République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté et d'Informatique Département Informatique

Projet pluridisciplinaire

Spécialité :
2eme année Ingenieur en informatique

Thème

ALGORITHMIQUE ET COMPLEXITE " La tour de Hanoï ".

Sujet Proposé par :
Dr. Hadjer MOULAI

Présenté par :

CHEMLI Ayoub
GUENANE Abdelkarim
CHELALI Mohamed Said Nail
DENADANI Safwane Souhil
BELKADI Adam
BOUCHAMA Imaddedine

Remerciement

Au moment où s'achève la réalisation de notre projet et la rédaction de ce Projet pluridisciplinaire, il nous est agréable de se retourner avec gratitude vers ceux qui nous ont apporté leur indispensable assistance dans notre travail. Mais avant tout, on remercie Dieu qui nous a donné le courage nécessaire pour mener à bien ce projet.

Nous tenons aussi à exprimer nos vifs remerciements A notre promotrice madame MOULAI Hadjer pour Son engagement et ses précieux conseils durant la période de notre travail.

Nous remercions les membres du jury pour avoir accepté d'examiner et de juger ce modeste travail.

A tous nos enseignants pour leurs efforts dans notre deux premieres annees de formation

Enfin, nous tenons à remercier toute personne qui a participé de près ou de loin à la réalisation de ce travail.

Tables des matières

Table des matières

Tour de hanoi

Introduction

La Tour de Hanoï est un problème classique en informatique qui consiste à déplacer une pile de disques d'une tour de départ à une tour d'arrivée en utilisant une tour auxiliaire, tout en respectant deux règles simples : un seul disque peut être déplacé à la fois, et aucun disque ne peut être placé sur un disque plus petit que lui-même. Bien que ce jeu puisse sembler simple, il met en lumière des concepts fondamentaux de l'informatique tels que la récursivité et la gestion de données.

Dans ce rapport, nous plongerons dans le monde fascinant de la Tour de Hanoï. Tout d'abord, nous expliquerons en détail les règles de ce jeu et son importance en informatique. Ensuite, nous explorerons différentes stratégies pour résoudre ce problème, en nous concentrant sur leur efficacité et leur simplicité. Nous analyserons également les implications de ces stratégies en termes de complexité algorithmique.

Après avoir discuté des concepts théoriques, nous passerons à la pratique. Nous explorerons à la fois des approches récursives et itératives, en expliquant le fonctionnement de chaque algorithme étape par étape. Nous testerons ensuite nos implémentations pour nous assurer qu'elles produisent les résultats attendus et que leur performance est satisfaisante.

En combinant une analyse approfondie avec une mise en pratique concrète, ce rapport vise à fournir une compréhension approfondie de la Tour de Hanoï, de ses applications en informatique et des principes algorithmiques fondamentaux qu'elle illustre.

I. Étude théorique du Problème

Historique :

Tour de Hanoï, un puzzle mathématique qui a captivé l'attention depuis sa création. Tout commence avec **François Edouard Anatole Lucas**, né le 4 avril 1842 à Amiens, France, et travaillant plus tard à Paris en tant qu'éminent théoricien des nombres. Lucas a consacré une partie de sa vie à la publication de ses **"Récréations mathématiques"**, dont le quatrième volume contient un puzzle mécanique appelé "Sigillum Salomonis", discuté dans le premier volume sous le nom de **"baguenaudier"**.

Ce puzzle antique semble avoir été le catalyseur de la création de la Tour de Hanoï. Cette dernière a été officiellement introduite dans un contexte mathématique en novembre **1883** par **Lucas** lui-même. Il l'a décrite comme **un jeu de combinaison** pour appliquer le système de numération binaire. L'inscription originale de Lucas, présente sur la boîte contenant le jeu, est conservée au Musée des Arts et Métiers à Paris, offrant ainsi une trace historique de ses débuts.

Figure 1 The original Tower of Hanoi © Musee des arts et metiers-Cnam Paris / photo M. Favareille

Dès sa création, l'idée de la Tour de Hanoï a été rapidement diffusée à travers le monde, avec des brevets accordés aux États-Unis et au Royaume-Uni dans les années qui ont suivi. En 1888, Lucas a généreusement donné le puzzle original au Conservatoire national des arts et métiers à Paris, marquant ainsi un moment crucial dans son histoire.

La couverture de la boîte originale du puzzle (voir **Figure 2**), illustrée d'un paysage fantastique, ainsi que les détails intrigants tels que le tatouage sur le ventre et la présence d'une grue tenant une feuille de papier, ajoutent une dimension artistique et culturelle à cette histoire.

Figure 2 La plaque de couverture de la Tour de Hanoï.

Mais pourquoi le choix du nom "La Tour de Hanoï" ? le jeu n'ait pas de lien direct avec la ville de Hanoï, le choix du nom pourrait être attribué à l'intérêt et à la présence française au Vietnam à cette époque, notamment à Hanoï qui était un sujet d'actualité dans les journaux français.

Lucas a créé une légende autour du jeu, racontant l'histoire de brahmanes dans le grand temple de Bénarès qui déplaçaient 64 disques d'or sur trois aiguilles de diamant, une tâche qui, si elle était réalisée selon certaines règles strictes, signalerait la fin du monde. Cette histoire fictive a été conçue pour ajouter un élément mystique et exotique au puzzle, qui ne comportait en réalité que huit disques en bois dans sa version originale.

La légende et le nom peuvent donc être vus comme un reflet de l'imaginaire colonial de l'époque, où des lieux exotiques comme Hanoï captivaient l'attention et l'intérêt du public français. Cela a probablement contribué à la popularité du jeu et à son nom mémorable.


LA TOUR D'HANOI

VÉRITABLE CASSE-TÊTE ANNAMITE

JEU RAPPORTÉ DU TONKIN

PAR LE PROFESSEUR N. CLAUS (DE SIAM)

PARIS, PÉKIN, YÉDO et SAÏGON

Règles et pratique du Jeu de la TOUR D'HANOI

Figure 3 Recto et verso du dépliant accompagnant le puzzle de la Tour de Hanoï

Ainsi, l'histoire de la Tour de Hanoï est riche en détails historiques, mathématiques et culturels, offrant un aperçu fascinant de son évolution à travers le temps et les continents.

Définition Formelle du Problème :

Le problème classique de la Tour de Hanoï se compose de trois piquets verticaux, nommés A, B et C, et de (n) (≥1) disques de différentes tailles.

Au départ, tous les disques sont placés sur le piquet source, empilés du plus grand au plus petit, formant ainsi une tour, avec le plus grand disque à la base et le plus petit en haut.

L'objectif est de transférer la tour du piquet source vers le piquet de destination en effectuant des mouvements autorisés.

Un mouvement autorisé est défini comme l'action de déplacer le disque du haut d'un piquet vers le haut d'un autre piquet sans violer les règles suivantes :

- **Chaque fois, un seul disque est déplacé** ;
- **Seul le disque supérieur peut être déplacé** ;
- **À aucun moment, un disque ne peut résider sur un disque plus petit**

La figure 4 illustre les états initial et final d'un problème de la Tour de Hanoï avec trois disques. Il y a trois disques, D1, D2 et D3, de taille croissante.

Initialement, tous les disques se trouvent sur le piquet A, le piquet source, tandis que le piquet cible est le piquet C.

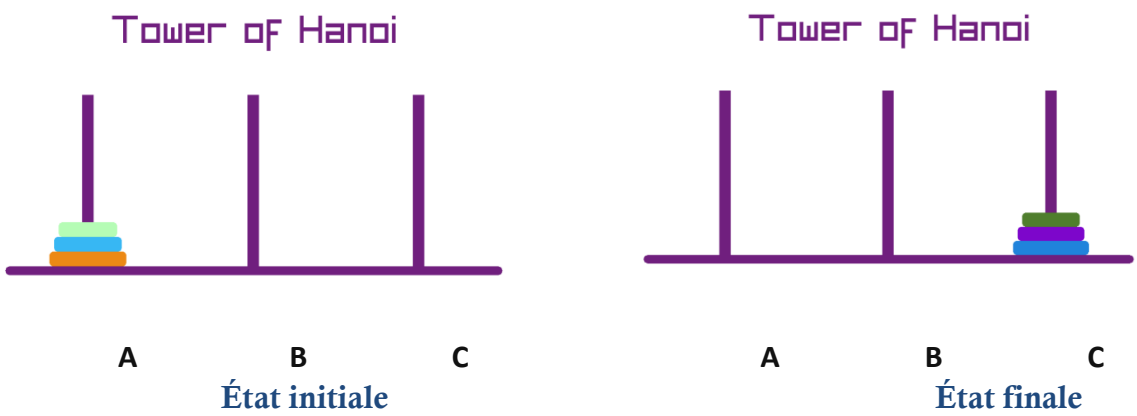


Figure 4 la Tour de Hanoï à trois piliers.

Conception des structures de données :

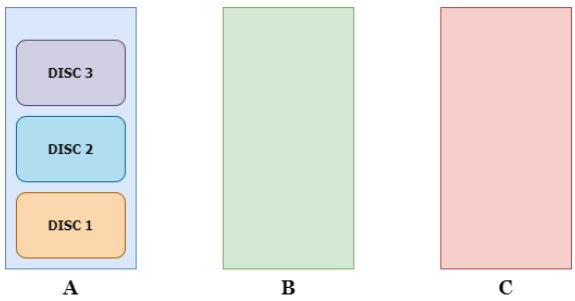
Le problème de la Tour de Hanoï peut être modélisé efficacement à l'aide de structures de données simples et élégantes.

Dans notre implémentation, nous utilisons des **piles** pour représenter les piquets et des **entiers** pour représenter les disques.

Piles (Piquets) : Chaque piquet est représenté par une pile, une structure de données linéaire suivant le principe LIFO (Last In, First Out). Cela signifie que seul l'élément situé au sommet de la pile peut être retiré ou ajouté à tout moment, ce qui correspond parfaitement à la règle du jeu qui stipule que seul le disque supérieur d'un piquet peut être déplacé.

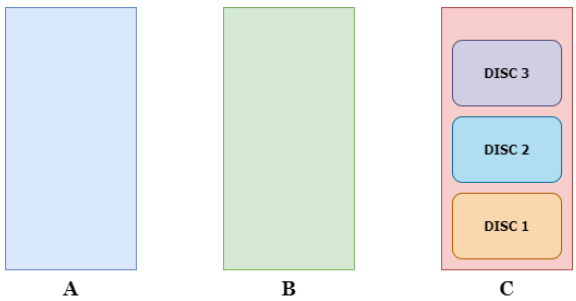
Entiers (Disques) : Les disques sont représentés par des entiers, où un entier plus petit correspond à un disque de diamètre plus petit. Cette représentation permet de comparer facilement la taille des disques lors des déplacements, assurant ainsi le respect de la règle qui interdit de placer un disque sur un autre plus petit.

État initiale de la pile :



Etat initiale

État finale de la pile :



Etat finale

Résolution du problème :

1. Solution générale :

a) Cas d'un seul disque :

- Déplacez simplement le disque de piquet de départ à piquet d'arrivée.

b) Cas de Plusieurs disques :

- Déplacez les n-1 disques supérieurs sur le piquet intermédiaire, en laissant le plus grand disque sur le piquet de départ.
- Déplacez ensuite le plus grand disque sur le piquet d'arrivée.
- Enfin, déplacez les n-1 disques du piquet intermédiaire vers le piquet d'arrivée, en les empilant sur le plus grand disque.

c) Nombre du coups optimal :

Nous avons tout d'abord tenté de résoudre le jeu avec un petit nombre n de disques.

Pour $n = 1$, il n'y a évidemment qu'un seul coup à jouer.

Pour $n = 2$, on peut résoudre le jeu en 3 coups.

Pour $n = 3$, on peut résoudre le jeu en 7 coups.

Démontrons par récurrence que le nombre de coups minimal pour résoudre le jeu avec n disques est égal à

$$\underline{2^n - 1}$$

Initialisation :

Le nombre de déplacements pour un anneau est égal à 1 et $2^1 - 1 = 2 - 1 = 1$ donc la récurrence est fondée au rang 1.

Hérédité :

Supposons que pour un certain entier n supérieur ou égal à 1, le nombre minimal de coups à jouer pour déplacer les n disques d'un piquet à l'autre soit égal à $2^n - 1$. Montrons alors que le nombre minimal de déplacements pour $n + 1$ disques est égal à $2^{n+1} - 1$. Avec $n + 1$ disques , on doit d'abord déplacer n disques du piquet de départ au piquet intermédiaire. Cela nécessite $2^n - 1$ déplacements. On peut alors déplacer le plus grand disque du piquet de départ au piquet d'arrivée ce qui nécessite 1 déplacement. Il faut ensuite redéplacer les n disques du piquet intermédiaire vers le piquet d'arrivée ce qui nécessite $2^n - 1$ déplacements.

Au final, le nombre de déplacement est égal à :

$$(2^n - 1) + 1 + (2^n - 1) = 2^n + 2^n - 1 = 2^{n+1} - 1.$$

On a démontré que la propriété est héréditaire.

Donc , quel que soit l'entier n supérieur ou égal à 1, le nombre minimal de déplacements pour résoudre le jeu est bien égal à $2^n - 1$.

2. Solution itérative :

a) Présentation de l'algorithme de Resolution :

```
ALGORITHME Resolution_Iteratif ;
 Var
 A, B, C : Pile ;
 n, indic, deplacementMin, i : entier ;
 DÉBUT
 // Déterminer la parité de n
 Si ( n MOD 2 = 0) Alors
      indic ← 1 ; // n est pair
 Sinon
      indic ← -1 ; // n est impair
 Fsi ;
 // Calculer le nombre de déplacements minimum
 deplacementMin ← puissance(2,n) – 1 ;
 // Boucle de déplacement des disques
 Pour i de 1 À deplacementMin Faire
      // Cas où n est pair
      Si (indic = 1) Alors
           Si (i MOD 3 = 0) Alors
                // B1
                Si ( Non vide(A) Et (vide(B) Ou sommet(A) < sommet(B)) ) Alors
```

```
                empiler(B, depiler(A)) ;
            Sinon
                empiler(A, depiler(B)) ;
             Fsi ;
        Sinon Si ( i MOD 3 = 1) Alors
            // B2
             Si (Non vide(A) Et (vide(C) Ou sommet(A) < sommet(C))) Alors
                empiler(C, depiler(A)) ;
            Sinon
                empiler(A, depiler(C)) ;
             Fsi ;
         Sinon
            // B2''
            Si (Non vide(B) Et (vide(C) Ou sommet(B) < sommet(C))) Alors
                empiler(C, depiler(B)) ;
            Sinon
                empiler(B, depiler(C)) ;
             Fsi;
        Fsi ;
// Cas où n est impair
Sinon
        Si (i MOD 3 = 0) Alors
             Si ( Non vide(A) Et (vide(C) Ou sommet(A) < sommet(C)) )Alors
                empiler(C, depiler(A)) ;
            Sinon
```

```
                    empiler(A, depiler(C)) ;
                Fsi ;
            Sinon Si ( i MOD 3 = 1) Alors
                Si (Non vide(A) ET (vide(B) Ou sommet(A) < sommet(B)))Alors
                    empiler(B, depiler(A)) ;
                Sinon
                    empiler(A, depiler(B)) ;
                Fsi ;
            Sinon
                Si (Non vide(C) ET (vide(B) OU sommet(C) < sommet(B))) Alors
                    empiler(B, depiler(C)) ;
                Sinon
                    empiler(C, depiler(B)) ;
                Fsi ;
            Fsi ;
        Fsi ;
        // Afficher le numéro de déplacement et l'état des piquets
        Ecrire("déplacement numéro : ", i ) ;
        afficherPiquets(A, B, C)  ;
        Fait ;  // fin pour
    FIN  .
```

b) Calcul de la complexité

```
/*Calcul de la complexité théorique de la fonction résolution itératif*/
/*
 int indic = 0 ;  -> 1 op (affectation)
 ----------------------------------------
  if (n % 2 == 0) -> 2 op ( opération du mod + comparaison du résultat avec le 0)
   {
        indic = 1;  -> 1 op (affectation)
    } else {
        indic = -1;
    }
    les deux blocs sont équivalents dans le nombre d'instructions , donc on
choisit l'un des deux
    => 3 op dans le total
----------------------------------------
 int deplacementMin = (1<<n)-1; -> n+2 op (1<<n ( n opération de décalage de bits
+ l'inititalisation à 1 ) + 1<<n -1 (soustraction))
 ----------------------------------------
 le calcul du complexité de la boucle for (int i = 0; i < deplacementMin; i++) :


 1. le calcul de nombre d'itérations
 nb d'itérations = ( deplacementMin -1 - 0 +1 )/1 = deplacementMin = (2^n) - 1


 2.le calcul du nombre de comparaisons
 int i = 0 -> 1op
 2^n -1 comparaison + la comparaison de la sortie de la boucle pour => 2^n
comparaison
 i <- i+ 1 ; 2op (addition + affectation) => le total égale à 2*((2^n)-1) =
2^(n+1) - 2


 3.le calcul de la complexité du plus grand bloc if else (if indic == 1)
 on remarque que les deux blocs sont équivalents dans le nombre d'instructions
donc choisit l'un des deux blocs
 => calcul de la complexité du bloc if


 if (indic == 1 ) => 1op (comparaison)
 on a 3 blocs , le bloc de if (i%3==0) , aprés le bloc else if (i%3==1),aprés le
bloc else (i%3==2),aprés le bloc else

 donc la complexité du bloc if (indic ==1) égale à la complexité de la condition
if(i%3==0) + le max (B1,B2)=B2
 dans B2 on choisit l'un des deux blocs B2' ou B2'' => B2'
```

```
---> calcul du complexité du Bloc B2'
 ---> calcul de la complexité du bloc if(i%3==1)
 if (i%3==1) -> 2op (le mod + la comparaison avec le un )
 on a deux blocs , le bloc if et le bloc else , les deux blocs ont le même nombre
d'instructions
 donc on choisit l'un des deux blocs => bloc if

  if (!vide(p1) && (vide(p3) || sommet(p1) < sommet(p3))) -->

 1....sommet (p1) -> 1 op (on accede au sommet du pile p1 et on le retourne)

 2....sommet (p3) -> 1 op (on accede au sommet du pile p3 et on le retourne)

 3.... sommet (p1) < sommet(p3) -> 1op (comparaison)

 4.... (vide (p3) <=> vide (p3)== 1) -> 3op (comparaison du sommet du pile avec le
- 1 + le retour de la fonction + comparaison avec 1)

 5....  4||3 -> 1 op (opération du ou)

 6....(!vide(p3)<=>vide(p3)!=1) -> 3op (comparaison du sommet du pile avec le - 1
+ le retour de la fonction + la comparaison avec 1)

 7.... 6&&5 -> 1op (opération du et )


  empiler(p3, depiler(p1)) ->
  dépiler(p1)-> 2op (accés a la pile et décrementation du sommet)
  empiler(p3,dépiler(p1))-> 2op (accés a la pile et incrémentation du sommet +
affectation du résultat de dépilement du p2)

-------------------------------------------------------------------------------
- le total = 1 + 3 + n+2 + 1 + 2^n + (2^(n+1))-2  + ((2^n)-
1)*(1+2+2+1+1+1+3+1+3+1+2+2) = 5+n+(2^n)+2^(n+1)+20*(2^n)-20 = 23*2^n+n-15
- comme 2^n > n quelque soit n>=0 donc on garde seulement le plus grand
coefficient qui est 2^n ce qui implique que la complexité égale à
O(2^n)
-------------------------------------------------------------------------------
```

c) Présentation de l'algorithme de verification :

ALGORITHME Verif_iterative ;

Var

A, B, C : **Pile** ;

DEBUT

Si (vide(A**) ET vide(**B**) ET pileDecroissante(**C**)) Alors**

Retourner 1 ;

Sinon

Retourner -1 ;

Fsi ;

Fin .

d) Calcul de la complexité theorique :

```
/*******Complexité théorique de la fonction verif_iterative***********/

/*Calculons la complexité du bloc if else
if (vide(&A)&&vide(&B)&&pileDecroissante(&C))
{
return 1;}
->
1.... (vide (&A) <=> vide (&A)== 1) -> 3op (comparaison du sommet du pile avec le
- 1 +  retour de fonction  + comparaison avec 1  )

2.... (vide(&B) <=> vide (&B)== 1) -> 3op (comparaison du sommet du pile avec le -
1 + retour de fonction  + comparaison avec 1  )

3.... deux opérations du et -> 2op


4....calcul de la complexité de la fonction (pileDecroissante(&C))->

---int x = depiler(p) -> 3op ( décrementation du sommet + retrun + affectation )
```

```
---while (p->sommet!=-1) <=> while (!vide(p)) donc on va  dépiler tous les n-1
éléments (dans le pire cas )de la pile ce qui implique que le nombre d'itérations
égale à n-1

-> calcul de la complexité de la boucle while

n vérification de (p->sommet!=-1)
int next = dépiler(p) -> 2op * (n-1) de dépilement + n-1 affectation -> 3(n-1)
le pire cas c'est que chaque fois le next > x donc :
la vérification de next < x -> (1op *n-1)
x = next -> 1op * n-1


return 1 -> 1op

=> le total = 3+n+3(n-1)+n-1+n-1+1 = 6n-1

5. return 1 ; -> 1op
donc la complexité du bloc if égale a  : 3+3+2+6n-1+1 =6n+8

d'où la complexité total de la fonction = 6n+8 => O(n)
```

e) Calcul de la complexité spatial :

```
/*****Complexité spatial*****/
//1.vérification
/**Comme on vérifie que la pile A et B sont vides ainsi on vérifie que la pile C
est remplie dans l'ordre décroissant donc on réserve une espace mémoire
pour les 3 piles de taille n (nombre de disques)+ la taille réservé pour la
variable next et x (dans la fonction piledécroissante) ce qui implique que
complexité spatiale = 3n+2 => O(n)
donc on octets c'est (3n+2)*4octets car chaque entier est representé en 4octets*/

//2.exécution
/**On fait les déplacements entre les 3 piquets donc on réserve une espace mémoire
pour les 3 piles de taille n + la variable indic + la variable n +la variable i
+la variable déplacementMin =>
 3n+1+1+1+1 = 3n+4 => O(n) en octets c'est (3n+4)*4octets car un entier est
représenté en 4octets */
```

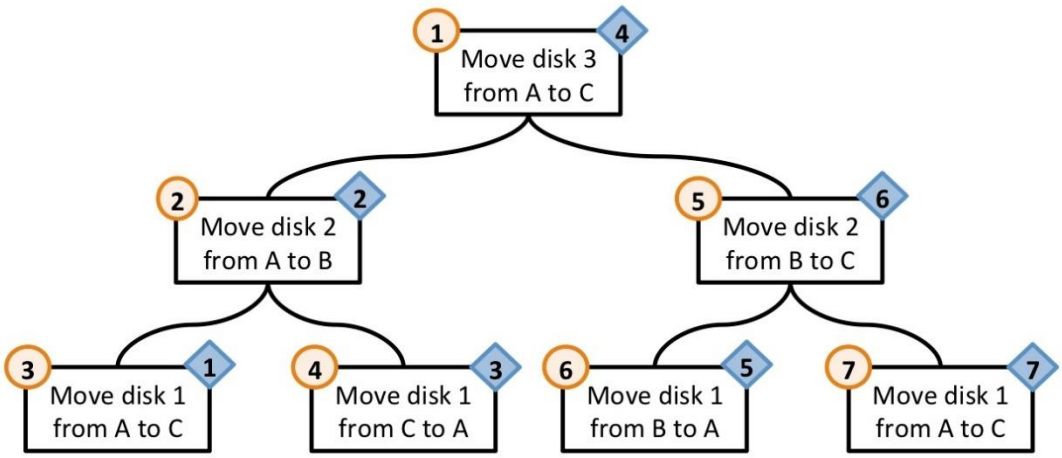
3. Solution récursive :

a) Présentation de l'algorithme :

```
ALGORITHME Hanoi
Var
    n : entier
    source, auxiliary, destination, finalDestination : caractère ;
DEBUT
    // Appeler la fonction récursive
    HanoiRecursif(n, source, auxiliary, destination, finalDestination) ;
FIN .

PROCEDURE HanoiRecursif (n,source, auxiliary, destination, finalDestination)
DEBUT
        Si( n = 1) Alors
            Ecrire("Déplacer le disque 1 de ", source, " vers ", destination) ;
        Sinon
            HanoiRecursif(n - 1, source, destination, auxiliary, finalDestination, pile) ;
            Ecrire("Déplacer le disque ", n, " de ", source, " vers ", destination)
            HanoiRecursif(n - 1, auxiliary, source, destination, finalDestination, pile) ;
        Fsi
 FIN ;
```

b) Calcul de la complexité :



- L'algorithme récursif pour résoudre les tours de Hanoï se décompose comme suit :

1. Déplacer les n-1 disques du poteau source au poteau intermédiaire.
2. Déplacer le disque restant du poteau source au poteau de destination.
3. Déplacer les n-1 disques du poteau intermédiaire au poteau de destination.

- Récurrence de la complexité temporelle :

La récurrence qui décrit la complexité temporelle (T(n)) de cet algorithme est :

$T(n) = 2T(n-1) + 1$

où :
- (T(n)) est le temps nécessaire pour déplacer n disques,
- (2T(n-1)) correspond au temps nécessaire pour déplacer les n-1 disques deux fois (une fois vers le poteau intermédiaire et une fois vers le poteau de destination),
- et (1) correspond au déplacement du plus grand disque directement au poteau de destination.

- Résolution de la récurrence :

1. Développer la récurrence :

$T(n) = 2T(n-1) + 1$
$T(n-1) = 2T(n-2) + 1$
$T(n-2) = 2T(n-3) + 1$

2. Substituer (T(n-1)) dans l'équation de (T(n)) :

T(n) = 2(2T(n-2) + 1) + 1

T(n) = 2^2.T(n-2) + 2 . 1 + 1

T(n) = 2^2.T(n-2) + 2 + 1

3. Continuer le processus pour (T(n-2)) :

T(n) = 2^2.(2T(n-3) + 1) + 2 + 1

T(n) = 2^3.T(n-3) + 2^2 + 2 + 1

4. Généraliser cette substitution jusqu'à atteindre (T(1)) :

T(n) = 2^k.T(n-k) + 2^{k-1} + 2^{k-2} + ... + 2 + 1

Lorsque k = n-1,

T(n) = 2^{n-1}T(1) + (2^{n-2} + 2^{n-3} + ... + 2 + 1)

5. Simplifier la somme géométrique :

T(n) = 2^{n-1} . 1 + (2^{n-1} - 1)

T(n) = 2^{n-1} + 2^{n-1} - 1

T(n) = 2 . 2^{n-1} - 1

T(n) = 2^n - 1

- Conclusion :

Ainsi, le nombre total de mouvements nécessaires pour déplacer n disques est (2^n - 1). La complexité temporelle de l'algorithme des tours de Hanoï est donc :

T(n) = 2^n - 1

b) Présentation de l'algorithme de verification :

Pour s'assurer que les disques sont correctement empilés dans l'ordre sur la destination finale, on doit ajouter quelque conditions dans l'algorithme de résolution .

PROCEDURE HanoiRecursif (n,source, auxiliary, destination, finalDestination)

DEBUT

Si(n = 1**) Alors**

Ecrire("Déplacer le disque 1 de ", source, " vers ", destination**) ;**

// Si la destination est la destination finale, empile le disque

Si (destination = finalDestination)alors

empiler(pile ,n) ;

// Si la source est la destination finale, dépile le disque

SinonSi (source == finalDestination)**Alors**

dépiler(pile) ;

Fsi ;

Retourner ;

Sinon

HanoiRecursif(n - 1, source, destination, auxiliary, finalDestination, pile**) ;**

Ecrire("Déplacer le disque **",** n, " de ", source, " vers ", destination**) ;**

// Si la destination est la destination finale, empile le disque

Si (destination == finalDestination)alors

empiler(pile ,n)

// Si la source est la destination finale, dépile le disque

Sinon Si (source == finalDestination)**Alors**
dépiler(pile)
Fsi ;

```
            HanoiRecursif(n - 1, auxiliary, source, destination, finalDestination, pile) ;
        Fsi ;
FIN ;

Fonction verification(n: entier, pile: Pile) : booléen
Var
        a , count : entier ;
DEBUT
count ← 0;
        Tantque (Non vide(pile) ) faire
                a ← dépiler(pile) ;
                count ← count + 1 ;
                Si ( non vide(pile) ET a > sommet(pile) ) Alors
                        Ecrire( "Solution invalide") ;
                        Retourner faux ;
                Fsi ;
        Fait ;
        Si (count != n )alors
                Ecrire( "Solution invalide") ;
                Retourner faux ;
        Fsi ;
        Ecrire( "Solution valide") ;
        Retourner vrai ;
Fin ;
```

c) Calcul de la complexité theorique :

```
/*******Complexité théorique de la fonction verif_recursive***********/
/*

1.... int count = 0 -> 1op

/*Calculons la complexité du bloc while

1.... (!vide(pile) <=> !vide(pile) == 1) -> 3op (comparaison du sommet du pile
avec le - 1 // return de la fonction //comparaison de la negation du retour de
fonction avec 1 )

2.... int a = pop(pile); -> 3op (decrementation du p->sommet// return de la
fonction // affectation du resultat de la fonction à a )

3.... count ++ -> 1op (incrementation du count)

4....Calcul de la complexité du bloc if :
    --(!vide(pile) <=> !vide(pile) == 1) -> 3op (comparaison du sommet du pile
avec le - 1 // return de la fonction // comparaison de la negation du retour de
fonction avec 1 )
    --a > sommet(pile) -> 2op (return de la fonction // comparaison du return avec
a )
    -- && -> 1op

    puisque au pire cas on ne realise jamais la condition <=> true alors on peut
negligé le "return false -> 1op"

    DONC total => 6op

while (!vide(p)) -> au pire cas on a  dépilé tous les n éléments de la pile ce qui
implique que le nombre d'itérations égale à n

DONC la complexité du bloc while égale a  : 13n


3.... (count != n) -> 1op (compraison)
4.... return de la fonction 1op



En conclusion la complexité total de la fonction = 1+13n+1+1 = 13n+3 => O(n)
```

d) Calcul de la complexité spatial :

```
/*****Complexité spatial*****/

/*Comme on vérifie  que la pile est remplie dans l'ordre décroissant donc on
réserve une espace mémoire
de taille n (nombre de disques)+ la taille réservé pour les variables n, count et
a. Ce qui implique que complexité spatiale = n+3 => O(n)
DONC en octets c'est (n+3)*4octets car chaque entier est representé en 4octets*/
```

Présentation d’une instance avec solution :

n = 3

1. État Initial (état 0) :

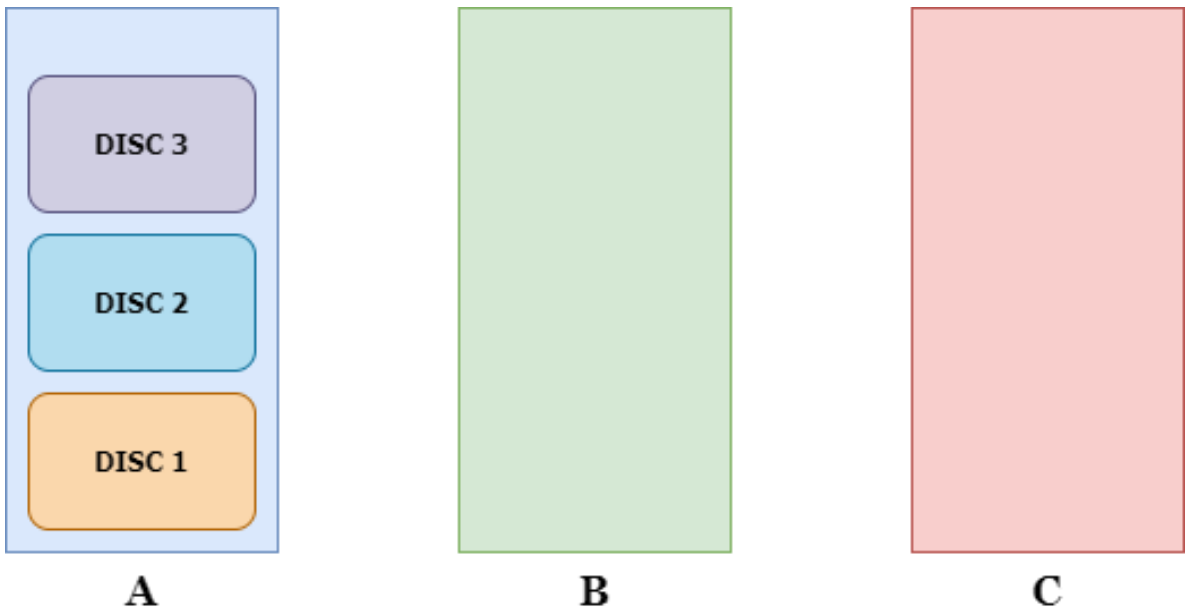
- Pile A (source) : Cette pile représente la tour d'origine où sont initialement empilés tous les disques.
Chaque disque est représenté par un nombre entier unique et est empilé du plus grand (en bas) au plus petit (en haut). Par exemple, si nous avons trois disques, la pile A pourrait ressembler à ceci :

1. Disque 1
2. Disque 2
3. Disque 3

- Pile B (auxiliaire) : Cette pile est initialement vide et est utilisée comme pile temporaire pour le déplacement des disques.

- Pile C (destination) : Cette pile est également initialement vide et représente la tour cible où nous voulons déplacer tous les disques à partir de la pile A.
Elle suit la même convention d'empilement que la pile A.

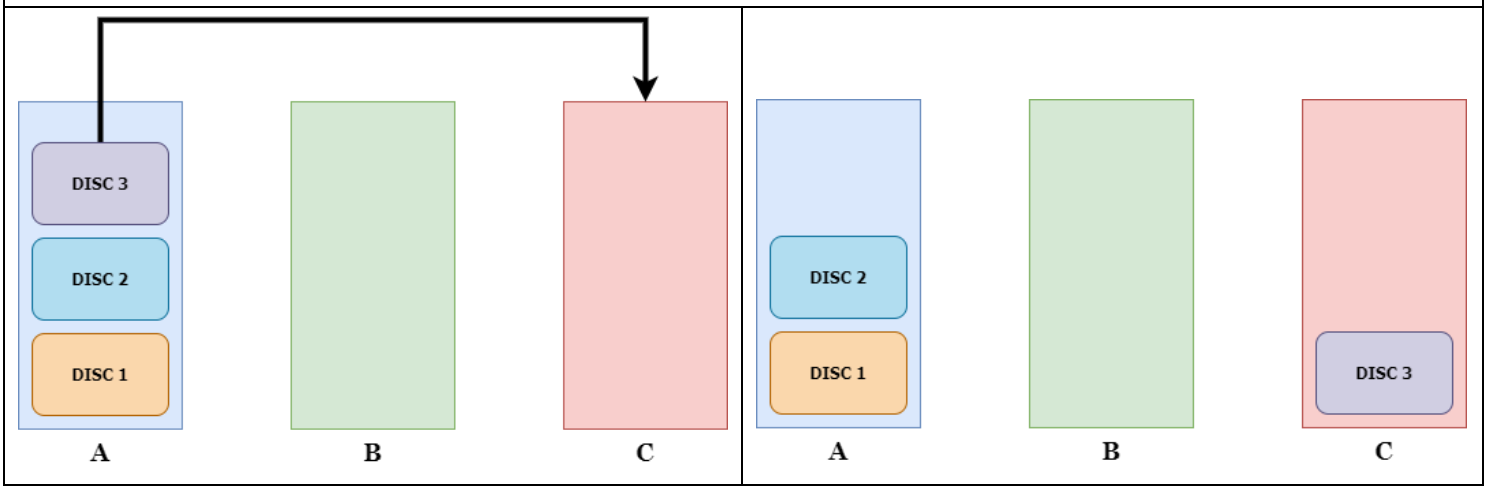
Dans cet état initial, tous les disques sont empilés sur la tour A, du plus grand (en bas) au plus petit (en haut), tandis que les piles B et C sont vides et prêtes à être utilisées comme piles auxiliaires pour déplacer les disques selon les règles du jeu.



2. Déroulement de la pile :

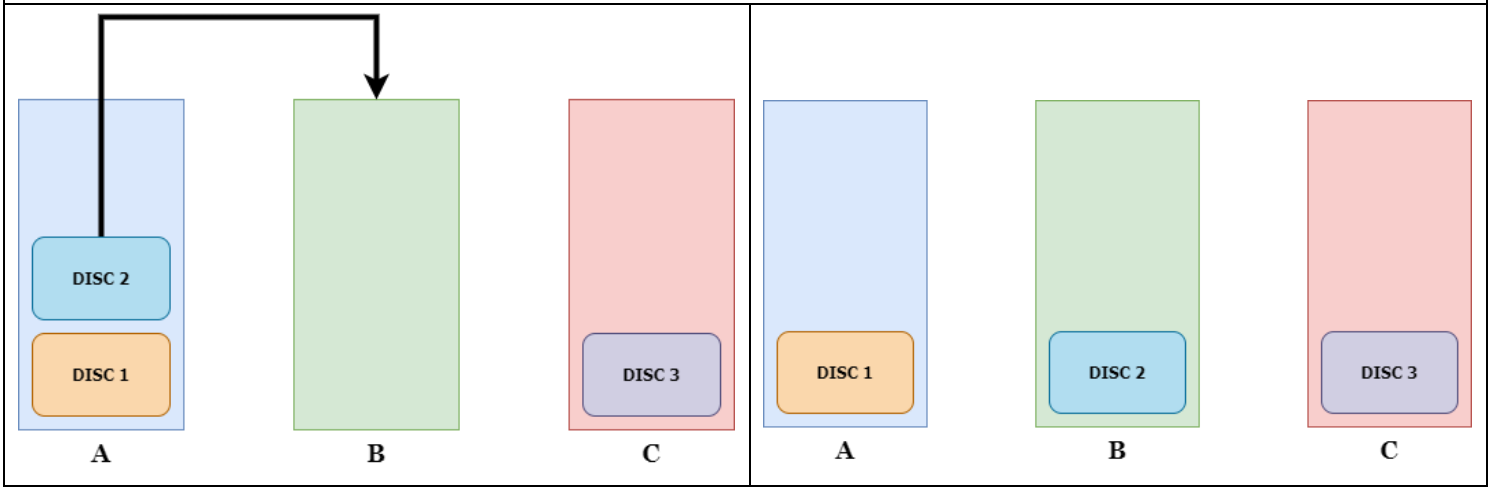
1. Déplacer le disque 3 de la tour A vers la tour C :

- **Pile A :** Les disques 1 et 2 restent sur la pile A.
- **Pile B :** Vide.
- **Pile C :** Contient uniquement le disque 3.



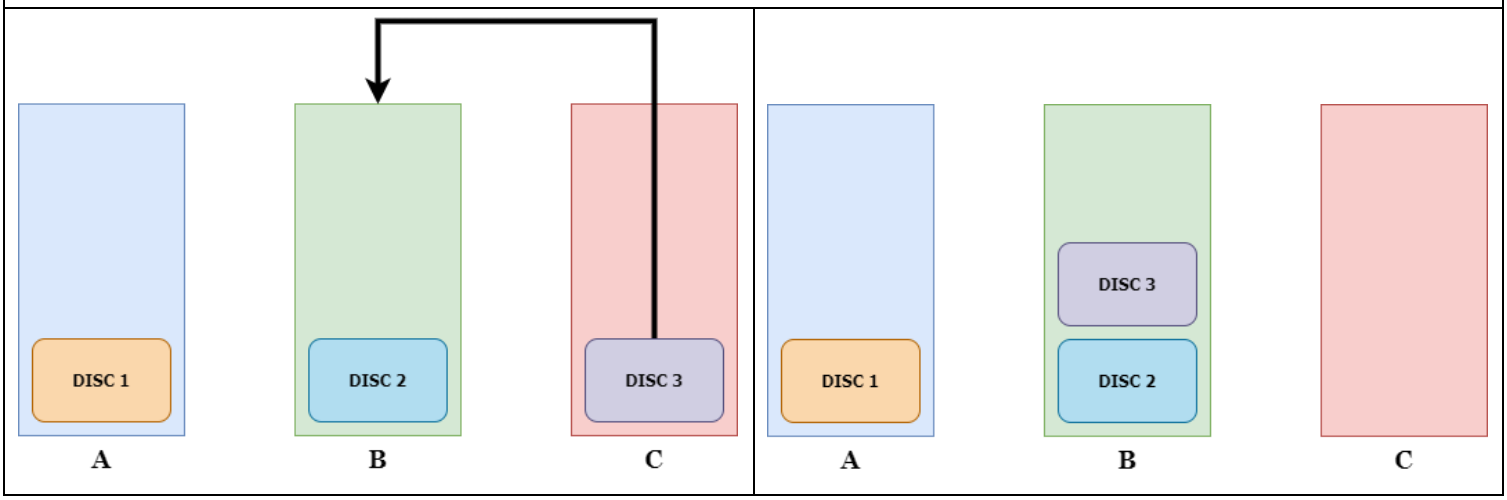
2. Déplacer le disque 2 de la tour A vers la tour B :

- **Pile A :** Le disque 1 reste sur la pile A.
- **Pile B :** Contient uniquement le disque 2.
- **Pile C :** Contient uniquement le disque 3.



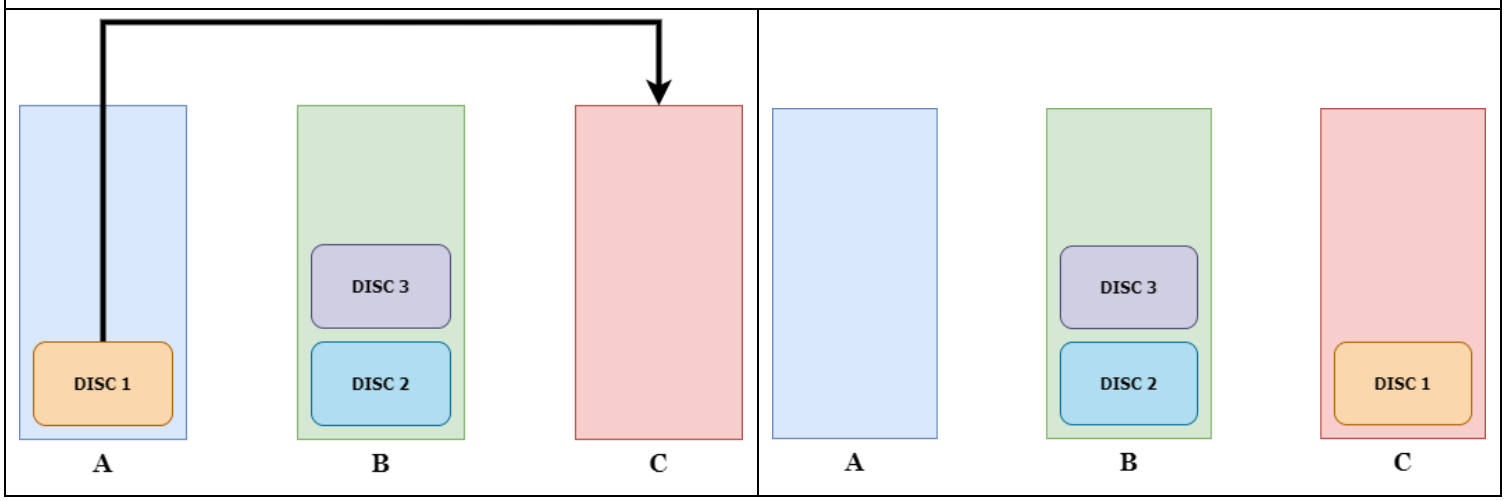
3. Déplacer le disque 3 de la tour C vers la tour B :

- **Pile A :** Le disque 1 reste sur la pile A.
- **Pile B :** Contient uniquement les disques 2 et 3.
- **Pile C :** vide.



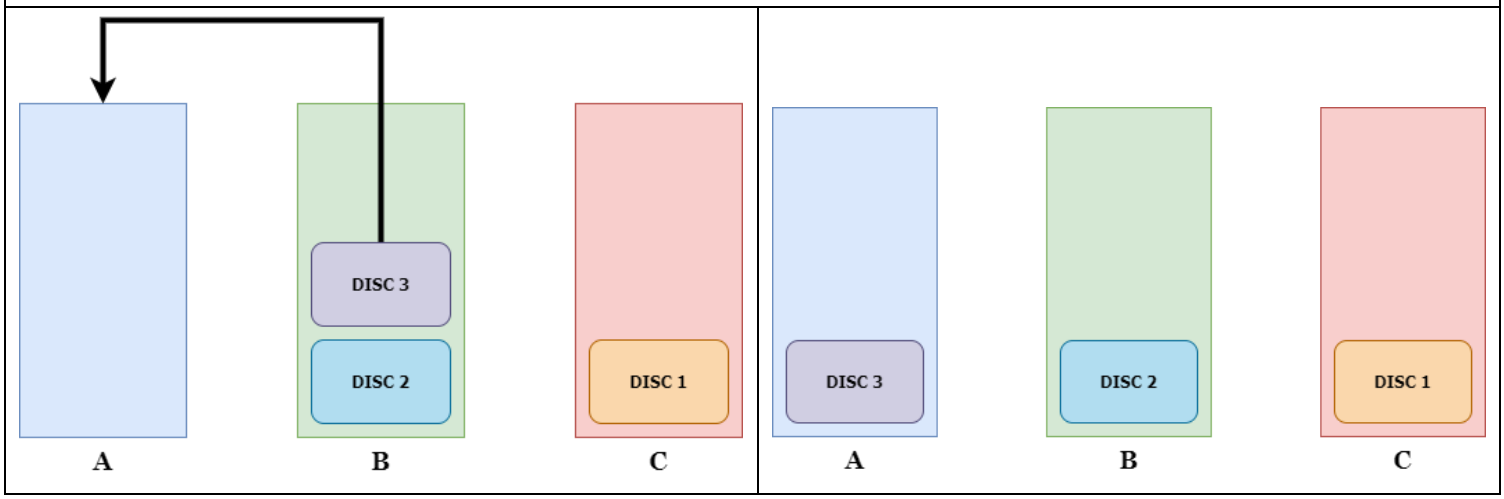
4. Déplacer le disque 1 de la tour A vers la tour C :

- **Pile A :** vide.
- **Pile B :** Contient uniquement les disque 2 et 3.
- **Pile C :** Contient uniquement le disque 1.



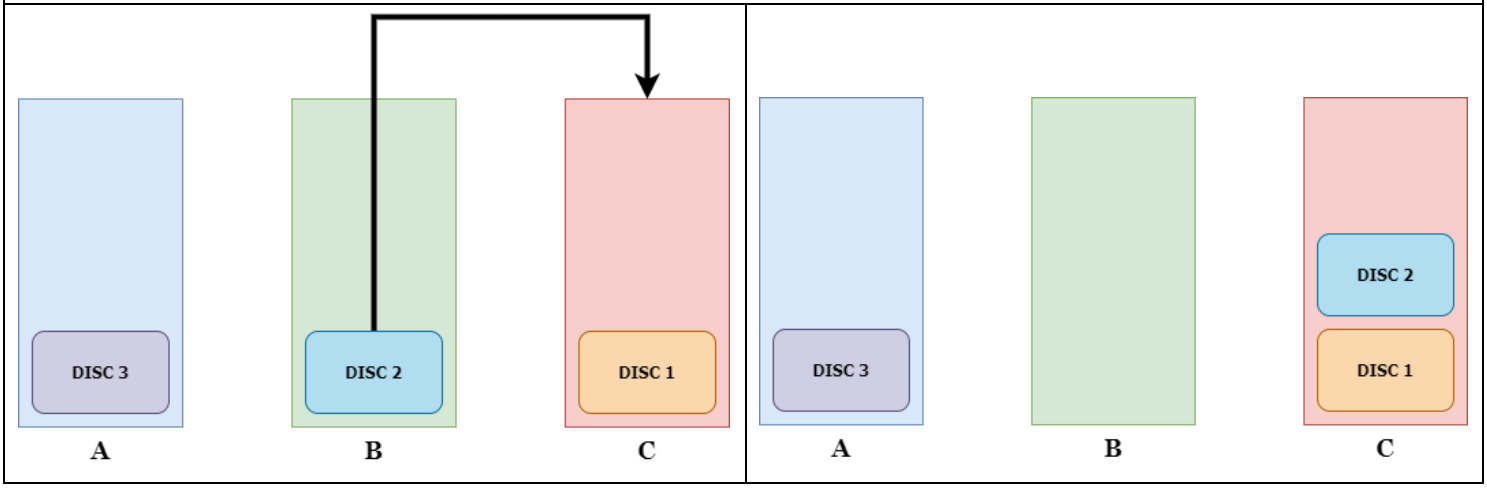
5. Déplacer le disque 3 de la tour B vers la tour A :

- **Pile A :** Contient uniquement le disque 3.
- **Pile B :** Contient uniquement le disque 2.
- **Pile C :** Contient uniquement le disque 1.



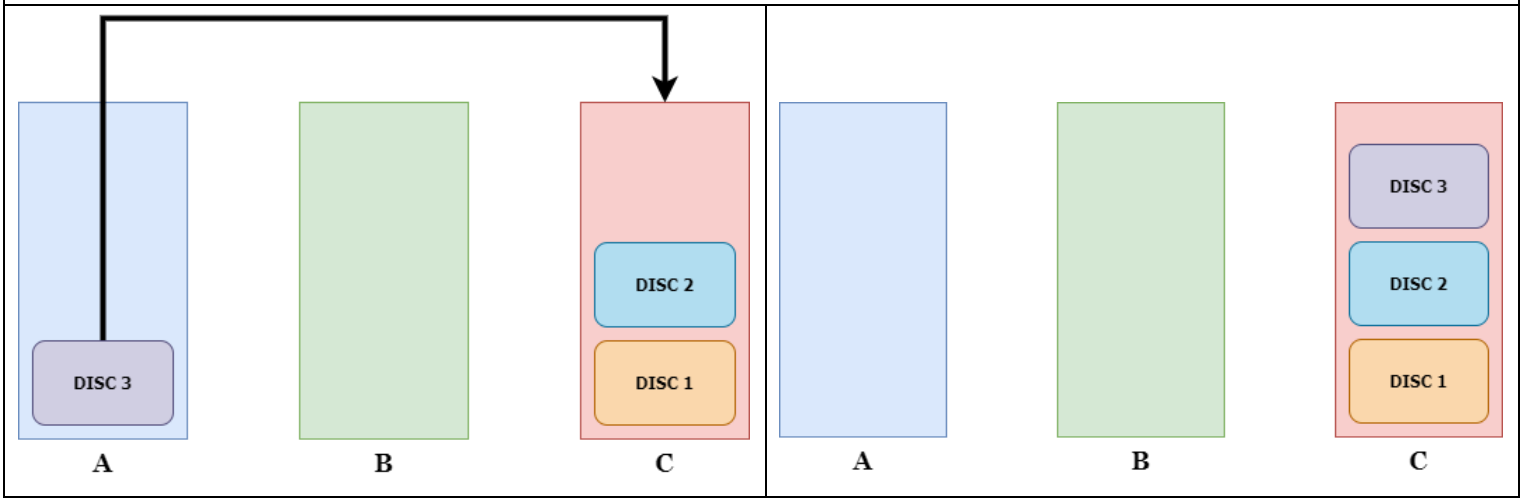
6. Déplacer le disque 2 de la tour B vers la tour C :

- **Pile A :** Contient uniquement le disque 3.
- **Pile B :** Vide.
- **Pile C :** Contient uniquement le disque 1 et 2.



7. Déplacer le disque 3 de la tour A vers la tour C :

- **Pile A :** vide.
- **Pile B :** Vide.
- **Pile C :** Contient tout les disques 1, 2 et 3.



II. Étude Expérimentale du Problème

Dans cette partie, on a simulé les différentes complexités (Temporelle et Spatiale) pour les algorithmes itératifs et récursifs de la résolution des Tours de Hanoï ainsi l'algorithme de vérification.

Itérative :

Nombre de disques (n)	**1**	**3**	**5**	**7**	**9**	**11**
Temps d'exécution (s)	0.000002	0.000002	0.000004	0.000004	0.000009	0.000049
Nombre de déplacements effectués	1	7	31	127	511	2047
Temps de vérification (nanoseconds)	109	170	217	260	262	314
Complexité spatiale de l'execution (octets)	7*4=28	52	76	100	124	148
Complexité spatiale de vérification(octets)	5*4 = 20	44	68	92	116	140

Nombre de disques (n)	**13**	**15**	**17**	**19**	**21**
Temps d'exécution(s)	0.000131	0.000466	0.001803	0.007017	0.0266846
Nombre de déplacements effectués	8191	32767	131071	524287	2097 151
Temps de vérification (nanoseconds)	329	384	456	515	573
Complexité spatiale de l'execution (octets)	172	196	220	244	268
Complexité spatiale de vérification(octets)	164	188	212	236	260

Nombre de disques (n)	23	25	27	29	31
Temps d'exécution(s)	0.1107017	0.706010	3.145218	11.386999	45.367207
Nombre de déplacements effectués	8388607	33554431	134217727	536870911	2147483647
Temps de vérification (nanoseconds)	624	637	758	784	814
Complexité spatiale de l'execution (octets)	292	316	340	364	388
Complexité spatiale de vérification(octets)	284	308	332	356	380

- Formule de calcul complexité spatiale de vérification = **(3n+2) *4octets**
- Formule de calcul complexité spatiale de l'exécution = **(3n+4)*4octets**
- Methode de simulation de la complexité de résolution itérative :

```
// Initialiser le compteur de temps au moment actuel
start = clock();

// Appeler la fonction resolutionIteratif
resolutionIteratif(&p1, &p2, &p3, n);

// Arrêter le compteur de temps
end = clock();

// Calculer le temps écoulé (en secondes)
float time = (float)(end - start) / CLOCKS_PER_SEC;

// Afficher le temps d'exécution
printf("\t\tLe temps d'execution est : %f\n\n", time);
```

Récursive :

Nombre de disques (n)	1	3	5	7	9	11
Temps d'exécution (sec)	200 (nanosec)	800 (nanosec)	900 (nanosec)	1700 (nanosec)	2800 (nanosec)	8200 (nanosec)
Nb déplacements	1	7	31	127	511	2047
Complexité spatiale (octets) O(2n-1)	11	27	43	59	75	91
Temps de vérification (nanoseconds)	2883	4564	8642	9755	11066	12316
Spatiale (octets)	16	24	32	40	48	56

Nombre de disques (n)	13	15	17	19	21
Temps d'exécution (sec)	25400 (nanosec)	0,0001845	0.001	0.004	0.007
Déplacements	8191	32767	131071	524287	2097151
Spatiale (octets) O(2n-1)	107	123	139	155	171
Temps d'exécution (NanoSeconds)	17324	20337	22933	24261	26246
Spatiale (octets)	64	72	80	88	96

Nombre de disques (n)	23	25	27	29	31
Temps d'exécution (sec)	0.029	0.09	0.307	1.21	5.344
Déplacements	8388607	33554431	134217727	536870911	2147483647
Spatiale (octets) O(2n-1)	187	203	219	235	251
Temps d'exécution (NanoSeconds)	28995	31604	23733	35371	36595
Spatiale (octets)	104	112	120	128	136

Différentes nuances de complexité:

Pour des données de même taille, un algorithme n'effectue pas nécessairement le même nombre d'opérations élémentaires. Pour cela, on distingue 3 types de complexité :

1. Complexité au pire des cas.
2. Complexité dans le meilleur des cas.
3. Complexité en moyenne des cas

Algorithme récursif :

- Meilleur cas :

Le problème des tours de Hanoï n'a pas de véritable "meilleur cas" en termes de complexité, car chaque instance du problème avec n disques nécessite exactement $2^n - 1$ déplacements pour être résolue. Chaque état de départ avec n disques suit le même schéma de résolution.

Complexité : La complexité reste $O(2^n)$ en temps et $O(n)$ en espace (à cause de la profondeur de la pile d'appels récursifs).

- Moyen cas :

Le cas moyen correspond également à n'importe quelle exécution de l'algorithme pour un nombre donné de disques, car le nombre de déplacements et la complexité ne varient pas pour différentes instances du même nombre de disques.

Complexité : La complexité reste $O(2^n)$ en temps et $O(n)$ en espace.

- Pire cas :

De même, le pire cas pour l'algorithme récursif est identique au cas moyen et au meilleur cas en termes de nombre de déplacements et de temps d'exécution. Il n'y a pas de cas où la complexité dépasse $O(2^n)$.

Complexité : La complexité reste $O(2^n)$ en temps et $O(n)$ en espace.

- Justification :

L'algorithme des tours de Hanoï consiste à déplacer n disques d'une tige à une autre en utilisant une tige intermédiaire, en suivant des règles précises : un seul disque peut être déplacé à la fois, et un disque plus grand ne peut jamais être placé sur un disque plus petit. La solution de cet algorithme suit un processus récursif bien défini.

Et comme l'algorithme est déterministe et suit toujours le même processus pour une taille d'entrée donnée, il n'y a pas de variabilité dans le nombre de mouvements nécessaires. Par conséquent, il n'y a pas de "meilleur" ou de "pire" cas, car chaque exécution de l'algorithme avec n disques

nécessitera exactement (2^n - 1) mouvements. Cette constance fait que l'algorithme est catégorisé comme ayant une complexité de Θ(2^n), indépendamment de toute autre considération de cas.

Algorithme itératif :

- Meilleur cas :

Comme pour l'algorithme récursif, l'algorithme itératif des tours de Hanoï n'a pas de véritable "meilleur cas" en termes de complexité temporelle. Chaque instance nécessite exactement $2^n - 1$ déplacements.

Complexité : La complexité reste O (2^n) en temps et O (n) en espace, car l'algorithme itératif utilise 3 pile de taille n.

- Moyen cas :

Le cas moyen correspond également à n'importe quelle exécution de l'algorithme pour un nombre donné de disques. Chaque instance aura le même coût moyen pour n disques.

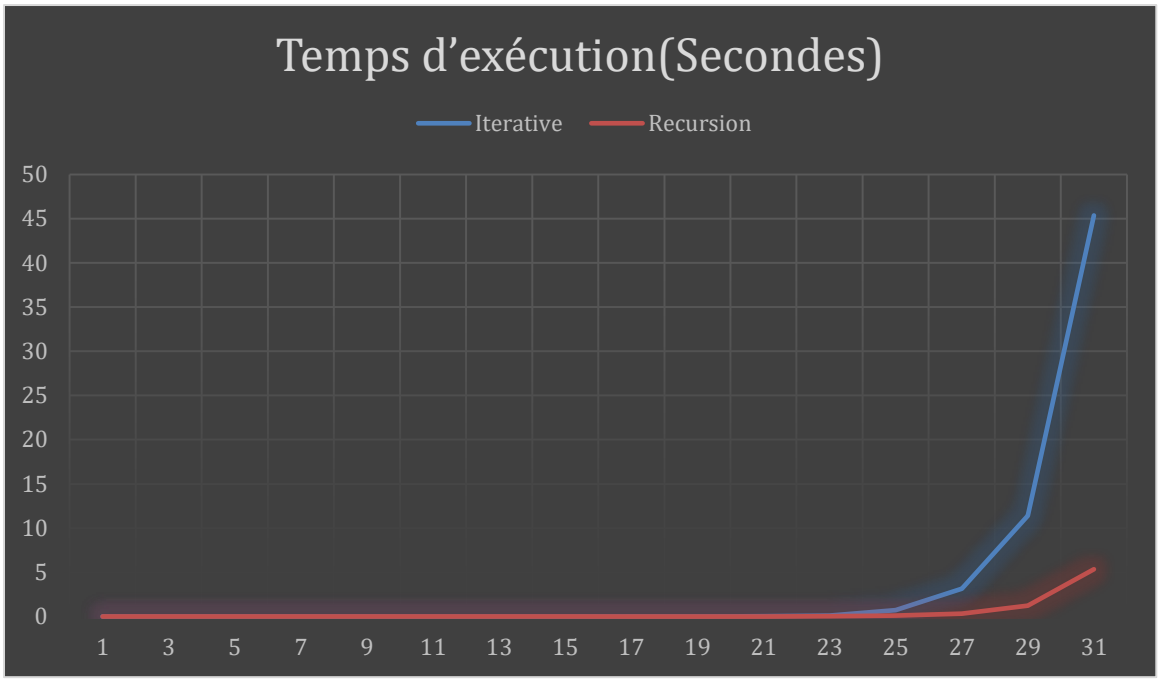
Complexité : La complexité reste O (2^n) en temps et O (n) en espace car l'algorithme itératif utilise 3 pile de taille n.

- Pire cas :

Le pire cas pour l'algorithme itératif est aussi déterminé par le nombre de disques. La complexité temporelle est O (2^n) et l'espace utilisé est O (n) .

Complexité : La complexité reste O (2^n) en temps et O (n) en espace car l'algorithme itératif utilise 3 pile de taille n.

Analyse des resultats :



Le graphe présenté compare le temps d'exécution des algorithmes de résolution itératif et récursif de la Tour de Hanoï pour différentes valeurs de n (nombre de disques). Voici une analyse détaillée des résultats observés :

Observation des Résultats

1. **Croissance Exponentielle**:
    - Les deux algorithmes montrent une croissance exponentielle du temps d'exécution à mesure que n augmente, ce qui est attendu étant donné que la complexité temporelle théorique de la Tour de Hanoï est $O(2\^n)$.
2. **Comparaison des Performances**:
    - Pour des valeurs de n petites à moyennes, les temps d'exécution des deux algorithmes (itératif et récursif) sont très proches.
    - À partir de n environ égal à 25, on observe que l'algorithme itératif commence à prendre beaucoup plus de temps que l'algorithme récursif.
3. **Écarts pour les Grandes Valeurs de n**:
    - Pour n autour de 27 et plus, le temps d'exécution de l'algorithme itératif augmente de manière significative par rapport à l'algorithme récursif.

Interprétation des Résultats

1. **Efficacité de l'Implémentation Récursive**:

    - L'algorithme récursif est plus performant pour des grandes valeurs de n. Cela pourrait être dû à des optimisations intrinsèques de la récursion dans certaines conditions ou à une meilleure gestion de la pile d'appels par le compilateur.

2. **Limites de l'Itératif**:

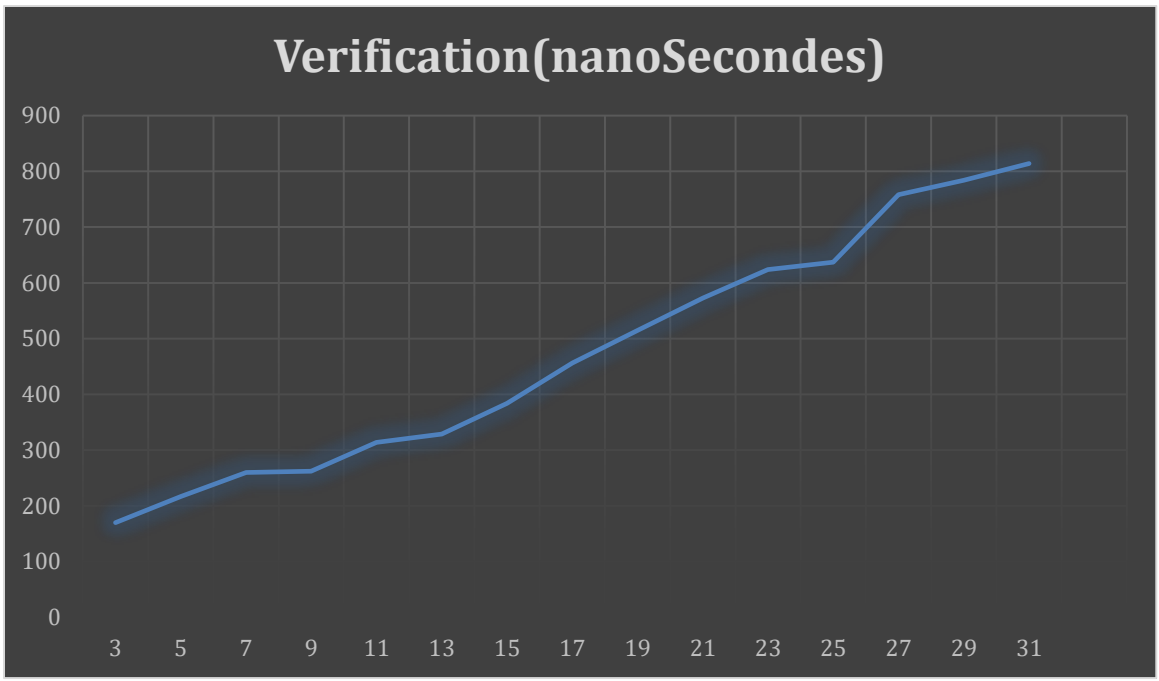
    - L'algorithme itératif semble souffrir d'une surcharge pour de grandes valeurs de nn. Cette surcharge pourrait être due à des opérations supplémentaires nécessaires pour gérer explicitement la pile dans l'implémentation itérative.

3. **Gestion de la Mémoire**:

    - L'implémentation récursive utilise implicitement la pile d'exécution du programme, ce qui peut être plus efficient que la gestion explicite de la pile nécessaire pour l'itératif.

4. **Explication pour différence très grandes**:

    - Les résultats pourraient être influencés par les optimisations spécifiques du compilateur pour les appels récursifs, comme la mise en cache des résultats ou l'élimination des appels de fonction.



Le graphe présenté montre le temps d'exécution de l'algorithme de vérification en fonction du nombre de disques n en nanosecondes. Voici une analyse détaillée des résultats observés:

Observation des Résultats

1. **Croissance Linéaire Apparente** :
    - Le graphe montre une tendance à la hausse relativement linéaire du temps d'exécution à mesure que le nombre de disques n augmente.
    - La vérification semble avoir une complexité $O(n)$ exactement comme les études thorique l'on prédis.
2. **Temps d'Exécution :**
    - Le temps d'exécution commence autour de 100 nanosecondes pour n=3 et atteint environ 800 nanosecondes pour n=31.
3. **Stabilité de la Croissance** :
    - La croissance du temps d'exécution est relativement stable, avec des augmentations progressives et sans grandes variations soudaines.

Interprétation des Résultats

1. **Complexité Linéaire** :

    - La vérification d'une solution de la Tour de Hanoï implique de vérifier les trois piles du code à la fin.

2. **Efficacité de l'Algorithme :**

    - L'algorithme de vérification semble efficace, avec des temps d'exécution très courts même pour des valeurs élevées de n.
    - Cette efficacité est importante pour l'utilisation pratique, car cela signifie que les solutions peuvent être validées rapidement même pour des problèmes de grande taille.

III. Conclusion

Au cours de ce projet, nous avons analysé le problème classique des tours de Hanoï et implémenté deux solutions en langage C : l'une utilisant une approche récursive et l'autre itérative. Les études théoriques ont révélé que la complexité en temps pour ces deux méthodes est de O(2^n), tandis que la complexité en espace est de O(n) pour la solution récursive.

Les simulations expérimentales ont confirmé cette complexité exponentielle, montrant une augmentation rapide du temps d'exécution et du nombre de déplacements à mesure que le nombre de disques augmente. Par ailleurs, nous avons observé que l'algorithme de vérification demande beaucoup moins de ressources en termes de temps et d'espace par rapport à l'algorithme de résolution.

En résumé, même si les approches récursive et itérative ont des complexités théoriques similaires, l'approche itérative peut offrir des avantages pratiques, notamment en termes de gestion de la mémoire. Cette analyse met en lumière l'importance de choisir la méthode la plus appropriée en fonction des contraintes spécifiques du problème et de l'environnement d'exécution.

IV. Refeferances :

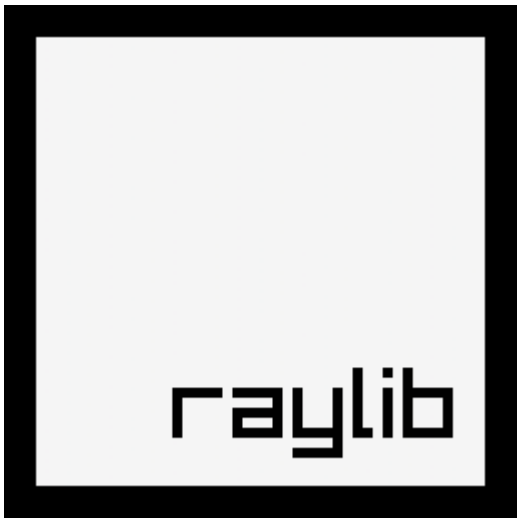
- Hinz, Andreas M., et al. *The tower of Hanoi-Myths and maths*. Berlin: Springer Basel, 2013.
- Li, Jie, and Rui Shen. "An Evolutionary Approach to Tower of Hanoi Problem." *Genetic and Evolutionary Computing: Proceeding of the Eighth International Conference on Genetic and Evolutionary Computing, October 18-20, 2014, Nanchang, China*. Springer International Publishing, 2015.
- Benoît Rittaud , Les Tours de Hanoï : L'histoire et la Mathématique. Accromath.
- Documentation officiel de Raylib
- Guide MSYS2

V. Implementation du code en C :

Introduction à la bibliothèque Raylib en C :

Raylib est une bibliothèque simple et facile à utiliser pour le développement de jeux vidéo, spécialement conçue pour la programmation de jeux. Écrite en C, elle offre des fonctionnalités pour les graphismes, l'audio et la gestion des entrées, ce qui en fait un outil puissant pour créer des jeux en 2D et 3D. Raylib est connue pour sa simplicité et sa facilité d'utilisation, ce qui la rend idéale pour notre presetation de projet.

Raylib fournit une API directe et puissante pour le développement de jeux, en faisant un choix approprié pour implémenter le jeu de la Tour de Hanoï. Sa simplicité et sa documentation exhaustive la rendent accessible aux développeurs de tous niveaux de compétence.



Pourquoi Raylib ?

- **Simplicité** : API minimaliste conçue pour être facile à apprendre et à utiliser.
- **Multi-plateforme** : Compatible avec Windows, Linux, MacOS, et plus encore.
- **Richesse fonctionnelle** : Offre une large gamme de fonctions pour les graphismes, l'audio et les entrées.
- **Open Source** : Libre d'utilisation et de modification sous la licence zlib/libpng.

Exemples de méthodes utiliser en Raylib :

Voici quelques méthodes clés de Raylib que nous avons utiliser dans notre presentation de la Tour de Hanoï :

1. **InitWindow(int width, int height, const char *title)**
   - **Description** : Initialise la fenêtre et le contexte OpenGL pour le rendu.
   - **Exemple** : **InitWindow(800, 600, "Tour de Hanoï");**
   - **Explication** : Cette fonction configure les dimensions de la fenêtre et son titre pour le jeu.
2. **BeginDrawing()**
   - **Description** : Commence le processus de rendu.
   - **Exemple** : **BeginDrawing();**
   - **Explication** : Cette fonction doit être appelée avant toute opération de dessin. Elle signale le début de la phase de dessin.
3. **ClearBackground(Color color)**
   - **Description** : Efface l'écran avec la couleur de fond spécifiée.
   - **Exemple** : **ClearBackground(RAYWHITE);**
   - **Explication** : Cette fonction est utilisée pour effacer l'écran avec une couleur donnée au début de chaque frame.
4. **DrawRectangle(int posX, int posY, int width, int height, Color color)**
   - **Description** : Dessine un rectangle à l'écran.
   - **Exemple** : **DrawRectangle(100, 100, 200, 50, RED);**
   - **Explication** : Cette fonction est utile pour dessiner les disques et les piquets dans le jeu de la Tour de Hanoï.
5. **EndDrawing()**
   - **Description** : Termine le processus de rendu et échange les buffers.
   - **Exemple** : **EndDrawing();**
   - **Explication** : Cette fonction doit être appelée après toutes les opérations de dessin pour afficher la frame rendue à l'écran.
6. **CloseWindow()**
   - **Description** : Ferme la fenêtre et désinitialise le contexte OpenGL.
   - **Exemple** : **CloseWindow() ;**
   - **Explication** : Cette fonction nettoie les ressources et ferme la fenêtre lorsque le jeu est terminé.

Comment compiler et tester le code ?

Pour compiler et tester notre application de la Tour de Hanoï construite avec Raylib, suivez les étapes suivantes :

Étape 1 : Installer MSYS2

Rendez-vous sur le site suivant : MSYS2. MSYS2 est un gestionnaire de paquets simple à installer et à utiliser, qui fournit un environnement de développement complet pour Windows.

Étape 2 : Installer MSYS2 et Ouvrir le Terminal

Après avoir téléchargé et installé MSYS2, exécutez-le. Un terminal MSYS2 s'ouvrira.

Étape 3 : Installer GCC et Raylib

Dans le terminal MSYS2, vous devez exécuter deux commandes pour installer le compilateur GCC et les fichiers requis de Raylib.

1. Installer le compilateur GCC : `pacman -S mingw-w64-ucrt-x86_64-gcc`
2. Télécharger et installer Raylib : `pacman -S mingw-w64-x86_64-raylib`

Étape 4 : Compiler le Projet

Ouvrez votre terminal MSYS2 et naviguez vers le dossier de votre projet contenant le fichier main.c. Utilisez la commande cd pour changer de répertoire, par exemple :

```sh
cd /chemin/vers/votre/projet/hanoi
```

Ensuite, exécutez la commande suivante pour compiler votre programme :

```sh
gcc main.c -o main -lraylib
```

Cette commande crée un fichier exécutable main.exe prêt à être exécuté.

Voici le lien vers l'implémentation du code : (dans le dossier ZIP)

VI. distribution des tâches

	Tache principale	Tache Secondaire
DENADANI Safwan Souhil	-Code C et algorithme itérative de Résolution -Calcule complexité théorique de l'algorithme de résolution itérative -Simulation de l'algorithme de résolution itérative	
GUENANE Abdelkarim	-Historique et présentation -Définition formelle -modélisation de la solution -écriture pseudo code itératif -écriture pseudo code récursive -meilleur cas, moyen cas, pire cas de chaque algorithme	
CHEMLI Ayoub	-chef d'equipe (organisation des taches et assurer le bien deroulement du projet) -redaction du repport -Code C et algorithme récursive de Résolution -Calcule complexité théorique de l'algorithme de résolution récursive -Exemple du problème avec sa solution	
BELKADI Adam	-Visualisation Graphique -Analyse des résultats	-Conclusion
BOUCHAMA Imaddedine	-Code C et algorithme Vérification (Version itérative) -Calcule complexité théorique et spatial de la vérification -Simulation de l'algorithme de vérification -Conclusion	-Analyse des résultat

CHELALI **Mohamed Said** **Nail**	-Code C et algorithme de Vérification (Version récursive) -Calcule complexité théorique et spatial de la vérification -Simulation de l'algorithme de résolution récursive -Simulation de l'algorithme de vérification	