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

Lecture 7

September 26, 2019

Instructor: Sepehr Assadi

1 Runtime of Randomized Quick Sort

Let us be precise by what we mean by expected worst case runtime. Suppose A is an array of length n. We define the *random variable* $S(A)$ to denote the runtime of the randomized quick sort algorithm on this fixed array A. Note that $S(A)$ is a random variable as its value depends on the random choices of the algorithm. The *expected* runtime of the randomized quick sort on input A of size n is then $\mathbf{E}[S(A)]$. Now for any integer $n \geq 1$, we define $T(n)$ to denote the *maximum* value of $\mathbf{E}[S(A)]$ for all choices array A of length n. In other words,

$$T(n) = \max_{A:|A|=n} \mathbf{E}[S(A)] \, .$$

In English, this is precisely the *expected worst case runtime* of the randomized quick sort, the expectation referring to taking $\mathbf{E}[S(A)]$ and the worst case referring to taking A to be the worst array.

Runtime Analysis: We now analyze the runtime of randomized quick sort. Let A be *any* arbitrary array of length n. Suppose we choose an element with the correct position q to be the pivot: then the algorithm recurses on $A[1 : q-1]$ and $A[q+1 : n]$. Note that q is also a random variable in this step. We thus have,

$$\begin{aligned}
\mathbf{E}[S(A)] &= \sum_{i=1}^{n} \Pr(q=i) \cdot \mathbf{E}[S(A) \mid q=i] && \text{(by definition of expected value)} \\
&\leq \sum_{i=1}^{n} \Pr(q=i) \cdot \mathbf{E}[S(A[1:i-1]) + S(A[i+1:n]) + c \cdot n \mid q=i] \, ,
\end{aligned}$$

as we recursively solve the problem on the two subproblems and since it takes $O(n) \leq c \cdot n$ for some constant $c > 0$ time to run the partition algorithm. Now note that the randomness in the runtime of $S(A[1 : i-1])$ and $S(A[i+1 : n])$ is *independent* of the event $q = i$ itself and so we can 'drop' this conditioning:

$$\begin{aligned}
\mathbf{E}[S(A)] &\leq \sum_{i=1}^{n} \Pr(q=i) \cdot \mathbf{E}[S(A[1:i-1]) + S(A[i+1:n]) + c \cdot n \mid q=i] \\
&= \sum_{i=1}^{n} \Pr(q=i) \cdot \mathbf{E}[S(A[1:i-1]) + S(A[i+1:n]) + c \cdot n] \\
&= \sum_{i=1}^{n} \frac{1}{n} \cdot \mathbf{E}[S(A[1:i-1]) + S(A[i+1:n]) + c \cdot n] \\
&\qquad \text{(we choose the pivot uniformly at random and so each value of } q \text{ has the same probability of } \tfrac{1}{n}) \\
&= \left(\frac{1}{n} \cdot \sum_{i=1}^{n} \mathbf{E}[S(A[1:i-1])] + \mathbf{E}[S(A[i+1:n])]\right) + c \cdot n && \text{(by linearity of expectation)}
\end{aligned}$$

By taking A to be the worst case array, $T(n) = \mathbf{E}[S(A)]$. Moreover, $\mathbf{E}[S(A[1:i-1])] \leq T(i-1)$ and $\mathbf{E}[S(A[i+1:n])] \leq T(n-i)$ by definition of the $T(\cdot)$ function (as we are taking maximum). This implies,

$$T(n) \leq \left(\frac{1}{n}\sum_{i=1}^{n} T(i-1) + T(n-i)\right) + c \cdot n.$$

We are almost done now: we obtained a recurrence for $T(n)$ similar to all the previous times that we found a recurrence. Even though this recurrence does not look like any of the previous recurrences we have seen, our goal is now clear: solve this recurrence and compute a closed form solution for $T(n)$.

It turns out the correct answer for this recurrence is $T(n) = O(n \log n)$. There are multiple ways to prove this but none of them are that simple (and somewhat beyond the level of your homeworks or exams). In the following, we show how to use induction to prove $T(n) \leq 2c \cdot n \log n$ for interested readers.

Solving the recurrence (optional): We prove by induction that $T(n) \leq 2c \cdot n \log n$. The base case is true since $T(\Theta(1)) = \Theta(1)$ and so we can take c to be a large enough constant to make this equation true. We now prove the induction step. Suppose this is true for all integers $< n$ and we prove it for n. We have,

$$\begin{aligned}
T(n) &\leq \left(\frac{1}{n}\sum_{i=1}^{n} T(i-1) + T(n-i)\right) + c \cdot n \\
&= \left(\frac{2}{n}\sum_{i=1}^{n} T(i-1)\right) + c \cdot n && \text{(since } \textstyle\sum_{i=1}^{n} T(i-1) = \sum_{i=1}^{n} T(n-i)\text{)} \\
&\leq \left(\frac{2}{n}\sum_{i=1}^{n} 2c \cdot (i-1) \cdot \log{(i-1)}\right) + c \cdot n \\
&\qquad \text{(since } T(i-1) \leq 2c \cdot (i-1) \cdot \log{(i-1)} \text{ by induction hypothesis as } i-1 < n\text{)} \\
&= \left(\frac{2}{n} \cdot 2c \cdot \sum_{i=0}^{n-1} (i) \cdot \log{(i)}\right) + c \cdot n && \text{(by change of variable)} \\
&\leq \left(\frac{2}{n} \cdot 2c \cdot (\frac{n^2 \log n}{2} - \frac{n^2}{4}) + c \cdot n\right) && (\textstyle\sum_{i=0}^{n-1} i \log i \leq (\frac{n^2 \log n}{2} - \frac{n^2}{4})) \\
&= (2c \cdot n \log n - c \cdot n) + c \cdot n \\
&= 2c \cdot n \log n.
\end{aligned}$$

This proves the induction step and hence $T(n) \leq 2c \cdot n \log n$ for all n. This implies that $T(n) = O(n \log n)$.

2 Sorting in Linear Time

So far, we have seen and carefully analyzed two efficient sorting algorithms: merge sort and quick sort. Despite being efficient, these algorithms still require $\Omega(n \log n)$ time. Can we have even faster algorithms for sorting? For instance, can we sort n numbers in only $O(n)$ time – such a solution would be *ideal* because it means that the runtime of our algorithm would be a constant factor more than the time needed to even read the input once.

This question however does not have a simple (or single) answer. All the algorithms we discussed so far are *comparison-based* algorithms: they gain order information between the elements of the array by comparing them to each other (this includes also the non-efficient algorithms such as selection sort and insertion sort that we discussed, as well as heap sort algorithm that we did not cover but is also efficient and run in $O(n \log n)$ time). One can prove that *any* comparison-based sorting algorithm, no matter how clever it is or how it works, still needs $\Omega(n \log n)$ time in the worst case. We will not cover this proof in this course because it is somewhat orthogonal to the topic of algorithm design that we are focusing on; however, a short proof is available in Chapter 8.1 of the CLRS book in case you are interested in seeing how it is possible to prove that *no* algorithm can solve a problem faster than some bound.

The above paragraph seems to suggest that the answer to the question of having faster algorithms for sorting is then 'no'. After all, how else can we sort n numbers without comparing them with each other? We are going to see one simple example of such algorithms: the *counting sort* algorithm that does not involve any comparison and still can sort the input quite efficiently <u>in certain scenarios</u>.

Counting Sort

Unlike the previous sorting algorithms that did not make any assumption about the input array A, counting sort is specifically designed for sorting arrays of *positive integers* – moreover, it is going to be efficient only when the values in the array are not "too large" (we will get back to this point later).

Counting Sort Algorithm: Given an input array $A[1:n]$ with values in $\{1, 2, \ldots, M\}$.

1. Create an array $C[1:M]$ and initialize it to be all 0.
2. For $i = 1$ to n: increase $C[A[i]]$ by one.
3. Let $p = 0$ and for $j = 1$ to M:
    - While $C[j] > 0$: decrease $C[j]$ by one; let $A[p] = j$, and increase p by one.

Example: Consider sorting the array $[5, 2, 3, 2, 4, 1, 6, 7, 4]$ using counting sort (here $n = 9$ and $M = 7$):

- The array C is updated as follows (underline shows the pointer i in Line (2) of the algorithm):

$$\begin{array}{ll}
A = [5,2,3,2,4,1,6,7,4] & C = [0,0,0,0,0,0,0] \\
A = [\underline{5},2,3,2,4,1,6,7,4] & C = [0,0,0,0,1,0,0] \\
A = [5,\underline{2},3,2,4,1,6,7,4] & C = [0,1,0,0,1,0,0] \\
A = [5,2,\underline{3},2,4,1,6,7,4] & C = [0,1,1,0,1,0,0] \\
A = [5,2,3,\underline{2},4,1,6,7,4] & C = [0,2,1,0,1,0,0] \\
A = [5,2,3,2,\underline{4},1,6,7,4] & C = [0,2,1,1,1,0,0] \\
A = [5,2,3,2,4,\underline{1},6,7,4] & C = [1,2,1,1,1,0,0] \\
A = [5,2,3,2,4,1,\underline{6},7,4] & C = [1,2,1,1,1,1,0] \\
A = [5,2,3,2,4,1,6,\underline{7},4] & C = [1,2,1,1,1,1,1] \\
A = [5,2,3,2,4,1,6,7,\underline{4}] & C = [1,2,1,2,1,1,1].
\end{array}$$

- The array A is then updated based on C (overline shows the pointer j in Line (3) of the algorithm and underline shows pointer p):

$$\begin{array}{ll}
C = [1,2,1,2,1,1,1] & A = [5,2,3,2,4,1,6,7,4] \\
C = [\overline{1},2,1,2,1,1,1] & A = [\underline{1},2,3,2,4,1,6,7,4] \\
C = [0,\overline{2},1,2,1,1,1] & A = [1,\underline{2},3,2,4,1,6,7,4] \\
C = [0,\overline{1},1,2,1,1,1] & A = [1,2,\underline{2},2,4,1,6,7,4] \\
C = [0,0,\overline{1},2,1,1,1] & A = [1,2,2,\underline{3},4,1,6,7,4] \\
C = [0,0,0,\overline{2},1,1,1] & A = [1,2,2,3,\underline{4},1,6,7,4] \\
C = [0,0,0,\overline{1},1,1,1] & A = [1,2,2,3,4,\underline{4},6,7,4] \\
C = [0,0,0,0,\overline{1},1,1] & A = [1,2,2,3,4,4,\underline{5},7,4] \\
C = [0,0,0,0,0,\overline{1},1] & A = [1,2,2,3,4,4,5,\underline{6},4] \\
C = [0,0,0,0,0,0,\overline{1}] & A = [1,2,2,3,4,4,5,6,\underline{7}].
\end{array}$$

Proof of Correctness: We first observe that after Line (2) of the algorithm, for every $1 \leq j \leq M$, $C[j]$ is equal to the number of times number j appears in the array A (this is a very basic observation and we do not need to prove it really).

We now consider Line (3): for any iteration j of the for-loop in this line, define p_j as the value of pointer p after this line. The proof is by induction over index j. Our induction hypothesis is that for every $1 \leq j \leq M$, after iteration j of the for-loop, all the numbers in the *original* array A that were $\leq j$ are now placed in a *sorted* order in the *new* array $A[1 : p_j - 1]$.

- *Base case:* For $j = 1$, the while-loop places $C[1]$ many copies of number of 1 in the array $A[1 : p_1]$ (which may be zero copies). Since $C[1]$ was equal to the number of times number 1 appears in the old array A, this means that after iteration 1, we copied #of 1's copies of 1 in $A[1 : p_1]$, proving the induction step.
- *Induction step:* Suppose this is true for all integers up to j and we prove it for $j + 1$. By induction hypothesis, we already placed all copies of $\{1, \ldots, j\}$ in the array $A[1 : p_j - 1]$. In iteration $j + 1$, the while-loop places $C[j + 1]$ many copies of number $j + 1$ in the array $A[p_j : p_{j+1} - 1]$ – this makes the array $A[1 : p_{j+1} - 1]$ contains all elements in the array A that are $\leq j + 1$ in a sorted order (since every element in $A[p_j : p_{j+1} - 1]$ is equal to $j + 1$ and is thus larger than all previous elements that were sorted by induction hypothesis for j). This proves induction step.

We can now apply our induction hypothesis to $j = M$ and have that the array $A[1 : p_M - 1]$ contains all elements in the original array A that are in the range $\{1, \ldots, M\}$ in a sorted order. Since all elements of A were in this range, this means that the new array is now a sorted version of A, proving the correctness.

Runtime Analysis: The first line of the algorithm takes $O(M)$ time to initialize the array C. The second line takes $O(n)$ time to iterate over all elements. The third and fourth lines together take $O(n + M)$ time since each iteration of the for-loop increases j by 1 and j can only increase M time, and each iteration of the while-loop increases p and p can only increase n times *in total*. Hence, the runtime is $O(n + M)$.

Final thoughts about counting sort: As we saw, counting sort does not make *any* comparison at all and is thus able to bypass the $\Omega(n \log n)$ lower bound for comparison-based sorting. In particular, when $M = O(n)$, counting sort runs in $O(n)$ time only, namely, *linear* time. Hence, we can achieve our ideal goal of sorting n numbers in linear time when the numbers are positive integers in the range $\{1, \ldots, O(n)\}$ using the counting sort algorithm.

3 Concluding Remarks on Searching and Sorting

This lecture concludes our study of searching and sorting algorithms. We covered the following topics related to searching and sorting:

- Binary search for finding an element in $O(\log n)$ time in a *sorted* array of length n.
- Selection sort and insertion sort for sorting an array of size n in $O(n^2)$ time – recall that these two algorithms were very simple but not that efficient.
- Merge sort for sorting an array of size n in $O(n \log n)$ time.
- Quick sort that still may require $\Theta(n^2)$ time to sort an array of length n in the worst case.
- A randomized variant of quick sort that can sort any array in expected worst case runtime of $O(n \log n)$.
- Counting sort as a non comparison-based sorting algorithm for sorting an array of size n with integer entries in the range $\{1, \ldots, M\}$ in $O(n + M)$ time – in particular, when $M = O(n)$, this algorithm runs in $O(n)$ time which is faster than all comparison-based sorting methods (including merge sort and quick sort).

What we did *not* cover on this front?

- Other examples of comparison-based sorting algorithms that run in $O(n \log n)$ time, for instance, the heap sort algorithm (if you are interested, you can look it up in Chapter 6 of the CLRS book);
- Other methods of non comparison-based sorting algorithms, for instance, the radix sort algorithm (if you are interested, you can look it up in Chapter 8.3 of the CLRS book);
- Computing other order statistics (beside maximum and minimum), for instance, for finding the median in $O(n)$ time (if you are interested, you can look it up in Chapter 9 of the CLRS book);
- An alternative for searching using hashing instead of sorting, including hash tables and hash functions (if you are interested, you can look up basics of hashing in Chapter 11 of the CLRS book);
- And definitely many more topics ...