Data Structures and Algorithms
Assignment-2 (Report)

Aditya Prasad
2022036

Q1)

a)

Given the constraints of custom data structure DS1, Selection Sort emerges as the optimal sorting algorithm. It iteratively identifies and places the smallest element from the unsorted section of the array at the beginning. DS1 facilitates the necessary operations of accessing and modifying items at specific indices, crucial for this process. Selection Sort's efficiency lies in minimizing the number of swaps (or set operations), which is particularly beneficial considering the costly nature of the set operation in DS1, requiring O(k log k) time. Therefore, its capability to execute a maximum of k swaps is advantageous in this scenario.

b)

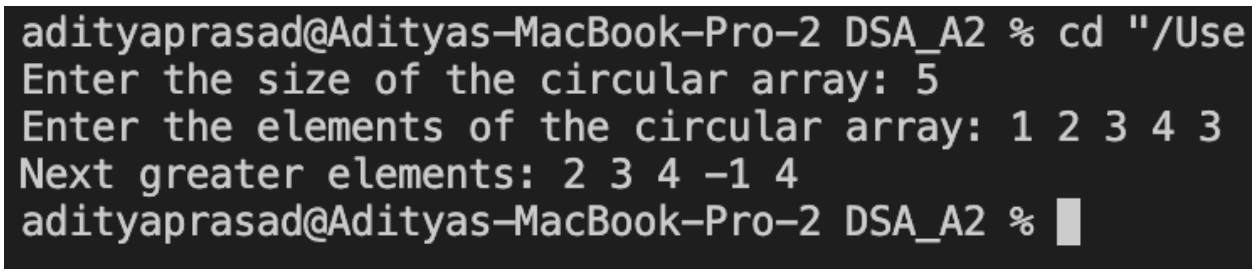
Merge Sort stands out as the optimal sorting approach due to its time complexity of O(n log n) for comparison-based sorting methods. It excels because comparing pairs of items only requires O(log n) time. However, it's worth noting that the actual time complexity is O(n log^2 n) because the comparison operation itself demands O(log n) time.

c)

Insertion sort would be the optimal sorting approach here.When a sorted array undergoes merely log log n swaps, it becomes nearly sorted. Insertion Sort would thus emerges as an excellent option due to its time complexity of Θ(n) for nearly sorted arrays.

Q2)

OUTPUT:



The provided code defines two data structures, Stack and Queue, along with a function nextGreaterElements to find the next greater elements in a circular array. Stack is implemented using a linked list and supports push, pop, peek, and isEmpty operations, while Queue is also implemented using a linked list and supports enqueue, dequeue, peek, and isEmpty operations. The nextGreaterElements function takes an array of integers representing a circular array and returns an array of integers representing the next greater elements for each element in the circular array. It utilizes a stack to efficiently find the next greater element for each element in the array by iterating through the array twice. The function iterates backward through the array, maintaining a stack of indices of elements whose next greater element is yet to be determined. It updates the result array accordingly and returns it. Finally, the main function takes input from the user, calls nextGreaterElements, and prints the resulting array of next greater elements.

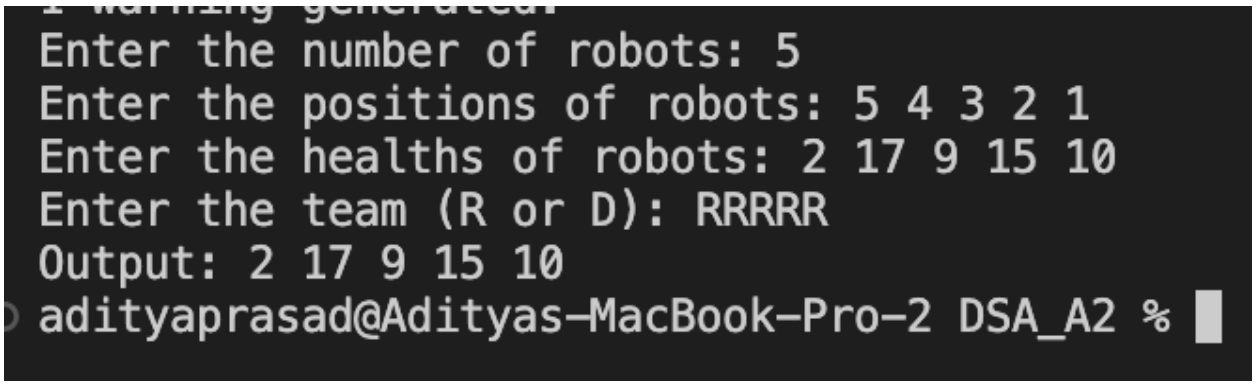
Q3)
OUTPUT

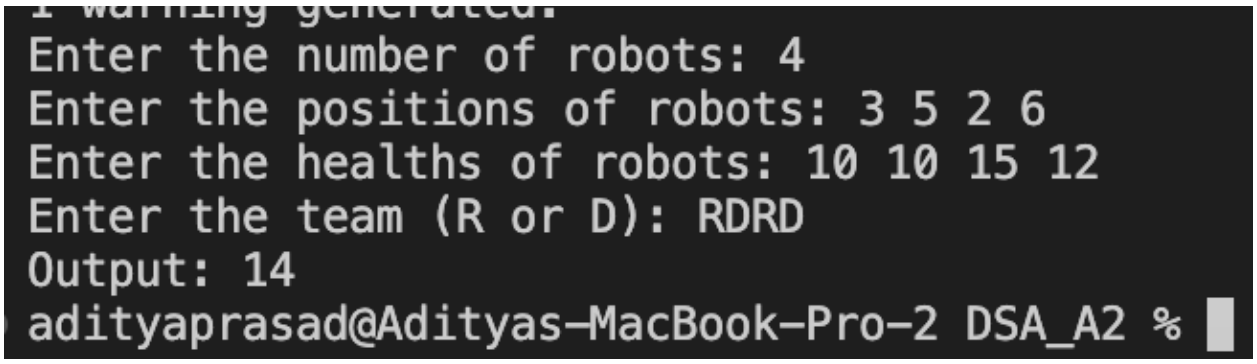
```
adityaprasad@Adityas-MacBook-Pro-2 DSA_A2 % cd "/Users/ad
Recipe: BBBSSC
Number of bread, salmon, and corn(n_b, n_s, n_c): 6 4 1
Price of bread, salmon, and corn(n_b, n_s, n_c): 1 2 3
Amount(r): 4
OUTPUT: 2
adityaprasad@Adityas-MacBook-Pro-2 DSA_A2 % cd "/Users/ad
Recipe: BSC
Number of bread, salmon, and corn(n_b, n_s, n_c): 1 1 1
Price of bread, salmon, and corn(n_b, n_s, n_c): 1 1 3
Amount(r): 1000000000000
OUTPUT: 200000000001
adityaprasad@Adityas-MacBook-Pro-2 DSA_A2 %
```

The provided code implements a program to calculate the maximum number of sushi rolls that can be made within a given budget, considering the availability of ingredients and their prices. It begins by prompting the user for input regarding the sushi recipe, the quantities of ingredients (bread, salmon, and corn), their respective prices, and the available budget. Using binary search, the program

iteratively adjusts the number of sushi rolls produced until it finds the maximum possible quantity within the budget constraints. It does this by calculating the cost of producing a certain number of sushi rolls and comparing it against the budget. If the cost is within budget, it increases the lower bound of the search range; otherwise, it decreases the upper bound. Finally, the program outputs the maximum number of sushi rolls that can be made within the budget. Overall, the code efficiently determines the optimal number of sushi rolls to produce given the available resources and budget.

Q4)
OUTPUT





This C++ code is designed to simulate a scenario involving robots from two teams, 'R' and 'D', engaging in a battle. The code takes input regarding the number of robots, their positions, healths, and respective teams. It then processes this data to determine the health of surviving robots after the battle. The surviving robots' health is calculated based on the specified rules of engagement. The program utilizes a custom Stack data structure to handle the processing logic efficiently. Additionally, it employs the Merge Sort algorithm to sort the robots' positions before performing battle calculations, ensuring optimal performance. Finally, the code outputs the healths of the surviving robots.

Overall, the program effectively models the battle scenario and computes the desired outcome.