Indian Statistical Institute

Semester-I 2024–2025

M.Tech.(CS) - First Year

Assignment (Due date: 20 December, 2024)

Subject: Computing Laboratory

Total: 4×15 = 60 marks

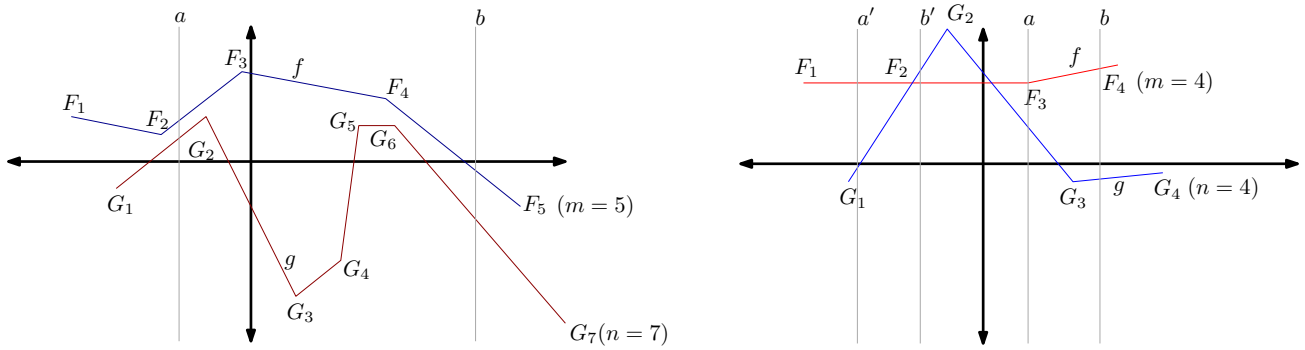
INSTRUCTIONS

1. You may consult or use slides / programs provided to you as course material, or programs that you have written yourself as part of classwork / homework for this course, but please **do not** consult or use material from other Internet sources, your classmates, or anyone else.
2. Unless otherwise specified, all programs should take the required inputs from stdin, and print the desired outputs to stdout. Please make sure that your programs adhere strictly to the specified input and output format. **Your program may not pass the test cases provided, if your program violates the input and output requirements.**
3. Submissions from different students having significant match will **not be evaluated**.
4. To avoid mismatches between your output and the provided output, please store all floating point numbers in `double` type variables.

Q1. (Q5 from Lab Test 1.)

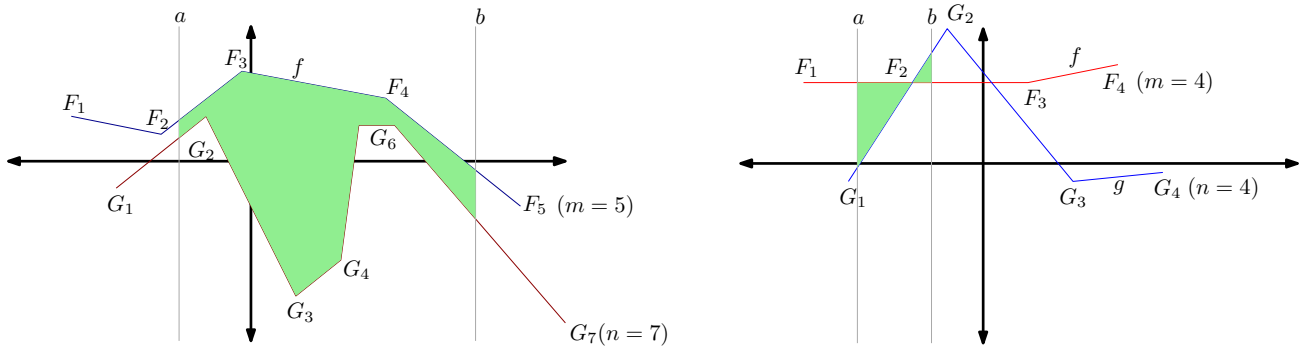
You will be given two piecewise linear functions, $f : \mathbb{R} \to \mathbb{R}$ and $g : \mathbb{R} \to \mathbb{R}$, and an interval $[a, b]$ in $\mathbb{R}$, as inputs. Write a program to determine (i) whether f and g are separable in $[a, b]$, and (ii) the area enclosed between f and g in $[a, b]$.

Recall from Lab Test 1 that f and g are said to be *separable* in the interval $[a, b] \subseteq \mathbb{R}$ if f and g are both defined in the interval $[a, b]$ and either $f(x) > g(x)$ for all $x \in [a, b]$ or $f(x) < g(x)$ for all $x \in [a, b]$. For example, in Figure 1(a), functions f and g are separable in the interval $[a, b]$, but in Figure 1(b), f and g are not separable in the interval $[a', b']$.



(a) (b)

Figure 1: Separable and non-separable piecewise linear functions



(a) (b)

Figure 2: Separable and non-separable piecewise linear functions

Addendum: if $\max(x_1^{(f)}, x_1^{(g)}, a) < \min(x_m^{(f)}, x_n^{(g)}, b)$, then irrespective of whether the two functions are separable, the lines corresponding to f and g enclose an area between them within the interval $[\max(x_1^{(f)}, x_1^{(g)}, a), \min(x_m^{(f)}, x_n^{(g)}, b)]$, as shown in Figure 2.

Input format: The input to your program will comprise 3 lines.

The first line will specify the function f: it will consist of a positive integer $m \geq 2$, followed by the x and y coordinates of m points, $F_1(x_1^{(f)}, y_1^{(f)}), F_2(x_2^{(f)}, y_2^{(f)}), \ldots, F_m(x_m^{(f)}, y_m^{(f)})$. **The points will be listed in increasing order of their x coordinates,** i.e., $x_1^{(f)} < x_2^{(f)} < \cdots < x_m^{(f)}$. In the interval $[x_i^{(f)}, x_{i+1}^{(f)}]$, the function f corresponds to the line segment joining F_i and F_{i+1}; f is not defined if $x \notin [x_1^{(f)}, x_m^{(f)}]$.

The second line will specify the function g in the same format, i.e., it will consist of a positive integer $n \geq 2$, followed by the x and y coordinates of n points. As for f, the points will be listed in increasing order of their x coordinates.

The third line will specify the two numbers a and b defining the interval for which you have to calculate separability and area. Addendum: you may assume that $\min(x_1^{(f)}, x_1^{(g)}) \leq a < b \leq \max(x_m^{(f)}, x_n^{(g)})$.

Output format: Your program should print either SEPARABLE or NOT SEPARABLE, depending on whether the given functions are separable or not in the given interval, followed by the area between the curves corresponding to f and g within the interval $[a, b]$, correct to 4 decimal places. Addendum: if $\max(x_1^{(f)}, x_1^{(g)}, a) \geq \min(x_m^{(f)}, x_n^{(g)}, b)$, print UNDEFINED for the area.

Q2. Write a program to simulate a Snakes-and-Ladders game between k players, P_1, P_2, ..., P_k

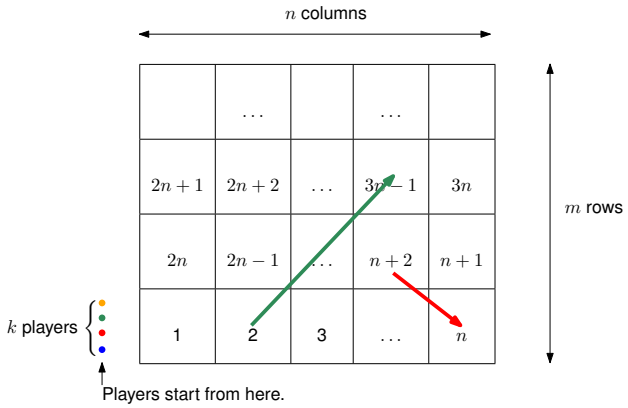


Figure 3: An $m \times n$ Snakes-and-Ladders board, with a green arrow representing a ladder (from 2 to $3n-1$), and a red arrow representing a snake (from $n+2$ to n).

Input format: The input will consist of the following 3 parts.

(a) Five positive integers k, m, n, l, s, r representing the number of players, rows in the board, columns in the board, ladders, snakes, and finally, the number of rounds for which the players play the game, respectively.

(b) $l+s$ pairs of positive integers, corresponding to the end-points of the l ladders and s snakes respectively. The starting point of a ladder is guaranteed to be smaller than its end point; conversely, the starting point of a snake is guaranteed to be larger than its end point. Addendum: you may also assume that any square has at most one starting point (of either a snake or a ladder). However, the end point of a ladder/snake may be the starting point of another ladder/snake. Thus, ladders and snakes may form chains that are either linear, or contain a loop.

(c) $r \times k$ integers (each between 1 and 12), representing the outcomes when the k players roll the dice in turn, for r rounds. **When one player reaches the end, the result of her rolling the dice is ignored.**

Rules of the game:

- Each player starts at "cell 0" as shown. If the player rolls $d \in \{1, 2, \ldots, 12\}$), she moves forward by d cells. If, as a result, she lands on a cell where a ladder/snake (or a chain of ladders and/or snakes) starts, her final position for the current round is at the end point of the ladder / snake / chain consisting of ladders and/or snakes. If the chain forms a loop, then the player goes into a "LOOPING" state, her position is indeterminate, and the result of her rolling the dice in subsequent rounds is ignored.
- If her final position for the current round is the same as the current position of some other player(s), the other player(s) have to move back to the starting position ("cell 0").
- When a player is on cell $mn-d$ for $d \in \{1, 2, \ldots, 12\}$ and rolls e, the player is permitted to move iff $e \leq d$, i.e., a player is not permitted to move, if the move would result in the player landing on a (hypothetical) cell numbered more than mn.

Output format: Your program should print one line of output per player, starting with the serial number of the player ($1 \ldots k$), the current position of the player if it is between 1 and $mn-1$, or COMPLETED if the player has reached the end (cell mn), or LOOPING, if appropriate, followed by the number of rounds played by the player.

Q3. Many programming languages provide *map*() and *filter*() functions that work as follows.

map. Given two sets X, Y, $map(L, f)$ takes as input a list $L = [l_0, l_1, \ldots, l_{N-1}]$ of elements from X, and a function $f : X \to Y$, and returns $f(L) \triangleq [f(l_0), f(l_1), \ldots, f(l_{N-1})]$, a list of elements from Y.

filter. Given a set X, $filter(L, g)$ takes as input a list $L = [l_0, l_1, \ldots, l_{N-1}]$ of elements from X, and a function $g : X \to \{\text{TRUE}, \text{FALSE}\}$, and returns $L' \triangleq [l_t \mid 0 \leq t < N,\ g(l_t) = \text{TRUE}]$.

Implement the `map` and `filter` functions in C. The functions should have the following prototypes:

```
void *map(void *L, unsigned int N,
          size_t domain_elt_size, size_t range_elt_size,
          void (*f)(void *input, void *output))

int filter(void *L, unsigned int N, size_t domain_elt_size,
             int (*g)(void *input))
```

where

- `L` is the input list;
- `N` is the number of elements in `L`;
- `domain_elt_size` (and `range_elt_size`, resp.) correspond to the size of each element of the domain and range of $f : X \to Y$;
- `f` is a pointer to a C function that implements f (`input` is a pointer to an element of the domain X, and `output` is a pointer to an existing chunk of memory that is just big enough to store an element of the co-domain Y); and
- `g` is a pointer to a C function that implements g (`input` is a pointer to an element of the domain X; `g` returns a non-zero value if $g(*\texttt{input}) = \text{TRUE}$, and `0` otherwise).
- `map` should allocate memory for the array containing $f(L)$; the user of `map` is responsible for freeing this memory after use.
- `filter` should rearrange the elements of L so that all elements l for which $g(l)$ is TRUE appear in their original order before all elements l' for which $g(l')$ is FALSE (these elements should also appear in their original order). `filter` should return the number of elements of L for which $g(l)$ is TRUE.

Example. Suppose $X = \mathbb{Z}$ (the set of integers), $f(x) = x^2$ and $g(x) = \text{TRUE}$ iff x is even. Let $L = [-1, 3, -8, 2]$. Then, `map(L, f)` returns a new array of `int`s containing `[ 1, 9, 64, 4 ]`; `filter(L, g)` changes `L` to `[ -8, 2, -1, 3]` and returns `2`.

Submission format: You should upload a C file (say `map-filter.c`) containing your implementation of the functions. We will write a program, say `tester.c`, which will contain the line `#include "map-filter.c"`. It will also define a number of functions that can be used as `f` or `g`, and test whether your implementation produces the same outputs that we get.

Q4. Your task in this problem is to implement and measure the performance of a *Bloom filter*, a data structure used for inexact searching in sets. Suppose S is a set stored using a Bloom filter. In response to a search for an element x, the Bloom filter returns one of two answers: x is not in S, and I am sure of that, or I think x is in S, but I could be mistaken (this is called a *false positive* error when x is not actually in S). At the end of this question, there is an example from `https://en.wikipedia.org/wiki/Bloom_filter` of a situation where inexact searching using Bloom filters is useful.

How a Bloom filter works[1]**.** An empty Bloom filter is ~~a bit~~ an array A of m ~~bits~~ flags, all set to 0. There are also k different hash functions, say h_1, h_2, ..., h_k, each of which maps an element of S to one of the m array positions in a uniformly random manner.

To add an element x to the Bloom filter, the bits $A[h_1(x)], A[h_2(x)], \ldots, A[h_k(x)]$ are each set to 1. To query for an element, say y, i.e., to test whether $y \in S$, we check $A[h_1(y)], A[h_2(y)], \ldots, A[h_k(y)]$. If any of these bits is 0, then $y \notin S$ (if $y \in S$, then all the bits would have been set to 1 when y was inserted). If all are 1, then the Bloom filter returns "y may be in S". If these bits were set by chance to 1 during the insertion of other elements, this would be a false positive result. Intuitively, if m and k are large, the chances of a false positive are small; as more and more elements are inserted into the Bloom filter, the chance of a false positive increases.

Note that, the space required by a Bloom filter to store a set of n items remains fixed at m, while the space required by exact search structures such as balanced search trees (BSTs) usually grows linearly. On the other hand, the probability of a false positive error for a Bloom filter grows with n, whereas it is always zero for BSTs.

Problem statement. The objective of this question is to study the false positive rate vs. space-efficiency tradeoff for Bloom filters. Given a sequence of non-negative integers ~~(as command-line arguments)~~, possibly with repetitions, insert them in (a) an AVL tree[2], and (b) the Bloom filter defined below.

Bloom filter definition. For a fixed m and k, to insert a non-negative integer x in the Bloom filter, call `srand(x)`; then compute `rand() % m` k times to get k values. Set the ~~bits~~ flags in these k positions to 1 in the Bloom filter.

~~For each element, report whether it was actually inserted (print INSERTED), or if it was already contained in the tree (print DUPLICATE). Measure the time taken to process the sequence; also, compute the total number of nodes in your final balanced search tree, and thus, estimate the total storage space required (in bytes) for the tree.~~

If your data structure reports that the input value was already present, print the name of the data structure (AVL or BLOOM), the serial number of the input (serial numbers start from 0), and its value, followed by a newline. If the value is actually inserted, no output should be printed.

Note that if the AVL tree reports that an input has already been inserted earlier, the Bloom filter will do the same. **The output for the AVL tree should appear before the output for the Bloom filter.** However, the output from the Bloom filter would sometimes be wrong, i.e., your program would print BLOOM, the serial number of the input and its value for some integers that are actually appearing for the first time.

After all input has been processed, print the number of false positive errors made by the Bloom filter. ~~As before, measure the time taken to process the sequence; also, compute the false positive error rate per cent.~~

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

[2] You may use the code provided as course material or any libraries (e.g., GDSL) for this purpose.

~~NOTE: You may use a byte instead of a bit to store 0 and 1, but you will get full credit only if you store the 0 and 1 at the bit level.~~

Input format: the Bloom filter parameters (i.e., the values of m and k), the total number of input values, followed by the actual sequence of input numbers.

Output format: One line for each input that is regarded as a repeat, in the format described above; the number of false positive errors made by the Bloom filter; the values stored in the AVL tree in preorder (all values should be printed on one line, with a space between 2 successive values).

~~Your program should print to stdout a sequence of INSERTED and DUPLICATE that corresponds to the given input. It should also print to stderr the number of bytes and the time taken to process the sequence using an AVL tree, m, k, and the time taken to process the sequence using a Bloom filter.~~

Example: A Web server can use a Bloom filter to determine whether a Web object (a page, an image, etc.) has been requested earlier. If it has been requested at least once earlier, only then it is stored in a cache.[3] If a cached object is requested again (i.e., thrice or more in all), it can be served quickly from the cache; otherwise, the object is served more slowly from the hard disk where it is stored. The reason for this strategy is that a very large proportion of Web objects are requested only once; there is no benefit to storing such objects in the cache.

One way to implement this strategy would involve storing the identifiers of the requested objects in an exact search structure, such as a balanced search tree (BST). Since Web servers service requests for a very large number of objects, such a BST would be large. In response to the question: "Has object x been requested before?", a BST would always correctly reply YES or NO.

Instead, if the identifiers were stored in a Bloom filter, the amount of space needed would be *much* less. Of course, this space saving would come at a cost. In response to the above question, a NO would be guaranteed to be a correct answer, but a YES would sometimes be mistaken, i.e., occasionally, an object being requested for the first time would appear to have been requested earlier, and would end up being cached unnecessarily.

[3] A *cache* is a faster, but much smaller, storage space that is intended to store a frequently accessed subset of the complete data stored in a larger, but much slower, storage medium.