Analysis of Algorithm

Introduction

What is Analysis of an Algorithm?

- ✓ Analyzing an algorithm means calculating/predicting the resources that the algorithm requires.
- ✓ Analysis provides theoretical estimation for the required resources of an algorithm to solve a specific computational problem.
- ✓ Two most important resources are computing time (time complexity) and storage space (space complexity).

Why Analysis is required?

- ✓ By analyzing some of the candidate algorithms for a problem, the most efficient one can be easily identified.

Efficiency of Algorithm

- The efficiency of an algorithm is a measure of the amount of resources consumed in solving a problem of size n.
- An algorithm must be analyzed to determine its resource usage.
- Two major computational resources are execution time and memory space.
- Memory Space requirement can not be compared directly, so the important resource is computational time required by an algorithm.
- To measure the efficiency of an algorithm requires to measure its execution time using any of the following approaches:
  1. **Empirical Approach:** To run it and measure how much processor time is needed.
  2. **Theoretical Approach:** Mathematically computing how much time is needed as a function of input size.

How Analysis is Done?

Empirical (posteriori) approach

- Programming different competing techniques & running them on various inputs using computer.
- Implementation of different techniques may be difficult.
- The same hardware and software environments must be used for comparing two algorithms.
- Results may not be indicative of the running time on other inputs not included in the experiment.

Theoretical (priori) approach

- Determining mathematically the resources needed by each algorithm.
- Uses the algorithm instead of an implementation.
- The speed of an algorithm can be determined independent of the hardware/software environment.
- Characterizes running time as a function of the input size n, considers all possible values.

Time Complexity

- Time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- Running time of an algorithm depends upon,
    1. Input Size
    2. Nature of Input
- Generally time grows with the size of input, for example, sorting 100 numbers will take less time than sorting of 10,000 numbers.
- So, running time of an algorithm is usually measured as a function of input size.
- Instead of measuring actual time required in executing each statement in the code, we consider how many times each statement is executed.
- So, in theoretical computation of time complexity, running time is measured in terms of number of steps/primitive operations performed.

Linear Search – Example

Search for 1 in given array

2	9	3	1	8

Comparing **value of i^th^ index** with the given element one by one, until we get the required element or end of the array

Step 1: i=1

2	9	3	1	8

i

$2 \neq 1$

Step 2: i=2

2	9	3	1	8

i

$9 \neq 1$

Step 3: i=3

2	9	3	1	8

i

$3 \neq 1$

Step 4: i=4

2	9	3	1	8

i

Element found at i^{th} index, i=4

Linear Search - Analysis

- The required element in the given array can be found at,

Case 1: element 2 which is at the first position so minimum comparison is required

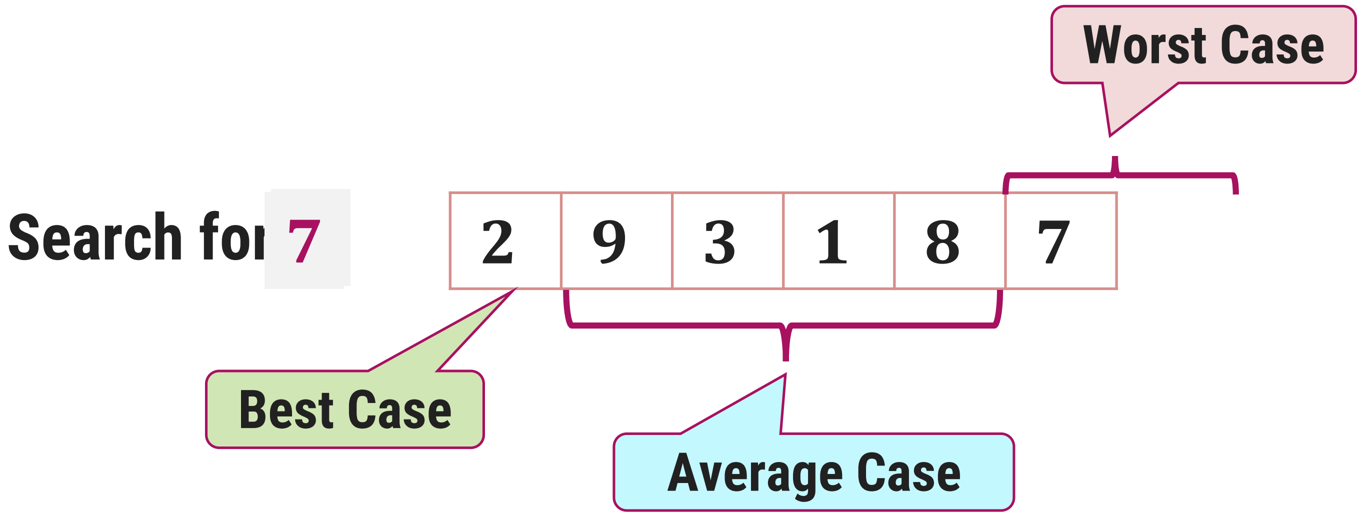
Best Case

Case 2: element 3 anywhere after the first position so, an average number of comparison is required

Average Case

Case 3: element 7 at last position or element does not found at all, maximum comparison is required

Worst Case



Analysis of Algorithm

Best Case	Average Case	Worst Case
Resource usage is minimum	Resource usage is average	Resource usage is maximum
Algorithm's behavior under optimal condition	Algorithm's behavior under random condition	Algorithm's behavior under the worst condition
Minimum number of steps or operations	Average number of steps or operations	Maximum number of steps or operations
Lower bound on running time	Average bound on running time	Upper bound on running time
Generally do not occur in real	Average and worst-case performances are the most used in algorithm analysis.	

Number Sorting - Example

- Suppose you are sorting numbers in Ascending / Increasing order.
- The initial arrangement of given numbers can be in any of the following three orders.

Case 1: Numbers are already in required order, i.e., Ascending order
No change is required

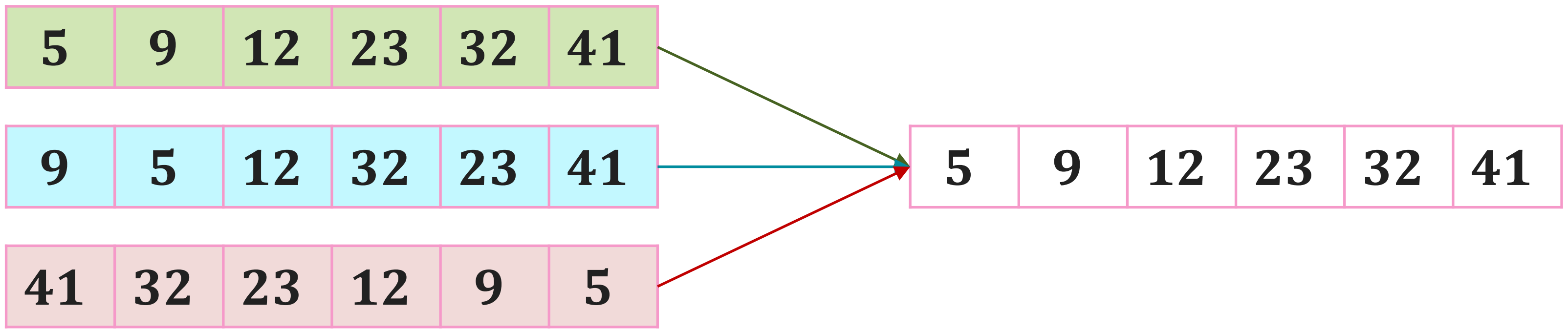
Best Case

Case 2: Numbers are randomly arranged initially. Some numbers will change their position

Average Case

Case 3: Numbers are initially arranged in Descending order so, all numbers will change their position

Worst Case



Best, Average, & Worst Case

Problem	Best Case	Average Case	Worst Case
Linear Search	Element at the first position	Element in any of the middle positions	Element at last position or not present
Book Finder	The first book	Any book in-between	The last book
Sorting	Already sorted	Randomly arranged	Sorted in reverse order

Asymptotic Notations

Introduction

- The theoretical (priori) approach of analyzing an algorithm to measure the efficiency does not depend on the implementation of the algorithm.
- In this approach, the running time of an algorithm is describes as Asymptotic Notations.
- Computing the running time of algorithm’s operations in mathematical units of computation and defining the mathematical formula of its run-time performance is referred to as Asymptotic Analysis.
- An algorithm may not have the same performance for different types of inputs. With the increase in the input size, the performance will change.
- Asymptotic analysis accomplishes the study of change in performance of the algorithm with the change in the order of the input size.
- Using Asymptotic analysis, we can very well define the best case, average case, and worst case scenario of an algorithm.

Asymptotic Notations

- Asymptotic notations are mathematical notations used to represent the time complexity of algorithms for Asymptotic analysis.
- Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.
  1. Ο Notation
  2. Ω Notation
  3. θ Notation
- This is also known as an algorithm's growth rate.
- Asymptotic Notations are used,
  1. To characterize the complexity of an algorithm.
  2. To compare the performance of two or more algorithms solving the same problem.

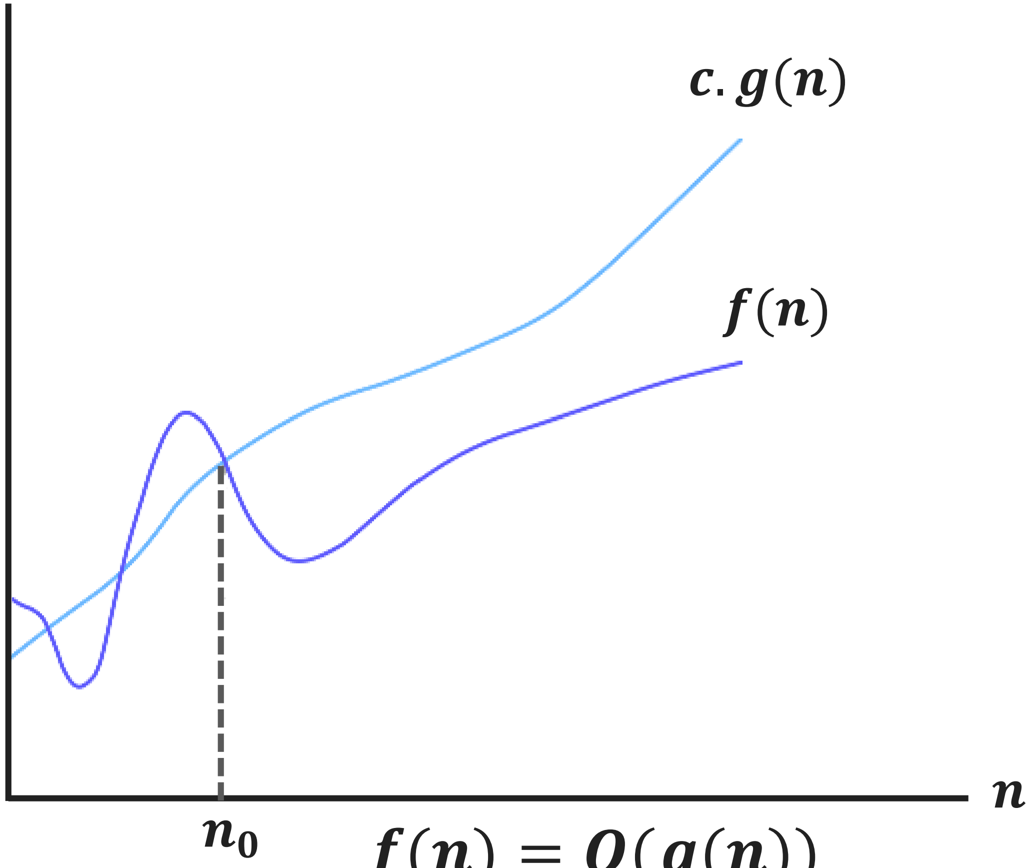
1. O-Notation (Big O notation) (Upper Bound)

- The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time.
- It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.
- 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 $0 \leq f(n) \leq cg(n)$ for all $n_0 \leq n$}

n	$f(n) = n^2$	$g(n) = 2^n$	
1	1	2	$f(n) < g(n)$
2	4	4	$f(n) = g(n)$
3	9	8	$f(n) > g(n)$
4	16	16	$f(n) = g(n)$
5	25	32	$f(n) < g(n)$
6	36	64	$f(n) < g(n)$

Big(O) Notation



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

- $g(n)$ is an asymptotically **upper bound** for $f(n)$.
- $f(n) = O(g(n))$ implies:
  $$f(n) \text{“} \leq \text{”}\, c.g(n)$$
- An upper bound $g(n)$ of an algorithm defines the maximum time required, we can always solve the problem in at most $g(n)$ time.
- Time taken by a known algorithm to solve a problem with worse case input gives the upper bound.

2. Ω-Notation (Omega notation) (Lower Bound)

- Big Omega notation (Ω) is used to define the lower bound of any algorithm or we can say the best case of any algorithm.
- This always indicates the minimum time required for any algorithm for all input values, therefore the best case of any algorithm.
- When a time complexity for any algorithm is represented in the form of big-Ω, it means that the algorithm will take at least this much time to complete it's execution. It can definitely take more time than this too.
- For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions,

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

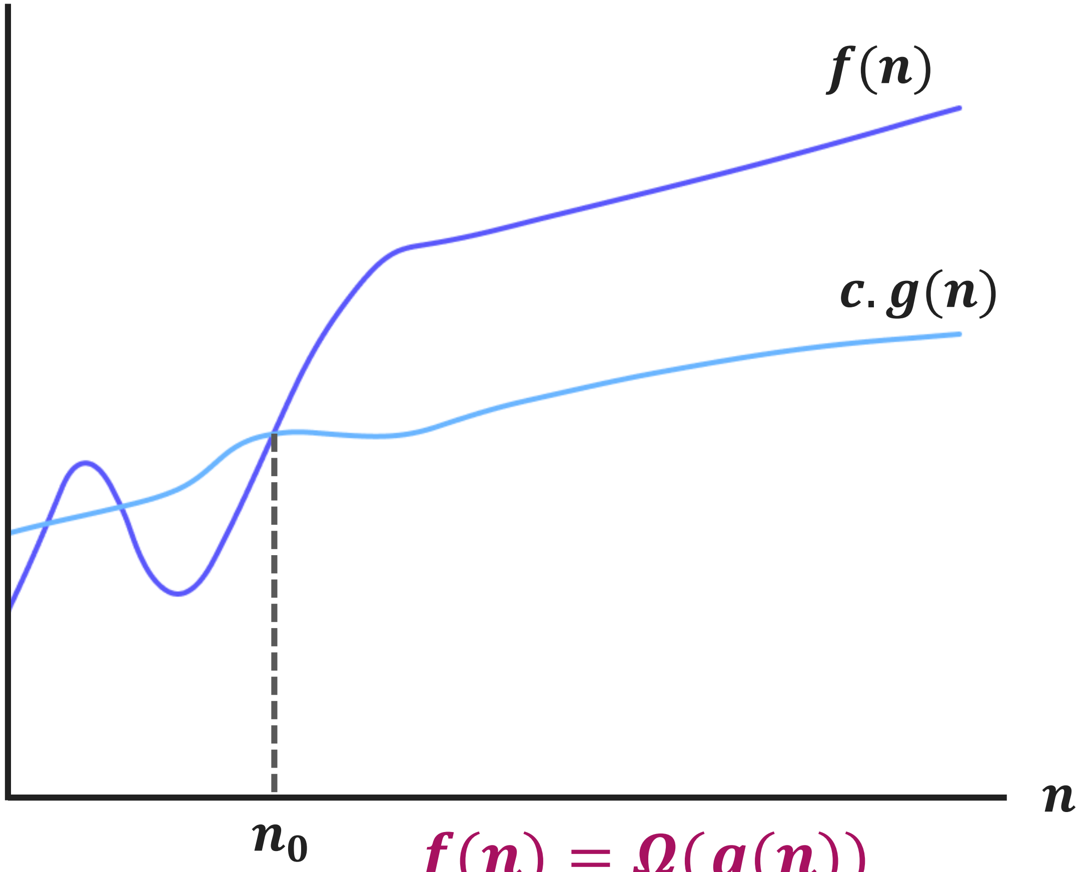
2. Ω-Notation (Omega notation) (Lower Bound)

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

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

n	$g(n) = 2^n$	$f(n) = n^2$	
1	2	1	$f(n) > g(n)$
2	4	4	$f(n) = g(n)$
3	8	9	$f(n) < g(n)$
4	16	16	$f(n) = g(n)$
5	32	25	$f(n) > g(n)$
6	64	36	$f(n) > g(n)$

Big(Ω) Notation



$f(n) = \Omega(g(n))$

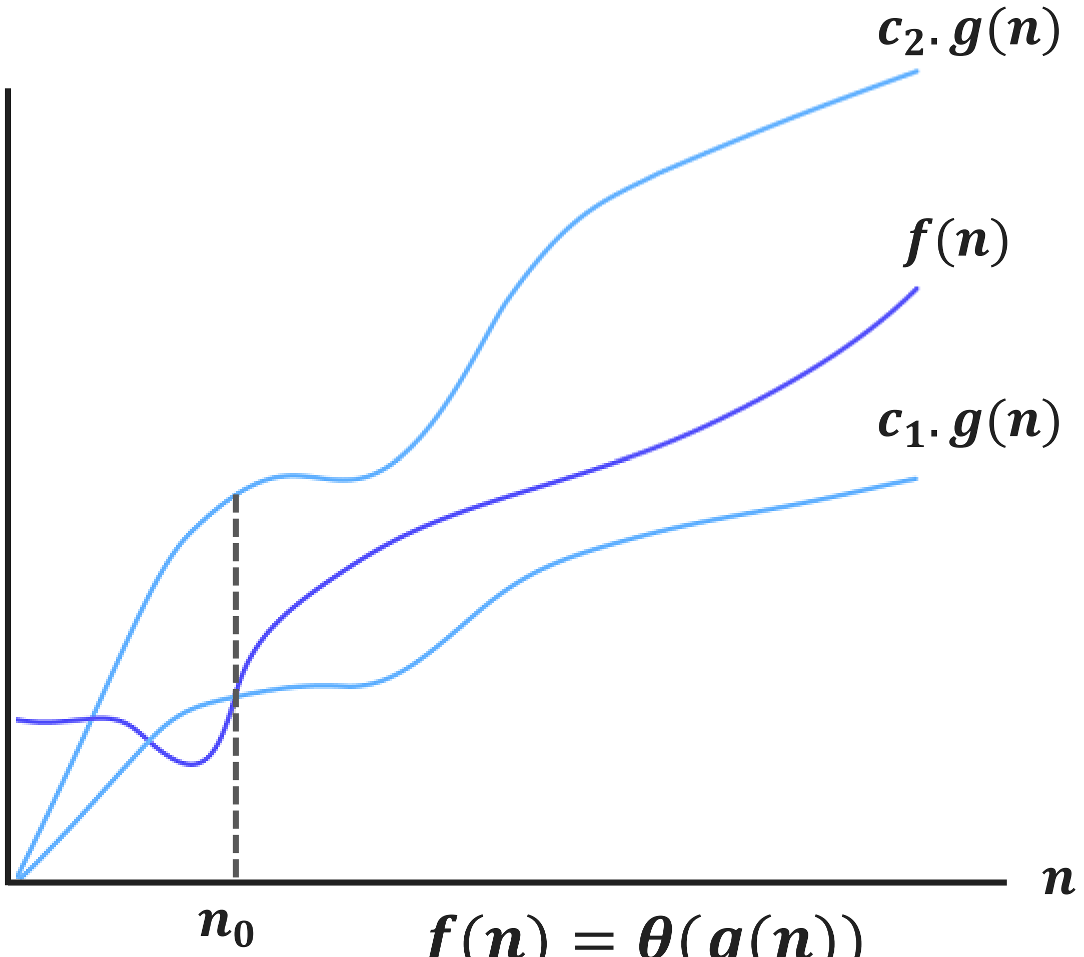
- $g(n)$ is an asymptotically **lower bound** for $f(n)$.
- $f(n) = \Omega(g(n))$ implies:
  $$f(n) \text{“} \geq \text{”}\ c.g(n)$$
- A lower bound $g(n)$ of an algorithm defines the minimum time required, it is not possible to have any other algorithm (for the same problem) whose time complexity is less than $g(n)$ for random input.

3. θ-Notation (Theta notation) (Same order)

- The notation $\theta(n)$ is the formal way to enclose both the lower bound and the upper bound of an algorithm's running time.
- Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average case complexity of an algorithm.
- The time complexity represented by the Big-θ notation is the range within which the actual running time of the algorithm will be.
- So, it defines the exact Asymptotic behavior of an algorithm.
- For a given function $g(n)$, we denote by $\theta(g(n))$ the set of functions,

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

θ-Notation



$f(n) = \theta(g(n))$

- $\theta(g(n))$ is a set, we can write $f(n) \epsilon\, \theta(g(n))$ to indicate that $f(n)$ is a member of $\theta(g(n))$.
- $g(n)$ is an asymptotically tight bound for $f(n)$.
- $f(n) = \theta(g(n))$ implies:

$$f(n)\text{“} = \text{”}\, c.\, g(n)$$

Asymptotic Notations

1. O-Notation (Big O notation) (Upper Bound)

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

$\mathbf{f(n) = O(g(n))}$

2. Ω-Notation (Omega notation) (Lower Bound)

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

$\mathbf{f(n)} = \Omega(\mathbf{g(n)})$

3. θ-Notation (Theta notation) (Same order)

$\theta(g(n))$ = $\{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $\mathbf{0} \leq \boldsymbol{c_1 g}(\boldsymbol{n}) \leq \boldsymbol{f}(\boldsymbol{n}) \leq \boldsymbol{c_2 g}(\boldsymbol{n})$ for all $n_0 \leq n\}$

$\mathbf{f(n) = \theta(g(n))}$

Common Orders of Magnitude

n	$\log n$	$n \log n$	n^2	n^3	2^n	$n!$
4	2	8	16	64	16	24
16	4	64	256	4096	65536	2.09×10^{13}
64	6	384	4096	262144	1.84×10^{19}	1.26×10^{29}
256	8	2048	65536	16777216	1.15×10^{77}	∞
1024	10	10240	1048576	1.07×10^{9}	1.79×10^{308}	∞
4096	12	49152	16777216	6.87×10^{10}	10^{1233}	∞

Asymptotic Notations in Equations

- Consider an example of buying elephants and goldfish:

Cost = cost_of_elephants + cost_of_goldfish

Cost ≈ cost_of_elephants (approximation)

Negligible

- **Maximum Rule**: Let, f, g: $N \rightarrow R^{+}$ the max rule says that:

$$\boxed{O(f(n)+g(n))=O(\max(f(n),g(n)))}$$

1. $n^4 + 100n^2 + 10n + 50$ is $\mathbf{O(n^4)}$
2. $10n^3 + 2n^2$ is $\mathbf{O(n^3)}$
3. $n^3 - n^2$ is $\mathbf{O(n^3)}$

- The low order terms in a function are relatively insignificant for large $\boldsymbol{n}$

$$n^4 + 100n^2 + 10n + 50 \approx n^4$$

Asymptotic Notations

1. O-Notation (Big O notation) (Upper Bound)

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

$\mathbf{f(n) = O(g(n))}$

2. Ω-Notation (Omega notation) (Lower Bound)

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

$\mathbf{f(n)} = \Omega(\mathbf{g(n)})$

3. θ-Notation (Theta notation) (Same order)

$\theta(g(n))$ = $\{f(n)$: there exist positive constants c_1, c_2 and n_0 such that $\mathbf{0} \leq \boldsymbol{c_1 g}(\boldsymbol{n}) \leq \boldsymbol{f}(\boldsymbol{n}) \leq \boldsymbol{c_2 g}(\boldsymbol{n})$ for all $n_0 \leq n\}$

$\mathbf{f(n) = \theta(g(n))}$

Analyzing Control Statements

For Loop

```
# Input     : int A[n], array of n integers
# Output    : Sum of all numbers in array A


Algorithm: int Sum(int A[], int n)
{
    int s=0;
    for (int i=0; i<n; i++)
        s = s + A[i];
    return s;
}
```

1

n+1

n

Total time taken = n+1+n+2 = 2n+3
Time Complexity f(n) = 2n+3

Running Time of Algorithm

- The time complexity of the algorithm is : $\boldsymbol{f(n) = 2 \cdot n + 3}$
- Estimated running time for different values of $\boldsymbol{n}$:

$\boldsymbol{n = 10}$	23 steps
$\boldsymbol{n = 100}$	203 steps
$\boldsymbol{n = 1000}$	2,003 steps
$\boldsymbol{n = 10000}$	20,003 steps

- As n grows, the number of steps grow in linear proportion to $\boldsymbol{n}$ for the given algorithm Sum.
- The dominating term in the function of time complexity is $\boldsymbol{n}$: As n gets large, the +3 becomes insignificant.
- **The time is linear in proportion to $\boldsymbol{n}$.**

Analyzing Control Statements

Example 1:

$$sum = a + b; \quad \mathbf{c}$$

- Statement is executed once only
- So, The execution time $T(n)$ is some constant $\mathbf{c} \approx \boldsymbol{O}(\mathbf{1})$

Example 2:

$$for\ i = 1\ to\ n\ do \quad \boldsymbol{c_1 * (n+1)}$$
$$sum = a + b; \quad \boldsymbol{c_2 * (n)}$$

- Total time is denoted as,

$$\boldsymbol{T(n) = c_1 n + c_1 + c_2 n}$$
$$\boldsymbol{T(n) = n(c_1 + c_2) + c_1 \approx O(n)}$$

Example 3:

$$for\ i = 1\ to\ n\ do \quad \boldsymbol{c_1\ (n+1)}$$
$$for\ j\ =\ 1\ to\ n\ do \quad \boldsymbol{c_2\ n\ (n+1)}$$
$$sum = a + b; \quad \boldsymbol{c_3 * n * n}$$

- Analysis

$$T(n) = c_1(n+1) + c_2 n(n+1) + c_3 n(n)$$
$$T(n) = c_1 n + c_1 + c_2 n^2 + c_2 n + c_3 n^2$$
$$T(n) = n^2(c_2 + c_3) + n(c_1 + c_2) + c_1$$
$$T(n) = an^2 + bn + c$$
$$\boldsymbol{T(n) = O\left(n^2\right)}$$

Sorting Algorithms

- Bubble Sort, Selection Sort, Insertion Sort

Introduction

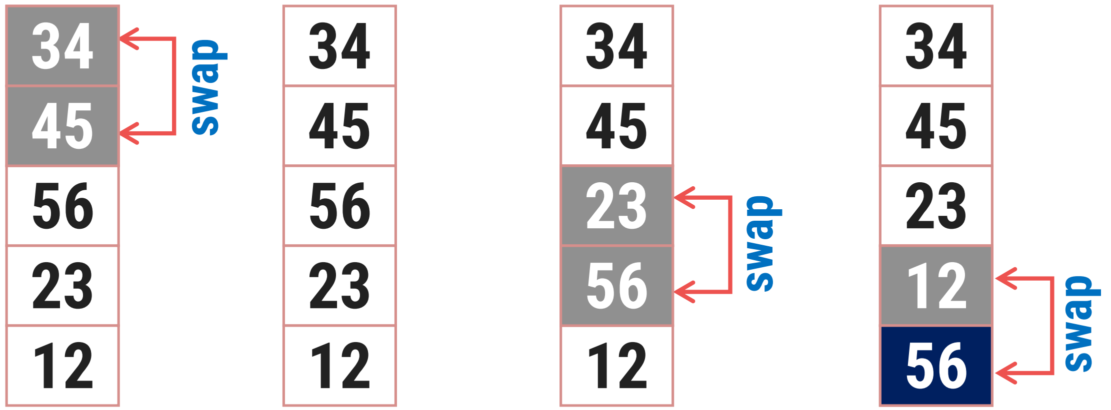
- Sorting is any process of arranging items systematically or arranging items in a sequence ordered by some criterion.
- Applications of Sorting
  1. Phone Bill: the calls made are date wise sorted.
  2. Bank statement or Credit card Bill: transactions made are date wise sorted.
  3. Filling forms online: “select country” drop down box will have the name of countries sorted in Alphabetical order.
  4. Online shopping: the items can be sorted price wise, date wise or relevance wise.
  5. Files or folders on your desktop are sorted date wise.

Bubble Sort – Example

Sort the following array in Ascending order

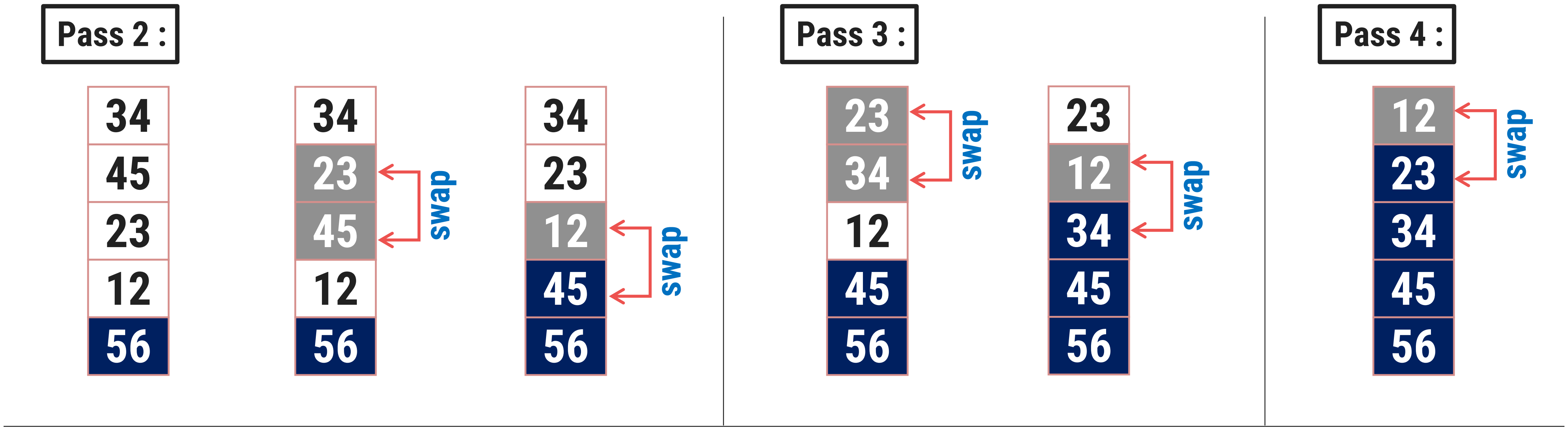
45	34	56	23	12

Pass 1 :



$$if\,(A[j] \;>\; A[j+1])$$
$$swap(A[j], A[j+1])$$

Bubble Sort – Example



$$
\begin{aligned}
&if\,(A[j] \;>\; A[j+1])\\
&\qquad swap(A[j], A[j+1])
\end{aligned}
$$

Bubble Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Bubble_Sort(A)
for i ← 1 to n-1 do
    for j ← 1 to n-i do
      if  A[j] > A[j+1] then
          temp ← A[j]
          A[j] ← A[j+1]
          A[j+1] ← temp
```

$\theta(n)$

$\theta(n^2)$

Bubble Sort

- It is a simple sorting algorithm that works by comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- As it only uses comparisons to operate on elements, it is a comparison sort.
- Although the algorithm is simple, it is too slow for practical use.
- The time complexity of bubble sort is $\boldsymbol{\theta(n^2)}$

Bubble Sort Algorithm – Best Case Analysis

```
# Input: Array A
# Output: Sorted array A
Algorithm: Bubble_Sort(A)
int flag=1;
for i ← 1 to n-1 do
      for j ← 1 to n-i do
            if  A[j] > A[j+1]  then
                  flag = 0;
                  swap(A[j],A[j+1])
      if(flag == 1)
            cout<<"already sorted"<<endl
            break;
```

Condition never becomes true

Pass 1 : i = 1

12
23
34
45
59

j = 1
j = 2
j = 3
j = 4

Best case time complexity = $\theta(n)$

Selection Sort – Example 1

Sort the following elements in Ascending order

5	1	12	-5	16	2	12	14

Step 1 :

Unsorted Array

5	1	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

Step 2 :

Unsorted Array (elements 2 to 8)

-5	1	12	5	16	2	12	14
1	2	3	4	5	6	7	8

Swap

- **Minj** denotes the current index and **Minx** is the value stored at current index.
- **So, Minj = 1, Minx = 5**
- Assume that currently **Minx** is the smallest value.
- Now find the smallest value from the remaining entire Unsorted array.

Index = 4, value = -5

Selection Sort – Example 1

Step 3 :

Unsorted Array (elements 3 to 8)

-5	1	12	5	16	2	12	14
1	2	3	4	5	6	7	8

- **Now Minj = 2, Minx = 1**
- Find min value from remaining unsorted array

Index = 2, value = 1

No Swapping as min value is already at right place

Step 4 :

Unsorted Array (elements 4 to 8)

-5	1	2	5	16	12	12	14
1	2	3	4	5	6	7	8

Swap

- **Minj = 3, Minx = 12**
- Find min value from remaining unsorted array

Index = 6, value = 2

Selection Sort – Example 1

Step 5 :

Unsorted Array
(elements 5 to 8)

-5	1	2	5	16	12	12	14
1	2	3	4	5	6	7	8

- **Now Minj = 4, Minx = 5**
- Find min value from remaining unsorted array

Index = 4, value = 5

No Swapping as min value is already at right place

Step 6 :

Unsorted Array
(elements 6 to 8)

-5	1	2	5	12	16	12	14
1	2	3	4	5	6	7	8

Swap

- **Minj = 5, Minx = 16**
- Find min value from remaining unsorted array

Index = 6, value = 12

Selection Sort – Example 1

Step 7 :

Unsorted Array
(elements 7 to 8)

-5	1	2	5	12	12	16	14
1	2	3	4	5	6	7	8

Swap

- **Now Minj = 6, Minx = 16**
- Find min value from remaining unsorted array

Index = 7, value = 12

Step 8 :

Unsorted Array
(element 8)

-5	1	2	5	12	12	14	16
1	2	3	4	5	6	7	8

Swap

- **Minj = 7, Minx = 16**
- Find min value from remaining unsorted array

Index = 8, value = 14

The entire array is sorted now.

Selection Sort

- Selection sort divides the array or list into two parts,
  1. The sorted part at the left end
  2. and the unsorted part at the right end.
- Initially, the sorted part is empty and the unsorted part is the entire list.
- The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array.
- Then it finds the second smallest element and exchanges it with the element in the second leftmost position.
- This process continues until the entire array is sorted.
- The time complexity of selection sort is $\boldsymbol{\theta(n^2)}$

Selection Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Selection_Sort(A)
for i ← 1 to n-1 do
     minj ← i;
     minx ← A[i];
     for j ← i + 1 to n do
          if A[j] < minx then
               minj ← j;
               minx ← A[j];
     A[minj] ← A[i];
     A[i] ← minx;
```

$\theta(n)$

$\theta(n^2)$

Selection Sort – Example 2

```
Algorithm: Selection_Sort(A)
for i ← 1 to n-1 do
    minj ← i; minx ← A[i];
    for j ← i + 1 to n do
        if A[j] < minx then
            minj ← j ; minx ← A[j];
    A[minj] ← A[i];
    A[i] ← minx;
```

Pass 1 :

i = 1

minj ← 2

minx ← 34 No Change

j = 2 3

A[j] = 56

Sort in Ascending order

45	34	56	23	12
1	2	3	4	5

Selection Sort – Example 2

```
Algorithm: Selection_Sort(A)
for i ← 1 to n-1 do
    minj ← i; minx ← A[i];
    for j ← i + 1 to n do
        if A[j] < minx then
            minj ← j ; minx ← A[j];
    A[minj] ← A[i];
    A[i] ← minx;
```

Pass 1 :

i = 1

minj ← 5

minx ← 12

j = 2 3 4 5

A[j] = 12

Sort in Ascending order

45	34	56	23	12
1	2	3	4	5

Swap

Unsorted Array

12	34	56	23	45
1	2	3	4	5

12	23	34	45	56

Insertion Sort – Example

Sort the following elements in Ascending order

5	1	12	-5	16	2	12	14

Step 1 :

Unsorted Array

5	1	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

Step 2 :

j

5 1	1	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

Shift down

$\boldsymbol{i = 2, x = 1}$ $\boldsymbol{j = i - 1 \; and \; j > 0}$

while $x < T[j]$ do

$T[j+1] \leftarrow T[j]$

$j--$

Insertion Sort – Example

Step 3 :

	j						
1	5	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

$i = 3, x = 12$ | $j = i - 1$ and $j > 0$

while $x < T[j]$ do
$T[j+1] \leftarrow T[j]$
$j--$

No Shift will take place

Step 4 :

j		j					
1 -5	5	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

Shift down
Shift down
Shift down

$i = 4, x = -5$ | $j = i - 1$ and $j > 0$

while $x < T[j]$ do
$T[j+1] \leftarrow T[j]$
$j--$

Insertion Sort – Example

Step 5 :

j

-5	1	5	12	16	2	12	14
1	2	3	4	5	6	7	8

No Shift will take place

$i = 5, x = 16$ | $j = i - 1 \; and \; j > 0$

while $x < T[j]$ do
$T[j+1] \leftarrow T[j]$
$j--$

Step 6 :

$i = 6, x = 2$ | $j = i - 1 \; and \; j > 0$

j j

-5	1	5 2	12	16	2	12	14
1	2	3	4	5	6	7	8

Shift down Shift down Shift down

while $x < T[j]$ do
$T[j+1] \leftarrow T[j]$
$j--$

Insertion Sort – Example

Step 7 :

					j		
-5	1	2	5	12	12	12	14
1	2	3	4	5	6	7	8

Shift down

$i = 7, x = 12$ | $j = i - 1$ *and* $j > 0$

while $\boxed{x < T[j]}$ do
$T[j+1] \leftarrow T[j]$
$j--$

Step 8 :

$i = 8, x = 14$ | $j = i - 1$ *and* $j > 0$

while $\boxed{x < T[j]}$ do
$T[j+1] \leftarrow T[j]$
$j--$

						j	
-5	1	2	5	12	12	14	14
1	2	3	4	5	6	7	8

Shift down

The entire array is sorted now.

Insertion Sort - Algorithm

```
# Input: Array T
# Output: Sorted array T

Algorithm: Insertion_Sort(T[1,…,n])
for i ← 2 to n do
     x ← T[i];
     j ← i - 1;
     while x < T[j] and j > 0 do
          T[j+1] ← T[j];
          j ← j - 1;
     T[j+1] ← x;
```

$\theta(n)$

$\theta(n^2)$

Insertion Sort Algorithm – Best Case Analysis

```
# Input: Array T
# Output: Sorted array T

Algorithm: Insertion_Sort(T[1,…,n])
for i ← 2 to n do
      x ← T[i];
      j ← i – 1;
      while x < T[j] and j > 0 do
            T[j+1] ← T[j];
            j ← j – 1;
      T[j+1] ← x;
```

$\theta(n)$

Pass 1 :

12			
23	i=2	x=23	T[j]=12
34	i=3	x=34	T[j]=23
45	i=4	x=45	T[j]=34
59	i=5	x=59	T[j]=45

The best case time complexity of Insertion sort is $\theta(n)$
The average and worst case time complexity of Insertion sort is $\theta(n^2)$

Thank You