Homework #5
Introduction to Algorithms
601.433/633
Spring 2020

Mou Zhang
Due on: Saturday, March 28th, 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.)

Solution: We need to set a variable MaxIndex to remember the place of the highest-order 1. If the counter is zero or we just reset the counter, then the MaxIndex is set to -1, else we will update MaxIndex with the increment steps. This value is designed to limit the number of operation we need to reset the counter.

Algorithm 1 Increment and reset

```
1:  procedure INCREMENT(A)
2:      i ← 0
3:      while i < A.length and A[i] = 1 do
4:          A[i] ← 0
5:          i ← i + 1
6:      end while
7:      if i < A.length then
8:          A[i] ← 1
9:          if i > MaxIndex then
10:             MaxIndex ← i
11:         end if
12:     else
13:         MaxIndex ← −1
14:     end if
15: end procedure
16: procedure RESET(A)
17:     for i ← 0, MaxIndex do
18:         A[i] ← 0
19:     end for
20:     MaxIndex ← −1
21: end procedure
```

Proof of correctness:

Proof. The increment step is mainly what is taught in the class and on the book, which is already proved. The only difference is about the update of MaxIndex,which is obviously right since it's only recording the highest-order 1. If current place i is larger than MaxIndex, then i is the new MaxIndex. In the Reset Step, since each bit that need to be reset must become the high-order1 at some time before high-order gets to MaxIndex, their place are all between 0 and MaxIndex. So we just simply set all bits between 0 and MaxIndex to 0 and set MaxIndex to 0. □

Proof of running time:

Proof. We are going to use the accounting method to prove this problem. Let's assume that it costs 1$ to flip a bit. It also cost 1$ to update MaxIndex. Each step will need 4$ for increment step and 1$ for reset step. In the increment steps, 1$ will be paid to set one bit to 1, and we place 1$ on the bit as credit to be used later when we flip the bit back to 0. Besides, we also need 1$ to update the MaxIndex, and if MaxIndex increases ,we will place an additional 1$ of credit on the new high-order 1 saved for the reset step. For the resetting part, we only need 1$ credit to be set MaxIndex to -1. Since each bit that need to be reset must become the high-order 1 at some time before high-order gets to MaxIndex, they all have 1 credit prepared for the reset step. So we only need to pay 1$ in the reset step to reset MaxIndex. Since each increment step needs 4$ and each reset step needs 1$, the operation time is O(n). □

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.

Solution: We can use a array A to solve the problem. The algorithm is shown as below.

- INSERT(S, x) inserts x into S.
  We can just add the new element to the end of the array A.
- DELETE-LARGER-HALF(S) deletes the largest $\lceil |S|/2 \rceil$ elements from S.
  We can find the medien number *mid* using the quickselect algorithm(which is designed to find the kth smallest number and takes O(n) time). After that,we can compare each number to the *mid* to find the half number of integers that are larger than or equal to *mid* and delete them.

Proof of correctness:

Proof. In the Insert part, I have insert the element in the array so it's obviously right. In the DELETE-LARGER-HALF Step, since we have found the larger half of all numbers bigger or euqal to *mid*, these numbers are the largest $\lceil |S|/2 \rceil$ elements from S. Deleting these numbers, then we delete the largest $\lceil |S|/2 \rceil$ elements from S □

Proof of complexity:

Proof. In the Insert Step, it is obvious that it takes O(1) time. In the DELETE-LARGER-HALF Step, since we are using a linear time to find the medien and doing linear time of comparison, the running time of this is O(m)(m is the number of elements in S when doing DELETE-LARGER-HALF Step). Let's assume the time spent is km, in which k is a constant.
Assuming S_i is the state after the *i*th step and m_i is the number of elements after the *i*th step
Let's define a potential function $\Phi(S_i) = 2 \times k \times m_i$
In the Insert part, the cost of each time of insertion is 1, so $\hat{c_i} = c_i + \Phi(S_i) - \Phi(S_{i-1}) = 1 + 2 \times k \times m_i - 2 \times k \times m_{i-1} = 1 + 2k$ ($m_i = m_{i-1} + 1$). Therefore, the amortized cost of Insert Step is O(1).
In the DELETE-LARGER-HALF Step, Since we know the cost of DELETE-LARGER-HALF Step is km, then the amortized cost is $\hat{c_i} = c_i + \Phi(S_i) - \Phi(S_{i-1}) = k \times m_{i-1} + 2 \times k \times m_i - 2 \times k \times m_{i-1}$. Since $m_i = m_{i-1}/2$, then $\hat{c_i} = k \times m_{i-1} + 2 \times k \times \frac{m_{i-1}}{2} - 2 \times k \times m_{i-1} = 0$. Then the amortized cost for DELETE-LARGER-HALF Step is 0.
Therefore, the amortized running time of each step is O(1). □

2 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.

Solution: Let's define a vertex to be finished in DFS if a recursive call for its DFS is over, i.e., all descendants of the vertex have been visited. Then if there exist central vertex, then one of the central vertices is the last finished vertex in DFS. The algorithm below is to find the the last finished vertex.

Algorithm 2 Find Mother Vertex

```
1:  Set vis[A.length] all false
2:  procedure DFSTRAVERSE(Graph G, vertex s)
3:      mark s as visited in vis[] array
4:      for all neighbors w of s in G do
5:          if w is not visited then DFSTraverse(G,w)
6:          end if
7:      end for
8:  end procedure
9:  procedure FIND MOTHER VERTEX(GraphG)
10:     for all nodes s in Graph G do
11:         if s is not visited then
12:             DFSTraverse(G, s)
13:             v ← s
14:         end if
15:     end for
16:     Clear vis[] array
17:     DFSTraverse(G, v)
18:     for all nodes s in Graph G do
19:         if s in not visited then
20:             return false
21:         end if
22:     end for
23:     return true
24: end procedure
```

Proof of correctness:

Proof. What we need to prove is that if there exist central vertex (or vertices), then one of the central vertices is the last finished vertex in DFS. Assume the last finished vertex is v. What we need to prove is that there is no other non-central vertex u such that edge $\rightarrow$ exists. There are two cases.
The first case is DFSTraverse for u is called before v. In this case, if u $\rightarrow$ v exists, than u must be finished after v because u is able to reach v and DFSTraverse tends to finish all u's descendants before to finish u.
The second case is DFSTraverse for v is called before u.If so, if u $\rightarrow$ v exists, than u must either be reachable to v(which means u is a central vertex too. Contradict.) or be finished after v(which means v is not the last finished vertex. Contradict.)
So, there is no other non-central vertex u such that edge $\rightarrow$ exists. Therefore, if there exist central vertex (or vertices), then one of the central vertices is the last finished vertex in DFS. Thus what we need to do is to check whether it is a central vertex, which can be done with single DFSTraverse starts from v. □

Proof of running time For lines 10-15, it's a DFS in the whole graph G and it is well-known that the time complexity of DFS in a Graph is O(V + E).This is because that each vertex and edge in graph G is only used at most twice in the whole DFS. The vertex is only used once and then it is marked to not use it again. And since each vertex is only used once, each edge is only used at most twice - for vertices on both end of the edge. For line 17, the DFSTraverse takes less or equal time than DFS in the whole graph G, which means its running time is O(v + E). For lines 18-22, the running time is O(V). So the total running time is O(V + E) + O(V + E) + O(V) = O(V + E).

3 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.

Solution:
Step 1: Run through the whole data and skip/delete all data with time earlier than the time *x* when fake news first appears.
Step 2: Create a hashset A of computer number to save the computer number of all computers that has fake data. Initially, A only contains the computer number of

initially affected computer C_a.
Step 3: Get to the next time t_k in trace data, and find all trace data that is in time t_k. Using these data to establish a graph G, each trace data is an edge and the 2 mentioned computers are 2 vertices.
Step 4:Find all connected component in G with DFS. If two vertices can find each other in the same DFS traverse, than they are in the same connected component.
Step 5: For each component, check whether each vertices is in A. If any vertices in a component is in A, which means it contains fake data, then the who component contains fake data now. So we should add every vertices in that component to A.
Step 6: Repeat Step 3, 4 and 5 as a loop until the next time we get is bigger than *y*. Stop the repeating. At this time, A contains all computers which has got the fake data. Return whether C_b is in A.

Proof of correctness:

Proof. This algorithm is obviously correct because it is a simulation of how the fake news spread. In Step 1, we delete the data before time x because at that time the fake news has not generated so these data are useless. In Step 2, once a computer has fake data, it will always has fake data. So keep a set of all computers that have fake data is right. In Step 3, 4 and 5,it is okay to spread fake news with 2 trace data k and r when t_k to be equal to t_r. So in the same time t_k, one computer in the connected component gets the fake data means that all computers in the connected component get fake data too. So what I do is to find all connected component with DFS and treat each connected component as a whole. And In Step 6, since all computers with fake news are in A, we can simply check whether C_b is in A to decide whether C_b has fake data. □

Proof of running time

Proof. In Step 1, deleting data at most takes O(m) time because there is only m lines of data. In Step 2, creating hashset only takes O(1) time. In Step 3, we set the graph. Since every edge needs to be generated with one trace data, the sum of all number of edges in all graphs generated in the Step 3 in every loop is no more than O(m). The sum of all vertices that are generated is also O(m) since 1 edge has at most 2 vertices. So the total time to generate graphs in all loops is O(m). In Step 4, the total time of DFS in all loops takes O(m) time since there's only O(m) vertices and edges. In Step 5, since A is a hashset, checking whether it contains a number

or not takes O(1) time. Since there are summing up O(m) vertices in all loops, this checking only takes O(m) time in total. Adding component to A takes O(n) time in total in all loops since there's at most n vertices. In Step 6, checking whether C_b is in A takes O(1) time. In summary, each step takes at most O(m) or O(n) time, so the whole algorithm takes O(m + n) time. □