EE360C: Algorithms
The University of Texas at Austin — Exam #2
Dr. David Soloveichik — October 31, 2018

Name: **EID:**

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

Exam #2

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.**

When asked to describe an algorithm, you may describe it in English or in pseudocode. You can use results and proofs from class without re-proving them. You can use algorithms and data structures from class without re-explaining how they work. If you use a data structure or algorithm we haven't covered in class, you will need to prove all relevant properties of it and explain how to implement it (you can't take it for granted.)

If you write a correct answer but include additional incorrect information in response to a question, you will lose points.

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

Master Theorem Reminder:
Let $T(n)$ be defined by the recurrence $T(n) = aT(n/b)+f(n)$ (where $a \geq 1$ and $b > 1$ are constants). Then $T(n)$ can be bounded asymptotically as follows:

(a) If $f(n) = O(n^{\log_b a-\epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.

(b) If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \lg n)$

(c) If $f(n) = \Omega(n^{\log_b a+\epsilon})$ for some constant $\epsilon > 0$ and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Problem 1: Divide and Conquer

[25 points]

Suppose you are choosing between the following three algorithms:

- Algorithm A solves problems of size n by dividing them into five subproblems of size $n/2$, recursively solving each subproblem, and then combining the solutions in linear time.
- Algorithm B solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.
- Algorithm C solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.

What are the running times of each of these algorithms (in asymptotic notation), and which would you choose (i.e., which is asymptotically faster)?

Hints: $\log_2 5 \approx 2.32$. Also note that the recurrence for Algorithm C isn't of the form needed for the Master Theorem.

Solution

Algorithm A: $a = 5, b = 2$. We compare $f(n) = \Theta(n)$ with $n^{\log_b(a)} \approx n^{2.32}$. The first is polynomially smaller (with $\epsilon \approx 1.32$) so we apply Case 1 of the Master Theorem to get running time $\Theta(n^{2.32})$.

Algorithm B: $a = 9, b = 3$. We compare $f(n) = O(n^2)$ with $n^{\log_b(a)} = n^2$. They are asymptotically equal so we apply Case 2 of the Master Theorem to get running time $O(n^2 \log n)$.

For Algorithm C we use recursion tree: it is a binary tree of depth n, so the number of nodes in it is $\Theta(2^n)$. So with constant work per node, we have $\Theta(2^n)$ algorithm.

Algorithm B is asymptotically fastest among the options.

Problem 2: Understanding Huffman's Algorithm

(a) [12.5 points] Consider running Huffman's Algorithm on a set of three symbols $\{a, b, c\}$. Suppose frequency $f(a) < 1/3$. Show that no matter what $f(b), f(c)$ are, symbol a will not be assigned an encoding of length 1.
Hint: In order for a to have an encoding of length 1, which symbols must Huffman choose as the two lowest frequency symbols first? Arrive at a contradiction.

For reference, find the pseudocode for Huffman's algorithm below:

```
HUFFMAN(C, f):          // C = set of symbols, f = their frequencies
    If |C| = 2:
        Let T := both symbols are children of the root node
        Return T
    Let x, y be the two lowest frequency symbols in C
    Form C' and f' by removing x and y, and adding a new symbol z with f(z) = f(x) + f(y)
    T' = HUFFMAN(C', f')
    Form T from T' by replacing z with a node having children x and y
    Return T
```

Solution

In the first call of the Huffman algorithm, the two lowest frequency symbols are selected. Since $f(a) + f(b) + f(c) = 1$ and $f(a) < \frac{1}{3}$, it must be that $f(b) + f(c) \geq \frac{2}{3}$. Then $f(b)$ or $f(c)$ is greater than or equal to $\frac{1}{3}$. Therefore a must be one of the two lowest frequency symbols. Then a and the other lowest frequency symbol will be merged in the first step. This means the depth of a is two, and thus its encoding is length two.

(b) [12.5 points] Now consider running Huffman's Algorithm on larger sets of symbols. Prove that if all symbols have frequency less than 1/3, then Huffman's algorithm will not assign an encoding of length 1 to any symbol.
Hint: Use part (a).

Solution

Suppose not. Then there is symbol a with $f(a) < 1/3$ that ends up with an encoding of length 1. Consider the second to last recursive call of Huffman. It will be on 3 symbols, say $\{a, b, c\}$, where b, c are (possibly) merged symbols with $f(b)$ and $f(c)$ being the summed frequencies of some original symbols. In part (a) we already proved that no matter what $f(b)$ and $f(c)$ are, a cannot end up with a length-1 encoding.

Problem 3: Spanning Trees
[25 points]

You are given an undirected, connected graph G, and each edge is colored red or blue (there are no edge weights). G has n vertices and m edges. Give an algorithm with running time $O(m)$ to find a spanning tree with the minimum number of blue edges. Justify the running time.

Hint: You don't have enough time to run the regular Prim's or Kruskal's. Think about how to modify Prim's to take linear time in this case. For reference, find the pseudocode for Prim's algorithm below:

```
MST-PRIM(G, s)
 1  for each vertex v
 2       do u.d := ∞
 3          u.π :=NIL
 4  s.d := 0
 5  Q := all vertices
 6  while Q ≠ ∅
 7       do v := EXTRACT-MIN(Q)
 8          for each edge (v, u)
 9               do if u ∈ Q and w(v, u) < u.d
10                    then u.d := w(v, u)
11                         u.π := v
```

Solution

Prim's algorithm has time complexity $O(m \log n)$ because the heap uses $O(\log n)$ for update operations (i.e., lines 7 and 10 above). The idea here is that since we only have red and blue edges (rather than arbitrary weights) we can modify Prim's algorithm to perform these operations in constant time. Instead of a heap to store vertices that are not yet part of the growing MST, we use two (doubly-)linked-lists for this: one storing vertices adjacent to the growing MST by a red edge (R) and one storing all the other vertices not yet in the MST (B). We preferentially extract a vertex from R, and only if no vertices are in it, we extract from B. When relaxing on edge (v, u), we set u's parent to v if u is still in B. We then move u from B to R if the edge (v, u) is red. Note that if a vertex v has been relaxed by a red edge, then is relaxed again by a blue edge we want to keep the red edge and not the blue edge (and keep v in R).

All list operations needed are constant time. Note that we can check if $u \in B$ in constant time by setting an additional bit field or by checking if $u.\pi = \text{nil}$. It takes $O(n)$ time for preprocessing each vertex. We iterate through vertices to check all edges, taking $O(m)$ time since all list operations are in constant time. The overall time complexity is $O(m)$ since $m > n$.

Pseudocode for completeness:

```
BLUE-MST(G)
 1  B := ∅
 2  R := ∅
 3  for each vertex v
 4        do v.d := ∞
 5           v.π := nil
 6           APPEND(B, v)
 7  while R ≠ ∅ ∧ B ≠ ∅
 8        do if R ≠ ∅
 9              then v := POP-HEAD(R)
10              else  v := POP-HEAD(B)
11           for each edge (v, u)
12                 do if u ∈ B
13                       then u.π := v
14                            do if (v, u) = red
15                                  then REMOVE(B, u)
16                                       APPEND(R, u)
```

Problem 4: Longest paths

[25 points]

Consider a directed graph G with n vertices and m edges, and weights assigned to every edge. Assume all weights are positive. Suppose we are interested in the *longest* (highest weight) path in G. Note that if there is a cycle in G, then the length of the longest path is not well defined (since it can get arbitrarily large by going around the cycle arbitrarily many times). Here we assume that G has no cycles. Give an $O(m+n)$ algorithm to find the length of the longest path in G, and justify its running time.

Solution

LONGEST PATH($G = (V, E)$)

```
1  Topological sort G
2  Initialize all dist(v) to 0
3  for each u ∈ V in topological order
4        do for each edge (u, v) out of u
5            if dist(v) < dist(u) + w(u, v) then dist(v) := dist(u) + w(u, v)
6  return max_{v∈V}{dist(v)}
```

The big picture is that since we process all vertices in topological order, whatever the longest path is, we will process its vertices in their order on the path. The update procedure (line 5) updates $dist(v)$ to be the length of the longest path ending on v.

Doing a topological sort has a running time of $O(n+m)$. The total number of times the comparison in line 5 is done is $O(m)$. Looking for the maximum to be returned takes $O(n)$. Therefore the overall running time of the algorithm is $O(m+n)$.