Analysis of Algorithms I: Randomized Algorithms

Xi Chen

Columbia University

Randomized algorithms: Algorithms that make random choices / decisions by, e.g., fliping coins or sampling random numbers.

1. Call Random$(0, 1)$ to get a bit b which is 0 with $1/2$ probability and 1 with $1/2$ probability (an unbiased coin flip).
2. Sometimes we flip biased coins: a p-biased coin, where $p : 0 \leq p \leq 1$, returns 0 with probability $1 - p$ and 1 with probability p.
3. Call Random$(1, n)$ to get an integer drawn between 1 and n uniformly at random: for every $i : 1 \leq i \leq n$, it equals i with probability $1/n$.

The behavior / performance of a randomized algorithm depends on the random samples it gets. Consider the following algorithm ALG:

1. For $i = 1$ to n do
2. Flip a p-biased coin
3. If it is 1 then perform an operation that takes m steps

Clearly the running time of ALG depends on the number of 1's in the outcome of the n biased coins.

We assume true randomness: Samples we get are independent from each other. E.g., the outcome of the ith coin does not depend on the outcomes of previous coins, and will not affect coins in the future.

But keep in mind that in practice, we get randomness from pseudorandom generators.

We now start with some terminologies and notation that will be used in describing and analyzing randomized algorithms.

Sample space: Set of all possible outcomes (or sample points). Most of the time, we deal with finite and discrete sample spaces.

1. If an algorithm flips n coins, then the sample space is $\{0,1\}^n$, where each sample point consists of n bits

$$\mathbf{b} = (b_1, b_2, \ldots, b_n) \in \{0,1\}^n$$

2. If an algorithm draws k random numbers between 1 and n, then the sample space is $\{1,2,\ldots,n\}^n$, and each sample point consists of k numbers

$$\mathbf{a} = (a_1, a_2, \ldots, a_k) \in \{1,2,\ldots,n\}^k$$

Each sample point (outcome) in the sample space is associated with a probability:

1. If we flip n independent (usually assumed by default) and unbiased coins, then every sample point $\mathbf{b} \in \{0,1\}^n$ has probability $1/2^n$, or $\Pr(\mathbf{b}) = 1/2^n$.
2. If we flip n p-biased coins (1 with probability p), then the probability of $\mathbf{b}$ depends on the number of 1's in $\mathbf{b}$:

$$\Pr(\mathbf{b}) = p^{\text{number of 1's in } \mathbf{b}} \cdot (1-p)^{\text{number of 0's in } \mathbf{b}} \qquad (1)$$

3. If we draw k integers between 1 and n uniformly at random, the probability of each $\mathbf{a} \in \{1,\ldots,n\}^k$ is $1/n^k$: $\Pr(\mathbf{a}) = 1/n^k$.

Let S denote the sample space, then we always have

$$\sum_{x \in S} \Pr(x) = 1.$$

Let S denote the sample space, then an event A is a subset of S, and its probability is defined as

$$\Pr(A) = \sum_{x \in A} \Pr(x)$$

where the sum is over all sample points in A.

For example, if we flip n coins, then

$$A = \left\{ \mathbf{b} \in \{0,1\}^n \,\middle|\, \text{there are three 1's in } \mathbf{b} \right\}$$

is an event. And

$$\Pr(A) = \sum_{\mathbf{b} \in A} \Pr(x).$$

If the coins are independent and p-biased, we have

$$\Pr(A) = \binom{n}{3} \cdot p^3 \cdot (1-p)^{n-3} \tag{2}$$

because $|A| = \binom{n}{3}$ (check the appendix if you are not familiar with binomial coeffients) and for every $\mathbf{b} \in A$, $\Pr(\mathbf{b})$ follows from (1).

Given two events A and B (both are subsets of the sample space S), we usually use $\Pr(A \text{ and } B)$ to denote $\Pr(A \cap B)$, the probability that both events happen. Moreover, we say events A and B are independent if

$$\Pr(A \text{ and } B) = \Pr(A) \cdot \Pr(B).$$

In general, events are not independent. E.g., consider A as the event that the first 3 bits are 1 and B as the event that there are four 1's in total. Show that they are not independent.

Now let's rewind back, and see where $\Pr(\mathbf{b}) = 1/2^n$ comes from when we flip n independent and unbiased coins. This is because

$$\begin{aligned}
\Pr(\mathbf{b}) &= \Pr(\text{coin 1 is } b_1 \text{ and coin 2 is } b_2 \text{ and } \cdots \text{ and coin } n \text{ is } b_n) \\
&= \Pr(\text{coin 1 is } b_1) \cdot \Pr(\text{coin 2 is } b_2 \text{ and } \cdots \text{ and coin } n \text{ is } b_n) \\
&\cdots \\
&= \Pr(\text{coin 1 is } b_1) \cdot \Pr(\text{coin 2 is } b_2) \cdots \Pr(\text{coin } n \text{ is } b_n) \\
&= 1/2^n
\end{aligned}$$

The second equation and the equations that we skip all come from the assumption of independence. Try to derive $\Pr(\mathbf{b})$ when the coins are biased.

Let S denote the sample space, then a random variable is a map from S to real numbers (or it maps each sample point in S to a real number). For example,

1. The number of 1's in n coin flipping is a random variable (it maps each sample point to the number of 1's in it).
2. Given an input instance I, the running time $t(I)$ of a randomized algorithm is a random variable (it maps each sample point in the sample space of the algorithm to its running time on I using this sample point, because given I and a sample point, the behavior of the algorithm is deterministic and its running time is a fixed number).

Given a random variable X, its expectation is

$$E[X] = \sum_x x \cdot \Pr(X = x)$$

where the sum is over all possible values x of X. In particular, given a randomized algorithm and an input I, we are interested in its expected running time

$$E[t(I)] = \sum_i i \cdot \Pr(t(I) = i)$$

We measure the performance of a randomized algorithm using its worst-case expected running time:

$$\begin{aligned} T(n) &= \max_{I \text{ with input size } n} \text{expected running time on } I \\ &= \max_{I \text{ with input size } n} E[t(I)] \end{aligned}$$

For example, we will show that for any input instance I of length n, the expected running time of Randomized Quicksort is $O(n \lg n)$. This implies that its worst-case expected running time is $O(n \lg n)$.

Come back to ALG described earlier:

1. Let X denote the number of 1's in the n coins, and let t denote the running time of ALG (note that we use t instead of $t(I)$ because ALG takes no input).
2. These two random variables X and t satisfies

$$t = m \cdot X + O(n)$$

where $O(n)$ accounts for the time lines 1 and 2 take. (Formally, an equation involving multiple random variables means given any sample point in the sample space, the values of the random variables involved satisfy the equation.)

As a result, we have

$$E[t] = m \cdot E[X] + O(n)$$

For basic properties of expectations (e.g, why we can take m and $O(n)$ out the expectation), check the appendix.

Question: what is $E[X]$. By the definition, we have

$$E[X] = \sum_{i=0}^{n} i \cdot \Pr(X = i)$$

Here the sum is from 0 to n because these are the only possible values of X (the number of 1's cannot be negative and cannot be larger than n no matter what sample point we get).

Using (2), we get

$$E[X] = \sum_{i=0}^{n} i \cdot \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}$$

There seems no way to simplify this sum, unless you are good at dealing with binomial coefficients. Actually there is a different and much simpler way to calculate $E[X]$.

For each $i : 1 \leq i \leq n$, let X_i denote the following (indicator) random variable: $X_i = 1$ if the ith coin is 1; $X_i = 0$ otherwise. Then we get the following equation:

$$X = X_1 + X_2 + \cdots + X_n$$

because for any sample point $\mathbf{b} \in \{0, 1\}^n$, X_i contributes 1 to the right hand side if the outcome of the ith coin is 1 and contributes 0 otherwise. Thus, the right hand side is always equal to the number of 1's in the sample point $\mathbf{b}$, which is X by definition.

Here comes the step that greatly simplifies the calculation of $E[X]$. Using the linearity of expectations, we get

$$E[X] = E[X_1 + X_2 + \cdots + X_n] = E[X_1] + E[X_2] + \cdots + E[X_n]$$

Moreover, because X_i is a 0-1 (indicator) random variable,

$$E[X_i] = 0 \cdot \Pr(X_i = 0) + 1 \cdot \Pr(X_i = 1) = \Pr(X_i = 1).$$

Because the coins are p-biased, we have

$$\Pr(X_i = 1) = p \quad \Rightarrow \quad E[X] = pn \quad \Rightarrow \quad E[t] = pmn + O(n)$$

We use another example to demonstrate the use of indicator variables to simplify the calculation of expectations. Max-Cut is the following problem: Let $G = (V, E)$ be an undirected graph, where $V = \{1, 2, \ldots, n\}$ has n vertices and E has m edges. Given a partition of the vertices, that is, a map f from V to $\{0, 1\}$, an edge $ij \in E$ is called a cut edge if $f(i) \neq f(j)$ (or the two vertices lie in different parts of the partition). In Max-Cut, we are given a graph G and need to find a partition that maximizes the number of cut edges.

This is a very difficult problem. Later in the course, we will introduce the complexity class NP and Max-Cut is one of the first problems shown to be NP-complete (or one of the most difficult problems in NP). However, we now give an extremely simple randomized algorithm. Even though it does not give us the largest cut, the expected number of cut edges we get is $m/2$. (This is not too bad because there are m edges in total so the size of the largest cut can be no more than m.) Considering how simple this randomized algorithm is, it does a good job in demonstrating the power of randomization.

Randomized-Max-Cut(G), where $G = (V, E)$, $V = \{1, \ldots, n\}$

1. For $i = 1$ to n do
2. flip an unbiased coin b, and set $f(i) = b$
3. output f

Running time: $\Theta(n)$. This is a typical randomized algorithm where the outcome of the random samples does not affect its running time (in contrast to Randomized Quicksort). To see how good the partition f is, we need to understand the following random variable:

$$X : \text{ number of cut edges in the partition } f$$

Here X is a random variable because it is a map from the sample point f to nonnegative integers between 0 and m.

However, working on X directly seems very difficult:

$$E[X] = \sum_{i=0}^{m} \Pr(X = i)$$

How to calculate $\Pr(X = i)$? One way is the following: For each edge $ij \in E$, where $i \neq j$ (no self-loop in G), we have

$$\begin{aligned}
& \Pr(ij \text{ is a cut edge}) \\
= \ & \Pr((f(i) = 0 \text{ and } f(j) = 1) \text{ or } (f(i) = 1 \text{ and } f(j) = 0) \\
= \ & \Pr(f(i) = 0 \text{ and } f(j) = 1) + \Pr(f(i) = 1 \text{ and } f(j) = 0) \\
= \ & 1/4 + 1/4 = 1/2
\end{aligned}$$

So every edge $ij \in E$ is a cut edge with probability $1/2$. From the argument used earlier in calculating the probability that there are i 1's in n unbiased coins, we have

$$\Pr(X = i) = \binom{m}{i} / 2^m$$

However, the equation above is not correct. The reason is because the m events, [ij is a cut edge], where $ij \in E$, are not independent (unlike the coins that we assume to be independent).

For example, consider the following graph $G = (V, E)$ with three vertices $V = \{1, 2, 3\}$ and $E = \{12, 23, 31\}$. We have

$$\Pr(12, 23, 13 \text{ are all cut edges}) = 0$$

because in no partition these three can all be cut edges. (Why?) This implies that the following three events [12 is a cut edge], [23 is a cut edge] and [31 is a cut edge] are not independent, because if they are independent then the probability should be $1/8$.

So, how do we calculate $E[X]$? We use indicator random variables and the linearity of expectations! For each edge $ij \in E$, we let:

$$X_{i,j} : \text{ 1 if } ij \text{ is a cut edge and 0 otherwise}$$

From the earlier analysis, we already have

$$E[X_{i,j}] = \Pr(X_{i,j} = 1) = \Pr(ij \text{ is a cut edge}) = 1/2.$$

It is also clear that

$$X = \sum_{ij \in E} X_{i,j}$$

because $X_{i,j}$ contributes 1 to the right hand side if ij is a cut edge and contributes 0 otherwise, so the right hand side is equal to the total number of cut edges.

As a result, using the linearity of expectation, we get

$$E[X] = \sum_{ij \in E} E[X_{i,j}] = \sum_{ij \in E} 1/2 = m/2$$

So the expected cut size is $m/2$.

We next introduce Quicksort:

- Divide-and-Conquer
- "in place" sort (using only constant extra space)
- Good performance in practice

MergeSort: recursively merging; Quicksort: recursively partitioning.

- **Divide**: Given an array A of length n, use $x = A[1]$ as a pivot to partition A. More exactly, the goal is to exchange elements of A so that it becomes a reordering of the input array:

$$A[1], A[2], \ldots, A[i-1], A[i] = x, A[i+1], \ldots, A[n]$$

  for some $i : 1 \leq i \leq n$, such that $A[1], A[2], \ldots, A[i-1] \leq x$ and $A[i+1], \ldots, A[n] > x$.

- **Conquer**: Sort recursively the lower subarray $A[1 \ldots i-1]$ and the upper subarray $A[i+1 \ldots n]$ using Quicksort.

- **Combine**: Trivial.

If we do not need the "in place" feature, the partition operation is easy to implement: Set the pivot x to be the first element in the array to be partitioned; create two empty arrays; make a pass of all elements in the array to be partitioned except the first one; for each of them compare it with x and move it to one of the arrays depending on the comparison result.

Here is a beautiful implementation of "in place" partition:

Partition (A,p,q), which partitions $A[p, p+1, \ldots, q]$ using $A[p]$

1. Set the pivot $x = A[p]$ and $i = p$
2. for $j = p + 1$ to q do
3. if $A[j] \leq x$ then set $i = i + 1$ and exchange $A[i]$ and $A[j]$
4. exchange $A[i]$ and $A[p]$
5. return i (where do we use this returned value?)

The loop invariant we use to prove the correctness of Partition is the following: At the beginning of each loop, we have $A[p] = x$,

$$A[k] \leq x \quad \text{for all } k : p+1 \leq k \leq i \tag{3}$$

$$A[k] > x \quad \text{for all } k : i+1 \leq k \leq j-1. \tag{4}$$

Before the first loop, we have $i = p$ and $j = p+1$ so (3) and (4) are trivially satisfied (because there is no k between $p+1$ and $i = p$ or between $i+1 = p+1$ and $j-1 = p$).

Induction Step: Assuming that the loop invariant holds at the beginning of the jth loop, for some $j : 1 \leq j \leq n$, we show that it holds at the beginning of the next loop. Two cases to consider:

If $A[j] > x$, then the only action in the loop is to increment j. But after j is incremented, (4) still holds because we know that $A[j-1]$ (or $A[j]$ before j is incremented) satisfies $A[j-1] > x$.

If $A[j] \leq x$, then after i is incremented, we know $A[i] > x$ due to the loop invariant (or inductive hypothesis). After exchanging $A[i]$ and $A[j]$, we know $A[k] \leq x$ for all $k : p + 1 \leq k \leq i$ and $A[k] > x$ for all $k : i + 1 \leq k \leq j$. The loop finishes by incrementing j and thus, the loop invariant holds.

By induction, we conclude that the loop invariant holds at the beginning of each loop and in particular, the last loop where $j = n + 1$. From this, the correctness of Partition follows. It is also easy to see that the running time of Partition is $\Theta(q - p)$, linear in the length of the input array. An equivalent characterization of its running time is

$$\Theta(\text{comparisons between } A[j] \text{ and the pivot } x \text{ made in line 3})$$

We will find the latter expression more helpful in the analysis of Randomized Quicksort next class.

Quicksort (A, p, q)

1. If $q = p$ return
2. $r =$ Partition (A, p, q) (Check Partition to see what it returns)
3. Quicksort $(A, p, r - 1)$
4. Quicksort $(A, r + 1, q)$

Using induction, one can prove the correctness of Quicksort. So given any input sequence A or length n, we can call Quicksort $(A, 1, n)$ to sort it. But, what about its performance?

We show that the worst-case complexity of Quicksort is $\Theta(n^2)$: If the input sequence A is already sorted, e.g., $A = (1, 2, \ldots, n)$, then Quicksort takes $\Omega(n^2)$ steps. To see this, we use $u(k)$ to denote the number of steps that Quicksort takes when the input is $(n-k, n-(k-1), \ldots, n-1, n)$ of length $k+1$, for each integer $k = 0, 1, \ldots, (n-1)$. We get the following recurrence:

$$u(0) = \Theta(1)$$
$$u(k) = u(k-1) + \Theta(k) \quad \text{for all } k \geq 1$$

where $\Theta(k)$ accounts for the running time of Partition as well as a recursive call to Quicksort with an empty input array.

To solve this recurrence, we first put back the constants:

$$\begin{aligned} u(0) &= c_1 \\ u(k) &= u(k-1) + c_2 \cdot k \quad \text{for all } k \geq 1 \end{aligned}$$

for some positive constants c_1 and c_2. Then

$$\begin{aligned} u(k) &= u(k-1) + c_2 \cdot k \\ &= (u(k-2) + c_2 \cdot (k-1)) + c_2 \cdot k \\ &= (u(k-3) + c_2 \cdot (k-2)) + c_2 \cdot (k-1) + c_2 \cdot k \\ &\cdots \\ &= (u(0) + c_2 \cdot 1) + c_2 \cdot 2 + \cdots + c_2 \cdot k \\ &= \Omega(k^2). \end{aligned}$$

On the other hand, in Section 7.4.1 of the textbook, it is proved that the worst-case running time of Quicksort over any sequence of length n is $O(n^2)$. The proof uses the substitution method because the Master theorem does not apply.

We conclude that the worst-case complexity of Quicksort is $\Theta(n^2)$.

So why do we call it Quicksort even though it has quadratic worst-case complexity? The intuition is in a "typical" sequence (imagine a random permutation), the first element which we use as a pivot may not be smallest / largest (this happens if we hit a jackpot). Usually the partition operation produces two subarrays of comparable sizes (unlike the worst-case example in which one subarray is empty and the other contains all elements except the pivot). And if every pivot we pick produces two subarrays of comparable sizes, one can show that its running time is $O(n \lg n)$.

However, an adversary may always feed a sorted sequences so that Quicksort takes quadratic time, essentially because it knows how Quicksort picks pivots. What if we change the way the pivots are selected? E.g., what if we pick the second element or the last element as a pivot? No use. As long as the adversary knows how the pivot is picked, it can always construct a sequence that leads to $\Omega(n^2)$ running time.

Then, what if we randomly pick a pivot? We will see in next class that by adding randomness into Quicksort, no adversary can come up with a bad sequence. Indeed, for any sequence of length n, we will show that the expected running time of Randomized Quicksort is $O(n \lg n)$.