CS 161 W19: Recitation 3 Problems

January 2019

Exercise 0

In this problem, we prove that the average depth of a node in a randomly built binary search tree with n nodes is $O(\log n)$. A *randomly built binary search tree* with n nodes is one that arises from inserting the n keys in random order into an initially empty tree, where each of the $n!$ permutations of the input keys is equally likely.

Let $d(x, T)$ be the depth of node x in a binary tree T (The depth of the root is 0). Then, the average depth of a node in a binary tree T with n nodes is

$$\frac{1}{n} \sum_{x \in T} d(x, T).$$

(a) Let the *total path length* $P(T)$ of a binary tree T be defined as the sum of the depths of all nodes in T, so the average depth of a node in T with n nodes is equal to $\frac{1}{n}P(T)$. Show that $P(T) = P(T_L) + P(T_R) + n - 1$, where T_L and T_R are the left and right subtrees of T, respectively.

(b) Let $P(n)$ be the expected total path length of a randomly built binary search tree with n nodes. Show that $P(n) = \frac{1}{n}\sum_{i=0}^{n-1}(P(i) + P(n-i-1) + n - 1)$.

(c) Show that $P(n) = O(n \log n)$. You may cite a result previously proven in the context of other topics covered in class.

(d) Design a sorting algorithm based on randomly building a binary search tree. Show that its (expected) running time is $O(n \log n)$. Assume that a random permutation of n keys can be generated in time $O(n)$.

Exercise 1

We are given an unsorted array A with n numbers between 1 and M where M is a large but constant positive integer. We want to find if there exist two elements of the array that are within T of one another.

(a) Design a simple algorithm that solves this in $O(n^2)$.

(b) Design a simple algorithm that solves this in $O(n \log n)$.

(c) How could you solve this in $O(n)$? (Hint: modify bucket sort.)

Exercise 2

In this exercise, we'll explore the difference between average-case runtime (which shows up in CLRS) and expected runtime (which is almost always what we use in class). Recall that an algorithm $\textsc{Alg}$ has expected runtime at most $f(n)$ if for every input i of length n, $\mathbb{E}[\#\text{ operations to run } \textsc{Alg} \text{ on } i] \leq f(n)$. Whereas, the average-case runtime of an algorithm depends on the distribution of possible inputs to the algorithm. $\textsc{Alg}$ has average-case runtime at most $f(n)$ if $\mathbb{E}[\#\text{ operations to run } \textsc{Alg} \text{ on a random input}] \leq f(n)$.

(a) Argue that an algorithm with expected runtime $O(f(n))$ also has average-case runtime $O(f(n))$.

(b) Say we want to sort a list containing the distinct integers from 1 to n, where we expect each of the $n!$ permutations of $[1, \ldots, n]$ to occur with equal probability. Our friend has already written an algorithm Alg for this problem with an *average-case* runtime of $O(n)$. Use Alg to design a new algorithm, Alg2, with *expected* runtime $O(n)$.

(c) Although in part (b) we were able to turn an average-case runtime into an expected runtime, this won't always be possible. Let's consider a slightly different sorting problem. We want to sort a list of length n where each element is an integer between 1 and n^2, and we expect each of the $(n^2)^n$ possible lists to occur with equal probability (this is equivalent to saying each element of the array is independent and uniformly chosen from $[1, \ldots, n^2]$). We'll consider the following algorithm for this problem (a generalization of the BucketSort you saw in lecture):

Algorithm 1: BucketSort2(A)

```
n = len(A)
buckets = array of length n, initialized to hold n empty linked lists
for i ∈ [0, n − 1] do
    bucket_index = ⌈A[i]/n⌉ − 1
    add A[i] to buckets[bucket_index]
sorted = []
for i ∈ [0, n − 1] do
    sorted_bucket = InsertionSort(buckets[i])
    add the elements in sorted_bucket to sorted
return sorted
```

Essentially, we place the elements of the array A into n buckets where bucket i gets elements in the range $[i \times n, (i+1) \times n - 1]$, then sort each bucket and concatenate the results. Give intuition that suggests the average-case runtime of BucketSort2 is $O(n)$ (see CLRS 8.4 for details).

(d) Finally, give an example input of length n for which BucketSort2 will have runtime $\Theta(n^2)$. Why can we not just randomize this input into an "average-case" input like we did in part (b)?

Exercise 3

In class we saw the RadixSort algorithm, which sorts integers starting with the least-significant digit (in some base). This question is meant to explore the decision between starting with the least-significant digit (LSD) and most-significant digit (MSD).

(a) Review: what is the runtime of LSD radix sort? What is the space required for LSD radix sort?

(b) Design a version of radix sort which starts with the most-significant digit.

(c) What is the runtime of your algorithm? Compare it with the runtime of LSD radix sort.

(d) What is the space required for your algorithm? Can you do better? Compare it with the space required for LSD radix sort.

(e) Advanced (Take Home) - Can you come up with a radix sort that uses sub-linear additional memory (in-place radix sort)?