DSAPS Assignment 2

Deadline: 28 September 2025, 11:59pm

Important Points

1. The assignment has 3 questions. You need to do ALL THREE questions. Each question weighs 100 marks.
2. Only C++ is allowed.
3. Directory structure to be followed for submission:

   ```
   202520xxxx_A2
   |--202520xxxx_A2_Q1.cpp
   |--202520xxxx_A2_Q2.cpp
   |--202520xxxx_A2_Q3.cpp
   |--README.md
   ```

   Replace your roll number in place of 202520xxxx.
4. Submission Format: Follow the above mentioned directory structure and zip the RollNo_A1 folder and submit RollNo_A2.zip on Moodle.

   Note: All submissions which are not in the specified format will be awarded 0 in the assignment.
5. C++ STL (including vectors) is not allowed for any of the questions unless specified otherwise in the question. So `#include <bits/stdc++.h>` is not allowed.
6. A brief description of your approach for solving each question (data structures, algorithms, and any optimizations used) and step-by-step instructions on how to compile and execute your code for each question should be mentioned in README.
7. You can ask queries by posting on Moodle.
8. Late Submission Rule: Deadlines for assignments are final and will not be extended. Late submissions cost 5% loss of marks for each late day up to 3 days (i.e., 5%, 10%, ...). Submissions beyond 3 days from the deadline shall receive 0 marks.

NOTE: In case of plagiarism in any of the questions, the students involved will be awarded -10 as the final marks of Assignment 1.

1 Question 1. Deque

1.1 A. [80 marks]

Problem Statement: Implement Deque

What is deque?

- Deque is the same as dynamic arrays with the ability to resize itself automatically when an element is inserted, with their storage being handled automatically by the container.
- They support insertion and deletion from both ends in amortized constant time.
- Inserting and erasing in the middle is linear in time.

Operations: The C++ standard specifies that a legal (i.e., standard-conforming) implementation of deque must satisfy the following performance requirements: (consider the data type as T)

1. **deque()** - initialize an empty deque. Time complexity: O(1)
2. **deque(n)** - initialize a deque of length n with all values as default value of T. Time complexity: O(n)
3. **deque(n, x)** - Initialize a deque of length n with all values as x. Time complexity: O(n)
4. void push_back(x) - append data x at the end. Time complexity: constant amortized time
5. **void pop_back()** - erase data at the end. Time complexity: constant amortized time
6. **void push_front(x)** - append data x at the beginning. Time complexity: constant amortized time
7. **void pop_front()** - erase an element from the beginning. Time complexity: constant amortized time
8. **T front()** - returns the first element(value) in the deque. If the first element is not present, return the default value of T. Time complexity: O(1)
9. **T back()** - returns the last element(value) in the deque. If the last element is not present, return the default value. Time complexity: O(1)
10. **T D[n]** - returns the nth element of the deque. You need to overload the [ ] operator. If nth element is not present return default value of T. Time complexity: O(1)
11. **bool empty()** - returns true if deque is empty else returns false. Time complexity: O(1)
12. **int size()** - returns the current size of deque. Time complexity: O(1)

13. **void resize(n)** - change the size dynamically to new size n. Time complexity: O(n)
    - If the new size n is greater than the current size of the deque, then insert new elements with the default value of T at the end of the queue.
    - If the new size n is smaller than the current size, then keep n elements from the beginning of the deque.
14. **void resize(n, d)** - change the size dynamically to new size n. Time complexity: O(n)
    - If the new size n is greater than the current size of the deque, then insert new elements with value d at the end of the queue.
    - If the new size n is smaller than the current size, then keep n elements from the beginning of the deque.
15. **void reserve(n)** : change the capacity of deque to n, if n > current capacity; otherwise do nothing. Time complexity: O(n)
16. **void shrink_to_fit()** - reduce the capacity of the deque to current size. Time Complexity: O(size())
17. **void clear()** - remove all elements of deque. Time complexity: O(n)
18. **int capacity()** - return the current capacity of deque. Time complexity: O(1)

Note:

1. Your deque should be generic type i.e. it should be datatype independent and can support primitive data types like integer, float, string, etc. Hint: Use template in C++ (Link)
2. For 1, 2 & 3 You can either define a constructor for the class or initialize the class object using void return type functions.
3. C++ STL is not allowed (including vectors, design your own if required)
4. D[0] - element at index 0 (i.e. first element from the front), D[1] - element at index 1 (i.e. second element from the front), D[-1] - element at last index (i.e. first element from the back), D[-2] - element at second last index (i.e. second element from the back)
5. Size of the deque is the number of elements currently present in your deque.
6. Capacity of the deque is the number of elements your deque can accommodate with currently held memory.
7. During Operation 1 both size and capacity of the deque should be set to zero.
8. If size is equal to capacity and a new element is inserted, then the capacity is doubled, unless capacity is zero, then it will become one.
9. If you have doubts about deciding the new capacity in any of the operations, refer to the behavior of the member functions of STL vector containers.

1.2 B. [20 marks]

Problem Statement: Using the deque implemented above, you are supposed to implement a randomized queue. Implement the following operations:

1. **void enqueue(x)**: add the item to the randomized queue
2. **T dequeue()**: remove and return a random item
3. **T sample()**: return a random sample (but do not remove it)

The above operations must be done in constant amortized time. Randomness of fetched items will be checked.

Randomized queue: A randomized queue is similar to a stack or queue, except that the item removed is chosen uniformly at random from items in the data structure.

Constant amortized time: Refer this (Link)

Note: Each element in the randomized queue must have an equal probability of being selected. Only these three operations will be checked in this part of the question.

Input Format: Design an infinitely running menu-driven main function. Each Time the user inputs an integer corresponding to the serial number of the operation listed above. Then, take necessary arguments related to the selected operation and execute the respective method. Finally, the program must exit with status code 0, when 0 is provided as a choice.

2 Question 2. Priority Queue

2.1 A. [80 marks]

Problem Statement: Implement priority queue

What is a priority queue?

- A priority queue is a data structure that maintains a collection of elements, each associated with a priority or value. Elements are stored in a way that allows the retrieval of the element with the highest (or lowest) priority quickly.
- Unlike deques or lists, a priority queue doesn't maintain the elements in any specific order based on their values, except for ensuring that the highest (or lowest) priority element is readily accessible.

Operations: The C++ standard specifies that a legal (i.e., standard-conforming) implementation of priority queue must satisfy the following performance requirements.

1. **priority_queue()** - initialize an empty priority queue. Time complexity: O(1)
2. **int size()** - returns the current size of the priority queue. Time complexity: O(1)
3. **void push(int el)** - insert an element el in the priority queue. Time complexity: O(log(size()))
4. **int top()** - returns the top (highest or lowest priority) element in the priority queue. Time complexity: O(1)
5. **void pop()** - remove the top element of the priority queue. Time complexity: O(log(size()))
6. **bool empty()** - returns true if the priority queue is empty else returns false. Time complexity: O(1)

2.2 B. [20 marks]

Problem Statement:

David's Bakery has introduced a new promotion for students. Under this promotion, he monitors his daily sales, and if the sales on a specific day is greater than or equal to the **combined sum of the median sale for a trailing number of d days and the median sale from the first day of the promotion**, (follow through the example included below to understand better) then he offers free 'Cheeeeeese Maggi' to the first five customers on the following day. However, David will only provide this free offer if he has transaction data for at least the trailing number of d prior days. Given the number of trailing days d and the daily sales record for David's Bakery over a period of n days, can you determine how many days David will offer free maggi to students during this entire period of n days?

Input Format:

- The first line contains two space-separated integers *n and d*, the number of days of daily sales data, and the number of trailing days respectively.

- The second line contains n space-separated non-negative integers where each integer *i denotes sales[i].*

Output Format:

- *int*: the total number of days David will offer free maggi to students.

Constraints:

$1 <= n <= 2 * 10^5$

$1 <= d < n$

$1 <= sales[i] <= 10^5$

Sample input:

8 5

2 3 4 2 1 6 3 6

Sample output:

2

Explanation:

For the first five days, David will not offer any free maggi to students because he didn't have sufficient sales data.

On the sixth day, David has d = 5 days worth of trailing sales data, which is 2, 3, 4, 2, 1. The median sale of the preceding five days is 2, and he also has the entire sales data from the beginning, which remains 2, 3, 4, 2, 1, with a median sale value of 2 as well. Since, David's sales on the 6th day is greater than the combined sum, David offers free Maggi to students for the 6th day.

On the seventh day, David has d = 5 days worth of trailing sales data, which is 3, 4, 2, 1, 6. The median sale of the preceding five days is 3, and he also has the entire sales data from the beginning, which is 2, 3, 4, 2, 1, 6, with a median sale value of (2 + 3) / 2 = 2.5 . Since, David's sales on the 7th day is less than the combined sum, he will not offer free Maggi to students for the 7th day.

On the eighth day, David has d = 5 days worth of trailing sales data, which is 4, 2, 1, 6, 3. The median sale of the preceding five days is 3, and he also has the entire sales data from the beginning, which is 2, 3, 4, 2, 1, 6, 3, with a median sale value of 3 as well. Since, David's sales on the 8th day is equal to the combined sum, David offers free Maggi to students for the 8th day.

Therefore, in total David will offer free maggi for 2 days (6th and 8th day).

Note: You need to implement priority queue for Integer data type only. You need to implement both minimum and maximum priority queue (Implementation details are up to you).

Question 3: The Lost Library of IIIT Hyderabad

Problem

Mtech PG1 people while roaming, found a mysterious, sealed-off wing in IIIT Hyderabad! Inside is a single, massive room with a long, continuous row of n bookshelves, called "bins". Each bin i contains multiple copies of a single, unique book with a numerical ID. The bins are arranged in sorted order of these book IDs.

A special robotic librarian is the only one that can navigate this wing. To find any book, the robot must follow a pre-programmed **Binary Search Plan**. And library was sealed because accessing the book there was not efficient. So to revive this library our task is to design the *most efficient* plan possible.

Part 1: The Ideal Search Plan [30 marks]

Let's first design the most logically efficient plan, ignoring all physical costs. In this model, the "cost" or time of finding a book is simply the **number of bins that are checked** during its search path. The total time of a plan is the sum of these individual search costs, weighted by how frequently each book is requested.

Input: An array $F[1 \dots n]$, where $F[i]$ is the monthly request frequency for the book in bin i.

Example:
Let's say $n = 3$ and the frequency array is $F = [10, 2, 5]$. A naive binary search strategy would be to always check the middle bin of any range. For the initial range $[1 \dots 3]$, the middle is **bin 2**. Let's analyze the cost of this naive plan:

- To find book 2: We check bin 2 (1 check). Time $= 1 \times F[2] = 1 \times 2 = 2$.
- To find book 1: We check bin 2, then bin 1 (2 checks). Time $= 2 \times F[1] = 2 \times 10 = 20$.
- To find book 3: We check bin 2, then bin 3 (2 checks). Time $= 2 \times F[3] = 2 \times 5 = 10$.

Total Cost for this naive plan: $2 + 20 + 10 = 32$. This example demonstrates that simply choosing the middle bin is not always optimal. Your task is to find a plan that beats this naive approach and finds the true minimum possible total time.

Part 2: Introducing Basic Robot Costs[30 marks]

Now, let's create a plan for a more realistic robot. The robot starts at a charging station at position 0.

New Cost Model:

- Moving from bin i to j costs $|i - j|$ seconds.
- For this part, the search for a book is "finished" once the bin is scanned.

Example:
Let's continue with our Part 1 example and the naive plan of probing bin 2 first.

- **Time for book 2:** (move $0 \to 2$) $= 2$. Total cost contribution $= 2 \times F[2] = 4$.
- **Time for book 1:** (move $0 \to 2 \to 1$) $= |2-0|+|1-2| = 3$. Total cost contribution $= 3 \times F[1] = 30$.
- **Time for book 3:** (move $0 \to 2 \to 3$) $= |2-0|+|3-2| = 3$. Total cost contribution $= 3 \times F[3] = 15$.

The total cost for this non-optimal plan is $4 + 30 + 15 = 49$.

Your Goal: Find the search plan that minimizes the total expected time with this new cost model.

Your Tasks for Part 2:

1. Adapt your approach from Part 1 to work with this more complex cost model. How does your recursive definition for the optimal cost change?
2. Describe an **efficient algorithm** to compute the minimum possible search time.
3. Analyze the time and space complexity of your algorithm.

Part 3: The Full Realistic Model[40 marks]

Finally, let's create a plan for the real-world robot with all of its complexities.

Real time Model:

- Moving from bin i to j costs $\alpha \cdot |i - j|$ seconds, for a given constant α.
- Scanning the book ID at any bin costs β seconds.
- **Reversing direction** (e.g., moving right then left) costs an additional γ seconds. For example, a sequence of moves $0 \to 5 \to 2$ would incur one reversal penalty.
- After the robot finds the correct book at bin i, it must **return to the start**, costing an additional $\alpha \cdot i$ seconds.

Example: Continuing with the above example only , let $\alpha = 2$, β=10 and $\gamma = 4$.

- **Time for book 2:** $=$ (move $0 \to 2$) + (scan) + (return $2 \to 0$) $= (2{\cdot}2)+10+(2{\cdot}2) =$ 18. No reversal. Total cost $= 18 \times F[2] = 36$.
- **Time for book 1:** $=$ (move $0 \to 2$) + (scan) + (reverse) + (move $2 \to 1$) + (scan) + (return $1 \to 0$) $= (2 \cdot 2) + 10 + 4 + (2 \cdot 1) + 10 + (2 \cdot 1) = 32$. Total cost $= 32 \times F[1] = 320$.
- **Time for book 3:** $=$ (move $0 \to 2$) + (scan) + (move $2 \to 3$) + (scan) + (return $3 \to 0$) $= (2 \cdot 2) + 10 + (2 \cdot 1) + 10 + (2 \cdot 3) = 32$. No reversal. Total cost $= 32 \times F[3] = 160$.

Current non-optimal total cost for this plan is $36 + 320 + 160 = 516$.

Your Tasks for Part 3:

1. The reversal penalty γ is tricky because the cost of a move now depends on the *previous* move. Explain how you would need to change your recursive definition to handle this. What new information must your subproblem definition keep track of?
2. Describe the final, efficient algorithm that solves this complete problem.
3. Analyze the time and space complexity of this final algorithm.

Constraints

For all parts of the problem, you can assume the inputs will follow these constraints:

- $1 \leq n \leq 80$
- $1 \leq F[i] \leq 1,000,000$
- $1 \leq \alpha, \beta, \gamma \leq 1,000$
- The total cost may exceed the capacity of a 32-bit integer.

Testing Your Algorithm: Input/Output Format

Your final program should be able to read the query type and corresponding parameters from standard input to solve the appropriate version of the problem.

Query Type 1 (for Part 1)

Input Format:

```
1
n
F_1 F_2 ... F_n
```

Example Input:

```
1
3
10 2 5
```

Example Output:

```
26
```

Hint: Path:1→3→2

Query Type 2 (for Part 2)

Input Format:

```
2
n
F_1 F_2 ... F_n
```

Example Input:

```
2
3
10 2 5
```

Example Output:

```
29
```

Hint: Path:1→2→3

Query Type 3 (for Part 3)

Input Format:

```
3
n
F_1 F_2 ... F_n
alpha beta gamma
```

Example Input:

```
3
3
10 2 5
2 10 4
```

Example Output:

```
392
```

Hint: Path:1→3→2

Notes:

- The reversal penalty only applies to movements during the search phase, not for the final "return home" trip.
- You are not allowed to use vector and all, so if you require any function make your own.