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 $1 - \delta$.

Solution:

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

1. The probability that a random (u, v) pair is not an edge where u is a left vertex and v is a right vertex is $\frac{1}{n} - \frac{1}{n^2}$.
2. By union bounding over all n edges chosen, the probability that M is not a perfect matching is at most $1 - \frac{1}{n}$.
3. The previous part implies that M has at least $\frac{1}{n}$ probability of being a perfect matching, which means a perfect matching exists in G.
4. The failure probability of `SpamBlindMatching` is bounded by $\left(1 - \frac{1}{n}\right)^T$.
5. Setting $T = n \ln(1/\delta)$ works because $\left(1 - \frac{1}{n}\right)^n \leq \frac{1}{e}$.
6. Uniform ± 1 has expected value 0 but half chance of exceeding 0.
7. Consider the random variable that is $t\mu$ with probability $1/t$ and 0 with probability $(t-1)/t$.
8. $\mathbf{E}[\boldsymbol{X}] = \frac{k}{n}\ell$.
9. Defining $\boldsymbol{X}_i$ as the random variable that the i-th sample is red, and using independence of the $\boldsymbol{X}_i$ we have

$$\mathbf{Var}[\boldsymbol{X}] = \mathbf{Var}[\boldsymbol{X}_1 + \cdots + \boldsymbol{X}_\ell] = \mathbf{Var}[\boldsymbol{X}_1] + \cdots + \mathbf{Var}[\boldsymbol{X}_\ell] = \ell \frac{k}{n}\left(1 - \frac{k}{n}\right).$$

10. The algorithm is to output $\frac{n}{\ell}\boldsymbol{X}$. $\mathbf{E}\left[\frac{n}{\ell}\boldsymbol{X}\right] = k$ and $\mathbf{Var}\left[\frac{n}{\ell}\boldsymbol{X}\right] = \frac{kn}{\ell}\left(1 - \frac{k}{n}\right) \leq \frac{kn}{\ell}$. This quantity deviates from k by $\frac{1}{\sqrt{\delta}}\sqrt{\frac{kn}{\ell}}$ with probability at most δ. We wish to choose ℓ so that $\frac{1}{\sqrt{\delta}}\sqrt{\frac{kn}{\ell}} < \varepsilon$. This happens when $\ell = \frac{n}{\varepsilon^2\delta}$.

1 Estimating Votes

Suppose we have a stream of votes of form (Id, Yes) or (Id, No) which has person's Id (that is unique to them) and whether they voted Yes/No. We would like to estimate the fraction of Yes votes. Unfortunately, many people have voted multiple times. People who voted multiple times voted for the same option each time.

The Distinct Elements algorithm takes a stream as inputs and outputs $\tilde{n} \in [(1-\epsilon)n, (1+\epsilon)n]$ with probability $1-\delta$, where n the number of distinct elements seen in the stream, using small memory. Let $S(n, \epsilon, \delta)$ be the space complexity Distinct Elements uses in terms of n, ϵ, δ.

Using the Distinct Elements algorithm as a black box, provide an algorithm for estimating the fraction of "Yes" votes within a factor of, say, $(1+3\epsilon)$. You should count the vote given by each Id only once. State its space complexity in terms of S.

Challenge: Justify the error bound on your algorithm (you may assume $\epsilon < 1/3$ for simplicity).

Solution: The algorithm is just to run two copies of Distinct Elements. Pass the ID of "Yes" votes to the first, and pass the ID of all votes to the second. When queried for the fraction, we return the ratio of the outputs of the two data structures.

The space complexity is at most $2 \cdot S(n, \epsilon, \delta)$ if n is the number of distinct voters.

By a union bound, if the number of Yes votes is m, then the first Distinct Elements data structures returns a value in $[(1-\epsilon)m, (1+\epsilon)m]$ with probability $1-\delta$, and the second returns a value in $[(1-\epsilon)n, (1+\epsilon)n]$ with probability $1-\delta$. So our algorithm returns a value in $[\frac{1-\epsilon}{1+\epsilon}\frac{m}{n}, \frac{1+\epsilon}{1-\epsilon}\frac{m}{n}]$ with probability $(1-\delta)^2 \geq 1-2\delta$. For $\epsilon \leq 1/3$, we have $(1+\epsilon)/(1-\epsilon) \leq 1+3\epsilon$, and $(1-\epsilon)/(1+\epsilon) \geq 1-3\epsilon$. So our algorithm's output is in $[(1-3\epsilon)m/n, (1+3\epsilon)m/n]$ with probability at least $1-2\delta$. Put another way, using $O(1)$ copies of this data structure only worsens their collective error guarantee by $O(1)$.

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?

(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.

Solution:

(a) n bits. Intuitively, this is because at the end of the preprocessing, there are 2^n different 'states' the algorithm has to be in, one for each bitstring. The number of states of a machine with ℓ bits of memory is 2^ℓ, so $\ell \geq n$. A more detailed argument follows.

The preprocessing algorithm is a function $f\colon \{0,1\}^n \to \{0,1\}^\ell$, where ℓ is the number of bits of memory needed to answer queries. The query algorithm is a function $q\colon [n] \times \{0,1\}^\ell \to \{0,1\}$. Observe that we can use the query algorithm to invert f on its image: if $g(y) = (q(1,y), q(2,y), \ldots, q(n,y))$, then $g(f(x)) = x$ for all $x \in \{0,1\}^n$. Hence f is injective, which means that $\ell \geq n$.

(b) We can prove this by reduction from the previous problem. For any string of bits $b_1, \ldots, b_\ell$, we define a stream of integers $0, 0, (i,i)_{b_i=1}$. Now we can query b_i by adding i to the stream and checking if it is the majority element. The length of the sequence is $n \leq 2\ell + 1$, so the memory usage is at least $\frac{n-1}{2}$ bits.

An alternative approach is for the stream to be $((-1)^{b_i} \cdot i)_{i \in [n]}$. Then we can query b_i by adding $(i, -i)$ to the stream. If $-i$ is the majority element, then $b_i = 1$, otherwise $b_i = 0$.