CS3230 Design & Analysis of Algorithms **February 3, 2022**

Lecture 4: Average-case Analysis & Randomized Algorithms

Lecturer: Warut Suksompong

The goal of this lecture is to introduce you to the role of randomness in computing. We will see two distinct ways in which the analysis of algorithms can be probabilistic: (i) average-case analysis, where the input is drawn from a distribution, and (ii) randomized algorithms, where the algorithm itself randomizes its execution.

4.1 Average-Case Analysis

The algorithm we use to illustrate average-case analysis is QUICKSORT. Assume for simplicity that the input array A consists of distinct elements.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

Here, PARTITION rearranges the array $A[p\,..\,r]$ around a *pivot*. In this section, we select the pivot deterministically to be the first element of the original array A. PARTITION returns an index q and rearranges A so that the pivot is at position q, all the elements smaller than the pivot lie in $A[p\,..\,q-1]$, and all the elements greater than the pivot lie in $A[q+1\,..\,r]$. The following is an in-place[2] implementation.

```
PARTITION(A, p, r)
1  // Choose the first element as pivot
2  x = A[p]
3  i = r + 1
4  for j = r downto p + 1
5      if A[j] > x
6          i = i − 1
7          Swap A[i] and A[j]
8  Swap A[i − 1] and A[p]
9  return i − 1
```

4.1.1 Correctness

The correctness of PARTITION follows from the following loop invariant.

Claim 4.1.1. *At the beginning of any iteration of the loop of lines 4–7 in* PARTITION*, the following hold true:*

[1] See also Chapters 5 and 7 of CLRS.

[2] *In-place* means that the temporary storage required by the algorithm is bounded by a constant. For example, in PARTITION, each swap of two elements requires only one number to be temporarily stored. This is in contrast to MERGE-SORT where temporary storage of size $\Omega(n)$ is needed (to temporarily store the output of each recursive invocation of MERGE-SORT).

(i) All elements in $A[i\,..\,r]$ are greater than x

(ii) All elements in $A[j+1\,..\,i-1]$ are less than x

(iii) $A[p]$ equals x

Proof. (iii) is clear, as $A[p]$ is not modified by the loop. To prove (i) and (ii), we use induction. The base case, where $j = r$ (and $i = r+1$), is trivially true. Now, suppose (i) and (ii) hold for $j = J+1$ and we want to show that it holds for $j = J$. Let A' be the array and $i = I'$ at the start of iteration $J+1$, and A the array and $i = I$ at the start of iteration J. There are two cases.

- $A'[J+1] < x$. In this case, $I = I'$, and $A = A'$; so, (i) is trivially maintained. (ii) also holds as by induction, $A[J+2\,..\,r]$ are already known to be less than x, and this case assumes $A[J+1] < x$.
- $A'[J+1] > x$. In this case, $I = I'-1$, $A[I] = A'[J+1]$ and $A[J+1] = A'[I]$ (while at other locations, A and A' are identical). Then, (i) holds because $A[I] = A'[J+1] > x$ here, while $A[I+1\,..\,r] = A'[I'\,..\,r] > x$ by induction. (ii) holds because the inductive hypothesis implies $A[J+1] = A'[I'-1] < x$, and also, $A[J+2\,..\,I-1] = A'[J+2\,..\,I'-2]$ are less than x.

□

At termination, Claim 4.1.1 shows that $A[i\,..\,r]$ are greater than x while $A[p+1\,..\,i-1]$ are less than x. Hence, swapping $A[i-1]$ and $A[p]$ ensures that the array is correctly partitioned with the pivot x occurring at position $i-1$.

The correctness of QUICKSORT is now immediate by strong induction. For the base case $p \le r$ where the array is empty or of size one, the algorithm does nothing which is correct. Otherwise, the above shows PARTITION places the pivot at the correct position, while strong induction ensures that QUICKSORT correctly sorts the left and right sub-arrays which are of size strictly smaller than $n = r - p + 1$.

4.1.2 Worst-case running time

Let $T(n)$ be the worst-case running time of QUICKSORT on input arrays of length n. If the input A can be arbitrary, it is possible that A is already in sorted order. In this case, at any every recursive call, one sub-array is empty while the other sub-array is of length one smaller than the original. Hence for such arrays:

$$T(n) = T(n-1) + \Theta(n); \qquad T(1) = \Theta(1).$$

Here, $\Theta(n)$ includes the cost of PARTITION as well as the handling of the empty left sub-array. Applying the recursion-tree approach, we see that there are n levels in the recursion tree. Adding up the costs for each level, we get that $T(n)$ is upper bounded by $\sum_{i=1}^{n} ci < cn^2$ for some constant c and also lower bounded by $c'n^2$ for some other constant c', and hence $T(n) = \Theta(n^2)$.

Informally, the reason for the quadratic scaling is because the recursion tree is very unbalanced. The MERGE-SORT type recurrence $T(n) = 2T(n/2) + \Theta(n)$ occurs when both sides of the recurrence are of equal size, or even roughly equal size.

4.1.3 Average-case running time

We have seen that QUICKSORT has $\Theta(n^2)$ running time on arrays which are already sorted. The same argument also shows $\Theta(n^2)$ bound for arrays which are reverse-sorted. But maybe these are some rare, pathological cases, and for a 'generic' input, we would have asymptotically smaller runtime?

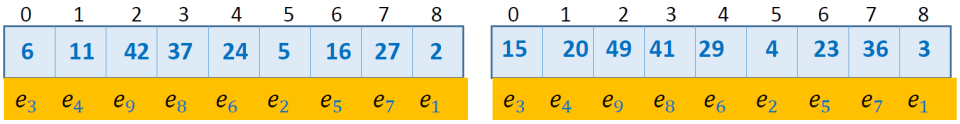
To address this question formally, we turn away from the idea of worst-case analysis and consider *average-case analysis*. For an algorithm Alg, suppose the inputs are drawn from a set U. For $n \ge 1$, let $\mathcal{D}_n$ be a probability distribution on U restricted to inputs of size n. Then, $A(n)$, the average-case running time of Alg with respect to $\mathcal{D}_n$ is defined as:

$$A(n) = \mathop{\mathbb{E}}_{x \sim \mathcal{D}_n} [\text{Runtime of Alg on } x]$$

(For a review on probability distributions and expectations, consult the supplementary material prepared by the TAs or Appendix C of CLRS.)

The tricky aspect about average-case analysis is the choice of the distribution $\mathcal{D}_n$. Ideally, we want it to reflect the distribution of inputs in real-life. But typically, this is something that we don't really know! If we really have no information about the relative importance of inputs, and if the set of inputs of size n is finite, then a common[3] choice is to let $\mathcal{D}_n$ be uniform over the inputs of size n.

Now, let's return to QUICKSORT, and consider an average-case analysis of its runtime. We immediately hit an obstacle: the set of arrays of length n is infinite, and so, we cannot define a uniform distribution on them. But a little thought reveals that because QUICKSORT is purely a comparison-based algorithm, its behavior only depends on the relative ordering of the array elements, and not on the elements themselves. More precisely, given an array A of n distinct elements, let e_i denote the i-th smallest number. Then, there exists a permutation π such that $A[i] = e_{\pi(i)}$ for all[4] $i \in [n]$. The following figure shows two different arrays with the same permutation $\pi = (3, 4, 9, 8, 6, 2, 5, 7, 1)$.



The behavior of QUICKSORT only depends on the permutation π; so, we may as well consider π to be the input to the algorithm. The number of permutations of n elements is $n!$, and we let $\mathcal{D}_n$ be the uniform distribution over these $n!$ permutations.

Claim 4.1.2. *The average-case running time of* QUICKSORT *with respect to the uniform distribution on permutations of length n is $O(n \log n)$.*

Proof. Let π be a random permutation over n elements, and let T_π be the random variable that is the number of comparisons made by QUICKSORT on π. On input π, let q be the output of PARTITION, let π' be the permutation on $q-1$ elements that is the input of the first recursive QUICKSORT call, and let π'' be the permutation on $n-q$ elements that is the input of the second recursive QUICKSORT call. Then:

$$\mathop{\mathbb{E}}_{\pi}[T_\pi] = \mathop{\mathbb{E}}_{\pi',q,\pi''}[n-1+T_{\pi'}+T_{\pi''}] = n-1+\sum_{i=1}^{n}\left(\mathop{\mathbb{E}}_{q,\pi'}[T_{\pi'} \mid q=i] + \mathop{\mathbb{E}}_{q,\pi''}[T_{\pi''} \mid q=i]\right)\cdot \Pr[q=i] \tag{4.1}$$

For the first equality, we used that PARTITION makes $n-1$ comparisons (regardless of q). For the second equality, we use linearity of expectations and the fact that for any random variable X and Y, we have $\mathbb{E}_X[Y] = \sum_x \mathbb{E}_X[Y \mid X = x] \cdot \Pr[X = x]$.

Lemma 4.1.3. $\Pr[q = i] = \frac{1}{n}$.

Proof. Note that $q = \pi(1)$ because the pivot is selected to be the first element. Since π is a random permutation, $\pi(1)$ is equally likely to be any element in $[n]$. □

Lemma 4.1.4. *Conditioned on $q = i$, π'' is a uniformly chosen permutation on $n - i$ elements.*

Proof. It is easy to verify that PARTITION does not change the order of the elements larger than the pivot. Any ordering of these elements is equally likely under π. □

Lemma 4.1.5. *Conditioned on $q = i$, π' is a uniformly chosen permutation on $i - 1$ elements.*

[3] You can justify this choice using the principle of maximum entropy.
[4] We use $[n]$ to denote the set $\{1, \ldots, n\}$.

Proof. Unlike Lemma 4.1.4, the relative ordering of elements smaller than the pivot may not be preserved by PARTITION! E.g., applying it to $[4, 8, 6, 2, 1, 7]$ yields $[1, 2, 4, 8, 6, 7]$ which flips the order of 1 and 2.

We claim the loop invariant that at the start of every iteration of the **for** loop in PARTITION, $A[j+1\,..\,i-1]$ is randomly ordered[5]. Note that by Claim 4.1.1(ii), these elements are all less than the pivot.

The base case is trivial because the array is empty. To prove maintenance, let A' be the array before the iteration with $j = J+1$ and let A be the array before the $j = J$ iteration. Suppose $i = I'$ in the former and $i = I$ in the latter iteration. There are two cases:

- $A'[J+1]$ is smaller than the pivot. Here, $I = I'$. Note that $A[J+2\,..\,I-1] = A'[J+2\,..\,I'-1]$ is randomly ordered by induction, and because of the uniformity of π, how $A[J+1]$ compares to $A[J+2\,..\,I-1]$ is unconstrained.
- $A'[J+1]$ is larger than the pivot. Here, $I = I'-1$. The set of elements in $A[J+1\,..\,I]$ and $A'[J+2\,..\,I']$ are the same. The only difference is that the last element of $A'[J+2\,..\,I']$ is moved to the first element of $A[J+1\,..\,I]$. If the first array is randomly ordered, the second 'rotated' array is also.

At termination, this implies that $A[p+1\,..\,i-1]$ are randomly ordered. Line 8 moves $A[i-1]$ to $A[p]$, and as in the second case above, this does not affect the uniformity of $A[p\,..\,i-2]$, which corresponds to π'. □

Remark. The last two lemmas were not explicitly discussed during lecture. I am including them here to have a fully rigorous argument.

Let $A(n)$ be the average-case number of comparisons with respect to the uniform distribution on permutations of length n. Plugging in the above lemmas into (4.1):

$$A(n) = n - 1 + \frac{1}{n}\sum_{i=1}^{n}(A(i-1) + A(n-i))$$

The latter sum can be simplified. To see this, note that each of $A(0), A(1), \ldots, A(n-1)$ occurs twice in the sum. Setting $A(0) = 0$, we get:

$$A(n) = n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i)$$

Now, we are ready to show the asymptotic bound on $A(n)$ using the substitute-and-check method. We claim that $A(n) \le 2n \ln n$.

The base case $n = 1$ definitely holds. Assume by strong induction that $A(i) \le 2i \ln i$ for all $i < n$. Then:

$$\begin{aligned} A(n) &= n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} A(i) \\ &\le n - 1 + \frac{2}{n}\sum_{i=1}^{n-1} 2i \ln i \\ &\le n - 1 + \frac{2}{n}\int_{i=1}^{n} 2i \ln i \end{aligned}$$

The last line uses the fact that for an increasing function f, $\sum_{i=1}^{n-1} f(i) \le \int_{i=1}^{n} f(i)di$. This is a simple observation about area under the curve. Integrating:

$$A(n) \le n - 1 + \frac{2}{n}\left(n^2 \ln n - \frac{n^2}{2} + \frac{1}{2}\right) \le 2n \ln n.$$

□

[5]More formally, the permutation corresponding to $A[j+1\,..\,i-1]$ is uniformly distributed over permutations on $i-j-1$ elements.

4.2 Randomized Algorithms

The shine of Claim 4.1.2 is marred by the fact that it seems to strongly use the assumption of a uniform input permutation. Can we modify QUICKSORT so that we get the $O(n \log n)$ runtime guarantee even in the worst case? We already saw that the earlier implementation of QUICKSORT has $\Theta(n^2)$ runtime in the worst case if the first element is always picked as the pivot.

The idea behind randomized algorithms is to purposefully inject randomness into the execution of the algorithm and **not** assume that the input is random. Given a randomized algorithm RandAlg, its *(worst-case) expected running time* is:

$$E(n) = \max_{\text{input } x \text{ of size } n} \mathbb{E}[\text{Runtime of RandAlg on } x]$$

where the expectation is over the randomness of the algorithm.

For example, a randomized version of QUICKSORT is to first randomly permute the input array and then feed it into the usual algorithm.

PERMUTE-QUICKSORT(A)
1 $\tau =$ uniformly chosen permutation on $A.length$ elements
2 Apply permutation τ on A
3 **return** QUICKSORT($A, 1, A.length$)

Then, the average-case guarantee of QUICKSORT translates into a bound on the expected running time of PERMUTE-QUICKSORT. Let's argue this more slowly. Recall that we can consider the input for a comparison-based sorting algorithm to be a permutation π on n elements (n as usual is the length of the input). Then the expected runtime is:

$$\begin{aligned}
&\max_{\pi} \mathbb{E}[\text{Runtime of PERMUTE-QUICKSORT on } \pi] \\
&\quad = \max_{\pi} \mathop{\mathbb{E}}_{\tau}[\text{Runtime of QUICKSORT on } \tau \circ \pi] \\
&\quad = \mathop{\mathbb{E}}_{\tau}[\text{Runtime of QUICKSORT on } \tau]
\end{aligned}$$

because no matter what π is, if τ is a random permutation, then $\tau \circ \pi$ is also completely random. Hence, by Claim 4.1.2, the expected running time of PERMUTE-QUICKSORT is $O(n \log n)$.

There is another way to randomize quick sort that yields the same expected running time guarantee and is also somewhat simpler. The idea is to use a randomized partition function that chooses the pivot to be a random element (instead of the first element) from the original array.

RANDOMIZED-PARTITION(A, p, r)
1 $i =$ RANDOM(p, r)
2 Swap $A[p]$ and $A[i]$
3 **return** PARTITION(A, p, r)

Here, RANDOM(p, r) returns a uniform random element from the set $\{p, p+1, \ldots, r\}$. Then, RANDOMIZED-QUICKSORT is the same as QUICKSORT, except that it calls RANDOMIZED-PARTITION instead of PARTITION in line 2.

Claim 4.2.1. *The expected running time of* RANDOMIZED-QUICKSORT *is* $O(n \log n)$.

Proof. Fix an arbitrary input permutation π of size n, and let T_π be the random variable denoting the number of comparisons made by RANDOMIZED-QUICKSORT on π. RANDOMIZED-PARTITION makes $n-1$ comparisons and returns a uniformly random index q in[6] $[n]$. Let π'_q and π''_q be the inputs to the two recursive

[6] Recall again that $[n]$ is our shorthand of $\{1, \ldots, n\}$.

calls. Note that for a particular choice of q, π'_q and π''_q are fixed as π itself is not random. Then:

$$\mathbb{E}[T_\pi] = \mathop{\mathbb{E}}_{q,\dots}[n-1+T_{\pi'_q}+T_{\pi''_q}] = n-1+\sum_{i=1}^{n}\left(\mathop{\mathbb{E}}_{q,\dots}[T_{\pi'_q} \mid q=i] + \mathop{\mathbb{E}}_{q,\dots}[T_{\pi''_q} \mid q=i]\right)\cdot \Pr[q=i] \tag{4.2}$$

The averaging is over the randomness in the top-level call to RANDOMIZED-PARTITION as well as the internal randomness in the two recursive calls to RANDOMIZED-QUICKSORT. In the above, the choice of q corresponds to the first source of randomness, while the second is denoted by the dots $(\dots)$.

I want to emphasize that (4.2) and (4.1) may look almost identical, but they are two very different assertions! In (4.2), π is not random, while in (4.1), the averaging is over the choice of π. It turns out that analyzing (4.2) is also a bit simpler than (4.1). Let $E(n)$ denote the worst-case expected number of comparisons made by RANDOMIZED-QUICKSORT. Since conditioned on $q=i$, π'_q and π''_q are inputs of length $i-1$ and $n-i$ respectively, we can write:

$$E(n) \le n-1+\frac{1}{n}\sum_{i=1}^{n}(E(i-1)+E(n-i))$$

This is exactly the same recurrence as that for $A(n)$ in Claim 4.1.2, and hence, $E(n) \le O(n\log n)$. □

As another example of a randomized algorithm, consider the following algorithm for the *selection problem*. Here, you are given an input array $A = (A[p], A[p+1], \dots, A[r])$ of $n = r-p+1$ distinct elements, as well as an integer $k \in [n]$, and the goal is to output the k-th smallest element of A.

RANDOMIZED-SELECT(A, p, r, k)
1 **if** $p == r$
2 **return** $A[p]$
3 $q =$ RANDOMIZED-PARTITION(A, p, r)
4 **if** $k == q-p+1$
5 **return** $A[q]$
6 **elseif** $k < q-p+1$
7 **return** RANDOMIZED-SELECT$(A, p, q-1, k)$
8 **else return** RANDOMIZED-SELECT$(A, q+1, r, k-(q-p+1))$

Correctness should be clear, using strong induction and correctness of RANDOMIZED-PARTITION. As for the running time analysis, it seems the main difference from RANDOMIZED-QUICKSORT is that now we make one recursive call instead of two. But this leads to an asymptotically smaller expected runtime.

Claim 4.2.2. *The expected running time of* RANDOMIZED-SELECT *is* $O(n)$.

Proof. Let's use the same notation as in the proof of Claim 4.2.1.

$$\mathbb{E}[T_\pi] \le n-1+\max(\mathbb{E}[T_{\pi'_q}], \mathbb{E}[T_{\pi''_q}]).$$

Just like what we did in (4.2), we get:

$$E(n) \le n-1+\frac{1}{n}\sum_{i=1}^{n}\max(E(i-1), E(n-i)).$$

We can assume that E is non-decreasing. So, $\max(E(i-1), E(n-i)) = E(\max(i-1, n-i))$. Hence[7]:

$$E(n) \le n-1+\frac{2}{n}\sum_{i=(n-1)/2}^{n-1} E(i)$$

[7] I am assuming n is odd here for simplicity. Otherwise, we need to do floors and ceilings, which as explained in a previous lecture, doesn't change the analysis.

Let's show that $E(n) \leq 4n$. For the base case ($n \leq 1$), this is trivially true. For the induction, we check:

$$\begin{aligned} E(n) &\leq n - 1 + \frac{2}{n} \sum_{i=(n-1)/2}^{n-1} 4i \\ &= n - 1 + \frac{8}{n} \frac{n+1}{2} \left(\frac{n-1}{2} + \frac{n-3}{4} \right) \\ &\leq n - 1 + \frac{8}{n} \frac{n+1}{2} \frac{3(n-1)}{4} \\ &\leq n - 1 + 3n \leq 4n \end{aligned}$$

For the second inequality, we used a formula for the sum of an arithmetic sequence: $\sum_{i=0}^{N-1}(a+i) = N\left(a + \frac{N-1}{2}\right)$. □

Randomized-Quicksort and Randomized-Selection are examples of *Las Vegas* algorithms. These algorithms are always correct. Their running time is a random variable that may sometimes be large but are bounded in expectation. On the other hand, there also exist *Monte Carlo* algorithms for which correctness is a random event. In other words, such algorithms may give an incorrect output with some small probability.

Example 1. Consider a timed version of Randomized-Quicksort which aborts after making $\lfloor 10n \ln n \rfloor$ comparisons and just returns the original input. Here, the running time is trivially $O(n \log n)$. On the other hand, it may be incorrect in the event that Randomized-Quicksort makes more than $\lfloor 10n \ln n \rfloor$ comparisons, which happens with small but positive probability. Hence, this version of quick sort is a Monte Carlo algorithm.

Example 2. A classic example of Monte Carlo algorithms is Karger's algorithm for the *minimum-cut problem*. Here, one is given an undirected graph, and the goal is to compute the smallest number of edges whose removal will make the graph disconnected. See the Wikipedia page for the beautifully simple algorithm.

There are many problems for which we know of randomized algorithms that are much simpler than any deterministic algorithm with similar performance. Some classic examples are Welzl's algorithm for the smallest circle problem or the Miller-Rabin primality test. You are encouraged to take CS5330, our module on randomized algorithms, for a detailed tour through this beautiful area.