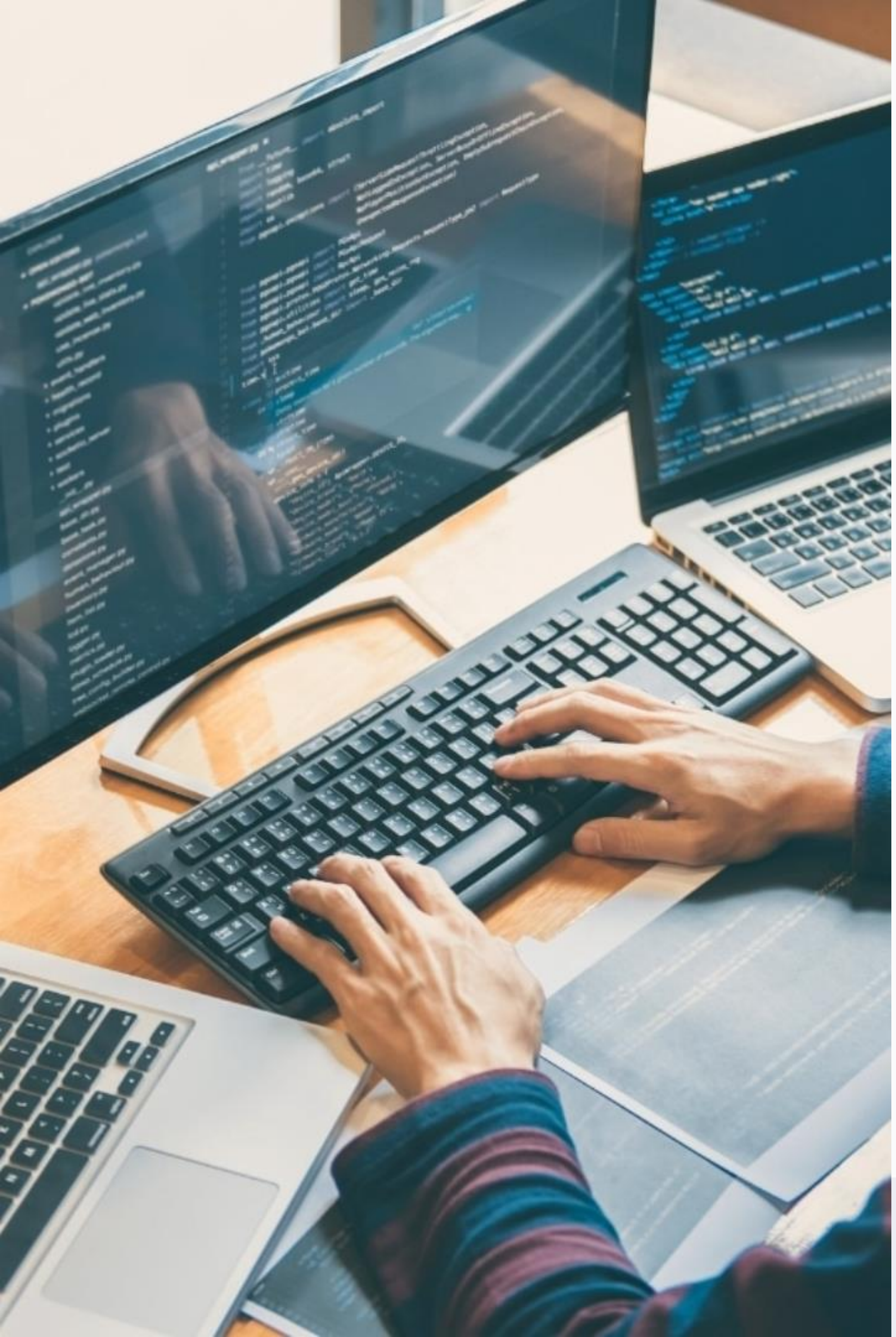

Advanced Data Structures and Algorithms (R1UC503B)

Time & Space Complexity

Dr. Gopal Chandra Jana
Assistant Professor, SCSE
Galgotias University

Understanding Time and Space Complexity

The goal of the analysis of algorithms is to **compare algorithms (or solutions) mainly in terms of running time and/or memory** but also in terms of other factors (e.g., developer effort, scalability, Adaptability, etc.)

- **Efficient algorithms save resources** (time and memory)

Running Time Analysis?

It is the process of determining how processing time increases as the size of the problem (input size) increases. Input size is the number of elements in the input, and depending on the problem type, the input may be of different types.

The following are the common types of inputs.

- Size of an array
- Polynomial degree
- Number of elements in a matrix
- Number of bits in the binary representation of the input
- Vertices and edges in a graph.

How to Compare Algorithms?

A few objective measures to be considered while comparing algorithms:

- **Execution times?** Not a good measure as **execution times are specific to a particular computer**.
- **Number of statements executed?** Not a good measure, since the number of statements **varies with the programming language as well as the style of the individual programmer.**
- **Feasible solution?** Let us assume that we express the running time of a given algorithm as a function of the input size n (i.e., f(n)) and compare these different functions corresponding to running times. This kind of comparison is independent of machine time, programming style, etc.

Understanding Time and Space Complexity

There are three types of analysis:

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

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

- **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.**

General rules to help us **determine the running time** of an algorithm.

Loops:

```
for( i=1; i<=n; i++)
    break;                  //constant time
```

Total Time = c 1 → O(1)

Loops:

```
for( i=1; i<=n; i++)        //Executes n times
    sum=sum+2;              //constant time
```

Total Time = c x n → O(n)

General rules to help us **determine the running time** of an algorithm.

Nested Loops:

```
for( i=1; i<=n; i++)            //Outer loop Executes n times
    for( j=1; j<=n; j++)        //Executes n times
        sum=sum+2;              //const time
```

Total Time = c x n x n → $O(n^2)$

General rules to help us **determine the running time** of an algorithm.

Consecutive Statements:

```
K = k-1;                  //Constant time
for( i=1; i<=n; i++)      //Executes n times
    sum=sum+2;            //constant time

for( i=1; i<=n; i++)      //Outer loop Executes n times
    for( j=1; j<=n; j++)  //Executes n times
        m=m-2;            //constant time
```

Total Time = c1 + c2 x n + c3 x n x n → $O(n^2)$

General rules to help us **determine the running time** of an algorithm.

If-then-else:

```
If(length()==0)                          //constant time
        return false;                    //constant time
else
{
for(int  n=0; n<length(); n++)           //Executes n times
    if( !Arr[n].equals(Arr2[n]))         //constant time
                return false;            //constant time
}
```

Total Time = $c1 + c2 + (c3+c4) \times n \rightarrow O(n)$

General rules to help us **determine the running time** of an algorithm.

Logarithmic:

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

2,4,8,…→ $2^n = m$

Total Time = O(log m)

Logarithmic:

```
for( i=n; i>=1; )
    i=i/2;
```

Total Time = O(log m)

General rules to help us **determine the running time** of an algorithm.

Nested Loops:

```
for( i=1; i<=n; i++)           //Outer loop Executes n times
    for( j=1; j<=n; j=j*2)     //Executes log(n) times
        Print();               //const time
```

Total Time = c x n x log(n) → O(nlog(n))

Nested Loops:

```
for( i=1; i<=n; i++)           //Outer loop Executes n times
    for( j=1; j<=n; j=j++)     //Executes n times
        if ( j%2==1)
            break;             //const time
```

Total Time = c x n x 1 → O(n)

General rules to help us **determine the running time** of an algorithm.

Square Root:

```
for( i=1; i x i<=n; i++ )
    print();
```

Total Time = O(sqrt(n))

Cube Root:

```
for( i=1; i x i x i<=n; i++ )
    print();
```

Total Time = O(CubeRoot(n))

General rules to help us **determine the running time** of an algorithm.

Multiple Loops:

```
for( j=1; j<N; j++)            //Executes N times
    print();                   //const time
for( i=1; i<M; i++)            //Executes M times
    print();                   //const time
```

Total Time = c x N x M → O(N+M)

Space Complexity = O(1)

Space Complexity

The term Space Complexity is misused for Auxiliary Space at many places.

Following are the correct definitions of Auxiliary Space and Space Complexity.

***Auxiliary Space* is the extra space or temporary space used by an algorithm.**

***Space Complexity* of an algorithm is the total space taken by the algorithm with respect to the input size.** Space complexity includes both Auxiliary space and space used by input.

For example, if we want to compare standard sorting algorithms on the basis of space, then Auxiliary Space would be a better criterion than Space Complexity. Merge Sort uses O(n) auxiliary space, Insertion sort, and Heap Sort use O(1) auxiliary space. The space complexity of all these sorting algorithms is O(n) though.

Space complexity is a parallel concept to time complexity.
If we need to create an array of size n, this will require O(n) space.
If we create a two-dimensional array of size n*n, this will require O(n^2) space.

Set of Problems :

```
for( j=1; j<=n; j++)   // j<=n
    print();
```

Time Complexity = O(?   )

```
for( j=1; j<n; j++)              // j<n
    print();
```

Time Complexity = O(?   )

```
If (x%2==0)
    print();
```

Time Complexity = O(?   )

General rules to help us **determine the running time** of an algorithm.

```
for( j=1; j<=n; j=j+2)
    print();
```

// **j=j+2**

Time Complexity = O(?   )

```
for( j=1; j * j * j <= n; j++)
```

// **j ***

j * j <=n

```
    print();
```

```
for( j=1; j <= n; j=j * 3)
    print();
```

// **j =j*3**

Time Complexity = O(?   )

General rules to help us **determine the running time** of an algorithm.

```
for( j=n; j>=1; j=j/x)          // j=j/x
    print();
```

Time Complexity = O(?)

```
for( j=1; j <= n; j++)
    for(i=1; i<=j; i++)
        print();
```

Time Complexity = O(?)

```
for( j=1; j <= n; j++)
    for(i=1; i<=n; i++)
      if (i%2==0)
             break;
```

Time Complexity = O(?)