# Introduction to Sorted Sorted Arrays

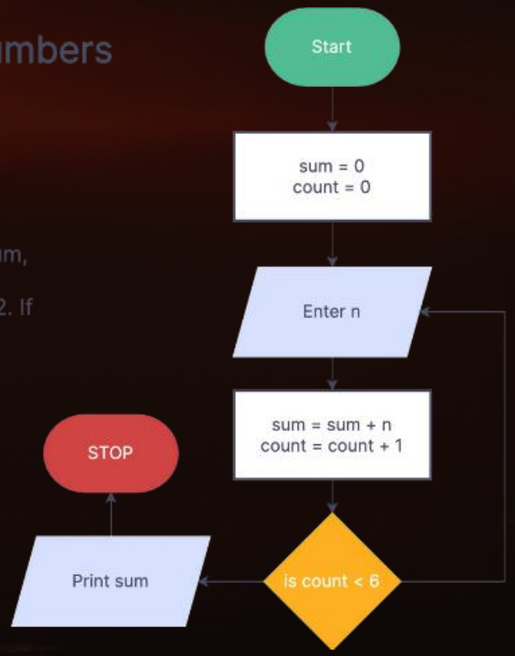
A sorted array is a data structure where elements are arranged in a specific order, either ascending or descending. This order allows for efficient searching and retrieval of elements.

**by keshav reddy**

# Sorting Algorithms Overview

Find the sum of 6 numbers algorithm

1. Initialize sum = 0 and count = 0
2. Enter n
3. Find sum + n and assign it to sum, then increment count by 1.
4. Is count < 6. If YES, go to step 2. If No, print sum.




1 **Bubble Sort**

This algorithm repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.

2 **Insertion Sort**

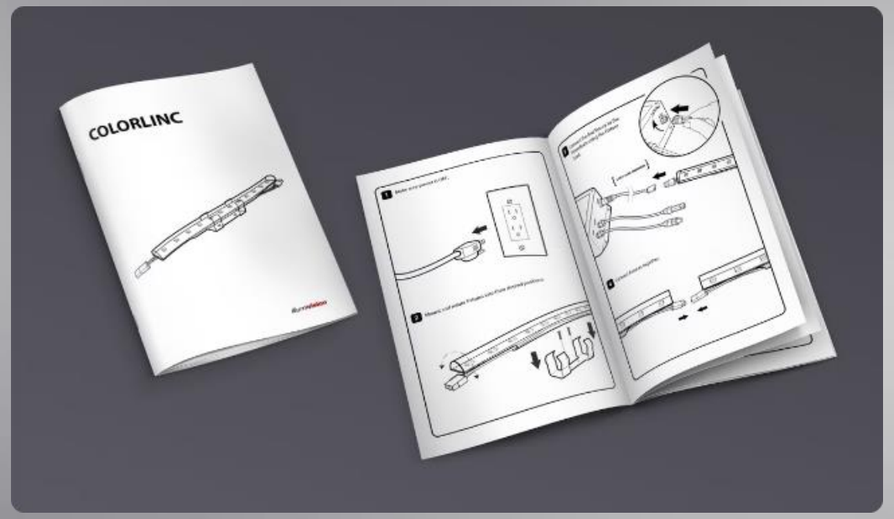
It iterates through the list, inserting each element at its correct position in the already sorted sublist.

3 **Selection Sort**

This algorithm repeatedly selects the minimum element from the unsorted sublist and swaps it with the first element of the sublist.

4 **Merge Sort**

This algorithm divides the list into smaller sublists, sorts them recursively, and then merges them back together.




# Sorting through Instructions

1

## Instruction Input

The user provides a series of instructions to manipulate the array.

2

## Instruction Parsing

The program interprets the instructions, identifying operations and target elements.

3

## Element Manipulation

The instructions are applied to the array, performing operations like swapping, inserting, or deleting elements.

# Implementing Sorting Instructions

## Instruction Set

Define a set of instructions that can be used to manipulate the array, such as "swap," "insert," "delete," or "move."

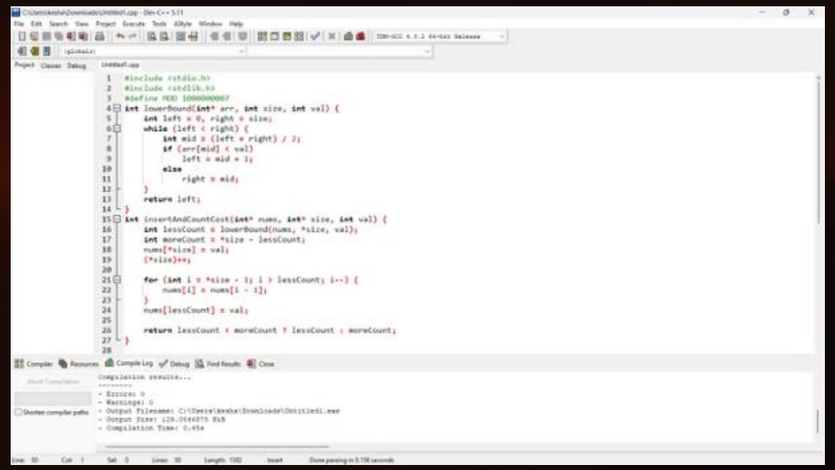
## Instruction Parsing

Implement a parser that can read and interpret the instructions provided by the user, identifying the operation and the target elements.

## Array Manipulation

Implement functions for each instruction that modify the array according to the parsed instructions.

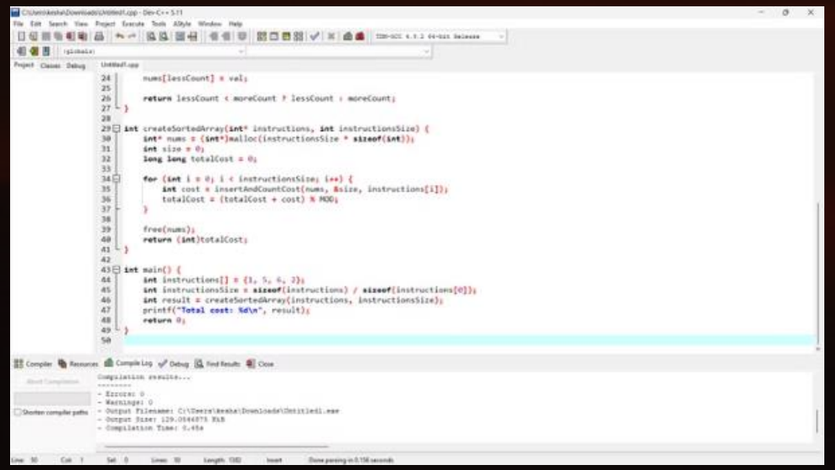
# Code and Output



```c
#include <stdio.h>
#include <stdlib.h>
#define MOD 1000000007
int lowerBound(int* arr, int size, int val) {
    int left = 0, right = size;
    while (left < right) {
        int mid = (left + right) / 2;
        if (arr[mid] < val)
            left = mid + 1;
        else
            right = mid;
    }
    return left;
}
int insertAndCountCost(int* nums, int* size, int val) {
    int lessCount = lowerBound(nums, *size, val);
    int moreCount = *size - lessCount;
    nums[*size] = val;
    (*size)++;

    for (int i = *size - 1; i > lessCount; i--) {
        nums[i] = nums[i - 1];
    }
    nums[lessCount] = val;

    return lessCount < moreCount ? lessCount : moreCount;
}
```



```c
    nums[lessCount] = val;

    return lessCount < moreCount ? lessCount : moreCount;
}

int createSortedArray(int* instructions, int instructionsSize) {
    int* nums = (int*)malloc(instructionsSize * sizeof(int));
    int size = 0;
    long long totalCost = 0;

    for (int i = 0; i < instructionsSize; i++) {
        int cost = insertAndCountCost(nums, &size, instructions[i]);
        totalCost = (totalCost + cost) % MOD;
    }

    free(nums);
    return (int)totalCost;
}

int main() {
    int instructions[] = {1, 5, 6, 2};
    int instructionsSize = sizeof(instructions) / sizeof(instructions[0]);
    int result = createSortedArray(instructions, instructionsSize);
    printf("Total cost: %d\n", result);
    return 0;
}
```

C:\Users\kesha\Downloads\U

```
Total cost: 1

--------------------------------
Process exited after 0.02792 seconds with return value 0
Press any key to continue . . .
```

# Optimizing Instruction-based Sorting

1

## Instruction Batching

Group similar instructions together to minimize the number of array accesses.

2

## Instruction Preprocessing

Analyze the instructions beforehand to identify potential optimizations.

3

## Data Structure Optimization

Use efficient data structures like heaps or binary search trees to store the array and enable faster manipulation.

# Handling Edge Cases and Errors

| | |
|---|---|
| Empty Array | Handle the case where the input array is empty. |
| Invalid Instructions | Validate user input to ensure that the instructions are valid and correctly formatted. |
| Out-of-Bounds Access | Prevent errors that occur when trying to access elements outside the array bounds. |

# Performance Analysis of Instruction-Based Sorting Sorting

1

## Best Case

The best-case scenario for an instruction-based sorting algorithm occurs when the input array is already sorted in ascending order. In this case, the algorithm can simply apply the given instructions without needing to perform any significant rearrangements, making the time complexity O(n), where n is the size of the array.

2

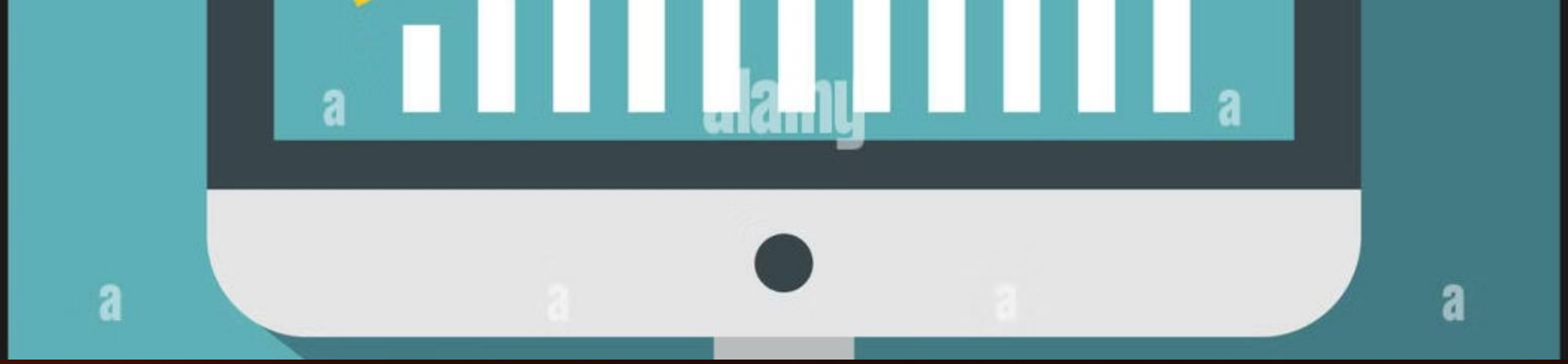
## Worst Case

The worst-case scenario happens when the input array is in descending order, the reverse of the desired sorted state. This requires the algorithm to perform the maximum number of operations, such as repeatedly swapping or inserting elements, to sort the array. In the worst case, the time complexity of the algorithm becomes O(n^2), resulting in a significant performance impact.

3

## Average Case

The average case refers to the typical performance of the algorithm when the input array is in a random order. In this scenario, the number of operations required to sort the array falls somewhere between the best and worst cases, with a time complexity of O(n log n). This is the expected performance for most inputs, making it a crucial metric for understanding the overall efficiency of the instruction-based sorting algorithm.



# Analyzing Time and Space Complexity

## Time Complexity

Analyze the time taken by the algorithm to sort the array based on the number of elements.

## Space Complexity

Determine the amount of additional memory used by the algorithm, excluding the space occupied by the input array.

# Conclusion and Key Takeaways Takeaways

Instruction-based sorting allows for flexible and customizable array sorting by providing a clear and concise way to specify the desired order. By carefully considering edge cases, optimizing instructions, and analyzing complexity, we can create efficient and reliable sorting solutions.