CS 161 W19: Recitation 3 Solutions

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 $O(n)$.

Solution 0

(a) Let $r(T)$ denote the root of tree T. Note the depth of node x in T is equal to the length of the path from $r(T)$ to x. Hence, $P(T) = \sum_{x \in T} d(x, T)$.

For each node x in T_L, the path from $r(T)$ to x consists of the edge $(r(T), r(T_L))$ and the path from $r(T_L)$ to x. The same reasoning applies for nodes x in T_R. Equivalently, we have

$$d(x, T) = \begin{cases} 0, & \text{if } x = r(T) \\ 1 + d(x, T_L), & \text{if } x \in T_L \\ 1 + d(x, T_R), & \text{if } x \in T_R \end{cases}$$

Then,

$$\begin{aligned}\sum_{x\in T} d(x,T) &= d(r(T),T) + \sum_{x\in T_L} d(x,T) + \sum_{x\in T_R} d(x,T)\\ &= 0 + \sum_{x\in T_L} [1 + d(x,T_L)] + \sum_{x\in T_R} [1 + d(x,T_R)]\\ &= |T_L| + |T_R| + \sum_{x\in T_L} d(x,T_L) + \sum_{x\in T_R} d(x,T_R)\\ &= n - 1 + P(T_L) + P(T_R).\end{aligned}$$

It follows that $P(T) = P(T_L) + P(T_R) + n - 1$.

(b) Let T be a randomly built binary search tree with n nodes. Without loss of generality, we assume the n keys are $\{1, \ldots, n\}$. (This is WLOG because our algorithm is solely comparison-based.)

By definition, $P(n) = \mathbb{E}_{\mathbb{T}}[P(T)]$. Then, $P(n) = \mathbb{E}_{\mathbb{T}}[P(T_L) + P(T_R) + n - 1] = n - 1 + \mathbb{E}_{\mathbb{T}}[P(T_L)] + \mathbb{E}_{\mathbb{T}}[P(T_R)]$, where T_L and T_R are the left and right subtrees of T, respectively. Note

$$\mathbb{E}_{\mathbb{T}}[P(T_L)] = \sum_{i=1}^{n} \mathbb{E}_{\mathbb{T}}[P(T_L)|r(T) = i] \cdot Pr(r(T) = i).$$

Since each element is equally likely to be the root of T, $\Pr(r(T) = i) = \frac{1}{n}$ for all i. Conditioned on the event that element i is the root, T_L is a randomly built binary search tree on the first $i - 1$ elements. To see this, assume we picked element i to be the root. From the point of view of the left subtree, elements $1, \ldots, i - 1$ are inserted into the subtree in a random order, since these elements are inserted into T in a random order and subsequently go into T_L in the same relative order. Hence, $\mathbb{E}_{\mathbb{T}}[P(T_L)|r(T) = i] = P(i - 1)$. Putting these together, we get

$$\mathbb{E}_{\mathbb{T}}[P(T_L)] = \sum_{i=1}^{n} \frac{1}{n} P(i - 1).$$

Similarly, we get $\mathbb{E}_{\mathbb{T}}[P(T_R)] = \sum_{i=1}^{n} \frac{1}{n} P(n - i)$. Then,

$$\begin{aligned}P(n) &= n - 1 + \mathbb{E}_{\mathbb{T}}[P(T_L)] + \mathbb{E}_{\mathbb{T}}[P(T_R)]\\ &= n - 1 + \frac{1}{n} \sum_{i=1}^{n} [P(i - 1) + P(n - i)]\\ &= n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} [P(i) + P(n - i - 1)],\end{aligned}$$

where we changed the indexing of the sumamtion in the last equality.

(c) This is the same recurrence that appears in the analysis of Quicksort.

(d) The algorithm is 1) construct a randomly built binary search tree T by inserting given elements in a random order; and 2) do the inorder traversal on T to get a sorted list. Note step 2 can be done in $O(n)$ time. We argue that step 1 takes $O(n \log n)$ time in expectation. We observe that given the final state of tree T, we can compute the amount of work spent to construct T. To insert a

node x at depth d, we traversed exactly the path from the root to the parent of x, at depth $d-1$, to insert it. Hence, we can upper bound the total work done to construct T by $O(P(T))$. From part (c), we know that $P(T) = O(n \log n)$ in expectation. It follows that Step 1 takes $O(n \log n)$ time in expectation. Overall, the algorithm runs in $O(n \log n)$ in expectation.

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.)

Solution 1

(a) Compare all pairs of numbers to see if any are within T of each other.

(b) Sort the array, then compare only adjacent elements to see if they are within T of each other.

(c) Because we know that the elements are from 1 to M, if they are integers, we can simply bucket sort the elements and check subsequent elements to see if they are T apart. If we cannot create M buckets due to memory constraints, or if the array elements are real numbers, we could split the items up into buckets of size T. If any bucket has 2 or more items, then those elements are within T of each other. Otherwise, each bucket holds at most 1 element (indeed, the elements are sorted) and we only need to check pairs of elements in adjacent buckets, and there are at most $n-1$ such pairs.

If creating M/T buckets would take up too much memory, you can create a hierarchy and only split up the non-empty cells. For example, a hierarchy of depth 2 could split the range of 1 to M into buckets of size T^2, then take only the non empty buckets and split those into size T.

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 ALG has expected runtime at most $f(n)$ if for every input i of length n, $\mathbb{E}[\#$ operations to run ALG on $i] \leq f(n)$. Whereas, the average-case runtime of an algorithm depends on the distribution of possible inputs to the algorithm. ALG has average-case runtime at most $f(n)$ if $\mathbb{E}[\#$ operations to run ALG 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)?

Solution 2

(a) Let $g(n) = O(f(n))$ be the expected runtime of Alg. Then the average-case runtime is

$$\begin{aligned}\mathbb{E}[\#\text{ operations to run Alg on a random input}] &= \sum_{\text{inputs } i} P(i)\mathbb{E}[\#\text{ operations to run Alg on } i] \\ &\leq \sum_{\text{inputs } i} P(i)g(n) = g(n).\end{aligned}$$

Thus the average-case runtime is at most the expected runtime, so it is also $O(f(n))$.

(b) Alg2 takes in a list L, randomly permutes it in time $O(n)$, then runs Alg on the permuted list.

(c) Initializing the buckets and placing the elements of the array into buckets can be done in linear time, as can the final concatenation step. The only potentially expensive operation is the InsertionSort run on each bucket. However, because each element of the list is independent and uniformly chosen from $[1, \ldots, n^2]$, on average each bucket will have only a small constant number of elements. Thus we intuitively expect InsertionSort to only take a small constant number of operations per bucket, so linear time overall. Thus we expect the algorithm to have runtime $O(n)$.

(d) Any permutation of the input $[1, 2, \ldots, n]$ will cause BucketSort2 to take $\Theta(n^2)$ operations, because all of the elements will end up in the same bucket and then get insertion-sorted.

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)?

Solution 3

(a) The runtime of LSD radix sort using digits of base r on n integers upper-bounded by M is $O((n+r)(\lfloor \log_r M \rfloor + 1))$. The space required for LSD radix sort is used by the r buckets (which can be emptied and reused at each iteration). Thus if we represent the buckets as linked lists, this uses $O(r+n)$ space.

(b) The following algorithm implements most-significant digit radix sort on array A in base r. When calling MSDRadixSort, level should initially be set to $\lfloor \log_r M \rfloor$.

Algorithm 2: MSDRadixSort(A, r, level)

```
n = len(A)
if level < 0 or n ≤ 1 then
    return A
buckets = array of length r, initialized to hold r empty linked lists
for i ∈ [0, n − 1] do
    bucket_index = ⌊A[i]/r^level⌋ mod r
    add A[i] to buckets[bucket_index]
sorted = []
for i ∈ [0, r − 1] do
    sorted_bucket = MSDRadixSort(buckets[i], r, level-1)
    add the elements in sorted_bucket to sorted
return sorted
```

(c) The runtime of this algorithm is $O(nr(\lfloor \log_r M \rfloor + 1))$. This holds because there are the same number of levels, and at each level, each item will have a constant amount of work (extracting the item's bucket number and placing the item in the bucket). However, we also have to initialize the r buckets at *each* sub-problem, and there can be up to n non-empty sub-problems at each level.

(d) At each level in the recursion we have to allocate r new buckets (as opposed to r buckets overall in LSD radix sort)—we can't reuse the higher-level buckets because they are still holding lists that need to be sorted. This can result in as much as $O(n + r(\lfloor \log_r M \rfloor + 1))$ space required.

(e) Think of the binary representation of numbers. First, sort by the most-significant bit. Keep track of the last element of the zeros group (starting at -1) and the first element of the ones group (starting at n). Work both groups in: if you see a number with a leading 1 on the left side, swap it with the position before the start of the 1's group. If you see a number leading with a zero on the right side, swap it with the element after the end of the zeros groups. For subsequent levels you can either 1) spawn recursive calls with the group boundaries you found 2) store the group boundaries and iterate through the groups 3) go through the array using comparisons on the leading bits to find where the group boundaries are. While option 3 saves space it increases computation time.