RAPPORT DU TD 00 (INF 5059): ADVANCED ALGORITHMS AND DATA STRUCTURES

Nom : TEGOMO DYVANE DEGAR

Matricule : 20V2299

Lien Github : https://github.com/divane33/TD-00-INF-5059-ADVANCED-ALGORITHMS-AND-DATA-STRUCTURES

Exercice1 : Recherche Binaire (Binary Search)

Représentation du Problème :
Étant donné une liste triée d'entiers, l'objectif est de trouver l'indice d'une valeur cible (valeur recherchée) en utilisant une méthode de recherche efficace.

Solution :
J'implémente l'algorithme de recherche binaire qui divise l'espace de recherche en deux à chaque itération. L'algorithme vérifie l'élément central (du milieu du tableau ou d'un sous-tableau) et réduit la recherche en fonction de si la cible (valeur recherchée) est supérieure ou inférieure à cet élément.

Exemple d'Entrée/Sortie :

- Entrée (tableau trié) : [1, 3, 5, 7, 9], Cible ou Valeur à rechercher : 5
- Sortie (résultat): 2 (indice de la cible)

Analyse de la Complexité pour cet algorithme :

- **Recherche Binaire :** Complexité de **O(log n)**. Par exemple, pour un tableau de 16 éléments, la recherche binaire trouve l'élément cible en seulement 4 étapes maximum (car $\log_2(16) = 4$).
- **Recherche Linéaire :** Complexité de **O(n)**. Dans le même tableau de 16 éléments, une recherche linéaire peut nécessiter jusqu'à 16 comparaisons dans le pire des cas.

Conclusion :
La recherche binaire est nettement plus rapide pour des grands ensembles de données triés.

<u>Exercice2 :</u> Parcours de Graphe (BFS et DFS)

Représentation du Problème :
Dans une carte de ville modélisée comme un graphe non orienté, les nœuds représentent des emplacements et les arêtes représentent des chemins. L'objectif est d'explorer tous les nœuds et de déterminer la connectivité entre eux, c’est-à-dire de savoir si il est possible d’y créer un chemin.

Solution :

- **BFS (Recherche en Largeur) :** Ici, on explore les nœuds niveau par niveau en utilisant une file (queue).
- **DFS (Recherche en Profondeur) :** Ici, on explore les nœuds en profondeur avant de revenir en arrière. Ceci est implémenté à l'aide de la récursion.

Exemple d'Entrée/Sortie :

- **Graphe :** A-B, A-C, B-D, C-E
- **BFS à partir de A :** A B C D E
- **DFS à partir de A :** A B D C E

Analyse de la Complexité :

- **Complexité : O(V + E)**, où V est le nombre de sommets et E le nombre d'arêtes.
- Par exemple, dans un graphe avec 5 nœuds et 4 arêtes, BFS et DFS visitent chaque nœud et chaque arête exactement une fois.

Conclusion :
Le BFS est idéal pour trouver **le chemin le plus court,** tandis que le DFS est efficace pour la recherche de **chemins profonds.**

Exercice3 : Programmation Dynamique (Problème du Sac à Dos)

Représentation du Problème :
Étant donné des objets avec des valeurs et des poids, et une limite de poids maximale, l'objectif est de maximiser la valeur totale sans dépasser la limite de poids.

Solution :
On implémente l'algorithme du **sac à dos 0/1** en utilisant la programmation dynamique avec un **tableau 2D dp[i][w]**, où **i** représente **le nombre d'objets** considérés et **w** la **capacité de poids actuelle.**

Exemple d'Entrée/Sortie :

- **Objets :** (60,10), (100,20), (120,30), Capacité : 50
- **Sortie :** Valeur Maximale = 220 (objets (100,20) et (120,30))

Analyse de la Complexité :

- **Complexité : O(n * W)**, où n est le nombre d'objets et W la capacité du sac.
- Par exemple, pour 3 objets et une capacité de 50, l'algorithme remplit un tableau de 3 x 50 = 150 cases.

Conclusion :
La programmation dynamique optimise les performances en réduisant le nombre de calculs redondants c’est-à-dire qui se répètent.

Exercice4 : Fusion d'Intervalles (Merge Intervals)

Représentation du Problème :
Dans une application de calendrier, il est nécessaire de fusionner des intervalles de temps qui se chevauchent afin de simplifier la planification.

Solution :
L'algorithme trie les intervalles par heure de début, puis fusionne de manière itérative les intervalles qui se chevauchent (qui se confondent).

Exemple d'Entrée/Sortie :

- **Entrée des différents intervals:** [(1,3), (2,6), (8,10), (9,12)]
- **Sortie des intervals fusionnés:** [(1,6), (8,12)]

Analyse de la Complexité :

- **Complexité : O(n log n)** pour le tri des intervalles, puis **O(n)** pour la fusion.

- Par exemple, pour 4 intervalles, le tri prend environ $\log_2(4)$ = **2 comparaisons par étape.**

Conclusion :
Nous pouvons voir à ce niveau que l'efficacité de l'algorithme dépend principalement de l'étape de tri.

<u>Exercice5 :</u> Somme Maximale de Sous-Tableau (Algorithme de Kadane)

Représentation du Problème :
Étant donné un tableau d'entiers, l'objectif est de trouver le sous-tableau contigu avec la somme maximale.

Solution :
L'algorithme de Kadane suit la somme courante du sous-tableau et la réinitialise si elle devient négative, garantissant ainsi une solution optimale.

Exemple d'Entrée/Sortie :

- **Tableau en Entrée :** [-2, 1, -3, 4, -1, 2, 1, -5, 4]
- **Sous-Tableau en Sortie :** 6 (sous-tableau [4, -1, 2, 1])

Analyse de la Complexité :

- **Complexité : O(n)** car chaque élément est traité une seule fois.
- Par exemple, pour un tableau de 8 éléments, l'algorithme effectue exactement 8 comparaisons.

Conclusion :
En conclusion, nous pouvons constater que l'algorithme de Kadane est **extrêmement efficace** pour des tableaux de grande taille.