RANDOMIZATION AND SUBLINEAR ALGORITHMS

SIDDHANT CHAUDHARY

These are my notes for the **Advanced Algorithms** course. The main theme covered in the course was *randomization*. My favorite reference book for randomization is: *Probability and Computing, Randomized Algorithms and Probabilistic Analysis by Michael Mitzenmacher and Eli Upfal*.

Contents

Date: 21 September, 2021.

1. Introduction

1.1. **Polynomial Identity Testing.** I have covered this nicely in my notes for a talk I gave at PROMYS 2021; the content should be available on my website.

1.2. **Max-cut in a graph.** Let us consider the *max-cut* problem. Given a graph $G = (V, E)$, we need to find the cardinality of the maximum cut of the graph. This problem is known to be **NP**-complete, and hence one cannot expect to find a polynomial time algorithm to solve this problem. Instead, we can use randomization. The algorithm description is given below.

(1) We maintain two sets S_1 and S_2, each initialised as empty sets.
(2) For each $v \in V$, toss a fair coin; depending upon the outcome, put v in the set S_1 or S_2. Do the same for each vertex independently.

Note that the maximum possible value of the max-cut in G is $|E|$. Now, for each edge $e \in E$, define an indicator random variable X_e as follows: $X_e = 1$ if after the algorithm, e is a cut edge, otherwise 0. Clearly,

$$\mathbf{P}\left[e \text{ is a cut edge}\right] = \frac{1}{2}$$

for each edge e. Hence,

$$\mathbf{E}\left[X_e\right] = \frac{1}{2}$$

So, it follows that the expected value of the number of cut edges is

$$\mathbf{E}\left[\sum_{e \in E} X_e\right] = \sum_{e \in E} \mathbf{E}\left[X_e\right] = \frac{|E|}{2}$$

So, our output is $\frac{1}{2}$ of the optimum value in expectation. Such algorithms are called *half-approximate algorithms*.

1.2.1. *Derandomization using conditional Expectation.* We will now *derandomize* our algorithm using the notion of conditional expectations.

In the previous discussion, let $C(S_1, S_2)$ denote the size of the cut. We showed that

$$\mathbf{E}\left[C(S_1, S_2)\right] = \frac{|E|}{2}$$

This means that there exists some cut in the graph with size atleast $|E|/2$. We now give a deterministic algorithm which finds such a cut in the graph.

Suppose the vertices of the graph are $v_1, ..., v_n$. Our algorithm will decide the placement of each v_i in either S_1 or S_2. Call these placement $x_1, ..., x_n$, where each $x_i \in \{A, B\}$. First we note a bunch of observations.

- We have that

$$\mathbf{E}\left[C(S_1, S_2) \mid x_1 = S_1\right] = \mathbf{E}\left[C(S_1, S_2) \mid x_1 = S_2\right]$$

This is intuitive to see because if we interchange S_1, S_2 in our algorithm, the expected values shouldn't change. Formally, this can proved by considering n-tuples $(x_1, ..., x_n)$, although I won't do that here. This means that

$$\mathbf{E}\left[C(S_1, S_2)\right] = \mathbf{E}\left[C(S_1, S_2) \mid x_1 = S_1\right] = \mathbf{E}\left[C(S_1, S_2) \mid x_1 = S_2\right]$$

- Let $1 \leq i < n-1$. Then, we have the following simple equation.

$$\mathbf{E}\left[C(S_1,S_2) \mid x_1 = s_1, x_2 = s_2, ..., x_i = s_i\right] = \sum_{j=1}^{2} \frac{1}{2}\mathbf{E}\left[C(S_1,S_2) \mid x_1 = s_1, ..., x_i = s_i, x_{i+1} = S_j\right]$$

Also, we have the following.

$$\mathbf{E}\left[C(S_1,S_2) \mid x_1 = s_1, x_2 = s_2, ..., x_{n-1} = s_{n-1}\right] = \sum_{j=1}^{2} \frac{1}{2}\text{CutSize}(x_1 = s_1, ..., x_{n-1} = s_{n-1}, x_n = S_j)$$

The above equation simply means that if the placements of $v_1, ..., v_{n-1}$ are decided, then the expectation can be easily computed.

The first of the two formulas above also shows the following.

$$\mathbf{E}\left[C(S_1,S_2) \mid x_1 = s_1, ..., x_i = s_i\right] \leq \max_{j=1,2}\left(\mathbf{E}\left[C(S_1,S_2) \mid x_1 = s_1, ..., x_i = s_i, x_{i+1} = S_j\right]\right)$$

This inequality gives us a simple algorithm to find a cut with $\geq |E|/2$ edges. The algorithms works as follows: suppose that the placements $x_1, ..., x_i$ have been decided, for some i. Then, compute the two expectations given above (the ones over which the maximum is chosen), and go with the choice of x_{k+1} that has the larger expectation.

- The only thing that remains now is to figure out how to compute the given expectations. Suppose we want to compute the expectation

$$\mathbf{E}\left[C(S_1,S_2) \mid x_1 = s_1, ..., x_i = s_i, x_{i+1} = S_j\right]$$

where $j \in \{1,2\}$. Note that in this expectation, the placements of the vertices $x_1, ..., x_{i+1}$ have already been decided; so, we can compute easily how much these vertices among themselves contribute to the cut. Now, consider all the other edges, which have atleast one endpoint not among $\{v_1, ..., v_{i+1}\}$. It is easy to see that all these edges contribute $1/2$ to this expectation. So, the expectation can be easily calculated in time $O(E)$.

- We can do even better; instead of calculating the expectations for $j = 1, 2$, note that we only need to decide which one is larger. Observe that this is only dependent on the placement of vertex v_{i+1}; we only need to count how many edges this vertex has with vertices $\{v_1, ..., v_i\}$ when $S_j = A, B$. We go by that choice which has a higher count of such edges.

So, we have obtained a greedy deterministic algorithm which gives us the required cut.

1.3. **Karger's Min Cut Algorithm.** Now we will see a randomized algorithm for finding a minimum cut in a graph. This algorithm is called *Karger's algorithm*. Here is how the algorithm works.

- An edge of the graph G is picked uniformly at random. After picking an edge, the edge is *contracted*. Multi-edges are maintained, and there are never any self loops.
- The algorithm is repeated until there are only two vertices left. Then, the number of (multi) edges between the two vertices is output.

We now analyze this algorithm. We begin with a two simple observations.

Lemma 1.1. *Once two vertices are merged, the algorithm never outputs a cut separating them.*

Proof. When an edge is contracted, we are effectively deleting that edge from our output. So, a cut separating the two vertices can never be output. ■

Lemma 1.2. *Any cut set of an intermediate graph is a cut set of the original graph. Hence, the algorithm always outputs some cut set of the graph.*

Proof. This is trivially true; given a cut set of some intermediate graph, we can get a cut set of the original graph by partitioning the vertices in the original graph according to the partition in the intermediate graph. ■

Next, compute the probability that a minimum cut is output. To do this, fix some minimum cut C in G. In what event is C output by our algorithm? C is output only if the algorithm never contracts an edge belonging to C. We use this crucial observation to prove the following theorem.

Theorem 1.3. *Let C be a minimum cut of G. Then the cut C is output by the algorithm with probability atleast $\dfrac{2}{n(n-1)}$, where n is the number of vertices.*

Proof. Let $|C| = k$. Note that the total number of iterations in the algorithm is $n-2$.

Let E_i be the event that the edge contracted in iteration i is not in C. Let $F_i = \bigcap_{j=1}^{i} E_i$, i.e F_i is the event that no edge of C has been contracted till iteration i. We need to give a lower bound on F_{n-2}.

First, consider the event $E_1 = F_1$. Since the minimum cut has size k, each vertex has degree atleast k. Hence,

$$|E| \geq \frac{nk}{2}$$

So, the probability that no edge of C is chosen in the first iteration is

$$\left(1 - \frac{2k}{nk}\right) = \frac{n-2}{n}$$

Now, observe that we have

$$\mathbf{P}\left[F_i\right] = \mathbf{P}\left[E_i \mid F_{i-1}\right]$$

In the ith stage, there are $n-i+1$ vertices. If we assume the event F_{i-1}, then clearly the size of the min cut is still k, and hence each vertex has degree atleast k. So, the number of edges at the ith stage is bounded below by

$$\frac{k(n-i+1)}{2}$$

So, it follows that

$$\mathbf{P}\left[F_i\right] = \mathbf{P}\left[E_i \mid F_{i-1}\right] \geq \left(1 - \frac{2k}{k(n-i+1)}\right) = \frac{n-i-1}{(n-i+1)}$$

So, it follows that

$$\begin{aligned}
\mathbf{P}\left[F_{n-2}\right] &= \mathbf{P}\left[E_{n-2} \mid F_{n-1}\right] \\
&\geq \frac{1}{3}\mathbf{P}\left[F_{n-1}\right] \\
&\quad \vdots \\
&\geq \left(\frac{1}{3}\right)\left(\frac{2}{4}\right)\left(\frac{3}{5}\right)\cdots\left(\frac{n-3}{n-1}\right)\left(\frac{n-2}{n}\right) \\
&= \frac{2}{n(n-1)}
\end{aligned}$$

This proves the claim. ■

Corollary 1.3.1. *If G is any graph with n vertices, then the number of minimum cuts in the graph is upper bounded by $\binom{n}{2}$.*

Proof. Let $C_1, ..., C_M$ be all the min cuts in G. We want to bouny the previous claim, we see that

$$\mathbf{P}\left[C_i \text{ survives}\right] \geq \frac{2}{n(n-1)} = \frac{1}{\binom{n}{2}}$$

The above is true for each i. Also, note that the events $\{C_i \text{ survives}\}$ and $\{C_j \text{ survives}\}$ are disjoint, if $i \neq j$. So, we see that

$$\mathbf{P}\left[\text{some min cut survives}\right] = \sum_{i=1}^{M} \mathbf{P}\left[C_i \text{ survives}\right] \geq \frac{M}{\binom{n}{2}}$$

Since a probability can't be greater than 1, the claim follows. ■

A detailed discussion of this problem can be found at this link: https://www.cs.cmu.edu/~avrim/451f13/lectures/lect0905.pdf.

1.4. **Coupon Collector's Problem.** In this problem, there are n different types of coupons. In each trial, we get exactly one coupon. The question is to figure out what is the number of trials required to get all of the coupons. It is assumed that each trial gives us coupons with uniform probability (i.e each coupon has the same probability of being picked up).

Let X denote the number of trials required to obtain all the different n coupons. Similarly, let X_i denote the number of trials to get the ith distinct coupon, provided that we already have $i-1$ distinct coupons. Clearly, we have the following.

$$X = \sum_{i=1}^{n} X_i$$

Let us figure out what $\mathbf{E}\left[X\right]$ is.

Suppose currently we have $i-1$ distinct coupons. Let p_i be the probability of finding the ith distinct coupon in the next trial. Clearly,

$$p_i = \frac{n-i+1}{n}$$

So,

$$\mathbf{P}\left[X_i = k\right] = \left(\frac{i-1}{n}\right)^{k-1}\left(\frac{n-i+1}{n}\right) = (1-p_i)^{k-1}p_i$$

So, X_i is a geometric random variable. So,

$$\begin{aligned}\mathbf{E}[X] &= \sum_{i=1}^{n} \mathbf{E}[X_i] \\ &= \sum_{i=1}^{n} \frac{1}{p_i} \\ &= n \sum_{i=1}^{n} \frac{1}{n-i+1} \\ &= n \sum_{i=1}^{n} \frac{1}{i} \\ &= n(\ln n + \Theta(1)) \\ &= \Theta(n \ln n)\end{aligned}$$

Hence, we require around $n \ln n$ trials to get all the coupons.

1.4.1. *How likely do we need a large number of trials?* Even though the above computation gave us an average number of trials required, nothing guarantees that the number of trials needed is not huge. So now, we will try to bound the likelihood of requiring a large number of trials to get all the coupons. To do this, we will use *Markov's* and *Chebyshev's inequalities*.

Again, let X be the same random variable as above. For any $c > 0$, we have by Markov's Inequality

$$\mathbf{P}[X \geq c\mathbf{E}[X]] \leq \frac{\mathbf{E}[X]}{c\mathbf{E}[X]} = \frac{1}{c}$$

So, if we put $c = 2$, we see that the chance that we need more than $2nH_n$ trials (here H_n is the nth harmonic number) is no more than half.

Next, let us try to apply Chebyshev's inequality to X. To do that, we need to find $\mathbf{Var}(X)$. Observe that the random variables X_i are all independent. So,

$$\mathbf{Var}(X) = \sum_{i=1}^{n} \mathbf{Var}(X_i)$$

Now, X_i's are all geometric random variables, and we know that the variance of a geometry random variable Y with parameter p is

$$\mathbf{Var}(Y) = \frac{1-p}{p^2} \leq \frac{1}{p^2}$$

So, we see that

$$\mathbf{Var}(X) = \sum_{i=1}^{n} \mathbf{Var}(X_i) \leq \sum_{i=1}^{n} \left(\frac{n}{n-i+1}\right)^2 = n^2 \sum_{i=1}^{n} \frac{1}{i^2} \leq \frac{\pi^2 n^2}{6}$$

So by Chebyshev's Inequality we see that for any $c > 0$,

$$\mathbf{P}[|X - nH_n| \geq c] \leq \frac{\pi^2 n^2}{6c^2}$$

In particular, for $c = nH_n$, we have

$$\mathbf{P}[|X - nH_n| \geq nH_n] = O\left(\frac{1}{\ln^2 n}\right)$$

This is a fairly weak bound. Using a trivial union bound, we can do even better.

Consider the probability of not obtaining the ith distinct coupon even after $n \ln n + cn$ steps. This probability is simply the following.

$$\left(\frac{i-1}{n}\right)^{n(\ln n+c)} \leq \left(1-\frac{1}{n}\right)^{n(\ln n+c)} \leq e^{-(\ln n+c)} = \frac{1}{ne^c}$$

By a simple union bound, we see that the probability that some coupon has not been obtained even after $n \ln n + cn$ steps is only e^{-c}. This is much better than what we obtained via Chebyshev's or Markov's inequalities.

1.5. **Parallel Algorithm for Bipartite Perfect Matchings.** In this section, we will assume the following: the determinant of an $n \times n$ matrix can be computed in parallel with $O(n^{3.5})$ processors and $O(\log^2 n)$ time. Also, we will be assuming the knowledge of the *polynomial identity testing problem* and its application to bipartite perfect matchings.

First, let us assume that we have an oracle A that, given a graph G, tells us whether A has a perfect matching or not. Also, assume that the oracle uses a parallel algorithm with $O(n)$ processors and $O(\log^2 n)$ time.

First, assume that we are dealing with only those graphs which either don't have perfect matchings, or have *unique* perfect matchings. Let G be such a graph. The following lemma is easy to see.

Lemma 1.4. *Let G be a graph with a unique perfect matching. Then, the edge (u, v) belongs to the perfect matching iff. $G \setminus \{u, v\}$ has a perfect matching.*

Proof. First, suppose (u, v) is an edge of the perfect matching. Then clearly, $G \setminus \{u, v\}$ has a perfect matching as well.

Conversely, suppose $G \setminus \{u, v\}$ has a perfect matching. Since G has a unique perfect matching, it must be the case that (u, v) belongs to the perfect matching. ■

The above claim gives us a very simple parallel algorithm. We will use n^2 processors (one for every edge, since there at potentially $O(n^2)$ edges). The processors are denoted $P_1, ..., P_{n^2}$. Suppose the edges are $e_1, ..., e_{n^2}$. The following is the description of the algorithm for processor P_i. Also, suppose the oracle A returns **true** if the input graph has a perfect matching, and **false** otherwise. Clearly, this is a valid parallel algorithm

Algorithm 1 Algorithm Description for Processor i

```
Input: A graph G which has a unique perfect matching (if it has one at all), oracle A
(u, v) ← e_i
if A(G \ {u, v}) = true then
    Add e_i to the perfect matching
else
    Don't add e_i to the perfect matching
end if
```

to find a perfect matching of such inputs G.

Now, let us deal with the cases where G need not have a unique perfect matching. In this case, we take the following strategy. The idea is to get a *minimum weight* perfect matching by assigning weights to edges randomly.

(1) If e is an edge, assign a weight to e, where the weight is picked uniformly at random from the set $\{1, 2, ..., 2|E|\}$.

(2) What is the probability that the minimum weight perfect matching is unique? Clearly, we still can't guarantee the uniqueness of the minimum weight matching.
(3) How do the processors find the minimum weight perfect matching in parallel?

Let us consider point number (3) first. Suppose we have been lucky enough to bypass point number (2) and get a unique minimum weight perfect matching by assigning weights randomly. Let this matching be M. How do the processors find it?

Define a matrix D as follows (the definition is very similar to the symbolic adjacency matrix/Tutte matrix).

$$D_{ij} = \begin{cases} 2^{W_{ij}} & , \quad \text{if } (i,j) \text{ is an edge} \\ 0 & , \quad \text{otherwise} \end{cases}$$

where W_{ij} is the weight of the edge (i,j). The question now is: if the minimum weight matching is unique, then is $\det(D) = 0$? The answer is no, as we prove below.

Lemma 1.5. *Let D be the matrix as above. If G the minimum weight matching in G is unique, then* $\det(D) \neq 0$.

Proof. We know that

$$\det(D) = \sum_{\sigma} \epsilon(\sigma) \prod_{i=1}^{n} D_{i,\sigma(i)}$$

Clearly, the term corresponding to σ is non-zero if and only if σ is a perfect matching. In that case, the term corresponding to sigma is the following.

$$2^{\sum_{i=1}^{n} W_{i,\sigma(i)}}$$

Suppose $W_{\min}$ is the weight of the minimum weight matching. Since this matching is unique, we see that $\det(D)$ is of the following form.

$$\det(D) = \pm 2^{W_{\min}} \pm C(2^{W_{\min}+1})$$

where C is some integer. So, the determinant is non-zero. ■

Continuing this forward, we have the following lemma.

Lemma 1.6. *Let D and G be defined as above. Then the following are true.*

(1) $\det(D) = 0$ *if G has no perfect matching.*
(2) *If M is a unique minimum weight perfect matching with weight W_0 then W_0 is the largest power of 2 that divides* $\det(D)$.
(3) *If minimum weight perfect matching is not unique then either* $\det(D) = 0$ *or the largest power of 2 that divides the* $\det(D)$ *is* $\geq W_0$.

Proof. (1) and (2) were more or less proved in the previous lemma. Let us prove (3). Suppose the weight of the minimum weight perfect matching is W_0. In addition, suppose that $\det(D) \neq 0$. Then, two cases are possible:

- The terms corresponding to weight 2^{W_0} all cancel out. In this case, since $\det(D) \neq 0$, higher order terms are remaining, and clearly each higher order term is $> 2^{W_0}$.
- If the terms corresponding to weight 2^{W_0} don't cancel out, then the highest power of 2 dividing $\det(D)$ is clearly W_0.

■

So, provided that we have a unique minimum weight perfect matching, we can use the following algorithm. We describe the role of processor P_{ij}, for $1 \leq i, j \leq n$. So, given a graph G (with assigned weights to edges) such that G has atmost one minimum weight perfect matching, P_{ij} processes the edge (i, j) and returns **true** if this edge is a part of the minimum weight perfect matching, and otherwise returns **false**. We now

Algorithm 2 Case of Unique Minimum Weight Perfect Matchings, Role of Processor P_i

Input: G (has atmost one minimum weight perfect matching), W_{ij} (weights), D
if $\det(D) = 0$ **then**
  There is no perfect matching
end if
$Q_0 \leftarrow$ Largest power of 2 dividing $\det(D)$
$M_{ij} \leftarrow$ Matrix obtained by removing ith row and jth column in D
if $\det(M_{ij} = 0)$ **then**
  return false
else
  $Q_{ij} \leftarrow$ Largest power of 2 dividing $\det(M_{ij})$
  if $Q_{ij} + W_{ij} = Q_0$ **then**
    return true
  else
    return false
  end if
end if

prove the correctness of the algorithm.

Proposition 1.7. *If the input G in the above algorithm has a unique minimum weight perfect matching, then processor P_{ij} returns* **true** *if and only if the edge (i, j) belongs to the minimum weight perfect matching.*

Proof. Since G has a unique minimum weight perfect matching, $\det(D) \neq 0$, which we know by **Lemma** 1.5. Now, consider the processor P_{ij}, where (i, j) is an edge of the graph. Let M_{ij} be the matrix obtained by removing the ith row and the jth column in D. Clearly, M_{ij} then describes the bipartite graph $G \setminus \{i, j\}$. If (i, j) is not part of *any* perfect matching, then clearly $\det(M_{ij}) = 0$ (because if it was non-zero, it would imply that $G \setminus \{i, j\}$ has a perfect matching, which would then imply that (i, j) is part of some perfect matching). In that case, the algorithm returns **false** as required.

So suppose (i, j) is part of *some* perfect matching. Then, either (i, j) is part of the unique minimum weight perfect matching, or it is part of some perfect matching whose weight is strictly larger than this minimum weight matching. We deal with these two cases separately.

(1) In the first caes, (i, j) is an edge of the unique minimum weight perfect matching of G. Clearly, in that case, the graph $G \setminus \{i, j\}$ has a unique minimum weight perfect matching as well. Again, by **Lemma** 1.5, we see that $\det(M_{ij}) \neq 0$. Infact, by **Lemma** 1.6, we see that the largest power of 2 that divides $\det(M_{ij})$ is infact that weight of the unique minimum weight matching of $G \setminus \{i, j\}$. So, it clearly follows that
$$Q_{ij} + W_{ij} = Q_0$$
and hence the algorithm will return **true** as desired.

(2) In the second case, (i, j) is an edge of a perfect matching whose weight is *strictly* greater than Q_0, the weight of the unique minimum weight perfect matching. In that case, by invoking **Lemma** 1.6, we see that

$$Q_{ij} + W_{ij} > Q_0$$

and hence the algorithm will return **false**.

This completes the proof. ■

So, as a consequence of **Proposition** parallelPMCorrectness, our algorithm will be correct given that the randomized weights lead to a unique minimum weight perfect matching. We will now find the probability that these weights lead to a minimum weight perfect matching. To do this, we will prove the so called *Isolation Lemma*.

Theorem 1.8 (**Isolation Lemma**)**.** *Let S be any set with $|S| = m$, and let $S_1, ..., S_k \subseteq S$ be subsets of S. Suppose each $x \in S$ is assigned a weight $w(x) \in \{1, 2, ..., n\}$ uniformly at random, where n is some natural number.*

$$\mathbf{P}\left[\exists \text{ a unique minimum weight set}\right] \geq 1 - \frac{m}{n}$$

Proof. Suppose there are two sets S_i, S_j of minimum weight. Since the sets are distinct, wlog there is some element $e \in S_i$ such that $e \notin S_j$. We call such an element a *tied element*.

Now, let $e \in S$ be any element. We will bound the probability of e being a tied element. Suppose the weights of all elements other than e have been fixed. Now, define the following.

$$A^- := \{S_i \mid e \notin S_i\}$$
$$A^+ := \{S_i \mid e \in S_i\}$$

Next, define the following.

$$W^- := \min_{S_i \in A^-} w(S_i)$$
$$W^+ := \min_{S_j \in A^+} w(S_j)$$

where above, $w(S_k)$ is the weight of the set S_k (the weight of a set is defined as the sum of the weights of its elements). Also, in the calculation of W^+, the element e is not included in the weight calculation (since it's weight has not been defined yet).

Now, suppose the weight of e is picked uniformly at random from the set $\{1, ..., N\}$. Oberve that when e is picked, the minimum weight of any set in A^+ will be $W + w(e)$. Also, for e to be a tied element, the following equation has to be true.

$$W^+ + w(e) = W^-$$

which implies

$$w(e) = W^- - W^+$$

So, the probability that e is a tied element is equal to the probability that $w(e) = W^- - W^+$. Clearly,

$$\mathop{\mathbf{P}}_{w(e)}\left[w(e) = W^- - W^+\right] \leq \frac{1}{n}$$

Note that the probability is *not equal to* $\frac{1}{N}$ because $W^- - W^+$ could be outside the range $\{1, ..., n\}$.

Now, observe that the minimum weight set is *not unique* if and only if atleast some $x \in S$ is a tied element. By a simple union bound, we therefore see that

$$\mathbf{P}\left[\text{Minimum weight set is not unique}\right] \leq \frac{m}{n}$$

and hence the claim follows. ■

Corollary 1.8.1. *If the weights to edges in G are assigned uniformly at random from the set $\{1, ..., 2|E|\}$, then with probability atleast 0.5, the minimum weight perfect matching is unique.*

Proof. This is just an application of **Theorem** 1.8 (**Isolation Lemma**); here the set S is the set of all edges, and the subsets are the perfect matchings in the graph G. ■

Remark 1.8.1. Ofcourse, if we want a higher accuracy, we can replace the number $2|E|$ with $c|E|$ where c is a large constant.

1.6. **Network Reliability.** In this section, we will study the *network reliability problem*. We are given a connected, undirected graph $G = (V, E)$. Each edge of G *fails* independently with probability $p \in (0, 1)$. We want to estimate p_{fail}, i.e the probability that the graph gets disconnected. The following bound is trivially true.

Proposition 1.9. *If the minimum cut size in G is c, then*

$$p_{\text{fail}} \geq p^c$$

Proof. Clearly, the graph gets disconnected if all the edges in the min cut fail. The probability that all the edges of the min cut fail is p^c. Hence the claim follows. ■

Our goal is to find a *fully polynomial-time randomized approximation scheme* (FPRAS) for this problem. We now define what this means for thte network reliability problem.

Definition 1.1. A *fully polynomial-time randomized approximation scheme* (FPRAS) is an algorithm such that on input (G, p, ϵ), outputs a value Z such that

$$\mathbf{P}\left[(1-\epsilon)p_{\text{fail}} \leq Z \leq (1+\epsilon)p_{\text{fail}}\right] \geq \frac{3}{4}$$

and runs in time poly $\left(n, \frac{1}{\epsilon}\right)$.

To solve this problem, we will consider two cases.

(1) In the first case, we will have

$$p^c \geq \frac{1}{n^4}$$

In this case, we will use the *unbiased estimator approach.*

(2) In the second case, we have

$$p^c < \frac{1}{n^4}$$

In this case, we will use a reduction to the DNF counting problem.

Let us analyze these two cases.

1.6.1. *Case 1.* In this case, we will follow the *unbiased estimator approach.* We will do the following: remove each edge of G independently with probability p and check if G is disconnected. Repeat this experiment t times.

For the ith experiment, we define the random variable X_i as follows.

$$X_i = \begin{cases} 1 & , \quad \text{if } G \text{ gets disconnected in the } i\text{th experiment} \\ 0 & , \quad \text{otherwise} \end{cases}$$

Then, our estimate of p_{fail} is the following quantity.

$$X = \frac{\sum_{i=1}^{t} X_i}{t}$$

Now, observe that

$$\mathbf{E}\left[X_i\right] = p_{\text{fail}} := \mu$$

Also,

$$\mathbf{Var}\left(X_i\right) = \mathbf{E}\left[X_i^2\right] - \left(\mathbf{E}\left[X_i\right]\right)^2 = \mu - \mu^2 \leq \mu$$

Because all the X_i are independent, we see that

$$\mathbf{Var}\left(X\right) = \frac{\mathbf{Var}\left(X_i\right)}{t} = \frac{\mu - \mu^2}{t}$$

and that

$$\mathbf{E}\left[X\right] = \frac{t\mu}{t} = \mu$$

By **Chebyshev's Inequality**, we then see that

$$\mathbf{P}\left[|X - \mu| \geq \epsilon\mu\right] \leq \frac{\mathbf{Var}\left(X\right)}{\epsilon^2\mu^2} \leq \frac{\mu}{t\epsilon^2\mu^2} = \frac{1}{t\epsilon^2\mu}$$

For an FPRAS, we want

$$\frac{1}{t\epsilon^2\mu} \leq \frac{1}{4}$$

which gives us the bound

$$t \geq \frac{4}{\epsilon^2\mu}$$

Now, because we assumed that $p_{\text{fail}} = \mu = \Omega(1/n^4)$, we clearly see that $t = O(n^4/\epsilon^2)$ works.

1.6.2. *Case 2.* Now consider the second case, i.e

$$p^c < \frac{1}{n^4}$$

The idea here is that most of the failure probability is due to smaller cuts (say of size atmost αc), and that we can ignore large cuts. The goal is to pick α such that

$$\mathbf{P}\left[\text{Some cut} \geq \alpha c \text{ fails}\right] \leq \epsilon p_{\text{fail}}$$

Also, we face the question: how to bound the error for *α-min cuts* (i.e cuts with size atmost αc).

Next, we claim that there are atmost $n^{2\alpha}$ α-min cuts in any graph with n vertices. For a proof, refer to **Claim 11.3** in this link: https://people.eecs.berkeley.edu/~sinclair/cs271/n11.pdf.

Now, suppose $E_1, ..., E_t$ are the α-min cuts. For any edge e, define the random variable X_e as follows.

$$X_e = \begin{cases} 1 & , \quad \text{if } e \text{ fails} \\ 0 & , \quad \text{otherwise} \end{cases}$$

Then, define

$$C_i = \bigwedge_{e \in E_i} X_e$$

So, C_i will be 1 if and only if all the edges in the cut E_i fail. Now, let

$$\Phi = \bigvee_{i=1}^{t} C_i$$

Clearly, Φ is a disjunction. So, clearly we see that

$$\mathbf{P}\left[\text{Some } \alpha\text{-min cut fails}\right] = \text{Weighted Average of solutions to } \Phi$$

We will soon see an FPRAS for DNF counting (in the next section), i.e computing the right hand side above.

Now, let us determine what α value to set. Let $C_1, ..., C_N$ be cuts of size $\geq \alpha c$, and let $c_i = |C_i|$. Arrange $c_1, c_2, ..., c_N$ in non-decreasing order, i.e let

$$c_1 \leq c_2 \leq ... \leq c_N$$

Now, consider the first $n^{2\alpha}$ cuts (if there are fewer such cuts, we get an even better bound). Clearly,

$$\mathbf{P}\left[C_i \text{ fails}\right] \leq p^{\alpha c}$$

and so

$$\mathbf{P}\left[\bigvee_{i=1}^{n^{2\alpha}} C_i \text{ fails}\right] \leq n^{2\alpha} p^{\alpha c}$$

Now, because we know that

$$p^c < \frac{1}{n^4}$$

this implies that for some $\delta > 2$

$$p^c \leq n^{-(2+\delta)}$$

So, this implies that

$$\mathbf{P}\left[\bigvee_{i=1}^{n^{2\alpha}} C_i \text{ fails}\right] \leq n^{2\alpha} p^{\alpha c} \leq n^{2\alpha} n^{(-2-\delta)\alpha} = n^{-\delta\alpha}$$

Now, for any β, we know that there are atmost $n^{2\beta}$ cuts of size $\leq \beta c$. Now, again, consider the list $C_1, ..., C_N$. For $k = n^{2\beta}$, we clearly have that $c_k \geq \beta c$ Now, let $\beta = \frac{1}{2}\frac{\log k}{\log n}$ (which means $k = n^{2\beta}$). Then,

$$\mathbf{P}\left[C_k \text{ fails}\right] \leq p^{\beta c} \leq p^{\frac{c}{2}\frac{\log k}{\log n}} \leq p^{-(2+\delta)(\log_n k)/2} = k^{-(1+\delta/2)}$$

So, it follows that

$$\mathbf{P}\left[\bigvee_{i > n^{2\alpha}}\right] \leq \sum_{k > n^{2\alpha}} k^{-(1+\delta/2)} \leq \int_{n^{2\alpha}}^{\infty} x^{-(1+\delta/2)} dx = \frac{2}{\delta} n^{-\delta\alpha} < n^{-\delta\alpha}$$

Putting the above two bounds together, we see that

$$\mathbf{P}\left[\text{Some cut of size} \geq \alpha c \text{ fails}\right] \leq 2n^{-\delta\alpha}$$

Finally, we want

$$2n^{-\delta\alpha} \leq \epsilon p^c \leq \epsilon p_{\text{fail}}$$

and so we can choose

$$\alpha = 2 + \frac{1}{2}\log_n(2/\epsilon)$$

1.7. **DNF Counting.** In this section, we will devise an FPRAS for the *DNF Counting* problem. Let us first describe what the problem is about.

We have a logical formula Φ in *Disjunctive Normal Form* (DNF), i.e the formula can be written as

$$\Phi = T_1 \vee T_2 \vee \cdot \vee T_m$$

where each T_i is a *conjunction*, i.e it is an and over a certain number of literals. For example, Φ could be the following.

$$\Phi = (x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_3 \wedge x_5)$$

We will make the following assumptions.

- The formula Φ has m terms and n variables.
- No term T_i contains both x and $\neg x$, where x is some variable. If such a T_i exists, it will never be satisfiable and can be safely removed from Φ.

The second assumption above implies that all the terms T_i in Φ have exactly one satisfying assignment; to see this, note that to satisfy a conjunction, all of its literals must be 1, and hence there is a unique assignment of *variables in* T_i that satisfies T_i.

Now, define the set S_i as follows.

$$S_i := \text{Set of assignments of all variables satisfying the term } T_i$$

The cardinality of S_i can be easily counted.

$$|S_i| = 2^{n-q_i}$$

where q_i is the number of variables in S_i. This is straightforward: there is a unique assignment of variables in S_i that satisfies S_i; all the other variables of Φ can be set independently to either 0 or 1. Our goal is to calculate the following cardinality.

$$c(\Phi) = \left|\bigcup_{i=1}^{m} S_i\right|$$

Clearly, the above cardinality is the total number of satisfying assignments of Φ. Hence the problem is called *DNF Counting*. We will assume that each variable has equal probability for being 0 or 1, i.e $p = 1/2$ (recall the Network Reliability problem setting). We will see how to solve this problem when p is any number after solving the problem for $p = 1/2$.

1.7.1. *The Naive Unbiased Estimator.* To estimate the value of $|\bigcup_{i=1}^{m} S_i|$, we will do the following: pick a random assignment, and check if it satisfies the formula Φ. Repeat the experiment a number of times, and average out the result. Formally, suppose T is the number of times we repeat the experiment. We will maintain a counter X which is initialised to zero. Every time our random sample is a satisfying assignment for Φ, we increment X by 1. Then, after T iterations, we output the number

$$Y = \frac{X}{T} 2^n$$

and we hope that this is a good approximation to the number of satisfying assignments (note that 2^n is the total number of possible assignments).

Now, for this method to be an FPRAS, it can be checked that we need the quantity $|\bigcup_{i=1}^{m} S_i| / 2^n$ to be polynomial in n. This is asking a lot, because this will mean that the number of satisfying assignments for Φ is in the order of 2^n, which means there are a lot of satisfying assignments. We will next cover a much better algorithm for this, which works in all cases.

1.7.2. *Karp-Luby Algorithm.* In this section, we will see how to sample uniformly at random from the set $c(\Phi)$, i.e we will sample *only* satisfying assignments. This is not obvious, but there is a nice way to do this. Define the set S as follows.

$$S := \{(a, i) \mid \text{Assignment } a \text{ satisfies term } T_i\}$$

We immediately see that

$$|S| = \sum_{i=1}^{m} |S_i|$$

and hence $|S|$ is easy to compute, since we we can compute each $|S_i|$ easily. Let

$$r = c(\Phi) = \left| \bigcup_{i=1}^{m} S_i \right|$$

and we want to approximate r. Let the satisying assignments of Φ be $a_1, ..., a_r$. Call the pair (a_i, j) to be *special* if j is the *first term* satisfied by a_i. Clearly,

$$\#\text{ of special pairs} = r$$

Since we wanted to approximate r, it is enough to approximate the number of special pairs. To do this, we will use an unbiased estimator wherein we will sample points uniformly at random from S (in a moment we will see how to do this). See the pseudocode of the algorithm on the next page.

Algorithm 3 Karp-Luby Algorithm

$X \leftarrow 0$
for $j = 1$ to T **do**
  Pick (a, i) uniformly at random from S ▷ We will show how to do this
  if (a, i) is special **then**
    $X \leftarrow X + 1$
  end if
end for
Output $\frac{X}{T} \cdot |S|$ ▷ Our estimate of the number of special pairs

Assume that we know how to pick (a, i) from the set S uniformly at random. Let X_i be the random variable which is 1 if the pair picked at the ith step is special, and 0 otherwise. So, for each $1 \leq i \leq T$, we see that

$$\mu = \mathbf{E}[X_i] = \mathbf{P}[X_i = 1] = \frac{\#\text{ of special pairs}}{|S|} = \frac{r}{|S|}$$

Suppose

$$Y = \frac{X}{T}|S|$$

Then, we see that

$$\mathbf{E}[Y] = \frac{\mathbf{E}[X] \cdot |S|}{T} = \frac{T\mu}{T} \cdot |S| = \mu|S| = r$$

Also, because the X_is are all independent, we see that

$$\mathbf{Var}(Y) = \frac{\mathbf{Var}(X) \cdot |S|^2}{T^2} = \frac{\sum_{i=1}^{T} \mathbf{Var}(X_i) \cdot |S|^2}{T^2} = \frac{T(\mu - \mu^2) \cdot |S|^2}{T^2} \leq \frac{\mu|S|^2}{T} = \frac{r|S|}{T}$$

So, by **Chebyshev's Inequality**, we see that

$$\mathbf{P}[|Y - r| \geq \epsilon r] \leq \frac{\mathbf{Var}(Y)}{\epsilon^2 r^2} \leq \frac{r|S|}{T\epsilon^2 r^2} \leq \frac{r^2 m}{T\epsilon^2 r^2} = \frac{m}{T\epsilon^2}$$

where in the above step, we have used the trivial fact that $|S| \leq rm$. So, for an FPRAS, we need

$$\frac{m}{T\epsilon^2} \leq \frac{1}{4}$$

which gives us the bound

$$T \geq \frac{4m}{\epsilon^2}$$

Clearly, the above algorithm is an FPRAS, because with probability $\geq 3/4$, Y is in the interval $[(1 - \epsilon)r, (1 + \epsilon)r]$.

Finally, we will now show how to sample points (a, i) uniformly at random from the set S. This will be done as follows.

(1) Pick term i (T_i) with probability

$$\frac{|S_i|}{\sum_{i=1}^{m} |S_i|}$$

i.e we are picking a term with probability proportional to it's size.

(2) Set variables in the term T_i so that they satisfy T_i (as we have noted before, there is precisely one such assignment of these variables), and pick values for the rest of the variables uniformly at random.

Let us show that this is really sampling points from S using the uniform distribution on S. But this is clear; if $(a, i) \in S$, then clearly the probability of picking (a, i) is simply the probability of picking term T_i multiplied by the probability of picking the values of a on the rest of the variables (which as we know is just $1/2^{n-q_i}$), which is equal to

$$\frac{|S_i|}{\sum_{i=1}^{m} |S_i|} \frac{1}{2^{n-q_i}} = \frac{1}{\sum_{i=1}^{m} |S_i|} = \frac{1}{|S|}$$

and hence we are done.

1.8. **Improving success for 2-sided errors: Medians of Means.** In this section, we will consider again the FPRAS we devised for the DNF Counting problem in the last section. We will see how to efficiently improve the error probability, even though we have a two-sided error in this case. The method we will use is called the *median of means* method.

Recall the random variable Y from the last section. We showed that

$$\mathbf{P}\left[|Y - r| \geq \epsilon r\right] \leq \frac{m}{T\epsilon^2}$$

Let $\delta > 0$ be the error that we want. So, we want

$$\frac{m}{T\epsilon^2} < \delta$$

which gives us the following bound on T.

$$T > \frac{m}{\delta\epsilon^2}$$

So, we will need $O(1/\delta)$ steps to get a small error. If $\delta = 1/2^n$ (exponentially small error), we will need $O(2^n)$ steps to get this small error; this is clearly not fast. We want to do better.

To optimize our solution, we will use the so called *median of means method*. The idea is simple: we will run the estimator for $2s + 1$ steps, where s will be determined in a moment. Then, we will output the *median* of the $2s + 1$ outputs that we get. Let O_i be the outputs for $1 \leq i \leq 2s + 1$, and suppose they are ordered, i.e

$$O_1 \leq O_2 \leq \cdots \leq O_{2s+1}$$

Our output will be O_{s+1}. Now, we will estimate

$$\mathbf{P}\left[O_{s+1} \notin [(1-\epsilon)r, (1+\epsilon)r]\right]$$

Clearly, the event $O_{s+1} \notin [(1-\epsilon)r, (1+\epsilon)r]$ is a subset of the event that $s+1$ outcomes are not in the given interval; this is true because if O_{s+1} is not in the given interval, then either $\{O_1, ..., O_s\}$ are not in the interval, or $\{O_{s+2}, ..., O_{2s+1}\}$ are not in the given

interval. So, using the independence of the $2s+1$ trials, we have the following.

$$\begin{aligned}\mathbf{P}\left[O_{s+1} \notin [(1-\epsilon)r, (1+\epsilon)r]\right] &\leq \mathbf{P}\left[\geq s+1 \text{ values are outside the interval}\right] \\ &\leq \sum_{i \geq s+1} \binom{2s+1}{i} \left(\frac{1}{4}\right)^{i} \left(\frac{3}{4}\right)^{2s+1-i} \\ &\leq \left(\frac{1}{4}\right)^{s+1} \left(\frac{3}{4}\right)^{s} \sum_{i \geq s+1} \binom{2s+1}{i} \\ &\leq \left(\frac{1}{4}\right)^{s+1} \left(\frac{3}{4}\right)^{s} 2^{2s+1} \\ &\leq \left(\frac{3}{4}\right)^{s}\end{aligned}$$

Now, we want

$$\left(\frac{3}{4}\right)^{s} < \delta$$

which gives us the bound

$$s \log(3/4) < \log(\delta)$$

which gives us

$$s \log(4/3) > \log(1/\delta)$$

which gives us

$$s > \frac{\log(1/\delta)}{\log(4/3)}$$

and hence we now need only $O(\log(1/\delta))$ steps to get a small error bound.

2. Probabilistic Method

2.1. **An example: k-SAT.** In this section, we will see how powerful and simple the *probabilistic method* can be to show the existence of certain structures.

Consider a k-SAT formula ϕ, i.e

$$\phi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$$

where each C_i is a *disjunction* of k literals. Also, suppose there are k variables. If we take a random assignment a to the variables, we see that

$$\mathbf{P}\left[a \text{ does not satisfy } C_j\right] = \frac{1}{2^k}$$

So, using a simple union bound, we see that

$$\mathbf{P}\left[a \text{ does not satisfy } \phi\right] \leq \frac{m}{2^k}$$

If $m < 2^k$, then we see that the above probability is strictly less than 1. In that case, we see that

$$\mathbf{P}\left[a \text{ satisfies } \phi\right] > 0$$

and hence, since the probability is non-zero, there is some satisfying assignment for ϕ. This is known as the *probabilistic argument/probabilistic method*.

2.2. **Lovasz Local Lemma.** In this section, we will state and prove a very useful tool to be applied in the probabilistic method.

Theorem 2.1 (**Lovasz Local Lemma**). *Let $E_1, ..., E_n$ be a set of bad events associated to a random experiment, such that the following hold.*

(1) $\mathbf{P}[E_i] \leq p < 1$ *for all* i.
(2) *Each* E_i *depends on atmost* d *other* E_j*s.*
(3) $p(d+1) \leq 1/e$, *where* e *is Euler's constant.*

Then,

$$\mathbf{P}\left[\bigcap_{i=1}^{n} \overline{E_i}\right] > 0$$

Remark 2.1.1. Condition (3) above can also be replaced by the condition $4dp \leq 1$; the proof of that statement is given in the book.

Proof. First, using induction, we will show the following: for any $S \subseteq [n]$ and $i \in [n]$, we have

$$\mathbf{P}\left[E_i \mid \bigcap_{j \in S} \overline{E_j}\right] \leq \frac{1}{d+1} \tag{2.1}$$

Having proven this, the statement of the theorem will follows; to see this, using the above inequality, we have the following.

$$\begin{aligned}
\mathbf{P}\left[\bigcap_{i=1}^{n} \overline{E_i}\right] &= \mathbf{P}\left[\overline{E_1} \mid \bigcap_{i=2}^{n} \overline{E_i}\right] \cdot \mathbf{P}\left[\bigcap_{i=2}^{n} \overline{E_i}\right] \\
&\vdots \\
&= \prod_{i=1}^{n-1} \mathbf{P}\left[\overline{E_i} \mid \bigcap_{j=i+1}^{n} \overline{E_j}\right] \cdot \mathbf{P}\left[\overline{E_n}\right] \\
&\geq \left(1 - \frac{1}{d+1}\right)^{n-1} \cdot (1-p) \\
&\geq \left(1 - \frac{1}{d+1}\right)^{n} \\
&> 0
\end{aligned}$$

where in the second last step, we used the fact that

$$1 - p \geq 1 - \frac{1}{d+1}$$

So, it is enough to prove inequality (2.1). We will do this now, and we will prove this using induction on $|S|$. For the base case, suppose $|S| = 0$. In that case, the inequality is trivial, since

$$\mathbf{P}[E_i] = p \leq \frac{1}{(d+1)e} < \frac{1}{d+1}$$

So, assume that the statement holds for all sets S with $|S| < m$, and we will show that statement for $|S| = m$.

Now, write $S = S_1 \cup S_2$, where S_2 consists of those j for which E_j is independent of E_i. Now, we have the following.

$$\mathbf{P}\left[E_i \mid \bigcap_{j\in S} \overline{E_j}\right] = \frac{\mathbf{P}\left[E_i \cap (\bigcap_{j\in S_1} \overline{E_j}) \mid \bigcap_{j\in S_2} \overline{E_j}\right]}{\mathbf{P}\left[\bigcap_{j\in S_1} \overline{E_j} \mid \bigcap_{j\in S_2} \overline{E_j}\right]}$$

We will give an upper bound on the numerator of the LHS and a lower bound on the denominator of the LHS. First, consider the numerator. We bound it as follows.

$$\mathbf{P}\left[E_i \cap (\bigcap_{j\in S_1} \overline{E_j}) \mid \bigcap_{j\in S_2} \overline{E_j}\right] \leq \mathbf{P}\left[E_i \mid \bigcap_{j\in S_2} \overline{E_j}\right] = \mathbf{P}\left[E_i\right]$$

where the first step is the trivial bound, and in the second step we used independence. Second, let us consider the denominator. Without loss of generality, suppose $S_1 = \{1, ..., r\}$, and let $S_2 = \{r+1, ..., n\}$. Then, we have the following.

$$\begin{aligned}\mathbf{P}\left[\bigcap_{j\in S_1} \overline{E_j} \mid \bigcap_{j\in S_2} \overline{E_j}\right] &= \prod_{k=1}^{r} \mathbf{P}\left[\overline{E_k} \mid \bigcap_{j=k+1}^{r} \overline{E_j} \cap \bigcap_{j\in S_2} \overline{E_j}\right] \\ &= \prod_{k=1}^{r} \left(1 - \mathbf{P}\left[E_k \mid \bigcap_{j=k+1}^{r} \overline{E_j} \cap \bigcap_{j\in S_2} \overline{E_j}\right]\right)\end{aligned}$$

Now, by induction hypothesis, each term in the product on the RHS can be bounded below by $1 - \frac{1}{d+1}$. So, we see the following.

$$\begin{aligned}\mathbf{P}\left[\bigcap_{j\in S_1} \overline{E_j} \mid \bigcap_{j\in S_2} \overline{E_j}\right] &\geq \left(1 - \frac{1}{d+1}\right)^r \\ &\geq \left(1 - \frac{1}{d+1}\right)^d\end{aligned}$$

where above we are using the fact that $r \leq d$, since each E_i depends on atmost d other E_js. Finally, note that

$$\left(1 - \frac{1}{d+1}\right)^d \geq \frac{1}{e}$$

and this is the lower bound we'll use for the denominator. Finally, putting everything together, we have the following.

$$\mathbf{P}\left[E_i \mid \bigcap_{j\in S} \overline{E_j}\right] \leq \frac{\mathbf{P}\left[E_i\right]}{\frac{1}{e}} = e\mathbf{P}\left[E_i\right] \leq ep \leq \frac{1}{d+1}$$

and hence the claim follows by induction. This completes the proof of the claim. ■

3. Sublinear Algorithms

We will begin this section by looking at some examples.

3.1. **Outputting index of an even number.** Suppose we are given an array A containing a permutation of numbers in the set $[n]$. Our goal is to come up with an algorithm that outputs an index containing an even number. First, we will show that there is no deterministic algorithm that can solve this in sublinear time.

Proposition 3.1. *There is no deterministic algorithm that solves the above problem in $o(n)$ time.*

Proof. Let $S \subseteq [n]$ be the set of indexes that the algorithm queries. Also, assume that $|S| \leq n/2$. Clearly, an adversary can give an input array in which all the indexes in S have odd numbers, and all other indexes have even numbers. In that case, the algorithm will have to query atleast $n/2 + 1$ numbers. This proves the claim. ■

Let us now consider bringing in randomization. So, let us look at the following simple algorithm.

(1) Pick an index $i \in [n]$ uniformly at random.
(2) Repeat the above constantly many times until you get an even number.

Note that for an index i,

$$\mathbf{P}\left[A[i] \text{ is even}\right] = \frac{1}{2}$$

So, we can just choose some constant T such that the success probability in T trials is atleast 99 percent. Clearly, this algorithm runs in time $O(1)$, i.e this is a sublinear algorithm.

3.2. **Diameter of a point set.** We are given a set of m points, and we are given pairwise distances between the points satisfying the triangle inequality. Using this information, we want to determine the diameter of the set. Just as a note, observe that here the input has size $\Omega(m^2)$.

The first question we ask is: can we solve this problem exactly in sublinear time using a deterministic algorithm? In the following claim, we prove that this is *not* possible.

Proposition 3.2. *There is no deterministic algorithm that solves the above problem in exactly sublinear time, i.e in time $o(m^2)$.*

Proof. To be completed. ■

Motivated by the above claim, we will seek an approximate answer to the problem. So, consider the following deterministic algorithm.

(1) Pick a point $i \in [m]$.
(2) Determine the farthest point $k \in [m]$ from l.
(3) Output l_{ik}, i.e the distance between i and k.

Proposition 3.3. *The above algorithm approximates the diameter by a factor of 2. The running time of the algorithm is $O(m)$.*

Proof. To prove this, suppose l_{uv} is the diameter of the set, where $u, v \in [m]$. We want to show that

$$l_{uv} \leq 2l_{ik}$$

To prove this, consider the edges ui and iv. By the triangle inequality, we know that

$$\begin{aligned} l_{uv} &\leq l_{ui} + l_{iv} \\ &\leq l_{ik} + l_{ik} = 2l_{ik} \end{aligned}$$

Clearly, the running time of the algorithm is $O(m)$, because we need to scan all the distances from i to any other vertex to pick the farthest vertex. ■

3.3. **Two types of approximations.** Let us now look at the two types of approximations in the literature of approximation algorithms.

- *Classical Approximation.* When the true answer is t, the output is $\tilde{t}$ such that
  (1) $\frac{t}{\alpha} \leq \tilde{t} \leq \alpha t$ (multiplicative)
  (2) $t - \beta \leq \tilde{t} \leq t + \beta$ (additive)
- *Property testing.* This is usually studied in the realm of *decision problems*. In the classical setting of decision problems, we want to *accept* the YES instances of the problem with probability $\geq 2/3$ and we want to *reject* the NO instances of the problem with probability $\geq 2/3$. In the setting of *property testing*, we *don't care* about no instances which are *close* to being YES instances (we'll make this formal).

  Formally, an input is *ϵ-far* from YES if it has to be changed in $\geq \epsilon$ fraction of positions to become YES, where $\epsilon \in (0, 1)$

Example 3.1. Suppose our universe is all arrays of length n. Also, suppose the set of YES instances is the set of all sorted arrays of length n. By our definition, arrays which have to be changed in ϵn values to make them sorted are said to be *ϵ-far* from YES. In this case, we can say that an array is ϵ-far from YES if it's *Hamming distance* from every sorted array is $\geq \epsilon n$.

For instance, consider the following array.

$$A = [2, 1, 4, 3, \cdots, n, n-1]$$

This array is 1/2-far from being YES, because from every pair of consecutive elements, atleast one must be changed. In fact, we can change the elements $1, 3, ..., n-1$ to $2, 4, ..., n$ to make the array sorted.

3.4. **Property Testing algorithm for connectedness.** In this section, we will look at a property testing algorithm for testing connectedness of graphs. Clearly, there is no deterministic algorithm which can test the connectedness of a graph exactly in sublinear time.

Let us now describe the algorithm.

(1) Our input will be a graph $G = (V, E)$ on n vertices and m edges.
(2) The goal is to distinguish with high probability between
  (a) G is connected.
  (b) G is ϵ-far from being connected, for $\epsilon \in (0, 1)$.
(3) The *input representation* in this problem will be adjacency lists. With this input representation, G is ϵ-far from being connected if it differs from every n-vertex connected graph on $\geq \epsilon m$ edges, i.e we need to add atleast ϵm edges to the graph to make it connected (because it is useless to remove edges).
(4) The next question is: how do we access the input? After all, we are aiming for a sublinear time algorithm. To do this, we allow two types of queries on the graph.
  (a) *Neighbor queries.* The input to this query will be (v, i), and the query will return the ith neighbor of the vertex v. We assume that there is some oracle doing this for us.
  (b) *Degree queries.* The input here is a vertex v, and the output will be the degree of v. Again, we assume that an oracle does this for us.

It is easy to see that both of these queries take $O(1)$ time.

Theorem 3.4. *Connectedness of graphs can be ϵ-tested using $O\left(\left(\frac{1}{\epsilon\overline{d}}^3\right)\right)$ queries, where $\overline{d} = \frac{2m}{n}$.*

Remark 3.4.1. Note the significance of the statement of the theorem; the complexity is independent of the size of the input representation; it only depends on the ratio $\frac{m}{n}$.

Proof. We will prove this in a sequence of steps. First, we make some simple observations.

Suppose G is ϵ-far from connected. An noticed before, this means atleast ϵm edges need to be addded to G to make it connected. This clearly implies that G has atleast $\epsilon m + 1$ CCs; this is true because adding one edge decreases the number of CCs by atmost 1.

Call a CC of G *small* if it has $\leq \frac{2n}{\epsilon m} = \frac{4}{\epsilon\overline{d}}$ vertices. Otherwise, call the CC *big*.

With this definition, it is clear that the number of big CCs in G is $< \frac{\epsilon m}{2}$. This is clear because if x is the number of big CCs and if $x \geq \frac{\epsilon m}{2}$, then

$$x \cdot \frac{\epsilon m}{2} > \frac{2n}{\epsilon m} \cdot \frac{\epsilon m}{2} = n$$

which is a contradiction.

For the algorithm, look at the pseudocode in **Algorithm** 4. First, we claim that if G is connected, then the algorithm always *rejects*, provided $\epsilon \geq \frac{2}{m}$. This is clear, because in line 6 of the algorithm, we will never detect a CC, because we never explore all the n vertices.

Next, we claim that if G is ϵ-far from connected, then the tester rejects with probability $\geq 2/3$. So, our tester has only *one-sided error*. To prove this, suppose G is ϵ-far from connected. Then, as we proved above, there stricly fewer than $\frac{\epsilon m}{2}$ big CCs; so, this means there are $\geq \epsilon m + 1 - \frac{\epsilon m}{2} = \frac{\epsilon m}{2}$ small CCs. Because each of these small CCs contains a vertex, we see that

$$\mathbf{P}\left[\text{Sampling a vertex from a small CC}\right] \geq \frac{\epsilon m/2}{n} = \frac{\epsilon m}{2n} = \frac{\epsilon\overline{d}}{4}$$

This implies that

$$\begin{aligned}\mathbf{P}\left[\text{Not sampling a vertex from a small CC}\right] &\leq \left(1 - \frac{\epsilon\overline{d}}{4}\right)^{\frac{8}{\epsilon\overline{d}}} \\ &\leq e^{-2} < \frac{1}{3}\end{aligned}$$

where we have used the inequality $1 - x \leq e^{-x}$. This means that with probabilty $> \frac{2}{3}$, one of the sampled vertices falls into a small CC, i.e with probability $> \frac{2}{3}$, the tester will *reject*. This proves the claim.

Let us now compute the query complexity of the tester. Observe that there are $\frac{8}{\epsilon\overline{d}}$ iterations. In each iteration, we do a BFS/DFS until seeing $\leq \frac{4}{\epsilon\overline{d}}$ vertices. This means that the number of queries is $\leq \left(\frac{4}{\epsilon\overline{d}}\right)^2$. Hence, the query complexity is

$$O\left(\left(\frac{1}{\epsilon\overline{d}}\right)^3\right)$$

and this completes the proof of the claim. ■

Algorithm 4 ϵ-tester for connectedness

Input: $\epsilon \in (0,1)$, n, m; query access to graph G

for $t = 1$ to $\frac{8}{\epsilon \bar{d}}$ **do**
  Sample a vertex $v \in V$ u.a.r.
  Run a BFS/DFS up until seeing $\leq \frac{4}{\epsilon \bar{d}} = \frac{2n}{\epsilon m}$ vertices.
  reject if a connected component is detected upon seeing these many vertices.
end for
accept

3.5. **Estimating the number of connected components.** We will now look at an algorithm to estimate the number of connected components in the input graph. The algorithm will be very similar to what we saw in the last section. Section to be completed.

3.6. **Testing sortedness of array.** Suppose we have an array A of length n. Our goal is to distinguish with high probability between the following two cases.

(1) A is sorted.
(2) $\geq \epsilon n$ values in A need to be changed to make it sorted.

Also, we are given query access to the array.

Theorem 3.5. *ϵ-testing sortedness can be done in $O\left(\frac{\log n}{\epsilon}\right)$ queries, where $\epsilon \in (0,1)$ is called the proximity parameter. Moreover, there is a one-sided error tester that achieves this bound.*

Proof. We will use the following ideas to come up with a tester.

(1) Binary search always succeeds in a sorted array.
(2) Binary search fails w.h.p on a far from sorted array.

Look at **Algorithm** 5 for the pseudocode of our tester.

We first claim that if the array is sorted, then the tester always *accepts*. The proof is clear, because binary search on a sorted array will always work.

Next, we claim that if array A is ϵ-far from sorted, then the tester will *reject* with probability $\geq \frac{2}{3}$. Let us now prove this. We will prove this in a bunch of steps.

First, for $i \in [n]$, call $A[i]$ *good* if binary search successfully finds $A[i]$. Otherwise, call $A[i]$ *bad*. We claim that A restricted to good values is *sorted*. To prove this, suppose $A[i]$ and $A[j]$ are good elements, and suppose $i \leq j$. We will show that $A[i] \leq A[j]$. To see this, recall how binary search works: on a given range, it considers the middle element of the range, compares the value to be searched with the array value of the middle element, and moves left/right accordingly. So, consider the implicit *binary search tree* that is created by binary searching $A[i]$ and $A[j]$. The root of the tree is the element $A[n/2]$. Clearly, $A[i]$, $A[j]$ have a common ancestor (namely the root), and hence they have a least common ancestor. Let $A[k]$ be the least common ancestor of $A[i]$ and $A[j]$. Clearly, i lies to the left to k, while j lies to the right of k, i.e $i \leq k \leq j$. Also, because binary search is successful for $A[i]$ and $A[j]$, it must be true that $A[i] \leq A[k]$ and $A[j] \geq A[k]$, which clearly means that $A[i] \leq A[j]$. This proves our claim.

Now, we claim that we will "repair" A to make it a sorted array by changing it's values only on bad points. This is actually very easy to see. Suppose array A has good points at indices $i_1 \leq i_2 \leq \cdots i_k$. Then, we can make array A sorted by changing the

values in the range (i_{j-1}, i_j) to $A[j-1]$ for every $2 \leq j \leq k$, and by changing values in the range $[1, i_1)$ to the value $A[i_1]$.

So, the above claim means that the distance of A to sortedness $\leq$ the number of bad points in A. Because A was assumed to be ϵ-far from sortedness, we clearly see that the number of bad points in A is $\geq \epsilon n$.

Now we can compute the rejection probability, which is simply the probability of picking a bad element in the array. In one iteration,

$$\mathbf{P}\left[\text{Tester rejects in one iteration}\right] \geq \frac{\epsilon n}{n} = \epsilon$$

So, if we put $T = 2/\epsilon$, we have that

$$\mathbf{P}\left[\text{Tester rejects}\right] \geq 1 - (1-\epsilon)^{2/\epsilon} \geq 1 - e^{-2} \geq \frac{2}{3}$$

and hence this proves the claim.

Finally, let us compute the query complexity of the algorithm. In each iteration, tester makes $O(\log n)$ queries. So, the total query complexity is $O\left(\frac{\log n}{\epsilon}\right)$, and this completes the proof of the theorem. ■

Algorithm 5 Tester for array sortedness

1: **for** $t = 1$ to T **do**
2: Pick $i \in [n]$ u.a.r.
3: Perform a binary search for $A[i]$.
4: **reject** if binary search fails.
5: **end for**
6: **accept**.

3.7. **Witness Lemma.** In this section, we will prove a handy lemma, which is called the *Witness Lemma* (this is not a standard name).

Proposition 3.6 (Witness Lemma)**.** *Suppose there are n objects, out of which t are faulty. Suppose we draw r objects u.a.r (with replacement) from these these objects. Then,*

$$\begin{aligned}\mathbf{P}\left[\text{Finding a faulty object}\right] &= 1 - \mathbf{P}\left[\text{Not finding a faulty object}\right] \\ &= 1 - \left(1 - \frac{t}{n}\right)^r\end{aligned}$$

So, to find a faulty object with probability atleast $2/3$, it is enough to set $r = \frac{2n}{t}$.

Proof. The proof is trivial. Just use the inequality $1 - x \leq e^{-x}$, and the fact that $e^{-2} < \frac{1}{3}$. ■

3.8. **Testing monotonocity of boolean functions over the hypercube.** Let us first describe the problem statement. Suppose we have a boolean function $f : \{0,1\}^d \to \{0,1\}$. We have query access to the function. We also have some input $\epsilon \in (0,1)$. The goal is to distinguish with high probability between the following cases.

(1) f is monotone, i.e if $x \preceq y$ then $f(x) \leq f(y)$. Here $\preceq$ is the lexicographic ordering.
(2) f is ϵ-far from being monotonic, i.e $\geq \epsilon 2^d$ f-values must be changed to make it monotonic.

Our algorithm will be the following.

Algorithm 6 Algorithm for testing monotonicity of boolean functions

1: **Input**: f, $\epsilon \in (0,1)$, $d \in \mathbf{N}$ (dimension); query access to f.
2: **for** $t = 1$ to $T = \Theta\left(\frac{d}{\epsilon}\right)$ **do**
3: Sample an edge of the boolean hypercube u.a.r.
4: If f violates monotonicity of f, **reject**.
5: **end for**
6: **accept**

By an *edge* of the boolean hypercube we just mean a pair of boolean vectors in $\{0,1\}^d$ which differ in exactly one coordinate. Also, f is said to *violate* an edge if it is not monotonic on the vertices of the edge.

Clearly, the algorithm *never rejects* a monotonic function. So, this algorithm has one-sided error. So, it only remains to show that if f is ϵ-far from monotonic, then the tester rejects with high probability.

Proposition 3.7. *If f is ϵ-far from monotone, then there are $\geq \epsilon 2^{d-1}$ edges in the hypercube that violate monotonicity.*

Proof. We will prove this by proving the contrapositive. So, suppose f violates monotonicity on $VE_f < \epsilon 2^{d-1}$ edges. Let $VE_f^{(i)}$ denote the number of violated edges along the ith dimension for $i \in [d]$. It is clear that

$$VE_f = \sum_{i \in [d]} VE_f^{(i)}$$

We will now give a method to "repair" f, i.e a method to make f monotonic by changing less than $\epsilon 2^{d-1}$ edges, which will complete the proof.

We do the following.

(1) Iterate from $i = 1$ to $i = d$.
(2) For each i, "repair" the violated edges on dimension i. Here, "repair" means just changing the values of f on the vertices which are connected by violating edges along dimension i; suppose (x, y) is an edge along dimension i which is violated such that $y \prec x$ but $f(y) > f(x)$. Clearly, $f(y) = 1$ and $f(x) = 0$. We can repair by either making $f(x) = 1$, or by making $f(y) = 0$, or by making $f(x) = 1$ and $f(y) = 0$. Do *any* of these operations, but make sure to do the same operation along all edges.

We now claim that repairing a dimension *does not* increase the number of violated edges in other dimensions. The proof of this is really simple, and is done by case work. More specifically, we can show that if i, j are dimensions such that $i \neq j$, then repairing dimension i does not increase the number of violations in dimension j. Complete the cases. Draw a *square*, and just consider the total number of configurations of this square. In every configuration, check that the repair function does not increase the number of violations along the jth dimension.

So, it follows that the total number of repair operations will be $\leq \sum_{i \in [d]} 2 \cdot VE_f^{(i)} = 2 \cdot VE_f$; we multiply by 2 because for every edge, we change values of atmost 2 vertices. Also, note that

$$2VE_f < \epsilon 2^d$$

So, f can be made monotone by changing values on $< \epsilon 2^d$ vertices, which means that f is not ϵ-far from monotone. ∎

3.9. L_p **testing.** So far, we've defined ϵ-farness with respect to the *Hamming distance*, which is simply the number of coordinates in which the input differs from a particular property. But this notion of distance is not suitable for all problems. Instead, one can study more meaningful distance measures, one of which are L_p distances. We will now define these formally.

Definition 3.1. Suppose D is a finite domain, and let $f, g : D \to [0,1]$. For $p \geq 1$, we define

$$L_p(f,g) = ||f-g||_p = \left(\sum_{x \in D} |f(x)-g(x)|^p\right)^{\frac{1}{p}}$$

Also, we define the *relative distance* $d_p(f,g)$ as follows.

$$d_p(f,g) = \frac{||f-g||_p}{||\mathbf{1}||_p}$$

where $\mathbf{1}$ is the all ones function. Just as before, *property testing* and *ϵ-farness* can be defined in terms of L_p distances. Let $\mathcal{P}$ be any set of functions $h : D \to [0,1]$. The *distance* $d_p(f,\mathcal{P})$ of f from $\mathcal{P}$ is defined in the most natural way.

$$d_p(f,\mathcal{P}) = \inf_{h \in \mathcal{P}} \{d_p(f,h)\}$$

Given a property $\mathcal{P}$ of functions $h : D \to [0,1]$, the *L_p-testing* problem has the following goal: given some input $f : D \to [0,1]$, we want to distinguish with probability $\geq \frac{2}{3}$ between the following two cases.

(1) $f \in \mathcal{P}$.
(2) $d_p(f,\mathcal{P}) \geq \epsilon$.

The farness parameter $\epsilon \in (0,1)$ is given as input, along with oracle access to the function f.

3.10. L_1**-testing monotonicity of functions on a grid.** Let n, d be positive integers. Consider the grid $[n]^d$. We can naturally define a partial order on this grid as follows: given $x, y \in [n]^d$, we say that $x \preceq y$ if $x_i \leq y_i$ for all $1 \leq i \leq d$. A function $f : [n]^d \to [0,1]$ is said to be *monotone* if $x \preceq y$ implies $f(x) \leq f(y)$. Let $\mathcal{M}$ denote the class of monotone functions $h : [n]^d \to [0,1]$. We will give an algorithm for L_1 testing of the property $\mathcal{M}$.

Theorem 3.8 (Characterization Theorem (Berman et. al)). *Let $f : [n]^d \to [0,1]$ and let $t \in [0,1]$. Define*

$$f_t(x) = \begin{cases} 1 & , \quad f(x) \geq t \\ 0 & , \quad f(x) < t \end{cases}$$

So, f_t is a boolean function. Then,

$$L_1(f,\mathcal{M}) = \int_0^1 L_1(f_t,\mathcal{M})\,\mathrm{d}t$$

Proof. See the proof of **Lemma 2.1** in this paper: https://cs-people.bu.edu/sofya/pubs/Lp-testing-stoc-published.pdf. ∎

Corollary 3.8.1. *If f and f_t are defined as above, then*

$$d_1(f,\mathcal{M}) = \int_0^1 d_1(f_t,\mathcal{M})\,\mathrm{d}t$$

Proof. The proof is trivial and just follows from the definition. ■

Theorem 3.9. *Suppose T is a non-adaptive 1-sided error tester for monotonicity of functions of the form $f : [n]^d \to \{0, 1\}$, then T is also an L_1-tester for monotonicty of functions of the form $f : [n]^d \to [0, 1]$.*

Remark 3.9.1. This theorem essentially reduces the problem of monotonicity testing of general real valued functions to monotonicity testing of boolean functions. This technique is also called *range reduction*.

Proof. Recall that a tester is *non-adaptive* if a query made by the tester does not depend upon answers to its previous queries. Hence, the tester can decide all of the queries it wants to make before testing. So, the tester T works as follows.

(1) The tester samples $Q \subseteq [n]^d$ according to some distribution.
(2) It rejects only if there is a violation of monotonicity among f values in Q, i.e only if there are $x, y \in Q$ such that $x \prec y$ but $f(x) > f(y)$.

So, we propose the following tester for real functions: just run T on input f with oracle access to f. Note that the tester T itself does not use any property of boolean functions; it just samples a bunch of points and tests for monotonocity over those points. So, T will still work work the input f.

Now note that if f is monotone, then T will *always* accept f. This is clear from the description of T.

So now suppose f is ϵ-far from monotone, i.e suppose $d_1(f, \mathcal{M}) \geq \epsilon$. We will show that with probability $\geq \frac{2}{3}$, the tester will reject. By **Theorem** 3.8 (**Characterisation Theorem**), there exists some $t^* \in [0, 1]$ such that $d_1(f_{t^*}, \mathcal{M}) \geq \epsilon$. Note that f_{t^*} is a boolean function. Now, suppose $\mathcal{M}'$ is the class of boolean monotone functions on D. Clearly, we have that

$$d_1(f_t^*, \mathcal{M}') \geq d_1(f_t^*, \mathcal{M}) \geq \epsilon$$

where the above inequality is true because the infimum cannot grow if we take larger sets. Moreover, observe that $d_1(f_t^*, \mathcal{M}')$ is just the *Hamming distance* between f and $\mathcal{M}'$. So, if T runs with query access to f_{t^*}, then with probability $\geq \frac{2}{3}$, Q contains x, y such that $x \prec y$ but $f_{t^*}(x) = 1$ and $f_{t^*}(y) = 0$. This clearly means that $f(x) \geq t$ and $f(y) < t$, which implies that $f(x) > f(y)$. So, the tester T will reject f with probability $\geq 2/3$, and this completes the proof of the claim. ■

3.11. **The Reverse Markov Inequality.** In this small section, we will prove a very useful inequality.

Proposition 3.10 (**Reverse Markov's Inequality**)**.** *Let X be a random variable such that $\mathbf{E}[X] \geq \mu$, and assume that X is bounded above by 1. Then,*

$$\mathbf{P}\left[X \geq \frac{\mu}{2}\right] \geq \frac{\mu}{2}$$

Proof. The proof of this claim is very simple. First, using the fact that X is bounded above by 1, we have the following.

$$\mu \leq \mathbf{E}[X] \leq \mathbf{P}\left[X \geq \frac{\mu}{2}\right] \cdot 1 + \mathbf{P}\left[X < \frac{\mu}{2}\right] \frac{\mu}{2} \leq \mathbf{P}\left[X \geq \frac{\mu}{2}\right] + \frac{\mu}{2}$$

This proves the claim. ■

3.12. **Monotonicity testing of boolean functions over the general grid.** Inspired from the previous section, we now only need to come up with a tester for testing monotonicity of boolean functions of the form $f : [n]^d \to \{0, 1\}$.

We visualise the domain $[n]^d$ as an undirected graph as follows: the vertices are simply all the points in $[n]^d$; moreover, $x, y \in [n]^d$ have an edge iff. there is some $i \in [d]$ such that $x_i = y_i + 1$ and $x_j = y_j$ for all $j \neq i$. In simple words, two vertices are connected iff. they differ by atmost one in exactly one coordinate, and they are equal on all other coordinates.

An *axis-parallel line* is defined to be a set of points in $[n]^d$ that are identical on all but one coordinate. Clearly, such a line will contain n points.

Theorem 3.11. *ϵ-testing monotonicity of a function $g : [n] \to \{0, 1\}$ can be done in $O\left(\frac{1}{\epsilon}\right)$ query complexity.*

Proof. To be completed. ■

Theorem 3.12. *Let $\mathcal{M}$ be the class of all monotone boolean functions on $[n]^d$. For $f : [n]^d \to \{0, 1\}$,*

$$\underset{\text{u.a.r axis parallel line } l}{\mathbf{E}} \left[d_0(f|_l, \mathcal{M})\right] \geq \frac{d_0(f, \mathcal{M})}{2d}$$

The expectation above is taken is taken over all axis parallel lines taken uniformly at random. As usual, d_0 is the Hamming distance.

Proof. The proof was not covered in the course. ■

Corollary 3.12.1. *If $f : [n]^d \to \{0, 1\}$ is ϵ-far from $\mathcal{M}$, then the expected (relative) distance to $\mathcal{M}$ of $f|_l$ is $\geq \frac{\epsilon}{2d}$.*

Proof. Let X be the random variable

$$X := d_0(f|_l, \mathcal{M})$$

Clearly, X takes values in the range $[0, 1]$. By the previous theorem, we know that

$$\mathbf{E}[X] \geq \frac{d_0(f, \mathcal{M})}{2d}$$

Now, f is ϵ-far from $\mathcal{M}$, then we see that $d_0(f, \mathcal{M}) \geq \epsilon$, and hence

$$\mathbf{E}[X] \geq \frac{\epsilon}{2d}$$

and this proves the claim. ■

From the above corollary, we can conclude something more. By **Proposition** 3.10 (**Reverse Markov**), we see that with probability $\geq \frac{\epsilon}{4d}$, $X \geq \frac{\epsilon}{4d}$, i.e the line l sampled is $\frac{\epsilon}{4d}$-far from monotonocity. This fact motivates the following tester.

Algorithm 7 ϵ-tester for monotonocity of $f : [n]^d \to \{0, 1\}$

1: **Input**: ϵ, d, n; oracle access to $f : [n]^d \to \{0, 1\}$.
2: **for** $t = 1$ to $t = 16\frac{d}{\epsilon}$ **do**
3: Sample a line l u.a.r from the hypergrid $[n]^d$.
4: Test monotonocity of $f|_l : [n] \to \{0, 1\}$ with parameter $\frac{\epsilon}{4d}$.
5: **reject** only if the above test rejects.
6: **end for**

Theorem 3.13. *The above tester is a one-sided error tester with query complexity* $O\left(\frac{d^2}{\epsilon^2}\right)$.

Proof. It is clear that the tester has only one-sided error, i.e if f is monotonic, the tester can never reject.

Now, suppose f is ϵ-far from $\mathcal{M}$. The only way the tester rejects is if either of the following occurs.

(1) The tester fails to sample a line that is $\frac{\epsilon}{4d}$-far from $\mathcal{M}$. As noted in the paragraph just above the pseudocode, the probability of this happening is $\leq 1 - \frac{\epsilon}{4d}$ for one iteration; so, the probability of never sampling such a line is

$$\leq \left(1 - \frac{\epsilon}{4d}\right)^{\frac{16d}{\epsilon}} \leq e^{-4} \leq \frac{1}{8}$$

(2) So now, suppose that we sample a good line, i.e a line which is $\frac{\epsilon}{4d}$-far from $\mathcal{M}$ in atleast one iteration. The failure possibility is that the test on the line fails. By boosting our test (which can be done by not blowing up the query complexity), we can ensure that the test on the line fails with probability $\leq \frac{1}{8}$.

So, it follows that the failure probability is $\leq \frac{1}{8} + \frac{1}{8} = \frac{1}{4}$.

Using **Theorem** 3.11, it is clear that the query complexity is $O\left(\frac{d^2}{\epsilon^2}\right)$. This completes the proof. ■

3.13. **An even better strategy for boolean functions.** In this section, we will expand on the previous section by coming up even a faster tester for ϵ-testing monotonicity of functions $f : [n]^d \to \{0, 1\}$ w.r.t the Hamming distance. Our result will rely on the following theorem.

Theorem 3.14. *Let X be a random variable taking values in $[0, 1]$. Suppose $\mathbf{E}[X] \geq \mu$. Let $p_i = \mathbf{P}\left[X \geq \frac{1}{2^i}\right]$ and let $k_i = \Theta\left(\frac{1}{2^i \mu}\right)$. Then,*

$$\prod_{i=1}^{\log \frac{4}{\mu}} (1 - p_i)^{k_i} \leq \frac{1}{3}$$

Proof. This was not proved in the course. ■

Algorithm 8 A new tester for the problem.

1: **Input**: ϵ, d, n; query access to $f : [n]^d \to \{0, 1\}$.
2: **for** $i = 1$ to $i = \left\lfloor \log \frac{4}{\mu} \right\rfloor$ **do**
3: Sample $k_i = \Theta\left(\frac{1}{2^i \mu}\right)$ lines.
4: For each line, run a monotonicity tester with distance parameter $\frac{1}{2^i}$.
5: *reject* only if any of the above tests reject.
6: **end for**

The proof of the correctness of this tester is very similar to the correctness proof of the previous tester. Let us analyze the query complexity. Clearly, the query complexity is

$$\sum_{i=1}^{\log \frac{4}{\mu}} \Theta\left(\frac{1}{2^i \mu}\right) \cdot O(2^i) = O\left(\frac{1}{\mu} \log \frac{1}{\mu}\right) = O\left(\frac{d}{\epsilon} \log \frac{d}{\epsilon}\right)$$

A lot of details are missing from this proof.

4. Streaming Algorithms in the Data Streaming Model

4.1. **The Streaming Model of Computation.** Let us briefly describe the *streaming model of computation*. In this model, there is an algorithm A which needs to output something interesting from a given input. However, the way the input is presented to the algorithm is a bit different.

(1) The input is given to the algorithm as a *continuous stream*, i.e the input data points are available one at a time at each time step.
(2) The algorithm A can see each data element only once (a 1-pass).
(3) There is often a limited working memory, like being logarithmic in the size of the input.
(4) The algorithm A must quickly process each element.

Example 4.1. Suppose we have a stream of m numbers from the universe $[n]$.

$$\langle a_1, a_2, ..., a_m \rangle \in [n]^m$$

In the streaming model, a desirable bound on the memory will be polylog(m, n); for instance, we are only allowed to store upto $\log(m)$ data points at once. This is helpful because in the real of streaming algorithms, we are dealing with really large streams.

We want to sample a uniformly random element from this stream. If we know the length of the stream, i.e if we know m in advance, we can just do the following: sample a point $t \in [m]$ u.a.r, and output a_t. Clearly, the memory required to do this is $\log m + \log n$, which satisfies our memory constraints. But the question is: what if we don't know m (which is the more realistic situation)? We will see the answer to this in the next section.

4.2. **Reservoir Sampling.** Consider the last question posed in the example in the previous section. We don't know the length of the stream (m), and we want to sample a data point from the stream uniformly at random. We will follow the following simple strategy, known as *reservoir sampling*.

(1) Initially, we let $s \leftarrow a_1$.
(2) When the tth element arrives, we set $s \leftarrow a_t$ with probability $1/t$.

Let us now see why this strategy works. Consider some time point t. For $i \leq t$, we have

$$\begin{aligned}
\mathbf{P}[s = a_i] &= \frac{1}{i} \cdot \left(1 - \frac{1}{i+1}\right) \cdots \left(1 - \frac{1}{t}\right) \\
&= \frac{1}{i} \cdot \frac{i}{i+1} \cdots \frac{t-1}{t} \\
&= \frac{1}{t}
\end{aligned}$$

And this is exactly what we wanted. The space complexity for this algorithm is again $O(\log n + \log m)$; the factor of $\log n$ is simply to store the data point a_i, and the fact $\log m$ comes from the fact that at each step t, we are tossing a t-sided die; since $t \leq m$, we need $\log m$ bits to store the result of this die.

4.3. **Counting number of distinct elements in a stream.** The problem is the following: given a stream of data, we need to determine the number of distinct elements in the stream. Again, suppose the data points come from the universe $[n]$, and suppose there are m data points.

There are some known lower bounds for the space complexity needed to solve this problem.

- To solve this problem *exactly* (even by using randomization), we need $\Omega(n)$ bits.
- To solve this problem deterministically (even up to an approximation), we need $\Omega(m)$ bits.

So, we need to make use of both randomization and approximation to hope to solve this problem better.

So we modify our goal: we want to estimate the number of distinct elements in stream $\langle a_1, ..., a_m \rangle$ up to a multiplicative factor $1 \pm \epsilon$ with probability $\geq \frac{2}{3}$. We will show an $O\left(\frac{\log n}{\epsilon^2}\right)$ space algorithm for this problem.

Algorithm 9 Counting number of distinct elements in a stream

1: **Input**: $\langle a_1, ..., a_m \rangle$.
2: Sample a random hash function $h : [n] \to [n]$.
3: Apply h to each stream element.
4: Let X denote the tth smallest hash value seen where $t = \frac{10}{\epsilon^2}$.
5: **return** $\tilde{r} = n \cdot \frac{t}{X}$.

Let r denote the number of distinct elements in the input stream. We will show that

$$\mathbf{P}\left[\tilde{r} \geq (1+\epsilon)r\right]$$

is a very small number. Let us now prove this. Let $u_1, ..., u_r$ denote the distinct elements of the stream. So,

$$\begin{aligned}\mathbf{P}\left[\tilde{r} \geq (1+\epsilon)r\right] &= \mathbf{P}\left[\frac{tn}{X} \geq (1+\epsilon)r\right] \\ &= \mathbf{P}\left[X \leq \frac{tn}{(1+\epsilon)r}\right]\end{aligned}$$

So, we want to bound the probability that the tth smallest hash value is $\leq \frac{tn}{(1+\epsilon)r}$. For each $1 \leq i \leq r$, let

$$Y_i = \mathbf{1}_{h(u_i) \leq \frac{tn}{(1+\epsilon)r}}$$

and let

$$Y = \sum_{i \in [r]} Y_i$$

Now, note that

$$\mathbf{P}\left[X \leq \frac{tn}{(1+\epsilon)r}\right] = \mathbf{P}\left[Y \geq t\right] \tag{4.1}$$

Also, we see that

$$\begin{aligned}\mathbf{E}[Y] &= \sum_{i\in[r]} \mathbf{E}[Y_i] \\ &= r\mathbf{E}[Y_1] \\ &= r\mathbf{P}[Y_1 = 1] \\ &= r\mathbf{P}\left[h(u_1) \leq \frac{tn}{(1+\epsilon)r}\right] \\ &\leq r \cdot \frac{tn}{(1+\epsilon)r} \cdot \frac{1}{n} \\ &= \frac{t}{(1+\epsilon)}\end{aligned}$$

Above, we have used the fact that h hashes a particular element of $[n]$ to any other element of $[n]$ with uniform probability.

Also, using the fact that Y_i's are all independent random variables, we have that

$$\begin{aligned}\mathbf{Var}(Y) &= \sum_{i\in[r]} \mathbf{Var}(Y_i) \\ &\leq \sum_{i\in[r]} \mathbf{E}\left[Y_i^2\right] \\ &= \sum_{i\in[r]} \mathbf{E}[Y_i] \\ &= \mathbf{E}[Y]\end{aligned}$$

Above, we have also used the fact that Y_is are 0-1 random variables. Now using the fact that $(1+\epsilon)\mathbf{E}[Y] \leq t$, we see that

$$\begin{aligned}\mathbf{P}[Y \geq t] &\leq \mathbf{P}[Y \geq (1+\epsilon)\mathbf{E}[Y]] \\ &\leq \mathbf{P}[|Y - \mathbf{E}[Y]| \geq \epsilon\mathbf{E}[Y]] \\ &\leq \frac{\mathbf{Var}(Y)}{\epsilon^2\mathbf{E}[Y]^2} \\ &\leq \frac{1}{\epsilon^2\mathbf{E}[Y]} \\ &= \frac{n}{\epsilon^2 r \cdot \left\lfloor \frac{tn}{(1+\epsilon)r} \right\rfloor}\end{aligned}$$

Complete the above inequality; show that the upper bound is 1/6. Similarly, it can be shown that

$$\mathbf{P}[\tilde{r} \leq (1-\epsilon)r] \leq \frac{1}{6}$$

So, it follows that with probability $\geq 1 - \frac{1}{3} = \frac{2}{3}$, $\tilde{r} \in [(1-\epsilon)r, (1+\epsilon)r]$ and this completes the analysis of the algorithm.

Let us now analyze the space complexity of this algorithm. To find the $(t = 10/\epsilon)$th smallest element, we need $O\left(\frac{\log n}{\epsilon^2}\right)$ memory. But note that to store the hash function, we need $O(n \log n)$ memory, which is bad (recall we only want polylog(m, n) memory).

To fix this problem, we do the following: we sample h from a 2-*wise independent family* $\mathcal{H}$ of hash functions. A family $\mathcal{H}$ of hash functions is said to be 2-*wise independent* if for all distinct $x, x' \in [n]$ and for all distinct $y, y' \in [n]$,

$$\mathbf{P}\left[h(x) = y, h(x') = y'\right] = \frac{1}{n^2}$$

Let p be a prime larger than n. Sample $a, b \in [p-1] \cup \{0\}$ u.a.r, and let $h(x) = ax + b$. In the next section, we wills show that this family of functions is indeed 2-wise independent. So now, to compute h, all we need to store is the numbers a, b; this requires only $O(\log p) = O(\log n)$ bits.

4.4. **k-wise independent hash families.** We will first begin by introducing a new definition.

Definition 4.1. Let $\mathcal{H}$ be a family of functions from $[A] \to [B]$. $\mathcal{H}$ is said to be 2-*wise independent* if for all $i_1 \neq i_2 \in [A]$ and for all $j_1, j_2 \in [B]$ it is true that

$$\mathop{\mathbf{P}}_{h \in \mathcal{H}}\left[h(i_1) = j_1 \wedge h(i_2) = j_2\right] = \frac{1}{B^2}$$

Here the probability is taken over the uniform distribution over $\mathcal{H}$. Similarly, $\mathcal{H}$ is said to be k-*wise independent* if for all *distinct* $i_1, i_2, ..., i_k \in [A]$ and for all $j_1, j_2, ..., j_k \in [B]$, it is true that

$$\mathop{\mathbf{P}}_{h \in \mathcal{H}}\left[h(i_1) = j_1 \wedge \cdots \wedge h(i_k) = j_k\right] = \frac{1}{B^k}$$

It can be checked that in **Algorithm** 9, we can sample our hash function from a 2-wise independent family, and the same guarantees will hold (Check this.) The only place where the proof of the analysis must be modified is when giving bounds on the variance and the expectation of the random variable Y.

Proposition 4.1. *k-wise independence implies $(k-1)$-wise independence, and hence it implies 1-wise independence, i.e for for all $i \in [A]$ and $j \in [B]$,*

$$\mathop{\mathbf{P}}_{h \in \mathcal{H}}\left[h(i) = j\right] = \frac{1}{B}$$

Proof. We will only prove this for $k = 2$; the proof for general k is exactly the same.

Suppose the family $\mathcal{H}$ is 2-wise independent. Let $i \in [A]$ and $j \in [B]$ be fixed; take some $i' \neq i \in [A]$. Then, we have the following.

$$\begin{aligned}
\mathop{\mathbf{P}}_{h \in \mathcal{H}}\left[h(i) = j\right] &= \sum_{j' \in [B]} \mathop{\mathbf{P}}_{h \in \mathcal{H}}\left[h(i) = j \wedge h('i) = j'\right] \\
&= \frac{B}{B^2} \\
&= \frac{1}{B}
\end{aligned}$$

and this proves our claim. ∎

4.4.1. *Constructing 2-wise independent hash families.* Let p be a prime number such that $n \leq p < 2n$ (such a prime exists). Let $a, b \in \mathbb{F}_p$, and define

$$h_{a,b}(x) = ax + b \bmod(p)$$

and let $\mathcal{H}$ be the family of all such functions.

Proposition 4.2. *$\mathcal{H}$ constructed above is a 2-wise independent family of functions $\mathbb{F}_p \to \mathbb{F}_p$.*

Proof. Suppose $x, y \in \mathbb{F}_p$ such that $x \neq y$, and let $u, v \in \mathbb{F}_p$. So, we have the following. Below, all equations are in the field $\mathbb{F}_p$.

$$\begin{aligned}\mathop{\mathbf{P}}_{h\in\mathcal{H}}[h(x) = u \wedge h(y) = v] &= \mathop{\mathbf{P}}_{h\in\mathcal{H}}[ax + b = u \wedge ay + b = v] \\ &= \mathop{\mathbf{P}}_{h\in\mathcal{H}}\left[a = \frac{u-v}{x-y} \wedge b = u - \frac{(u-v)}{x-y}x\right] \\ &= \frac{1}{p^2}\end{aligned}$$

and this proves the claim. ■

We will now state a fact without proof that is essential in constructing k-wise independent hash families.

Theorem 4.3. *Let q be any prime power, and consider the finite field $\mathbb{F}_q$ of order q. The set of all polynomials of degree $\leq k-1$ over $\mathbb{F}_q$ forms a k-wise independent hash function family of functions $\mathbb{F}_q \to \mathbb{F}_q$.*

Remark 4.3.1. So we can see that we only need $O(k \log q)$ space to store a hash function from such a family.

4.5. L_2 **norm estimation of frequency vector.** In this section, we will see algorithms to estimate the L_2 norm of a *frequency vector*.

Definition 4.2. Let $\langle a_1, ..., a_m\rangle$ be a stream, where $a_i \in [m]$. The *frequency vector* of the stream is defined as the vector $F = (f_1, ..., f_n)$, where f_i is the number of occurrences of i in the stream.

Our goal is the following: given $\epsilon > 0$, we want to estimate within $1 \pm \epsilon$ factor, the L_2 norm of F, i.e we want to estimate $||F||_2$. Let

$$F_2 := ||F||_2^2 = \sum_{i\in[n]} f_i^2$$

We will devise an $O\left(\frac{\log n + \log m}{\epsilon^2}\right)$-space algorithm for this problem. The algorithm is called the *AMS algorithm* (Alon, Matthias, Szegedy).

Algorithm 10 AMS algorithm for estimating L_2 norm of the frequency vector F

1: Sample a hash function $h : [n] \to \{-1, 1\}$ from a 4-wise independent hash family.
2: Initialise $X \leftarrow 0$.
3: For each stream element a, do $X \leftarrow X + h(a)$.
4: **return** X^2.

4.5.1. *How to sample such a hash function?* Let us now see how to sample a hash function of the given form. Let l be such that $n \leq 2^l < 2n$. We sample a 4-wise independent hash function from $\mathbb{F}_{2^l} \to \mathbb{F}_{2^l}$ (using degree 3 polynomials). Suppose g is this function. Then, we consider the function $h : \mathbb{F}_{2^l} \to \{-1, 1\}$ defined as follows.

$$h(x) = \begin{cases} 1 & , \quad \text{if } g(x) \text{ is even} \\ -1 & , \quad \text{otherwise} \end{cases}$$

So the function h divides the set of outputs of g into two parts. This is how we sample the function h in the algorithm. For simplicity, we will assume that n is a power of 2, i.e $n = 2^l$.

4.5.2. *Computing the expectation.* Let us now show that the algorithm is an unbiased estimator for F_2. To show this, we will show that $\mathbf{E}[X^2] = F_2$. To that end, let z_i be the random variable $z_i = h(i)$ for $i \in [n]$. With this definition, note that

$$X = \sum_{i \in [n]} f_i z_i$$

So, we get the following.

$$\begin{aligned}
\mathbf{E}\left[X^2\right] &= \mathbf{E}\left[\left(\sum_{i\in[n]} f_i z_i\right)^2\right] \\
&= \mathbf{E}\left[\sum_{i\in[n]} f_i^2 z_i^2 + \sum_{i\neq j\in[n]} f_i f_j z_i z_j\right] \\
&= \sum_{i\in[n]} f_i^2 \mathbf{E}\left[z_i^2\right] + \sum_{i\neq j\in[n]} f_i f_j \mathbf{E}\left[z_i z_j\right]
\end{aligned}$$

Now, observe that z_i^2 can only take the value 1, and hence it's expectation is 1. Moreover, if $i \neq j$, then note that

$$\mathbf{E}[z_i z_j] = \mathbf{E}[z_i]\,\mathbf{E}[z_j] = 0$$

where we are using 2-wise independence of the hash function family from which h is sampled (it is 4-wise independent, and hence 2-wise independent). So, it follows that

$$\mathbf{E}\left[X^2\right] = \sum_{i\in[n]} f_i^2 = F_2$$

and hence X^2 is indeed an unbiased estimator.

4.5.3. *A bound on the variance.* Now, we will prove that $\mathbf{Var}(X^2) \leq 2F_2^2$. Let us now prove this. We know that

$$\begin{aligned}
\mathbf{Var}\left(X^2\right) &= \mathbf{E}\left[X^4\right] - \left(\mathbf{E}\left[X^2\right]\right)^2 \\
&= \mathbf{E}\left[X^4\right] - F_2^2
\end{aligned}$$

By expanding out the expectation $\mathbf{E}[X^4]$, the claim can be shown (show this).

At this point, if we simply apply **Chebyshev's Inequality**, we will get that

$$\mathbf{P}\left[|X^2 - F_2| \geq \epsilon F_2\right] \leq \frac{\mathbf{Var}(X^2)}{\epsilon^2 F_2^2} \leq \frac{2F_2^2}{\epsilon^2 F_2^2} = \frac{2}{\epsilon^2}$$

The above bound is not very strong. To fix this problem, we will repeat this algorithm many times, as we will see next.

Algorithm 11 Modified AMS algorithm

1: Let $c \leftarrow \frac{6}{\epsilon^2}$.
2: Run c *independent and parallel* copies of the earlier AMS algorithm.
3: Let $X_1^2, X_2^2, ..., X_c^2$ be their outputs.
4: **return** $Y = \frac{1}{c}\sum_{i\in[c]} X_i^2$

4.5.4. *Improving the variance bound by repetitions.* Our new algorithm will be **Algorithm 11**.

Let us analyze this new algorithm. First, note that

$$\mathbf{E}[Y] = \frac{1}{c}\mathbf{E}\left[\sum_{i\in[c]} \mathbf{E}\left[X_i^2\right]\right] = \frac{1}{c}\cdot cF_2 = F_2$$

and again, Y is an unbiased estimator for F_2.

Let us now bound the variance. Because the X_is are independent, we see that

$$\mathbf{Var}(Y) = \frac{1}{c^2}\sum_{i\in[c]} \mathbf{Var}\left(X_i^2\right) = \frac{c\mathbf{Var}(X_i)^2}{c^2} \leq \frac{2F_2^2}{c}$$

Now, if we apply **Chebyshev's Inequality**, we get that

$$\mathbf{P}[|Y - F_2| \geq \epsilon F_2] \leq \frac{\mathbf{Var}(Y)}{\epsilon^2 F_2^2} \leq \frac{2F_2^2}{\epsilon^2 F_2^2 c} = \frac{2}{c\epsilon^2} = \frac{1}{3}$$

And hence, we see that our estimator gives a good estimation with probability $\geq \frac{2}{3}$.

Let us finally analyze the space complexity of our algorithm. Note that to maintain the values of X_i's, we need $O(c\log m)$ bits, because X_i can be atmost m. Moreover, note that we will have to maintain c hash functions too; each hash function needs $O(4\log n)$ space (as we need to maintain 4 coefficients), and hence the total space required for maintaining the hash functions is $O(c\log n)$. So, the total space required is

$$O(c\log m + c\log n) = O\left(\frac{\log m + \log n}{\epsilon^2}\right)$$

4.6. **Morris Counter.** In this section, we will see a new algorithm, which goes by the name of *Morris Counter*, to approximate the length of a stream of unknown length.

Algorithm 12 Morris Counter for approximating unknown lengths

1: $X \leftarrow 0$.
2: When a new item arrives,

$$X = \begin{cases} X+1 & \text{, with probability } \frac{1}{2^X} \\ X & \text{, with probability } 1 - \frac{1}{2^X} \end{cases}$$

3: **return** $2^X - 1$ as the estimate.

Essentially, the idea behind this algorithm is to maintain the logarithm of the length of the stream. Let us now analyze this algorithm.

Theorem 4.4. *Let $Y = 2^X$. Then, $\mathbf{E}[Y] = m + 1$, where m is the length of the stream.*

Proof. We will prove this by induction on m. For $i \geq 0$, let X_i be the random variable denoting the value of the counter X once i items of the stream have arrived. Let $Y_i = 2^{X_i}$.

For the base case, we have $m = 1$. In this case, note that $X_1 = 1$ with probability 1 and 0 with probability 0. So, it follows that Y_1 is 1 with probability 0 and 2 with probablity 1; hence,

$$\mathbf{E}[Y_1] = 2 = 1 + 1$$

and hence the base case is true.

Now we focus on the inductive step. Suppose for all $i < m$, it is true that $\mathbf{E}[Y_i] = i + 1$. We will prove that this holds for $i = m$ as well. Note that

$$\begin{aligned}
&\mathbf{E}[Y_m] \\
&= \mathbf{E}\left[2^{X_m}\right] \\
&= \sum_{j=0}^{m} 2^j \mathbf{P}[X_m = j] \\
&= \sum_{j=0}^{m} 2^j \left(\mathbf{P}[X_m = j \mid X_{m-1} = j] \cdot \mathbf{P}[X_{m-1} = j] + \mathbf{P}[X_m = j \mid X_{m-1} = j-1] \cdot \mathbf{P}[X_{m-1} = j-1]\right) \\
&= \sum_{j=0}^{m} 2^j \left(\left(1 - \frac{1}{2^j}\right) \mathbf{P}[X_{m-1} = j] + \frac{1}{2^{j-1}} \mathbf{P}[X_{m-1} = j-1]\right) \\
&= \sum_{j=0}^{m} 2^j \mathbf{P}[X_{m-1} = j] + \sum_{j=0}^{m} (2\mathbf{P}[X_{m-1} = j-1] - \mathbf{P}[X_{m-1} = j]) \\
&= \mathbf{E}[Y_{m-1}] + 2 - 1 \\
&= \mathbf{E}[Y_{m-1}] + 1 \\
&= m + 1
\end{aligned}$$

where in the last step above, we have used the inductive hypothesis. So this proves the claim, and shows that our estimator is indeed unbiased. ■

Definition 4.3. A real valued function $f : \mathbf{R} \to \mathbf{R}$ is convex if

$$f\left(\frac{x+y}{2}\right) \leq \frac{f(x) + f(y)}{2}$$

Theorem 4.5 (Jensen's Inequality). *Let Z be a random variable with $|\mathbf{E}[Z]| < \infty$. If f is convex, then*

$$f(\mathbf{E}[Z]) \leq \mathbf{E}[f(Z)]$$

We will use this to analyze $\mathbf{E}[X_m]$. Note that $Y_m = 2^{X_m}$ by definition. We know that the function $f(z) = 2^z$ is convex. So by the above inequality, we know that

$$f(\mathbf{E}[X_m]) \leq \mathbf{E}[f(X_m)]$$

which means that

$$2^{\mathbf{E}[X_m]} \leq \mathbf{E}[Y_m] = m + 1$$

and this means that $\mathbf{E}[X_m] \leq \log_2(m+1)$. Therefore, the expected space complexity needed to store X is $O(\log\log(m+1))$.

Theorem 4.6. $\mathbf{Var}(Y_m) = \dfrac{m(m-1)}{2}$

Proof. Reading exercise. ■

As before, if we directly apply **Chebyshev's Inquality**, we won't get a good bound (just like in the analysis of the AMS algorithm). Instead, we do the following (the same strategy used in the AMS algorithm): we run $\Theta\left(\frac{1}{\epsilon^2}\right)$ independent copies of the Morris Counter, and output the average of the counters. It can be shown that this strategy gives us a good estimator.

5. Lower Bounds

5.1. **Yao's Minimax Principle.** In this section, we will explore a result from complexity theory: *Yao's Minimax Principle.*

Theorem 5.1 (**Yao's Minimax Principle**). *Let some problem be fixed. Then the following are equivalent.*

(1) *There exists a probability distribution $\mathcal{D}$ over inputs of length n of the problem such that every deterministic algorithm with query complexity $q(n)$ fails with probability $> 1/3$.*
(2) *For all (randomized) algorithms with query complexity $q(n)$, there exists an input of size n on which the algorithm fails with probability $> 1/3$.*

Remark 5.1.1. Intuitively, this theorem allows us to prove a global statement about all randomized algorithms to a problem by generating a probability distribution on inputs and analyzing that distribution on a fixed deterministic algorithm.

Proof. We will only prove (1) $\implies$ (2), as that will be all we need. Let $x_1, ..., x_t$ be all inputs of size n. Let $A_1, ..., A_t$ be all *deterministic algorithms* for this problem; note that in our case an algorithm is defined by the output it gives on inputs $x_1, ..., x_t$. Since only finitely many combinations of outputs are possible, we see that there are finitely many *deterministic algorithms* for these inputs.

Now, (1) implies that for all algorithms A_i above, it is true that

$$\mathop{\mathbf{P}}_{x\sim\mathcal{D}}[A_i(x) \text{ is wrong}] > \frac{1}{3}$$

So, this means that for every distribution $\mathcal{A}$ over the deterministic algorithms A_i, we have

$$\mathop{\mathbf{P}}_{A\sim\mathcal{A}, x\sim D}[A(x) \text{ is wrong}] > \frac{1}{3}$$

Now, any *randomized algorithm* over the inputs $x_1, ..., x_t$ is really some probability distribution over A_is; if we fix the random string that the randomized algorithm uses (i.e the output of the random coins), the algorithm follows a fixed path, which is the path taken by some deterministic algorithm.

So now fix a randomized algorithm, and interpret it as a distribution $\mathcal{A}$ on A_i. Now, there must be some input $x \in \{x_1, ..., x_t\}$ such that

$$\mathop{\mathbf{P}}_{A\sim\mathcal{A}}[A(x) \text{ is wrong}] > \frac{1}{3}$$

because otherwise, the previous inequality would not hold (this is a pigeonhole like argument). This proves (2), and proves this direction. ■

Remark 5.1.2. This theorem holds for non-adaptive algorithms and one-sided error algorithms as well.

Example 5.1. Let us now see an example of **Yao's Minimax Principle** in action.

Consider the problem for ϵ-testing 0^*, i.e given an input string of length n, we want to test whether the string is equal to 0^n, or if the string is ϵ-far from 0^* (which means it contains atleast ϵn 1s).

We claim that every randomized tester to solve the above problem with probability $\geq 2/3$ needs $\Omega(1/\epsilon)$ queries. To prove this, we construct a distribution $\mathcal{D}$ on n-length binary strings as follows.

(1) With probability 1/2, we choose the string 0^n.
(2) With probability 1/2, we do the following: we divide our string into $\frac{1}{\epsilon}$ continuous blocks, each of size ϵn. Call these blocks $B_1, ..., B_{\frac{1}{\epsilon}}$. Then, we sample $i \in [\frac{1}{\epsilon}]$ uniformly at random, and the block B_i to all 1s, and set everything else to 0.

It is clear that with the above distribution, we are sampling 0^n with half probability, and something which is ϵ-far from 0^* with half probability.

Now, let A be a deterministic algorithm for this problem that makes $< \frac{1}{3\epsilon}$ queries to the input. We will analyze the error probability of A on this distribution on the inputs.

(1) Note that if A sees all 0s in all of its queries, then A has to *accept*; if not, then it would mean that A rejects the all 0s string, i.e it fails with probability $\geq \frac{1}{2} > \frac{1}{3}$.
(2) Next, let us analyze the error probability of A when it *accepts* seeing only 0s in its queries. Clearly, A makes an error (if any) only on the inputs of the second kind in our distribution. Because A makes $< \frac{1}{3\epsilon}$ queries, it can *hit* only $< \frac{1}{3\epsilon}$ blocks among blocks $B_1, ..., B_{\frac{1}{\epsilon}}$. Now, because in this case A sees only 0s in its queries and since A is deterministic, the indices of the points that A sees are fixed; let these indices be $i_1, ..., i_q$, where $q < \frac{1}{3\epsilon}$ is the number of queries that A makes. Clearly, there are $> \frac{1}{\epsilon} - \frac{1}{3\epsilon} = \frac{2}{3\epsilon}$ blocks that don't contain these indices. Any of these blocks can be the all 1s block, and hence A accepts $> \frac{2}{3}$ of the inputs of the second kind. So, it follows that the failure probability of A is $> \frac{1}{2} \cdot \frac{2}{3\epsilon} \cdot \epsilon = \frac{1}{3}$ (and note that we are using the fact that the all 1s block is a uniformly random block).

So in all cases, we have shown that A fails with probability $> \frac{2}{3}$ on this distribution. By **Theorem** 5.1 (**Yao's Minimax Principle**), it follows that this problem needs $\Omega(1/3\epsilon)$ queries to be solved.