BIG-O Complexities:

1. Doubly Linked List Markov:

1) add() method: Firstly, in this method, search function is called on a node to check if it is already there to avoid duplication. If it is already there, the node's vector is updated. If it is not present, then the insert() function is called and after its insertion, its vector is updated. Only type of insertion used in this assignment is insertion at the end which can easily be done by keeping the address of the last node stored in the tails pointer. Therefore, the big-O complexity or the worst case for the run of this method can be broken into the O(n) for the first search operation and then for adding the element in the vector has to be done for each and every element which is done using the push_back() function of the IVector(user defined class) class. This function in turn checks for the size of the array and if it exceeds, then make the n-1 movements after creating an array of double the size. It also makes 2+2(n-1) comparisons everytime it is called.

Search operation: worst time- O(n)
Push_back in the vector: worst time- O(n)
And the insertion operation: worst time- O(1)

This is the add function which is run n-1 times itself making the time complexities

```
void DLL_Markov<T>::add(T t1, T t2)
{
    DLL_Markov_Node<T>* node1 = find_node(t1);// O(n^2)

    if (node1 != NULL) {

        node1->addNew(t2);              //O(n^2)-- in the worst
case, otherwise
    }                                    //takes O(n) most of the
times
    else {

        DLL_Markov_Node<T>* temp = insert(t1);   //O(1) for n-1
nodes making the
        temp->addNew(t2);                        // complexity
O(n)
    }
}
```

```
Therefore, it can be said that the big-O complexity of this method
would be O(n^2)
```

2) generate() method: This method generates the string of a length which is provided by the user. So, the loop to generate the string implements exact no. of times as the length of the string. In this method, firstly a random node is selected and then that node's information is pushed into a vector, then the getRandom()

method is called on this node to find the next node and finally, the find_node method is used to find the given node. The getRandom() method generates a random integer lying in the range of the size of the vector and then uses it to return the element at that index. Therefore, its execution takes only O(1) time.

getRandom: worst time- O(1)
push_back: worst time- O(n)
Search operation: worst time- O(n)

```
void DLL_Markov<T>::generate(int len)  // this is method is called
only once
{
    int iSecret = 0;
    IVector<T> retV;

    srand(time(NULL));

    iSecret = rand() % (size());

    DLL_Markov_Node<T> *tmp = find_node(iSecret);     //search
operation is
    retV.push_back(tmp->info);                        //executed
‘len’ times which
                                                     //is the
length of the string
    for (int i = 0; i < len - 1; i++) {

        T loopy = tmp->getRandom();                 //also both
getRandom and Push
                                                    //functions are
also run len times
        retV.push_back(loopy);                    // if the length is
higher, there
                                                    //would be
significant difference
        tmp = find_node(loopy);                   // for a small number
it wont make
                                                    //much difference
        if (tmp == NULL) tmp = start;
    }
    retV.print_vector();

}
```

Time complexity overall can be O(n) if the length of the string is not much

2. **Multi Linked List Markov:**

1) add() method: Firstly, in this method, search function is called on a node to check if it is already there to avoid duplication. If it is already there, the successor node is checked for duplication. If none are present, then the insert() function is called and after their insertion, the second node is updated in the vector of the first node. Since its insertion was similar to the doubly linked list, it is implemented with a worst time of O(1). Similarily, the search operation is exactly similar to the doubly linked list as well. Therefore, the big-O complexity or the worst case for the run of this method can be broken into the O(n) for the first search operation and then for adding the element in the vector has to be done for each and every element which is done using the push_back() function of the IVector(user defined class) class. The difference from the DLL list lies in the fact that the search operation is run 2 times on both nodes, instead of the 1 time in the DLL implementation. The insertion operation is run 2 times too, but since each node has already been added as a successor before, it does not really make any difference at all.

Search operation: worst time- O(n)
Push_back in the vector: worst time- O(n)
And the insertion operation: worst time- O(1)

Here too the add method is called n-1 times, which ensures the insertion of all nodes, hence there is not much difference in time complexities from doubly linked list and are pretty much the same.

```
void ML_Markov<T>::add(T t1, T t2)
{

    ML_Markov_Node<T>* node1 = find_node(t1);  //O(n^2)

    if (node1 == NULL) {

        node1 = insert(t1);                    //O(1)...since
already inserted as
                                               //the successor
    }

    ML_Markov_Node<T>* node2 = find_node(t2);     //O(n^2)

    if (node2 == NULL) {

        node2 = insert(t2);                    //O(n)

    }

    node1->addNew(node2);  //addNew or pushback function…worst
case- O(n^2)

}
```

Overall time complexity- O(n^2)

2) generate() method: This method generates the string of a length which is provided by the user. So, the loop to generate the string implements exact no. of times as the length of the string. In this method, firstly a random node is selected and then that node's information is pushed into a vector, then the getRandom() method is called on this node to find the next node and finally, the find_node method is used to find the given node. The getRandom() method generates a random integer lying in the range of the size of the vector and then uses it to return the element at that index. Therefore, its execution takes only O(1) time.

getRandom: worst time- O(1)
push_back: worst time- O(n)
Search operation: worst time- O(n)

```
void ML_Markov<T>::generate(int len)
{
    int iSecret = 0;
    IVector<T> retV;

    srand(time(NULL));           // check if the corresponding
vector is empty

    iSecret = rand() % (size());

    ML_Markov_Node<T> *tmp = find_node(iSecret);
    retV.push_back(tmp->info);

    for (int i = 0; i < len - 1; i++) {

        tmp = tmp->getRandom();

        retV.push_back(tmp->info);
        //tmp = find_node(loopy);

        if (tmp->next == NULL) tmp = start;
    }
    retV.print_vector();

}
```

Again not much difference to the big-O complexity of the DLL.

3. **Skip List Markov:**

1) add() method: Insert function is called to insert the node. Since, skip lists are ordered lists, the check for duplication is ensured at the time of insertion itself. If it is

already there, the successor node is added to the vector and the method returns to the caller method. The implementation of this method takes the worst time of O(n) with an average time of O(log(n)), which is the biggest advantage of using skiplist despite its hard implementation. Similarily, the search operation also takes the same worst and average times as the insertion operation. Addition of the element in the vector is done again using the push_back() function of the IVector(user defined class) class. Biggest advantage of skip list over the sequential searching lists like doubly and multi linked lists is that it does search over ordered data by skipping some each time ensuring faster searches over large data-set.

Search operation: worst time- O(n)
Push_back in the vector: worst time- O(n)
And the insertion operation: worst time- O(n)

The add method is called n-1 times making the worst case for the insert operation in the add method O(n^2)

```
void SkipList_Markov<T>::add(T t1, T t2)
{
    probabilise1();

    insert(t1, t2);        //

}
    node1->addNew(node2);  //addNew or pushback function…worst
case- O(n^2)

}
```

Overall time complexity- O(n^2)

2) generate() method: Again this method generates the string of a length which is provided by the user. So, the loop to generate the string implements exact no. of times as the length of the string. In this method, firstly a random node is selected and then that node's information is pushed into a vector, then the getRandom() method is called on this node to find the next node and finally, the find_node method is used to find the given node. The getRandom() method generates a random integer lying in the range of the size of the vector and then uses it to return the element at that index. Then using the element returned, the search operation is performed again to get the next word for the string. Therefore, its execution takes only O(1) time.

getRandom: worst time- O(1)
push_back: worst time- O(n)
Search operation: worst time- O(n)

```
void SkipList_Markov<T>::generate(int len)
{
```

```
    int iSecret = 0;
    IVector<T> retV;

    SkipList_Markov_Node<T> *tmp = root[0];

    //pushing the first random element
    retV.push_back(tmp->info);

    //pushing the elements in the vector of the nodes
    for (int i = 0; i < len - 1; i++) {

        T loopy = tmp->getRandom();


        retV.push_back(loopy);
        tmp = find_node(loopy);

        if (tmp == 0) tmp = root[0];
    }

    //printing final string
    retV.print_vector();

}
```

If the length is increased to a very high number, the time complexity suffers since the loop is run way more times, however it is balanced by the average run time of skip-list search operation which has the time complexity of only O(lgn).