CS358: Applied Algorithms **Name:** _____

This exam contains 10 pages (including this cover page) and 8 problems. (The final problem is a bonus problem just for practice—I think the actual midterm will be a little shorter than this.) Check to see if the exam you received contains all the pages. **This is a 75 minute exam and the total number of points is 100.**

- Please enter your name at the top.
- Please read through the entire exam before attempting any problem.
- If you need clarification on a question you may ask the instructor.
- You may use any results we've seen in class (you do not need to reprove any results we saw in class unless explicitly asked). You may also refer to algorithms we have seen in class (for example, you may say "run DFS on the graph").
- All scratch work must be done on the scratch paper provided along with this exam—either the sheets at the end, or the blank reverse side of each page. You are not allowed to write on a personal notebook during the exam.
- You are allowed a 2-page 8x11 2-sided "cheat sheet." You can use it for reference, but you should not use it as scratch paper (use the scratch paper provided). You may collaborate on creating this with other students, and you do not need to hand it in.
- For each problem, the specifics of what it's asking for (a proof, an algorithm, a running time, etc.) is written in **bold**. Please check the bold instructions on every question to make sure you are answering it correctly!
- If you need additional space for a solution, you may use the scratch paper at the end of the exam, or the back of any sheet. Be sure to note where your solution is if you do this.
- **Remember that this exam has partial credit.** If you are unsure of the full solution, it's still a good idea to write down what you know.

1. (10 points) Which of the following is most expensive on most modern computers? **Circle** the most expensive operation. You do not need to explain your answer.

   a) A cache miss b) A branch misprediction c) Dividing two integers

   A cache miss

2. (10 points) **What is the Jaccard similarity** of the following two sets A and B? You do not need to explain your answer.

   $$A = \{a, b, c\} \qquad B = \{b, c, d, f\}$$

   $A \cap B = \{b, c\}$ and $A \cup B = \{a, b, c, d, f\}$, so the Jaccard similarity is 2/5.

3. (10 points) Let's say I am using a Locality-Sensitive Hash to find a close pair of items like in Assignment 4, and I want to *decrease* the number of repetitions necessary to find the close pair. Which of the following is the best method to do so? **Please circle one, and explain your answer in one-two sentences.**

   a) Make the number of concatenated hashes smaller

   b) Make the number of concatenated hashes larger

   c) Make the number of buckets in the hash table smaller

   d) Make the number of buckets in the hash table larger

   The best answer is to make the number of concatenated hashes smaller. The probability that two close elements have the same signature is j_1^k; decreasing k (the number of concatenated hashes) makes this number larger.

   Making the number of buckets in the hash table smaller will make the close elements slightly more likely to land in the same bucket: if there are B buckets, two elements with different signatures will be in the same bucket with probability B. But this is not as effective as concatenating the hashes for two reasons. First, we would need to drastically reduce the number of buckets to have the same impact as concatenating fewer hashes; second, this impacts close and far pairs equally.

4. (20 points) Let's say we want to find an item X in a sorted array of length n. For simplicity, assume that X is actually located in the array, at some arbitrary position. Here are two algorithms to do this:

   - **linear search**: iterate through each element of the array; if we ever see an element equal to X we return it. This takes $O(n)$ time.
   - **binary search:** look at the middle element of the array; if it is equal to X return it. If it is $< X$, recurse on the right side of the array; if it is $> X$ recurse on the left side of the array. This takes $O(\log n)$ time.

   For each of the following four questions, please **answer the question using big-O notation**; then **give a 1-sentence explanation** of your answer.

   What is the *cache efficiency* of linear search in the external-memory model? You should assume that n is much larger than M or B.

   > $O(n/B)$. We do a linear scan; each time I have a cache miss when accessing item i, we will not have a further cache miss until we access item $i + B$.

   What is the *cache efficiency* of binary search in the external-memory model? You should assume that n is much larger than M or B.

   > $O(\log_2 n/B)$. Each access to the "middle element" will likely cause a new cache miss until the array has size $\leq B$, after which the remaining accesses will be within the same cache line.
   >
   > [Note that since B is much less than n (being more specific, let's assume that $B < \sqrt{n}$, $\log_2 n = \Theta(\log_2 n/B)$—they are the same answer asymptotically and are both correct here. –Sam]

   About how many branch mispredictions will linear search incur while searching for X?

We check each item to see if it is equal to X. This will almost always evaluate to "false." Therefore, we will only have $O(1)$ branch mispredictions—we will get roughly n branches where the prediction is correct, and a constant number (when we find X, or in the first few "false" answers) where the prediction is incorrect.

About how many branch mispredictions will binary search incur while searching for X?

$O(\log n)$—each time we look at the middle element, we are equally likely to find that it is $\geq X$ and $\leq X$.

5. (20 points) You are given an array A of n integers. Your goal is to find four indices a, b, c, d such that $A[a] + A[b] = A[c] + A[d]$. We can solve this problem in $O(n^4)$ time by trying all n values for a, b, c, and d.

   Give a faster algorithm using the *meet-in-the-middle* strategy we used for the two towers problem. Your algorithm should use $O(n^2)$ space, and should run in either $O(n^2)$ or $O(n^2 \log n)$ time (either time bound is sufficient for full credit).

   First, we consider each pair of indices a and b. For each such a and b, we store the tuple $A[a] + A[b], (a, b)$ in a table. This table has $O(n^2)$ entries, so requires $O(n^2)$ space; the table can be built in $O(n^2)$ time.

   Then we sort the table by the first value, $A[a] + A[b]$. This takes $O(n^2 \log n)$ time.

   Now, we iterate through all possible indices c and d. For each such pair, we calculate $A[c] + A[d]$. Then, we binary search in the table to see if there is an a and b with $A[a] + A[b] = A[c] + A[d]$; if we find such a pair in the table, we can return a and b (from the table) and c and d. We do an $O(\log n)$-time binary search for $O(n^2)$ pairs c, d for $O(n^2 \log n)$ time.

   [Alternatively, we could store each tuple in a hash table, which would give us $O(n^2)$ total expected time. –Sam]

6. (20 points) Consider a stream of elements, but now each element of the stream is being *inserted* or *deleted*. The total number of times each element occurs is the number of times it is inserted, minus the number of times it is deleted.

   For example, in the following stream:

   Ins. 1, Ins. 2, Del. 1, Ins. 3, Ins. 2, Del. 2,

   1 appears a total of 0 times, 2 and 3 each appear once.

   We can naturally create a count-min-sketch like data structure for this kind of stream. We keep an array of w counters[1] and a hash h. (We'll only do "one row"—so only one hash h.) When an element e is inserted, we increment the counter at $h(e)$; when an element e is deleted, we decrement the counter at $h(e)$. Assume that the stream consists of N_i total insertions and N_d total deletions.

   For some query q, let $\hat{o}_q$ be the value returned by the data structure, and let i_q and d_q be the *true* number of times that q is inserted and deleted (respectively) in the stream. **Below, give the value of** $\mathrm{E}[\hat{o}_q]$ (i.e. the expected value returned by the data structure) in terms of i_q, d_q, w, N_i, and N_d. **Give the steps you use** to obtain your result.

   We obtain the value $\hat{o}_q$ by looking at counter $h(q)$ in the array. We know that the counter will be incremented i_q times due to q being inserted, and decremented d_q times due to q being deleted.

   Other elements will be inserted and deleted, however. For each element x, if $h(x) = h(q)$, then $h(q)$ will be incremented when x is inserted, and deleted when x is deleted.

   Therefore [I'm doing this fairly formally; you need not be so formal for full credit –Sam]

$$\hat{o}_q = \sum_{x \mid h(x)=h(q), x \neq q} \#\text{ times } x \text{ is inserted} - \#\text{ times } x \text{ is deleted}$$

   Let's rewrite this using random variables. Let $X_x = \#$ times x is inserted $-$ $\#$ times x is deleted if $h(x) = h(q)$, and $X_x = 0$ if $h(x) \neq h(q)$. Then we can rewrite $\hat{o}_q = i_q - d_q + \sum_x X_x$, so $\mathrm{E}[\hat{o}_q] = \mathrm{E}[i_q - d_q + \sum_{x \neq q} X_x]$.

   By linearity of expectation, this means that $\mathrm{E}[\hat{o}_q] = i_q - d_q + \sum_{x \neq q} \mathrm{E}[X_x]$. By definition,

$$\mathrm{E}[X_x] = \frac{1}{w} \cdot (\#\text{ times } x \text{ is inserted} - \#\text{ times } x \text{ is deleted})$$

[1] We used $w = e/\varepsilon$ for the CMS in class, but we'll just call it w here to keep things easier to work with.

Substituting,

$$\begin{aligned}\mathrm{E}[\hat{o}_q] &= i_q - d_q + \sum_{x \neq q} \frac{1}{w} \cdot (\#\text{ times } x \text{ is inserted} - \#\text{ times } x \text{ is deleted}) \\ &= i_q - d_q + (N_i - i_q - N_d + d_q)/w.\end{aligned}$$

[I am looking for linearity of expectation to be mentioned here (if it is used) for full points. –Sam]

[Note that in our analysis in class, we just used N_i rather than $N - i_q$. This is because the analysis in class was just an upper bound, and here we are calculating the exact expectation. –Sam]

7. (20 points) Consider a simple cuckoo filter with n items, cn slots (this is a basic cuckoo filter, with 1 "slot per bucket"), fingerprints of length f, and two independent hash functions h_1 and h_2.

   Give an upper bound on the false positive probability of the cuckoo filter in terms of c and f.

   The cuckoo filter returns a positive on a query $q \notin S$ if $h_1(q)$ or $h_2(q)$ contains a fingerprint equal to $f(q)$. We will examine the probability of each and combine them using a nion bound.

   First, consider the probability that the fingerprint in $h_1(q)$ matches. For this to happen, $h_1(q)$ needs to contain a fingerprint; this occurs with probability $1/c$. Then, if there is a fingerprint contained, we have a false positive when the fingerprint stored matches the fingerprint of q; this occurs with probability $1/(2^f - 1)$. [The -1 is because the 0 fingerprint is not used; I think I'd probably give full credit with this small term missing –Sam] Multiplying, we have a false positive for h_1 with probability $\frac{1}{c(2^f-1)}$.

   The exact same analysis applies for h_2; we have a false positive with probability $\frac{1}{c(2^f-1)}$.

   Adding together using the union bound, there is a false positive with probability $\frac{2}{c(2^f-1)}$.

8. (10 points) The following is a **bonus problem**—I think it is too similar to other problems on the practice exam so I removed it, but it's not a bad use to use it as practice.

   You are given a *sorted* array A of n integers. The goal is to solve the following problem. We are given a target value T, and we want to find indices i and j such that $A[i]+A[j] = T$.

   Because the array is sorted, we can solve this problem using the following strategy. We keep two pointers i and j; to start, $i = 0$ and $j = n-1$. We compare $A[i]+A[j]$ to T. If $A[i]+A[j] = T$, we are done. If $A[i]+A[j] < T$, we increment i and continue. Otherwise $A[i]+A[j] > T$, so we decrement j and continue. We keep going until $i > n-1$ or $j < 0$, at which point we return that there is no solution.

   To summarize, i starts at 0, j starts at $n-1$, and at each iteration of the algorithm either i is incremented or j is decremented.

   Analyze this algorithm in the external memory model. Give the number of cache misses using big-O notation. You should assume that n is much larger than M or B.

   If we have a cache miss when accessing element i, we do not have a cache miss for the next B values of i. Same for j: when we have a cache miss when accessing element j, we do not have a cache miss for the next B values of j. (One cache line for each is kept in cache.)

   Therefore, the total number of cache misses is $O(n/B)$.

Scratch Paper