Midterm Solution Sketches

- Don't Panic.
- The midterm contains six problems (and one just for fun). You have 100 minutes to earn 100 points.
- The midterm contains 20 pages, including this one and 3 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	Similarity Tester	22	
2	Case Study	12	
3	Parallel Heavy Hitters	10	
4	It Takes Three	20	
5	The Enemy of My Enemy	20	
6	Odds and Ends	16	
Total:		100	

Student Number: _____

Problem 1. Twice the Arrays, Twice the Fun

Assume you have two arrays A and B of size n containing 0's and 1's. Each array has exactly k cells containing a 1 and $(n-k)$ cells containing a 0, for some $k < n/4$.

Problem 1.a. True, False, and Explain:

Which of the following statements are always true? For each part, explain why in less than ten words. (Since we are interested in worst-case analysis, you may assume that both arrays A and B are prepared by an adversary.)

Let i be an index chosen uniformly at random from $[1, n]$.
Let j be an index chosen uniformly at random from $[1, n]$.

$$\Pr[A[i] = B[j]] \leq k/n$$ **TRUE** **FALSE**

Solution: False. The correct probability is $(k/n)^2 + (1 - k/n)^2 > k/n$.

Let i be an index chosen uniformly at random from $[1, n]$.

$$\Pr[A[i] = B[i] = 1] \geq 1/n^2$$ **TRUE** **FALSE**

Solution: False. There may be no indices i where $A[i] = B[i] = 1$.

Let i be an index chosen uniformly at random from $[1, n]$.
Let $X = A[i] - B[i]$.

$$\mathrm{E}[X] = 0$$ **TRUE** **FALSE**

Solution: True, by linearity of expectation.

Let S be a set of indices of size $s = n/k$, where each $i \in S$ is chosen independently and uniformly at random from $[1, n]$. Let X be the number of 1's in the set S.

$$\Pr[X \geq 5] \leq 1/5$$

TRUE **FALSE**

Solution: True, by Markov's Inequality.

Let $s = 2/\epsilon^2$. Let S be a set of indices of size s, where each $i \in S$ is chosen independently and uniformly at random from $[1, n]$. Let $X = \sum_{j \in S} (n/k) A[i]$.

$$\Pr\left[|X - 2/\epsilon^2| > 2/\epsilon\right] \leq 1/3$$

TRUE **FALSE**

Solution: False. The random variable X is not bounded by some constant, so you have to scale the Hoeffding Bound. As a counterexample, consider the case where $k = 1/n$, and notice that for any constant ϵ, it is false as n gets large.

Problem 1.b. Two-Array Similarity Tester:

Here we continue to test the two arrays A and B, each with k cells with value 1 and $(n-k)$ cells with value 0. Give an algorithm for testing whether the two arrays are identical. Your algorithm should guarantee that with probability at least 2/3:

- It returns TRUE if the two arrays are (exactly) the same.
- It returns FALSE if the two arrays are ϵ-different, i.e., there are at least $> \epsilon n$ entries for which the two arrays differ.
- Otherwise, it can return TRUE or FALSE.

Two-Array Similarity Tester Algorithm:

Solution:

- Repeat $2/\epsilon$ times:
  - Choose an index j. If $A[j] \neq B[j]$ return FALSE.
- Return TRUE.

Prove that your algorithm is correct.

What is the query complexity: $O(1/\epsilon)$

Solution: Analysis: If the two arrays are the same, it always returns TRUE. If the two arrays are ϵn different, then there are ϵn indices for which the arrays differ. In that case, there is a $\epsilon n/n$ probability of choosing a differing index, and hence it returns TRUE with probability at most $(1-\epsilon)^{2/\epsilon} \leq 1/e^2 \leq 1/3$.

Problem 2. Case Study

Below, we consider two case studies. In each case, please indicate whether the specified algorithm is a good choice for the problem, in terms of meeting the desired goals and in the context of other available options. Consider precisely the algorithm from class (not an optimized version).

Problem 2.a. Alice is analyzing a social network presented as a graph with n nodes, where each node represents a person and each edge represents a friendship relationship. Some people have a very large number of friends, but the average number of friends of everyone in the network is 8. The graph is presented as a stream of edges, and Alice has only a limited amount of space. Alice wants to find the best possible approximation to the diameter of the graph. (Alice is not worried about the time complexity, only the space.)

Alice first computes a $\log(n)$-stretch spanner, using the algorithm from class for finding approximate distances in a stream. Then, she considers every pair of nodes and returns the maximum pairwise distance as an approximation of the diameter.

Does this approach return the approximate diameter? YES NO
(Either a constant factor approximation or a $O(\log n)$-factor approximation is acceptible to Alice. A n^ϵ-factor approximation is not acceptable.)

Is this a good approach? YES NO

Explain your answer:

Solution: This is *not* a good solution. It does indeed find an approximation of the diameter as specified. However, for this graph, it is inefficient. In total, the graph has only $4n$ edges in total, and so Alice can store the entire graph using on $8n \log n$ bits (i.e., two identifiers for each edge). This is about the same amount of space as the spanner algorithm from class, and provides a better approximation, i.e., the exact diameter.

Problem 2.b. Bob is analyzing an airline route network represented as a graph with n nodes. Each node represents a city, and there are about 40,000 cities in the dataset. Each edge represents a possible trip, and the maximum degree of the graph is 10. For each edge connecting cities i and j, assign them a weight $w(i, j)$, which is the time taken on the best route from i to j. The time calculated is end-to-end, including the time spent waiting to change planes. Time is measured in hours, where the shortest route is one hour, and the longest flight is 52 hours (involving four different flights).

Bob wants to estimate the weight of the MST as fast as possible. He wants an estimate that is within 1% of the correct answer. He therefore chooses to use the sublinear time approximation algorithm for finding an MST discussed in class.

Does this approach return the approximate MST weight? YES NO

Is this a good approach? YES NO

Explain your answer:

Solution: This is not a good solution. It does work. However, recall that the running time for the algorithm presented in class depends on dW^4/ϵ^3. For $d = 10$, $\epsilon = 0.01$, and $W = 52$, this value will significantly exceed the cost of a standard MST algorithm. Even were you to use the best known sublinear time approximation algorithm, you will not see any significant benefits until $\epsilon > 0.1$. At $\epsilon = 0.01$, you might as well just run Kruskal's Algorithmn.

Problem 3. Parallel Heavy Hitters.

We have seen several different algorithms this semester for solving the Heavy Hitters Problem: for a stream of M integers with values ranging from 1 to N, given some $\epsilon > 0$, output every item that appears at least $2\epsilon M$ times, and never output an item that appears less than ϵM times. The result should be correct with probability at least $1 - 1/N$.

Assume you have p different servers, each of which has $O(\epsilon^{-1} \log N \log M)$ space. The stream will be split among the p servers, with each processing M/p items.

In more detail, before the streams begin, the servers may communicate. Then, each server processes a separate stream of M/p integers. During the processing of the stream, the servers may not communicate. At the end, one designated FINAL server must report all the heavy hitters for the combined stream, i.e., all the items that appeared at least $2\epsilon M$ times, and no item that appeared less than ϵM times, collectively over all the streams. Your algorithm should be correct with probability at least $1 - 1/N$.

Explain how you can modify/use one of the algorithms from class to solve this problem. Be sure to specify all the necessary parameters so that the algorithm provides the correct guarantees.

Solution: The Count-Min sketch seen on the problem set solves this problem. In more detail, each server uses an array of $(2\log(N), 2/\epsilon)$ counters $C_{i,j}$. The servers share a collection of $2\log(N)$ hash function $h_1, h_2, \ldots$ that map from $[1, N] \to [1, 2/\epsilon]$. When receiving an item x, the server increments counters $C_{i,h_i(x)}$, for all i.

At the end, all the servers send their counts to the FINAL server, which then sums the counts and returns every value x such that $\min_i C_{i,h_i(x)} \geq 2\epsilon M$. Notice that the result is identical to if one server had executed the entire algorithm by itself.

Specifically, for each i and for each x, $\mathrm{E}\left[C_{i,h_i(x)} - \mathit{count}(x)\right] \leq \epsilon M/2$, and so the probability that the error is more than ϵM is $\leq 1/2$. The probability that the error is bad for all $2\log M$ values of i is thus at most $1/2^{2\log M} = 1/N^2$. Taking a union bound over all possible values yields a probability of error at most $1/N$, as desired.

Problem 4. Sometimes, it takes three.

The Pointy-haired Boss has hired Dogbert to improve the organization of his workforce, specfically, to divide the workers into teams of three (known herein as "triads.") The workforce consists of n people $p_1, p_2, \ldots, p_n$.

Dogbert is presented with a stream, where each element in the stream is an (unordered) pair (p_i, p_j) of two people who can work together on a team. Assume that each pair (p_i, p_j) appears at most once in the stream.

At the end of the stream, Dogbert's algorithm should output a set of triads (p_i, p_j, p_k) where we know that (p_i, p_j) and (p_j, p_k) are both legal team pairs. (Note that it is not necessary for (p_i, p_k) to be a legal pair, since p_j can always act as a mediator.) No person p_i should be part of more than one triad.

For example, you might have a stream consisting of:

`(Alice, Wally), (Alice, Dilbert), (Asok, Colbert), (Ted, Dilbert), (Asok, Phil)`

In this case, Dogbert might output the triads: `(Wally, Alice, Dilbert), (Colbert, Asok, Phil)`.

Dogbert develops the following algorithm for processing the stream:

---

1. For each person p_k: set $T[k] = \emptyset$.
2. Set $S = \emptyset$.
3. For each pair (p_i, p_j) in the stream:
   (a) If p_i or p_j is already in a triad in S, then skip this item.
   (b) Otherwise, if $T[i] = \ell$, then add (p_ℓ, p_i, p_j) to S and delete ℓ, i, j from all $T[.]$.
   (c) Otherwise, if $T[j] = \ell$, then add (p_ℓ, p_j, p_i) to S and delete ℓ, i, j from all $T[.]$.
   (d) Otherwise, set $T[i] = j$ and $T[j] = i$.
4. Return S.

After initializing the T array and the set S to be empty, Dogbert processes each new pair (p_i, p_j) in the following manner. If either person in the pair is already in a triad in S, then Dogbert continues to the next pair. Otherwise, Dogbert checks $T[i]$ and $T[j]$, which contain potential third parties for the triad. If he finds a legal third, then the triad is added to S and all three people in the triad are deleted from T. Otherwise, if no triad is possible, then Dogbert sets $T[i] = j$ and $T[j] = i$ to indicate that j is a possible third party for a triad involving i and vice versa.

Problem 4.a. Prove that the output is correct, i.e., that every triad is valid and that no person appears in more than one triad.

Solution: To show that the solution is correct, we need to show two things: every triad is valid, and no person is in more than one triad. First, notice that $T[i] = \ell$ only if there has been a previous pair (i, ℓ) or (ℓ, i). Therefore if we see pair (p_i, p_j) and $T[i] = \ell$, it is valid to output (p_ℓ, p_i, p_j). Second, notice that if p_i is added to S, then it is deleted from all T. From that point on, any pair involving i will be skipped.

Problem 4.b. Prove that the output is maximal, i.e., that if there are two pairs (p_i, p_j) and (p_j, p_k) that appear in the stream, then at least one of the people p_i, p_j, p_k is included in a triad in the output set S.

Solution: We now prove that if there are two pairs (p_i, p_j) and (p_j, p_k) that appear in the stream, then at least one of the people p_i, p_j, p_k is included in the set S. Assume, without loss of generality, that pair (p_i, p_j) appears before pair (p_j, p_k). Assume for the sake of contradiction that none of the three people appears in the set S.

When pair (p_i, p_j) appears in the stream, since neither is in S, it is not skipped. It also must not be the case that $T[i] = \ell$ or $T[j] = \ell$, as otherwise p_i and p_j would be added to S. Thus, after the pair is processed, we know that $T[i] = j$ and $T[j] = i$. When pair (p_j, p_k) is processed, again it is not skipped because both p_j and p_k are not in S. In this case, however, we know that $T[j] = i$, and so we add (p_i, p_j, p_k) to S, contradicting our assumption.

Thus, we conclude that the output is maximal.

Problem 4.c. Prove that the output is a 3-approximation of optimal. That is, if the largest output set S^* contains s triads, then the output set S contains at least $s/3$ triads.

Solution: We prove that it is a 3-approximation of optimal using a charging argument. For each triad t in S^*, we charge one unit to a triad in S such that every triad in S has at most three units charged to it. Let $t = (p_i, p_j, p_k)$ be a triad in S^*.

- If $t \in S$, then charge one unit to t.
- If p_i is in a triad t' in S, then charge one unit to t'.
- If p_j is in a triad t' in S, then charge one unit to t'.
- If p_k is in a triad t' in S, then charge one unit to t'.

We have already argued (in the previous part) that the output set S is maximal. Since (p_i, p_j, p_k) appears in S^*, we can conclude that (p_i, p_j) and (p_j, p_k) appear in the stream (otherwise it would not be a valid triad), which means that at least one of the people p_i, p_j, p_k is included in a triad in the output set S. Thus, we can always charge for t in one of these four ways.

We also observe that a triad (p_i, p_j, p_k) can only be charged at most three times: once for a triad in S^* containing p_i, once for a triad in S^* containing p_j, and once for a triad in S^* containing p_k. (Each of these people can appear in at most one triad in S^*.)

Finally, we conclude that if there are x items in S, then the total number of triads charged to some triad in S is $3x$, i.e., there are at most $3x$ items in S^*. This implies that the algorithm is a 3-approximation of optimal.

Problem 5. The enemy of my enemy is my friend?

Socialogist Dr. I. Haitewe is investigating the claim that "the enemy of my enemy is my friend." Luckily, he has access to a large collection of data from the popular anti-social networking site EnemyBook. He is given a large graph $G = (V, E)$ stored as an adjacency list indicating which people on EnemyBook are enemies. There are n nodes in the graph, where each node represents a user, and each edge (i, j) represents a pair of people that are enemies. Each person has at most d enemies. (After that, they are kicked out of EnemyBook for being too hateful.)

Dr. Haitewe observes that if the graph G is triangle-free, then it is possible that for each user, the enemy of their enemy is their friend! If, however, there is a triangle in the graph, then this is impossible. Thus Dr. Haitewe wants to test if the graph G is triangle-free.

Dr. Haitewe decides that a graph G with n nodes is ϵ-far from triangle-free if you have to delete ϵdn edges from the graph in order to remove all the triangles.

Help Dr. Haitewe by developing an algorithm that, with probability at least 2/3:

- Returns FREE if the graph is triangle-free.
- Returns FAR if the graph is ϵ-far from triangle-free.

Your goal is to find an algorithm that minimizes the running time.

Problem 5.a. Describe your algorithm and explain how it works.

Solution:

- Repeat $s = 2/\epsilon$ times:
  - Choose an edge $e = (u, v)$ at random.
  - Check whether u and v share any common neighbors.
  - If u and v have a shared neighbor, return FAR.
- Return FREE.

Problem 5.b. Prove that your algorithm is correct.

Solution: If the graph is triangle-free, then for every selected edge there are no shared neighbors, and hence the algorithm correctly returns FREE. If the graph is ϵ-far from triangle-free, then there are at least ϵdn edges that are part of triangles. In total, there are no more than dn edges in the graph. Thus the probability of choosing such a edge is at least $\epsilon dn/dn \geq \epsilon$. The probability that no edge in the sample is part of a triangle is $(1-\epsilon)^{2/\epsilon} \leq e^{-2} \leq 1/3$. Thus with probability at least $2/3$, the algorithm returns FAR.

Problem 5.c. What is the running time of your algorithm? $O(d/\epsilon)$

Solution: The running time is $O(d/\epsilon)$, since the sample is of size $O(1/\epsilon)$ and each edge takes time $O(d)$ to check if it is part of a triangle.

Problem 6. Odds and Ends

Imagine you are given a stream of integers from the range $[1, N]$. The goal is to decide whether at least one item in the stream appears an odd number of times.[1]

We are going to focus for now on the following version of the problem, designing an algorithm with the following properties:

- If every item in the stream appears an even number of times, output EVEN.
- If at least one item appears an odd number of times, output ODD.

Your algorithm should be correct with probability at least $1 - 1/N$.

Hint: For every integer i we know that $i \oplus i = 0$, where $\oplus$ is the bitwise XOR operation, which is commutative and associative. (Recall that for two bits: $(0 \oplus 0) = (1 \oplus 1) = 0$, and $(0 \oplus 1) = 1$.)

Problem 6.a. Let h be a hash function that randomly maps $[1, N]$ to $[1, N]$. (You may treat h as a uniform random function.) Let S be a set of distinct integers. Prove that if S contains at least one item, then with probability at least $1 - 1/N$: $\oplus_{j \in S}(h(j)) \neq 0$. Here $\oplus$ represents the XOR operator, which is applied to the hash of every element in S.

Solution: Let X be the XOR of the hash of all the elements in S except for a single item i. Thus $\oplus_{j \in S}(h(j)) = X \oplus h(i)$. Notice that there is only exactly one value of $h(i)$ that will cause this to be zero, i.e., the case where $h(i) = X$. Otherwise, $\oplus_{j \in S}(h(j)) \neq 0$. Since h is a random function, there is at most a $1/N$ probability that $h(j) = X$.

[1] For example, consider the problem of deciding if a graph has an Eulerian Cycle, i.e., a tour that crosses every edge exactly once. A connected graph has an Eulerian Cycle if and only if every node has even degree. If the graph is given as a stream of edges, you might use this algorithm to decide whether or not it has an Eulerian Cycle.

Problem 6.b. Describe your algorithm and explain how it works. Try to minimize the amount of space used.

Solution: Let $H = \oplus_{j \in S}(h(j))$ be the XOR of the hash of all the integers in the stream. If $H = 0$, return EVEN. If $H \neq 0$, then return ODD.

Problem 6.c. How much space does your algorithm use? $O(\log N)$

Problem 6.d. Prove that your algorithm is correct.

Solution: If every item appears an even number of times, then $H = 0$ since any even number of items always XORs to zero. Assume that there is at least one odd item. Let S be the set containing each odd item once. All the other items in the stream (and their hashes) XOR to zero. Thus we return ODD if and only if $\oplus_{j \in S} = 0$. We have already shown, however, that the probability of $\oplus_{j \in S} = 0$ is at most $1/N$.

Problem 7. (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