

# Design and Analysis of Algorithms

---

*Dina El-Manakhly, Ph. D.*

*dina\_almnakhly@science.suez.edu.eg*

# 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.

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

- **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  $O(g(n)) = O(n^2)$

# 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** ( $\Omega(g(n))$ ), 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** ( $\Theta(g(n))$ ), Big-Theta of  $g$  of  $n$ , the Asymptotic Tight Bound)

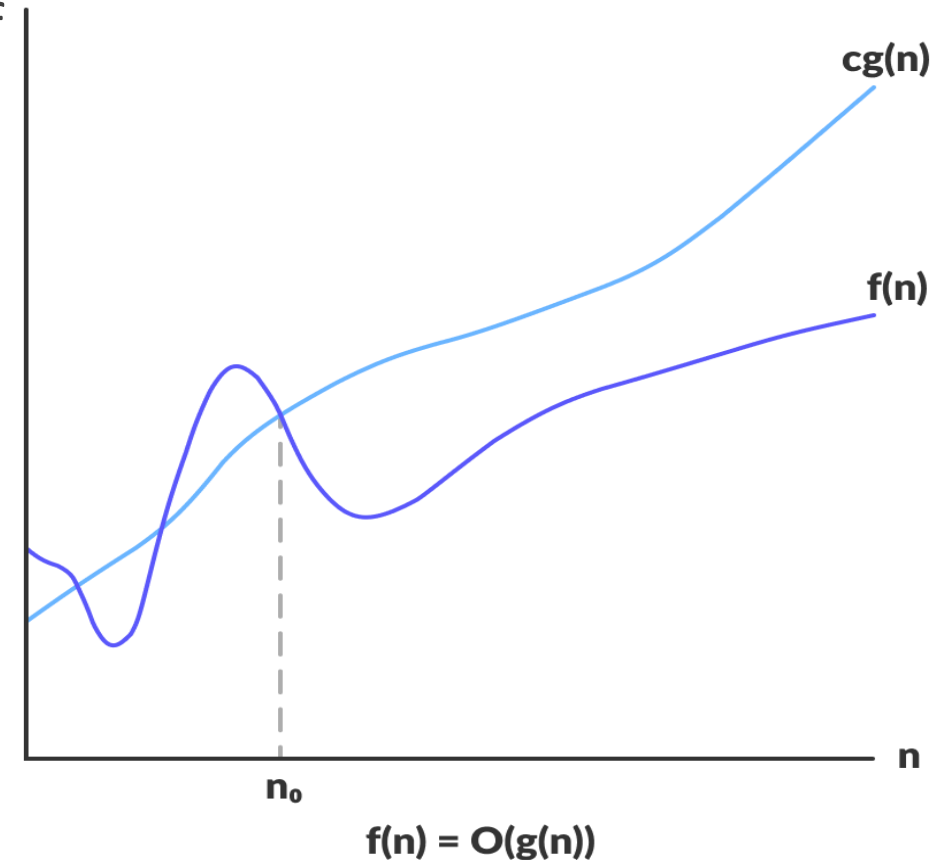
- Average case is amount of time a program is expected to take using "typical" input data.

# Big-O Notation

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

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

- $f(n) = O(g(n))$  means there exists some constant  $c$  such that  $f(n)$  is always  $\leq cg(n)$  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

$$f(n) \leq cg(n) \\ n^2 + n + 5 \leq cn^2$$

If  $c=2$   
 $n^2 + n + 5 \leq 2n^2$       $n \geq 3$

$$f(n) \text{ is } O(n^2) \\ n \geq 3, c=2$$

$$C=2 \\ n^2 + n + 5 \leq 2n^2 \quad n=1 \quad \text{X}$$

$$C=2 \\ n^2 + n + 5 \leq 2n^2 \quad n=2 \quad \text{X}$$

$$C=2 \\ n^2 + n + 5 \leq 2n^2 \quad n=3 \quad \checkmark$$

# Big-O Notation (Example 2)

---

□ Show that  $2n + 3$  is  $O(n)$

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

Thus we want to prove the following

$$f(n) \leq cg(n)$$

$$2n + 3 \leq cn$$

If  $c = 5$

$$2n + 3 \leq 5n \quad n \geq 1$$

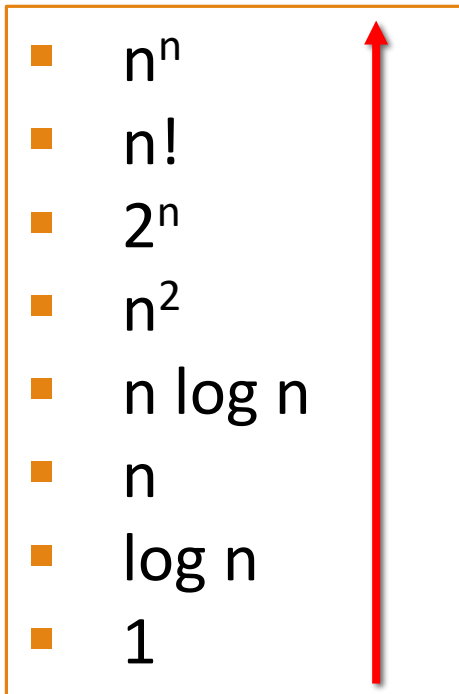
$f(n)$  is  $O(n)$

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

# 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  $O(\log n)$  ✗



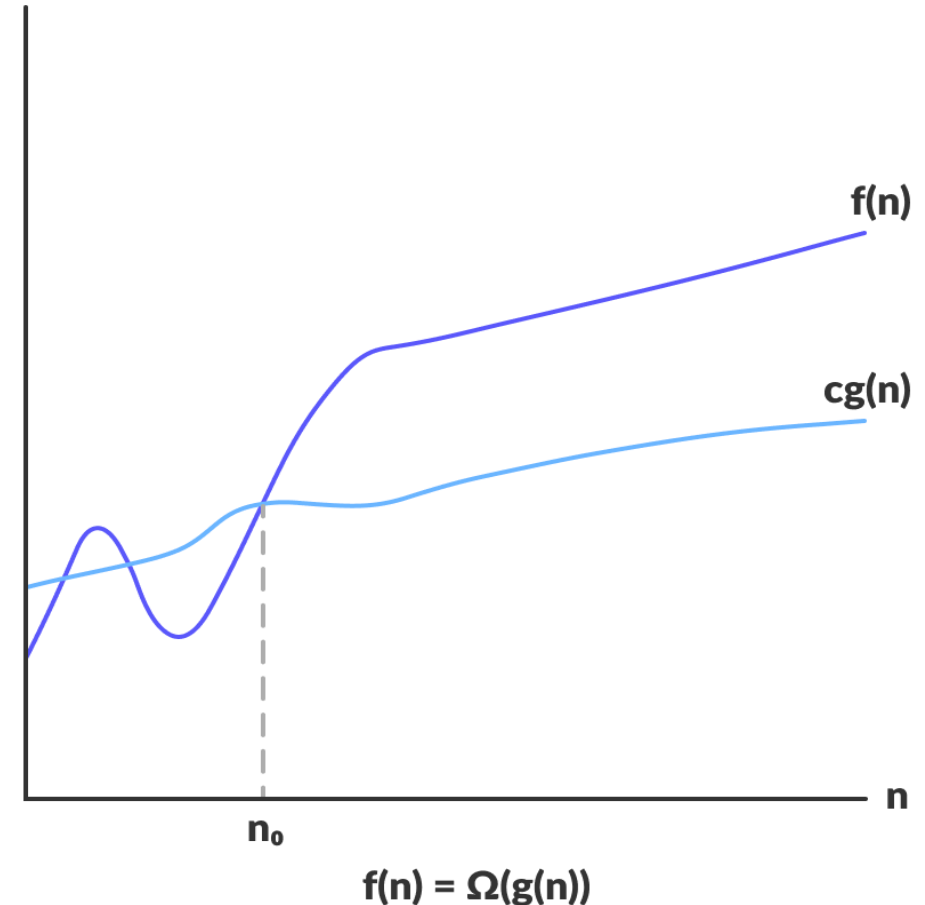
# Omega Notation ( $\Omega$ -notation)

□ For a given function  $g(n)$ , we denote by  $\Omega(g(n))$  the set of functions:

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

□  $f(n) = \Omega(g(n))$  means that there exists some constant  $c$  such that  $f(n)$  is always  $\geq cg(n)$  for large enough  $n$ .

□ We use  $\Omega$ -notation to give an asymptotic lower bound of a function, to within a constant factor.





# Omega Notation (Example 1)

---

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

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

Thus we want to prove the following

$$f(n) \geq cg(n)$$

$$2n + 3 \geq cn$$

If  $c=1$

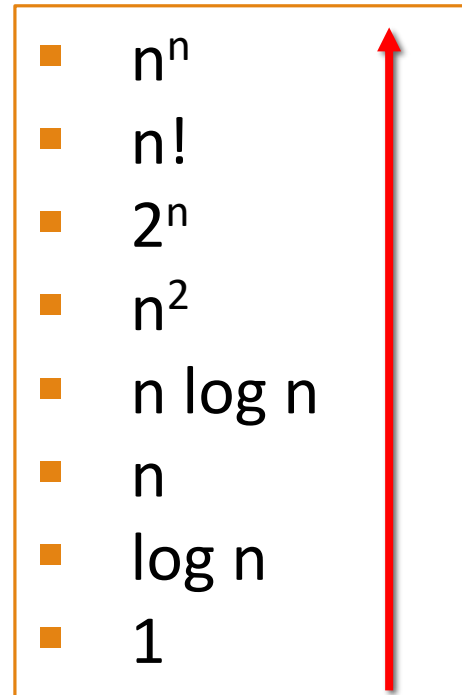
$$2n + 3 \geq n \quad n \geq 0$$

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

$$n \geq 0, c=1$$

# Omega Notation (Example 1)

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



# Theta Notation ( $\Theta$ -notation)

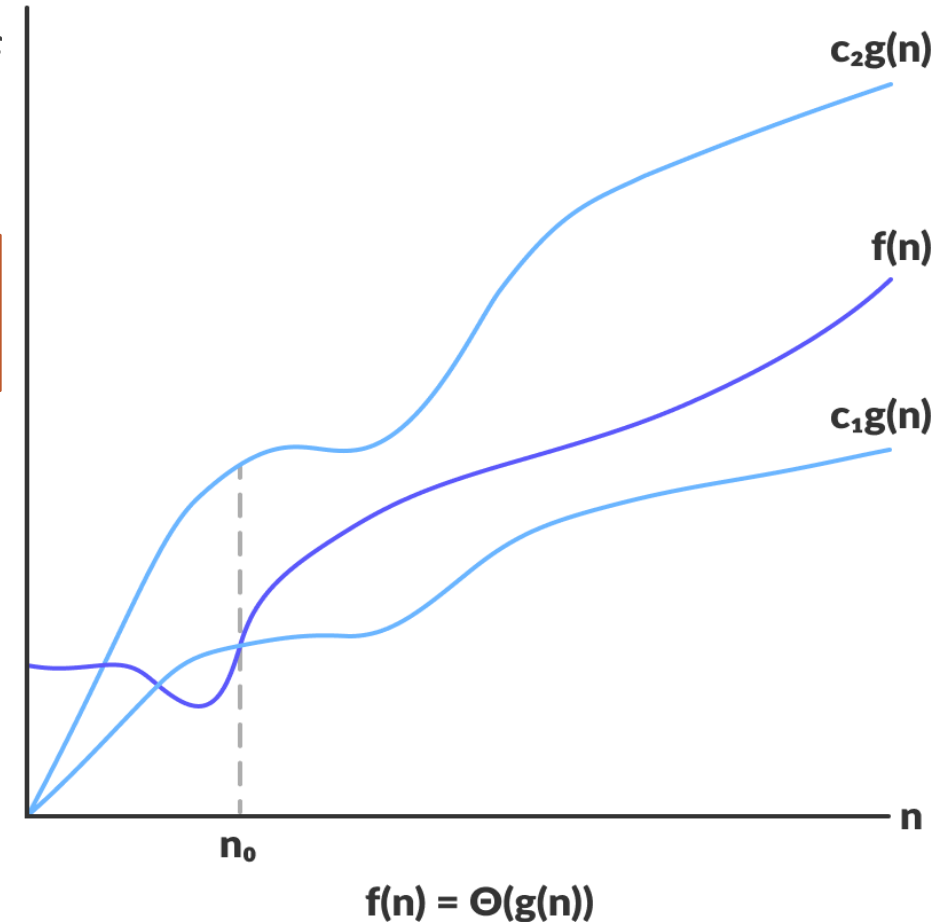
□ For a given function  $g(n)$ , we denote by  $\Theta(g(n))$  the set of functions:

$$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1g(n) \leq f(n) \leq c_2g(n) \text{ for all } n \geq n_0 \}$$

□ A function  $f(n)$  belongs to the set  $\Theta(g(n))$  if there exist positive constants  $c_1$  and  $c_2$  such that it can be sandwiched between  $c_1g(n)$  and  $c_2g(n)$ .

□  $f(n) = \Theta(g(n))$

$$c_1g(n) \leq f(n) \leq c_2g(n)$$



# 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

$$\begin{aligned} f(n) &\leq cg(n) \\ 2n+3 &\leq cn \end{aligned}$$

If  $c=5$

$$2n+3 \leq 5n \quad n \geq 1$$

$f(n)$  is  $O(n)$   
 $n \geq 1, c=5$

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

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

Thus we want to prove the following

$$\begin{aligned} f(n) &\geq cg(n) \\ 2n+3 &\geq cn \end{aligned}$$

If  $c=1$

$$2n+3 \geq n \quad n \geq 0$$

$f(n)$  is  $\Omega(n)$   
 $n \geq 0, c=1$

# 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)**”.

# Strategy

---

- A **Strategy** is an approach or a design for solving a computational problem.

- **Example**

- ☐ Greedy method
- ☐ Dynamic programming
- ☐ Backtracking
- ☐ Branch and bound
- ☐ Brute Force
- ☐ Divide and conquer

# Brute Force

---

- A brute-force algorithm solves a problem in the most **simple, direct way**.
- Brute Force search is the naive approach (**intuitive**).
- A brute force algorithm solves a problem through exhaustion: **it goes through all possible choices until a solution is found**.
- **Example:**

If there is a lock of 4-digit PIN. The digits to be chosen from 0-9 then the brute force will be trying all possible combinations one by one like 0001, 0002, 0003, 0004, and so on until we get the right PIN. In the worst case, it will take 10,000 tries to find the right combination.
- Brute force algorithms **are simple but very slow**.

# Advantages and Disadvantages of Brute Force

---

## ■ Advantages

- Widely Applicable
- Easy (Do not think much to solve it)
- Good for small problems

## ■ Disadvantages

- Often inefficient for large input sizes Because the **complexity** is high.

**Complexity of an algorithm** is a measure of the amount of time and/or space required by an algorithm for an input of a given size ( $n$ ).

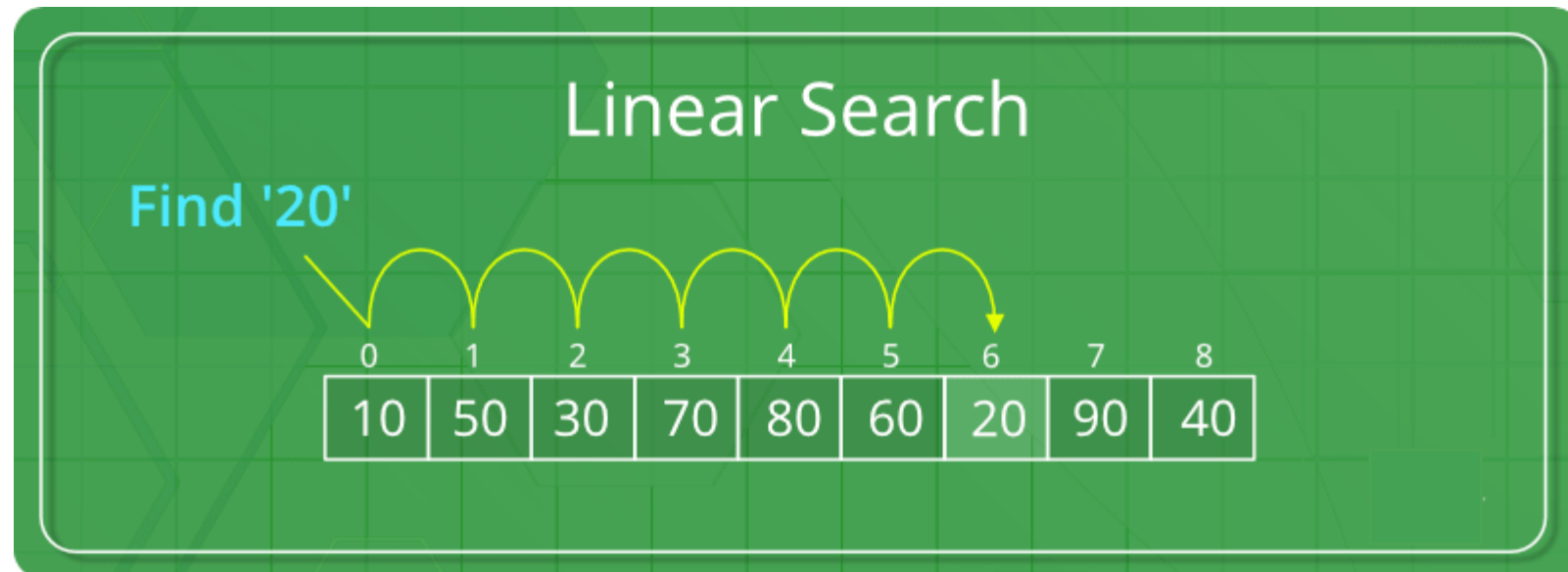
# Some standard algorithms that follow Brute Force algorithm

---

- ❑ Linear Search.
- ❑ Selection Sort.
- ❑ Merging Problem.

# Linear Search (Sequential Search)

- **Linear Search** is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found, otherwise the search continues till the end of the data set. It is the easiest searching algorithm



# Linear Search (Sequential Search)

---

- **Problem Definition:** *Given an array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and an element  $k$ . Find the smallest index,  $pos$ , of an occurrence of the element  $k$  in  $A$  if it exist. Otherwise,  $pos$  is equal to zero.*

## Examples

Example 1: Given  $A=(2,4,9,6,3,10,7,1)$  and  $k=7$  then  $\text{search}(A,k)=7$

Example 2: Given  $A=(2,4,9,6,3,10,7,1)$  and  $k=17$  then  $\text{search}(A,k)=0$

Example 3: Given  $A=(2,4,9,6,9,10,9,1)$  and  $k=9$  then  $\text{search}(A,k)=3$

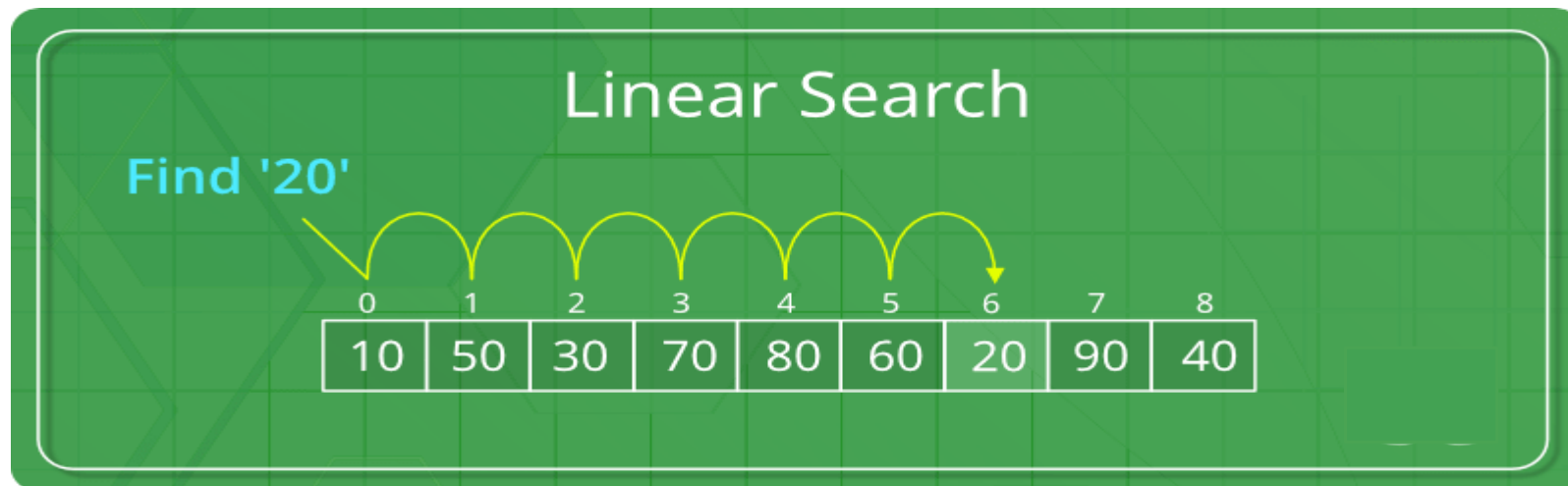


# Linear Search (Sequential Search)

- The **main idea** of sequential search algorithm is scanning the array from the start element to final element in the array.

- In each iteration  $i$ ,  $1 \leq i \leq n$ , we do the following test:

if the element at index  $i$  is equal to the element  $k$  then return  $i$ . **Otherwise** increase  $i$  by one. Until the element  $k$  is found or return  $0$ .



## Pseudo Code

**Algorithm:** Sequential\_Search

**Input:** An array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and the element  $k$ .

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

start

pos=0; i=1; found = false

while  $i \leq n$  and not found do

    if  $a_i = k$  then

        pos= i

        found=true

    else

        i=i+1

end while

return pos

# Tracing

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

34

**Algorithm:** Sequential\_Search

**Input:** An array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and the element  $k$ .

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

start

pos=0; i=1; found = false

while  $i \leq n$  and not found do

    if  $a_i = k$  then

        pos= i

        found=true

    else

        i=i+1

end while

return pos

# Tracing

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

34

$i=1$

Pos=0

Found= false

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

**Algorithm:** Sequential\_Search

**Input:** An array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and the element  $k$ .

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

start

pos=0;  $i=1$ ; found = false

while  $i \leq n$  and not found do

    if  $a_i = k$  then

        pos=  $i$

        found=true

    else

$i=i+1$

end while

return pos

# Tracing

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

34

$i=2$

Pos=0

Found= false

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

**Algorithm:** Sequential\_Search

**Input:** An array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and the element  $k$ .

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

start

pos=0;  $i=1$ ; found = false

while  $i \leq n$  and not found do

    if  $a_i = k$  then

        pos=  $i$

        found=true

    else

$i=i+1$

end while

return pos

# Tracing

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

34

$i=3$

Pos=0

Found= false

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

**Algorithm:** Sequential\_Search

**Input:** An array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and the element  $k$ .

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

start

pos=0;  $i=1$ ; found = false

while  $i \leq n$  and not found do

    if  $a_i = k$  then

        pos=  $i$

        found=true

    else

$i=i+1$

end while

return pos

## Tracing

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

34

$i=4$

Pos=0

Found= false

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

**Algorithm:** Sequential\_Search

**Input:** An array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and the element  $k$ .

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

start

pos=0;  $i=1$ ; found = false

while  $i \leq n$  and not found do

    if  $a_i = k$  then

        pos=  $i$

        found=true

    else

$i=i+1$

end while

return pos

# Tracing

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

34

$i=5$

Pos=0

Found= false

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

**Algorithm:** Sequential\_Search

**Input:** An array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and the element  $k$ .

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

start

pos=0;  $i=1$ ; found = false

while  $i \leq n$  and not found do

    if  $a_i = k$  then

        pos=  $i$

        found=true

    else

$i=i+1$

end while

return pos



# Tracing

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

34

$i=6$

Pos=0

Found= false

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

**Algorithm:** Sequential\_Search

**Input:** An array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and the element  $k$ .

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

start

pos=0;  $i=1$ ; found = false

while  $i \leq n$  and not found do

    if  $a_i = k$  then

        pos=  $i$

        found=true

    else

$i=i+1$

end while

return pos

## Tracing

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

34

$i=7$

Pos=7

Found= true

1	2	3	4	5	6	7	8	9	10
12	4	60	19	3	71	34	66	1	34

**Algorithm:** Sequential\_Search

**Input:** An array  $A=(a_1, a_2, \dots, a_n)$  of  $n$  elements and the element  $k$ .

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

start

pos=0;  $i=1$ ; found = false

while  $i \leq n$  and not found do

    if  $a_i = k$  then

        pos=  $i$

        found=true

    else

$i=i+1$

end while

return pos

# 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  $A=(a_1, a_2, \dots, a_n)$  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

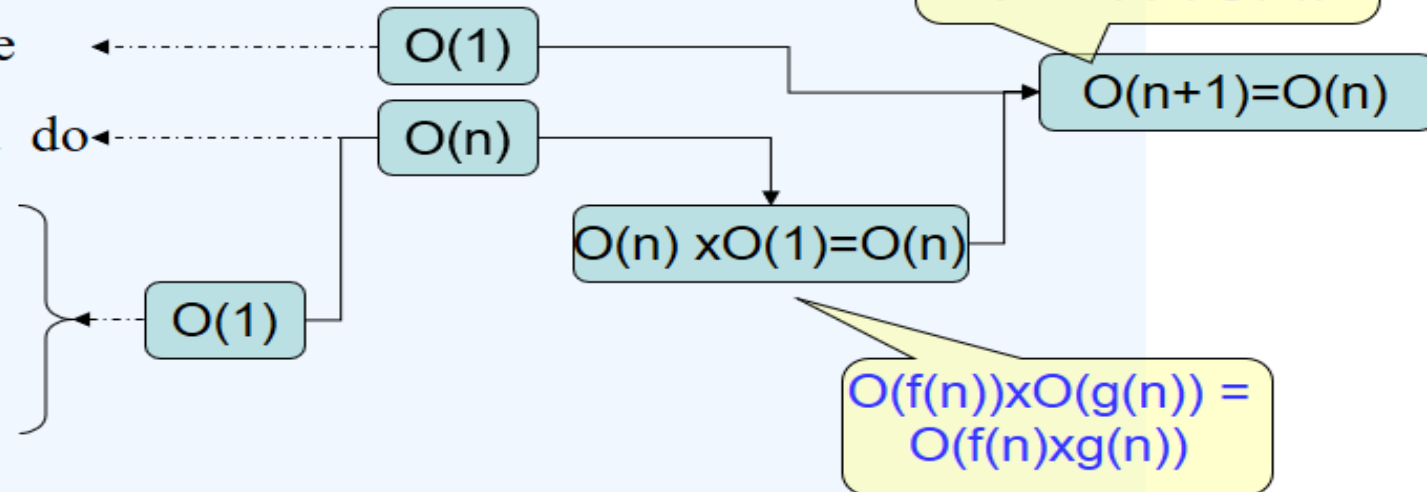
    if  $a_i = k$  then

$pos = i$

$found = true$

    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  $A=(a_1, a_2, \dots, a_n)$  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

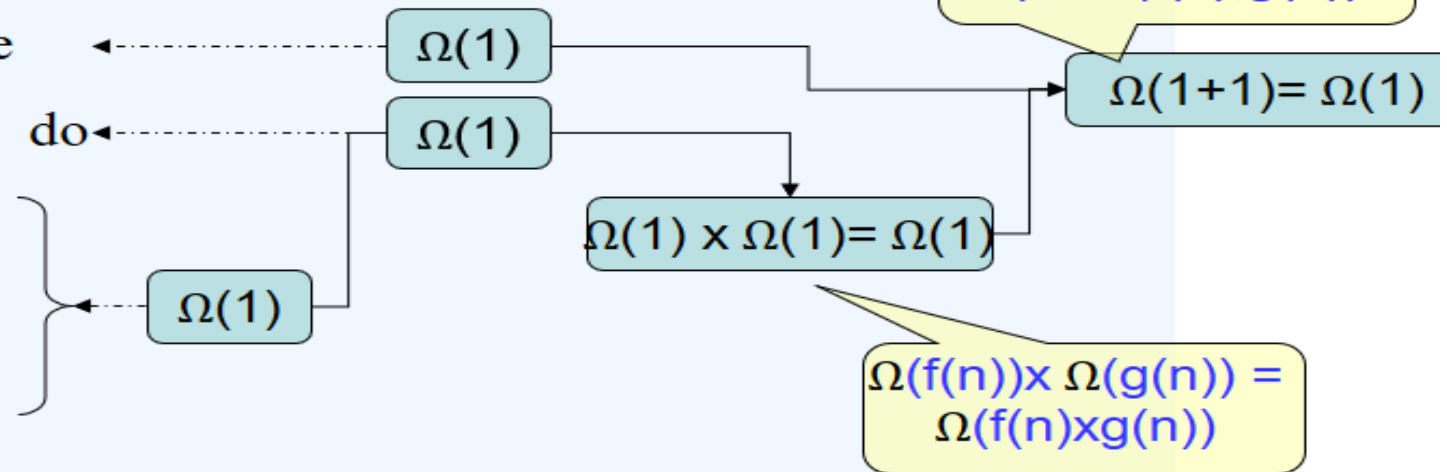
    if  $a_i = k$  then

$pos = i$

$found = true$

    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 =  $1 + 2 + \dots + N = \frac{N * (N+1)}{2}$  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  $\Theta(N)$ .

## 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.	$\Omega(1)$
Average Case	Average of all possible cases	$\Theta(n)$

Space Complexity of Linear Search	$O(1)$
-----------------------------------	--------