Data Structures

Introduction and time complexity

An **algorithm** is the step-by-step unambiguous instructions to solve a given problem.

1. The set of instructions should produce correct output.

2. Should take minimum possible resources.

Five steps for thinking for an algorithm:

1. Constraints
2. Ideas Generation
3. Complexities
4. Coding
5. Testing

Constraints

1. Knowing the algorithms is not sufficient to designing a good software.
2. There may be some data, which is missing that you need to know before beginning to solve a problem.
3. In this step, we will capture all the constraints about the problem. We should never try to solve a problem that is not completely defined.

For example : When the problem statement says that write an algorithm to sort numbers.

Basic guideline for the Constraints**:**

1. **What is data type of values needs to sort**
2. **If the data is numeric (int or float)**
    1. **How many numbers of elements in the array**
    2. **What is the range of value in each element.**
    3. **Does the array contain unique data or not?**

String data

1. **Min and max length**
2. **Unique data or not**

Idea Generation

Following is the strategy that you need to follow to solve an unknown problem:

1. Try to simplify the task in hand.
2. Think of a suitable data-structure.
3. Think about similar problems you have already solved.

Complexities

1. The solution should be fast and should have reasonable memory requirement.
2. You should be able to do Big-O analysis.
3. Sometime taking some bit more space saves a lot of time and make your algorithm much faster.

Coding

1. Now, after capturing all the constraints of the problem, deriving few solutions, evaluating the complexities of the various solutions, you can pick one solution to do final coding.
2. Never ever, jump into coding before discussing constraints, Idea generation and complexity
3. Small functions need to be created so that the code is clean and managed.

Testing

1. Once coding is done ,it is a good practice that you go through your code line by line with some small test case. This is just to make sure your code is working as it is supposed to work.

Algorithm analysis: Algorithm analysis helps us to determine which algorithm is most efficient in terms of time and space consumed.

Goal of the Analysis of Algorithms: The goal of the analysis of algorithms is to compare algorithms (or solutions) mainly in terms of running time but also in terms of other factors (e.g., memory, developer effort, etc.)

To compare algorithms: will you consider

Execution times ?

Number of statements ?

Rate of growth ?

Types of Analysis

- **Worst case**
  - Defines the input for which the algorithm takes a long time (slowest time to complete).
  - Input is the one for which the algorithm runs the slowest.

- **Best case**
  - Defines the input for which the algorithm takes the least time (fastest time to complete).
  - Input is the one for which the algorithm runs the fastest.

- **Average case**
  - Provides a prediction about the running time of the algorithm.
  - Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.
  - Assumes that the input is random.

For a given algorithm, we can represent the best, worst and average cases in the form of expressions. As an example, let f(n) be the function which represents the given algorithm.

$f(n) = n^2 + 500$ for worst case
$f(n) = n + 100n + 500$ for best case

Big-O Notation [Upper Bounding Function]

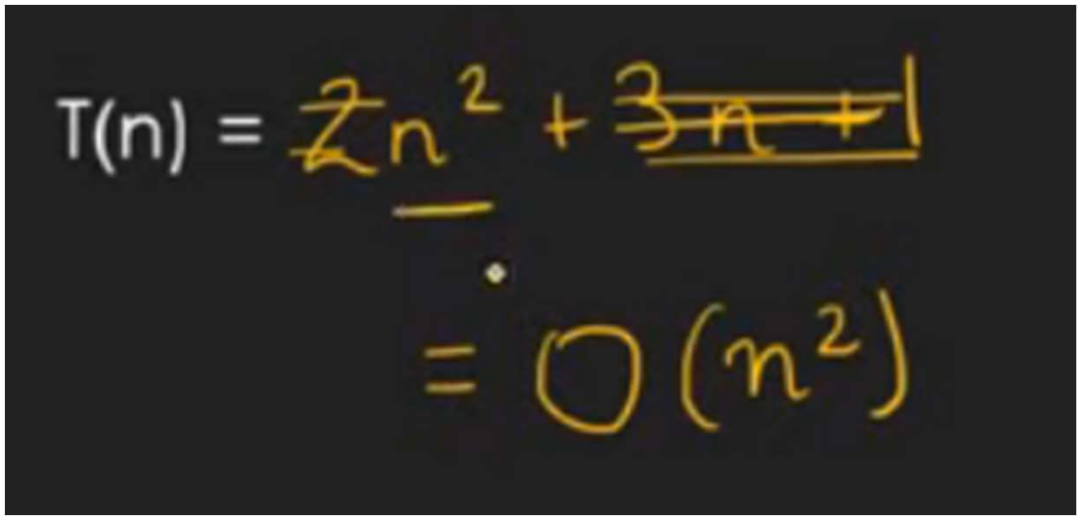
Omega-Q Notation [Lower Bounding Function]

Theta-Θ Notation [Order Function]

We generally focus on the upper bound (O) because knowing the lower bound Omega(Ω) of an algorithm is of no practical importance, and we use the ThetaΘ notation if the upper bound (O) and lower bound (Ω) are the same.

$T(n) = 2n^2 + 3n + 1$

- Drop lower order terms
- Drop all the constant multipliers



1) Loop

```
for( i = 1; i<=n; i++ ){
        x=y+z;
}
```

2) Nested Loop

```
for( i = 1; i<=n; i++ ){
        for( j = 1; j<=n; j++ ){
                x=y+z;
        }
}
```

```
for( i = 1; i<=n; i++ ){
        x=y+z; //constant time = c
}
```

$= cn$

$= O(n)$

```
for( i = 1; i<=n; i++ ){ //n times
        for( j = 1; j<=n; j++ ){ //n times
                x=y+z; //constant time
        }
}
```

$= O(n^2)$

4) If-else statements

```
if(condition)
{
  ---  O(n)
}

else
{
  ---  O(n²)
}
```

3) Sequential Statements

```
i) a = a + b;  // constant time = c1

ii) for( i = 1; i<=n; i++ ){
        x = y + z;          c2 n
    }

iii) for( j = 1; j<=n; j++ ){
        c = d + e;          c3 n
    }
```

```
if(condition)
{
  ---  O(n)
}

else
{
  ---  O(n²)
}
```

= O(n²)

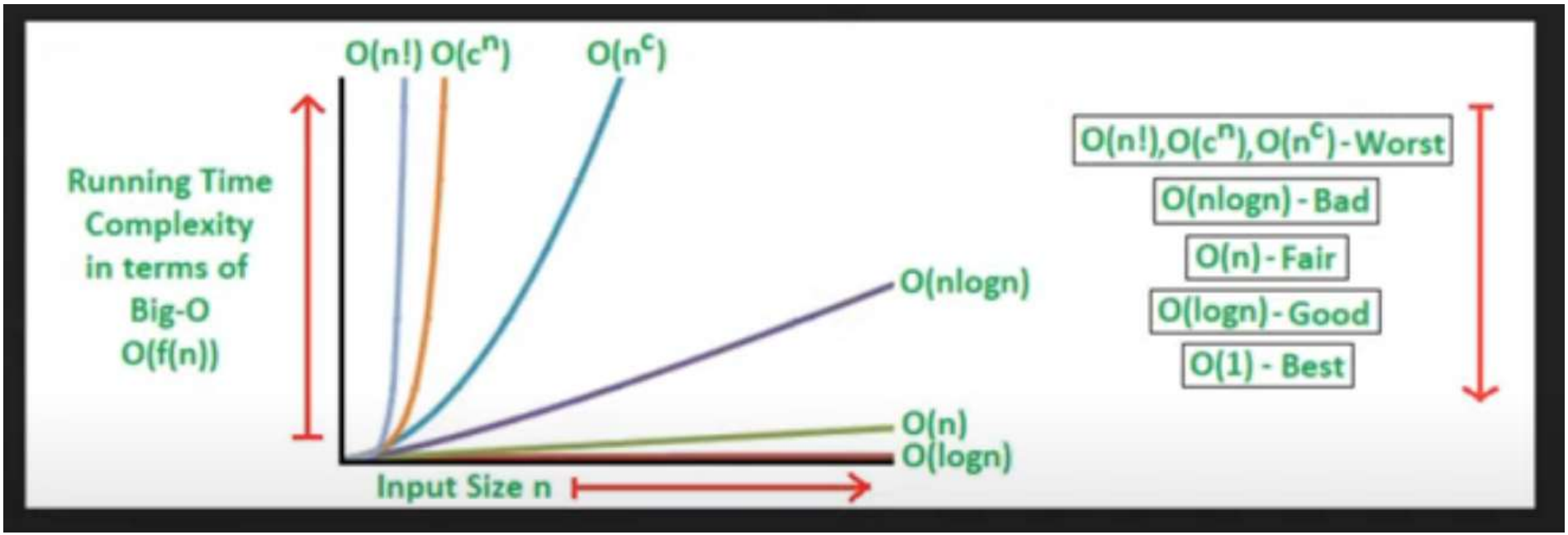
```
i) a = a + b;  // constant time = c1

ii) for( i = 1; i<=n; i++ ){
        x = y + z;          c2 n
    }

iii) for( j = 1; j<=n; j++ ){
        c = d + e;          c3 n
    }
```

= c1 + c2 n + c3 n

= O(n)



$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^c) < O(n!)$$

```
if(i > j ){
   j>23 ? cout<<j : cout<<i;
}
```

O(1)

```
for(i= 0 ; i < n; i++){
   cout<< i << " " ;
   i++;
}
```

O(n)

```
for(i= 0 ; i < n; i++){
   for(j = 0; j<n ;j++){
      cout<< i << " ";
   }
}
```

$O(n^2)$

```
int i = n;
while(i){
   cout << i << " ";
   i = i/2;
}
```

O(log n)

```
for(i= 0; i < n; i++){
   for(j = 1; j < n; j = j*2){
      cout << i << " ";
   }
}
```

O(n log n)

```
int a = 0;
for (i = 0; i < N; i++) {
        for (j = N; j > i; j--) {
                a = a + i + j;
        }
}
```

O(N*N)

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
        for (j = 2; j <= n; j = j * 2) {
                k = k + n / 2;
        }
}
```

O(nLogn)

```
int a = 0, i = N;
while (i > 0) {
        a += i;
        i /= 2;
}
```

O(log N)

```
function(int n)
{
        if (n==1)
        return;
        for (int i=1; i<=n; i++)
        {
                for (int j=1; j<=n; j++)
                {
                        printf("*");
                        break;
                }
        }
}
```

```
static void function(int n)
{
        int count = 0;
        for (int i = n / 2; i <= n; i++)
                for (int j = 1; j <= n; j = 2 * j)
                        for (int k = 1; k <= n; k = k * 2)
                                count++;
}
```

1

```
void function(int n)
{
   int i, count =0;
   for(i=1 ; i*i <=n ; i++)
      count ++;
}
```

2

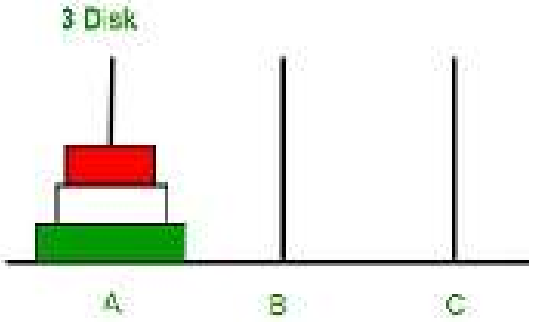
```
void funtion(int n)
{
  for(int I = 1;i<=n ; i++)
     for(int j = 1; j <= n ; j+=i)
        printf("*");
}
```

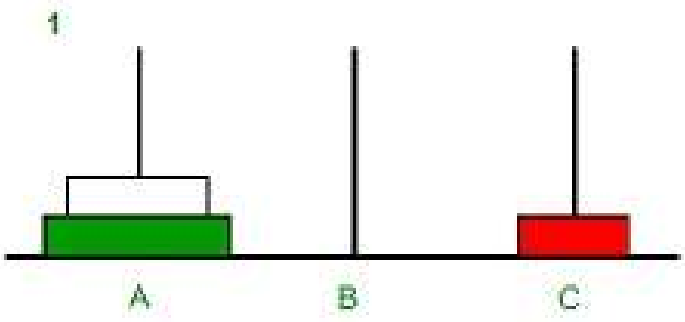
3

```
int fun(int n) {
    int i = 0, j = 0, k = 0, m = 0;
    for (i = 0; i<n ; i++)
      for(j = I; j<n ; j++)
        for(k=j+1 ; k<n ; k++)
            m+=1;
    return m;
 }
```

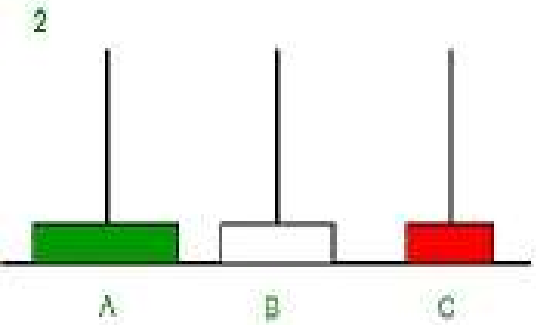
Time complexity with examples:

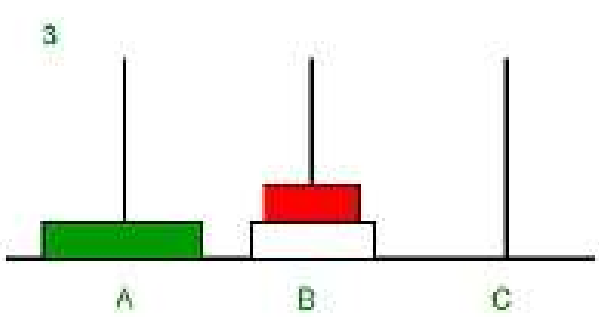
Time Complexity	Name	Example
1	Constant	Adding an element to the front of a linked list
$logn$	Logarithmic	Finding an element in a sorted array
n	Linear	Finding an element in an unsorted array
$nlogn$	Linear Logarithmic	Sorting n items by 'divide-and-conquer' - Mergesort
n^2	Quadratic	Shortest path between two nodes in a graph
n^3	Cubic	Matrix Multiplication
2^n	Exponential	The Towers of Hanoi problem

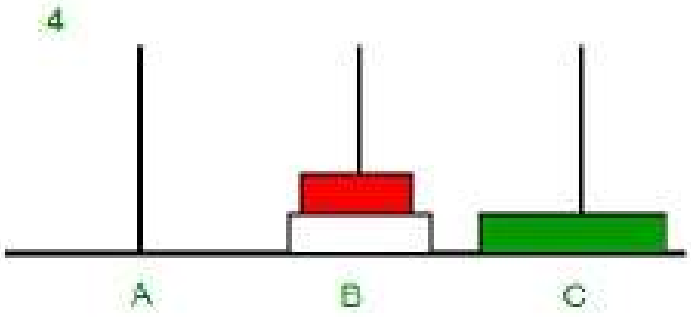


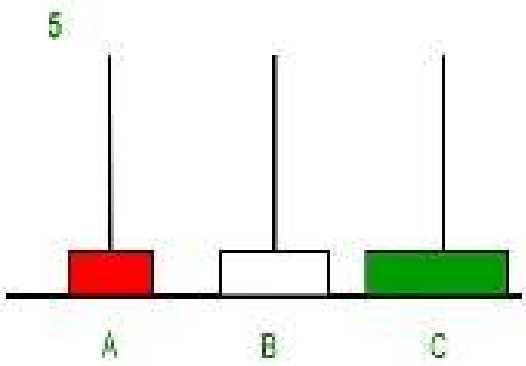


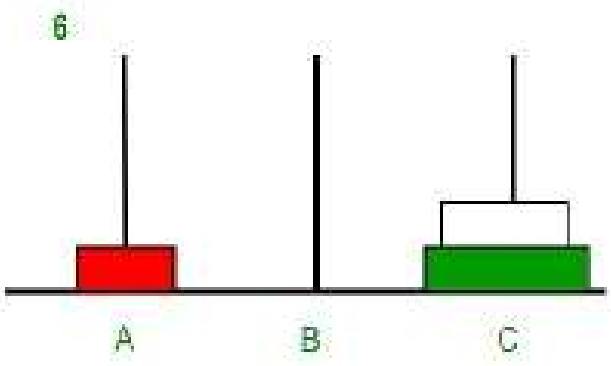
3 disks = 8 times

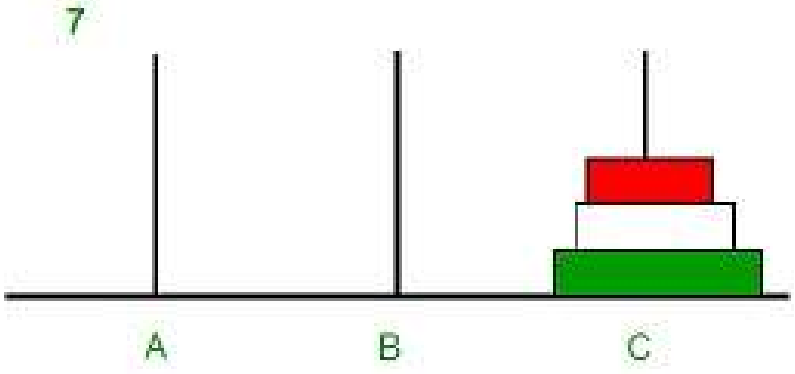












5 disks

2^5 times

Data Structures

Data Type:

• A data type defines a domain of allowed values and the operations that can be performed on those values. For example int, float, char data types are provided in C

• If an application needs to use a data type which is not provided as primitive data type of the language, then it is programmer's responsibility to specify the values and operations for that data type and implement it. For example data type for date is not provided in C, and we need dates to be stored and processed in our program then we need to define and implements the date data type.

Abstract Data Type:

• ADT is a concept that defines a data type logically. It specifies a set of data and collection of operations that can be performed on that data.

•ADT only mentions what operations are to be performed but not how these operations will be implemented.

•It does not specify how data will be organized in memory and what algorithms will be used for implementing operations.

•It is called abstract because it gives an implementation independent view.

•ADT is just like a black box which hides the inner structure and design of the data type.

Example of ADT:

List ADT

A list contains elements of same type arranged in sequential order and following operations can be performed on the list:

1. Initialize() - Initialize the list to be empty
2. get() - returns an element from the list at any given position
3. insert() - Inserts a new element at any position of the list
4. remove() - Removes the first occurrence of any element from a non empty list
5. removeAt() - Removes the element from a specific location from a non empty list
6. replace() - Replace an element at any position by another element
7. size() - Returns number of elements in the list
8. isEmpty() - Returns true if the list is empty, otherwise false
9. isFull() - Returns true if the list is full, otherwise false

Data Structures:

•Data structures is a programming construct used to implement an ADT.
•It is physical implementation of ADT.
•It contains collection of variables for storing data specified in the ADT.
•All operations specified in ADT are implemented through functions.

ADT is logical view of data and the operations to manipulate the data while Data Structures is the actual representation of data in memory and the algorithms to manipulate the data
ADT is a logical description while Data Structure is concrete
ADT is what is to be done and data structure is how to do it.
ADT is used by client program.

Advantages of Data Structures are-

Efficiency
Reusability
Abstraction

List of DS:
- Array
- Stack
- Queue
- Linked List
- Binary Tree
- BST
- Hash
- Graph

- searching and sorting algorithms
- time complexity and space complexity