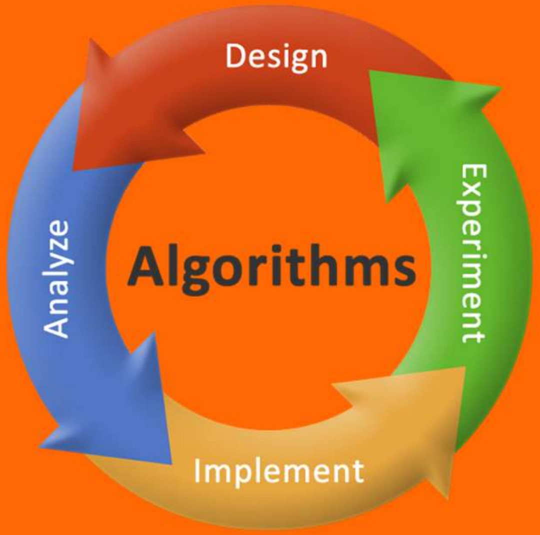
Unit – 1
Analysis of Algorithms



Topics to be covered

- The efficient algorithm
- Average, Best and Worst case analysis
- Asymptotic Notations
- Analyzing control statement
- Sorting Algorithms and analysis: Bubble sort, Selection sort, Insertion sort, Shell sort and Heap sort
- Sorting in linear time : Bucket sort, Radix sort and Counting sort
- Amortized analysis

Analysis of Algorithm

Analysis of Algorithm

- What is the analysis of an algorithm?
  - Analyzing an algorithm means **calculating/predicting the resources** that the algorithm requires.
  - 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.

How Analysis is Done?

1. 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.

2. 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 $\boldsymbol{n}$, considers all possible values.

Problem & Instance

- **Instance**: An Instance of a problem consists of the input needed to compute the solution to the problem.
- Example:

  Problem: to multiply two positive numbers

  Instance: $981\ X\ 1234$
- Instance size: Any integer (generally $\boldsymbol{n}$) that in some way measures the **number of components** in an instance.
- Examples:
  1. Sorting problem: Instance size is **number of elements** to be sorted.
  2. Graph problem: Instance size is **number of nodes or edges** or both.

Efficiency of Algorithm

- The efficiency of an algorithm is a measure of the amount of **resources consumed** in solving a problem of size n.
- The important resource is time, i.e., time complexity
- To measure the efficiency of an algorithm requires **to measure its time complexity** using any of the following approaches:
  1. To run it and measure how much processor time is needed. Empirical Approach
  2. Mathematically computing how much time is needed as a function of input size.

Theoretical Approach

Efficiency of Algorithm

- 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.
- *In theoretical computation of time complexity, Running time is measured in terms of number of steps/primitive operations performed.*

Linear Search

- Suppose, you are given a jar containing some business cards.
- You are asked to determine whether the name “Mukesh Ambani" is in the jar.
- To do this, you decide to simply go through all the cards one by one.
- How long this takes?
- Can be determined by how many cards are in the jar, i.e., Size of Input.

Linear Search

- Linear search is a method for **finding a particular value** from the given list.
- The algorithm checks each element, **one at a time** and in sequence, until the desired element is found.
- Linear search is the simplest search algorithm.
- It is a special case of **brute-force search**.

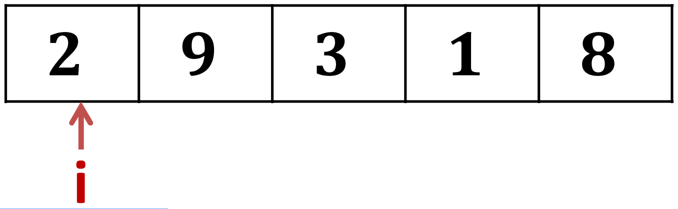
Linear Search - Example

Search for 1 in given array

2	9	3	1	8

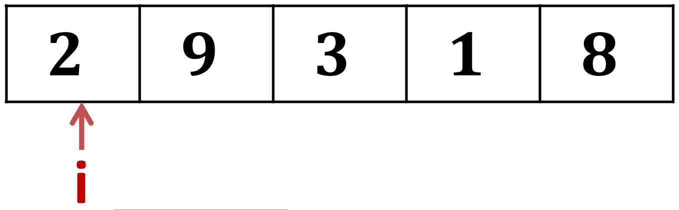
Comparing **value of ith index** with the given element one by one, until we get the required element or end of the array

Step 1: i=0



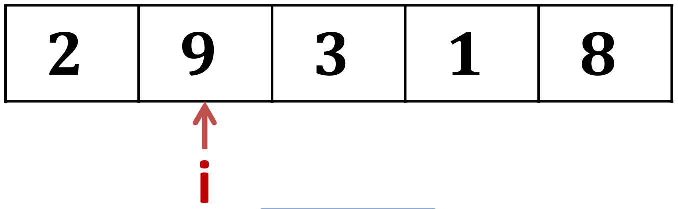
$2 \neq 1$

Step 2: i=1



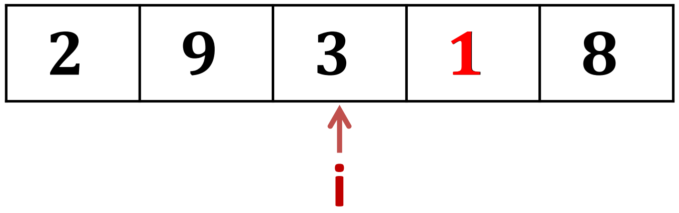
$9 \neq 1$

Step 3: i=2



$3 \neq 1$

Step 4: i=3



Element found at ith index, i=3

Linear Search Algorithm

```
# Input    : Array A, element
# Output   : First index of element in A or
             -1 if not found

Algorithm: Linear_Search
for i = 1 to last index of A:
      if A[i] equals element:
            return i
return -1
```

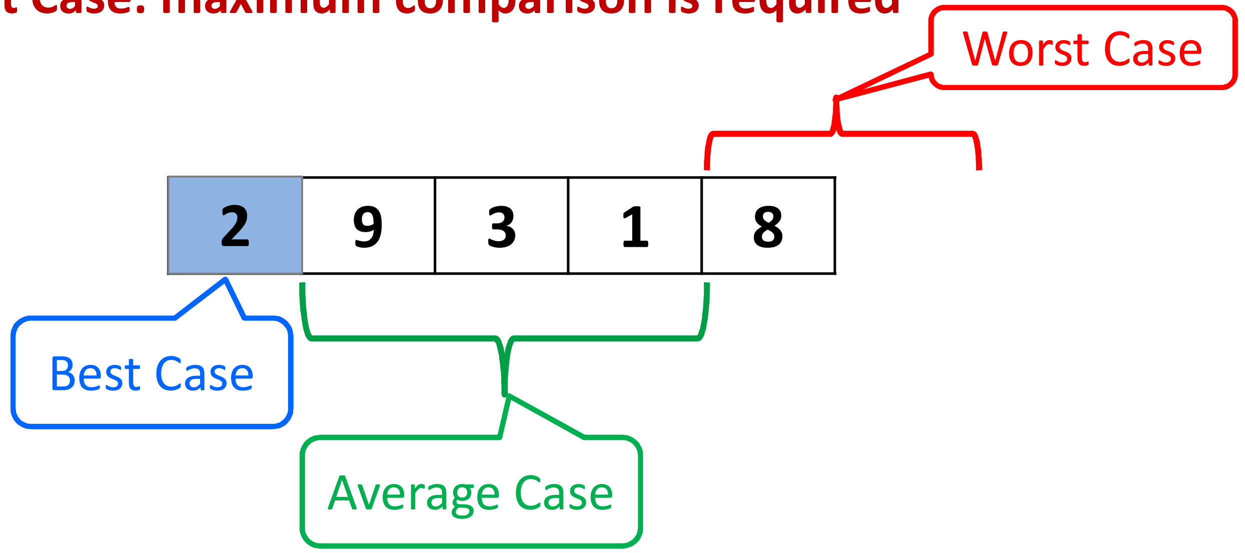
Linear Search

- The required element in the given array can be found at,
  1. E.g. 2: It is at the first position

     Best Case: minimum comparison is required
  2. E.g. 3 or 1: Anywhere after the first position

     Average Case: average number of comparison is required
  3. E.g. 8 or 7: Last position or does not found at all

     Worst Case: maximum comparison is required



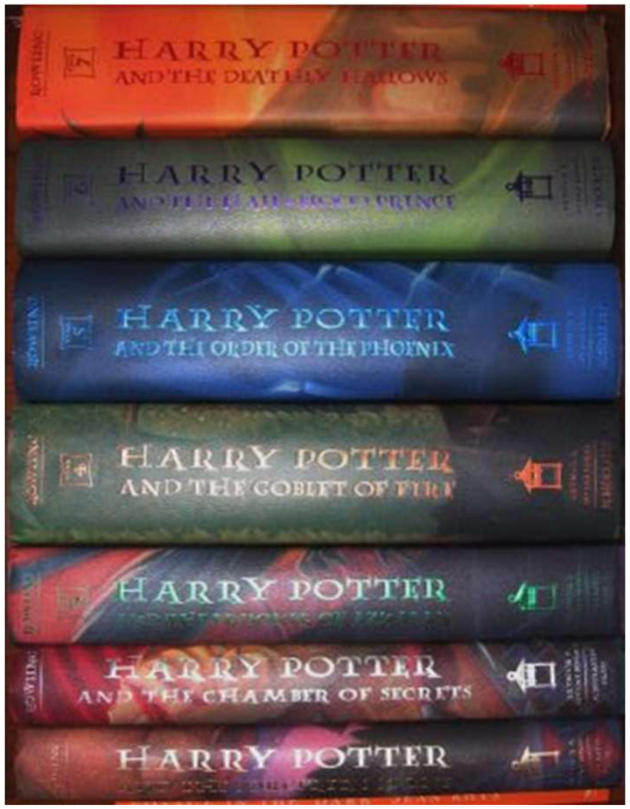
Analysis of Algorithm

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.**	

Asymptotic Notations

Book Finder

- Suppose, you are writing a program **to find a book** from the shelf.
- For any required book, it will start checking books one by one from the bottom.
- If you wanted Harry Potter 3, it would only take 3 actions (or tries) because it’s the third book in the sequence.
- If Harry Potter 7 — it’s the last book so it would have to check all 7 books.
- What if there are total 10 books? How about 10,00,000 books? It would take 1 million tries.


HARRY POTTER AND THE DEATHLY HALLOWS
HARRY POTTER AND THE HALF-BLOOD PRINCE
HARRY POTTER AND THE ORDER OF THE PHOENIX
HARRY POTTER AND THE GOBLET OF FIRE
HARRY POTTER
HARRY POTTER AND THE CHAMBER OF SECRETS
HARRY POTTER

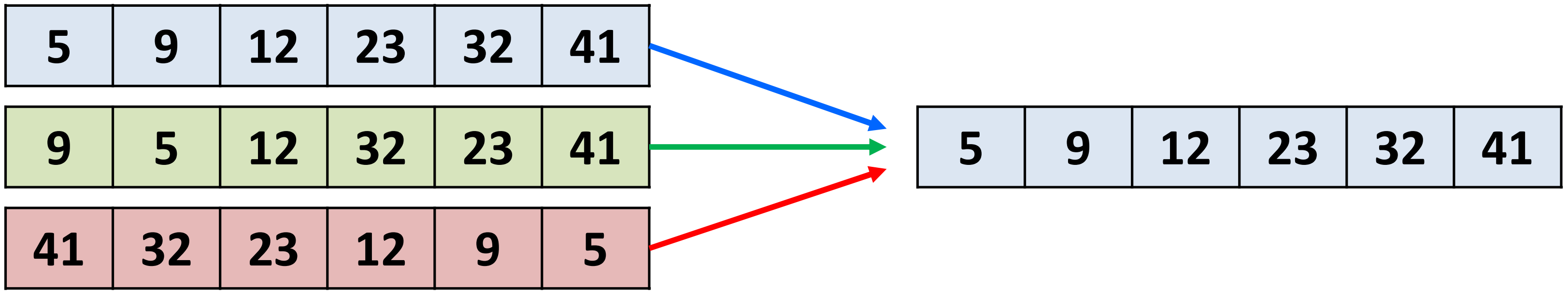
Number Sorting

- Suppose you are sorting numbers in **Ascending / Increasing order**.
- The initial arrangement of given numbers can be in **any of the following** three orders.
  1. Numbers are already in required order, i.e., Ascending order

     No change is required – Best Case
  2. Numbers are randomly arranged initially.

     Some numbers will change their position – Average Case
  3. Numbers are initially arranged in Descending or Decreasing order.

     All numbers will change their position – Worst Case



Analysis – Best Case

1. Linear search
   1. Element at the first position
   2. Element in any of the middle position
   3. Element at last position or not present
2. Finding book from shelf
   1. The first book
   2. Any book in-between
   3. The last book
3. Sorting numbers
   1. Already sorted
   2. Randomly arranged
   3. Sorted in reverse order

Best Case:

- Minimum number of search / swap is required.
- Minimum time is consumed to find solution.
- Known as the LOWER BOUND of running time.
- Running time is denoted as Ω (Omega) - Notation

Analysis – Average Case

1. Linear search
   1. Element at the first position
   2. Element in any of the middle position
   3. Element at last position or not present
2. Finding book from shelf
   1. The first book
   2. Any book in-between
   3. The last book
3. Sorting numbers
   1. Already sorted
   2. Randomly arranged
   3. Sorted in reverse order

Average Case:

- Average number of search / swap is required.
- Average time is consumed to find solution.
- Known as the TIGHT BOUND of running time.
- Running time is denoted using θ (Theta) - Notation

Analysis – Worst Case

1. Linear search
    1. Element at first position
    2. Element in any of the middle position
    3. Element at last position or not present
2. Finding book from shelf
    1. The first book
    2. Any book in-between
    3. The last book
3. Sorting numbers
    1. Already sorted
    2. Randomly arranged
    3. Sorted in reverse order

Worst Case:

- Maximum number of search / swap is required.
- Maximum time is consumed to find solution.
- Known as the UPPER BOUND of running time.
- Running time is denoted using O-Notation (Big O notation)

Asymptotic Notations

- Asymptotic Notations (**Big O, θ - Theta and Ω - Omega**) allow us to analyze an algorithm’s running time.
- This is also known as an algorithm’s **growth rate**.
- The running time of an algorithm is compared with **some common orders of growth** in the complexity analysis. Such as,

Time Complexity	Growth rate	n=10	n=100
$f(n) = O(1)$	**growth is constant**	**Does not depend on n**	
$f(n) = O(\log n)$	**growth is logarithmic**	**3.32**	**6.64**
$f(n) = O(n)$	**growth is linear**	**10**	**100**
$f(n) = O(n \log n)$	**growth is faster than linear**	**33.2**	**664**
$f(n) = O(n^2)$	**growth is quadratic**	**100**	**10000**
$f(n) = O(2^n \; or \; n!)$	**growth is exponential**	**1024**	**1.2676506e+30**

Asymptotic Notations

- 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)

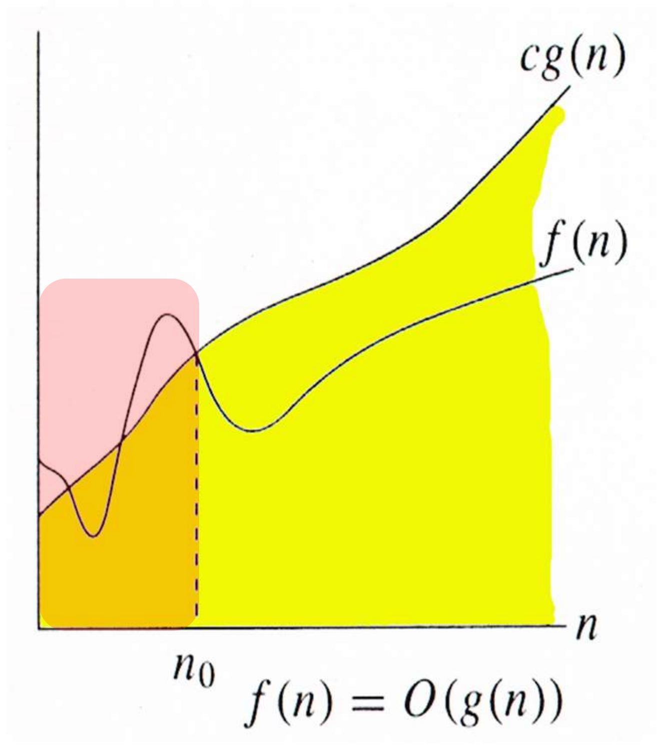
For a given function $g(n)$, we denote by $O(g(n))$ the set of functions,

$O(g(n))$ = {$f(n)$: *there exist positive constants c and* n_0 *such that*

$0 \leq f(n) \leq cg(n)$ *for all* $n_0 \leq n$}

- $g(n)$ is an asymptotically **upper bound** for $f(n)$.
- $f(n) = O(g(n))$ implies:

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



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

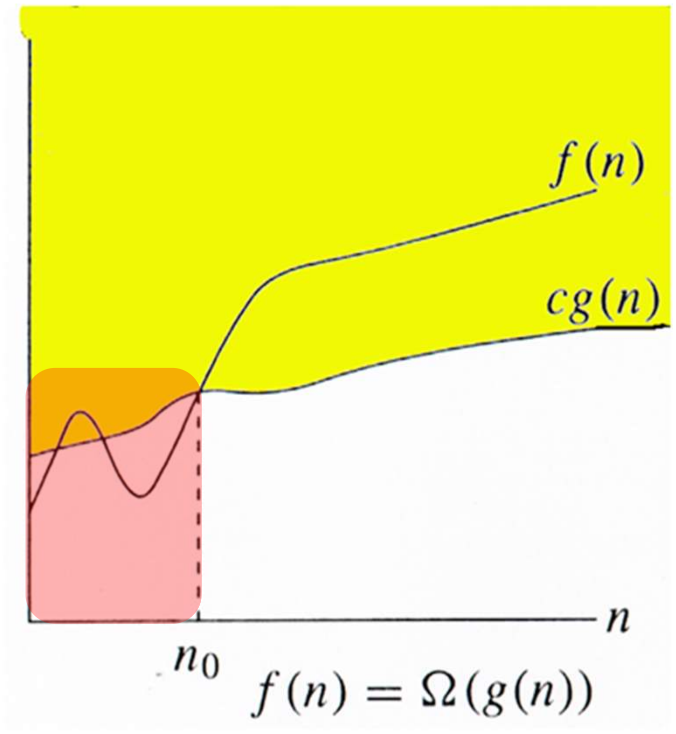
For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions,

$\Omega(g(n))$ = {$f(n)$: *there exist positive constants c and* n_0 *such that*

$0 \leq cg(n) \leq f(n)$ *for all* $n_0 \leq n$}

- $g(n)$ is an asymptotically **lower bound** for $f(n)$.
- $f(n) = \Omega(g(n))$ implies:

$$\boldsymbol{f(n) \text{“} \geq \text{”} c.g(n)}$$



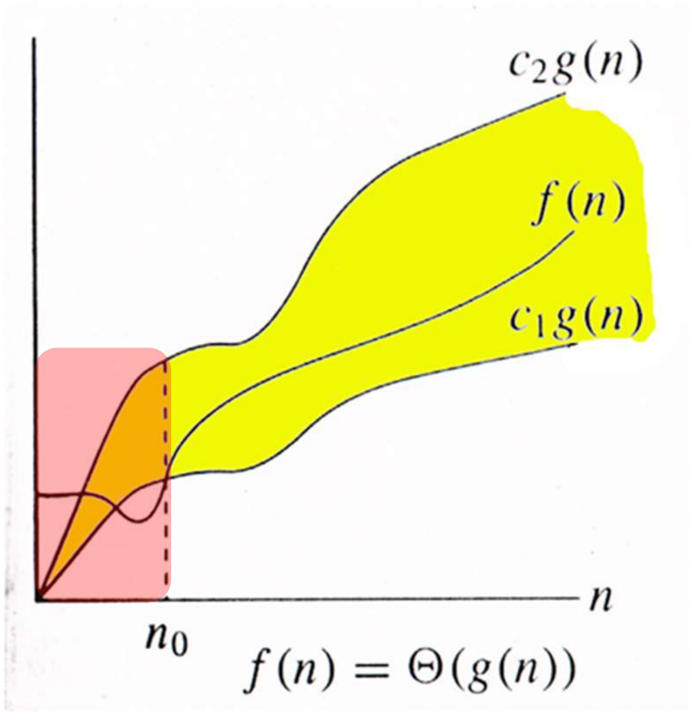
3. θ-Notation (Theta notation) (Same order)

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

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

- $\theta(g(n))$ is a set, we can write $f(n) \in \theta(g(n))$ 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:

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



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 $\boxed{0 \leq f(n) \leq cg(n)}$ for all $n_0 \leq n\}$

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

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

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 $\boxed{0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)}$ for all $n_0 \leq n\}$

Asymptotic Notations

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

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

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

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

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

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

Asymptotic Notations

- 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.

Asymptotic Notations - Example

- Example 1:

$f(n) = n^2$ *and* $g(n) = n$

Algo. 1 running time

Algo. 2 running time

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

n	$f(n) = n^2$	$g(n) = n$
1		
2		
3		
4		
5		

- Example 2:

$f(n) = n$ *and* $g(n) = n^2$

Algo. 1 running time

Algo. 2 running time

$f(n) \leq g(n)$ | $f(n) = O(g(n))$

n	$f(n) = n$	$g(n) = n^2$
1	1	1
2	2	4
3	3	9
4	4	16
5	5	25

Asymptotic Notations - Example

- Example: Let $\boxed{f(n) = n^2}$ and $\boxed{g(n) = 2^n}$

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

Algo. 1 running time

Algo. 2 running time

n	$f(n) = n^2$	$g(n) = 2^n$		
1	1	2		
2	4	4		
3	9	8		
4	16	16		
5	25	32		Here for $n \geq 4$, $f(n) \leq g(n)$ $so, n_0 = 4$
6	36	64		
7	49	128		

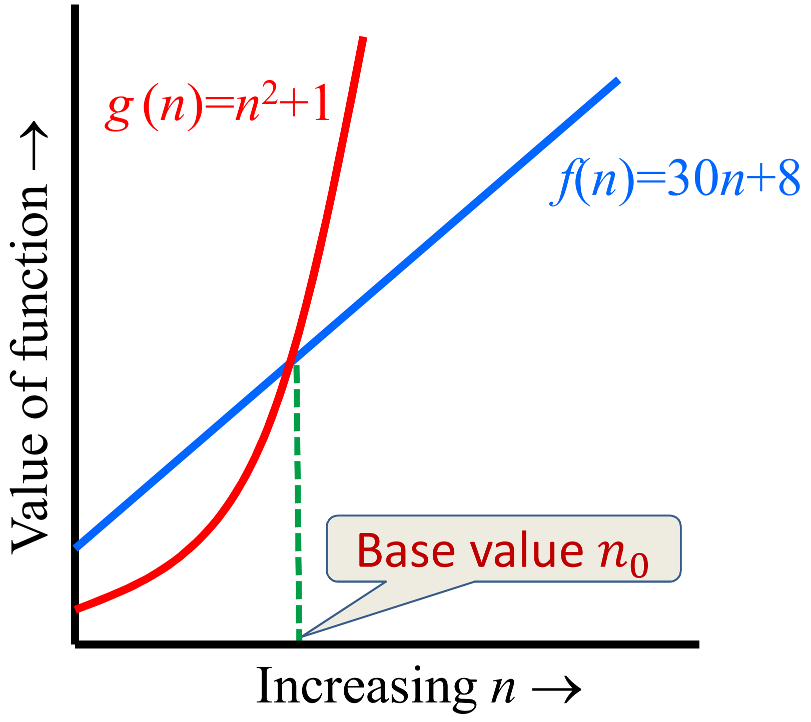
Asymptotic Notations - Example

- Example:

$\boxed{\mathrm{f(n) = 30n + 8}}$ is in the order of n, or $\underline{\mathrm{O(n)}}$.

$\boxed{\mathrm{g(n) = n^2 + 1}}$ is order $\mathrm{n^2}$, or $\underline{\mathrm{O(n^2)}}$.

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



In general, any $O(n^2)$ function is faster-growing than any $O(n)$ function.

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}	∞

Growth of Function (HW)

- Arrange the given notations in the increasing order of their values.

1

n	$nlogn$	2^n	$logn$	$\sqrt{n}$	e^n	$n^2 + logn$	n^2	$loglogn$	n^3	$(logn)^2$	$n!$

2

$\boldsymbol{n^8}$	$\boldsymbol{(n^2 - n + 1)^4}$	$\boldsymbol{n^n}$	$\boldsymbol{n^{(1+e)}}$	$\boldsymbol{(1 + e)^n}$	$\boldsymbol{n^2/logn}$	$\boldsymbol{nlogn}$

Exercises

For each of the following pairs of functions, either $f(n)$ is $O(g(n))$, $f(n)$ is $\Omega(g(n))$, or $f(n) = \theta(g(n))$. Determine which relationship is correct.

$f(n) = n;\ g(n) = \log n^2$	
$f(n) = \log\log n;\ g(n) = \log n$	
$f(n) = n;\ g(n) = log^2 n$	
$f(n) = n\log n + n;\ g(n) = \log n$	
$f(n) = 10;\ g(n) = \log 10$	
$f(n) = 2^n;\ g(n) = 10n^2$	
$f(n) = 2n;\ g(n) = 3^n$	

Asymptotic Notations in Equations

- **Maximum Rule**: Let, $f, g\colon N \rightarrow R^{+}$ 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* $O(n^4)$
2. $10n^3 + 2n^2$ *is* $O(n^3)$
3. $n^3 - n^2$ is $O(n^3)$

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

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

- Consider the example of buying elephants and goldfish:

Cost = cost_of_elephants + cost_of_goldfish

Negligible

Cost ≈ cost_of_elephants (approximation)

- $n^4 + 100n^2 + 10n + 50$ *and* n^4 *have the same rate of growth.*

Exercises

- Express the function $n^3/1000 - 100n^2 - 100n + 3$ in terms of θ notation.

  $\boxed{\theta(n^3)}$

- Express $20n^3 + 10n \log n + 5$ in terms of O notation.

  $\boxed{O(n^3)}$

- Express $5n \log n + 2n$ in terms of O notation.

  $\boxed{O(n \log n)}$

Home Work

1. Prove (i) Is $2^{n+1} = O(2^n)$? (ii) Is $2^{2n} = O(2^n)$?
2. Check the correctness for the following equality.

$$5n^3 + 2n = O(n^3)$$

3. Find θ notation for the following function

$$F(n) = 3 * 2^n + 4n^2 + 5n + 2$$

4. Find O notation for the following function

$$a.\quad F(n) = 2^n + 6n^2 + 3n$$

$$b.\quad F(n) = 4n^3 + 2n + 3$$

5. Find Ω notation for the following function

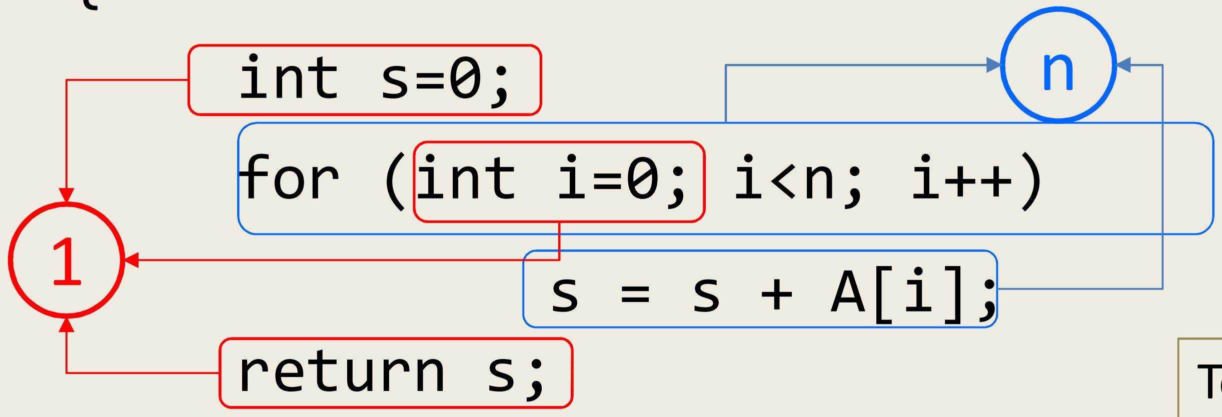
$a.\ F(n) = 4 * 2^n + 3n$

$b.\ F(n) = 5n^3 + n^2 + 3n + 2$

Sum of n Elements of Array

- Algorithm

```
//Input: int A[n], array of n integers
//Output: Sum of all numbers in array A
int Sum(int A[], int n)
{
```



```
    int s=0;
    for (int i=0; i<n; i++)
        s = s + A[i];
    return s;
```

```
}
```

Total: $2n + 3$

The time complexity of the algorithm is :

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

Running Time of Algorithm

- Estimated running time for different values of n :

The time complexity *of the* algorithm is :

$$f(n) = 2 \cdot n + 3$$

$n = 10$	23 steps
$n = 100$	203 steps
$n = 1000$	2,003 steps
$n = 10000$	20,003 steps

- As n grows, the number of steps grow in linear proportion to n for the given algorithm **Sum.**

Running Time of Algorithm

- The dominating term in the function of time complexity is $\boldsymbol{n}$:

The time complexity *of the* algorithm is :

$$f(n) = 2 \cdot n + 3$$

$n = 10$	23 steps
$n = 100$	203 steps
$n = 1000$	2,003 steps
$n = 10000$	20,003 steps

- As n gets large, the $+3$ becomes insignificant.
- **The time is linear in proportion to $\boldsymbol{n}$.**

Analyzing Control Statement

Example 1:

$sum = a + b$ — c

Statement is executed once only

The execution time $T(n)$ is some constant $c \approx O(1)$

Example 2:

```
$for\ i = 1\ to\ n\ do$    $c_1 * (n + 1)$
    $sum = a + b;$    $c_2 * (n)$
```

Total time is denoted as,

$T(n) = c_1 n + c_1 + c_2 n$

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

Example 3:

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

Analysis

$T(n) = c_1(n + 1) + c_2 n(n + 1) + c_3 n(n)$

$T(n) = c_1 n + c_1 + c_2 n^2 + c_2 n + c_3 n^2$

$T(n) = n^2(c_2 + c_3) + n(c_1 + c_2) + c_1$

$T(n) = \bigcirc\!\!\!\!\!\!\ an^2 + bn + c$

$\boldsymbol{T(n) = O(n^2)}$

Analyzing Control Statement

Example 4:

```
$l = 0$
$for\ i = 1\ to\ n\ do$
    $for\ j = 1\ to\ i\ do$
        $for\ k = j\ to\ n\ do$
            $l = l + 1$
```

$$t(n) = \theta(n^3)$$

Example 5:

```
$l = 0$
$for\ i = 1\ to\ n\ do$
    $for\ j = 1\ to\ n^2\ do$
        $for\ k = 1\ to\ n^3\ do$
            $l = l + 1$
```

$$t(n) = \theta(n^6)$$

Example 6:

```
$for\ j = 1\ to\ n\ do$                          $\theta(n^2)$
    $for\ k = 1\ to\ j\ do$
        $sum = sum + j * k$
$for\ l = 1\ to\ n\ do$                          $\theta(n)$
        $sum = sum - l + 1$
printf("sum is now %d", $sum$)                  $\theta(1)$
```

$$t(n) = \underline{\theta(n^2)} + \theta(n) + \theta(1)$$
$$t(n) = \theta(n^2)$$