Advent of Code 2022 Day 6

Jacob Reinhart

January 6, 2023

1 Part 1

1.1 Problem Formulation

Given a Stream S of characters and an integer N, how many characters will need to be streamed from S before the previous N characters are all different?

Input: Stream S; Integer N
Output: $C :=$ Smallest $i \geq N$ such that $S[i], S[i-1], ..., S[i-(N-1)]$ all different

In the problem, $N = 4$ and S is the provided file input.

1.2 Naive Algorithm

For simplicity, assume a data structure wrapping S that has

- $S.readNext()$ which scans the next character from S and operates in $O(U)$ time.
- $S.getPrevious(i)$ which, for $1 \leq i \leq N$, returns the char scanned i scans ago, and operates in $O(G)$ time.
- $S.getNumRead()$ which returns the of calls to $S.next()$ made over the course of the algorithm, and can be trivially implemented in constant time.
- a set of underlying data structures that take up $O(P)$ additional space

The most naive implementation would be to read from S, and after every read, scan the previous N characters for a duplicate in any other of the previous N characters, stopping when no duplicates are found. This way, we would expect each read take $O(U + N^2G)$ time.

1.3 Improved Algorithm

However, the naive algorithm does a lot of duplicate work because **any pair of duplicates not involving the most-recently read character ($S.getPrevious(1)$) would have been identified on a previous read.** Thus, there should be a way to get the solution by only scanning the most recently read character for duplicates against, at most, the rest of the previous N.

Define a variable J with the invariant over the reading loop that $\boldsymbol{J}$ **is the minimal value such that** $\boldsymbol{S.getPrevious(J) = S.getPrevious(i), \exists i < J}$. Once $J > N$, then there are no duplicates in the previous N reads and the end condition is satisfied.

Initially, $J = 1$. Inductively, when we read another character we would increment J, then if $\exists j < J$ such that $S.getPrevious(j) = S.getPrevious(1)$, then we should update $J :=$ the smallest such j. This way, we would expect each read to add $O(U + NG)$ time with no additional space compared to the naive method.

Algorithm 1: FindStepsUntilLastNAllDiff

Input: Stream S; Integer N
Output: $C :=$ Smallest $i \geq N$ such that $S[i], S[i-1], ..., S[i-(N-1)]$ all different

```
1  J := 1;
2  while J ≤ N do
3      J++;
4      S.readNext();
5      x := S.getPrevious(1) ;                    // the char just scanned
6      for i = 1, ..., J do
7          if x = S.getPrevious(i) then
8              J := i;
9              break;
10 return S.getNumRead();
```

Time: $O(U + NG)$ per read, so $O(C(U + NG))$ overall.
Space: $O(P)$

1.4 Implementing The Reader

We now turn our attention to a specific implementation of S to optimize for G, U, and P.

1.4.1 Naive Implementation

The most naive implementation would be to store S as a basic array:

- Initialize(): $P := [] ; c := 0$;
- Read(): $P.append(scanner.next()); c$++;
- Get(i): $P[c - i]$

Under this interpretation $G = O(1)$, but $P = O(C)$. We don't know C at the start, but can reasonably infer that $C >> N$, plus we will have to be periodically allocating more memory on each scan, leading to $U = O(1)$ amortized, so this is not a good solution.

1.4.2 Improved Implementation

Another approach might be to recognize that *we never need more than the last N previous*. That would allow us to bound the size of P to $O(N)$. One such implementation looks like this:

- Initialize(): $P[1, ..., N]$ empty; $c := 0$;
- Scan(): $P[1, ..., N] := P[2, ..., N] + scanner.next()$; c++;
- Get(i): $P[i]$;

Under this method, we bring the space down to $P = O(N)$, but also bump up the time complexity to $U = O(N)$ because we would have to have $P[1, ..., N - 1] := P[2, ..., N]$; in every scan.

1.4.3 Efficient Implementation

A better solution would be to use a "clock-face" implementation:

- Initialize(): $headIdx := -1$; $P[0, ..., N - 1]$ empty; $c := 0$
- Scan(): $headIdx := (headIdx + 1)\%N$; $P[headIdx] := scanner.next(); c$++;
- Get(i): $P[(headIdx - i)\%N]$;

This solution keeps $P = O(N)$ and leaves us both U and G at constant time.

Thus, most efficient algorithm and most efficient implementation lead to a complexity of $\boldsymbol{O(CN)}$ time ($O(N)$ time per read) with $\boldsymbol{O(N)}$ additional space.

2 Part 2

Exactly the same, but with larger $N = 14$.