Probability for Randomized Algorithms

1 Probability Basics

1.1 Why Randomized Algorithms?

In order to analyze the work and span of randomized algorithms, we will often find ourselves using probabilty and related topics. In particular, when analyzing the work of a randomized algorithm, we typically use **expected value**. Looking at span requires us to use a different probability tool – the **high probability bound**. To understand both of these, we'll also introduce some other concepts, such as random variables, which will allow us to more easily apply probability to randomized algorithms.

But why do we even use randomized algorithms in the first place? As it turns out, the best randomized algorithms are often asymptotically equivalent to the best deterministic ones, and in some cases, they can be better. Moreover, deterministic algorithms often have worst-case inputs where they might perform particularly badly. Randomized algorithms can avoid this problem since their inherent randomness can prevent adversarial inputs from leading to extremely long runtimes.

Implementing randomized algorithms can also be simpler than their deterministic counterparts. An example of this comes from the problem of primality testing, where we need to return whether or not an integer is prime. The Miller-Rabin randomized algorithm for this problem is very simple and can be implemented quickly, in around 20 lines of Python or a similar language. The actual details of this algorithm are outside the scope of this course, but it is notable for running in expected polynomial time. For 30 years after its creation, it was unknown whether a deterministic polynomial time algorithm for this problem even existed, and when one was found, it turned out to be very complex.

Randomness can also be used to break symmetry in algorithms. There are sometimes places in our algorithms where we are presented with multiple options to explore, and our runtime is heavily dependent on which option we choose. We'll see this later, when quicksort chooses an arbitrary pivot with which it splits the array. In these cases, randomness can be used to select one of the options, ensuring that each one is used some of the time. In contrast, deterministically selecting the same choice every time could lead to always getting an undesirable runtime.

1.2 Disadvantages of Randomized Algorithms

Definition: Two Types of Randomized Algorithms

A **Las Vegas** algorithm is an algorithm whose cost bounds are a random variable, but always will return the correct result.

A **Monte Carlo** algorithm may not generate the right answer every time, but does get the right answer with some positive probability. Thus, running a Monte Carlo algorithm many times increases the probability that the right answer is found at least once.

In our course, we will focus on the study of Las Vegas algorithms. When writing such an algorithm, we ideally want to prove that when it returns, it is always going to return the correct result. This means that, if it is called with the same inputs repeatedly, it should output the same things every time. Despite this, it will use randomness within its execution, meaning that the execution paths that it goes down may vary between runs. This can lead to unpredictability in terms of runtime and resource usage, which is sometimes an undesirable property. In fact, the runtime of a randomized algorithm with sufficiently unlucky randomness can be unbounded! The probability of this happening is typically small enough that it is not problematic in practice, but is at least worthy of note here.

In a parallel context, further concerns regarding runtime emerge. If we run a randomized process on several parallel threads and wait for them all to finish, then our overall runtime is equal to that of the slowest single thread. Even if all of the other threads finish quickly, having to wait for one slow thread can lead to inefficiency. Finally, randomized algorithms can also be difficult to analyze, since we have to account for the randomness when finding the work and span of such an algorithm.

2 Example: The Random Distance Run

The CMU Random Distance Run is an annual event put on by SCS, involving two large fuzzy six-sided dice. The two dice are both rolled, and the sum of the numbers on top of the dice is used to determine how many laps the competitors have to run around the CMU track. The race can be as short as half a mile, if two 1s are rolled, and can get up to 3 miles (just under 5 kilometers) if two 6s are rolled. We can use the Random Distance Run to explore some basic probability definitions.

The first definition we want to consider is that of a **sample space**, typically denoted Ω. A sample space represents the set of possible outcomes in a situation. We also consider a **probability measure**, which is a function P that maps the power set of Ω to $\mathbb{R}$ with some restrictions. In particular, for any event A we have $0 \leq P(A) \leq 1$ – all probabilities are between 0 and 1. It also needs to hold that $P(\Omega) = 1$, meaning that one of the possibilities in Ω always happens. Finally, if A and B are disjoint events, then $P(A)+P(B)$ must be equal to $P(A \cup B)$. Essentially, a probability measure is a function which tells us the probabilities of **events** – an event is defined as a subset of the sample space. Our last important definition is that of a **random variable**. Random variables are typically denoted by capital letters such as X. They are deterministic

functions which map the sample space to the real numbers.

Example: Random Distance Run

In the Random Distance Run, we can define two random variables. We can let D_1 be the value that the first die rolls, and D_2 be the value that the second die rolls. Our sample space is the set of possible rolls for the two dice, and D_1 and D_2 each take outcomes from the sample space and map them to real numbers in $\{1,2,3,4,5,6\}$.

A special type of random variable is the **indicator random variable**, which is always either equal to 1 or 0 based on whether some event occurs or does not occur. We will use indicator random variables frequently when analyzing algorithms.

Once we have random variables to work with, we can start considering their expected values. The **expected value** of a random variable can be thought of as a weighted sum of its possible outcomes. More formally, we define the expected value of X as:

$$\mathbb{E}[X] = \sum_{a \in \Omega} \Pr(a) \cdot X(a)$$

Here, we sum over all possible outcomes in the sample space. For each outcome, we multiply the probability of it occurring by the value of X if it does occur.

Example

The expected value of D_1 is $7/2$. To find this, we can note that the random variable takes on the values 1, 2, 3, 4, 5, and 6, each with probability $1/6$. Then our computation is:

$$1 \cdot \frac{1}{6} + 2 \cdot \frac{1}{6} + \cdots + 6 \cdot \frac{1}{6} = \frac{1+2+3+4+5+6}{6} = \frac{7}{2}$$

The expected value of D_2 is the same as that of D_1, since it is also a fair six-sided die – assuming it is independent of D_1.

But what happens if we wanted to find the expected number of laps that the runners have to run? This would be the expected value of $D_1 + D_2$, the sum of two random variables. It turns out that we can use a theorem known as **linearity of expectation** here!

Theorem: Linearity of Expectation

For two random variables X and Y, it holds that $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$.

Example

The expected value of $D_1 + D_2$, the sum of the dice, is $7/2 + 7/2 = 7$ by linearity of expectation.

What if we wanted to know the expected product of two dice? This problem is a bit harder. Let's define a new random variable $D = D_1 D_2$, equal to the product of the two dice. This expected

value is a bit harder, since multiplying two expected values does not always give the expected value of the product. There are still some cases where this is allowed, though.

Theorem: Products of independent random variables

For two *independent* random variables X and Y, $\mathbb{E}[XY] = \mathbb{E}[X]\mathbb{E}[Y]$.

We say that two random variables are **independent** if $\Pr[X = a, Y = b] = \Pr[X = a]\Pr[Y = b]$ for all possible outcomes a and b – that is, the value of X does not affect the value of Y, and vice versa.

Example

The expected value of the product of the two dice is $(7/2) \cdot (7/2) = (49/4)$, assuming the dice are independent.

What if we don't have independence? In this case, we have to compute our expected value from the definition, by finding each of the different potential values and multiplying them by their probabilities, then taking the sum.

Example

Let's say the two dice are still fair and six-sided, but now are rigged in a way that leads to them always rolling the same number.

The expected value of the sum of the dice is still 7. Each die has an expected value of 7/2, and while the dice are not independent, linearity still holds. However, the expected value of the product is more complicated. There is a 1/6 probability of both dice being 1, the same probability of them both being 2, and so on and so forth. Thus, we sum:

$$1 \cdot \frac{1}{6} + 4 \cdot \frac{1}{6} + \cdots + 36 \cdot \frac{1}{6} = \frac{1+4+9+16+25+36}{6} = \frac{91}{6}$$

Notice that this expected value is **not** the same as the expected product of the two independent dice that we computed earlier!

Other functions, such as the max function, are even more complicated to compute when applied to random variables. We can systematically calculate the value of the function on every outcome in the sample space, then multiply each by the probabilities, eventually giving us our desired answer. In general, arbitrary functions do not compose well with expected values for random variables.

3 Further Probability Definitions

The **union bound** can be used to provide an upper bound on the probability that at least one event out of some set of events occurs. For two events A and B, it states that the probability that either A or B happens is upper bounded by $\Pr[A] + \Pr[B]$ – that is, $\Pr[A \cup B] \leq \Pr[A] + \Pr[B]$.

More generally, it states that the probability of the union of events is upper bounded by the sum of the probabilities of each of the individual events. This gives us a powerful tool for bounding the probability that no "bad" events occur in certain situtations.

The **conditional probability** of A given B, often written $\Pr[A \mid B]$, is the last definition we will introduce. It is defined as:

$$\Pr[A \mid B] = \frac{\Pr[A \cap B]}{\Pr[B]}$$

3.1 Harmonic Numbers

The *harmonic numbers* show up frequently in the analysis of randomized algorithms.

Definition: Harmonic Numbers

The n^{th} **harmonic number** H_n is defined as the sum of the reciprocals of the first n positive integers. Mathematically:

$$H_n = \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n} = \sum_{i=1}^{n} \frac{1}{i}$$

Harmonic numbers are nice to see because they are small.

Theorem: Harmonic numbers are logarithmic

The n^{th} harmonic number H_n, satisfies

$$\ln n < H_n < \ln n + 1$$

In simpler terms, $H_n = \Theta(\log n)$, which is a quantity we are usually happy to see in algorithms!

Example: Car Clusters

Suppose n cars are driving along a straight line, initially spaced so that none are touching. Each car is assigned a distinct maximum speed, chosen uniformly at random. Cars accelerate up to their maximum speed, but if a car reaches the one in front, it slows down to match that car's speed.

A *cluster* is a group of cars that end up driving together at the same speed. How many clusters should we expect?

Observe that car i (counting from front to back) forms a new cluster if and only if its speed is less than the speeds of all cars in front of it. Since speeds are assigned uniformly at random, this occurs with probability $1/i$.

Let X_i be the indicator variable for the event that car i forms a new cluster. Then $\mathbb{E}[X_i] =$

$1/i$, and the total number of clusters is

$$C = \sum_{i=1}^{n} X_i.$$

By linearity of expectation,

$$\mathbb{E}[C] = \sum_{i=1}^{n} \frac{1}{i} = H_n = \Theta(\log n).$$

4 Tail Bounds

When analyzing a randomized algorithm, we primarily care about what will happen most of the time. This is not to say we don't care about the things that are unlikely to happen, though. In fact, these cases represent how our algorithm is doing when it is running badly. We'd like to consider how often we end up with a particularly bad runtime, and this is where **tail bounds** can help us. These bounds are inequalities telling us how often we will end up with a runtime greater than some threshold, and are often based on the expected runtime that we calculate.

One common tool we use for tail bounds is called Markov's inequality.

Theorem: Markov's Inequality

For a random variable X which always is greater than or equal to 0:

$$\Pr[X \geq a] \leq \frac{\mathbb{E}[X]}{a}$$

What does this mean intuitively? If we have a probability distribution for X with some expected value $\mathbb{E}[X]$, then Markov's inequality says that we can't be very far above $\mathbb{E}[X]$ too often. To show this, let's look at what would happen if the probability $\Pr[X \geq a]$ was greater than $\mathbb{E}[X]/a$. The rest of the time, X is at least 0, which "contributes" a nonnegative value to the expected value calculation. However, with probability greater than $\mathbb{E}[X]/a$, X is at least a, meaning that the expected value is greater than $(\mathbb{E}[X]/a) \cdot a$: a contradiction.

5 Analyzing Algorithms

The **quicksort** algorithm is a randomized algorithm. It operates on an array A by picking a pivot randomly, then splitting the array into two parts – the elements less than A, and the elements greater than A. It then recursively quicksorts both sub-parts, and then appends them to get the final solution.

Being a randomized algorithm, it's possible for this algorithm to perform very badly with an unfortunate choice of pivots. But how likely is this outcome? As it turns out, this happens with

a very small probability. Thus, it's not worth worrying about too much. Rather, we concern ourselves with the expected work, and the span that occurs with high probability. Our goal is to analyze the work and span in our fork-join model. This means that we might need to consider cost graphs that are random in nature, and whose work and span are represented as random variables rather than integers. We'll do the analysis of the quicksort algorithm itself later, but we first need to define these tools.

To find the work, we can add the expected work of each part of an algorithm using linearity of expectation. The span is more complicated, because the span of two things in parallel uses the max function. This means that we cannot use linearity of expectation, and thus, for span we try to use high-probability bounds rather than expected values.

5.1 High Probability Bounds

Definition: High Probability

We say that a random variable $W(n) \leq f(n)$ **with high probability** (w.h.p.) if there exists a constant n_0 such that for any integer value of $k \geq 1$ and any $n \geq n_0$:

$$\Pr[W(n) \leq k \cdot f(n)] \geq 1 - \frac{1}{n^k}$$

We will also write $W(n) \in O(f(n))$ w.h.p. if $W(n) \leq c \cdot f(n)$ w.h.p. for some constant c.

This is a way of expressing a very stringent tail bound on the random variable $W(n)$. As we increase k, the bound is loosened but must be met with higher probability. Typically, this is done by showing that $\Pr[W(n) > k \cdot f(n)] < 1/n^k$, for sufficiently large n.

Why do we insist on being able to do this with all values of k? The answer is that it allows us to apply a **union bound** over polynomially many events in n that each hold w.h.p., and show that all of them hold simultaneously w.h.p. For example, this works well for bounding the span of parallel programs. If I run an algorithm that forks n processes, the span of the overall algorithm is equal to the maximum span of any of the forked processes.

Theorem: Max Preserves w.h.p Bounds

Let $S(n)$ be a non negative random variable, and let $T(n) = \max(S(n), \ldots, S(n))$, where there are n (not necessarily independent) copies of $S(n)$ in the max. If $S(n) \leq f(n)$ w.h.p. then $T(n) \leq 2f(n)$ w.h.p. It follows that if $S(n) \in O(f(n))$ w.h.p. then so is $T(n)$.

Proof. Denote the n instances of $S(n)$ as $S_1(n), \ldots, S_n(n)$. The event that $T(n) > (k+1)f(n)$ is the same as the event that at least one of the events $S_i(n)$ satisfies $S_i(n) > (k+1)f(n)$. Thus we can apply the union bound and infer that $\Pr[T(n) > (k+1)f(n)] \leq n \Pr[S(n) > (k+1)f(n)]$. But we know that $\Pr[S(n) > (k+1)f(n)] < \frac{1}{n^{k+1}}$. It follows that $\Pr[T(n) > (k+1)]f(n) < \frac{1}{n^k}$. Since $k \geq 1$ we have $2k \geq k+1$, and can then infer that $\Pr[T(n) > k \cdot (2f(n))] < \frac{1}{n^k}$. Therefore $T(n) \leq 2f(n)$ w.h.p. □

5.2 The Skittles Game

Let's now consider another game to illustrate high probability bounds.

Problem: Skittles Game

The Skittles game is played with a fair coin and a pile of n Skittles. It is a single-player game played in rounds. Initially there are $s = n$ Skittles. Each round consists of the player flipping the coin once. If it comes up heads, then the player eats $\lceil \frac{s}{2} \rceil$ of the Skittles and there are $\lfloor \frac{s}{2} \rfloor$ remaining. If it comes up tails, the player proceeds to the next round without eating any Skittles. The game ends when there are no Skittles remaining. We are interested in the random variable $R(n)$ which is the number of rounds the game lasts.

In the worst case, the game takes forever, because we might flip tails on every round and never eat any Skittles. But this outcome has probability zero. Intuitively, you might guess that the game will last $O(\log n)$ rounds, because the number of Skittles will roughly be halved every couple of rounds, whenever a head is flipped. In fact, $O(\log n)$ is a high probability bound for the length of the Skittles game. Let's prove this formally.

Claim: Bounds on the Skittles Game

The Skittles game will end in $O(\log n)$ rounds with high probability.

Proof. To simplify things, let's change the game a little bit. We change the number of skittles we eat when we get heads to be $s/2$ rather than $\lfloor s/2 \rfloor$. The modified game ends when the number of skittles goes below 1. Let $T(n)$ be the random variable for the modified game, while $R(n)$ is for the original game. For any run of the game (using the same source of random coin flips we know that $R(n) \leq T(n)$. So from now on we focus on $T(n)$.

The outline of the proof is to consider the expected number of Skittles left after i rounds. If we model this as a random variable, then we can apply Markov's inequality to it to find how many rounds it will take for Pr[≥ 1 Skittle remains] to become sufficiently low.

So let X_r be the random variable equal to the number of Skittles remaining after r rounds have finished. Let's find the value of X_i in terms of i. To do this, we'll consider each individual round to see how that round's coin toss might affect the number of Skittles remaining.

On any given round, we have a $1/2$ probability of flipping heads, which multiplies the current size of the pile by $1/2$. We also have a $1/2$ probability of flipping tails, which leaves the pile size unchanged. Therefore,

$$X_{r+1} = \begin{cases} \frac{1}{2} X_r & \text{prob. } \frac{1}{2} \\ X_r & \text{prob. } \frac{1}{2} \end{cases}$$

$$\mathbb{E}[X_{r+1}] = \frac{1}{2} \cdot \mathbb{E}\left[\frac{1}{2} X_r\right] + \frac{1}{2}\mathbb{E}[X_r] = \frac{1}{4} \cdot \mathbb{E}[X_r] + \frac{1}{2}\mathbb{E}[X_r] = \frac{3}{4}\mathbb{E}[X_r]$$

Additionally, all rounds are independent, so we unroll the recurrence to get

$$\mathbb{E}[X_r] = n \cdot \left(\frac{3}{4}\right)^r.$$

The event that $T(n) > r$ is exactly the same as the event that $X_r \geq 1$, because the latter event is equivalent to saying that least one more round after r will occur. So:

$$\Pr[T(n) > r] = \Pr[X_r \geq 1].$$

So at this point, to finish the proof, we need to find the value of r such that $\Pr[X_r \geq 1]$ is small enough. The trick is to apply Markov's inequality to turn our bound on $\mathbb{E}[X_r]$ into a bound on $\Pr[X_r \geq 1]$. Specifically Markov's inequality is the $\leq$ in the following statement.

$$\Pr[X_r \geq 1] \leq \frac{\mathbb{E}[X_r]}{1} = n \cdot \left(\frac{3}{4}\right)^r.$$

Now we are ready to match this to the definition of *with high probability*. Specifically, for any k, we'd like to find a number of rounds r in terms of k after which the probability of having finished is at least $1 - 1/n^k$. This is equivalent to saying that after r rounds, our probability of having 1 or more Skittles left needs to be less than $1/n^k$. So we need a value of r such that the following is satisfied:

$$n \cdot \left(\frac{3}{4}\right)^r < \frac{1}{n^k}.$$

Rewriting this in terms of r, we get

$$r > \log_{\frac{4}{3}}(n^{k+1}) > 2.409(k+1)\log_2 n.$$

Since $2k \geq k+1$ and $5 > 2 \cdot 2.409$, if we choose a $r = k \cdot 5\log_2 n$, then it is big enough to give us the desired bound. Thus we have shown that

$$\Pr[T(n) > k \cdot 5\log_2 n] < \frac{1}{n^k}.$$

It now follows that $T(n) \leq 5 \cdot \log_2 n$ w.h.p. and $T(n) \in O(\log n)$ w.h.p. □

We can generalize this phenomenon in a reusable lemma which we will invoke in the next few lectures.

Lemma: Skittles Lemma

Let $X_0 = n$, and let X_{r+1} be computed by a random process from X_r that guaranties that $0 \leq X_{r+1} \leq X_r$, and also that $\mathbb{E}[X_{r+1}] \leq \alpha\,\mathbb{E}[X_r]$ where $0 < \alpha < 1$ is a constant. Let $T(n)$ be the random variable which is the first round r when $X_r < 1$.

It follows that $T(n)$ is $O(\log n)$ w.h.p. (More specifically $T(n) \leq \beta \log_2 n$ w.h.p., where $\beta = -2/\log_2 \alpha$.)