UPLOAD THE SOLUTION OF Q1 AND Q2 TOGETHER AS A SINGLE FILE.

Q1 [20M]. Consider the following three functions and answer the following:

- **a)** Write the time complexity of function foo() and justify in one statement.
- **b)** **Derive** the time complexity for function foo1().
- **c)** Write recurrence equation for the function foo2() and solve it using recursion tree method. Make a complete tree showing atleast 3 levels, and all the subsequent steps.

Q1(a)

```
void foo (int N) {
   int x=1, y;
   while (x <= N)
   {
      for (y=1 ; y<N ; y++)
      {
         printf ("Hello World!\n");
         break;
      }
      x = x + 1;
   } }
```

Q1(b)

```
void foo1 (int N)
{
   int i, j;
   for (i=1 ; i<=N ; i++)
   {
     for (j=1 ; j<i ; j*=2)
        printf ("Hello World! ");
   }
}
```

Q1(c)

```
void foo2(int N) {
   int j, k, x=0, temp=2;
   if (N == 2)
      return;
   for (j=1; j<=N-1; j++)
   {
      for (k=1; k<=temp;k++)
       x++;
      temp*=2;
   }
   foo2(N-1);
   foo2(N-1);
}
```

Sol1(a) [2Marks]: O(N) → No partial marks.

Sol 1(b) [6 Marks]: Outer loop runs N times; and for each value of N, inner loop runs O (lg N) times. Therefore, time complexity is $\sum_{i=1}^{N} lg\ N$ = lg(1*2*3*.....*N) = lg (N!) = O(N lg N).

Here you were supposed to show all the steps. If final time complexity is correct (but intermediate steps were not correct or were wrong) then 2 Marks were given.

Sol 1(c) [12 Marks]: Recurrence equation is $T(N) = 2T(N-1) + 2^N$

Size of sub-problem at depth i = (N-i).

Depth at which size of sub-problem becomes 2 is: (N-2). This is the height of the tree.

Cost of each level (except the last level) = 2^N

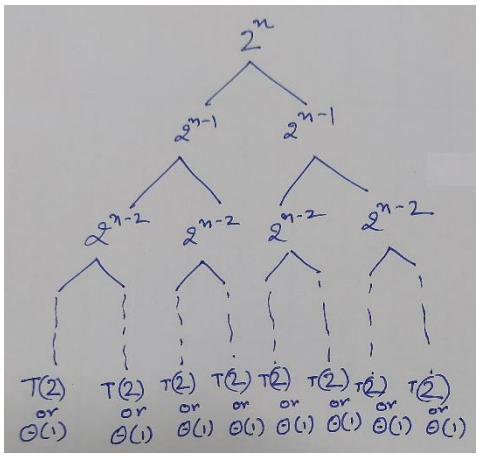
Cost of last level = number of leaf nodes x number of nodes

$= 2^{N-2} \times O(1)$

Therefore, total cost = $(N-2)2^N + 2^{N-2} O(1)$

$< N2^N$

Therefore, time complexity = $O(N2^N)$



Marking Scheme:

0/2 Marks for recurrence equation.

0/4 Marks for "complete" recursion tree. If the tree was incomplete (i.e. all levels including the leaves should be made), zero marks were given

0/4 Marks for total cost equation.

0/2 Marks for final time complexity.

Q2(a) [4M] Experiment reveals that when N integers were given to insertion sort algorithm, it performed 3000 comparisons. If we now give 2N integers to the same insertion sort algorithm, approximately how many comparisons will be done. Justify your answer.

Sol: Number of comparisons is approx. 12000. [1 Mark for final answer; 3Marks for justification]

Q2(b) [6M] Consider that a stack is implemented using a Linked List. It is claimed that the following three stack operations can be performed in O(1) time.

a) Push (Stack S, element E): Normal push operation
b) Pop (Stack S): Normal pop operation, i.e. delete the top-of-stack element
c) Pop_Min (Stack S): Delete the minimum element from the stack.

Is the claim correct? Justify your answer. As a justification, you have to prove your statements. That is, if you say that "something" is possible (or not possible), then prove it formally.

[**Note**: *You can use theorems done in the class for your justification and need not prove them.*]

Sol: No, the claim is not correct. If it would have been correct, sorting would happen in O(N) time.

Many of you have written that "the time complexity of pop_min() operation cannot be O(1). Rather, it has to be O(n)". This is a conclusive statement; and not a proof. So, such answers were awarded zero marks. No partial marks were given.

UPLOAD THE SOLUTION OF Q3 AND Q4 TOGETHER AS A SINGLE FILE.

Q3 [20M]. With respect to the quicksort algorithm done in class, answer the following:

a) Assume that quicksort always uses last element as a pivot. To sort the following array in ascending order, what are the possible values from which the last element (shown as X) can be chosen such that most balanced partition is performed by quicksort.

[11 6 4 10 23 25 29 28 X]

Sol: [0/4 Marks] Any value between 12 to 22 (or 11 to 23 or 12 to 23 or 11 to 22) is good.

b) [6M] Using one value of X from the range specified in part (a) above, show the first two rounds of quicksort.

After first partition, the value of X and 23 should be swapped. [0/3M]

After second partition, either the first part would be [6 4 10 11] or second part would be [23 29 28 25]. [0/3M]

c) Consider the following iterative version of quicksort. Considering temp array as a stack, complete the code.

```
void quicksort (int arr[], int N) {
  int x, y, temp[N], top = -1;
  temp[++top] = 0;
  temp[++top] = N-1;
  while (top >= 0) {
    x = temp[top--];
    y = temp[top--];
    int p = partition (arr, y, x); //assume partition algorithm as done in class
    //write code to check if there are elements to the left of pivot.
    //If Yes, push left side to temp.
     if (p - 1 > y) {
       temp[++top] = y;
       temp[++top] = p - 1;   //   0/5 Marks for this block. No further partial marks.
    }
    //Correspondingly check for elements on the right of pivot.
   if (p + 1 < x) {
       temp[++top] = p + 1;
       temp[++top] = x;       //   0/5 Marks for this block. No further partial marks.
    }

  } }
```

Q4 [20M]. Consider a deque abstract data type (ADT) which allows insertions and deletions from either end of the queue. Consider a variant of deque, that is, dequeRear, which allows insertions at either end of the queue but deletions only from the rear. You have to implement dequeRear using two stacks. Write algorithms for the following operations of dequeRear ADT using two stacks:

(a) [3 Marks] enqueueAtRear (): It inserts the element at rear end.

(b) [5.5 Marks] enqueueAtFront (): It inserts the element at front end.

(c) [3 Marks] dequeueFromRear (): deletes the element from rear end.

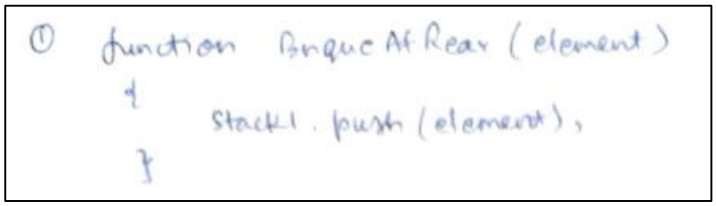
(d) [5.5 Marks] front (): returns the element at front.

(e) [3 Marks] rear (): returns the element at rear.

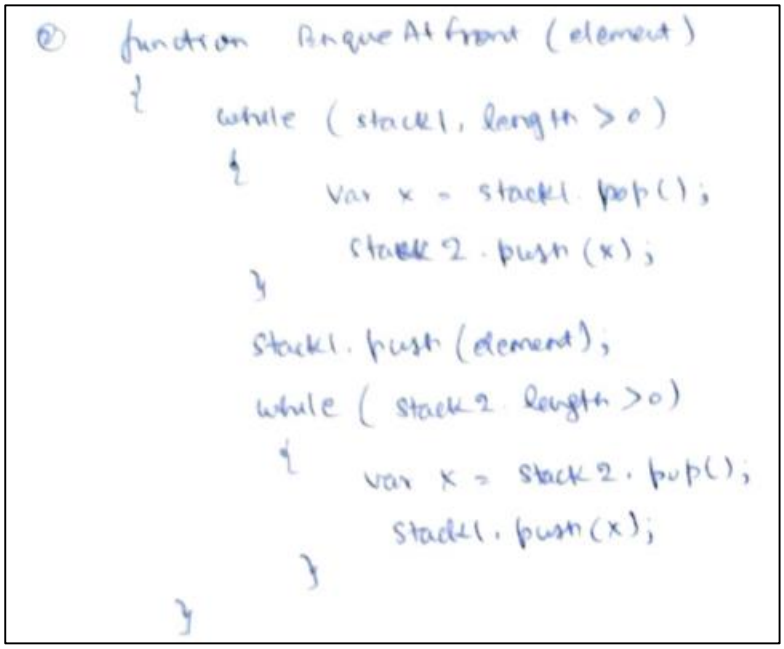
Also, in a tabular form, write the time and space complexity of the above operations. Marks would be given for efficient and neatly written algorithms.

Function	Marks for writing correct algorithm. No partial marks.	Time Complexity	Marks for writing correct Time Complexity	Space complexity	Marks for writing correct Space Complexity
enqueueAtRear	0/1M	O(1)	0/1M	O(1)	0/1M
enqueueAtFront	0/3.5M	O(n)	0/1M	O(n)	0/1M
dequeueFromRear	0/1M	O(1)	0/1M	O(1)	0/1M
front	0/3.5M	O(n)	0/1M	O(n)	0/1M
rear	0/1M	O(1)	0/1M	O(1)	0/1M

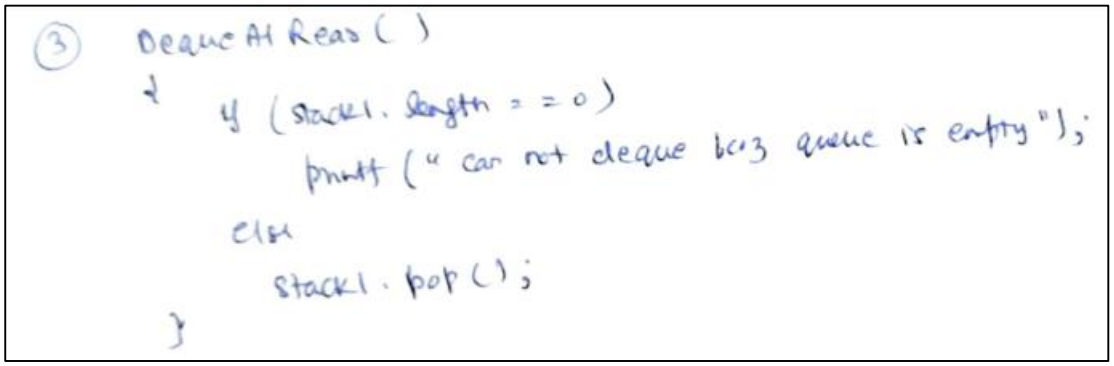
Time and space complexity marks were awarded only if corresponding algorithm is correct.



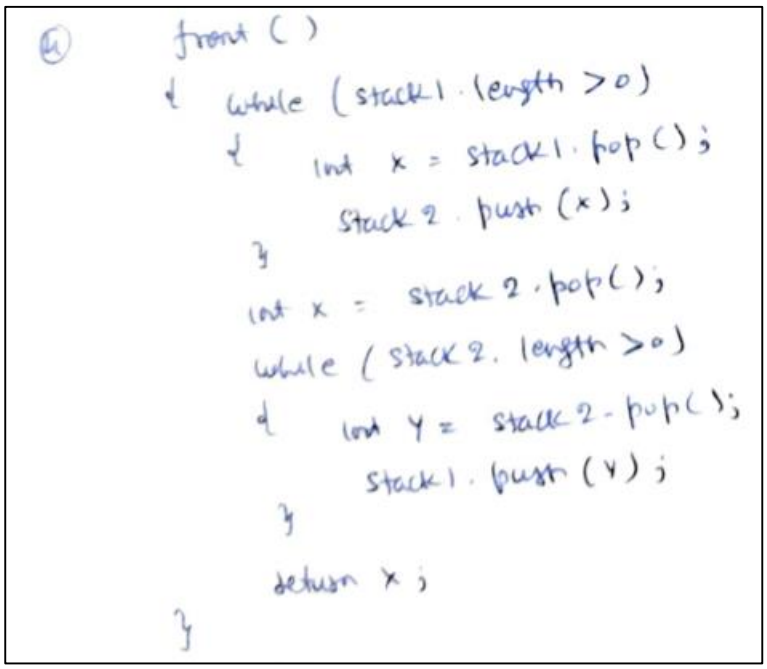
```
① function EnqueAtRear (element)
  {
      Stack1.push(element);
  }
```



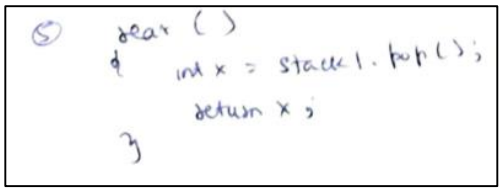
```
② function EnqueAtFront (element)
  {
      while (stack1.length > 0)
      {
          var x = stack1.pop();
          stack2.push(x);
      }
      stack1.push(element);
      while (stack2.length > 0)
      {
          var x = stack2.pop();
          stack1.push(x);
      }
  }
```



```
③ DequeAtRear ( )
  {
      if (stack1.length == 0)
          printf("can not deque bcz queue is empty");
      else
          stack1.pop();
  }
```



```
front ()
{
  while (stack1.length > 0)
  {
    int x = stack1.pop();
    stack2.push(x);
  }
  int x = stack2.pop();
  while (stack2.length > 0)
  {
    int y = stack2.pop();
    stack1.push(y);
  }
  return x;
}
```



```
rear ()
{
  int x = stack1.top();
  return x;
}
```

UPLOAD THE SOLUTION OF Q5 AND Q6 TOGETHER AS A SINGLE FILE.

Q5 [16M]. The function Interchange () takes the address of the first node of a singly linked list and interchange the first and the last node. For example, if the linked list is: 1-> 2 -> 3 -> 4 -> 5, the function modifies the list to be 5 -> 2 -> 3 -> 4 -> 1. Complete the following implementation of the function, ***strictly as instructed in the comments***. Worst case time complexity of the function is O(N), performs the required task in single traversal of the list, and **work by only modifying the pointers**. This means, you are not allowed to modify the data field of any node, create new nodes, use any data structure for temporary storage (like arrays, queues, etc.), and so on.

Write the complete code as your answer (along with required comments).

```
typedef struct node
{
   int data;
   struct node *next;
} NODE;
```

```
NODE* InterChange (NODE *first)
{
   NODE *temp1, *temp2;
   //Do not declare any new pointer variable(s).
   //Write code below considering the list has either zero or one node.
   if (first == NULL || first->next == NULL)
     return first;                                  //0/2M
   temp1 = first;
   //Write code below to point temp1 to second last node.
    while (temp1->next->next != NULL)
     temp1 = temp1->next;                         //  0/4M
   //Using temp2 as an additional variable, write code below to perform
   //the required task of interchanging.
   temp2 = temp1->next;
   temp2->next = first->next;
   temp1->next = first;
   first->next = NULL;
   first = temp2;                          //  0/8 Marks. No partial with this block.
   //Finally, check if the above code also works for 2-element list.
   //Write "YES IT WORK" if it works; "NO, IT WONT WORK" if it doesn't.
   //If your answer is NO, write code below to make it work.
   if (first->next == first)
     first->next = temp1;                          //0/2  Marks
   return first;
}
```

Q6 [7M] Write an efficient algorithm that counts the number of occurrences of each character in a string using two stacks, and analyse its time complexity in the worst and best-case scenarios.

```
Algorithm Count_Occurrences(String S) (3 marks for correctness of algorithm + 1 mark for efficiency)
    {S1 and S2 are two stacks that store objects of type OBJ}
    {Each object of type OBJ has two fields: char and count}
    l ←S.length()   {l is the length of S}
    for i←0 to l-1 do
        while(!(S1.isEmpty()))
            OBJ obj1 ←S1.top()
            S1.pop()
            if obj1.char is the same as S[i] then
                obj1.count ← obj1.count + 1
                S1.push(obj1)
                break;
            else
                S2.push(obj1)
        If(S1.isEmpty())
            obj1 ← get an object of type OBJ
            obj1.char ← S[i]
            obj1.count ← 1
            S1.push(obj1)
        While(!(S2.isEmpty()))
            OBJ obj1 ←S2.top()
            S2.pop()
            S1.push(obj1)
    Output the contents of S1
```

Worst-case time complexity: $O(n^2)$ (n is the length of input string) (1.5 marks)

Best-case time complexity: O(n) (1.5 marks)

(If the algorithm is wrong, then no marks for time complexity)