K.R. MANGALAM UNIVERSITY
THE COMPLETE WORLD OF EDUCATION

# Course: Data Structure

(Course Code : ENCS205)

## UNIT-1: Foundations of Data Structures

School of Engineering & Technology
K.R. Mangalam University

# SESSION 8:
# Space and Time Complexity

# Recapitulation (Previous Session)

## Quiz

Q1: The asymptotic notation used to describe the upper bound of an algorithm's running time is called ________________.

Q2: The asymptotic notation θ (n log n) describes an algorithm that has both an upper and lower bound of ______________.

Q3: The notation Ω(n) provides a lower bound on the time complexity of an algorithm, meaning it will take at least _____________ time in the worst case.

Q4: O (n^2) describes an algorithm that is more efficient than one with a complexity of O(n log n). (True/False)

Q5: An algorithm with a time complexity of O (n^2) is considered to have exponential time complexity. (True/False)

# Recapitulation (Previous Session)

## Quiz (Answers)

A1: Big O

A2: n log n

A3: linear

A4: False

A5:  True

# Time Complexity

- Time complexity: estimate of time taken by the algorithm
- It is estimated by counting the number of elementary operations, where each elementary operation takes a fixed amount of time to perform.
- We use **asymptotic notations** for describing the time complexity of an algorithm.
- We ignore hardware, operating system, processors, etc.

# Space Complexity

- Space complexity : no. of memory units used w.r.t given input size.

- For any algorithm memory may be used for the following:
  - Variables (This include the constant values, temporary values)
  - Program Instruction
  - Execution

- Sometimes **Auxiliary Space** is confused with Space Complexity. But Auxiliary Space is the extra space or the temporary space used by the algorithm during it's execution.

- **Space Complexity = Auxiliary Space + Input space**

# Common Time Complexities

**O(1) is *constant-time* complexity.**

**The number of operations for the algorithm doesn’t actually change as the problem size increases.**

```
void Constant-Time(int arr[ ]) {
    printf("First element of array = %d", arr[0]); }
```

This function runs in O(1) time (or "constant time") relative to its input.

**A loop that runs a constant number of times is also considered as O(1).**

```
for (int i = 1; i <= C; x++)  // C is some constant
{ // some O(1) expressions }
```

## Common Time Complexities

### O(n): Linear-Time complexity

- Running time increases at most linearly with the size of the input.

- There is a constant $c$ such that the running time is at most $cn$ for every input of size $n$.

- Linear time is the best possible time complexity in situations where the algorithm has to sequentially read its entire input.

- This function runs in O(n) time, where n is the number of items in the array.

```
void Linear-Time(int arr[])
{
    printf("First element of array = %d",arr[i]);
}
```

# Common Time Complexities

## O(n): Linear-Time complexity (Cont..)

**Examples**

1. Traversing an array
2. Traversing a linked list
3. Linear Search
4. Deletion of a specific element in a Linked List (Not sorted)
5. Comparing two strings
6. Checking for Palindrome
7. Counting/Bucket Sort

# Common Time Complexities

## O($N^2$) time complexity

- O($N^2$) - performance is directly proportional to the square of the size of the input data set.
- Is common with algorithms that involve nested iterations over the input.

```
void Nsquare (int arr[], int size=N)
{
   for (int i = 0; i < size; i++)
   {
      for (int j = 0; j < size; j++)
            printf("%d = %d\n", arr[i],
arr[j]);
    }
}
```

**If the array has 10 items, we have to print 100 times. If it has 1000 items, we have to print 1000000 times**

# Common Time Complexities

```
int count = 0;
for (int i = 0; i < N; i++)
    for (int j = 0; j < i; j++)
        count++;
```

The running time of statement inside the nested loop is product of size of loops.

```
for (i=0; i<n; i++)
for (j=0; j<k; j++)
a++;
```

The running time is $O(n \times k)$

So the time complexity will be $O(N^2)$.

## Common Time Complexities

### O(log n) is logarithmic complexity

- Time Complexity of a loop is considered as O(Log n) if the loop variables is divided / multiplied by a constant amount.

**For example Binary Search has O(logn) time complexity.**

```
for (int i = 1; i <=n; i *= c)
{ // some O(1) expressions }

for (int i = n; i > 0; i /= c)
{ // some O(1) expressions }
```

**The series that we get in first loop is 1, c, $c^2$, $c^3$, … $c^k$. If we put k equals to $Log_c n$,**
**we get $C^{Log_c n}$ which is n.**

# Common Time Complexities

## O(LogLogn)

- if the loop variables is reduced exponentially by a constant amount.

```
// Here c is a constant greater than 1
for (int i = 2; i <=n;i = pow(i, c))
{ // some O(1) expressions }

//Here fun is sqrt or cuberoot or any other constant root
for (int i = n; i > 0; i = fun(i))
{ // some O(1) expressions }
```

# Common Time Complexities

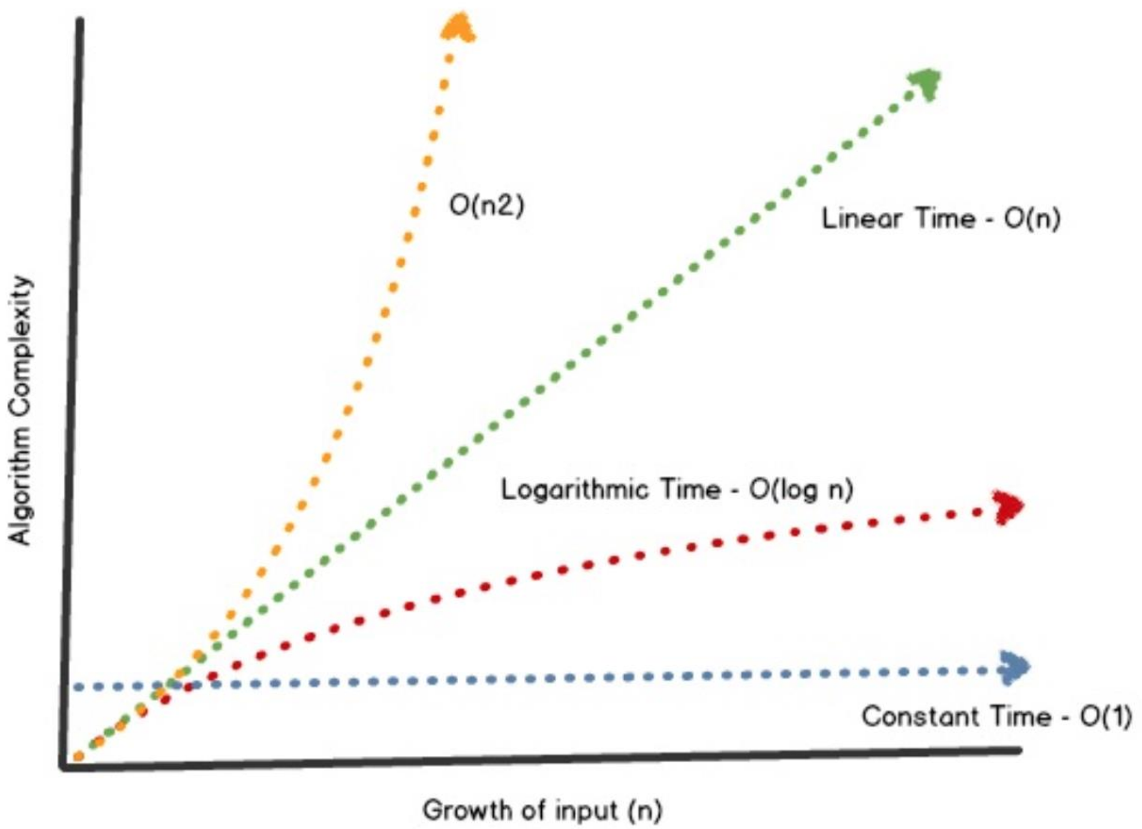
- **$O(n^3)$, $O(n^4)$, $O(n^5)$, etc. : *polynomial* complexity**
- **$O(2^n)$ : *exponential* complexity.**

| | |
|---|---|
| $2^5$ | 32 |
| $2^{10}$ | 1024 |
| $2^{100}$ | 1.2676506002282E+30 |

**General ordering among functions.**

**$O(1) < O(\log n) < O(n) < O(n \log n) < O(n*n) < O(n*n*n) < O(n^k) < O(2^n)$**

# O(1),O(n), O(N2) VS O(log n) time complexities



# Comparison Between Various Complexities

| n | $\log_2 n$ | n | $n\log_2 n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|
| 10 | 3.3 μs | 10 μs | 33 μs | 100 μs | 1 ms | 1 ms |
| $10^2$ | 6.6 μs | 100 μs | 664 μs | 10 ms | 1 s | $4\times10^{16}$ years |
| $10^4$ | 13 μs | 10 ms | 133 ms | 1.7 minutes | 11.6 days | $10^{2997}$ years |
| $10^6$ | 20 μs | 1 s | 20 s | 11.6 days | 32000 years | $10^{300000}$ years |

*Assuming one million operations per second

**Important points to note:**

- $O(\log n)$ excellent complexity.
- $O(n\log n)$ is pretty good: hard to complain about it.
- $O(n^k)$ could be bad, depending on $k$
- $\Omega(k^n)$ is a disaster.

**If you have an algorithm with a higher complexity than necessary, no amount of clever programming will make up for it**

# Space Complexity

```
int ADD(int x, int y, int z) {
int r = x + y + z;
return r;}
```

requires 3 memory units w.r.t parameters and 1 for the local variable, and this never changes, so this is O(1)

```
SpaceComplexityDemo(A,n)
{
  for i=1 to n
   A[i]=1
}
```

Here input size is n, we are using just one extra variable here i.e i

```
int ADD(int arr[], int n){
int k = 0;
  for (int i = 0; i < n; ++i) {
    k += arr[i];}
return k
}
```

requires 'n' units for arr, plus space for n, k and i, so it's O(n)

```
ADD()
{
int Z = P + Q + R;
  return(z);
}
```

Variables P,Q,R and Z are all integer types, hence they will take up 2 bytes each, so total memory requirement will be (8 + 2) = 10 bytes, additional 2 bytes is for return value. here space requirement is fixed, hence it is called Constant Space Complexity.

# Examples of Space Complexity

```
// n is the length of array a[]
int ADD(int A[], int n)
{           int x = 0;  // 2 bytes for x
            for(int i = 0; i < n; i++)
        // 2 bytes for i
            { x  = x + A[i];}
            return(x); }
```

```
void ADD(int x[], int y[], int z[], int n) {
  for (int i = 0; i < n; ++i) {
     z[i] = x[i] + y[i]    }
}
```

- here 2*n bytes for the array A[] elements.
- It is using 'n' memory units of its own space
- 2 bytes each for x, n, i and the return value.

Hence the total memory requirement is (2n + 8), It is Linear Space Complexity

requires 'n' units for x[], 'n' units for Y[] and 'n' units for z[] and 1 unit for i and n.
amount of storage= n+n+n+1+1=3n+2

# QUIZ (Time Complexity)

## What is the time, space complexity of following code:

```
int x = 0, y = 0;
for (i = 0; i < N; i++) {
    x = x + rand();
}
for (j = 0; j < M; j++) {
    y = y + rand();
}
```

**Options:**

1. O(N * M) time, O(1) space
2. O(N + M) time, O(N + M) space
3. O(N + M) time, O(1) space
4. O(N * M) time, O(N + M) space

**Output:**

3. O(N + M) time, O(1) space

**The first loop is O(N) and the second loop is O(M). Since we don't know which is bigger, we say this is O(N + M). This can also be written as O(max(N, M)).**

**Since there is no additional space being utilized, the space complexity is constant / O(1)**

# QUIZ (Time Complexity)

## What is the time complexity of following code:

```
int c = 0;
for (x = 0; x < N; i++) {
    for (y = N; y > x; y--) {
        c = c + x + y;
    }
}
```

1. **O(N)**
2. **O(N*log(N))**
3. **O(N * Sqrt(N))**
4. **O(N*N)**

## Output:

4. O(N*N)

**Explanation:**

The above code runs total no of times

= N + (N – 1) + (N – 2) + … 1 + 0

= N * (N + 1) / 2

= 1/2 * N^2 + 1/2 * N

O(N^2) times.

# QUIZ (Time Complexity)

**What does it mean when we say that an algorithm A is asymptotically more efficient than B?**

**Options:**

1. A will always be a better choice for small inputs
2. A will always be a better choice for large inputs
3. B will always be a better choice for small inputs
4. B will always be a better choice for all inputs

**Output:**

**2. A will always be a better choice for large inputs**

**Explanation:** In asymptotic analysis we consider growth of algorithm in terms of input size. An algorithm A is said to be asymptotically better than B if A takes smaller time than B for all input sizes n larger than a value n0 where n0 > 0.

# QUIZ (Time Complexity)

**The complexity of linear search algorithm is**

a) O(n)
b) O(log n)
c) O($n^2$)
d) O(n log n)

**Answer: a**
**Explanation: The worst case complexity of linear search is O(n).**

# QUIZ (Time Complexity)

**What is the time complexity of following code:**

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

**Options:**

1. O(N)
2. O(Sqrt(N))
3. O(N / 2)
4. O(log N)

**Output:**

4. O(log N)

**Explanation:**

We have to find the smallest x such that

N / 2^x N

x = log(N)

# QUIZ (Space Complexity)

```
// n is the length of array a[]
int sum(int a[], int n)
{
    int x = 0;
  // 4 bytes for x
    for(int i = 0; i < n; i++)
  // 4 bytes for i
    {
      x  = x + a[i];
    }
    return(x);
}
```

- 4*n bytes of space is required for the array a[] elements.
- 4 bytes each for x, n, i and the return value.

Hence the total memory requirement will be (4n + 12), which is increasing linearly with the increase in the input value n, hence it is called as Linear Space Complexity.

# QUIZ (Space Complexity)

```
Procedure SpaceComplexityDemo(A,n)
{  for i=1 to n
   A[i]=1
}
```

Here input size is n, we are using just one extra variable here i.e i

```
void add(int a[], int b[], int c[], int n) {
    for (int i = 0; i< n; ++i) {
        c[i] = a[i] + b[0]
    }
}
```

Which requires N units for a, M units for b and L units for c and 1 unit for i and n. So it will need N+M+L+1+1 amount of storage

# QUIZ (Space Complexity)

```
def calculate(a,b,c):
      return a+b/c
```

$S(p) = 1 + 1 + 1 = 3 \implies O(1)$

```
def sum(a, n):
 s = 0
for i in range(n):
  s += a[i]
  return s
```

S(p) = (n*1 + 1 + 1) + 1 ==> O(n) and auxiliary S(p) = O(1)

a. 'n' for Array of size n
b. '1' each for variable s,i,n
c. Additional space of 1 unit for var s

```
public static String[]
arrayOfHiNTimes(int n) {
   String[] hiArray = new String[n];
   for (int i = 0; i < n; i++) {
      hiArray[i] = "hi";     }
   return hiArray; }
```

This method takes *O*(*n*) space (the size of hiArray scales with the size of the input)

# Practice Questions

**Q1: Calculate space complexity of following program:**

```
// n is the length of array a[]
int sum(int a[], int n)
{
    int x = 0;
  // 4 bytes for x
    for(int i = 0; i < n; i++)
 // 4 bytes for i
    {
      x  = x + a[i];
    }
    return(x);
}
```

# Practice Questions

**Q2: Calculate space complexity of following program:**

```
public static int getLargestItem(int[]
    items) {
   int largest = Integer.MIN_VALUE;
   for (int item : items) {
      if (item > largest) {
         largest = item;
      }
   }
   return largest;
}
```

# Practice Questions (Answers)

**A1:**

- **4*n bytes of space is required for the array a[] elements.**
- **4 bytes each for x, n, i and the return value.**

**Hence the total memory requirement will be (4n + 12), which is increasing linearly with the increase in the input value n, hence it is called as Linear Space Complexity.**

**A2:** **This method takes $O(n)$ space (the size of the array scales with the size of the input)**

# Practice Questions- Try Yourself

**Q.1 Time complexity of binary search?**

A. O(1)

B. O(logn)

C. O((logn)2)

D. O(n)

**Q. 2 What is Time Complexity of**

```
ans = 0
for i = 1 to n:
for j = 1 to log(i):
ans += 1
```

A. O(n)

B. O(nlogn)

C. O(n2)

D. O(n3)

**Q.3 What is Time complexity of**

```
def f(a):
        n = len(a)          j = 0
        for i = 0 to n - 1:
        while (j < n and a* < a[j]):
                j += 1
```

A. O(logn)

B. O(n)

C. O(nlogn)

D. O(n2)

**Q. 4 What is Time Complexity of this program:**

```
ans = 0
while (n > 0):
ans += n % 10
n /= 10;
```

A. O(log2n)

B. O(log3n)

C. O(log10n)

D. O(n)

# THANK YOU