

INFO0902 Projet 2 - Arbres binaires de recherche

(Rapport)

GERARD Manon, DUCHEMIN Catherine

TOTAL POINTS

20.5 / 22

QUESTION 1

1 Pseudocode bstRangeSearch **4.5 / 5**

- **0 pts** Correct
- **5 pts** Pas réponse ou réponse totalement incorrecte.
- ✓ - **0.5 pts** *Erreur mineure.*
 - **2 pts** Code inefficace (parcours complet de l'arbre)
 - **1 pts** Petite erreur ou problème de style.
 - **2 pts** Petites erreurs ou problèmes de style.
 - **3 pts** Plusieurs erreurs ou problème, ou erreur majeure.
 - **4 pts** Beaucoup d'erreurs ou problèmes de style.
- 1 Erreur: ne traite pas correctement le cas où plusieurs clés identiques sont dans l'arbre. Il ne faut pas créer un cas particulier si ce cas rentre dans le cas général plus bas.

QUESTION 2

2 Complexités **3 / 4**

- **0 pts** Correct
- **4 pts** Pas de réponse ou réponse totalement incorrecte
- **0.5 pts** Meilleur cas arbre équilibré pas correct ou manquant

✓ - **0.5 pts** *Meilleur cas arbre quelconque pas correct ou manquant*

- **0.5 pts** Pire cas arbre équilibré pas correct ou manquant
- **0.5 pts** Pire cas arbre quelconque pas correct ou manquant
- **0.5 pts** Petite erreur ou petit manque dans les justifications
- **1 pts** Quelques erreurs ou manques dans les justifications
- **2 pts** Beaucoup d'erreurs ou manques dans les justifications
- ✓ - **0.5 pts** *\$\$k\$\$ pas pris comme un paramètre de valeur fixée.*

2 La distinction n'est pas très claire. On demandait la complexité en fonction de $\Theta(k)$. Ca veut dire que les formules de complexité devait faire apparaître $\Theta(k)$.

3 Non. Si k_{max} est < que la valeur à la racine et qu'il n'y a pas de sous-arbre à gauche, on ne descend même pas dans la branche => $\Theta(1)$.

4 Idem. $\Theta(1)$.

QUESTION 3

3 Implémentation 3 / 3

✓ - 0 pts Correct

- 3 pts Pas de réponse ou réponse totalement incorrecte

- 0.5 pts Quelques détails manquants ou pas suffisamment clairs.

- 1 pts Plusieurs détails importants manquants ou pas suffisamment clairs.

- 2 pts Beaucoup de détails importants manquants ou pas suffisamment clairs.

- 0 pts Solution ABR inefficace (2 arbres par exemple).

- 0 pts pdctBallSearch erronée (oublie le filtrage ou bien autre erreur, par exemple, positions pas stockées dans l'arbre).

- 0 pts Solution bst2dBallSearch erronée ou inefficace.

QUESTION 4

4 Profondeurs moyennes 3 / 3

✓ - 0 pts Correct

- 3 pts Pas de réponse ou réponse totalement incorrecte

- 0.5 pts Discussion manquante ou incomplète (par ex., pas de mention que l'évolution est bien logarithmique).

- 1 pts Résultats partiellement erronés.

- 1 pts Discussion manquante ou incomplète (par ex., pas de mention que l'évolution est bien logarithmique).

- 2 pts Résultats complètement erronés

QUESTION 5

5 Temps de calcul 5 / 5

✓ - 0 pts Correct

- 5 pts Pas de réponse ou réponse totalement incorrecte

- 1 pts Temps de création dictionnaire manquant (et pas discuté).

- 1.5 pts Discussions manquantes.

- 0.5 pts Petites erreurs ou manques dans la discussion

- 1 pts Discussions incomplètes.

- 0.5 pts Quelques valeurs aberrantes ou manquantes dans les tables.

- 1 pts Temps arbre 2d incorrectes ou manquants.

- 2 pts Impact du rayon pas étudié.

5 La rayon de recherche n'a absolument aucun impact sur la création du dictionnaire et sur les recherches exactes. Il ne fallait pas en parler. Ce que vous observez ne peut être que du à l'aléatoire.

QUESTION 6

6 Conclusion 1 / 1

✓ - 0 pts Correct

- 0.5 pts Ne conclut pas sur les trois approches.

- 1 pts Pas de réponse ou réponse incorrecte

QUESTION 7

7 Endroit de Porto 1 / 1

✓ - 0 pts Correct

- 0.5 pts Nombre de taxis pas donné.

- 1 pts Pas de réponse ou réponse incorrecte

NB : Nous avons utilisé la convention de comparaison au sein de l'arbre de l'énoncé initial.

1 Pseudo-code de la fonction BSTRANGESEARCH

```
BSTRANGESEARCH(binarySearchTree, keyMin, keyMax)
1 list = LISTNEW()
2 root = binarySearchTree.root
3 comp = binarySearchTree.COMPFN(keyMin, keyMax)
4 // If keyMin > keyMax or binarySearchTree is empty, return an empty list
5 if comp > 0 or root == NIL
6     return list
7 // If keyMin = keyMax, search for the exact node inside the binarySearchTree
8 if comp == 0 ①
9     node = BSTSEARCH(binarySearchTree, keyMin)
10    if node != NIL
11        LISTINSERTFIRST(list, node.value)
12    return list
13 INORDERTREEWALK(binarySearchTree, root, list, keyMin, keyMax)
14 return list

INORDERTREEWALK(binarySearchTree, node, list, keyMin, keyMax)
1 if n == NIL
2     return
3 key = node.key
4 compMin = binarySearchTree.COMPFN(keyMin, key)
5 compMax = binarySearchTree.COMPFN(keyMax, key)
6 // Tree path left (recursive call)
7 if compMin ≤ 0
8     INORDERTREEWALK(binarySearchTree, node.left, list, keyMin, keyMax)
9 // Insert the value corresponding to the key only if keyMin ≤ key ≤ keyMax
10 if compMin ≤ 0 and compMax ≥ 0
11     LISTINSERTLAST(list, node.value)
12 // Tree path right (recursive call)
13 if compMax > 0
14     INORDERTREEWALK(binarySearchTree, node.right, list, keyMin, keyMax)
```

Les fonctions LISTNEW, BSTSEARCH et LISTINSERTFIRST étaient déjà implémentées, ainsi que la fonction COMPFN, qui correspond à la fonction ptCompare. Nous n'avons donc pas représenté leurs pseudo-codes.

2 Analyse de la complexité de BSTRANGESEARCH

Notre algorithme commence par :

- la création une liste vide avec la fonction **ListNew**
- la récupération de la racine de l'arbre avec la fonction **bstRoot**
- la comparaison des clés k_{min} et k_{max} passées en argument à la fonction **bstRangeSearch**

Ces 3 opérations ont chacune une complexité $\Theta(1)$ en temps et en espace dans tous les cas (pire cas comme meilleur cas), que l'arbre soit équilibré ou non, et quels que soient N et k .

Ensuite, notre algorithme est composé de 3 cas distincts et exclusifs les uns des autres (une fois entré dans un cas, on ne rentrera jamais dans un autre cas) :

1 Pseudocode bstRangeSearch 4.5 / 5

- **0 pts** Correct
- **5 pts** Pas réponse ou réponse totalement incorrecte.
- ✓ **- 0.5 pts** *Erreur mineure.*
- **2 pts** Code inefficace (parcours complet de l'arbre)
- **1 pts** Petite erreur ou problème de style.
- **2 pts** Petites erreurs ou problèmes de style.
- **3 pts** Plusieurs erreurs ou problème, ou erreur majeure.
- **4 pts** Beaucoup d'erreurs ou problèmes de style.

1 Erreur: ne traite pas correctement le cas où plusieurs clés identiques sont dans l'arbre. Il ne faut pas créer un cas particulier si ce cas rentre dans le cas général plus bas.

NB : Nous avons utilisé la convention de comparaison au sein de l'arbre de l'énoncé initial.

1 Pseudo-code de la fonction BSTRANGESEARCH

```
BSTRANGESEARCH(binarySearchTree, keyMin, keyMax)
1 list = LISTNEW()
2 root = binarySearchTree.root
3 comp = binarySearchTree.COMPFN(keyMin, keyMax)
4 // If keyMin > keyMax or binarySearchTree is empty, return an empty list
5 if comp > 0 or root == NIL
6     return list
7 // If keyMin = keyMax, search for the exact node inside the binarySearchTree
8 if comp == 0 ①
9     node = BSTSEARCH(binarySearchTree, keyMin)
10    if node != NIL
11        LISTINSERTFIRST(list, node.value)
12    return list
13 INORDERTREEWALK(binarySearchTree, root, list, keyMin, keyMax)
14 return list

INORDERTREEWALK(binarySearchTree, node, list, keyMin, keyMax)
1 if n == NIL
2     return
3 key = node.key
4 compMin = binarySearchTree.COMPFN(keyMin, key)
5 compMax = binarySearchTree.COMPFN(keyMax, key)
6 // Tree path left (recursive call)
7 if compMin ≤ 0
8     INORDERTREEWALK(binarySearchTree, node.left, list, keyMin, keyMax)
9 // Insert the value corresponding to the key only if keyMin ≤ key ≤ keyMax
10 if compMin ≤ 0 and compMax ≥ 0
11     LISTINSERTLAST(list, node.value)
12 // Tree path right (recursive call)
13 if compMax > 0
14     INORDERTREEWALK(binarySearchTree, node.right, list, keyMin, keyMax)
```

Les fonctions LISTNEW, BSTSEARCH et LISTINSERTFIRST étaient déjà implémentées, ainsi que la fonction COMPFN, qui correspond à la fonction ptCompare. Nous n'avons donc pas représenté leurs pseudo-codes.

2 Analyse de la complexité de BSTRANGESEARCH

Notre algorithme commence par :

- la création une liste vide avec la fonction **ListNew**
- la récupération de la racine de l'arbre avec la fonction **bstRoot**
- la comparaison des clés k_{min} et k_{max} passées en argument à la fonction **bstRangeSearch**

Ces 3 opérations ont chacune une complexité $\Theta(1)$ en temps et en espace dans tous les cas (pire cas comme meilleur cas), que l'arbre soit équilibré ou non, et quels que soient N et k .

Ensuite, notre algorithme est composé de 3 cas distincts et exclusifs les uns des autres (une fois entré dans un cas, on ne rentrera jamais dans un autre cas) :

- Lignes 4 à 6 : $k = 0$ (ou arbre vide mais on ne tient pas compte de ce cas pour le calcul de la complexité puisqu'on suppose N très grand).
- Lignes 7 à 12 : $k \in [0, 1]$
- Lignes 13 et 14 : $k \in [0, N]$

Cas 1 : $k = 0$

La comparaison effectuée entre keyMin et keyMax montre que le nombre de valeur dans l'intervalle est nul donc on retourne une liste vide.

Ces 2 opérations ont une complexité $\Theta(1)$ en temps et en espace dans tous les cas, que l'arbre soit équilibré ou non, et quel que soit N .

Cas 2 : $k \in [0, 1]$

La comparaison effectuée entre keyMin et keyMax montre que l'intervalle se limite à une seule valeur donc on effectue une recherche exacte en appelant la fonction **bstSearch**.

Cette fonction a une complexité dans le pire cas $\Theta(h)$ en temps et $\Theta(1)$ en espace, où h est la hauteur de l'arbre. Ce pire cas correspond à celui où on doit parcourir tout l'arbre, soit parce que la valeur recherchée ne s'y trouve pas, soit parce qu'elle se trouve à la dernière position visitée. On distingue :

- pour un arbre équilibré qui a pour propriété $h \in \log N$:
 - La complexité en temps au pire cas est $\Theta(\log N)$
 - La complexité en espace au pire cas est $\Theta(1)$
- pour un arbre non équilibré :
 - La complexité en temps au pire cas est $\Theta(N)$
 - La complexité en espace au pire cas est $\Theta(1)$

Le meilleur cas correspond, pour les deux types d'arbre, au cas où la valeur de la clé recherchée correspond à la valeur de la clé de la racine. La complexité est alors $\Theta(1)$ en temps comme en espace, quelle que soit la valeur de N .

On vérifie ensuite si le noeud dont la clé correspond à la valeur de clé recherchée existe ou non, et s'il existe on insère la valeur associée à ce noeud dans la liste, puis on retourne la liste. Etant donné qu'on n'effectue un seul appel à la fonction **listInsertFirst**, la complexité de ces opérations est $\Theta(1)$ en temps et en espace dans tous les cas, que l'arbre soit équilibré ou non, et quel que soit N .

La complexité de notre algorithme dans ce deuxième cas est donc égale à la complexité de la fonction **bstSearch**.

Cas 3 : $k \in [0, N]$

La comparaison effectuée entre keyMin et keyMax montre que l'intervalle ne se limite pas à une seule valeur. On effectue un appel à la fonction **inOrderTreeWalk**, qui parcourt l'arbre dans l'ordre des clés, puis on retourne la liste des k valeurs qui correspondent aux clés faisant partie de l'intervalle de recherche.

C'est donc la complexité de **inOrderTreeWalk** qui détermine la complexité de notre algorithme dans ce troisième cas.

inOrderTreeWalk commence par des opérations de vérification, d'assignation et de comparaison qui sont effectuées à un temps constant qui ne dépend ni de N ni de k .

Ensuite, on a :

- un premier appel récursif à **inOrderTreeWalk** vers le fils gauche du noeud visité tant que la valeur de la clé est supérieure à keymin et donc appartient possiblement à l'intervalle de recherche.

- une fois qu'on a atteint un noeud pour lequel la valeur de clé est supérieure ou égale à keyMin, on vérifie si cette valeur est également inférieure ou égale à keyMax ; si oui la clé du noeud visité appartient bien à l'intervalle de recherche, donc on insère la valeur associée au noeud dans la liste à l'aide de la fonction **listInsertLast**.
- un second appel récursif à **inOrderTreeWalk** vers le fils droit du noeud visité tant que la valeur de la clé de ce noeud est inférieure à keyMax et donc appartient possiblement à l'intervalle de recherche.

Complexité en temps

La complexité en temps de **inOrderTreeWalk** dépend du nombre de noeuds visités n , ce qui donne une complexité $\Theta(n)$, tandis que la complexité en temps de **listInsertLast** est $\Theta(1)$ à chaque insertion dans la liste (puisque elle ne dépend pas du nombre d'objets déjà insérés dans la liste), pour un total d'insertions k , ce qui donne une complexité $\Theta(k)$.

- Au pire cas, toutes les valeurs de clé de l'arbre sont comprises dans l'intervalle de recherche. On doit donc visiter les N noeuds de l'arbre et insérer $k = N$ valeurs dans la liste, soit $N + N = 2N$ et on a donc une complexité $\Theta(N)$ pour les deux types d'arbres (équilibrés et non équilibrés).
- Au meilleur cas, aucune des valeurs de clé de l'arbre n'est comprise dans l'intervalle de recherche. On n'effectue donc qu'un seul parcours en profondeur, à gauche ou à droite, et on n'insère rien dans la liste. On a donc une complexité $\Theta(h)$ et on doit distinguer :
 - Le cas de l'arbre équilibré, pour lequel la complexité est alors $\Theta(\log N)$
 - Le cas de l'arbre non équilibré, pour lequel la complexité est alors $\Theta(N)$ 3

Complexité en espace

La complexité en espace de **inOrderTreeWalk** dépend de la hauteur de l'arbre h , ce qui donne une complexité $\Theta(h)$, tandis que la complexité en espace de **listInsertLast** est $\Theta(k)$.

- Au pire cas, toutes les valeurs de clé de l'arbre sont comprises dans l'intervalle de recherche donc on doit insérer $k = N$ valeurs dans la liste. On a donc une complexité $\Theta(N)$ identique pour les deux types d'arbres.
- Au meilleur cas, aucune des valeurs de clé de l'arbre n'est comprise dans l'intervalle de recherche. On n'effectue donc qu'un seul parcours en profondeur, à gauche ou à droite, et on n'insère rien dans la liste. On a donc une complexité $\Theta(h)$ et on doit distinguer :
 - Le cas de l'arbre équilibré, pour lequel la complexité est alors $\Theta(\log N)$
 - Le cas de l'arbre non équilibré, pour lequel la complexité est alors $\Theta(N)$ 4

Tableau récapitulatif des complexités pour un arbre de N noeuds

			$k = 0$	$k \in [0, 1]$	$k \in [0, N]$
Temps	Meilleur cas	Équilibré	$\Theta(1)$	$\Theta(1)$	$\Theta(\log N)$
		Non Équilibré		$\Theta(\log N)$	$\Theta(N)$
	Pire cas	Équilibré		$\Theta(N)$	
		Non Équilibré			
Espace	Meilleur cas	Équilibré	$\Theta(1)$	$\Theta(1)$	$\Theta(\log N)$
		Non Équilibré			
	Pire cas	Équilibré			$\Theta(N)$
		Non Équilibré			

2 Complexités 3 / 4

- **0 pts** Correct
 - **4 pts** Pas de réponse ou réponse totalement incorrecte
 - **0.5 pts** Meilleur cas arbre équilibré pas correct ou manquant
 - ✓ - **0.5 pts** *Meilleur cas arbre quelconque pas correct ou manquant*
 - **0.5 pts** Pire cas arbre équilibré pas correct ou manquant
 - **0.5 pts** Pire cas arbre quelconque pas correct ou manquant
 - **0.5 pts** Petite erreur ou petit manque dans les justifications
 - **1 pts** Quelques erreurs ou manques dans les justifications
 - **2 pts** Beaucoup d'erreurs ou manques dans les justifications
 - ✓ - **0.5 pts** *\$\$k\$\$ pas pris comme un paramètre de valeur fixée.*
- 2 La distinction n'est pas très claire. On demandait la complexité en fonction de $\Theta(k)$. Ca veut dire que les formules de complexité devait faire apparaître $\Theta(k)$.
- 3 Non. Si k_{\max} est < que la valeur à la racine et qu'il n'y a pas de sous-arbre à gauche, on ne descend même pas dans la branche => $\Theta(1)$.
- 4 Idem. $\Theta(1)$.

3 Explication de nos implémentations

Arbre binaire de recherche simple

pdctCreate On a créé un dictionnaire à l'aide d'un arbre simple. Il a fallu créer l'arbre et lui associer une fonction de comparaison, on a donc repris la fonction `ptCompare`. Ensuite, il a fallu remplir l'arbre en insérant tous les éléments de la liste `lpoints` comme clés. En ce qui concerne leurs valeurs associées, il a fallu créer une structure "Value" qui reprendre à la fois la valeur contenue dans `lvalues`, mais aussi la clé de `lpoints`. Cela a dû être fait pour permettre la recherche dans une boucle où l'on a besoin de retrouver la clé alors qu'en utilisant `bstRangeSearch`, nous avons accès à la valeur.

pdctBallSearch On a suivi les indications données dans l'énoncé. Tout d'abord, on a fait un premier filtrage des clés à l'aide de la fonction `bstRangeSearch`. Les clés minimales et maximales sont les coins bas gauche et haut droit du carré circonscrit au cercle. On obtient donc une liste avec tous les "Value" des points dont la coordonnée x est dans l'intervalle x_{min}, x_{max} . Ensuite, on doit filtrer les points appartenant à la boule ou non. On parcourt toute la liste et quand un point de la liste rentrée par `bstRangeSearch` n'appartient pas à la boule, on le supprime de la liste. En revanche, quand le point appartient à la boule, on change sa valeur qui était de type "Value" afin de ne conserver que la valeur demandée. Enfin, on retourne cette liste filtrée.

Arbre binaire de recherche 2d

bst2dBallSearch Quand l'arbre est vide, il suffit de retourner une liste vide. La partie intéressante est quand il existe un arbre. Une fonction `bst2dBallSearchRec` a été créée afin de pouvoir l'appeler quand on décide de continuer à explorer une branche. Cette fonction est initialement appelée avec la racine. Dedans, on vérifie si le nœud appartient à la boule, dans quel cas on l'ajoute à la liste. Ensuite, on fait un appel à cette même fonction avec le successeur droit et gauche s'ils existent et sont susceptibles d'être encore dans la boule.

Ainsi, `continueLeft` et `continueRight` ont été créés pour déterminer cela. Aux profondeurs paires, on comparera la coordonnée x avec la coordonnée la plus à gauche (resp. à droite) de la boule. Aux profondeurs impaires, on comparera la coordonnée y avec la coordonnée la plus basse (resp. haute) de la boule. Si on trouve que c'est inférieur ou égale (resp. supérieur), on continue à explorer à gauche (resp. à droite).

pdctCreate On a créé un dictionnaire à l'aide d'un arbre 2d. Ainsi, il a fallu créer cet arbre puis le remplir en insérant tous les éléments de la liste `lpoints` comme clés et leurs valeurs associés dans `lvalues`.

3 Implémentation 3 / 3

✓ - 0 pts Correct

- 3 pts Pas de réponse ou réponse totalement incorrecte
- 0.5 pts Quelques détails manquants ou pas suffisamment clairs.
- 1 pts Plusieurs détails importants manquants ou pas suffisamment clairs.
- 2 pts Beaucoup de détails importants manquants ou pas suffisamment clairs.
- 0 pts Solution ABR inefficace (2 arbres par exemple).
- 0 pts pdctBallSearch erronée (oublie le filtrage ou bien autre erreur, par exemple, positions pas stockées dans l'arbre).
- 0 pts Solution bst2dBallSearch erronée ou inefficace.

4 Profondeur moyenne des noeuds

Ces valeurs ont été moyennées sur 10 essais dans chacun des cas présentés dans le tableau :

Profondeur moyenne			
Taille (N)	10^4	10^5	10^6
Arbre simple	15,42	20,29	24,97
Arbre 2d	15,56	20,42	24,59
Base du log	$\approx 1,80$	$\approx 1,76$	$\approx 1,74$

En théorie, la profondeur moyenne d'un noeud dans un arbre binaire de recherche construit aléatoirement - ce qui est le cas dans notre projet - est $\Theta(\log N)$.

A partir des données du tableau, on peut constater que de la profondeur moyenne d'un noeud suit bien une évolution logarithmique fonction du nombre de noeuds N de l'arbre. La base moyenne du logarithme, sur ces 10 essais, se situe entre 1,74 et 1,80 et est donc très proche de la valeur théorique de 2.

On constate également que l'évolution de la profondeur moyenne est bien similaire dans les deux cas (arbres simples et arbres 2d).

Les résultats de notre algorithme sont donc conformes à la théorie.

5 Temps de calcul des trois approches

Temps de calcul [s] pour un nombre de recherches fixé à 10^4				
Rayon : 0.01, Nombre de points : 10^4				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	0,000001	0,27315	0,384116	0,37011
Arbre 1d	0,001795	0,00177	0,000457	0,068836
Arbre 2d	0,001832	0,002635	0,000678	0,008389
Rayon : 0.01, Nombre de points : 10^5				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	0,000001	3,397066	4,847243	5,142195
Arbre 1d	0,047295	0,005655	0,000483	1,490889
Arbre 2d	0,039891	0,006656	0,000675	0,073700
Rayon : 0.01, Nombre de points : 10^6				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	0,000001	55,286948	90,973317	91,908659
Arbre 1d	1,555353	0,020469	0,000603	49,031961
Arbre 2d	1,393792	0,018749	0,000910	1,139594
Rayon : 0.05, Nombre de points : 10^5				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	0,000001	3,603755	5,245774	5,992293
Arbre 1d	0,051058	0,006362	0,000593	12,362265
Arbre 2d	0,046234	0,006398	0,000827	0,919155
Rayon : 0.10, Nombre de points : 10^5				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	<0,000000	3,671344	5,579683	6,833554
Arbre 1d	0,052558	0,006985	0,000687	25,319091
Arbre 2d	0,048573	0,006416	0,000694	3,065354

4 Profondeurs moyennes 3 / 3

✓ - 0 pts Correct

- 3 pts Pas de réponse ou réponse totalement incorrecte
- 0.5 pts Discussion manquante ou incomplète (par ex., pas de mention que l'évolution est bien logarithmique).
- 1 pts Résultats partiellement erronés.
- 1 pts Discussion manquante ou incomplète (par ex., pas de mention que l'évolution est bien logarithmique).
- 2 pts Résultats complètement erronés

4 Profondeur moyenne des noeuds

Ces valeurs ont été moyennées sur 10 essais dans chacun des cas présentés dans le tableau :

Profondeur moyenne			
Taille (N)	10^4	10^5	10^6
Arbre simple	15,42	20,29	24,97
Arbre 2d	15,56	20,42	24,59
Base du log	$\approx 1,80$	$\approx 1,76$	$\approx 1,74$

En théorie, la profondeur moyenne d'un noeud dans un arbre binaire de recherche construit aléatoirement - ce qui est le cas dans notre projet - est $\Theta(\log N)$.

A partir des données du tableau, on peut constater que de la profondeur moyenne d'un noeud suit bien une évolution logarithmique fonction du nombre de noeuds N de l'arbre. La base moyenne du logarithme, sur ces 10 essais, se situe entre 1,74 et 1,80 et est donc très proche de la valeur théorique de 2.

On constate également que l'évolution de la profondeur moyenne est bien similaire dans les deux cas (arbres simples et arbres 2d).

Les résultats de notre algorithme sont donc conformes à la théorie.

5 Temps de calcul des trois approches

Temps de calcul [s] pour un nombre de recherches fixé à 10^4				
Rayon : 0.01, Nombre de points : 10^4				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	0,000001	0,27315	0,384116	0,37011
Arbre 1d	0,001795	0,00177	0,000457	0,068836
Arbre 2d	0,001832	0,002635	0,000678	0,008389
Rayon : 0.01, Nombre de points : 10^5				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	0,000001	3,397066	4,847243	5,142195
Arbre 1d	0,047295	0,005655	0,000483	1,490889
Arbre 2d	0,039891	0,006656	0,000675	0,073700
Rayon : 0.01, Nombre de points : 10^6				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	0,000001	55,286948	90,973317	91,908659
Arbre 1d	1,555353	0,020469	0,000603	49,031961
Arbre 2d	1,393792	0,018749	0,000910	1,139594
Rayon : 0.05, Nombre de points : 10^5				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	0,000001	3,603755	5,245774	5,992293
Arbre 1d	0,051058	0,006362	0,000593	12,362265
Arbre 2d	0,046234	0,006398	0,000827	0,919155
Rayon : 0.10, Nombre de points : 10^5				
Fonction	pdctCreate	pdctExactPosSearch	pdctExactNegSearch	pdctBallSearch
Liste	<0,000000	3,671344	5,579683	6,833554
Arbre 1d	0,052558	0,006985	0,000687	25,319091
Arbre 2d	0,048573	0,006416	0,000694	3,065354

Impact du nombre de points stockés dans la structure

- sur la création du dictionnaire de points :
 - Liste : le nombre de points n'a aucun impact mesurable sur le temps de création du dictionnaire. Cet algorithme a le meilleur temps de création du dictionnaire dans tous les cas.
 - Arbre 1d et Arbre 2d : le temps de création du dictionnaire croît plus vite que le nombre de points car il est multiplié par un facteur de l'ordre de 25 à 35 à chaque fois que le nombre de points est multiplié par 10. Cette croissance conduit à un temps de création du dictionnaire supérieur à 1 seconde dans le cas où le nombre de points est de 10^6 . Les temps de calculs sont fort similaires pour ces deux algorithmes.
- sur la réponse à des requêtes de type exactes :
 - Liste : le temps de calcul a une croissance plus rapide que le nombre de points. Cet algorithme est le plus lent des trois dans tous les cas. On constate également que le temps de calcul en cas de recherche négative est plus élevé qu'en cas de recherche positive, ce qui est logique car on se trouve alors dans le pire cas pour l'algorithme.
 - Arbre 1d et Arbre 2d : le temps de calcul augmente de manière linéaire avec le nombre de points, d'un facteur de l'ordre de 3 donc moins rapidement que le nombre de points. Les temps de calculs sont similaires pour les deux algorithmes. Les temps de calculs sont moins élevés dans tous les cas pour les recherches négatives par rapport aux recherches positives.
- sur la réponse à des requêtes de type boules :
 - Liste : la croissance du temps de calcul est plus rapide que celle du nombre de points, mais cette croissance est moins rapide que celle de l'arbre simple. Néanmoins, c'est cet algorithme qui a le temps de calcul le plus élevé dans tous les cas.
 - Arbre 1d : cet algorithme a la croissance la plus rapide en temps de calcul. Le temps de calcul s'approche même de celui de la liste dans le cas où le nombre de points est de 10^6 et il est donc peu efficace pour de grands nombres de points.
 - Arbre 2d : Pour cet algorithme aussi, le temps de calcul augmente plus rapidement que le nombre de points, mais c'est lui qui a la croissance la plus lente en temps de calcul. Il a également le meilleur temps de calcul dans tous les cas.

Impact du rayon de recherche

- sur la création du dictionnaire de points :
 - Liste : le rayon de recherche n'a aucun impact mesurable sur le temps de création du dictionnaire. Cet algorithme a le meilleur temps de création du dictionnaire dans tous les cas.
 - Arbre 1d et Arbre 2d : le rayon de recherche a un impact extrêmement modéré sur le temps de création du dictionnaire : on obtient des temps de calculs presque identiques pour tous les rayons. Les temps de calculs sont fort similaires pour ces deux algorithmes.
- sur la réponse à des requêtes de type exactes :
 - Liste : le temps de calcul augmente très peu avec le rayon. Cet algorithme est le plus lent des trois dans tous les cas. On constate également que le temps de calcul en cas de recherche négative est plus élevé qu'en cas de recherche positive, ce qui est logique car on se trouve alors dans le pire cas pour l'algorithme.
 - Arbre 1d et Arbre 2d : le temps de calcul augmente très peu avec le rayon. Les temps de calculs et donc les performances sont similaires pour les deux algorithmes. Les temps de calculs sont moins élevés dans tous les cas pour les recherches négatives par rapport aux recherches positives.

- sur la réponse à des requêtes de type boules :
 - Liste : La croissance du temps de calcul est faible par rapport à l'évolution du rayon. Cet algorithme est le moins efficace des trois pour un rayon de 0,01, mais il devient plus efficace que l'algorithme de l'arbre 1d lorsque le rayon augmente.
 - Arbre 1d : le temps de calcul augmente plus vite que la taille du rayon lorsque celui-ci passe de 0,01 à 0,05, puis la croissance du temps de calcul ralentit lorsque le rayon passe de 0,05 à 0,10. Comparativement aux deux autres, c'est cet algorithme qui a la croissance la plus rapide en temps de calcul en valeur. Il est peu efficace pour de grands rayons car il a alors les moins bonnes performances en temps de calculs.
 - Arbre 2d : comme pour l'arbre 1d, le temps de calcul augmente plus vite que la taille du rayon lorsque celui-ci passe de 0,01 à 0,05, puis la croissance du temps de calcul ralentit lorsque le rayon passe de 0,05 à 0,10. Cette croissance du temps de calcul est plus rapide que celle de la liste. Néanmoins, en valeur, il a le meilleur temps de calcul dans tous les cas.

6 Conclusion quant à l'intérêt relatif des trois approches.

On peut en tirer les conclusions suivantes :

- Par liste : C'est l'algorithme qui a le meilleur temps de création du dictionnaire dans tous les cas. Ses performances en temps de calculs sont peu modifiées lorsqu'on augmente la taille du rayon, mais elles diminuent drastiquement lorsqu'on augmente le nombre de points. C'est également l'algorithme le plus lent dans presque tous les cas de figure.
- Par arbre binaire de recherche simple : Ses performances en temps de calcul dans le cas de la recherche dans une boule diminuent drastiquement lorsqu'on augmente le nombre de points comme la taille du rayon. Il est par contre efficace dans tous les cas pour les recherches exactes.
- Par arbre binaire de recherche à deux dimensions : Ses performances en temps de calcul dans le cas de la recherche dans une boule diminuent lorsqu'on augmente le nombre de points comme la taille du rayon, mais ils restent dans des valeurs acceptables et meilleures que celles des deux autres algorithmes dans tous les cas. Il est également efficace dans tous les cas pour les recherches exactes.

En conclusion, l'algorithme de l'arbre binaire de recherche à deux dimensions est à privilégier par rapport aux deux autres car ses performances sont comparativement meilleures.

7 Utilisation de l'implémentation la plus efficace pour trouver l'endroit à Porto qui a la densité de taxis la plus élevée.

Nous avons privilégié l'implémentation par arbre binaire de recherche à deux dimensions.

Nombre de taxis maximal 94049

La position longitude = -8.585478 et latitude = 41.148732

Correspondance avec un endroit à Porto La gare de Campanha

C'est donc la gare de Campahna qui suscite le plus de départ de taxi à Porto.

5 Temps de calcul 5 / 5

✓ - 0 pts Correct

- 5 pts Pas de réponse ou réponse totalement incorrecte
- 1 pts Temps de création dictionnaire manquant (et pas discuté).
- 1.5 pts Discussions manquantes.
- 0.5 pts Petites erreurs ou manques dans la discussion
- 1 pts Discussions incomplètes.
- 0.5 pts Quelques valeurs aberrantes ou manquantes dans les tables.
- 1 pts Temps arbre 2d incorrectes ou manquants.
- 2 pts Impact du rayon pas étudié.

5 La rayon de recherche n'a absolument aucun impact sur la création du dictionnaire et sur les recherches exactes. Il ne fallait pas en parler. Ce que vous observez ne peut être que du à l'aléatoire.

- sur la réponse à des requêtes de type boules :
 - Liste : La croissance du temps de calcul est faible par rapport à l'évolution du rayon. Cet algorithme est le moins efficace des trois pour un rayon de 0,01, mais il devient plus efficace que l'algorithme de l'arbre 1d lorsque le rayon augmente.
 - Arbre 1d : le temps de calcul augmente plus vite que la taille du rayon lorsque celui-ci passe de 0,01 à 0,05, puis la croissance du temps de calcul ralentit lorsque le rayon passe de 0,05 à 0,10. Comparativement aux deux autres, c'est cet algorithme qui a la croissance la plus rapide en temps de calcul en valeur. Il est peu efficace pour de grands rayons car il a alors les moins bonnes performances en temps de calculs.
 - Arbre 2d : comme pour l'arbre 1d, le temps de calcul augmente plus vite que la taille du rayon lorsque celui-ci passe de 0,01 à 0,05, puis la croissance du temps de calcul ralentit lorsque le rayon passe de 0,05 à 0,10. Cette croissance du temps de calcul est plus rapide que celle de la liste. Néanmoins, en valeur, il a le meilleur temps de calcul dans tous les cas.

6 Conclusion quant à l'intérêt relatif des trois approches.

On peut en tirer les conclusions suivantes :

- Par liste : C'est l'algorithme qui a le meilleur temps de création du dictionnaire dans tous les cas. Ses performances en temps de calculs sont peu modifiées lorsqu'on augmente la taille du rayon, mais elles diminuent drastiquement lorsqu'on augmente le nombre de points. C'est également l'algorithme le plus lent dans presque tous les cas de figure.
- Par arbre binaire de recherche simple : Ses performances en temps de calcul dans le cas de la recherche dans une boule diminuent drastiquement lorsqu'on augmente le nombre de points comme la taille du rayon. Il est par contre efficace dans tous les cas pour les recherches exactes.
- Par arbre binaire de recherche à deux dimensions : Ses performances en temps de calcul dans le cas de la recherche dans une boule diminuent lorsqu'on augmente le nombre de points comme la taille du rayon, mais ils restent dans des valeurs acceptables et meilleures que celles des deux autres algorithmes dans tous les cas. Il est également efficace dans tous les cas pour les recherches exactes.

En conclusion, l'algorithme de l'arbre binaire de recherche à deux dimensions est à privilégier par rapport aux deux autres car ses performances sont comparativement meilleures.

7 Utilisation de l'implémentation la plus efficace pour trouver l'endroit à Porto qui a la densité de taxis la plus élevée.

Nous avons privilégié l'implémentation par arbre binaire de recherche à deux dimensions.

Nombre de taxis maximal 94049

La position longitude = -8.585478 et latitude = 41.148732

Correspondance avec un endroit à Porto La gare de Campanha

C'est donc la gare de Campahna qui suscite le plus de départ de taxi à Porto.

6 Conclusion 1 / 1

✓ - 0 pts Correct

- 0.5 pts Ne conclut pas sur les trois approches.

- 1 pts Pas de réponse ou réponse incorrecte

- sur la réponse à des requêtes de type boules :
 - Liste : La croissance du temps de calcul est faible par rapport à l'évolution du rayon. Cet algorithme est le moins efficace des trois pour un rayon de 0,01, mais il devient plus efficace que l'algorithme de l'arbre 1d lorsque le rayon augmente.
 - Arbre 1d : le temps de calcul augmente plus vite que la taille du rayon lorsque celui-ci passe de 0,01 à 0,05, puis la croissance du temps de calcul ralentit lorsque le rayon passe de 0,05 à 0,10. Comparativement aux deux autres, c'est cet algorithme qui a la croissance la plus rapide en temps de calcul en valeur. Il est peu efficace pour de grands rayons car il a alors les moins bonnes performances en temps de calculs.
 - Arbre 2d : comme pour l'arbre 1d, le temps de calcul augmente plus vite que la taille du rayon lorsque celui-ci passe de 0,01 à 0,05, puis la croissance du temps de calcul ralentit lorsque le rayon passe de 0,05 à 0,10. Cette croissance du temps de calcul est plus rapide que celle de la liste. Néanmoins, en valeur, il a le meilleur temps de calcul dans tous les cas.

6 Conclusion quant à l'intérêt relatif des trois approches.

On peut en tirer les conclusions suivantes :

- Par liste : C'est l'algorithme qui a le meilleur temps de création du dictionnaire dans tous les cas. Ses performances en temps de calculs sont peu modifiées lorsqu'on augmente la taille du rayon, mais elles diminuent drastiquement lorsqu'on augmente le nombre de points. C'est également l'algorithme le plus lent dans presque tous les cas de figure.
- Par arbre binaire de recherche simple : Ses performances en temps de calcul dans le cas de la recherche dans une boule diminuent drastiquement lorsqu'on augmente le nombre de points comme la taille du rayon. Il est par contre efficace dans tous les cas pour les recherches exactes.
- Par arbre binaire de recherche à deux dimensions : Ses performances en temps de calcul dans le cas de la recherche dans une boule diminuent lorsqu'on augmente le nombre de points comme la taille du rayon, mais ils restent dans des valeurs acceptables et meilleures que celles des deux autres algorithmes dans tous les cas. Il est également efficace dans tous les cas pour les recherches exactes.

En conclusion, l'algorithme de l'arbre binaire de recherche à deux dimensions est à privilégier par rapport aux deux autres car ses performances sont comparativement meilleures.

7 Utilisation de l'implémentation la plus efficace pour trouver l'endroit à Porto qui a la densité de taxis la plus élevée.

Nous avons privilégié l'implémentation par arbre binaire de recherche à deux dimensions.

Nombre de taxis maximal 94049

La position longitude = -8.585478 et latitude = 41.148732

Correspondance avec un endroit à Porto La gare de Campanha

C'est donc la gare de Campahna qui suscite le plus de départ de taxi à Porto.

7 Endroit de Porto 1 / 1

✓ - 0 pts Correct

- 0.5 pts Nombre de taxis pas donné.

- 1 pts Pas de réponse ou réponse incorrecte



UNIVERSITÉ DE LIÈGE
FACULTÉ DES SCIENCES APPLIQUÉES

Projet 2 : arbres binaires de recherches

INFO0902-1 : Structure de données et algorithmes

Auteurs :
Catherine DUCHEMIN s202046
Manon GERARD s201354

Professeur :
P. GEURTS

3^e année de Bachelier Ingénieur Civil
Année académique 2022 - 2023