CSM 61B Spring 2019

Final Review

Mentoring 13: April 29, 2019

1 Running Out of Time

1.1 Give a tight runtime bound for `mystery` as a function of N, the length of the input array.

```
public static int[] mystery(int[] array) {
    boolean done = false;
    while (!done) {
        done = true;
        for (int i = 1; i < array.length; i++) {
            if (array[i-1] > array[i]) {
                done = false;
                int tmp = array[i-1];
                array[i-1] = array[i]
                array[i] = tmp;
            }
        }
    }
    return array;
}
```

This sorting algorithm, known as bubble sort, makes repeated passes through the array, swapping adjacent elements if they are out of order. The algorithm terminates if it completes a pass without swapping any elements.

Each pass must look through all elements of the array, and in the worse case, if the initial array is in reverse sorted order, makes N passes through the array. The runtime for this function is $O(N^2)$.

1.2 Give a tight asymptotic bound for `convoluted` as a function of N, the length of the input arrays `a` and `b`. If possible, give a $\Theta(\cdot)$ bound for the overall runtime. Otherwise, provide a $\Theta(\cdot)$ bound for both the best case and worst case runtime.

```
int[] convoluted(int[] a, int[] b) {
    assert a.length == b.length;
    int[] result = new int[a.length];
    for (int i = 0; i < result.length; i += 1) {
        for (int j = 0; j <= i; j += 1) {
            result[i] += a[j] * b[i];
        }
    }
    return result;
}
```

$\Theta(N^2)$

For the first iteration of the outer for loop, the inner loop will run once. The next iteration, the inner loop runs two times. Each subsequent iteration of the outer loop increases the number of iterations of the inner loop by 1. The total runtime is given by $\sum_{i=1}^{N} = \Theta(N^2)$

1.3 Give a tight asymptotic bound for `debugBinarySearch` as a function of N, the length of the input array, `a`. If possible, give a $\Theta(\cdot)$ bound for the overall runtime. Otherwise, provide a $\Theta(\cdot)$ bound for both the best case and worst case runtime.

```
boolean debugBinarySearch(int[] a, int target) {
    int start = 0;
    int end = a.length - 1;

    while (start <= end) {
        int mid = ((end - start) / 2) + start;
        if (a[mid] == target) {
            return true;
        } else if (a[mid] < target) {
            start = mid + 1;
        } else {
            end = mid - 1;
        }

        System.out.print("Searching: [ ");
        for (int i = start; i <= end; i++) {
            System.out.print(a[i] + " ");
        }
        System.out.println("]");
    }
    return false;
}
```

$\Theta(N)$ in the worst case, $\Theta(1)$ in the best case.

Normally binary search runs in $\Theta(log(N))$ in the worst case. However, for this function, it also prints out the values of the list it is considering for each iteration of the while loop. In the worst case, you would perform $log(N)$ iterations of your while loop, and your for loop would run $\frac{N}{2}+\frac{N}{4}+...+4+2+1$ times, which would be $\Theta(N)$ runtime. However, in the best case, you could find your target immediately and return in constant time.

1.4 Give a tight asymptotic bound for `mystery`.

```
int mystery(int N) {
    if (N == 0) {
        return 0;
    }
    return mystery(N/3) + mystery(N/3) + mystery(N/3);
}
```

$\Theta(N)$

This is a recursive function with a branching factor of 3. If you draw a tree with all the recursive calls, there will be log(N) levels before the function terminates. There is 1 function call at the top level, 3 recursive calls at the next level, 9 recursive calls at the third level, and each subsequent level triples the number of recursive calls. Since each recursive call does constant work, the total runtime is $3^0+3^1+3^2+...+3^{log(N)} = 1+3+9+...+N = \Theta(N)$

2 Out of Sorts

2.1 Each column below gives the contents of a list at some step during sorting. Match each column with its corresponding algorithm.

· Merge sort · Quicksort · Heap sort · LSD radix sort · MSD radix sort

For quicksort, choose the topmost element as the pivot. Use the recursive (top-down) implementation of merge sort.

	Start	A	B	C	D	E	Sorted
1	4873	1876	1874	1626	9573	2212	1626
2	1874	1874	1626	1874	7121	8917	1874
3	8917	2212	1876	1876	9132	7121	1876
4	1626	1626	1897	4873	6973	1626	1897
5	4982	3492	2212	4982	4982	9132	2212
6	9132	1897	3492	8917	8917	6152	3492
7	9573	4873	4873	9132	6152	4873	4873
8	1876	9573	4982	9573	1876	9573	4982
9	6973	6973	6973	1897	1626	6973	6152
10	1897	9132	6152	3492	1897	1874	6973
11	9587	9587	7121	6973	1874	1876	7121
12	3492	4982	8917	9587	3492	9877	8917
13	9877	9877	9132	2212	4873	4982	9132
14	2212	8917	9573	6152	2212	9587	9573
15	6152	6152	9587	7121	9587	3492	9587
16	7121	7121	9877	9877	9877	1897	9877

From left to right: unsorted list, quicksort, MSD radix sort, merge sort, heap sort, LSD radix sort, completely sorted.

MSD Look at the left-most digits. They should be sorted. Mark this immediately as MSD.

LSD One of the digits should be sorted. Start by looking at the right most digit of the remaining sorts. Then check the second from right digit of the remaining sorts and so on. As soon as you find one in which at

least something is sorted, mark that as LSD.

Heap Max-oriented heap so check that the bottom is in sorted order and that the top element is the next max element.

Merge Realize that the first pass of merge sort fixes items in groups of 2. Identify the passes and look for sorted runs.

Quick Run quicksort using the pivot strategy outlined above. Look for partitions and check that 4873 is in its correct final position.

3 T,F,G,V,E

3.1 State if the following statements are True or False, and justify. For all graphs, assume that edge weights are positive and distinct, unless otherwise stated.

(a) Adding some positive constant k to every edge weight does not change the shortest path tree from vertex S.

False.

(b) Doubling every edge weight does not change the shortest path tree.

True.

(c) Adding some positive constant k to every edge weight does not change the minimum spanning tree.

True.

(d) Doubling every edge weight does not change the minimum spanning tree.

True.

For the four parts above, we can consider when graph transformations affect the two algorithms:
MST algorithms depend on the relative order of *edge weights*. Hence, adding a constant, or doubling the edge weights does not alter the MST. (More broadly, any monotonically increasing function can be applied, such as squaring the edge weights, assuming they are all positive.)

Shortest path algorithms depend on the relative order of *sums of edge weights*. More specifically, we are concerned about sums of edge weights that represent paths to vertices in the graphs. We can see then that adding a constant k to all edge weights does alter the relative order of these sums. In fact, as k increases, the algorithm becomes more biased towards paths that are shorter in *hop-length*, i.e. number of vertices in the path. One intuitive way to think about this would be to make k a very large number, tending towards infinity. Then all edge weights are approximately the same length, and shortest path algorithms will find the shortest path by hop-length, just like BFS. On the other hand, if we double every edge weight, the relative order of sums does not change. $2w_1+2w_2+2w_3 = 2(w_1+w_2+w_3)$. We see that we can factorize out the multiplier, and the ordering is still dependent on the original sums of edge weights. More broadly, we can consider any positive multiplication of edge weights to not affect shortest path trees.

(e) Let $(S, V - S)$ be a specific cut of the graph. If an edge e is not the lightest edge across this cut, it cannot be a part of any MST.

False. Consider the graph $\{(A, B, 1), (B, C, 2)\}$. Even though edge (B, C) is not the lightest edge across the cut $\{A\}, \{B, C\}$, it is necessarily still a part of all MSTs (since this graph is a tree).

(f) If an edge e is the lightest edge connected to vertex S, it must be a part of the shortest path tree from vertex S.

True. If e connects S to T, then that must be the shortest path from S to T. Assume there is some shorter path to T from S. Then it must exit S via edge e' which has strictly larger weight than e, creating a contradiction.

4 Roleplaying Game

4.1 You are the king of a large kingdom! In order to manage your kingdom, you have appointed lords to rule towns within your kingdom. Every lord can govern over his town and any town that he is connected to by road. Your job as king is to figure out the optimal way to allocate lords and build roads.

Formally, consider a graph G with vertices V and edges E. Each vertex v represents a town. It has an associated cost c, the cost of installing a lord in the town. Each edge e represents a potential road. It has an edge weight w, the cost of building that road. Devise an algorithm that can efficiently compute which towns to install lords in and which roads to build, such that every town in the kingdom is governed (either has a lord in it or is connected by some number of roads to a town with a lord in it).

We can formulate this problem as a Minimum Spanning Tree problem. We create a dummy node S. We connect S to every vertex with edge weight c, the cost of that vertex. We then find the MST of this modified graph, and that is the solution. Why does this method work? We know that the MST must include S, by definition of spanning tree. Every edge in te MST that is outgoing from S represents a selected town. In the MST, every vertex is either directly connected to S, i.e. a town with a lord, or connected to S by a series of selected edges, i.e. connected to some town with a lord via some roads. Since the MST finds the minimum cost solution, this is our desired arrangement of lords and roads

5 Largest Perimeter Triangle

5.1 Given an array A of positive lengths, return the largest perimeter of a triangle with non-zero area, formed from 3 of these lengths. Recall the Triangle Inequality, which states that for any triangle, the sum of the lengths of any two sides must be greater than or equal to the length of the remaining side ($a + b > c$). If it is impossible to form any triangle of non-zero area, return 0.

For example, A = [2, 1, 2] returns 5. A = [1, 2, 1] returns 0. A = [3, 2, 3, 4] returns 10.

What is the runtime of your solution?

```
public int largestPerimeter(int[] A) {




}
```

Note: this problem was adapted from LeetCode (https://leetcode.com/problems/largest-perimeter-triangle/).

Note: this solution was adapted from LeetCode (https://leetcode.com/problems/largest-perimeter-triangle/solution/).

Without loss of generality, say the side lengths of the triangle are $a \leq b \leq c$. The necessary and sufficient condition for these lengths to form a triangle of non-zero area is $a + b > c$.

Say we knew c already. There is no reason not to choose the largest possible a and b from the array. If $a + b > c$, then it forms a triangle, otherwise it doesn't.

This leads to a simple algorithm: sort the array. For any c in the array, we choose the largest possible $a \leq b \leq c$: these are just the two values adjacent to c. If this forms a triangle, we return the answer. Else, we return 0.

Thus, we complete our function as follows:

```
public int largestPerimeter(int[] A) {
    Arrays.sort(A);
    for (int i = A.length - 3; i >= 0; --i) {
        if (A[i] + A[i+1] > A[i+2]) {
            return A[i] + A[i+1] + A[i+2];
        }
    }
    return 0;
}
```

This function takes $O(N \log N)$ time to sort A and $O(N)$ time to iterate through the for loop. Thus, the overall runtime is given by $O(N \log N)$.