CSCE 629-601 Homework 3
28th September, 2016

Name: Adil Hamid Malla **UIN:** 425008306

1. Exercise 17.1-2 on page 456

Solution:
Now apart from the INCREMENT functionality we also have a DECREMENT functionality associated with the k-bit counter.
Now the DECREMENT function will decrement the k-bit counter by one value. As in the book we will use an array A$[0, \ldots..k-1]$ to represent the bit counter. Where A.length = k, as the counter. A binary number x that is stored in the counter has its lowest order bit in $A[0]$ and its highest order bit in $A[k-1]$, so that

$$x = \sum_{i=0}^{k-1} A[i] \cdot 2^i$$

The increment algorithm is given below.

Algorithm 1 Increment Functionality

```
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:      end if
10: end procedure
```

Now we have already seen the Increment function of the counter in book. We will implement the Decrement function using the below algorithm. To subtract $1\left(modulo2^k\right)$ to the value in the counter, we will use the following procedure.

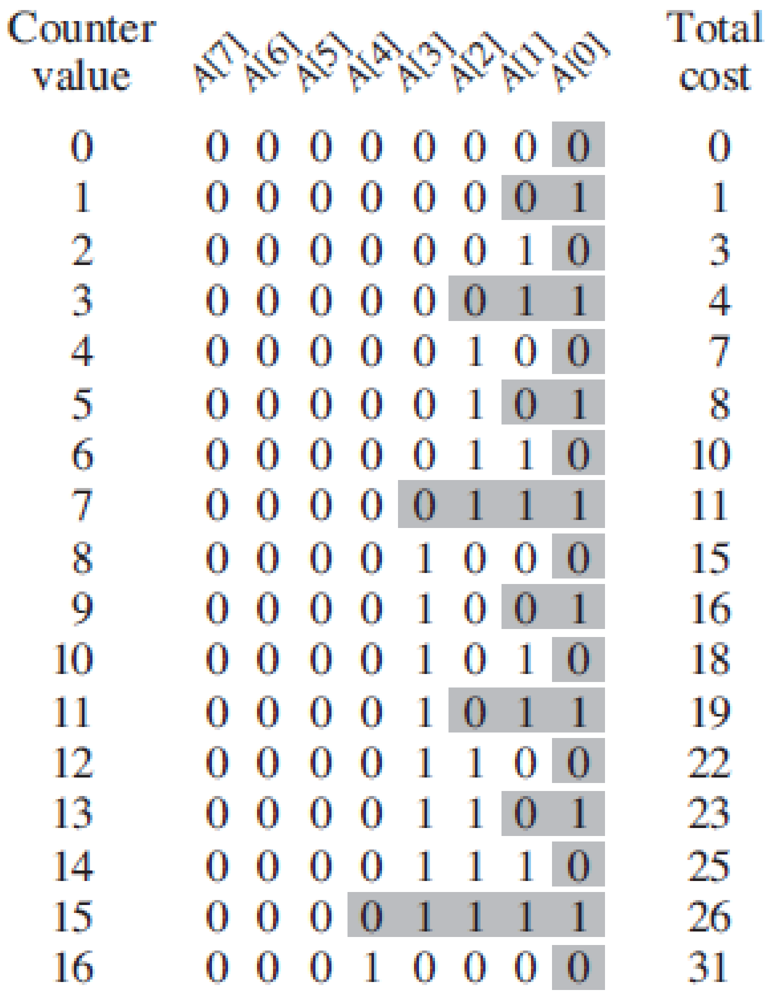
Algorithm 2 Decrement Functionality

```
1:  procedure DECREMENT($A$)
2:      i = 0
3:      while $i < A.length$ and $A[i] == 0$ do
4:          $A[i] = 1$
5:          $i = i + 1$
6:      end while
7:      if $i < A.length$ then
8:          $A[i] = 0$
9:      end if
10: end procedure
```

Now we see here that the Decrement function traverse all the k bits and inverts the bits. But if see the structure of the bits getting changed it follows same as that of increment i.e first column bit changes every time, and the next changes after every two entries and so. The general concept of the bits it changes are shown below

$$A[i] \text{ Flips } = \frac{n}{2^i} \text{ times}$$

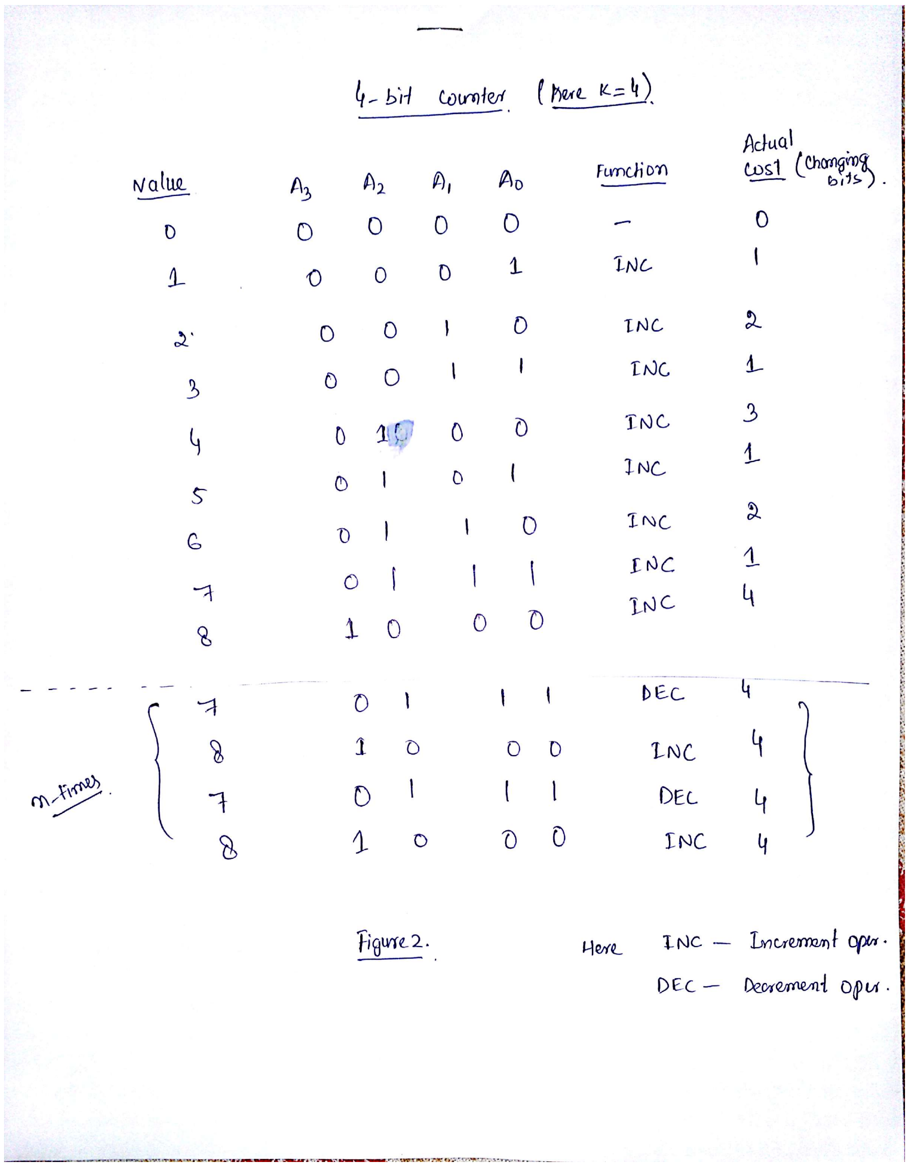
So while incrementing you will see the pattern of bits being actually changed and if you go reverse then also we can see actual number of bits being changed.
But for the amortized analysis we have to check the worst case scenarios. What if don't follow proper incrementing till the limit and then decrement, what i mean is what if we increment and then next operation we follow is decrement. This will result in changing all the bits in k-bit counter.
Generally speaking, if we have n operation we can do on this k-bit counter. We can either do the n/2 increments first and then n/2 decrements. but this will give us somewhat $O(n)$ result.



Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Figure 1: K-Bit Counter- Taken from Book

But when we do the increment and decrement of the k bit counter repeatedly then we have a problem as everytime it will increment the counter and change the number of bits and then decrement will do the opposite of changing the same number of bits, thus it won't follow the general rule of when we are only incrementing or decrementing as in Figure 1. As in the figure given below Figure 2, we have taken a 4 bit counter, when the counter reaches the value of 8, after that if we repeat the n operation like increment and decrement, then we will always change the k bits, hence each operation of increment will take $\theta(k)$ and n operations will take $\theta(nk)$ .Thus time complexity of this increment and decrement functionality will be $\theta(nk)$ times.


4-bit counter (Here K=4)

	Value	A_3	A_2	A_1	A_0	Function	Actual cost (changing bits).
	0	0	0	0	0	–	0
	1	0	0	0	1	INC	1
	2	0	0	1	0	INC	2
	3	0	0	1	1	INC	1
	4	0	1	0	0	INC	3
	5	0	1	0	1	INC	1
	6	0	1	1	0	INC	2
	7	0	1	1	1	INC	1
	8	1	0	0	0	INC	4
	7	0	1	1	1	DEC	4
n-times.	8	1	0	0	0	INC	4
	7	0	1	1	1	DEC	4
	8	1	0	0	0	INC	4

Figure 2.

Here INC — Increment opr.
DEC — Decrement opr.

Figure 2: When we have INC and DEC operations alternatively

1. Exercise 17.2-1 on page 458

Solution:
Here, the question asks as to using the accounting method to detect the amortized cost for the n operations on stack, where the operations can be PUSH, POP, MULTIPOP and BACKUPSTACK.
Since, to backup the stack each element needs to be read and saved on some other format, so backing up the stack is like reading each element and save it so the cost associated is 1. So if we have n elements in the stack that means we will take n read operations to get the data to backup. Consider it like a reading but here it takes all the elements from the stack and we can further POP or use MULTIPOP to pop the elements.
Now lets take the amortized cost for all the stack operations.

Operation	Actual Cost	Amortized Cost
PUSH	1	3
POP	1	0
MULTIPOP	min(k,s)	0
BACKUPSTACK	s	0

Here k is integer passed to pop number of items from stack and s is the size of the stack at a given point.
Now for amortized costs are given to our operations, we have to make sure that in any sequence of n operations we have to make sure that the amortized costs are actually the higher bound to the actual cost of the operation. We can see now that we can pay any sequence of the stack operations by charging the amortized costs.

1. If we are doing the PUSH operation say n times, lets assume each cost unit is equal to \$1 then we are paying \$3 upfront for the PUSH operation, means it is greater than the actual cost and rest of the amount goes to credit. Now lets see what happens if we do any other operations, only thing we have to ensure is that our credit should not go negative.

2. If we do n POP operations, then each operation's actual cost is \$1 per operation, so for every pushed item we are poping then we are not violating the credit score to be negative. Now if we do MULTIPOP or BACKUPSTACK, since there are no items in stack so no operation will be done.

3. If we do MULTIPOP(k, s) operation on the stack then also we won't charge anything as we already have paid for the every pushed item upfront.

4. Finally for the case of BACKUPSTACK say after k operations we are left with n-k elements in the stack and we want to backup the data for further reference. Now since during the pushing of each element we have already paid \$2 extra amount per push. And after k operations of POP and MULTIPOP we have n-k elements left in the stack, so we by default have $\$2 * (n - k)$ left in the credit to pay for backing up the data using the BACKUPSTACK function and other n-k operations that may be POP or MULTIPOP. So by making the amortized cost of 0 to BACKUPSTACK, we have actually made sure that we never have negative credit score.

Thus, for any sequence of the n PUSH, POP, MULTIPOP and BACKUPSTACK operations, the total amortized cost is the upper bound on the total actual cost. So the total amortized cost is 3*n which is $O(n)$.

1. Exercise 22.2-7 on page 602

Solution:
This problem is quite tricky. It wants to judge the notion of our understanding of the graphs and traversal and how to apply them. Since we need two class of people in our graphs i.e. baby faces and heels. So we can say that we have to see of the graph can be divided into two sets where one is represented as Babyfaces and other Heels. So it is basically the graph bipartite problem. There are many ways to check the bipartite graph. The one i am using here is called 2-coloring of graph. We will try to color all the nodes either Blue aka BabyFaces or Red aka Heels. If such a distinction is possible then we can say that we have a rivalry between the Babyfaces and Heels. If no such distinction is possible then we can say it is not possible to say that rivalry is between two kind of the people.

INPUT: We have n wrestlers , which we will consider as nodes, and also we have r pair of wrestlers who have rivalry between them since it needs two guys to have rivalry, then r pair acts as an edge between two nodes i.e. wrestlers here.Since we rivalry between say u and v then rivalry is not one sided, its from both the sides, that is $u \to v$ and $v \to u$. So basically we are saying the rivalry is an undirected edge between two nodes u and v. Now we can use the adjaceny list to store the Nodes i.e. n wrestlers and their rivalry nodes i.e. edge from one node to other. Lets represent the graph by $G(n, r)$

OUTPUT: The output of this problem is whether nodes can be separated in to two sets where there is no edge between the nodes of same section, then we can say the edges only appears between the sections, here edge represents the rivalry. And also we will output the two groups. If its not possible then we will say no such partition possible.
Algorithm: The Algorithm is quite easy and simple. We do the BFS traversal of the graph, we start with the one node and give the node a color, say BLUE here and then when we discover all the neighbors, and color all of them as RED, and then their neighbors as BLUE and so on. If we are able to give the distinct color at each level without coming around already visited node with opposite color, then we can say we have a rivalry between BLUE aka BabyFaces and RED aka Heels. If we get one node while traversing which has opposite color as compared to which it is suppose to be then we say we can't make a clear distinction into two groups. If we get a node which has the same color as the parent node in BFS tree that mean we are not sure whether to put that node in babyfaces or heels, so such distinction is not possible.
Lets see the algorithm for enhanced BFS with solution to this problem by coloring. Here we will be using few value holders like LIST of BLUE and RED nodes. whether a node is un-discovered, discovered or visited by keeping the present condition of the node saved.
If our algorithm returns the true that means we were successful in getting the two sections of the nodes and thus we were able to make the rivalry exist between two sets of people that is Babyface wrestlers and Heel guys.
The proof of correctness of this algorithm lies in the idea itself that if we are trying to give a blue color to a guy who is already red, or vice versa that means it a conflicting node which cannot be put on either side of the group rivalry, thus we end of saying such rivalry doesn't exist between two groups.

Algorithm 3 Graph Two-Coloring Problem

```
procedure TWOCOLORBFS($G, s$)
    for $\forall u \in G.n$ except s do
        $u.state = undiscovered$
        $u.color = white$                    ▷ all nodes will have basically same color
    end for
    $s.state = discovered$
    $s.color = BLUE$
    Let Q be a queue to store the values and initially $Q = \phi$
     Enqueue($Q, s$)
    Let $BFBLUE$ [...] be list of wrestlers being baby face and $HLRED$ [..] be list of wrestlers being heels
    add $s \rightarrow BFBLUE$
    while $Q! = \phi$ do
        $u = DEQUEUE\left(Q\right)$
        for $v \in G.adj\left(u\right)$ do
            if $v.state == undiscovered$ then
                $v.state = discovered$
                $v.color = Complement\left(u.color\right)$                    ▷
complement of color means opposite of color, that is if u is blue then red and vice versa
                if $v.color == BLUE$ then
                    add $v \rightarrow BFBLUE$
                else
                    add $v \rightarrow HLRED$
                end if
            else
                if $v.state == discovered$ or $visited$ and $v.color == u.color$ then
                    return "No such Distinction Possible"
                end if
            end if
        end for
        $u.state = visited$
    end while
    return true
end procedure
```

The complexity of this algorithm will be same as that of the BFS algorithm where the graph is represented by adjacency list. The time complexity of the algorithm for BFS is $O\left(V + E\right)$ where V is number of vertices and E is number of edges. Total time devoted to queue operations is $O\left(V\right)$ and adjaceny list is scanned equal to length of the list in total i.e. $\theta\left(E\right)$ and time spent on scanning adjaceny list is $O\left(E\right)$. Here also the algorithm will have a time complexity of $O\left(n + r\right)$ where we have n as number of nodes/wrestlers and r as number of edges/rivalry as we are doing the BFS of the graph.