République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumédiène

Faculté d'Electronique et d'Informatique
Département Informatique

Master Systèmes Informatiques intelligents

Module : Conception et Complexité des Algorithmes

Rapport de projet 2 de TP

Réalisé par :
AIT AMARA Mohamed, 181831072170
BOUROUINA Rania, 181831052716
CHIBANE Ilies, 181831072041
HAMMAL Ayoub, 181831048403

Année universitaire : 2021 / 2022

Table des matières

Chapitre 1

la Tour de Hanoi, historique et présentation du problème

1.1 Introduction Historique

La Tour de Hanoi est inspirée à Lucas par l'un de ses amis, le professeur Claus de Siam qui raconte une histoire ayant lieu au coeur du temple Kashi Vishwanath : Trois aiguilles de diamant, plantées dans une dalle d'airain. [. . .] Sur une de ces aiguilles, Dieu enfila au commencement des siècles, 64 disques d'or pur, le plus large reposant sur l'airain, et les autres, de plus en plus étroits, superposés jusqu'au sommet.

C'est la tour sacrée du Brahmâ. Nuit et jour, les prêtres se succèdent sur les marches de l'autel, occupés à transporter la tour de la première aiguille sur la troisième, sans s'écarter des règles fixes que nous venons d'indiquer, et qui ont été imposées par Brahma. Quand tout sera fini, la tour et les brahmes tomberont, et ce sera la fin des mondes ! [1]

1.1.1 Fait amusant

De ce qui était mentionné, si on utilise à 64 disques on aura besoin de $2^{64} - 1$ déplacements. Supposant que chaque déplacement dure 1 seconde, on obtiendra 86 400 déplacements par jour, le jeu se terminera après 584,5 milliards d'années approximativement, soit une quarantaine de fois l'âge de l'univers.[1]

1.2 Présentation du problème

Les tours de Hanoï sont un jeu de réflexion imaginé par le mathématicien français Édouard Lucas, qui consiste à déplacer des disques de diamètres différents d'une tour de « départ » à une

1. Edouard Lucas, Récréations mathématiques, tome 3, (1892), réédité par la Librairie Albert Blanchard, (1979), p. 58

tour d'« arrivée » en passant par une tour « intermédiaire », et ceci en un minimum de coups, tout en respectant les règles suivantes[2] :

-> On peut déplacer au maximum un seul disque par déplacement. -> On n'a pas le droit de poser un disque sur un autre plus petit.

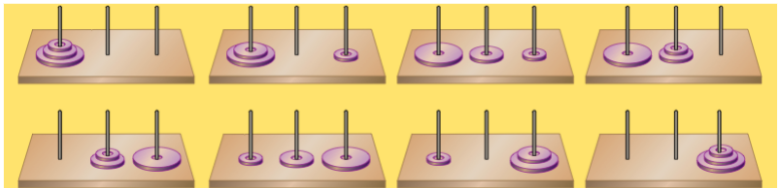

FIGURE 1.1 – Exemple d'une solution dans le cas de trois disques

1.2.1 Polémique

Il existe toujours cette confusion dans certains articles scientifiques qui mentionnent que le problème de la tour de Hanoi est NP alors que ce n'est pas le cas.

Dans ce problème les étapes correctes peuvent être déterminées dès le départ sans avoir besoin d'une recherche d'essais-erreurs pour prendre la bonne décision. Par conséquent, ce problème n'est pas un problème de décision, auquel des classes telles que NP s'appliquent.

C'est plus un problème de fonction. La complexité temporelle est en effet déterminée par le nombre d'étapes à sortir, qui est de 2^{n-1}, c'est-à-dire $\mathcal{O}(2^n)$. La classe correspondante serait donc FEXPTIME-Complete, le préfixe F signifiant Function, et Complete signifiant que cela ne peut se faire en un temps inférieur à exponentiel (comme P). Elle est analogue à la classe EXPTIME-Complete pour les problèmes de décision, c'est-à-dire $\mathcal{O}(2^{polynomial(n)})$.

Il faut savoir que la classe EXPTIME En théorie de la complexité, est la classe de complexité qui est l'ensemble des problèmes de décision décidés par une machine de Turing déterministe en temps exponentiel. Mais il existe des classes plus grosses comme EXPSPACE et NEXPTIME (respectivement les problèmes pouvant être résolus en espace exponentiel, et en temps exponentiel sur machine non déterministe). L'illustration ci-dessous montre le diagramme d'inclusions des classes de complexité classiques :

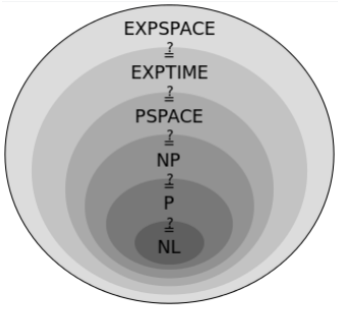

FIGURE 1.2 – Diagramme d'inclusions des classes de complexité classiques.

1.2.2 Définition formelle du problème

Il est facile de démontrer par récurrence que si n est le nombre de disques, il faut $2^n - 1$ coups au minimum pour parvenir à ses fins, quantité qui augmente très rapidement avec n. En effet, soient A, B et C les trois emplacements des tours ; notons x_n le nombre de déplacements de disques nécessaires au déplacement d'une tour complète. Pour déplacer une tour de n disques de A vers C, on effectue ces trois étapes[1] :

déplacer la tour des $n-1$ premiers disques de A vers B (étape qui nécessite x_{n-1} déplacements, d'où la récurrence) ; déplacer le plus grand disque de A vers C (un déplacement supplémentaire) ; déplacer la tour des n-1 premiers disques de B vers C (à nouveau x_{n-1} déplacements). Le nombre de déplacements de disques vérifie donc la relation de récurrence :

$$x_0 = 0$$
$$x_n = 2x_{n-1} + 1 \text{ si } n >= 1$$

ce qui donne bien :

$$x_n = 2^n - 1$$

On peut montrer que la méthode décrite ici donne la séquence optimale (celle qui nécessite le moins de coups), et que celle-ci est unique. En effet, pour déplacer la tour de n disques de A vers C, on devra forcément, à un moment ou à un autre, déplacer le plus grand disque de A vers C, et pour ce faire, on devra avoir empilé les $n-1$ premiers disques en B.

1.3 Conclusion

Dans ce chapitre on a parlé de l'origine du problème de la tour de hanoi et on l'a expliqué d'une manière classique et générale. Dans ce qui suit, on présentra notre solution récursive ainsi que des exemples démonstratifs.

Chapitre 2

Présentation de la solution

2.1 Modélisation de la solution

Dans ce problème, chaque anneau porte un numéro séquentiel unique $a_i \in [1, n]$ qui représente sa taille tel que n est le nombre maximal d'anneaux (E.g. l'anneau avec le nombre 1 est plus petit que l'anneau avec le nombre 3).

De plus, on modélise chaque tour par un tableau T_j d'une taille égale au nombre maximum d'anneaux n. Si un niveau i de la tour j contient un anneau $a_{i'}$, alors $T[j, i] = a_{i'}$, sinon $T[j, i] = 0$. Le niveau le plus bas de la tour (la base de la tour) est placé à la dernière case du tableau ; $\forall j$ $T[j, n]$ est le niveau le plus bas de la tour (voir Figure 2.1).

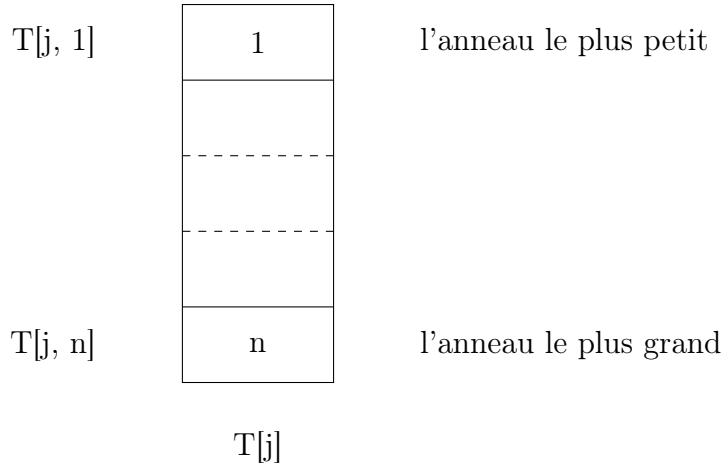

FIGURE 2.1 – Exemple d'une tour avec tous les anneaux

Par conséquent, le bord de jeu peut être représenté par une matrice, colonne par colonne, où chaque colonne est en réalité une tour du jeu.

$$\textbf{bord} = \begin{pmatrix} T[1,1] & T[2,1] & ... \\ ... & & \\ T[1,n] & T[2,n] & ... \end{pmatrix}$$

Un exemple d'initialisation classique de trois tours avec trois anneaux placés sur la première tour :

$$\textbf{bord} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \end{pmatrix}$$

Règles de changement d'état Pour passer d'un état de bord vers le suivant, on ne peut bouger qu'un seul anneau du haut d'une tour vers une autre tour, à condition que l'anneau supérieur de la tour destination ait un nombre supérieur à l'anneau qu'on veut bouger.

Plus formellement, on peut déplacer le premier élément non nul d'une colonne s'il existe vers une autre colonne, s'il y a encore de la place et que le premier element non nul de la colonne destination est supérieur à celui qu'on déplace.

2.2 Algorithme de résolution

Cet algorithme récursif permet de produire la séquence exacte d'actions pour résoudre le problème des tours de Hanoi.

On commence d'abord par déplacer les $n-1$ disques de la tour de départ vers la tour intermédiaire par un appel récursif. Puis, le plus grand disque restant est transporté vers la tour d'arrivée. Ensuite, les $n-1$ disques qui se trouvaient sur la tour intermédiaire sont déplacés vers la tour d'arrivée par le même processus récursif.

Algorithme 1 : Hanoi

Données : bord : matrice [1, 3][1, n] d'entiers, depart, arrivee, intermediaire : 1..3, nbdisques : entier
Résultat : bord : matrice [1, 3][1, n] d'entiers
début
  si $nbdisques = 1$ **alors**
    `// S'il reste un seul disque à déplacer, on le déplace directement`
    deplacer(board, depart, arrivee)
  sinon
    Hanoi(bord, depart, intermediaire, arrivee, nbdisques - 1);
    deplacer(board, depart, arrivee); `// Deplacer le disque supérieur de la tour depart vers la tour arrivee`
    Hanoi(bord, intermediaire, arrivee, depart, nbdisques - 1);
  fin si
fin

La fonction *deplacer* permet de déplacer le disque de niveau supérieur d'une tour vers un autre.

Fonction deplacer(Entrée/sortie : bord : matrice [1, 3][1, n] d'entiers, Entrée : depart, arrivee : 1..3)

Variable :
i, j : entier;
début
  `// Trouver le disque supérieur de la tour depart`
  $i \leftarrow 1$;
  tant que $i \leq n$ *et* $bord[depart][i] = 0$ **faire**
    $i \leftarrow i + 1$;
  fin tq
  `// Trouver le disque supérieur de la tour arrivee`
  $j \leftarrow 1$;
  tant que $j \leq n$ *et* $bord[arrivee][j] = 0$ **faire**
    $j \leftarrow j + 1$;
  fin tq
  $bord[arrivee][j-1] \leftarrow bord[depart][i]$;
  $bord[depart][i] \leftarrow 0$;
fin

Calcul de la complexité La complexité est exprimée en matière de nombre d'opérations de déplacement effectuées. En l'occurrence, le nombre de déplacements est exprimé selon la suite

numérique suivante :

$$h(1) = 1$$

$$h(n) = 2 * h(n-1) + 1$$

où n représente le nombre total de disque à déplacer.

En remplaçant $h(n-1)$ par la formule réccurente, on obtient :

$$h(n) = 2 * (2 * h(n-2) + 1) + 1$$

$$h(n) = 2 * (2 * (2 * h(n-3) + 1) + 1) + 1$$

$$...$$

$$h(n) = 2^n - 1$$

Ce résultat peut être démontré par récurrence comme suit :
Cas de base ou pour $n = 1$:

$$h(1) = 2^1 - 1 = 1$$

Donc la formule est correcte pour $n = 1$.
Supposons que la proposition $h(i) = 2^i - 1$ est correcte pour $\forall i \leq n$. Montrons qu'elle est aussi correcte pour $n + 1$:

$$h(n+1) = 2 * h(n) + 1$$

D'après l'hypothèse :

$$h(n+1) = 2 * (2^n - 1) + 1$$

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

Donc la proprosition est correct $\forall n \in \mathbb{N}$.

Et ainsi la complexité est égale à $\mathcal{O}(2^n)$.

2.3 Algorithme de vérification

Le problème des tours de Hanoi à trois tours admet une unique solution, sous forme d'une suite de déplacements qui génèrent un séquencement d'états intermédiaires.

Pour vérifier la validité d'une solution quelconque, nous devons nous assurer que chaque déplacement est bien réglementaire (concerne le disque le plus haut et ne pose pas un disque sur un autre disque de taille plus petite) et que le dernier état engendré correspond effectivement à l'état but, c.à.d. que toutes les tours sont vides sauf la tour cible. De plus, la séquence de déplacement doit être exactement de longueur $2^n - 1$ tel que n représente le nombre de disques, car la solution est unique, donc égale à la solution calculée par l'algorithme exact.

Nous nous assurons que l'algorithme de génération de solutions produit des séquences de déplacements qui respectent ces conditions susmentionnées. Par conséquent, nous devons vérifier que le dernier état qui doit correspondre à l'état but dans lequel les anneaux sont alignés sur la tour cible ou destination.

```
Fonction verification(Entrée : bord : matrice [1, 3][1, n] d'entiers, arrivee : 1..3) : booleén
    Variable :
    début
        pour i ← 1 à n faire
            si bord[arrivee][i] ≠ i alors
                retourner faux;
            fin si
        fin pour
        retourner vrai;
    fin
```

Calcul de complexité La complexité de l'algorithme de vérification est triviale, elle est égale à la complexité du parcours séquentiel des éléments d'un tableau (la tour d'arrivée).

La complexité temporelle est égale à $\mathcal{O}(n)$ tel que n est le nombre de disques.

Quant à la complexité spatiale, sachant que la fonction de vérification ne prend que le dernier état comme paramètre, elle est donc égale à la taille de la matrice d'entiers : $3 * n * n \cong \mathcal{O}(n^2)$.

2.4 Illustration d'une instance du problème

Nous étudions dans ce qui suit une instance du problème des tours de Hanoi avec une disposition de trois tours et trois disques sur la tour initiale.

$$\textbf{bord} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 0 \end{pmatrix}$$

L'unique solution à cette disposition est la séquence suivante de 7 déplacements :

— On déplace le disque de taille 1 de la tour 1 vers la tour 3.

$$\textbf{bord} = \begin{pmatrix} 0 & 0 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 1 \end{pmatrix}$$

— On déplace le disque de taille 2 de la tour 1 vers la tour 2.

$$\textbf{bord} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 3 & 2 & 1 \end{pmatrix}$$

— On déplace le disque de taille 1 de la tour 3 vers la tour 2.

$$\textbf{bord} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 3 & 2 & 0 \end{pmatrix}$$

— On déplace le disque de taille 3 de la tour 1 vers la tour 3.

$$\textbf{bord} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 2 & 3 \end{pmatrix}$$

— On déplace le disque de taille 1 de la tour 2 vers la tour 1.

$$\textbf{bord} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 2 & 3 \end{pmatrix}$$

— On déplace le disque de taille 2 de la tour 2 vers la tour 3.

$$\textbf{bord} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 2 \\ 1 & 0 & 3 \end{pmatrix}$$

— On déplace le disque de taille 1 de la tour 1 vers la tour 3.

$$\textbf{bord} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 3 \end{pmatrix}$$

Chapitre 3

Étude expérimentale

3.1 Complexité théorique de l'algorithme de résolution

Complexité temporelle La complexité de l'algorithme de résolution comme calculée précédemment est de l'ordre de $\mathcal{O}(2^n)$, et elle est précisément égale à $2^n - 1$ qui est une complexité exponentielle.

Le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme de résolution selon la variation de la taille du problème (le nombre de disques) :

N	1	10	50	100	150	250	500	750	1000
t(ns)	2	1024	1,1259E+15	1,26765E+30	1,42725E+4	1,80925E+75	3,27339E+150	5,92239E+225	1,07151E+301

La figure suivante (voir Figure 3.1) représente l'évolution du temps d'exécution théorique selon le nombre de disques :

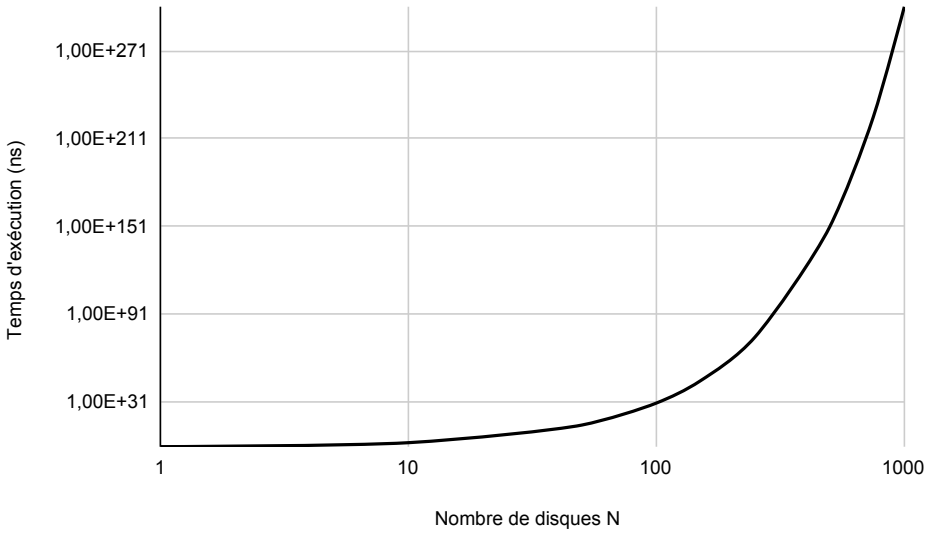

FIGURE 3.1 – Temps d'exécution théorique de l'algorithme de résolution

Depuis le graphe, nous observons que le temps d'exécution évolue de manière exponentielle selon le nombre de disques de départ. Il atteint rapidement des temps d'exécution incommensurables le rendant inexploitable.

Complexité spatiale L'algorithme exploite une matrice $3 * n$ tel que n est le nombre de disques. Chaque disque est représenté par un entier de taille 4 octets. De plus, la taille de la représentation est statique au cours de l'exécution. Par conséquent, la complexité spatiale est égale à $4 * 3 * n$ octets donc de l'ordre de $\mathcal{O}(n)$.

Cependant, l'algorithme récursif fait au maximum n appels récursifs (le nombre maximum d'appels est le nombre maximum de disques pouvant être déplacés à la fois). L'adresse de retour étant stockée sur 8 octets, la taille maximale de la pile d'appel de fonctions exploitée est donc égale à $8 * n$ octets qui est de l'ordre de $\mathcal{O}(n)$.

La complexité spatiale totale est donc de l'ordre de $\mathcal{O}(n)$.

3.2 Complexité théorique de l'algorithme de vérification

Complexité temporelle La complexité temporelle de l'algorithme de vérification est linéaire et elle est égale à $n \cong \mathcal{O}(n)$, car l'algorithme parcourt la tour d'arrivée et vérifie si les disques sont bien rangés dans le bon ordre.

Le tableau suivant représente les temps d'exécution théorique en nanoseconde de l'algorithme de vérification selon la variation de la taille du problème (le nombre de disques) :

N	1	10	50	100	150	250	500	750	1000
t(ns)	1	10	50	100	150	250	500	750	1000

La figure suivante (voir Figure 3.2) représente l'évolution du temps d'exécution théorique selon le nombre de disques :

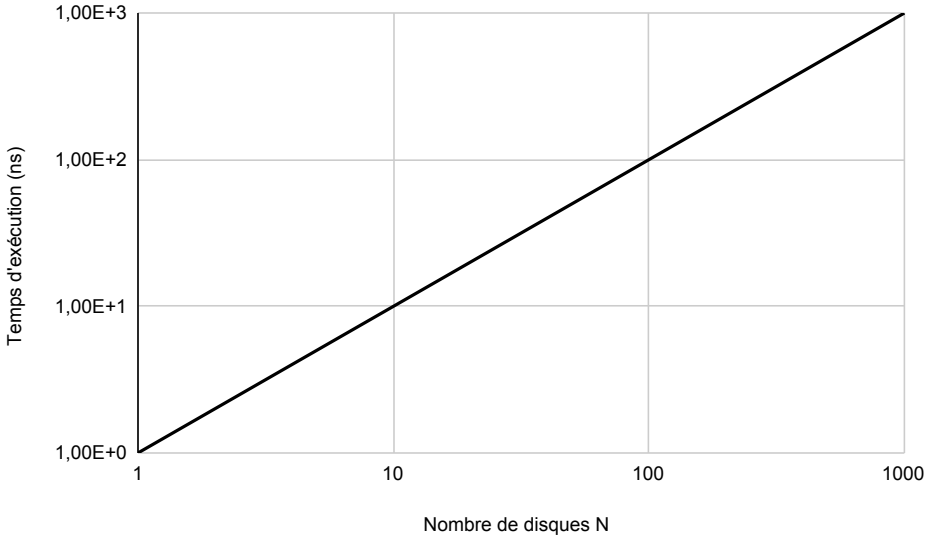

FIGURE 3.2 – Temps d'exécution théorique de l'algorithme de vérification

Depuis le graphe, nous observons que le temps d'exécution évolue avec une tendance linéaire avec l'augmentation du nombre de disques.

Complexité spatiale La complexité spatiale de l'algorithme de vérification est égale à la taille de la matrice plus l'indice de la tour cible qui est un entier, c.à.d. $3 * n + 4$ octets donc de l'ordre $\mathcal{O}(n)$.

3.3 Expérimentation

3.3.1 Algorithme de résolution

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme de résolution récursif selon la variation du nombre de disques.

N	3	5	7	10	15	20	25
t1(ns)	663	2029	7829	59701	2168020	90192200	3190280000
t2(ns)	595	2099	8018	60986	2648340	88261800	3188540000
t3(ns)	642	2118	11230	62336	2206080	89520400	3176230000
t4(ns)	597	2085	8118	61247	2211980	89318100	3179730000
t5(ns)	596	2021	7724	61253	2168540	88693300	3187330000
t6(ns)	882	2042	8044	61308	2275150	88450100	3175950000
t7(ns)	867	2067	7977	63335	2258050	88788800	3181730000
t8(ns)	615	2059	8085	63063	2301650	88529200	3178430000
t9(ns)	593	2077	7942	60909	2211830	88074100	3176490000
t10(ns)	619	2182	7955	61240	2221290	89301200	3182450000
moyenne(ns)	667	2078	8292	61538	2267093	88912920	3181716000

La figure suivante (voir Figure 3.3) représente l'évolution du temps d'exécution selon le nombre de disques.

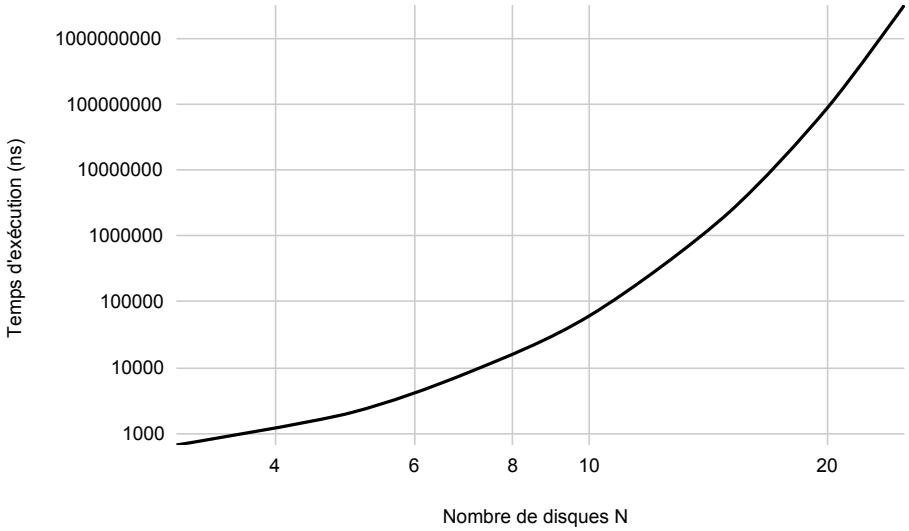

FIGURE 3.3 – Temps d'exécution de l'algorithme de résolution

Depuis la figure 3.3, nous remarquons bien que le temps d'exécution de l'algorithme de résolution est exponentiel, conforme à la complexité de $\mathcal{O}(2^n)$.

3.3.2 Algorithme de génération de solution

Cet algorithme fait au minimum $2^n - 1$ itérations, qui est équivalent au nombre d'étapes de la solution. Si l'algorithme génère à une étape un déplacement non autorisé, ce déplacement est rejeté et un autre est généré (si à son tour il n'est pas autorisé en génère un autre et ainsi de suite). Donc il est de complexité $\mathcal{O}(2^n)$.

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme de génération de solution pour le problème des tours de hanois selon la variation du nombre de disques.

N	3	5	7	10	15	20	25
t1(ns)	16484	25099	73290	304737	11255800	401042000	14478900000
t2(ns)	13480	23469	72592	310506	11503700	402620000	14470300000
t3(ns)	15804	22381	70229	321948	11607800	401610000	14440400000
t4(ns)	16334	24410	69875	312012	11348400	401075000	14418900000
t5(ns)	15780	23887	79824	329052	11008400	402276000	14439500000
t6(ns)	14275	21296	72074	321122	11444300	402046000	14443800000
t7(ns)	13965	30082	52879	309217	1,17E+07	405221000	14440000000
t8(ns)	14042	22535	62667	310238	1,14E+07	405030000	14471900000
t9(ns)	13832	22679	49077	311514	11376700	405483000	14419800000
t10(ns)	15764	22572	69374	311673	11215200	402717000	14449700000
moyenne(ns)	14976	23841	67188	314202	11386730	402912000	14447320000

La figure suivante (voir Figure 3.4) représente l'évolution du temps d'exécution selon le nombre de disques.

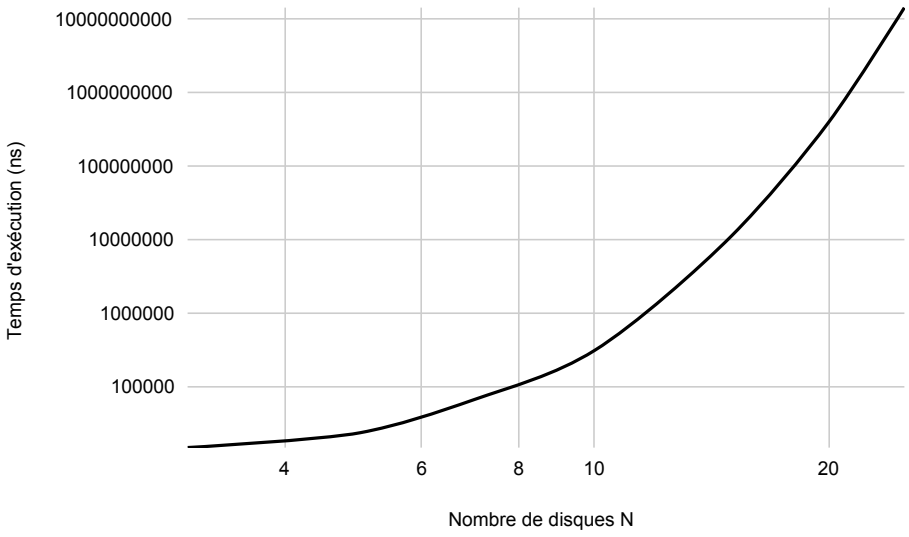

FIGURE 3.4 – Temps d'exécution de l'algorithme de génération de solution

Depuis la figure 3.4, nous remarquons bien que le temps d'exécution de l'algorithme de résolution est exponentiel, conforme à la complexité de $\mathcal{O}(2^n)$. Cependant, les temps d'exécution sont bien plus élevés que ceux de l'algorithme de résolution, car même si l'algorithme de génération fait $2^n - 1$ itération, certains déplacements sont rejetés.

3.3.3 Algorithme de vérification de solution

Le tableau suivant représente les temps d'exécution en nanoseconde de l'algorithme de vérification de solution pour le problème des tours de hanois selon la variation du nombre de disques.

N	3	5	7	10	15	20	25
t1(ns)	71	61	113	80	75	146	182
t2(ns)	61	68	111	68	77	94	61
t3(ns)	66	95	66	80	143	84	156
t4(ns)	68	68	82	66	101	86	143
t5(ns)	76	75	97	108	65	141	197
t6(ns)	57	67	81	110	64	104	179
t7(ns)	58	90	77	104	180	148	91
t8(ns)	74	73	70	78	137	147	102
t9(ns)	78	75	74	117	105	126	105
t10(ns)	60	66	68	80	68	78	164
moyenne(ns)	67	74	84	89	102	115	138

La figure suivante (voir Figure 3.5) représente l'évolution du temps d'exécution selon le nombre de disques.

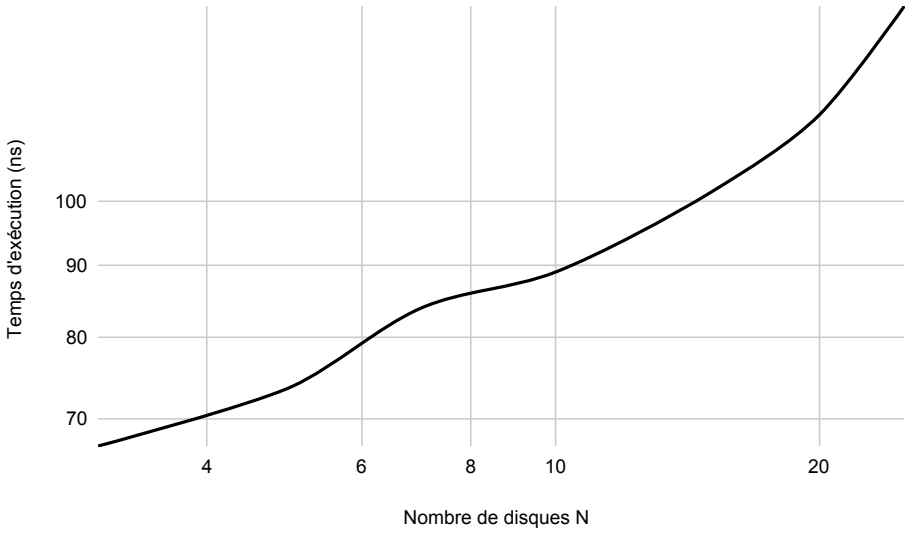

FIGURE 3.5 – Temps d'exécution de l'algorithme de vérification de solution

Depuis la figure 3.5, nous observons une trajectoire presque linéaire, qui est représentative de la complexité $\mathcal{O}(n)$. Les fluctuations sont dûes au nombre de disques relativement bas.

Discussion des résultats On constate que la complexité de l'algorithme de résolution pour le problème des tours de hanoi évolue bien de manière exponentielle comme l'a montré l'étude

expérimentale. Le temps d'exécution pour une instance du problème de 20 disques est de l'ordre de dizaine de secondes, celle avec 30 disques dépasse les quinzaines de minutes. Avec de tels résultats, il est clair que l'exploitation effective de ce genre de complexités est inenvisageable, voire impossible.

Quant à l'algorithme de vérification, il a une simple complexité linéaire. Il ne fait que vérifier l'état de la dernière tour, les déplacements quant à eux, sont supposés corrects et valides par l'algorithme de génération de solution.

Chapitre 4

Conclusion générale

4.1 Conclusion

Dans ce rapport, nous explorons comment le jeu d'enfant classique de la tour de Hanoï peut être utilisé à un niveau élaboré ou élémentaire. Il peut être exploité pour la vulgarisation des concepts mathématiques et pousse le joueur débutant à la réflexion en résolvant le jeu avec 3 ou 4 disques, tandis que les plus avancés peuvent se poser plusieurs questions sur les liens de la structure avec les systèmes de numération, la modification des règles et d'autres questions qui sont encore l'objet des recherches d'aujourd'hui.
Nous avons commencé l'exploration en présentant le problème, son origine et sa définition formelle pour trouver la solution.
Dans le deuxième chapitre, nous avons entamer le processus de la résolution en modélisant cette dernière. Nous avons ensuite détaillé les algorithmes de solution et de vérification nécessaires ainsi que le calcul de leurs complexités. Finalement, nous avons illustrer une instance du problème avec la solution générée.
Nous avons fini notre étude avec des expériences pour calculer les complexités théroriques (temporelle et spatiale) de nos algorithmes et les comparer aux résultats expérimentaux obtenus sur plusieurs essais.
L'expérimentation avec ce jeu offre à chacun l'occasion d'apprécier les ressorts du jeu, tels que ses aspects algorithmiques, les structures de données, les mathématiques appliquées, etc.
Par ailleurs, le jeu de la tour de Hanoï est un cas idéal pour illustrer la puissance de la notion d'algorithme récursif et donne un exemple du calcul de factorielle n. En effet, son efficacité est impressionnante pour seulement quelques lignes de code. Contrairement à ce dernier type d'algorithme, un algorithme itératif de résolution de la tour de Hanoï est en général beaucoup moins facilement trouvé et est toujours plus long.

Bibliographie

[1] Accessed at 31/12/2021. [Online]. Available : https://fr.wikipedia.org/wiki/Tours_de_Hano%C3%AF

[2] Accessed at 31/12/2021. [Online]. Available : https://accromath.uqam.ca/2016/02/les-tours-de-hanoi-et-la-base-trois/

Chapitre 5

Annexe

main.cpp

```
#include <iostream>
#include <stdlib.h>
#include<time.h>
#include <bits/stdc++.h>

#include "AlgoHanoi.hpp"

using namespace std;

int main()
{
    srand(time(0));
    cout <<"\t\t\tBienvenu dans la partie 2 du projet de Conception, analyse et complexite des algorithmes"<<endl;
    cout <<"Veuille choisir l'une des options suivante :"<<endl;
    cout <<"1) Algorithme des d'Hanoi recursif"<<endl;
    cout <<"2) Algorithme des d'Hanoi recursif avec affichage"<<endl;
    cout <<"3) Algorithme des d'Hanoi iteratif"<<endl;
    cout <<"4) Algorithme des d'Hanoi iteratif avec affichage"<<endl;
    cout <<"5) Solution aleatoire"<<endl;

    int ch, NbrDisque;
    Pilier TourHanoi[3];
    cin >> ch;
    chrono::time_point<chrono::steady_clock> start, stop;
    chrono::duration<double, nano> duration;
    switch(ch)
    {
        case 1:
            cout << "Choisissez le nombre de disque" << endl;
            cin >> NbrDisque;
```

```
31                  NDisque = NbrDisque;
32                  init(TourHanoi, NbrDisque);
33                  start = chrono::steady_clock::now();
34                  HanoiRecursif(NbrDisque, 0, 2, TourHanoi);
35                  stop = chrono::steady_clock::now();
36                  duration = stop - start;
37                  cout << "Tour d'hanoi resolu en " << deplacement << "
    ↪  deplacements" <<endl;
38                  cout << "la fonction de resolution de la tour d'hanoi aura prit "
    ↪  << duration.count()
39                       << "ns afin de terminer son execution" <<endl;
40                  break;
41
42              case 2:
43                  cout << "Choisissez le nombre de disque" << endl;
44                  cin >> NbrDisque;
45                  NDisque = NbrDisque;
46                  init(TourHanoi, NbrDisque);
47                  AfficheLesTours(TourHanoi, NbrDisque);
48                  Sleep(500);
49                  HanoiRecursifAffichage(NbrDisque, 0, 2, TourHanoi);
50                  cout << "Tour d'hanoi resolu en " << deplacement << "
    ↪  deplacements" <<endl;
51                  break;
52
53              case 3:
54                  cout << "Choisissez le nombre de disque" << endl;
55                  cin >> NbrDisque;
56                  NDisque = NbrDisque;
57                  init(TourHanoi, NbrDisque);
58                  start = chrono::steady_clock::now();
59                  HanoiIteratif(NbrDisque, 0, 2, TourHanoi);
60                  stop = chrono::steady_clock::now();
61                  duration = stop - start;
62                  cout << "Tour d'hanoi resolu en " << deplacement << "
    ↪  deplacements" <<endl;
63                  cout << "la fonction de resolution de la tour d'hanoi aura prit "
    ↪  << duration.count()
64                       << "ns afin de terminer son execution" <<endl;
65                  break;
66
67              case 4:
68                  cout << "Choisissez le nombre de disque" << endl;
69                  cin >> NbrDisque;
70                  NDisque = NbrDisque;
71                  init(TourHanoi, NbrDisque);
72                  AfficheLesTours(TourHanoi, NbrDisque);
```

```
                Sleep(500);
                HanoiIteratifAffichage(NbrDisque, 0, 2, TourHanoi);
                cout << "Tour d'hanoi resolu en " << deplacement << "
↪ deplacements" <<endl;
                break;

            case 5:
                cout << "Choisissez le nombre de disque" << endl;
                cin >> NbrDisque;
                NDisque = NbrDisque;
                init(TourHanoi, NbrDisque);
                start = chrono::steady_clock::now();
                Aleatoire(NbrDisque, TourHanoi);
                if (Verification(NbrDisque, TourHanoi))
                    cout << "Solution valide" << endl;
                else
                    cout << "Solution invalide" << endl;
                stop = chrono::steady_clock::now();
                duration = stop - start;
                cout << "la fonction de generation de solution de tour d'hanoi et
↪ de verification aura prit " << duration.count()
                     << "ns afin de terminer son execution" <<endl;
                break;

            default:
                cout << "erreur "<<endl;
        }
    }
```

Affichage.hpp

```
#ifdef _WIN32
#include <Windows.h>
#else
#include <unistd.h>
#endif

#include <iostream>
#include <cstdlib>

#include "structure.hpp"

using namespace std;

//affiche un char n fois
void affiche(char c, int n)
```

```
16      {
17        for (int i = 0; i < n; ++i)
18          cout << c;
19      }
20
21      //affiche une tour
22      void AfficherTour(int d, int n)
23      {
24        //pas disque affihe la poutre du pilier
25        if (d == 0) {
26          affiche(' ', n-1);
27          cout << '|';
28          affiche(' ', n);
29        }
30        //affichage du pilier selon sa taille
31        else {
32          affiche(' ', n-d);
33          affiche('-', 2*d-1);
34          affiche(' ', n-d+1);
35        }
36      }
37
38      //affiche des 3 tours
39      void AfficheLesTours(Pilier TH[], int n)
40      {
41        for (int i = 0; i < n; ++i) {
42          AfficherTour(TH[0].Pilier[i], n);
43          AfficherTour(TH[1].Pilier[i], n);
44          AfficherTour(TH[2].Pilier[i], n);
45          cout << endl;
46        }
47
48        affiche('#', 6*n-1);
49        cout << endl << endl;
50        Sleep(500);
51      }
```

AlgoHanoi.hpp

```
1       #include <iostream>
2       #include <stdlib.h>
3       #include <math.h>
4       #include <vector>
5       #include <time.h>
6
7
```

```cpp
#include "Affiichage.hpp"

using namespace std;

struct Dep
{
    int o, d;
};

//variable globale afin de sauvegarder le nombre de deplacements et de disques
int deplacement = 0, NDisque = 0;

//resolution recursif des tours d'hanoi
void HanoiRecursif(int n, int origine,int destination, Pilier TH[])
{
    if (n != 0)
    {
        //on determine le pilier intermediaire
        int auxiliaire = autre(origine, destination);
        //premier appel recursif ou on deplace le disque de son pilier actuel au pilier intermediaire
        HanoiRecursif(n-1, origine, auxiliaire, TH);
        //on effectue le deplacement
        deplacement++;
        deplacer(TH[origine], TH[destination],NDisque);
        //second appel recursif ou on deplace le disque du pilier intermediaire au pilier destiination
        HanoiRecursif(n-1, auxiliaire, destination, TH);
    }
}

//resolution recursif des tours d'hanoi + affichage
void HanoiRecursifAffichage(int n, int origine,int destination, Pilier TH[])
{
    if (n != 0)
    {
        //on determine le pilier intermediaire
        int auxiliaire = autre(origine, destination);
        //premier appel recursif ou on deplace le disque de son pilier actuel au pilier intermediaire
        HanoiRecursifAffichage(n-1, origine, auxiliaire, TH);
        //on effectue le deplacement
        deplacement++;
        deplacer(TH[origine], TH[destination],NDisque);
        //affichage intermediaire des tours
        AfficheLesTours(TH, NDisque);
```

```
51              //second appel recursif ou on deplace le disque du pilier
    ↪ intermediaire au pilier destiination
52              HanoiRecursifAffichage(n-1, auxiliaire, destination, TH);
53          }
54      }
55
56      /*
57      Afin de resoudre ce problème iterativement on applique les regles suivantes
    ↪ :
58      - déplacer le plus petit disque d'un emplacement à l'emplacement suivant (de
    ↪ TH[0] vers TH[1], de TH[1] vers TH[2], de TH[2] vers TH[0]) ;
59      - déplacer un autre disque ;
60      */
61      void HanoiIteratif(int n, int origine,int destination, Pilier TH[])
62      {
63          //on initialise le pilier d'origine ainsi que le pilier destination du
    ↪ premier deplacement
64          int op = 0, dp = 1;
65          while(TH[2].Pilier[0] == 0)
66          {
67              //on effectue le deplacement du plus petit disque
68              deplacer(TH[op], TH[dp],NDisque);
69              deplacement++;
70              //on determine les 2 piliers n'occupant pas le plus petit disque
71              Chemin ep = pilierInter(dp);
72              //si le pilier destination n'est pas deja remplie
73              if(TH[2].Pilier[0] == 0)
74              {
75                  //on recupere les disques se trouvant au sommet des 2 piliers
76                  int a = sommet(TH[ep.o],NDisque), b = sommet(TH[ep.d],NDisque);
77
78                  /*ensuite on compare lequel des 2 piliers possedent le disque le
    ↪ plus petit ce dernier sera par la suite
79                  deplacer vers le second, dans le cas ou un des piliers est vide
    ↪ on deplace le disque de l'autre pilier
80                  automatique*/
81                  if(a== NDisque)
82                  {
83                      deplacer(TH[ep.d], TH[ep.o],NDisque);
84                      deplacement++;
85                  }
86                  else if (b == NDisque)
87                  {
88                      deplacer(TH[ep.o], TH[ep.d],NDisque);
89                      deplacement++;
90                  }
91                  else if(TH[ep.o].Pilier[a] > TH[ep.d].Pilier[b])
```

```
92                  {
93                      deplacer(TH[ep.d], TH[ep.o],NDisque);
94                      deplacement++;
95                  }
96                  else
97                  {
98                      deplacer(TH[ep.o], TH[ep.d],NDisque);
99                      deplacement++;
100                 }
101             }
102             //on determine le pilier actuel du plus petit disque ainsi que sa
    ↪  destination
103             op = op == 2 ? 0 : op+1;
104             dp = dp == 2 ? 0 : dp+1;
105         }
106     }
107
108     //meme chose que la procedure precedente avec ajout des procedure
    ↪  d'affichage
109     void HanoiIteratifAffichage(int n, int origine,int destination, Pilier TH[])
110     {
111         int op = 0, dp = 1;
112         while(TH[2].Pilier[0] == 0)
113         {
114         deplacer(TH[op], TH[dp],NDisque);
115         AfficheLesTours(TH, NDisque);
116         deplacement++;
117         Chemin ep = pilierInter(dp);
118         if(TH[2].Pilier[0] == 0)
119         {
120             int a = sommet(TH[ep.o],NDisque), b = sommet(TH[ep.d],NDisque);
121
122             if(a == NDisque)
123             {
124             deplacer(TH[ep.d], TH[ep.o],NDisque);
125             deplacement++;
126             }
127             else if (b == NDisque)
128             {
129             deplacer(TH[ep.o], TH[ep.d],NDisque);
130             deplacement++;
131             }
132             else if(TH[ep.o].Pilier[a] > TH[ep.d].Pilier[b])
133             {
134             deplacer(TH[ep.d], TH[ep.o],NDisque);
135             deplacement++;
136             }
```

```
137             else
138             {
139             deplacer(TH[ep.o], TH[ep.d],NDisque);
140             deplacement++;
141             }
142             AfficheLesTours(TH, NDisque);
143         }
144         op = op == 2 ? 0 : op+1;
145         dp = dp == 2 ? 0 : dp+1;
146         }
147     }
148 
149     //fonction generant des deplacement aleatoire
150     vector<Dep> Aleatoire(int n, Pilier TH[])
151     {
152         // le nombre de deplacements a generer est 2^n - 1
153         vector<Dep> solution;
154         srand(time(NULL));
155         Dep dep;
156         int top_o, top_d;
157         bool dep_autorise;
158         for (int i = 0; i < pow(2, n) - 1; ++i)
159         {
160             dep_autorise = false;
161             //on genere un deplacement aleatoire valide
162             while (!dep_autorise)
163             {
164                 dep.o = rand() % 3;
165                 if ((top_o = sommet(TH[dep.o], n)) == n) // origine vide
166                     continue;
167                 dep.d = rand() % 3;
168                 if (dep.d == dep.o) // destination meme que origine
169                     continue;
170                 top_d = sommet(TH[dep.d], n);
171                 if (top_d == n || TH[dep.o].Pilier[top_o] <
    ↪ TH[dep.d].Pilier[top_d])
172                     dep_autorise = true;
173             }
174             //on l'ajoute a la liste de deplacement
175             solution.push_back(dep);
176 
177             cout << "on deplace le disque de taille "<<
    ↪ TH[dep.o].Pilier[sommet(TH[dep.o], n)]
178                  <<" du pilier "<< dep.o <<" au pilier "<< dep.d << endl;
179 
180             deplacer(TH[dep.o], TH[dep.d], n);
181         }
```

```
182         cout << "la solution contient " << solution.size() << " deplacements" <<
    ↪ endl;
183 
184         return solution;
185     }
186 
187     //verifiation de la solution aleatoire
188     bool Verification(int n, Pilier TH[])
189     {
190         for (int i = 0; i < n; ++i)
191         {
192             //si le pilier final ne contient pas tout les disque dans le bon
    ↪ ordre alors on retourne false
193             if (TH[2].Pilier[i] != i + 1)
194                 return false;
195         }
196         return true;
197     }
```

structure.hpp

```
1     #include <iostream>
2 
3     using namespace std;
4 
5     struct Pilier
6     {
7       int *Pilier;
8     };
9 
10    struct Chemin
11    {
12        int o,d;
13    };
14 
15    //initalisation du pilier
16    void init(Pilier TH[], int n)
17    {
18      //creation des 3 piliers
19      for (int i = 0; i < 3; ++i)
20        TH[i].Pilier = (int *)malloc(n * sizeof(int));
21 
22      // chaque pilier est representer par sa taille le plus petit est
    ↪ representer par l'indice 1 et le plus grand par l'indice n
23      // on represente l'abscence de pilier par l'indice 0
24      for (int i = 0; i < n; ++i)
```

```
25        {
26          TH[0].Pilier[i] = i+1; // premier pilier contient l'integralite des
    ↪ disque dans l'odre de leur taille variant de 1 a n
27          TH[1].Pilier[i] = 0; // pilier vide
28          TH[2].Pilier[i] = 0; // pilier vide
29        }
30      }
31
32      //on retourne le disque se trouvant au sommet du pilier
33      int sommet(Pilier p, int n)
34      {
35        int top;
36        //on parcourt le pilier jusqu'a atteindre le premier disque du tableau
37        for (top = 0; (top < n) && (p.Pilier[top] == 0); ++top);
38          return top;
39      }
40
41      //on deplace un disque d'un pilier a un autre
42      void deplacer(Pilier &origine, Pilier &destination, int n)
43      {
44        //on recupere le disque a deplacer
45        int top1 = sommet(origine, n);
46
47        if (top1 < n)
48        {
49          /*on verifie que le disque a deplacer n'est pas plus grand que le disque
    ↪ se rouvant au sommet du pilier destination
50          et que le pilier destination n'est pas autrement on retourne une
    ↪ erreur*/
51          int top2 = sommet(destination,n);
52          if ((top2 < n) && (destination.Pilier[top2] < origine.Pilier[top1]))
53          {
54            cerr << "ERREUR : on ne peut pas deplacer une disque de taille "
55                 << origine.Pilier[top1] << " sur un disque de taille "
56                 << destination.Pilier[top2] << " !" << endl;
57            return;
58          }
59
60          //on ajoute le disque au pilier destination et on l'enleve du pilier
    ↪ origine
61          destination.Pilier[top2-1] = origine.Pilier[top1];
62          origine.Pilier[top1] = 0;
63        }
64      }
65
66      //defini le pilier intermediaire
67      int autre(int p1, int p2)
```

```
68      {
69        return 3 - p1 -p2;
70      }
71
72      //retoune les 2 piliers ne contenant pas le plus petit disque
73      Chemin pilierInter(int a)
74      {
75        Chemin ep;
76        //la variable 'a' contient la position pilier contenant le plus petit
    ↪   disque on retourne par consequent les 2 autres piliers
77        switch(a)
78        {
79          case 1 :
80            ep.o = 0;
81            ep.d = 2;
82            break;
83          case 0 :
84            ep.o = 1;
85            ep.d = 2;
86            break;
87          case 2 :
88            ep.o = 0;
89            ep.d = 1;
90            break;
91        }
92
93        return ep;
94      }
```