ALGORITHMS

Chapter 1 Algorithms Analysis and Asymptotic Notations

Basic criteria 1) Input 2) Output 3) Definiteness 4) Finiteness
5) Effectiveness

Algorithm Developments Stages

Step1: Identify the problems
step2: Identify the Conditions
step3: Design logic

Strategy to design logic

- Divide & conquer
- Greedy Method
- Dynamic program
- Branch & Bound etc.

Step4 Validation (Make Assume and prove it)
Step5 · Analysis

The process of comparing algorithms rate of growth with respect to time, space, number of register, bandwidth etc is called analysis

Analysis are of two types

1. Posterior Analysis
2. Priory Analysis

Different between

Priory Ana.	Postriory Ana.
• Analysis is done before executing	• Analysis is done after executing
• It is independ of OS, System or chitecture, CPU	• It is dependent of OS, System Architecture CPU
• It provide estimated values	• It provides exact values
• It provide uniform values	• It provides non-uniform values

Step6: Implementation

Step7: Testing and debugging.

Analysis

The "Analysis" deals with performance evaluation (Complexity Analysis)

Types of Analysis / Behavior / Algorithm

* Worst Case

- Provide an upper bound on running time.
- An absolute guarantee that the algorithm would not run longer, no matter what the input are.

* Best Case

- Provides a lower bound on runtime.
- Input is the one for which the algorithm runs the fastest.

* Average Case

- Provides a Prediction about the running time.
- Assume that the input is random.

Lower Bound Running Time Upper Bound.

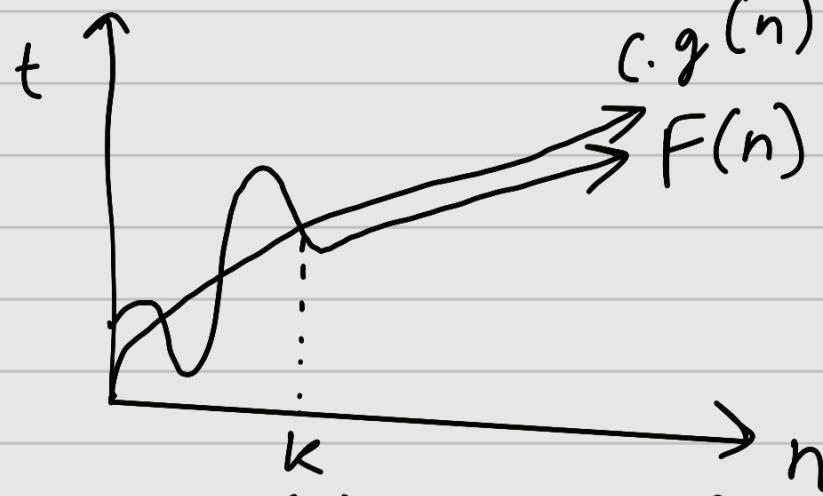
Asymptotic Notation

These are useful to represent rate of growth of two algorithm with respect to time and Space

- O (Big-OH) (upper bound)
- Ω (Big- omega) (lowerbound)
- Θ (theta) (tight Bound)
- o (Little -oh)
- ω (Little -omega)

Defination

Big oh - (O)



$f(n) = Og(n)$

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

$c > 0$

$n \geq k$

$k \geq 0$

★ Worst case
★ Upper bound (At most) (Least Upper bound)

↓

i.e closest value

Example $f(n) = 2n^2 + 2$

$2n^2 + n \leq c \cdot g(n^2)$

$2n^2 + n \leq 3n^2$

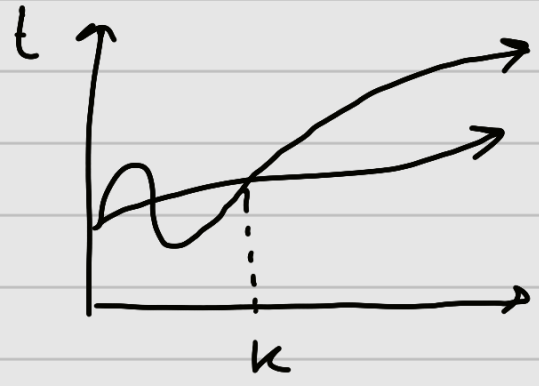
$2+1 = 3$

$n \leq n^2$

$1 \leq n$ for $n \geq 1$ this function will Hold.

2. Big Omega (Ω)

(कम से कम कितना time लगेगा)



→ Best case

→ Lower Bound (At least)

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

$F(n) \geq c \cdot g(n)$

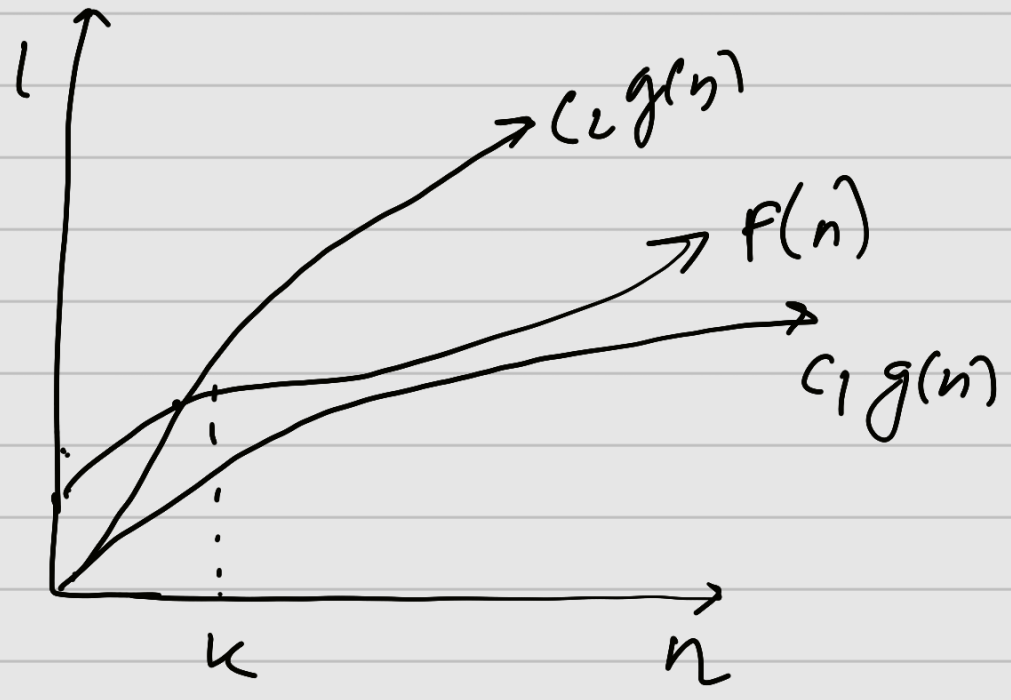
$2n^2 + n \geq c \cdot n^2$ ← (We have to choose least value close Or nearest to this)

$2n^2 + n \geq 2n^2$

$n \geq 0$

3) Theta (Θ)

(Average)



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

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

$2n^2 \leq 2n^2 + n \leq 3n^2$

Dominance Ranking:

$n! \geq 3^n \geq 2^n \geq n^3 \geq n\log n \geq n \geq \log_n^k$
$\geq \log\log(n) \geq \log n \geq 1$

Highest
$n!$
3^n
2^n
n^3
$n\log n$
n
$\log_n^k$
$\log\log(n)$
$\log n$
1
lowest

If base 2 is provided solving by placing value i.e substitution but always place large values

If $F(n) = a_0 + a_1 n + a_2 n^2 + \cdots a_m n^m$ where $a_m \neq 0$
then $F(n) = O(n^m)$
i.e Highest power

Properties of Big (OH) text Book

1. If $F(n) = O(g(n))$
then $a \cdot F(n) = O(g(n))$
for $a > 0$

2. If $f(n) = O(h(n), g(n) = O(k(n)))$
then
$F(n) + g(n) = O(\max(h(n), k(n)))$
$F(n) \cdot g(n) = O(h(n) \cdot k(n))$

Little -Oh (o)

$f(n) = O(g(n))$ iff $F(n) < c \cdot g(n)$

$\forall n \geq k, \forall c > 0$

$k > 0$

(not equals)

Little ω

Def: $F(n) = \omega(g(n))$ iff $F(n) > c \cdot g(n)$

$\forall n \geq k, \forall c > 0, k > 0$

Defination of Small oh (o) and Small Omega (ω)

The function $F(n) = o(g(n))$ iff $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

The function $F(n) = \omega(g(n))$ iff $\lim_{n \to \infty} \frac{g(n)}{F(n)} = 0$

Properties of Asymptonic Notations

1. Transitivity: this meta

$F(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$

implys $F(n) = \Theta(h(n))$

$F(n) = O(g(n))$ and $g(n) = O(h(n))$

imply $F(n) = O(h(n))$

$f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$
imply $f(n) = \Omega(h(n))$

$f(n) = o(g(n))$ and $g(n) = o(h(n))$
imply $f(n) = o(h(n))$.

$f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$
imply $f(n) = \omega(h(n))$

2. Reflexivity:

$f(n) = \Theta(f(n))$, $f(n) = O(f(n))$
$f(n) = \Omega(f(n))$

3. Symmetry:

$f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$
↑ (theta average)

4. Transpose Symmetry:

$f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
$f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

Property table

Property Notation	Symmetric	Reflexive	Transitive	Transpose
$O\ (\leq)$	x	✓	✓	$O \to \Omega$
$\Omega\ (\geq)$	x	✓	✓	$\Omega \to O$
$\Theta\ (=)$	✓	✓	✓	$\Theta \to \Theta$
$o\ (<)$	x	x	✓	$o \to \omega$
$\omega\ (>)$	x	x	✓	$\omega \to o$

Asympotically Smaller And larger terminology

Time Complexity

Time taken by algorithm to Complete

two methods

1. A Priori analysis ⟶ Independent of Machine
2. Posterior Analysis → dependent on Machine

CHAPTER 2
DIVIDE & Conquer

2.1 The General Method

* Divide: Divide the input data S into two or more disjoint subset S_1, S_2

* Recursion: Solve the sub-problems recursively.

* Conquer: Combine the Solutions for S_1, S_2 into a solution for S

The Control Abstraction of divide & Conquer

* A control abstraction means a procedure whose flow of control is clear, but whose primary operations are specified by other procedures whose precise meaning is left undefined.

The Complexity of recurrence algorithm is given by the recurrence of the form.

$$T(n) = T(1) \quad , n = 1$$
$$= aT(n/b) + f(n) , n > 1$$

2.2 Finding Maximin

Straight forward Algorithm

```
Algorithm Straight Maxmin (a, n, max, min)
{
    max = min = a[1];
    for i = 2 to n do
    {
        if (a[i] > max) then max = a[i];
        if (a[i] < max) then min = a[i];
    }
}
```

- If the elements in a[1:n] are polynomial vectors, very large numbers, or strings of characters, the cost of an elements comparision is much higher than the cost of other operations.

- Above algorithm has time Complexity 2(n-1) for best, worst and average

Case.

- We can Optimiz 2(n-1) to n-1 by adding if else in loop
- Now the best case occurs when the elements are in increasing order. The number of comparisons is n-1.
- The worst Case Occurs when the elements are in decreasing order i.e 2(n-1).
- Average number of Comparisions is $\frac{3n-1}{2}$.

Now DANDC Solution (Divide and Conquer Solution)

a[1:n] ← array
Parameter i and j are integer $1 \le i \le j \le n$.

Algorithm

```
MaxMin(i, j, max, min)
{
    If (i == j) then max = min = a[i];
    else if (i = j-1) then
    {
        if (a[i] < a[j]) then
```

```
{
        max = a[j];
        min = a[i]
    }
else
    {
        max = a[i];
        min = a[j]
    }
}
else
{
// If array is not small, divide P into Sub-problem //
    mid = [(i+j)+2]
// Solve the Subproblem
    MaxMin (i, mid, max, min)
    MaxMin (mid+1, j, max1, min1);
// Combine the Solutions
    If (max < max1) then max = max1
    If (min > min1) then min = min1
}
}
```

Merge Sort

Algorithm

MergeSort(arr[], l, r)

If r > l

1. Find the middle point to divide array in to two halves

   middle = $m = l + (r-l)/2$

2. Call merge sort for first halves

   mergesort(arr, l, m)

3. Call merge Sort for Second half

   Mergesort(arr, m+1, r)

4. Merge the two halves Sorted in Step 2 and 3:

   Merge(arr, l, m, r)

Quick Sort Algorithm

Basically Quick Sorts Say an elements in array is sort iff the left hand side has small element than it and right hand side has largest element. (not fast but name is quick)

Partition Algorithm

```
Partition(l, h)
{
    pivot = A[l]
    i=l; j=h
    while (i<j)
    {
        do
        { i++; }
        while (A[i] > pivot)
        do
        { j--; }
        while (A[i] ≤ pivot)
        If (i<j)
        {
            Swap(A[i], A[j]);
        }
    }
```

Find the element greater than pivot

Find an element smaller than pivot

If j is greater than i

```
    Swap(A[l], A[j]);      If not i > j
    return j;
}
QuickSort(l, h)
{
    if(l < h)
    {
        j = partition(l, h);
        QuickSort(l, h);
        QuickSort(j+1, h);
    }
}
```

QuickSort Analysis

We can say that

Time Complexity of QuickSort $O(n \log n)$

Best Case $O(n \log n)$

↖ Only if partition is in middle.

* Median

Median is the middle element of Sort list

Worst Case of Quicksort

* Partition is at begining
* Worst case time is $O(n^2)$ (If list is already Sort then also it will run)

Removing Worst Case of Quick Sort

1. Select Middle elements as pivot
2. Select Random element as pivot

Quick Sorts take Stack Size $\log n$ to n

$\log n$ ↑ Best

n ↑ Worst

Maximum Stack Space is $O(\log n)$ Quick Sort

Program for Quick

```
#include <stdio.h>

Void printArray ( int* A , int n) {
    for (int i = 0; i < n ; i++)
    {
        print ("%d ", A[i]);
    }
    printf( "\n");
}

int partition ( int *A , l, h)
{
    int pivot = A[low] // Considering starting element as pivot
    int temp;
    int i = low+1;
    int j = high;
    while (A[i] <= pivot){
        i ++                    Greater than pivot
    }
    while ( A[j] > pivot )
    { j --                      less than pivot
    }
    if (i<j) {
        temp= A[i];
        A[i] = A[j];
        A[j] = temp;
    }
```

(code is incomplete refer Harry's video)

```
Void QuickSort ( int A[], int low,
                 int High) {
    if (low < high) {
```

```
    int partitionIndex;
    PartitionIndex = partition(A, low, High)
    QuickSort ( A, low, partitionIndex-1)
    Quicksort ( A, partitionIndex+1, High)
  }
}
```

Binary Search

Algorithm

```
RBinSearch (l, h, key)
{
    if(l==h)
    { if(A[l] == key)
        return l;
      else
        return 0;
```

```
}
else
{
    mid = (l+h)/2;
    if (key == A[mid])
        return mid;
    if (key < A[mid])
        return RBinSearch(l, mid-1, key);
    else
        return RBinSearch(mid+1, h, key);
}
```

① ——

① ——

1 ($n/2$) →

or

1 ($n/2$) ——

↑ Because it is dividing

Time Complexity

$$T(n) = \begin{cases} 1 & n = 1 \\ T(n/2) + 1 & n > 1 \end{cases}$$

$a = 0$

$b = 2$

$f(n) = 1$

by Master term

$O(\log n)$

Strassen's Matrix Multiplication

Key point

* Number of column of 1st matrix is equal to number of row of 2nd matrix.

* $$C_{ij} = \sum_{k=1}^{n} A_{ik} * B_{kj}$$

$$A \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \times B \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = C \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

$m \times n \quad = \quad m \times n$

Traditional method

```
for(i=0; i<n; i++)
  {  for(j=0; j<n; j++)
       {
         c[i,j] = 0;
         for(k=0; k<n; k++)
           {
             c[i,j] += A[i,k] * B[k,j];
           }
       }
  }
```

$O(n^3)$

3 3 3

Divide and Conqun Strategy

Consider a Base Case

* let Say 2x2 is a small matrix multiplication.

Algorithm-

n = dimensions

```
MM(A, B, n)
{
  if(n <= 2)  If small
  {
    C = 4 Formulas
  }
  else (If large)
  {
    mid = n/2
    MM(A11, B11, n/2) +
    MM(A12, B21, n/2)
```

- $MM(A_{11}, B_{12}, n/2) + MM(A_{12}, B_{22}, n/2)$
- $MM(A_{21}, B_{11}, n/2) + MM(A_{22}, B_{21}, n/2)$
- $MM(A_{21}, B_{12}, n/2) + MM(A_{22}, B_{22}, n/2)$

Time complexity

$$T(n) = \begin{cases} 1 & n \leq 2 \\ 8T(n/2) + n^2 & n > 2 \end{cases}$$

$\Rightarrow \Theta(n^3)$

$\log_2 8 = 3$

$n^4 \neq n^2$

$n = 2$

Space khata hai

Recursion stack is also using

Strassen's Matrix Multiplication

Master's Theorem

$$T(n) = aT(n/b) + f(n)$$

$a \geq 1 \quad b > 1 \quad f(n) = \Theta(n^k \log^p n)$

methods

① $\log_b a$

② k

Case 1: If $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: If $\log_b a = k$

If $p > -1$ $\Theta(n^k \log^{p+1} n)$

If $p = -1$ $\Theta(n^k \log \log n)$

If $p < -1$ $\Theta(n^k)$

Case 3:

If $\log_b a < k$

if $p \geq 0$ $\Theta(n^4 \log^p n)$

if $p < 0$ $O(n^4)$

Examples

$T(n) = 27(n/2) + 1$

$T(n) = 4T(n/2) + n$

$\log_2 4 = 2$

$k = n^4$

$4 = 1$

$2 > 1$

$\boxed{\Theta(n^2)}$

$$T(n) = 8T(n/2) + n$$

$$\log_2 8 = 3$$

$$k = 1$$

$$\therefore \quad 3 > 1$$

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

$$T(n) = 9T(n/3) + 1$$

$$\log_3 9 = 2$$

$$k = n^4$$

$$k = 0$$

$$9 > 0$$

$\Theta(n^2)$

$T(n) = 4T(n/2) + n$

$\log_2^4 = 2$

$4 < 1$

$\Theta(n^2)$

$T(n) = 8T(n/2) + n\log n$

$3 > 1$

$\Theta(n^3)$

$T(n) = 2T(n/2) + n'$

$\log_2^2 = 1 \quad 4 = 1 \quad p = 0$

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

$$T(n) = 2T(n/2) + n/\log n$$

$\log_2^2 = 1 \quad k = 1 \quad p = -1$

$\Rightarrow \Theta(n \log\log n)$

Case 2.2

Examples from text books

$T(n) = 3T(n/2) + n^2$

$\log_2^3 =$

$k = 2$

$\log_2^1 < k$

$\Rightarrow \Theta(n^2)$ Case 3

$T(n) = 4T(n/2) + n^2$

$\log_2^4 = 2$

$k = 2$

$\therefore \quad p = 0$

$(n^2 \log n)$ Case 2

$T(n) = T(n/2) + 2^n$

$\log_2^1 =$

$2^n = 2^k$

$k = n$

Let us say

$1 < n$

$\Rightarrow \Theta(2^n)$

2.6 $T(n) = 2^n T(n/2) + n^n$

$a = 2^n$

is not constant here.

2.8 $T(n) = 16T(n/4) + n$

$\log_4^{16} = 2$

$k = 1$

$\Rightarrow \Theta(n^2)$

2.9 $T(n) = 2T(n/2) + n\log n$

$k = 1$

$\log_2^2 = 1$

$p = 1$

$(n \log^2 n)$ case 2.3

2.10 $T(n) = 2T(n/4) + n^{0.51}$

$\log_b^a = \log_4^2 = 0.50$

$0.50 < 0.51$

$\Rightarrow \Theta(n^{0.51})$

$T(n) = 0.5T(n/2) + 1/n$

$a < 1$

doesnot apply master theorm

2.11 $T(n) = 16T(n/4) + n!$

$\log_b a = \log_4 16 = 2$

$n^4 = (n!)$

Seems $n!$ is greater than 2 (obviously)

$\therefore \Rightarrow \Theta(n!)$

$T(n) = \sqrt{2T(n/2) + \log n}$

$= a = \sqrt{2}$

$b = \sqrt{2}$

$n = 0$

$\Rightarrow \Theta(\sqrt{n})$

$T(n) = 3T(n/3) + \sqrt{n}$

$T(n) = \Theta(n^2)$

$T(n) = 64T(n/8) - n^2 \log n$

Does not apply ($f(n)$ is not positive)

CHAPTER 3

Greedy Method

General method

*Basically we are given 'n' inputs and we have to find the optimal solution for the given input.

The Greedy method Control abstraction

```
Algorithm GREEDY (A, n)
{
    /* A(1:n) Contain the n input
                Solution ← ∅ */
    // initialize the solution to empty
    for i ← 1 to n do
        x ← Select (A)
        IF FEASIBLE (solution, x) then
        solution ← UNION (solution, x)
    endIf
    repeat
    return (solution);
}
```

A Func selects val from A and assign to X

FEASIBLE determines if the fun' is feasible

UNION combines the solution with previous solution to get final optimal solution.

3.2 Knapsack Problem

Ex

Algorithm for greedy Strategy for the knapsack problem
(Tutorials point Algorithm)

```
For i = 0 to n
    do x[i] = 0
weight = 0
for i = 1 to n
    If weight + w[i] ≤ W then
        x[i] = 1
        weight = weight + w[i]
    else
        x[i] = (W - weight) / w[i]
        weight = W
        break
return x
```

(x is array of item that accept in bags that value of x for each item is 1 so know we are return x)
(If we want to calculate this just add for loop and multiply with cost of item)

Job Sequence with deadline

* Given n process associated with each job is an integer deadline $d_i > 0$ and a profit $p_i > 0$. For any job i the profit p_i is earned if any only if the job is completed by its deadline.

* To complete Job one has to process the Job on a machine for one unit of time. Only one machine is available for processing JOb.

* A feasible solution for this problem is a subset J of jobs such that each job in this subset can be completed by its deadline. The value of a feasible solution J in the sum of the profit of the Job in J.

* A optimal solution is a feasible Solution with maximum value.

Since problem involves the identification of a subset it fits the Subset paradigm.

Example

P_1 P_4 P_3 P_2

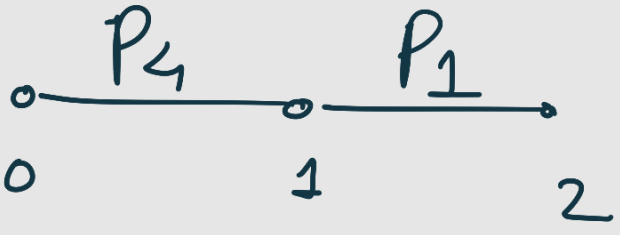
Let $n = 4$

$P1 = 100$ $d_1 = 2$

$P_2 = 10$ $d_2 = 1$

$P_3 = 15$ $d_3 = 2$ ⟵ max(2)

$P_4 = 27$ $d_4 = 1$



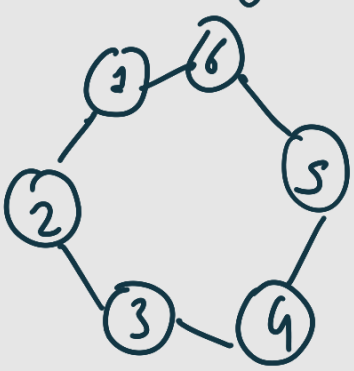
$[P_4 + P_2]$

$\Rightarrow 127$

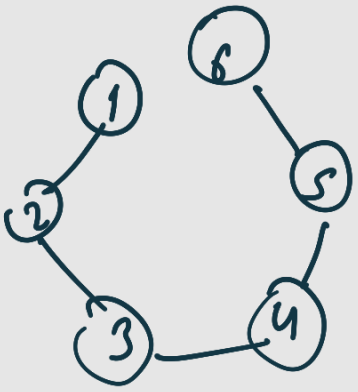
Minimum Cost Spanning tree

Basic of graph

* Let $G = (V, E)$ be an undirected connected graph. A sub-graph $t = (V, E')$ of G is a Spanning tree of G Iff it is a tree.



This is graph with cycle



This is graph but no cycle and it is a tree.

* One of the application of Spanning tree is that it can be used to obtain an independent set of circuits equation for an electric network. First a Spanning tree for the electric network is obtained.
* A minimal sub-graph is one with the fewest number of edges.
* Any connected graph with n vertices must have at $n-1$ edges and all connected

graph with n-1 edges are trees.

* If the nodes of G represents cities and the edges represent possible communication links connecting two cities, then the minimum number of links, needed to connect the n cities is n-1. The Spanning of G represents all feasible choices.

* Number of possible choices of Connected graph

For we have Given n edges out of which n' edges are possible minimum choices.

then possible choices are

$$n C_{n'} - (\text{No of cycles in graph})$$

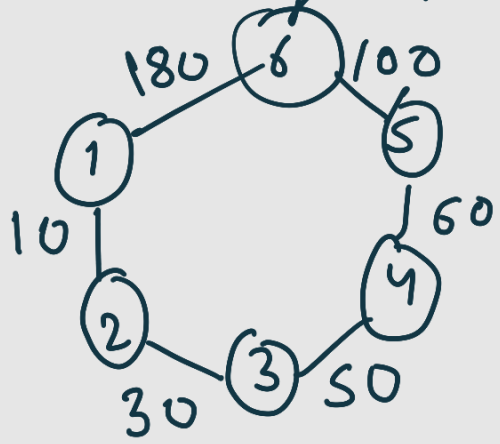
Refer Abdul bari sir video for same.

Prim's Algorithms

* A greedy method to obtain minimum Spanning tree. builds this tree edge by edge.

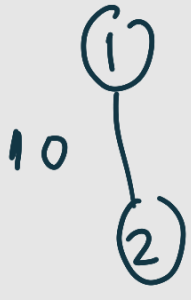
* Explain with examples

So If we are given a graph



← This is weighted graph G

* Prims algorithm Say to forms a minimum Spanning tree at first consider the Smallest edge of all given set of edges

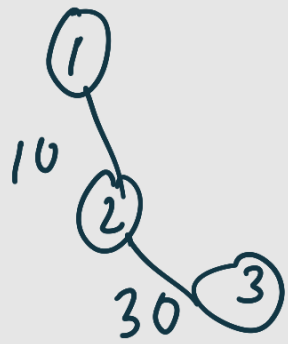


2 →

* The find the next smallest but the smallest are Selected from vertices that already taken from previous Step

* According to our example 1, 2 are the already selected vertices.

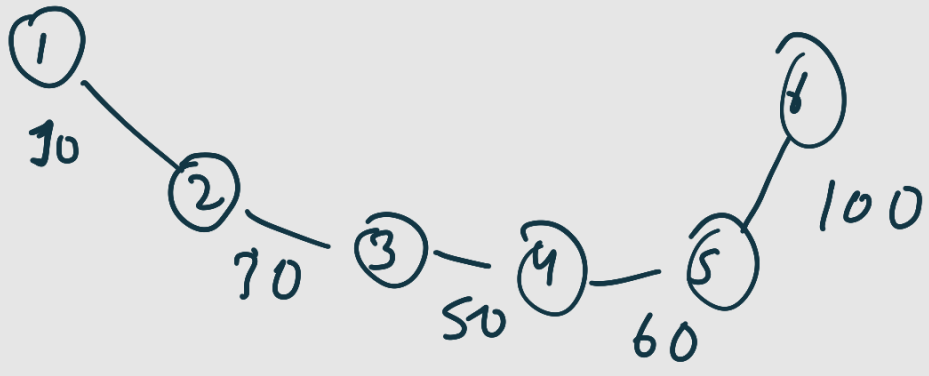
∴



repeat ② until there is n-1 edges

Here n is number of vertices (obiously)

*



↑

This our minimum cost spanning tree according to prim's Algorithm.

Sudo code for Prims Algorithm

↓

```
Prim(E, cost, n, t)
[
  Let (k, l) be an edges of minimum cost in E;
  mincost = cost[k,l]
  t[1,1] = k
  t[1,2] = l
  For i = 1 to n do
    if (cost[i,1] < cost[i,k])
        then near[i] = l;
    else near[i] = k;
  near[k] = near[l] = 0
  For i = 2 to n-1 do
    [
      Let j be an index such that near[j] ≠ 0
      and
      cost(j, near[j]) is minimum;
      t[i,1] = j
      t[i,2] = near[j]
      mincost = mincost + cost[j, near[j]];
      near[j] = 0
      For k = 1 to n do
        if (near[k] ≠ 0)
            and (cost((k, near[k]) > cost[k,j])
```

```
        then
            near[k] = j;
    }
    return mincost;
}
```

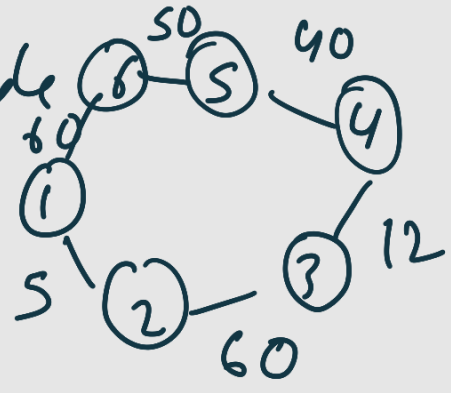
This Algorithm takes time $O(n^2)$ where n is the number of vertices in the graph G.

Kruskal's Algorithm

- In kruskal's Algorithm the optimization criteria are that, the edges of the graph are considered in non-decreasing order of cost.

- The set of edges so far selected to the Spanning tree be such that it is possible to complete t into tree. Thus t may not be a tree at all stages in the algorithm.

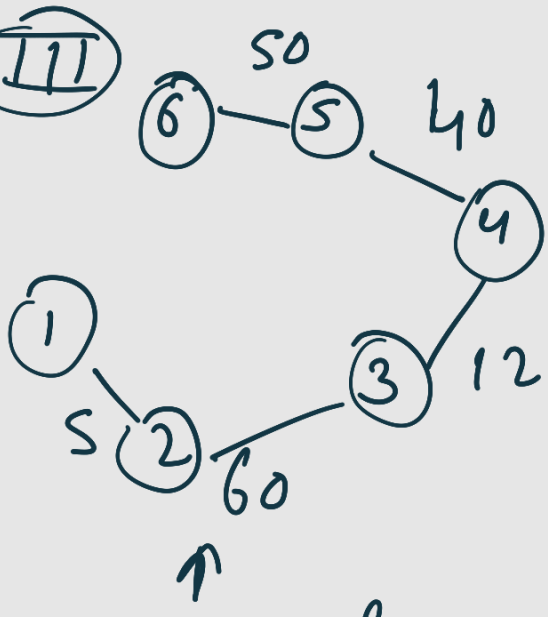
Example



Step I select minimum

Step II select next minimum

repeat III



↑ required result.

Algorithm

```
Kruskal(E, cost, n, t)
{
  Construct a heap out of the edges
   cost using
  Heapify;
  for i = 1 to n do
     parent[i] = -1;
  i = 0;
  mincost = 0.0;
  while ((t < n-1) and heap not empty))
     do
        {
           Delete a minimum cost edge
           (u,v) from the heap and
          reheapify using Adjust;
          j = Find(u); k = Find(v);
          if (j ≠ k) then
          {
              i = i+1
              t[i,1] = u;
              t[i,2] = v;
```

```
            mincost = mincost + cost[u,v];
            Union(j,k);
        }
    }
    If (i ≠ n-1) then write ("NO spanning tree")
    else return mincost
}
```

Analysis of Algorithm

- If edges are maintain in minheap then next edges to consider can be obtain in $O(\log |E|)$ time, where E is the set of edges of Graph G.
- Construction of heaps takes $O(|E|)$ time

$\therefore$ $O(|E| \log |E|)$

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

Optimal Merge Pattern

* So Basically we are given some sorted list with diff Size and we have to merge them optimal so that time required to merge is minimum.
* Since this problem calls for an ordering among the pair to be merged it fits the ordering paradigm.
* Merging an n-record file and an m-record file required (n+m) record moves.
* At each steps, merging the two smallest size file.
* Two-Way merging pattern (each steps involves merging of two files).
* The two-way merge pattern can be represented by binary merge trees.
* The leaf nodes are drawn as Squares and represent the given files. These nodes are called external nodes. The remaining nodes are drawn as circle and are called Internal nodes.

* Each internal nodes has exactly two children and it represents the file obtained by merging the files represented by its two children. The number in each nodes is lenght (i.e number of records) of the file represented by that node.

Algorithm to generate two Way merge tree

```
treenode = record
           {
              treenode *lchild;
              treenode *rchild;
              integer weight;
           }

Algorithm Tree(n)
    {
      for i = 1 to n-1 do
          {
            pt = new treenode;
              // Get a new tree node
          (pt → lchild) = Least(list);
```

```
// Merge two trees with
(pt → rchild) = Last(list)
// smallest lenghts.
(pt → tweight) = ((pt → lchild) → weight)
                 + ((pt → rchild) →
                    weight) ;
Insert(list, p)
} // end of for loop
return Last(list);
   //tree left in first is merge tree.
}
```

*** The greedy rule to generate optimal merge tree is

At each step choose a sub-tree with least lenght of merging.

Huffman Code

Suppose

A message

X = BCCDAABCCABCAAC

↑

To send this string as message we required ASCII code for this

ASCII code is 8-bit charactor

So total lenght of messeg will be

8 x lenght(x)

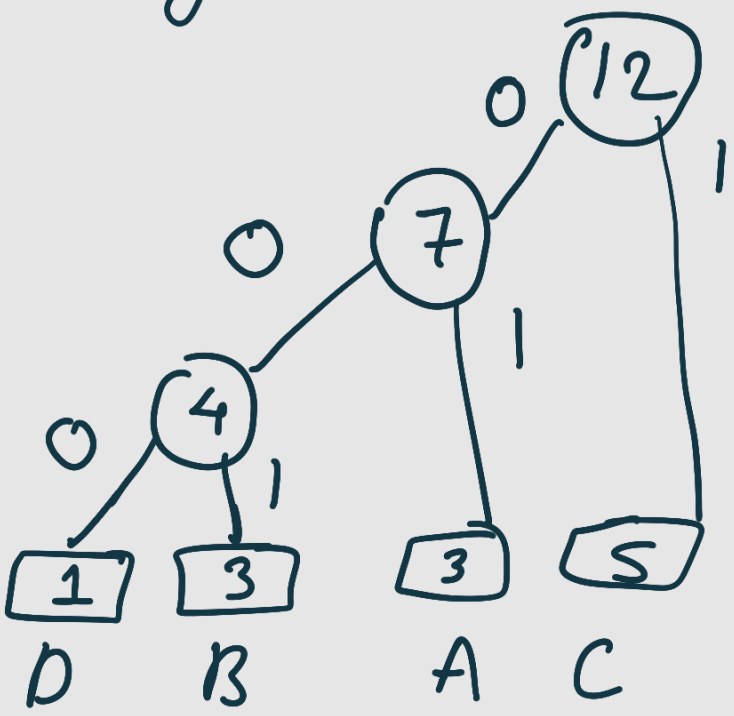
which is greater.

So we reduce this using Huffman coding

Create table

Char	Count
A	5
B	3
C	6
D	1

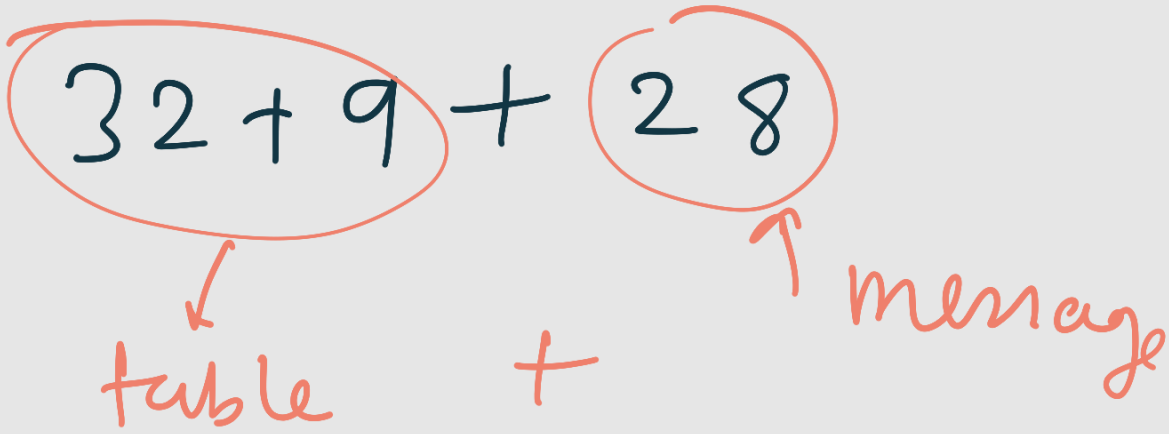
We are going to create 2-way merge list



Consider 12 as root right side represents 1 and left side represent 0.

Now we can create code

Char	code	count	number of bit
A	01	5 x (2) = 10	
B	001	3 x (3) = 9	
C	1	6 x 1 = 6	
D	000	1 x 3 = 3	
4 x 8	9		28 bit



=

69 bit for total Message

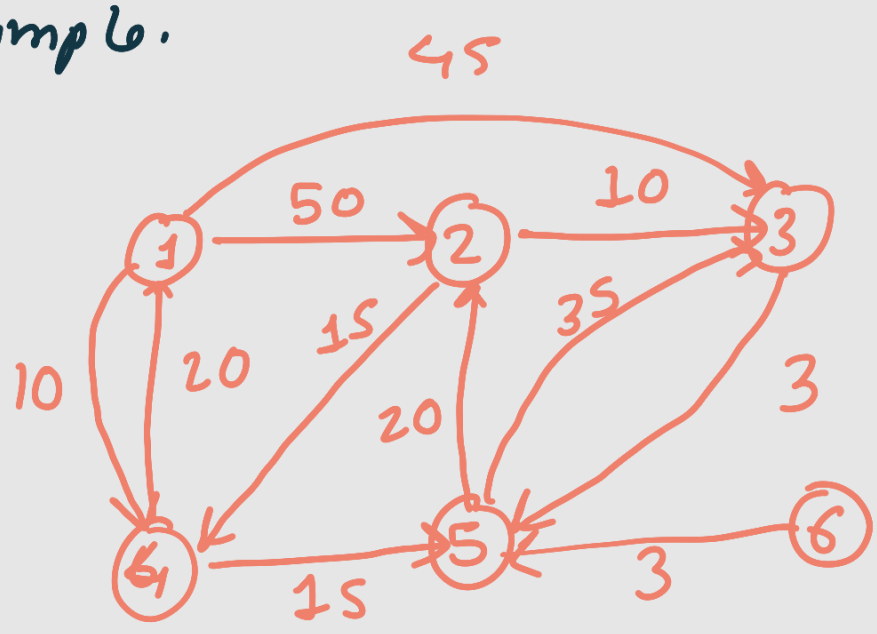
This is Humans codes

Single Source Shortest Paths (Dijkstra Algorithm)

* Ordering Paradigm
* Suppose we have given a graph G which directed as well as weighted graph.

* If we have already constructed + shortest paths, then using this Optimization measure, the next path to be constructed should be the next shortest minimum length path.

For better understant Consider the example.



Suppose we start with Vertex 1

	1	2	3	4	5	6
1	1	50	∞	10	∞	∞
4	0	50	∞	10	25	
5	0	50	∞	10	25	∞
2	0	50	60	10	25	∞
3	0	50	60	10	25	∞

Here we apply relaxation in each instance

Consider above example

If we select 1 the nearest and minimum cost is 4 from 4 5 is minimum cost and from 5 again 2 is minimum cost and from 2, 3 is minimum cost

We applied relaxation here

Relation

$$if\ (d[u] + c(u,v) < d[v])$$

$$d[v] = d[u] + c(u,v)$$

d = distance

c = cost between two edge i.e weight.

Dijkstra Shortest source path Algorithm

```
ShortestPaths(v, cost, dist, n)
{
    for i = 1 to n do
    { // initialize S
        S[i] = false;
        dist[i] = cost[v, i]
    }
    S[v] = true;
    dist[v] = 0.0 ; // Put v in S
    for num = 2 to n-1 do
    {
    Choose u from among those
      vertices not in S such
      that dist[u] is minimum:
        S[u] = true ; // Put u in S
    for (each w adjacent to u with
            S[w] = false)
        do
            //update distance.
            if (dist[w] > dist[u] +
                          cost[u,w]))
                then
```

```
            dist[w] = dist[u] + cost[u,w];
        }
    }
```

Gate Smashers Algorithm for Dijkstra

```
Dijkstra (Graph, source)
    {
     Create vertex Set Q
    for each vertex v in graph
     dist[v] = ∞                    ] O(V)
     add v to Q  (Build Heap)       ] O(V)
    dist[source] = 0
    while Q is not empty
     u = Extract-min[Q]             -(V log V)
    For each neighbour v of u
      Relax (u, v)                  -(E log V)
```

Analysis

$$O(V) + O(V) + O(V \log V) + O(E \log V)$$

Biggest

Also

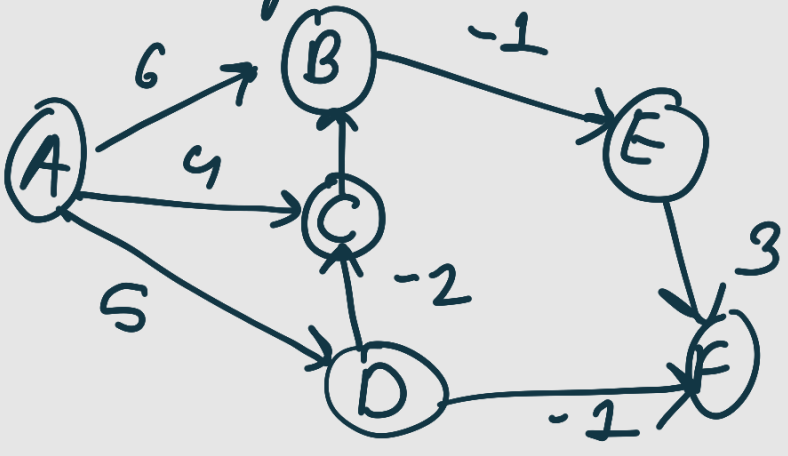
$$\therefore O(E \log V)$$

Single Source Shortest Path

Bellman ford

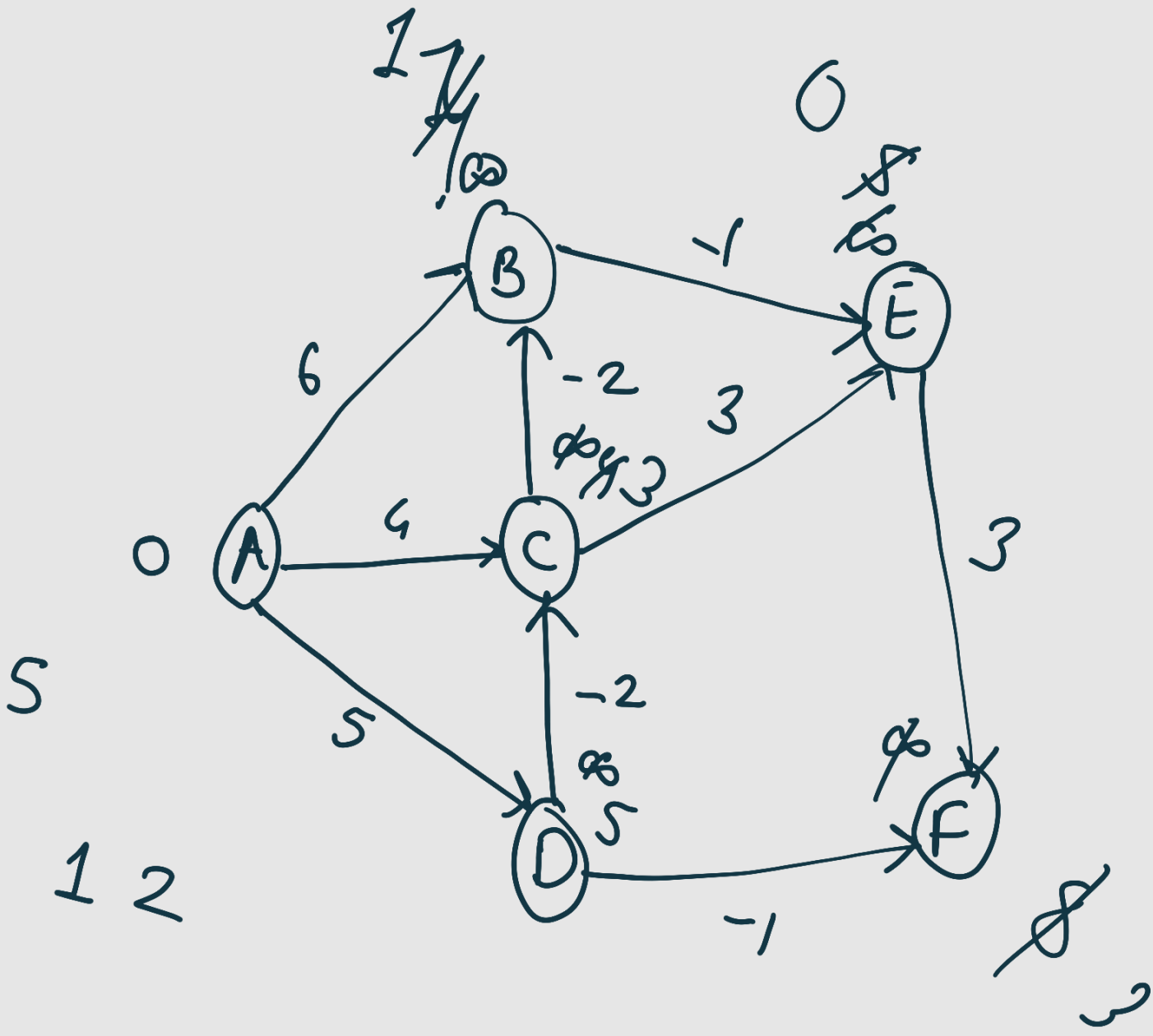
So Basically if graph is given we are relaxing all the edges $n-1$ times. If n is number of vertices.

For Example



Consider edges

5



Theory

- The Bellman ford algorithm uses relaxation to find single source paths on directed graphs that contain negative weight edges. The algorithm will also

detect if there are any negative weight cycles (such that there is no solution)

Basically the algorithms works as follows:

* Initialize the initial distance as 0 and other vertex as ∞

this takes $O(V)$ times

* Loop $|V|-1$ times through all edges checking the relaxing condition to compute minimum distance

$O(|V-1|) \cdot O(E)$

∵ Because we calculate distance using edge value and vertex.

∴ $O(VE)$

* Loop through all edges checking for negative weight cycle which occurs if any of the relaxation condition fail

$\Rightarrow O(E)$

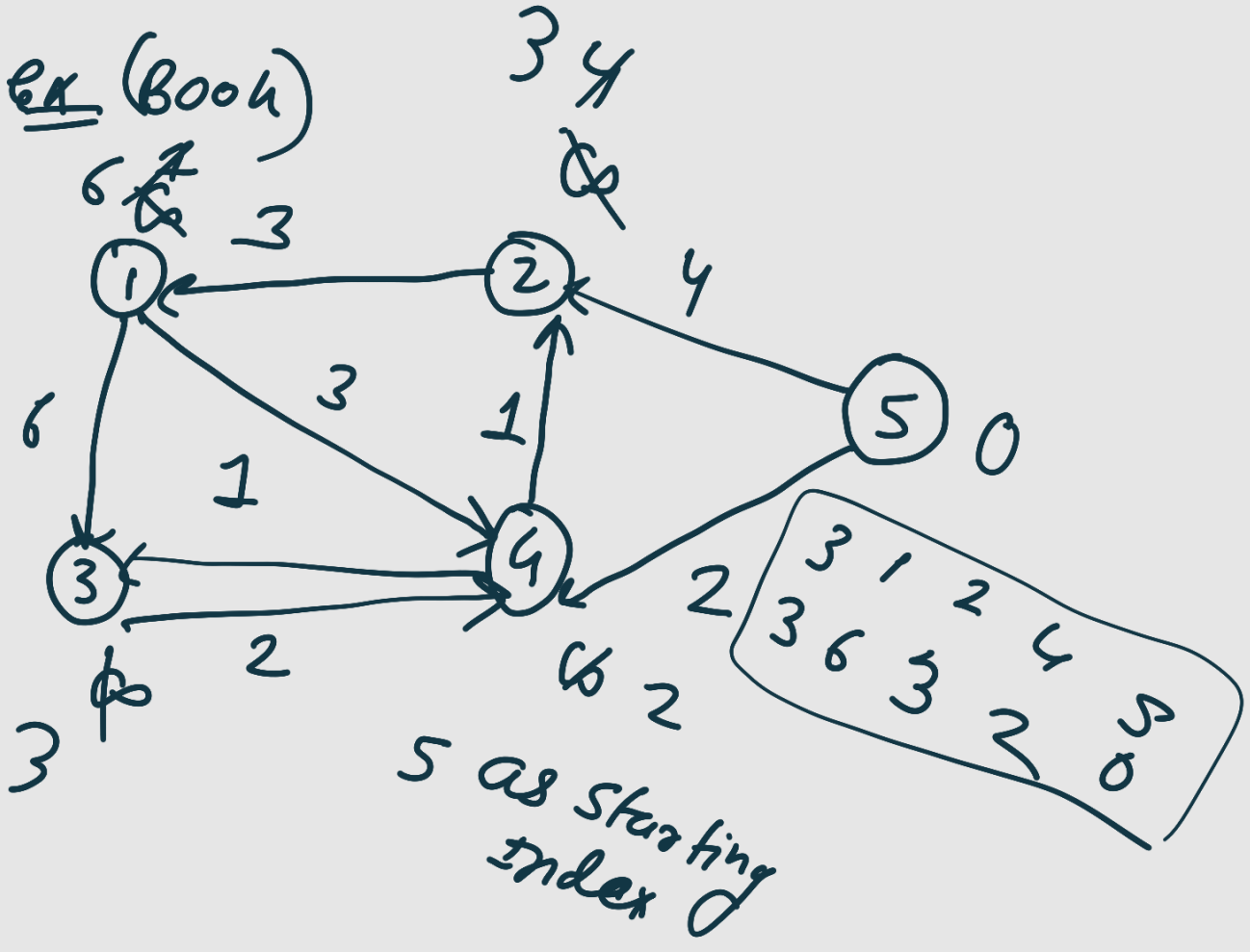
∴ Total time complexity $O(E+OV-1E)$

$\therefore O(OV)$

Note:

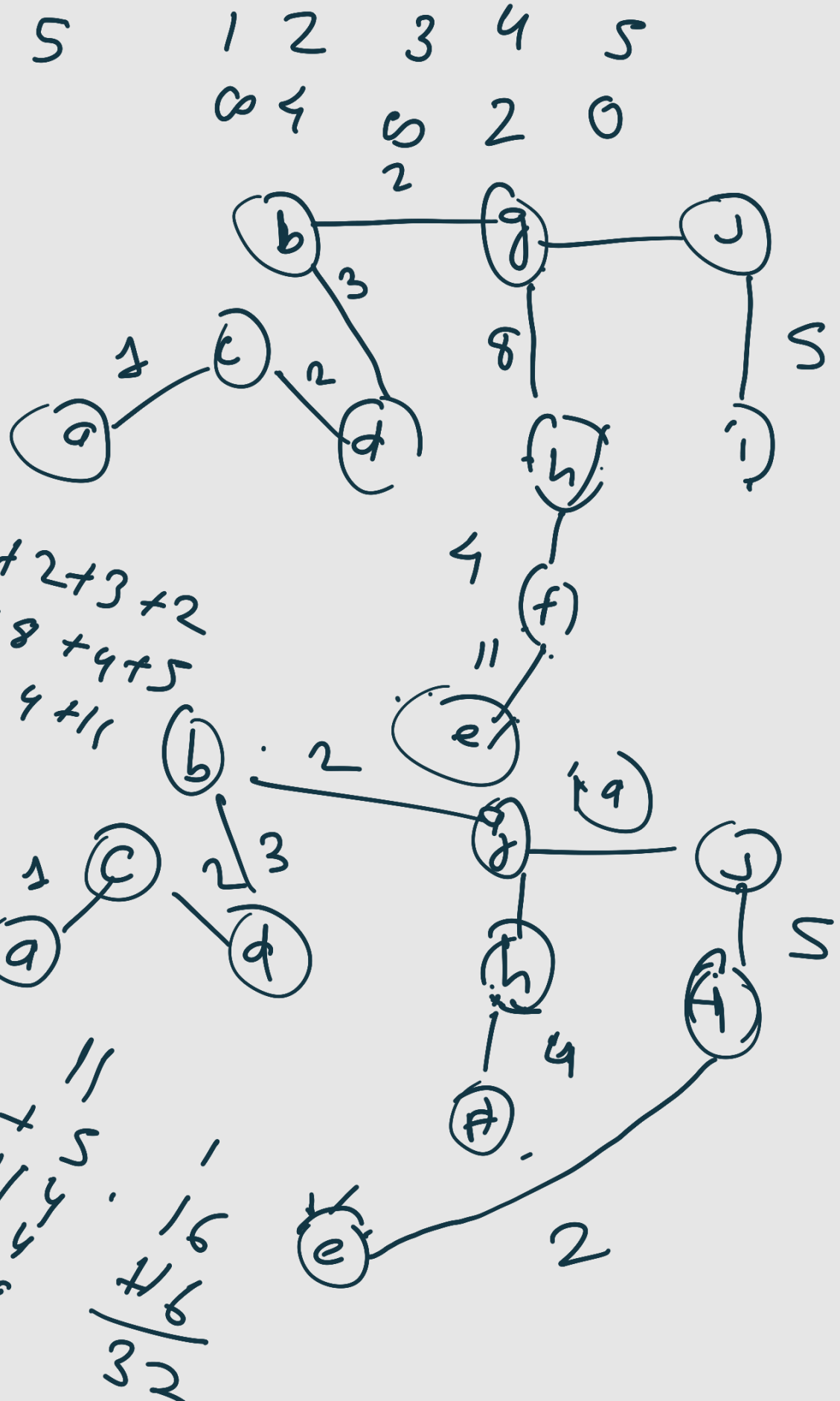
If the graph is DAG (Directed Acyclic graph) we can make Bellman fords algoritnm more efficient by first topologically sorting $O(V+E)$.

This because we are running it E times as there is no cycle we are traversing and modifying only one time.



5 Vertices

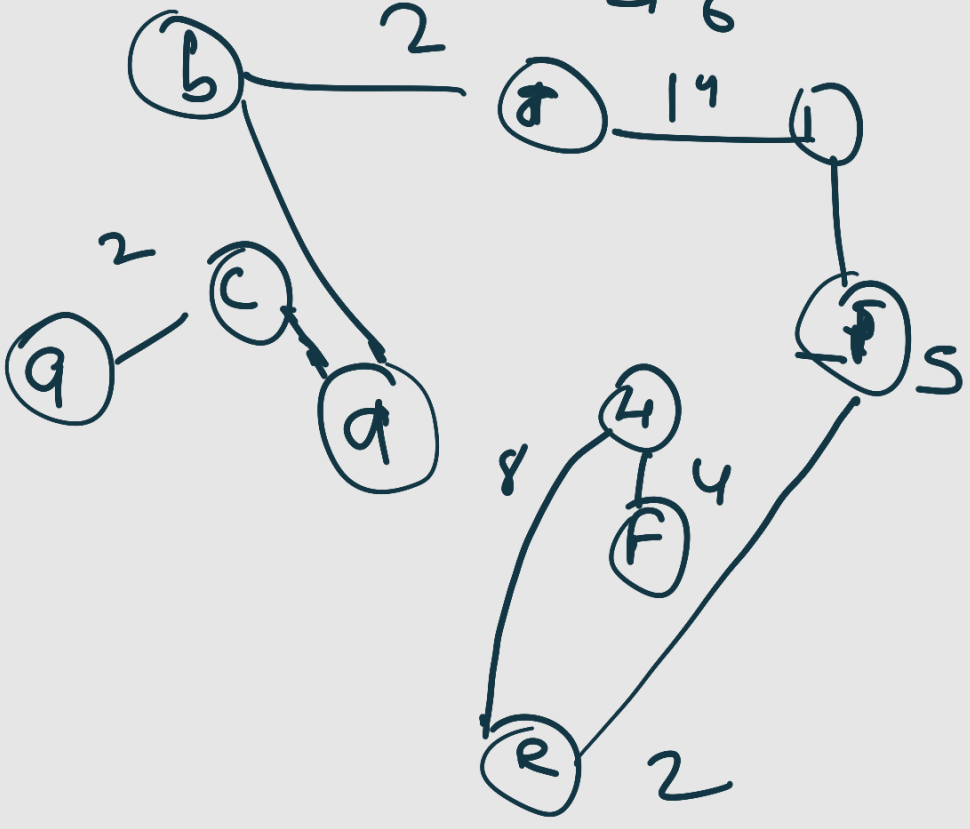
4 iteration (n-1)



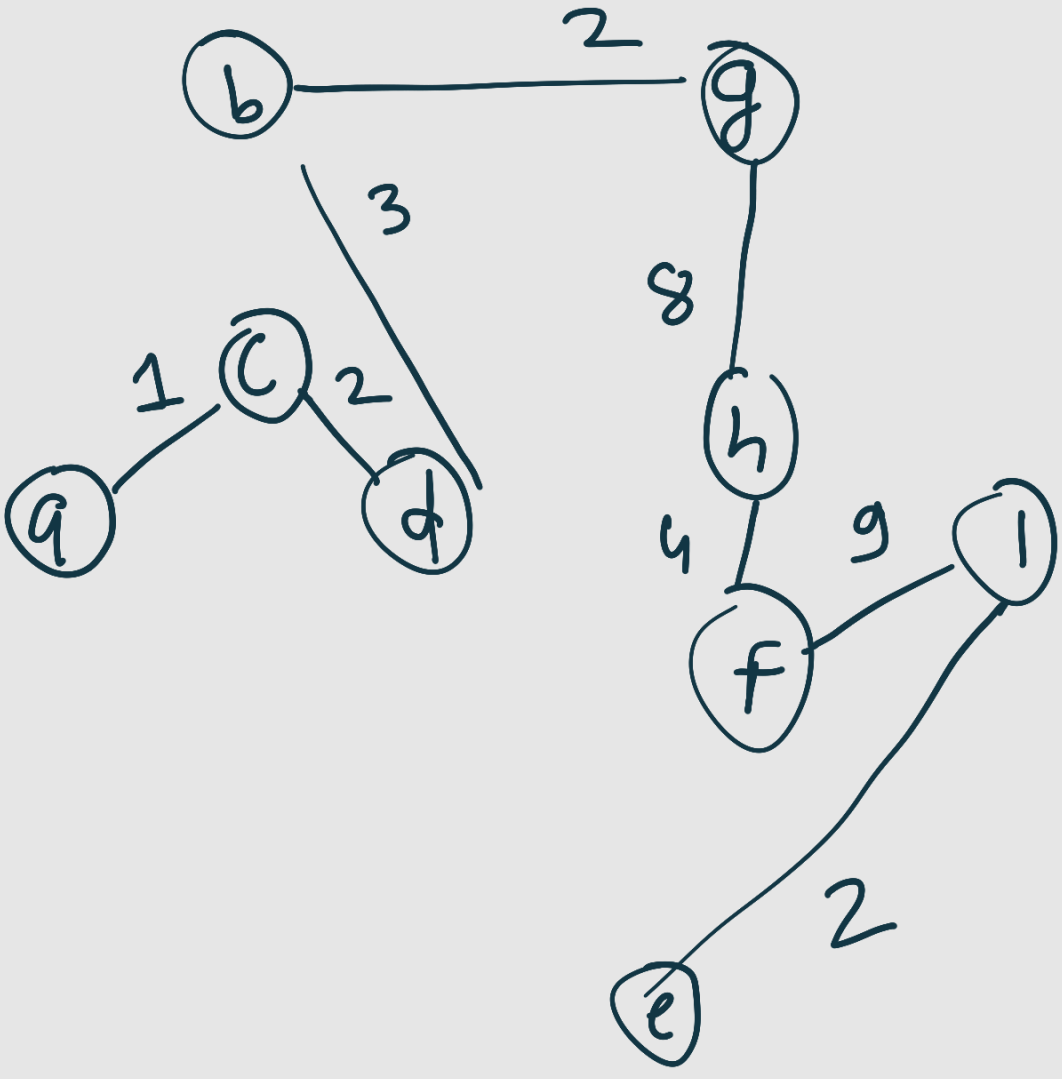
×2
×2
×1

1+2+3+2+19+5+4+8
+2

32
26 + 8 + 8 + 19 + 5
8
50 22 1
5
27 27
9
46



2+1+3+2+19+5+2
+8+4



$8+4+2+2+3+2+1$

$+9$

$17+9+1$

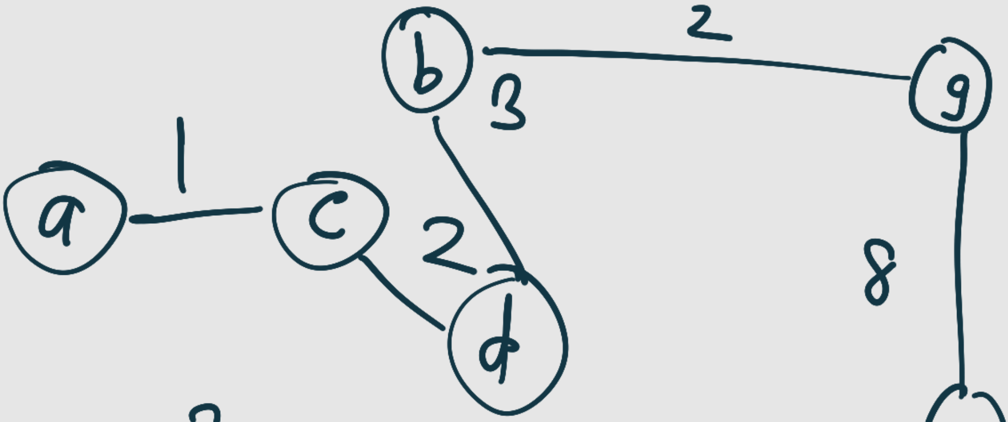
27

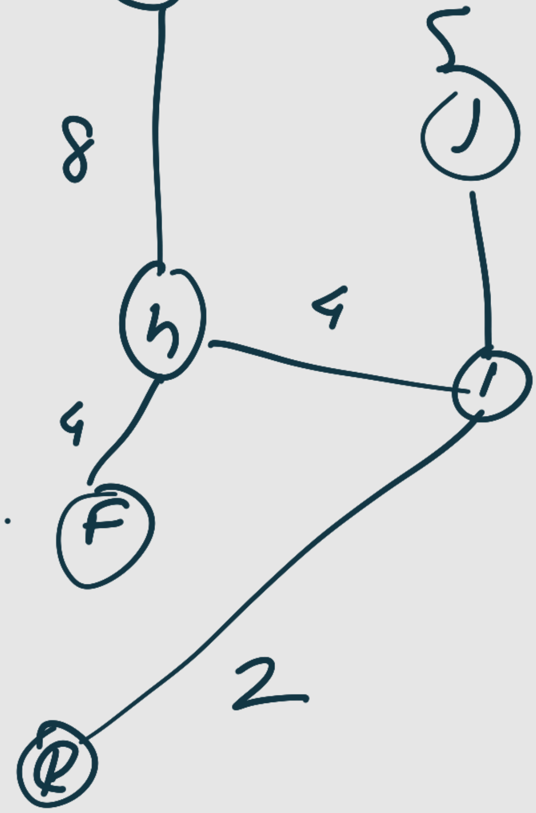
1

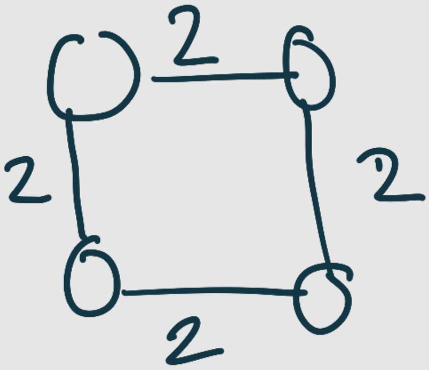
27

$+5$

32

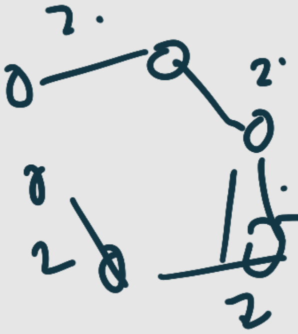






2+2+2

6



1 + 2 + 3 + 2 + 8 + 4 + 5 + 4 + 2

2 + 2 + 2 + 2 + 2 10

16 + 8 + 2 + 5

2(n)-2

2(4)-2

18
8

26
+5
---

7 5

31

* key points from Answer

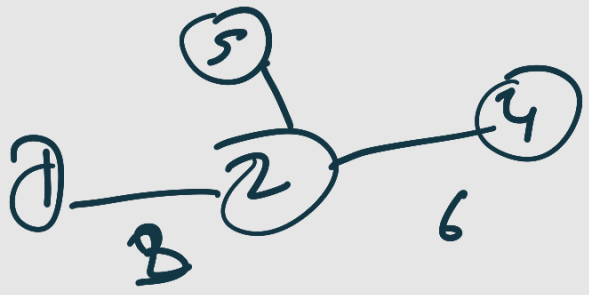
* Every minimum Spanning tree of G must contain min e
* If max e is in minimum Spanning tree then its removed must disc G.
* G has a unique minimum Spanning tree.

If e has weight w in Spanning tree

* then is a minimum Spanning tree containing e
* If e is not in a minimum Spanning tree T, then the cycle formed by Adding e to T, all edges have the same weight-
* Every minimum Spanning tree has edge e weight w.

$(1+2)+(2+3)+(3+4)$

2 5 7



$6(4)-11$

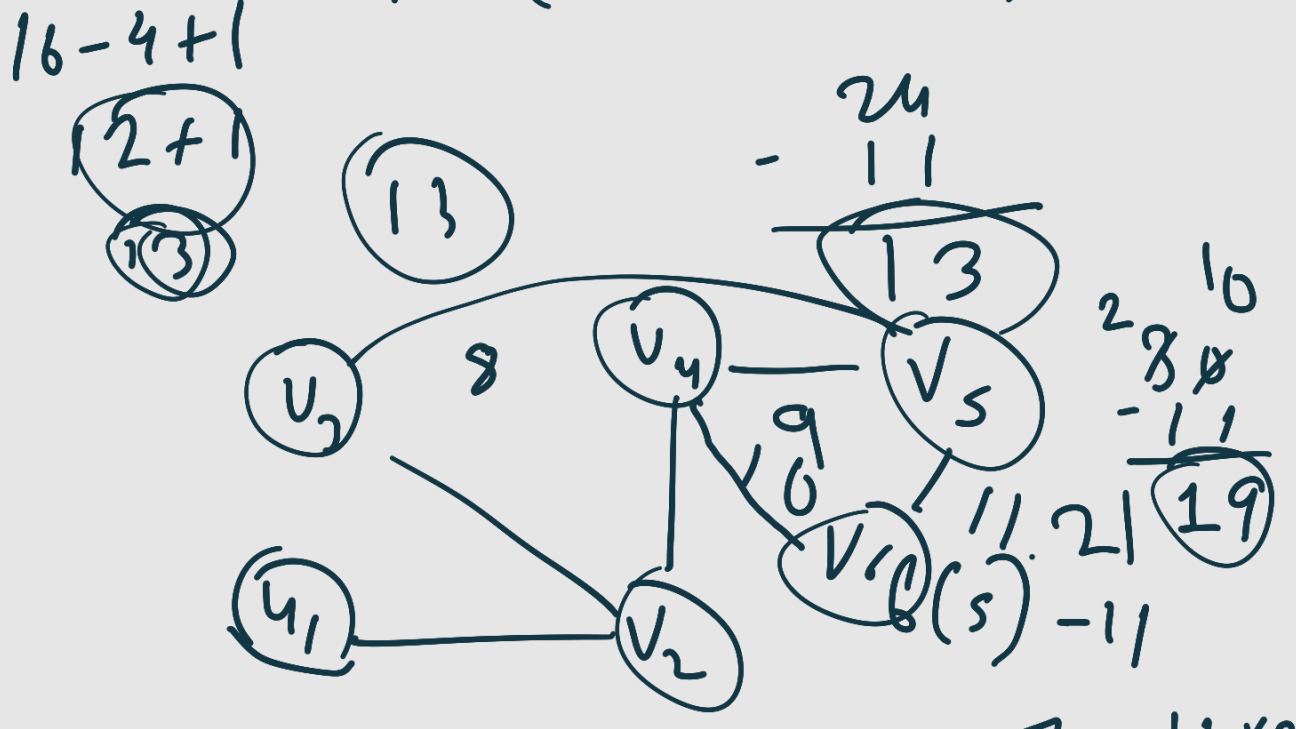
$24-11$

$n=4$

$\frac{1}{12}(11(4)^2-5(4))$

$\frac{1}{12}(11\times16-20)$

$16-4+1$



$21-3+4+6+8$

$5-5+1$

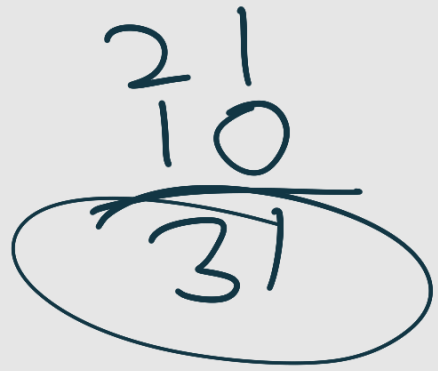
111 7 11x25

111 +6 -25

13

+8

21


21
10
31

2
1
1.0
0
0
0.6
1
1
0
0.24
1
0
0.13
1
0.36
0
1
0.05
0.09
0.11
0.17
0.19
0.40
b
d
f
c
a
e

a = 011

b =


1
1/2
1/4
1/8
1/16
1/32
1/32
1/16
1/8
1/4
1/2
1
b=01
f=1

Chapter 4
Graph Technique Components.

Heap

Depth-first Search (DFS)

* Visit all vertices and edges of G.
* Determine whether G is connected
* Compute the connected components of G.
* Compute the spanning forest of G.

* DFS on a graph takes n vertices and m edges takes O(n+m) time.

* It can also be used to solve follow.. Prob
  - Find and report a path between two given vertices
  - Find a cycle in the graph.

Algorithm DFS(G) (old video version)

DFS uses stack to remember to get next vertex to start a search.

Rule 1: Visit adjencent unvisited vertex and mark it has visited, display it and push it in stack

Rule 2: If no adjacent vertex is found popup a vertex from stack.

Rule 3: Repeat rule 1 and 2 until stack is empty.

Main Algorithm (DFS)

```
depth First Search (v)
  {
    label vertex v as reached
    for(each unreached vertex u adjacent
                                from v)
      depth First Search(u);
  }
```

DFS Algorithm (New video version)

DFS traversal of a graph produces Spanning tree as final result i.e a graph without loops - we use stack data structure with max size of total no of vertices in the graph to implement DFS traversal of graph

Algorithm :-

Step 1: Define a stack of size total no of vertices in graph

Step 2: Select any vertex as starting point for traversal. Visit that vertex and

push it on to the stack.

Step 3: Visit any one of the adjacent vertex of the vertex which is at top of the stack, which is not visited & push it on to the stack.

Step 4: Repeat step 3. Until there are no new vertex to be visited from the vertex on top of stack

Step 5: When there is no new vertex to be visit then use backtracking and pop one vertex from the stack.

Step 6: Repeat step 3,4,5 until stack becomes empty.

Step 7: When Stack becomes empty, then produce final Spanning tree by removing unused edge from the graph.

properties of DFS

Property 1

DFS (G,V) visit all the vertices and edges in the connected component of V

Property 2

The discovery edge labeled by DFS(G,V) from a Spanning tree of the connected component of V

Analysis of DFS

- Setting/getting a vertex label takes $O(1)$ time.
- Each vertex is labeled twice
  - * once as unexplored
  - * once as visited.
- Each edge is labeled twice
  - * once as UNEXPLORED
  - * one as Discovery or BACK
- Method incidentEdges is called once for each vertex.
- DFS runs in $O(n+m)$ time provided the graph is represented by the adjacency list structure.

Recal that $\sum_v \deg(v) = 2m$

I don't understand this may be related to Exam by Ryan Hard this.

BFS (Breadth First Search)

* BFS produces a spanning tree as a final result i.e graph without loops.
* We use queue d.s with maximum size of total number of vertices in the graph to implement BFS of a graph.

Algorithm :

Step1 : Define a queue of size total number of vertices in the graphs.

Step2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the queue.

Step3: Visit all the adjacent vertices of the vertex which is at point of queue, which is not visited and insert them into the queue

Step4: When there is no new vertex to be visit from the vertex at point of the queue then delete that vertex from the queue.

Step5: Repeat Step 3 & 4, until queue becomes empty.

Step6: When queue becomes empty, then produce final spanning tree by removing unused edges from the graph.

* A BFS traversal of a graph G
  1) Visit all the vertices and edges of G
  2) Determine whether G is connected.
  3) Compute the connected component of G
  4) Compute a Spanning forest of G.
* BFS on a graph with n vertices and m edges takes $O(n+m)$ time.
* Other problem solved using BFS
  1) Find and reports minimum number of paths between two given vertices
  2) Find a simple cycle if there is one.

Properties

Notation: G_s: Connected Component of s

Property 1: BFS(G, s) visits all the vertices and edges of G_s

Property 2: The discovery edge labeled by BFS(G, s) form a spanning tree T_s and G_s

Property 3: For each Vertex V in L_i
  * The path of T_s from s to v has

i-edges

* Every path from s to v in G_s has atleast i edges.

Analysis

- Setting / getting a vertex / edge label takes $O(1)$
- Each vertex is labeled twice
  - once as unexplored
  - once as unvisited
- Each edge is labeled twice
  - once as unexplored
  - once as Discovery or cross
- Each vertex is inserted once into a sequence L_i
- method incident edges is called once for each vertex.
- BFS runs in $O(n+m)$ time provided the graph is represented by the adjacency list structure.
- Recall that $\sum_v \deg(v) = 2m$

Pseudo Code for BFS

```
BFS(G)
{
  for all v ∈ G.vertices
    {
      setlabel(v, UNVISITED)
    }
  for all e ∈ G.edges
    {
      setlabel(e, UNDISCOVERED)
    }
// Now we have to choose arbitary
   start vertex.
for all s ∈ G vertices
 if (s.label == unvisited)
{  List.addEnd(s);
   setlabel(s, VISITED);
 while (List.Notempty())
    {  v = List.removefront();
     for all  e ∈ incident on v
         { if (e.label == undiscovered)
            {
              // checking adjacent vertex last say w is adjvertex
               w = adj_vertex(v, e);
               if(w.label = visited)
                  {
                    setlabel(e, CROSS)
                  }
               if (w.label = UNVISITED)
                  {
                    setlabel(e, DISCOVERED)
                    setlabel(w, VISITED);
                    List.addEnd(w);
                  }
            }
         }
    }
}
```

DFS - Pseudo code

i/p - Graph Aim: Traverse

Initial: All vertices = unvisited
All edges = undiscovered

Choose arbitrary start vertex

Visit vertex

↓

Current vertex

↳ undiscovered edg ?

↳ found

↳ unvisited vertex ?
Edge = Disconnected
Visit VERTEX

↳ visited vertex ?
edge = BACK edge.

not found ←

↳ Back track

Algorithm

```
DFS(G)
{
  for all V ∈ G.vertices
  {
    Setlabel ( V, UNVISITED)
  }
  for all e ∈ G.edges
  {
    setlabel (e, UNDISCOVERED)
  }
  // Chose arbitary start

  for all V ∈ G.Vertices
    If (V.label == unvisited)
      { visit (V, G);
```

```
    }
}

Visit (v, G)
{
  SetLabel(V, VISITED);
// go through all incident edges to V
  for e ∈ V.incidentEdges
    {
      if (e.label == UNDISCOVERED)
        {
          w = adj-Vertex(v, e)
          If (w.label == 'UNVISITED')
            {
              SetLabel (e, DISCOVERED)
              Visit (w, G)
            }
          If (w.label == VISITED)
            {
              SetLabel (e, BACK)
            }
        } // If edges is undiscovered
    }
```

DFS Vs BFS

Application	DFS	BFS
Spanning Forest	✓	✓
Connected Components	✓	✓
paths Cycle	✓	✓

Shortest path ✓

Biconnected Components ✓

Back edge (v, w)

* w is an ancestor of v in the tree of discovery.

Cross edge (v, w)

* w is in the same level as v or in the next level in the tree of discovery edges.

Applications of DFS

- Unidirected graph

  Connected Component, articulation pt, bridges, biconnected Components

- Directed graph

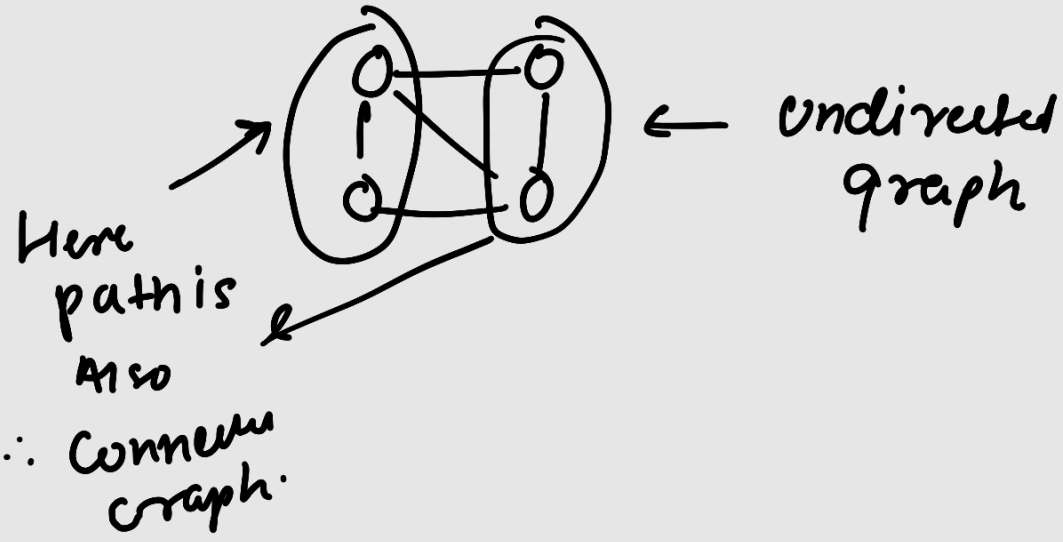
  Cyclic / acylic graph

  Topological sort

  Strongly Connected Components

Connected Components

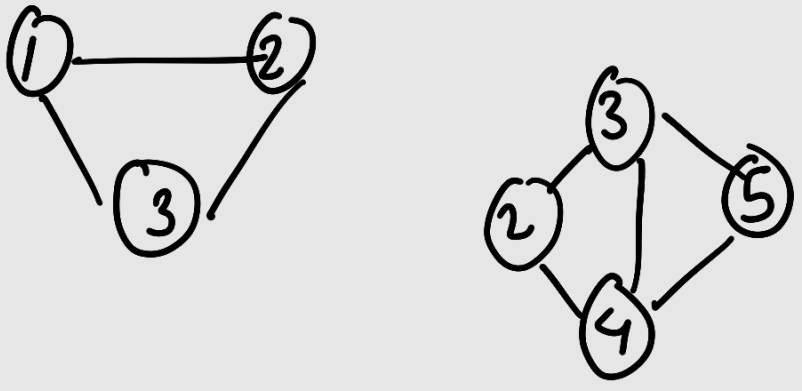
An undirected graph is called a connected graph if there is a path between any two two vertices.



Connected Component

A connected component is a subgraph in which any two vertices r connected two each other by paths & which is connected to no addition vertices of super graph (main graph)

Eg.



Bi-Connected Components

A graph is biconnected if it contains no 'articulation' point.

↓

is any vertex of graph G whose removal results in disconnected graph.

Book defination

- Biconnected Component of a graph G

* A maximal Biconnected Sub-graph of G.

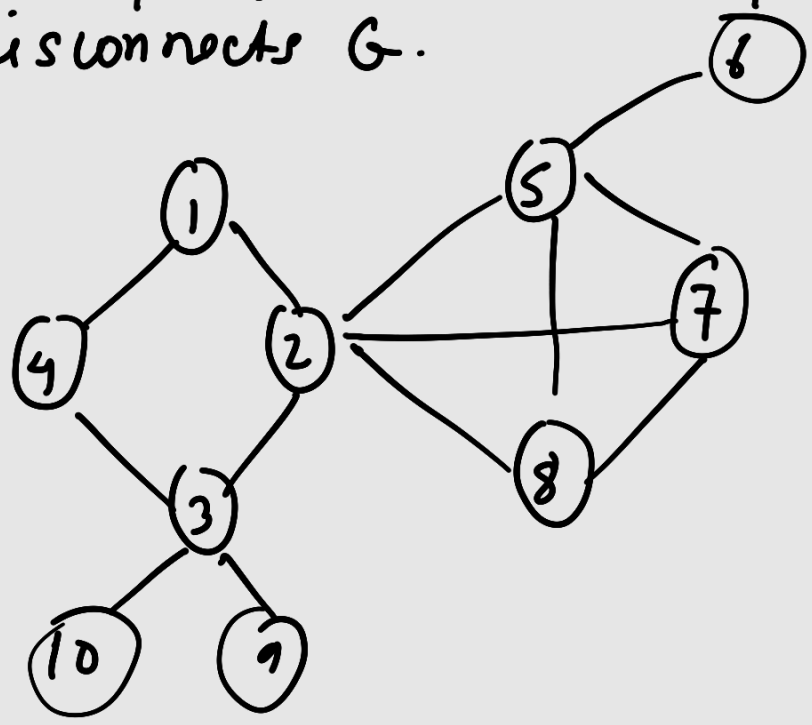
or

* A sub-graph consisting of a seperation edges of G and its end vertices.

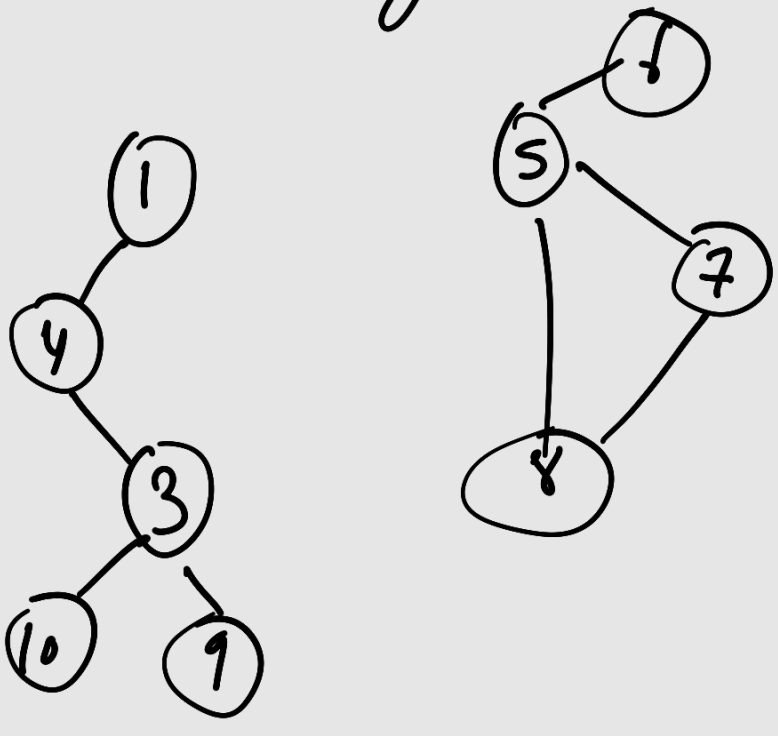
* A vertex V in a connected graph G is an Articulation point if and only if the deletion of vertex V together with all edges incident to V disconnects the graph into two or more non empty components.

* A graph G is Biconnected $\Leftrightarrow$ it contain no Articulation points.

* A Bridge of G is an edge whose removal disconnects G.



deleting 2



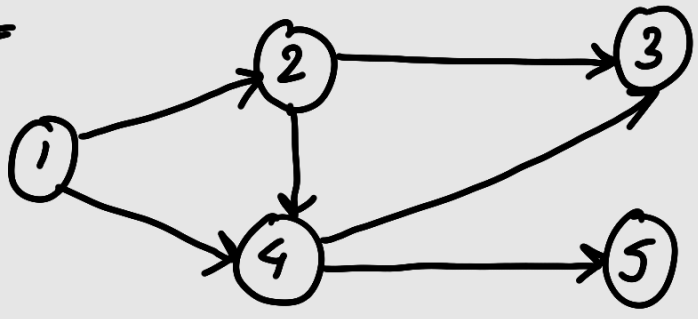
4.3 Topological Sorting

It is a linear ordering of its vertices such that for every directed edge uv for vertex u to v u comes before vertex v in the ordering.

Graph should be DAG
(Directed Acyclic graph)

- Every DAG will have atleast one topological Ordering

Ex



How find Topological Order for this?

Step1 Check Cycle If yes not possible for TS

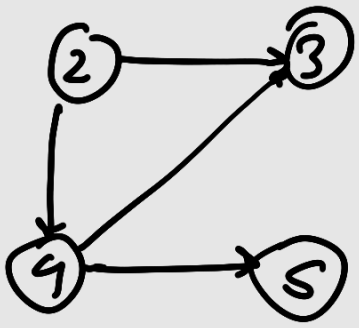
Step2: Calculate indegree of Every vertex

	Indegree
1 →	0
2 →	1
3 →	2
4 →	2
5 →	1

Step3: Choose vertex having in degree zero

for ex Select Vertex 1 fm above

Step 4: Delete Out going edges of selected Vertex

$\therefore$ G = 

Step 5: update in degree

2 → 0

4 → 1

3 → 2

5 → 1

Step 6 repeat 3, 4, 5

Step 7 Get topological order

Ex. 1 2 4 3 5 — 1 posibility

1 2 4 5 3 — 2 posibility

Algorithm for topological Sorting

```
TopologicalSort(G)
    H ← G      // Tempory copy of G
    n ← G.numVertices()
    while H is not empty do
    Let V be a vertex with no outgoing edges
    Label V ← n
    n ← n - 1
    Remove V from H
```

Run time : $O(n+m)$

Simulate the Algorithm Using depth-First Search

topologicalDFS(G)

```
Input DAG G
output topological ordering G
    n ← G.numVertices()
    for all u ∈ G.vertices()
        setLabel(u, UNEXPLORED)
    for all e ∈ G.edges()
        setLabel(e, UNEXPLORED)
    for all v ∈ G.vertices()
        if getLabel(v) = UNEXPLORED
            topologicalDFS(G, v)

Algorithm topologicalDFS(G, v)
    Input graph G and a start vertex
v of G
output labelling of the vertices of G in
the connected component of v.

setLabel(v, VISITED)
    for all e ∈ G.incidentEdges(v)
        if getLabel(e) = UNEXPLORED
```

```
w ← opposik(v, e)
if getLabel(w) = UNEXPLORED
    setLabel(e, DISCOVERY)
    topologicalDFS(G, w)
else
    {e is forward or cross edge}
Label v with topological number n
```

∴ **Run time : O(n+m)**

Priority Queue

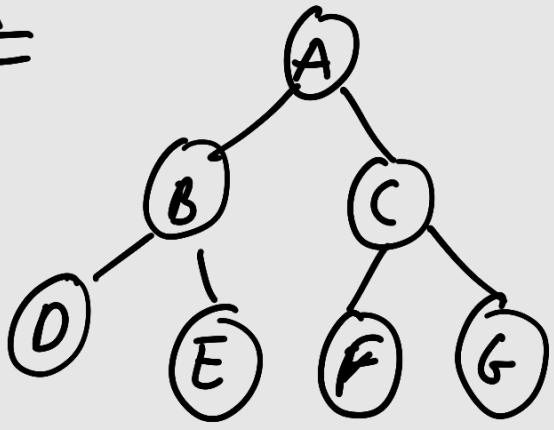
- Data structure that supports the operations of search min (or max) and delete min (or max respectively) is called a priority queue.

Heaps

- A max/min heap is complete binary tree with j property that the value at each node is at least as large as

or as small as the value at its children (If they exist)

Ex



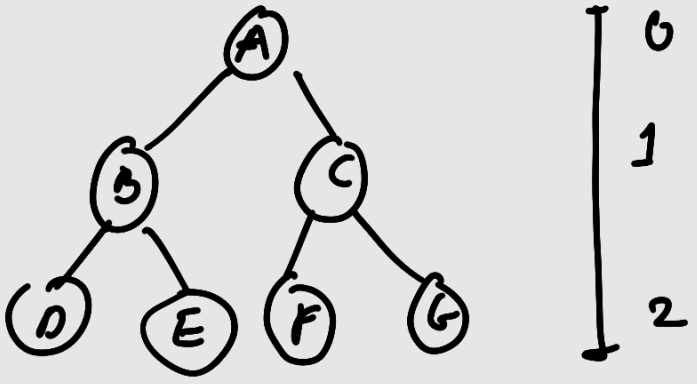
if a node is at index i

its left child is at — $2*i$

its right child is at — $2*i+1$

its parent is at — $\lfloor \frac{i}{2} \rfloor$

A	B	C	D	E	F	G
1	2	3	4	5	6	7

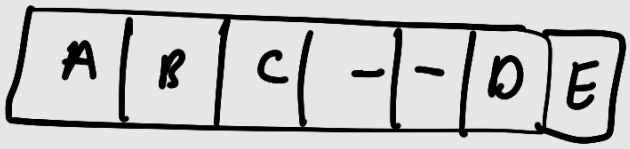


↑ Full Binary tree

If h is height $2^{h+1}-1$ nodes

A	B	C	D	E

← This is complete binary tree as no element is missing



← Here element is missing so this not Complete Binary

Every full Binary tree is Complete Binary tree
vise versa is not possible

Height of Complete Binary tree is always $\log n$

Heap

* Heap is a Complete Binary tree

. A max/min heap is complete binary tree with j property that the value at each node is at least as large as or as small as the value at its children (If they exist

Insertion in Heap

```
Insert (a, n)
{
    i = n ;
    iteam = a[n];
```

```
while ((i > 1) and a[i/2] < iteam))
                                  // Checking iteam
                                  with parent
                                  item
    do {
        a[i] = a[i/2]
        i = i/2
      }
    a[i] = item
    return true;
}
```

Deletion From a heap

```
DelMax (a, n, x)
  {
    if (n = 0) then
      {
        while ("heap is empty");
        return false;
      }
  x = a[1];
```

```
    a[1] = a[n];
    Adjust (a, 1, n-1);
     return true;
  }
Algorithm Adjust (a, i, n)
    {
      j = 2i;
      item = a[i];
      while (j<n) do
        {
          if ((j<n) and [a[j]<a[j+1]])
              {
                 then j = j+1
              }
          if (item >= a[j])
              then break;
              a[j/2] = a[j];
              j = 2j;
        }
        a[j/2] = item
```

```
}
```

Heap Sort Algorithm

```
Algorithm sort (a,n)
    {
    for i = 1 to n do insert (a,i);
    for i = n to 1 step 1 do
        {
            DelMax (a,i,x);
            a[i] = x
        }
    }
```

time Complexity

Insertion Best O(1) worst O(logn)

Deletion O(logn)

Sort O(nlogn) → Conform it

Radix Sort (Bucket Sort)

* Lexicographic Sort

Explanation using expample

consider a tuple

(2016, 10) (2016, 18) (2015, 8) (2015, 9)

sort in asc. order of year within each year sort in asc. order of marks

(Basically Sorting in n dimension)

O/P : 2015 8
2015 9
2016 8
2016 10

Here

Broad : year

Narrow : marks

sort wrt marks (1^{st} sort with narrow dimension)

(2016, 8) (2015, 8) (2015, 9) (2016, 10)

Note: Sorting Should be Stable Sort.

sort wrt year (2^{nd} sort with broad dimension)

(2015, 8) (2015, 9) (2016, 8) (2016, 10)

Radix-sort is a Lexicographic Sort which Stable sort algorithm i.e bucket Sort.

Algorithm Radix Sort

Radix Sort (S, N)

↑ Seq — Range

(consider this a last dimension)

```
{
   for (i = d → i = 1)
   {
     S = BucketSort(S, N, i)
   }
 return S;
}
```

Analysis time for Radix-sort

Bucketsort is $O(N+n)$

Radix sort → ∴ $O(d(N+n))$

↑ Since we are calling d times.

Radix Sort of Binary Number

	sorting this	Already sorted	This sorted
001	100	100	001
100 →	001 →	001 →	100
111	111	111	111

(so we sort the binary number i.e. already sort)

time Complexity $O(kn)$

k ↑ number of bit

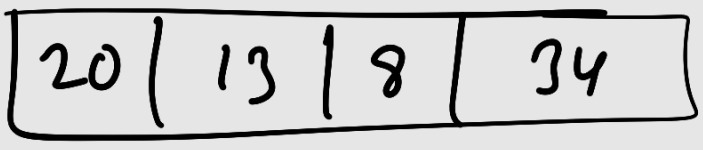
n → item to be sorted

Selection Sort

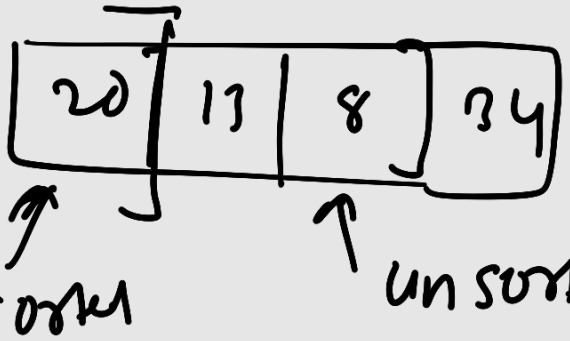
- Basically We compare get smallest element in arry and swap it with the start position of array
- In each iteration the index moves one step forward i.e size of array becomes small.
- The sort goes upto n-1 array elements.

∴ It has time Complexity $\Theta(n^2)$.

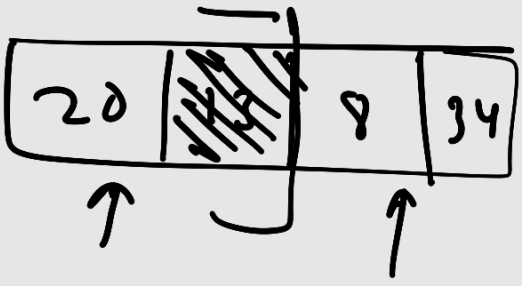
Insertion sort



binde into two array



take 3 = key

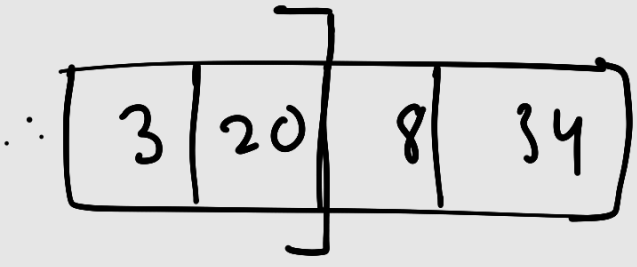


array

Compare 3 with sorted elements and

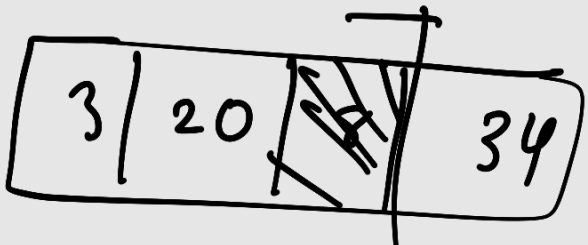
if key < element

move element to right

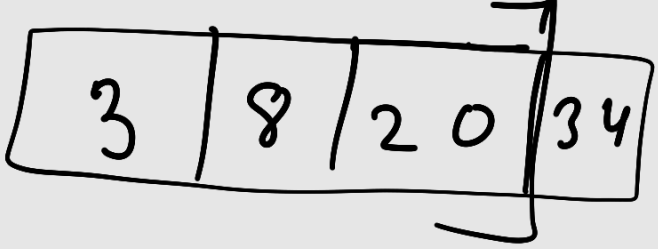


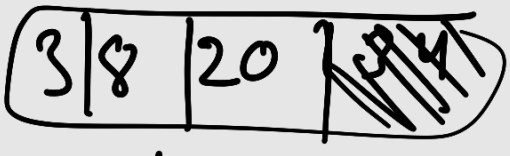
place 3

Now repeating same for 8 and 34



key = 8

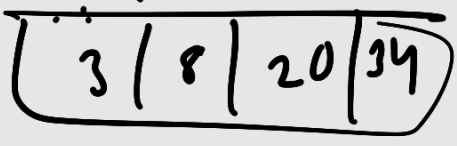




key = 34

Already at it place.

Sorted array



CHAPTER 5 Dynamic Programming

- Dynamic Programming is an algorithm design method that can be used when the solution to a problem can be viewed as the result of a sequence of decision

- **Principle of Optimality**: The principle of optimality states that an optimal sequence of decision has the property that whatever the initial state and decision are, the remaining decision must constitute and optimal decision sequence with regard to the state resulting from the first decision.

Multi stage graphs

- The multistage graph problem is to find a minimum-cost path from s to t.
- The graph is given $G = (V, E)$

Using forward approach, the cost can be calculated as

- $cost(i, j) = \min\{c(j, l) + cost(i+1, l)\}$

$= l \in V_{i+1}$

$\langle j, l \rangle \in E$

Algorithm Using Forward Approach

```
Algorithm FGraph (G, k, n, p) {
    cost[n] = 0.0;
    for j = n-1 to 1 step -1 do
    {
        Let r be a vertex such that (j, r)
        is an edge of G and c[j, r]
        + cost[r] is minimum;
        ∴ cost[j] = c[j, r] + cost[r];
        d[j] = r;
    }
```

// Find a minimum cost path

```
p[1] = 1;
p[k] = n;
for j = 2 to k-1 do
{
    p[j] = d[p[j-1]]
}
```

Time Complexity Analysis

Without memoization
(Brute force method)
Exponential time.

Dynamic Problem approach.

$$O(V+E) \leftarrow$$

because we are traveling Each Vertex and Edges only one time. (Memoization)

All pair Shortest path (Dynamic programming approach)

- Let $G = (V, E)$ be a directed graph with n vertices
- Let Cost be a cost adjacency matrix for G such that $cost(i,i) = 0$. $1 \le i \le n$
- Then $cost(i,j)$ is the length (or cost) of edge $\langle i,j \rangle$ if $\langle i,j \rangle \in E(G)$ and $cost(i,j) = \infty$ if $i \neq j$ and $\langle i,j \rangle$ not $\in E(G)$

Output of Algo

- The all-pair shortest path problem is to determine a matrix A such that $A(i,j)$ is the length of a shortest path from i to j.

Time Complexity of All pair shortest path

The all pair shortest path required $O(n^3)$ time two matrix A can be obtained in $O(n^3)$ time.

Sudo formula

$$A(i,j) = \min_{1 \le k \le n} \{ A^{k-1}(i,k) + A^{k-1}(k,j) . cost(i,j) \}$$

Generalized formula

$$A^k(i,j) = \min_{k \ge 1} \{ A^{k-1}(i,j), A^{k-1}(i,k) + A^{k-1}(k,j) \}$$

Algorithm for all-pair shortest path

```
AllPaths(A, Cost)
{
    For i = 1 to n do
      for j = 1 to n do
        A[i,j] = Cost[i,j]; // Copy cost into A
      For k = 1 to n do
          for i = 1 to n do
              for j = 1 to n do
            A[i,j] = min(A[i,j], A[i,k] +
                                 A[k,j]);
}
```

The time required by All Pair shortest path is $O(n^3)$

Application

* Internet paket routing
* Flight reservation
* Driving routes

Property 1:

A subpath of a shortest path is itself a shortest path.

Property 2:

There is a tree of shortest path from a start vertex to all the other vertices.

Optimal Binary Search Tree (Dynamic programming)

No of possible trees

$$T(n) = \frac{^{2n}C_n}{n+1}$$

Successful Search and Unsuccessful probability

* If there are n key then there are n+1 Square nodes

According to my knowledge

* The Successfull search probability is a Search probability for Sucessful Search this means that there is high probability the item is in the tree.
* Opposite of above is unsuccessful search

Cost of Searching

Based on probability

Let n = total number of nodes (including Square nodes)

$$\therefore \text{total cost} = \sum_{i=1}^{n} \text{level of node } i * \text{probability of node } i$$

Here i = Level of node in Graph

Note: For Square node level is $(i-1)$

Bari Sir's Formula

$$Cost[0,n] = \sum_{1 \le i \le n} P_i * level(q_i) + \sum_{0 \le i \le n} q_i * (level(E_i)-1)$$

Dynamic Program Aproch For OBST

$$C[i\,j] = \min_{i<k\le j} \{ C[i, k-1] + C[k, j] \} + w[i, j]$$

where $w[i,j] = w(i, j-1] + P[j] + q[j]$

and $w[i,i] = q[i]$; $r[i,i] = 0$

$C[i,i] = 0$

Algorithm

```
OBST(P, q, n)
{
  for i = 0 to n-1 do
      {
        w[i,i] = q[i];
        r[i,i] = 0;
        c[i,i] = = 0.0;
      w[i, i+1] = q[i] + q[i+1] + P[i+1];
       r[i, i+1] = i+1;
       c[i, i+1] = q[i] + q[i+1] + P(t+1];
  }

w[n,n] = q[n];
r(n,n) = 0
 c[n,n] = 0.0
for m = 2 to n do
 // Find optimal tree with m nodes
for i=0 to n-m do
   {
     j = j+m
   w[i,j] = w[i,j-1] + P[j] + q[j];
   k = Find(c, r, i, j);
```

```
        c[i,j] = w[i,j] + c[i,k-1] + c[k,j];
        r[i,j] = k;
    }
    write (c[0,n], w[0,n], r[0,n]);
}

Algorithm Find(c, r, i, j)
    {
        min = ∞
        for m = r[i,j-1] to r[i+1, j] do
        if ((c[i,m-1] + c[m,j]) < min) then
            {
                min = c[i,m-1] + c[m,j];
                l = m
            }
            return l;
    }
```

0/1 KNAPSACK

Given A set S of n items, associated with each item i having

$w_i \rightarrow$ a positive weight

$b_i \rightarrow$ a profit

Goal → Choose item with maximum total profit but with weight at most W.

Objective: Maximize $\sum_{i \in t} b_i$

Constraint: $\sum_{i \in T_1} w_i \leq W$

Set theory 0/1 knapSACK Algorithm

```
Algorithm DKP(P, W, n, m)
{
   S^0 = {(0,0)}
  for i = 1 to n-1 do
   {
     S_1^{i-1} = {(P,W) | (P-Pi, W-Wi) ∈ S^{i-1} and W ≤ m}
     S^i = MergePurge(S^{i-1}, S^{i-1});
   }
(PX, WX) = last pair in S^{n-1};
(PY, WY) = (P' + Pn, W' + Wn) where
        W' is the largest W in any pair S^{n-1}
      such that W + Wn < m;
```

if (Px > PY) then $x_n = 0$
else
$x_n = 1$;
TraceBackFor($x_{n-1}, \cdots, x_1$);
}

C++ basic Program (Easy)
bari Sir program

```
main()
  {
int P[5] = {0,1,2,5,6},
int wt[5] = {0,2,3,4,5},
int m=8, n=4;
int k[5][9];
for(i=0; i<=n; i++)
 {
  for(int w=0; w<=m; w++)
```

① If (i==0 || w==0)
{
k[i][w] = 0;

② else if(wt[i] <= w)
k[i][w] = max(P[i] + k[i-1][w-wt[i]],
k[i-1][w]);

```
    ③ else
        k[i][w] = k[i-1][w];
    }
  }
  cout << k[n][w];
}
```

Backtracking result

```
while (i>0 && j>0) {
  if (k[i][j] == k[i-1][j])
      {
       cout << i << " = 0 " << endl, i--;
      }
  else
   {
    cout << i << " = 1 " << endl;
    i--;
    j = j - w[i];
   }
}
```

Travelling Sales Person Problem (TSP)

Let $G = (V, E)$ be a directed graph with edge cost c_{ij} the variable c_{ij} is defined such that $c_{ij} > 0$ for all i and j and $c_{ij} = \infty$ if $\langle i, j \rangle \notin E$

Let $|V| = n$ when $n > 1$

- A tour G is a directed Simple Cycle that includes every vertex in V.
- The cost of a tour is the sum of the cost of the edges on the tour.
- The travelling Salesperson Problem is to find a tour of minimum cost.

All pair Shortest Path (Floyd-Warshall).

Dynamic Programming.

Basically we check path from each vertises and calculate minimum shortest path.

Formula

$$d_{ij}^{(u)} = \begin{cases} w_{ij} & \text{if } u = 0 \\ \min(d_{ij}^{(u-1)}, d_{iu}^{(u-1)} + d_{uj}^{(u-1)}) & \text{if } u \geq 1 \end{cases}$$

Thus we represent Optimal values when $u = n$ in a matrix as

$$D^{(n)} = \{d_{ij}^{(n)}\} = \{\delta(i,j)\}$$

Algorithm

```
FLOYD(W)
{
  Get Initial matrix

  for u = 1 to n
    let D^(u) = (d^(w)_ij) be a
    a new n x n matrix

    for i = 1 to n
        for j = 1 to n
```

$$d_{ij} = \min\left(d^{(k-1)}_{ij},\ d^{(u-1)}_{iu} + d^{(u-1)}_{uj}\right)$$

if $d^{(k-1)}_{ij} \leq d^{(k-1)}_{ju} + d^{(u-1)}_{uj}$

$$\pi^{(w)}_{ij} = \pi^{(k-1)}_{ij}$$

else

$$\pi^{(w)}_{ij} = \pi^{(u-1)}_{uj}$$

return $D^{(n)}$

Time complexity $O(n^3)$

Transitive closure

Floyd-Warshall can be used to determine whether or not a graph has transitive closure. i.e whether or not there are path between all vertices

- Assign all edges in the graph to have weight = 1
- Run Floyd-Warshall
- check if all $d_{ij} < n$.

Longest Common Subsequence (LCS)

Given two array we have to find matching sets in the array; it not necessary that they are linear

Ex A L G O R I T H M

G E O M E T R Y

⇒ G O M] This two are posible
⇒ G O T] max subsequence.

We can design algorithm using two method

① Recursive formulation
② optimal substructure choice

DP - Recursive Formulation

- Sub-problem:

  Lcs (i, j) = longest Common Subsequence of x_i and y_j

  Lcs (m, n) = main problem.

- Lenght of the lcs (i, j) : $c[i, j]$
- We need $c[m, n]$

Recursive Formula

$$c[i,j] = \begin{cases} 0 & \text{If } i = 0 \text{ or } j = 0 \\ c[i-1, j-1] + 1 & \text{If } i, j > 0 \text{ and } x_i = y_j \\ \max[c(i-1, j), c(i, j-1)] & \text{If } i, j > 0 \text{ and } x_i \neq y_j \end{cases}$$

Optimal Substruce choice

(sudo code)

- Find the sub-problem

$\Rightarrow x_i = prefix(x, i) = \langle x_1, \dots x_i \rangle = x[1 \dots i]$

$\Rightarrow y_j = prefix(y, j) = \langle y_1 \dots y_j \rangle = y[1 \dots j]$

- Theorm
  - Let $Z = \langle z_1, \dots z_k \rangle$ be any lcs of X and Y.
  - If $x_m = y_n$ then $z_k = x_m = y_n$

and Z_{u-1} is a LCS of X_{m-1} and Y_{n-1}

* If $X_m != Y_n$ and $Z_u != X_m$ then Z is a LCS of X_{m-1} and Y_n

* If $X_m != Y_n$ and $Z_u != Y_n$ then Z is a LCS of X_m and Y_{n-1}

- Proof by contradiction
- Conclusion: A LCS of two Sequence contain a prefix LCS of prefixes of the sequences.

Bottom up implementation

```
Lcs(x, y, m, n)
  For (i = 1 ... m)
    c[i,0] = 0 // base cases of the recurrence
  For(j = 0 ... n)
    c[0,j] = 0 //..
```

```
For (i = 1 ... m)
 For (j = 1 .... n)
 IF ( x[i] = y[i])
   C[i,j] = c[i-1, j-1] + 1
    b[i, j] = 'diag'
ELSE IF (c[i-1 ,j] >= c[i,j-1])
      c [i,j] = c[i-1, j]
      b[i,j] ← 'up'
ELSE
      c[i,j] = c [i,j-1]
      b[i,j] = 'left'
RETURN c and b
```

// Start from first row and coloum

Complexity : $\Theta(m*n)$ - both time and Space

MATRIX MULTIPLICATION

Basically we are given matrix

Let's say $A_1 \times A_2 * A_3$

matrix 1 matrix 2

So in which sequence the number of internal multiplication will be minimum (i.e multiplication of number of rows and numb. of colm) so that we get answer of multiplicat

Bottom-up Implementation

```
Matrix-chain(p[0...n])
    For(i=1...n)
        m[i,i] = 0 // Stop condition
    For(l=2...n)
                // 1-dimensional
```

sub problem.

For (i = 1 ... n - l + 1) // Start index

j = i + l + 1 // end Index

m[i, j] = INF

For (k = i ... j - 1)

// find optimal choice

$q = m[i,k] + m[k+1,j] + p[i-1] * p[k] * p[j]$

IF (q < m[i, j])

m[i, j] = q

s[i, j] = k

return m and s

Complexity

- Space : $\Theta(n^2)$

Number of Sub-problem : One solution for each

- Time : $O(n^3)$

NS : Total Number of subproblem $O(n^2)$

NC: Number of choices at each Step $O(n)$

For DP, the complexity is usually NS x NC.