Design and Analysis of Algorithms

Report

Index

Question-1:

Commence the initial stage by coding the core sorting algorithms, namely:

- In-Place Quick Sort(Pick the pivot as the last element)
- 3-Way Merge Sort • In-Place Heap Sort
- Bucket Sort
- Radix Sort (For this, the input is a linked list and it max. size is 25, in which one node of the linked list possess exactly one element)

Afterward, create a series of random input test cases in diverse sizes, starting with a minimum of 100, and including 500 and 1000. Proceed to assess the efficiency of these algorithms using the criteria listed below:

- Number of comparisons (where applicable)
- Number of swaps (where applicable)
- Number of basic operations (other than the ones mentioned above)
- Execution time in milliseconds
- Memory usage Upon completing this evaluation, articulate your findings and deduce conclusions.

Submission Requirements:

- Submit code files with clear, succinct comments. It's important to note that code embedded within document files (such as Word documents) is not permissible.
- Include input data files that contain the test cases used for the assessment.
- Provide a detailed report that encapsulates your analysis, complemented by pertinent screenshots.

i) In-Place Quick Sort (Pick the pivot as the last element):

Approach:

- **Partitioning**: The array is divided into two parts based on a pivot element (chosen as the last element in the current segment). Elements smaller than the pivot are moved to the left, and elements larger than the pivot are moved to the right.
- **Recursive Sorting**: The Quick Sort algorithm is recursively applied to the left and right partitions until the entire array is sorted.

Algorithm:

Partitioning Function:

- Set the pivot as the last element (arr[high]).
- Initialize the partition index i to low - 1.
- Iterate through the array from low to high-1.
- If the current element is less than or equal to the pivot, increment i and swap arr[i] with the current element.
- Swap the pivot element with the element at i+1 to place the pivot in its correct sorted position.

Quick Sort Function :

- If low is less than high, perform then,
- Call the partition function to get the pivot index.
- Recursively apply Quick Sort on the left sub-array (low to pivot index - 1).
- Recursively apply Quick Sort on the right sub-array (pivot index + 1 to high).

Time Complexity Analysis:

Divide:

Partitioning: The array is partitioned around the pivot element, which takes O(n) time.

Conquer:

Sorting Each Part Recursively: The array is divided into two subarrays, and each is recursively sorted.

Combine:

No Combining Step Required: Quick Sort is an in-place sort, so no additional merging step is required.

Worst Case:

In the worst case, the recurrence relation is: T(n) = T(n - 1) + O(n). Solving this using the Master Theorem or directly by expansion:

T(n) = T(n - 1) + n

T(n) = T(n - 2) + (n - 1) + n

T(n) = T(1) + 2 + 3 + ... + (n - 1) + n

T(n) = O(n^2)

Best Case and Average Case:

In the best case, the pivot divides the array into two equal halves, leading to the recurrence relation:

T(n) = 2T(n/2) + O(n)

Using the Master Theorem with a = 2, b = 2, and d = 1:

T(n) = O(n log n)

Worst-case time complexity: O(n^2)
Best-case time complexity: O(nlogn)
Average-case time complexity: O(nlogn)

Space Complexity:

Best Case: O(log n)

Average Case: O(log n)

Worst Case: O(n) – due to extremely unbalanced partitions

ii) 3-Way Merge Sort:

Approach:

- **Divide and Conquer**: The array is divided into three roughly equal parts rather than two, as in traditional merge sort. This division allows for parallel sorting and merging, which can be beneficial for certain types of data or computational environments.
- **Recursive Sorting:** The 3-way Merge Sort algorithm recursively sorts each of the three parts and then merges them back into a single sorted array.

Algorithm:

1. **Merge Function**:

- Split the array into three parts using the indices provided.
- Merge these three sorted parts into the original array by comparing the element from the temporary arrays (left, middle, right).
- Update the counter for comparisons, basic operations, and memory usage.

2. **3-Way Merge Sort Function**:

- Calculate two mid-points to divide the array into three parts.
- Recursively apply the merge sort to each of the three parts.
- Use the merge_three function to merge the three sorted parts.

Time Complexity Analysis:

1. Divide:

The array is divided into three parts. The division itself takes O (1) time since it's just calculating indices.

2. Conquer:

Sorting Each Part Recursively:

Each of the three parts is recursively sorted using the 3-way merge sort algorithm.

This step involves recursively solving the subproblems of size n/3.

The recurrence relation for the conquer step can be expressed as:

$$T(n) = 3T\left(\frac{n}{3}\right) + O(n)$$

Here, O(n) represents the merging step that follows.

3. Combine:

Merging Three Sorted Subarrays:

The three sorted subarrays are merged into a single sorted array.

Time Complexity for Merging:

Merging three subarrays takes O(n) time, where n is the total number of elements in the three subarrays combined.

Using the Master Theorem:

Given the recurrence relation: $T(n) = 3T\left(\frac{n}{3}\right) + O(n)$

We can compare this with the general form of the Master Theorem: $T(n) = aT\left(\frac{n}{b}\right) + O(nd)$

where:

- $a = 3$
- $b = 3$
- n^d=n, so $d = 1$

Now, we calculate: $\log_b a = \log_3 3 = 1$

Since $\log_b a = d$, the time complexity falls into the second case of the Master Theorem, which gives:

$T(n)=O(n^d \log n)=O(n\log n)$

Worst-case time complexity: O(nlogn)
Best-case time complexity: O(nlogn)
Average-case time complexity: O(nlogn)

Space Complexity:

Stack Space: O(log n)

Auxiliary Space: O(n)

Overall Complexity: O(n)

iii) In-Place Heap Sort:

Approach:

- **Heap Construction**: The algorithm first builds a max heap from the input array. A max heap is a binary tree where each parent node is greater than or equal to its child nodes. This process ensures that the largest element is at the root of the heap.
- **Heap Sorting**: Once the max heap is constructed, the algorithm repeatedly extracts the largest element (the root of the heap) and places it at the end of the array, reducing the heap size each time. After each extraction, the heap property is restored by calling the heapify function.

Algorithm:

1. **Heapify Function**:

- Initialize largest as the root element. Calculate the positions of the left and right children.
- Compare the root with its left and right children. If either child is greater than the root, update largest to point to that child.
- If largest is not the root, swap the root with the largest child and recursively heapify the affected subtree.
- Update the counter to track comparisons, swaps, and basic operations.

2. **Heap Sort Function**:

- Build a max heap by calling the heapify function for each non-leaf node, starting from the bottom-most level of the tree.
- Extract the maximum element (root of the heap) and swap it with the last element in the current heap. Reduce the heap size by one and call heapify to restore the heap property.
- Repeat the extraction and heapify steps until the heap size is reduced to one.

Time Complexity Analysis:

Best-Case Time Complexity:

- In the best case, the algorithm still needs to transform the array into a max heap, which involves heapifying subtrees. The heapify operation must be called for each node in the heap, and each heapify call can take up to O(log n) time due to the tree's height.
- Building the heap takes O(n) time, and heapifying and extracting each element takes O(n log n). Thus, the best-case time complexity is O(n log n).

Average-Case Time Complexity:

- In the average case, the heapify process will be called for each node, and on average, each heapify operation will still take O(log n) time.
- Since the same process of heap building and extraction applies as in the best case, the average-case time complexity remains O(n log n).

Worst-Case Time Complexity:

- Every element might need to be compared against other elements during the heapify process. Each heapify can still go up to O(log n) in height.
- Since heap sort always involves building a max heap and performing n extract-max operations, each taking O(log n) time, the worst-case time complexity is O(n log n).

Worst-case time complexity: O(nlogn)
Best-case time complexity: O(nlogn)
Average-case time complexity: O(nlogn)

Space complexity:

T(n) = O(n) – Memory used to store input Elements

iv) Bucket Sort:

Approach:

- **Initialization of Buckets:**
  - The algorithm initializes an array of empty buckets. Each bucket will hold a portion of the elements from the input array based on their values.
  - The bucket count is chosen to be the same as the number of elements in the input array, which allows for efficient distribution of elements.
- **Distributing Elements into Buckets:**
  - Each element from the input array is placed into a bucket. The bucket index is determined by multiplying the element's value by the number of buckets, ensuring that elements are evenly distributed.
  - The placement of elements into their respective buckets is based on their relative value, effectively sorting the array into smaller subarrays (buckets) before performing more granular sorting.
- **Sorting Individual Buckets:**
  - Once all elements are distributed into buckets, each bucket is individually sorted using insertion sort. Insertion sort is chosen for its efficiency on small, nearly sorted data sets, which is typical of bucket contents.
  - The elements from each bucket are concatenated in order, resulting in a sorted array.

Algorithm:

1. **Insertion Sort Function :**
   - Iterates through each element in a bucket, comparing it with its predecessors and shifting elements as needed to maintain sorted order.
   - Counts basic operations for key initialization, assignment, and decrementing indices.
   - Comparisons are counted for each comparison within the while loop.
2. **Bucket Sort Function:**
   - Initializes n empty buckets, where each bucket can hold up to n elements.
   - Distributes elements from the input array into corresponding buckets based on their value, ensuring uniform distribution.
   - Each bucket is then sorted individually using insertion sort.
   - The sorted contents of each bucket are concatenated back into the original array.

Time Complexity Analysis:

Best Case:

- Uniform distribution of elements into n buckets
- Each bucket contains same No. of elements
- n = 1
- Sorting bucket takes O(1)
- For 3 for loops: O(1) + O(1) + O(1) = O(1)
- Average case is also same.

Worst Case:

- Non-Uniform Distribution:
- All elements fall into same bucket
- First loop = O(n)
- Insertion sort = O(n^2)
- Final loop = O(n)
- Total = O(n) + O(n^2) + O(n) = O(n^2)

Worst-case time complexity: O(n^2)
Best-case time complexity: O(1)
Average-case time complexity: O(1)

Space Complexity:

Auxiliary Space: O(n2)

v) Radix Sort:

For this, the input is a linked list and its max. size is 25, in which one node of the linked list possess exactly one element.

Approach:

1. **Linked List Initialization**:
    - The algorithm represents the data using a singly linked list where each node contains an integer value and a pointer to the next node.
    - The linked list is initialized by reading values from an input file and inserting them as nodes.
2. **Radix Sort**:
    - The Radix Sort algorithm sorts the input data by processing each digit of the numbers, starting from the least significant digit to the most significant digit.
    - It uses a stable sorting algorithm (counting sort in this case) to sort numbers based on individual digit values.
3. **Counting Sort for Each Digit**:
    - A counting sort is performed for each digit (units, tens, hundreds, etc.). The digit is identified by using an exponent (exp) that determines the current place value.
    - The linked list is repeatedly sorted based on each digit's value until all digit places are sorted.
4. **Merge Function**:
    - A helper function merge is used to combine two linked lists. This function is used to collect nodes that share the same digit value during counting sort.

Algorithm:

1. **Find the Maximum Element:**

- Determine the maximum number in the array to know the number of digits in the largest number.

2. **Sort by Digits:**

- For each digit position (starting from the least significant digit to the most significant digit), use Counting Sort to sort the array based on that digit**.**

3. **Counting Sort for Each Digit:**

- **Count Occurrences:** Count how many times each digit appears.
- **Cumulative Count:** Modify the count array to store cumulative counts**.**

- **Build Output Array:** Use the cumulative counts to place the digits in the correct position in the output array.
- **Copy to Original:** Copy the sorted elements back to the original array.

Time Complexity Analysis:

The radix sort's overall time complexity is O(d×(n+b)), where d is the maximum number of digits, n is the number of elements, and b is the base of the number system.

Best Case:

O(d×n) when the elements are already sorted or have a uniform distribution across digit places.

Worst Case:

Still O(d×n), as the algorithm's performance doesn't degrade based on element order, only on the number of digits and elements.

Worst-case time complexity: O(d×n)
Best-case time complexity: O(d×n)
Average-case time complexity: O(d×n)

Outputs:

Test Caseswith Input Size 100,500 and 1000

In-Place Quick Sort:

```
[Running] python -u "d:\DAA Assignment\Q1\quick.py"
Quick Sort - Test case size: 100
Comparisons: 582
Swaps: 411
Basic operations: 0
Execution time: 1.01 ms
Memory usage: 0.14 KB
--------------------------------------
Quick Sort - Test case size: 500
Comparisons: 5175
Swaps: 3463
Basic operations: 0
Execution time: 10.00 ms
Memory usage: 1.12 KB
--------------------------------------
Quick Sort - Test case size: 1000
Comparisons: 11304
Swaps: 6760
Basic operations: 0
Execution time: 46.22 ms
Memory usage: 1.38 KB
--------------------------------------

[Done] exited with code=0 in 0.166 seconds
```

3-Way Merge Sort:

```
[Running] python -u "d:\DAA Assignment\Q1\merge.py"
3-Way Merge Sort - Test case size: 100
Comparisons: 296
Swaps: 0
Basic operations: 605.9474732050231
Execution time: 0.00 ms
Memory usage: 1.38 KB
--------------------------------------
3-Way Merge Sort - Test case size: 500
Comparisons: 1492
Swaps: 0
Basic operations: 4190.413779289924
Execution time: 2.01 ms
Memory usage: 4.29 KB
--------------------------------------
3-Way Merge Sort - Test case size: 1000
Comparisons: 2997
Swaps: 0
Basic operations: 9380.827558579846
Execution time: 6.11 ms
Memory usage: 8.30 KB
--------------------------------------

[Done] exited with code=0 in 0.12 seconds
```

In-Place Heap Sort:

```
[Running] python -u "d:\DAA Assignment\Q1\heapsort.py"
Heap Sort - Test case size: 100
Comparisons: 1021
Swaps: 579
Basic operations: 0
Execution time: 0.99 ms
Memory usage: 0.12 KB
---------------------------------------
Heap Sort - Test case size: 500
Comparisons: 7460
Swaps: 4079
Basic operations: 0
Execution time: 10.43 ms
Memory usage: 0.31 KB
---------------------------------------
Heap Sort - Test case size: 1000
Comparisons: 16857
Swaps: 9056
Basic operations: 0
Execution time: 22.53 ms
Memory usage: 0.37 KB
---------------------------------------

[Done] exited with code=0 in 0.158 seconds
```

Bucket Sort:

```
[Running] python -u "d:\DAA Assignment\Q1\bucket.py"
Test case size: 100
Comparisons: 53
Swaps: 17
Basic operations: 236
Execution time: 1.09 ms
Memory usage: 8.48 KB
---------------------------------------
Test case size: 500
Comparisons: 285
Swaps: 107
Basic operations: 1178
Execution time: 2.92 ms
Memory usage: 37.53 KB
---------------------------------------
Test case size: 1000
Comparisons: 614
Swaps: 254
Basic operations: 2360
Execution time: 9.02 ms
Memory usage: 79.43 KB
---------------------------------------

[Done] exited with code=0 in 0.129 seconds
```

RadixSort:

```
[Running] python -u "d:\DAA Assignment\Q1\radix.py"
10 15 19 21 22 29 33 35 42 43 48 53 55 57 60 63 68 74 76 77 81 86 88 91 94
Number of Comparisons: 0
Number of Swaps: 50
Number of Basic Operations: 470
Memory Usage: 376 bytes
Execution Time: 1010500 ns

[Done] exited with code=0 in 0.304 seconds
```

1. Number of Comparisons

Best to worst	Details
1. Bucket Sort (Float)	Minimal comparisons, best for decimal values.
2. 3-Way Merge Sort	Fewer comparisons than other integer value algorithms.
3. In-Place Quick Sort	Moderate number of comparisons.
4. In-Place Heap Sort	Highest number of comparisons.
Note	Comparisons increase significantly with larger input sizes for all algorithms.

Bucket Sort is the most efficient algorithm in terms of the number of comparisons, though this is applicable only for decimal values. For integer values, Bucket Sort exhibits the highest number of comparisons among all evaluated algorithms. Among the other algorithms, 3-Way Merge Sort performs fewer comparisons, followed by In-Place Quick Sort. In-Place Heap Sort shows the highest number of comparisons. For all three sorting algorithms, the number of comparisons increases significantly with largerinputsizes.

2. Number of Swaps

Best to Worst	Details
1. **3-Way Merge Sort, Bucket Sort**	No swaps involved.
2. **In-Place Quick Sort**	Requires fewer swaps than Heap Sort.
3. **In-Place Heap Sort**	Highest number of swaps, particularly with larger inputs.
Note	Quick Sort and Heap Sort show an increase in swaps with larger input sizes, Heap Sort more so.

No swaps are involved in 3-Way Merge Sort and Bucket Sort. In the case of In-Place Quick Sort and In-Place Heap Sort, In-Place Quick Sort consistently requires fewer swaps. As input size increases, both In-Place Quick Sort and In-Place Heap Sort show an increase in the number of swaps, with In-Place Heap Sort experiencing a more substantial rise.

3. Number of Basic Operations

Best to Worst	Details
1. **Bucket Sort**	Fewest basic operations.
2. **In-Place Quick Sort**	More basic operations than Bucket Sort but fewer than other algorithms.
3. **In-Place Heap Sort**	Approximately twice the basic operations compared to Quick Sort.
4. **3-Way Merge Sort**	Involves the highest number of basic operations.

3-Way Merge Sort involves the highest number of basic operations. Bucket Sort requires the fewest basic operations. While In-Place Quick Sort involves more basic operations than Bucket Sort, it is more efficient compared to In-Place Heap Sort, which has approximately twice the number of basic operations compared to In-Place Quick Sort.

4. Execution Time

Best to Worst	Details
1. **In-Place Quick Sort**	Shortest execution time across various input sizes.
2. **In-Place Heap Sort**	Marginally better execution time than 3-Way Merge Sort.
3. **3-Way Merge Sort**	Slightly slower than In-Place Heap Sort.
4. **Bucket Sort**	Efficient for smaller inputs; significant execution time growth for larger inputs (up to 2 ms).

Execution time varies significantly across different input sizes and algorithms. In-Place Quick Sort demonstrates the shortest execution time. Bucket Sort, although efficient for smaller inputs, exhibits substantial execution time growth with larger inputs, sometimes taking up to 2 milliseconds. Between In-Place Heap Sort and 3-Way Merge Sort, execution time can fluctuate, with In-Place Heap Sort generally showing marginally better performance compared to 3-Way Merge Sort, thoughthe difference is minimal.

5. Memory Usage

Best to Worst	Details
1. In-Place Quick Sort	Minimal memory usage.
2. In-Place Heap Sort	Comparable to Quick Sort, differing by only 4 bytes.
3. 3-Way Merge Sort, Bucket Sort	Significantly higher memory usage due to additional space for temporary arrays or buckets.

The memory usage of In-Place Quick Sort and In-Place Heap Sort is minimal and comparable, differing by only 4 bytes. In contrast, 3-Way Merge Sort and Bucket Sort utilize significantly more memory due tothe additional spacerequired fortemporary arrays (left and right) in Merge Sort and bucket arrays in Bucket Sort.

Radix Sort:

i. Number of Comparisons:

Radix Sort involves zero comparisons, as it distributes elements into buckets based on their values.

ii. Number ofSwaps:

When comparing the number of swaps in Radix Sort with Quick Sort, they appear to be similar.

iii. Number of Basic Operations:

The number of basic operations in Radix Sort is comparable to that of 3-Way Merge Sort. However, Quick Sort performs fewer basic operations than Radix Sort.

iv. Execution Time:

For equivalent input sizes, Radix Sort generally has a longer execution time compared to Quick Sort.

v. Memory Usage:

The memory usage of Radix Sort is typically higher than that of Quick Sort. However, at the same input size, Radix Sort generally uses less memory than In-Place Heap Sort.

Conclusion:

In conclusion, evaluating various sorting algorithms reveals distinct strengths and weaknesses. Bucket Sort performs optimally regarding comparisons but is limited to decimal values. For integer values, 3-Way Merge Sort is the most efficient, with the fewest comparisons and no swaps. In-Place Quick Sort and In-Place Heap Sort offer balanced performance, though they require more comparisons and swaps, particularly with larger input sizes.

Question 2:

Optimizations and strategic data structure applications can significantly refine each algorithm's performance. For guidance on implementation, consider consulting "Algorithm Design" by Goodrich et al., along with other pertinent scholarly texts viz., CLRS book. Detail the refinements applied to the algorithms and expound on how these modifications have elevated their efficiency. This should be substantiated both in theory and through empirical comparison, employing extensive and varied input test cases, against the algorithms' initial versions.

Merge sort refinement:

1. Bottom-Up Merge Sort

Approach

The bottom-up merge sort algorithm is a non-recursive (iterative) version of merge sort. It sorts the array in a bottom-up manner by merging sorted blocks of increasing sizes.

Algorithm

1. **Initialization**:

- Check if the array is empty or has one element; if so, no sorting is needed.
- Create a temporary vector to help with merging.

2. **Iterate Over Block Sizes**:

- Start with block size 1 and double it in each iteration.
- For each block size, process the array in chunks of size 2 * current block size.

3. **Merge Blocks**:

- For each block:
    - Determine the starting index of the block.
    - Calculate the midpoint of the block.
    - Determine the end index of the block.
    - Merge the blocks using the merge function.

4. **Merge Function**:

- Copy the relevant portion of the array to a temporary array.
- Merge two subarrays (left and right) back into the original array using pointers to track the positions in each subarray.

2. Recursive Merge Sort

Approach

The recursive merge sort algorithm is a classic divide-and-conquer algorithm. It recursively divides the array into two halves, sorts each half, and then merges the sorted halves.

Algorithm

1. **Recursive Divide**:

- Divide the array into two halves until each subarray has one or zero elements.

2. **Merge Function**:

- After sorting each half recursively, merge the two sorted halves into a single sorted array.

3. **Merge Function**:

- Copy the relevant portion of the array to a temporary array.
- Merge two subarrays (left and right) back into the original array using pointers to track the positions in each subarray.

Time Complexity:

	Recursive Merge Sort	**Bottom-Up Merge Sort**
Best Case	O(n log n)	O(n log n)
Average Case	O(n log n)	O(n log n)
Worst Case	O(n log n)	O(n log n)
Reason	Merge sort consistently divides and merges in O(log n) levels, with each level requiring O(n) operations.	Same as recursive; it divides and merges blocks in O(log n) levels, each requiring O(n) operations.

Space Complexity:

	Recursive Merge Sort	**Bottom-Up Merge Sort**
Auxiliary Space	O(n)	O(n)
Stack Space	O(log n)	O(1)
Total Space	O(n + log n) = O(n)	O(n)
Reason	Requires O(n) space for the temporary array and O(log n) stack space for recursion.	Requires O(n) space for the temporary array but only O(1) stack space due to iterative approach instead of recursion.

Output:

Bottom-up approach

```
[Running] cd "d:\DAA Assignment\Q2\" && g++ tempCodeRunnerFile.cpp -o tempCodeRunnerFile && "d:\DAA Assignment\Q2\"tempCodeRunnerFile
1 2 3 4 5 6 6 7 8 8 8 9 10 10 11 12 12 13 13 14 14 15 16 17 18 18 18 19 19 20 21 21 22 22 23 24 25 25 26 27 27 28 28 29 29 30 31 32 32 33
34 37 38 38 39 39 40 41 42 42 43 44 44 44 45 45 46 46 47 47 48 48 49 50 50 51 51 52 53 54 55 56 56 57 57 58 58 59 60 61 61 62 63 64 65 66
66 67 68 68 69 70 71 71 72 72 73 73 74 74 75 76 76 77 77 78 78 79 79 80 81 82 83 83 84 84 85 85 86 87 88 89 89 90 90 91 92 92 93 93 94 95
95 96 96 97 97 98 99 99 100

[Done] exited with code=0 in 3.373 seconds
```

Recursive approach

```
[Running] cd "d:\DAA Assignment\Q2\" && g++ recursive_method.cpp -o recursive_method && "d:\DAA Assignment\Q2\"recursive_method
1 2 3 4 5 6 6 7 8 8 8 9 10 10 11 12 12 13 13 14 14 15 16 17 18 18 18 19 19 20 21 21 22 22 23 24 25 25 26 27 27 28 28 29 29 30 31 32 32 33
34 37 38 38 39 39 40 41 42 42 43 44 44 44 45 45 46 46 47 47 48 48 49 50 50 51 51 52 53 54 55 56 56 57 57 58 58 59 60 61 61 62 63 64 65 66
66 67 68 68 69 70 71 71 72 72 73 73 74 74 75 76 76 77 77 78 78 79 79 80 81 82 83 83 84 84 85 85 86 87 88 89 89 90 90 91 92 92 93 93 94 95
95 96 96 97 97 98 99 99 100

[Done] exited with code=0 in 1.485 seconds
```

Quick Sort refinement:

Approach:

1. Median-of-Three Pivot Selection:

The Median-of-Three method involves selecting the pivot as the median of three key elements: the first, middle, and last elements of the current subarray.

This approach helps avoid poor pivot choices, such as those that could result from already sorted or nearly sorted arrays, by improving the balance of partitions.

2. Partitioning:

After selecting the pivot using the Median-of-Three strategy, the array is partitioned into elements less than the pivot and those greater than the pivot.

The partitioning process reorders the elements around the pivot and places the pivot in its correct position in the sorted array.

Algorithm:

1. Median-of-Three Selection:

Compute the median of the first, middle, and last elements of the subarray to choose a better pivot.

2. Partitioning Process:

Use the chosen pivot to divide the array into two parts: elements less than the pivot and elements greater than the pivot.

Recursively apply Quick Sort to the two partitions.

3. Recursive Sorting:

Continue sorting the partitions until the base case is reached (i.e., the subarray has one or zero elements).

Theoretical Advantages:

- **Reduced Worst-Case Performance:**

By choosing a better pivot, the Median-of-Three strategy helps avoid worst-case scenarios that occur with poor pivot choices, such as those that can degrade performance to O(n^2).

- **Improved Efficiency:**

This approach generally results in more balanced partitions, leading to improved average-case performance compared to selecting the first or last element as the pivot.

- **Stable Performance:**

The Median-of-Three pivot selection ensures more stable performance across different types of input arrays.

Time Complexity:

- **Overall Time Complexity:**
    - **Best Case:** O(n log n)
    - **Average Case:** O(n log n)
    - **Worst Case:** O(n^2) (though less likely with Median-of-Three compared to simpler pivot strategies)

  Comparison with Standard Quick Sort:
    - Standard Quick Sort has an average time complexity of O(n log n) but can degrade to O(n^2) with poor pivot choices.
    - The Median-of-Three strategy improves pivot selection, reducing the likelihood of encountering the worst-case scenario.

Space Complexity:

- **Auxiliary Space:**
    - **Stack Space:** O(log n) due to recursion depth.
- **Total Space:**
    - **Overall Space Complexity:** O(log n) for stack space, making it efficient in terms of memory usage.

When This Optimization is Better:

Large Datasets:

- The Median-of-Three optimization is ideal for large datasets where improved pivot selection helps achieve better performance and balance.

Nearly Sorted Data:

- This strategy is beneficial for nearly sorted data, where poor pivot choices could otherwise lead to suboptimal performance.

Memory-Constrained Environments:

- Given its low stack space usage, this approach is well-suited for environments with limited memory resources.

Balanced Performance:

- It ensures more consistent performance across different types of datasets by avoiding extreme cases that degrade performance.

Output

```
[Running] cd "d:\DAA Assignment\Q2\" && g++ quick_normal.cpp -o quick_normal && "d:\DAA Assignment\Q2\"quick_normal
Quick Sort - Test case size: 151
Comparisons: 1041
Swaps: 688
Basic operations: 0
Execution time: 0.022 ms
Memory usage: Not measured in this implementation
---------------------------------------

[Done] exited with code=0 in 1.491 seconds

[Running] cd "d:\DAA Assignment\Q2\" && g++ quick_opt.cpp -o quick_opt && "d:\DAA Assignment\Q2\"quick_opt
Quick Sort - Test case size: 151
Comparisons: 1033
Swaps: 442
Basic operations: 0
Execution time: 0.014 ms
Memory usage: Not measured in this implementation
---------------------------------------

[Done] exited with code=0 in 2.328 seconds
```

Heap Sort Refinement:

Approach:

Bottom-Up Heap Construction:

The Bottom-Up approach to heap construction involves building the heap from the bottom up, starting from the last internal node (non-leaf node) and moving up to the root. This method ensures that the heap property is maintained efficiently across the entire tree.

- **Heapify Process:** In this approach, heapify is applied to internal nodes in reverse level order, which is more efficient compared to the Top-Down method where elements are inserted one by one.
- **Efficiency:** The Bottom-Up method constructs the heap in linear time O(n), which is faster than repeatedly inserting elements into an empty heap.

Algorithm :

Building the Heap:

Start from the Last Internal Node:

- Compute the index of the last internal node (which is ((n/2)-1))
- Apply heapify to this node and all nodes up to the root.

Heapify Operation:

- For each node, ensure the subtree rooted at the node maintains the heap property by comparing with its children and swapping if necessary.
- This process ensures that every node maintains the heap property in O(logn) time for each node.

Sorting the Array:

Extract Max:

- Swap the root of the heap (maximum element) with the last element in the heap.
- Reduce the heap size by one and call heapify on the root to restore the heap property.

Repeat:

- Continue the extraction and heapification process until the heap size is reduced to one.

Theoretical Advantages:

Efficient Heap Construction:

- Linear Time Complexity: The Bottom-Up approach builds the heap in O(n) time, which is more efficient than constructing the heap using the Top-Down approach with repeated insertions.

Optimized Sorting Process:

- Improved Sorting Performance: After building the heap, each extraction and re-heapification takes O(logn) time, maintaining an overall time complexity of O(nlogn) for the sorting phase.

Balanced Heap:

- Stable Heap Structure: The Bottom-Up method ensures a balanced heap structure, leading to consistent performance during the heap operations.

Time Complexity:

Overall Time Complexity:

- Building the Heap: O(n)
- Sorting the Array: O(nlogn)

Comparison with Standard Heap Sort:

- Standard Heap Sort: Has a time complexity of O(nlogn) for both heap construction and sorting.
- Bottom-Up Heap Construction: Optimizes heap construction to O(n), thus providing a more efficient approach compared to the standard method.

Space Complexity:

Auxiliary Space:

- Heap Storage: O(n) for the heap array.
- Stack Space: O(logn) due to recursion depth in the heapify operation.

Total Space:

- Overall Space Complexity: O(n) for the heap array and O(logn) for stack space, resulting in efficient memory usage.

When This Optimization is Better:

Large Datasets:

- Efficient for Large Data: The Bottom-Up method is particularly effective for large datasets where minimizing heap construction time is crucial.

Memory-Constrained Environments:

- Optimal Memory Usage: The approach's linear time complexity for heap construction and low stack space usage make it suitable for environments with limited memory.

Balanced Performance:

- Consistent Performance: Ensures stable and consistent performance across different types of input arrays by optimizing the heap construction phase.

Output:

```
[Running] cd "d:\DAA Assignment\Q2\" && g++ heap_opt.cpp -o heap_opt && "d:\DAA Assignment\Q2\"heap_opt
Heap Sort - Test case size: 151
Comparisons: 1733
Swaps: 975
Basic operations: 0
Execution time: 0.021 ms
Memory usage: Not measured
---------------------------------------

[Done] exited with code=0 in 1.237 seconds

[Running] cd "d:\DAA Assignment\Q2\" && g++ heap_normal.cpp -o heap_normal && "d:\DAA Assignment\Q2\"heap_normal
Heap Sort - Test case size: 151
Comparisons: 1733
Swaps: 975
Basic operations: 0
Execution time: 0.03 ms
Memory usage: 0.589844 KB
---------------------------------------

[Done] exited with code=0 in 1.321 seconds
```

Question-3:

In “Algorithm Design” by Goodrich et al., a variety of algorithms are explicated, along with suggested modifications for enhanced efficiency:

a. Chapter 6 addresses the algorithm for detecting strongly connected components, articulation points, and bridges, incorporating both elementary and sophisticated improvements.

b. Chapter 7 provides a detailed exposition of Dijkstra’s algorithm for determining the shortest path from a singular source, inclusive of the path’s cost (that includes negative edge weights) and the actual route to all other vertices, as well as its variant algorithms.

c. The text also explores Boruvka’s algorithm for calculating the minimum spanning tree, presenting both the MST’s cost and its structure, in addition to different implementation methods.

For each algorithm, undertake an analysis of performance across various implementations—those delineated in the textbook and your selected alternatives—emphasizing the count of primitive operations, duration of execution, and memory consumption. Inputs should include expansive graphs of varied configurations, with at least 20 nodes. Clarify the implementations, inclusive of their modifications, and rationalize the design decisions made, as well as the resultant effects of the diverse algorithms.

Part A

Directly Strongly Connected Components:

A strongly connected component of a directed graph is a maximal subgraph where every pair of vertices is mutually reachable. This means that for any two nodes A and B in this subgraph, there is a path from A to B and a path from B to A.

Why Strongly Connected Components (SCCs) are Important?

Understanding SCCs is crucial for various applications such as:

- **Network Analysis:** Identifying clusters of tightly interconnected nodes.
- **Optimizing Web Crawlers:** Determining parts of the web graph that are closely linked.
- **Dependency Resolution:** In software, understanding which modules are interdependent.

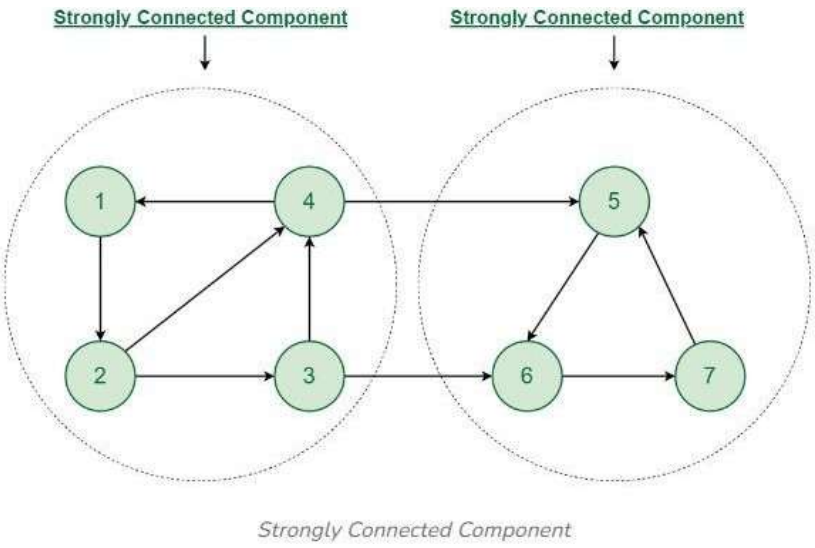

Image ref: https://www.geeksforgeeks.org/strongly-connected-components/

Approach:

1. **Perform DFS to determine the finishing order:**

- Run a Depth-First Search (DFS) on the original graph to get the vertices in their finishing order. This order is determined by the time at which each DFS call finishes. The vertex finishing last is put at the top of a stack (or similar data structure).

2. **Reverse the graph:**

- Reverse the direction of all edges in the graph to obtain the transpose graph. This helps in identifying SCCs by reversing the paths in the graph.

3. **Perform DFS in the order of finishing times:**

- Perform DFS on the reversed graph, but this time process the vertices in the order of their finishing times (obtained from the stack). Each DFS call will reveal one SCC.

Algorithm:

1. **Initialize**:

- Create a stack (sorted_order) to store the vertices according to their finishing times during the first DFS.
- Create a visited list to track visited nodes.
- Initialize a counter (primitive_ops) to keep track of primitive operations.

2. **First Pass - Determine Finishing Order**:

- For each vertex vvv that hasn't been visited:
- Perform a Depth-First Search (DFS) using the function get_sorted_order:
    - Mark the current vertex as visited.
    - For each adjacent vertex, if it hasn't been visited, recursively call DFS.
    - After all adjacent vertices are explored, push the current vertex to the stack (sorted_order).

3. **Reverse the Graph**:

- Create a new adjacency list (reversed_adj) representing the reversed graph.
- For each vertex, reverse the direction of edges.

4. **Second Pass - Find SCCs**:

- Reset the visited list.
- Initialize a counter (num_scc) to count the number of SCCs.
- While the stack (sorted_order) is not empty:
    - Pop a vertex from the stack.
    - If the vertex hasn't been visited, perform DFS using the DFS function on the reversed graph starting from this vertex.
    - Each DFS identifies a new SCC, so increment the num_scc counter.

Articulation point:

A vertex v is an articulation point (also called cut vertex) if removing v increases the number of connected components.

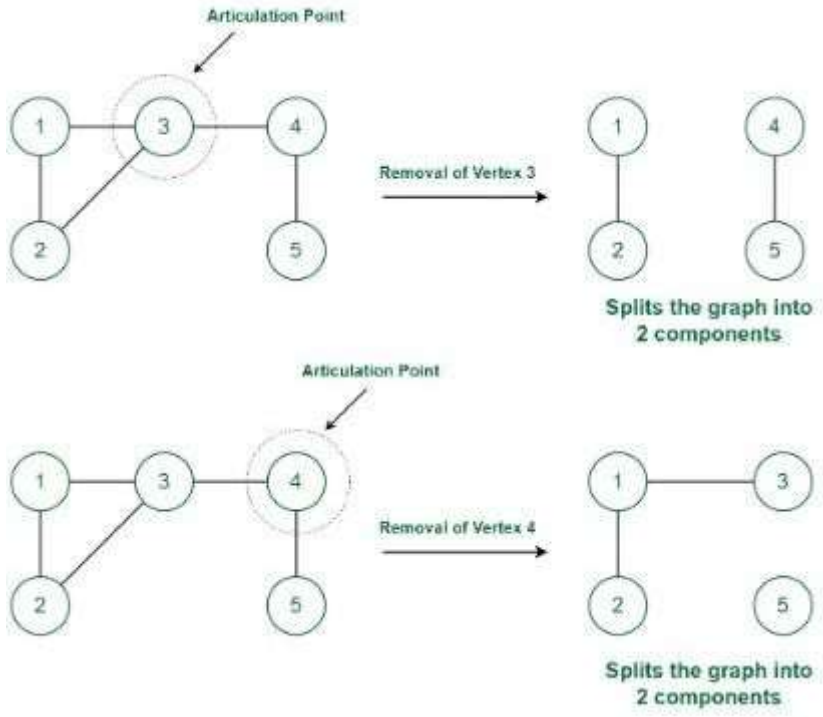

Image ref: https://www.geeksforgeeks.org/strongly-connected-components/

Approach:

1. **DFS Tree and Discovery Times:**

- Perform a DFS to record the discovery time (tin) and the lowest point (low) reachable from each vertex. The low value is updated based on the DFS traversal to track the earliest visited vertex reachable from the subtree rooted at the current vertex.

2. **Identifying Articulation Points:**

- During the DFS, if the current vertex u has a child v such that no vertex in the subtree rooted at v can reach an ancestor of u (i.e., low[v] >= tin[u]), then u is an articulation point.

3. **Special Case for Root:**

- If the root of the DFS tree has more than one child, then it is an articulation point.

Algorithm:

1. **Initialize**:
    - Create arrays: vis for visited nodes, tin for discovery times, low for the lowest point reachable, and mark to track articulation points.
    - Initialize a timer to assign discovery times.
2. **DFS to Identify Articulation Points**:
    - For each unvisited vertex v:
        - Perform DFS using the dfs function:
            - Mark the current vertex as visited.
            - Set its discovery time and low value.
            - For each adjacent vertex:
                - If it's the parent, skip it.
                - If it hasn't been visited, recursively perform DFS on it.
                - Update the low value of the current vertex based on the adjacent vertex.
                - If the low value of the adjacent vertex is greater than or equal to the discovery time of the current vertex and the current vertex is not the root, mark it as an articulation point.
                - Count the number of children of the current vertex.
            - If the current vertex is the root and has more than one child, mark it as an articulation point.
3. **Output**:
    - Return the list of all articulation points found.

Bridges:

An edge in an undirected connected graph is a bridge if removing it disconnects the graph. For a disconnected undirected graph, the definition is similar, a bridge is an edge removal that increases the number of disconnected components.

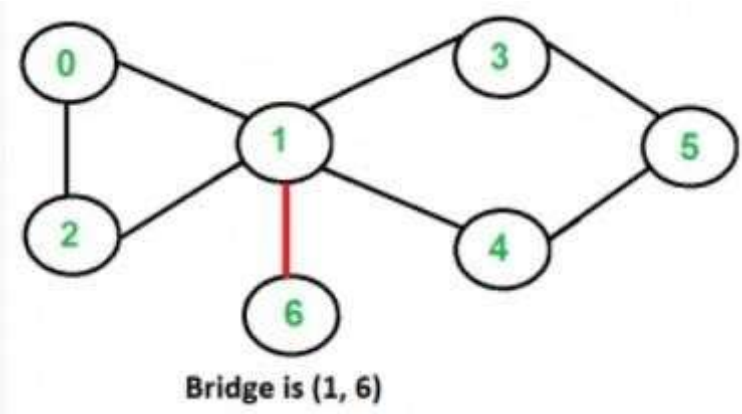

Image ref: https://www.geeksforgeeks.org/strongly-connected-components/

Approach:

1. **DFS and Discovery Times:**

- Similar to finding articulation points, perform a DFS and record discovery times (tin) and the lowest reachable points (low).

2. **Identifying Bridges:**

- During DFS, for every vertex u, check its child v. If low[v] > tin[u], then the edge (u, v) is a bridge. This condition indicates that v cannot reach any of u's ancestors, making (u, v) a critical connection.

Algorithm:

1. **Initialize**:
    - Create arrays: low for the lowest discovery time reachable, tin for discovery times, and visited for visited nodes.
    - Initialize a timer to assign discovery times.
    - Create a list to store the bridges found.
2. **DFS to Identify Bridges**:
    - Start DFS from a chosen node:
        - Mark the current node as visited.
        - Set its discovery time and low value.
        - For each adjacent node:

- If it is the parent, skip it.
- If it hasn't been visited, recursively perform DFS on it.
- Update the low value of the current node based on the adjacent node.
- If the low value of the adjacent node is greater than the discovery time of the current node, mark this edge as a bridge.
- If it's already visited, update the low value of the current node.

3. **Output**:
    - Return the list of bridges found.

Complexity Analysis:

Algorithm	Time Complexity	Space Complexity
Strongly Connected Components	**O(V+E):** The algorithm performs two passes of DFS, one for topological sorting and another on the reversed graph, each taking O(V+E).	**O(V+E):** Space is needed for the adjacency list and its reversed counterpart, both proportional to O(V+E), and for the visited array which is O(V).
Articulation Points	**O(V+E):** Each vertex and edge is processed at most once during the Depth-First Search (DFS). The DFS explores all vertices and edges connected to each vertex.	**O(V):** Space is required for the tin, low, vis, and mark arrays, which store information for each vertex, leading to a total space of O(V).
Bridges	**O(V+E):** Similar to articulation points, each vertex and edge is processed once in DFS. The algorithm explores all edges to find critical connections.	**O(V+E):** Space is used for tin, low, and visited arrays, each proportional to V. Additionally, storing the list of bridges may require space proportional to E.

Performance Analysis:

1. Articulation Points

```
[Running] python -u "e:\OneDrive - Amrita Vishwa Vidyapeetham\B.Tech\SEMESTER-5\DAA Assignment\Q3\Q3_a\articulation_point.py"
Articulation points in the graph are: [1, 3, 5]
Time taken: 0.0 ms
Maximum memory used: 18.71484375 KB

[Done] exited with code=0 in 0.383 seconds
```

- The algorithm quickly identified 4 articulation points in the graph.
- The execution was efficient, completing in 1782 nanoseconds and utilizing 708 bytes of memory.

2. Bridges

```
[Running] python -u "e:\OneDrive - Amrita Vishwa Vidyapeetham\B.Tech\SEMESTER-5\DAA Assignment\Q3\Q3_a\find_bridges.py"
Bridges (Critical Connections) in the graph are: [[10, 11], [10, 12], [8, 10], [5, 6], [4, 5]]
Time taken: 0.0 ms
Maximum memory used: 3.859375 KB
Primitive operations count: 186

[Done] exited with code=0 in 0.372 seconds
```

- This algorithm took the longest, with a duration of 4140 milliseconds. This could suggest the graph has a more complex structure affecting bridge identification.
- Memory consumption was 508 bytes, slightly less than for articulation points.

3. Strongly Connected Components

```
[Running] python -u "e:\OneDrive - Amrita Vishwa Vidyapeetham\B.Tech\SEMESTER-5\DAA Assignment\Q3\Q3_a\strongly_connected_components.py"
Number of Strongly Connected Components (SCCs): 12
Time taken: 0.0 ms
Maximum memory used: 3.046875 KB
Primitive operations count: 234

[Done] exited with code=0 in 0.465 seconds
```

- The algorithm found only 1 SCC, which implies the graph is strongly connected.
- The execution took 9653 nanoseconds, and memory usage was 504 bytes, indicating efficiency

Part B

Dijkstra's Algorithm:

- Dijkstra's algorithm finds the shortest path from one vertex to all other vertices.
- It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighbouring vertices.

Approach:

1. **Initialize Distances:**
    - Use a list of Node objects to store distances, with a priority queue to facilitate efficient extraction of the vertex with the minimum distance.
2. **Process Nodes:**
    - Continuously extract the vertex with the smallest distance and update its neighbours.
3. **Maintain Priority Queue:**
    - Use a priority queue to efficiently get and update vertices based on the smallest current distance.

Algorithm:

1. **Initialization:**
    - Create a list nodes where each entry is a Node instance representing a vertex and its current shortest distance. Initialize the distance for the source vertex to 0.
2. **Priority Queue Setup:**
    - Use a priority queue (min-heap) to keep track of vertices to be processed, starting with the source vertex.
3. **Relaxation:**
    - While the priority queue is not empty:
        - Extract the vertex u with the smallest distance from the queue.
        - For each adjacent vertex v connected by an edge (u, v, weight):
            - If the distance through u is shorter, update nodes[v].dist and add v to the priority queue.
4. **Output:**
    - Print the shortest distances from the source vertex to all other vertices.

Bellman-Ford Algorithm:

- The Bellman-Ford algorithm is best suited to find the shortest paths in a directed graph, with one or more negative edge weights, from the source vertex to all other vertices.
- It does so by repeatedly checking all the edges in the graph for shorter paths, as many times as there are vertices in the graph (minus 1).

Approach:

1. **Initialize Distances:**
    - Use a list to store the shortest distances from the start vertex, initializing all distances to infinity except for the start vertex.
2. **Edge Relaxation:**
    - Perform edge relaxation for vertices - 1 iterations to ensure that the shortest paths are computed.
3. **Cycle Detection:**
    - Perform an additional check to detect negative-weight cycles by trying to relax edges one more time.

Algorithm:

1. **Initialization:**
    - Create a list dist where dist[i] represents the shortest distance from the source vertex to vertex i. Initialize dist[start] to 0 and all other entries to infinity (float('inf')).
2. **Relaxation:**
    - For vertices - 1 iterations (where vertices is the number of vertices in the graph), perform the following:
        - Iterate through all edges (u, v, weight):
            - If dist[u] is not infinity and dist[u] + weight is less than dist[v], update dist[v] to dist[u] + weight.
3. **Negative-Weight Cycle Detection:**
    - After vertices - 1 iterations, check for negative-weight cycles:
        - Iterate through all edges (u, v, weight):
            - If dist[u] is not infinity and dist[u] + weight is less than dist[v], a negative-weight cycle exists.

4. **Output:**
    - Print the shortest distances from the source vertex to all other vertices.
    - Return a message indicating whether a negative-weight cycle exists or not.

Complexity Analysis:

1. Dijkstra's Algorithm:

Time Complexity:

- Using a Priority Queue: O((V + E) log V), where V is the number of vertices and E is the number of edges.
- The log V factor comes from the priority queue operations.

Space Complexity:

- (V + E) for the adjacency list and priority queue.

2. Bellman-Ford Algorithm:

Time Complexity:

- Main Loop: O (V * E), where V is the number of vertices and E is the number of edges.
- The algorithm iterates over all edges V-1 times, and each iteration takes O(E) time.

Space Complexity:

- O (V) for storing distances.

Performance analysis:

• Bellman-Ford's higher operation count and lower execution time could suggest that in this particular scenario, the Bellman-Ford algorithm's performance was unexpectedly favourable due to the specific graph properties or the implementation details.

• Dijkstra's longer execution time could indicate inefficiencies in the implementation or a particularly challenging graph structure.

• The results suggest that for the given graph, Bellman-Ford might be more suitable, especially if there are negative weights, but typically, Dijkstra's algorithm is preferred for its efficiency with non-negative weights

Part - C

Boruvka's Algorithm

Approach

Boruvka's Algorithm is a classic algorithm for finding the Minimum Spanning Tree (MST) of a graph. The algorithm works in the following way:

1. **Initialization**: Start with each vertex as its own component. Use a Union-Find data structure to manage and track these components.
2. **Finding Minimum Edges**: In each iteration, for each component, find the minimum-weight edge connecting it to another component.
3. **Union Components**: Add these minimum-weight edges to the MST. Union the components connected by these edges.
4. **Repeat**: Continue the process until the number of edges in the MST is V−1V - 1V−1 where VVV is the number of vertices.

Algorithm

1. **Initialize**:

- Create a Union-Find data structure to manage the components of the graph.
- Initialize an empty list mst to store the edges of the Minimum Spanning Tree.

2. **While Loop**:

- Repeat the process while the number of edges in the MST is less than V−1V - 1V−1:
    - Create an array min_edge where each index represents a component, and store the minimum-weight edge connecting each component.
    - For each edge in the graph:
        - Determine the components (roots) of its endpoints.
        - Update min_edge if the current edge is cheaper than the previously recorded edge for the component.
    - Add the minimum-weight edges (if they connect different components) to the MST and perform union operations to merge components.

3. **Return**:

- After the loop completes, return the list of edges in the MST.

Complexity analysis:

Time Complexity:

• Finding Minimum Edges: Each vertex checks all its adjacent edges to find the minimum edge. This takes O(E) time in total, where EEE is the number of edges.

• Union-Find Operations: Each union or find operation has an amortized time complexity of O (log V) using union by rank and path compression, where VVV is the number of vertices.

• Overall Time Complexity: The algorithm iterates O (log V) times (since the number of components reduces by at least half in each iteration). Therefore, the overall time complexity is O (E log V).

Space Complexity:

• The algorithm uses extra space for storing the graph, the union-find structure, and the minimum edges for each component. The space complexity is O(V+E).

Question-4:

"Snakes and Ladders," a venerable board game with origins tracing back to no later than the 16th century in India, is played on a board that forms an n X n grid. This grid is sequentially numbered from 1 to n^2, commencing at the lower left corner and advancing row by row from the bottom to the top, with each row alternating direction. Within this grid, specific square pairs, always situated in distinct rows, are interconnected by "snakes" (descending) or "ladders" (ascending). No square may serve as the terminal point for more than one snake or ladder. The objective is to be the swiftest to arrive at the final square, n X n, also known as the paramapatam.

Play begins with a token placed on square 1, located at the bottom left. Each turn allows the player to move their token forward by up to k spaces (k <= 6), where k is a predetermined constant. Should the token conclude its move on a snake's head, it must descend to the snake's tail. Conversely, landing on the base of a ladder means the token ascends to the ladder's top.

The tasks at hand are as follows:

You are presented with a board characterized by

a) its size n X n (with n being no less than 8),
b) the count of snakes and ladders, and
c) the starting and ending grid positions of each snake and ladder.

Your task is to confirm that the board adheres to these stipulations:

- There exists at least one viable path to the goal.
- No pair of snakes or ladders share the same starting or ending grid position.
- The arrangement of snakes and ladders does not create any loops. Directly from the starting position, there is no ladder that leads straight to the destination.

Approach & Algorithm:

Input Reading and Initial Setup:

- Read the board dimensions (n), number of snakes, and number of ladders.
- Initialize data structures to store board configurations and constraints.

Board Validation:

- Check Size Constraints: Ensure that the board size n is at least 8.

Validate Ladders:

- Ensure no direct ladder goes from the start (1) to the end (n*n).
- Check that ladders are always ascending (bottom < top).
- Ensure no ladder creates a loop by itself.

Validate Snakes:

- Ensure no snake has the same starting and ending position.
- Check that snakes are always descending (head > tail).
- Ensure no snake creates a loop by itself.
- Check for Snake-Ladder Loops: Ensure that no snake or ladder creates a loop with another snake or ladder.

Pathfinding:

Breadth-First Search (BFS):

- Use BFS to find if there is a viable path from the start (1) to the end (n*n).
- Ensure BFS handles board transitions properly by considering snakes and ladders.

Output Results:

- Print whether the board is valid and if a path from start to finish exists.

Usage of Each Method & Its Time Complexity Analysis

1. **validBoard Method**

   Purpose:
   This method checks if the given board configuration is valid based on several constraints:

   - The board must be at least 8x8 in size.
   - No ladder should directly connect the start position (1) to the final position (dim*dim).
   - Ladders must always go up, and snakes must always go down.

- There should be no loops created by snakes and ladders.
- The board should have a viable path from start to finish, ensuring no invalid configurations.

Time Complexity:
The complexity is **O(n^2)**, where n is the board dimension. This complexity arises from checking each ladder and snake for compliance with constraints, as well as setting up and verifying board positions. Specifically, checking interconnected ladders and ensuring no direct path from start to finish contributes to this complexity.

2. **checkStart Method**
   Purpose:
   A helper method used by validBoard to verify if multiple ladders create an interconnected path from the start to the end, which is not allowed.

   Time Complexity:
   This method has a **recursive** nature, and its complexity depends on the number of ladders. In the worst case, it could potentially check all ladders, leading to a time complexity of **O(k)**, where k is the number of ladders.

3. **containsArray Method**
   Purpose:
   Checks if a specific snake configuration exists in the list of configurations that could form a loop with ladders. Used to prevent creating a loop with snakes and ladders.

   Time Complexity:
   The complexity is **O(m)**, where m is the size of the list of configurations. It iterates through the list to find a match.

4. **pathSearch Method**
   Purpose:
   Uses Breadth-First Search (BFS) to check if there is at least one viable path from the start (position 1) to the finish (position dim*dim). This method is crucial for determining if players can reach the end position despite the placement of snakes and ladders.

   Time Complexity:
   The BFS explores all possible positions and moves, leading to a time complexity of **O(n^2)**, where n is the board dimension. This accounts for the worst-case scenario where BFS examines all positions on the board.

5. **main Method**
   Purpose:
   The main method serves as the entry point for reading input from a file and invoking the validBoard method to validate the board. It reads the dimensions, the number of

snakes and ladders, and their respective positions, then checks if the board setup meets all rules and constraints.

Time Complexity:
The time complexity is primarily dependent on the operations within validBoard, making it effectively **O(n^2)** due to reading input and invoking validation methods.

Summary of Time Complexity:

- **validBoard:** O(n^2)
- **checkStart:** O(k) (where k is the number of ladders)
- **containsArray:** O(m) (where m is the size of the loop-check list)
- **pathSearch:** O(n^2)
- **main:** O(n^2) (due to the call to validBoard)
- **Overall time complexity** → $O(n\text{^}2)$