LEUNA FIENKAK NKEHEUP - 20U2698

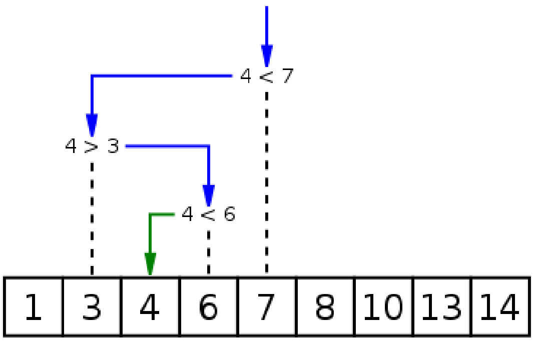
Lien Github

PARTIE 1

Exercise 1: Binary Search

Représentation des problèmes

L'algorithme de recherche binaire est conçu pour trouver efficacement une valeur cible dans une liste triée d'entiers. Au lieu d'effectuer une recherche séquentielle comme dans une recherche linéaire, la recherche binaire divise la liste en deux et détermine quelle moitié peut contenir la cible, ce qui réduit considérablement l'espace de recherche à chaque étape.



Exemple de représentation :

Considérons le tableau trié suivant :
1,3,5,7,9,11,13,15,17,19

Si nous recherchons **7**, la recherche binaire procédera comme suit :

1. Vérifier l'élément du milieu (index **4**, valeur **9**).
2. Comme **7 < 9**, rechercher dans la moitié gauche.
3. Le nouvel élément du milieu est à l'index **2** (valeur **5**).
4. Comme **7 > 5**, rechercher dans la moitié droite de ce sous-tableau.
5. Trouver **7** à l'index **3**.

Solution

La recherche binaire suit l'approche « diviser pour régner », en réduisant l'espace de recherche de moitié à chaque étape.

```python
def binary_search(arr, target):
    left, right = 0, len(arr) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1
```

Résultats

Input & Output:

Input List	Target	Output (Index)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]	7	3
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]	10	-1

Analyse de la complexité en temps

- **Recherche binaire** : La complexité en temps est **O(log n)**, car à chaque étape, la taille du problème est réduite de moitié.
- **Recherche linéaire (comparaison)** : La complexité en temps est **O(n)**, car chaque élément est vérifié un par un.

Comparaison:

Algorithm	Best Case	Worst Case
Linear Search	O(1)	O(n)
Binary Search	O(1)	O(log n)

Conclusion

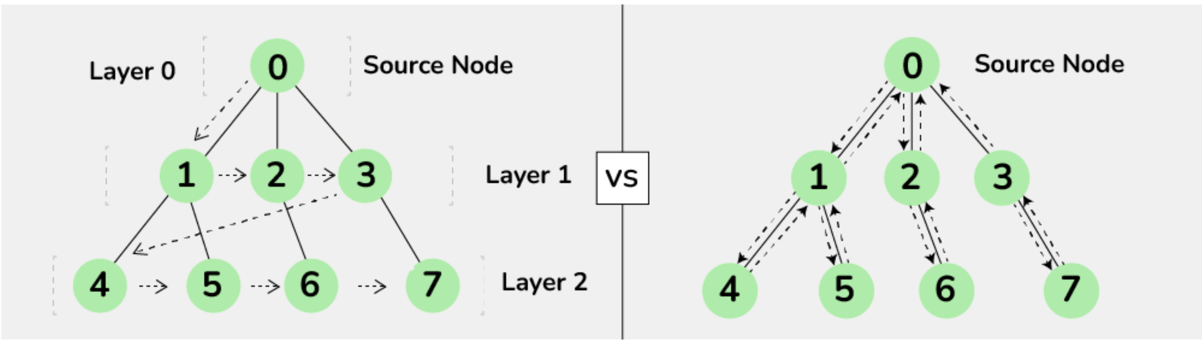
La recherche binaire est un algorithme de recherche très efficace pour les listes triées, qui surpasse largement la recherche linéaire dans les grands ensembles de données. Elle s'exécute en temps logarithmique O(log n), ce qui optimise les opérations de recherche.

Exercise 2: Graph Traversal (BFS and DFS

Problème

Nous avons un graphe non orienté représentant une carte de ville, avec des nœuds pour les emplacements et des arêtes pour les routes. L'objectif est d'explorer le graphe avec BFS et DFS, de vérifier la connectivité entre deux emplacements et de trouver le chemin le plus court entre eux. **Graphe a**

BFS **DFS**



Solution

- **BFS** explore le graphe niveau par niveau à l'aide d'une file (queue).
- **DFS** explore en profondeur à l'aide d'une récursion.
- **Vérification de connectivité** en vérifiant si le nœud cible est atteint en BFS.
- **Chemin le plus court** en utilisant BFS pour reconstruire le chemin optimal.

```python
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])
    traversal_order = []

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            traversal_order.append(node)
            queue.extend(graph[node] - visited)

    return traversal_order

def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    traversal_order = [start]

    for neighbor in graph[start] - visited:
        traversal_order.extend(dfs(graph, neighbor, visited))

    return traversal_order

def is_connected(graph, node1, node2):
    return node2 in bfs(graph, node1)

def shortest_path_bfs(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()

    while queue:
        node, path = queue.popleft()
        if node == goal:
            return path

        if node not in visited:
            visited.add(node)
            for neighbor in graph[node] - visited:
                queue.append((neighbor, path + [neighbor]))

    return None  # No path found
```

Résultats

Exécution avec le graphe donné :

```
BFS Traversal: ['0', '2', '1', '3', '6', '4', '5', '7']
DFS Traversal: ['0', '2', '6', '1', '4', '5', '3', '7']
Is 0 connected to 5?: True
Shortest path from 0 to 5: ['0', '1', '5']
Is 3 connected to 0?: True
```

Conclusion

BFS est efficace pour trouver le chemin le plus court (O(V + E)), tandis que DFS est utile pour explorer les connexions. L'algorithme est bien adapté à la navigation urbaine et peut être étendu à des graphes plus complexes.

Exercise 3: Dynamic Programming (Knapsack Problem)

Problème

Une entreprise d'emballage veut optimiser le rangement des produits dans des conteneurs afin de maximiser la valeur totale tout en respectant une limite de poids. Chaque produit a une valeur et un poids, et nous devons choisir les produits à inclure pour obtenir la valeur maximale possible.

Solution

- **Représentation** : Les articles sont représentés sous forme de tuples (valeur, poids).
- **Approche dynamique** :
  - Utilisation d'une table `dp[i][w]` où `i` est le nombre d'articles considérés et `w` est la limite de poids actuelle.
  - `dp[i][w]` stocke la valeur maximale pouvant être obtenue avec les `i` premiers objets et un poids maximal `w`.
  - On remplit la table en comparant l'inclusion ou l'exclusion d'un article.
- **Retourne** la valeur maximale et la liste des articles sélectionnés.

```python
def knapsack(items, max_weight):
    n = len(items)
    dp = [[0] * (max_weight + 1) for _ in range(n + 1)]

    for i in range(1, n + 1):
        value, weight = items[i - 1]
        for w in range(max_weight + 1):
            if weight <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weight] + value)
            else:
                dp[i][w] = dp[i - 1][w]

    # Backtracking to find the selected items
    w = max_weight
    selected_items = []
    for i in range(n, 0, -1):
        if dp[i][w] != dp[i - 1][w]:
            selected_items.append(items[i - 1])
            w -= items[i - 1][1]

    return dp[n][max_weight], selected_items[::-1]
```

Résultats

Exécution avec les articles `[(60, 10), (100, 20), (120, 30)]` et une capacité de 50 :

```
Maximum value: 220

Selected items: [(100, 20), (120, 30)]
```

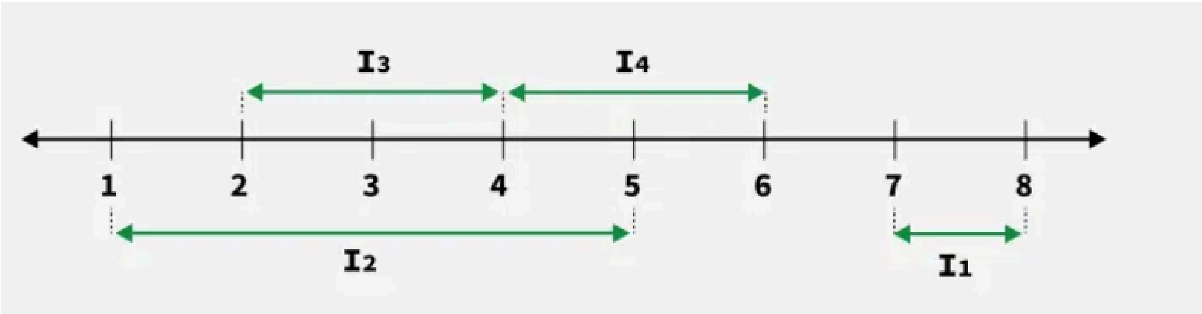
Conclusion

L'algorithme permet d'optimiser efficacement l'emballage avec une complexité de `O(n * W)`, ce qui est optimal pour un problème de sac à dos 0/1. Il offre une solution précise pour la maximisation des profits en tenant compte des contraintes de poids.

Exercise 4: Merge Intervals

Problème

Dans une application de calendrier, il est nécessaire de fusionner les plages horaires qui se chevauchent afin d'optimiser la planification. **Figure 1**



Solution

- **Représentation** : Les intervalles sont donnés sous forme de tuples `(start_time, end_time)`.
- **Approche** :
  - Trier les intervalles par heure de début.
  - Parcourir les intervalles triés et les fusionner s'ils se chevauchent.
- **Retourne** la liste des intervalles fusionnés.

```
def merge_intervals(intervals):
    if not intervals:
        return []

    intervals.sort(key=lambda x: x[0])
    merged = [intervals[0]]

    for current in intervals[1:]:
        last_merged = merged[-1]
        if current[0] <= last_merged[1]:  # Overlapping intervals
            merged[-1] = (last_merged[0], max(last_merged[1], current[1]))
        else:
            merged.append(current)

    return merged
```

Résultats

Exécution avec les intervalles de la **Figure 1**:

`(7, 8), (1, 5), (2, 4), (4, 6):`

`Merged intervals: [(1, 6), (7, 8)]`

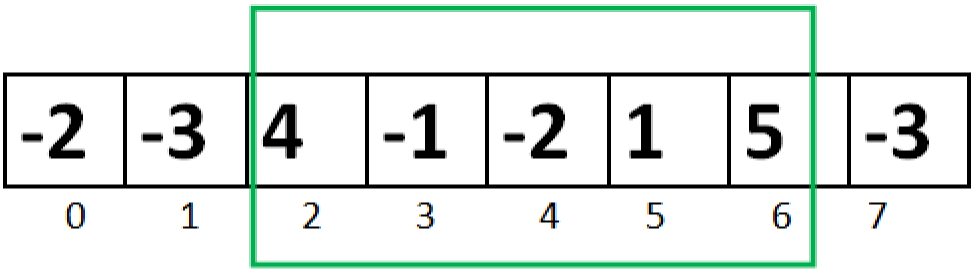
Conclusion

L'algorithme optimise efficacement les plages horaires avec une complexité de `O(n log n)`, due au tri initial. Il est idéal pour des applications nécessitant la gestion de créneaux horaires, comme les calendriers et les plannings.

Exercise 5: Maximum Subarray Sum (Kadane's Algorithm)

Problème

Dans l'analyse des prix des actions, nous devons identifier la séquence continue avec le profit maximal.



Solution

- **Représentation** : Un tableau d'entiers représentant les variations de prix.
- **Approche (Algorithme de Kadane)** :
    - Parcourir le tableau tout en maintenant une somme courante.
    - Réinitialiser la somme si elle devient négative.
    - Suivre les indices du sous-tableau optimal.
- **Retourne** la somme maximale et le sous-tableau correspondant.

```python
def max_subarray_sum(arr):
    if not arr:
        return 0, []

    max_sum = float('-inf')
    current_sum = 0
    start = end = s = 0

    for i in range(len(arr)):
        current_sum += arr[i]

        if current_sum > max_sum:
            max_sum = current_sum
            start = s
            end = i

        if current_sum < 0:
            current_sum = 0
            s = i + 1

    return max_sum, arr[start:end+1]
```

Résultats

Exécution avec `[-2, 1, -3, 4, -1, 2, 1, -5, 4]` :

```
Maximum Subarray Sum: 6
Subarray: [4, -1, 2, 1]
```

Conclusion

L'algorithme de Kadane trouve efficacement le sous-tableau optimal en $O(n)$, bien plus rapide qu'une approche brute-force $O(n^2)$.

PARTIE 2

Enoncez et expliquez les principes utilisés pour les opérations suivantes suivants:

Sélection

- **Roulette Wheel Selection:** Est une méthode de sélection qui représente les chromosomes/individu sur une roue en fonction de leur pourcentage par rapport à leur fitness value. Dans ce processus, La roue possède un pointeur initialement à 0, elle subit ensuite un mouvement de rotation aléatoire et une fois le mouvement terminé, le chromosome x que le pointeur indique est sélectionné pour la suite (un individu avec une grande proportion a plus de chance d'être selectionné)

  l'algorithme est le suivant:

  - Calcule la valeur fitness de chaque individu dans la population.
  - Pour chaque individu, calcule sa probabilité d'être choisi, en divisant son aptitude par la somme des aptitudes de toute la population. (en fonction de si on veut minimiser ou maximiser)
  - Calcule les probabilités cumulées pour chaque individu (en ajoutant les probabilités de tous les individus précédents).
  - Tire un nombre aléatoire entre 0 et 1 (le pointeur)
  - Si ce nombre est plus petit que la première probabilité cumulée, choisis le premier individu. Sinon, choisis l'individu dont la probabilité cumulée est juste au-dessus du nombre tiré.
  - Répète les étapes 4 et 5 autant de fois qu'il y a d'individus pour remplir le groupe de sélection.

- **Boltzmann Selection:** C'est une méthode de sélection qui permet de donner une chance à tous les individus d'être choisis, même ceux qui ne sont pas les meilleurs. Une sélection contrôlée par une température T.

  l'algorithme est le suivant:

  - Au début, on veut explorer un maximum (même les solutions pas très bonnes).

- Avec le temps, on veut exploiter les bonnes solutions (ne choisir que les meilleurs).
- Donc, on utilise une température T(qui diminue petit à petit) pour contrôler la sélection (et rechercher une certaine équilibre entre l'exploration et l'exploitation)
- Plus T est grande, plus tous les individus ont une chance.
- Plus T est petite, plus seuls les meilleurs sont choisis.

Mutation

- **Réinitialisation aléatoire:** On prend une caractéristique dans un chromosome donné et on remplace sa valeur par une nouvelle valeur prise au hasard sans tenir compte de la valeur de départ ou de l'ordre.

  l'algorithme est le suivant:

  - Choisir un gène au hasard dans un individu.
  - Générer une nouvelle valeur aléatoire dans le domaine autorisé.
  - Remplacer l'ancienne valeur par la nouvelle.

- **Mutation par inversion:** Elle consiste à prendre une portion du chromosome et à inverser l'ordre des éléments dans cette portion. Elle peut être utilisé lorsque l'ordre des chromosomes est prise en compte (le cas du voyageur par exemple) on garde les même valeurs, mais dans un ordre different.

  l'algorithm est le suivant:

  - On choisit deux positions aléatoires dans le chromosome
  - Inverser les éléments entre ces deux positions.
  - On reforme le chromosome avec cette portion inversée.

- **Mutation par brouillage:** Elle consiste à choisir une portion du chromosome, puis à mélanger aléatoirement les gènes de cette portion (au lieu de simplement les inverser comme dans la mutation par inversion).

  L'algorithme est le suivant:

- On choisit deux indices au hasard dans le chromosome (début et fin d'une portion).
- Extraire cette portion.
- Mélanger aléatoirement les éléments de la portion.
- Réinsérer la portion mélangée dans le chromosome (A partir de l'indice du début choisit a la 1ere etape)