CS 344: Design and Analysis of Computer Algorithms **Rutgers: Fall 2019**

Lecture 6

September 23, 2019

Instructor: Sepehr Assadi

1 Quick Sort

We proved the correctness of quick sort in the previous lecture notes but deferred the runtime analysis to this lecture.

Recall that in the quick sort algorithm introduced before, we can take any element as the pivot. Partitioning the array using this pivot takes $O(n)$ time. Also, assuming we pick the element with correct position q as pivot, we then need to recursively sort the arrays $A[1 : q-1]$ and $A[q+1 : n]$. Hence, if we use $T(n)$ to denote the worst-case runtime of quick sort, we have,

$$T(n) \leq \max_{q \in \{1,\ldots,n\}} \{T(q-1) + T(n-q)\} + O(n).$$

We can verify that the maximum value of the right hand side equation happens when $q = 1$ or $q = n$ (we will do so shortly). Using that, we have,

$$T(n) \leq T(n-1) + O(n) \implies T(n) = O(n^2). \qquad \text{(this is the same recurrence as insertion sort!)}$$

So first, why did we say $q = 1$ or $q = n$ which results in $O(n^2)$ runtime is the worst example? In other words, can quick sort run even slower? The answer is *no* because after each run of the partition algorithm that takes $O(n)$ time, we remove one element from consideration in recursive calls (the pivot element). Hence, for each of the n elements, we may spend at most $O(n)$ time processing it (in partition) and so we can only spend $O(n^2)$ time. Hence, quick sort runs in $O(n^2)$ time.

Now the second question is that whether $T(n) = O(n^2)$ is *too pessimistic* (and not accurate) and we could proved a better upper bound? The answer to this question is also *no*: consider an array that is sorted in non-increasing order, i.e., exactly the reverse order of what we want – for this array, by picking the last element as pivot, we have to recursively solve the problem for $n-1$ elements (on a subarray that is still sorted in the non-decreasing order); hence, for such an array we have to spend $\Omega(n)$ time for running the partition algorithm for $\Omega(n)$ first choices of pivot, making the total runtime $\Omega(n^2)$.

The above implies that the runtime of quick sort is $\Theta(n^2)$ in the worst case. Wait what? Something must be wrong because quick sort is supposed to be "quick", namely, $O(n \log n)$ time; so what just happened? The catch is that quick sort can be really slow if we "end up unlucky" and pick a "bad" pivot in every step (roughly speaking, a bad pivot is a one which partitions the array in a very lopsided way, the extreme example being $n-1$ elements in one side and none in the other – this corresponds to picking either the *maximum* or *minimum* of the array as the pivot). What would happen instead if we are really "lucky" and get the "best" pivot in every step (the best pivot is when we partition the array into two roughly the same size subarrays – this corresponds to picking the *median* of the array as the pivot): *for this particular example*, we can write $T(n) \leq 2T(n/2) + O(n)$; this is the same as the recurrence for merge sort and thus we get $T(n) = O(n \log n)$ which would be good.

The above part tells us we need to have some "luck" on our side, but how can we do that? By picking the pivot *randomly* instead of arbitrarily! We thus need to consider a randomized quick sort. Before that, we do a very quick recap of the probabilistic background we need.

2 Probabilistic Background

We review basic probabilistic background using a simple running example: Consider the probabilistic process of rolling two dice and observing the result.

- *Probability space*: The set of all possible outcomes of the probabilistic process. Here, all the 36 combinations of answers, i.e., $(1,1),(1,2),(1,3),\cdots,(6,5),(6,6)$.
- *Event*: Any subset of the probability space. Here, one example of an event would be 'sum of the two dice is equal to 7'; another example is 'both dice rolled an even number'.
- *Probability distribution*: an assignment of $\mathbf{Pr}(e) \in [0,1]$ to every element e of the probability space such that $\sum_e \mathbf{Pr}(e) = 1$. Here, every one of the 36 elements of have the same probability $1/36$, i.e., $\mathbf{Pr}((1,1)) = 1/36, \cdots, \mathbf{Pr}((6,6)) = 1/36$.
- *Probability*: for any event E, probability of E is $\mathbf{Pr}(E) = \sum_{e\in E} \mathbf{Pr}(e)$, i.e., the sum of the probabilities assigned by the probability distribution to the elements of this event. Here, for example,

$$\mathbf{Pr}(\text{sum of the two dice is 7}) = \sum_{e\in\{(1,6),(6,1),(2,5),(5,2),(3,4),(4,3)\}} \mathbf{Pr}(e) = 6\cdot\frac{1}{36} = \frac{1}{6};$$

$$\mathbf{Pr}(\text{both dice roll even}) = \sum_{e\in\{(2,2),(2,4),(2,6),(4,2),(4,4),(4,6),(6,2),(6,4),(6,6)\}} \mathbf{Pr}(e) = 9\cdot\frac{1}{36} = \frac{1}{4}.$$

- *Random variable*: Any function from the elements of the probability space to integers or reals. Here, an example of a random variable X is the sum of the two dice, e.g., $X((1,3)) = 4$ and $X((2,5)) = 7$. Another example is a random variable Y that assigns one to the events that both dice roll even and is zero otherwise, e.g. $Y((2,2)) = 1$ and $Y((2,3)) = 0$ (this type of random variable that assigns 1 to a particular event and is zero otherwise is called an *indicator* random variable for that event).
- *Expected value*: The expected value of a random variable X, denoted by $\mathbf{E}[X]$, is the average of its value *weighted* according to the probability distribution, i.e., $\mathbf{E}[X] = \sum_e \mathbf{Pr}(e)\cdot X(e)$. Here, for instance, for the random variables X and Y defined above, we have,

$$\mathbf{E}[X] = \sum_e \Pr(e)\cdot X(e) = \sum_{i\in\{2,\ldots,12\}} \mathbf{Pr}(X=i)\cdot i = 7;$$

$$\mathbf{E}[Y] = \sum_e \Pr(e)\cdot Y(e) = \sum_e \Pr(Y(e)=1) = \frac{1}{4}.$$

  A very important property of expected value is its *linearity* (often called *linearity of expectation*): for any two random variables X, Y, $\mathbf{E}[X+Y] = \mathbf{E}[X] + \mathbf{E}[Y]$.

The above is an extremely short refresher of the probabilistic background. This cannot by any means replace a proper introduction to this amazing concept. You are strongly encouraged to take a look back at the materials from your previous courses on probability, or Chapter 5 of the CLRS book for more details.

3 Randomized Quick Sort

We now analyze a randomized variant of quick sort. The algorithm is literally the same as before with the exception that we now pick the pivot *uniformly at random* from the elements of the array, instead of picking an arbitrary one. We repeat the algorithm here for completeness.

Randomized Quick Sort Algorithm: The input is an array $A[1:n]$.

1. If $n = 0$ or $n = 1$ return the current array.
2. Pick p *uniformly at random* from $\{1, 2, \ldots, n\}$.
3. Use the partition algorithm to partition A with pivot p. Let q be the correct position of $A[p]$ in the sorted array computed by the partition algorithm.
4. Recursively run randomized quick sort on $A[1 : q-1]$ and $A[q+1 : n]$.

The correctness of this algorithm is exactly as before: we proved that *any* choice of p would work in the algorithm so a uniformly at random chosen one should work as well. The interesting part is to analyze the runtime of the algorithm. But first, we should re-examine what a runtime of a randomized algorithm should be? If we take the worst-case runtime to mean *literally* the worst-case over all choices of the input *and* the randomness of the algorithm, then randomization cannot in anyway improve the runtime of algorithms (do you see why?). The problem with this approach is that we are again ignoring the randomness of the algorithm and fix it to be worst case also.

It turns out the correct notion of runtime for randomized algorithms is *expected worst case* running time: we still assume that the input can be arbitrary (or worst case), but now take the *expected value* of the runtime of the algorithm over all choices of random bits that it uses. This way, randomization can indeed help with the running time as we are going to see shortly.

We will analyze the expected worst case runtime of the randomized quick sort algorithm in the next lecture.