CS3230 - Design and Analysis of Algorithm: Practice Set #2 - Randomised Algorithms and Analysis

Eldon Chung
eldon.chung@u.nus.edu

National University of Singapore — February 24, 2019

Introduction

This is a practice and **completely optional** practice problem set that revolves around analysis of randomised algorithms, and some bounds. Solutions will not be provided! Post on the LumiNUS forums if you would like to attempt the questions, and help will be provided in the form of either verification or guidance. Alternatively you may email or text me and I'll definitely reply and help.

A few comments about convention: The algorithms provided will be in pseudocode. Arrays here will be 1-indexed. For *this* problem set you may also assume that array access, comparison, and arithmetic is done in $\Theta(1)$ time. Arrays will be passed by reference in function calls and returns. Allocating space for or initialising n-sized arrays (to whatever value) will take $\Theta(n)$ time. Also there are no space limitations unless otherwise stated. In fact, any deviation from this standard will be explicitly stated.

In terms of notation, an n-sized array A will be denoted as $A[1..n]$. Accessing the i^{th} element in an array is denoted by $A[i]$. Variable assignments are denoted by $\leftarrow$, e.g. $x \leftarrow 5$ would mean variable x is being assigned a value of 5. As is standard, ∞ will be used to denote infinity, and despite not being an actual realisable concept in most machines and programming languages, it is ostensibly `INT_MAX` in C++ or `Integer.MAX_VALUE` in Java. Meanwhile $\emptyset$ will be used to denote empty data structures (not to be confused with null pointers). E.g. $S = \emptyset$ can be a stack that contains no elements, yet it is possible to push elements into S, and invoking a pop operation on S when it is empty would just return nothing instead of crashing as most languages do.

Additionally: $Random(a, b)$ will be a random number generator that returns any integer in the range (inclusive) $[a, b]$ with equal probability, that is: $\frac{1}{b-a+1}$.

Some Useful Intuition

Now I don't want to spoonfeed, and I don't want to do your thinking for you, it's the best way to learn! But I'll list out a few intuitions that I've found useful so far that can potentially be very helpful. Most of these will be the same as what I've already mentioned in class, I'm just listing them out in case you'd like to see them again or missed them the first time. Also go read the lecture notes, I'll not be repeating most of the definitions. Also the notion of random variables, expectation etc is already covered in CS1231 as well. So go revise the relevant content!

0.1 Are We Counting? Or Are We Flipping Coins?

There's two main random variables[1] that we focus on (though of course there's many more than that): indicator random variables and geometric random variables, and although in some sense similar, there are a few ways to decide the between the two.

Indicator random variables, are random variables that can only take on values 1 (with probability p) and 0 (with probability $1-p$). So the expectation is just p. Now these guys usually are what we usually use to count events.

On the other hand, geometric random variables model events like coin flipping. It's like you're given a coin that gives heads with probability p and tails with probability $1-p$, and the random variable gives you the number of times you need to flip until you get a heads. So with probability p you only need 1 flip, with probability $(1-p)\cdot p$ you need two flips, et cetera. So in general with probability $(1-p)^{n-1}\cdot p$ you need exactly n flips. And you can try to prove it but here I'll take for granted that the expectation for such a random variable, is $\frac{1}{p}$.

Try to recognise what sort of situation you're doing, is it count the number of occurences of a single event? Or are you trying to count the number of tries before something happens? This question usually helps to distinguish between the two. Disclaimer: I can't promise that it always works that way since my experience is also somewhat limited, so please take this guide with a pinch of salt, of course you'll need to be sure yourself on which variable you should be using.

0.2 Linearity is So Convenient!

Linearity of expectation is super helpful! When you're counting something, you could try to express it as a sum of other random variables, then just apply the linearity of expectation.

[1] If this term sounds unfamiliar to you, really, please revise the lecture notes.

Questions

Exercise 1 Finding the Minimum (Randomised)

Surprise surprise! You're given the minimum finding algorithm again! But this time, *randomised*.

```
1: procedure FIND-MIN(A[1..n])
2:     minElement ← ∞
3:     for i = 1 to k do
4:         r ← Random(1, n)
5:         if minElement > A[r] then
6:             minElement ← A[r]
7:         end if
8:     end for
9:     return minElement
10: end procedure
```

In essence, this algorithm looks at k elements randomly (and notice that it may look at the same element more than once), then updates the $minElement$ as necessary.

1. What is the probability (with respect to k) that after k samples, we find the minimum?

2. For an array of size n, what value should k be, with respect to n, so that the probability that we correctly find the minimum is $\geq \frac{1}{2}$?

3. Using the same idea, for an array of size n, what value should k be, with respect to n, so that the probability that we correctly find the minimum is $\geq p$?

4. Notice that the runtime of the algorithm is $\Theta(k)$, based on your answer for the previous question, if you want to be correct with probability at least $1 - \frac{1}{n}$, would this run faster than the deterministic $\Theta(n)$ time solution?

Exercise 2 Randomised Search

Consider the problem where we were given an unsorted array, and were asked to search for an item and return its index. We now know that in the worst case this takes n comparisons to find where it is. Instead what we now do is shuffle the array first, then perform the linear search.

1. Find the expected number of checks before we find the item (assuming it exists).

Exercise 3 Fun With Randomness

Now this is a question I always ask people because I think it's fun! Say you were given a broken random generator $Broken - Random(0, 1)$ that returns 1 with probability p (not necessarily $\frac{1}{2}$), and 0 with probability $1 - p$.

1. Design a randomised algorithm, that uses $Broken - Random(0, 1)$ such that it returns 0 with probability $\frac{1}{2}$ and 1 with probability $\frac{1}{2}$.

2. Prove that your algorithm is correct.

3. What is the expected runtime of your algorithm?

Exercise 4 Fun With Randomness 2: Electric Boogaloo

Okay now let's try to generalise! Say you're given $Random(0, 1)$, and this works, so it returns 0 with probability $\frac{1}{2}$ and 1 with probability $\frac{1}{2}$.

1. Using $Random(0, 1)$, make $Random(a, b)$. i.e. design an algorithm that returns any integer in the range $[a, b]$ with equal probability.

2. Prove that your algorithm works.

3. What is the expected runtime of your algorithm?

Exercise 5 Counting Inversions Strikes Again!

Recall that given an n sized array $A[1..n]$ (you may assume with distinct values), an inversion is when $A[i] > A[j]$ and $i < j$. Intuitively this means that the values need to be inverted to be sorted.

** 1. Say we shuffled any given array so that we may obtain any of the $n!$ permutations with equal probability. Find the number of inversions in expectation. Come up with as tight a bound as possible, including constant factors. (Hint: Perhaps one of the indicator random variables will be useful here.)

Exercise 6 Everyday I'm shufflin

Say we had a notion of distance for an element, where after we shuffle, the distance of an element is the number of places it is away from it's place if it were sorted. E.g $[3, 6, 4, 2, 1, 5]$, the distance for element 3 is 2, and the distance for 6 is 4, the distance for 4 is 1, the distance for 2 is 2 and the distance for 1 is 4, and the distance for 5 is 1. So the total distance here is: $2 + 4 + 1 + 2 + 4 + 1$.

* 1. Find the expected total distance. Come up with as tight a bound as possible, including constant factors.

** Exercise 7 Average Case Arguments

We all know that bubble sort is quite slow in the worst case, what about if we shuffle the array first to mitigate the worst case?

1. First it's highly recommended you find the average number of inversions in a random permutation. I.e. solve the previous question on inversions.

2. Bound asymptotically the largest number of inversions that can be reduced by the inner loop of bubble sort.

* 3. Using the expected number of inversions that can be reduced by each (full) run of the inner loop, and the total expected number of inversions that can happen on a randomly permuted/shuffled array, find a bound asymptotically on the expected number of times the outer loop should run. Conclude with the average case complexity (asymptotically) for bubble sort.

*** Exercise 8 Average Case Arguments 2

Okay let's consider the majority finding problem again in lecture 5. Say we were given an $2n+1$ sized array $A[1..2n+1]$ with only 0's and 1's. We also know we can stop when we've seen either more than half 0's or more than half 1's (more than half here means $\geq n+1$). This exercise will consider separate randomised scenarios:

1. Say there was an adversary that gave either gave us n 1's and $n+1$ 0's or $n+1$ 1's and n 0's and we had to decide between which, also assume further that the adversary knew whatever algorithm we would implement. Clearly deterministic algorithms do not work here, since the adversary can force the worst case consistently. Instead now let us consider the following algorithm where you firstly shuffle the array, then loop from left to right and stop the moment you see either $n+1$ 0's or 1's. (Basically by then ,like we mentioned you would know what is the majority). What is the expected runtime? Give your solution including the constant factor.

1. Say now instead this felt like too much work and we don't care about being right all the time, being right some of the time is good enough. Instead now what we do is sample $2k+1$ numbers ($k \leq n$, and without replacement), then output the majority of these $2k+1$ numbers as the majority of the array. What is the probability that we are right? i.e. the majority in our sample was actually the same as the array.

** 1. Okay let's change the situation a little. Now we are given a deterministic algorithm that loops from the first element to the last, and again it keeps the counter and outputs the majority that it sees, and again it stops early the moment it knows which is the majority. Instead, every element has an equal probability of being either 0 or 1. In other words, an $2n+1$ sized array as 2^{2n+1} possibilities. What is the expected runtime?

** 2. Given the same situation as before but now again we randomly sample $2k+1$ elements ($k \leq n$, and without replacement), then output the majority of these $2k+1$ numbers as the majority of the array. What is the probability that we are right?