Homework #5
Algorithms I
Student: Arpan Ghosh
600.463
Spring 2017

Due on: Saturday, March 18th, 11:59pm
Late submissions: will NOT be accepted
Format: Please start each problem on a new page.
Where to submit: On Gradescope, under HW5
Please type your answers; handwritten assignments will not be accepted.
To get full credit, your answers must be explained clearly,
with enough details and rigorous proofs.

March 24, 2017

Problem 1 (20 point)

Let $G = G(V, E)$ be a directed graph represented by an adjacency list. G is a bipartite graph if it is possible to partition the vertices of G into two disjoint sets, i.e. $V = V_1 \cup V_2$ and $V_1 \cap V_2 = \emptyset$ such that there are no edges between vertices in the V_1 and there are no edges between vertices in the V_2. Design an efficient algorithm that works in $O(|E| + |V|)$ time and checks if G is bipartite. Prove the correctness of your algorithm and analyze the running time.

Problem 1 Solution

Analysis:
In order to do this in linear time where we check the edges and vertices, the best way to do so is using a Depth for Search to figure out whether a graph is bipartite and by coloring the graph in one of two colors: for this solution let's choose red and black. The following algorithm is based off implementing a stack that will conduct operations in constant time to keep track of the colors. **Proof:** The bipartite check of our function operates such that when the base case is satisfied with V containing only one element, then the graph will be bipartite as there will be no adjacent vertices. In the inductive step we know that every edge will be checked only once and we can check adjacent vertices of the edge that is being checked. If the graph is bipartite, then our edge will be colored accordingly, if it is not capable of being colored by the below algorithm then we know the graph is not bipartite. The while loop will run the amount of times equal to the number of vertices. This is because each vertex is pushed to the stack, and such the loop will check each vertex. Moreover, as the vertices are checked, the edges are checked as well but will only be checked once in each iteration and not doubly checked as new vertices will contain different adjacent edges and such the total runtime will be O($|V| + |E|$).

Bipartite-Check(G)
Input: $G = G(V, E)$, a directed graph represented by an adjacency list.
Output: Returns true if bipartite, false if not

The following are in this function: stackVertices (a stack that contains the vertices), visit() a method that checks each vertex, newColor() that will color a certain vertex and edge a certain color depending on its place.

```
for each v ∈ V
    visit(v) = false
    newColor = red
while (visit(v) = false for all ∈ V)
    visit(v) = true
    newColor(v) = white
    stackVertices = [v] //contains v's
        while (stackVertices != null)
            e = pop(stackVertices)
            for each (e, v) ∈ E
                if visit(v) = false
                    visit[v] = true
```

```
                push(stackVertices,v)
            if newColor(v) = red
                if newColor(e) = blue
                    newColor(v) = white
                if newColor(e) = white
                    newColor(v) = blue
                        else if newColor(v) = white
                            if newColor(e) != blue
                                return false
                            else if newColor(v) = blue
                                if newColor(e) != white
                                    return false
return true
```

Problem 2 (20 points)

Problem 2.1 (10 points)

Suppose we wish not only to increment a counter 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 a counter as an array of bits 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.)

Problem 2.1 Solution

Analysis and Runtime: An algorithm for this counter is quite easy to implement, as we will be incrementing a binary counter. Or in this case, a counter that also resets all bits to 0. We must only keep a pointer to the high order 1 and as we encounter a bit with value 0 we keep incrementing until we reach a bit that is 1, then we flip it. The cost of each increment operation is linear in the number of bits flipped. We use an array of A(0...k-1) bits where we are implementing a k-bit binary counter and A.length = k. This algorithm is very similar to the one seen in chapter 17 of the book, the only difference being there will be a pointer to the high-order 1.Similarly, the runtime is analyzed pretty well in the book to prove that the INCREMENT and RESET operations take $O(n)$ time on an initially zero counter, the total number of flips in the sequence as the sum from i=0 to k-1 iterations of floor n/2^i is less than 2n, or the sum of i iterations to infinity as 1/2^i (see page 455 for this).

Algorithm: Please consult CLRS page 454 for the 6 line algorithm. Only thing we are adding is a pointer, and a reset option that will reset all bits below the high-order 1 bit (which the pointer is taking care of, so all bits below the pointer to the array position), to zero.

Problem 2.2 (10 points)

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

$INSERT(S, x)$ inserts x into S.

$DELETE\text{-}LARGER\text{-}HALF(S)$ 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 $DELETE\text{-}LARGER\text{-}HALF$ operations runs in $O(m)$ time. Your implementation should also include a way to output the elements of S in $O(|S|)$ time.

Problem 2.2 Solution

Analysis and Runtime: The following algorithm will use an array implementation. The array will hold elements of S and since we don't need to worry about duplicate values we can just push elements into the array at the end and resize the array when necessary (double the size of array when it reaches a certain size). The DELETE-LARGER-HALF(S) will delete the largest elements from S so we can sort the array in nlog(n) time and then find the median of the sorted array and delete the elements from the median until the last element of the array. Overall, we will have a runtime of $O(|S|)$ as deletion will take linear time - as we will be resizing the array to copy the elements we want back into the array.

Algorithm for INSERT:

```
Input: x into Array(S)
S[] = size 10
while (x elements to insert)
i= Array(S).length
    if (S.length > 9)
        new TEMP Array = Array(S).length * 2
            copy each element into new array, same name
    end if
S[i] = x
i++;
```

DELETE-LARGER-HALF:

Input: Array(S)
Output: Array(S) without larger half

MergeSort on Array(S) using mergesort function from class/CLRS

median = median(Array(S)); //median of sorted array

for each element in new sorted array,
    copy elements from S[0] ... S[median] into new Array(TEMP).
return Array(TEMP)