Lecture 1

In this lecture, we will learn about

1. Definition of Data Structure and algorithm
2. Finding the complexity(복잡성)of algorithms
3. Sequential Search and Binary Search

Data Structure means data and structure. In this subject we learn how to **store** (위험·재난·손해 등에서)and **access (**면회, 출입 **)** data. To access data means performing (실행하다, 다하다) operations on data i.e. Insertion, deletion, traversal.
Insertion means to insert (끼워 넣다, 삽입하다, 꽂다) data in a structure.
Deletion means to delete (삭제하다, 지우다) data from a structure.
Traversal (가로지르다, 가로지르고 있다;가로질러 가다, 건너다) means to get data from the structure.

Structure is a holder which hold data. You already used one data structure in C language. **Array** is a **data structure** which can hold data of same type. We can perform insert, delete and traverse operation on data structure and we will do this later in the course.

Before starting with data structure, there are few mathematical concepts (구상, 발상) which we should understand. It is about **analysis (**분석, 해석;분해**) of algorithms**. We analyze a algorithm by finding its **complexity**. An algorithm is a step by step solution to a problem. Let us try to understand analysis or complexity of an algorithm through an example.

Here is a simple code to print the elements of the array.

```
//reading elements into array of size n
int a[10]; // complexity is 0

int n=5;   // complexity is 1

for(i=0;i<n;i+)
        scanf("%d",&a[i]); // This loop execute n times O(n)

for(i=0;i<n;i++)
        printf("%d",a[i]); // This loop execute n times O(n)
```

To save time and space, I missed writing the main(). This algorithm takes the input from the user, stores input in an array and then output or display all the elements of the array.

When we do analysis of above algorithm, we try to give our answer as a function of number of elements or **n**. There are 2 loops in the above program.

If the value of n is 5, 1st loop will execute (실행[수행, 달성]하다) 5 times and second loop will also execute 5 times. i.e. Total number of executions are 10.

If the value of n is 6, 1st loop will execute 6 times and similarly second loop will also execute 6 times. i.e. Total number of executions are 12.

If the value of n is m, 1st loop will execute m times and second loop will execute m times i.e. Total number of executions are 2m

If the value of n is n, 1st loop will execute n times, and second loop will execute n times, i.e. Total number of executions are 2n.

So, complexity of this algorithm is directly proportional (α)(비례하는) to 2n or O(2n) or **O(n)**.
O(n) is read as order of n.
Let us take an another example.
This example search an element in a array.

```
void main(){
int a[10];          // complexity is 0

int i,e;            // complexity is 0

boolean flag=false;     // complexity is 1

for(i=0;i<10;i++)
        scanf("%d",&a[i]); // complexity is n O(n)

printf("enter the element you want to search");        // complexity is 0
scanf("%d",&e);         // complexity is 0

for(i=0;i<10;i++)
{
        if(a[i] ==  e)
        {                                               // complexity is ???????
                flag=true;
                break;
        }
}

if (flag == false)
        printf("element not found");
else                                            // complexity is 0
        pritnf("element found");
}
```

Do you see the ????? for the second for loop. What is the complexity of that for loop. It is bit difficult to find the complexity of that loop. Why?
Because we are searching an element in an array and if we are lucky (운**(chance);**운명, 천명, 천운)we can find the match(결혼 상대) at the first position (위치;장소, 곳;소재지). But if we are unlucky, we may find the match at the last position Or we may find the match some where in between.

Or We may not even find the match

So there are 4 cases.
We may not find a match.
We may find a match at the first place.
We may find a match at the last place.
We may find a match at the middle.

To make our answer easier, we divide our problem into three parts now.

First is best case.
Best case is if we are able to find the element in the first place. So in this case, How many time does the 2nd for loop execute? Only 1 time, right?
So complexity is O(1) i.e. constant (불변의, 일정한).

Second is worst case.
Worse case is if element we are searching is not in the list or is present at the last place.
In both the cases, for loop will traverse through all the n elements in the array until it reaches the last element in the array. So, for loop executes n number of times. Hence complexity is O(n).

Third and last case is Average case.
Average case means element is present in the middle of the array. If the size of array is n, then middle is either n/2 or (n+1)/2.
That means our loop execute at-most n/2 or (n+1)/2.

Sequential Search	***Best Case***	***Worst Case***	***Average Case***
First for loop	O(n)	O(n)	O(n)
Second for loop	O(1)	O(n)	O(n/2)
Complexity of Algorithm	O(1)	O(n)	O(n)

Let us take another example.
The objective or purpose of this algorithm is same as the previous algorithm. This algorithm searches an element in an array. But the logic is different.

Let me first explain the logic. If I ask you to search for a word “hello” in the dictionary. How will you search it? Will you start from the first page? Will you compare your word “hello” with the first word on first page in the dictionary, then second word, then third word and so on.
This is what we did in the last searching. This kind of search is also called as **linear search or sequential search**. Obviously if you are searching in a small array of size 10 or 20 or 100, sequential search is acceptable but if the size of array is 100,000 or 100,000 or bigger than this. We cannot search sequentially, in that case it will take a lot of time.
Since dictionary has many words, and we cannot use sequential search for searching a

word in dictionary. So how do we search in dictionary?
Well, I open the approximate middle page of the dictionary. There are three possibilities.
If the word “hello” is before the middle page, my search reduces to first page to middle page.
If the word “hello” is after the middle page, my search reduces to middle page to last page.
If the word “hello” is on the middle page, I am finished.

Same method i can use for searching in an array, and it is better than sequential searching. This method of searching is called as **binary search**.

Here is the program for binary search.

```
void main(){
int e;    // complexity is 0
int a[10];        // complexity is 0
boolean flag=false;     // complexity is 1
int low=0;        // complexity is 1
high=9;           // complexity is 1

for(i=0;i<10;i++)
        scanf(“%d”,&a[i]);                      // complexity is O(n)

printf(“enter the element you want to search”);
scanf(“%d”,&e);

while (low <= high )
{
        mid = (low+high)/2;
        if (a[mid] == e)
        {
                flag=true;                      // what is the complexity??????
                break;
        }
        else if (e < a[mid])
                high = mid – 1;
        else
                low = mid + 1;
}

if (flag == false)
        printf(“element not found”);
else
        printf(“element found”);

}
```

There are two loops. One for loop accepts data from the user and store in the array.

Second for loop does the searching.
This example represent a different type of problem. Analysis of this type of problem is different from the analysis of previous problem.
Let us try to understand it. This problem is also called as **divide and conquer** problem. There are many sorting algorithms based on divide and conquer which we will study later. Analysis of those sorting algorithm will be same as this analysis.

We started searching in an array of size n.
After 1 step (i.e. Finding the middle), our search reduced into 2 parts, first half or second half. In other words, i can say, now we only need to search within n/2 elements. So after 1^{st} step our search reduced from size n to n/2.
After 2^{nd} step, our search will reduce to n/4.
After 3^{rd} step, our search will reduce to n/8.
.
.
.
Then finally at one step,we will find the element.

We can represent this using a mathematical notation.
$T(n) = T(n/2) + c.$
$T(1) = 1;$

Where T(n) is the complexity of binary search in an array of size n.
c represents a constant.

$T(n) = T(n/2) + c$
$T(n) = [\ T(n/2^2) + c] + c$
$T(n) = [\ [T(n/2^3) + c] + c] + c$
.
.
.
$T(n) = T(n/2^k) + k.c$ (1)

When $n/2^k = 1$
$\log n = \log 2^k$
$\log n = k \log 2$
$k = \log n / \log 2$
$k = \log_2 n$

So eq(1) is now
$T(n) = T(1) + c \log_2 n$
$T(n) = 1 + c \log_2 n$ (because T(1) = 1)

hence $T(n) = O(\log_2 n)$

log n grows much more slowly than n.

n	$\log_2 n$
1	0
1.5	0.41
2	0.69
5	1.61
8	2.08
20	3
50	3.91
80	4.38
100	4.61

Binary Search	***Best Case***	***Worst Case***	***Average Case***
	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$