A Normal(?) Sequence

Aresh Pourkavoos

January 23, 2022

This idea is about OEIS entry A330731, which I submitted, and most of the information here is also available there. Nonetheless, I wanted to explore the sequence in a bit more detail with less of a wait time before publication, hence the separate source.

The sequence is infinite and binary, i.e. its entries are all either 0 or 1. I designed it to be *normal*, meaning that as the number of terms increases, every subsequence of the same length appears with equal frequency. For example, a randomly chosen subsequence of length 5 has a 1 in 32 chance of being 01001. At the moment, I still have not proven this property, but experimentally, this seems to be the case. However, I hope to prove that it is normal eventually, and in fact, I believe that the frequency of a given substring converges to its final value *faster* than a "random" sequence.

The sequence is defined by the following procedure to create the next digit, starting from the empty string:

- List all of the tails, or suffixes, of the sequence so far in order from longest to shortest. For example, the tails of 011 are 011, 11, 1, and the empty string.
- For each tail, count the number of times it occurs elsewhere in the sequence (i.e. other than the end), and note how many times 0 or 1 appears afterward. For the example above, 011 and 11 do not appear anywhere other than the end, but 1 appears in one other place (the middle), where it is followed by a 1.
- The *longest* tail where one bit appears afterward more often than the other decides which bit comes next, namely the bit which appears *less* often. Since 1 is followed by 0 zero times and by 1 once, the next bit is 0. This is why the occurrences of the empty string (and the bits that come after them) were not counted in the previous step: these numbers do not matter for the result. If they did, though, 0 would still be added since the empty string is followed by 0 once and by 1 twice.
- In the edge case where all of these comparisons result in a tie, 0 is the next bit. For example, in 010011, the tails are 010011, 10011, 0011, 011, 11, 1, and the empty string. All but the last two don't occur elsewhere, 1 is followed by 0 once and by 1 once, and the empty string is followed by both three times. Hence the next bit is 0 by default.

The naive algorithm to generate the sequence per the definition takes $O(n^3)$ time for the first n terms. This is because to generate each new term, the frequencies of $O(n)$ different tails are checked (from the whole sequence so far down to a relatively short tail), and each counting requires a pass over all terms, which takes $O(n)$ time. In other words, each term takes $O(n^2)$ time to generate. However, with the right data structures, I was able to reduce the time per term to $O(n)$, bringing the time for the first n terms to $O(n^2)$.

The following page contains a C program that prints the first 8192 terms of the sequence.

```c
#include <stdio.h>
#define N_TERMS 8192
int a[N_TERMS];
int b[N_TERMS];
int c[N_TERMS];
int cEnd = 0;
int main() {
  for (int n = 0; n < N_TERMS; n++) {
    c[n] = 0;
    c[cEnd] = n;
    cEnd = n;
    int newD = 0;
    int freq0 = 0;
    int freq1 = 0;
    int prevTail = -1;
    int j = c[0];
    while (j != 0) {
      int currTail = b[j-1];
      if (currTail != prevTail) {
        if (freq1 != freq0) break;
        freq0 = 0;
        freq1 = 0;
      }
      if (a[n-j] == 0) freq0++;
      else freq1++;
      prevTail = currTail;
      j = c[j];
    }
    if (freq1 < freq0) newD = 1;
    for (int numVisited = 0; numVisited < n; numVisited++) {
      int k = c[j];
      if (a[n-k] == newD) {
        b[k-1]++;
        j = k;
      } else {
        b[k-1] = 0;
        c[j] = c[k];
        if (cEnd == k) cEnd = j;
        c[k] = 0;
        c[cEnd] = k;
        cEnd = k;
      }
    }
    a[n] = newD;
    b[n] = 0;
    printf("%d", newD);
  }
  printf("\n");
  return 0;
}
```

A line-by-line breakdown:

- The arrays a, b and c are responsible for the algorithm's $O(n)$ space complexity. Since the program generates a fixed number of terms here, the arrays are initialized to their full length. However, they could also be implemented as unbounded arrays and have an element appended to them at the beginning of each iteration of the main loop.
- The $O(n^2)$ time complexity comes from the nested for loops. Each outer loop on line 8 generates a single term, and the loops on lines 17 and 30 take $O(n)$ steps each.