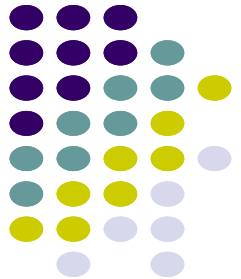
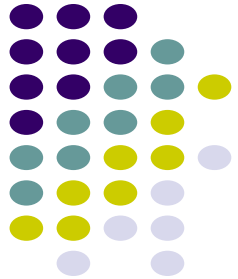
Asymptotic Analysis:
Big-O Notation
Time and Space Complexity



Asymptotic Analysis

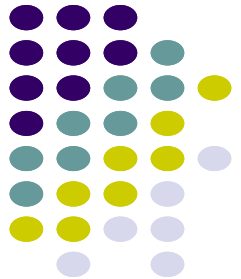


- The efficiency of an algorithm depends on the amount of time, storage and other resources required to execute the algorithm.
- The efficiency is measured with the help of asymptotic notations.
- An algorithm may not have the same performance for different types of inputs.
- With the increase in the input size, the performance will change.
- The study of change in performance of the algorithm with the change in the order of the input size is defined as asymptotic analysis.

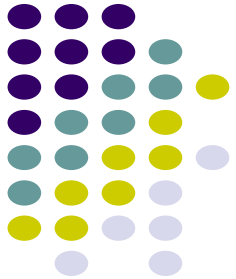
Asymptotic Notations

- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.
- For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear i.e. the best case.
- But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.
- When the input array is neither sorted nor in reverse order, then it takes average time.
- These durations are denoted using asymptotic notations.

Big O Notation: Understanding Time and Space Complexity in Algorithms

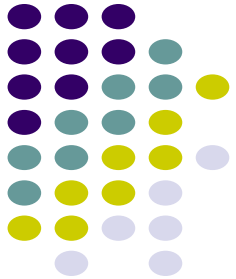


Big O Notation



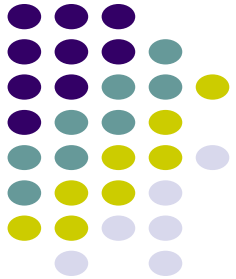
- In the field of computer science and algorithm analysis, understanding the performance characteristics of algorithms is of paramount importance.
  - How fast it runs? Time Complexity
  - How much space does it consume? Space Complexity
- The Big O notation is a powerful tool used to express the time and space complexity of algorithms.
- It allows us to compare and contrast different algorithms, predicting how they will scale with larger inputs and identifying potential bottlenecks in their execution.

What is Big O Notation?



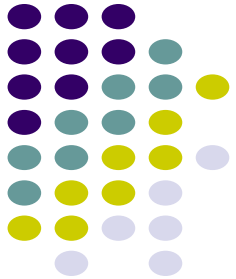
- Big O Notation is a way of describing the efficiency of a piece of code (usually a function).
- It helps us have a standardized way of answering the following questions:
  - How fast is this piece of code (function)?
  - How much additional space does this piece of code (function) consume?
- The “O” in Big O stands for “order of” and the “f(n)” represents the function that defines the growth rate of the algorithm as a function of the input size.
- The function f(n) can be any mathematical expression or formula that represents the number of operations performed or the amount of memory used by the algorithm.

Time Complexity



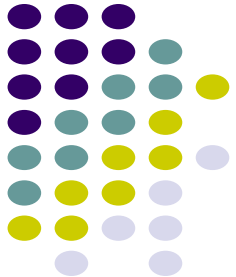
- Time complexity in Big O notation is a measure of how an algorithm’s running time increases with the size of its input.
- It provides an estimate of the worst-case time required to execute an algorithm as a function of the input size.
- In other words, it gives us an upper bound on the time taken by the algorithm to complete its task.
- There are several common time complexities expressed using Big O notation

Time Complexity



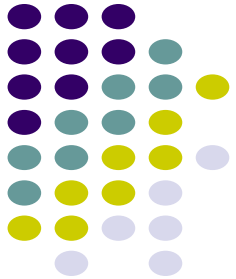
- **O(1) — Constant Time**: The algorithm's running time does not depend on the size of the input; it performs a fixed number of operations.
- **O(log n) — Logarithmic Time**: The algorithm's running time grows logarithmically with the size of the input.
- **O(n) — Linear Time**: The algorithm's running time scales linearly with the size of the input.
- **O(n log n) — Linearithmic Time**: The algorithm's running time grows in proportion to n times the logarithm of n.
- **O(n²) — Quadratic Time**: The algorithm's running time is directly proportional to the square of the input size.
- **O(2^n) — Exponential Time**: The algorithm's running time doubles with each increase in the input size.

Space Complexity



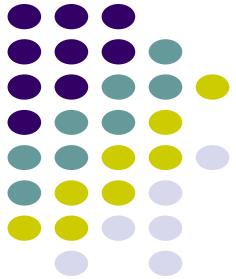
- Space complexity in Big O notation measures the amount of memory used by an algorithm with respect to the size of its input.
- It represents the worst-case memory consumption as the input size increases.
- Similar to time complexity, space complexity also has different notations:

Space Complexity



- **O(1) — Constant Space**: The algorithm uses a fixed amount of memory that does not depend on the input size.
- **O(n) — Linear Space**: The algorithm's memory usage grows linearly with the input size.
- **O(n^2) — Quadratic Space**: The algorithm's memory usage increases proportionally to the square of the input size.

Counting Analysis



- A method to calculate the time and space complexity of a function with respect to the input.
- It makes some assumptions to simplify Big O calculation.
- Mathematical, relational, assignment and return statement count as 1 operation. In reality, they are multiple operations but from the perspective of computing Big O, they are constant, thus we ignore the details.

Example

```
public static int SumToNOneTwo(num)
    {
        return num * (num + 1) / 2;
    }
```

int SumToNOneTwo(num)			
return num * (num + 1) / 2;	1 + 1 + 1 + 1	4	1 addition, 1 Multiplication, 1 division, 1 return
		O(4)	
		O(1)	

Time complexity -> O(1)
Space complexity -> O(1). Space used by num is constant with respect the input (num)

Example

```
public static int SumToNOne(int num)
    {
        int sum = 0;
        for (int i = 0; i <= num; i++)
        {
            sum += i;
        }
        return sum;
    }
```

int SumToNOne(int num)			
int sum = 0;	1	1	1 assignment
for (int i = 0; i <= num; i++)	1 + (1 + 1) * N	1 + 2N	1 assignment, 1 comparison 1 increment done N times
sum += i;	(1 + 1) * N	2N	1 addition 1 assignment done N times
return sum;	1	1	1 return
		4N + 3	
		O(N)	

Time complexity -> O(N). N refers to the input here.
Space complexity -> O(1). Spaces used by variables sum, i are constant with respect the input (N).

Example

```
int a = 0, b = 0;
for (i = 0; i <= N; i++){
  a = a + 5;
}
for(j = o; j <= M; j++){
  b = b + 6;
}
```

int a = 0, b = 0;	1 + 1	2	2 assignments
for (i = 0; i <= N; i++) {	1 + (1 + 1) * N	2N + 1	1 assignment, 1 increment 1 comparison done N times.
a = a + 5;	(1 + 1) * N	2N	1 addition 1 assignment done N times.
}			
for (j = 0; j <= M; j++) {	1 + (1 + 1) * M	2M + 1	1 assignment, 1 increment 1 comparison done M times.
b = b + 6;	(1 + 1) * M	2M	1 addition 1 assignment done M times.
}			
		4N + 4M + 4	
		O(N + M)	

Time complexity -> O(N + M)
Space complexity -> O(1). Spaces used by a, b, i, j, N, M are constant with respect to inputs (N, M)