Introduction
to
Data Structures

Alok Kumar Jagadev

What is Algorithm

- An algorithm is "a finite set of precise instructions for performing a computation or for solving a problem". It will take some value or set of values, as input, and produces some value or set of values, as output.
- Example:
  - Directions to somebody's house is an algorithm
  - A sorted, non-decreasing sequence of natural numbers of non-zero, finite length

Some algorithms are harder than others

- Some algorithms are easy
  - Finding the largest (or smallest) value in a list
  - Finding a specific value in a list
- Some algorithms are a bit harder
  - Sorting a list
- Some algorithms are very hard
  - Finding the shortest path between Miami and Seattle
- Some algorithms are essentially impossible
  - Factoring large composite numbers

Characteristics of an Algorithm

- Well-Defined Inputs: Inputs, it should be well-defined.
- Well-Defined Outputs: Must clearly define the output.
- Definiteness: the steps are defined precisely.
- Correctness: should produce the correct output.
- Finiteness: should end up with finite number of steps.
- Effectiveness: each step must be able to be performed in a finite amount of time.
- Generality: the algorithm should be applicable to all problems of a similar form.
- Language Independent: Just plain instructions that can be implemented in any language.

Algorithm: Example

Algorithm to add two numbers

Step 1: Start
Step 2: Declare variables num1, num2 and sum.
Step 3: Read values num1 and num2.
Step 4: Add num1 and num2 and assign the result to sum.
sum←num1+num2
Step 5: Display sum
Step 6: Stop

Algorithm to find all roots of a quadratic equation $ax^2+bx+c=0$.

Step 1: Start
Step 2: Declare variables a, b, c, D, x1, x2, rp and ip;
Step 3: Calculate discriminant
$D \leftarrow b^2-4ac$
Step 4: If $D \geq 0$
$r1 \leftarrow (-b+\sqrt{D})/2a$
$r2 \leftarrow (-b-\sqrt{D})/2a$
Display r1 and r2 as roots.
Else
Calculate real part and imaginary part
$rp \leftarrow -b/2a$
$ip \leftarrow \sqrt{(-D)}/2a$
Display rp+j(ip) and rp-j(ip) as roots
Step 5: Stop

What is a Good Algorithm?

- Efficient:
  - Running time (small)
  - Space used (less)

- Efficiency as a function of input size:
  - The input size
  - Number of data elements

Space Complexity of Algorithms

For any algorithm memory may be used for the following:

- Variables (includes constant values, temporary values)
- Program Instruction
- Execution

- Space complexity: Amount of memory used by the algorithm (including the input values to the algorithm) to execute and produce the output.
- While executing, algorithm uses memory space for three reasons:
  - Instruction Space: memory used to save the compiled version of instructions.
  - Environmental Stack: In function call, a system stack is maintained.
  - Data Space: Amount of space used by the variables and constants.

- while calculating the Space Complexity of any algorithm,
  - only Data Space is considered (neglecting Instruction Space and Environmental Stack.)

Calculating the Space Complexity

Example:

```
{
    int z = a + b + c;
    return(z);
}
```

In this expression,

- variables a, b, c and z are all integer types, take 4 bytes each.
- Total memory = (4(4) + 4) = 20 bytes,
- Additional 4 bytes is for return value.
- Because this space requirement is fixed, called Constant Space Complexity.

Example:

```
// n is the length of array a[]
int sum(int a[], int n) {
    int x = 0;                          // 4 bytes for x
    for(int i = 0; i < n; i++)  {       // 4 bytes for i
        x  = x + a[i];
    }
    return(x);
}
```

- In this code, 4*n bytes of space is required for the array a[] elements.
- 4 bytes each for x, n, i and the return value.
- Total memory requirement is (4n + 16), increasing linearly with increase in the input value n. Called as Linear Space Complexity.

Data Structures and Algorithms

- Algorithm: Outline, the essence of a computational procedure, step-by-step instructions
- Program: an implementation of an algorithm in some programming language
- Data structure:
  - Organization of data needed to solve the problem
  - A way of organizing all data items that considers not only the elements stored but also their relationship to each other.

Time Complexity

- Time complexity of an algorithm signifies the total time required by the program to run till its completion.
- Time Complexity is most commonly estimated by counting the number of elementary steps performed by any algorithm to finish execution.
- Any problem can have number of solutions.
- Two different algorithms to find square of a number:

```
m=0
for i=1 to n
   do m = m + n
return m
```

- loop will run n number of times
- time complexity will be n atleast
- As the value of n will increase the time taken will also increase

```
return n*n
```

- For this code, time complexity is constant
- Never be dependent on the value of n, so always give the result in 1 step.

Time Complexity

- Performance of algorithm may vary with different input data
- worst-case Time complexity is considered because that is the maximum time taken for any input size.

- The most common metric for calculating time complexity is Big O notation.
- This removes all constant factors so that the running time can be estimated in relation to n, as n approaches infinity.

Time Complexity

statement;

- A single statement.
- Its Time Complexity will be Constant.
- The running time of the statement will not change in relation to n.

```
for(i=0; i < n; i++)  {
    statement;
}
```

- The time complexity for this algorithm will be Linear.
- The running time of the loop is directly proportional to n.
- When n doubles, so does the running time.

Time Complexity

```
for(i=0; i < n; i++)  {
    for(j=0; j < n; j++)   {
            statement;
    }
}
```

- Time complexity will be Quadratic.
- Running time of the two loops is proportional to the square of n.
- When n doubles, the running time increases by n * n.

```
while(low <= high) {
    mid = (low + high) / 2;
    if (target < list[mid])
        high = mid - 1;
    else if (target > list[mid])
        low = mid + 1;
    else break;
}
```

- This is an algorithm to break a set of numbers into halves, to search a particular value.
- Logarithmic Time Complexity.
- The running time of the algorithm is proportional to the number of times n can be divided by 2.
- Because the algorithm divides the working area in half with each iteration.

Asymptotic Analysis

- Asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run-time performance.
- Asymptotic analysis is input bound
  - no input to the algorithm, concluded to work in a constant time.
  - Other than the "***input***" all other factors are considered constant.

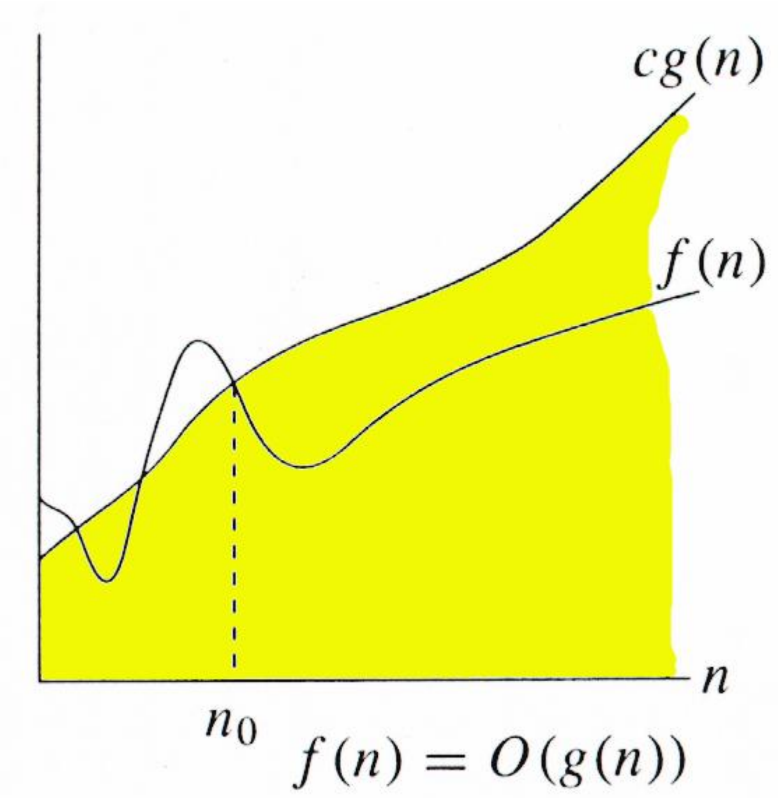
- Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation.
  - Example: running time of one operation is computed as f(n) and may be for another operation it is computed as $g(n^2)$
- Describes the algorithm efficiency and performance in a meaningful way.
- Describes the behaviour of time or space complexity for an algorithm

Asymptotic Notation

- Commonly used asymptotic notations to calculate the running time complexity of an algorithm.
  - O Notation
  - Ω Notation
  - θ Notation

Big Oh Notation, O

- The Big Oh notation 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.
- Definition: $O(g(n)) = \{ f(n) :$ $\exists$ positive constants c and $n_0,$ such that $\forall n \geq n_0,$
- we have $0 \leq f(n) \leq cg(n) \}$
- Intuitively: Set of all functions whose rate of growth is the same as or lower than that of g(n).



Big Oh Notation: Example

Example:

Compute Big-Oh notation for

$f(n) = 10n^2+4n+2$

Given $f(n)= 10n^2+4n+2$

$f(n) \leq c * g(n)$

$10n^2+4n+2 \leq 10n^2+4n+n$, for $n \geq 2$

$10n^2+4n+2 \leq 10n^2+5n$

$10n^2+4n+2 \leq 10n^2+n^2$, for $n \geq 5$

$10n^2+4n+2 \leq 11n^2$ where $c= 11$,

$g(n) = n^2$ and $n_0=5$

Hence $f(n) = O(n^2)$

Example:

Compute Big-Oh notation for

$f(n) = 1000n^2+100n- 6$

Given $f(n) = 1000n^2+100n-6$

$f(n) \leq c * g(n)$

$1000n^2+100n-6 \leq 1000\ n^2+100n$ for all values of n

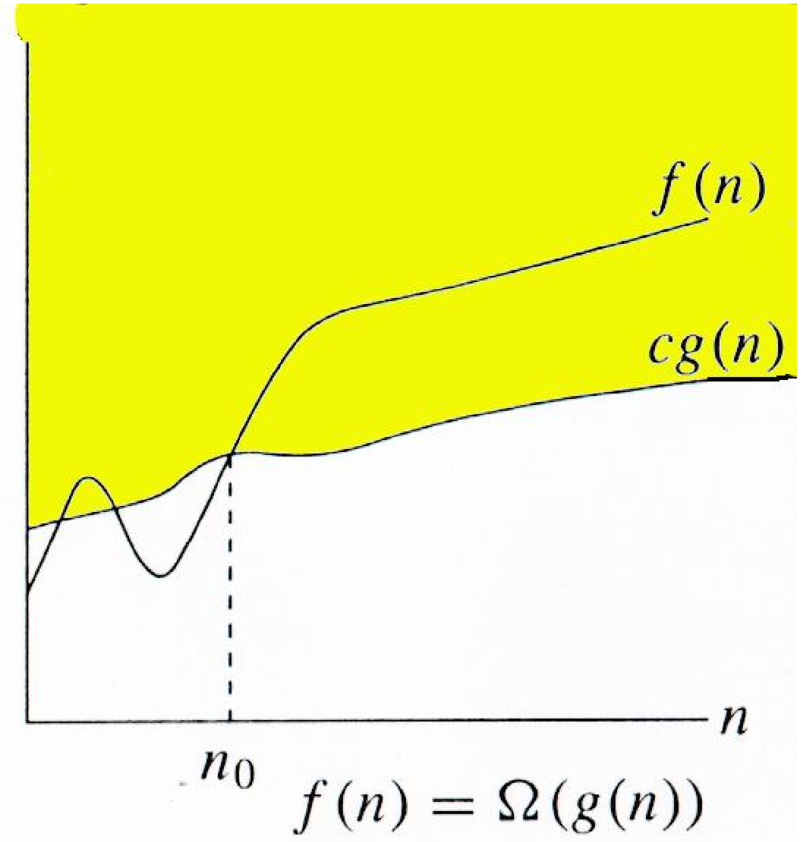
$1000n^2+100n-6 \leq 1000\ n^2+n^2$, for $n \geq 100$

$1000n^2+100n-6 \leq 1001\ n^2$, where $c =1001$, $g(n) = n^2$ and $n_0=100$

Hence $f(n)=O(n^2)$

Omega Notation, Ω

- The Omega notation is the formal way to express the lower bound of an algorithm's running time.
- It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.
- Definition: $\Omega(g(n)) = \{ f(n) :$ $\exists$ positive constants c and n_0, such that $\forall n \geq n_0$,
- we have $0 \leq cg(n) \leq f(n)\}$
- Intuitively: Set of all functions whose rate of growth is the same as or higher than that of g(n).



Omega Notation, Ω

Example:

Compute omega notation for $f(n)= 10n^2+4n+2$

Given $f(n)= 10n^2+4n+2$

$f(n) \geq c * g(n)$

$10n^2+4n+2 \geq 10n^2$ for all values of n $(n\geq 0)$

where $c=10$, $g(n)=n^2$ and $n_0=0$

Hence $f(n) = \Omega(n^2)$

Example:

Compute omega notation for $f(n)= 4n^3+2n+3$

Given $f(n)= 4n^3+2n+3$

$f(n) \geq c * g(n)$

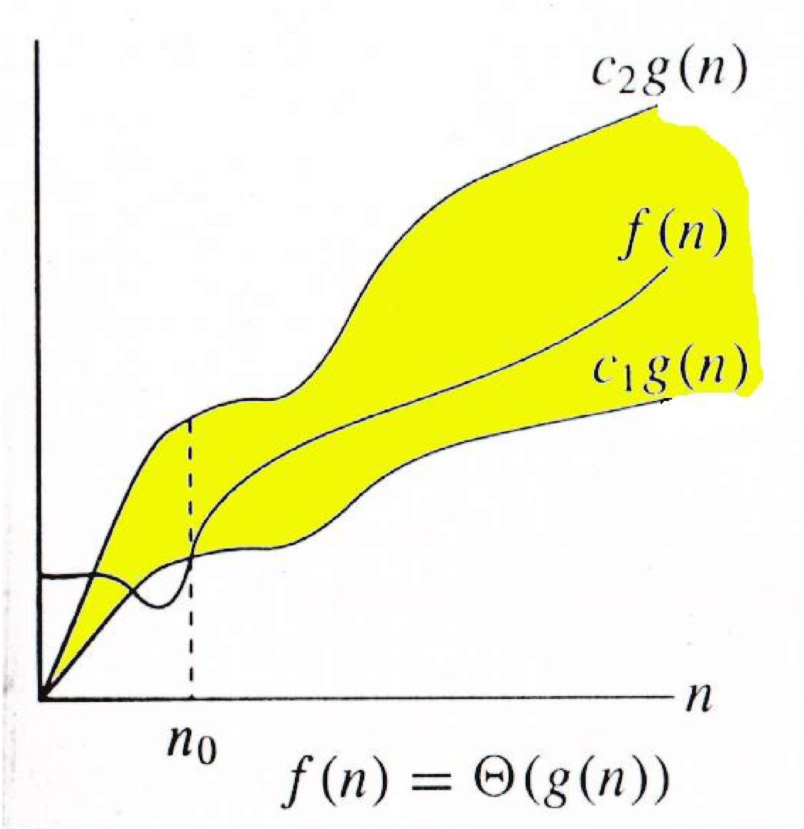
$4n^3+2n+3 \geq 4n^3$ for all values of n $(n\geq 0)$

where $c=4$, $g(n)=n^3$

and $n_0=0$

Theta Notation, θ

- The Theta notation is the formal way to express both the lower bound and the upper bound of an algorithm's running time.
- Definition: $\Theta(g(n)) = \{f(n)$: $\exists$ positive constants c_1, c_2, and n_0, such that $\forall n \geq n_0$, $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}$
- *Intuitively*: Set of all functions that have the same *rate of growth* as $g(n)$.
- g(n) is an asymptotically tight bound for f(n).



Theta Notation, θ

Example:

Compute theta notation for $f(n)=3n+2$

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

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

Compute $f(n) \leq c_2 * g(n)$

$3n+2 \leq 3n+n$, for $n \geq 2$

$3n+2 \leq 4n$ where $c_2=4$ and $g(n)=n$

Compute $c_1 * g(n) \leq f(n)$

$3n \leq 3n+2$ for all values of n

where $c_1=3$, $g(n)=n$

Hence, $f(n)= \theta(n)$

Example:

Compute theta notation for $f(n)=10n2+4n+2$

Given $f(n)=10n^2+4n+2$

$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$

Compute $f(n) \leq c_2 * g(n)$

$10n^2+4n+2 \leq 10n^2+4n+n$, for $n \geq 2$

$10n^2+4n+2 \leq 10n^2+5n$

$10n^2+4n+2 \leq 10n^2+n^2$, for $n \geq 5$

$10n^2+4n+2 \leq 11n^2$, where $c_2=11$ and $g(n)=n^2$

Compute $c_1 * g(n) \leq f(n)$

$10n^2 \leq 10n^2+4n+2$ for all values of n

where $c1=10$, $g(n)=n^2$

Hence, $f(n)= \theta(n^2)$

Common Asymptotic Notations

Following is a list of some common asymptotic notations:

- constant − $O(1)$
- logarithmic − $O(\log n)$
- linear − $O(n)$
- n log n − $O(n \log n)$
- quadratic − $O(n^2)$
- cubic − $O(n^3)$
- polynomial − $n^{O(1)}$
- exponential − $2^{O(n)}$

Analysis of Algorithms

1. O(1): Time complexity of a function (or set of statements) is considered as O(1) if it doesn't contain loop, recursion and call to any other non-constant time function.

    // set of non-recursive and non-loop statements

    - Example: swap() function has O(1) time complexity.
    - A loop or recursion that runs a constant number of times is also considered as O(1). For example the following loop is O(1).

```
// Here c is a constant
for (int i = 1; i <= c; i++) {
    // some O(1) expressions   }
```

Analysis of Algorithms

2) O(n): Time Complexity of a loop is considered as O(n) if the loop variables is incremented / decremented by a constant amount.

Example:

```
// Here c is a positive integer constant
   for (int i = 1; i <= n; i += c) {
        // some O(1) expressions
   }
   for (int i = n; i > 0; i -= c) {
        // some O(1) expressions    }
```

Analysis of Algorithms

3. $O(n^c)$: Time complexity of nested loops is equal to the number of times the innermost statement is executed.

Example:

```
for (int i = 1; i <=n; i += c) {
    for (int j = 1; j <=n; j += c) {
      // some O(1) expressions   }
 }

 for (int i = n; i > 0; i -= c) {
    for (int j = i+1; j <=n; j += c) {
      // some O(1) expressions
 }
```

For example Selection sort and Insertion Sort have $O(n^2)$ time complexity.

Analysis of Algorithms

4) O(Logn): Time Complexity of a loop is considered as O(log n) if the loop variables is divided / multiplied by a constant amount.

```
for (int i = 1; i <=n; i *= c) {
    // some O(1) expressions    }
for (int i = n; i > 0; i /= c) {
    // some O(1) expressions    }
```

Example: Binary Search (refer iterative implementation) has O(log n) time complexity.

Analysis of Algorithms

5. O(LogLogn) Time Complexity of a loop is considered as O(log log n) if the loop variables is reduced / increased exponentially by a constant amount.

```
// Here c is a constant greater than 1
 for (int i = 2; i <=n; i = pow(i, c)) {
    // some O(1) expressions
 }
 //Here func() is sqrt or cuberoot or any other constant root
 for (int i = n; i > 0; i = func(i)) {
    // some O(1) expressions
 }
```

Analysis of Algorithms

How to combine time complexities of consecutive loops?

When there are consecutive loops, we calculate time complexity as sum of time complexities of individual loops.

```
for (int i = 1; i <=m; i += c) {
    // some O(1) expressions
}
for (int i = 1; i <=n; i += c) {
    // some O(1) expressions
}
```

- Time complexity of above code is O(m) + O(n) which is O(m+n)
- If m == n, the time complexity becomes O(2n) which is O(n).

Question Discussion

What is time complexity of func()?

```
int func(int n)  {
  int count = 0;
  for (int i = n; i > 0; i /= 2)
     for (int j = 0; j < i; j++)
        count += 1;
  return count; }
```

(A) $O(n^2)$
(B) O(nLogn)
(C) O(n)
(D) O(nLognLogn)

Question Discussion

What is time complexity of func()?

```
int func(int n)  {
  int count = 0;
  for (int i = n; i > 0; i /= 2)
     for (int j = 0; j < i; j++)
        count += 1;
  return count; }
```

(A) $O(n^2)$

(B) O(nLogn)

(C) O(n)

(D) O(nLognLogn)

Answer: (C)

Explanation:

- For a input integer n, the innermost statement of func() is executed following times.

  $$n + n/2 + n/4 + \ldots 1$$

- So time complexity T(n) can be written as: $T(n) = O(n + n/2 + n/4 + \ldots 1) = O(n)$

Question Discussion

What is the time complexity of func()?

```
int func(int n)  {
  int count = 0;
  for (int i = 0; i < n; i++)
     for (int j = i; j > 0; j--)
        count = count + 1;
  return count;
}
```

(A) O(n)

(B) $O(n^2)$

(C) O(n*Logn)

(D) O(nLognLogn)

Question Discussion

What is the time complexity of func()?

```
int func(int n)  {
  int count = 0;
  for (int i = 0; i < n; i++)
     for (int j = i; j > 0; j--)
        count = count + 1;
  return count;
}
```

(A) O(n)

(B) $O(n^2)$

(C) O(n*Logn)

(D) O(nLognLogn)

Answer: (B)

Explanation:

- Time complexity can be calculated by counting number of times the expression "count = count + 1;" is executed.
- The expression is executed 0 + 1 + 2 + 3 + 4 + …. + (n-1) times.
- Time complexity = O(0 + 1 + 2 + 3 + .. + n-1) = O(n*(n-1)/2) = $O(n^2)$

Question Discussion

Which of the given options provides the increasing order of asymptotic complexity of functions f1, f2, f3 and f4?

$f1(n) = 2^n$ $f2(n) = n^{(3/2)}$ $f3(n) = nLogn$ $f4(n) = n^{(Logn)}$

(A) f3, f2, f4, f1

(B) f3, f2, f1, f4

(C) f2, f3, f1, f4

(D) f2, f3, f4, f1

Question Discussion

Which of the given options provides the increasing order of asymptotic complexity of functions f1, f2, f3 and f4?

$f1(n) = 2^n$ $\quad$ $f2(n) = n^{(3/2)}$

$f3(n) = n\text{Log}n$ $\quad$ $f4(n) = n^{(\text{Log}n)}$

(A) f3, f2, f4, f1

(B) f3, f2, f1, f4

(C) f2, f3, f1, f4

(D) f2, f3, f4, f1

Ans: (A)

Explanation:

$f1(n) = 2^n$ $\quad$ $f2(n) = n^{(3/2)}$

$f3(n) = n\text{Log}n$ $\quad$ $f4(n) = n^{(\text{Log}n)}$

- Except f3, all other are polynomial/ exponential.
- f3 is definitely first in output. Among remaining, $n^{(3/2)}$ is next.
- One way to compare f1 and f4 is to take Log of both functions.
  - Order of growth of Log(f1(n)) is O(n) and order of growth of Log(f4(n)) is O(Logn * Logn).
  - Since O(n) has higher growth than Θ(Logn * Logn), f1(n) grows faster than f4(n).

Question Discussion

Which of the following is not $O(n^2)$?

(A) $(15^{10}) * n + 12099$

(B) $n^{1.98}$

(C) $n^3 / (\mathrm{sqrt}(n))$

(D) $(2^{20}) * n$

Question Discussion

Which of the following is not $O(n^2)$?

(A) $(15^{10}) * n + 12099$

(B) $n^{1.98}$

(C) $n^3 / (sqrt(n))$

(D) $(2^{20}) * n$

Answer: (C)

Explanation: The order of growth of option c is $n^{2.5}$ which is higher than n^2.

Question Discussion

What is the time, space complexity of following code:

```
int a = 0, b = 0;
for (i = 0; i < n; i++) {
    a = a + rand();
}
for (j = 0; j < m; j++) {
    b = b + rand();
}
```

A. O(n * m) time, O(1) space
B. O(n + m) time, O(n + m) space
C. O(n + m) time, O(1) space
D. O(n * m) time, O(n + m) space

Question Discussion

What is the time, space complexity of following code:

```
int a = 0, b = 0;
for (i = 0; i < n; i++) {
    a = a + rand();
}
for (j = 0; j < m; j++) {
    b = b + rand();
}
```

A. O(n * m) time, O(1) space
B. O(n + m) time, O(n + m) space
C. O(n + m) time, O(1) space
D. O(n * m) time, O(n + m) space

C. O(n + m) time, O(1) space

Explanation: The first loop is O(n) and the second loop is O(m). Since we don’t know which is bigger, we say this is O(n + m). This can also be written as O(max(n, m)).

Since there is no additional space being utilized, the space complexity is constant / O(1)

Question Discussion

What is the time complexity of following code:

```
int a = 0;
for (i = 0; i < n; i++) {
    for (j = n; j > i; j--) {
        a = a + i + j;
    }
}
```

A. O(n)
B. O(n*log(n))
C. O(n * sqrt(n))
D. O(n*n)

Question Discussion

What is the time complexity of following code:

```
int a = 0;
for (i = 0; i < n; i++) {
    for (j = n; j > i; j--) {
        a = a + i + j;
    }
}
```

A. O(n)
B. O(n*log(n))
C. O(n * sqrt(n))
D. O(n*n)

D. O(n*n)

Explanation: The above code runs total no of times

$= n + (n - 1) + (n - 2) + \dots 1 + 0$

$= n * (n + 1) / 2$

$= 1/2 * n^2 + 1/2 * n$

$O(n^2)$ times.

Question Discussion

What is the time complexity of following code:

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

A. $O(n)$
B. $O(n \log n)$
C. $O(n^2)$
D. $O(n^2 \log n)$

Question Discussion

What is the time complexity of following code:

```
int i, j, k = 0;
for (i = n / 2; i <= n; i++) {
    for (j = 2; j <= n; j = j * 2) {
        k = k + n / 2;
    }
}
```

A. $O(n)$
B. $O(n \log n)$
C. $O(n^2)$
D. $O(n^2 \log n)$

B. $O(n \log n)$

Explanation:If you notice, j keeps doubling till it is less than or equal to n. Number of times, we can double a number till it is less than n would be log(n).

Let's take the examples here.

for n = 16, j = 2, 4, 8, 16

for n = 32, j = 2, 4, 8, 16, 32

So, j would run for O(log n) steps.

i runs for n/2 steps.

So, total steps = O(n/ 2 * log (n)) = O(n*logn)

Question Discussion

What is the time complexity of following code:

```
int a = 0, i = n;
while (i > 0) {
    a += i;
    i /= 2;
}
```

A. O(n)
B. O(sqrt(n))
C. O(n / 2)
D. O(log n)

Question Discussion

What is the time complexity of following code:

```
int a = 0, i = n;
while (i > 0) {
    a += i;
    i /= 2;
}
```

A. O(n)
B. O(sqrt(n))
C. O(n / 2)
D. O(log n)

D. O(log n)

Explanation: We have to find the smallest x such that $n / 2^x$ n

x = log(n)

Question Discussion

```
for(i=1; i<=n; i=i*2)
        for(j=n; j>=1; j=j/2)
                statement;
```

Time Complexity $O((\log n)^2)$

Question Discussion

```
void function(int n){
        int i,j,k;
        for(i=n/2; i<=n; i++)
                for(j=1; j + n/2<=n; j++)
                        for(k=1; k<=n; k= k * 2)
                                count++;
}
```

Ans: $O(n^2 * \log n)$. In both these type of answers, we should give the marks because students do not know mathematical computation of time complexity.

Question Discussion

What is time complexity of the function func()?

```
void func(int arr[], int n) {
        int i, j = 0;
        for(i=0; i<n; i++)
                while(j < n && arr[i] < arr[j])
                        j++;
}
```

Ans: O(n) (Note: Since the j is not initialized inside the outer while loop, the inner while loop will be executed nearly n number of times)

Question Discussion

What does it mean when we say that an algorithm X is asymptotically more efficient than Y?

A. X will always be a better choice for small inputs
B. X will always be a better choice for large inputs
C. Y will always be a better choice for small inputs
D. X will always be a better choice for all inputs

B. X will always be a better choice for large inputs

Explanation: In asymptotic analysis we consider growth of algorithm in terms of input size. An algorithm X is said to be asymptotically better than Y if X takes smaller time than Y for all input sizes n larger than a value n0 where n0 > 0.

Data Structures and Algorithms

- Data structure affects the design of both structural & functional aspects of a program.

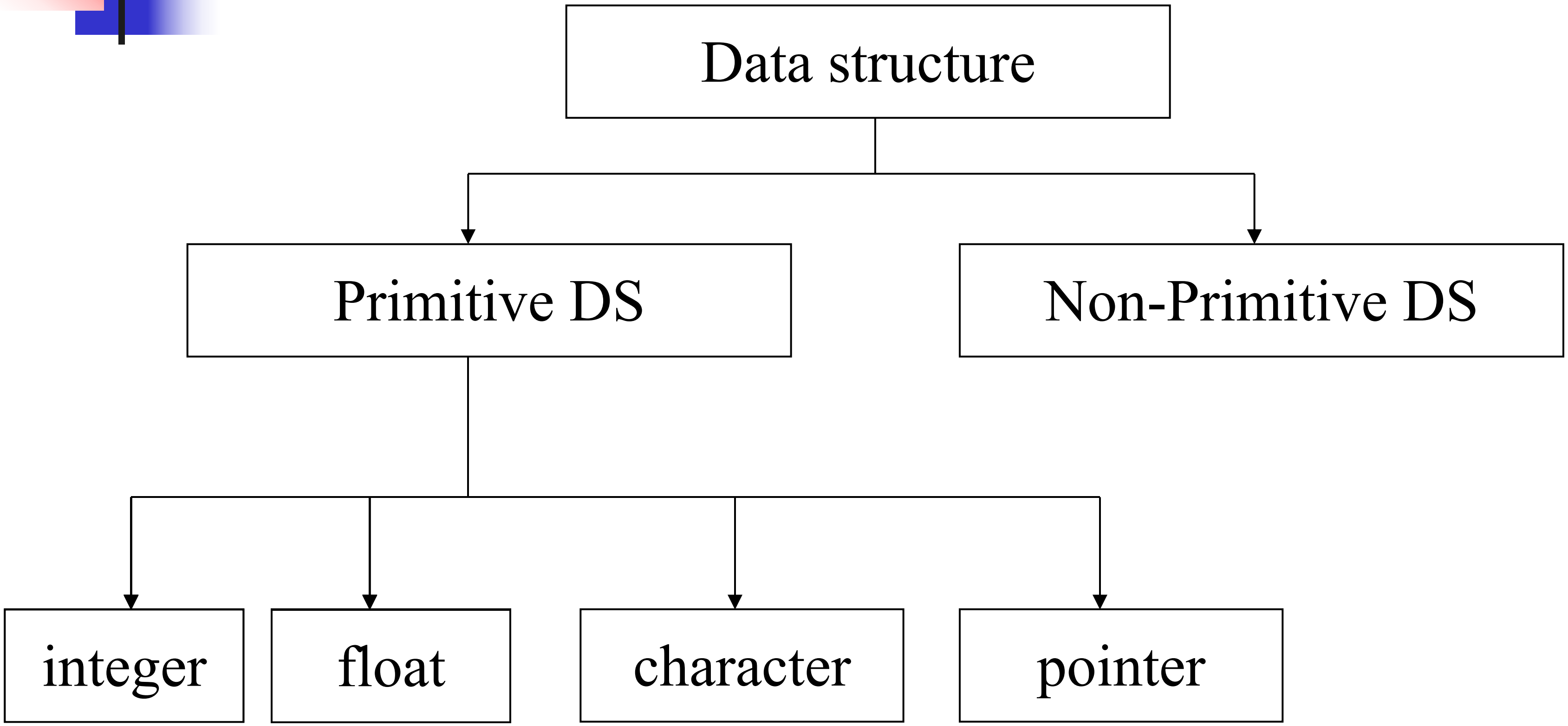
  Program=Algorithm + Data Structure

- Algorithm is a step by step procedure to solve a particular function.

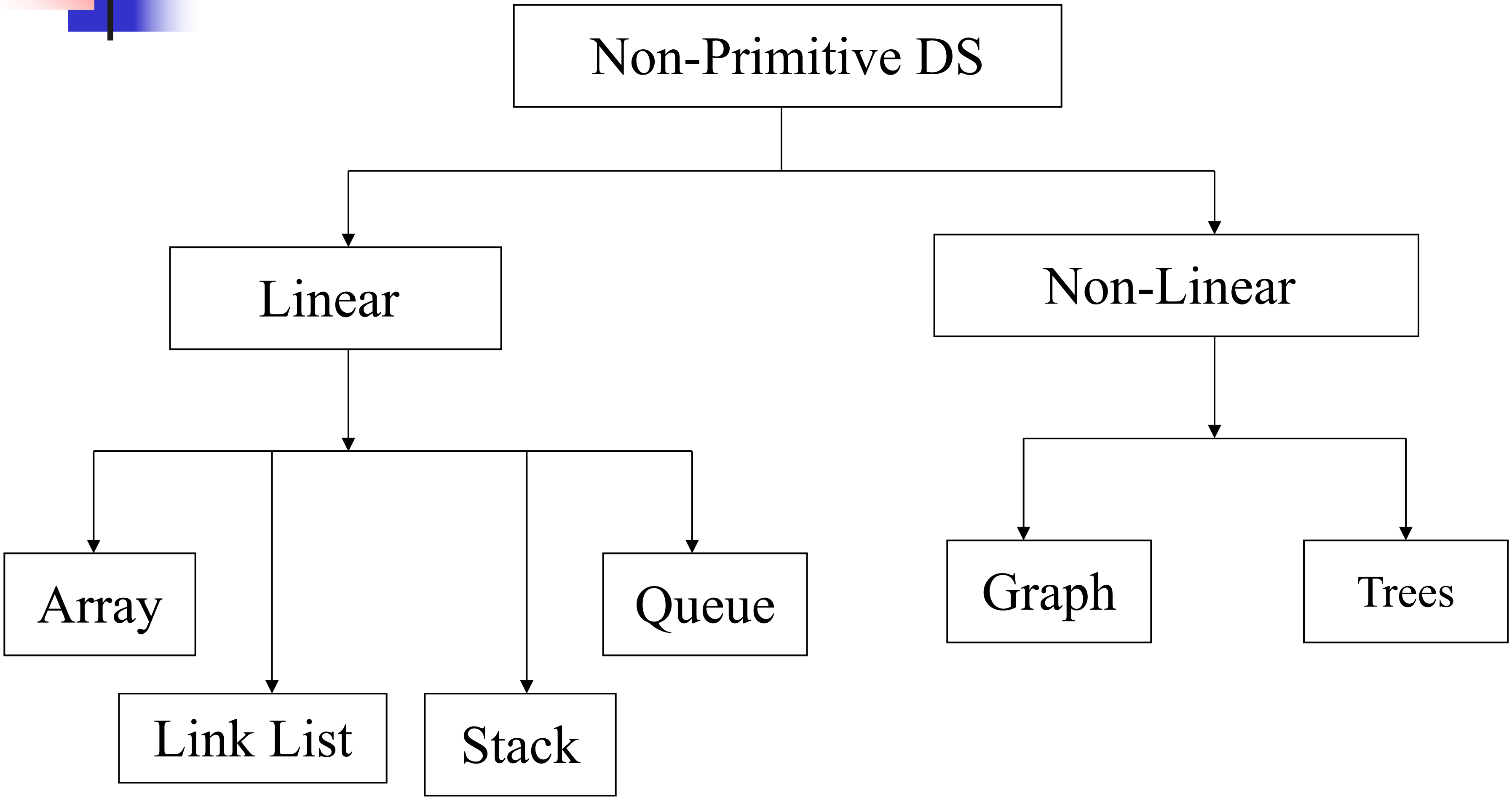
Classification of Data Structure

- Data structure are normally divided into two broad categories:
  - Primitive Data Structure
  - Non-Primitive Data Structure

Classification of Data Structure



Classification of Data Structure



Primitive Data Structure

- There are basic structures and directly operated upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.

Non-Primitive Data Structure

- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.

Non-Primitive Data Structure

- The most commonly used operation on data structure are broadly categorized into following types:
  - Create
  - Selection
  - Updating
  - Searching
  - Sorting
  - Merging
  - Destroy or Delete

Difference between them

- A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.
- A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as
  - linked-list, binary search tree, AVL Tree, graph etc.

Description of various Data Structures: Arrays

- An array is defined as a set of finite number of homogeneous elements or same data items.
- It means an array can contain one type of data only, either all integer, all float-point number or all character.

Arrays

- Simply, declaration of array is as follows:

  int arr[10]

- Where int specifies the data type or type of elements arrays stores.
- “arr” is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

Arrays

- Following are some of the concepts to be remembered about arrays:
  - The individual element of an array can be accessed by specifying name of the array, following by index or subscript inside square brackets.
  - The first element of the array has index zero[0]. It means the first element and last element will be specified as:arr[0] & arr[9] respectively.

Arrays

- The elements of array will always be stored in the consecutive (continues) memory location.
- The number of elements that can be stored in an array, that is the size of array or its length is given by the following equation:

  (Upperbound-lowerbound)+1

Arrays

- For this array it would be

  (9-0)+1=10,where 0 is the lower bound of array and 9 is the upper bound of array.
- Array can always be read or written through loop.
  - If we read a one-dimensional array it require one loop for reading and other for writing the array.

Arrays

- For example: Reading an array

```
for(i=0;i<=9;i++)
    scanf("%d", &arr[i]);
```

- For example: Writing an array

```
for(i=0;i<=9;i++)
    printf("%d", arr[i]);
```

Arrays

- If we are reading or writing two-dimensional array it would require two loops.
  - And similarly the array of a N dimension would required N loops.
- Some common operation performed on array are:
  - Creation of an array
  - Traversing an array

Arrays

- Insertion of new element
- Deletion of required element
- Modification of an element
- Merging of arrays

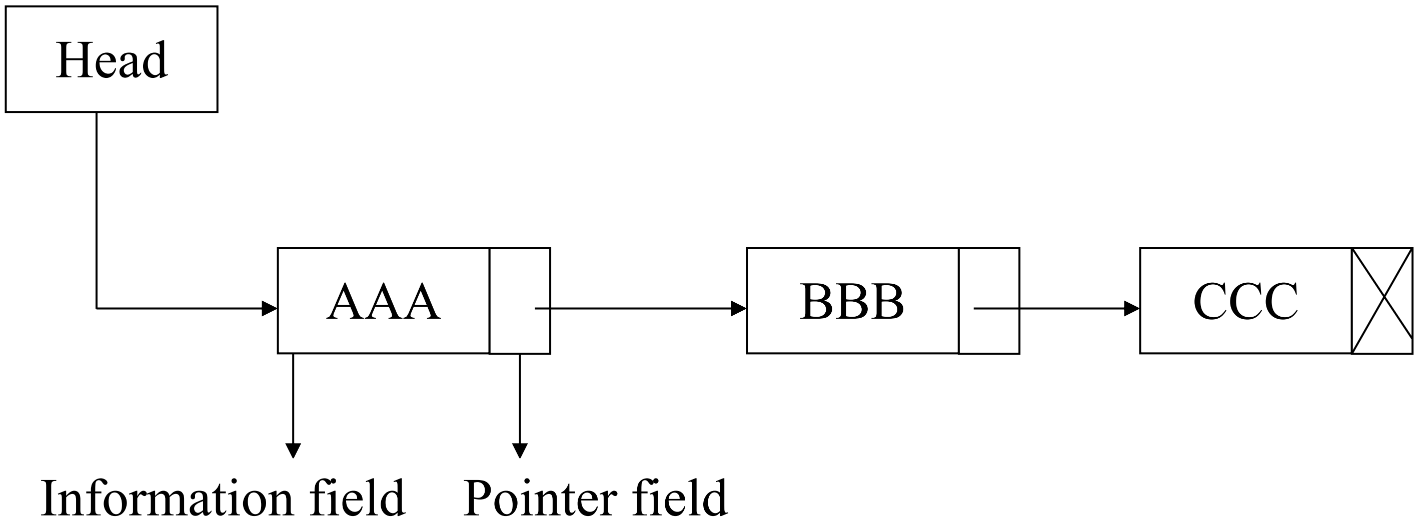
Lists

- A lists (Linear linked list) can be defined as a collection of variable number of data items.
- Lists are the most commonly used non-primitive data structures.
- An element of list must contain at least two fields:
  - one for storing data or information and other for storing address of next element.
- For storing address, a special data structure of list the address must be pointer type.

Lists

- Technically each such element is referred to as a node, therefore a list can be defined as a collection of nodes as show bellow:

[Linear Liked List]



Lists

- Types of linked lists:
  - Single linked list
  - Double linked list
  - Single circular linked list
  - Double circular linked list

Stack

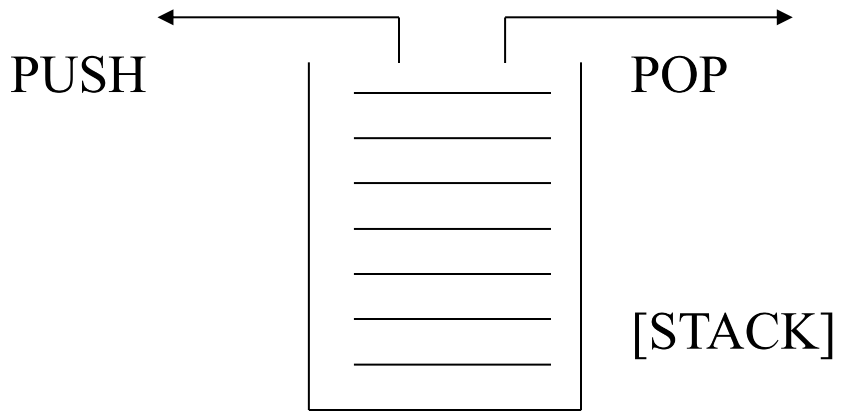
- A stack is also an ordered collection of elements like arrays, but it has a special feature that deletion and insertion of elements can be done only from one end called the top of the stack (TOP)
- Due to this property it is also called as last in first out type of data structure (LIFO).

Stack

- It could be through of just like a stack of plates placed on table in a party, a guest always takes off a fresh plate from the top and the new plates are placed on to the stack at the top.
- It is a non-primitive data structure.
- When an element is inserted into a stack or removed from the stack, its base remains fixed where the top of stack changes.

Stack

- Insertion of element into stack is called PUSH and deletion of element from stack is called POP.
- The bellow show figure how the operations take place on a stack:



Stack

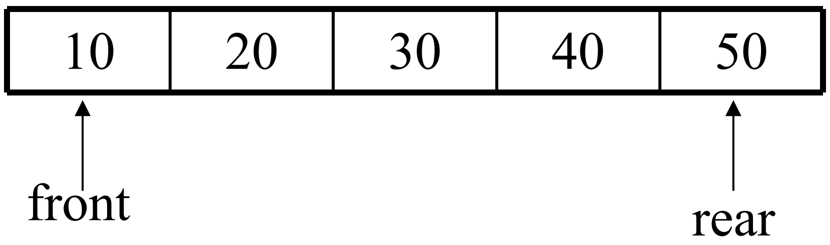
- The stack can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Queue

- Queue are first in first out type of data structure (i.e. FIFO)
- In a queue new elements are added to the queue from one end called REAR end and the element are always removed from other end called the FRONT end.
- The people standing in a railway reservation row are an example of queue.

Queue

- Each new person comes and stands at the end of the row and person getting their reservation confirmed get out of the row from the front end.
- The bellow show figure how the operations take place on a stack:



Queue

- The queue can be implemented into two ways:
  - Using arrays (Static implementation)
  - Using pointer (Dynamic implementation)

Trees

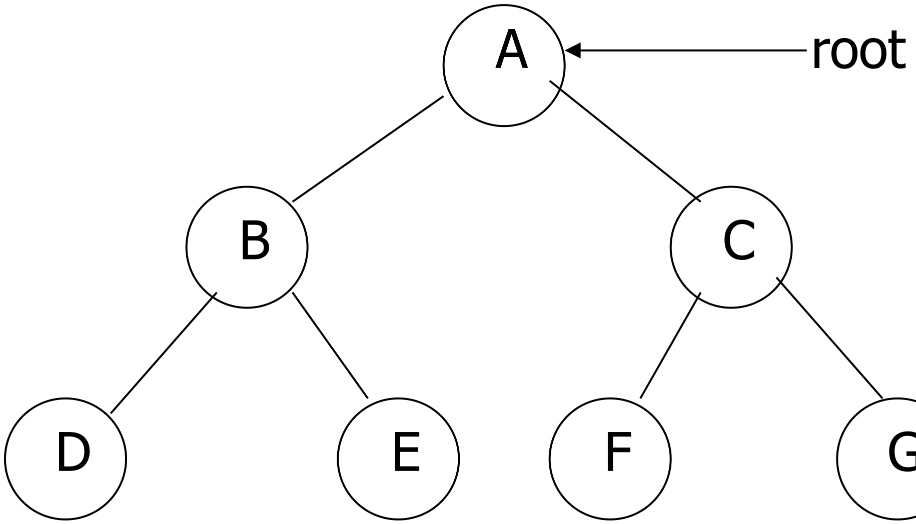
- A tree can be defined as finite set of data items (nodes).
- Tree is non-linear type of data structure in which data items are arranged or stored in a sorted sequence.
- Tree represent the hierarchical relationship between various elements.

Trees

- In trees:
- There is a special data item at the top of hierarchy called the *root* of the tree.
- The remaining data items are partitioned into number of mutually exclusive subset, each of which is itself, a tree which is called the sub tree.
- The tree always grows in length towards bottom in data structures, unlike natural trees which grows upwards.

Trees

- The tree structure organizes the data into branches, which related the information.



Graph

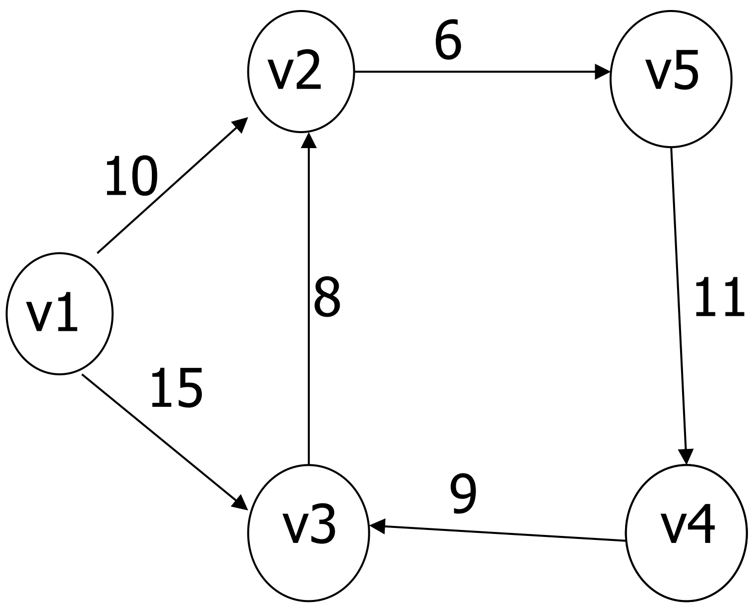
- Graph is a mathematical non-linear data structure capable of representing many kind of physical structures.
- It has found application in Geography, Chemistry and Engineering sciences.
- Definition: A graph G(V,E) is a set of vertices V and a set of edges E.

Graph

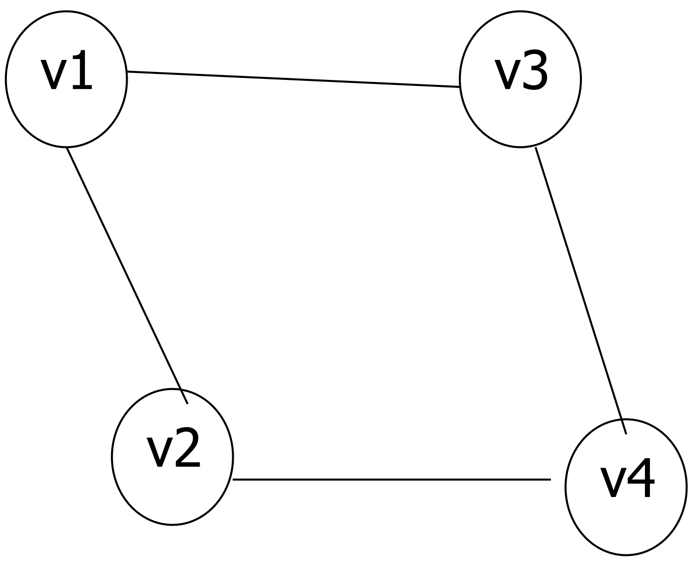
- An edge connects a pair of vertices and many have weight such as length, cost and another measuring instrument for according the graph.
- Vertices on the graph are shown as point or circles and edges are drawn as arcs or line segment.

Graph

- Example of graph:



[a] Directed & Weighted Graph



[b] Undirected Graph

Graph

- Types of Graphs:
  - Directed graph
  - Undirected graph
  - Simple graph
  - Weighted graph
  - Connected graph
  - Non-connected graph