CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 4

For: January 30–February 3

Problem 1. Time Complexity Analysis
Analyse the following code snippets and find the best asymptotic bound for the time complexity of the following functions with respect to n.

(a)
```
public int niceFunction(int n) {
    for (int i = 0; i < n; i++) {
        System.out.println("I am nice!");
    }
    return 42;
}
```

O(n)

(b)
```
public int meanFunction(int n) {
    if (n == 0) return 0;
    return 2 * meanFunction(n / 2) + niceFunction(n);
}
```

T(n) = T(n/2) + n
= T(n/4) + n + n/2
= T(n/n) + n + n/2 + n/4 ... = 1+ 1/2 + 1/4 ...
= n

(c)
```
public int strangerFunction(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < i; j++) {
            System.out.println("Execute order?");
        }
    }
    return 66;
}
```

O(n^2)

I= 1,2...n
j=0,1,2...n
(1+n)n/2 = n^2

(d)
```
public int suspiciousFunction(int n) {
    if (n == 0) return 2040;

    int a = suspiciousFunction(n / 2);
    int b = suspiciousFunction(n / 2);
    return a + b + niceFunction(n);
}
```

O(nlogn)

T(n) = T(n-1) + T(n-2) + 1
=O(2^n)

(e)
```
public int badFunction(int n) {
    if (n <= 0) return 2040;
    if (n == 1) return 2040;
    return badFunction(n - 1) + badFunction(n - 2) + 0;
```

O(2^n)

```
    }
```

log(1) + log(2)...log(n)
log(1*2*3...n)
log(n!) -> can be lower bound by nlogn

as log(n!) <= log(n) + log(n)...log(n)
hence log(n!) <= nlog(n)

log(n!) = n*n-1*n-2..(n/2) <= (n/2)(n/2)(n/2)...(n/2)
= log((n/2)^(n/2))
= n/2(log(n/2)) = nlog(n)

(f)
```
    public int metalGearFunction(int n) {
        for (int i = 0; i < n; i++) {
            for (int j = 1; j < i; j *= 2) {
                System.out.println("!");
            }
        }
        return 0;
    }
```
O(nlogn)

(g)
```
    public String simpleFunction(int n) {
        String s = "";
        for (int i = 0; i < n; i++) {
            s += "?";
        }
        return s;
    }
```
O(n^2)

Problem 2. Sorting Review

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

(b) Consider an array of pairs (a, b). Your goal is to sort them by a in ascending order. If there are any ties, we break them by sorting b in ascending order. For example, [(2, 1), (1, 4), (1, 3)] should be sorted into [(1, 3), (1, 4), (2, 1)].

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.

Sort B by selection to order priority, then sort A by merge Sort, as it is stable, the order remains for B even after sorting

Problem 3. 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

use a global variable pointer to keep track of the stack pointer and queue pointer

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

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)

(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.)

(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.

use stack, if) added on empty stack, throw error
if using integer, if (+1, if) -1, if end of
operation int != 0 or <0, throw error

Problem 4. 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.

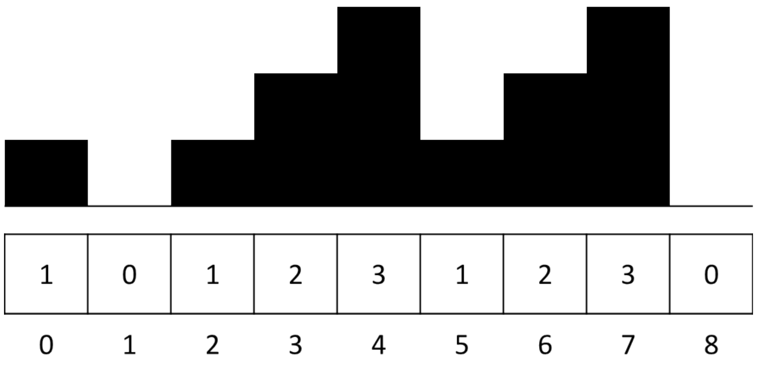
The tribe's houses are arranged in a single line, with one house at each index. When it rains, any area that is enclosed between 2 places of a higher elevation will be flooded (refer to the example below). Only the peaks that are not enclosed are safe.

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:

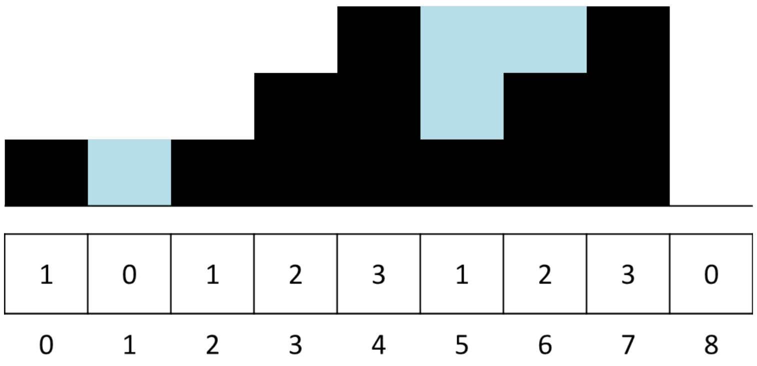
$$[1, 0, 1, 2, 3, 1, 2, 3, 0]$$

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.



(5,5,1) since we know there is no other max after the 5 on left, the algorithm works

Problem 5. Sorting with Queues
(Optional) Sort a queue using another queue with $O(1)$ additional space.

a) void recursive_insertion_sort(int arr[], int n) {
```
   // Base case
   if (n <= 1)
      return
   // Sort first n-1 elements
   recursive_insertion_sort( arr, n-1 )
   int val = arr[n-1]
   int pos = n-2
   while (pos >= 0 && arr[pos] > val) {
      arr[pos+1] = arr[pos]
      pos = pos - 1
   }
   arr[pos+1] = val
}
```

b)

c)
```
 for (curr_size = 1; curr_size <= n-1;
                curr_size = 2*curr_size)
     {

         // Pick starting point of different
         // subarrays of current size
         for (left_start = 0; left_start < n-1;
                  left_start += 2*curr_size)
         {
            // Find ending point of left subarray. mid+1 is starting point of right
            int mid = Math.min(left_start + curr_size - 1, n-1);

            int right_end = Math.min(left_start
                     + 2*curr_size - 1, n-1);

            // Merge Subarrays arr[left_start...mid]
            // & arr[mid+1...right_end]
            merge(arr, left_start, mid, right_end);
         }
     }
```