CS 161, Lecture 5 Quicksort
Scribe: Sam Keller (2015) **Date:** April 13, 2016

1 Introduction

Today we'll study another sorting algorithm: Quicksort. You may wonder why we want to study a new sorting algorithm. We have already studied MergeSort, which we showed to perform significantly better than the trivial $O(n^2)$ algorithm. While MergeSort achieves an $O(n \log n)$ worst-case asymptotic bound, in practice, there are a number of implementation details about MergeSort that make it tricky to achieve high performance. Quicksort is an alternative algorithm, which is simpler to implement in practice. Quicksort will also use a divide and conquer strategy but will use randomization to improve the performance of the algortihm in expectation. Java, Unix, and C stdlib all have implementations of quicksort as one of their built-in sorting routines.

2 Quicksort Overview

As in all sorting algorithms, we start with an array A of n numbers; again we assume without loss of generality that the numbers are distinct. The idea for quicksort is the following:

(1) Pick some element $x \leftarrow A[i]$. We call x the pivot.

(2) Split the rest of A into $A_<$ (the elements less than x) and $A_>$ (the elements greater than x).

(3) Rearrange A into $[A_<, x, A_>]$.

(4) Recurse on $A_<, A_>$.

Steps 1-3 define a "partition" function on A. The partition function of Quicksort can vary depending on how the pivot is chosen and also in the implementation. Quicksort is often used in practice because we can implement this step in linear time, and with very small constant factors. In addition, the rearrangement (Step 3) can be done in-place, rather than making several copies of the array (as is done in merge sort). In this lecture we will not use an in-place implementation for simplicity, but pseudocode for it can be found in CLRS.

3 Speculation on the Runtime

The performance of Quicksort depends on which element is chosen as the pivot. Assume that we choose the k^{th} smallest element; then $|A_<| = k - 1, |A_>| = n - k$.

This allows us to write a recurrence; let $T(n)$ be the runtime of quicksort on an n-element array. We know the partition step takes $O(n)$ time; therefore the recurrence is

$$T(n) \leq cn + T(k-1) + T(n-k).$$

One way to define the partition function is to pick the *median* as the pivot. In the above recurrence this would mean that $k = \lceil \frac{n}{2} \rceil$. We showed in the SELECT algorithm in Lecture 3 that we can find the median element in linear time. Therefore the recurrence becomes $T(n) \leq cn + 2T(\frac{n}{2})$. This is exactly the same recurrence as merge sort, which means that this algorithm is guaranteed to run in $O(n \log n)$ time.

Unfortunately, the median selection algorithm is not practical; while it runs in linear time, it has much larger constant factors than we would like. To improve it, we will explore some alternative methods for choosing a pivot.

4 Choosing the Last Element as Pivot

As an alternative to the median, we'll try deterministically choosing the last element as the pivot. The partition function takes the array A and two indices in the array p, r, meaning that it will only be partitioning $A[p : r]$. If $p \geq r$, the size of $A[p : r]$ is 1 so the function does nothing. This is the base case. Otherwise, it performs the split of A using $A[r]$ and returns the index where $A[r]$ ends up being placed at. The pseudocode is below.

Algorithm 1: Partition(A, p, r)

```
$x \leftarrow A[r]$;
$A_< \leftarrow \{A[i] : A[i] < x\}$;
$A_> \leftarrow \{A[i] : A[i] > x\}$;
$a \leftarrow |A_<|$;
$A[p, p + a - 1] \leftarrow A_<$;
$A[p + a] \leftarrow x$;
$A[p + a + 1, r] \leftarrow A_>$;
return $p + a$;
```

Algorithm 2: Quicksort(A, p, r)

```
if $p < r$ then
    $q = $ Partition$(A, p, r)$;
    Quicksort$(A, p, q - 1)$;
    Quicksort$(A, q + 1, r)$;
```

To run quicksort on an array A of length n, call Quicksort$(A, 0, n - 1)$. This method sorts the array as it goes, and so does not return anything.

As an example, let's consider running this version of quicksort on a *sorted* array with $n = 5$.

1. At the first level, 5 is selected as the pivot. $A_< = [1, 2, 3, 4]$, and $A_>$ is empty.
2. In the the recursive call to $A_<$, 4 is selected at the pivot. $A_< = [1, 2, 3]$, and $A_>$ is empty.
3. In the the recursive call to $A_<$, 3 is selected at the pivot. $A_< = [1, 2]$, and $A_>$ is empty.
4. In the the recursive call to $A_<$, 2 is selected at the pivot. $A_< = [1]$, and $A_>$ is empty.
5. $A_<$ has only one element, so it is already sorted.

We see that each call to Partition splits the current array into a single piece of size one less while comparing every element to the pivot and hence spending $\Omega(n)$ time. This gives a new recurrence:

$$T(n) = cn + T(n - 1).$$

Using the recursion tree approach, we see that this costs $c(n - i)$ at each level i. Therefore the total runtime is

$$T(n) = \sum_{i=0}^{n-1} c(n - i) = c \cdot \frac{n(n + 1)}{2} = \Omega(n^2).$$

This is asymptotically much worse than the runtime we had by selecting the median! It also shows that there are inputs (even silly ones such as a sorted array) for which the algorithm performs *at least* a quadratic number of operations.

This leads us to ask two questions:

- Are there any inputs which take longer than $\Theta(n^2)$?
- How does this algorithm perform on most inputs?

5 Analysis

We first present a claim that will make the overall runtime easier to analyze.

Claim 1. *Quicksort runs in $O(X + n)$ time, where X is the number of total comparisons done over the entire algorithm.*

Proof of Claim 1. Each call Partition(A, p, r) takes $O(r - p)$ time, since we do constant work on each element in the range. Furthermore, there are exactly $r - p$ comparisons done. Therefore the total time spent in a call to Partition is $O(X)$, where X is the number of total comparisons done over the entire algorithm. Additionally, Quicksort will perform a constant amount of work in the base case, when the array to sort only has one element. Quicksort will be called on single-element arrays once for each element in the original array, which gives a total of $O(X + n)$ time. □

This immediately answers are first question from the previous section: since each call to Partition takes at most n comparisons, and there are at most n such calls, the runtime is always $O(n^2)$.

The sorted case performs badly because the last element does not evenly partition the remaining elements. Let's suppose instead that the pivot is always inside the range $[\frac{n}{10}, \frac{9n}{10}]$ of the sorted array (in other words, it is not too close to either extreme). We can now update our recurrence again:

$$T(n) \leq cn + \max_{\frac{n}{10} \leq k \leq \frac{9n}{10}} T(k) + T(n-k).$$

We can use a recursion tree to evaluate this recurrence. Note that the sum of the sizes of the arrays at each level of the tree is n (because n splits into $k, n-k$, etc.), and so $O(n)$ work is done at each level. Furthermore, since $k, n - k \leq \frac{9}{10}$, we know that the largest array at each level is at most $\frac{9}{10}$ of the largest array at the level above it. This means that there are at most $\log_{\frac{10}{9}} n = O(\log n)$ levels, which means that this algorithm takes $O(n \log n)$ time.

The above case happens for *most* inputs. Very few inputs actually have the last element be picked from only $2n/10$ choices (the $n/10$ smallest and largest elements). So on an average input, the algorithm performs well!

The question is, can we make the algorithm perform well on all inputs? It turns out that the answer is yes, if one uses a randomized algorithm and instead of worst case runtime one considers runtime in expectation.

There are two ways to convert the Quicksort algorithm that performs well on average inputs to a version that performs well on all inputs, in expectation:

- Randomly shuffle the array before beginning quicksort, and then run Quicksort on the shuffled array (choosing the last element for each pivot as before).
- Randomly select a pivot in each call to Partition.

These two strategies turn out to be equivalent; we will study the second.

6 Proof of Expected Runtime

To modify the pseudocode for Partition and Quicksort, we must make the following change:

- Change Partition to RandPartition: Before each call to Partition, select a random index i and swap the element at position i with the last element; then partition as before.
- Change Quicksort to RandQuicksort: Make recursive calls to RandPartition and RandQuicksort.

Our goal is to show that RandQuicksort has **expected runtime** $O(n \log n)$. This means that, for every input array A of size n, the average running time over many executions of RandQuicksort$(A, 0, n-1)$ will be $\leq c \cdot n \log n$ for some constant c.

To do this, we define a random variable Y to represent the runtime of Quicksort. Furthermore, let X be a random variable for the number of comparisons required.

By Claim 1, $E[Y] = O(E[X+n]) = O(n + E[X])$.

Let the elements of the array be $z_1 < z_2 < \cdots < z_n$. Define an indicator variable X_{ij}, such that $X_{ij} = 1$ if z_i, z_j are compared at some point in a run of quicksort and 0 otherwise. Let p_{ij} be the probability that these elements are compared. By linearity of expectation,

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

We will show rigorously next lecture that $p_{ij} = \frac{2}{j-i+1}$. The intuition for this result is that i, j are compared if and only if either i, j is chosen as a pivot before any element $z_{i+1}, z_{i+2}, \cdots, z_{j-1}$; since pivots are chosen randomly, this happens with probability $\frac{2}{j-i+1}$.

Assuming the claim, we have

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2\sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{1}{k+1} \leq 2n \sum_{k=1}^{n} \frac{1}{k} \leq 2n(1 + \ln n) \leq O(n \log n).$$

Therefore the expected runtime is $O(n \log n)$, as desired.