Cost-Effective Sorting: Minimizing Insertion Cost to Create a Sorted Array

# CAPSTONE PROJECT REPORT

***Submitted by***

D. Madhan kumar (192111559)

D.Charan (192211577)

P.Pavan kumar (192211590)

*Under the guidance of*

Ms. Joy Priyanka

# 

***in partial fulfillment for the completion of course***

# CSA0637- DESIGN AND ANALYSIS OF ALGORITHMS FOR

# DYNAMIC PROGRRAMMING

**SIMATS ENGINEERING THANDALAM**



# APRIL 2024

**Problem statement:**

You are presented with an integer array called 'instructions'. Your task is to construct a sorted array from the elements in 'instructions'. Beginning with an empty container named 'nums', you sequentially insert each element from 'instructions' into 'nums' from left to right. The cost of each insertion is determined as the minimum of the number of elements currently in 'nums' that are strictly less than the current element, and the number of elements currently in 'nums' that are strictly greater than the current element. After inserting all elements from 'instructions' into 'nums', your goal is to compute and return the total cost incurred during this process. It's noteworthy that since the resulting answer might be substantial, you must return it modulo 10^9+7.

# Proposed design work:

The approach emphasizes inserting elements from 'instructions' into 'nums' to create a sorted array, minimizing insertion costs by considering element counts. The objective is to compute the total cost, ensuring accuracy and handling large results with modulo 10^9+7.

1. **Identifying the key components:**
   * **Input Array 'instructions':** The given integer array from which a sorted array needs to be constructed.
   * **Container nums:** An empty container where elements from 'instructions' are sequentially inserted to create the sorted array.
   * **Insertion process**: Iteratively inserting each element from 'instructions' into 'nums' from left to right.
   * **Insertion cost:** Determining the cost of each insertion as the minimum of the count of elements in 'nums' that are strictly less than the current element and the count of elements in 'nums' that are strictly greater than the current element.
   * **Total cost calculation:** Summing up the costs of all insertions to compute the total cost incurred during the construction of the sorted array.
   * **Handling Large Results:** Considering the possibility of a substantial result and ensuring it's returned modulo 10^9+7.

# Functionality:

* **Insertion of Elements into a Sorted Array:** The insert function allows for the insertion of elements into a sorted array while maintaining its sorted order**.** It iterates through the array to find the correct position for the new element and shifts elements to make space for it.
* **Calculation of Insertion Cost**: It counts the number of elements in the array that are strictly less than and strictly greater than the current element.
* **Total Cost Computation:** computes the total cost incurred during the construction process. It iterates through each element in the 'instructions' array, calculates the insertion cost using the calculateCost function, and sums up these costs.
* **Handling Large Results:** The result of the total cost computation is returned modulo 10^9+7 to handle large results and ensure it fits within the bounds of a 32- bit signed integer.

# Architectural Design:

* **Calculation of insertion cost:**

int calculateCost(int\* nums, int size, int val) { int lessThan = 0, greaterThan = 0;

for (int i = 0; i < size; i++) { if (nums[i] < val) {

lessThan++;

} else if (nums[i] > val) { greaterThan++;

}

}

return (lessThan < greaterThan) ? lessThan : greaterThan;

}

# Insertion of elements into sorted array:

void insert(int\* nums, int\* size, int val) { int index = \*size;

while (index > 0 && nums[index - 1] > val) { nums[index] = nums[index - 1];

index--;

}

nums[index] = val; (\*size)++;

}

# Total cost computation:

int totalCost(int\* instructions, int instructionsSize) { int nums[1000];

int size = 0;

int totalCost = 0;

for (int i = 0; i < instructionsSize; i++) {

int cost = calculateCost(nums, size, instructions[i]); totalCost += cost;

insert(nums, &size, instructions[i]);

}

return totalCost % 1000000007;

}

# Main function:

int main() {

int instructions[1000]; int instructionsSize;

// Input the size of the instructions array

printf("Enter the number of elements in the instructions array: "); scanf("%d", &instructionsSize);

// Input the elements of the instructions array printf("Enter the elements of the instructions array:\n"); for (int i = 0; i < instructionsSize; i++) {

scanf("%d", &instructions[i]);

}

int result = totalCost(instructions, instructionsSize); printf("Total Cost: %d\n", result);

return 0;

}

# UI Design:

1. **Layout Design:**

**Flexible layout:** The system begins with an empty container named 'nums' where elements from the 'instructions' array are sequentially inserted. The insertion process is managed to ensure that elements are added in sorted order. The cost of each insertion is calculated based on the number of elements in 'nums' that are strictly less than or greater than the current element. This flexible layout enables a clear understanding of the flow from data input to sorted array construction.

# User Friendly:

* + Use an intuitive layout that guides users through the process of providing input and viewing results.
  + Clearly label input fields and provide informative messages or tooltips.

# Color selection:

Background: Light neutral color (e.g., white or off-white) for the overall background of the code.

Text: Black or dark gray for all textual elements including comments, function names, variable names, and code statements.

Function names: Blue for function names such as calculateCost, insert, totalCost, and main to distinguish them from other elements.

Variable names:Green for variable names like nums, size, val, instructions, and instructionsSize to make them stand out.

Control flow: Gray for loop constructs like for and while loops.

Input prompts: Orange for input prompts such as "Enter the number of elements in the instructions array: ".

User input: Purple for user input text such as the entered elements of the instructions array.

# Feasable elements used:

**Elements positioning:** In the user interface, the elements are positioned as follows: A section labeled "Instructions" is provided where users can input the elements of the integer array. Each instruction is represented by a separate input field. Below the instruction input fields, there is a button labeled "Calculate Cost" that initiates the computation process. Finally, the result of the calculation, the total cost, is displayed below the button.

**Accessibility:** Ensure that all UI elements are accessible, especially for users with disabilities.

Use appropriate contrast and text sizes for readability.

# Element function:

**Instruction Input Field:** Allows users to input individual integers representing instructions for constructing the sorted array.

**Calculate Cost Button:** Triggers the computation of the total cost incurred during the construction process based on the provided instructions.

**Total Cost Display:** Shows the result of the cost calculation, i.e., the total cost incurred during the construction process.

These elements collectively facilitate the input of instructions, computation of the total cost, and display of the result to the user.

# Login template:

#include <stdio.h> #include <stdlib.h>

#define MOD 1000000007

int compare(const void \*a, const void \*b) { return (\*(int \*)a - \*(int \*)b);

}

int minCost(int\* instructions, int instructionsSize) {

int\* nums = (int\*)malloc(instructionsSize \* sizeof(int)); int cost = 0;

int i;

for ( i = 0; i < instructionsSize; i++) { int lowerCount = 0, greaterCount = 0; int j;

for ( j = 0; j < i; j++) {

if (instructions[j] < instructions[i]) { lowerCount++;

} else if (instructions[j] > instructions[i]) { greaterCount++;

}

}

cost = (cost + (lowerCount < greaterCount ? lowerCount : greaterCount)) % MOD; nums[i] = instructions[i];

qsort(nums, i + 1, sizeof(int), compare);

printf("Sorted list after inserting %d: ", instructions[i]); int k;

for ( k = 0; k <= i; k++) { printf("%d ", nums[k]);

}

printf("\n");

}

free(nums); return cost;

}

int main() { int n;

printf("Enter the number of elements in the array: "); scanf("%d", &n);

int\* instructions = (int\*)malloc(n \* sizeof(int));

printf("Enter the elements of the array:\n"); int i;

for ( i = 0; i < n; i++) { scanf("%d", &instructions[i]);

}

int totalCost = minCost(instructions, n); printf("Total cost of insertion: %d\n", totalCost);

free(instructions); return 0;

}

# Conclusion:

In conclusion, this task involves constructing a sorted array from a given set of instructions. The process entails sequentially inserting each element from the instructions into an initially empty container, while calculating the cost of each insertion based on the number of elements in the container that are strictly less than or greater than the current element being inserted. After inserting all elements, the total cost incurred during the construction process needs to be

computed and returned, considering the possibility of a substantial result by returning it modulo 10^9+7.

To achieve this, we can develop a C program that takes input for the instructions array, iteratively inserts elements into a sorted array, and calculates the cost of each insertion. Finally, the program computes and outputs the total cost incurred. Overall, the goal is to efficiently construct the sorted array while minimizing the total cost, ensuring that the resulting array adheres to the specified insertion criteria and handling the potential computational complexity of large result values.