Note: Your TA probably will not cover all the problems. This is totally fine, the discussion worksheets are not designed to be finished in an hour. They are deliberately made long so they can serve as a resource you can use to practice, reinforce, and build upon concepts discussed in lecture, readings, and the homework.

Philosophy of analyzing randomized algorithms. The first step is to always identify a *bad event.* I.e. identify when your randomness makes your algorithm fail. We will review some techniques from class using the following problem as our "test bed".

Let G be a bipartite graph with n left vertices, and n right vertices on $n^2 - n + 1$ edges.

- Prove that G always has a perfect matching.
- Give a polynomial in n time algorithm to find this perfect matching.

We will analyze the following algorithm `BlindMatching`:

- Let $\boldsymbol{\pi}$ and $\boldsymbol{\sigma}$ be independent and uniformly random permutations of $[n]$.
- If $\{\boldsymbol{\pi}(1), \boldsymbol{\sigma}(1)\}, \{\boldsymbol{\pi}(2), \boldsymbol{\sigma}(2)\}, \ldots, \{\boldsymbol{\pi}(n), \boldsymbol{\sigma}(n)\}$ is a valid matching output it.
- Else output `failed`.

Union Bound. Suppose $\boldsymbol{X}_1, \ldots, \boldsymbol{X}_n$ are (not necessarily independent) $\{0, 1\}$ valued random variables, then

$$\Pr[\boldsymbol{X}_1 + \cdots + \boldsymbol{X}_n \geq 1] \leq \Pr[\boldsymbol{X}_1 = 1] + \Pr[\boldsymbol{X}_2 = 1] + \cdots + \Pr[\boldsymbol{X}_n = 1].$$

Now we analyze our algorithm using union bound. An output $M = (\{\boldsymbol{\pi}(1), \boldsymbol{\sigma}(1)\}, \ldots, \{\boldsymbol{\pi}(n), \boldsymbol{\sigma}(n)\})$ is a valid perfect matching exactly when all edges of the form $\{\boldsymbol{\pi}(i), \boldsymbol{\sigma}(i)\}$ are present in G. A "bad event" happens if any of those pairs are not edges in G.

Let $\boldsymbol{X}_i$ be the indicator of the event that $\{\boldsymbol{\pi}(i), \boldsymbol{\sigma}(i)\}$ is *not* present in our graph.

1. What is the probability that $\boldsymbol{X}_i = 1$?
2. Use the union bound to upper bound the probability that M is *not* a valid perfect matching.
3. Conclude that G has a valid perfect matching.

The upper bound obtained on the probability of our bad event, i.e. of M not being a valid perfect matching, is fairly high. In light of this, we introduce the technique of *amplification.*

Amplification. The philosophy of amplification is that if we have a randomized algorithm that fails with probability p, we can repeat the algorithm many times and aggregate the output of all the runs to produce a new output such that the failure probability of the randomized algorithm is significantly smaller. Now consider the following algorithm `SpamBlindMatching`.

- Run `BlindMatching` independently T times.
- If at least one of the runs outputted a valid perfect matching, return the output of such a run.
- Else output `failed`.

1. What is the failure probability of `SpamBlindMatching`?
2. How large should we set T if we want a failure probability of δ?

Now we switch gears and turn our attention to concentration phenomena and its usefulness in analyzing randomized algorithms.

Markov's inequality. Let $\boldsymbol{X}$ be a *nonnegative valued* random variable, then for every $t \geq 0$:

$$\Pr[\boldsymbol{X} \geq t\mathbf{E}[\boldsymbol{X}]] \leq \frac{1}{t}.$$

1. Markov's inequality is *false* for random variables that can take on negative values! Give an example.
2. Give a tight example for Markov's inequality. In particular, given μ and t, construct a random variable $\boldsymbol{X}$ such that $\mu = \mathbf{E}[\boldsymbol{X}]$ and $\Pr[\boldsymbol{X} \geq t\mu] = \frac{1}{t}$.

Chebyshev's inequality. Let $\boldsymbol{X}$ be any random variable with well-defined variance[1], then

$$\Pr\left[|\boldsymbol{X} - \mathbf{E}[\boldsymbol{X}]| > t\sqrt{\mathbf{Var}[\boldsymbol{X}]}\right] \leq \frac{1}{t^2}.$$

To see the above inequality in action, consider the following problem:

> Let B be a bag with n balls, k of which are red and $n-k$ of which are blue. We do not have knowledge of k and wish to estimate k from ℓ independent samples (with replacement) drawn from B.

Let $\boldsymbol{X}$ be the number of red balls sampled.

1. What is $\mathbf{E}[\boldsymbol{X}]$?
2. What is $\mathbf{Var}[\boldsymbol{X}]$?
3. Choose a value for ℓ and give an algorithm that takes in n and $\boldsymbol{X}$ and outputs a number $\widetilde{k}$ such that $\widetilde{k} \in [k - \varepsilon\sqrt{k}, k + \varepsilon\sqrt{k}]$ with probability at least $1 - \delta$.

1 4-cycles

We use $G(n,p)$ to denote the distribution of graphs obtained by taking n vertices and for each pair of vertices i, j placing edge $\{i, j\}$ independently with probability p.

(a) Compute the expected number of edges in $G(n,p)$?

(b) Compute the expected number of 4-cycles in $G(n,p)$?

(c) Give a polynomial time randomized algorithm that takes in n as input and in poly(n)-time outputs a graph G such that G has no 4-cycles and the expected number of edges in G is $\Omega(n^{4/3})$.

2 Lower Bounds for Streaming

(a) Consider the following simple 'sketching' problem. Preprocess a sequence of bits $b_1, \ldots, b_n$ so that, given an integer i, we can return b_i. How many bits of memory are required to solve this problem exactly?

[1]In this course, all random variables will have well-defined variance

(b) Given a stream of integers $x_1, x_2, \ldots$, the *majority element* problem is to output the integer which appears most frequently of all of the integers seen so far. Prove that any algorithm which solves the majority element problem exactly must use $\Omega(n)$ bits of memory, where n is the number of elements seen so far.

3 Streaming Integers

In this problem, we assume we are given an infinite stream of integers x_1,x_2, $\ldots$, and have to perform some computation after each new integer is given. Since we may see many integers, we want to limit the amount of memory we have to use in total. For all of the parts below, give a brief description of your algorithm and a brief justification of its correctness.

(a) Show that using only a single bit of memory, we can compute whether the sum of all integers seen so far is even or odd.

(b) Show that we can compute whether the sum of all integers seen so far is divisible by some fixed integer N using $O(\log N)$ bits of memory.

(c) Assume N is prime. Give an algorithm to check if N divides the product of all integers seen so far, using as few bits of memory as possible.

(d) Now let N be an arbitrary integer, and suppose we are given its prime factorization: $N = p_1^{k_1} p_2^{k_2} \ldots p_r^{k_r}$. Give an algorithm to check whether N divides the product of all integers seen so far, using as few bits of memory as possible. Write down the number of bits your algorithm uses in terms of $k_1, \ldots, k_r$.