Lecture Notes: Random shuffling and sampling

Yufei Tao
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

1 Apr, 2012

In this lecture, we will discuss a fundamental problem: given a set S of size n, how to obtain a random sample set R of S in $O(n)$ time? At first glance, this may seem to be trivial: if we want an R of size s, how about generating a random number t from 1 to n, taking the t-th element of S as a sample, and repeating this s times? But another moment's thought would reveal that this algorithm may not always suit our needs. First, it samples *with* replacement. In other words, one can imagine that, after taking an element of S as a sample, this element is put back into S (i.e., replaced) so that it can be sampled *again*. Sampling with replacement may create a *bag* of samples, i.e., a multi-set, because the same element can have several copies in R. In some scenarios, we want to have distinct samples: for example, if you want to interview s random people to do a survey, it makes no sense to interview the same person twice. Another limitation of the algorithm is that it needs to know the number n in advance (otherwise, what is the range of your random number?). This is not a desired property in data streams: for example, if you want to keep track of a sample set of the creditcard transactions tomorrow *continuously* as they take place, you have no idea about n, i.e., the total number of transactions.

You may easily come up with another simple algorithm: if the sampling rate is p, why not just independently include each element of S with probability p. The correctness is obvious, and the time is apparently $O(n)$. Well done, but the algorithm again falls short if we want the sample set R to have a *specific* size. For example, if our memory for storing creditcard transactions can accommodate only 10000 transactions, then we do not want R to have more than 10000 transactions. How would you choose your p in such a scenario when you do not know n? Furthermore, even if you *do* know n, it is difficult to make sure that your sample set never exceeds size s. Note that, even though the sample set clearly has *expected* size np, its *actual* size may be any number from 0 to n, albeit perhaps with a small probability. What if we want $|R|$ to be exactly s *for sure*?

In this lecture, we will learn another sampling algorithm that allows us to take a sample set R of a desired size in $O(n)$ time, by looking at each element of S only once. In fact, this algorithm has a lot to do with a closely related problem: *shuffling*, as we will also study, actually, will study first.

1 Random shuffling

Let us first consider the following *random shuffling* problem. Let S be an array of size n. We denote the i-th ($1 \leq i \leq n$) element of the array as $S[i]$. The goal is create a random permutation of S. Namely, when our algorithm finishes, the list of elements $(S[1], S[2], ..., S[n])$ should be any of the $n!$ permutations with the same probability.

Algorithm. We will discuss an algorithm discovered by Durstenfeld [1]. This algorithm is also often called the *Fisher-and-Yates algorithm*. The algorithm processes each element of S in turn. In processing the i-th element $S[i]$, it generates a random number x in the range $[1, i]$ (note that

the range increases with i). Then, it swaps $S[i]$ with $S[x]$ (of course, if $i = x$, then the swap has no effect). At the end of the algorithm, the list of elements currently in the array $(S[1], ..., S[n])$ is the final permutation output.

Analysis. The algorithm clearly runs in $O(n)$ time. Now we prove that it is correct:

Lemma 1. *The list of elements $(S[1], ..., S[n])$ at the end of the algorithm can be any of the $n!$ permutations with the same probability.*

Proof. We will prove:

Claim: After processing $S[k]$ $(1 \leq k \leq n)$, the list $(S[1], ..., S[k])$ is any of the $k!$ permutations with respect to these elements with the same probability.

The correctness of the above claim immediately implies the correctness of the lemma. We prove the claim by induction. First, the claim is obviously correct for $k = 1$. Assuming that the claim holds for $k = i$, next we show that it also holds for $k = i + 1$.

Let P be the list $(S[1], ..., S[i])$ *before* processing $S[i + 1]$, and similarly, let P' be the list $(S[1], ..., S[i+1])$ *after* processing $S[i+1]$. Let x be the random number generated in the processing of $S[i + 1]$. Then, P' is a function of the pair (P, x). Note that P has $i!$ possible choices, while x has $i + 1$ choices. In other words, there are $(i + 1)!$ different pairs of (P, x). Next, we will prove:

- Fact 1: each pair of (P, x) happens with the same probability in the algorithm.
- Fact 2: each pair of (P, x) produces a distinct P'.

Combining both facts establishes the correctness of the claim for $k = i + 1$.

Proof of Fact 1. By our inductive assumption, each P happens with probability exactly $1/i!$. The value of x is independent from P, and can take any value in $[1, i + 1]$ with probability $1/(i + 1)$. Hence, each pair (P, x) happens with probability $1/(i + 1)!$.

Proof of Fact 2. Let (P_1, x_1) and (P_2, x_2) be two distinct pairs. Namely, at least one of the following is true: $P_1 \neq P_2$, $x_1 \neq x_2$. Let P_1' and P_2' be the P' produced by the first and second pairs, respectively. Our aim is to show that $P_1 \neq P_2$. We distinguish two cases:

- Case 1: $P_1 = P_2$. Since $x_1 \neq x_2$, P_1' differs from P_2' in the $(i + 1)$-st element (by the way the algorithm works). Hence, $P_1 \neq P_2$.
- Case 2: $P_1 \neq P_2$. Let j be the smallest number such that the j-th element of P_1 is different from that of P_2. If $x_1 \neq x_2$ or $x_1 = x_2 \neq j$, then P_1' still differs from P_2' in the j-th element. Otherwise (i.e., $x_1 = x_2 = j$), then P_1' still differs from P_2' in the $(i + 1)$-st element.

This completes the proof. □

2 Random sampling

Now we are ready to tackle random sampling. The problem setup is as follows. Let S be an *unbounded* stream of elements, namely, its elements keep arriving. We are allowed to spend only constant time on each element, i.e., when it arrives. We want to maintain a random sample set R of size s. More specifically, if n is the number of elements we have seen so far, then any subset with size s of those elements should have exactly the same probability of being our R. Furthermore, this has to be true at *all* times.

Algorithm. The algorithm we are presenting now is called the *reservoir algorithm* [2]. We denote the i-th element of R as $R[i]$. At the beginning, we fill up R with the first s elements of the stream. Starting from the $(s+1)$-st element, however, we go through a random process to decide whether to sample the element, and if yes, which existing sample is to be replaced. Specifically, let e be the n-th element in the stream where $n \geq s+1$. We generate a random number x from 1 to n. If $x > s$, then e is discarded. Otherwise, we set $R[x] \leftarrow e$, i.e., essentially discarding the original $R[x]$ and using e to take its place. It is clear that each element is processed with constant time.

Correctness. To show that the algorithm is correct, we need to prove:

Lemma 2. *After processing n elements, R is any subset with size s of those n elements with the same probability.*

Note that it is *insufficient* to prove that each of the first n elements has the same chance of being in R (why?).

Proof. We prove the lemma with induction on n. The lemma is obviously true for $n = s$. Assuming that the lemma is correct for $n = k$, next we show that it is also correct for $n = k+1$.

Let e be the $(k+1)$-st element in the stream. Let R be the sample set *before* processing e, and R' the sample set *after* processing e. Furthermore, let x be the random number generated in the processing of e. Note that R has $\binom{k}{s}$ difference choices, while x has $k+1$ choices.

Consider one specific subset T of the first $k+1$ elements in the stream. We will prove that $R' = T$ with probability exactly $\frac{1}{\binom{k+1}{s}} = \frac{s!(k+1-s)!}{(k+1)!}$. For this purpose, we distinguish two cases:

- Case 1: T does not include e, or equivalently, $T = R$ and $x > s$. By our inductive assumption, the probability that $T = R$ equals $\frac{1}{\binom{k}{s}} = \frac{s!(k-s)!}{k!}$. As $x > s$ (independently from $T = R$) takes place with probability $\frac{k+1-s}{k+1}$, we know that any T of Case 1 can be R' with probability $\frac{s!(k-s)!}{k!} \cdot \frac{k+1-s}{k+1} = \frac{s!(k+1-s)!}{(k+1)!}$.
- Case 2: T includes e. This case implies that the following two conditions must hold simultaneously:
  - Condition $C1$: T differs from R in only one element e'.
  - Condition $C2$: x equals exactly the position of e' in R.

  Let us analyze the probability that $C1$ holds. How many R can differ from T in only one element? To answer this question, we construct such an R as follows. First, add all the elements in T but e to R. Second, add to R any of the $k-(s-1)$ elements that are in the first k elements of the stream, but are not already in R. Hence, it is clear that the number of such R is $k-s+1$. By our inductive assumption, each such R occurs with probability $\frac{1}{\binom{k}{s}} = \frac{s!(k-s)!}{k!}$. Hence, $C1$ holds with probability $\frac{s!(k-s)!}{k!} \cdot (k-s+1) = \frac{s!(k-s+1)!}{k!}$

  Condition $C2$, on the other hand, holds with probability $1/(k+1)$, because x has to be precisely the index of e' in R, out of the $k+1$ possible choices. As $C1$ and $C2$ are independent, the probability that they bold hold equals $\frac{s!(k-s+1)!}{k!} \cdot \frac{1}{k+1} = \frac{s!(k-s+1)!}{(k+1)!}$.

This completes the proof. □

References

[1] R. Durstenfeld. Algorithm 235: Random permutation. *Communications of the ACM (CACM)*, 7(7):420, 1964.

[2] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.