Design and Analysis of Algorithms

Dina El-Manakhly, Ph. D.

dina_almnakhly@science.suez.edu.eg

Lecture 6(13/11/2022)

Asymptotic analysis

- The notation we use to describe the asymptotic running time of an algorithm are defined in terms of functions whose domain are the set of natural numbers.

$$N=\{0,1,2,\dots\dots\dots\}$$

- **Example**:
  - $f(n)= n^2 + n + 5$

    Asymptotic Analysis is $O(n^2)$

  - $f(n)= n^2 + n + 5, \quad g(n)=n^2$

    Asymptotic Analysis is $O(g(n))=O(n^2)$

    $f(n)= O(g(n))$

Three asymptotic notations

There are mainly three asymptotic notations:

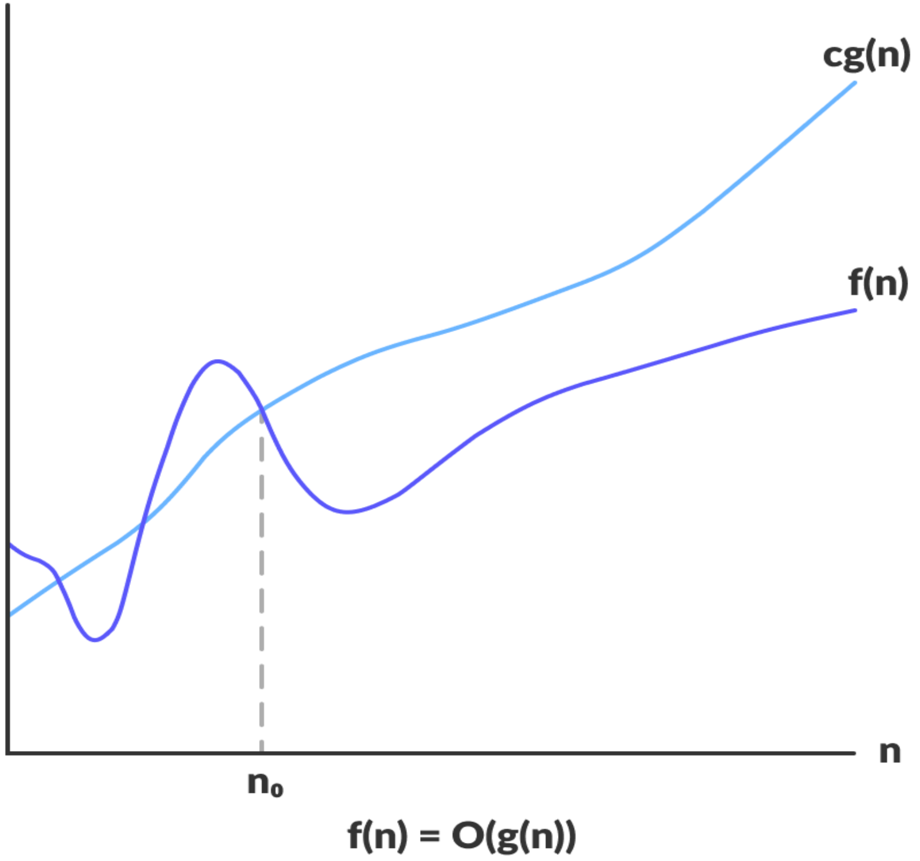
- **Big-O notation** (O(g(n)), Big-Oh of g of n, the Asymptotic Upper Bound)
  - It gives the worst-case complexity of an algorithm. Worst case is amount of time program would take with worst possible input configuration. Worst case is easier to find and we are always interested in the worst-case scenario.
- **Omega notation** (Ω(g(n)), Big-Omega of g of n, the Asymptotic Lower Bound)
  - It gives the best case complexity of an algorithm. Best case is amount of time program would take with best possible input configuration.
- **Theta notation** (Θ(g(n)), Big-Theta of g of n, the Asymptotic Tight Bound)
  - Average case is amount of time a program is expected to take using "typical" input data.

Big-O Notation

- For a given function g(n), we denote by O(g(n)) the set of functions:

> O(g(n)) = { f(n): there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0$ }

- f(n)=O(g(n)) means there exists some constant **c** such that f(n) is always **<= cg(n)** for large enough **n.**
- We use O-notation to give an asymptotic upper bound of a function, to within a constant factor.



Big-O Notation (Example 1)

- **Show that $n^2 + n + 5$ is $O(n^2)$**

$f(n)= n^2 + n + 5, \quad g(n)=n^2$

Thus we want to prove the following

$$f(n) \le cg(n)$$

$$n^2 + n + 5 <= cn^2$$

If c=2

$n^2 + n + 5 <= 2n^2 \quad n>=3$

f(n) is $O(n^2)$

n>=3, c=2

C=2

$n^2 + n + 5 <= 2n^2 \quad n=1$ X

C=2

$n^2 + n + 5 <= 2n^2 \quad n=2$ X

C=2

$n^2 + n + 5 <= 2n^2 \quad n=3$ ✓

Big-O Notation (Example 2)

- **Show that $2n + 3$ is $O(n)$**

$f(n) = 2n+3$, $g(n) = n$

Thus we want to prove the following

$$f(n) \le cg(n)$$
$$2n+3 \le cn$$

If $c=5$

$2n+3 \le 5n$ $n \ge 1$

$f(n)$ is $O(n)$

$n \ge 1$, $c=5$

Big-O Notation (Example 2)

- $2n + 3$ is $O(n)$ ✓
- $2n + 3$ is $O(n^2)$ ✓
- $2n + 3$ is $O(n^3)$ ✓
- $2n + 3$ is $O(n^n)$ ✓
- $2n + 3$ is $O(2^n)$ ✓
- $2n + 3$ is $O(\log n)$ X

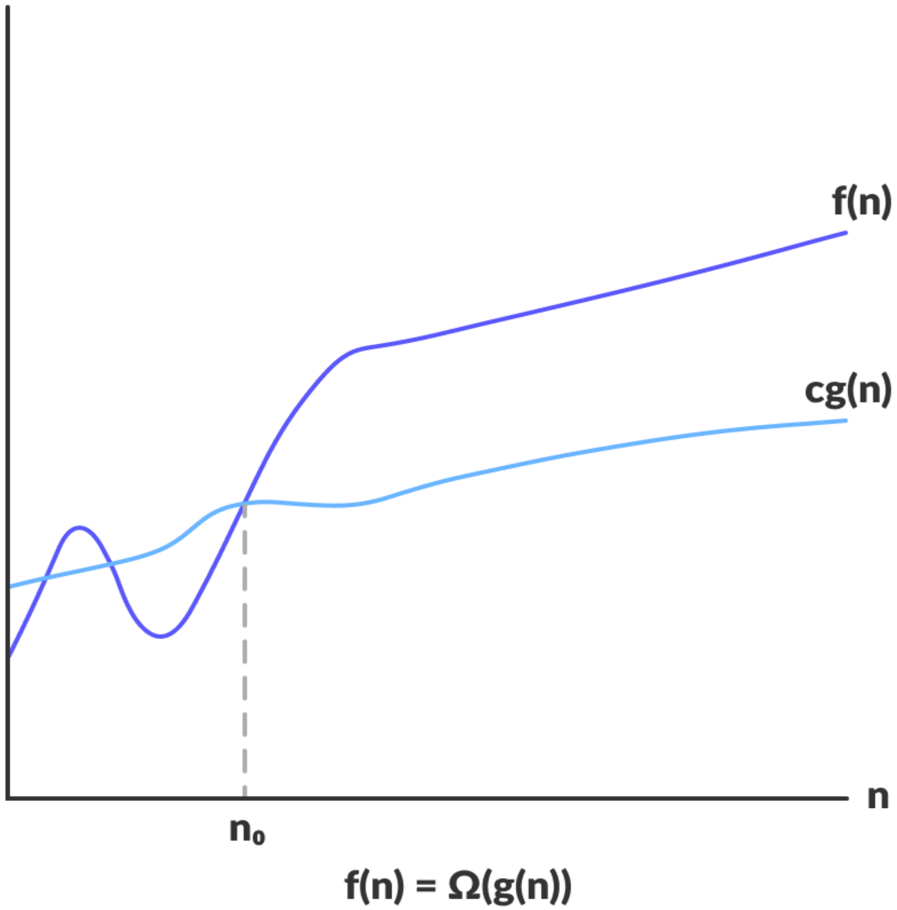
- n^n
- $n!$
- 2^n
- n^2
- $n \log n$
- n
- $\log n$
- 1

Omega Notation (Ω-notation)

- For a given function g(n), we denote by Ω(g(n)) the set of functions:

$$\Omega(g(n)) = \{\, f(n): \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \,\}$$

- f(n)= Ω(g(n)) means that there exists some constant **c** such that f(n) is always **>= cg(n)** for large enough **n.**
- We use Ω-notation to give an asymptotic lower bound of a function, to within a constant factor.



Omega Notation (Example 1)

- **Show that $2n + 3$ is $\Omega(n)$**

$f(n)= 2n+3$, $g(n)=n$

Thus we want to prove the following

$$f(n) >= cg(n)$$
$$2n+3 >= cn$$

If $c=1$

$2n+3 >= n$ $n>=0$

$f(n)$ is $\Omega(n)$

$n>=0$, $c=1$

Omega Notation (Example 1)

- 2n + 3 is $\Omega(n)$ ✓
- 2n + 3 is $\Omega(\log n)$ ✓
- 2n + 3 is $\Omega(1)$ ✓
- 2n + 3 is $\Omega(n^n)$ x

- n^n
- $n!$
- 2^n
- n^2
- $n \log n$
- n
- $\log n$
- 1

Theta Notation (Θ-notation)

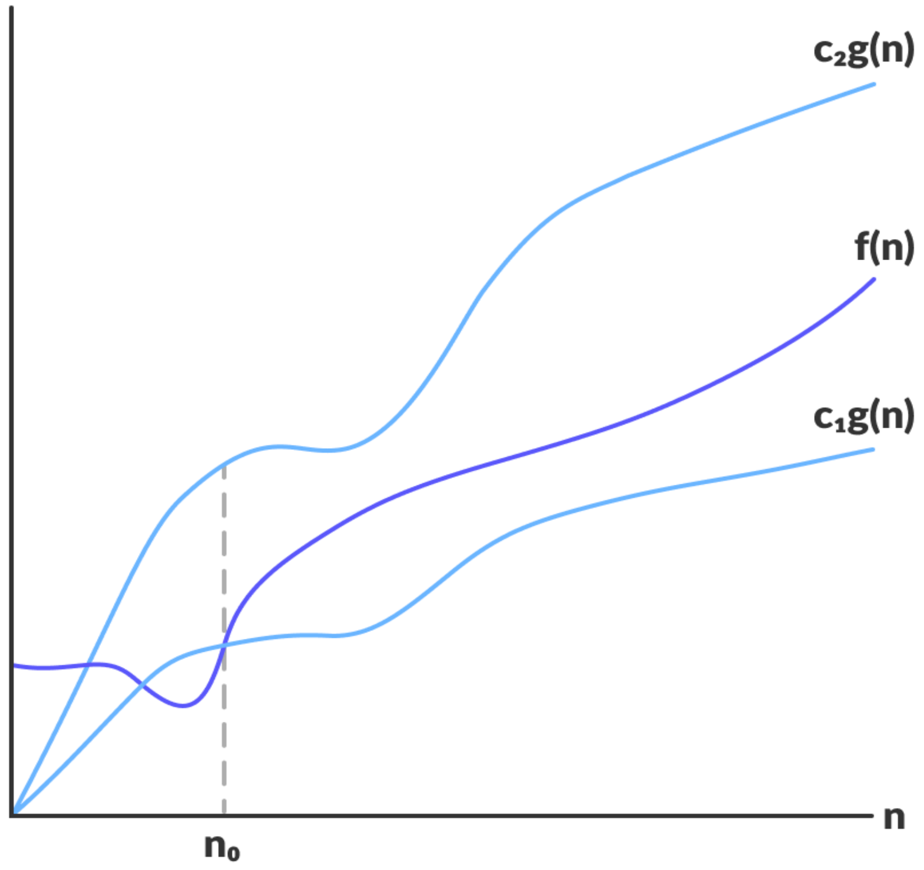
- For a given function g(n), we denote by Θ(g(n)) the set of functions:

$$\Theta(g(n)) = \{\ f(n): \text{there exist positive constants } c_1,\ c_2 \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \quad \text{for all } n \ge n_0\ \}$$

- A function f(n) belongs to the set Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1 g(n)$ and $c_2 g(n)$.

- $f(n)= \Theta(g(n)$

$$c_1 g(n) \le f(n) \le c_2 g(n)$$



f(n) = Θ(g(n))

Theta Notation (Example 1)

Show that $2n+3 = \Theta(n)$

Show that 2n + 3 is O(n)

$f(n) = 2n+3, \quad g(n)=n$

Thus we want to prove the following

$$f(n) \le cg(n)$$

$$2n+3 <= cn$$

If c=5

$2n+3 <= 5n \qquad n>=1$

f(n) is O(n)

n>=1, c=5

Show that 2n + 3 is Ω(n)

$f(n) = 2n+3, \quad g(n)=n$

Thus we want to prove the following

$$f(n) >= cg(n)$$

$$2n+3 >= cn$$

If c=1

$2n+3 >= n \qquad n>=0$

f(n) is $\Omega(n)$

n>=0, c=1

Space Complexity

- Space complexity is the amount of memory space that an algorithm or a problem takes during the execution of that particular problem/algorithm.
- Space complexity is not only calculated by the space used by the variables in the algorithm it also includes and considers the space for input values with it.
- There is a sort of confusion among people between the space complexity and the auxiliary space.

Space Complexity = Auxiliary Space + Space used for input values

The extra space used by an algorithm

Space Complexity (Auxiliary Space + Space used for input values)

```
#Sum Of N Natural Number
int sum(int n)
{
 int i,sum=0;
 for(i=n;i>=1;i--)
 sum=sum+i
 return sum;
}
```

- Input value is 'n'
- Auxiliary space is 'i' and 'sum'

Space Complexity (Auxiliary Space + Space used for input values)

Language C compiler takes the following space:

Type	Size
bool, char, unsigned char, signed char, __int8	1 byte
__int16, short, unsigned short, wchar_t, __wchar_t	2 bytes
float, _int32, int, unsigned int, long, unsigned long	4 bytes
double, __int64, long double, long long	8 bytes

Example 1

```
#Sum Of N Natural Number
int sum(int n)
{
 int i,sum=0;
 for(i=n;i>=1;i--)
 sum=sum+i
 return sum;
}
```

- **sum** variable will take "**4 bytes**" of space.
- **i** variable will also take "**4 bytes**" of space.
- **n** variable will take "**4 bytes**" of space.
- **for loop and return** function these all comes under the **auxiliary space** and lets assume these all will take "**4 bytes**" of space.
- **Total space complexity** is a **4*4=16 bytes**

This is a **fixed complexity** and because of the same variables inputs, such space complexities are considered as **constant space complexities** or called **O(1)** space complexity.

Example 2

```
function sum_of_numbers(arr[],N){
    sum=0
    for(i = 0 to N){
      sum=sum+arr[i]
    }
    print(sum)
}
```

- **array(arr)**, the size of array is "**N**" and each element will take "**4bytes**" so the space taken by "**arr**" will be "**N * 4 bytes**".
- "**sum**" will take "**4 bytes**" of space.
- "**i**" will take "**4 bytes**" of space.
- **function call, initialization of for loop and print function** these all comes under the **auxiliary space** and lets assume these all will take "**4 bytes**" of space.
- **Total space complexity = (4*N + 12) bytes** But these 12 bytes are constant so we will not consider it and after removing all the constants we can finally say that this algorithm has a complexity of "**O(N)**".

Example 3

```
factorial(N){
    int fact=1;
    for (int i=1; i<=N; i++)
    {
      fact*=i;
    }
    return fact;
}
```

- "**Fact**" will take "**4 bytes**" of space.
- "**N**" will take "**4 bytes**" of space.
- "**i**" will take "**4 bytes**" of space.
- **function call, initialization of for loop and return function** these all comes under the auxiliary space and lets assume these all will take "**4 bytes**" of space.
- **Total space complexity= 4*4=16 bytes**
- There is no variable which just constant value(16), it means that this algorithm will take constant space that is "**O(1)**".

Time & Space Complexity of Linear Search

Analysis of Worst Case Time Complexity of Linear Search:

- The worst case will take place if:
  1) The element to be search is in the last index.
  2) The element to be search is not present in the list.

- In both cases, the maximum number of comparisons take place in Linear Search which is equal to N comparisons.

- Hence, the **Worst Case Time Complexity** of Linear Search is **O(N)**.

- Number of Comparisons in Worst Case: N.

Analysis of Worst Case Time Complexity of Linear Search:

Algorithm: Sequential_Search

Input: An array $A=(a_1, a_{2,\ldots,} a_n)$ of n elements and the element k.

Output: return i if the element k equals the element a_i. Otherwise, return 0.

Beg

1. pos=0; i=1; found = false
2. while $i \leq n$ and not found do

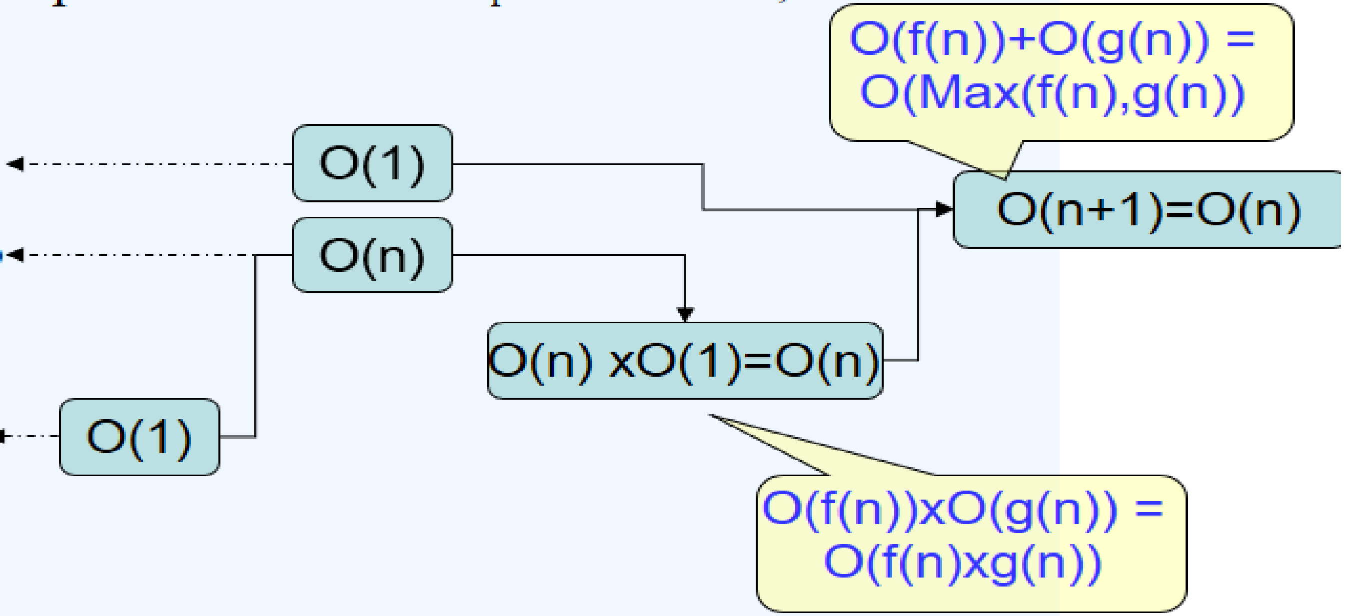
    if $a_i = k$ then

        pos= i

        found=true

    else i=i+1

End.



Analysis of Best Case Time Complexity of Linear Search

- The Best Case will take place if:

  1) The element to be search is on the first index.

- The number of comparisons in this case is 1.

- There force, **Best Case Time Complexity** of Linear Search is $\Omega(1)$.

Analysis of Best Case Time Complexity of Linear Search

Algorithm: Sequential_Search

Input: An array $A=(a_1, a_{2,\ldots,} a_n)$ of n elements and the element k.

Output: return i if the element k equals the element a_i. Otherwise, return 0.

Beg

1. pos=0; i=1; found = false
2. while $i \leq n$ and not found do

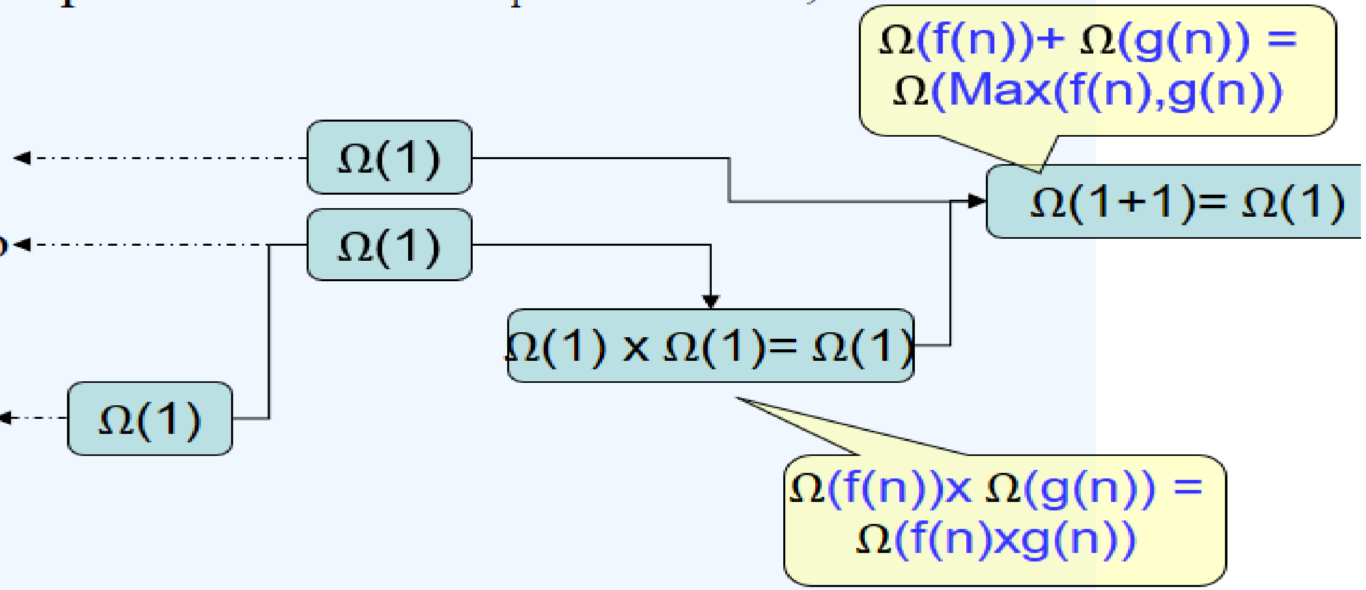
   if a_i= k then

      pos= i

      found=true

   else i=i+1

End.



Analysis of Average Case Time Complexity of Linear Search

Average of all possible cases

We need to find element P

- There are two cases:
  - Case 1: The element P can be in N distinct indexes.
  - Case 2: There will be a case when the element P is not present in the list.
  - There are N case 1 and 1 case 2. So, there are N+1 distinct cases to consider in total.

- Number of comparisons for all cases in case 1 = 1 + 2 + ... + N = $\frac{N * (N+1)}{2}$ comparisons.

  if element P is not in the list, then Linear Search will do N comparisons in case 2.

  Therefore, total number of comparisons for all N+1 cases = $\frac{N * (N+1)}{2} + N = N * (\frac{(N+1)}{2} + 1)$

- Average number of comparisons = $\frac{(N * (\frac{(N+1)}{2} + 1))}{(N+1)} = \frac{N}{2} + \frac{N}{N+1}$

- The dominant term in "Average number of comparisons" is N/2. So, the Average Case Time Complexity of Linear Search is **Θ(N)**.

Complexity Analysis of Linear Search

Cases	When	Time
Worst Case	Worst case occurs when the element k exist at the end of the array or the element k does not exist in the array.	O(n)
Best Case	Best case occurs when the element k exist at the front (beginning) of the array.	Ω(1)
Average Case	Average of all possible cases	ϴ(n)

Space Complexity of Linear Search	**O(1)**