Stream Sampling

1. I'm going to start listing numbers off to you. I'll stop at some arbitrary point and ask you to give me a uniformly sampled number from the ones I listed.

   a. Can you design a simple algorithm to do this?

      Solution: We'll store all the numbers in a list as they come in. Then we'll choose a random number i between 1 and n (inclusive) and return the number at i in the list.

   b. Now what if I ask you to give me a uniform sample of k numbers without replacement from the ones I gave (I'll tell you k up front). Does your solution easily adapt?

      Solution: We'll do what we did in a. except instead of choosing one index from 1 to n, we'll first choose one from 1 to n, remove that element from our list and add it to a new list to track our sample. Then we'll choose another index from 1 to $n-1$, remove it from the list and add it to the sample, and so on until we have a k element sample.

   c. Let's think about our solution and get a sense of how efficient it is. In computer science, we measure efficiency in few different metrics. Two of the most important are *time* and *space*.

      We usually consider algorithms time-efficient and/or space-efficient if they use $O(n)$ time and/or $O(\log n)$ space respectively. Suppose I ultimately list off n numbers. Does our algorithm meet these standards?

      Solution: We take $O(1)$ time per number we consume in the stream, and then $O(k)$ time to ultimately produce a sample. We unfortunately use $O(n)$ space, because we store every number we're given.

2. Let's think about why we use as much space as we do and how we can use less.

   a. Let's simplify the problem for a moment by only sampling a single number—i.e. fixing $k = 1$.

      If I ask you to give me the minimum number I list instead of a random one, can you come up with a more space-efficient solution?

      Solution: We'll create a variable to store the "running" minimum, and initialize it to the first number we get. As we get numbers from the stream, we'll set the running minimum to the minimum of itself and the new element. At the end, we'll just return it as our sample.

   b. Can we turn the random-sampling problem into a min-finding problem?

      Hint: instead of just storing the numbers as they come in, what if we also store a random "tag" sampled uniformly from $[0, 1)$ *? You may assume that we can take such samples at will.*

      Solution: As each number comes in, we'll add a tag sampled from $[0, 1)$ to the number. We'll store the number with the "running" minimum tag as in a. Then we can return the number with the smallest tag at the end.

      Why does this work? Every number has a uniform probability of having the minimum tag, because the tag is sampled uniformly from $[0, 1)$ and is chosen independently for each number.

   c. Can we scale this solution into one that works for a sample of k numbers (again without replacement)? What are the time and space costs of this solution?

      For time, you should note that it is possible to maintain a collection of k numbers using $O(k)$ space[1] such that it costs:

      - $O(1)$ time to find the maximum
      - $O(\log k)$ time to remove the maximum
      - $O(\log k)$ time to add a new item to the collection. [2]

      Solution: We'll want to track the numbers with the k smallest tags. To do so, we'll store the first k numbers in our max-heap (see footnote 2). Whenever we get a new

[1] I'm making the simplifying assumption here that it takes $O(1)$—that is, constant—space to write down a number.
[2] This can be done by arranging the k elements into a data structure called a binary heap.

number, we'll generate a tag. If the tag is smaller than the maximum in our heap, it must be one of the k smallest we've seen, so we'll remove the maximum from the heap and add our new number to the heap.

For each new number, it only costs us $O(1)$ to generate the tag and lookup the maximum in our heap. If we have to put out new element into the heap, it'll cost us $O(\log k)$. Space-wise, the only storage we have is the heap, so we use $O(k)$ space.

d. Suppose that while I'm listing numbers, I can stop you multiple times to get up-to-then random samples. Is there a statistical difference bewteen the samples given by solutions 1 and 2?

Solution: There is. Suppose I feed the stream of integers 1 through 100 into the sampler. After I feed it 50, I ask for a sample. Suppose without loss of generality we get back 25. Then we'll ask for another sample after we finish the stream.

In solution 1, the only thing we know about the sample at the end of the stream is that it's uniform. In solution 2, we know that the sample can't possibly be any number $1 - 50$ (inclusive) except for 25, because they've all been thrown away.

3. Can we do even better?

   a. Let's again go back to the $k = 1$ case. Let E_i denote the event in which we take the ith number as our current sample. We'll always take the first number we are given as the sample, so $\mathbb{P}[E_1] = 1$. What is $\mathbb{P}[E_2]$? $\mathbb{P}[E_i]$?

      Solution: If I hand you i random tags what's the chance that any particular one I point to is the smallest? Because the random tags are chosen uniformly and indpendently, each has exactly the same chance to be the minimum. Then each tag has a $\frac{1}{i}$ chance of being the smallest.

      This means that the ith number has a $\frac{1}{i}$ chance to be chosen as the new sample.

   b. How can we use this idea to make a new version of our sampler? What are the time and space efficiency of this new version?

      Solution: Instead of storing the tags, we'll just use the probabilities we determined. On the ith element of the stream, we'll flip a $\frac{1}{i}$-biased coin ($\frac{1}{i}$ probability of coming up heads). If it comes up heads, we'll replace our current sample with this number.

   c. Can you prove that this solution gives a uniform random sample? I.e. let X be a random variable for the output of the sampler. Can you prove that for every number x in a list of n numbers, $\mathbb{P}[X = x] = \frac{1}{n}$? Assume that every number in the list is distinct. *Hint: try induction on n, the number of numbers I ultimately list.*

      Solution: In the base case $n = 1$. The first number is certain to be chosen as the sample per our scheme. A uniform random sample of one number has to be that number with probability 1, so this is correct.

      In the inductive case, we assume that our strategy works for all stream sizes 1 through n. Suppose we've processed n numbers, and then we get an $(n + 1)$st. When we get a new number, we'll chose it as the new sample with probability $\frac{1}{n+1}$.

      Then the new number has probability $\frac{1}{n+1}$ of being the sample, which is correct. By the inductive hypothesis, each of the previous numbers has probability $\frac{1}{n}$ of having been the sample prior to the $(n + 1)$st number. At the $(n + 1)$st number, we have a $1 - \frac{1}{n+1} = \frac{n}{n+1}$ chance of keeping the old sample. This means that each of the first n numbers has a $\frac{1}{n} \cdot \frac{n}{n+1} = \frac{1}{n+1}$ chance of being the sample, as desired.

d. Can we generalize this solution to work for a size k sample?

Solution: We'll track the sample in a list of size k. The first k nuymbers go into the list. Thereafter, for the ith number (i includes the fact that we've passed k numbers, so the first i we do this for is $i = k+1$), we decide to keep the number with probability $\frac{k}{i}$. We want every number in the current sample to have an equal chance of being replaced, so we'll choose a random integer index between 1 and k (inclusive), and replace the number at that index in the sample with the new number.

e. This final solution is called *reservoir sampling*. It's deployed in the real world in order to take samples from massive real-time data sets. What are it's time and space costs?

Solution: It costs $O(1)$ time per each new element, and then $O(1)$ time to ultimately return the final sample. Again, like solution 2, we only use $O(k)$ space.