CMIMC 2017
Power Round

INSTRUCTIONS

1. Do not look at the test before the proctor starts the round.
2. This test consists of several problems, some of which are short-answer and some of which require proofs, to be solved within a time frame of **60 minutes**. There are **150 points** total.
3. Answers should be written and clearly labeled on sheets of blank paper. Each numbered problem should be *on its own sheet*. If you have multiple pages, number them as well (e.g. 1/3, 2/3).
4. Write your team ID on the upper-right corner and the problem and page number of the problem whose solution you are writing on the upper-left corner on each page you submit. Papers missing these will not be graded. Problems with more than one submission will not be graded.
5. Write legibly. Illegible handwriting will not be graded.
6. In your solution for any given problem, you may assume the results of previous problems, even if you have not solved them. You may not do the same for later problems.
7. Problems are not ordered by difficulty. They are ordered by progression of content.
8. No computational aids other than pencil/pen are permitted.
9. If you believe that the test contains an error, submit your protest in writing to Doherty Hall 2302 prior to the end of lunch.

1 Introduction

Randomness is a phenomenon present in all areas of mathematics and science, for it allows us to model very complex systems in a way that is rather well-tamed by mathematicians. The CMIMC 2017 Power Round aims to explore how randomness can be used as a powerful tool in algorithms.

2 Random Variables

A *random variable* is a variable that takes a given value with a certain probability. The set of all possible values a given random variable can take is called its *sample space*; how often it takes a given value depends on its *probability distribution*. We will only deal with discrete sample spaces and distributions. For instance, here are the three major distributions we will look at:

- *Discrete uniform distribution*: We say $X \sim \text{Discrete Uniform}(N)$ if X takes the values $\{1, 2, \ldots, N\}$ with equal probabilities.
- *Bernoulli distribution*: We say a random variable X has the Bernoulli distribution – denoted $X \sim \text{Bernoulli}(p)$ – if X takes the value 1 with probability p and 0 with probability $1 - p$.
- *Geometric distribution*: Let X be the random variable corresponding to the number of samples from $\text{Bernoulli}(p)$ until we get a 1. Then $X \sim \text{Geometric}(p)$.

A very important point to note is the difference between random variables and random numbers: random variables describe an entire distribution and have no numerical value, whereas random numbers are the numerical result of sampling from a distribution. For instance, the random variable corresponding to a coin toss describes all the possible outcomes and their respective probabilities; however, flipping the coin once and observing it to be heads corresponds to a random number.

Definition 2.1. We call two random variables X and Y *independent* – denoted $X \perp Y$ – if

$$\mathbf{Pr}[(X = a) \cap (Y = b)] = \mathbf{Pr}[X = a] \cdot \mathbf{Pr}[Y = b]$$

for all a, b. Here $\mathbf{Pr}[X = a]$ denotes the probability that sampling from the random variable X yields the result a.

We will now look at a very useful statistic in describing the distributions of random variables.

Definition 2.2. Suppose X can take the values $X_1, X_2, \ldots$. The *expected value* of a random variable X, denoted by $\mathbf{E}(X)$, is defined to be

$$\mathbf{E}[X] = X_1 \cdot \mathbf{Pr}[X = X_1] + X_2 \cdot \mathbf{Pr}[X = X_2] + \cdots = \sum_{k=1}^{\infty} X_k \cdot \mathbf{Pr}[X = X_k].$$

Note that if the number of outcomes is some finite number N, then $\mathbf{Pr}[X = X_j] = 0$ for all $j \geq N + 1$, so this sum is well-defined.

1. [**6**] Suppose $X \sim \text{Geometric}(p)$. Prove that $\mathbf{E}[X] = \frac{1}{p}$.

Here are four very important properties of the expected value:

- $\mathbf{E}[c \cdot X] = c \cdot \mathbf{E}[X]$ and $\mathbf{E}[c + X] = c + \mathbf{E}[X]$, where c is a constant.
- $\mathbf{E}[X \cdot Y] = \mathbf{E}[X] \cdot \mathbf{E}[Y]$, if $X \perp Y$.
- (Linearity of expectation) $\mathbf{E}[X + Y] = \mathbf{E}[X] + \mathbf{E}[Y]$.

2. [**8**] Prove these four properties.
3. [**4**] Suppose $X \sim \text{Bernoulli}(0.75)$ and $Y \sim \text{Discrete Uniform}(6)$ are independent random variables. Define X' and Y' to be random variables satisfying $X' = 6 \cdot X + 4$ and $Y' = 3 \cdot Y + 2$. What is $\mathbf{E}[X' + Y']$ and $\mathbf{E}[X' \cdot Y']$?
4. [**7**] Suppose we randomly distribute $m \geq 1$ balls into $n \geq 1$ initially empty bins, so that each ball has an equal chance of being placed in each bin. What is the expected number of balls in the first bin? What is the expected number of empty bins?

3 Algorithms

An *algorithm* is a set of instructions for performing a task. For instance, the following two algorithms determine if a nonnegative integer is odd or even:

```
Data: a nonnegative integer n
Result: whether n is odd or even
while n ≥ 0 do
    if n = 0 then
        return n is even
    end
    else if n = 1 then
        return n is odd
    end
    else
        n ← n − 2
    end
end
```

```
Data: a nonnegative integer n
Result: whether n is odd or even
for i = 0, 2, 4, 6, ... do
    if n = i then
        return n is even
    end
    else if n = i + 1 then
        return n is odd
    end
end
```

We will now detail a common convention in writing algorithms called *pseudocode*. **It is not required that you follow these conventions, but you must be able to clearly describe the steps of your algorithms.** There are only a few conventions that we need to introduce:

[variable name] ← [value]	assigns (or reassigns) [value] to [variable name]
if [condition] **then**	perform the indented instruction if [condition] is satisfied
else if [condition] **then**	perform the indented instruction if the previous **if** condition isn't satisfied but [condition] is satisfied
else	perform the indented instruction if none of the previous **if** conditions are satisfied
while [condition] **do**	iteratively perform the indented instruction as long as [condition] is satisfied
for [variable name] = [sequence] **do**	iteratively perform the indented instruction for each value of [variable name] in the sequence
return [value]	the output of the algorithm

Algorithms generally require *correctness* – the algorithm always outputs the right answer – and *termination* – the algorithm doesn't run infinitely.

5. [**4**] Prove that the above two algorithms are correct and always terminate.

We will also introduce a way to generate random numbers.

Definition 3.1. Let the function $\mathbf{B}(p)$ return a random number $X \sim \text{Bernoulli}(p)$, and let the function $\mathbf{D}(N)$ return a random number $X \sim \text{Discrete Uniform}(N)$.

With these two basic functions, we can simulate many complex probabilistic events. For example, suppose we want to generate a random number X according to the following distribution:

value	0	1	2
$\mathbf{Pr}[X = \text{value}]$	1/4	1/2	1/4

One algorithm that can simulate such a distribution is:

Data: none
Result: a random number with the above distribution
$u \leftarrow \mathbf{B}(1/2)$
$v \leftarrow \mathbf{B}(1/2)$
return $u + v$

6. [**2**] Prove that this algorithm accurately returns a random number with the above distribution.

7. [**10**] Let p be any real number in $[0, 1]$. Using only calls to $\mathbf{B}(1/2)$, devise an algorithm that terminates with probability 1 and returns a random number $X \sim \text{Bernoulli}(p)$. Prove that your algorithm is correct and terminates with probability 1. **Partial credit of at most 4 points will be awarded for solving the case of $p = 1/3$.**

Finally, we will discuss how to shuffle a deck of cards. Suppose we have $N \geq 3$ cards $C_1, C_2, \ldots, C_N$ lined up in a row, and we want to shuffle the deck so that it is equally likely to see any possible outcome. We have devised the following algorithm to perform this task:

Data: the cards $C_1, C_2, \ldots, C_N$
Result: a uniformly random permutation of $C_1, C_2, \ldots, C_N$
for $i = 1, 2, \ldots, N$ **do**
  $j \leftarrow \mathbf{D}(N)$
  swap the positions of C_i and C_j
end

8. [**7**] Prove that this algorithm is incorrect, i.e., it does not properly shuffle the cards.

9. [**7**] Fix the algorithm, and prove correctness.

4 Randomized Algorithms

Finally we meet the significant subject: randomized algorithms. A *randomized algorithm* is an algorithm that uses random variables. For instance, when the CMIMC staff plays chess, we determine who plays as white by flipping a coin, which is a simulation of $\mathbf{B}(0.5)$. A *deterministic algorithm* is an algorithm that always produces the same output when run on the same input. Note: this does not mean all randomized algorithms are nondeterministic.

Let's examine why randomized algorithms might be useful. We all know the game of battleship, where each player places his ships, and then each player tries to guess the coordinates of the ships. Well, the game of one-dimensional battleship involves the first player setting k of the numbers $A[1], A[2], \ldots, A[4k]$ equal to 1 and the rest equal to 0. The second player's goal is to guess some index i for which $A[i] = 1$, and after each guess the first player reveals if it is correct or not.

0	0	1	0	0	0	1	1	0	1	0	0	0	0	0	0

You are the second player in a game of one-dimensional battleship.

10. [**2**] Find, with proof, a deterministic algorithm for playing one-dimensional battleship that requires at most $3k + 1$ guesses.

11. [**2**] Suppose you play with a deterministic algorithm. Prove that there exists some configuration by the first player for which you require at least $3k + 1$ guesses.

12. [**6**] Find, with proof, a randomized algorithm for playing one-dimensional battleship such that the expected number of guesses your strategy requires is a constant independent of k. Your algorithm does not necessarily need to terminate.

Sometimes, when we design an algorithm, instead of always outputting the right answer, we can design an algorithm that will output the wrong answer with a very small probability.

Definition 4.1. A *Las Vegas algorithm* is an algorithm for which computation time and space may be large, but it always outputs the right answer. A *Monte Carlo algorithm* is an algorithm for which the worst-case run-time is small, but it errs with a small probability.

To demonstrate this idea, we will present a Monte Carlo algorithm to determine whether or not a number is prime. But first, we will state without proof that there is a function IsComposite(n, a) that takes n and a number $a \in \{2, 3, \ldots, n-1\}$ with the following properties:

- If n is prime, then IsComposite(n, a) always returns **false**.
- If n is composite, then IsComposite(n, a) incorrectly returns **false** for less than half of the numbers $a \in \{2, 3, \ldots, n-1\}$.

Knowing this function, we can now describe this primality test algorithm:

Data: a positive integer n
Result: whether or not n is prime
for $i = 1, 2, \ldots, 100$ **do**
  $a \leftarrow 1 + \mathbf{D}(n-2)$
  if IsComposite(n, a) **then**
    return n *is composite*
  end
end
return n *is prime*

13. [**5**] What is the probability this algorithm outputs "n is composite" when n is actually prime? What is the probability that this algorithm outputs "n is prime" when n is actually composite? How can we decrease these two probabilities?

5 Sorting

The problem of sorting a list is a classical problem in any text on algorithms. Here we will present two sorting algorithms, one deterministic and one randomized, and compare their efficiencies. Sorting algorithms' efficiencies are generally measured by the number of comparisons they make, i.e., the number of times we check if $A[i] < A[j]$ for some i, j.

Consider the following deterministic sorting algorithm:

Algorithm: Sorting Algorithm #1
Data: a list $L = (L[1], L[2], \ldots, L[n])$ of integers
Result: L sorted in increasing order
for $i = 1, 2, \ldots, n$ **do**
  for $j = i + 1, \ldots, n$ **do**
    if $A[j] < A[i]$ **then**
      swap $A[i]$ and $A[j]$
    end
  end
end

14. [**4**] Prove that this algorithm is correct and terminates.

15. [**2**] Find the number of comparisons this algorithm always makes.

The second algorithm uses a technique called *recursion*, which means it divides the problem into sub-problems of the same type as the original problem, solves the sub-problems, and combines the results. Remember, this is perfectly acceptable as long as we cover the base cases. So consider the following randomized sorting algorithm:

Algorithm: Sorting Algorithm #2
Data: a list $L = (L[1], L[2], \ldots, L[n])$ of integers
Result: L sorted in increasing order
if $n = 0, 1$ **then**
  return L
end
$p \leftarrow \mathbf{D}(n)$ ($L[p]$ is called a *pivot* in this case)
$L_1, L_2 \leftarrow \emptyset$
for $i = 1, 2, \ldots, p-1, p+1, \ldots, n$ **do**
  if $L[i] \leq L[p]$ **then**
    add $L[i]$ to L_1
  end
  else
    add $L[i]$ to L_2
  end
end
use this algorithm to sort L_1, L_2
return $[L_1, L[p], L_2]$

16. [**6**] Prove that this algorithm is correct and terminates.

17. [**8**] Let C be a variable denoting the number of comparisons this algorithm makes, and suppose the resulting sorted array is $\ell_1 \le \ell_2 \le \cdots \le \ell_n$. Furthermore, let A_{ij} denote the event that this algorithm at some point compares ℓ_i and ℓ_j. Prove that

$$\mathbf{E}[C] = \sum_{1 \le i < j \le n} \mathbf{Pr}[A_{ij}].$$

18. [**6**] Prove that A_{ij} occurs if and only if the first pivot chosen from $\ell_i, \ldots, \ell_j$ is either ℓ_i or ℓ_j.

19. [**6**] Prove that $\mathbf{Pr}[A_{ij}] = \frac{2}{j-i+1}$.

20. [**6**] Let

$$H_n := \frac{1}{1} + \frac{1}{2} + \cdots + \frac{1}{n}$$

denote the n^{th} harmonic number. Prove that $\mathbf{E}[C] = 2(n+1)H_n - 4n$.

21. [**4**] In the worst case, how many comparisons are made?

22. [**8**] Now you will compare the efficiencies of Sorting Algorithm #1 and Sorting Algorithm #2. Let a_n be the answer to Problem 15. Prove that

$$\lim_{n \to \infty} \frac{2(n+1)H_n - 4n}{a_n} = 0.$$

6 Randomized Approximation Algorithms

Often, it is unreasonable to find the exact solution to a problem because maybe finding the exact solution requires checking 2^{100} different cases. Therefore, sometimes it suffices to develop an *approximation algorithm*, which is an algorithm that simply finds a very good solution to a problem. For example, we do not know any efficient algorithms (and none exist if $\mathsf{P} \neq \mathsf{NP}$!) to find an assignment of boolean variables that satisfy a given set of boolean formulas; however, there is an approximation an algorithm that can find an assignment satisfying an expected 87.5% of the formulas.

We will investigate the following problem: there are n people at a party, some of whom know each other. A subset of these people is called a *cover* if, after we remove those people from the party, no two distinct people at the party know one another. The problem is, given a list of who knows who at the party, to find a cover of minimal size. The following approximation algorithm determines a cover that is close to minimal size:

Data: a list L of pairs $\{u, v\}$ of people who know each other
Result: a cover that is close to minimal size
$S \leftarrow \emptyset$
for $\{u, v\} \in L$ **do**
  if $u, v \notin S$ **then**
    randomly choose u or v with equal probability
    add the chosen vertex to S
  end
end
return S

23. [**4**] Prove that this algorithm indeed returns a cover.

24. [**8**] Let C denote any cover of minimal size. Let S_i denote the contents of S after completing the ith iteration of the loop. Prove that, for all $i \geq 0$,

$$\mathbf{E}[|S_i \cap C|] \geq \mathbf{E}[|S_i \backslash C|].$$

25. [**6**] Conclude that, after the algorithm terminates,

$$\mathbf{E}[|S|] \leq 2 \cdot |C|.$$

We have proven that this simple algorithm produces a cover that doesn't deviate too far from the minimal size of a cover.

Now consider the following variant of the original problem: for each person v at the party, we assign a number $0 < w_v \leq 1$ that describes how much we want him to stay at the party, where 1 means we really want him to stay at the party, and 0.001 means we really want to kick him out of the party. The number w_v is called a *weight*. The problem is to find a cover of minimal total weight. Notice that the original problem is the special case where all the weights are 1.

26. [**8**] For a set T, let

$$W(T) := \sum_{v \in T} w_v$$

We can modify the above algorithm by changing "randomly choose u or v with equal probability" to "randomly choose u with probability p_{uv} and v with probability $1 - p_{uv}$." Redefine C to be any cover of minimal weight, instead of minimal size. Find, with proof, the value of p_{uv} that ensures that for all $i \geq 0$,

$$\mathbf{E}[W(S_i \cap C)] \geq \mathbf{E}[W(S \backslash C)]$$

27. [**4**] Conclude that, after the algorithm terminates,

$$\mathbf{E}[W(S)] \leq 2 \cdot W(C).$$