EE360C: Algorithms
The University of Texas at Austin — Exam #1
Dr. David Soloveichik — October 3, 2018

Name: **EID:**

Section (circle): morning 11am-12:30pm / evening 5pm-6:30pm

Exam #1

Instructions. WRITE LEGIBLY.
No calculators, laptops, or other devices are allowed. This exam is **closed book**. Write your answers on the test pages. **If you need scratch paper, use the back of the test pages, but indicate where your answers are.** Write down your process for solving questions and intermediate answers that may earn you partial credit.

If you are unsure of the meaning of a specific test question, write down your assumptions and proceed to answer the question on that basis. When asked to describe an algorithm, you may describe it in English or in pseudocode.

If you write information in response to a question, and that information is incorrect, you will not earn full credit. In the same vein, if a question asks for a finite number of things, and you provide "extra" things, we will ignore anything extra, and grade only the first answers.

You have **90 minutes** to complete the exam. The exam is **100 points** total.

Some sums we used in class, which you may or may not find useful:

Arithmetic series:

$$\sum_{k=1}^{n} k = \frac{1}{2}n(n+1)$$

From computing the number of nodes in a full binary tree of height h:

$$\sum_{l=0}^{h} 2^l = 2^{h+1} - 1$$

From BuildHeap analysis:

$$\sum_{k=0}^{\infty} k/2^k = 2$$

Problem 1: Asymptotic Analysis

(a) [10 points] Prove or disprove: $3^n = O(2^n)$.

Solution

False. Using limit theorem: $\lim_{n\to\infty} 3^n/2^n = \lim_{n\to\infty}(3/2)^n = \infty$ since $3/2 > 1$. Thus 3^n grows asymptotically faster than 2^n.

(b) [10 points] Prove that $f(n) + g(n) = \Theta(\max(f(n), g(n)))$, assuming both f and g are non-negative. Hint: don't use the limit theorem (the limit might not exist).

Solution

For the upper bound, note that always $f(n) + f(n) \leq 2\max(f(n), g(n))$. Thus we satisfy the definition of O with $n_0 = 1$ and $c = 2$.

For the lower bound, note that always $\max(f(n), g(n)) \leq f(n)+f(g)$ if f, g are non-negative. Thus we satisfy the definition of Ω with $n_0 = 1$ and $c = 1$.

Problem 2: Stable Matching
[10 points]

The stable roommates problem is a version of the stable marriage problem in which each person ranks *all of the others* in order of preference. (There are no separate "men" or "women".) Naturally, we say that a matching is unstable if there are two persons, each of whom prefers the other to his partner in the matching. In other words, the following is an example of an instability: p_1 is matched to p_2, and p_3 is matched to p_4, but p_1 prefers p_3 to p_2 and p_3 prefers p_1 to p_4. The goal is to assign roommate pairs so that none are unstable. (We assume there is an even number of people so there is always a way to match them up.)

Consider the following rankings (as usual, the rankings are from most favored to least favored. I.e., A ranks B his top choice, then C, then D):

A	B	C	D
B	C	A	D
C	A	B	D
D	A	B	C

Find a stable matching for the above set of candidate roommates or demonstrate that one does not exist.

Solution

Does not exist. A valid solution for this exam question tries all possible matchings (there are 3 of them), showing an instability in every case.

Problem 3: Build-a-heap

[10 points]

Recall from class that given an array A in arbitrary order, we can use Heapify-Down repeatedly to construct a min-heap from A in $\Theta(n)$ time. Describe how you can use Heapify-Up repeatedly (instead of Heapify-Down) to construct a min-heap from A.

Prove the asymptotically tight worst-case running time of your algorithm. In other words, give a function $f(n)$ such that the running time is $\Theta(f(n))$ and prove it.

If your worst-case time complexity is worse than $\Theta(n)$, explain at the high level why using Heapify-Down is faster than Heapify-Up in this context.

Solution

We go through $A[i]$ for $i = 1$ to n, calling Heapify-Up on each element. If $A[1]$ through $A[i-1]$ is already a heap, then calling Heapify-Up on $A[i]$ is equivalent to the procedure for inserting a new element into a heap that we proved in class.

Running time:

Upper bound: We run Heapify-Up on n elements, with each call taking $O(\log n)$ time, so overall this is $O(n \log n)$.

Lower bound: Suppose that n is such that the last layer of the tree is completely filled. Further, suppose that originally A is sorted from largest to smallest. Consider the work needed to just call Heapify-Up on the bottom layer (leaves): each subsequent Heapify-Up call must move each of these elements all the way to the top of the tree since it is the smallest element so far. So for every leaf we must do $\Omega(\log n)$ work. Our binary tree of n elements has $(n+1)/2$ nodes in the bottom layer (leaves), which is $\Omega(n)$. Thus overall the work is $\Omega(n \log n)$.

Combining the upper and lower bounds we get $\Theta(n \log n)$.

The reason why this is less efficient than the BuildHeap procedure using Heapify-Down described in class, is that we are doing most work for the layers of the tree with the most nodes (i.e., the bottom layers). In contrast, when we used Heapify-Down in class, we were doing the most work for the top layers with the fewest nodes, and least work for the bottom nodes with the most nodes.

Problem 4: Algorithm Design

Given a *sorted* array S of n integers and another integer z you want to determine if S contains two numbers x and y that sum to z.

Your friend, who hasn't taken Algorithms, comes up with the following algorithm:

(assume one-indexed arrays)

```
1  for i = 1 to n − 1
2    for j = i + 1 to n
3      if S[i] + S[j] == z return YES
4  return NO
```

(a) [10 points] Prove the asymptotically tight worst-case running time of your friend's algorithm. In other words, give a function $f(n)$ such that the running time is $\Theta(f(n))$ and prove it.

Solution

The worst case running time is when such x and y do not exist. The inner loop runs $n-1, n-2, \ldots, 2, 1$ times, for a total of $\sum_{k=1}^{n-1} = \frac{1}{2}(n-1)(n)$ by the arithmetic series formula. This is $\Theta(n^2)$.

(b) [10 points] You point out to your friend that their algorithm doesn't use the fact that the array S is sorted, and that you can do better. Come up with a faster algorithm and prove its asymptotically tight worst-case running time.

Solution

Go through S and for each element x use binary search to try to find $y = z - x$. If found, return YES; otherwise return NO.

Running time: In the worst case, we go through n elements, and for each the binary search takes $\Theta(\log n)$ time. (Note that the worst case running time is when such a y is never found, and the binary search takes $\Omega(\log n)$ time.) So overall this is $\Theta(n \log n)$.

(c) [20 points] (This is challenging—try to do the rest of the exam first)
Come up with an $O(n)$ algorithm for this problem. Prove its correctness and its worst-case running time.

Solution

```
1  l = 1
2  r = |A|
3  while l < r :
4     if (S[l] + S[r] == z) return YES
5     if (S[l] + S[r] > z)
6        r = r − 1
7     else  l = l + 1
8  return NO
```

Running time: The while loop runs at most n times, and each operation in the while loop takes $O(1)$ time, for an overall $O(n)$ time.

Proof of correctness:
If S does not contain x and y that sum to z, then the algorithm will return NO since the equality test must always fail.
Otherwise, S contains x and y that sum to z. How do we know the algorithm won't "miss" these? Without loss of generality, assume the left pointer reached x before the right pointer reached y, i.e., $S[l] = x$ and $S[r] > y$. Then the algorithm would decrement r without changing l such that eventually $S[r] = y$. A similar argument can be made if the right pointer r reached y before l reached x.

Problem 5: Checking strong connectivity
[20 points]

Give an $O(n+m)$ algorithm for checking if a directed graph is strongly connected. (Recall n counts the nodes and m counts the edges. Recall that a directed graph is strongly connected if, for every two nodes u and v, there is a path from u to v and a path from v to u.) Prove your algorithm's correctness and running time.

Hint: BFS/DFS solves the problem of finding all nodes reachable from a given node u. Can you think of how to solve the problem of finding all nodes from which u is reachable?

Solution

Algorithm: Pick any node s. Run BFS (or DFS) from s in G. If not all nodes reached, return false. Otherwise, construct G_{rev} which is the same as G but with all edges reversed (i.e., edge (u, v) becomes edge (v, u)). Run BFS (or DFS) from s in G_{rev}. If not all nodes reached, return false; otherwise return true.

Running time analysis: Running BFS/DFS twice is $O(n + m)$. Generating G_{rev} from G can be done by a single traversal of all nodes and edges, so in $O(n + m)$ time. So overall this algorithm is $O(n + m)$.

Proof of correctness:

First, note that G is strongly connected iff every node is reachable from s and s is reachable from every node. Proof: ($\Rightarrow$) If some node v is not reachable from s or s is not reachable from v then clearly G is not strongly connected. ($\Leftarrow$) If every node is reachable from s and s is reachable from every node, then there is path from any node to any other by going through s.

Finally, observe that the algorithm is checking for the condition "every node is reachable from s and s is reachable from every node." The first part is checked by the first run of BFS/DFS. The second part is checked by the second run of BFS/DFS: the existence of a path from s to v in G_{rev} is equivalent to the existence of a path from v to s in the original G.