1. The following approach is often called reservoir sampling. (It is useful, naturally, in streaming algorithms; you should think about why this might be.) Suppose that we have a sequence of items, passing by one at a time. We want to maintain a sample of one item that has the property that it is uniformly distributed over all the items that we have seen at each step. Moreover, we want to accomplish this without knowing the total number of items in advance or storing all of the items that we see.

Consider the following algorithm, which stores just one item in memory at all times. When the first item appears, it is stored in the memory. When the \( k \)-th item appears, it replaces the item in memory with probability \( \frac{1}{k} \).

Explain why this algorithm solves the problem.

Now suppose instead we want a sample of \( s \) items instead of just one, without replacement. That is, we don’t want to get the same item multiple times in our sample. (If this wasn’t an issue, we could get a sample of \( s \) items with replacement just by running \( s \) independent copies of the above.) Generalize the above process to that case. (Hint: start by taking the first \( s \) items and storing them as your sample. With what probability should each new item come into the sample?)

2. In this problem we consider a reduction lower bound for a streaming algorithm. We’ll use a reduction to the Index problem from Communication Complexity. Alice has a sequence of \( n \) bits \( x_1, x_2, \ldots, x_n \). Bob has an index \( i \). Bob wants to compute \( x_i \) after receiving a single message from Alice. Is it known that answering this question requires \( \Omega(n) \) bits (even for randomized algorithms that succeed with probability greater than say \( 2/3 \)).

Use the Index problem to show that any algorithm for the following problem, Max-Connected-Component \((k)\), requires \( \Omega(m) \) space for any 1-pass algorithm. In the Max-Connected-Component problem, one is given a stream of edges in a graph that is a forest on \( m \) vertices. (Recall a forest is a graph without a cycle.) The question is whether there is a connected component of size at least \( k \geq 3 \) in the graph.

(Hint: keep in mind you’re suppose to show that if you can solve the streaming problem in \( o(m) \) space, you can solve the Index problem with \( o(n) \) bits of communication. Start with an index problem. Let the graph consist of 3 sets of vertices: \( V_l = \{\ell_1, \ell_2, \ldots, \ell_n\} \), \( V_r = \{r_1, r_2, \ldots, r_n\} \), and \( V_d = \{d_1, d_2, \ldots, d_{k-2}\} \). Given an index problem, let Alice’s sequence of bits corresponds to edges \( \{(l_j, r_j) \mid x_j = 1\} \). What should Bob’s index correspond to, in terms of edges?)

3. Cuckoo hashing is a variant on the "multiple-choice" hashing frameworks. Its simplest variation is the following: there is a table with \( n \) cells. Each element \( x \) can hash into exactly two locations, given by hash functions, \( H_1(x) \) and \( H_2(x) \). When an item is placed into the hash table, if at least one of these two locations is free, the item is placed in a free location.

If both locations are not free, \( x \) chooses one of the two locations, and kicks the element \( y \) that is in that location out. Then \( y \) must try to go to its alternative location. If that location is free, then all is well, and \( y \) is placed there; otherwise, \( y \) must kick out the element in that location, and this new element must try to move to its alternative location, and so on.

It is possible that, at some point, an element cannot be placed, in which case the above process will loop. The loop can either be found explicitly, or a limit on the number of times elements can be kicked out for a single placement can be enforced.

One way to generalize this when each element has more than two choices is to choose which element to kick out randomly at each step.

Implement cuckoo hashing. hen hashing an item into a table, you can use an idealized pseudo-random hash
function. This can be accomplished by using a pseudo-random number generator to determine where each element goes, and recording it in a table so you always put the element in the same place for the trial.

In your experiments, use a table of size 32768, and add elements until the first time you cannot add an element. (For convenience, you may assume an element cannot be added if, after repeating the kick out step 100 times, you are not done.) Using 2 hash functions and 3 hash functions, and running the experiment 1000 times, examine how full the hash table can be before problems start to occur. Discuss your findings. (Feel free to do further experiments, on larger tables or varying other parameters, if you wish.)

For this problem, please turn in your code (in an appendix, where I hopefully won’t have to read it).

4. In this problem, you will experiment with the (multistage) filters of the Mice and Elephants/Count-Min sketches papers, both with and without conservative update. Your input stream will consist of the elements 1 through 9,100. For $1 \leq i \leq 9$, elements $1000 \cdot (i - 1) + 1$ to $1000 \cdot i$ will appear $i$ times in the stream. That is, elements 1 to 100 will appear once in the stream; 1001 to 2000 will appear twice; and so on. Elements $9000 + i$, for $1 \leq i \leq 100$, will appear $i^2$ times in the stream. For example, element 9100 will appear 10,000 times. (Each time an element appears in the stream, it has a count of 1 associated with it.)

You must consider the stream in 3 different orders:

1. all appearances of 1, followed by all appearances of 2, followed by all appearances of 3, etc. (forward order).
2. all appearances of 9100, followed by all appearances of 9109, followed by all appearances of 9108, etc. (reverse order)
3. all elements in a random order.

You should do 100 trials, both with conservative update and without. Use 4 independent hash tables and functions on each trial. Each hash table should have 500 counters. When hashing an item into a table, you can use an idealized pseudo-random hash function. This can be accomplished by using a pseudo-random function to determine where each element goes, and recording it in a table so you always put the element in the same place for the trial.

Over the 100 trials, 3 different orders, and with and without conservative update, record the following after all elements in the stream have passed:

1. How many elements are, according to the filter data structure, potentially responsible for 1% or more of the total load. Compare your result with the true answer.
2. What the presumed count is for element 9,100.

Present these results in a concise, organized manner.

Based on your results discuss the importance of order on filter performance, the importance of conservative update on filter performance, and your opinion overall on how the filter performed on this data stream.

For this problem, please turn in your code (in an appendix, where I hopefully won’t have to read it).

5. Alice wants to send Bob the result $X$ of a fair coin flip over a binary symmetric channel that flips each bit with probability $0 < p < 1/2$. To avoid errors in transmission, she encodes heads as a sequence of $2k + 1$ zeroes and tails as a sequence of $2k + 1$ ones.
1. Consider the case where \( k = 1 \), so heads is encoded as 000 and tails as 111. For each of the eight possible sequences of three bits that can be received, determine the probability that \( X \) was heads conditioned on Bob receiving that sequence.

2. Argue that for general \( k \), Bob minimizes the probability of error by deciding that \( X \) was heads if at least \( k + 1 \) of the bits are 0. (For partial credit, just prove this for \( k = 1 \).)

3. Give a formula for the probability that Bob makes an error that holds for general \( k \) and \( p < 1/2 \). Give a plot of the formula for \( p \in (0.01, 0.1) \) and \( k \) ranging from 1 to 6. (If you don’t like plots, give a table for \( p = 0.01, 0.02, \ldots, 0.10 \).)

6. I have encoded a message using Reed-Solomon codes. My encoding works as follows. Each letter corresponds to a number: \( a \) to 1, \( b \) to 2, etc. To simplify matters, I do all work modulo 29, which is prime. My message is a four-letter word, and I used the four letters as the coefficients of a polynomial. (If my word was “aaaa”, my polynomial would be \( 1 + x + x^2 + x^3 \).) Below I provide the values \( P(1), \ldots, P(6) \), at most one of which is in error. Determine the message.

\[
P(1) = 15, P(2) = 13, P(3) = 22, P(4) = 24, P(5) = 1, P(6) = 22.
\]

7. Consider the following channel: the sender can send a symbol from the set \( \{0, 1, 2, 3, 4\} \). The channel introduces errors; when the symbol \( j \) is sent, the receiver receives \( j + 1 \mod 5 \) with probability \( 1/2 \), and receives \( j - 1 \mod 5 \) with probability \( 1/2 \). The errors are mutually independent when multiple symbols are sent.

We can define \( (k, n) \) encoding and decoding functions for this channel. The encoding function \( En \) maps numbers from \( \{0, \ldots, k - 1\} \) into sequences from \( \{0, 1, 2, 3, 4\}^n \), and the decoding function \( De \) maps sequences from \( \{0, 1, 2, 3, 4\}^n \) into a number in \( \{0, \ldots, k - 1\} \).

There are \( (2, 1) \) encoding and decoding functions with zero probability of error. The encoding function maps 0 to 0 and 1 to 1. When a 0 is sent, the receiver will receive either a 1 or 4, so the decoding function maps 1 and 4 back to 0. When a 1 is sent, the receiver will receive either a 2 or 0, so the decoding function maps 2 and 0 back to 1. This guarantees that no error is made. Hence at least one bit can be sent without error per channel use.

1. Show that there are \( (5, 2) \) encoding and decoding functions with zero probability of error (by giving such functions). Argue that this means that on average more than one bit of information can be sent per use of the channel.

2. Show that if there are \( (k, n) \) encoding and decoding functions with zero probability of error, then \( n \geq \log_2 k / (\log_2 5 - 1) \).