Date: / /
Page No.:

Searching And Sorting

Bishu and Soldiers

Problem Statement

Bishu loves war games. His country has an army with N soldiers, each having some power. When Bishu goes to war, he can fight with soldiers of enemy countries. There are Q rounds, and in each round, an enemy soldier appears with a certain power. Bishu has to count how many soldiers in his army have power ≤ that enemy soldier's power and find the total power of those soldiers.

Algorithm

1. Read an integer N (number of soldiers).
2. Read N space-separated integers representing the power of each soldier.
3. Read an integer Q (number of enemy soldiers).
4. Read Q queries, where each query is an integer P (power of an enemy soldier).
5. Sort the array of soldier's power in ascending order.

Rolex

Date: / /
Page No.:

6) Compute a prefix sum array:
- Create an array prefix_sum where:
  prefix_sum[i] = sum of all soldiers' power from index 0 to i.

7) For each enemy power P:
- Use binary search (bisect_right in Python) to find the number of soldiers with power ≤ P.
- Find the sum of their powers using the prefix sum array.

8) Print the count of soldiers ≤ P and their total power.

```
import bisect

#Input
n=int(input()) # Number of soldiers
soldiers = list (map(int, input().split()))
q=int(input()) # Number of enemy soldiers

#Sorting soldiers' powers
soldiers.sort()

#Prefix sum array
prefix_sum = [0] * n
prefix_sum[0] = soldiers[0]
for i in range(1, n):
    prefix_sum[i] = prefix_sum[i-1] + soldiers[i]
```

Date: / /
Page No.:

```
#Answering queries
for _ in range(q):
    power = int(input())
    idx = bisect.bisect_right(soldiers, power)
                        #upper bound index
    count = idx
    total_power = prefix_sum[idx-1] if idx>
                                0 else 0
    print(count, total_power)
```

Input : 7
1 2 3 4 5 6 7
3
3
10
2

Step 1: Read Input

1. n = 7 → Number of soldiers.
2. soldiers = [1, 2, 3, 4, 5, 6, 7] → Power of each soldier.
3. q = 3 → Number of queries.
4. Queries:
   - 3 → First enemy soldier's power.
   - 10 → Second enemy soldier's power.
   - 2 → Third enemy soldier's power.

Step 2: Sorting the Soldiers' Powers
Since the soldiers' powers are already sorted ([1, 2, 3, 4, 5, 6, 7]), no change happens. But in general, sorting ensures that we can efficiently use binary search.

Rolex

Step 3: Compute Prefix Sum
We create an array to store the cumulative sum of soldier powers.

Index	Soldiers' Power	Prefix Sum
0	1	1
1	2	1+2=3
2	3	3+3=6
3	4	6+4=10
4	5	10+5=15
5	6	15+6=21
6	7	21+7=28

So, prefix_sum = [1, 3, 6, 10, 15, 21, 28]

Step 4: Answering Queries Using Binary Search
We use bisect_right(soldiers, P), which finds the first index where P should be inserted while maintaining order. This also gives the count of elements ≤ P.

Query 1: P=3
- idx = bisect_right(soldiers, 3) = 3 (Elements ≤ 3: [1, 2, 3])
- Count = 3
- Total Power = prefix_sum[2] = 6
- Output: 3 6

Query 2: P = 10
- idx = bisect_right(soldiers, 10) = 7 (Elements ≤ 10: [1, 2, 3, 4, 5, 6, 7])
- Count = 7
- Total Power = prefix_sum[6] = 28
- Output: 7 28

Query 3: P=2

- idx = bisect_right(soldiers, 2) = 2 (Elements ≤ 2: [1, 2])
- Count = 2
- Total Power = prefix_sum[2] = 3
- Output: 2 3

Final Output

3 6
7 28
2 3

Time Complexity Analysis

Operation	Complexity
Sorting Soldiers	$O(N \log N)$
Prefix Sum Calculation	$O(N)$
Query Processing (Binary Search)	$O(Q \log N)$
Total Complexity	$O(N \log N + Q \log N)$

This is efficient for large values of N and Q.

Date: / /
Page No.:

Stack And Queues

Stack Permutations

A stack permutation is a rearrangement of elements using a stack (LIFO structure). Given an input sequence, a stack permutation is a possible output sequence that can be obtained by pushing elements onto a stack and popping them in some order.

Algorithm:

1. Initialize an empty stack to simulate the push and pop ~~elements~~ operations.

2. Initialize two pointers:
   - i = 0 to traverse input_seq.
   - j = 0 to traverse output_seq.

3. Iterate through input_seq:
   a. Push input_seq[i] onto the stack.
   b. Increment i by 1.
   c. While the stack is not empty and the top of the stack matches output_seq[j]:
      - Pop the top element from the stack.
      - Increment j by 1.

4. After processing all elements from input_seq, check if the stack is empty:
   - If yes, return True (valid stack permutation).
   - If no, return False (invalid stack permutation).

Rolex

Date: / /
Page No.:

Code.

```
def is_stack_permutation(input_seq,
                         output_seq):
stack = [ ]
i = 0  # Pointer for input sequence
j = 0  # Pointer for output sequence

while i < len(input_seq):
    stack.append(input_seq[i]) # Push element
                               onto stack.
    i += 1

    while stack and stack[-1] == output_seq[j]:
        stack.pop()
        j += 1

    return not stack

input_seq = [1, 2, 3]
output_seq = [2, 1, 3]
print(is_stack_permutation(input_seq, output
                           _seq))
```

Time Complexity

O(n) because each element is pushed and popped at most once.