Answer to the Question no : 1

1. Define a structure for a node in the sorted linked list. Each node should contain an integer value and a pointer to the next node.

```
class SortedLinkedList{
    public:
    class Node
        int value;
        Node* next;
        public:
        Node(int value){
            this->value=value;
            this->next=nullptr;
        }
    };

    int length;
    Node* head;
    Node* tail;
    SortedLinkedList(){
        int length=0;
        this->head=nullptr;
        this->tail=nullptr;
    }
```

Task1 : Insert Sorted Function

```
  void insert_sorted(int value){
    Node* newNode=new Node(value);

    if(head== nullptr || head->value >= newNode->value){
        newNode->next=head;
        head=newNode;
        return;
    }
    Node* temp=head;

    while(temp->next != nullptr){
        if(temp->value < value && temp->next->value > value){
            newNode->next=temp->next;
            temp->next=newNode;
```

```
            return;
        }
        temp=temp->next;
    }
    //Code reaches here meaning the new Node is greater than all other
Nodes
    temp->next=newNode;}
```

Task 2 : Print Sorted Function

```
 void print_sorted(){
        Node* current = head;
        while (current != nullptr) {
            std::cout << current->value << " -> ";
            current = current->next;
        }
        std::cout << "Null pointer" << std::endl;
    }
};
```

Answer to the question no: 2

The running time is the number of times a statement is executed . To calculate it we shall use asymptotic notations for worst case analysis ie BigO notation .

```
template <class itemType>
void sortedType<itemType> ::
    insertItem(itemType item){
        int location=0; ——————————————————O(1)
        bool found=false;--------------------------------O(1)

        while(location<length && !found){ —------ O(n)
            if(item>info[location]){
                location++;
            }else{
                found=true;
            }
        }
        for(int index=lenght;index<length;index--){  —-----O(n)
            info[index]=info[index-1];
            info[location]=item;
            length++;
        }
}
```

Hence the total running time of this code in worst case asymptotic notation would be = O(1) + O(1) + O(n) + O(n)

= O(n)

Answer: O(N)

Main differences between Static and Dynamic Allocation :

Static Allocation	Dynamic Allocation
Memory is allocated in Stack	Heap memory is used
Memory is allocated in runtime , Meaning when the code is running	Memory is allocated in Compile time , Meaning before the code is running
Memory is automatically cleared	Manually clearing of the memory is necessary
Stack memory is less and limited compared to heap memory	Heap memory is greater and more usable than stack memory
Since static allocation used stack , its relatively faster	Since dynamic allocation requires working with heap memory and pointers its relatively slow
Example : Int arr[10]	int * arr = new int[10]

Answer to the question no 3

1) O(n)

Because there is only one loop where the statement declared inside it runs for n-1 number of times . o(n-1) == O(n)

2) $O(n^2)$

The outer loop runs for n number of times and the inner loop runs for n number of times

3) $O(n^3)$

Outer runs for n number of time

Inner runs for n*n which is n square number of times

4) $O(n^2)$

The outer loop runs for n number of times

The inner loop runs for i number of time which is again n number of times . therefore O(n*n)

5) $O(n^5)$

The outer loop runs for i=n-1 time O(n-1) == O(N)

The middle loop runs for i*i times O((n-1)(n-1)) == $O(n^2)$

The inner loop runs for k<j times O((n-1)(n-1)-1)times ==$O(n^2)$

Therefore the complexity is : $O(n^2 * n^2 * n)$ $O(n^5)$

Answer to the question no:4

Q. Implement using stack to check for palindrome

```
#include <bits/stdc++.h>
using namespace std;

bool palindromeChecker(string inp){
   int length=inp.length();
   int mid=length/2;
   //creating a stack to store half of the input
   stack<char> myStack;
   int i;
   for(i=0;i<mid;i++){
      myStack.push(inp[i]);
   }

   //Popping and comparing with rest half of the input
      //increment the i if the length is odd
   if(length%2 != 0){
      i++;
   }
   char temp;
   while(inp[i] != '\0'){
      temp=myStack.top();
      myStack.pop();
      if(temp != inp[i]){
         return false;
      }
      i++;
   }
   cout<<"";
   return true;
}

int main(){
   string input;
   cout<<"ENter the string: "<<endl;
   cin>>input;

   if(palindromeChecker(input)){
      cout<<"Yes its A Palindrome";
   }else {
      cout<<"No its not a palindrome";
   }
```

Answer to the question no : 5

Write the C++ code for evaluating a postfix expression (The algorithm was discussed in Stack and Queue Slide).

```
int postFixFinder(string expression){
   stack<int> myStack;
   for(int i=0;i<expression.length();i++){
       if(isdigit(expression[i])){
           myStack.push(expression[i] - '0');
       }else{
           int value1 = myStack.top();
           myStack.pop();
           int value2=myStack.top();
           myStack.pop();

           if(expression[i]=='+'){
               myStack.push(value1+value2);
           }else if(expression[i]=='-'){
               myStack.push(value2 - value1);
           }else if(expression[i]=='*'){
               myStack.push(value1*value2);
           }else if(expression[i]=='/'){
               myStack.push(value2/value1);
           }
       }
   }
   return myStack.top();
}

int main(){
   string expression="231*+6*";
   cout<<postFixFinder(expression)<<endl;
   return 0;}
```

Answer to the question no : 6

Creating a circular Linked List

```cpp
#include <iostream>

class CircularLinkedList {
private:
    struct Node {
        int data;
        Node* next;
        Node(int value) : data(value), next(nullptr) {}
    };

    Node* tail;

public:
    CircularLinkedList() : tail(nullptr) {}

    // Add a new node at the end
    void insert(int value) {
        Node* newNode = new Node(value);
        if (tail == nullptr) {
            // First node points to itself
            tail = newNode;
            tail->next = tail;
        } else {
            // Insert newNode after tail and update tail
            newNode->next = tail->next; // newNode points to head
            tail->next = newNode;       // old tail points to newNode
            tail = newNode;             // update tail to newNode
        }
    }

    // Delete a node by value
    void remove(int value) {
        if (tail == nullptr) return;  // Empty list

        Node* current = tail->next;
        Node* prev = tail;

        // Special case: deleting the only node in the list
        if (current == tail && current->data == value) {
            delete tail;
            tail = nullptr;
            return;
        }

        // General case: search for the node to delete
```

```
        do {
            if (current->data == value) {
                prev->next = current->next;
                if (current == tail) {
                    // If we're deleting the tail, update tail
                    tail = prev;
                }
                delete current;
                return;
            }
            prev = current;
            current = current->next;
        } while (current != tail->next);
    }

    // Display the list
    void display() const {
        if (tail == nullptr) {
            std::cout << "List is empty\n";
            return;
        }

        Node* current = tail->next; // Start from head
        do {
            std::cout << current->data << " ";
            current = current->next;
        } while (current != tail->next);
        std::cout << std::endl;
    }

    // Check if the list is empty
    bool isEmpty() const {
        return tail == nullptr;
    }

    // Destructor to free up memory
    ~CircularLinkedList() {
        if (tail == nullptr) return;

        Node* current = tail->next; // Start at head
        while (current != tail) {
            Node* nextNode = current->next;
            delete current;
            current = nextNode;
        }
        delete tail;
    }
};                                                  // Mushtasin 2322175642
```

Creating the Main function :

Int main(){

```
a. C->next->info
b. B->previous->info
c. A->next->next
d. C->back->next
e. B->back->back
f. C->back->back
g. A->back
```

}

Answer to the question no 7

// Friend function to overload == operator for stack comparison

Definition of the Function template

```
friend bool operator==(const stackType& s1, const stackType& s2) {
        if (s1.size != s2.size) {
            return false;
        }

        // Traverse and compare each element
        Node* temp1 = s1.topNode;
        Node* temp2 = s2.topNode;

        while (temp1 != nullptr && temp2 != nullptr) {
            if (temp1->data != temp2->data) {
                return false;  // If any data is different, stacks are not
    equal
            }
            temp1 = temp1->next;
            temp2 = temp2->next;
        }

        return true;  // All elements are the same
    }
};
```

Usage:

```
int main(){
    stackType stack1;
    stackType stack2;

    stack1.push(1);
    stack1.push(2);
    stack1.push(3);
```

```
    stack2.push(4);
    stack2.push(5);
    stack2.push(6);

    if(stack1==stack2){
        std::cout<<"They are same";
    }else{
        std::cout<<"Stacks are different";
    }

}                                                    // Mushtasin 2322175642
```

Answer to the Question no 8

Given a queue of integers, write an algorithm that, using only the queue ADT, calculates and prints the sum and the average of the integers in the queue without changing the contents of the queue

```
void calcSumAvg(Queue<int>& myQueue) {
    if (myQueue.isEmpty()) {
        std::cout << "Queue is empty." << std::endl;
        return;
    }

    int sum = 0;
    int count = 0;
    int firstElement = myQueue.front();  // Marker to detect
when we've completed one full cycle
    bool isFirstCycle = true;

    while (isFirstCycle || myQueue.front() != firstElement) {
        isFirstCycle = false; // After the first element is
processed, this becomes false
        int current = myQueue.front();  // Get the front
element
        myQueue.dequeue();                    // Remove it from
the queue
        // Process the element
        sum += current;
        count++;
        myQueue.enqueue(current);
    }

    // Calculate the average
    double average = sum / count;
```

```
    // Print the results
    std::cout << "Sum: " << sum << std::endl;
    std::cout << "Average: " << average << std::endl;
}                                                    // Mushtasin 2322175642
```