Introduction to Algorithms
Assignment 3

Sourabh S. Shenoy
(UIN:225009050)

September 23, 2016

Contents

1. Question 1 **2**
- 1.1 K-bit Counter . . . 2
- 1.2 Solution . . . 2

2. Question 2 **2**
- 2.1 Stack Problem . . . 2
- 2.2 Solution . . . 2

3. Question 3 **3**
- 3.1 Wrestling Problem . . . 3
- 3.2 Solution . . . 3
  - 3.2.1 Idea: . . . 3
  - 3.2.2 Pseudocode: . . . 4
  - 3.2.3 Proof of Correctness: . . . 5
  - 3.2.4 Time Complexity: . . . 6

1. Question 1

1.1 K-bit Counter

Show that if a DECREMENT operation were included in the k-bit counter example, n operations could cost as much as $\theta(nk)$ time.

1.2 Solution

For n operations on a k-bit counter to cost $\theta(n * k)$ time, we ideally need to flip all k bits in every operation. This can happen when the counter value is $2^k - 1$. For instance, consider a 4-bit counter as shown below. Suppose the counter value is $2^4 - 1 = 15$, then, the operations can be applied as follows:

Initial: 1 1 1 1
Increment: 0 0 0 0
Decrement: 1 1 1 1
Increment: 0 0 0 0
Decrement: 1 1 1 1
Increment: 0 0 0 0
.
.
and so on..

Thus, we can see that k bits are flipped at every steps. Suppose we have n operations, with alternating Increment and Decrement as shown above, then we have a worst case time complexity of $\theta(nk)$

2. Question 2

2.1 Stack Problem

Suppose we perform a sequence of stack operations on a stack whose size never exceeds k. After every k operations, we make a copy of the entire stack for backup purposes. Show that the cost of n stack operations, including copying the stack, is O(n) by assigning suitable amortized costs to the various stack operations.

2.2 Solution

Let us first check the time complexity using **aggregate analysis** method. Our assumption here is that the basic stack operations have costs as mentioned in the table below.

Push	1
Pop	1
Copy (1 element)	1

Suppose we have 'n' operations performed on the stack, the worst case would be if all n operations are Push. This is because having all Push operations will maximize the number of elements to be copied.

In this case, it would cost O(n) time complexity to push the elements and another O(n) time to copy each element to the backup stack. Thus, the total complexity is O(n+n)=O(2n)=O(n), which is linear.

Now, let us perform **amortized analysis** on the same problem. Let us consider the cost of operations as follows:

Push	3
Pop	0
Copy (1 element)	0

Now, suppose we have 'n' operations, all being pushes as in the previous case, then we have an amortized cost of 3n associated with it. This means that, in essence, we have already "paid" in time complexity for copying those 'n' elements. Hence, now the copying does not take any time. The complexity now becomes O(3n+0)=O(3n)=O(n), which is again linear. Also, when we performed amortized analysis, we have obtained 'n' additional cost associated with pop, although we have not performed any pop operation. However, this does not affect the fact that it is still linear in complexity.

3. Question 3

3.1 Wrestling Problem

There are two types of professional wrestlers: "babyfaces" ("good guys") and "heels" ("bad guys"). Between any pair of professional wrestlers, there may or may not be a rivalry. Suppose we have n professional wrestlers and we have a list of r pairs of wrestlers for which there are rivalries. Give an O(n+r) time algorithm that determines whether it is possible to designate some of the wrestlers as babyfaces and the remainder as heels such that each rivalry is between a babyface and a heel. If it is possible to perform such a designation, your algorithm should produce it.

3.2 Solution

3.2.1 Idea:

We can use the concept of graph theory to solve this problem. Essentially, we are looking at whether we can build a bipartite graph. A bipartite graph is one whose vertices can be

divided into 2 disjoint sets, say U and V, such that every edge connects a vertex in U to a vertex in V.[1] Suppose we are able to prove that the graph we built is bipartite, it implies that the edges connecting the two disjoint sets are the rivalries. Also, it will mean that there are no edges between vertices in the same set (No rivalry among babyfaces or among heels).

To identify the bipartiteness, we use 2-Color algorithm[2]. The algorithm works by assigning Color-1 to a random node, Color-2 to it's neighbors, Color-1 to their neighbors etc, until all the nodes are colored. From this, it is obvious that if we have an odd cycle, we can not assign colors without assigning 2 nodes the same color. Hence, if the graph consists only of "even cycles", then the graph can be colored with 2 colors and hence bipartite.

3.2.2 Pseudocode:

We treat each wrestler as a node and the rivalries will be represented by edges.

Step 1: Construct an undirected graph with n vertices and add the 'r' edges by connecting the corresponding wrestlers

Step 2: Initialize all nodes with color-0 (No color)

Step 3: Initialize a Queue and add the first node to the queue and assign Color-1 to it. Now, traverse through all the edges connected to Node 1, while adding them to the queue, and assign them Color-2

Step 4: Once all the edges of node 1 are done, pop out the next node in the queue. It is already colored with color-2. Traverse through all it's neighbors and assign them Color-1 and so on..

Step 5: Suppose we encounter a situation where the color of the current node is same as the color of one of the nodes it is connected to, it means that we have an odd cycle and the graph cannot be colored with 2 colors. Hence, we return "No". If that does not happen, the algorithm will keep coloring till all nodes are colored properly and return "Yes".

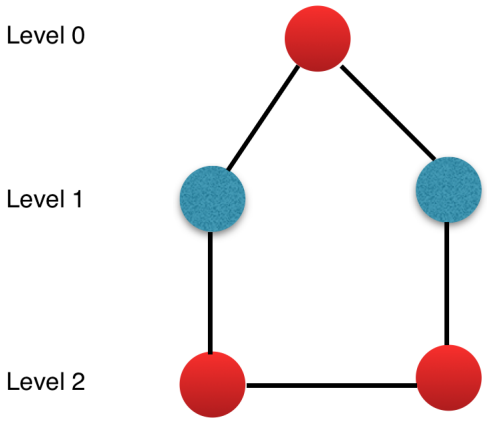
Step 6: Since there would be wrestlers with no rivalry with anyone, there would be disjoint nodes present in the set. Hence, they have to be covered too. Also, we may have multiple disjoint graphs. The process has to be repeated for each disjoint sub-graph and all have to be 2-colorable

Step 7: Return "No" if any of the subgraphs are not 2-colorable

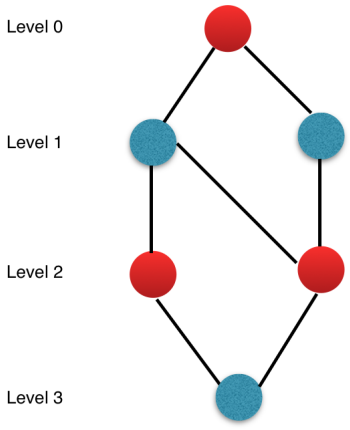
[1] *https://en.wikipedia.org/wiki/Bipartite_graph*
[2] *https://en.wikipedia.org/wiki/Graph_coloring*

3.2.3 Proof of Correctness:

Consider a random node to be at depth 0. The nodes connected to it would be at depth 1 and so on.. We do not have any back edge or forward edges in our undirected graph. Hence, every node at the same level will be assigned the same color. If we have an odd cycle, this results in a cross edge (Edge between nodes of same level), which implies that those two nodes will have the same color, and hence the graph wont be bipartite. An example of graph with cycle length 5 (odd) is shown below:



However, if the cycle length had been 6 (even), then we would not have had the problem and the graph would have been 2-colorable, as shown below.



3.2.4 Time Complexity:

Creating the graph by adding nodes and the edges takes O(|V|+|E|), where |V| is the number of vertices and |E| is the number of edges. Similarly, since we use Breadth First Search with modification to detect the bipartiteness, the time complexity of the main algorithm is also O(|V|+|E|). In our case, we have 'n' vertices, and the number of edges will be 'r', since we have 'r' pairs of rivalries. Hence, the total time complexity of the algorithm is O(2n+2r) or O(n+r).