CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 4

For: January 31–February 4

Problem 1. Sorting Review

sort(1…n) {return insert(n, sort(1…n-1))}
T(n) = (n-1)*T(n-1) = n!

(a) How would you implement insertion sort recursively? Analyse the time complexity by formulating a recurrence relation.

Nested if loop, first check for a, then check for b. No need to go into the b check if a is already different

(b) Consider an array of pairs (a,b). Your goal is to sort them by a in ascending order first, and then by b in ascending order. For example, [(2, 1), (1, 4), (1, 3)] should be sorted into [(1,3), (1,4), (2,1)].

MergeSort is stable so we could sort b using SelectionSort first, then sort the a using MergeSort

You are given 2 sorting functions, which are a MergeSort and a SelectionSort. You can use each sort at most once. How would you sort the pairs? Assume you can only sort by one field at a time.

(c) We have learned how to implement MergeSort recursively. How would you implement MergeSort iteratively? Analyse the time and space complexity.

Is it just the merge function can be converted into a loop, incrementing list 1 and list 2?

Problem 2. Queues and Stacks Review

Recall the Stack and Queue Abstract Data Types (ADTs) that we have seen in CS1101S. Just a quick recap, a Stack is a "LIFO" (Last In First Out) collection of elements that supports the following operations:

- **push**: Adds an element to the stack
- **pop**: Removes the **last** element that was added to the stack
- **peek**: Returns the last element added to the stack (without removing it)

Queue: Same as stack but you keep track of the first nonnull element in the array also. Enqueue will happen at the back and dequeue will happen at the front. peek is arr[i].

And a Queue is a "FIFO" (First In First Out) collection of elements that supports these operations:

- **enqueue**: Adds an element to the queue
- **dequeue**: Removes the **first** element that was added to the queue
- **peek**: Returns the next item to be dequeued (without removing it)

Stack: Have external counter for number of nonnull elements in array, so pushing or popping happens at that index, either setting to null, or value. peek is just arr[i]. increment when necessary.

(a) How would you implement a stack and queue with a fixed-size array in Java? (Assume that the number of items in the collections never exceed the array's capacity.)

(b) A Deque (double-ended queue) is an extension of a queue, which allows enqueueing and dequeueing from both ends of the queue. So the operations it would suppport are *enqueue_front*, *dequeue_front*, *enqueue_back*, *dequeue_back*. How can we implement a Deque with a fixed-size array in Java? (Again, assume that the number of items in the collections never exceed the array's capacity.)

Keep track of the two values mentioned earlier and either set to null or value and increment/ decrement

Nothing left in queue - return "done"
Queue too full - return "try again later"

(c) What sorts of error handling would we need, and how can we best handle these situations?

(d) A set of parentheses is said to be balanced as long as every opening parenthesis "(" is closed by a closing parenthesis ")". So for example, the strings "()()" and "(())" are balanced but the strings ")(())(" and "((" are not. Using a stack, determine whether a string of parentheses are balanced.

If "(" then increment counter
Else check if arr[counter] == "(" else false

Problem 3. Stac and Cue
(Adapted from Trapping Rain Water)

You've been promoted to the chief of a tribe called Stac on the island of Cue! To prove your right to rule, you must first pass a divine test. Your tribe's astrologer predicted that a long season of rain will befall the town. You became concerned, as some of your tribe members live in low-lying areas, which will likely be flooded as a result of the rain. Your duty is to ensure that all of your affected residents are evacuated safely.

The tribe's houses are arranged in a single line, with one house at each index. When it rains, any area that is enclosed between 2 places of a higher elevation will be flooded (refer to the example below). Only the peaks that are not enclosed are safe.

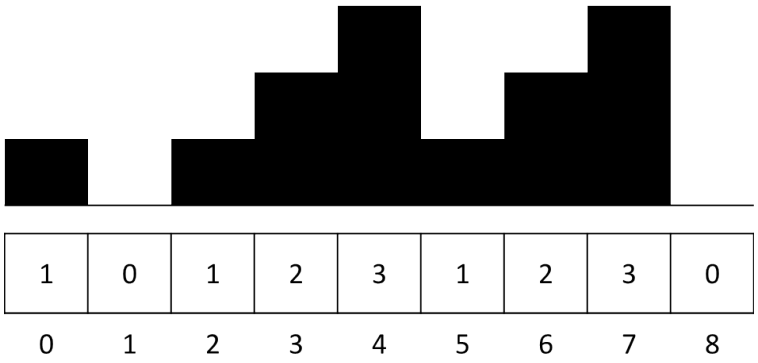
Given the map of your entire village (which shows the elevation of each house in consecutive order), determine the **number of houses** that needs to be evacuated from the area (i.e. how many houses / indexes would have water above them).

Example:

[1, 0, 1, 2, 3, 1, 2, 3, 0]

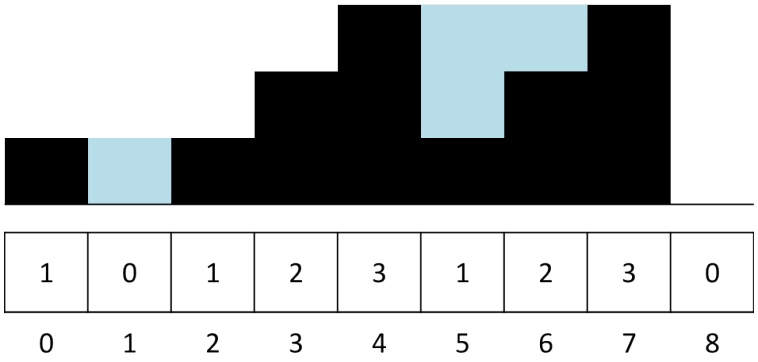
loop through index 1 to len - 2 and check whether there's something on both sides that's taller, doesn't have to be right beside

The array above can be represented with the diagram below.



For the house at index 1, the rain will flood the place, because it is surrounded by index 0 and 2 (which are of elevation 1). For the houses at index 5 and 6, the rain will also flood the place, because the elevation at index 4 and 7 is higher. Therefore, only 3 houses would need to be evacuated (houses 1, 5 and 6).

For simplicity, we would assume that the houses at the edges will not be flooded.



Problem 4. Sorting with Queues

(Optional) Sort a queue using another queue with $O(1)$ additional space.