INDIAN STATISTICAL INSTITUTE

MTech(CS) I year 2025-2026

Subject: Computing Laboratory

Lab Test 4 (May 15, 2026)

Total: 60 marks Duration: 4 hours

GENERAL INSTRUCTIONS

1. All programs should take the required input via the standard input (terminal/keyboard), and print the desired output to the terminal.
2. Please make sure that your programs adhere strictly to the specified input and output format. Do not print extra strings asking the user for input, debugging messages, etc. These will cause the automatic checking system to fail.

Problem 1 / Q1. (12 marks)

Your task in this problem is to implement and measure the performance of a **Bloom Filter** (BF), a data structure used for **inexact searching** in sets. It supports the standard INSERT operation, as well as an INEXACT-SEARCH operation, which works as follows. Suppose S is a set of elements inserted into a BF. In response to an inexact search for an element x, the BF returns one of two answers:

- x is definitely not in S, or
- x appears to be in S, but this could be a mistake (this is called a **false positive** error when x is actually not in S).

For this problem, we will assume that the elements stored in a BF are all integers.

How a Bloom Filter works[1]**.** A BF consists of an array A of m flags (all set to 0 or FALSE initially), and a set of k different functions, say h_1, h_2, ..., h_k. For each of these functions, the domain is the set of integers, and the range is $\{0, 1, 2, \ldots, m-1\}$. The Bloom Filter works as follows.

- INSERT(x): to store an integer x in the BF, the elements $A[h_1(x)], A[h_2(x)], \ldots, A[h_k(x)]$ are each set to 1 (or TRUE).
- SEARCH(y): to search whether an integer y has been stored in the BF, we check $A[h_1(y)]$, $A[h_2(y)]$, ..., $A[h_k(y)]$. If any of these flags is 0/FALSE, then the BF returns "y is definitely not in S." This answer is guaranteed to be correct, because if y were inserted into the BF, then all the flags would have been set to 1/TRUE when y was inserted.

  If all the flags are 1/TRUE, then the Bloom filter returns "y appears to be in S". It is possible that y was not actually inserted earlier, but these bits were set by chance to 1 during the insertion of other elements. This would then count as a **false positive** error.

Write a program that first inserts n_1 integers x_1, x_2, ..., x_{n_1} into the BF defined above (m, k and h_1, h_2, ..., h_k will also be given to you). The program should then read n_2 more integers y_1, y_2, ..., y_{n_2}, and do an inexact search in the BF for each y_1, y_2, ..., y_{n_2}. Your program should also determine whether the search result is a false positive.

Input format: The integers m, k, n_1 and n_2, followed by x_1, x_2, ..., x_{n_1}, y_1, y_2, ..., y_{n_2}. You will also be given a C file named `bf-functions.c`. You should `#include` this file in your program. To compute $h_i(x)$, you should call the C function `h(i, m, x)` (defined in `bf-functions.c`).

[1] `https://en.wikipedia.org/wiki/Bloom_filter`

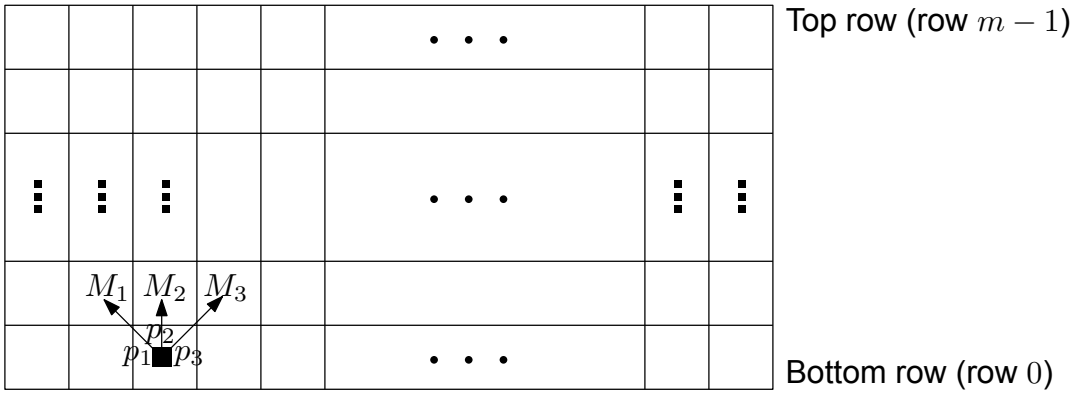
Output format: For each of the search keys, your program should print the search key itself (i.e., y_i), and the output of the Bloom filter (either NO or YES). If the output is YES, you should also print whether the output is a false positive. To determine whether the output is a false positive, you may use any search algorithm (including linear / binary search). [12]

Problem 2 (16 + 8 = 24 marks)

You are given an $m \times n$ checkerboard (like a chess board) and a checker (like a chess piece). You must move the checker from the bottom row of the board (row 0) to the top row (row $m-1$) according to the following rule. At each step, one of three possible moves is permitted (see the diagram given below):

- Move M_1: move the checker to the square that is one row up and one column to the left (but only if the checker is not already in the leftmost column), or
- Move M_2: move the checker to the square immediately above, or
- Move M_3: move the checker to the square that is one row up and one column to the right (but only if the checker is not already in the rightmost column).

For each square, each permitted move has an associated score. Depending on the current position (i, j) $(0 \leq i < m,\ 0 \leq j < n)$ and the chosen move (M_1, M_2 or M_3), you score $p_1(i,j), p_2(i,j)$ or $p_3(i,j)$ points (these scores are not necessarily positive).



Q2. Write a program to compute the maximum total score that can be obtained by moving a checker from a bottom square to a top square. Any square in the bottom row is a valid starting point, and any square in the top row is a valid destination.

Input format: In order, the values of m, n and the scores (a triple (p_1, p_2, p_3) of integers) for each square. The score triples will be given to you in row-major order, i.e., starting with $\langle p_1(0,0), p_2(0,0), p_3(0,0)\rangle$ and ending with $\langle p_1(m-1,n-1), p_2(m-1,n-1), p_3(m-1,n-1)\rangle$. For simplicity, 3 scores will be specified for each square, even when some of these moves are not permitted (e.g., $p_1(0,0), p_3(0,n-1)$ and all scores for the top row should be ignored, since these do not correspond to valid moves).

Output format: The maximum total score obtainable across all paths starting in the bottom row, and ending in the top row.

For full credit, your algorithm should run in $O(mn)$ time, and use $O(mn)$ extra space. [16]

Q3. Modify your program to compute the actual path from a bottom square to a top square that the checker should follow in order to obtain the maximum total score. As above, any square in the bottom row is a valid starting point, and any square in the top row is a valid destination.

Input format: As above.

Output format: The printed path should consist of the column numbers of the squares comprising the path, starting from the bottom row and ending in the top row. If multiple paths result in the same maximum score, your program should print the path that corresponds to choosing the leftmost option at every stage. [8]

Problem 3 (16 + 8 = 24 marks)

In a *K-ary tree*, any node can have upto K children. A node in a K-ary tree may be implemented as follows.

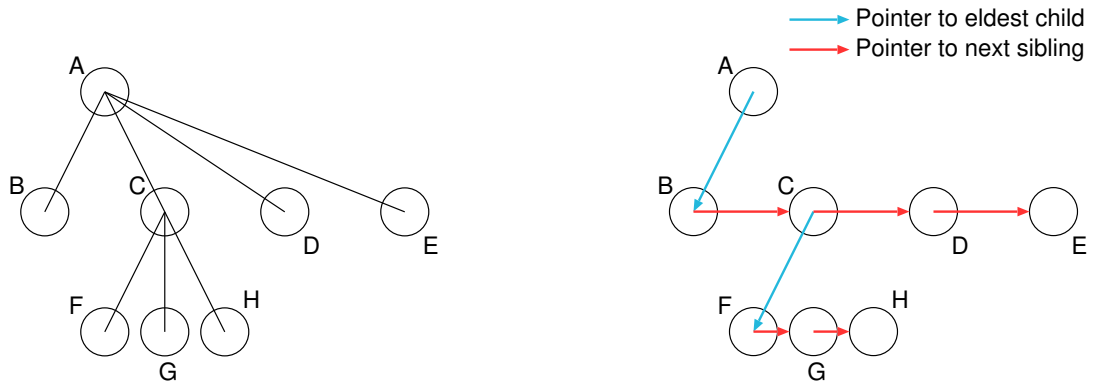
```
typedef struct {
    int data;
    int num_children;
    int indices_of_child_nodes[K];
} TREENODE;
```

If K is large, but in practice, most nodes have only a small number of children, then a non-trivial amount of memory is wasted by the above implementation. We would therefore like to use the following type definition to represent a node in a K-ary tree.

```
typedef struct {
    int data;
    int eldest_child, next_sibling; // instead of left, right
    int num_children, parent, level; // additional information to be added later
} TREENODE;
typedef struct {
    int root, num_nodes; // for this problem, root will be equal to 0
    TREENODE *nodelist;
} TREE;
```

That is, each node points to its eldest child, and to its right sibling (if these exist). In this definition, each K-ary tree node holds **exactly two** indices (equivalent to pointers), just like a node in a conventional binary tree. This definition also works for a tree of **arbitrary arity**, i.e., even when there is no fixed upper bound on the number of children. As usual, an index value of -1 is used instead of NULL pointers.

The conventional representation of a tree and its representation using the node definition given above is shown in the following figure. The figure uses uppercase letters as data (instead of integers) to avoid any confusion between the index of a node and the data stored in it. Note that the indices of A, B, C, etc. **remain the same** in both the representations.



Index	Data	No. children	Indices of children
0	A	4	1 2 3 4
1	B	0	
2	C	3	5 6 7
3	D	0	
4	E	0	
5	F	0	
6	G	0	
7	H	0	

Index	Data	Eldest child	Next sibling
0	A	1	-1
1	B	-1	2
2	C	5	3
3	D	-1	4
4	E	-1	-1
5	F	-1	6
6	G	-1	7
7	H	-1	-1

Table 1: Input format for a tree of arbitrary arity (left), and its desired representation (right).

Q4. Write a program that reads in a tree of arbitrary arity in the format specified below, and stores it using the type definitions given above. **The index of each data item in `nodelist` should remain the same as in the original representation.** Your program should also compute and fill in the parent index for each node, as well as its level[2].

Your program should finally print the above representation of the tree in the usual tabular format. Note that all lines (except the first line) in the output will contain 6 columns, but the number of columns in the input will, in general, vary across lines.

Input format: The first line will contain n, the number of nodes in the tree. This will be followed by n more lines, each corresponding to a node. Each of these lines will contain the data (an integer) stored in the node, the number of its children, followed by the indices of its children. The first node (index=0) is assumed to be the root. See the left-side of Table 1 to get an idea about the input format.

Output format: The number of nodes, followed by one line per node. Each of these lines will contain 6 integers, corresponding to the 6 fields of `TREENODE`, printed in the order in which they appear in the definition. As mentioned above, **the index of each data item in `nodelist` should remain the same as in the original representation.**

For full credit, your program should allocate dynamic memory only for the final representation (i.e., you should not need to store the input data in any other temporary array or data structure). Also, the fields of the desired representation should all be filled in one **single** pass over the input data (initialising the fields to default values is acceptable, and will not incur any penalty). [16]

Q5. Write a program which, given the indices of any two nodes in a `TREE` (our implementation of a tree of arbitrary arity), prints the index of their lowest common ancestor.

Input format: A tree in the same format as that used for the output of Q1, followed by the indices of two vertices in the tree.

Output format: A single integer. [8]

[2]Recall that the *level* of the root is defined to be 0; if m is a child node of n, then $level(m) \triangleq 1 + level(n)$.