6)

```

```
Voulez-vous continuer ?0
```

2.3 Show and display a solution of the problem.

Si nous lançons notre programme "Djikstra affichage de la recherche.py" nous obtiendrons tous les chemins de g1 vers les autres destinations avec leur distance totale et les villes de passage. Dans la plupart des cas, un chemin direct entre Gotham(g1) et une autre ville est le plus court. En effet nous obtenons le chemin suivant :

```

-----
Pour aller de g1 vers g1, il vaut mieux :
- ne pas bouger parce qu'on part de là.

Pour aller de g1 vers g2, il vaut mieux :
- aller de g2 vers g1
- aller de g1 vers g10

Pour aller de g1 vers g3, il vaut mieux :
- aller de g3 vers g1

Pour aller de g1 vers g4, il vaut mieux :
- aller de g4 vers g1
- aller de g1 vers g9

Pour aller de g1 vers g5, il vaut mieux :
- aller de g5 vers g1

Pour aller de g1 vers g6, il vaut mieux :
- aller de g6 vers g1

Pour aller de g1 vers g7, il vaut mieux :
- aller de g7 vers g1

Pour aller de g1 vers g8, il vaut mieux :
- aller de g8 vers g1

Pour aller de g1 vers g9, il vaut mieux :
- aller de g9 vers g1

Pour aller de g1 vers g10, il vaut mieux :
- aller de g10 vers g1

```

3 Organize the Jokers

Le logarithme utilisé dans les calculs de complexité des parties 3 et 4 est le logarithme binaire.

```

""" Une recherche naïve et gloutonne """
def naifSearch (unsortedListe, researched_number):

    # On regarde toute les cases
    for i in range (0,len(unsortedListe)):
        if unsortedListe[i]==researched_number:
            return i
    # si aucunes des cases sont égales à notre nombre alors on retourne -1
    return -1

```

Question 1 :

Il s'agit d'une simple boucle for : On regarde les valeurs de la liste une par une. Sa complexité est donc en $O(n)$

```

""" La fonction recursive de divideAndConquerSearch """
def divideAndConquerSearchRec (unsortedListe, l, r, researched_number):

    # Si nous n'avons pas parcouru l'ensemble des elements
    if r >= l:
        mid = l + (r - 1)//2
        # Si le milieu est notre element recherche
        if unsortedListe[mid] == researched_number:
            return mid
        else:
            # On regarde des deux cotes car la liste n'est pas triee
            return max(divideAndConquerSearchRec(unsortedListe, mid + 1, r, researched_number),
                       divideAndConquerSearchRec(unsortedListe, l, mid-1, researched_number))
    else:
        # Si notre element n'est pas present on retourne -1
        return -1

""" Une recherche naive et gloutonne mais en diviser pour regner sur une liste non triee """
def divideAndConquerSearch (sorterListe, researched_number):
    return divideAndConquerSearchRec (sorterListe, 0, len(sorterListe)-1, researched_number)

```

Question 2 :

Il s'agit d'un diviser pour régner sur une liste non triée, on doit donc regarder les valeurs des deux côtés :

Pour obtenir sa complexité, nous pouvons écrire la formule suivante :

$$T(n) = \underbrace{1}_{\text{taille de la liste}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{partie droite de la liste}} + \underbrace{T\left(\frac{n}{2}\right)}_{\text{partie gauche de la liste}}$$

Soit $n = 2^t$

$$\begin{aligned} \text{On a } T(2^t) &= 1 + T(2^{t-1}) + T(2^{t-1}) \\ &= 1 + 2 + T(2^{t-2}) + T(2^{t-2}) + T(2^{t-2}) + T(2^{t-2}) \\ &\xrightarrow{\text{par itération}} = \dots \\ &= 1 + 2 + 4 + \dots + 2^{t-1} \\ &= \frac{1 - 2^t}{1 - 2} \\ &= 2^t - 1 \end{aligned}$$

$$T(n) = n - 1$$

Donc, on a une complexité en $\boxed{O(n)}$

```

""" Un quicksort afin de trier notre liste de maniere plutot efficace """
def quicksort(unsortedListe):
    if unsortedListe == []:
        return []
    else:
        pivot = unsortedListe[0]
        l1 = []
        l2 = []
        for x in unsortedListe[1:]:
            if x<pivot:
                l1.append(x)
            else:
                l2.append(x)
        return quicksort(l1)+[pivot]+quicksort(l2)

""" La fonction recursive de divideAndConquerSearch """
def dichotomieSearchRec (sorterdListe, l, r, researched_number):

    # Si nous n'avons pas parcouru l'ensemble des elements susceptible d'etre notre valeur
    if r >= l:

        mid = l + (r - 1)//2

        # Si le milieu est notre element recherche
        if sorterdListe[mid] == researched_number:
            return mid

        # Si l'element est plus grand que le milieu alors il ne peut que se situer a droite
        elif sorterdListe[mid] < researched_number:
            return dichotomieSearchRec(sorterdListe, mid + 1, r, researched_number)

        # Si l'element est plus petit que le milieu alors il ne peut que se situer a gauche
        else:
            return dichotomieSearchRec(sorterdListe, l, mid-1, researched_number)
    else:
        # Si notre element n'est pas present on retourne -1
        return -1

```

Question 3 :

On utilise le tri rapide (Quick sort) afin de trier efficacement la liste.

Déterminons sa complexité :

La complexité de la partition est de αn avec α un entier positif
le pivot sépare le tableau en deux parties de (l) éléments et $(n-l)$ éléments

On obtient donc : $T(n) = T(l) + T(n-l) + \alpha n$

On peut ainsi observer deux cas limites :

- Pire cas: $\boxed{l=1}$

\Rightarrow le pivot est la plus petite ou la plus grande valeur

$$\text{On a } T(n) = T(1) + T(n-1) + \alpha n$$

$$= [T(n-2) + T(1) + \alpha(n-1)] + T(1) + \alpha n$$

$$= \dots$$

par itération

$$= n T(1) + \alpha (n-1+1+\dots+n-2+n-1+n)$$

$$= n T(1) + \alpha \left(\frac{n(n-1)+(n-2)(n-1)}{2} \right)$$

$$= n T(1) + \frac{\alpha}{2} (n^2 - 2n + n^2 - 2n - n - 2)$$

$$= n T(1) + \alpha n^2 - \frac{5}{2} n - 2$$

Donc au pire la complexité est en $\boxed{O(n^2)}$

• Meilleur cas : $\boxed{h = \frac{n}{2}}$

\hookrightarrow Le pivot est la valeur centrale

On a $T(n) = 2T\left(\frac{n}{2}\right) + \alpha n$

On pose $n = 2^t$

$$\begin{aligned} T(2^t) &= 2T(2^{t-1}) + \alpha 2^t \\ &= 4T(2^{t-2}) + 2 \times \alpha 2^{t-1} + \alpha 2^t \\ &= 4T(2^{t-2}) + 2 \times \alpha 2^t \\ &= 8T(2^{t-3}) + 3 \times \alpha 2^t \\ \text{par iteration} \rightarrow &= \dots \\ &= 2^t T(1) + t \alpha 2^t \end{aligned}$$

Donc $T(n) = nT(1) + \log(n) \alpha n$

On a donc au mieux une complexité en $\boxed{O(n \log n)}$

Avec une liste triée, nous pouvons maintenant effectuer une dichotomie pour rechercher une valeur.

Déterminons sa complexité :

On a : $T(n) = 1 + T\left(\frac{n}{2}\right)$ récursion sur une moitié de la liste
comparaison

On pose $n = 2^t$

$$\begin{aligned} T(2^t) &= T\left(\frac{2^t}{2}\right) + 1 \\ &= T(2^{t-1}) + 1 \\ &= T(2^{t-2}) + 2 \end{aligned}$$

$$\text{par iteration} \rightarrow \dots = T(1) + t$$

Donc $T(n) = T(1) + \log(n)$

La complexité du tri par dichotomie est donc de $\boxed{O(\log(n))}$

Question 4: On crée notre base de données de manière aléatoire :

Affichage de la liste non triee:

```
[9181, 7495, 9218, 3349, 3505, 6403, 9282, 9731, 3512, 9314, 5972, 3042, 6973, 3979, 6980, 686, 7990, 4826, 9075, 7062, 6874, 6050, 5864, 9571, 2386, 8250, 5074, 1241, 1305, 5453, 2795, 4806, 8696, 1634, 2273, 3578, 8506, 3949, 6432, 6083, 7876, 8331, 2587, 6975, 5310, 3623, 2081, 9531, 4611, 4926, 1672, 1567, 4191, 8185, 4380, 5135, 574, 7581, 2506, 2040, 5663, 2722, 8900, 3069, 8204, 1468, 365, 7257, 7949, 6609, 7771, 144, 8807, 2956, 2153, 9203, 1599, 8723, 4326, 1191, 4031, 4931, 4000, 2786, 6903, 5427, 8428, 8592, 5541, 4942, 9325, 5376, 4593, 9234, 5089, 4016, 1249, 6263, 3850]
```

Affichage de la liste triee:

```
[144, 365, 574, 686, 1191, 1241, 1249, 1305, 1468, 1567, 1599, 1634, 1672, 2040, 2081, 2153, 2273, 2386, 2506, 2587, 2722, 2786, 2795, 2956, 3042, 3069, 3349, 3505, 3512, 3578, 3623, 3850, 3949, 3979, 4000, 4016, 4031, 4191, 4326, 4380, 4593, 4611, 4806, 4826, 4926, 4931, 4936, 4942, 5074, 5089, 5135, 5310, 5376, 5427, 5453, 5541, 5663, 5864, 5972, 6050, 6083, 6263, 6403, 6432, 6609, 6874, 6903, 6975, 6980, 7062, 7257, 7495, 7581, 7771, 7876, 7949, 7990, 8185, 8204, 8250, 8331, 8428, 8506, 8592, 8696, 8723, 8807, 8900, 9075, 9181, 9203, 9218, 9282, 9314, 9325, 9531, 9571, 9731]
```

Et nous obtenons les résultats de test suivants :

Test de recherche dans les listes (la position "-1" indique que la valeur n'est pas dans la liste) :

Liste non triee :

```
La valeur 3505 est, d'apres la recherche naive, dans la liste non triee en position : 4  
La valeur 12256 est, d'apres la recherche naive, dans la liste non triee en position : -1  
La valeur 3505 est, d'apres la recherche diviser pour regner naive, dans la liste non triee en position : 4  
La valeur 12256 est, d'apres la recherche diviser pour regner naive, dans la liste non triee en position : -1  
La valeur 3505 est, d'apres la recherche dichotomique, dans la liste non triee en position : -1  
La valeur 12256 est, d'apres la recherche dichotomique, dans la liste non triee en position : -1  
On peut observer un resultat faux pour cette derniere methode car celle ci ne fonctionne correctement qu'avec une liste triee
```

Liste triee :

```
La valeur 3505 est, d'apres la recherche naive, dans la liste triee en position : 27  
La valeur 12256 est, d'apres la recherche naive, dans la liste triee en position : -1  
La valeur 3505 est, d'apres la recherche diviser pour regner naive, dans la liste triee en position : 27  
La valeur 12256 est, d'apres la recherche diviser pour regner naive, dans la liste triee en position : -1  
La valeur 3505 est, d'apres la recherche dichotomique, dans la liste triee en position : 27  
La valeur 12256 est, d'apres la recherche dichotomique, dans la liste triee en position : -1
```

D'un point de vue de complexité, il est plus intéressant, si on veut effectuer une unique recherche, d'utiliser l'algorithme naïf ne nécessitant pas de liste triée (en effet la fonction de tri est en $O(n\log(n))$) tandis que la recherche naïve est seulement en $O(n)$). Cependant pour de multiple recherche, il devient plus intéressant de trier la liste car la recherche dichotomique est en $O(\log(n))$ contrairement à la recherche naïve qui est en $O(n)$.

4 The Jokefather

On crée la data base de manière aléatoire :

Affichage de la base de donnees brute:

```
[9794, 5303, 6395, 3342, 7068, 3007, 7183, 6691, 4287, 2716, 7039, 4861, 3172, 6562, 605, 712, 7091, 2420, 6622, 4762, 8951, 6346, 6588, 1817, 7182, 8378, 7965, 7260, 6826, 9949, 4180, 8396, 4482, 8026, 3573, 7200, 5670, 6936, 1779, 3760, 2911, 7722, 8295, 3284, 7776, 3185, 2705, 4714, 1889, 9094, 446, 7013, 7653, 6495, 3228, 5846, 9017, 6420, 492, 1768, 9403, 6998, 7110, 657, 4442, 5032, 5409, 9563, 1664, 9301, 1238, 1173, 5404, 1806, 877, 3017, 4895, 4064, 3252, 1842, 1322, 8953, 80, 5853, 5669, 7026, 6571, 4478, 1312, 1479, 6512, 7732, 8127, 887, 2243, 7263, 6153, 6573, 5198, 5476]
```

Question 1 : Les deux affichages sont ici tronqués sur le rapport car trop longs mais vous pouvez faire tourner le programme pour avoir un meilleur aperçu :

Affichage PreOrder de l'arbre BST:

```
0| 9794
1| 5303
2| 3342
3| 3007
4| 2716
5| 605
6| 446
7| 80
7| 492
6| 712
7| 657
7| 2420
8| 1817
9| 1779
10| 1768
11| 1664
12| 1238
13| 1173
14| 877
15| 887
13| 1322
14| 1312
14| 1479
10| 1806
9| 1889
10| 1842
10| 2243
8| 2705
5| 2911
4| 3172
5| 3017
5| 3284
6| 3185
7| 3228
8| 3252
3| 4287
4| 4180
5| 3573
6| 3760
7| 4064
4| 4861
5| 4762
6| 4482
7| 4442
8| 4478
7| 4714
5| 5032
```

Affichage InOrder de l'arbre BST:

```
7| 80
6| 446
7| 492
5| 605
7| 657
6| 712
14| 877
15| 887
13| 1173
12| 1238
14| 1312
13| 1322
14| 1479
11| 1664
10| 1768
9| 1779
10| 1806
8| 1817
10| 1842
9| 1889
10| 2243
7| 2420
8| 2705
4| 2716
5| 2911
3| 3007
5| 3017
4| 3172
6| 3185
7| 3228
8| 3252
5| 3284
2| 3342
5| 3573
6| 3760
7| 4064
4| 4180
3| 4287
7| 4442
8| 4478
6| 4482
7| 4714
5| 4762
4| 4861
6| 4895
5| 5032
6| 5198
1| 5303
6| 5404
5| 5489
7| 5476
6| 5669
```

Test de recherche dans l'arbre BST:

La valeur 7068 est dans la base de donnee : True

La valeur 12097 est dans la base de donnee : False

Partie 4 :

Question 1:

- Remplir l'arbre binaire avec les données :

la fonction insert qui permet de rentrer une donnée à une complexité en $O(1)$

On cette fonction est utilisée sur chaque élément de la liste afin de créer et de remplir l'arbre binaire de recherche

Ainsi la complexité de cette création et remplissage est $\boxed{\text{en } O(n)}$

- Recherche d'une valeur dans l'arbre binaire de recherche :

On peut observer deux cas limites :

- Pire cas: l'arbre binaire de recherche ne possède que des nœuds n'ayant qu'un seul fils
(cas arrivant notamment en utilisant à l'origine une liste triée)

Ainsi on doit parcourir tous les nœuds afin de trouver la valeur car le problème ne peut être divisé.
La complexité au pire est donc $\boxed{\text{en } O(n)}$

- Mieux cas: l'arbre est bien équilibré.

On a donc autant de nœuds de chaque côté

Ainsi on a : $T(n) = T\left(\frac{n}{2}\right) + \alpha$ avec α en entier positif
partie gauche ou droite de l'arbre en fonction de la valeur

On pose $n = 2^t$

$$\begin{aligned} \text{On a donc } T(2^t) &= T(2^{t-1}) + 1\alpha \\ &= T(2^{t-2}) + 2\alpha \end{aligned}$$

par itération $= \dots$

$$\text{Ainsi } T(n) = T(1) + t\alpha$$

La complexité au mieux est donc en $\boxed{O(\log n)}$

Question 2: Les deux affichages sont ici tronqués sur le rapport car trop longs mais vous pouvez faire tourner le programme pour avoir un meilleur aperçu :

Affichage PreOrder de l'arbre AVL:

```
0| 6395
1| 3342
2| 1817
3| 712
4| 492
5| 446
6| 80
5| 605
6| 657
4| 1322
5| 1238
6| 887
7| 877
7| 1173
6| 1312
5| 1768
6| 1664
7| 1479
6| 1779
7| 1806
3| 3007
4| 2420
5| 1889
6| 1842
6| 2243
5| 2716
6| 2705
6| 2911
4| 3185
5| 3172
6| 3017
5| 3252
6| 3228
6| 3284
2| 4861
3| 4287
4| 3760
5| 3573
5| 4180
6| 4064
4| 4714
5| 4478
6| 4442
```

```
Affichage InOrder de l'arbre AVL:  
6| 80  
5| 446  
4| 492  
5| 605  
6| 657  
3| 712  
7| 877  
6| 887  
7| 1173  
5| 1238  
6| 1312  
4| 1322  
7| 1479  
6| 1664  
5| 1768  
6| 1779  
7| 1806  
2| 1817  
6| 1842  
5| 1889  
6| 2243  
4| 2420  
6| 2705  
5| 2716  
6| 2911  
3| 3007  
6| 3017  
5| 3172  
4| 3185  
6| 3228  
5| 3252  
6| 3284  
1| 3342  
5| 3573  
4| 3760  
6| 4064  
5| 4180  
3| 4287  
6| 4442  
5| 4478  
6| 4482  
4| 4714  
5| 4762  
2| 4861  
6| 4895  
5| 5032
```

```
Test de recherche dans l'arbre AVL:  
La valeur 7068 est dans la base de donnee : True  
La valeur 12097 est dans la base de donnee : False
```

Question 2:

Les AVL sont des arbres binaires de recherche toujours équilibrés. Ainsi cela change aussi la complexité de ses méthodes :

- Remplir l'AVL avec les données :

La fonction d'insertion est maintenant en $O(\log n)$ à cause du rééquilibrage de l'arbre à fin de chacune d'elle.

Ainsi, puisque cette fonction est utilisée sur l'intégralité de la liste, la création et le remplissage de l'AVL est en $\boxed{O(n \log n)}$

- Recherche d'une valeur dans un AVL :

L'arbre, étant toujours équilibré, nous nous situons à chaque fois dans le meilleur cas pour la recherche dans un arbre binaire de recherche.

Ainsi la complexité de la recherche est toujours en $\boxed{O(\log n)}$

Question 3:

```
Affichage LevelOrder de l'arbre B:
0] [1779, 3342, 6395, 7183]
1] [605, 1238] [1889, 2716, 3172] [4287, 4861, 5303, 5670] [6562, 6622, 6936, 7068] [7722, 7965, 8378, 9094]
2] [80, 446, 492] [657, 712, 877, 887, 1173] [1312, 1322, 1479, 1664, 1768] [1806, 1817, 1842] [2243, 2420, 2705]
[2911, 3007, 3017] [3185, 3228, 3252, 3284] [3573, 3760, 4064, 4180] [4442, 4478, 4482, 4714, 4762] [4895, 5032,
5198] [5404, 5409, 5476, 5669] [5846, 5853, 6153, 6346] [6420, 6495, 6512] [6571, 6573, 6588] [6691, 6826] [6998,
7013, 7026, 7039] [7091, 7110, 7182] [7200, 7260, 7263, 7653] [7732, 7776] [8026, 8127, 8295] [8396, 8951, 8953,
9017] [9301, 9403, 9563, 9794, 9949]
```

```
Affichage PreOrder de l'arbre B:  
0| [1779, 3342, 6395, 7183]  
1| [605, 1238]  
2| [80, 446, 492]  
2| [657, 712, 877, 887, 1173]  
2| [1312, 1322, 1479, 1664, 1768]  
1| [1889, 2716, 3172]  
2| [1806, 1817, 1842]  
2| [2243, 2420, 2705]  
2| [2911, 3007, 3017]  
2| [3185, 3228, 3252, 3284]  
1| [4287, 4861, 5303, 5670]  
2| [3573, 3760, 4064, 4180]  
2| [4442, 4478, 4482, 4714, 4762]  
2| [4895, 5032, 5198]  
2| [5404, 5409, 5476, 5669]  
2| [5846, 5853, 6153, 6346]  
1| [6562, 6622, 6936, 7068]  
2| [6420, 6495, 6512]  
2| [6571, 6573, 6588]  
2| [6691, 6826]  
2| [6998, 7013, 7026, 7039]  
2| [7091, 7110, 7182]  
1| [7722, 7965, 8378, 9094]  
2| [7200, 7260, 7263, 7653]  
2| [7732, 7776]  
2| [8026, 8127, 8295]  
2| [8396, 8951, 8953, 9017]  
2| [9301, 9403, 9563, 9794, 9949]
```

Test de recherche dans l'arbre B:

La valeur 7068 est dans la base de donnee : True

La valeur 12097 est dans la base de donnee : False

Question 3 :

Les B-tree sont assez proches des AVL dans leurs avantages et dans leur conception. Ils possèdent donc des complexités similaires pour leurs différentes méthodes :

- Remplir le B-Tree avec les données:

La fonction d'insertion est en $O(\log n)$ à cause de la fonction split qui la compose.

Ainsi le remplissage du B-Tree est en $O(n \log n)$

- Recherche d'une valeur dans le B-Tree

Le B-Tree, étant équilibré, possède tout comme l'AVL une complexité pour la recherche en $O(\log n)$

