CHEDJOUN KENGUEP Dave
Matricule: 20U2757
Niveau: Master 2
Spécialité: SIGL
Url Git of the project: https://github.com/DavePhil/TD_00_SBSE.git

TD 00: Algorithms for problem solving

Exercice 1

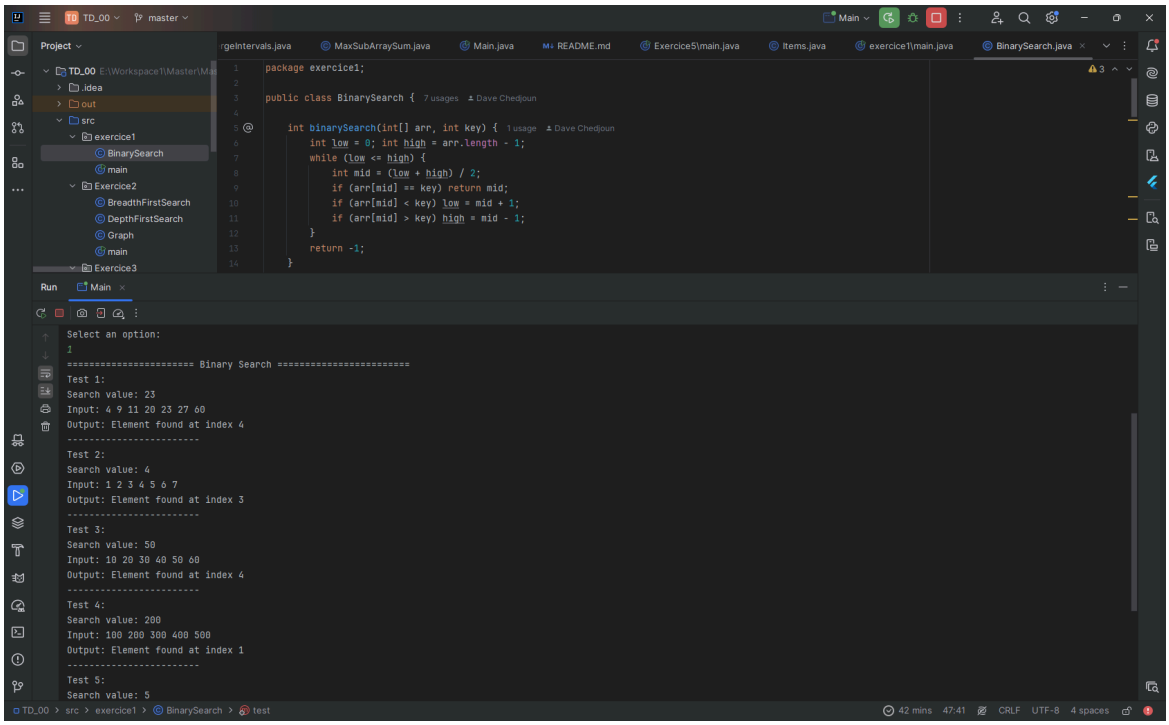
La recherche binaire suit une approche "diviser pour régner" en réduisant l'espace de recherche de moitié à chaque itération.

Input: int arr[] = {4, 9, 11, 20, 23, 27, 60}; int x = 23;

Output: The element found at index 4

Solution & Résultats

1. Comparer l'élément cible avec l'élément au milieu du tableau.
2. Si c'est une correspondance, retourner l'index.
3. Sinon, si la cible est plus petite, rechercher dans la moitié gauche.
4. Si la cible est plus grande, rechercher dans la moitié droite.
5. Répéter jusqu'à ce que la cible soit trouvée ou que l'espace de recherche devienne vide.



Conclusion

Complexité temporelle : Dans le meilleur cas on a : O(1) (si l'élément est au centre) et dans le cas moyen/pire on a : O(log n)

Exercice 2

BFS (Parcours en largeur) : Explore tous les voisins d'un nœud avant de passer aux suivants.

DFS (Parcours en profondeur) : Explore un chemin en profondeur avant de revenir en arrière.

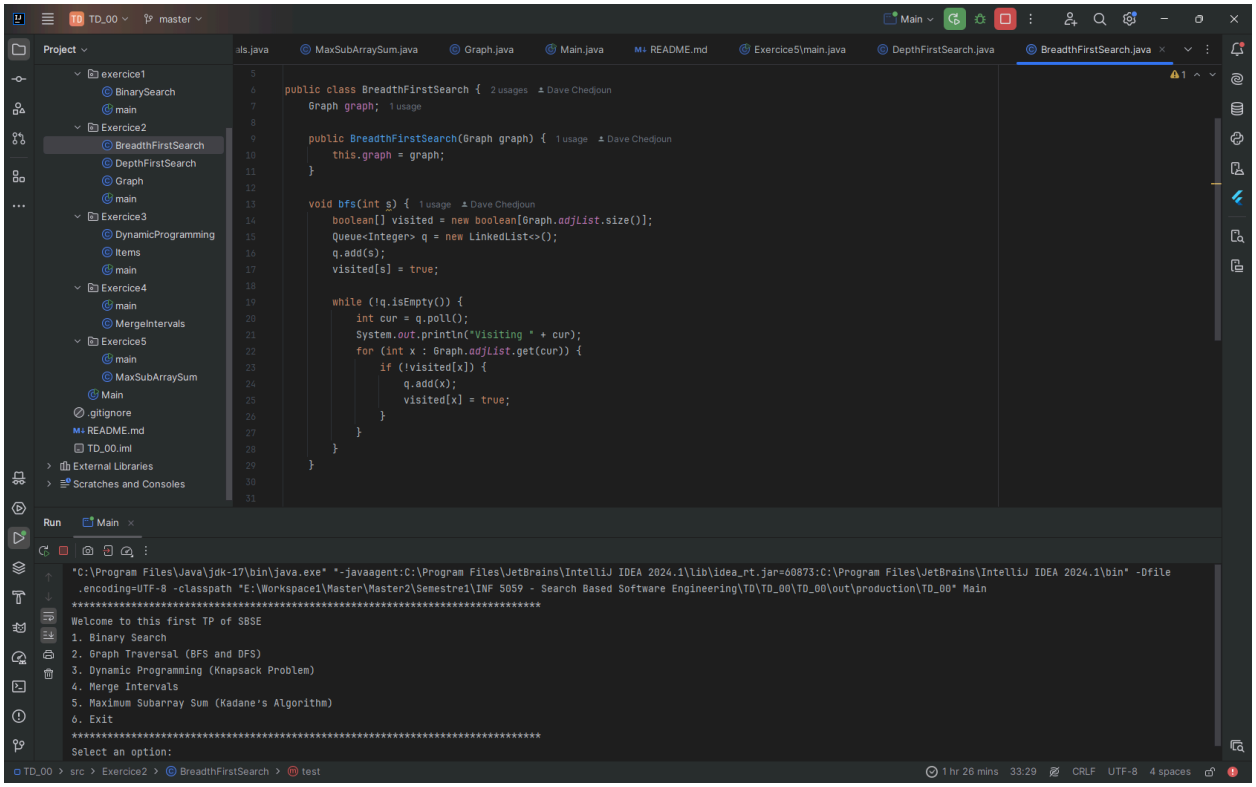
Input: adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]];

Output: 1 2 0 3 4

Solution & Résultats

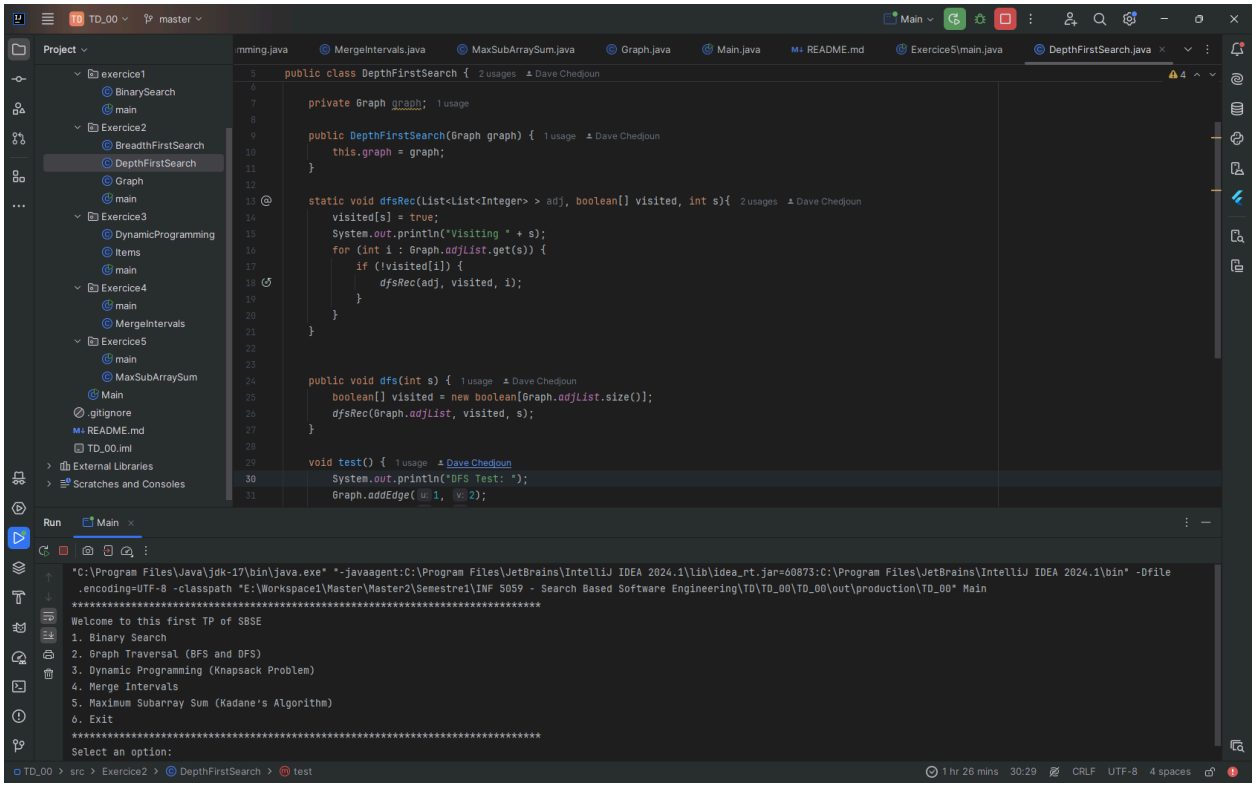
BFS (Breadth-First Search - Parcours en largeur)

- Utiliser une file (queue) pour stocker les nœuds à visiter.
- Marquer le nœud comme visité et l'ajouter à la file.
- Tant que la file n'est pas vide :
- Défilez un nœud et affichez-le.
- Ajoutez tous ses voisins non visités à la file.



DFS (Depth-First Search - Parcours en profondeur)

- Utiliser une pile (stack) ou récursion pour parcourir le graphe.
- Commencer par un sommet et marquer tous les sommets visités.
- Explorer récursivement tous les voisins non visités.



Shortest Path (Plus court chemin)

Initialisation

- Vérifie si `start` et `end` sont les mêmes : dans ce cas, le chemin est simplement `[start]`.
- Initialise un tableau `visited[]` pour éviter les cycles.
- Utilise une **file (queue)** pour le parcours BFS.
- Un `Map<Integer, Integer>` (`parent`) est utilisé pour **mémoriser le parent** de chaque nœud, afin de reconstruire le chemin plus tard.

Parcours BFS

- On commence par `start`, qu'on marque comme visité et on l'ajoute à la file.
- Tant que la file n'est pas vide :
    - Extraire un sommet `cur` de la file.

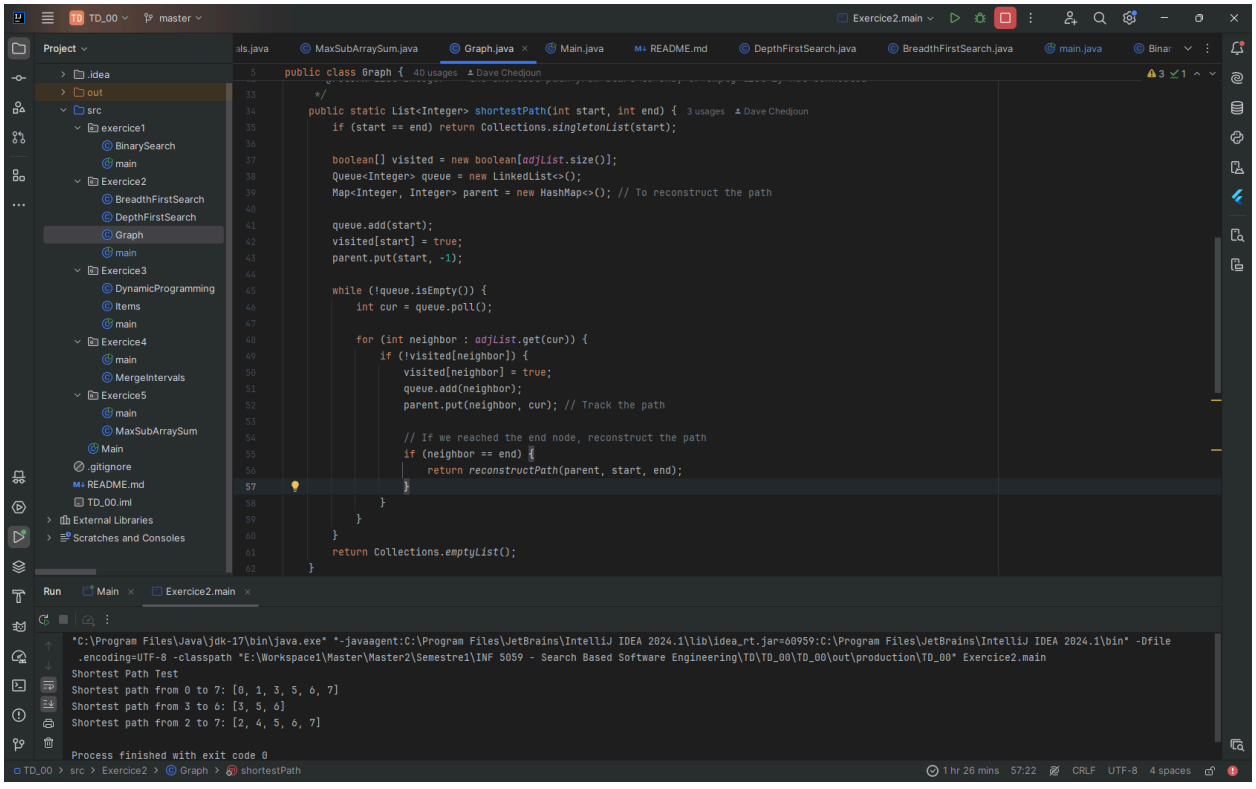
- Parcourir tous ses voisins (`neighbor`).
- Si un voisin `neighbor` n'a pas été visité :
  - Le marquer comme visité.
  - L'ajouter à la file.
  - Enregistrer `cur` comme son parent.
  - Si `neighbor == end`, on arrête et on **reconstruit le chemin**.

Reconstruction du chemin

- Si `end` est atteint, une fonction auxiliaire `reconstructPath` est appelée pour retrouver le chemin en remontant via `parent`.

Retour du résultat

- Si un chemin est trouvé, on le retourne sous forme de liste.
- Sinon, la fonction renvoie une liste vide (`Collections.emptyList()`).



Conclusion

Complexité : `O(V + E)`, où `V` est le nombre de sommets et `E` le nombre d'arêtes.

Exercice 3

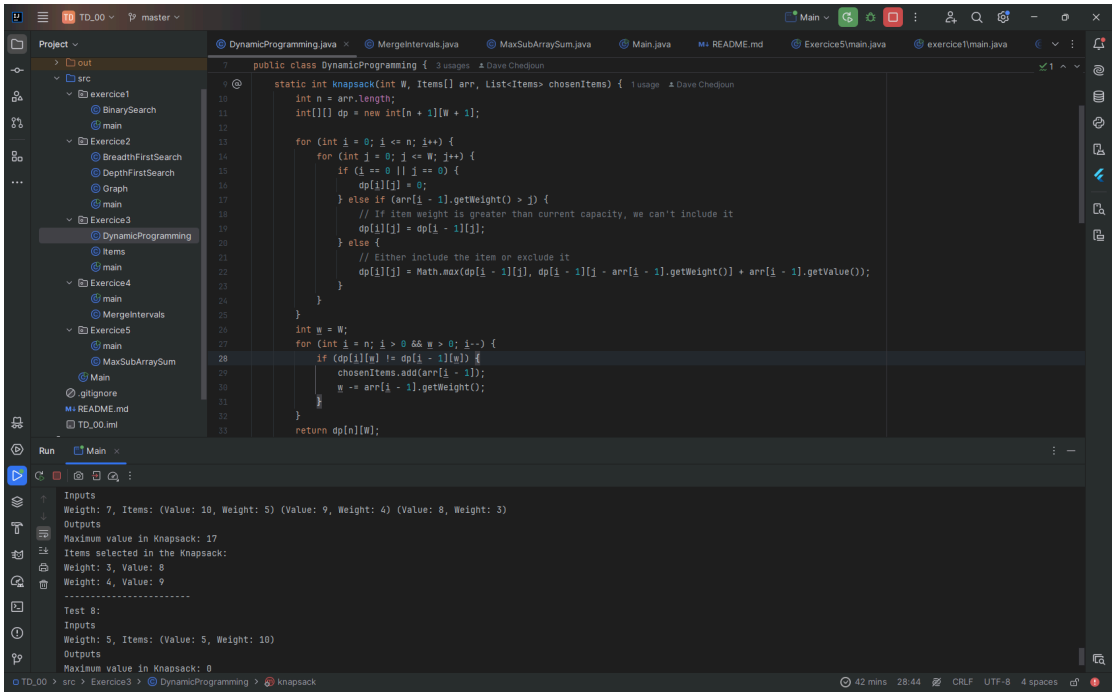
Nous avons un sac à dos de capacité W. Nous avons N objets, chacun avec un poids weight[i] et une valeur value[i]. Objectif : maximiser la somme des valeurs sans dépasser W.

Input : Capacité W = 5 Items = [(Valeur: 2, Poids: 3), (Valeur: 3, Poids: 4), (Valeur: 4, Poids: 5), (Valeur: 5, Poids: 6)]

Output : Valeur maximale dans le sac à dos : 3 Objets sélectionnés : [(Valeur: 3, Poids: 4)]

Solution & Résultats

- Définir `dp[i][w]` comme la valeur maximale pouvant être obtenue avec `i` objets et une capacité `w`.
- Cas de base : `dp[0][w] = 0` (pas d'objet, donc valeur nulle).
- Pour chaque objet `i` :
    - ❖ Si `weight[i] > w`, ne pas inclure l'objet : `dp[i][w] = dp[i-1][w]`.
    - ❖ Sinon, choisir entre inclure ou non l'objet : `dp[i][w] = max(dp[i-1][w], value[i] + dp[i-1][w - weight[i]])`
- La réponse est donnée par `dp[N][W]`.



Conclusion

Complexité : `O(N * W)`, où `N` est le nombre d'objets et `W` la capacité du sac.

Exercice 4

Description

On nous donne `N` intervalles sous forme `(début, fin)`. Si deux intervalles `[a, b]` et `[c, d]` se chevauchent (`b >= c`), on les fusionne en `[a, d]`.

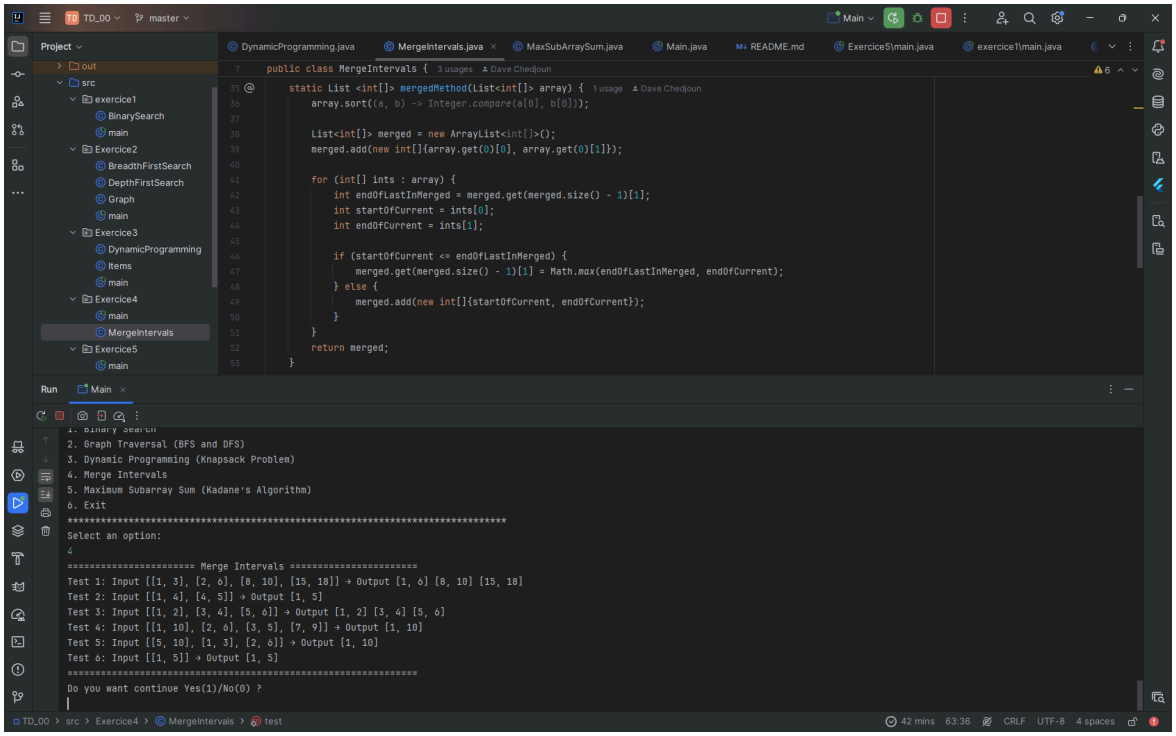
Input: `Intervalles = [(1, 3), (2, 6), (8, 10), (15, 18)]`

Output: `Intervalles fusionnés = [(1, 6), (8, 10), (15, 18)]`

Solution & Résultats

1. Trier les intervalles par ordre croissant de début.
2. Initialiser une liste de résultat avec le premier intervalle.
3. Pour chaque intervalle suivant :

- S'il chevauche le dernier intervalle fusionné, fusionner (`max(fin1, fin2)`).
- Sinon, ajouter un nouvel intervalle.



Conclusion

Complexité : `O(N log N)` pour le tri + `O(N)` pour la fusion ➜ **O(N log N)**.

Exercice 5

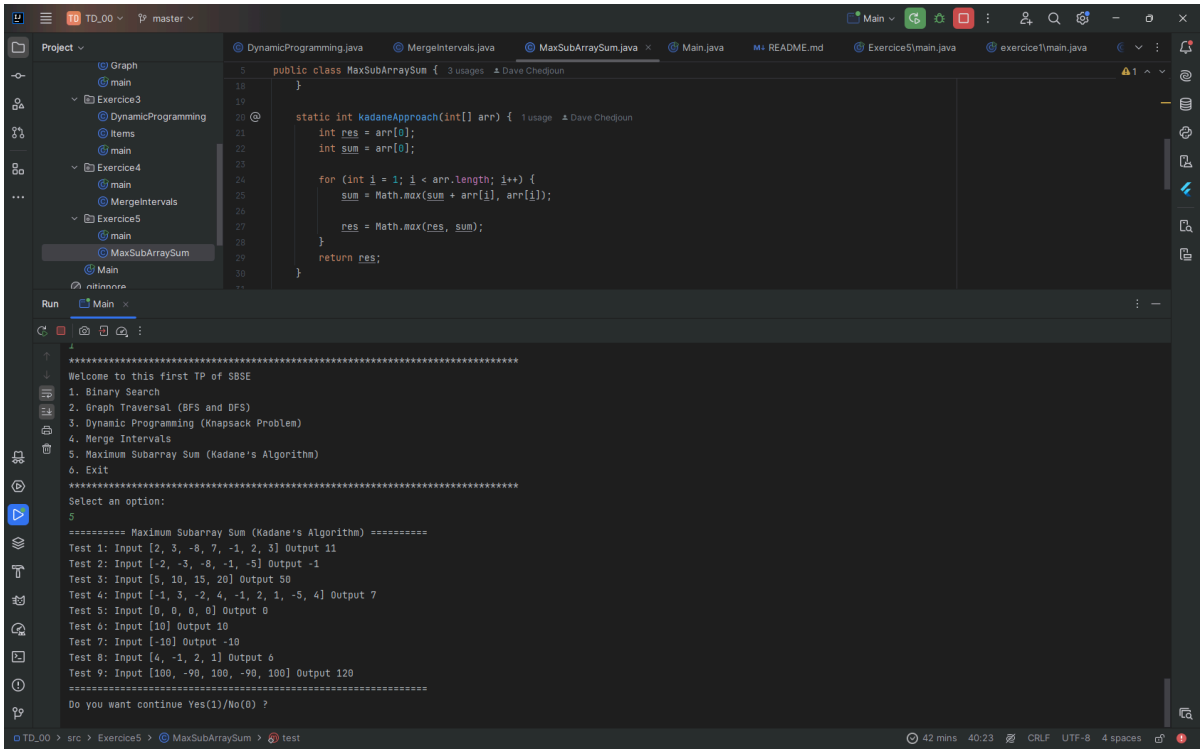
On nous donne un tableau d'entiers (positifs, négatifs, mixtes). Trouver la **sous-séquence contiguë** dont la somme est maximale.

Input: Tableau = [-2, 1, -3, 4, -1, 2, 1, -5, 4]

Output: Somme maximale = 6

Solution & Résultats

- Initialiser `max_global = nums[0]` et `max_courant = nums[0]`.
- Parcourir le tableau et pour chaque élément `nums[i]`
    - max_courant = max(nums[i], max_courant + nums[i])
    - max_global = max(max_global, max_courant)



Conclusion

Complexité : `O(N)`, où `N` est la taille du tableau.