1 Question 1

Note on notation: we will represent the set with name s as S.

Intuition

Let's model this problem with the help of disjoint sets. Given the array $A[1, \dots n]$, we will partition indices $1, n$ into disjoint sets by their solution to operation C. That is, $\forall i$ such that $OperationC(i) = k \iff i \in K$. That is, the indices with the same solution for operation C are in the same set K with "name" k. Note that all the indices in the set have to be a consecutive sequence. If it's not a consecutive sequence, then there exists such index j in between the indices in K such that $A[j] = 0$, and so $\exists h \in K, Operationc(h) \neq k$, violating our definition. Initially, since A contains only 0's, then we have n disjoint sets, one for each index. On setting $A[i] = 1$, we can see that $OperationC(i) = OperationC(i+1)$ - that is, since $A[i]$ is no longer a zero, the leftmost zero for i is now the same as the leftmost zero for $i + 1$. According to our definition above, i and $i + 1$ must now be in the same set. We can accomplish this by an $Union$ operation between the sets i and $i + 1$ belong to. When we need to recover which set i is in, that is the solution to operation C, we can simply call $Find(i)$. Something interesting to note is that $A[i]$ will become 1 only once, and so we only union i and $i + 1$ once. Given these requirements, let's keep track of these disjoint sets in another array U (in addition to A containing the 0-1 data). To implement $Union$ and $Find$ for U, we use the same algorithm taught in class using tree linking and path compression. Explicitly:

1. $OperationA(i)$: Call $Union(i, k)$, where k is the set name of the set that $i + 1$ belongs to. This can be found in constant time with the help of supplementary array Q described below. If K is smaller than I, $Union$ will merge K into I. However, $OperationC$ should return k, not i. Thus, we will change the name of the union'd set from i to k via a swap on the previous root nodes i and k after the union, thus making k the root of the union'ed set. This is done in constant time.
2. $OperationB(i)$: simply return $A[i]$
3. $OperatonC(i)$: $Find(i)$.

In addition to the union find data structure, we will also keep another array Q that we will we use to store additional information. The goal of this data structure is to determine the root/set of the leftmost element (smallest index) in some set I in constant time. Note that in this particular application of union find, all sets are within a contiguous section of the array and all "unions" will only occur between the head/root (since we flip $A[i]$ from 0 to 1) of a subarray/set and the tail (leftmost child) of another subarray/set. Let the following rules define array Q of size n.

1. Property: Let i be the name of a set that still exists (has not been absorbed into a larger set), then the following is true: $\exists k \geq 0$, ($Q[i] = k \iff Q[i-$

$k] = -k) \wedge j \in [i-k, i], j \in I$. This also means that $OperationC(j) = i$ from our definition above. This means that everything in the contiguous subarray from $i-k$ to i is in set I, thereby meaning the set I has size $k+1$ and that the leftmost right zero for all indices in this range is at index i.

2. Initially $Q[i] = 0 \forall i \in [1, n]$.

3. We need to maintain this property when we union disjoint sets. Note that a find will not impact this array at all since set membership does not change. We can only ever union sets i and the set that $i+1$ belongs to (i.e. disjoint sets when we flip $A[i] = 0$ to 1). In other words, in Q, the two sets are represented by two contiguous subarrays "next to each other". With our property above, we know that when we want to flip $A[i] = 1$, index $i+1$ belongs to set K such that $k = i+1-A[i+1]$. Now there are two cases when we call $Union(i, k)$. Note that both of these take $O(1)$ time as it is just array lookup and changing values.

   (a) If $|K| \geq |I|$, then we set $A[i - A[i]] -= A[k] + 1$ and we set $A[k] += 1 - A[i - A[i]]$. Note that this happens simultaneously (for example we store $A[k]$ in a temp variable so we increment using the old value than the new value).

   (b) If $|K| < |I|$, we do the exact same as above except that after we do the union, we notice that since the larger of the sets was I, the elements in the contiguous subarray $[i - A[i] \ldots k]$ no longer belong in K, but in I instead! This violates our property above! So, to account for this we change the name of set I to k instead. This can be easily done by keeping track of the root node of K before linking and swapping it with the root node of the union'd set (which contains i) after linking. Thus, the union'd set K now has a root of k. This keeps our property.

Analysis

1. Operation A: Finding the set name of $i+1$ is just constant time as it is array index lookup specified above. $Union$ given the roots of two sets is just $O(1)$ as it is just adjusting pointers. Don't forget we also update Q on $Union$. However, this simply consists of array indexing for i and $i+1$, which is $O(1)$. This results in a total runtime of $O(1)$.

2. Operation B: $O(1)$ as this is an array lookup.

3. Operation C: This is just amortized $O(\alpha(m.n))$ from the results in class.

Algorithm

Algorithm 1 Operations

procedure OperationA(A, i, Q, U)
  $k \leftarrow i + 1 - Q[i + 1]$ ▷ root of the second set (i + 1)
  $swap \leftarrow False$
  if $A[k] < A[i]$ **then**
    $swap \leftarrow True$
  $temp \leftarrow A[i - A[i]]$
  $A[i - A[i]] \leftarrow A[i - A[i]] - A[k] - 1$
  $A[k] \leftarrow A[k] - A[temp] + 1$
  U.Union(i, k)
  if $swap$ **then**
    ChangeName(i, k) ▷ change name of set I to k by a swap between the previous roots

procedure OperationB(A, i, Q, U)
  return $A[i]$

procedure OperationC(A, i, Q, U)
  return U.Find(i)