Homework #5 Solutions
Introduction to Algorithms
601.433/633
Spring 2020

Due on: Tuesday, March 24th, 12pm
Where to submit: On Gradescope, please mark the pages for each question

1 Problem 1 (25 points)

1.1 Problem 1.1 (10 points)

Suppose we wish not only to increment a counter of length $m \in \mathbb{N}$ but also to reset it to zero (i.e., make all bits in it 0). Counting the time to examine or modify a bit as $\Theta(1)$, show how to implement INCREMENT and RESET operations on a counter (represented as an array of bits) so that *any* sequence of n operations takes $O(1)$ amortized time per operation.

You may assume that the counter is initially zero and that you may access and modify any specific bit (say bit $j \in [m]$) in $O(1)$ time. Additionally, you may assume that the total count in the counter never exceeds $2^m - 1$ during the course of the n operations.

Prove correctness and running time of your algorithms.

(Hint: Keep a pointer to the highest-order 1.)

Proof. We introduce a new field $max[A]$ to hold the index of the high-order 1 in A. Initially, $max[A]$ is set to 1, since the low-order bit of A is at index 0, and there are initially no 1's in A. The value of $max[A]$ is updated as appropriate when the counter is incremented or reset, and we use this value to limit how much of A must be looked at to reset it. By controlling the cost of $RESET$ in this way, we can limit it to an amount that can be covered by credit from earlier $INCREMENT$s.

```
$INCREMENT(A)$
**while** $i < length[A]$ and $A[i] = 1$ **do**
```

```
$INCREMENT(A)$
$i \leftarrow 0$
while $i < length[A]$ and $A[i] = 1$ do
    $A[i] \leftarrow 0$
    $i \leftarrow i + 1$
    if $i < length[A]$ then
        $A[i] \leftarrow 1$
        if $i > max[A]$ then
            $max[A] \leftarrow i$
        end if
    else
        $max[A] \leftarrow -1$
    end if
end while
```

```
$RESET(A)$
for $i \leftarrow 0$ to $max[A]$ do
    $A[i] \leftarrow 0$
end for
$max[A] \leftarrow -1$
```

As for the counter in the book, we assume that it costs \$1 to flip a bit. In addition, we assume it costs \$1 to update $max[A]$.

Setting and resetting of bits by $INCREMENT$ will work exactly as for the original counter in the book: \$1 will pay to set one bit to 1; \$1 will be placed on the bit that is set to 1 as credit; the credit on each 1 bit will pay to reset the bit during incrementing.

In addition, we will use \$1 to pay to update max, and if max increases, we will place an additional \$1 of credit on the new high-order 1. (If max doesn't increase, we can just waste that \$1— it won't be needed.) Since $RESET$ manipulates bits at positions only up to $max[A]$, and since each bit up to there must have become the high-order 1 at some time before the high-order 1 got up to $max[A]$, every bit seen be $RESET$ has \$1 credit on it. So the zeroing of bits of A by $RESET$ can be completely paid for by the credit stored on the bits. We just need \$1 to pay for resetting max.

Thus, charging \$4 for each $INCREMENT$ and \$1 for each $RESET$ is sufficient, so the sequence of n $INCREMENT$ and $RESET$ operations takes $O(n)$ time. □

1.2 Problem 1.2 (15 points)

Design a data structure to support the following two operations for a set S of integers, which allows duplicate values:

- INSERT(S, x) inserts x into S.
- DELETE-LARGER-HALF(S) deletes the largest $\lceil |S|/2 \rceil$ elements from S.

Explain how to implement this data structure so that any sequence of m INSERT and DELETE-LARGER-HALF operations runs in amortized $O(1)$ time per operation. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

Prove the running time of your implementation.

Proof. We can use a simple array A to be our underlying data structure. Array A contains a set S of integers. The operations are as follows.

- $INSERT(S, x)$: The element can be inserted to the last element in A.
- $DELETE\text{-}LARGER\text{-}HALF(S)$: First, find the median mid of S using array A. We then partition S into two subsets, S_1 and S_2, where all the elements in S_1 are less than mid and the elements in S_2 are larger than and equal to mid by comparing each element in A with mid. Last, we delete S_2 and set $S = S_1$.

Suppose there are m operations performed on S. For each $i = 1, 2, \ldots, n$, let c_i be the actual cost of the ith operation and A_i be the data structure that results after applying the ith operation to data structure A_{i-1}. Since we use the linear time algorithm for finding the median, the running time of the algorithm for discarding the larger half is $O(m)$ for an m-element array. Suppose the actual time for discarding the larger half is km for some constant k. We define the *potential function* Φ on each data structure A_i to be

$$\Phi(A_i) = 2k \times |A_i|,$$

where $|A_i|$ is the number of elements in array A_i. For an operation i, we analyze the cost by considering the following two cases.

$INSERT(S, x)$: The actual cost c_i for insertion is 1. Then, the amortized cost

$$\widehat{c_i} = c_i + \Phi(A_i) - \Phi(A_{i-1}) = 1 + 2k \times |A_i| - 2k \times |A_{i-1}| = 1 + 2k,$$

since $|A_i| = |A_{i-1}| + 1$ when inserting an element to A_{i-1}. The amortized cost therefore is $O(1)$.

$DELETE\text{-}LARGER\text{-}HALF(S)$: As mentioned above, the actual cost for deleting the larger half is km if there are m elements. Then, the amortized cost

$$\begin{aligned}\widehat{c_i} &= c_i + \Phi(A_i) - \Phi(A_{i-1}) \\ &= k \times |A_{i-1}| + 2k \times |A_i| - 2k \times |A_{i-1}| \\ &= k \times |A_{i-1}| + 2k \times \frac{1}{2}|A_{i-1}| - 2k \times |A_{i-1}| \\ &= k \times |A_{i-1}| + k \times |A_{i-1}| - 2k \times |A_{i-1}| = 0\end{aligned}$$

since $|A_i| = \frac{1}{2}|A_{i-1}|$ when deleting the larger half of A_{i-1}. The amortized cost therefore is 0.

Therefore, the m operations run in $O(m)$ time. $\square$

Problem 2 (10 points)

Let $G = (V, E)$ be a directed graph. $a \in V$ is a *central* vertex if for all $b \in V$ there exists a path from a to b. Provide an $O(|V| + |E|)$ time algorithm to test whether graph G has a central vertex.

Prove the correctness of your algorithm and analyze the running time.

Proof. First, we notice that the directed graph G can be cyclic. Is there any method we can get rid of all the cycles? Let's consider one possible algorithm described as following.

1. Use Kosaraju's algorithm to find all strongly connected components of the given graph G.
2. Build a new graph G^* by shrinking each SCC of G into one representative vertex and keeping the edges between SCCs.
3. If there is more than one vertex that has no incoming edges, return FALSE; else, return TRUE.

Running Time: Assuming that the graph is presented by adjacency list, Kosaraju's algorithm runs in $O(|V| + |E|)$ time. Building the new graph G^* and finding outgoing-only vertices take $O(|V| + |E|)$. In total, the time complexity is $O(|V| + |E|)$.

Correctness: I eliminated some details here. Briefly, let's consider the graph G^*. Clearly, G^* is a DAG. DAG has at least one vertex with no incoming edges (we can use proof by contradiction, eliminated here). So the only case left, one vertex with no incoming edges is actually the central vertex. This is because all other vertices have at least one incoming edge and can be reached by one of vertices. If there are more than one such vertex, e.g. two vertices, one cannot be reached by the other, which contracts the definition of central vertex. □

2 Problem 3 (15 points)

You're helping some analysts monitor a collection of networked computers, tracking the spread of fake information. There are n computers in the system, labeled $C_1, C_2, ..., C_n$, and as input you're given a collection of trace data indicating the times at which pairs of computers communicated. Thus the data is a sequence of ordered triples (C_i, C_j, t_k); such a triple indicates that C_i and C_j exchanged bits at time t_k. There are m triples total.

We'll assume that the triples are presented to you in sorted order of time. For purposes of simplicity, we'll assume that each pair of computers communicates at most once during the interval you're observing. The analysts you're working with would like to be able to answer questions of the following form: If the fake information was generated by computer C_a at time x, could it possibly have been sent to C_b by time y? The mechanics of communicating the information are simple: if a computer containing the fake information C_i communicates with another computer C_j that hasn't received that information yet by time t_k (in other words, if one of the triples (C_i, C_j, t_k) or (C_i, C_j, t_k) appears in the trace data), then computer C_j receives the fake information, starting at time t_k.

The fake information can thus spread from one machine to another across a sequence of communications, provided that no step in this sequence involves a move backward in time. Thus, for example, if C_i has received the fake information by time t_k, and the trace data contains triples (C_i, C_j, t_k) and (C_j, C_q, t_r), where $t_k \leq t_r$, then C_q will receive the fake information via C_j. (Note that it is okay for t_k to be equal to t_r; this would mean that C_j had open connections to both C_i and C_q at the same time, and so the information would have been sent from C_i to C_q.)

Design an algorithm that answers questions of this type: given a collection of trace data, the algorithm should decide whether the fake information generated by computer C_a at time x could have been received by computer C_b by time y. The algorithm should run in time $O(m + n)$.

Prove correctness and running time as usual.

Proof. We will first create a graph $G = (V, E)$. For each element (C_i, C_j, t_k) in the trace data we create vertices (C_i, t_k) and (C_j, t_k) and add directed edges in both directions between the vertices. Additionally, we add an edge from (C_i, t') to (C_i, t_k) where t' is the preceding time step (the previously last one) C_i appeared in. We do the same for C_j.

Constructing this graph takes $O(m)$ time by maintaining an array of linked lists – a linked list of time steps (in descending order) associated with each machine C_i.

Given a pair of machines C_a, C_b and a pair of time steps x, y to check whether the fake information introduced at time x in machine C_a could have been received by machine C_b by time y – we run DFS from every vertex (C_a, t) such that $t \geq x$ is the smallest time step larger than x that appears in the linked list for C_a. If we reach a vertex (C_b, t') such that $t' \leq y$ then we return YES else we return NO.

Correctness If the algorithm returns YES, then there exists a path from C_a to C_b in increasing order of time starting at a time after x and ending at a time before y. Hence, by the rules by which fake information spreads, C_b could have received the information.

If the algorithm returns NO – assume for contradiction that C_a could have sent the information to C_b by time y if C_a received the information at time x. Then, by how we constructed the graph, there must be a sequence of machines with open connections in non-decreasing order of time. It is easy to show that there must be a directed path from a vertex (C_a, t) (for $t \geq x$) to another (C_b, t') (for $t' \leq y$) in G. This is a contradiction to the fact the algorithm returned NO.

Running Time The running time bound follows by showing that the time to create G is $O(m)$ when the trace data is presented in sorted order of time and running DFS requires $O(m+n)$ time.

$\square$