CS 161 Winter 2020 Section 4

January 30-31, 2020

1 More Sorting!

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.)

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

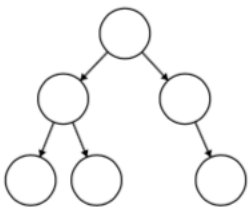
(b) Sort the array, then compare only adjacent elements to see if they are within T of each other.

(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.

2 Uniqueness of BST Structure

You are given a binary tree structure with n nodes and a set of n distinct keys (numbers). Prove or disprove: There is exactly one way to assign keys to the given tree structure such that the resulting tree is a valid binary search tree. **Example**: You are given the binary tree drawn below and the set of keys 1, 2, 3, 4, 5, 6. The question asks whether there is exactly one way to assign the keys to nodes such that the tree will be a binary search tree. (If you prove the statement, it should be for any input and not just this example.)



True. We prove by induction on n.
Inductive Hypothesis: Given a tree with n nodes, there is only one way to assign a set of n distinct values to the nodes such the resulting tree is a valid BST.
Base case: If $n = 0$, then the tree contains no nodes, and there is only one way to assign no value to the nonexistent node. (A silly base case, but it works and is necessary).
Inductive step: Let $k > 1$ be an integer. Assume the inductive hypothesis holds for up to n nodes, where $n < k$. Consider a tree with k nodes. Denote by $(a_1, ..., a_k)$ the set of keys in sorted order $(a_i < a_{i+1})$
Let r denote the root. Denote by ℓ the number of nodes in the left subtree of the root. Since the nodes in the left subtree must store a key that is less than the root and the nodes in the right subtree must have keys that are greater than the root, we conclude that the root must store the key $a_{\ell+1}$ In addition, the left subtree of the root must contain the keys $\{a_1, ..., a_\ell\}$ and the right subtree must contain the keys $\{a_{\ell+2}, ..., a_k\}$. Each of these subtrees is a binary tree with strictly less than k nodes, and by our induction hypothesis, we know that there is only one way to assign the values to the nodes in each subtree. We conclude that there is only one way to assign the k values to the nodes of the tree in a way that will result in a valid binary search tree.
Conclusion: We conclude that the inductive hypothesis holds for all n.

3 Approximate Sorting

(Note: This is a more difficult question from a previous offering's homework)

We say an array A of n distinct numbers is **k-approximately sorted** if the following property holds: for each $i = 1, 2, ..., n$, if $A[i]$ is the j-th smallest number in the array, then $|i - j| \leq k$. In other words, each number is at most k positions away from its actual sorted position. A k-approximate sorting algorithm takes an array of distinct numbers as input and produces an array that is k-approximately sorted.

(a) Show that given any n distinct numbers, the number of permutations of those numbers that are $\sqrt{n}$-approximately sorted is $O(n^{cn})$, for some $c < 1$. In other words, show that there are $O(n^{cn})$ possible permutations of the numbers such that each element lands within $\sqrt{n}$ distance of its sorted positions.

(b) Use part (a) to find a lower bound on the number of leaf nodes in the decision tree for any comparison-based $\sqrt{n}$-approximate sorting algorithm, and prove that any such $\sqrt{n}$-approximate sorting algorithm must have worst case complexity $\Omega(n \log n)$.

(c) Let $k > 1$ be an arbitrary constant. Can we construct an $\frac{n}{k}$-approximate sorting algorithm that does better than $O(nlogn)$? (Hint: the SELECT algorithm might help you here).

(a) **Solution.** Let $b_1, b_2, ..., b_n$ be n distinct numbers sorted in increasing order (think of this as where the n numbers are actually supposed to belong). If $a_1, a_2, ..., a_n$ is a $\sqrt{n}$-approximately sorted permutation of $b_1, ..., b_n$, then a_i equals b_j for some $i - \sqrt{n} \leq j \leq i + \sqrt{n}$, so it can take on at most $2\sqrt{n}+1$ different possible values. It's helpful to think of this as a $\sqrt{n}+1+\sqrt{n}$ size window of values that a_i could possibly correspond to. There can therefore be at most $(2\sqrt{n} + 1)^n$ approximately sorted permutations of the n numbers. Now, for all $n > 3^{10}$: $(2\sqrt{n}+1)^n \leq (3\sqrt{n})^n < (n^{0.1}\sqrt{n})^n = (n^{0.6})^n = n^{0.6n}$, and hence the number of approximately sorted permutations is $O(n^{0.6n})$. (Note: we picked 3^{10} a bit arbitrarily. For any constant $\epsilon > 0$ we can pick a constant n_0 depending on ϵ so that for all $n > n_0$, the number of approximately sorted permutations is $O(n^{(\frac{1}{2}+\epsilon)n})$).

(b) **Solution.** To start off, let's think of each leaf node in the decision tree as corresponding to a particular ordering of the input. To be more concrete, here are three examples example leaf nodes:

- "[third element from input, second element from input, tenth element, ...]"
- "[seventh element from input, first element from input, hundredth element, ...]"
- "[first element from input, second element from input, third element, ...]"

In our 'exact' sorting context (i.e. sorting an array pefectly, as discussed in lecture), we decided that the corresponding decision tree for comparison based models needs to have at least $n!$ leaves. This is because no two input orderings could take the same path down the decision tree to arrive at the same leaf node (e.g. "[seventh element from input, first element from input, hundredth element, ...]"), so we we need as many leaf nodes as there are input orderings, which is $n!$. In other words, to arrive at any particular leaf node in the decision tree, only one permutation of the input elements would have worked—for example, if our leaf node was "[seventh element from input, first element from input, hundredth element, ...]", then the only input element that would have led to that leaf would be the input whose seventh element is the smallest in the array, whose first element is the second smallest, whose hundredth element is the third smallest, etc.

In this $\sqrt{n}$-approximate sorting context, things are a bit different. In this case, we know that no more than $O(n^{cn})$ input orderings could take the same path down the decision tree to arrive at the same leaf node. To see this more clearly, consider the example leaf node of "[seventh element from input, first element from input, hundredth element, ...]". There are several input orderings that could arrive at that leave node as the answer (and at most $O(n^{cn})$ of them). The input orderings that would work are inputs where the seventh element is not exactly the smallest but is actually within a $\sqrt{n}$ window from being the smallest, where the first element is not exactly the second smallest but is actually within a $\sqrt{n}$ window from being the second smallest, where the hundredth element is not exactly the third smallest but is actually within a $\sqrt{n}$ window from being the third smallest.

Thus, since we're considering a *lower bound* on the number of leaf nodes needed, we can use this information to determine the fewest number of leaf nodes that we need to 'get by'. To 'get by', we just need to make sure that there are enough leaf nodes (i.e. enough output orderings) such that there's a valid leaf node (i.e. a valid output ordering) for any input. Since we've observed that we could have an 'overlap' of at most $O(n^{cn})$ (i.e. at most $O(n^{cn})$ input orderings could all experience the same path down the decision tree to hit the smae leaf node), then the least amount of leaf nodes we would need to serve as destinations for all our $n!$ input orderings is $\Omega(\frac{n!}{n^{cn}})$. To compare this with our 'exact' sorting context from lecture, we had that at most 1 input ordering can reach a particular leaf node, so the least amount of leaf nodes we would need to serve as destinations for all the $n!$ input orderings was $\Omega(\frac{n!}{1}$.

Now that we've decided that the lower bound on the number of leaves in the decision tree is $\Omega(\frac{n!}{n^{cn}})$, we can reason that the lower bound on the height of our decision tree (which corresponds to the lower bound on the number of steps, i.e. runtime, that any comparison based $\sqrt{n}$-approximate sorting algorithm would need to take) is $\Omega(\log(\frac{n!}{n^{cn}}))$. Using Stirling's approximation, we can see that this gives us a worst case runtime of $\Omega(n \log n)$, which is what we needed to show.

(c) **Solution.** We can use the selection algorithm to find the $\frac{n}{k}$-th smallest number in $O(n)$ time. Then the group of $\frac{n}{k}$ numbers smaller than or equal to it correspond to the $\frac{n}{k}$ smallest numbers, and this group can be found in linear time. We can the repeat the process to find the next $\frac{n}{k}$ smallest numbers, and so on so forth, until we have divided the list into k groups of $\frac{n}{k}$ numbers. We can then output these groups in increasing order, sorting within each group arbitrarily. As long as the groups are correct, no number will be off from their true sorted positions by more than

$\frac{n}{k}$ places, so our list will be $\frac{n}{k}$-approximately sorted. Our algorithm takes $O(kn) = O(n)$ time, since k is constant!