CS 344: Design and Analysis of Computer Algorithms **Rutgers: Spring 2022**

Lecture 7

February 8, 2022

Instructor: Sepehr Assadi

1 Probabilistic Background

We are going to define randomized algorithms in this lecture and talk about randomized quick sort. Before that however, a quick refresher on probabilistic background is in order. Let us use 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 Appendix "Discrete Probability" of your textbook for more details.

2 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[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.

[1] In this course, we only work with randomized algorithms that use randomization to speedup their running times but regardless of which random bits they use, they will always output a right solution. These algorithms are called *Las Vegas* randomized algorithms. There are other types of randomized algorithms that use randomization to obtain their correctness, meaning that for every input, they are correct with probability 99% but may output a wrong answer with the remaining 1% probability. These algorithms are called *Monte Carlo* randomized algorithms. We will not discuss the latter algorithms in this course (one reason is that proving correctness of those algorithms requires much more background on probability theory that is beyond the scope of this course).

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:

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

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 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,

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

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

This proves the induction step and hence $T(n) \leq 2c \cdot n \log n$ for all n. This implies that $T(n) = O(n \log n)$. So we proved that expected worst-case running time of the randomized quick sort algorithm is $O(n \log n)$.

3 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, insertion sort, or "Silly Sort" from Homework 1). 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.

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 in certain scenarios.

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):

$$A = [5,2,3,2,4,1,6,7,4] \qquad C = [0,0,0,0,0,0,0]$$
$$A = [\underline{5},2,3,2,4,1,6,7,4] \qquad C = [0,0,0,0,1,0,0]$$
$$A = [5,\underline{2},3,2,4,1,6,7,4] \qquad C = [0,1,0,0,1,0,0]$$
$$A = [5,2,\underline{3},2,4,1,6,7,4] \qquad C = [0,1,1,0,1,0,0]$$
$$A = [5,2,3,\underline{2},4,1,6,7,4] \qquad C = [0,2,1,0,1,0,0]$$
$$A = [5,2,3,2,\underline{4},1,6,7,4] \qquad C = [0,2,1,1,1,0,0]$$
$$A = [5,2,3,2,4,\underline{1},6,7,4] \qquad C = [1,2,1,1,1,0,0]$$
$$A = [5,2,3,2,4,1,\underline{6},7,4] \qquad C = [1,2,1,1,1,1,0]$$
$$A = [5,2,3,2,4,1,6,\underline{7},4] \qquad C = [1,2,1,1,1,1,1]$$
$$A = [5,2,3,2,4,1,6,7,\underline{4}] \qquad C = [1,2,1,2,1,1,1].$$

- 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):

$$C = [1,2,1,2,1,1,1] \qquad A = [5,2,3,2,4,1,6,7,4]$$
$$C = [\overline{1},2,1,2,1,1,1] \qquad A = [\underline{1},2,3,2,4,1,6,7,4]$$
$$C = [0,\overline{2},1,2,1,1,1] \qquad A = [1,\underline{2},3,2,4,1,6,7,4]$$
$$C = [0,\overline{1},1,2,1,1,1] \qquad A = [1,2,\underline{2},2,4,1,6,7,4]$$
$$C = [0,0,\overline{1},2,1,1,1] \qquad A = [1,2,2,\underline{3},4,1,6,7,4]$$
$$C = [0,0,0,\overline{2},1,1,1] \qquad A = [1,2,2,3,\underline{4},1,6,7,4]$$
$$C = [0,0,0,\overline{1},1,1,1] \qquad A = [1,2,2,3,4,\underline{4},6,7,4]$$
$$C = [0,0,0,0,\overline{1},1,1] \qquad A = [1,2,2,3,4,4,\underline{5},7,4]$$
$$C = [0,0,0,0,0,\overline{1},1] \qquad A = [1,2,2,3,4,4,5,\underline{6},4]$$
$$C = [0,0,0,0,0,0,\overline{1}] \qquad A = [1,2,2,3,4,4,5,6,\underline{7}].$$

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.