CS 124 Homework 5: Spring 2021

Your name: Sean Ty

Collaborators:

No. of late days used on previous psets: 0
No. of late days used after including this pset: 0

Homework is due Wednesday 2021-04-07 at 11:59pm ET. You are allowed up to **twelve** (college)/**forty** (extension school) late days through the semester, but the number of late days you take on each assignment must be a nonnegative integer at most **two** (college)/**four** (extension school).

Try to make your answers as clear and concise as possible; style will count in your grades. Be sure to read and know the collaboration policy in the course syllabus. Assignments must be submitted in pdf format on Gradescope. If you do assignments by hand, you will need to scan in your results to turn them in.

For all homework problems where you are asked to design or give an algorithm, you must prove the correctness of your algorithm and prove the best upper bound that you can give for the running time. Generally better running times will get better credit; generally exponential time algorithms (unless specifically asked for) will receive no or little credit. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail, but keep in mind pseudocode does NOT substitute for an explanation. Answers that consist solely of pseudocode will receive little or not credit. Again, try to make your answers clear and concise.

1. Suppose each person gets an indendent uniformly random hash value from the range $[1 \dots n]$. (For the case of birthdays, n would be 365.)

   (a) **(10 points)** Show that for some constant c_1, when there are at least $c_1\sqrt{n}$ people in a room, the probability that no two have the same hash value is at most $1/2$.

   Suppose that there are m people in the room. Then the probability that no two have the same hash value is $\prod_{j=1}^{m-1} \frac{n-j}{n}$, since the probability that person j does not share the same hash value as persons $1, 2, \dots j-1$ (conditional on them having different hashes) is $\frac{n-j+1}{n}$ ($n-j+1$ valid out of n possible, each equally likely by assumption).

   Now we find c_1. From the hint, we know that $\prod_{j=1}^{m-1}\left(1-\frac{j}{n}\right) \le \prod_{j=1}^{m-1}\exp(-j/n) = \exp\left(\frac{-m(m-1)}{2n}\right)$. So, we only need $-\frac{(m-1)m}{2n} \le -1$. To solve for c_1, it suffices to plug in $m = c_1\sqrt{n}$ to the above inequality, which then simplifies to $c_1\sqrt{n}(c_1\sqrt{n}-1) \ge 2n$, or $c_1^2\sqrt{n} - c_1 - 2\sqrt{n} \ge 0$.[1] This is a quadratic in c_1 and is thus satisfied when $c_1 \ge \frac{1+\sqrt{1+8n}}{2\sqrt{n}}$. Now the sequence $\{a_n\}_{n\ge 1}$ given by $a_n = \frac{1+\sqrt{1+8n}}{2\sqrt{n}} = \frac{1}{2\sqrt{n}} + \frac{1}{2}\sqrt{\frac{1}{n}+8}$ is strictly decreasing in n and is therefore bounded above by $1/2 + 3/2 = 2$ (plug in $n=1$). Thus, $c_1 = 2$ should suffice.

   (b) **(10 points)** Similarly, show that for some constant c_2 (and sufficiently large n), when there are at most $c_2\sqrt{n}$ people in the room, the probability that no two have the same hash value is at least 1/2. (You may wish to use the same hint.)

   (Set up copy pasted from 1a) Suppose that there are m people in the room. Then the probability that no two have the same hash value is $\prod_{j=1}^{m-1} \frac{n-j}{n}$, since the probability that person j does not share the same hash value as persons $1, 2, \dots j-1$ (conditional on them having different hashes) is $\frac{n-j+1}{n}$ ($n-j+1$ valid out of n possible, each equally likely by assumption).

   Now we find c_2. Note that from the hint, we have

$$\prod_{j=1}^{m-1}\left(1-\frac{j}{n}\right) \ge \prod_{j=1}^{n}\exp\left(-\frac{j}{n}-\frac{j^2}{n^2}\right) = \exp\left(-\frac{(m-1)m}{2n} - \frac{(m-1)m(2m-1)}{6n^2}\right).$$

   Therefore, if we want the probability for no two to have the same hash value to be at least $\frac{1}{2}$, it is enough to again plug in $m = c_2\sqrt{n}$ and solve the inequality

$$\frac{(m-1)m}{2n} + \frac{(m-1)m(2m-1)}{6n^2} \le \log 2.$$[2]

   Plugging in the value of m here and simplifying yields that it's equivalent to findind c_2 such that for all n we have

$$\frac{c_2^2}{2} - \frac{c_2}{2\sqrt{n}} + \frac{c_2\sqrt{n}(2c_2^2 n - 3c_2\sqrt{n}+1)}{6n^2} \le \log 2$$

$$\iff \frac{c_2^3}{3\sqrt{n}} + \left(\frac{1}{2}-\frac{1}{2n}\right)c_2^2 - \frac{1}{2\sqrt{n}}\left(1-\frac{1}{3n}\right)c_2 \le \log 2.$$

   Now note that the LHS of the last inequality is at most $\frac{c_2^3}{3\sqrt{n}} + \frac{c_2^2}{2}$, so it is sufficient to find c_2 such that for all positive integers n we have

$$\frac{c_2^3}{3\sqrt{n}} + \frac{c_2^2}{2} \le \log 2.$$

   The LHS above is maximized at $n=1$, so we only really need $\frac{c_2^3}{3} + \frac{c_2^2}{2} \le \log 2$. Now a short trip to WolframAlpha gives us a bound of $c_2 \le 0.926$, so c_2 exists and we can use $c_2 = 0.926$.

[1] This is because if we find a c_1 that works, then it necessarily exists.

[2] This is because if we find a c_2 that works, then it necessarily exists.

(c) **(0 points, optional)**[3] Make these constants as close to optimal as possible.
Hints: You may wish to use the fact that

$$(1-\frac{1}{n})^n < \frac{1}{e} < (1-\frac{1}{n})^{n-1}$$

for all $n > 1$.You may want to also use the facts that

$$e^{-x} \geq 1-x$$

and

$$e^{-x-x^2} \leq 1-x \quad \text{for } x \leq \frac{1}{2}.$$

You may feel free to find and use better bounds.

[3]We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.

2. (a) **(5 points)** In our description of Bloom filters in class, we could only add and look up items. Suppose we try to implement deletion of an item from a Bloom filter by changing the corresponding elements to 0s. Give an example sequence of operations (adds, deletes, and lookups) for which this Bloom filter has both a false positive and a false negative.

   Consider a bloom filter consisting of $n \geq 2$ hash tables with hash functions $h_1, h_2, \ldots h_n$. Then suppose that we add x, y such that $h_1(x) = h_1(y)$ and $h_i(x) \neq h_i(y)$ for all other i. Then deleting x turns all the bits $h_i(x)$ into zero, which means that when we lookup y we return that y is not in the bloom filter (since $h_1(y)$ is now 0). This is a false negative.

   Now consider some pair (z, w) which hash to the same values for all hash functions (we can assume this exists since the bloom filters must have small enough size that the pigeonhole principle holds and so two will always agree in hashes). Then add w into the bloom filter. If we lookup z, this would return that it exists since all its hashes would be 1, so this gives a false positive.

   (b) **(15 points)** Counting Bloom filters are a variant of Bloom filters where you do not use an array of bits, but an array of small counters. Instead of changing 0s to 1s when an element hashes to a Bloom filter location, you increment the counter. To delete an element, you decrement the counter. To check if an element is in the set, you just check if all of the counter entries are bigger than 0; if they are, then you return the element is in the set. If you see a 0, you say that the element is not in the set. Deletions will work properly for a Counting Bloom filter as long as the counters never overflow; once a counter overflows, you would not have an accurate count of the number of elements currently hashed to that location. A standard choice in this setting in practice to use 4 bit counters. Find an expression for the probability that a specific counter overflows when using 4 bit counters when n elements are hashed into m total locations using k hash functions. (You may have a summation in your expression.) Calculate this probability when $m/n = 8$ and $k = 5$, for $n = 1000, 10000$, and 100000, and calculate the expected number of counters that overflow for each case.

   Here, I will be assuming that the hash functions' ranges partition the larger table (i.e. no two hash functions can output the same number), since this is what was assumed in class. I was also confirmed this in Franklyn's and Richard's OHs.

   Fix a counter. Then note that the probability that it is incremented by a specific element is k/m, since there are m/k possible locations it is hashed to by a specific hash function. Since there are n items to hash, by story of the binomial, the number of elements that hash to the counter's cell is distributed as $\text{Bin}(n, k/m)$. Now using 4 bit counters, an overflow occurs if the counter reaches 16, or when there are 16 items hashed to the same counter's cell. So, the probability that this counter overflows is $P(n, m, k) = 1 - \sum_{i=0}^{15} \binom{n}{i} \left(\frac{k}{m}\right)^i \left(1 - \frac{k}{m}\right)^{n-i}$ by taking the complement. Then from linearity of expectation, the expected number of counters that overflow is $m \cdot P(n, m, k)$.

   A short trip to WolframAlpha yields $P(1000, 8000, 5) = 1.29 \times 10^{-17}$, $P(10^4, 8 \cdot 10^4, 5) = 1.42 \times 10^{-17}$, and $P(10^5, 8 \cdot 10^5, 5) = 1.42 \times 10^{-17}$, which means the expected number of overflowing counters is 1.29×10^{-14}, 1.42×10^{-13}, and 1.42×10^{-12} for the first, second and third cases respectively, according to our calculations before.

   (c) **(5 points)** Suppose that you use m counters for n elements and k hash functions for a Counting Bloom filter. What is the false positive probability (assuming there has never been an overflow), and how does it compare with the standard Bloom filter?

   Note that the probability of an overflow, computed below, is astronomically low, so we can effectively assume that this will never happen (else the math is horrendously ugly, and I was told this was fine by Masaoud). Now consider a counter. If this was a counting bloom filter, then the counter equals zero iff nothing has been hashed into it. In contrast, for a standard bloom filter the counter (bit) is zero iff nothing has been hashed into it. Therefore, all cells with counters at least 1 in a counting bloom filter are the same as the cells with a bit of 1 in the standard bloom filter, and vice-versa. It then follows that a counting bloom filter can be "converted" into a standard bloom filter by just changing the non-zero values into a 1, and so in particular the probability that a given cell is 0 is the same for both tables. It then follows that the probability of a false positive is the same in both cases: about $(1 - e^{-\frac{nk}{m}})^k$.[4]

[4] We can also just see this directly since we already know that the probability of some cell being 0 still is about $(1 - k/m)^n \approx e^{-nk/m}$, just as a normal bloom filter.

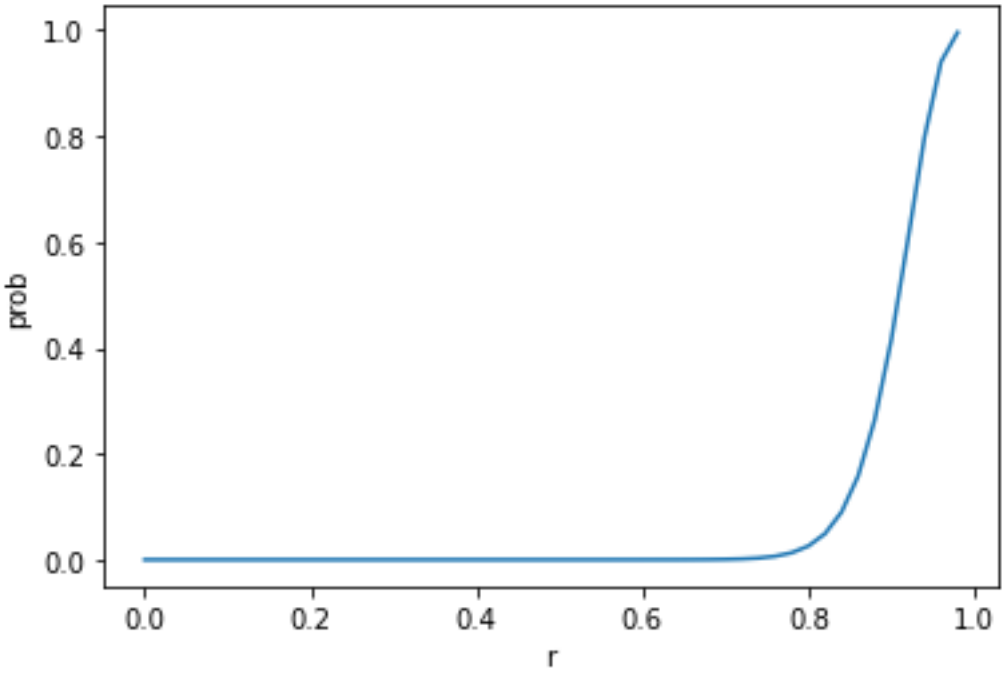
3. For the document similarity scheme described in class, it would be better to store fewer bytes per document. Here is one way to do this, that uses just 48 bytes per document: take an original sketch of a document, using 84 different permutations. Divide the 84 permutations into 6 groups of 14. Re-hash each group of 14 values to get 6 new 64 bit values. Call this the *super-sketch*. Note that for each of the 6 values in the super-sketch, two documents will agree on a value when they agree on all 14 of the corresponding values in the sketch.

   (a) **(5 points)** Why does it make sense to simply assume that this is the only time a match will occur?

   The only time a false positive match occurs is when two different values end up hashing to the same thing. But since we are hashing them to 64-bit values, and we have 2^{64} of them, assuming completely random hashing the probability of this happening is $\frac{1}{2^{64}}$, which is extremely small and basically never happens. So, it's safe to assume that a match indeed means it they agree on all 14 values in the sketch.

   (b) **(15 points)** Consider the probability that two documents with resemblance r agree on two or more of the six sketches. Write equations that give this probability and graph the probability as a function of r. Explain and discuss your results.

   By definition, the probability that two documents with resemblance r agree on a permutation is r. Therefore, the probability that they agree in a group is effectively r^{14} since there are 14 permutations in each. Now it follows that that the number of sketches they agree on (by story of binomial) is distributed as $\text{Bin}(6, r^{14})$, and so the probability they agree on exactly i (for $0 \leq i \leq 6$) is $\binom{6}{i}(r^{14})^i(1-r^{14})^{6-i}$. So, the probability that they agree on at least 2 of the 6 sketches is $1-(1-r^{14})^6 - 6r^{14}(1-r^{14})^5$. Here is a graph of this for $r \in [0,1]$:



   We also have the following table showing some of the values:

r	Sketch	Super sketch
0.5	1e-18	1e-8
0.8	0.006	0.026
0.9	0.58	0.42
0.95	0.988	0.879
0.96	0.997	0.940

   The slope here beyond 0.8 appears to be less steep than the slope in the original resemblance method (in the handouts). This, together with the above table, suggests that the probability that lower resemblance documents get flagged as similar is higher than before and higher resemblance have a lower chance than before, i.e. this is less accurate. This makes sense though, since we are using less space so some accuracy is being traded off.

   (c) **(10 points)** What happens if instead of using a 64 bit hash value for each group in the supersketch, we only use a 16 bit hash? An 8 bit hash?

If we instead use a 16 or 8 bit hash, the probabilities of a "false positive" are now 2^{-16} and 2^{-8} respectively. Since these are now not negligible probabilities, the probability of a match is (exactly) $r^{14}+2^{-k}(1-r^{14})$ where $k = 16, 8$ respectively (we're adding in the unconditional probability of a false positive–this existed before but was negligible so was ignored). The probability that two documents with resemblance r will agree on 2 of the 6 groups is then

$$1 - ((1 - r^{14})(1 - 2^{-k}))^6 - 6(r^{14} + 2^{-k}(1 - r^{14}))((1 - r^{14})(1 - 2^{-k}))^5,$$

(here k is the number of bits, say 8 or 16) which may pose some issues. The following table shows our probabilities for values of r and k:

r	k	Probability
0	8	2.26e-3
0	16	3.5e-9
0.5	8	2.34e-3
0.5	16	8.73e-8
0.8	8	0.030
0.8	16	0.026
0.9	8	0.422
0.9	16	0.415
0.95	8	0.881
0.95	16	0.879
0.99	8	0.9979
0.99	16	0.9998

e.g. if $k = 8$ and $r = 0$ then this probability is about 2e-3, which is certainly too likely! For $k = 16$ this is less of an issue since the probability would be around 3e-9, but this is still not negligible (at least compared to 64, which has a probability well under 1e-15).

4. Suppose we want to search for k patterns, each of length $m < n$, as substrings (not subsequences: "ac" is a subsequence but not a substring of "abc") of a document of length n, and report which of them are in the document.

   (a) **(5 points)** Give a simple $O(nk)$ algorithm for this problem. Using as a black box the document-searching scheme from class, your algorithm shouldn't take more than a sentence or two to describe.

   Do the same fingerprinting mod p for the patterns as in class then go through all m-length substrings (there are about $n - m = O(n)$ of them) and hash them in the same way, checking if they match with the patterns or not. If yes, we check if they are the same which takes $O(m)$, and since we can assume expected false positives is $O(1)$, total checking is $O(m)$, so run time is $O(nk + m) = O(nk)$.

   (b) **(24 points)** Give an algorithm to instead do the search in $O(n + km)$ time.

   Fingerprint every pattern mod p just like in class and put them in a hash table of size n. Then just as in lecture, we have an iterator representing which character we're at in the document. Note that by using the fingerprinting scheme in class, we can fingerprint every substring of length m in $O(n)$ time. Then, hash each substring in the document into a hash table. Note that by linearity of expectation, the expected number of items that hash into some cell in the hash table is at $(n - m + 1) \times \frac{1}{n}$ (there are $n - m + 1$ substring fingerprints being hashed, each has $\frac{1}{n}$ probability of hashing to the cell), which is at most $n/n = 1$, so the expected number of collisions in this hashing scheme is $O(1)$. So, the hash table construction is $O(n)$, and look ups are $O(1)$.

   Now for each pattern we fingerprint them according to the same scheme. This fingerprinting should take $O(km)$ time since there are k patterns and each has length m. Then hash each fingerprint to check if the fingerprint is in the hash table. If it is, we verify if the fingerprints match for the matched hashed values. If they don't, we discard since they then would not match as strings. If they do, we manually check if they match as strings (where we will use the index we stored for the first character), which takes $O(m)$ time, and return the pattern if we found an exact match (though we don't terminate the algorithm, we just go to the next pattern).[5] Since there are expected $O(1)$ false positives in the fingerprinting, the process of checking the matches according to the hash table will take $O(k)$ times, and so $O(km)$ time in total. If we found an exact string match, we remove the pattern from the string which takes $O(1)$ time (so $O(k)$ for all k patterns). Finally, since we have to hash each of the $O(n)$ substrings and check them in a hash table, this takes $O(n)$ time. This means that the run time is $O(n + km)$, as desired. The space complexity is $O(n)$.[6]

   For correctness, note that we are exhaustively searching the document for any substrings that match the pattern, and all positives are manually checked so false positives do not become an issue. Thus, this algorithm returns exactly the list of patterns that can be found in the document.

[5] To compare the substrings, just loop through using the index to iterate through the substring in the document, so this should take $O(1)$ space.

[6] I believe it is possible to make it $O(k)$ space by putting the patterns in the hash table instead and making it size k. Fingerprint every substring in the document and instead check if they're in the hash table. Once an exact match is found, remove the pattern from the hash table and continue until all substrings in the document are exhausted. From arguments earlier, deletions and lookups are $O(1)$ and we have $O(1)$ collisions.

5. (a) **(10 points)** Prove that 53586077 is composite by finding a witness in the form of a positive integer $a < 53586077$ such that $a^{53586077-1} \neq 1$. Give any information necessary to show that your witness in fact witnesses. (Note: do not use a factor as a witness! Sure, 53586077 is small enough that you can exhaustively find a factor, but we want to test your knowledge of the primality-testing algorithm without using awkwardly-big numbers.)

   Let $n = 53586077$. We coded in Python using the following algorithm: initialize a random a in the range $[2, n-2]$ and raise it to the $n-1$ power. If it's not 1, we output a, otherwise continue by generating another a and repeating the procedure. This found the number 38915485 which is clearly not a factor of n (double it and it is larger than n), and a short trip to WolframAlpha does verify that this works since $38915485^{n-1} \equiv 48961323 \not\equiv 1 \pmod{n}$.

   (b) **(10 points)** The number 294409 is a Carmichael number. Prove that it is composite by finding a witness in the form of a nontrivial square root of 1.

   Let $n = 294409$ and we express $n - 1 = 2^s \cdot t$ with s, t nonnegative integers and t odd. We used the following algorithm: if we have not found a witness yet, we select a random $a \in [2, n-2]$ and raise it to a^t modulo n. Keep squaring it until we hit $a^{2^s t}$. If at some point we got $a^{2^{k+1}t} \equiv 1$ but $a^{2^k t} \not\equiv 1$ (all mod n), then a is a witness and we output a. Otherwise, a is not a witness and we select another random a and repeat the process until we find a valid a. This algorithm works because we know that n is a Carmichael number, so such an a exists.

   From this algorithm, we get the number 154831 is a witness with $154831^{73602} \equiv 262144 \pmod{294409}$ and $154831^{147204} \equiv 1 \pmod{294409}$ so $n = 294409$ is composite.

   (You can find the answers by whatever method you like, but we recommend writing some code; if you use a language like C that doesn't natively deal with big integers nicely, you'll want some package that does. You don't need to submit code.)

6. **(0 points, optional)**[7] How many people do you need in the same room before it is more likely than not that some 3 people in the room share the same birthday? You may solve this problem purely mathematically (by developing an appropriate formula– be careful, this is a little tricky!) or by doing experiments by writing a program.

[7] We won't use this question for grades. Try it if you're interested. It may be used for recommendations/TF hiring.