CpSc 8400: Design and Analysis of Algorithms

Instructor: Dr. Brian Dean — Spring 2014
Webpage: `http://www.cs.clemson.edu/~bcdean/` — TTh 11:00-12:15
Handout 19: Final Exam Solutions — Vickery 100

1. BST (5 points). You are given a set of n numbers, each an integer in the range $1 \ldots 10n$. Design a fast algorithm for building a binary search tree of height $O(\log n)$ on these numbers.

First sort the numbers in $O(n)$ time using counting or radix sort. We then build a balanced BST from the sorted sequence $A[1 \ldots n]$ by taking the middle element $A[n/2]$ as the root, and then recursively building balanced BSTs for the left and right subtrees out of the sorted sequences $A[1 \ldots n/2-1]$ and $A[n/2+1 \ldots n]$. The running time of this processes satisfies $T(n) = 2T(n/2) + O(1)$, which solves to $O(n)$.

2. Carpool (5 points). You are given a directed graph G with n nodes and m edges, and you are told the cost $c_e \geq 0$ of traveling along each edge e. Dorian starts at node x and Ravi starts at node y. They would both like to travel to a common destination node z. If P_x represents edges along the $x \rightsquigarrow z$ path taken by Dorian and P_y the edges along the $y \rightsquigarrow z$ path taken by Ravi, the total cost of their travel is the cost of all the edges in $P_x \cup P_y$ (so it might be useful for them to arrange their paths so they overlap, allowing them to carpool and save money). Please describe a fast algorithm for computing paths P_x and P_y minimizing the cost of $P_x \cup P_y$.

Run Dijkstra's algorithm backward from z to compute the cost $c(i,z)$ from every node i to z. Then run Dijkstra forward from x and y to compute $c(x,i)$ and $c(y,i)$ for all nodes i, and select the node i (representing a meeting point along the two paths) minimizing the combined cost $c(x,i) + c(y,i) + c(i,z)$. Total running time is $O(m + n \log n)$.

3. Paranoid Max (5 points). Consider an array $A[1 \ldots n]$ of distinct elements that are randomly permuted (so each of the $n!$ orderings of these elements is equally likely). Suppose we scan sequentially through the array from position $i = 1$ up to position $i = n$ keeping a running maximum M. However, any time we encounter an element $A[i] > M$, we not only reset M to $A[i]$, but we also double-check all the elements in $A[1 \ldots i]$ (in $O(i)$ time) just to make absolutely certain that they are all no larger than M (and of course, this check will always succeed). What is the expected running time of this algorithm?

Let X_i denote the time we spend when visiting position i – either $O(i)$ time if $A[i]$ is the maximum of $A[1 \ldots i]$ (which happens with probability $1/i$), or $O(1)$ otherwise. Since $\mathbf{E}[X_i] = \frac{1}{i}O(i) + \left(1 - \frac{1}{i}\right)O(1) = O(1)$, linearity of expectation tells us that the total expected running time is $\mathbf{E}[X_1 + \ldots + X_n] = O(n)$.

4. k-Partition (5 points). You are given an array $A[1 \ldots n]$ of integers and also an integer k. We say that A has a k-partition if its elements can be divided into k non-overlapping contiguous subarrays each of which has the same sum (and the subarrays must collectively contain all the elements from A). Please give a fast algorithm for testing whether or not A has a k-partition.

Each block in the partition must sum to exactly $S = (A[1] + \ldots + A[n])/k$, which we compute

in advance in $O(n)$ time. Then scan forward through A maintaining a running sum, resetting the sum every time it reaches exactly S. If the sum ends on zero, A has a k-partition. Total running time: $O(n)$.

5. Security (5 points). The security log of the front door to a bank tells you the records of all people entering or exiting the building during the past day. These n records are of the form $(t_1, p_1, e_1) \ldots (t_n, p_n, e_n)$, where for each record i,

- t_i is an integer-valued time,
- p_i is the integer identifier for a person, and
- e_i is $+1$ if person p_i enters the bank at time t_i, or -1 if person p_i leaves the bank at time t_i.

All t_i's and p_i's are in the range $1 \ldots n^2$. The bank is initially empty at time $t = 0$. Please design a fast algorithm that determines if there are two times $t < t'$ (not necessarily integers) at which exactly the same people are in the bank, such that at least one entry or exit occurs between t and t'. For only partial credit, you can make the simplifying assumption that the p_i's or the t_i's (but not both) are at most $O(1)$.

We sort the records by time in $O(n)$ using radix sort and then simulate the process of people entering and exiting the bank over time, keeping a running hash h_t of the set of people inside at each interesting time t where an entry or exit occurs. By storing the h_t's in a universal hash table, we can easily detect in $O(1)$ expected time whenever we see two times $t < t'$ with $h_t = h_{t'}$ (where the hash has changed in the meantime). There are several ways to maintain our running hash, for example using a polynomial hash function. However, a particularly easy way is to generate a random integer $v(i)$ for each person i (this is generated and stored in a universal hash table associated with key i upon the first time we encounter person i). When we encounter a record (t_i, p_i, e_i), we update our running hash by adding $e_i v(p_i)$ to it. If the $v(i)$'s are chosen from a sufficiently large range, then unwanted hash collisions (leading to false positive detections) occur with extremely low probability. Total running time is $O(n)$ in expectation.

6. Inequalities (5 points). You are given a set of m strict inequality constraints over the variables $x_1 \ldots x_n$. For example, you might have $x_1 < x_2$, $x_2 < x_3$, and $x_3 < x_1$. As in this example, it might be impossible to assign values to the variables $x_1 \ldots x_n$ so that all m inequalities are satisfied. Our goal is to assign values to these variables so as to satisfy a maximum number of the inequalities. Please design a 2-approximation algorithm for this problem, which will always satisfy at least $OPT/2$ inequalities, where OPT denotes the maximum number that can be satisfied.

If we assign a random permutation of $\{1, 2, \ldots n\}$ to $x_1 \ldots x_n$, then each inequality is satisfied with probability $1/2$, so we satisfy $m/2 \geq OPT/2$ expected inequalities by linearity of expectation (note that $m \geq OPT$ trivially). Alternatively, consider assigning either $1, 2, \ldots, n$ or $n, n-1, \ldots, 1$ to $x_1 \ldots x_n$. Since each inequality must be satisfied in one of these two solutions, one of the two solutions must satisfy at least $m/2 \geq OPT/2$ inequalities.

7. Game (5 points). You are playing a game with your opponent in which you start with n numbers $A_1 \ldots A_n$ in a row. On your turn, you can select one of the two endpoints from this sequence and remove it. Then your opponent can select one of the two endpoints and remove it. The game proceeds in this fashion, until all the numbers have been removed; the winner is the player who removed the numbers with the highest total sum. Please design a fast algorithm that can tell if, moving first, you can guarantee winning.

We use dynamic programming. Let $M[i, j]$ denote the total value you can win, playing optimally, if you move first and only the numbers $A_i \ldots A_j$ remain. We now have $M[i, j] =$

$A_i + \ldots + A_j - \min(M[i+1, j], M[i, j-1])$**, which we can compute in** $O(1)$ **time by computing the range sum** $A_i + \ldots + A_j$ **in terms of a difference of two prefix sums. Solving all these subproblems takes** $O(n^2)$ **time, and if** $M[1, n] > (A_1 + \ldots + A_n)/2$**, we can guarantee winning.**

8. Frequent Numbers (5 points). You are given an array $A[1 \ldots n]$. Please design a fast *comparison-based* algorithm that will output every number occurring in at least 1% of all the entries in A. For example, if $n = 1000$, you should output any number occurring at least 10 times in A.

Note that we can't hash since this isn't comparison-based. In $O(n \log n)$ **time, we can sort (which groups identical elements together) and then scan through the sorted ordering counting the number of occurrences of each value. In** $O(n)$ **time, however, we can select out the order statistics at ranks** $n/100$**,** $2n/100$**,** $3n/100$**, etc. (this involves** $100 = O(1)$ **calls to our linear-time selection algorithm). The answer will be a subset of these 100 values, so we just check each one in** $O(n)$ **time to count occurrences. Our total running time is therefore just** $O(n)$**.**

9. Delete Largest Half (5 points). You would like to design a data structure supporting two operations: *insert* (which inserts a new element), and *delete-largest-half*, which deletes the largest $n/2$ elements from the structure. Please show how to build this data structure so that both operations run in $O(1)$ amortized time. For full credit, use a potential function in your analysis.

Store the elements in a simple linked list. To delete the largest half, select the median and filter out all elements larger than the median; let's say this takes n **units of time. Consider** $\phi = 2n$ **as our potential function, the amortized cost of an insertion is 3 (1 unit of actual cost, 2 units of increased potential), and the amortized cost of deleting the largest half is 0 (**n **units of actual cost, offset by a decrease in potential from** $2n$ **to** n**).**

10. Shortest Path (5 points). How can you quickly solve the single-source shortest path problem in a directed graph whose edge costs belong to $\{0, 1\}$?

Run BFS but when we examine each outgoing edge (i, j) **from a node** i**, we put** j **at the front of our queue if** (i, j) **has cost zero, and at the back of the queue otherwise (as would normally be done with BFS). The resulting algorithm therefore acts like a hybrid between BFS and DFS, and it still runs in** $O(m + n)$ **time.**