M24-CS_1.304 Data Structures and Algorithms for Problem Solving

Assignment–3

Important Points

- Only **C++** is allowed.
- Use only the **allowed headers** mentioned at the end of each question.
- This is an **in-lab assignment**. The lab will be held on **10th November 2025**.
- All of you need to **prepare all four questions**, and you will be asked to code only **one** of them in the lab.
- The lab starts at **8:30 AM** as usual. Please be on time.
- If you have any doubts please post them on Moodle.

1. Big Integer Library

Problem Statement: Create a big integer library, similar to the one available in Java. The library should provide functionalities to store arbitrarily large integers and perform basic math operations.

Operations:
Arithmetic Operations and Constraints:

- **Basic Arithmetic Operations:**
  - Addition (+)
  - Subtraction (-)
  - Multiplication (x, lowercase ”X”)
  - Division (/)
- **Examples:**
  - Input: 32789123+99893271223x9203232392-4874223
  - Output: 919340989462382970316
  - Input: 3423542525+6773442x5345345-213213197786/45647

  - Output: 36209803199102

- **Exponentiation:** Base will be a big integer and the exponent will be less than 2^{63}.
- **GCD of Two Numbers**
- **Factorial**

Constraints:

- For all operations, the number of digits in input, output, and intermediate results won't exceed 3000 digits.

Expected Time Complexity:

- Addition ($s_1 + s_2$): $O(n + m)$
- Subtraction ($s_1 - s_2$): $O(n + m)$
- Multiplication ($s_1 * s_2$): $O(n * m)$
- Division (s_1/s_2): $O(n * m)$
- Exponentiation (s_1^x): $O(n^2 \cdot \log(x))$
- GCD (s_1, s_2): $O(\max(n, m))$
- Factorial (s_1): $O(n^3)$

Input Format:

- First line will contain an integer value which denotes the type of operation. The integer value and operation mapping is as follows:
  - 1: Addition, Subtraction, Multiplication, & Division
  - 2: Exponentiation
  - 3: GCD
  - 4: Factorial
- The following line will contain input according to the type of operation:
  - For the 1st and 4th types of operation, there will be one string.
  - For the 2nd and 3rd types of operation, there will be 2 space-separated strings.

Sample Cases:

Sample Input 0:	1 1+2x6+13/5-2
Sample Output 0:	13
Sample Input 1:	2 2 10
Sample Output 1:	1024
Sample Input 2:	3 9 15
Sample Output 2:	3
Sample Input 3:	4 12
Sample Output 3:	479001600

Note:

1. **Negative Numbers:** Negative numbers won't be present in the intermediate or final output (i.e., no need to consider cases like $2 - 3$).
2. **Brackets:** There are **no** brackets in the input.
3. **Integer Division:** Perform integer division operation between two big integers, disregarding the remainder.
4. **Operation Precedence:** Addition, subtraction, multiplication, and division follow the same precedence and associativity rules as in Java/C++.
5. **Special Cases:** Ignore division by zero, $\gcd(0, x)$, $\gcd(x, 0)$.
6. **C++ STL:** C++ STL is not allowed except for string library (if you want to use vectors and stack; design your own).
7. **Regex Library:** You are not allowed to use the regex library.
8. **String Manipulation:** string, to_string, and string manipulation methods are allowed.
9. **Main Function Design:** Design your `main` function according to the sample input/output given.

2. Skip List

Problem Statement: The skip list is an example of a probabilistic data structure because it makes some of its decisions at random. While the skip list is not guaranteed to provide good performance, it will provide good performance with extremely high probability.

References: Skip List, Design Skiplist

Sr no.	Operation	Expected Time Complexity
1	Insertion	$O(\log N)$
2	Deletion	$O(\log N)$
3	Search	$O(\log N)$
4	Count element occurrences	$O(\log N)$
5	lower_bound	$O(\log N)$
6	upper_bound	$O(\log N)$
7	Closest element to a value	$O(\log N)$

Important Points:

- **Implement it using a class or struct.**
- The skip list should work with **integer data type only.**
- **Duplicates are allowed.**
- In your driver code, print the skip list (bottom level) after every operation.

Operations:

1. **void insert(e):** Inserts e into the skip list.
2. **void delete(e):** Deletes all the occurrences of the element e, if it is present in the skip list.
3. **bool search(e):** Returns `true` if e is present in the skip list, otherwise returns `false`.
4. **int count_occurrence(e):** Returns the count of occurrences of the element e.
   - **Example:** If the skip list has the elements: $1, 1, 2, 2, 2, 3$
     - `count_occurrence(2)` will return 3.
     - `count_occurrence(734)` will return 0.
5. **int lower_bound(e):** Returns the first element that is greater than or equal to e. If no such element exists, return 0.
   - **Example 1:** If the skip list has the elements: $1, 1, 2, 2, 2, 3$
     - `lower_bound(2)` will return 2 (the 3rd element from the left).
   - **Example 2:** If the skip list has the elements: $1, 1, 2, 5, 6, 6, 7$
     - `lower_bound(3)` will return 5 (the 4th element from the left).

6. **int upper_bound(e):** Returns the first element that is greater than e. If no such element exists, return 0.

   - **Example 1:** If the skip list has the elements: $1, 1, 2, 2, 2, 3$
     - `upper_bound(2)` will return 3 (the last element from the right).
   - **Example 2:** If the skip list has the elements: $1, 1, 2, 5, 6, 6, 7$
     - `upper_bound(7)` will return 0 (no element greater than 7).

7. **int closest_element(e):** Returns the element closest to e. If no such element exists, return 0.

   - **Example:** If the skip list has the elements: $1, 1, 2, 2, 2, 5, 7$
     - `closest_element(2)` will return 2.
     - `closest_element(3)` will return 2.
     - `closest_element(4)` will return 5.
     - `closest_element(-1472)` will return 1.
     - `closest_element(6)` will return 5 (If multiple closest elements are present, return the smaller one).

3. Strings Galore

String processing is a common problem across domains. With greater amounts of data being generated with time, the efficiency of algorithms for storage/retrieval has become increasingly important.

3A. Never Again MLE

- Comparing two strings has a time complexity of $\mathcal{O}(\min(n_1, n_2))$, where n_1 and n_2 denote the lengths of the two strings. In practice, when processing large amounts of data, various optimization techniques are applied to improve both time and memory efficiency.
- In this problem, you are required to design a system that efficiently checks for the presence of a set of strings while minimizing memory usage.
- For reference, you may incorporate the ideas of **string hashing** and **bloom filters**.
- You will be given n strings. For each string s_i, determine whether or not it has been encountered before.

Input

The first line of input contains a single integer, t, denoting the number of tests. In each test, the first line will contain a single integer n, the number of strings. Each of the next n lines will contain a single string, s_i.

Constraints

- $1 \leq t \leq 10^4$
- $1 \leq n \leq 10^5$
- $1 \leq \sum\_is\_i \leq 2 \cdot 10^8$ (summed across all tests)
- $1 \leq |s\_i| \leq 10^5$
- s_i is composed of lower-case English alphabet characters.

Memory Limit

- 512MB

Time Limit

- 6s

Output For each string, report 1 if it was encountered before in that test, 0 otherwise.

3B. Pooling Resources

- In this problem, you are required to efficiently store strings in an **associative array**.
- You will be given n queries. Queries can be of two types:
  - **Type 0 (put):** Insert or update an entry.
  - **Type 1 (get):** Retrieve an entry.
- Consider an initially empty associative array A.
- For **Type 0** queries, you will be given an integer i and a string s_i. This operation maps s_i to i in A.
- For **Type 1** queries, you will be given an integer q_i. You must output the corresponding value in A if the key q_i exists, and output 0 otherwise.
- For reference, you may incorporate the ideas of **string pooling** while implementing this problem.

For each string, report 1 if it was encountered before in that test, 0 otherwise.

3B. Pooling Resources

- In this problem, you are required to efficiently store strings in an **associative array**.
- You will be given n queries. Queries can be of two types:
  - **Type 0 (put):** Insert or update an entry.
  - **Type 1 (get):** Retrieve an entry.
- Consider an initially empty associative array A.
- For **Type 0** queries, you will be given an integer i and a string s_i. This operation maps s_i to i in A.
- For **Type 1** queries, you will be given an integer q_i. You must output the corresponding value in A if the key q_i exists, and output 0 otherwise.
- For reference, you may incorporate the ideas of **string pooling** while implementing this problem.

Input

The first line of input contains a single integer, t, denoting the number of tests. The first line of each test contains a single integer, n, the number of queries. Each of the next n lines will contain a single query.

(type: 0) queries follow the format: 0 i s_i

(type: 1) queries follow the format: 1 i

Constraints

- $1 \leq t \leq 10^4$

- $1 \leq \sum n \leq 10^6$
- $1 \leq i \leq 10^{18}$
- $1 \leq \sum \_is\_i \leq 2 \cdot 10^8$ (summed across all tests)
- $1 \leq |s\_i| \leq 10^5$
- s_i is composed of lower-case English alphabet characters.
- Let S denote the set of unique strings in the input.
- $1 \leq \sum \_s\_i \in S|s\_i| \leq 10^6$ (summed across all tests)

Memory Limit

- 512MB

Time Limit

- 5s

Output
For each query of type: 1, report the response - the associated string s_q_i or 0 if there is no such key.

Note:

- Only the headers `<iostream>`, `<string>`, and `<vector>` are allowed.
- Use of any other headers in the submission will lead to **0 marks** being awarded for the particular problem.

4. Graph Explorer

Problem Statement: Design a system to process queries on a dynamic graph (either directed or undirected). The system should efficiently support traversal, connectivity, and analysis operations.

Supported Operations:

- **BFS (Type 0):** Perform Breadth-First Search from a given source vertex and print the **lexicographically smallest BFS traversal order**.
- **DFS (Type 1):** Perform Depth-First Search from a given source vertex and print the **lexicographically smallest DFS traversal order**.
- **Cycle Detection (Type 2):** Check whether the graph contains a cycle. Works for both directed and undirected graphs.
- **Bipartiteness Check (Type 3):** Determine if the graph is bipartite. Applicable only for **undirected** graphs.
- **Connected Components (Type 4):** Count the number of connected components in the graph. For directed graphs, this refers to the number of **strongly connected components**.
- **Dijkstra's Algorithm (Type 5):** Compute the shortest path from a source vertex to a target vertex in a weighted graph.
- **Kosaraju's Algorithm (Type 6):** For **directed graphs**, print all strongly connected components (SCCs). Each SCC should be printed as a space-separated list on a new line, in lexicographical order.
- **Topological Sort (Type 7):** For **directed acyclic graphs (DAGs)**, print the lexicographically smallest topological order.
- **Add Edge (Type 8):** Dynamically add an edge (u, v, w) to the graph. Default weight is 1 if unweighted. The update should take effect immediately.
- **Add Vertex (Type 9):** Add a new vertex to the graph. The new vertex ID will be the next unused integer (starting from V). After adding the vertex, print its assigned ID.

Input Format:

- The first line contains three integers:

  $$\texttt{isDirected} \quad V \quad E$$

  where: `isDirected = 0` → undirected graph `isDirected = 1` → directed graph
- The next E lines contain edges as: $u\ v\ w$ (weight optional, default $w = 1$)
- Next line: Q — number of queries.
- Next Q lines — each line represents one query:

  - Type 0: `0 s` — BFS from vertex s
  - Type 1: `1 s` — DFS from vertex s
  - Type 2: `2` — Cycle detection
  - Type 3: `3` — Bipartiteness check (only if undirected)
  - Type 4: `4` — Connected components
  - Type 5: `5 s t` — Dijkstra's shortest path
  - Type 6: `6` — Kosaraju's SCCs (only if directed)
  - Type 7: `7` — Topological sort (only if directed)
  - Type 8: `8 u v w` — Add edge
  - Type 9: `9` — Add vertex (print new vertex ID)

Expected Time Complexity:

- BFS / DFS: $O(V + E)$
- Cycle Detection / Bipartiteness / Components: $O(V + E)$
- Dijkstra's: $O((V + E) \log V)$
- Kosaraju (SCC): $O(V + E)$
- Topological Sort: $O(V + E)$
- Add Edge / Vertex: $O(1)$ amortized

Notes:

- All traversals (BFS, DFS, Topo Sort) must follow lexicographically smallest order.
- Dynamic updates (edges, vertices) must reflect immediately.
- Skip unsupported operations based on graph type (e.g., don't perform bipartiteness on directed graphs).

Constraints:

- $1 \leq V \leq 10^4$ — Number of vertices
- $0 \leq E \leq 2 \times 10^5$ — Number of edges
- Edge weights: $1 \leq w \leq 10^6$
- $1 \leq Q \leq 10^4$ — Number of queries
- Graph may be disconnected
- Vertex indices: 0 to $V - 1$
- Self-loops and multi-edges are not allowed

Sample Test Case 1 (Undirected Graph):

```
Input:
0 4 3
0 1 1
1 2 1
2 3 1
3
0 0
3
4

Output:
0 1 2 3
1
1
```

Sample Test Case 2 (Directed Graph):

```
Input:
1 5 6
0 1 1
1 2 1
2 0 1
1 3 1
3 4 1
4 1 1
3
6
7
2

Output:
0 1 2
3 4
1
```

Sample Test Case 3 (Dynamic Updates):

```
Input:
0 3 2
0 1 1
1 2 1
5
0 0
9
8 2 3 1
0 0
4

Output:
0 1 2
```

```
3
0 1 2 3
1
```

Note: STL is allowed for this question.