— 2 —

The Alias Method

> "It is yet another Civilized Power, with its banner of the Prince of Peace in one hand and its loot-basket and its butcher-knife in the other. Is there no salvation for us but to adopt Civilization and lift ourselves down to its level?"
>
> ---
>
> To the person sitting in darkness, Mark Twain

2.1. Introduction

Given a list of n positive numbers $p_1, \ldots, p_n$ such that $\sum_i p_i = 1$, consider the basic task of picking an index i at random with probability p_i. Two constant-time operations are assumed to be available:

(I) rand(n): Given an integer n, returns a real number uniformly at random from$[0, n)$

(II) The floor function $\lfloor \cdot \rfloor$.

The ***alias method*** is described below. After linear-time preprocessing, it enables sampling from this discrete distribution in worst-case constant time.

2.2. The alias method

Idea. The algorithm redistributes the probabilities into n buckets, where each bucket is selected with probability $1/n$. Each bucket stores either one value or two original values, and a threshold specifies how to choose between them.

An elegant way to think about this is to view each probability p_i as an interval of length (you guessed it) p_i, with i being its ***label***. These intervals tile/cover the interval $[0, 1]$. Think about marking each interval by its label (i.e., the value i that it represents). Then, choosing a random value i according to this distribution is easy—we randomly pick a value $x \in [0, 1]$, and pick the interval in this partition that contains x. Naively, using binary search over prefix sums, this takes $O(\log n)$ time per query—terribly slow, unacceptable even.

The key insight to doing better is to break each input interval into several intervals (at most $2n$ intervals overall), and rearrange them, such that the tiling they generate has a special structure—the numbers i/n are always endpoints of intervals in this tiling. Furthermore, each fragment has the label of the original interval it came from. Thus, the total length of all the fragments labeled by i is still going to be exactly p_i. Thus, picking a label using the same algorithm would yield a random sample with the desired distribution.

The algorithm. The algorithm partitions the input intervals into three queues:

(I) Too small: $S = \{(i, p_i) \mid p_i < 1/n\}$.

(II) Too perfect: $M = \{(i, p_i) \mid p_i = 1/n\}$.

(III) Too large: $L = \{(i, p_i) \mid p_i > 1/n\}$.

As long as the queues are not empty, the algorithm continues running. There are several cases.

There is an interval of length $1/n$: If there is any interval in M, the algorithm uses it in the tiling (i.e., it is tiling the interval $[0, 1]$ from left to right) and moves on to the right.

No interval of length $1/n$: If M is empty, and the algorithm already generated i tiles, then the number of intervals in the two queues is exactly $n - i$. The remaining length to tile is $T = (n - i)/n$. Thus, the average length of the remaining intervals is exactly $T/(n - i) = 1/n$. By the pigeonhole principle, since none of remaining intervals can have length $1/n$ (as $M = \varnothing$), it must be that both the small queue and the large queue are not empty. Indeed, if all remaining intervals were $< 1/n$, the average would be $< 1/n$; if all were $> 1/n$, the average would be $> 1/n$. Thus, both $S \neq \varnothing$ and $L \neq \varnothing$.

There is a bad choice here[2.1], but the right choice is to pop a small element s from S. To complete it, we also pop ℓ a large element from L. We have $s < 1/n < \ell$. The new tile is going to be made of two parts s and

[2.1] The temptation is to take a large interval, break it into a tile of size $1/n$, and the remainder, tile the part of size $1/n$ and continue. The problem is that the size of the queues remains the same, so that is a bad idea.

$\delta = 1/n - s$, where the latter part δ is taken from ℓ. That is, we have our tile made of two fragments of original intervals s and δ (labeled with the label of s and the label ℓ, respectively). The interval ℓ however paid for this dearly—its length now has shrunk to $\ell' = \ell - \delta = \ell - (1/n - s)$. The algorithm now outputs the new tile (made of two fragments of original intervals, as promised), and the residual interval ℓ' is now pushed back into the appropriate queue (into L if $\ell' > 1/n$, into M if $\ell' = 1/n$, and into S if $\ell' < 1/n$).

Analysis. In the latter case, the algorithm popped two intervals, pushed back one subinterval, and output one tile. Namely, the algorithm preserved the invariant that at each step one tile of length $1/n$ was output, and the total size of the queues shrunk by one in each iteration (in the first case where an item is taken from M, one interval is popped and none pushed back, also reducing the total queue size by one).

The correctness of the algorithm is now clear, as inductively it keeps the above invariant till the end. The algorithm computes n tiles (i.e. *buckets*), numbered say $0, \ldots, n-1$, and each tile contains one or two fragments.

Running time: Construction. Clearly, each iteration takes $O(1)$ time, and there are exactly n iterations.

The query algorithm: Picking a value. The algorithm picks a value $X \in [0, n)$ by calling *rand*(n). It then computes the bucket index $Y = \lfloor X \rfloor \in \{0, \ldots, n-1\}$. If this bucket has only a single interval (with label i) associated with it, it returns the label i. Otherwise, the bucket contains a primary fragment of length s (say label α), designated in advance, and an alias fragment of length $1/n - s$ (say labeled β). The algorithm computes the fractional offset $\Delta = X - Y \in [0, 1)$—this quantity being the distance of the random number X from the beginning of the bucket/tile that contains it (note that $\Delta < 1/n$). If $\Delta \leq s$, it returns the label of the primary α and otherwise it returns the label of the alias β.

(All we did here was the good old search over the suffix sums, implemented in a faster way, because of the nice structure computed.)

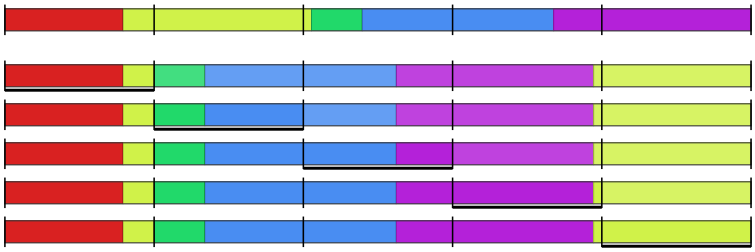

Figure 2.1: The alias method in action for $n = 5$.

Running time: Query. A few operations are performed, each one takes constant time, so the query time is $O(1)$ worst-case.

We thus got the following result.

Theorem 2.2.1. *Given a discrete distribution with n values, the above algorithm constructs, in $O(n)$ time, a data-structure that can compute samples according to this discrete distribution in $O(1)$ worst-case time per query. The algorithm relies on having access to the floor function, and the ability to sample uniformly from $[0, n)$ in constant time.*

2.3. Bibliographical notes

The linear-time algorithm described here is due to Vose [Vos91], building upon the original alias method developed by Walker [Wal74, Wal77]. The problem where the randomness source is simply an unbiased coin flip is also interesting, where one can achieve the entropy of the discrete distribution for the expected number of coin flips needed.

If the source of randomness is a number picked uniformly at random from some *discrete* set $\{1, \ldots, m\}$, then doing expected $O(1)$ time is possible. Using the alias method for the top level (pick the bucket), and then resolving the two values inside the bucket, reduces this case nicely to choosing only two values. Note that the expected query time is $O(1)$, but the worst case query time is unbounded, as no finite sequence of rational probabilities can lead to an irrational probability.

As a toy example, the reader can consider the problem of choosing between two values with probabilities $1/3$ and $2/3$ respectively, using an

unbiased coin. Describing the algorithm as a search tree over the binary strings leads easily to the conclusion that this tree has unbounded depth.

References

[Vos91] M. D. Vose. A linear algorithm for generating random numbers with a given distribution. *IEEE Transactions on Software Engineering*, 17(9): 972–975, 1991.

[Wal74] A. J. Walker. New fast method for generating discrete random numbers with arbitrary frequency distributions. *Electronics Letters*, 10(8): 127–128, 1974.

[Wal77] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Transactions on Mathematical Software*, 3(3): 253–256, 1977.