Université de sciences et de la technologie de Houari Boumediene

Master Systèmes Informatiques Intelligents (M1)

RAPPORT DE TP complexité

Arbres binaires

FERKOUS SARAH

08/12/2022

Introduction générale

Un arbre est un cas particulier de graphe non orienté connexe qui n'admet pas de cycle. Un sommet d'un arbre est appelé nœud.

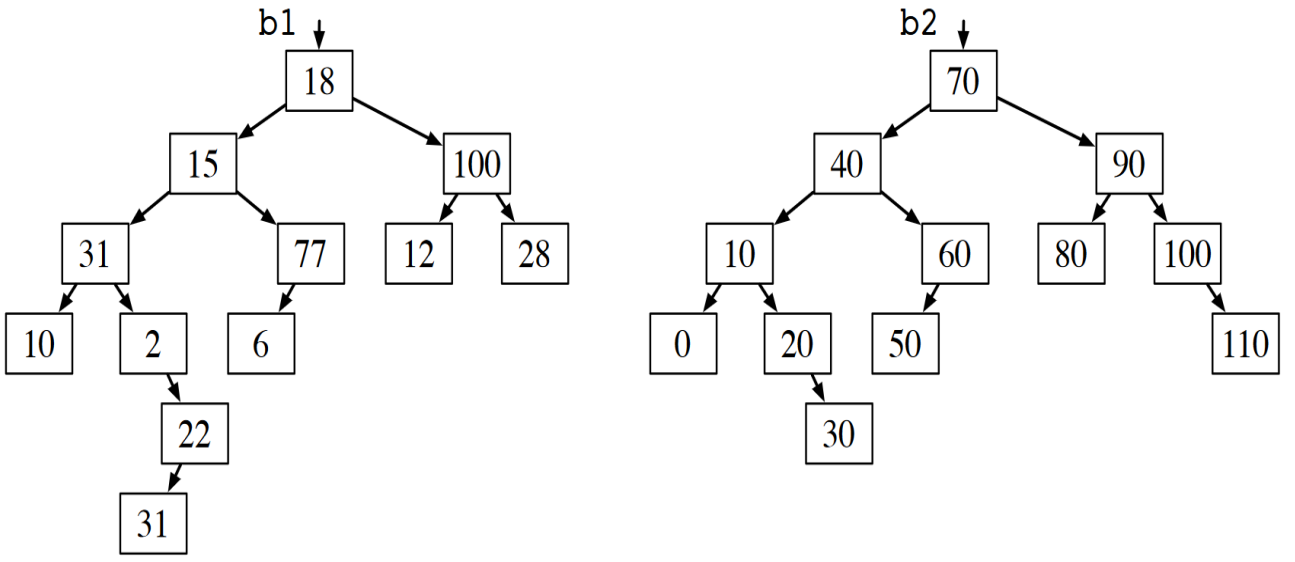
Un arbre binaire est un arbre dans lequel chaque nœud a au plus deux successeurs appelés respectivement fils gauche et fils droit.

On va étudier dans ce TP :

- La recherche d'un élément dans un arbre binaire.
- La recherche du minimum dans un arbre binaire.
- Le parcours en largeur d'un arbre binaire.

On représentera les deux versions récursive et itérative pour chaque problème, et aussi le calcule comparative de la complexité pour atteindre la meilleur solution entre eux.

Tout ça va être préparé à l'aide des autres structures de données qui s'appelle : les structures de donnes élémentaires : les piles, les files et les tableaux.



Partie 1 :

Explication détaillée des algorithmes utilisés

1. Un algorithme qui teste si un élément est dans l'arbre binaire.
   - **La version récursive**

     Le pseudo code :

```
entier findElementRec (bt : BTree, e : Element)
Debut
  Si (arbrevide (bt)) Alors  retourner 0 ;  //tester si l'arbre est vide ou pas
    Si non
      si (root (bt) == e)  Alors   retourner 1 ;   //élément trouve
        Si non retourner findElementRec(leftChild(bt),e)||findElementRec(rightChild(bt),e)
        //des appels récursives
      Fsi ;
  Fsi ;
Fin.
```

Explication :

Cette version de recherche consiste à faire des appels récursifs, à la racine de l'arbre en première fois puis son fils gauche ou fils de droite… et on s'arrête si l'élément est trouvé ou bien on a parcouru tous l'arbre.

- **La version itérative**

Le pseudo code :

```
entier findElemIter (x : *Node, e : Element)
Debut
  Si(x == NULL) Alors  retourner 0 ;  Fsi ;    //si l'arbre est vide on fait rien
  pt : *pile ;     //déclaration d'une pile qui vas contenir le noud et ses deux fils
  pt = new_pile(3) ;
  Empiler (pt, x) ;   //sauvegarder la racine
  Tant que(nonpilevide(pt))
  Faire
     x : *Node ;
     x = depiler(pt) ;  //on depile l'ement de la tete
     Si(x.elem == e) Alors retourner 1 ;  Fsi ;   //on teste s'il est égale a l'élément cherche
     Si(x.left) Alors empiler(pt, x.left) ;  Fsi ;   //si non on empile le FG s'il existe
     Si(x.right) Alors empiler(pt, x.right) ;  Fsi ;  //si non on empile le FD s'il existe

  Faits
    retourner 0 ;   //dans le cas où l'élément n'existe pas
Fin.
```

Explication :

On a utilisé une pile qui va sauvegarder l'élément racine et ses deux fils (gauche et droite), on insère à chaque fois l'élément racine après on le dépile et on insère ses deux fils et on reboucle avec tant que qui dépile le premier élément de la pile et tester si c'est l'élément recherché si non empiler ses deux fils…

2. Un algorithme qui renvoie le minimum des valeurs de l'arbre.

- **La version récursive**

Le pseudo code :

```
Element findMinRec (bt : BTree)
Debut
 Si (arbrevide(bt)) Alors ecrire('min impossible') ; Fsi ;  //si l'arbre vide élément n'existe pas
 Si (isLeaf(bt)) Alors root(bt) ;  //dans le cas où l'arbre contient un seul nœud
    Si non    m : Element ;
              m = root(bt) ;   //on initialise le min a la racine
              Si(non arbrevide(leftChild(bt))) Alors  //descend à gauche tant que possible
                   m=min(m, findMinRec(leftChild(bt))) ; //appel fct min prédéfinie
              Fsi ;
              Si(non arbrevide(rightChild(bt))) Alors  //descend à droite tant que possible
                   m=min(m, findMinRec(rightChild(bt))) ;  //appel fct min prédéfinie
              Fsi ;
              retourner m ;  //on retourne le min après le parcours de tous les nœuds
 Fsi ;
Fin.
```

Explication :

Un algorithme récursif qui envois l'élément racine comme minimum si l'arbre contient un seul nœud, sinon on initialise le minimum à la racine et on fait un parcours d'arbre avec comparaison entre le minimum choisit et le nœud courant et on fait une permutation dans le cas où le nœud courant est inférieur au minimum choisit, la condition d'arrêt c'est quand l'arbre devient vide.

- **La version itérative**

Le pseudo code :

```
entier findMinIter (x : *Node)
Debut
  x : entier ;
  si  (x == NULL) Alors retourner 0 ;  //si l'arbre vide élément n'existe pas
    si non  min = root(x) ;  //on fixe le min a la racine
  Fsi ;
  pt : *pile ;   //déclaration d'une pile pour le parcours itératif de l'arbre
  pt = new_pile(3) ;
  Empiler (pt, x) ;  //empiler la racine de l'arbre
  Tant que (nonpilevide(pt))  //TQ l'arbre n'est pas vide
  Faire
    x : *Node ;
    x = depiler(pt) ;  //on dépile le premier élément a la tête, première fois sera la racine
    Si (x.elem < min) Alors min = x.elem ; Fsi ; //si element empiler<min on mis a jour le min
    Si (x.left) Alors empiler (pt, x.left) ;  Fsi ;  //on avance à gauche dans l'arbre
    Si (x.right) Alors empiler (pt, x.right) ;  Fsi ;  //on avance à droite dans l'arbre
  Faits ;
  retourner min ;
Fin.
```

Explication :

Si l'arbre est vide alors le min n'existe pas, dans le cas contraire on initialise le min à la racine, on empile la racine et on rentre dans la boucle tant que et on dépile le premier élément qui est à la tête de pile, on fait la comparaison avec l'élément dépilé et le min. Si ce dernier est inférieur au min on permute entre les deux, sinon on continue notre parcours avec empilement de fils gauche et droit de l'l'élément dépiler et on reboucle.

3. Un algorithme qui affiche les nœuds de l'arbre dans l'ordre d'un parcours en largeur

- **La version récursive**

Le pseudo code :

```
entier height(node : *Node)   //retourner le profondeur de l'arbre
Debut
 Si (node == NULL)  Alors retourner 0 ;
  Si non
   Lheight = height(node.left) ;   //subtree left
   Rheight = heigh(node.right) ;   //subtree right
   Si (lheigh > rheigt)   //on choisit le max entre les deux
       retourner (lheight + 1) ;
       si non  retourne (rheight + 1) ;
  Fsi ;
 Fsi ;
Fin.
```

Explication :

Une fonction qui retourne le nombre la hauteur d'un arbre, elle prend le nombre de nouds maximum entre les une branches.

```
printCurrentLevel (root : *Node, level :entier)  //afficher les nœud de niveau actuel
Debut
  Si (root == NULL)  Alors retourner 0 ;  Fsi ;
  Si (level == 1)   Alors ecrire('root.elem') ;  //partie d'affichage des nœuds
    Si non si (level > 1)  Alors printCurrentLevel(root.left, level - 1) ;
                                 printCurrentLevel(root.right, level - 1) ;
            Fsi ;
  Fsi ;
Fin.
```

Explication :

Procédure récursive qui affiche tous les nœuds existent dans un niveau, si l'arbre est vide la valeur retournée est 0.
Si l'arbre n'est pas vide, si le niveau égal à 1 on affiche la racine si non appel récursive pour le fils gauche et le fils droit.

```
printWidthOrderRec(root : *Node)   //fonctionqui affiche le niveau dans un arbre
Debut
  h, i : entier ;
   h = height(root) ;
  pour(i de 1 à h)  //pour chaque niveau on fait une appel récursive
  Faire
    printCurrentLevel(root, i) ;
  Fait ;
Fin.
```

Explication :

On sauvegarde la hauteur de l'arbre dans la variable h, on fait appel à la fonction printCurrentLevel h fois avec le numéro du niveau à chaque fois.

- **La version itérative**

Le pseudo code :

```
printWidhtOrderIter(x : *Node)
Debut
  Si(x == NULL) Alors ecrire('Arbre vide') ;
    Si non
            file *q ;
            q = file_new() ;
            enfiler (q, x) ;   //enfiler la racine
            Tant que (q != NULL)   //TQ la file n'est pas vide
            Faire
              n : *Node ;
              n = defiler(q) ;  //on défile le premier élément en tête de file
              ecrire (root(n)) ;  //on l'affiche
              Si (n.left) Alors enfiler (q, n.left) ;   //avancer à gauche
              Si (n.right) Alors enfiler (q, n.right) ;   //avancer à droite
            Fait ;
Fsi ;
Fin.
```

Explication :

Le parcours itératif d'un arbre en largeur nécessite une file d'attente, on enfile la racine comme première itération après on entre dans la boucle tant que la file n'est pas vide on défile, on affiche, et on enfile les deux fils gauche et droit et on reboucle.

Partie 2 :

Calcule de la complexité théorique

Initiation

La complexité théorique a l'avantage de ne dépendre ni de l'ordinateur, ni du langage de programmation, ni même de l'astuce du programmeur. Elle permet de d'éviter l'effort de programmer inutilement un algorithme inefficace, comme elle peut renseigner sur l'efficacité des algorithmes étudiés.

I. La fonction **findElementRec()** :
La recherche se fait en parcourant les nœuds de l'arbre jusqu'à atteindre l'élément recherché. Au pire cas l'élément n'existe pas et donc le parcours se fait pour tous les nœuds de l'arbre, la complexité sera en **O(n)** tel que **n** est le nombre des nouds dans cet arbre.

II. La fonction **findElementIter()** :
Cette recherche a une complexité polynomiale linéaire **O(n)** car au pire cas l'élément n'existe pas et on passe par tous les nœuds (on rentre dans la boucle tant que).

Comparaison entre les deux versions théoriquement:
Les deux méthodes ont la même complexité **O(n)** mais l'algorithme récursif est meilleur, car il a moins de nombre d'instructions en plus l'algorithme itératif utilise une pile (plein d'empilement et de dépilement), en terme de temps d'exécution et d'espace occupé la version récursive est plus performant que la version itérative.

III. La fonction **findMinRec()** :
-Au pire cas le minimum est une feuille.
-On a que des branchements conditionnels et il n'y a pas des boucles.
-L'appel récursif se fait **n** fois (**n** nœuds avant le minimum).
-On a aussi un appel à la fonction **min ()** qui contient une seul instruction uniquement.
Et donc la complexité est en **O(n)**.

IV. La fonction **findMinIter()** :
Elle contient une boucle tant que qui va contenir tous les nœuds de l'arbre au pire cas, et donc la complexité : **O(n)**.

Comparaison entre les deux versions théoriquement:
Les deux versions sont en O(n), on décide qui est la meilleur avec le calcule de temps d'exécution dans la partie qui suit.

V. La fonction **printWidthOrderRec()** :
Cette fonction fait des appels à la fonction printCurrentLevel() pour chaque niveau de l'arbre, supposons que l'arbre contient **m** niveaux . et cette printCurrentLevel() fait des appels récursives a lui-même , supposons **n** le nombre des nœuds dans ce niveaux.
La complexité est **n*m**, au pire cas **n=m** et donc la complexité est en **O(n^2)** 'Complexité **quadratique**'.

VI. La fonction **printWidthOrderIter()** :
L'affichage d'un arbre consiste à parcourir tous les nœuds de l'arbre qui est **n**, dans cette fonction on a une boucle tant que pour la file qui va contenir tous les nœuds de l'arbre et donc la complexité est en **O(n)** '**Complexité linéaire**'.

Comparaison entre les deux versions théoriquement:
La version itérative est plus performante que la version récursive car :
O(n) < O(n^2) comme se figure ci-dessous (Figure 0):

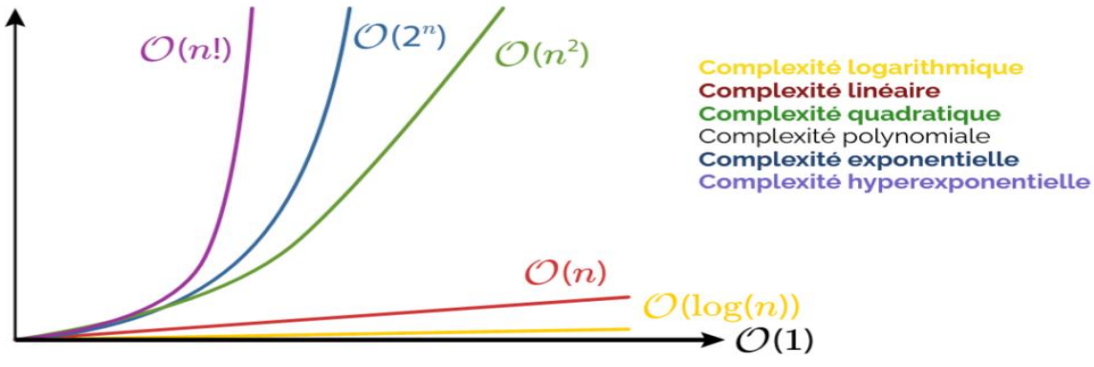


Figure 0 : photo explicative des types de complexité.

Parie 3 :

Illustration graphique des temps d'exécution des différents algorithmes

Algorithme	version	10000	50000	70000	80000	100000	500000	900000	1000000
`findElem`	récursive	0.00	0.001	0.001	0.00	0.002	0.00	0.017	0.015
	itérative	0.00	0.001	0.001	0.001	0.001	0.02	0.041	0.040
`findMin`	récursive	0.00	0.009	0.003	0.009	0.011	0.096	0.194	0.184
	itérative	0.00	0.01	0.00	0.01	0.00	0.061	0.102	0.101
`printWidthOrder`	récursive	1.1820	9.3250	14.367	18.862	26.417	/	/	/
	itérative	0.01	0.01	0.31	0.62	0.73	/	/	/

Tableau 1 : Résultats d'exécution des six algorithmes

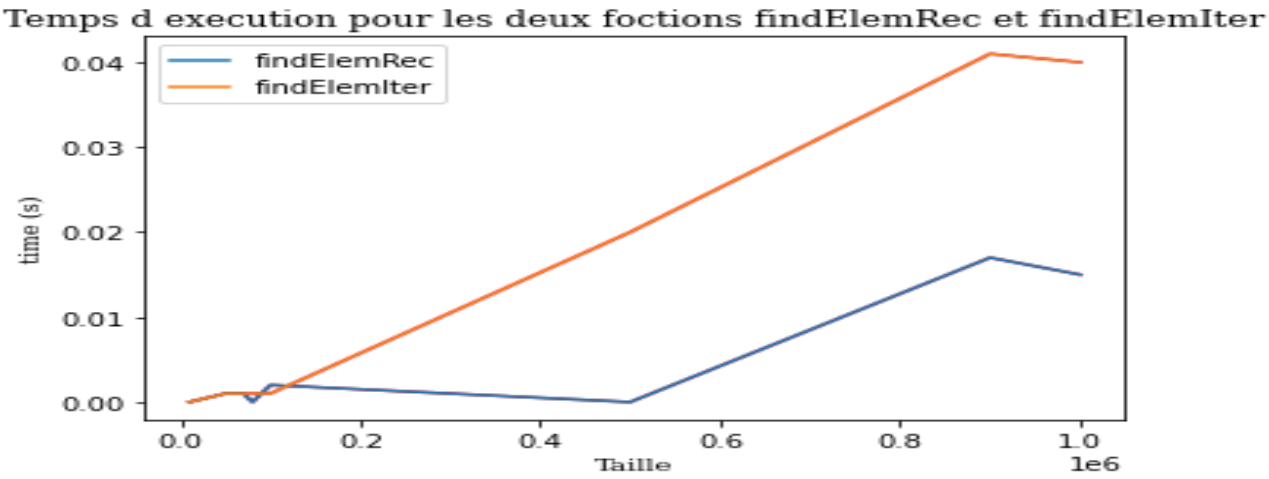


Figure 1 : Représentation graphique comparative des résultats d'exécution de la fonction findElemen

Analyse et comparaison :

On remarque à partir de **Figure 1** que les deux courbes sont presque les même pour des petites tailles, mais quand la taille augmente la courbe itérative passe plus rapide que celle de récursive. Malgré que les deux ont une complexité quadratique mais la version récursive est plus performante que la version itérative.

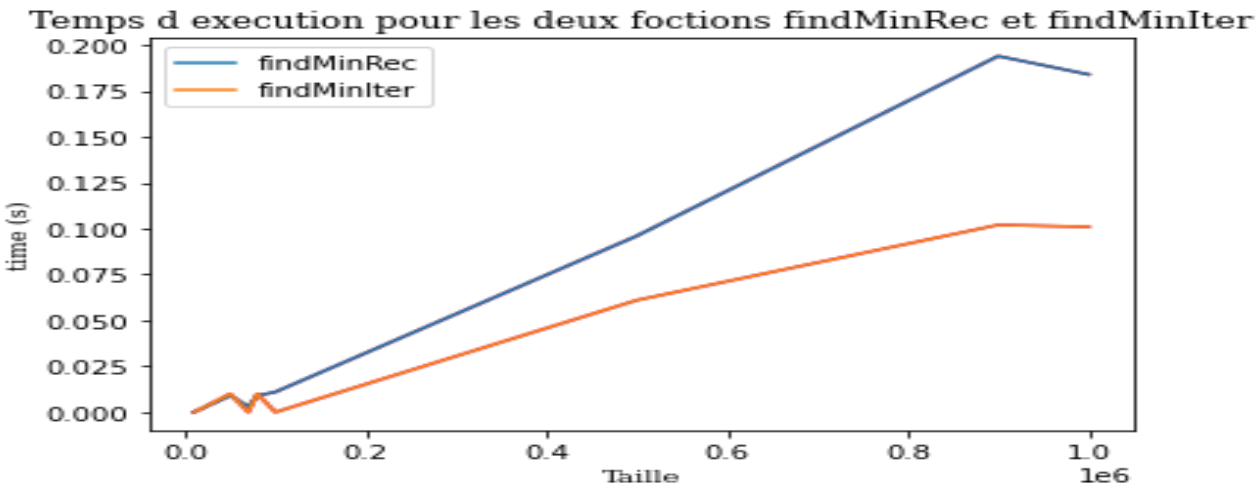


Figure 2: Représentation graphique comparative des résultats d'exécution de la fonction findMin

Analyse et comparaison :

On remarque à partir de **Figure 2** que les deux courbes sont presque les même pour des petites tailles, mais quand la taille augmente la courbe récursive passe plus rapide que celle de itérative. Malgré que les deux ont une complexité quadratique mais la version itérative est plus performante que la version récursive.

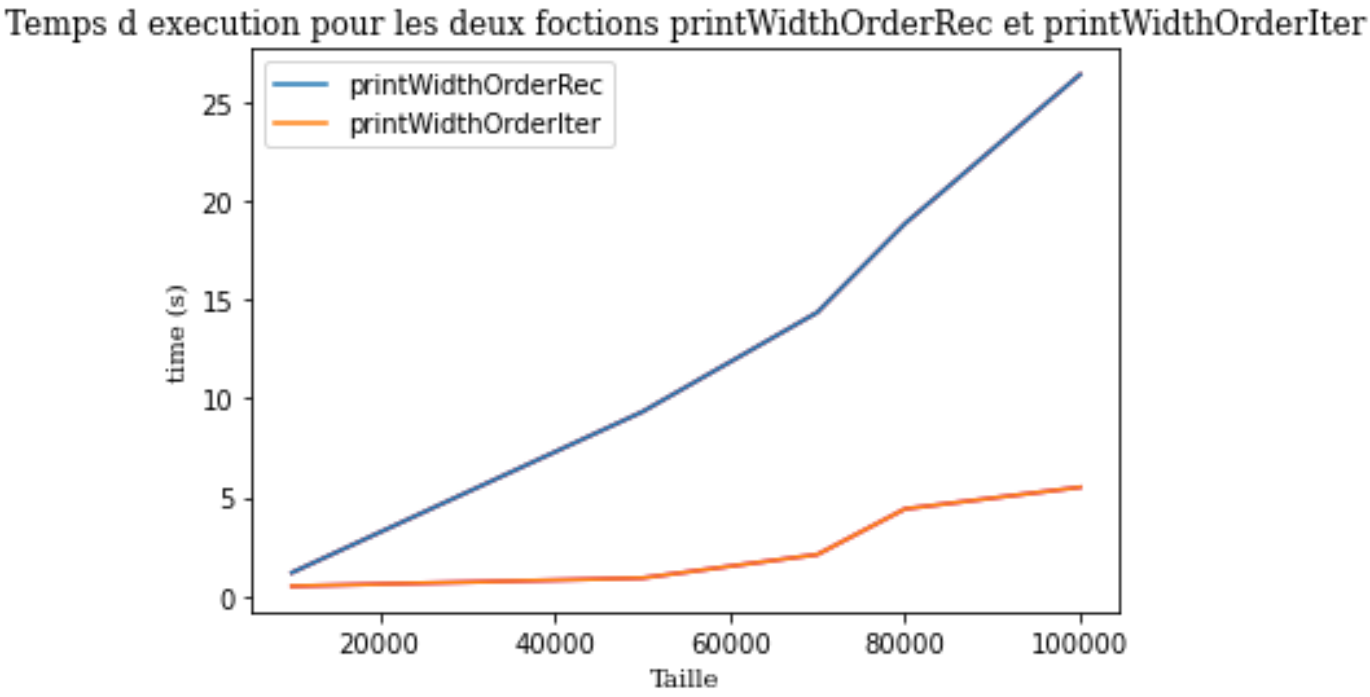


Figure 3 : Représentation graphique comparative des résultats d'exécution de la fonction printWidth

Analyse et comparaison :

On remarque à partir de **Figure 3** que la courbe de la complexité récursive est augmenté d'une manière très rapide (courbe quadratique) par contre la courbe de la complexité itérative accroit très légèrement (courbe linéaire).

La meilleure version pour ce parcours est la version itérative.

Conclusion générale

Un programme est dit **récursif** lorsqu'une entité **s'appelle elle-même**. Un programme est appelé **itératif** lorsqu'il y a une **boucle** (ou répétition).

Complexité du temps : Trouver la complexité temporelle de la récursion est plus difficile que celle de l'itération.

- **Récusion** : La complexité temporelle de la récursivité peut être trouvée en trouvant la valeur du n-ième appel récursif par rapport aux appels précédents. Ainsi, trouver le cas de destination en termes de cas de base, et le résoudre en termes de cas de base nous donne une idée de la complexité temporelle des équations récursives.
- **Itération** : La complexité temporelle de l'itération peut être trouvée en trouvant le nombre de cycles qui se répètent à l'intérieur de la boucle.

L'utilisation de l'une ou l'autre de ces techniques est un compromis entre la complexité temporelle et la taille du code. Si la complexité temporelle est le point central, et que le nombre d'appels récursifs est important, il est préférable d'utiliser l'itération. Cependant, si la complexité temporelle n'est pas un problème et que la taille du code l'est, la récursivité serait la solution.

Récursion : La récursivité consiste à appeler la même fonction à nouveau, et donc, a une très petite longueur de code. Cependant, comme nous l'avons vu dans l'analyse, la complexité temporelle de la récursion peut devenir exponentielle lorsqu'il y a un nombre considérable d'appels récursifs. Par conséquent, l'utilisation de la récursion est avantageuse dans le cas d'un code plus court, mais d'une complexité temporelle plus élevée.

L'itération est la répétition d'un bloc de code. Cela implique une plus grande taille de code, mais la complexité temporelle est généralement moindre que pour la récursivité.

Description de code utilise pour la représentation graphique

Pour réaliser les graphes de temps d'exécution de chaque algorithme on a eu recours au langage python à travers la bibliothèque **matplotlib** conçue spécialement pour dresser des courbes de nature différentes.

Le script trace les graphes en se basant sur une liste de nombre qui représente les temps d'exécution récupères a partir des fichiers génères par le programme c.

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as nb

xpoints = nb.array([10000,50000,70000,80000,100000,500000,900000,1000000])
ypoints1 = nb.array([0.00,0.009,0.003,0.009,0.011,0.096,0.194,0.184])
ypoints2 = nb.array([0.00,0.01,0.00,0.01,0.00,0.061,0.102,0.101])

font1 = {'family':'serif', 'color':'black', 'size':12}
font2 = {'family':'serif', 'color':'black', 'size':10}


plt.title('Temps d execution pour les deux foctions findMinRec et findMinIter', fontdict=font1)
plt.xlabel('Taille', fontdict=font2)
plt.ylabel('time (s)', fontdict=font2)
plt.plot(xpoints,ypoints1,'r')
plt.plot(xpoints,ypoints2,'m')
plt.plot(xpoints, ypoints1, label='findMinRec')
plt.plot(xpoints, ypoints2, label='findMinIter')
plt.legend()
#plt.grid(True)
plt.show()
```

Figure 4: Capture d'écran de code python pour le dessin des graphes.

Quelque exemple d'exécution

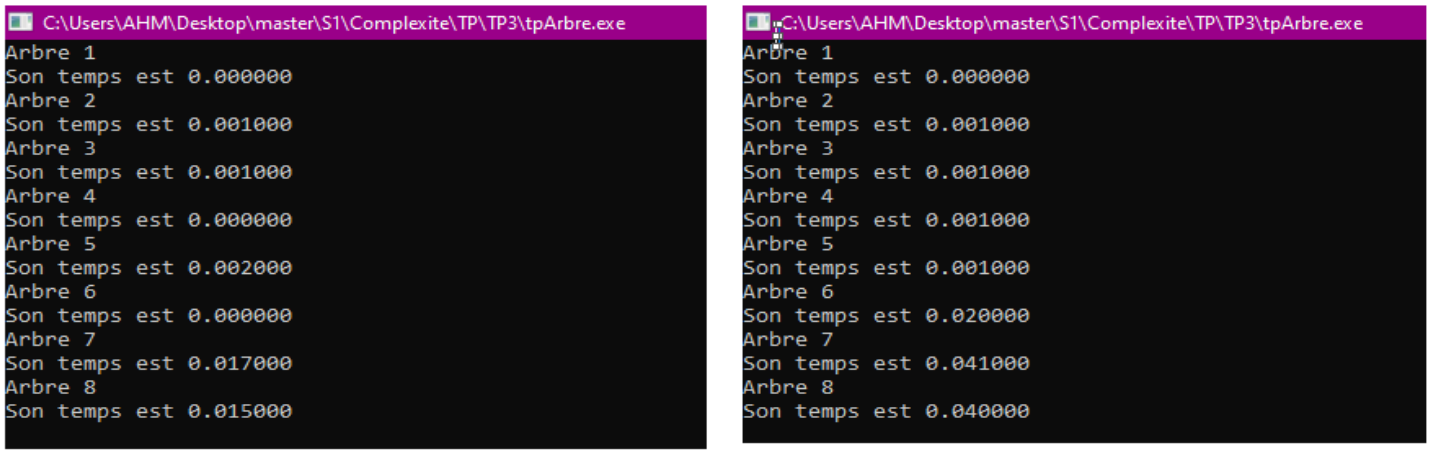


Figure 5 : Capture d'écran de l'exécution

Environnement expérimentale

Marque	Système d'exploitation	processeur	RAM
DELL i5	**Système d'exploitation 64 bits, processeur x64. Windows 10 Professionnel.**	**Intel(R) Core(TM) i5-7300U CPU @ 2.60GHz 2.70 GHz**	**8,00 Go**

Tableau 2 : Description de l'environnement expérimental.