Rapport Algorithmique

Projet 2 : Analyse des Transactions Bancaires avec les Arbres Binaires de Recherche

INFO-F-103 — Algorithmique I

Jordan Matin

Université Libre de Bruxelles (ULB)

Année académique 2022-2023

12 février 2026

Table des matières

1 Introduction **3**

2 Partie 1 : Dénombrement des ABR Structurellement Uniques **3**
2.1 Énoncé du problème . . . 3
2.2 Analyse et raisonnement . . . 3
2.3 Cas de base . . . 3
2.4 Illustration pour $T(3)$ . . . 4
2.5 Formule de récurrence . . . 4
2.6 Vérification . . . 4
2.7 Lien avec les nombres de Catalan . . . 5
2.8 Complexité de l'approche récursive . . . 5
2.9 Bonus : Méthode non récursive . . . 5

3 Partie 2 : Identification des Transactions les Plus Importantes **5**
3.1 Énoncé du problème . . . 5
3.2 Approche récursive . . . 6
3.2.1 Principe . . . 6
3.2.2 Implémentation . . . 6
3.2.3 Analyse de complexité . . . 6
3.3 Approche itérative . . . 7
3.3.1 Principe . . . 7
3.3.2 Implémentation . . . 7
3.3.3 Analyse de complexité . . . 7
3.4 Comparaison des deux approches . . . 8
3.5 Conclusion partielle . . . 8

4 Partie 3 : Correction des Transactions **8**
4.1 Énoncé du problème . . . 8

4.2 Exemple . . . 8
4.3 Approche implémentée . . . 9
4.3.1 Algorithme . . . 9
4.3.2 Analyse de complexité . . . 9
4.4 Approche alternative en $O(n \log n)$ . . . 9
4.4.1 Idée . . . 9
4.4.2 Structure de données proposée . . . 9
4.4.3 Pseudo-code . . . 10
4.4.4 Analyse de complexité . . . 10
4.5 Comparaison des approches . . . 10

5 Partie 4 : Fusion de Systèmes **11**
5.1 Énoncé du problème . . . 11
5.2 Approche implémentée . . . 11
5.2.1 Algorithme . . . 11
5.2.2 Implémentation . . . 11
5.3 Analyse de complexité . . . 12
5.4 Exemple . . . 12
5.5 Optimalité de l'approche . . . 12

6 Conclusion **12**
6.1 Récapitulatif des complexités . . . 13
6.2 Enseignements . . . 13
6.3 Perspectives . . . 13

Références **14**

1 Introduction

Dans le cadre du cours INFO-F-103 (Algorithmique I), ce projet vise à développer un système de gestion de transactions bancaires basé sur les **Arbres Binaires de Recherche** (ABR, ou *Binary Search Trees* — BST en anglais).

Les arbres binaires de recherche sont des structures de données hiérarchiques qui permettent d'organiser et de manipuler des données de manière efficace. Ils offrent des avantages significatifs en termes de performance pour les opérations d'insertion, de recherche et de suppression, avec une complexité moyenne de $O(\log n)$.

Ce rapport présente l'analyse algorithmique des différentes parties du projet :

1. **Partie 1** : Dénombrement des arbres binaires de recherche structurellement uniques
2. **Partie 2** : Identification des transactions les plus importantes
3. **Partie 3** : Correction des transactions
4. **Partie 4** : Fusion de systèmes de transactions

Pour chaque partie, nous présentons l'approche algorithmique adoptée ainsi qu'une analyse détaillée des complexités temporelles et spatiales.

2 Partie 1 : Dénombrement des ABR Structurellement Uniques

2.1 Énoncé du problème

Étant donné un entier N, il s'agit de déterminer combien d'arbres binaires de recherche **structurellement uniques** peuvent être construits à partir des identifiants allant de 1 à N.

2.2 Analyse et raisonnement

Considérons une liste d'entiers $[1, 2, 3, \ldots, N]$. En choisissant un indice i dans cette liste comme racine de l'arbre, nous observons que :

— Il y a $i - 1$ entiers à gauche de i (qui formeront le sous-arbre gauche)
— Il y a $N - i$ entiers à droite de i (qui formeront le sous-arbre droit)

Puisque chaque entier de la liste peut servir de racine, nous pouvons construire une relation de récurrence.

2.3 Cas de base

Définissons $T(n)$ comme le nombre d'ABR structurellement uniques pour n nœuds :

— $T(0) = 1$ (l'arbre vide est considéré comme un arbre valide)

— $T(1) = 1$ (un seul nœud forme un seul arbre)
— $T(2) = 2$ (deux configurations possibles)
— $T(3) = 5$ (cinq configurations possibles)

2.4 Illustration pour $T(3)$

Pour $N = 3$, les cinq arbres possibles sont :

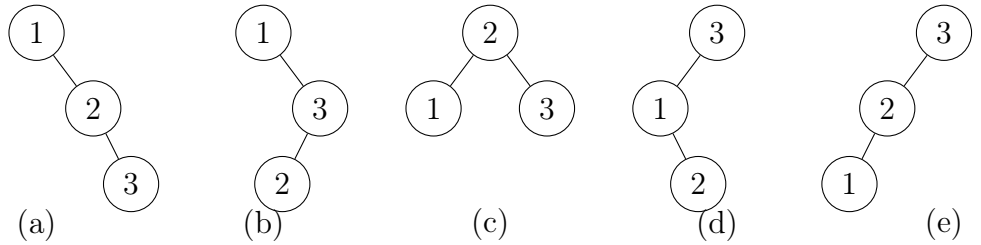

FIGURE 1 – Les 5 arbres binaires de recherche structurellement uniques pour $N = 3$

2.5 Formule de récurrence

En analysant la structure, nous déduisons que pour chaque choix de racine i (où $1 \leq i \leq N$), le nombre total d'arbres est le produit du nombre d'arbres possibles dans le sous-arbre gauche et dans le sous-arbre droit.

La formule générale est donc :

$$\boxed{T(N) = \sum_{i=1}^{N} T(i-1) \times T(N-i)} \tag{1}$$

Où :

— $T(i-1)$ représente le nombre d'ABR possibles avec les $i-1$ éléments du sous-arbre gauche
— $T(N-i)$ représente le nombre d'ABR possibles avec les $N-i$ éléments du sous-arbre droit

2.6 Vérification

Vérifions la formule pour $T(3)$:

$$\begin{aligned} T(3) &= T(0) \times T(2) + T(1) \times T(1) + T(2) \times T(0) \\ &= 1 \times 2 + 1 \times 1 + 2 \times 1 \\ &= 2 + 1 + 2 \\ &= 5 \quad \checkmark \end{aligned}$$

2.7 Lien avec les nombres de Catalan

Cette formule correspond exactement à la définition des **nombres de Catalan**. Le n-ième nombre de Catalan est donné par :

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)! \cdot n!} \tag{2}$$

Ainsi, $T(N) = C_N$.

2.8 Complexité de l'approche récursive

Aspect	Sans mémoïsation	Avec mémoïsation
Complexité temporelle	$O(3^n)$ (exponentielle)	$O(n^2)$
Complexité spatiale	$O(n)$ (pile d'appels)	$O(n)$ (cache + pile)

TABLE 1 – Analyse de complexité pour le dénombrement des ABR

2.9 Bonus : Méthode non récursive

En utilisant la formule fermée des nombres de Catalan, il est possible de calculer $T(N)$ en $O(n)$ sans récursion :

```python
def count_unique_bst(n):
    if n <= 1:
        return 1

    catalan = [0] * (n + 1)
    catalan[0] = 1
    catalan[1] = 1

    for i in range(2, n + 1):
        for j in range(i):
            catalan[i] += catalan[j] * catalan[i - j - 1]

    return catalan[n]
```

Listing 1 – Calcul itératif du nombre d'ABR uniques

3 Partie 2 : Identification des Transactions les Plus Importantes

3.1 Énoncé du problème

Étant donné un arbre binaire de recherche de transactions et un entier positif j, il s'agit de trouver la j-ième plus grande valeur de `transaction_amount` dans l'arbre.

3.2 Approche récursive

3.2.1 Principe

L'approche récursive repose sur un **parcours en ordre inverse** (*reverse in-order traversal*), c'est-à-dire : sous-arbre droit → racine → sous-arbre gauche.

Ce parcours visite les nœuds par ordre décroissant de leur clé. En maintenant un compteur, nous pouvons identifier le j-ième élément visité.

3.2.2 Implémentation

```
def jemePlusImportante(self, j):
    """Retourne la j-ieme plus grande transaction."""
    compteur = [0]  # Liste pour permettre la modification dans la recursion
    resultat = [None]
    self._jemePlusImportanteUtil(self.root, j, compteur, resultat)
    return resultat[0]

def _jemePlusImportanteUtil(self, node, j, compteur, resultat):
    if node is None or resultat[0] is not None:
        return

    # Parcours du sous-arbre droit (valeurs plus grandes)
    self._jemePlusImportanteUtil(node.right, j, compteur, resultat)

    # Traitement du noeud courant
    compteur[0] += 1
    if compteur[0] == j:
        resultat[0] = node
        return

    # Parcours du sous-arbre gauche (valeurs plus petites)
    self._jemePlusImportanteUtil(node.left, j, compteur, resultat)
```

Listing 2 – Méthode récursive `jemePlusImportante`

3.2.3 Analyse de complexité

— **Complexité temporelle** : $O(n)$ dans le pire des cas, car il peut être nécessaire de visiter tous les nœuds si $j = n$.

— **Complexité spatiale** : $O(h)$ où h est la hauteur de l'arbre, due à la pile d'appels récursifs. Dans le pire des cas (arbre dégénéré), $h = n$, donc $O(n)$.

3.3 Approche itérative

3.3.1 Principe

L'approche itérative utilise une pile explicite pour simuler le parcours en ordre inverse. Cette méthode évite les appels récursifs et offre un meilleur contrôle sur le flux d'exécution.

3.3.2 Implémentation

```
def jemePlusImportanteIterative(self, j):
    """Version iterative pour trouver la j-ieme plus grande
       transaction."""
    stack = []
    current = self.root
    compteur = 0

    while stack or current:
        # Aller le plus a droite possible
        while current:
            stack.append(current)
            current = current.right

        # Traiter le noeud
        current = stack.pop()
        compteur += 1

        if compteur == j:
            return current

        # Aller a gauche
        current = current.left

    return None  # j depasse le nombre de noeuds
```

Listing 3 – Méthode itérative `jemePlusImportanteIterative`

3.3.3 Analyse de complexité

— **Complexité temporelle** : $O(h + j)$ où h est la hauteur de l'arbre. Dans le pire des cas, $O(n)$.

— **Complexité spatiale** : $O(h)$ pour la pile. Dans le pire des cas, $O(n)$.

3.4 Comparaison des deux approches

Critère	**Récursive**	**Itérative**	**Préférence**
Complexité temporelle	$O(n)$	$O(h+j)$	Itérative
Complexité spatiale	$O(h)$	$O(h)$	Équivalent
Lisibilité	Bonne	Moyenne	Récursive
Risque de stack overflow	Oui	Non	Itérative
Arrêt anticipé	Possible	Plus naturel	Itérative

TABLE 2 – Comparaison des approches récursive et itérative

3.5 Conclusion partielle

L'approche itérative est préférable pour les raisons suivantes :

1. Elle permet un arrêt anticipé plus naturel dès que le j-ième élément est trouvé.
2. Elle évite le risque de dépassement de pile pour les arbres très profonds.
3. Elle ne nécessite pas de stocker tous les nœuds dans une liste triée.

4 Partie 3 : Correction des Transactions

4.1 Énoncé du problème

Pour chaque transaction dans un fichier, remplacer son montant par le **montant immédiatement supérieur** parmi les transactions qui la suivent dans le fichier. Si aucun montant supérieur n'existe, remplacer par -1.

4.2 Exemple

Fichier original		**Fichier corrigé**	
ID	Montant	ID	Nouveau montant
1	160	1	360
2	1160	2	1260
3	1420	3	1600
4	360	4	500
5	620	5	640
⋮	⋮	⋮	⋮

TABLE 3 – Exemple de correction des transactions

4.3 Approche implémentée

4.3.1 Algorithme

L'algorithme procède comme suit :

1. Parcourir le fichier et construire un ABR avec toutes les transactions.
2. Pour chaque transaction (dans l'ordre du fichier) :
   — Effectuer un parcours en ordre (in-order) pour obtenir une liste triée.
   — Chercher le montant immédiatement supérieur parmi les transactions suivantes.
   — Remplacer le montant ou mettre -1 si aucun montant supérieur n'existe.
3. Écrire les résultats dans le fichier de sortie.

4.3.2 Analyse de complexité

— **Parcours in-order** : $O(n)$ pour obtenir la liste triée.
— **Recherche du successeur** : $O(n)$ dans le pire des cas (boucle while).
— **Répétition pour chaque transaction** : n transactions.

Complexité totale :

— **Temporelle** : $O(n^2)$ dans le pire des cas.
— **Spatiale** : $O(n)$ pour stocker la liste triée.

Optimisation implémentée : Une fonction force l'arrêt de la boucle lorsqu'il n'y a pas d'élément plus grand, ce qui améliore les performances en pratique.

4.4 Approche alternative en $O(n \log n)$

4.4.1 Idée

En parcourant les transactions **de droite à gauche** (du dernier au premier), nous pouvons maintenir une structure de données ordonnée qui permet de trouver efficacement le successeur de chaque montant.

4.4.2 Structure de données proposée

Utiliser un **ABR auto-équilibré** (comme un arbre AVL ou un arbre Rouge-Noir) ou un **ensemble ordonné** (*TreeSet* en Java, `SortedList` en Python).

4.4.3 Pseudo-code

Algorithm 1 Correction des transactions en $O(n \log n)$

Entrée : Liste de transactions $T = [t_1, t_2, \ldots, t_n]$
Sortie : Liste des montants corrigés $R = [r_1, r_2, \ldots, r_n]$

structure $\leftarrow$ Ensemble ordonné vide
$R \leftarrow$ Liste de taille n initialisée à -1

for $i = n$ **jusqu'à** 1 **do**
    montant $\leftarrow T[i].montant$
    successeur $\leftarrow$ *structure*.trouverSuccesseur(*montant*)
    if *successeur* existe **then**
        $R[i] \leftarrow$ *successeur*
    else
        $R[i] \leftarrow -1$
    end if
    structure.inserer(*montant*)
end for

return R

4.4.4 Analyse de complexité

— **Insertion** dans un ABR équilibré : $O(\log n)$
— **Recherche du successeur** : $O(\log n)$
— **Répétition** pour n éléments : n fois

Complexité totale : $O(n \log n)$

4.5 Comparaison des approches

Aspect	Approche implémentée	Approche optimale
Complexité temporelle	$O(n^2)$	$O(n \log n)$
Complexité spatiale	$O(n)$	$O(n)$
Structure utilisée	ABR simple	ABR équilibré / TreeSet
Difficulté d'implémentation	Moyenne	Plus élevée

TABLE 4 – Comparaison des approches pour la correction des transactions

L'approche optimale est théoriquement meilleure, mais l'approche implémentée exploite directement les propriétés des ABR comme demandé dans l'énoncé.

5 Partie 4 : Fusion de Systèmes

5.1 Énoncé du problème

Étant donné deux arbres binaires de recherche, fusionner leurs éléments en une seule liste triée suivant un parcours en ordre (in-order).

5.2 Approche implémentée

5.2.1 Algorithme

L'algorithme procède en deux étapes :

1. **Parcours in-order** de chaque arbre pour obtenir deux listes triées.
2. **Fusion** (*merge*) des deux listes triées en une seule liste triée.

5.2.2 Implémentation

```python
def fusion(arbre1, arbre2):
    """Fusionne deux ABR en une liste triee."""
    # Etape 1 : Parcours in-order des deux arbres
    liste1 = arbre1.inorder_traversal()  # O(n)
    liste2 = arbre2.inorder_traversal()  # O(m)

    # Etape 2 : Fusion des deux listes triees
    return merge_sorted_lists(liste1, liste2)  # O(n + m)

def merge_sorted_lists(list1, list2):
    """Fusionne deux listes triees en une seule liste triee."""
    result = []
    i, j = 0, 0

    while i < len(list1) and j < len(list2):
        if list1[i] <= list2[j]:
            result.append(list1[i])
            i += 1
        else:
            result.append(list2[j])
            j += 1

    # Ajouter les elements restants
    result.extend(list1[i:])
    result.extend(list2[j:])

    return result
```

Listing 4 – Fusion de deux ABR

5.3 Analyse de complexité

Soit n le nombre de nœuds dans le premier arbre et m le nombre de nœuds dans le second arbre.

- **Parcours in-order du premier arbre** : $O(n)$
- **Parcours in-order du second arbre** : $O(m)$
- **Fusion des deux listes triées** : $O(n + m)$

Complexité totale :

- **Temporelle** : $O(n + m)$
- **Spatiale** : $O(n + m)$ pour stocker les listes et le résultat

5.4 Exemple

Arbre 1 Arbre 2

FIGURE 2 – Exemple de fusion de deux ABR

- **Liste 1** (in-order de l'arbre 1) : $[1, 3]$
- **Liste 2** (in-order de l'arbre 2) : $[2, 4, 5, 6]$
- **Résultat de la fusion** : $[1, 2, 3, 4, 5, 6]$

5.5 Optimalité de l'approche

L'algorithme de fusion utilisé est le même que celui du **Merge Sort** pour la phase de fusion. Sa complexité de $O(n + m)$ est **optimale** car il est nécessaire de parcourir tous les éléments des deux arbres au moins une fois.

6 Conclusion

Ce projet nous a permis d'explorer en profondeur les arbres binaires de recherche et leurs applications dans un contexte de gestion de transactions bancaires.

6.1 Récapitulatif des complexités

Partie	**Complexité temporelle**	**Complexité spatiale**
1. Dénombrement ABR	$O(n^2)$ avec mémoïsation	$O(n)$
2. j-ième plus importante (récursif)	$O(n)$	$O(h)$
2. j-ième plus importante (itératif)	$O(h+j)$	$O(h)$
3. Correction des transactions	$O(n^2)$ / $O(n \log n)$	$O(n)$
4. Fusion de systèmes	$O(n+m)$	$O(n+m)$

TABLE 5 – Récapitulatif des complexités pour chaque partie

6.2 Enseignements

1. Les **ABR** offrent une structure efficace pour organiser des données ordonnées.
2. Le choix entre approche **récursive et itérative** dépend du contexte et des contraintes.
3. Les **nombres de Catalan** apparaissent naturellement dans le dénombrement des structures arborescentes.
4. L'algorithme de **fusion** de listes triées est un outil fondamental en algorithmique.

6.3 Perspectives

Pour aller plus loin, il serait intéressant d'explorer :

— Les arbres auto-équilibrés (AVL, Rouge-Noir) pour garantir des complexités logarithmiques.

— Les applications en cryptographie et en sécurité des systèmes financiers.

— Les structures de données alternatives comme les B-arbres pour le stockage sur disque.

Références

Références

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.

[2] Stanley, R. P. (2015). *Catalan Numbers*. Cambridge University Press.

[3] Sedgewick, R., & Wayne, K. (2011). *Algorithms* (4th ed.). Addison-Wesley.