Answer to the question no-01

	Best case	Worst case	Average case
Bubble sort	**O(n)**	**O(n^2)**	**O(n^2)**
Insertion sort	**O(n)**	**O(n^2)**	**O(n^2)**
Merge sort	**O(n*logn)**	**O(n*logn)**	**O(n*logn)**

Answer to the question no-02

Array	Linked list
1. The size of the array has to be declared first.	1. Does not need to declare the size , we will use it until memory space is finished in ram.
2. It is static	2. It is dynamic.

We need a head or root node to set the first value in their position and we can get others' value from that position . If we do not do this , we will lose the first value as well as others' value. It assures that it is the starting point of linked kid. For that we need a head or root node in the linked list.

Answer to the question no-03

The basic idea of binary search is to create a position and compare the given value by increasing or decreasing that position. If the given value is larger than position , it will do a different work, otherwise if it is smaller, it will do another different work. It means that it will do that work as a time complexity O(logn).

It can easily differentiate the linear search to it by measuring time complexity. Another is, we do not need to go to all values. We need to go to a specific range value. But in linear search we have to go through all the values to search.

The requirement for it to be suitable for binary search has to be sorted into either increasing order or non-increasing order.

Answer to the question no-04

The time complexity of inserting an element at the beginning of a singly linked list is **O(1).** We have to create a head and node to hold the data and this work in O(1).

The time complexity of inserting an element at any index of a singly linked list is **O(index-1).** As we have to search that previous position of it from the beginning like linear search.

The time complexity of deleting an element at the beginning of a singly linked list is **O(1)**.

The time complexity of deleting an element at any index of a singly linked list is **O(index-1).** As we have to search for a pre-existing position.

Answer to the question no-05

If I provide a linked list below:

```
class node
{
public:
   int data;
   node *nxt;
};
class LinkedList
{
public:
   node *head;
   LinkedList()
   {
      head=NULL;
   }
   node *CreateNewNode(int value)
   {
      node *newnode=new node;
      newnode->data=value;
      newnode ->nxt=NULL;
   }
   void InsertAtHead(int value)
```

```
{
   node *a=CreateNewNode(value);
   if(head==NULL)
   {
      head=a;
      return;
   }
   else
   {
      a->nxt=head; head=a;
   }
}
```

If we observe the list properly, we can see that we need 2 memory spaces. Firstly in the 'node' class there are 2 variables declared. When we call 'newnode', we get 2 memory spaces. For that for 1 data, we need 2 variables, one is to store data and one is to store the next data location or pointer.

But in an array we do not need to use 2 variables. We need to only declare the size of the array to continue. And the array size is equal to the number of all the elements and the elements are stored in memory in serially. But in the linked list the memory locations are not serialized. For that the memory space is 2 times more than array.

Answer to the question no-06

The worst case of quick sort is O(n^2). The average case of quick sort is O(nlogn).

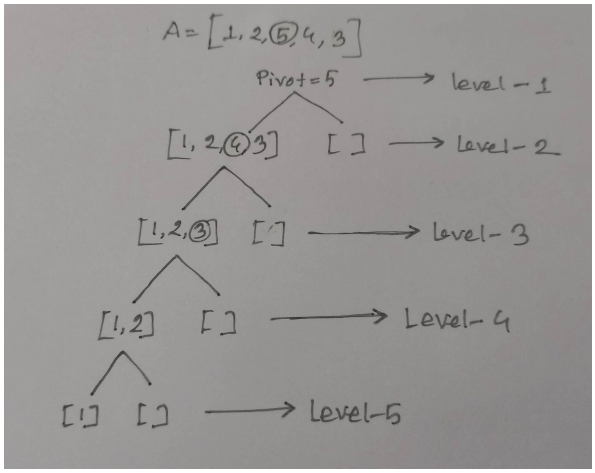
If I provide a example of quick sort here:

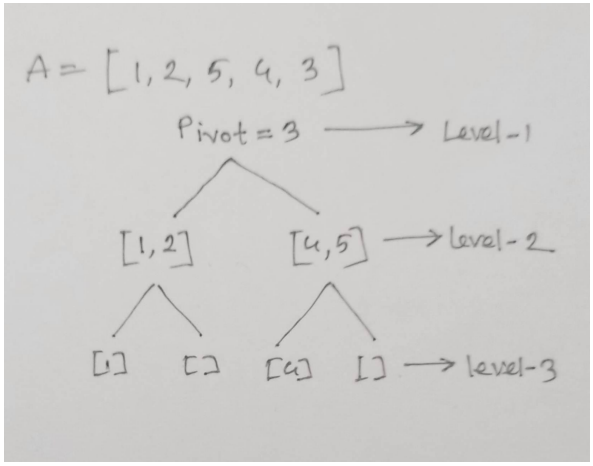
```
vector<int>quick_sort(vector<int>A)
{
   if(A.size()<=1){return A;}
   int P=rand()%(A.size());
   vector<int>B,C;
   for(int i=0;i<A.size();i++)
   {
      if(i==P){continue;}
      if(A[i]<=A[P]){B.push_back(A[i]);}
      else{C.push_back(A[i]);}
   }
   vector<int>Sort_B=quick_sort(B);
```

```
    vector<int>Sort_C=quick_sort(C);
    vector<int>Sort_A;
    for(int i=0;i<Sort_C.size();i++)
    {
        Sort_A.push_back(Sort_C[i]);
    }
    Sort_A.push_back(A[P]);
    for(int i=0;i<Sort_B.size();i++)
    {
        Sort_A.push_back(Sort_B[i]);
    }
    return Sort_A;
}
```

In quick sort we use a pivot to sort by 2 parts. If we make the pivot minimum or maximum value then we have to pass ‘n’ numbers of level and for this level we get the time complexity O(n^2).

But if we choose the middle of the element as a pivot , we can lessen the time complexity. It will be O(nlogn).



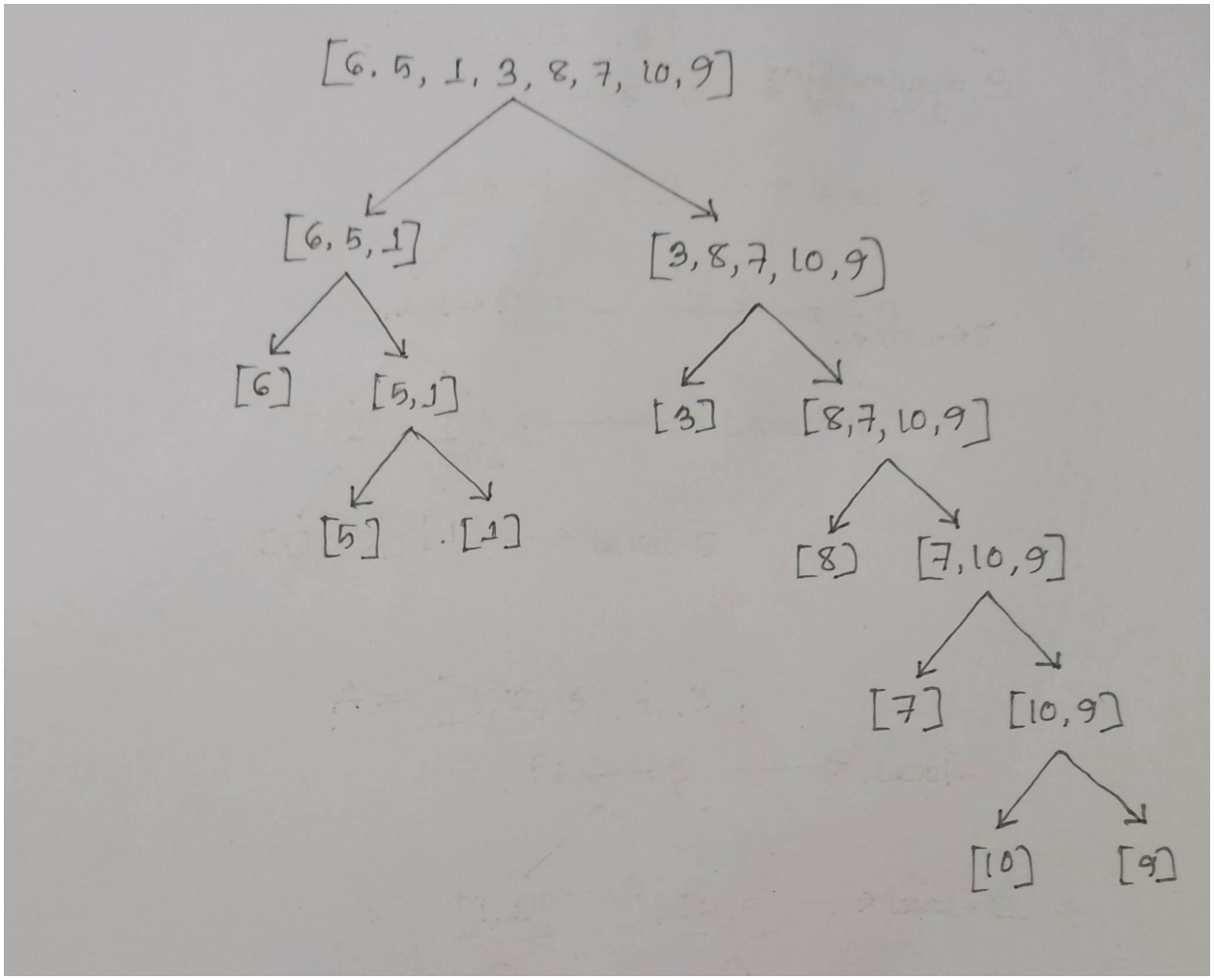


In the first picture, the level is ‘n’. Time complexity is O(n^2).

In the second picture, the level is less than ‘n’. Time complexity is O(nlogn).

Answer to the question no-07

The recursion call tree is below:



Answer to the question no-08

```
for (int i = 1; i * i <= n; i++) {
    if (n % i == 0) {
        cout<< i <<"\n";
        cout<< (n/i) <<"\n";
    }
}
```

In this segment the loop will run if the value of 'n' is a square number. Such as:
If,
n=1,the loop will run 1 time.
n=2,the loop will run 1 time.
n=3,the loop will run 1 time.
n=4,the loop will run 2 times.
n=5,the loop will run 2 times.

n=6,the loop will run 2 times.
n=7,the loop will run 2 times.
n=8,the loop will run 2 times.
n=9,the loop will run 3 times.
Here 1,4,9 are square numbers.
If we root these values, we will get loop run times.
So ,the time complexity of this code segment is **O(√n).**

Answer to the question no-09

Array is more suitable for this case.
In the linked list I can not use binary search. The first condition of binary search is to be sorted, increasing or non-increasing. If I do binary search in the linked list, I have to sort the elements first. But the time complexity will be very bad.
But if i use an array then the time complexity will be O((n+q)logn) for q queries. The value is smaller than doing binary search in the linked list. Also doing linear search in an array.
So, doing a binary search array is more comfortable and easy for me.

Answer to the question no-10

I think a doubly linked list is more suitable for this.
Because, in a singly linked list there is only one way which is moving forward, backward is more complex in this. Here I can do it but it will be more complex as it carries 2 variables in the node.
But in the doubly linked list, I can use 3 variables for moving forward or backward and it can be done easily. One for data, one for the next pointer, one for the previous pointer.

```
class node
{
public:
   int data;
   node *next;
};
```

```
class node
{
public:
   int data;
   node *next;
   node *previous;
};
```

So, a doubly linked list is more suitable for an undo-redo function.