CS2040S: Data Structures and Algorithms

Discussion Group Problems for Week 4

For: February 5–February 9

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;
}
```

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

(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);
}
```

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

```
}
```

Note: The nth Fibonacci number can be expressed as $F_n = \frac{\Phi^n - (1-\Phi)^n}{\sqrt{5}}$, where $\Phi = \frac{1+\sqrt{5}}{2}$. E.g. $F_0 = 0$, $F_1 = 1$.

(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;
}
```

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

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.

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
- **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 sequence 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.

(e) A sequence of opening and closing parentheses of three different types {}, (), and [] is given. We will define a hyperbalanced parenthesis sequence in a recursive way. For the recursion base, we say that empty parentheses sequence is a hyperbalanced one. Next, if "X" and "Y" are hyperbalanced sequences, then any of the sequences "{X}", "[X]", "(X)", and "XY" is hyperbalanced. For example, sequences "{}([{}])", "{}", "((((()))))" are hyperbalanced, but sequences "{(})", "]", "({}" are not. Determine whether a given sequence of parentheses is hyperbalanced.

Problem 4. Mountain stack

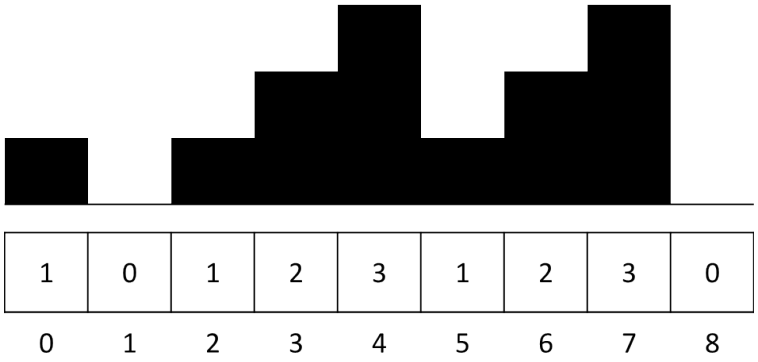
Adventure flows in your veins, and today, you embark on your journey through the mountains! The range consists of n hills, aligned linearly and numbered from 0 to $n-1$. Each hill, indexed at i, stands at a height of $a[i]$ meters and spans a length of 1 kilometer. As you plan your journey, you're keen to understand how far you can see to your left (you choose not to look right due to the prevailing winds in the mountains). When you stand atop hill i and look leftward, you'll see a point j kilometers away if and only if all hills from $i-j$ to i are no taller than $a[i]$ meters. For each hill i, determine the maximum distance you can see from it. If there are no hills $j < i$ taller than i then the answer for i is "infinity". Your goal is to solve this problem as effectively as possible.

(Hint: try using a stack to achieve $O(n)$ time complexity).

Example:

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

The array above can be represented with the diagram below.



For hills numbered 0, 2, 3, 4, and 7, there are no taller hills to their left; therefore, the answer for them is "infinity". For the hill with index 1, it is impossible to see even 1 km to the left, as the hill at index 0 is taller, and thus the answer is 0. The same applies to the hill numbered 5, as $a[4] > a[5]$. However, when standing on top of hill 6, one can observe 1 km to the left, since $a[5] \leq a[6]$ and does not limit the sight. But the view is limited beyond that, as $a[4] > a[6]$.

Problem 5. Sorting with Queues

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