CORE EXAMINATION
(Algorithms Part)
Department of Computer Science
New York University
September 26, 2008

This is the common examination for the M.S. program in CS. It covers three core computer science topics: Programming Languages, Operating Systems, and Algorithms. The exam has two parts. The Systems Part (PL and OS) lasts two and one half hours and covers the first two topics. The Algorithms Part (Algo) lasts one and one-half hours, and covers the last topic.

THE REST OF THESE INSTRUCTIONS CONCERN THE ALGORITHMS EXAM.

You will be assigned a seat in the examination room.

Use the proper booklet for each question. The three exam booklet for this Algorithms Part is marked with Question numbers ALG1, ALG2 and ALG3. DO NOT put your name on the exam booklet or answer sheet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope. You can keep the exam.

Good luck!

Algorithms

Question 1

Recall that a *max-heap* A with heapsize n is a special binary tree on nodes $\{1, 2, \ldots, n\}$. Each node i has an associated value, normally denoted by $\mathsf{key}[i]$. The max-heap property is that when $i = \mathsf{parent}[j]$, we must have $\mathsf{key}[i] \geq \mathsf{key}[j]$. The height of a node is the number of edges in the longest simple downward path from the node to a leaf.

a. [2 points] What is the maximum height as a function of n?

Solution

$\lfloor \lg n \rfloor$. (Comment: $\lg n$ is close enough).

---

b. [2 points] Asymptotically (but with the correct constant) how many nodes have height one (as a function of n)?

Solution

$\frac{n}{4}$. Half the nodes are leaves and a quarter are parents of leaves.

---

c. [2 points] Let S be the sum of the heights of all the nodes. Find (with argument) the asymptotics of S. Your answer should be of the form $S = \Theta(f(n))$ for some natural function $f(n)$.

Solution

$S = \Theta(n)$. There are $\frac{n}{2}$ with height zero, $\frac{n}{4}$ with height 1, $\frac{n}{8}$ with height 2, etc. This gives $\sim n(0(1/2) + 1(1/4) + 2(1/8) + \ldots)$, but the sum converges so the total is $\sim n$. (Comment: This is a tough one but part of the analysis in HEAPS. Some partial credit for $O(n \lg n)$.)

---

d. [2 points] Describe an algorithm for adding a new element, with key value x, to the heap. Analyze how long the algorithm will take in the worst case. How long will it take in the best case?

Solution

This is precisely `HEAP-INSERT`. Increment heapsize, set i to new heapsize, set $\mathsf{key}[i] = x$. Then while $i \neq 1$ and $\mathsf{key}[parent[i]] < \mathsf{key}[i]$ exchange $\mathsf{key}[i], \mathsf{key}[parent[i]]$, reset i to $parent[i]$. The maximal time is $\Theta(\lg n)$ if you go up to the root, the minimal time is $O(1)$ if the new value is smaller than its parent at the start.

---

e. [2 points] Let $u = \lfloor \sqrt{n} \rfloor$. Describe an algorithm for creating an array $B[1 \cdots u]$ consisting of the top u values in the heap, in descending order. How long (again your answer should be in the form $\Theta(f(n))$ for some natural function $f(n)$) does your algorithm take?

Solution

Apply `HEAP-EXTRACT-MAX` u times. Each time takes $\Theta(\lg n)$ so the total time is $\Theta(\sqrt{n} \lg n)$.

Question 2

In this question, a document is viewed as a sequence of words and a position refers to word position in that sequence. You are given N arrays of sorted integers representing the positions of N words in a single document. The following is an example with $N = 3$.

the: 7 14 21 35 51

magic: 10 16 31 44

A *text window* is a consecutive sequence of positions in a document, i.e. a sequence of positions $(i, i+1, i+2, \ldots, j)$ for some positive integers $i \leq j$. The size of the window is $j - i + 1$.

You are asked to find a text window of the smallest size in the document that contains each of the N words at least once.

Denote by k the largest number of occurrences of any of the N words in the text (i.e. k is the largest size of any of the N arrays).

a. [4 points] Assume that $N = 2$. Give an efficient algorithm to solve this problem. What is the time complexity of your algorithm? (*Hint: Merge the two arrays?*).

Solution

$O(k)$. Merge the two arrays in $O(k)$ time to obtain a single sorted array. Each array element holds a position as well as the word at that position. Now scan the array from left to right and whenever you find two consecutive array elements corresponding to two different words, take the difference in the positions as the size of a text-window that contains both the words. Maintain the minimum over all such text-windows.

---

b. [6 points] Give a general algorithm to solve this problem for $N \geq 2$. What is the time complexity of your algorithm?

Solution

$O(kN \log k)$. Merge the k arrays into a single sorted array. This takes $O(kN \log k)$ time by repeatedly merging pairs of arrays (Comment: This step is non-trivial. Partial credit for sub-optimal answers). Call the merged array $A[1, \ldots, kN]$ where each element holds a position and the corresponding word. Now the problem can be solved by one scan of the array A in time $O(kN)$ by maintaining and appropriately updating:

- Two pointers that indicate left and right endpoints of the current text-window.
- An array $\mathsf{count}[1, \ldots, N]$ that keeps a count of number of occurrences of each word in the current window.
- An integer variable W that keeps track of the number of words that appear at least once in the current text-window.

If $W < N$, then move the right pointer by one step and *enlarge* the window. If $W = N$, move the left pointer by one step and *shrink* the window. Other details are omitted, but students are expected to provide full details.

Question 3

Let $G = (V, E)$ be a directed *acyclic* graph where V is the vertex set and E is the set of edges. (Note: A directed graph is called acyclic if it does not contain any directed cycle).

a. [3 points] Prove that the graph must have a vertex t that has no outgoing edge.

Solution

Start with an arbitrary vertex, take an outgoing edge if possible, and repeat. This process must terminate since the graph has no cycle and that yields a vertex with no outgoing edge.

---

b. [3 points] Suppose $|V| = n$. A *topological ordering* of the acyclic graph is a labeling of its vertices by integers from 1 to n such that

- Any two distinct vertices receive distinct labels.

- Every (directed) edge goes from a vertex with a lower label to a vertex with a higher label.

Give a polynomial time algorithm to find a topological ordering of the graph. What is its running time? Don't worry about getting the optimal running time. Assume adjacency list representation of the graph.

Solution

For $i = n, n-1, \ldots, 1$, find a vertex with no outgoing edge, label it with i, delete that vertex, and repeat. This can clearly be implemented in polynomial time, say $O(n^2)$ time, since for each index i, we can find a vertex with no outgoing edge and then delete it in $O(n)$ time.

---

c. [4 points] Fix a node t that has no outgoing edge. For every node $v \in V$, let $P(v)$ be the number of distinct paths from v to t. Define $P(v) = 0$ if no such path exists and define $P(t) = 1$ for convenience. Give a polynomial time algorithm to compute $P(v)$ for every node v. What is its running time? Don't worry about getting the optimal running time.

Solution

Take the topological ordering of the graph. Initialize $P(t) = 1$ and $P(u) = 0 \ \forall \ u \neq t$. Initialize the *current vertex* to be the one immediately preceding t in the topological order. For the current vertex v, define $P(v)$ to be the sum over all $P(w)$ such that (v, w) is an edge. Move the current vertex to the one immediately preceding it in the topological order. This can clearly be implemented in polynomial time, say $O(n^2)$ time, since for each vertex, we can sum over all its outgoing neighbors in $O(n)$ time.
