

CHEDJOUN KENGUEP Dave

Matricule: 20U2757

Niveau: Master 2

Spécialité: SIGL

TD 00: Algorithms for problem solving

Exercice 1

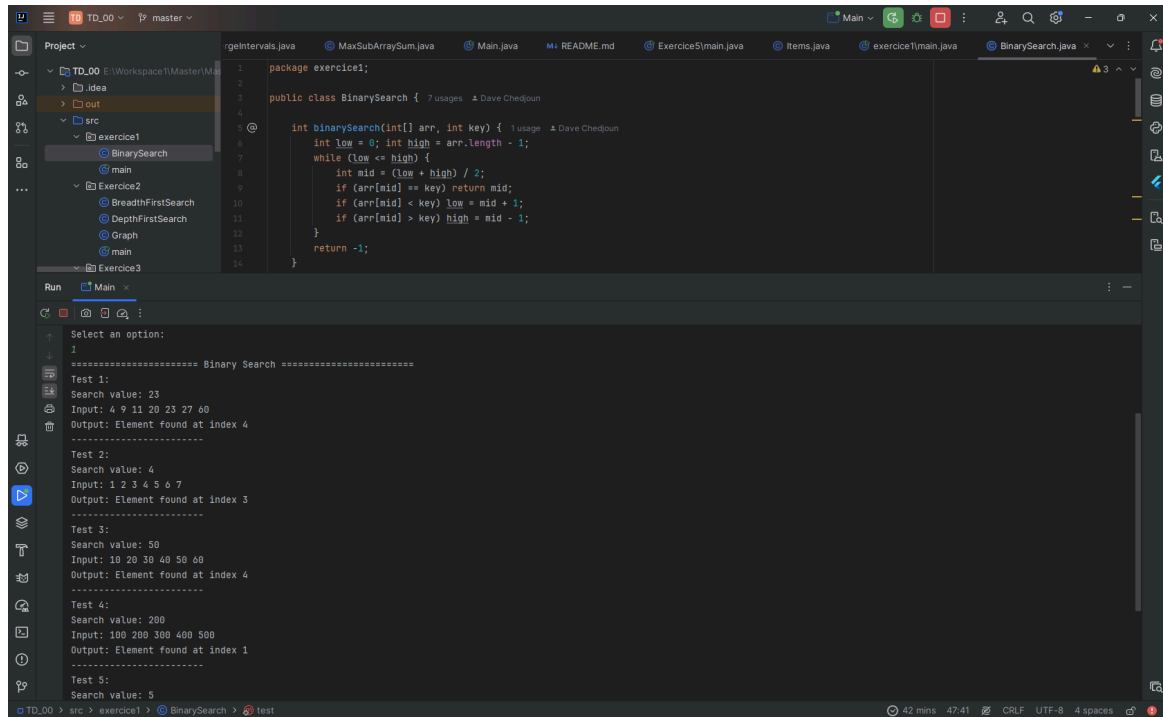
La recherche binaire suit une approche "diviser pour régner" en réduisant l'espace de recherche de moitié à chaque itération.

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.



```
package exercise1;

public class BinarySearch {
    int binarySearch(int[] arr, int key) {
        int low = 0; int high = arr.length - 1;
        while (low <= high) {
            int mid = (low + high) / 2;
            if (arr[mid] == key) return mid;
            if (arr[mid] < key) low = mid + 1;
            if (arr[mid] > key) high = mid - 1;
        }
        return -1;
    }
}
```

Run Main

Select an option:

1

***** Binary Search *****

Test 1:

Search value: 23

Input: 4 9 11 20 23 27 60

Output: Element found at index 4

Test 2:

Search value: 4

Input: 1 2 3 4 5 6 7

Output: Element found at index 3

Test 3:

Search value: 50

Input: 10 20 30 40 50 60

Output: Element found at index 4

Test 4:

Search value: 200

Input: 100 200 300 400 500

Output: Element found at index 1

Test 5:

Search value: 5

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.

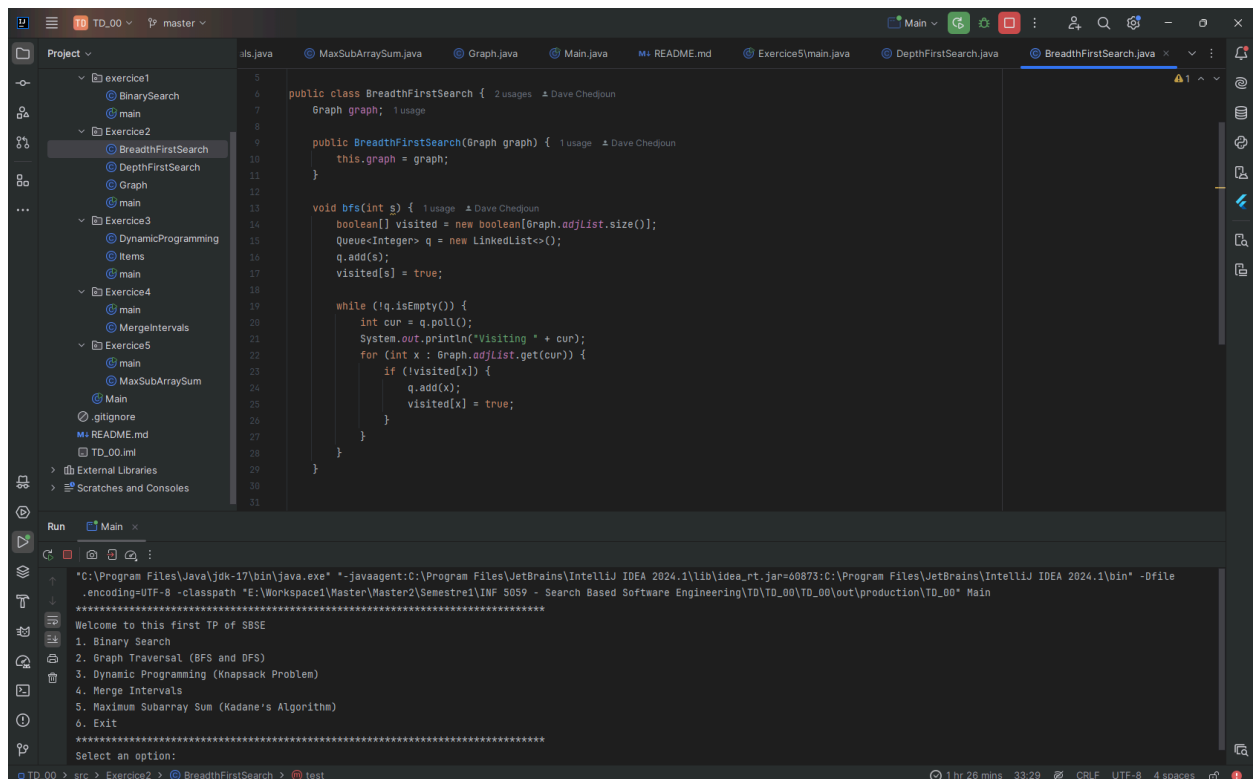
Input: `adj = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]];`

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.



```
public class BreadthFirstSearch {
    Graph graph;

    public BreadthFirstSearch(Graph graph) {
        this.graph = graph;
    }

    void bfs(int s) {
        boolean[] visited = new boolean[graph.adjList.size()];
        Queue<Integer> q = new LinkedList<>();
        q.add(s);
        visited[s] = true;

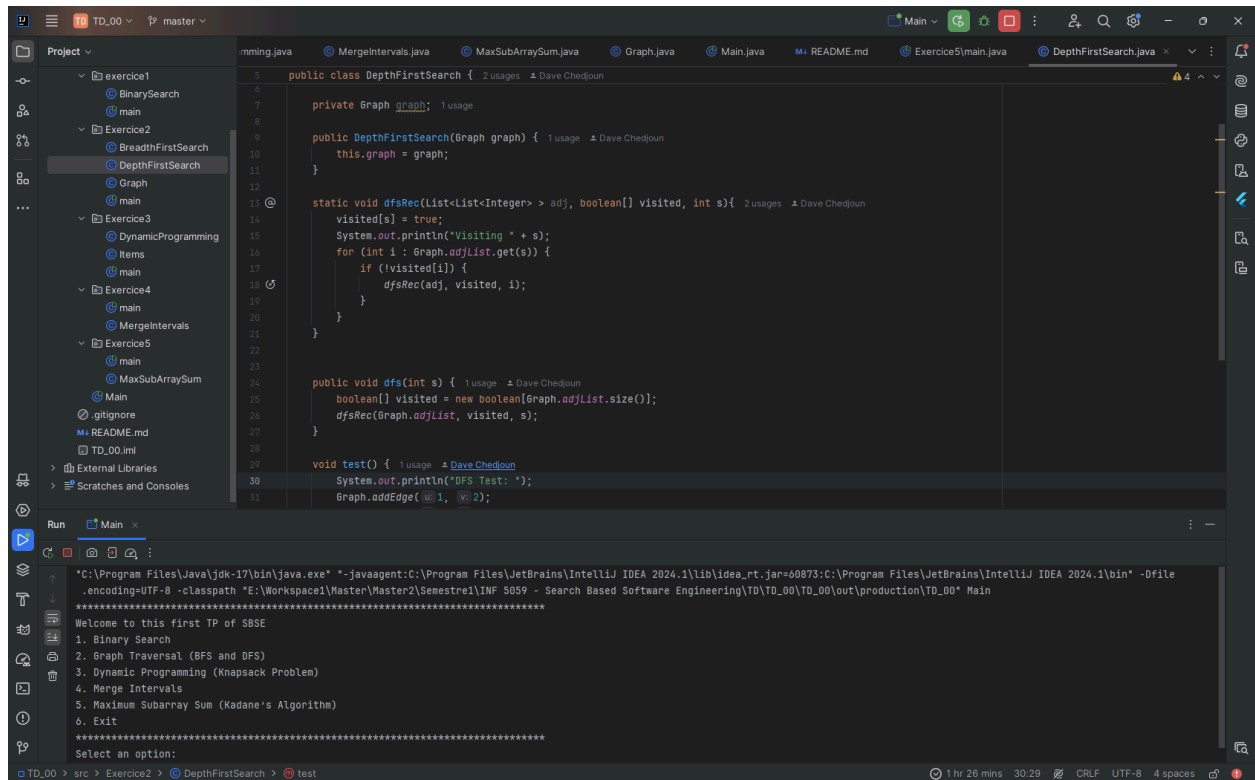
        while (!q.isEmpty()) {
            int cur = q.poll();
            System.out.println("visiting " + cur);
            for (int x : graph.adjList.get(cur)) {
                if (!visited[x]) {
                    q.add(x);
                    visited[x] = true;
                }
            }
        }
    }
}
```

Run Main

```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\lib\idea_rt.jar=60873:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin" -Dfile.encoding=UTF-8 -classpath "E:\Workspace1\Master\Master2\Semestre1\INF 5059 - Search Based Software Engineering\TD\TD_00\out\production\TD_00" Main
Welcome to this first TP of SBSE
1. Binary Search
2. Graph Traversal (BFS and DFS)
3. Dynamic Programming (Knapsack Problem)
4. Merge Intervals
5. Maximum Subarray Sum (Kadane's Algorithm)
6. Exit
Select an option:
```

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.



The screenshot shows the IntelliJ IDEA IDE. The left sidebar displays a project structure with folders for 'Exercise1' through 'Exercise5', each containing a 'main' file. The 'DepthFirstSearch' class is selected under 'Exercise2'. The main editor window shows the code for 'DepthFirstSearch.java'. The code includes a private 'Graph' field, a constructor, a recursive 'dfsRec' method, and a 'dfs' method that initializes a 'visited' array and calls 'dfsRec'. A 'test' method is also present. The bottom panel shows the 'Run' console with the command 'java.exe' and the output 'DFS Test: ' followed by a list of options: 1. Binary Search, 2. Graph Traversal (BFS and DFS), 3. Dynamic Programming (Knapsack Problem), 4. Merge Intervals, 5. Maximum Subarray Sum (Kadane's Algorithm), 6. Exit. The status bar at the bottom indicates '1 hr 26 mins' and '30:29'.

```
5 public class DepthFirstSearch {
6     private Graph graph;
7
8     public DepthFirstSearch(Graph graph) {
9         this.graph = graph;
10    }
11
12    static void dfsRec(List<List<Integer>> adj, boolean[] visited, int s){
13        visited[s] = true;
14        System.out.println("Visiting " + s);
15        for (int i : graph.adjList.get(s)) {
16            if (!visited[i]) {
17                dfsRec(adj, visited, i);
18            }
19        }
20    }
21
22    public void dfs(int s) {
23        boolean[] visited = new boolean[graph.adjList.size()];
24        dfsRec(graph.adjList, visited, s);
25    }
26
27    void test() {
28        System.out.println("DFS Test: ");
29        graph.addEdge(0, 1, 2);
30    }
31 }
```

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()**).

```

public class Graph {
    // ...
    public static List<Integer> shortestPath(int start, int end) {
        if (start == end) return Collections.singletonList(start);

        boolean[] visited = new boolean[adjList.size()];
        Queue<Integer> queue = new LinkedList<>();
        Map<Integer, Integer> parent = new HashMap<>(); // To reconstruct the path

        queue.add(start);
        visited[start] = true;
        parent.put(start, -1);

        while (!queue.isEmpty()) {
            int cur = queue.poll();

            for (int neighbor : adjList.get(cur)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                    parent.put(neighbor, cur); // Track the path

                    // If we reached the end node, reconstruct the path
                    if (neighbor == end) {
                        return reconstructPath(parent, start, end);
                    }
                }
            }
        }

        return Collections.emptyList();
    }
}

```

Run Main x Exercice2.main x

```

"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\lib\idea_rt.jar=60959:C:\Program Files\JetBrains\IntelliJ IDEA 2024.1\bin" -Dfile.encoding=UTF-8 -classpath "E:\Workspace1\Master\Semestre1\INF 5059 - Search Based Software Engineering\TD\TD_00\out\production\TD_00" Exercice2.main
Shortest Path Test
Shortest path from 0 to 7: [0, 1, 3, 5, 6, 7]
Shortest path from 3 to 6: [3, 5, 6]
Shortest path from 2 to 7: [2, 4, 5, 6, 7]
Process finished with exit code 0

```

Conclusion

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

Exercice 3

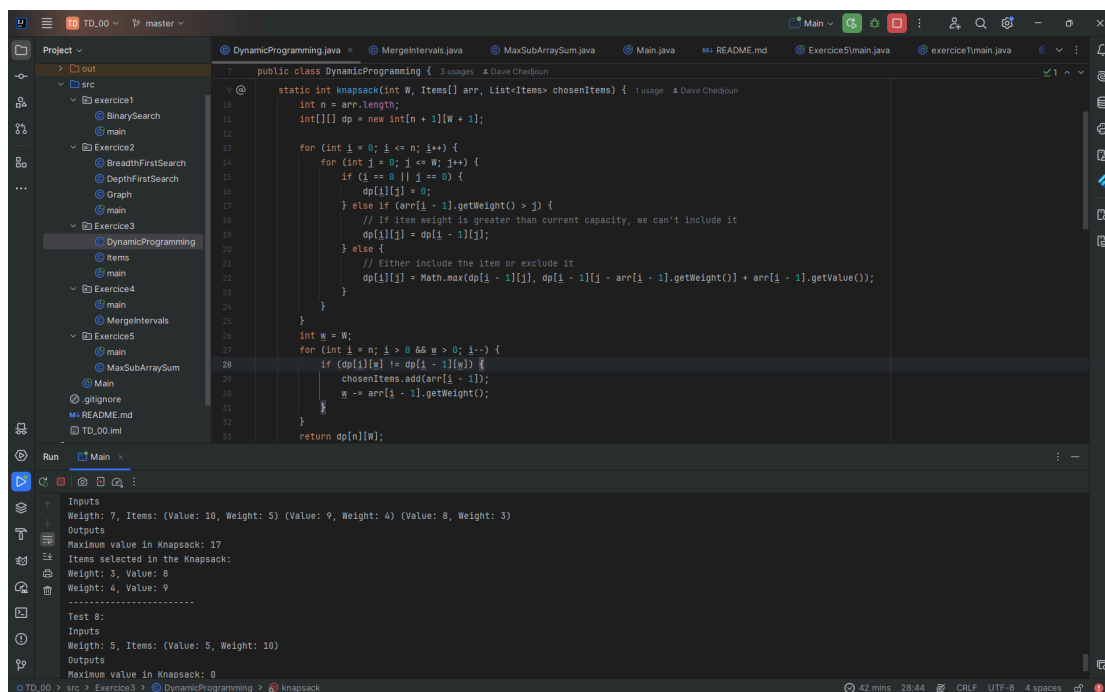
Nous avons un sac à dos de capacité W . Nous avons N objets, chacun avec un poids $weight[i]$ et une valeur $value[i]$. Objectif : maximiser la somme des valeurs sans dépasser W .

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 $dp[N][W]$.



```
public class DynamicProgramming {
    static int knapsack(int W, Items[] arr, List<Items> chosenItems) {
        int n = arr.length;
        int[][] dp = new int[n + 1][W + 1];

        for (int i = 0; i <= n; i++) {
            for (int j = 0; j <= W; j++) {
                if (i == 0 || j == 0) {
                    dp[i][j] = 0;
                } else if (arr[i - 1].getWeight() > j) {
                    // If item weight is greater than current capacity, we can't include it
                    dp[i][j] = dp[i - 1][j];
                } else {
                    // Either include the item or exclude it
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i - 1][j - arr[i - 1].getWeight()] + arr[i - 1].getValue());
                }
            }
        }

        int w = W;
        for (int i = n; i > 0 && w > 0; i--) {
            if (dp[i][w] != dp[i - 1][w]) {
                chosenItems.add(arr[i - 1]);
                w -= arr[i - 1].getWeight();
            }
        }

        return dp[n][W];
    }
}
```

Run

Inputs: Weight: 7, Items: (Value: 10, Weight: 5) (Value: 9, Weight: 4) (Value: 8, Weight: 3)

Outputs: Maximum value in Knapsack: 17
Items selected in the Knapsack:
Weight: 3, Value: 8
Weight: 4, Value: 9

Test 8:

Inputs: Weight: 5, Items: (Value: 5, Weight: 10)

Outputs: Maximum value in Knapsack: 0

Conclusion

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

Exercice 4

Description

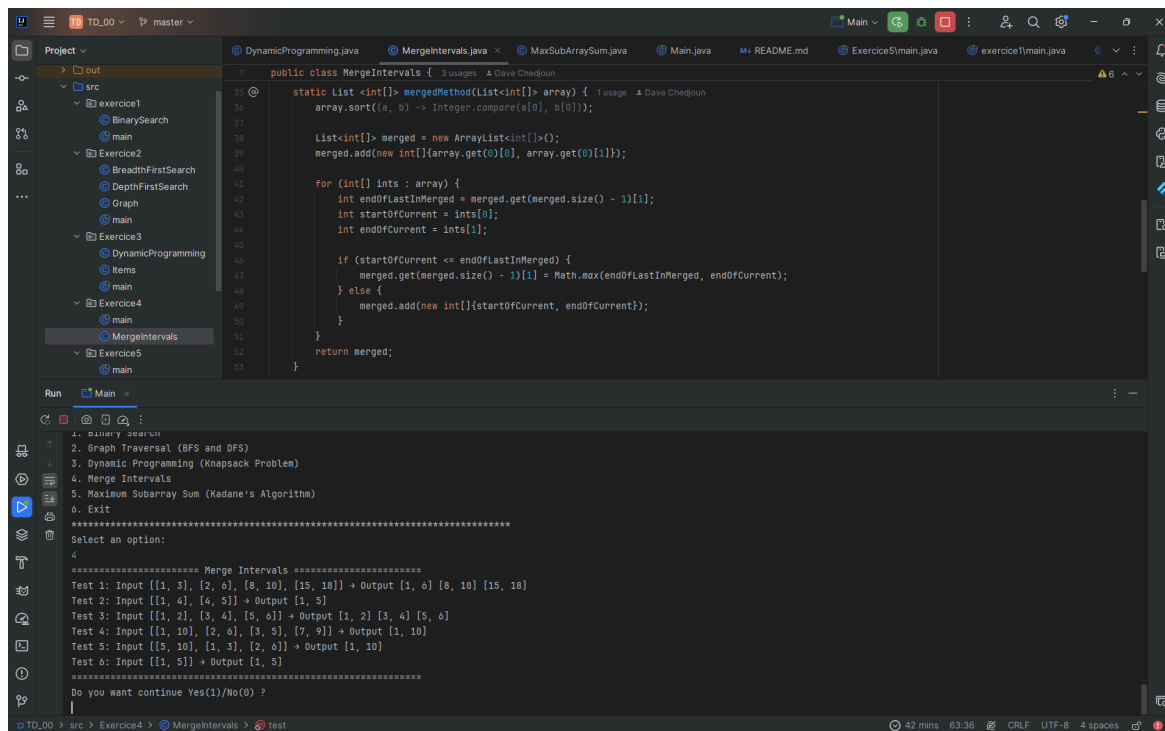
On nous donne N intervalles sous forme (début, fin). Si deux intervalles $[a, b]$ et $[c, d]$ se chevauchent ($b \geq c$), on les fusionne en $[a, d]$.

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 ($\max(\text{fin1}, \text{fin2})$).
 - Sinon, ajouter un nouvel intervalle.



The screenshot shows an IDE with the following components:

- Project Explorer:** Shows a project structure with folders for 'src' and 'out'. The 'src' folder contains subfolders for 'exercice1', 'exercice2', 'exercice3', 'exercice4', and 'exercice5'. The 'exercice4' folder contains 'MergeIntervals.java'.
- Code Editor:** Displays the code for 'MergeIntervals.java'. The code defines a static method 'mergedMethod' that takes a list of intervals and returns a merged list. The code sorts the intervals by their start value and then iterates through them, merging overlapping intervals.
- Run Console:** Shows the output of the program. It displays a menu with options 1 to 6. Option 4 is selected, and the program outputs the merged intervals for the input $[(1, 3), (2, 6), (8, 10), (15, 18)]$, which are $[(1, 6), (8, 10), (15, 18)]$.

Conclusion

Complexité : $O(N \log N)$ pour le tri + $O(N)$ pour la fusion $\rightarrow O(N \log N)$.

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)`

The screenshot shows an IDE with a project named 'TD_00'. The file explorer on the left shows a directory structure with 'Exercice5' containing 'Main.java' and 'MaxSubArraySum.java'. The editor displays the code for 'MaxSubArraySum.java', which implements the Kadane's Algorithm. The code defines a 'kadaneApproach' method that iterates through the array, updating 'max_courant' and 'max_global' as described in the solution. The 'Run' console at the bottom shows the output of the program, which includes a menu of options and test cases. The test case for the input [-2, 1, -3, 4, -1, 2, 1, -5, 4] shows an output of 7, which is slightly different from the expected output of 6 mentioned in the problem statement.

```
public class MaxSubArraySum {  
    static int kadaneApproach(int[] arr) {  
        int res = arr[0];  
        int sum = arr[0];  
        for (int i = 1; i < arr.length; i++) {  
            sum = Math.max(sum + arr[i], arr[i]);  
            res = Math.max(res, sum);  
        }  
        return res;  
    }  
}
```

Run Console Output:

```
*****  
Welcome to this first TP of SBSE  
1. Binary Search  
2. Graph Traversal (BFS and DFS)  
3. Dynamic Programming (Knapsack Problem)  
4. Merge Intervals  
5. Maximum Subarray Sum (Kadane's Algorithm)  
6. Exit  
*****  
Select an option:  
5  
***** Maximum Subarray Sum (Kadane's Algorithm) *****  
Test 1: Input [2, 3, -8, 7, -1, 2, 3] Output 11  
Test 2: Input [-2, -3, -8, -1, -5] Output -1  
Test 3: Input [5, 10, 15, 20] Output 50  
Test 4: Input [-1, 3, -2, 4, -1, 2, 1, -5, 4] Output 7  
Test 5: Input [0, 0, 0, 0] Output 0  
Test 6: Input [10] Output 10  
Test 7: Input [-10] Output -10  
Test 8: Input [4, -1, 2, 1] Output 6  
Test 9: Input [100, -90, 100, -90, 100] Output 120  
*****  
Do you want continue Yes(1)/No(0) ?
```

Conclusion

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