Columbia University
W4231: Analysis of Algorithms
Professor Luca Trevisan

Handout 15
December 14 1999

# Problem Set 3 Solutions

## Problem 1. Amortized analysis

Recall the problem of incrementing a counter (CLR p. 358). Using amortized analysis it can be shown that the worst-case time for a sequence of $n$ INCREMENT operations on an initially zero counter is $O(n)$ and thus the amortized cost of each operation is $O(1)$.

Now suppose we wish not only to increment a counter but also to reset it to zero (i.e. make all bits in it 0). Show how to implement a counter as a bit vector so that any sequence of $n$ INCREMENT and RESET operations takes time $O(n)$ on an initially zero counter.
(*Hint*: Keep a pointer to the high order 1).

**Solution**

The basic idea is to use an increment algorithm that is very similar to the one described in CLR: the only modification is that we mantain a pointer (a global variable, that we can call $p$ and that is initially 0) to the high order bit. Reset is implemented starting at the highest significant bit and iterating through the vector setting each bit to zero.

*Time complexity*

Using the potential method we define the potential $\phi(n)$ as

$$(\text{the number of 1s in the bynary representation of } n) + n$$

So when doing an insert on a counter whose value is $n$, suppose $k$ is the number of consecutive 1's in least significant positions of $n$, the actual cost for this operation is given by $k+1$. Thus the difference in potential after this insert is $2-k$. This is because 1-k is the cost for changing to 0 all the 1's in the least significant positions of $n$ and adding a new one, and we have to add 1 because $n$ becomes $n+1$.
The total amortized cost for insert is thus 3.

Reset requires time complexity $l = \lceil \log n \rceil$, where $l$ is the position of the position of the most significant bit of $n$.
The difference in potential is given by:
$l$ - number of 1's in $n$ $-n < 0$

## Problem 2. Fibonacci Heaps

The height of $n$-node Binomial Heap is always $O(\log n)$. Show that this is not the case for Fibonacci Heaps by exhibiting, for any positive integer $n$ a sequence of Fibonacci-heap operations that creates a Fibonacci heap consisting of just one tree that is a linear chain of $n$ nodes.

**Solution**

Starting from an empty Fibonacci heap $F$, the basic idea of the algorithm is to create, recursively, a Fibonacci heap that is a linear chain of $n-1$ nodes and then add one more node to the chain.
More precisely, we start creating a Fibonacci Heap of height 1, having root key $m$. Then we add the elements $m-1$ (a value less than the current minimum), $m+1$ (a value larger than the current minimum) and $m-2$ (an even smaller element that has to be deleted to force the consolidation) and delete $m-2$. The consequent consolidate step will generate the required Fibonacci Heap. In the following we supposed $n > 2$ (the case $n \leq 2$ is trivial).

*Pseudocode*

```
Linear-heap($F$,n, m)
    Linear-heap($F$,n-1, m+1)
    Insert($F$,min($F$)+1)
    Insert($F$,min($F$)-1)
    Insert($F$,min($F$)-2)
    Deletemin($F$)
    $x$=min($F$).secondchild
    Decreasekey($x$,min($F$)-2)
    Deletemin($F$)
    return
```

*Correctness*

The proof is by induction. The thesis is clearly correct for $n = 1$. Assume that the statement is true for $n = k$. For $n = k + 1$ the algorithm first creates a linear chain of $k$ nodes (by our inductive hypotesis we know that our algorithm can create such a tree), then it adds two new elements: $x$ that is less than the minimum, and $w$ that is greater than the minimum. Finally it adds an element $m$ that is guaranteed to be smaller than the minimum. When $m$ is removed there remain the chain of $k$ nodes $x$ and $w$. Note that $x$ and $w$ have both degree 0, so they are consolidated. We obtain a chain with two elements where $x$ is the root, having degree 1. Thus the chain of height two and the chain of height $k$ are consolidated. In this way, deleting $w$ we obtain the required chain.

## Problem 3. Graphs

Given a graph $G = (V, E)$ with $|V| = n$ (you can assume $V = \{1, \dots, n\}$)

**(a)** Present an algorithm that given $S_1, S_2 \subseteq V$, outputs $S_1 \cap S_2$ in time $O(n)$

**(b)** Present an algorithm that receiving as input the graph $G$ outputs a 4-cicle (if any exists) in time $O(n^3)$

**Solution**

(a) The basic idea is to sort $S_1$ and $S_2$ using counting sort and then use the algorithm from homework 2, to produce the intersection of $S_1 \cap S_2$ in linear time. Note that $S_1$ and $S_2$ can be sorted in linear time using counting sort. This is because the elements of $V$ are integers in the set $\{1, \dots n\}$

(b) We suppose that the graph is represented using an adjacency list. The strategy is, then, for each possible pair of vertices to determine if their adjacency lists have at least two elements in common. Indeed, in this case, the two vertices are part of a 4-cycle and that cycle can be returned.

```
Find-4Cycle(G,n)
for $i = 1$ to $n$
    for $j = 2$ to $n$
        if $(i \neq j)$ then
            $Int$ =intersection($Adj[i], Adj[j]$)
            if length($Int \geq 2$) then return($i, Int[0], Int[1], j$)
return "The graph contains no 4 cycles".
```

The correctnes of the above algorithm is clear. Its running time is given by $O(n^3)$