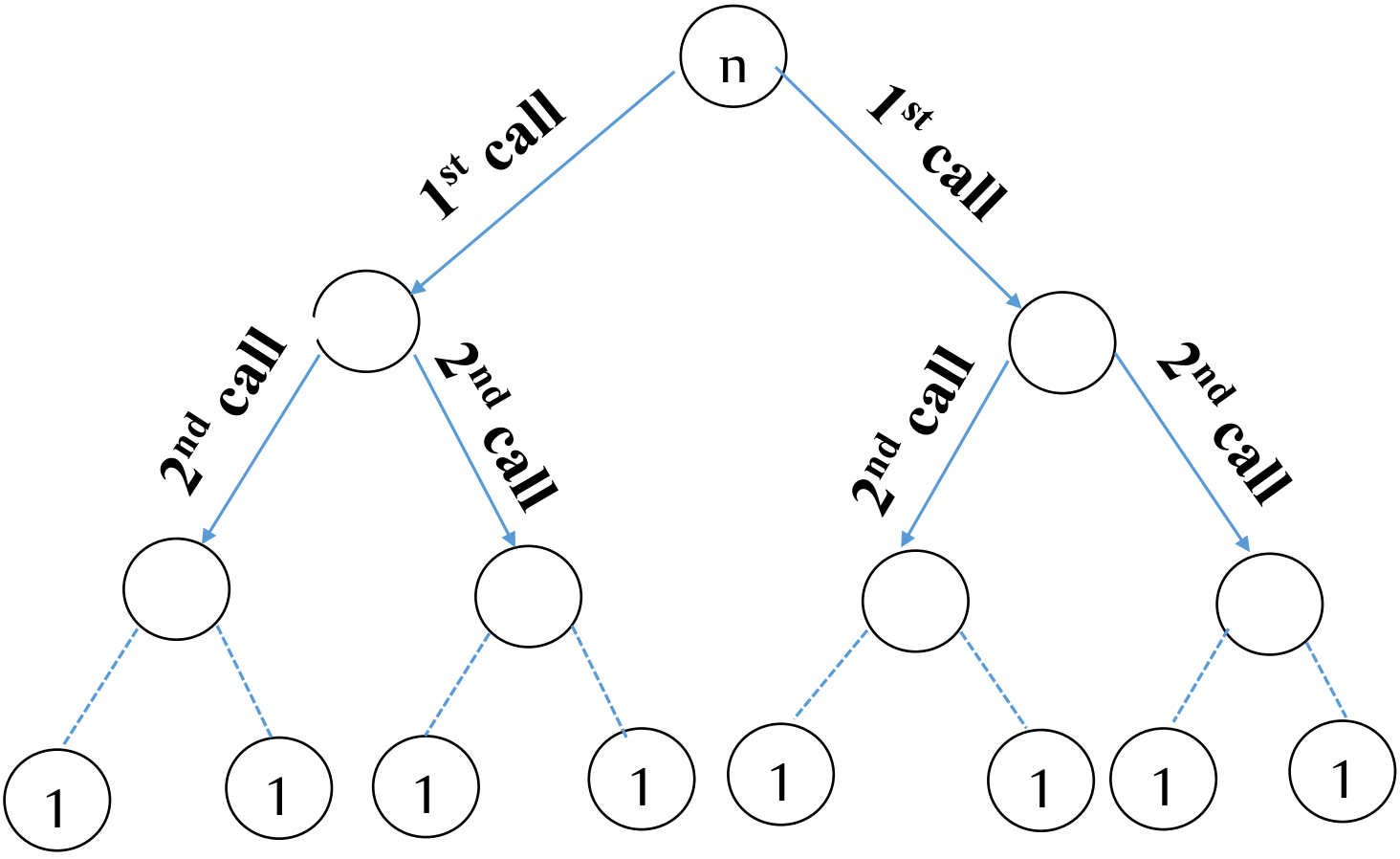
Name : Mohammed Arif Mainuddin
ID : 2211577042
CSE225.2

1.

Here the input size depends on n. Since the code contain recursive call it will run until its meet its base condition. And here the base condition is 1. If the base condition doesn't fulfil the recursive function will call twice. Let's visualize the figure of recursive calling:



Here the dotted lines means more recursive and then faced the base condition. We can see that each time, one recursive call is

generating more two recursive call. If we observe we can see that the total call is $T(n) = 2^1 + 2^2 + \ldots\ldots + 2^{n-1} + 2^n = 2^n$.

So the time complexity is $O(2^n)$.

2.

i) In the first step, we check the middle element of the array, which effectively divides the array into two halves.

ii) In the second step, you either:

a. Find the target element and return its index O(1) time.

b. Recursively search in the left half of the array if arr[mid] > num`. In this case, the left half will be approximately half the size of the original array.

c. Recursively search in the right half of the array if arr[mid] < num. Again, the right half will be approximately half the size of the original array.

iii) You repeat step 2 until we either find the target element or reduce the search space to an empty array.

Now, let's calculate the number of steps needed to reach the base case (an empty array or finding the target element).

In each step, we are effectively halving the size of the search space. So, the number of steps required to reach the base case can be expressed as follows:

$T(n)=n + n/2 + (n/2)/2 + ((n/2)/2)/2 +...$

This is a geometric progression where we keep dividing by 2. The number of steps required until we reach a single element or an empty array is approximately $\log2(n)$.

Therefore, the time complexity of the binary search algorithm is $O(\log n)$ because the number of steps required to complete the search is logarithmic in the size of the input array.

3.

The outer loop runs from j = 1 to j<length-1. This loop iterates through each element of the array till length. If the length is n this loop will run n times.

Total run time of this for loop is 1+2+3+4+…+n.

For the nested while loop, (i > -1 && A[i] > key), where i = j-1,

this loop will run j-1 times and for j=n , this loop will run n-1.

Total run time of while loop is 1+2+3+4+….+ n-1.

We can conclude that the sum of both loop is

$T(n) = 1+2+3+4+\ldots+n-1+n$

$T(n) = n*(n-1)/2 = (n^2 - n)/2 = n^2/2 - n/2$.

Time complexity is $O(n^2)$.

Hence, the time complexity of insertion sort is $O(n^2)$.

4.

Advantages of Unsorted List:

Simplicity: We don't need to iterate the whole list to add, delete in unsorted list.

Fast Accessing: While inserting the elements to the unsorted list, we will know the index. So we can easily access any elements using their index and the time complexity is O(1).

Disadvantages of Unsorted List:

Not good Searching: To find any element we need to iterate through the whole list, which time complexity is O(n).

No specific Order: Since unsorted list doesn't have any specific order, in the scenario where we may need to retrieve the elements in ascending or descending order we can't retrieve any elements.

Binary Search: To apply Binary Search on any linear data structure the structure has to be sorted in a specific order. Since the unsorted list is not sorted, we can't apply Binary Search.

5.

Linear Search:

- In linear search we search through the array until we find the key value or face the end of the list or array. It works on both sorted and unsorted lists.
- Linear search use the Brute Force algorithm.
- The time complexity of linear search is O(n) in the worst case. Where n is the size of list or array.
- Linear search is suitable for small lists or when we don't have any information about the order of elements. It's easy to implement and understand. But for the big size of linear search may not be best of option due to time complexity.

Binary Search:

- Binary search requires the elements to be sorted in ascending or descending order. It starts in the middle

of the array or list and repeatedly eliminates half of the remaining elements based on whether the target element is greater or smaller than the middle element.

- Binary search has a time complexity of O(log n) in the worst case.
- Binary search use divide and conquer algorithm.
- Binary search is highly efficient for large sorted array or lists.

6.

```
void UnsortedType::DeleteItem ( ItemType  item )
//  Pre: item's key has been inititalized.
//     An element in the list has a key that matches item's.
// Post: No element in the list has a key that matches item's.
{
    int  location  =  0 ;

  while (item.ComparedTo (info [location] )  !=  EQUAL )
          location++;

    // move last element into position where item was located
```

```
    info [location] = info [length - 1 ] ;
    length-- ;
}
```

We can see that the while loop will run until item.ComparedTo(info[location]) == EQUAL, this the run time depends on the size of info[].

So the time complexity is O(n) where n is the size.

7.

Linear Data Structure:

In Linear data structures, the data items are arranged in a linear sequence means the elements are connected only in single way respectively. Applying operation on linear structure is not complicated.

Example: Array

Non-Linear Data Structure:

In Non-Linear data structures, the data items are not in sequence means the elements can be connected to each other in multiple way. Applying operation on non-linear data structure we need to traverse by applying depth-first-search or breadth-first-search which is complicated.

8.

A two-dimensional (2D) array is a data structure that can store elements in a grid or matrix-like format, where elements are arranged in rows and columns.

Here's an example of declaring and initializing a 2D array in C++ :

```
#include<bits/stdc++.h>
#define
fast
ios::sync_with_stdio(false),cin.tie(NULL),cout.tie(NULL)
using namespace std;
int main(){
   fast;
int arr[5][5];
for(int i=0; i<5; i++){
  for(int j=0; j<5; j++){
    arr[i][j] = rand() % 100;
  }
 }
}
```

1D Array:

When we need to store a list of elements that have a single linear sequence.

Examples: Storing the city name, storing the name of students.

2D Array:

When we need to organize elements in a grid or matrix-like structure, where each element has both row and column positions.

Examples: Storing data in matrix, storing data in chessboard.