Midterm Solution Sketches

- Don't Panic.
- The midterm contains four problems (and one just for fun). You have 100 minutes to earn 100 points.
- The midterm contains 16 pages, including this one and 4 pages of scratch paper.
- The midterm is closed book. You may bring one double-sided sheet of A4 paper to the midterm. (You may not bring any magnification equipment!) You may not use a calculator, your mobile phone, or any other electronic device.
- Write your solutions in the space provided. If you need more space, please use the scratch paper at the end of the midterm. Do not put part of the answer to one problem on a page for another problem.
- Read through the problems before starting. Do not spend too much time on any one problem.
- Show your work. Partial credit will be given. You will be graded not only on the correctness of your answer, but also on the clarity with which you express it. Be neat.
- You may use any algorithm given in class without restating it—simply give the name of the algorithm and the running time. If you change the algorithm in any way, however, you must provide complete details.
- Good luck!

Problem #	**Name**	**Possible Points**	**Achieved Points**
1	True, False, Explain	24	
2	You are the computer	26	
3	Types of friends	25	
4	Three-way election	25	
Total:		100	

Student Number: _____

Problem 1. True, False, and Explain

This problem consists of a set of true/false questions. For each question, circle true or false and briefly (in *at most* one sentence) explain your answer.

Problem 1.a. Graph sketches. For the following three questions, assume $G = (V, E)$ is a graph that is given to you as a stream of edge additions and removals (i.e., it is presented as a stream in the usual way for this class).

In class we described an algorithm that uses a collection of (streaming) L0-samplers to decide whether a graph is connected. In each of the following three questions, assume you use the *exact same* data structure as for determining connectivity, stopping immediately after the stream is complete (and before the connectivity computation is performed). We call the data structure constructed in this way **a graph sketch**.

Using a graph sketch, once the stream is completed, you can use the graph sketch to find a spanning tree of the graph G (if it is connected).

TRUE **FALSE**

Solution: **True**: the connectivity argument, when it executes, actually constructs a spanning tree as part of the algorithm.

Using a graph sketch, once the stream is completed, you can use the graph sketch to answer any one query of the form "Is edge (u, v) in the graph G?"

TRUE **FALSE**

Solution: **False**: the graph sketch does not store all the edges in the graph, only a small subset of them.

Using a graph sketch, once the stream is completed, you can use the graph sketch to answer any one query of the form "Given a set of nodes $U \subseteq V$, is there any outgoing edge from the set U, i.e., is there any edge (u, v) where $u \in U$ and $v \notin U$?"

TRUE **FALSE**

Solution: **True:** This is exactly the guarantee of the graph sketch: the L0-sampler finds a non-zero entry in the vector, which represents an outgoing edge.

Problem 1.b. Sampling. The following three questions are related to sampling from either a graph or an array.

Assume you are a given a graph $G = (V, E)$ with n nodes and m edges. You choose a random node in the graph $u \in V$ and count the number of neighbors of u. Let X be the random variable representing the number of neighbors of the randomly chosen node. Then $\mathrm{E}[X] = m/n$.

TRUE **FALSE**

Solution: **False:** The correct calculation is $\mathrm{E}[X] = 2mn/$.

Assume you are given an arbitrary array $A[1..n]$ where each $A[i] \in [1..M]$ and $M > 10n$. Let

$$\bar{a} = \sum_{i=1}^{n} \frac{A[i]}{n}$$

be the average value of the array. Choose a random $i \in [1..n]$, and let X be the random variable representing $A[i]$. Then $\Pr[X \geq 4\bar{a}] \leq 1/4$.

TRUE **FALSE**

Solution: **True:** This follows from Markov's Inequality.

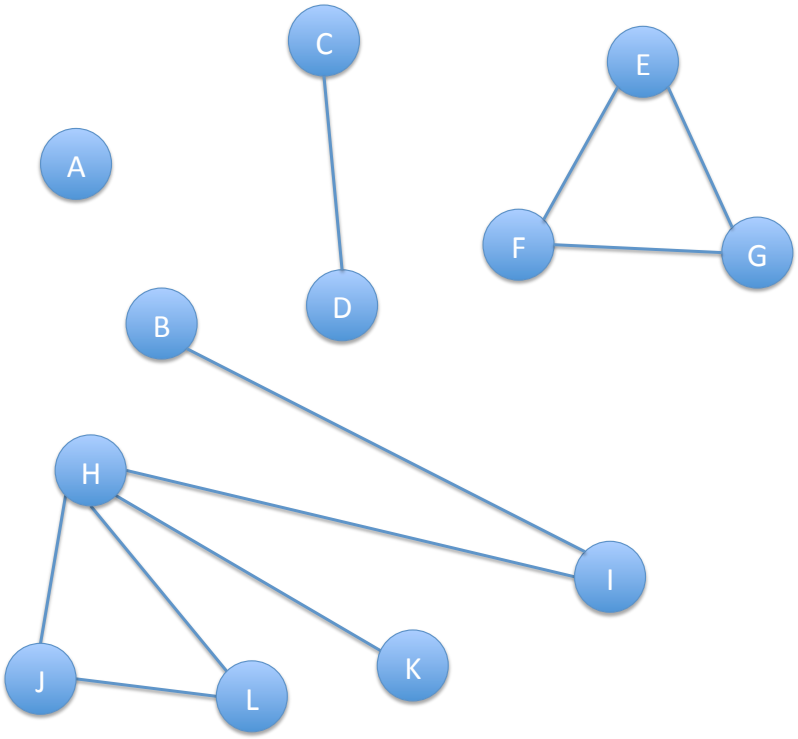
Assume you are given an arbitrary array $A[1..n]$ where each $A[i] \in [1..M]$ and $M > 10n$. Let

$$\bar{a} = \sum_{i=1}^{n} \frac{A[i]}{n}$$

be the average value of the array. Choose a random $i \in [1..n]$, and let X be the random variable representing $A[i]$. Then $\Pr[X \leq \bar{a}/4] \leq 1/4$.

TRUE **FALSE**

Solution: **False:** This is not always true. For example, consider an array where half the elements are M and half are 1.



Problem 2. Be the computer.

Problem 2.a. In this problem, you will execute the algorithm for finding the approximate number of connected components (using sampling) on the graph above, which has $n = 12$ nodes and maximum degree $d = 4$. Assume that $\epsilon = 1/2$. Use a version of the algorithm that succeeds with constant probability (e.g., probability 2/3).

During the execution, whenever you need to choose a random node in the graph, use the following sequence as your "random number generator":

C, K, F, D, C, K, F, D, C, K, F, D, C, K, F, D, C, K, F, D, C, K, F, D, C, K, F, D, C, K, F, D, ...

For example, the first time you need to choose a random node, use node C. The second time you choose a random node, use node K. Etc. If you run out of random nodes, then repeat the same sequence again. (Notice that this "random number generator" is not very random, and hence your execution may or may not find the "right" answer!)

Outer loop. The algorithm for finding the approximate number of connected components has an outer loop that determines the sample size. As a function of ϵ, n, and d, how many times do you execute the outer loop? (Use big-O notation.)

Loop count (big-O): **Solution:** $\Theta(1/\epsilon^2)$

Now compute the precise number of times (i.e., no big-O notation, no variables) that you will execute the outer loop. The constant depends on the probability of success, so there is no single correct value; as long as you choose something reasonable, you will get this part correct. However, to execute the algorithm and determine the final answer, you need to fix a number. (Remember, $\epsilon = 1/2$.)

Loop count (exact): **Solution:** 16, for example

Accumulated value. As the algorithm progresses, it accumulates a certain *sum* which is increased after every iteration of the outer loop. List below the value of the sum after the first four iterations. (Your execution may well consist of more than 4 iterations. Here we only ask about the first four iterations. **Explain briefly, to the right, how the new sum was calculated.** (You may use fractions; there is no need to generate a decimal value.)

After iteration 1: **Solution:** $1/2$

After iteration 2: **Solution:** $1/2 + 1/4 = 3/4$

After iteration 3: **Solution:** $1/2 + 1/4 + 1/3 = 13/12$

After iteration 4: **Solution:** $1/2 + 1/4 + 1/3 + 1/2 = 19/12$
A complete solution includes a brief description indicating that each iteration adds $max(\epsilon/2, 1/n(u))$.

Final answer. What is the *sum* after all the iterations is complete? What is the final number of connected components calculated by the algorithm? Give the precise answer: do **not** round to the nearest integer.

Final sum: **Solution:** $(19/12) \cdot 4 = 19/3$

Number of connected components calculated: **Solution:** $(19/3)(12/16) = 19/4 = 4.75$

Problem 2.b. Now consider the problem of finding the weight of the minimum spanning tree. Assume you have a graph with $n = 12$ nodes, maximum degree $d = 4$, and where all the edges have either weight 1 or weight 7. (The graph in this part is unrelated to the graph in the previous part.)

Recall that G_i is the graph below containing only edges of weight $\leq i$. Assume that graph G_1 has six connected components and graph G_7 has one connected component.

What is the weight of the minimum spanning tree? (You do not need to execute any specific algorithm from class. You do need to give the exact answer.) Explain your calculation briefly below.

Weight of MST: **Solution:** 41

There are two ways you might work this out. First, we say in class a formula for finding the MST:

$$n - W + sum_{i=1}^{6} CC(G_i).$$

In this case, that is: $12 - 7 + (6 \cdot 6) = 41$.

This can also be worked out directly . Since the graph G_1 has 6 connected components, you will need 5 edges of weight 7 at a cost of 35. The remaining $(11 - 5)$ edges will have weight 1, so those edges yield a cost of 6. That yields a total cost of 41.

Problem 3. Types of friends

Let $G = (V, E)$ be an undirected, unweighted graph representing a new social network: each node in the graph represents a person, and each edge (u, v) represents friendship between u and v. Assume there are n nodes and m edges. Each edge (u, v) has a *type*, for example, a friendship might be "family," or "close friend," or "high school friend," etc. (See the example on Page 9.) Let $t(u, v)$ be the type of edge (u, v). Assume every type is represented as an integer in the range $[1, T]$. Assume that $T = m$.

You are given the edges and their types as a stream (i.e., each item once, in arbitrary order)

$$(u_1, v_1, t_1), (u_2, v_2, t_2), \ldots, (u_m, v_m, t_m)$$

Each edge (and its associated type) will appear *exactly once* in the stream.

The goal is to determine the most common types of friendships in your network. For example, you might discover that "complete stranger" is the most common type of friendship!

More formally, given an integer k (that will be a parameter for your algorithm), your algorithm should, with probability at least 2/3, return a set of types containing:

- all the types t such that at least m/k edges have type t;
- no type t such that fewer than $m/(2k)$ edges have type t.

See the example on Page 9. In this problem (below and on the next page), you will describe an algorithm to solve this problem. As in all streaming algorithms, your primary goal is to use as little space as possible.

Describe briefly, in 1-2 sentences, the main idea behind your algorithm:

Solution: The main idea is to use a CountMin sketch (as on problem sets 2 and 4) to determine the frequency of each type in the stream. By setting the error $\epsilon = 1/(2k)$, the error in count will be within $m/(2k)$, as needed.

How much space does your algorithm use? (Give your answer as a function of n, m, and k. Use big-O notation.)

Space: **Solution:** $O(k \log m)$ counters or $O(k \log^2 m)$ bits

What is the update time for processing an edge in the stream? (Give your answer as a function of n, m, and k. Use big-O notation.)

Update time: **Solution:** $O(\log m)$

What is the query time for calculating the set of types, once the stream is complete? (Give your answer as a function of n, m, and k. Use big-O notation.)

Query time: **Solution:** $O(m \log m)$

(Briefly) describe any additional details of your algorithm and prove that it computes the right answer in the specified time and space:

Solution: The basic idea here is to use a CountMin sketch, where each entry in the vector is a type of friendship from 1 to T. Whenever you see the type t, the counters associated with t are incremented. (Since we are going to use the sketch as a block box, no further description of the details are needed, aside from the choice if ϵ.)

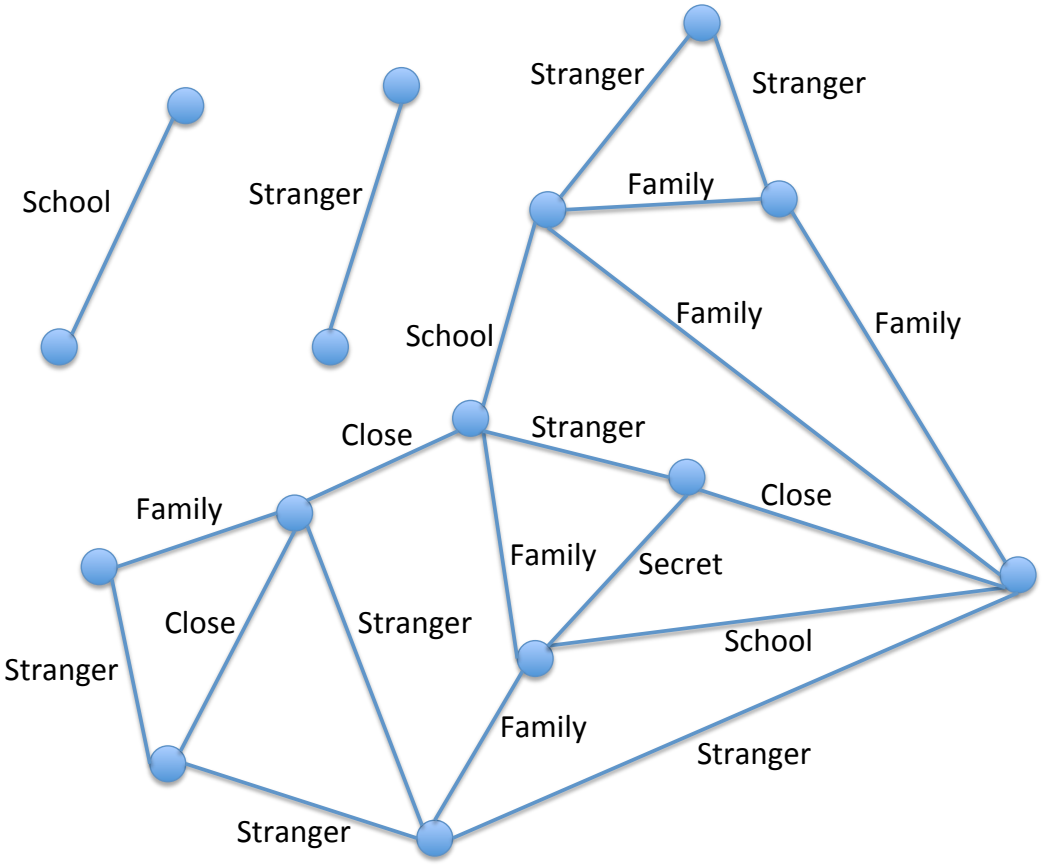
We should run the CountMin sketch with error $\epsilon = 1/(2k)$, which means that $B = 2/\epsilon = 4k$ and $A = O(\log T) = O(\log m$. This ensures that with probability at least $1-1/m^2$, we return for each t a count that is at least $edges(t)$ and at most $edges(t) + \epsilon T \leq edges(t) + m/(2k)$, where $edges(t)$ is the number of edges of type t.

Thus (and as we showed on the problem set) with probability at least $2/3$, for every type t, we can conclude that the count returned is at least $edges(t)$ and at most $edges(t)+m/(2k)$.

The algorithm should iterate over all the types and check the $count(t)$ returned by the CountMin sketch. It should return all types t where $count(t) \geq m/k$. For every type t that has more than m/k edges with type t, we will get $count(t) \geq m/k$ and return it in our set. For every type t that has $< m/(2k)$ edges with type t, we will get count at most $(m/(2k) + \epsilon m \leq m/(2k) + m/(2k) \leq m/k$, i.e., it will not be included. Therefore we conclude that it will return all the types that it should and none of the types that it shouldn't.

The total space used is $O(AB) = O(k \log m)$. The time for performing an update is $O(\log m)$, i.e., the cost of updating all the hash functions.

Implemented naively, the time for performing a query is $O(m \log m)$, i.e., the cost for looking up every single type. This can be improved by using a 1-sparse sampler (instead of a counter) for each item. That would reduce the query time to $O(k \log m)$.



Example for Problem 3: In this example, we have a graph with 21 edges. There are several different types of friendship edges: Family, Close, School, Stranger. There are 6 family edges, 3 close edges, 3 school edges, and 8 stranger edges.

When your algorithm runs with $k = 3$, it should return any friendship type that appears on at least 7 edges. Hence in this case, it *must* return "stranger." The algorithm may optionally return any friendship type that appears on at least 3.5 edges. Hence in this case, it *may* return "family." It may not return any of the other friendship types, as they are not sufficiently common.

Problem 4. Three-way elections.

Imagine you have an election where Alice, Bob, and Collin are running for President of Probabilityland. Let V be an array representing n votes: $V[1\,..\,n]$ where each $V[i] \in \{A, B, C\}$ represents a vote for Alice, Bob, or Collin.

Let x_A be the number of votes for Alice, i.e., the number of A's in the array. Let x_B be the number of votes for Bob, i.e., the number of B's in the array. We do not care about Collin. Collin is not allowed to be president.

The goal of this problem is to decide as efficiently as possible whether there are more votes for Alice or Bob in the array. (Remember, we do not care about Collin.) For a given parameter ϵ, give an algorithm with the following properties:

- If $x_A - x_B \geq \epsilon n$, return Alice, with probability at least 2/3.
- If $x_B - x_A \geq \epsilon n$, return Bob, with probability at least 2/3.
- Otherwise, return either Alice or Bob. (The election is too close to call.)

For example, assume $\epsilon = 1/4$, $n = 12$, and array B is as follows:

$$V = [A, B, A, B, A, B, C, B, B, C, B, C]$$

Here, $x_A = 3$ and $x_B = 6$. Since $\epsilon n = 3$, we conclude that $x_B - x_A \geq 3$ and return Bob. However, if there were one fewer vote for Bob, then we could return either Alice or Bob.

Your algorithm should run in sublinear time (e.g., $o(n)$ time), looking at as few votes as possible. That is, you should not actually compute the exact values of x_A and x_B. For partial credit, you can assume that Collin gets zero votes, i.e., all the votes in the array are either A or B.)

Describe briefly, in 1-2 sentences, the main idea behind your algorithm:

Solution: We already saw on the problem set how to estimate the fraction of 1's in an array. Use the same technique to estimate the fraction of A's and B's in the array, using error rate $\epsilon/4$, and choose A or B as the winner based on which has a larger fraction.

What is the query complexity of your algorithm? (That is, how many votes in the array does it look at? Give your answer as a function of n and ϵ. Use big-O notation.)

Queries: **Solution:** $O(1/\epsilon^2)$

(Briefly) describe any additional details of your algorithm and prove that it computes the right answer in the specified time:

Solution: This problem can be solved using the sampling algorithm from Problem Set 2. Recall that we know how to decide, via sampling, the fraction of A's in the array, plus or minus $\epsilon/4$, using $O(1/\epsilon^2)$ samples. The same holds for determining the fraction of B's in the array. (Notice that for the sampling algorithm, we are using an error rate of $\epsilon/4$ because we are going to compare the two fractions, so we need smaller error; anything $< \epsilon/2$ will work.)

Let x'_A be the estimated number of A's in the array determined via sampling, where $|x'_A - x_A| \leq \epsilon n/4$. Let x'_B be the estimated number of B's in the array determined via sampling, where $|x'_B - x_B| \leq \epsilon n/4$.

If $x'_A - x'_B \geq \epsilon n/2$, then return x'_A. If $x'_B - x'_A \geq \epsilon n/2$, then return x'_B. Otherwise, return either one.

We can see that this is correct because if $x_A - x_B \geq \epsilon n$, then $x'_A - x'_B \geq (x_A - \epsilon n/4) - (x_B + \epsilon n/4) \geq (x_A - x_B) - \epsilon n/2 \geq \epsilon n/2$. Similarly, if $x_B - x_A \geq \epsilon n$, we conclude that $x'_B - x'_A \geq \epsilon n/2$.

The problem can also be solved directly via sampling, e.g., by sampling a set of S locations in the array. (You have to be a little careful, because the number of A's and the number of B's in the sample are not independent random variables. However, each is close to its expectation, and you can sum the two probabilities of error via a union bound.)

Problem 5. (Just for fun!) [0 points]

Last week, I received an anonymous invitation to a strange party at a secret location. I went with my wife. There we met 4 other couples, who had also received anonymous invitations. Thus were ten of us at the party, each of whom had come with a partner.

Luckily, some of us at the party already knew each other. We each shook hands with the people we did not know. (People who already knew each other did not shake hands.) We all were wondering why we were here. To gain some information, I asked each of the nine other people in the room how many hands they had shook. Surprisingly, I got nine different answers! Each person had shaken a different number of hands!

How many people at the party did my wife not know before the party?

Scratch Paper

Scratch Paper

Scratch Paper

Scratch Paper