Adapted From Virginia Williams' lecture notes.

COMP 285 (NC A&T, Spr '22) Lecture 9

Randomized Algorithms and QuickSort

Today we will study another sorting algorithm: `Quicksort`, which was invented in 1959 by Tony Hoare. 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 algortithm in expectation. Java, Unix, and C stdlib all have implementations of `Quicksort` as one of their built-in sorting routines.

1 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[1]. Quicksort is very similar to the `QuickSort` algorithm we studied last lecture. The description of Quicksort is the following:

Algorithm 1: QuickSort(A)

if $n = 1$ **then**
  return A
end if
$p \leftarrow$ `RandomPivot`(A)
$A_< \leftarrow \{A[i] \mid A[i] < p\}$
$A_> \leftarrow \{A[i] \mid A[i] > p\}$
$A'_< \leftarrow$ `QuickSort`$(A_<)$
$A'_> \leftarrow$ `QuickSort`$(A_>)$
return $[A'_<, p, A'_>]$

The above steps define a "partition" function on A. The partition function of `Quicksort` can vary depending on how the pivot is chosen and also on 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 can actually be done in-place, rather than making several copies of the array (as is done in `MergeSort`). In these notes we will not describe the details of an in-place implementation, but the pseudocode can be found in CLRS.

[1] What does it mean to say that we can assume something without loss of generality? Why can we make this assumption without loss of generality in this proof?

2 Speculation on the Runtime

The performance of `Quicksort` depends on which element is chosen as the pivot. Assume that we choose the kth 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

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

For the worst pivot choice (the maximum or minimum element in the array[2]), the runtime satisfies the recurrence $T(n) = T(n-1) + O(n)$; hence $T(n) = O(n^2)$.

One way that seems optimal 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 previous lecture that the algorithm `Select` can find the median element in linear time. Therefore the recurrence becomes $T(n) \leq cn + 2T(n^2)$. This is exactly the same recurrence as `MergeSort`, 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.

We leave the proof of correctness of `Quicksort` as an exercise to the reader (hint: use induction).

3 Random Pivot Selection

Our discussion so far suggests two approaches to pivot selection: (1) We could pick an arbitrary element as a pivot. This is super simple and takes O(1) time (good), but the worst case running time of the resulting algorithm would be $O(n^2)$ (bad). (2) We could pick the median as the pivot. This takes $O(n)$ time and is somewhat complicated (bad), but gives a worst case running time of $O(n \log n)$ (good). Can we get the best of both worlds?

One method of "defending" against making a bad pivot choice is to choose a random element as the pivot. We observe that it is unlikely that the random element will be either the median (best-case) or the maximum or minimum (worst-case). Note that we have a uniform distribution over the n order statistics of the array (this is a fancy way of saying: for every $1 \leq i \leq n$ we pick the i-th highest element with the same probability $\frac{1}{n}$). Let's pause for a moment. This is the first time in this course that we have encountered an algorithm that uses randomness in its execution. Such an algorithm (that "flips coins" and takes actions based on the outcome of these coin flips) is called a randomized algorithm. How do we analyze a randomized algorithm?

[2] 2Why are these the worst pivot choices?