23CSE203
DATA STRUCTURES & ALGORITHMS
L-T-P-C: 3-1-2-5

Introduction to Data Structures: Need and Relevance - Abstract Data Types and Data Structures - Principles, and Patterns. Basic complexity analysis – Best, Worst, and Average Cases - **Asymptotic Analysis** - **Analyzing Programs** – Space Bounds, recursion- linear, binary, and multiple recursions. Arrays, Linked Lists and Recursion: Using Arrays - Lists - Array based List Implementation – Linked Lists – LL ADT – Singly Linked List – Doubly Linked List – Circular Linked List Stacks and Queues: Stack ADT - Array based Stacks, Linked Stacks – Implementing Recursion using Stacks, Stack Applications. Queues - ADT, Array based Queue, Linked Queue, Double-ended queue, Circular queue, applications.

Course Outcome:

COs	Course Outcome Description	BTL
CO1	Understand the concept and functionalities of Data Structures and be able to implement them efficiently	L3

J.UMA, AP-CSE

Amrita School of Computing

O(1) – Constant Time

Accessing an element in a list

```
def get_first_element(arr):
    return arr[0]
```

Time: O(1)

Simple arithmetic

```
def is_even(n):
    return n % 2 == 0
```

Time: O(1)

Swapping two variables

```
def swap(a, b):
    a, b = b, a
    return a, b
```

Time: O(1)

O(n) – Linear Time

- **Loops**: The running time of a loop is, at most, the running time of the statements inside the loop (including tests) multiplied by the number of iterations

```
// executes □    times
for (i=1; i<=n; i++)
m = m + 2; // constant time, c
```

$$\text{Total time} = \text{a constant } c \times n = c\,n = O(n).$$

O(n) – Linear Time

- **If-then-else statements:** Worst-case running time: the test, plus either the then part or the else part (whichever is the larger).

```
//test: constant
if(length( ) == 0 ) {
return false; //then part: constant
}
else { // else part: (constant + constant) * n
for (int n = 0; n < length( ); n++) {
// another if : constant + constant (no else part)
if(!list[n].equals(otherList.list[n]))
//constant
return false;
}
}
```

$$\text{Total time} = c_0 + (c_1 + c_2) * n = O(n).$$

O(n) – Linear Time

Sum of n numbers

```
def sum_n(n):
    total = 0
    for i in range(1, n+1):
        total += i
    return total
```

- Time: O(n)

Linear search

```
def linear_search(arr, key):
    for i in range(len(arr)):
        if arr[i] == key:
            return i
    return -1
```

Time: O(n)

O(n^2) – Quadratic Time

- **Nested loops:** Analyze from the inside out. Total running time is the product of the sizes of all the loops.

```
//outer loop executed n times
for (i=1; i<=n; i++) {
// inner loop executed n times
for (j=1; j<=n; j++)
k = k+1; //constant time
}
```

$$\text{Total time} = c \times n \times n = cn^2 = O(n^2).$$

O(n^2) – Quadratic Time

- Bubble Sort

Time: $O(n^2)$

```
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(n - i - 1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
```

- Printing all pairs

```
def print_pairs(arr):
    for i in arr:
        for j in arr:
            print(i, j)
```

Time: $O(n^2)$

O(n^2) – Quadratic Time

- **Consecutive statements:** Add the time complexities of each statement.

```
x = x +1; //constant time
// executed n times
for (i=1; i<=n; i++)
m = m + 2; //constant time
//outer loop executed n times
for (i=1; i<=n; i++) {
//inner loop executed n times
for (j=1; j<=n; j++)
k = k+1; //constant time
}
```

$$\text{Total time} = c_0 + c_1 n + c_2 n^2 = O(n^2).$$

O(log n) – Logarithmic Time

- **Logarithmic complexity:** An algorithm is O(log n) if it takes a constant time to cut the problem size by a fraction (usually by ½).

```
for (i=1; i<=n;)
i = i*2;
```

- If we observe carefully, the value of i is doubling every time. Initially i = 1, in next step i = 2, and in subsequent steps i = 4, 8 and so on.
- Let us assume that the loop is executing some k times. At kth step 2^k= n, and at (k + 1)th step we come out of the loop.
- Taking logarithm on both sides, gives O(log n) is the TC.

O(log n) – Logarithmic Time

- Binary Search

```
def binary_search(arr, key):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```

Time: O(log n)

O(log n) – Logarithmic Time

- Repeated division

```python
def divide_until_one(n):
    while n > 1:
        n //= 2
```

Time: O(log n)

$O(2^n)$ – Exponential Time

Recursive Fibonacci

```
def fib(n):
    if n <= 1:
        return n
    return fib(n-1) + fib(n-2)
```

At level 0 → 1 call

At level 1 → 2 calls

At level 2 → 4 calls

At level k → 2^k calls

So for depth ≈ n, we get:

Total calls ≈ $1 + 2 + 4 + \ldots + 2^n = O(2^n)$

Time: $O(2^n)$

O(√n)– Square root Time

- Floor Square root
- The loop condition is: i * i <= n
- Loop continues as long as the square of i is less than or equal to n.
- i will take values from 1 to floor(√n)

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

- **Time Complexity: O(√n)**

Loop 1: for(i = n/2; i <= n; i++)

Starts at n/2 and runs to n

$O(n)$

Loop 2: for(j = 1; j + n/2 <= n; j++)

$j \le n - n/2 = n/2$

runs approximately n/2 times

(Note: the bound doesn't depend on i, so it's independent.)

$O(n)$

Loop 3: for(k = 1; k <= n; k = k * 2)

k is doubling every time: $1, 2, 4, 8, \ldots, \le n$

This loop runs $\log_2(n)$ times$\Rightarrow$

$O(\log n)$

$O(n) \times O(n) \times O(\log n) = O(n^2 \log n)$

```
public void function(int n) {
    int i, j, k, count = 0;

    for(i = n/2; i <= n; i++)                // Loop 1
        for(j = 1; j + n/2 <= n; j++)        // Loop 2
            for(k = 1; k <= n; k = k * 2)    // Loop 3
                count++;
}
```

✅ Outer Loop (`i = n/2 to n`):

- Runs from `i = n/2` to `n`, i.e., approximately **n/2** times
- So: `O(n)`

✅ Middle Loop (`j = 1 to n`, doubling each time):

- Values of `j`: `1, 2, 4, 8, ..., up to n`
- This is a **logarithmic loop**, specifically **$\log_2(n)$** iterations
- So: `O(log n)`

✅ Inner Loop (`k = 1 to n`, doubling each time):

- Again, values of `k`: `1, 2, 4, ..., ≤ n`
- Another **logarithmic loop** — also `O(log n)`

```
public void function(int n) {
    int i, j, k, count = 0;

    // Outer loop: i from n/2 to n
    for(i = n / 2; i <= n; i++)
        // Middle loop: j = 1; j <= n; j *= 2
        for(j = 1; j <= n; j = 2 * j)
            // Inner loop: k = 1; k <= n; k *= 2
            for(k = 1; k <= n; k = k * 2)
                count++;
}
```

$$O(n) \times O(\log n) \times O(\log n) = O(n * \log^2 n)$$

TIME COMPLEXITY PROBLEM

- REFER TC PROBLEMS HOMEWORK NOTEPAD FOR MORE PROBLEMS