**Table Of Index**

|  |  |  |  |
| --- | --- | --- | --- |
| Sl | Excrement | Page | Remark |
| 1 | Linear Search Algorithm |  |  |
| 2 | Binary Search Algorithm |  |  |
| 3 | Insertion Sort Algorithm |  |  |
| 4 | Merge Sort Algorithm |  |  |
| 5 | Quick Sort Algorithm |  |  |
| 6 | Road Cutting Algorithm |  |  |
| 7 | 0/1 Knapsack Algorithm |  |  |
| 8 | Depth-First Search (DFS) Algorithm |  |  |
| 9 | Breadth-First Search (BFS) Algorithm |  |  |
| 10 | Bellman-Ford Algorithm |  |  |
| 11 | Dijkstra's Algorithm |  |  |
| 13 | Kruskal's Algorithm |  |  |
| 14 | Prim's Algorithm |  |  |

**Experiment Name: Linear Search Algorithm**

**Objective:**

To understand and implement the Linear Search algorithm to find an element in an unsorted list of numbers.

**Theory:**

Linear Search is one of the simplest searching algorithms. It checks each element of the array or list sequentially until the desired element is found or the list ends.

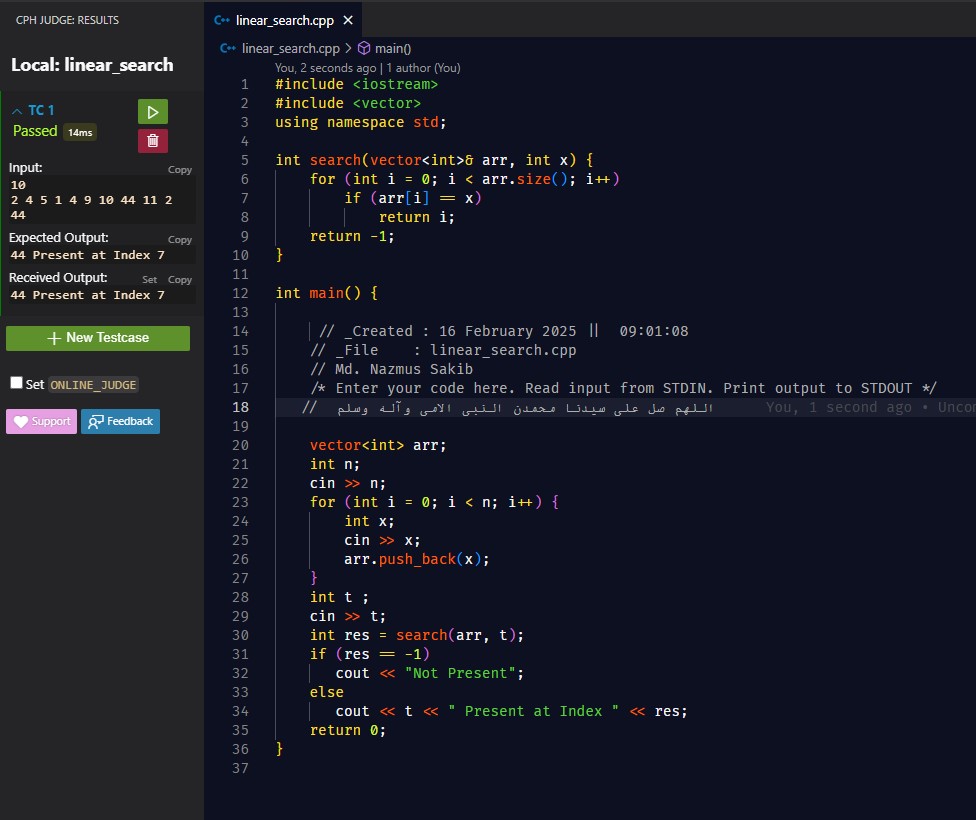
* **Working Principle:**
  + Start from the first element of the list.
  + Compare each element with the target value.
  + If the target value is found, return its position.
  + If the target is not found by the end of the list, return an indication of failure (e.g., -1).
* **Complexity:**
  + **Time Complexity:** O(n), where n is the number of elements in the list.
  + **Space Complexity:** O(1), as no additional data structures are used.
* **Advantages:**
  + Simple and easy to implement.
  + No need for sorted data.
* **Disadvantages:**
  + Not efficient for large datasets.
  + Average and worst-case time complexity is O(n).

**Algorithm Steps:**

1. Start from the first element of the list.
2. Compare the current element with the target value.
3. If they are equal, return the index of the current element.
4. If they are not equal, move to the next element.
5. Repeat steps 2-4 until the end of the list.
6. If the target value is not found, return -1.

**Time Complexity:**  
The time complexity of the Linear Search algorithm is **O(n)**, where n is the number of elements in the list. This is because the algorithm checks each element sequentially, and in the worst-case scenario, it has to examine all n elements.

**Space Complexity:**  
The space complexity of Linear Search is **O(1)**. This is because the algorithm does not require any additional data structures and uses only a few variables, independent of the size of the input list.



**Experiment Name:** Binary Search Algorithm  
**Objective:**  
To understand and implement the Binary Search algorithm to efficiently find an element in a sorted list of numbers.

**Theory:**  
Binary Search is a highly efficient search algorithm that works by repeatedly dividing the search interval in half. If the value of the target element is less than the value in the middle of the interval, the search continues in the lower half. Otherwise, it continues in the upper half. This process repeats until the target value is found or the interval is empty.

**Working Principle:**

1. Start with the middle element of the sorted list.
2. If the middle element is the target value, return its index.
3. If the target value is less than the middle element, narrow the search to the lower half of the list.
4. If the target value is greater than the middle element, narrow the search to the upper half of the list.
5. Repeat the process until the target is found or the search interval becomes empty.

**Complexity:**

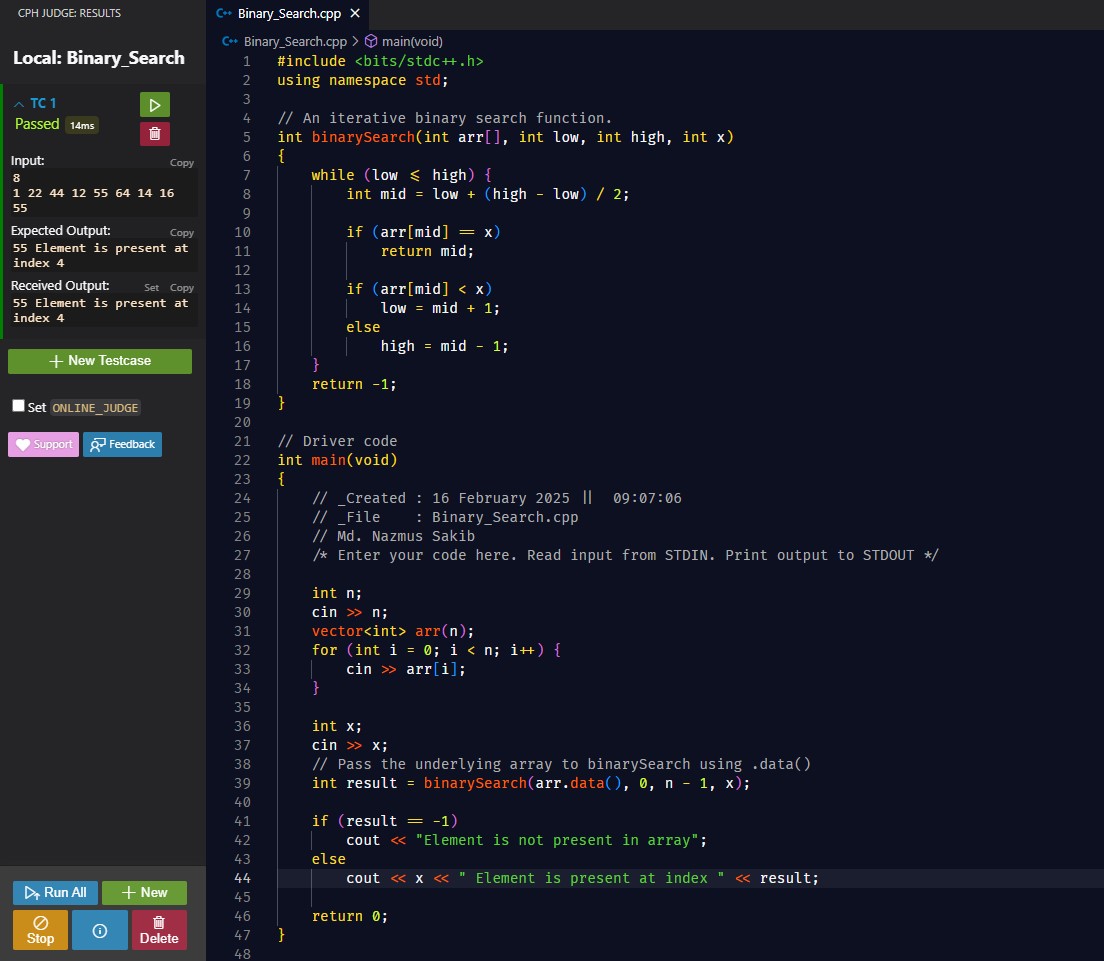
* **Time Complexity:** O(log n), where n is the number of elements in the list. Since the search space is halved at each step, the number of comparisons grows logarithmically with the size of the list.
* **Space Complexity:** O(1) for the iterative version, as no extra space is required other than the variables used for indexing. If a recursive implementation is used, the space complexity can be O(log n) due to the call stack.

**Advantages:**

* Very efficient for large datasets.
* Time complexity is logarithmic (O(log n)), making it faster than linear search for sorted data.

**Disadvantages:**

* Requires the list to be sorted before searching.
* Not suitable for unsorted data.



**Experiment Name:** Insertion Sort Algorithm  
**Objective:**  
To understand and implement the Insertion Sort algorithm to sort a list of numbers in ascending order.

**Theory:**  
Insertion Sort is a simple comparison-based sorting algorithm. It builds the final sorted array one element at a time. It works similarly to how we might sort playing cards in our hands: we take one card at a time and place it in the correct position relative to the already sorted cards.

**Working Principle:**

1. Start from the second element of the list.
2. Compare the current element with the previous elements, moving the larger elements one position to the right.
3. Insert the current element at the correct position in the sorted part of the list.
4. Repeat steps 2-3 for each element until the entire list is sorted.

**Complexity:**

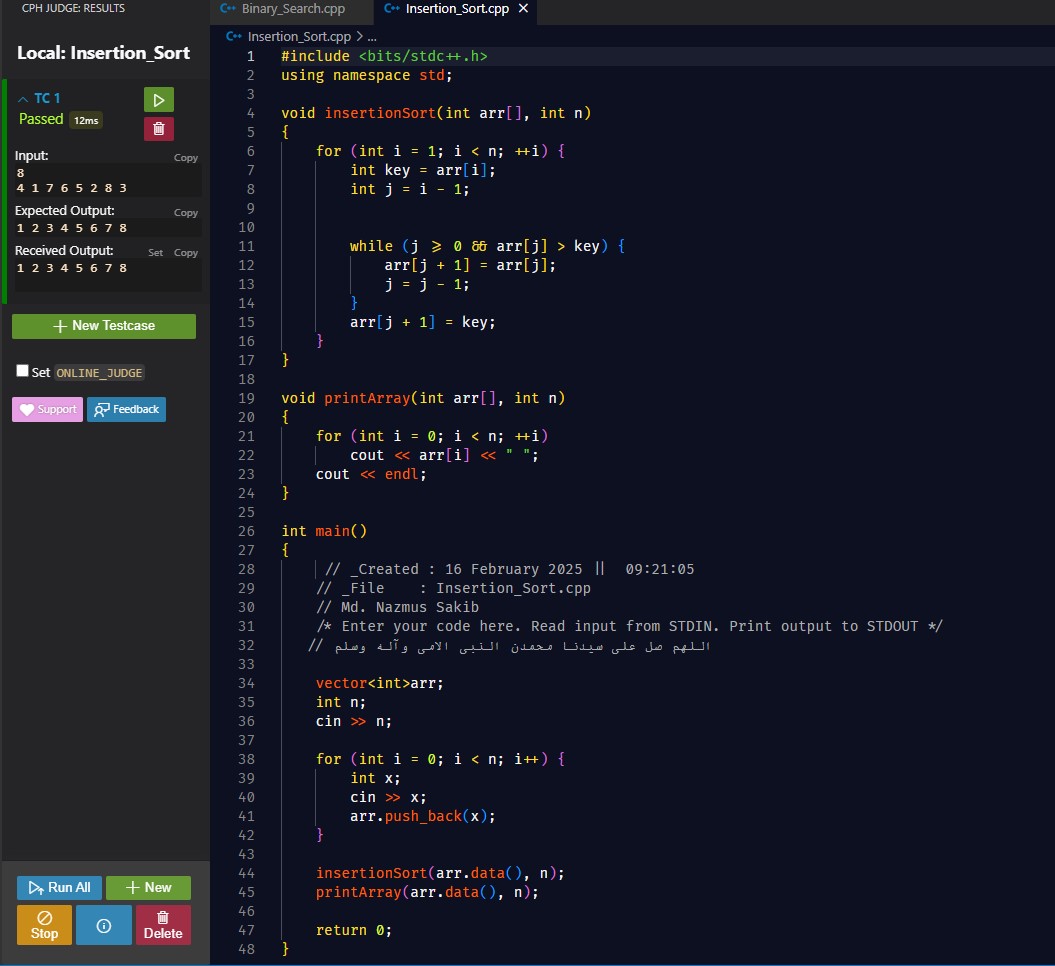
* **Time Complexity:**
  + Best case: O(n) when the list is already sorted.
  + Worst case: O(n²) when the list is in reverse order.
  + Average case: O(n²), as it typically requires comparing each element with every other element.
* **Space Complexity:** O(1), as it sorts the list in place and doesn't require any extra space other than a few variables.

**Advantages:**

* Simple to implement and understand.
* Efficient for small datasets or nearly sorted data.
* Performs well when the list is partially sorted.

**Disadvantages:**

* Not efficient for large datasets due to its O(n²) time complexity in the worst case.
* Comparisons and shifts can be expensive for large inputs.



**Experiment Name:** Merge Sort Algorithm  
**Objective:**  
To understand and implement the Merge Sort algorithm to efficiently sort a list of numbers.

**Theory:**  
Merge Sort is a divide-and-conquer sorting algorithm that divides the list into smaller sublists, sorts them, and then merges them back together in the correct order. It works by recursively splitting the list into halves, sorting each half, and then merging the sorted halves.

**Working Principle:**

1. Divide the unsorted list into two halves.
2. Recursively divide each half until each sublist contains a single element.
3. Merge the sublists back together in a sorted manner, comparing elements from each sublist.
4. Repeat the merging process until only one sorted list remains.

**Complexity:**

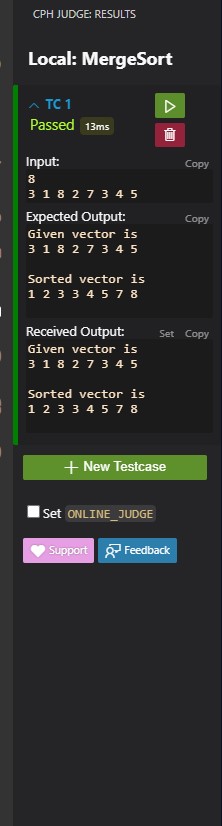
* **Time Complexity:**
  + Best, worst, and average case: O(n log n), as the list is divided in half (log n divisions), and merging each division takes O(n) time.
* **Space Complexity:** O(n), as it requires additional space to store the temporary sublists during the merging process.

**Advantages:**

* Stable sort, meaning the order of equal elements remains unchanged.
* Very efficient for large datasets with time complexity of O(n log n).
* Suitable for external sorting (sorting large datasets stored on disk).

**Disadvantages:**

* Requires extra space for temporary storage, which can be a limitation for very large datasets.
* Not as efficient for small datasets compared to simpler algorithms like Insertion Sort or Selection Sort.



**Experiment Name:** Quick Sort Algorithm  
**Objective:**  
To understand and implement the Quick Sort algorithm to efficiently sort a list of numbers.

**Theory:**  
Quick Sort is a divide-and-conquer sorting algorithm that works by selecting a pivot element from the list, partitioning the other elements into two sublists (those less than the pivot and those greater than the pivot), and then recursively sorting the sublists. This process repeats until the entire list is sorted.

**Working Principle:**

1. Choose a pivot element from the list (various methods can be used to select the pivot).
2. Partition the list such that all elements less than the pivot are on one side, and all elements greater than the pivot are on the other side.
3. Recursively apply the same process to the sublists formed by partitioning.
4. Continue until the entire list is sorted.

**Complexity:**

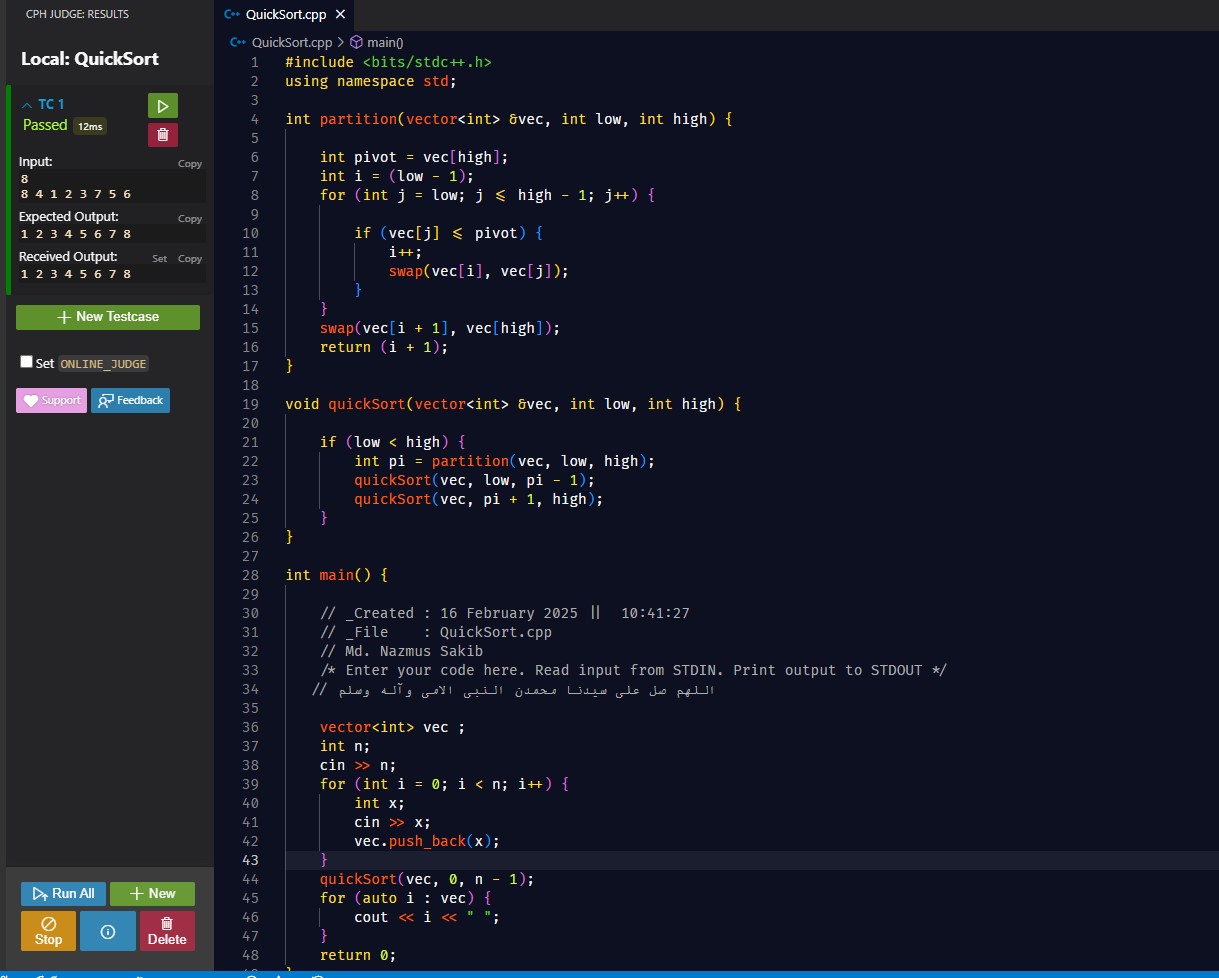
* **Time Complexity:**
  + Best case: O(n log n), when the pivot splits the list evenly.
  + Worst case: O(n²), when the pivot always ends up being the smallest or largest element.
  + Average case: O(n log n), which is typical for most random datasets.
* **Space Complexity:** O(log n) for the recursive call stack, making it more space-efficient than Merge Sort (which requires O(n) space).

**Advantages:**

* Generally faster in practice compared to other O(n log n) algorithms like Merge Sort, especially for large datasets.
* Can be implemented in-place with O(1) space complexity aside from recursion.

**Disadvantages:**

* In the worst case, Quick Sort can take O(n²) time if the pivot selection is poor (e.g., always selecting the smallest or largest element).
* Not stable, meaning that equal elements may not preserve their original order.



**Experiment Name:** Road Cutting Algorithm  
**Objective:**  
To optimize the division of a road into segments with minimum cost or optimal conditions.

**Theory:**  
The Road Cutting algorithm involves dividing a road (represented as an array) into smaller segments while minimizing or optimizing a given cost. It uses dynamic programming to find the best points to cut the road.

**Working Principle:**

1. Represent the road as an array with associated costs for each segment.
2. Calculate the optimal cut points incrementally using previously computed results (dynamic programming).
3. Combine subproblems to reach the optimal solution.

**Complexity:**

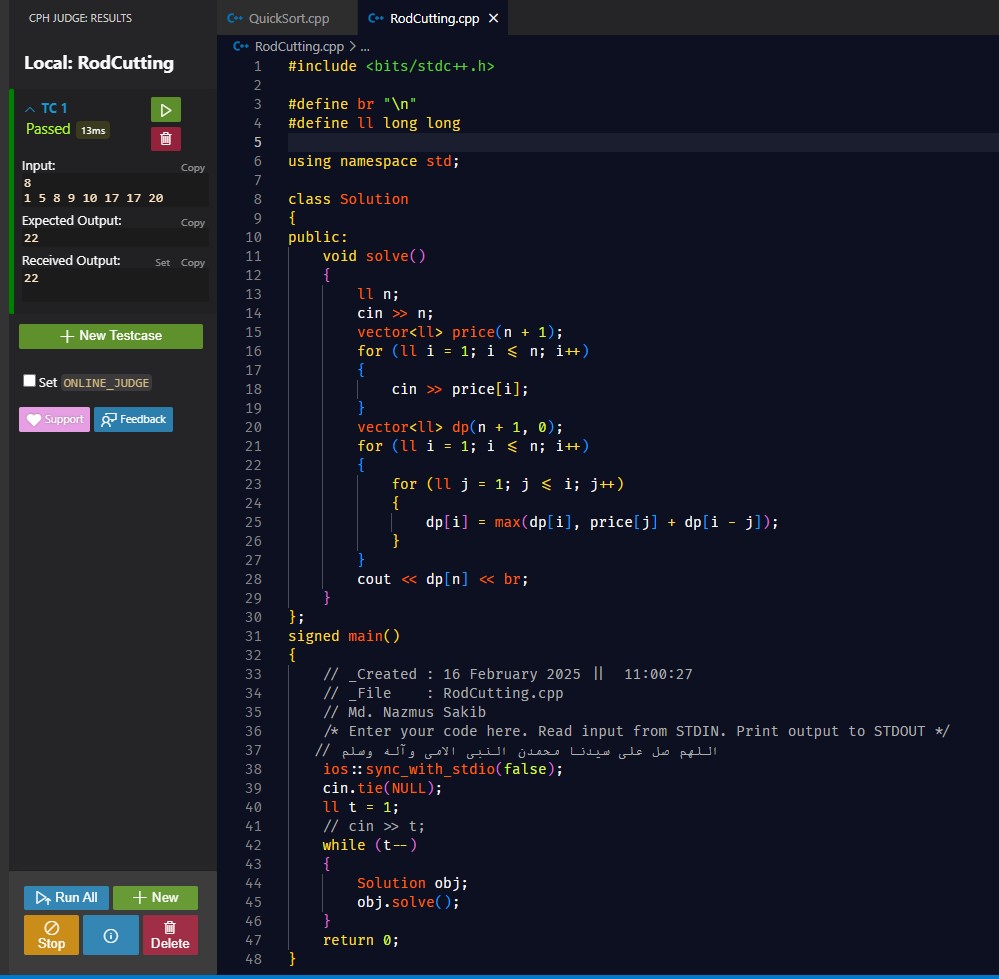
* **Time Complexity:** O(n²), where n is the number of sections.
* **Space Complexity:** O(n), for storing subproblem results.

**Advantages:**

* Solves optimization problems with interdependent decisions efficiently.

**Disadvantages:**

* Can be expensive for large datasets.



**Experiment Name:** 0/1 Knapsack Algorithm  
**Objective:**  
To understand and implement the 0/1 Knapsack problem to maximize the total value of items that can be carried, subject to a weight constraint.

**Theory:**  
The 0/1 Knapsack problem involves selecting items to put in a knapsack such that the total value is maximized, while the total weight does not exceed a given limit. Each item can either be included or excluded (0 or 1), hence the name 0/1 Knapsack.

**Working Principle:**

1. Given a list of items with corresponding weights and values, and a knapsack with a maximum weight capacity.
2. For each item, decide whether to include it in the knapsack based on its value and weight.
3. Use dynamic programming to store the optimal solutions of subproblems (different combinations of items and capacities).
4. Build up the solution incrementally by comparing the value with and without including the current item.

**Complexity:**

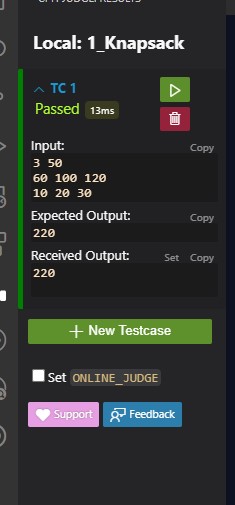
* **Time Complexity:** O(n \* W), where n is the number of items and W is the knapsack capacity.
* **Space Complexity:** O(n \* W), for storing the dynamic programming table.

**Advantages:**

* Provides an optimal solution for the problem.
* Well-suited for problems with limited resources (like weight or volume).

**Disadvantages:**

* Computationally expensive for large inputs due to time and space complexity.
* Not feasible for real-time large-scale problems without approximation methods.



**Experiment Name:** Factorial Knapsack Algorithm  
**Objective:**  
To understand and implement the Factorial Knapsack problem, which is an extension of the 0/1 Knapsack problem, where the number of items can be selected multiple times in specific quantities, often based on factorial constraints.

**Theory:**  
The Factorial Knapsack problem is a variation of the Knapsack problem where the items can be included a specific number of times, often governed by factorials or other constraints. Instead of just picking an item once or not at all, items can be chosen multiple times, but with limits (for example, an item can be chosen up to a factorial number of times).

**Working Principle:**

1. Given a list of items, each with a weight, value, and a constraint on the number of times it can be selected.
2. For each item, calculate the possible values for selecting it multiple times based on the factorial constraint.
3. Use dynamic programming or greedy strategies to find the optimal combination of items that maximizes the total value while adhering to the weight capacity and selection constraints.

**Complexity:**

* **Time Complexity:** Typically O(n \* W \* k), where n is the number of items, W is the knapsack capacity, and k is the maximum number of times an item can be selected (which may be related to factorials or a similar constraint).
* **Space Complexity:** O(n \* W \* k), for storing the results of subproblems in the dynamic programming table.

**Advantages:**

* Can be more flexible than the traditional 0/1 Knapsack problem.
* Suitable for cases where items can be picked in multiple quantities but under certain constraints.

**Disadvantages:**

* Computationally expensive for large numbers of items or large constraints.
* Involves more complex dynamic programming solutions compared to the standard 0/1 Knapsack.

**Experiment Name:** Depth-First Search (DFS) Algorithm  
**Objective:**  
To understand and implement the Depth-First Search (DFS) algorithm for traversing or searching through graphs and trees.

**Theory:**  
Depth-First Search (DFS) is a graph traversal algorithm that starts at a source node and explores as far as possible along each branch before backtracking. It uses a stack (either explicitly or through recursion) to keep track of the nodes to visit next. DFS can be applied to both directed and undirected graphs.

**Working Principle:**

1. Start from a selected node (root for a tree or any node for a graph).
2. Visit the node, mark it as visited, and push it onto the stack.
3. Explore each unvisited neighbor by recursively visiting the adjacent nodes.
4. Once all neighbors are explored, backtrack to the previous node and continue the process until all nodes are visited.
5. If implementing with recursion, the call stack serves as the stack for backtracking.

**Complexity:**

* **Time Complexity:** O(V + E), where V is the number of vertices (nodes) and E is the number of edges in the graph. Each node and edge is visited once in the traversal.
* **Space Complexity:** O(V), where V is the number of vertices, as space is needed to store the visited nodes and the stack used for traversal.

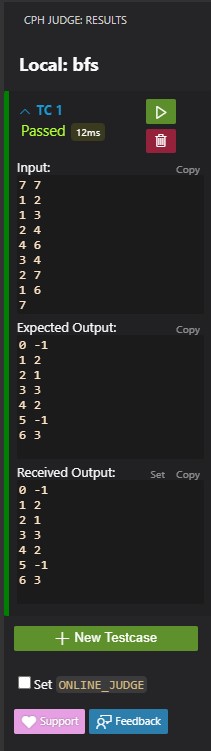
**Advantages:**

* Simple to implement and works well for both directed and undirected graphs.
* Can be used to solve problems like finding connected components, topological sorting, and solving puzzles (e.g., mazes).

**Disadvantages:**

* May not be optimal for finding the shortest path in unweighted graphs.
* Can be memory-intensive for large graphs due to the recursive call stack.

This provides a brief overview of the Depth-First Search (DFS) algorithm and its associated time and space complexities.



**Experiment Name:** Breadth-First Search (BFS) Algorithm  
**Objective:**  
To understand and implement the Breadth-First Search (BFS) algorithm for traversing or searching through graphs and trees.

**Theory:**  
Breadth-First Search (BFS) is a graph traversal algorithm that starts at a selected node and explores all its neighbors at the current depth level before moving on to the nodes at the next depth level. BFS uses a queue to keep track of the nodes to visit next, ensuring that nodes are explored level by level.

**Working Principle:**

1. Start from a selected node (root for a tree or any node for a graph).
2. Visit the node, mark it as visited, and enqueue it.
3. While the queue is not empty, dequeue a node, visit its unvisited neighbors, mark them as visited, and enqueue them.
4. Repeat this process until all reachable nodes are visited.

**Complexity:**

* **Time Complexity:** O(V + E), where V is the number of vertices (nodes) and E is the number of edges in the graph. Each node and edge is visited once in the traversal.
* **Space Complexity:** O(V), where V is the number of vertices, as space is needed to store the visited nodes and the queue used for traversal.

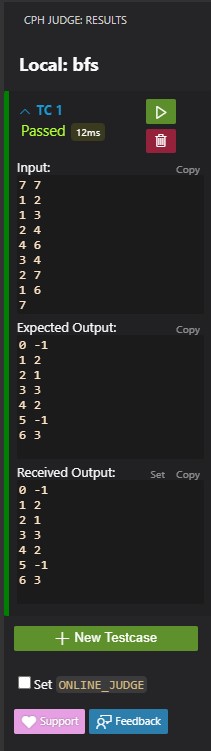
**Advantages:**

* Can be used to find the shortest path in an unweighted graph.
* Ensures that nodes are visited in the order of their distance from the starting node.

**Disadvantages:**

* Can be less memory-efficient for large graphs due to the need to store all nodes at each level in the queue.
* Not as efficient as DFS for some problems, such as finding a path to a specific node in a large tree.

This provides a concise overview of the Breadth-First Search (BFS) algorithm along with its time and space complexities.



**Experiment Name:** Bellman-Ford Algorithm  
**Objective:**  
To understand and implement the Bellman-Ford algorithm for finding the shortest paths from a single source node to all other nodes in a weighted graph, even if the graph contains negative weight edges.

**Theory:**  
The Bellman-Ford algorithm is a dynamic programming algorithm used to find the shortest paths in a weighted graph. Unlike Dijkstra’s algorithm, Bellman-Ford can handle graphs with negative weight edges. It works by iteratively relaxing the edges of the graph, updating the shortest path estimates, and repeating the process for all edges.

**Working Principle:**

1. Initialize the distance to the source node as 0 and all other nodes as infinity.
2. For each edge in the graph, update the shortest path estimate for the destination node if a shorter path is found.
3. Repeat this process for (V - 1) times, where V is the number of vertices (nodes).
4. After (V - 1) iterations, check for negative weight cycles by seeing if any edge can still be relaxed. If so, a negative weight cycle exists.

**Complexity:**

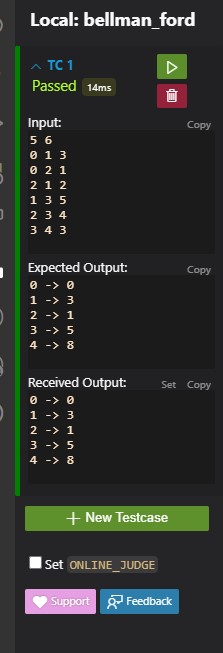
* **Time Complexity:** O(V \* E), where V is the number of vertices (nodes) and E is the number of edges in the graph. Each edge is relaxed V-1 times.
* **Space Complexity:** O(V), as space is required to store the shortest path estimates for each vertex.

**Advantages:**

* Can handle graphs with negative weight edges.
* Simple to implement and works even when negative weight cycles are present.

**Disadvantages:**

* Slower compared to other shortest path algorithms like Dijkstra’s for graphs with non-negative weights.
* Does not work efficiently for large graphs due to its O(V \* E) time complexity.



**Experiment Name:** Dijkstra's Algorithm  
**Objective:**  
To understand and implement Dijkstra's algorithm for finding the shortest paths from a single source node to all other nodes in a weighted graph with non-negative edge weights.

**Theory:**  
Dijkstra's algorithm is a greedy algorithm used to find the shortest paths in a graph with non-negative edge weights. It works by iteratively selecting the node with the smallest tentative distance, exploring its neighbors, and updating their tentative distances. The process repeats until all nodes have been processed.

**Working Principle:**

1. Initialize the distance to the source node as 0 and all other nodes as infinity.
2. Set the source node as the current node and mark it as visited.
3. For each neighbor of the current node, calculate the tentative distance by adding the edge weight to the current node's distance. If the tentative distance is smaller than the previously recorded distance, update it.
4. Select the unvisited node with the smallest tentative distance and mark it as the current node.
5. Repeat this process until all nodes are visited.

**Complexity:**

* **Time Complexity:**
  + Using a priority queue (min-heap): O((V + E) \* log V), where V is the number of vertices and E is the number of edges.
  + Without a priority queue, the time complexity is O(V²) for dense graphs.
* **Space Complexity:** O(V), as it requires storage for the tentative distances and visited nodes.

**Advantages:**

* Efficient for graphs with non-negative edge weights.
* Can find the shortest path to a single destination or all destinations.

**Disadvantages:**

* Cannot handle graphs with negative edge weights.
* Less efficient than Bellman-Ford for graphs with negative weights.



**Experiment Name:** Kruskal's Algorithm  
**Objective:**  
To understand and implement Kruskal's algorithm for finding the minimum spanning tree (MST) of a connected, undirected graph with weighted edges.

**Theory:**  
Kruskal's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a graph. It works by sorting all edges of the graph in non-decreasing order of their weights and adding them one by one to the MST, ensuring that no cycles are formed. The algorithm uses a union-find (disjoint-set) data structure to efficiently check and manage the connected components of the graph.

**Working Principle:**

1. Sort all the edges in non-decreasing order of their weights.
2. Initialize a disjoint-set (union-find) structure to keep track of the connected components.
3. Start with an empty MST. For each edge in sorted order:
   * If the edge connects two different components (no cycle is formed), add the edge to the MST.
   * If the edge would form a cycle (both vertices are already in the same component), skip it.
4. Repeat until the MST contains exactly (V-1) edges, where V is the number of vertices.

**Complexity:**

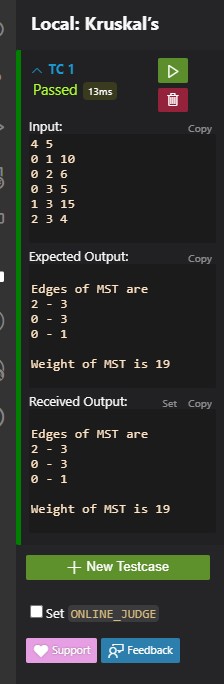
* **Time Complexity:** O(E log E), where E is the number of edges, because sorting the edges takes O(E log E) time, and the union-find operations take near constant time using path compression and union by rank.
* **Space Complexity:** O(V + E), where V is the number of vertices and E is the number of edges. Space is used to store the graph, the disjoint-set data structure, and the MST edges.

**Advantages:**

* Simple and easy to implement.
* Works efficiently for sparse graphs.
* Can handle graphs with negative weight edges.

**Disadvantages:**

* Not as efficient as Prim's algorithm for dense graphs.
* Requires sorting the edges, which can be a bottleneck in very large graphs.



**Experiment Name:** Prim's Algorithm  
**Objective:**  
To understand and implement Prim's algorithm for finding the minimum spanning tree (MST) of a connected, undirected graph with weighted edges.

**Theory:**  
Prim's algorithm is a greedy algorithm used to find the minimum spanning tree (MST) of a graph. The MST is a subset of edges that connects all the vertices without any cycles and with the minimum possible total edge weight. The algorithm starts with a single vertex and grows the MST by adding the smallest weight edge that connects a vertex inside the MST to a vertex outside of it.

**Working Principle:**

1. Start with an arbitrary node and include it in the MST.
2. Select the edge with the smallest weight that connects a node in the MST to a node outside of it.
3. Add the selected edge and its corresponding vertex to the MST.
4. Repeat the process until all vertices are included in the MST.

**Complexity:**

* **Time Complexity:**
  + Using a priority queue (min-heap): O((V + E) \* log V), where V is the number of vertices and E is the number of edges.
  + Without a priority queue, the time complexity is O(V²) for dense graphs.
* **Space Complexity:** O(V + E), to store the graph representation (e.g., adjacency list or matrix) and the data structures needed for processing vertices and edges.

**Advantages:**

* Efficient for dense graphs.
* Finds the optimal MST for a graph with non-negative edge weights.

**Disadvantages:**

* Not suitable for graphs with negative edge weights (though MST is always possible).
* Can be less efficient for sparse graphs when a simple greedy approach could be more optimal.

