The plan for this lecture is to finish the analysis of Randomized Quicksort, to show that the sorting algorithms we have seen so far are in some sense optimal, and to give a very different sorting algorithm that can sort in linear time when the numbers are not too big.

1 QuickSort Review

Here, we finish the analysis of QuickSort. For more details about QuickSort, including the notation used in these notes, see the notes for Lecture 5. Recall the claims about the runtime of QuickSort proved in the previous lecture.

Claim 1. *The runtime of QuickSort is $O(n + X)$ where X is the total number of comparisons made by the algorithm.*

Claim 2. *Let $z_1 < z_2 < ... < z_n$ refer to numbers in the input array A, but in* sorted order. *Let p_{ij} be the probability that z_i and z_j are compared at any point of the algorithm. Then, the expected number of comparisons made by QuickSort is*

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} p_{ij}.$$

Last lecture, we used these theorems to argue that if $p_{ij} = \frac{2}{j-i+1}$, then the runtime of RandQuickSort is $O(n \log n)$. Thus, to finish our proof of the run time of RandQuickSort, we need to analyze the probabilities p_{ij} and to show that they are indeed $p_{ij} = \frac{2}{j-i+1}$.

Claim 3. *For any i, j with $i < j$: $p_{ij} = \frac{2}{j-i+1}$.*

Proof. We first note that in RandQuickSort, the only time that two elements are compared is when one of them is selected as a pivot. Consider the set $Z = \{z_i, z_{i+1}, \ldots, z_j\}$, that is, the elements between z_i and z_j in the sorted order. Let RandQuickSort(A, p, r) be the *last call* of RandQuickSort for which all elements of Z are in $A(p : r)$. Thus, in this call, the pivot must be some $z_k \in Z$ (as otherwise all of Z will be in $A <$ or $A >$ and RandQuickSort(A, p, r) wouldn't be the last call for which $Z \subset A(p : r)$).

Let z_k be the pivot. We will consider two cases - either z_k is one of the endpoints, z_i or z_j, or z_k is some other element in Z.

Case 1: Either $z_k = z_i$ or $z_k = z_j$

In this case, z_i and z_j are clearly compared within this call.

Case 2: z_k is neither z_i nor z_j

In this case, z_k is one of $\{z_{i+1}, \ldots, z_{j-1}\}$. Then z_k is compared to z_i and z_j. Since $z_i < z_k$ and $z_k < z_j$, z_i will be in $A_<$ and z_j will be in $A_>$, so z_i and z_j will never be compared.

Thus, $p_{ij} = \Pr(z_i \text{ and } z_j \text{ are compared}) = \Pr(\text{pivot is } z_i \text{ or } z_j | \text{ pivot is in } Z)$. Because the pivot was picked uniformly at random, $\Pr(\text{pivot is } z_i \text{ or } z_j | \text{ pivot is in } Z) = \frac{2}{|Z|} = \frac{2}{j-i+1}$, completing this proof.

□

2 Lower Bounds for Comparison Sorting Algorithms

We call a sorting algorithm a *comparison-based* sorting algorithm if the sorted order is determined solely through pairwise comparisons of elements, and not the actual values that are being compared. The two most well-known comparison sorting algorithms are MergeSort and QuickSort, but there are many others which also share this property.

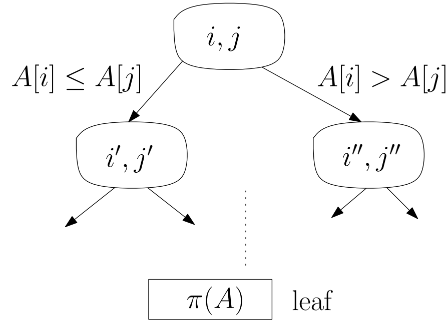

Figure 1: The general framework for a decision tree. In this case, i is compared to j at the root, which could lead to i' being compared to j' (if $A[i] \leq A[j]$) or i'' being compared to j'' (if $A[i] > A[j]$). The leaves of the tree are permutations of the entries of A.

In this section, we will show that comparison-based sorting algorithms all have inherent limitations. We'll show that any deterministic comparison sorting algorithm must perform $\Omega(n \log n)$ comparisons to sort n elements. That is, for any sorting algorithm that doesn't look at the actual values being compared, the algorithm's running time is lower bounded by $\Omega(n \log n)$.

To show this lower bound, we will completely disregard any operations besides comparisons. Then we can model all comparison sorting algorithms with a class of binary trees called *decision trees*.

Each node of the *decision tree* corresponding to a comparison-based algorithm represents a comparison that the algorithm makes between two elements of the array. The comparison stored at the root is the first comparison performed by the algorithm. Each node of the tree contains two (ordered) indices i, j of the array, meaning that $A(i)$ and $A(j)$ are two be compared. There are two possible cases - either $A(i) \leq A(j)$ or $A(i) > A(j)$. The left child of the node corresponds to the next comparison that the algorithm would make if $A(i) \leq A(j)$, and the right child corresponds to the next comparison whenever $A(i) > A(j)$. This way any internal node of the decision tree has exactly two children. At each leaf of the decision tree (the termination of the algorithm), there is a permutation of A representing the final sorted order based on the root to leaf path.

Each run of the comparison-based sorting algorithm on an input A corresponds to a path from the root to some leaf in the decision tree. The number of total comparisons made on input A is the length of the corresponding root to leaf path. Thus, the worst-case number of total comparisons for a comparison sorting algorithm will be the height of its decision tree. We use this to prove a lower-bound on the number of comparisons made by an arbitrary comparison sorting algorithm.

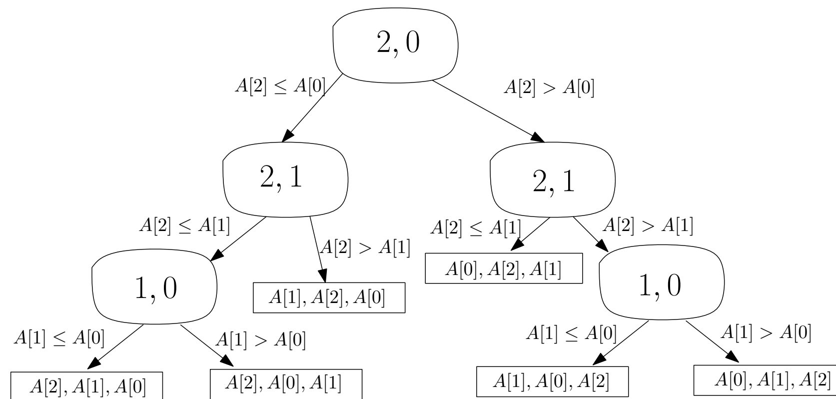

Figure 2: A decision tree for Quicksort running on an array of length 3.

Theorem 2.1. *Any comparison sorting algorithm must perform $\Omega(n \log n)$ comparisons to sort n elements.*

Proof. As stated earlier, the worst-case number of total comparisons for a comparison sorting algorithm will be the height h of its corresponding decision tree. Thus, we will prove this theorem by showing that $h \geq \Omega(n \log n)$.

We first note that every permutation of the n indices of the input must appear at one of the leaves of the decision tree; otherwise, there would exist some possible input that would not be correctly sorted. There are $n!$ permutations of these indices, so the number of leaves ℓ is at least $n!$. Additionally, because the tree is binary, a decision tree of height h will have no more than 2^h leaves. Thus, $n! \leq \ell \leq 2^h$ which implies that $h \geq \log(n!)$.

By Stirling's approximation, $n! \geq \Omega((\frac{n}{e})^n)$. Therefore, $\log(n!) \geq \Omega(\log n^n - \log e^n) = \Omega(n \log(n))$. Thus, we have $h \geq \Omega(n \log n)$, completing the proof.

□

3 CountingSort

We now look at an integer sorting algorithm called CountingSort, which requires some additional assumptions but has a much improved runtime. The main point is that this algorithm actually uses the values stored the input array to sort instead of purely comparisons. Because of this, for many inputs it can run in linear time.

3.1 Algorithm Description and Intuition

Input: Integer array A of length n where all elements are in $\{0, ...k\}$
Output: Sorted integer array B of length n

Given some element x in A, we now consider at which indices it will appear in the sorted array B. Let C_i be the number of elements that are less than or equal to i. Then, because the number of elements less than x is equivalent to the number of elements less than or equal to $x - 1$, we know that the first x in B will be in position C_{x-1}. The last occurrence of x will be in position $C_x - 1$. Thus, to sort A, it suffices to compute C_x for every $x \in \{0, ..., k\}$. The pseudocode for CountingSort is below.

Algorithm 1: CountingSort(A, k) `// |A| = n`

If $|A| = 1$, return A;
$B \leftarrow$ array of length n;
$C \leftarrow$ array of length k initialized to the array of all 0s;
for j *from* 0 *to* $n - 1$ **do**
    $C[A[j]] \leftarrow C[A[j]] + 1$;   `Find number of occurrences of each element`
for j *from* 1 *to* k **do**
    $C[j] \leftarrow C[j] + C[j - 1]$;   `Now,` $C[j] = C_j$ `as defined earlier`
for $j = n - 1 \rightarrow 0$ **do**
    $x \leftarrow A[j]$;
    $B[C[x] - 1] \leftarrow x$;
    $C[x] \leftarrow C[x] - 1$;
return B;

The correctness proof of the algorithm is as follows. The first for loop computes the number of occurrences of each x in A, storing the number in $C[x]$. Then the second for loop computes for each x, the number of elements $\leq x$. It does this inductively. As the base case, C_0 is exactly the number of 0s in A, so it equals $C[0]$. Suppose that $C[i-1]$ contains the number of elements $\leq i - 1$. Then the number of elements $\leq i$ is $C[i-1]+$ the number of occurrences of i. Thus, one can compute C_i by summing $C[i-1]$ and $C[i]$. Finally, the last loop places each element x in its correct position, via the argument we made earlier.

Claim 4. *CountingSort sorts n integer elements between 0 and k in $O(n+k)$ time.*

Proof. This runtime follows easily from the above pseudocode. The first for loop from iterates over n elements and performs constant-time operations on each, so it can be completed in time $O(n)$. Through similar reasoning, the second for loop can be completed in time $O(k)$, and the third for loop can be completed in time $O(n)$. Thus, the total runtime will be $O(n+k)$.

□

One nice additional property of CountingSort is that the occurences of some element x will appear in the same order in the sorted array B as they did in the original array A. This property is known as *stability* and is critical to CountingSort's application to RadixSort, another non-comparison sorting algorithm. It may also be useful if you are sorting objects by some key, but want to maintain the order of the objects who share a key.