

# CS 312 - Algorithm Design

## Midterm - March 13

This exam is open book and open notebook. You are expected to complete this exam in class. Please write your answers in the blue books provided. Please do not hesitate to ask questions if you are confused by the wording of any of these questions.

1. The median of a list of numbers is the number such that half the numbers have a greater value and half have a lesser value. For example, in the list 7, 1, 2, 99, 100, the median is 7 since there are 2 numbers less than 7 and 2 numbers greater than 7.

Here are 2 algorithms to find the median in a list of numbers. To keep things simple, assume that the list has an odd number of elements in it.

```
FindMedian1 (list) {
    n = size of list
    mid = n / 2 + 1;    // Use integer division
    for i = 1 to mid {
        largest = list[1]
        largestLocation = 1
        for j = 2 to n-(i-1) {
            if list[j] > largest {
                largest = list[j]
                largestLocation = j
            }
        }
        swap (list [n - (i-1)], list[largestLocation])
    }
    return list[mid]
}
```

```
FindMedian2 (list) {
    n = size of list
    mid = n / 2 + 1;    // Use integer division
    for i = 1 to mid {
        largest = list[n - (i-1)]
        for j = 1 to n-i {
            if list[j] > largest {
                largest = list[j]
                swap (list [n - (i-1)], list[j])
            }
        }
    }
    return list[mid]
}
```

- a. (5 pts.) What is the  $O()$  cost of each algorithm?

The outer loop of each algorithm always executes  $n/2 + 1$  times. The inner loop between 1 and  $n$  times. This gives a combined cost of  $O(n^2)$ .

- b. (5 pts.) Given the input list 7, 1, 2, 99, 100, what is the value of the list when FindMedian1 completes?

2 1 7 99 100

- c. (5 pts.) Given the input list 7, 1, 2, 99, 100, what is the value of the list when FindMedian2 completes?

2 1 7 99 100

- d. (5 pts.) What ordering of the values 7, 1, 2, 99, 100 in the input list would result in the fastest execution of FindMedian1?

The fastest execution time occurs when the if-statement is truly as infrequently as possible since everything else is fixed no matter what the input is. On the first iteration of the loop, the if-statement will never be true if the largest value is initially in position 1 of the list.

At the end of the first iteration of the outer loop, the entry in the first position will be swapped with the value in the last position. To avoid the if statement being true on the next iteration, this means the swap should move the 2<sup>nd</sup> largest value into the first position. So, the 2<sup>nd</sup> largest value should be in the last position initially.

Similarly, the next iteration of the outer for loop will move 99 to the second last position. We would like that swap to result in the 3<sup>rd</sup> largest value being swapped to the first position.

So, the fastest execution happens with this input:

100 1 2 7 99

- e. (5 pts.) What ordering of the values 7, 1, 2, 99, 100 in the input list would result in the fastest execution of FindMedian2?

This one is a bit easier to see. In this case, on the first iteration of the outer for loop, the if-statement will never be true if the largest value is already in the last position. On the next iteration, no swap will occur if the largest value is in the 2<sup>nd</sup> last position, and so on. As a result, a sorted list will have no swaps and the fastest execution:

1 2 7 99 100

- f. (10 pts) Provide an algorithm for finding a median that has a cost of  $O(n \log n)$ .

The simplest way to do this is to sort the array and return the value in the mid position.

- g. (10 pts) Provide an algorithm for finding a median that is  $O(n)$  for **some**, but not all, inputs. What ordering of the input list 7, 1, 2, 99, 100 allows this algorithm to be lin-

ear? (Don't even think about turning in the algorithm in Chapter 13 of the book! You will get no credit for that. Don't waste your time looking at it, either. You will get no insight from doing so.)

First, check if the list is already sorted. If it is, return the middle of the list. This can be done in  $O(n)$  time. If the list is not sorted, fall back on the  $O(n \log n)$  algorithm of part f. This will be linear for the order 1 2 7 99 100.

```
sorted = false
for i = 1 to n-1
    if list[i] > list[i+1]
        sorted = false
if sorted
    return list[mid]
else
    sort the list
    return list[mid]
```

2. Suppose you have  $N$  jobs to schedule on a computer. For each job, you know the length of time that job will take. There are no restrictions on when a job must start nor when it must be finished by. Your goal is to minimize the average completion time of the jobs.
- a. (15 pts.) It turns out that the optimal schedule can be found using a greedy algorithm. Provide a pseudocode greedy algorithm that finds the optimal schedule.

Sort the jobs by job length from shortest to fastest. Do the jobs in that order.

- b. (15 pts.) Prove that the schedule produced by your algorithm is always optimal.

Do a proof by exchange.

Any schedule with idle time can be improved by removing the idle time. Therefore, the optimal schedule must have no idle time. Note that the greedy algorithm produces schedules with no idle time.

If the greedy algorithm is not optimal, then there must be another algorithm that produces a better schedule. This schedule must have one or more inversions where job  $i$  follows job  $j$  even though length of job  $i$  is less than the length of job  $j$ . Find a pair of consecutive jobs that are inverted in this schedule. We next show that undoing the inversion can only improve the average completion time.

Swapping the order of jobs  $i$  and  $j$  does not affect the completion time of any other job. As a result, we only need to consider whether doing job  $i$  first or job  $j$  first will reduce the average completion time.

Define  $s_i$  as the start time for job  $i$  in the greedy schedule. Then, the finish time of job  $i$  in the greedy schedule is  $f_i = s_i + \text{length}_i$ . The finish time of job  $j$  in the greedy schedule is  $f_j = s_i + \text{length}_i + \text{length}_j$ .

In the schedule with an inversion, we have:

$$s_j' = s_i$$

$$f_j' = s_i + \text{length}_j$$

$$f_j' = s_i + \text{length}_i + \text{length}_j$$

So the last of jobs  $i$  &  $j$  finish at the same time in both schedules. As a result, the only thing that will affect the average completion time is the time at which the first of jobs  $i$  and  $j$  finishes. The greedy algorithm picks the shorter job and thus the completion time of job  $i$  in the greedy schedule is earlier than the completion time of job  $j$  in the schedule with an inversion. Therefore, the greedy algorithm will result in a smaller average completion time than a schedule that contains inversions.

3. (25 pts) Consider Prim's Algorithm for finding the minimum spanning tree and Breadth-First Search.

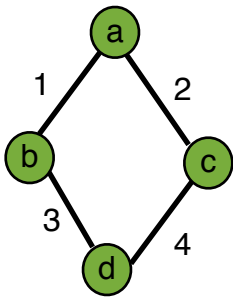
Is it the case that nodes are always, sometime or never added to the trees created by the 2 algorithms in the same order? That is, given a graph  $G$ , run Prim's Algorithm recording the order in which the nodes are added to the tree. Then run BFS on the same original graph and record the order in which nodes are added to the BFS tree. Are the nodes in these 2 lists always, sometimes or never in the same order.

If your answer is sometimes, provide 2 examples. In one example they should be added in the same order. In the other example, they should be added in different orders. Keep your examples simple but include at least 4 nodes and have at least one path that uses at least 2 edges.

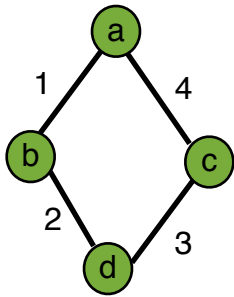
If your answer is always or never, explain your answer. You do not need a formal proof but your explanation should be convincing.

Sometimes.

Here is a graph in which the BFS algorithm and Prim's algorithm can pick the same ordering:



Prim's algorithm would pick the nodes in the following order: a b c d  
 BFS would pick the nodes either in that same order or in the order:  
 a c b d  
 In this example, the ordering is sometimes the same.



In this example, Prim's algorithm would pick the nodes in the following order: a b d c  
In this case, BFS cannot produce that ordering.