Assignment One

Your Name
Your.Name@Marist.edu

January 10, 2023

1 Problem One

1.1 The Data Structure

An n-element array of integer pairs: ($currentCount$, $predecessorSum$) will, once initialized in the preprocessing step, support Member, Less, and Range operations as described in Section 1.3 in $O(1)$ (constant) time.

For illustration, consider the case where $S = (1, 2, 3, 4, 7, 7, 7, 7, 7, 8, 9, 10, 10, 10, 10, 10, 10, 14, 16)$. Here we have $m = 19$ elements in S ranging in value from 1 through $n = 16$. The array for S after preprocessing is given in Figure 1.

1.2 Preprocessing

There are two preprocessing steps, Tally and Predecessor Sum, each of which is $O(m + n)$ as we will see in Equations 1 and 2 below. This makes the overall performance $O(m + n)$ because $2 \cdot O(m + n) = {}^{1}O(m + n)$.

1.2.1 Pass one: Tally

For each element i in S we'll store its number of occurrences at $A[i].currentCount$.

Data: collection S
Result: array A containing tallys for each element i in S
1 allocate $A[n]$;
2 **for** $j \leftarrow 1$ **to** n **do**
3 | $A[j].currentCount \leftarrow 0$;
4 **end**
5 **foreach** *element i in S* **do** // There are m elements in s.
6 | $A[i].currentCount \leftarrow A[i].currentCount + 1$;
7 **end**

Algorithm 1: Tally

[1] This is an abuse of the notation, but if it's okay in the CLRS [1] book I hope it's okay here.

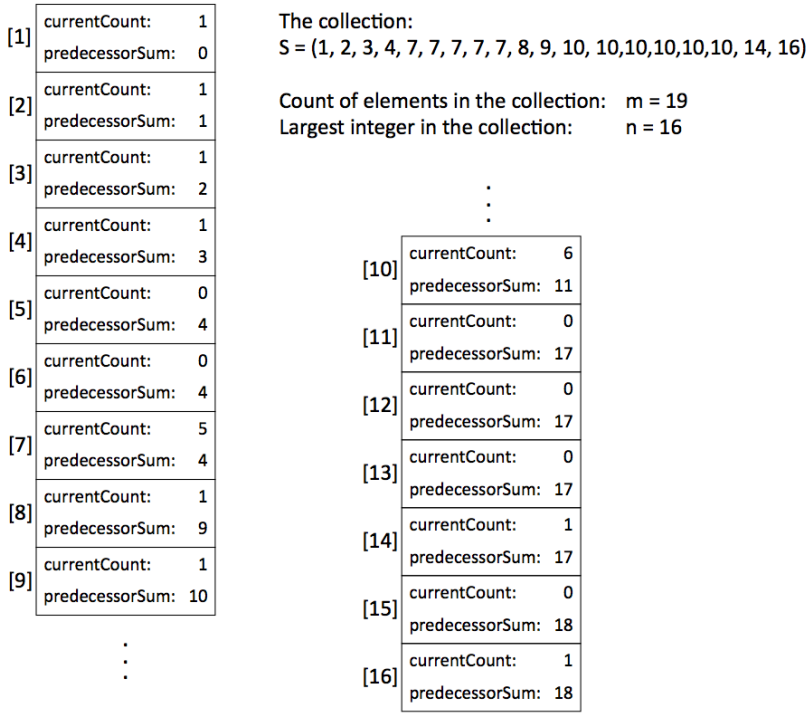

Figure 1: Example array built from 19 values, 10 of them unique.

Asymptotic Analysis
We are given m integer values (the size of the collection), each in the range $[1..n]$ (the size of the domain) and want to iterate over them, computing the tally for each. Consider Algorithm 1. Line 1 executes in constant time. Lines 2 through 4 execute in $O(n)$ time because we are iterating from 1 to n. Lines 5 through 7 execute in $O(m)$ time because we are iterating over all the elements of S, of which there are m.

So, for pass one, we have :

$$\begin{aligned} O(pass1) &= constant + O(n) + O(m) \\ &= O(n) + O(m) \\ &= O(m+n) \end{aligned} \tag{1}$$

1.2.2 Pass two: Predecessor Sum

Once the tally is done we need to make another pass over A to compute, for each $A[i] : i > 1$, the sum of all of its predecessors ($A[1]..A[i-1]$) and store that in $A[i].predecessorSum$.

Data: array A containing tallys for each element i in S
Result: a new and improved array A, now with the predecessor sums
1 $A[1].predecessorSum \leftarrow 0$;
2 **if** $n > 1$ **then**
3 | **for** $k \leftarrow 2$ **to** n **do**
4 | | $A[k].predecessorSum \leftarrow A[k-1].predecessorSum + A[k-1].currentCount$;
5 | **end**
6 **end**

Algorithm 2: Predecessor Sum

Asymptotic Analysis
Looking at Algorithm 2, line 1 is assignment, a constant time operation. If lines 3 through 5 execute at all, they do so in $O(n)$ time because we make $n-1$ iterations over line 4, which is assignment and array lookups, both constant time operations. For pass two, we have :

$$\begin{aligned} O(pass2) &= constant + O(n) \\ &= O(n) \\ &= O(m+n) \text{ which is ok so long as } m > 0 \end{aligned} \tag{2}$$

1.3 Operations

1.3.1 Member

Input: parameter i
Data: a new and improved array A, replete with tallys and predecessor sums
Output: $True$ if i exists in S, $False$ otherwise
1 **if** $(i \geq 1) \wedge (i \leq n)$ **then**
2 | return $(A[i].currentCount > 0)$;
3 **else**
4 | return $False$;
5 **end**

Algorithm 3: Member

Asymptotic Analysis
Looking at Algorithm 3, line 1 consists of comparisons, which are constant time operations. Line 2 is an array lookup and a comparison, both constant time operations. The remaining parts of the algorithm (including the rest of line 2) are for program control, and not considered in this analysis. Since all parts of the algorithm execute in constant time, the whole thing executes in constant time and is therefore $O(1)$.

1.3.2 Less

Input: parameter i
Data: a new and improved array A, replete with tallys and predecessor sums
Output: The number of elements in S that are strictly less than i.
1 **if** $(i \geq 1) \wedge (i \leq n)$ **then**
2 | return $A[i].predecessorSum$;
3 **else**
4 | return 0;
5 **end**

Algorithm 4: Less

Asymptotic Analysis
Looking at Algorithm 4, line 1 consists of comparisons, which are constant time operations. Line 2 is an array lookup, a constant time operation. The remaining parts of the algorithm (including the rest of line 2) are for program control, and not considered in this analysis. Since all parts of the algorithm execute in constant time, the whole thing executes in constant time, and is therefore $O(1)$.

1.3.3 Range

Input: parameters i and $j : i \leq j$
Data: a new and improved array A, replete with tallys and predecessor sums
Output: The number of elements in S that are in the range $[i..j]$.

```
1 if (i ≥ 1) ∧ (i ≤ n) ∧ (j ≥ 1) ∧ (j ≤ n) ∧ (i ≤ j) then
2 |   return (A[j].currentCount + A[j].predecessorSum − A[i].predecessorSum);
3 else
4 |   return 0;
5 end
```

Algorithm 5: Range

Asymptotic Analysis
Looking at Algorithm 5, line 1 consists of comparisons, which are constant time operations. Line 2 consists of array lookups, addition, and subtraction, all constant time operations. The remaining parts of the algorithm (including the rest of line 2) are for program control, and not considered in this analysis. Since all parts of the algorithm execute in constant time, the whole thing executes in constant time, and is therefore $O(1)$.

2 Problem Two

3 Problem Three

References

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2022.

4 Appendix

4.1 Some JavaScript source code listings

```
var A = [5,0,5,6,6,8,45,50];

function solve(A) {
    // Base case to stop the recursion.
    if (A.length == 1) {
        if (A[0] % 5 == 0) {
            var retVal = 1;
        } else {
            var retVal = 0;
        }
        return retVal;
    } else {
        // Divide.
        var divPoint = Math.floor( A.length / 2);
        var left  = A.slice(0, divPoint);
        var right = A.slice(divPoint, A.length);

        // Conquer.
        var left5s   = solve(left, level+1);
        var center5s = straddle(left, right);
        var right5s  = solve(right, level+1);

        // Combine.
        return Math.max(left5s, Math.max(center5s, right5s));
    }
}

function straddle(left, right) {
    var retVal = 0;
    if ((left[left.length-1] % 5 == 0) && (right[0] % 5 == 0)) {
        // Count back the 5's on the left going from right to left.
        var leftCount = 0;
        var index = left.length-1;
        while ( (index >= 0) && (left[index] % 5 == 0) ) {
            index--;
            leftCount++;
        }
        // Count forward the 5's on the right going from left to right.
        var rightCount = 0;
        while ( (rightCount < right.length) && (right[rightCount] % 5 == 0) ) {
            rightCount++;
        }
        // Return the sum of the straddling 5s on the left and right.
        retVal = leftCount + rightCount;
    }
    return retVal;
}
```