How to sample from a distribution?[1]

- In the previous two lectures, we saw applications of importance/non-uniform sampling which sample objects not from a universe uniformly at random but with probability proportional to some weight or importance. For instance, in the #DNF application it was sampling a set from a collection of sets with probability proportional to the size of the set, and in the Cohen-Lewis algorithm (which itself was an importance sampler), it was picking a row with probability proportional to the square of the sum of the row, etc. However, how does one do that? How much time does it take to sample from such a distribution?

- Before we go to non-uniform sampling, let us first address uniform sampling. We want to design an algorithm which given a positive number n returns an integer $i \in [n]$ with probability $1/n$. If we have only access to random bits (that is, the press of a button gives me 0 or 1 with probability $1/2$) and n is a power of 2, then we can simulate the uniform sampler with $\log_2 n$ random bits. Do you see how? Hint: every number has a binary representation. If n is not a power of 2, then one could use rejection sampling to get a Las-Vegas algorithm which runs in $O(\log n)$ expected time. For what follows, we are going to assume that we have the power to uniformly sample. That is, given any N, we can get a sample in $[N]$ in one unit of time. This is not an unreasonable assumption in many programming environments.

- *Set up.* We assume there is a universe U of n elements, and we are given weights w_1 to w_n which we are going to assume are non-negative integers. Our goal is to sample $i \in U$ such that

$$\mathbf{Pr}[i] = \frac{w_i}{\sum_{i=1}^{n} w_i}$$

  Assume you have the power to uniformly sample from a set of n elements.

- The sampling algorithm has two phases: a *preprocessing step* which reads the weights and does some work, and a *query* step which you think of as a button. You press the button and out pops the i as desired. We would like the preprocessing step to take $O(n)$ time since it takes that much time to even read the input. We would want the query time to be much faster.

- *Solution 1.* The idea is to think of the the w_i's as gaps between flags on a road which is $\sum_{i=1}^{n} w_i$ units long, and one samples uniformly a "mile" along this road and picks the first flag coming after it. (Ideally, I would draw a figure here.)

  To be more concrete, in the preprocessing step, one computes the array $S[0:n]$ defined as

$$\forall 1 \leq i \leq n,\ S[i] := \sum_{j \leq i} w_j$$

  It should be clear $S[0:n]$ can be computed in $O(n)$ time using running sums, and is sorted increasing.

  Then, in the query phase, we do the following:

[1] *Lecture notes by Deeparnab Chakrabarty. Last modified : 12th April, 2023*
These have not gone through scrutiny and may contain errors. If you find any, or have any other comments, please email me at deeparnab@dartmouth.edu. Highly appreciated!

a. Sample *uniformly* $r \in_R [\, S[n] \,]$.

b. Perform binary search to find smallest i such that $r \leq S[i]$.

c. Return i.

Exercise: *Convince yourself that for any $i \in \{1, 2, \ldots, n\}$ the probability we sample i in Step 3 above is $\frac{w_i}{S[n]}$.*

The query time is dominated by binary search and is $O(\log n)$. In particular, if we make K queries, then the total query time is $O(K \log n)$. This is not bad.

Remark: *What if the w_i's were real numbers? Well then we need a stronger primitive: we would need to have the ability to generate a random real number between $[0, 1]$ in $O(1)$ time. Hopefully, the reader can modify the above algorithm to work for real w_i's with this primitive.*

- *Solution 2, a slight "improvement" when K is known.* So, if you were making K queries, then the total time would be $O(n + K \log n)$. One can do a shade better if we knew the parameter K in advance.

  The idea is to run "Step 1" of the above algorithm K times up front. This takes $O(K)$ time. Then, we *sort* these K numbers, and this takes $O(K \log K)$ time. Let's say the sorted random numbers are $r_1 \leq r_2 \leq \cdots \leq r_K$. Next, we *merge* (as in merge sort) with the array S, and as we are merging for each r_k, $1 \leq k \leq K$, we figure out the smallest i_k such that $r_k \leq S[i_k]$. We return $(i_1, \ldots, i_K)$. I am not explaining all the details here, but hopefully you can figure this out. This can be done in $O(K + N)$ time.

  So, the total time is $O(K \log K + N)$, which doesn't look much better. Except, the merging step is a "cache-efficient" step because you can pull big chunks of the array into fast memory, merge, and then continue, while binary search is not cache-efficient as its access patterns on the array go all over the place. Once again, I am not being formal here because it'll take us away too much, but once again, hope this is clear.

  Next let's see a smart algorithm which makes the query time down to $O(1)$.

- *Solution 3: Walker's Alias Method*[2]. The idea is to pre-process cleverly. For this algorithm, we will actually assume the w_i's are non-negative reals and we have the ability to sample a uniform real number in $[0, 1]$.

  To begin with, let's define $p_i := w_i / \sum_i w_i$, and so we want to design a sampler which returns i with probability p_i. The idea above was to consider n flags in a one mile long road where p_i is the gap between the $(i-1)$th and ith flag, randomly land on this road, and then binary search to find the nearest flag ahead of you. The bulk of the time is taken in binary-searching.

  The alias method proceeds differently. It first processes the vector $(p_1, \ldots, p_n)$ to get another vector $(B_1, \ldots, B_n)$ where each B_t is a structure containing two tuples:

$$B_t = \left\{ (i, q_i), (j, q_j) \quad : 1 \le i \le n, 1 \le j \le n, \;\; 0 \le q_i, q_j \le \frac{1}{n} \quad , q_i + q_j = \frac{1}{n} \right\}$$

  and for all $1 \le i \le n$, if we sum up the q_i's over all the buckets containing an "i-tuple" then we get p_i.

  Let's unpack this. Imagine the n numbers as n different colors with p_i units of each color. The total amount $\sum_i p_i = 1$. Imagine the B_t's as "buckets" which can contain different colors. The processing leads to buckets where each bucket has (at most) two different colors such that the total amount in every bucket is the same, and thus equals $\frac{1}{n}$.

  A priori, it is not clear how to obtain this data structure. But let us show that if we do have this data structure, then one can sample i with probability p_i using $O(1)$ time.

  procedure ALIAS-QUERY$(p_1, \ldots, p_n)$:
  Sample $t \in \{1, 2, \ldots, n\}$ uniformly at random.
  Let $B_t = \{(i, q_i), (j, q_j)\}$ with, recall, $q_i + q_j = \frac{1}{n}$.
  Sample $z \in \left[0, \frac{1}{n}\right]$.
  If $z \le q_i$, return i; else, return j.

Claim 1. The above sampler returns a coordinate $i \in 1, \ldots, n$ with probability p_i.

Proof. Fix an i in $[n]$. For $t \in [n]$, let $z_i(t)$ be the amount of i in B_t. More precisely, if $(i, q_i) \in B_t$ then $z_i(t) = q_i$, otherwise it is 0. Note: $\sum_{t=1}^{n} z_t(i) = p_i$. The probability i is returned by the above sampler is

$$\sum_{t=1}^{n} \underbrace{\mathbf{Pr}[t \text{ sampled}]}_{=\frac{1}{n}} \cdot \mathbf{Pr}[i \text{ sampled} \mid t \text{ sampled}]$$

What's the probability i is sampled given bucket t is sampled? If $z_t(i) = 0$, then it's 0. Otherwise, it is $z_t(i)/(1/n) = n z_t(i)$. Substituting above gives the answer. □

- So, the interesting part is to get the data structure. One begins with an initial bucket configuration where bucket B_t contains only one tuple (t, p_t). One can think of another tuple $(i, 0)$ for an arbitrary i. We maintain the invariant that we never have more than two tuples in a bucket.

[2] Walker, A. J. (September 1977). An Efficient Method for Generating Discrete Random Variables with General Distributions. ACM Transactions on Mathematical Software. 3 (3): 253–256.

Let's use b_t to denote the total amount of "stuff" in bucket B_t; in particular, if $B_t = \{(i, q_i), (j, q_j)\}$, then $b_t := q_i + q_j$. In the end, we want all b_t's to be $1/n$.

Call a bucket B_t full if $b_t = \frac{1}{n}$. Call a bucket underfull if $b_t < \frac{1}{n}$. Call a bucket overfull if $b_t > \frac{1}{n}$. We will maintain the invariant that an underfull or an overfull bucket has only one tuple (i, q_i) with $q_i > 0$; initially this is true. The pre-processing algorithm keeps moving stuff from overfull buckets to underfull buckets.

```
1: procedure ALIAS-PREPROCESSING(p_1, ..., p_n):
2:     Initialize B_t := {(t, p_t)} for 1 ≤ t ≤ n.
3:     while there exists overfull bucket do:
4:         Pick an overfull bucket B_t = {(i, q_i)} with q_i > 1/n.
5:         Pick an underfull bucket B_s = {(j, q_j)} with q_j < 1/n.
6:         Modify: B_t = {(i, q_i + q_j − 1/n)} and B_s = {(j, q_j), (i, 1/n − q_i)}.
```

Note that Line 6 makes the bucket B_s full while B_t could remain overfull, or become underfull, or if $q_i + q_j = 2/n$, become full. In any case note: B_t still contains only one tuple, and the number of full buckets increases by 1. The latter implies there are at most n loops. Also note that we can keep the overfull, undefull, and full buckets in a list moving them when needed; these take $O(1)$ time. Thus the total preprocessing time is $O(n)$.