02-Oct-18
Algorithms

Longest Common Subsequence

(string) X = ABCDEFG

ABC
ABG
ADEF
AFG
} subsequence of (string) X

LCS

ABCDGH
AEDFHR
} ADH (length 3)

2^n subsequences of length n string.

~~n (n-1) (n-1)~~

Use of lcs —

i. comparison between two files (diff)
ii. bioinformatics

AGGTAB
GXTXAYB
} GTAB (4)

03-Oct-18
Algorithms

[1] Recursive Solution

Let the input sequences be X[0,..,m-1] and Y[0...n-1] of lengths m and n respectively. And let lcs_len(x[n-1], y[m-1]) be the length of the two sequences x and y. Following is the recursive definition of lcs_len[X[0,..m-1], Y[0,...n-2]

if last characters of both string/sequences match / X[m-1] == Y[n-1] ;then,

lcs_len[X[m-1], Y[n-1] = 1+ lcs_len(X[m-2], Y[n-

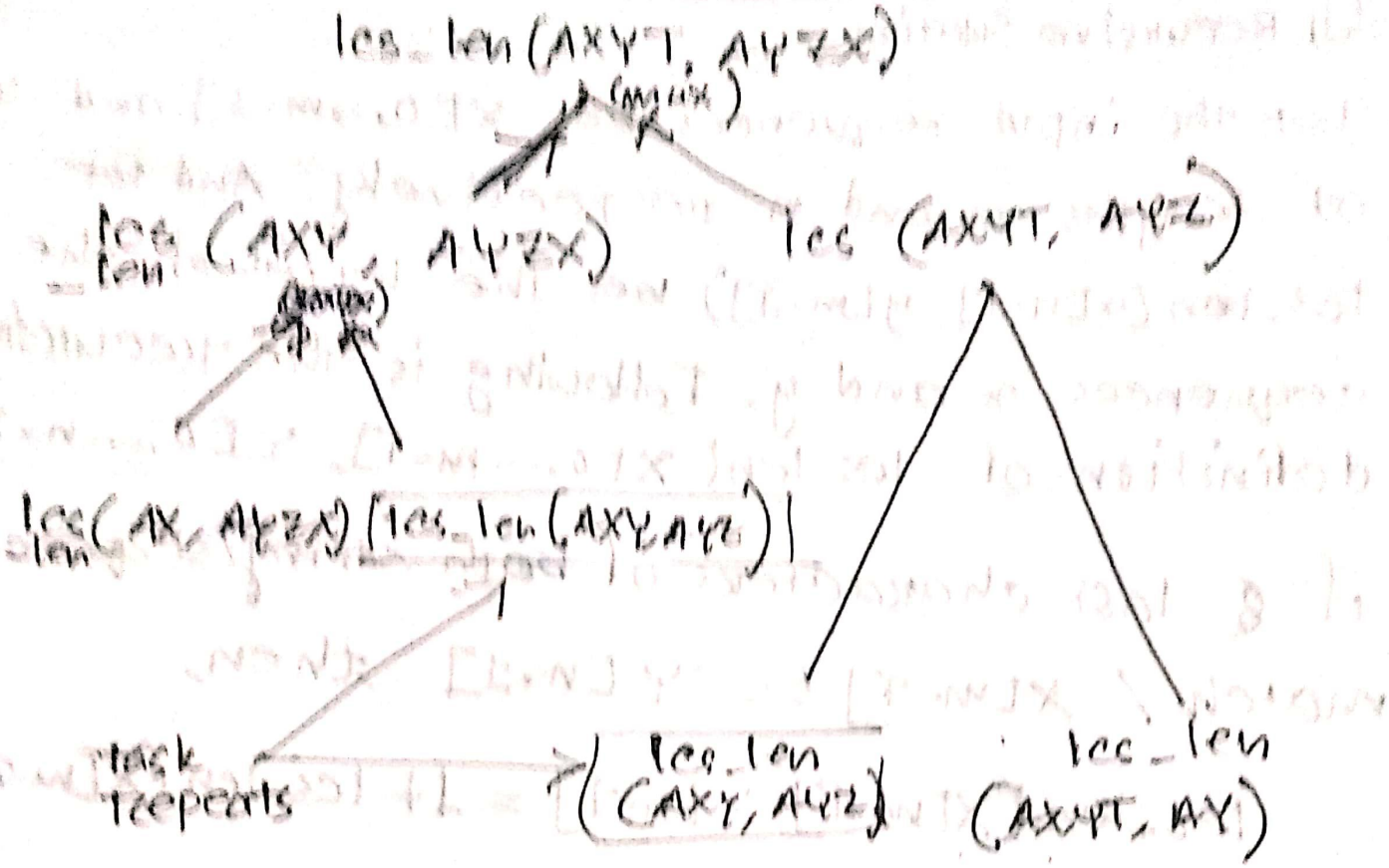
if the last characters of both sequences don't match —

lcs_len(X[m-1], Y[n-1]) =

max(lcs_len(X[m-2], Y[n-1]),
lcs_len(X[m-1], Y[n-2]))

[Complexity $O(2^n)$]

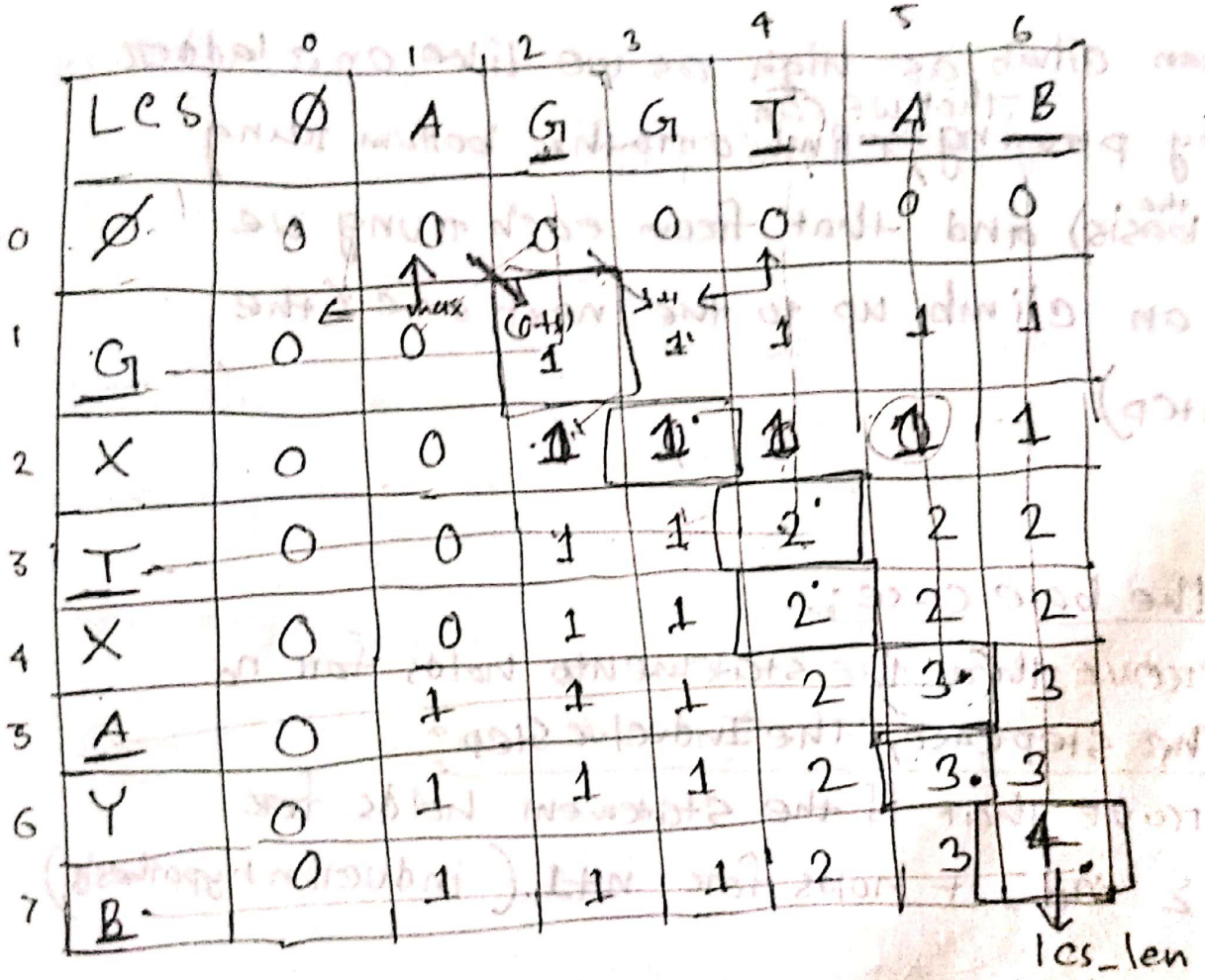
Worst-case: When all characters mismatch / lcs_



DP Implementation:

lcs_len_dp (AGGTAB, GXTXAYB)

A subsequence is a sequence that appears in the same relative order, but not necessarily contiguous.



Approach:

if the last characters match

lcs[i][j] = lcs[i-1][j-1] + 1

if the last characters do not match

lcs[i][j] = max(lcs[i-1][j], lcs[i][j-1])

17-Nov-18
Algo CT I

Proof by induction

→ Mathematical induction proves that we can climb as high as we like on a ladder by proving that we can climb onto the bottom rung (the basis) and that from each rung we can climb up to the next one (the step)

The base case:
Prove that the statements holds for n_0

The step case / The Inductive Step:
Prove that if the statement holds for $n \geq n_0$, it holds for $n+1$. (induction hypothesis)

Example:

Prove that P(n) holds for all natural numbers n where P(n) is the statement

$$0 + 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$$

$((n >= 0))$

Base case:

Showing that P(n) holds for $n = 0$

P(0) can easily be true —

$$0 = \frac{0(0+1)}{2}$$

Inducive Step:

Showing that if p(k) holds then also p(k+1) holds

Assuming that, p(k) holds,

Then p(k+1) can be written as —

$$(0+1+2+3+\cdots+k)+(k+1)$$

$$= \frac{(k+1)((k+1)+1)}{2}$$

Using the hypothesis p(k) holds, the LHS can be written as —

$$\frac{k(k+1)}{2} + (k+1) = \frac{k(k+1)+2(k+1)}{2}$$

$$= \frac{(k+1)(k+2)}{2} = \frac{(k+1)((k+1)+1)}{2}$$

thereby showing that p(k+1) holds Q.E.D

Example:

Assume an infinite supply of 4 dollar and 5 dollar coins. Induction can be used to prove that any whole amount of greater than 12 can be formed by a combinition of such coins.

//

show that, for $n >= 12$
there exist natural number
a, b such that
$n = 4a + 5b$
where 0 is included as natural number

//

$S(n)$: $n \geq 12 \Rightarrow \exists a, b \in \mathbb{N}.\ n = 4a + 5b$

Base Case:

$S(k)$ holds for $k = 12$ is trivial

let, $a = 3$ and $b = 0$

$4 \cdot 3 + 5 \cdot 0 = 12$

Step Case:

showing that $S(k)$ holds for some value of $k \geq 12$. We have to prove that $S(k+1)$ holds.

assume $S(k)$ holds

that is $k = 4a + 5b$ for some natural numbers k

have to prove that there exist some natural numbers a_1 & b_1 such that $k+1 = 4a_1 + 5b_1$

for $a \geq 1$

$$k+1 = 4a + 5b + 1$$

$$= 4a + 5b - 4 + 5$$

$$= 4(a-1) + 5(b+1)$$

$$= 4a_1 + 5b_1$$

for $a = 0$

$$k = 5b$$

$$k+1 = 5b + 1$$

$$= 5b - 15 + 16$$

$$= 5(b-3) + 4 \cdot 4$$

$$= 4 \cdot 4 + 5(b-3)$$

$$= 4a_1 + 5b_1$$

QED

$b = 0$

$k = 4a$

$k+1 = 4a+1$

$= 4a + 4 - 3 - 4 + 5$

~~$= 4a + 4(a+1)$~~

$= 4(a-1) + 5 \cdot 1$

$= 4a_1 + 5b_1$

Time & Space Complexity

Insertion Sort

```
for(i = 0; i <= n; i++){
    x = num[i];
    j = i - 1;
    while( j >= 1 && num[j] > x){
        num[j+1] = num[j]
        j--;
    }
    num[j+1] = x;
```

Best Case:
Lower bound on cost
Worst Case:
Upper bound on cost

[redacted]

Master Theorem Read See

If $T(n) = a\,T\left(\frac{n}{b}\right) + \theta(n^d)$

$a > 0 \quad b > 1 \quad d \geq 0$

then

$$T(n) = \begin{cases} \theta(n^d), & d > \log_b a \\ \theta(n^d \log_b n), & d = \log_b a \\ \theta(n^{\log_b a}), & d < \log_b a \end{cases}$$

Ex. $T(n) = T\left(\frac{2n}{3}\right) + 1$

$a = 1 \quad b = 3/2 \quad d = 0$

a, b should be constant, otherwise it won't be applicable

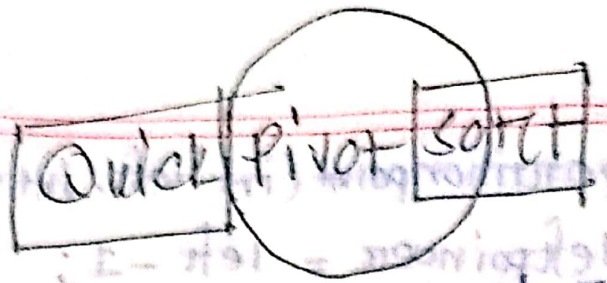
$T(n) = 1 \cdot T\left(\frac{n}{\frac{3}{2}}\right) + n^0$

$\log_b a = \log_{\frac{3}{2}} 1 = 0$ | $\theta(n^0 \log n)$ $\in \log n$

20-Dec-18 Algo I

Algo CT 2

Quick Sort



Quick sort is a divide and conquer algorithm.

It picks elements as pivots and partitions the given array around the picked pivot.

The key process of quick sort is to partition. The target of partition is, given an array and an element x of array as pivot, put x to its correct position in sorted array and put all smaller elements (smaller than x) before x and put all elements greater than x after x. All this should be done in linear time.

Implementation

```
void quicksort (int left, int right) {
        int pivot = arr[right];
        int partition_point = findpartitionpoint (left, right,
                                                  pivot);

        quicksort (left, partitionpoint - 1);
        quicksort (partitionpoint + 1, right);
}
```

```
int findpartitionpoint(int left, int right, int pivot){
    int leftpointer = left - 1;
    int rightpointer = right; // arr[right] is pivot
    while(true){
        while(arr[++leftpointer] < pivot){} //do nothing
        while(arr[--rightpointer] > pivot){} // do nothing
        if(leftpointer >= rightpointer) break;
        else swap(arr[leftpointer], arr[rightpointer]);
    }
    swap(arr[leftpointer], arr[right]); // swap
                                        // with pivot
    return leftpointer;
                // current position of pivot
}
```

Complexity Analysis:

findpartitionpoint(): $O(n/2)$

Finding Median

$$\text{Mean of an array} = \frac{\text{Sum of all elements}}{\text{Number of elements}}$$

Median : Middle element of an array when n is odd

Average of middle elements of an array when n is even

Finding Median Using quicksort() :

```
int quicksort&medianfind(int left, int right) {
        int pivot = arr[right];
        int partitionpoint = findpartitionpoint
                              (left, right, pivot);

        if (partitionpoint == k-1)
            return arr[partitionpoint];
                        // which is the kth element

    else if (partitionpoint < k-1)
            quicksort&medianfind (partitionpont+1, right);

        else quicksort&medianfind (left, partitionpoint-1);
}
```

Convex Hull

The convex polygon of smallest area that can cover all the given points.

All angles are less than 180° in convex polygon.

Graham's Scan [n log n]

```
bool compare(Point a, Point b) {
    if (a.y == b.y) return a.x < b.x;
    return a.y < b.y;
}
```

Implementation of Monotone Chain Convex Hull Algorithm

NJ SUB 17504880

LC

```
#include <bits/stdc++.h>
using namespace std;

struct Point { int x, y; };

Point inputpoints[300000];
int num_of_input;

Point hullpoints[600000];
int hp_index;

bool compare (Point p, Point q) {
    if (p.x == q.x) return p.y < q.y;
    return p.x < q.x;
}

int status (Point a, Point b, Point c) {
    det = a.x*b.y + b.x*c.y + c.x*a.y
        - a.y*b.x - b.y*c.x - c.y*a.x;
    if (det > 0) return 1; // Counterclockwise
    else if (det < 0) return -1; // Clockwise
    else return 0; // Linear
}
```

```
void makeconvexhull(){
  sort(inputpoints, inputpoints+num_of_inputs,
       compare);

  hp_index = 0;

  // building lower hull

  for(int i = 0; i < num_of_inputs; i++){
    while(hp_index >= 2 &&
          status(hullpoints[hp_index-2],
                 hullpoints[hp_index-1],
                 inputpoints[i]) < 0){
      hp_index--;
    }

    hullpoints[hp_index++] = inputpoints[i];
  }

  // building upper hull
  int t = hp_index+1;

  for(int i = num_of_inputs-2; i>=0; i--){
    while(hp_index >= t &&
          status(hullpoints[hp_index-2],
                 hullpoints[hp_index-1],
                 inputpoints[i]) < 0){
      hp_index--;
    }
    hullpoints[hp_index++] = inputpoints[i];
  }
```

```
// last point is the same as the first point, so decrease
// size by one
hp_index--;
}
```

inputpoints - take function - call hullpoints - output	main()

Disjoint Set Union

VJ SUB: UVa 10685, 11503 (identical)

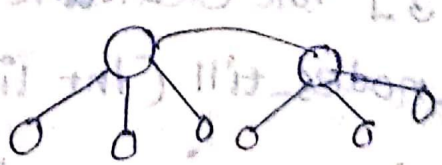
Shofayetsplanet.com/?p=763

```
#include
#define maxi 100

int parent[maxi];
int size;

void makeSet(int n){
    parent[n] = n;
}

int findRepresentitive(int representitive){
    if(parent[representitive] == representitive)
        return representitive;

    return parent[representitive]
        = findRepresentitive(parent[representitive]);
}
                    // also compresses here
                    // in one set ***

void makefriend(int a, int b){
    int u = findRepresentitive(a);
    int v = findRepresentitive(b);
    if(u == v) cout << "Already Friends" << endl;
    else parent[u] = v;
}

main(){
    for(i=0 to size-1) makeset('a'+i); // shofayet example
                                       // next page
```

&compress

&compress

```
makefriend ('a','b');
makefriend ('c','a');
makefriend ('d','e');
makefriend ('d','a');

for int (i=0; i<5; i++){

    cout << "Parent of "
         << (char)('a'+i)
         << " is "
         << (char) parent['a'+i]
         << endl;
}

for(int i = 0; i < 5; i++){
    findrepresentitive('a'+i);
    // tree minimise করার জন্য আবারও...
}
```



আকলে

type হওয়ার জন্য

//print

22-Dec-18

Implementation along with rank

Rank means: in this set, the parent have how many nodes/child along with itself

```
#includes - basics, map

int parent [100009];
int ranks [100009];

// parent[i] contains parent of
// the ith node.
// The ultimate parents, when compressed
// fully. Otherwise na confirm about the
// ultimate parents.

// Initialize nodes from 1 to limit
// Node 0 for either found or not
// [UVa 11503] for clearance
void initialize_nodes_till (int limit) {
    for (int i = 0; i <= limit; i++) {
        parent [i] = i; // Initially, parent to ourself
        ranks [i] = 1; // Initially, the node has only
                       // one element & this is the node
                       // itself
    }
}
```

```
int find_parent_and_compress (int node){
    // finds the ultimate parent of node and
    // compresses all the parents of node to ultimateparent
    if (parent[node] == node) return node;
    return parent[node] =
        find_parent_and_compress(parent[node]);
}

int make_friendship_and_count_friends_in_circle
        (int node1, int node2) {
    int u = find_parent_and_compress(node1);
    int v = find_parent_and_compress(node2);
    if (u == v) return ranks[u]; // already
                                 // friends/ in
                                 // the same set
    if (ranks[u] > ranks[v]) {
        parent[v] = u;
        ranks[u] += ranks[v];
        return rank[u];
    }
    else{ parent[u] = v;
        ranks[v] += ranks[u];
        return ranks[v];
```

Implementation:

Node given in string, not defined

Converting it to integer node —

```
#include <map> + basics

scan — string name1, name2;
init nodeAdded = 0;
if
(in_Curr_Set_which_Node[name1] == 0) {
    // name1 নামের কোন নোড এখানো নাই
    nodeAdded ++; // A new node added
                  // to the next
                  // index of last node
    in_Curr_Set_which_Node[name1]
        = nodeAdded;
}
friend1_node = in_Curr_Set_which_Node[name1];

// Similar way for name2, and so on
```

Representation of Graphs

Shafaet Book Chapter 1, 2
GFG Article

Breadth First search

BFS is used to find the shortest path in an unweighted graph.

Approach:

1. Level of source node: zero
2. The directly connected nodes to source is level 1 node
~~3. The directly connected nodes~~

If any node is directly connected to (n-1) level node and still unleveled then it will be leveled as level n.

// See GFG Article /breadth-first-search-or-bfs-for-a-graph/

Complexity : $O(V+E)$

Implementation:
Just Traversing

```
#include<iostream>
#include<list>
using namespace std;

class Graph
{
    int V;
    list<int> *adj;
    // Pointer to an array containing
    // adjacency list
    public:
    Graph(int V);
    void addEdge(int v, int w);
    void BFS(int source);
};

Graph::Graph(int V){
    this->V = V;
    adj = new list<int>[V];
}
```

```
void Graph::AddEdge(int u, int v){
    adj[u].push_back(v);
}
```

```
void Graph::BFS(int source){
    // Mark all vertices not visited
    bool *visited = new bool[V];
    for(int i = 0; i < V; i++)
        visited[i] = false;

    // Create a queue for BFS
    list<int> queueee;
    list<int>::iterator it;
    // Mark the current node visited and enqueue it
    visited[source] = true;
    queueee.push_back(source);

    while(!queueee.empty()){
        int curr = queueee.front();
        queueee.pop_front();
        for(it = adj[curr].begin(); it != adj[curr].end(); ++it){
            if(!visited[*it]){
                visited[*it] = true;
                queueee.push_back(*it);
            }
        }
    }
}
```

```
    for (it = adj[source].begin() to end) {
        if (!visited[*it]) {
            visited[*it] = true;
            queueee.push_back(*it);
        }
    }
}//while
}// fn
```

Levelling:

```
procedure BFS(G, source)
    Q ← queue();
    level[] ← infinity  // memset
    level[source] = 0;
    while Q is not empty
        u ← q.pop();
        for all edges from u to v —
            if (level[v] = inf):
                level[v] = level[u] + 1;
                q.enqueue(v);
            end if
        end for
    end while
    return distance;
```

Direction Array

```
int fx[] = { 1, -1, 0, 0 };
int fy[] = { 0, 0, 1, -1 };

for (int k = 0; k < 4; k++) {
    int nx = x + fx[k];
    int ny = y + fy[k];
    if (valid(nx, ny)) {
        // DO SOMETHING
    }
}
```

→ four iteration for four direction

Bfs in 2D Grid

UVa 10653

Checking bicolorability using BFS

See: SPOJ BUGLIFE on VJ + Sheet
UVa 10004

```
procedure bicoloring (G1, source):
    Q ← queue()
    Q.enqueue(source)
    color[] ← white //memset
    color[source] ← red

    while Q is not empty
        u ← Q.pop()

        for all edges from u to v in
        G1.adjacentEdges(u) do:

        if color[v] = white
            if color[u] = red
                color[v] = blue
            else color[v] = red
            end if

            Q.enqueue(v)
        end if
        if color[v] == color[v]
            return false
        end if ✓ end while ✓ return true
```

1. No node having an edge to itself.
11. Graph is nondirected
strongly connected: there will be at least one path from any node to any other node.

26-Dec-18

[BFS Path Printing]

```
start = node_no[fin];   fin, init are strings
stop  = node_no[init];

list<string> anslist;

while (start != stop) {
    ans_list.push_front(node[start]); // map(int) to
    kount++;
    start = g.prev[start];
}

ans_list.push_front(init);
```

BFS Problems:

UVA 429
762
439
10653
567
336
10004

SPOJ BUGLIFE

26-Dec-18

BFS Path Printing

```
start = node_no[fin];
stop  = node_no[init];
```
fin, init are strings

```
list<string> ans_list;

while (start != stop) {
    ans_list.push_front(node[start]); // map<string, int> node
    kount ++;
    start = g.prev[start];
}

ans_list.push_front(init);
```

BFS Problems:

UVA
- 429
- 762
- 439
- 10653
- 567
- 336
- 10004

SPOJ
- BUGLIFE

28-Dec-18
01:00 AM

Minimum Spanning Tree

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges ~~connected~~ of a connected, edge-weighted and undirected graph that connects all the vertices together, without any cycle and the minimum possible total edge weight.

Prim's Algorithm

See gfg/prims-minimum-spanning-tree-mst-greedy-algo-5/

(Video)

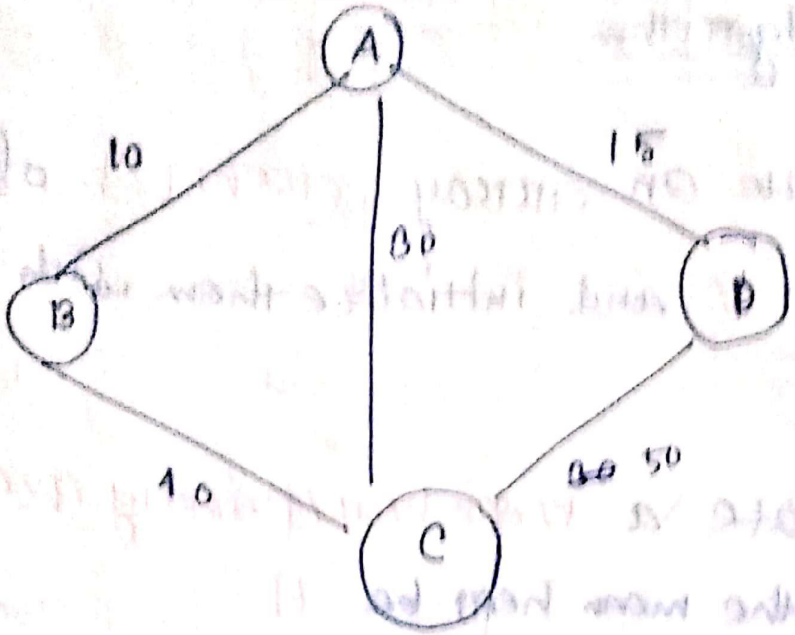
Req:

~~ptqu~~

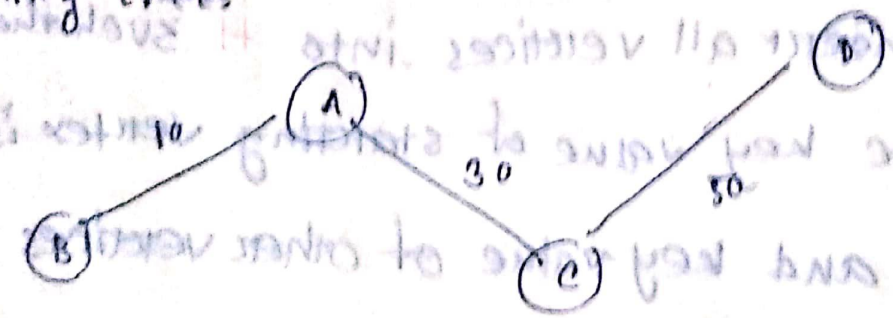
priority_queue

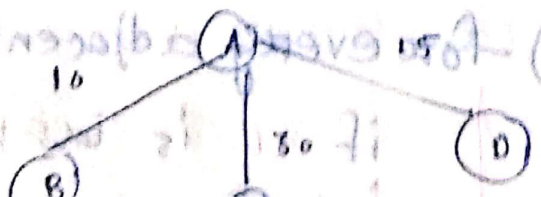
binary heap (optional)

min heap (optional)



Spanning Trees





Prim's Algorithm

1. Create an array parent[] of size V and initialize them with NIL.

2. Create a mean heap/priority que of size V
   Let the mean heap be H

3. Insert all vertices into H and initialize the key value of starting vertex is 0 and key value of other vertices infinite.

4. while H is not empty:
   a) u = ExtractMin(H) // also pop
   b) for every adjacent v of u:
      i) if u is not processed yet
         i. if edge u-v is smaller than current key value of v:
            update key value of v with u-v
         ii. parent[v] = u.

end if
end for
end while

Complexity of PRIME MST Algo: O((V+E) x log V)
O(V log V + E log V)

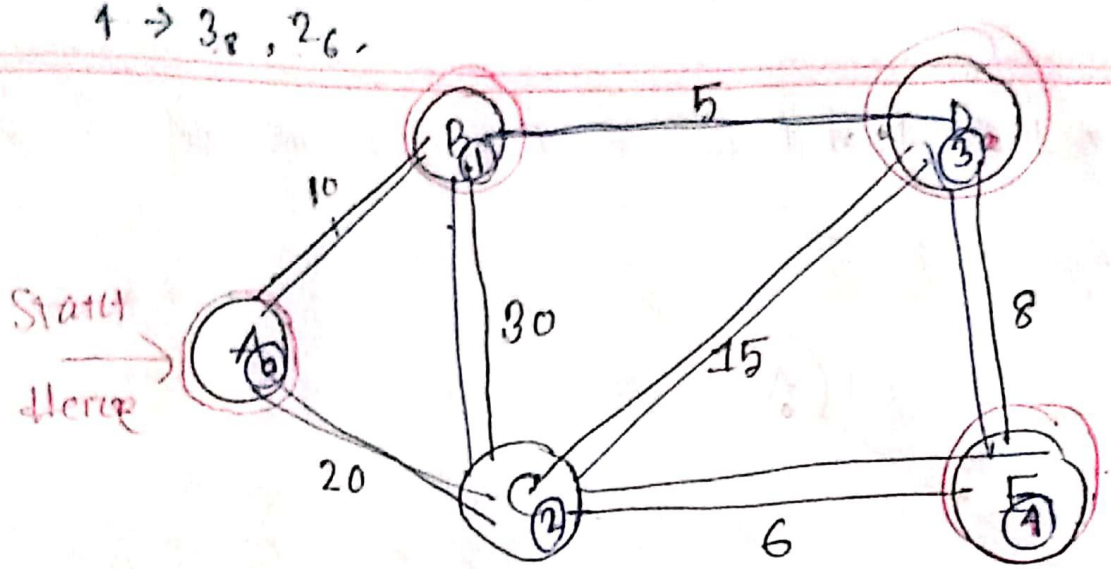
$2 \to 0_{20}, 1_{30}, 3_{15}, 4_{6}$

$3 \to 1_{5}, 2_{15}, 4_{8},$

$1 \to 3_{8}, 2_{6},$

$0 \to 1_{10}, 2_{20}$

$1 \to 0_{10}, 2_{30}, 3_{5}, 4_{\infty}$



Vertex	key	Parent
A ✓	0	NIL
B ✓	$\alpha \to 10$	NIL → A
C	$\alpha \to 20 \to 15$	NIL → A → D → E
D ✓	$\alpha \to 5$	NIL → B
E ✓	$\alpha \to 8$	NIL → D

Problems of Prim's Algo:

vjudge.net / ~~contest~~/277422

C++ Implementation:

```
add_edges (int u, int v, int edge_val){
        pair<int, int> tp;
        tp.first = v;
        tp.second = edge_val;
        adj_list[u].pb(tp);
        tp.first = u;
        tp.second = edge_val;
        adj_list[v].pb(tp);
}

extract_min(){
    return the index of minimum valued key which
    is not taken yet
}

make_MST(source){
    taken[] = false, key[] = inf, parent[] = nil;
    list<pair<int, int>> :: iterator it;
    while(1){
        u = extract_min();
        if(u == -1) break;
        taken[u] = true;
        for(it = adj_list[u].begin to end-1){
            pair<int, int> v = *it;
            if(taken[v.first]) continue;
            if(key[v.second] < key[v.first]){
                key[v.first] = v.second; parent[v] = u; }
```

01-Jan-19

There are (num_of_vertices - 1) edges in a spanning tree.

There can be

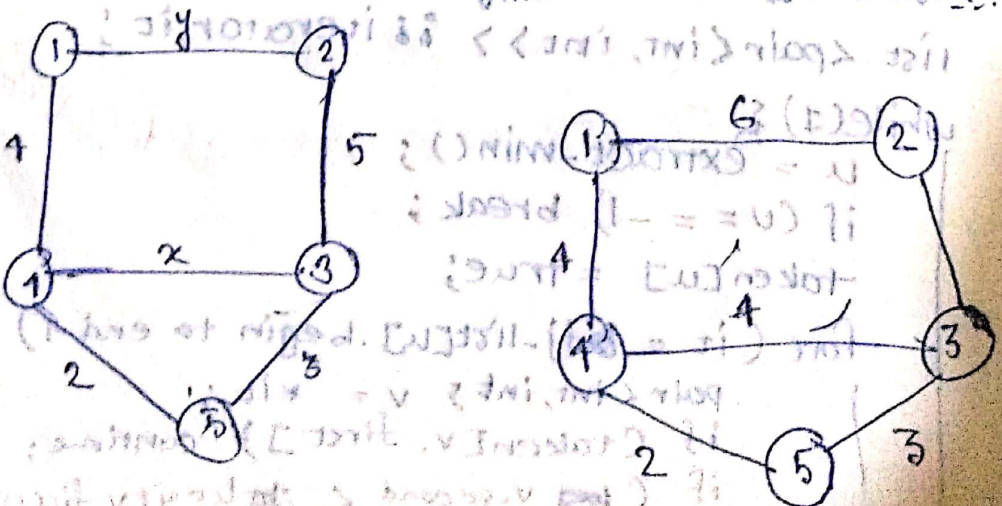
$$C_{(num\_of\_vertices\_in\_graph - 1)}^{num\_of\_edges\_in\_graph}$$

spanning tree in a graph.

Kruskal Algorithm

Kruskal algorithm may work for non connected graph also, but it may give spanning tree for non connected components also.

minimum value of x & y ???



Kruskal Algorithm:

1. Sort all the edges in non decreasing / increasing order of their weight.
2. Pick the smallest edge, check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include the edge. Else, discard it.
3. Repeat step 2 until there are (v-1) edges in the spanning tree.

#

```
MST_Kruskal (G, w):
    A = ∅
    St = { }
    Esorted = sort_by_weight (G.E)
    parent[N] = NULL
    for i from 1 to N do
        makeSet (N)
    for all the edges (u, v) in Esorted do
        if FindRepresentitive (u) != FindRepresentitive (v)
            Union (u, v)
            add (u, v) to St
        end if
    end for
    — Return St
```

Code: Shantu Book Pg 153

Implementation of Kruskal:

Shafayet.

```
struct Edge{
    int u;
    int v;
    int w; //weight
    bool operator < (const edge& p) const
    {
    return w < p.w;
    }
};

int parent[MAXN];
vector<edge> e;

int find_parents (int r) {
    if (parent[r] == r) return r;
    else return find_parents(parent[r]);
```

```
int mst ( int n ) {
    sort (e.begin(), e.end());
    for(int i = 0; i< n ; i++){
        parent[i] = i;
    }

    int kount = 0, s = 0;

    for (int i = 0; i <(int) e.size(); i++) {
        int u = find_e[i].u
        int u = find-parent (e[i].u);
        int v = find-parent (e[i].v);
        if (u != v) {
            parent[u] = v;
            kount++;
            s += e[i].w;
            if (kount == n-1) break;
        }
    }
    return s
}
```

03 JAN 19

HUFFMAN CODING

- Huffman coding is a lossless data compression algorithm. The idea is to assign variable-length codes to input characters, lengths of the assigned codes are based on the frequencies of corresponding characters.

// youtube.com/watch?v=co4_ahEDCho/

- A compression technique
- Used for reducing the size of data/message

msg = B C C A B B D D A E C C B B A E D D C C

msg.length() = 20

Normally this message contains 20 X 8 bits = 160 bits

ASCII - 8 bits

A - 65 - 01000001

B - 66 - 01000010

C - 67

D - 68

E - 69

Fixed size Coding

Character	Count/Frequency	Code
A	3	000
B	5	001
C	6	010
D	1	011
E	2	100

3 bit representation.

$20 \times 3 = 60$ bits (Reduced)

~~For decoding tables~~

~~$\frac{5 \times 8 \text{ bits}}{3} = \frac{40 \text{ bits}}{15}$~~

ASCII

$= 5 \times 8$ bits $= 40$ bits for characters

5×3 bits $= 15$ bits

$40 + 15 = 55$

Msg 60 bits, Table 55 bits, Total = 115 bits (Reduced)

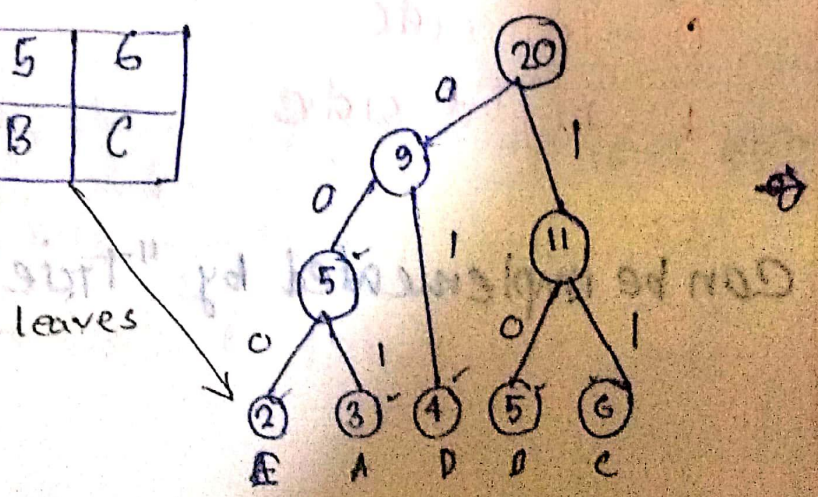
Variable Size Coding - Huffman Coding - A Greedy Approach

- We don't have to take fixed size codes for the alphabets.
- Some characters/alphabets may be appearing few times less number of times and may be appearing more number of times.

Character	Count/Frequency	Code	Size (bits)
A	3	001	3x3 = 9
B	5	10	2x5 = 10
C	6	11	2x6 = 12
D	4	01	2x4 = 8
E	2	000	3x2 = 6
8x5 = 40 bits	20	12 bits	45 bits

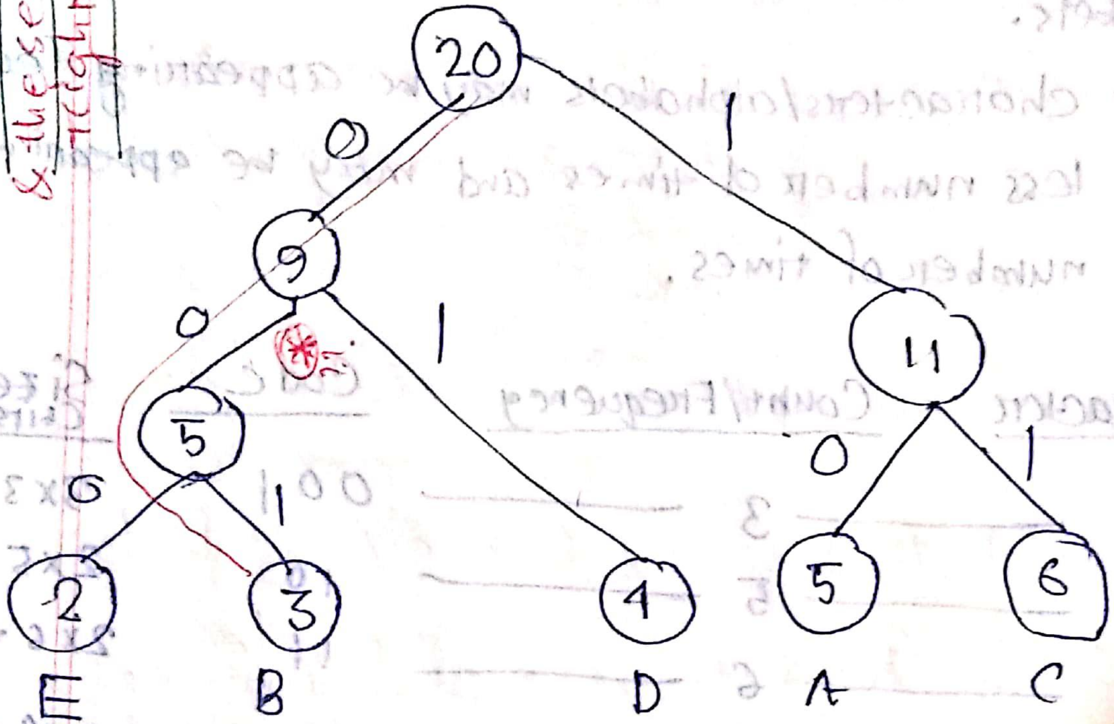
- All the alphabets should be arranged in the increasing order of their count.

2	3	4	5	6
E	A	D	B	C



(*) try to make the first extracted left (lesser) & the second extracted right (bigger)

Decoding



B = 001

0 - left side

1 - right side

can be implemented by "Trie"

The most frequent character gets the smallest code and the least frequent character gets the largest code.

gfg / huffman-coding-greedy-algo-3/

video

Huffman coding

- lossless data compression algorithm
- we assign variable length codes to input characters, length of which depends on frequency of characters
- the variable-length codes assigned to input characters are prefix codes

{0, 11}

Prefix code

Uniquely Decodable

{0, 1, 11}

Non prefix code

'1' is a prefix of '11'

Character	Frequency
a	5
b	9
c	12
d	13
e	16
f	15

Huffman tree building approach:

[1] Create a leaf node for each unique character and build a mean heap for all leaf nodes.

[2] Extract two nodes with the minimum frequency from the min heap.

[3] Create a new internal node with frequency equal to the sum of the two nodes frequencies. Make

the first extracted node as its left child and other extracted node as its right child. Add the node to the min heap.

[4] Repeat 2 & 3 until the heap contains only one node. The remaining node is the root node and the tree is complete.

⇓

```
while( size(meanheap) > 1){
        left = extractmin(meanheap);
        right = extractmin(meanheap);

        top = new node( left->freq + right->freq

        meanheap.push(top);

}
```

Solve : UVA 240

Submission on VJ :

08 JAN 19

Array Compression

~~#include <map>~~

arr[] = { 1, 0, 0, 2, 5, 2, 1, 0, 4, 5, 1, 2 }

Given a number, say in which positions the number exists.

www.shafaetsplanet.com/?p=1388

if, given numbers are in range $[-2^{30}, 2^{30}]$.

"your_node_to_my_node" technique ;)

[04 JAN 19]

-Heap Sort // Read "Heap" first from Jan's document

-array remove() function calling.

Complexity: O(n log n)

```
int remove() {
    int ret = arr[1]; //will be returned at last
    arr[n+1] = arr[1]
    arr[1] = arr[n];
    n--;
    curr = 1
    while (2*curr <= n) {
        int next = 2*curr;
        if (next+1 <= n) {
            if (arr[next+1] < arr[next]) {
                next++;
            }
        }
        if (arr[next] < arr[curr]) {
            swap(arr[next], arr[curr]);
            curr = next;
        }
        else break
    }
```

Priority Queue

A priority queue is typically used implemented using heap data structure.

Applications:

i. Dijkstra's Shortest Path Algorithm using priority queue: When the graph is stored in the form of adjacency list or matrix, priority queue can be used to extract minimum efficiently when implementing dijkstra's algorithm.

ii. CPU scheduling

iii. In finding min Minimum Spanning Tree

iv. Extract min in huffman coding

Priority queue —

Every item has a priority associated with it.

An element with high priority is dequeued before an element with low priority.

If two elements have the same priority, they are served according to their order in queue.

C++ STL

empty() — whether the queue is empty.
size() — size of the queue
top() — topmost element in the queue
push(g) — adds element g in queue.
pop() — delets the top element

Generally, maxheap is executed in STL

priority_queue <datatype> pq;

- to get minheap

operator '<' should be overloaded as '>'

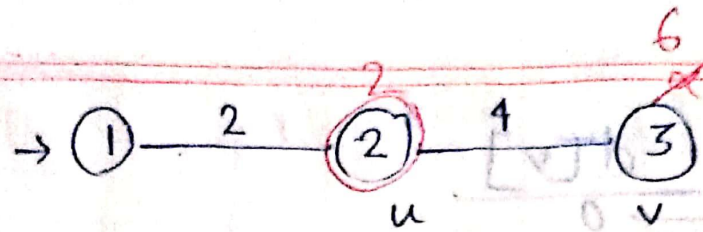
//See Shanto book pg

04 JAN 19

DIJKSTRA ALGORITHM

//

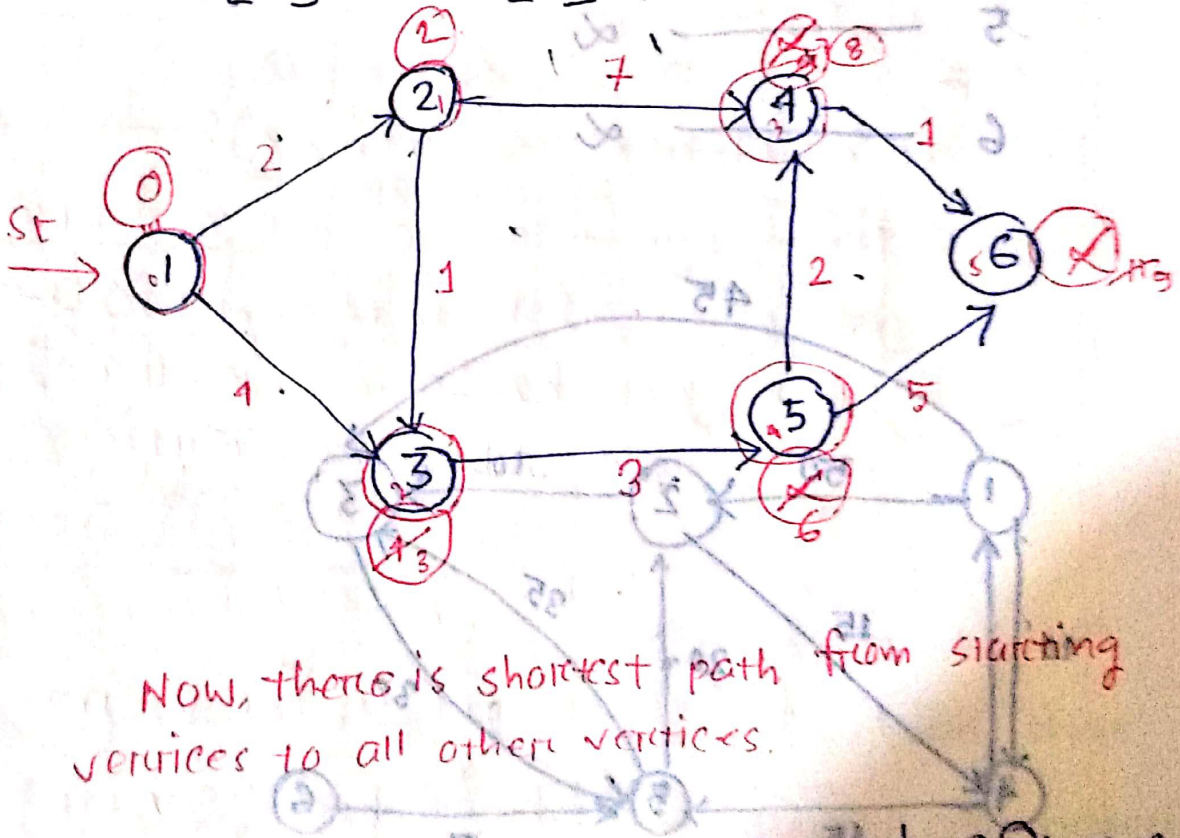
- Single source shortest path problem
- maybe direct path / via other vertices
- Shortest path is a minimization problem
- Minimization problem is an optimization problem
- Optimization problems can be solved in greedy approach
- Greedy methods says that a problem should be solved in stages by taking one step at a time and considering one input at a time to get the optimised result.
- In Greedy methods, there are predefined procedures to get optimal soln.
- A dijkstra algorithm can work on a directed as well as non directed graphs.



Path Relaxation

Relaxation:

```
if (dis[u] + cost[u][v] < dis[v])
    dis[v] = dis[u] + cost[u][v];
```



Now, there is shortest path from starting vertices to all other vertices.

$n = |V|$

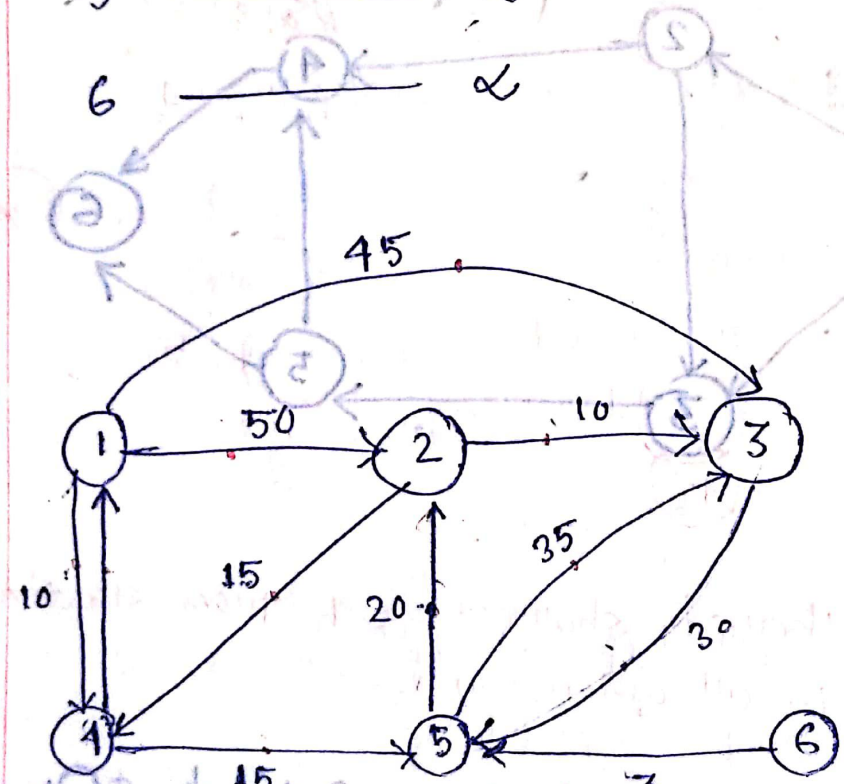
① $O(n \times n)$: Relaxing

$O(n^2)$

Using Priority Queue

$O(V \log V + E)$

vertex	d[v]
1	0
2	50
3	45
4	10
5	∞
6	∞



$|V| = n$

① $(n \times n)$: Relaxing

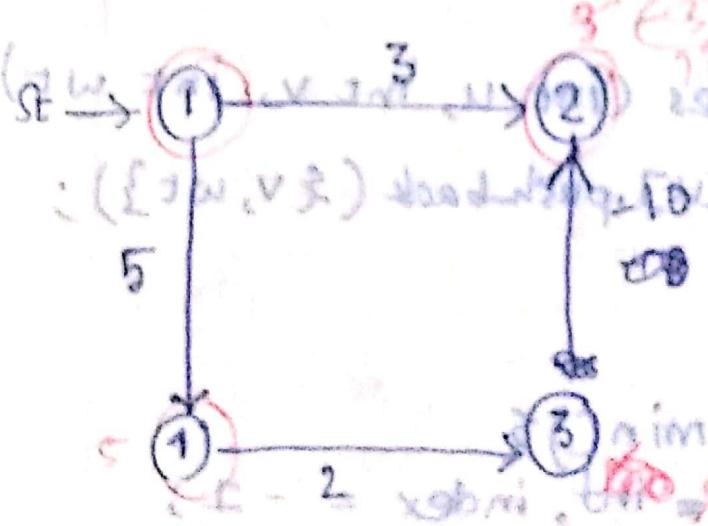
$O(n^2)$

Using Priority Queue

$O(V \log k + E)$

Selected vertex ↓	1 ✓	2 ✓	3 ✓	4 ✓	5 ✓	6 ✓	
NO, init	0	∞	∞	∞	∞	∞	
1	0	50	15	10	∞	∞	
4	0	50	15	10	25	∞	
5	0	45	15	10	25	∞	
2	0	15	15	10	25	∞	
3	0	15	15	10	25	∞	
6	0	15	15	10	25	∞	

Drawback / Disadvantages



Dijkstra has not considered that if there are negative edges then what we should do.

06 JAN 19

Implementation of Dijkstra's

```
class Graph{
public:
    int num_of_vertices;
    list<pair<int,int>> *adj_list;
    bool *taken;
    int *key;

    Graph(int num_of_vertices){
        this->num_of_vertices = num_of_vertices;
        adj_list = new list<pair<int,int>>[num_of_vertices];
        taken = new bool[num_of_vertices];
        key = new int[num_of_vertices];
    }

    void add_edges(int u, int v, int wt){
        adj_list[u].push_back({v, wt});
    }

//  int extract_min(){
//      int curr = inf, index = -1;
//      for(int i = 0; i < numof_vertices; i++){
//          if(key[i] < curr && taken[i] == false){
//              curr = key[i];
//              index = i;
//          }
//      return index;
```

Heap - store without extracting min (with priority queue (log n))

```
struct element {                    node   weight
    int first, second;
    element() { }
    element(int _first, int _second) { first = _first; second = _second; }
    bool operator<(const element &a) const { return a.second > b.second; }
};

void Dijkstra(int source) {

    for(int i = 0; i < num_of_vertices; i++) {
        key[i] = inf;
        parent[i] = nil;
        taken[i] = false;
    }
    priority_queue<element> q; q.push(element(source, 0));
    list<pair<int, int>>::iterator it;
    key[source] = 0;

    // while(1) {
    while(!q.empty()) {
        // int u = extract_min();
        // if (u == -1) break;
        int u = q.top().first;
        q.pop();
        taken[u] = true;

        for(it = adj_list[u].begin(); it != adj_list[u].end(); it++) {
            pair<int, int> v = *it;
            if (taken[v.first]) continue; // for both
            if (key[u] + v.second < key[v.first]) {
                key[v.first] = key[u] + v.second;
                parent[v.first] = u;
                q.push(element(v.first, key[v.first]));
            }
        }
    }
```

07 JAN 19

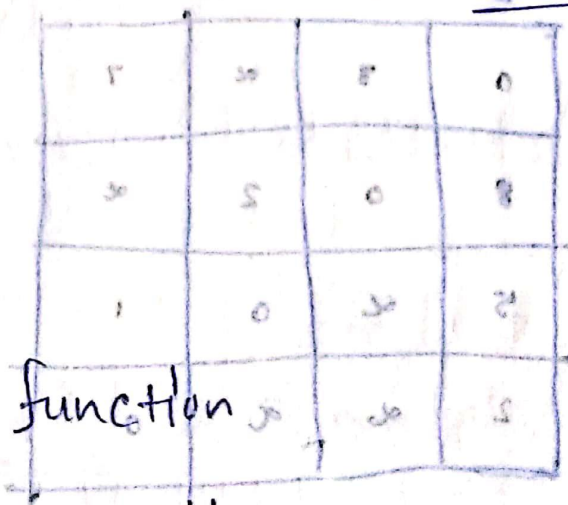
FINDING THE SECOND BEST SHORTEST PATH IN A GRAPH

~~Link to problem statement:~~

VJ SUBMISSION : 17644801

```
struct element {----} //see prev page

void dijkstra(int source){
    for(int i=0; i< num_of_vertices; i++){
        d[i] = inf; sd[i] = inf;
    }
    // d[i] = 1st min dis, sd[i] = sec min dis
    list<pair<int,int>>::iterator it;
    d[source] = 0;

    priority_queue<element> pq;
    pq.push({source, 0});
    while(!pq.empty()){
        element uu = pq.top(); pq.pop();
        int u = uu.node;
```

```
for (it = adj_list[u].begin(); it != adj_list[u].end(); it++) {
    pair<int, int> v = *it;
    if ( uu.val + v.second < d[v.first] ) {
        sd[v.first] = d[v.first];
        d[v.first] = uu.val + v.second;
        pq.push({v.first, d[v.first]});
                 ↓ element
    else if
    ( uu.val + v.second > d[v.first]
      uu.val + v.second < sd[v.first] ) {

        sd[v.first] = uu.val + v.second;
        q.push({v.first, sd[v.first]});
}
} // end of function
```



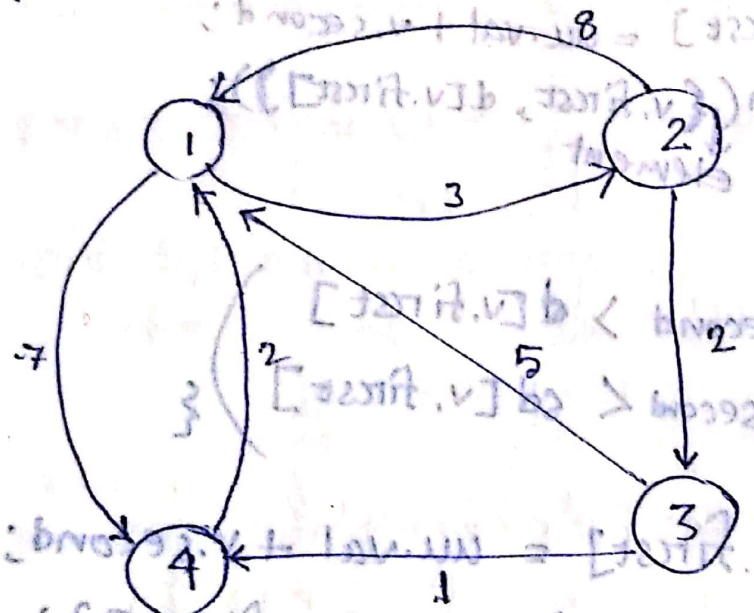
See Problems on Dijkstra:

vjudge.net/contest/278180

09 JAN 19

All Pairs Shortest Path / Floyd Warshall

Find the shortest path between every pair of vertices. // Batch 4.2



A^0 =

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞
3	5	∞	0	1
4	2	∞	∞	0

Dynamic programming says that the problem should be solved by taking sequence of decision, in each stage we will take the decision.

$A^1 =$

	1	2	3	4
1	0	3	∞	7
2	8	0	2	∞ 15
3	5	∞ 8	0	1
4	2	∞ 5	∞	0

$A^0[2,3]$ $\quad A^0[2,3] + A^0[1,3]$

2 $<$ $8 + \infty = \infty$

$A^0[2,4]$ $\quad A^0[2,1] + A^0[1,4]$

∞ $>$ $8 + 7 = 15$

A^2, A^3, A^4 Dynamically A^4 (last)

minimum path

$$A^k[i,j] = \min\{A^{k-1}[i,j]$$

Code:

midpoint

```
for (k = 1; k <= n; k++) {
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n; j++) {
            A[i, j]
            = min (A[i][j],
                   A[i][k] + A[k][j])
        }
    }
}
```

next[i][j] = next[i][k]

```
findpath (i, j)
    path ← [i]
    while i ≠ j :
        i ← next[i][j]
        path.append(i)
    end while
return path
```

See Problems:
vjudge.net/contest/278424

Complexity of Dijkstra

In worst case, graph will be a complete graph
ie total edges $\frac{v(v-1)}{2}$

1. With Adjacency list and priority queue

$$O((v+e)\log v)$$

in worst case $e >> v$

$$\therefore O(e \log v)$$

2. Matrix & priority queue

$$O(v^2 + e \log v)$$

$$\Rightarrow O(e \log v)$$

3. With Adjacency list & priority queue -

$$O(e + v \log v)$$

4. Using unsorted array

$$O(|v|^2)$$

19 JAN 19

0/1 Knapsac Problem

(Tushar Roy) - (video)

Wt	Val
1	1
3	4
4	5
5	7

Max weight can be picked = 7

make table

Val - Wt	0	1	2	3	4	5	6	7
(1) 1	0	1	1	1	1	1	1	1
(4) 3	0	1	1	4	5	5	5	5
(5) 4	0	1	1	4	5	6	6	9
(7) 5	0	1	1	4	5	7	8	9

List the selected weights

(see video)

for all (i, j) in the table —

```
if (j < wt[i]) {
    T[i][j] = T[i-1][j]; }
else {
    T[i][j]
    = max { val[i] + T[i-1][j - wt[i]]
          { T[i-1][j]
```

Time Complexity : O(total_no_of_items × max_wt)

Recursive Solution:

```
int knapsack(int i, int w) {
    if (i >= num_of_items) return 0;
      // items are 0 to no.-1
    if (dp[i][w] != -1) return dp[i][w];
    profit1 = profit2 = 0;
    if (w + weight[i] <= capacity)
        profit1 = cost[i] + knapsack(i+1, w+wt[i]);
    profit2 = knapsack(i+1, w);
    return dp[i][w] = max(profit1, profit2);
}
```

31 JAN 19

Backtracking is an algorithmic technique for solving problems recursively by building solution incrementally one piece at time, removing solutions that fail to satisfy the constraints of a problem at any point of time

Backtracking

All Permutation Generation of a given string

```
// gfg article
// shafayet
// call with generate(0)
procedure generate(idx)
    if idx == n
        print position
        return
    for i from 0 to n
        if taken[i] = false
            taken[i] = true
            position[idx] = i
            generate(idx+1)
            taken[i] = false  // backtrack
```

[01 Feb 19]

Backtracking

All Permutation Generation of a string/set

A permutation, also called an "arrangement number" or "order" is a rearrangement of the elements of an ordered list S into a one-to-one correspondence with S itself.

A string of length n has n! permutations.

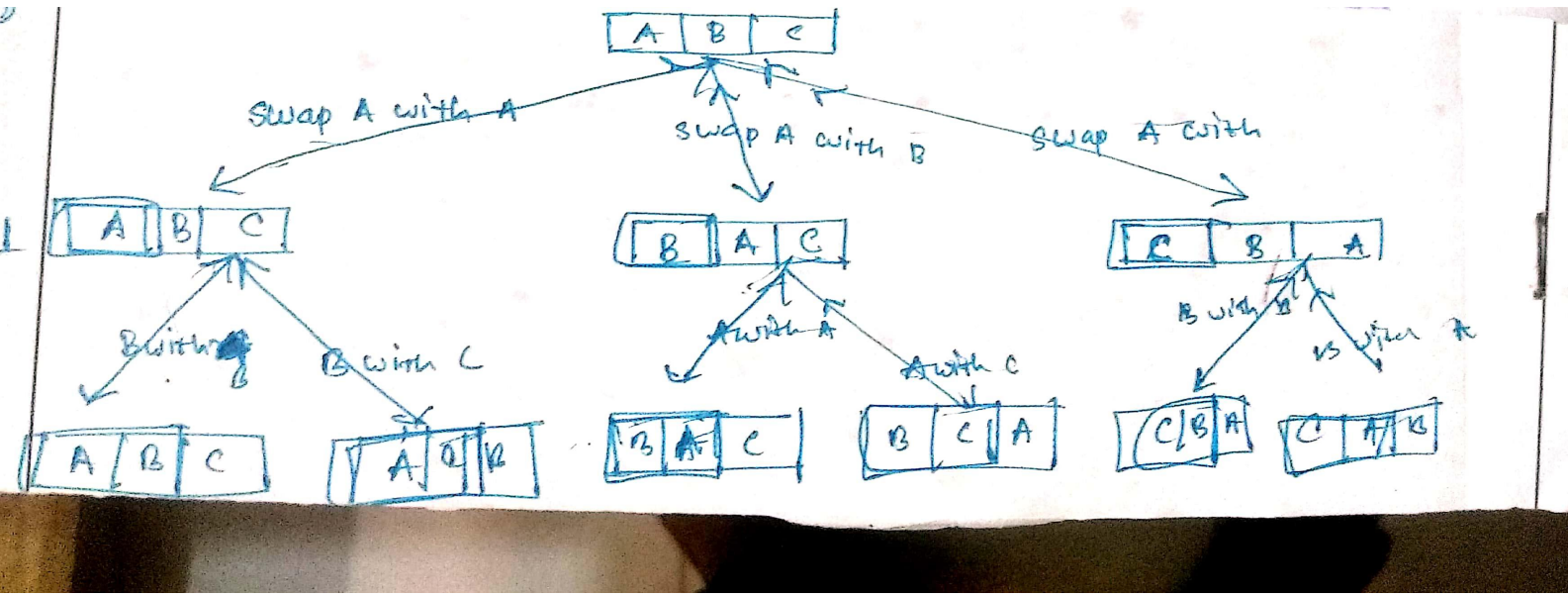
// see gfg article's image

```
void permute (char *a, int l, int r){
    int i;
    if (l == r) {cout << a<<endl; return;}

    for(i = l ; i <=r; i++) {
        swap (a[l], a[i]);
        permute (a, l+1, r);
        swap(a[l], a[i]; // Backtrack
    }
}   // This solution prints duplicates also.
```

Complexity: $n \times (n!)$

n → depth of every permutation

$n!$ → num of permutation


A B C
Swap A with A
Swap A with B
Swap A with
A B C
B A C
C B A
B with B
B with C
A B C
A B C
B A C
B C A
C B A
C A B

N Queen Problem

The problem of placing N queens in a NxN chessboard.

Naive Algorithm

```
while there are untried configurations
{
    generate the next configuration
    if queens don't attack in this
    configuration
    {
        print this configuration
    }
}
```

Backtracking algorithm:

The idea is to place different queens one by one in different columns, starting from the leftmost column. When we place a queen in a column we check for clashes

with already placed queens. In the current column, if we find a row for which there is no clash, we mark this row and column as a part of the solution. If we do not find such a row due to clashes then we backtrack & return false.

Recursive Solution: vj submission: 16695612

```
int position[10];
int table[92][8];
int tab_index = 0;

bool issafe(int ro, int col){
    for(int kol = col; kol > 0; kol--)
        if(position[kol] == ro) return false;

    for(int i = ro, j = col; i > 0 && j > 0; i--, j--)
        if(position[j] == i) return false;

    for(int i = ro, j = col; i > 0 && j > 0; i++, j--)
        if(position[j] == i) return false;

    return true;
}
```

```
void call(int col){
    if(col==9){
        for(int i=1; i<=8; i++){
            table[tab_index][i-1] = position[i];
        }
        tab_index++;
        return;
    }

    for(int i=1; i<=8; i++){
        if(issafe(i, col)){
            position[col] = i;
            call(col+1);
            *position[col]=1;
        }
    }
}
```

01 Feb 19

Lab Solution / Branch & Bound

```
#include <bits/stdc++.h>
#define N 4
using namespace std

struct point { int row; int col; };

bool isSafe(int col[], int myRow, int myCol) {
    int cr;
    for(cr=0; cr<=myRow; cr++) {
        if ( col[cr]==myCol
        || (abs(col[cr]-myCol))
        == (abs(cr-myRow))).
        return false
    }
    return true;
}
```

```
int main(){
    int col[4];
    int r = 0;
    int c = 0;

    stack<point> memory;
    point position;

    while (r < N){
        if (c<N && isSafe(col,r,c)){
            col[r] = c;
            position.row = r;
            position.col = c;
            memory.push(position);
            r++;
            c = 0;
        }
        else{
            c = memory.top().col;
            r = memory.top().row;
            c++;
            if (c >= N){
                c = memory.top().col;
                r--;
                memory.pop();
            }
        }
    }
```

Graph Coloring

Read - gfg/m-coloring-problem-backtracking-5

vertex, color

```
isSafe (v, c){
    // find adjacent of v
    // ⇒ find 1's for row ~~in~~ adj[v]
    // if these vertices have the same color
    for(node=1 to adj[v]){
        if (adj[v][node] == 1){
            if (color[node] == c)
                return false;
        }
    }
    return true;
}

AssignColor(v)
    if (v == V) ← base
    for color = 1 to n
        if (isSafe(v, color))
            given_color[v] = color;
            AssignColor(v+1);
        end if ~~given_color[v] = 0;~~ // backtrack
    end for → given_color[v] = 0; // backtrack // assigned color doesn't
```

Travelling Salesman Problem | TSP

gfg/traveling-salesman-problem-tsp-implementation

Given a set of cities and distances between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns to the starting point.

Hamilton Cycle:

Hamilton Path in an undirected graph is a path that visits each vertex exactly once. A hamilton cycle or hamilton circuit is a hamilton path such that there is an edge in graph from the last vertex to first vertex of the hamilton path.

Difference between hamilton path & TSP:

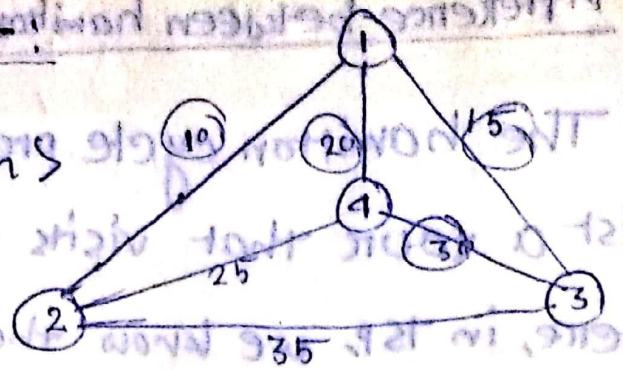
The hamilton cycle problem is to find if there exist a tour that visits every city exactly once. Here, in TSP, we know the hamilton circuit exists, as the graph is complete. And, in fact, many tour can exist. The TSP problem is to find a minimum weight hamilton cycle.

> TSP is famous NP hard problem.
> There is no polynomial time known solution for this problem.

Way of implementation:

1. Consider city 1 as the starting and ending point. Since tcoute is cyclic, we can consider any point as starting point.
2. Generate $(n-1)!$ permutations of cities.
3. Calculate cost of every permutation and keep track of minimum cost permutation.
4. Return the permutation with minimum cost.

Implementation in C++:



10 + 20 + 30 + 15 = 80

```
#include <bits/stdc++.h>
using namespace std;
#define V 4

int travellingSalesmanProblem
    ( Graph[][V], int s) {
                    source

    vector<int> vertex;

    for (int i = 0; i < V ; i++)
        if (i != s)
            vertex.push_back(i);

    int min_path = INT_MAX
```

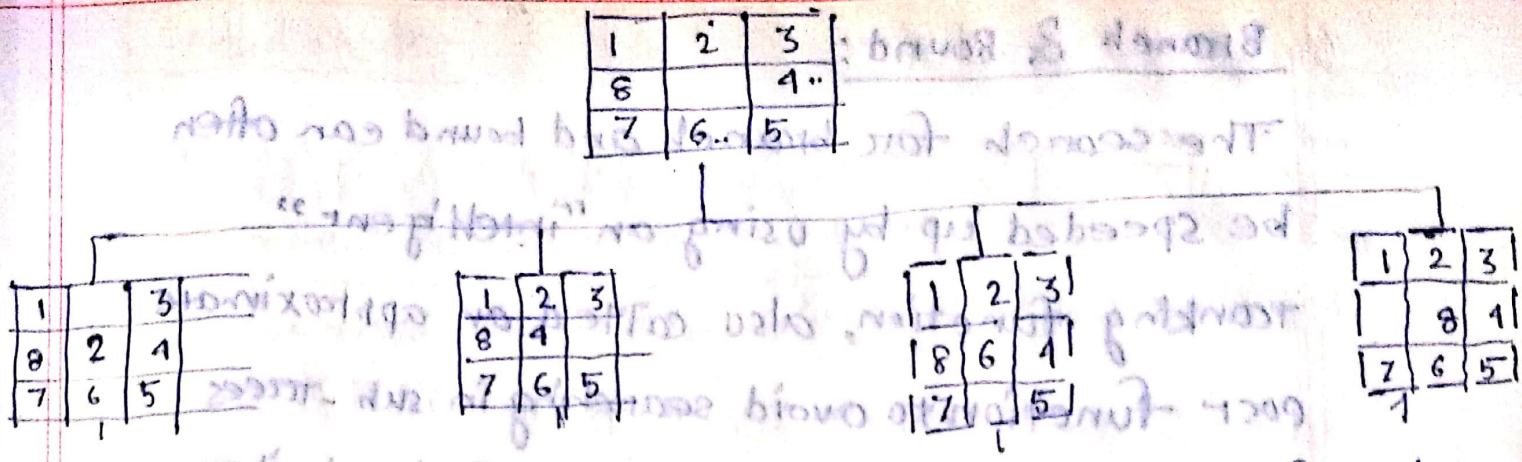
```
do {
    int current_pathweight = 0;
    int k = s;
    for (int i = 0; i < vertex.size(); i++){
        current_pathweight += graph[k][vertex[i]];
        k = vertex[i];
    }
    current_pathweight += graph[k][s];

    min_path = min(min_path, current_pathweight);

} while (next_permutation(vertex.begin(), vertex.end()));

return min_path;
}
```

8 Puzzle Problem

Given a 3x3 board with 8 tiles (every tile has one number from 1 to 8) and one empty space. The objective is to place the numbers on tiles to match final configuration using the empty space. We can slide four adjacent (left, right, above and below) tiles into the empty space.

DFS / Brute Force Solution

We can perform DFS on state space (set of all configurations of a given problem i.e all states that can be reached from initial state) tree.



In this solution, successive moves can take us away from the goal rather than bringing closer. The search of state space tree follows leftmost path from the root regardless of initial state. An answer node may never be found in this approach. (?)

BFS / Brute Force Solution

We can perform Breadth First Search on state space tree. This always finds a goal state nearest to the root. But no matter what the initial state is, the algorithm attempts the same sequence of moves like DFS.

Branch & Bound:

The search for branch and bound can often be speeded up by using an "intelligent" ranking function, also called an approximate cost function to avoid searching in sub-trees do not contain an answer node. It is similar to backtracking technique but uses BFS-like search.

There are basically three types of nodes involved in branch and bound—

Live Node:

Live node is a node that has been generated but whose children have not yet been generated.

E Node:

E node is a live node whose children are currently being explored. E node is a node which is currently being expanded.

Dead Node:

Dead node is a generated node that is not to be expanded or explored any further. All children of a dead node have already been expanded.

Cost Function:

Each node x in a search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is one with the least cost. The function is defined as

$$c(x) = G(x) + H(x)$$

where, $G(x)$ = cost of reaching current node from the root.

$H(x)$ = Cost of reaching an answers node from x

So, $c(x) = f(x) + h(x)$

where, $f(x)$ is the length of path from root to x or the number of moves so far

$h(x)$ is the number of blank tiles not in their goal positions (the number of misplaced tiles). There are at least $h(x)$ moves to transform state x to a goal state

Complete Algorithm:

//*Algorithm LCsearch uses c(x) to find an answer node

*LCsearch uses [Least()] and [Add()] to maintain the list of live nodes

*[Least()] finds a Live Node with least c(x), deletes it from the list and returns it

*[Add(x)] adds x to the list of live nodes *///

```
struct list_node {
    list_node *next;
    list_node *parent;
    float cost;
};
```

SEE GFG

```
algorithm LCSearch (list_node *t) {

    if (*t is answer node) {
        print this & return
    }

    E = t ;  // Expanding Node

    Initialize the list of live nodes to be empty

    while (true) {
        For each child x of E
        {
            if x is an answer node
            {
                print the path from x to t
                return
            }
            Add (x) : // Add to live nodes
            x -> parent = E
        }
```

if there is no more live node {
print a msg;
return;
}

E = least(); // least cost

} // end of while

} // end of function

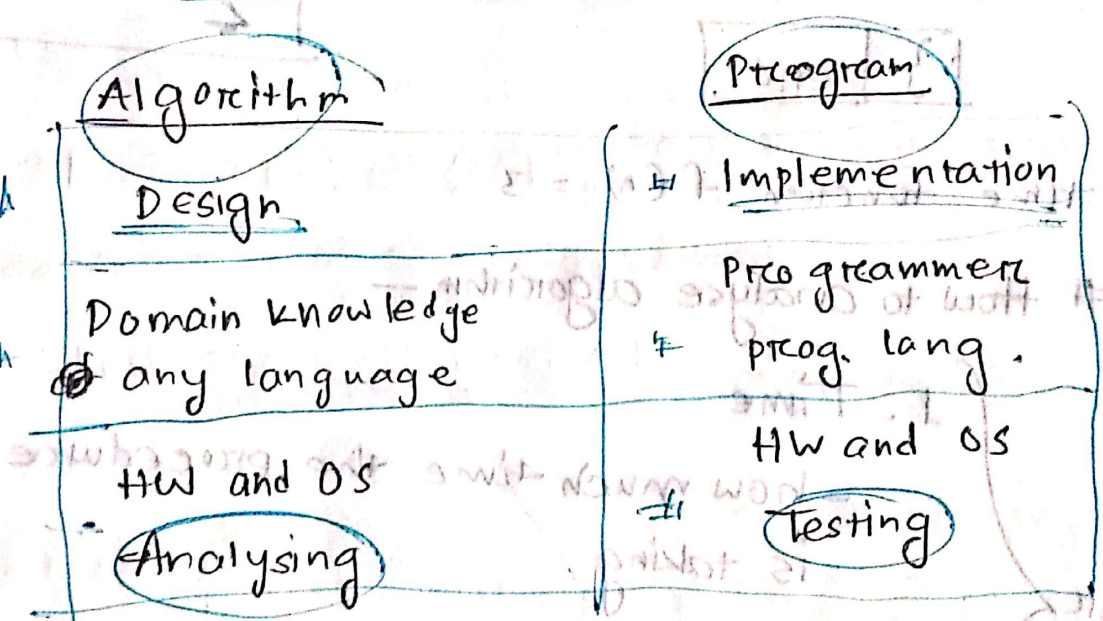
* # What is algorithm?

⇒ An algorithm is a step by step procedure for solving a problem.

Software problem

— Design phase

— Implementation phase



* # Characteristics of algorithm —

1. Input — 0, 1, 2, or more
2. Output — at least one
3. Definiteness — use defined values, solvable approaches
4. Finiteness — Must terminate at any state
5. Effectiveness — should be effective

Writing algorithm

Algorithm swap (a, b) { begin

1 — temp := a ;

2 — a := b ;

3 — b := temp ;

} end

:= assignment symbol

Time function $f(n) = 3$

How to analyse algorithm

Major

1. Time
   - how much time the procedure is taking.
2. Space
   - How much memory space the algorithm will consume.
3. Network consumption
   - in cloud or any network
   - how much data transfer is needed

4. Power Consumption
5. CPU Registers (how many)

Speed is fun.

Asymptotic Analysis

- in assymptotic analysis, we calculate the performance in terms of input size ().
- Orderer of growth / Rate of growth.

Asymptotic analysis is not perfect, but the best way available for analysing algorithms.

$1000n \log n \equiv O(n \log n)$

$2n \log n \equiv O(n \log n)$

practically different performance

asymptotically same performance

Assymptotically slower but faster for software

Assymptotic analysis —

- Worst Case
- Average Case
- Best Case

Worst-Case Analysis —

- Upper Bound of running time
- Case that causes the maximum number of operation

Average Case Analysis —

- take all possible inputs and calculate the time for computing those
- sum all the calculated values and divide them by the total num of inputs
- distribution of cases is predictable
- uniform distribution

- not easy, rarely done (mathematically)

$$\text{Avg case time} = \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$

$$= \frac{\theta\left(\frac{(n+1)(n+2)}{2}\right)}{(n+1)}$$

$$= \theta(n)$$

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

Best Case Analysis –

- lower bound on running time
- case that causes minimum number of operation
- Bogus

For some cases - all cases are same

- Ex: Merge sort

$O(n \log n)$ in worst, best, avg

Asymptotic Notations:

⊖ Notation

- bounds function from above and below, so it defines exact asymptotic behaviour.

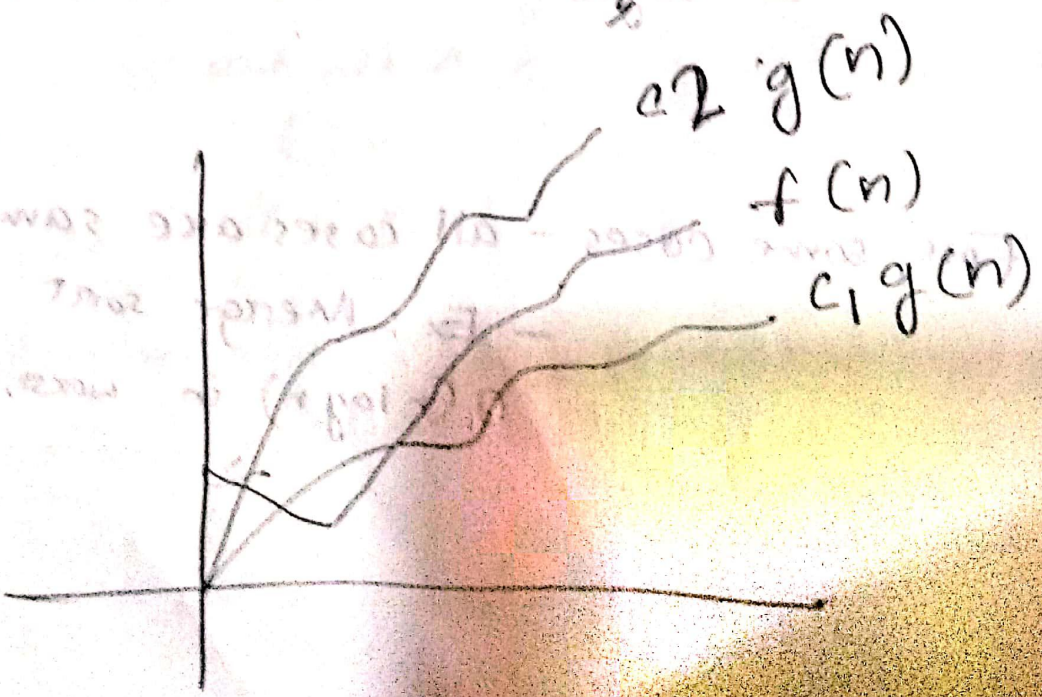
$$3n^3 + 6n^2 + 6000 = \Theta(n^3)$$

$\Theta(g(n))$

= { $f(n)$: there exist positive constants c1, c2 and n0 such that

$0 <= c1 * g(n) <= f(n) <= c2 * (g(n))$

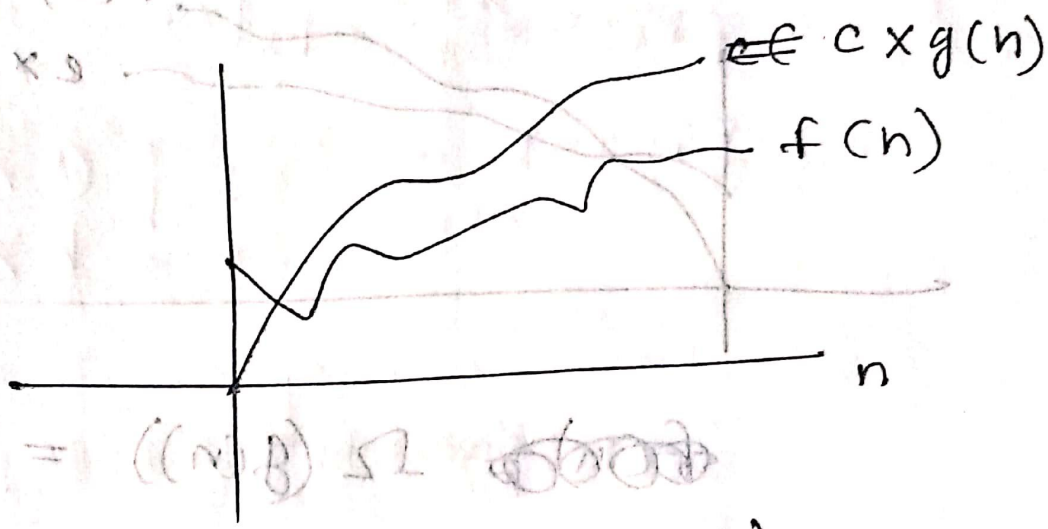
for all $n >= n0$ }



Big O Notation

= defines the upper bound of an algorithm

= bounds only from above



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

$O(g(n))$

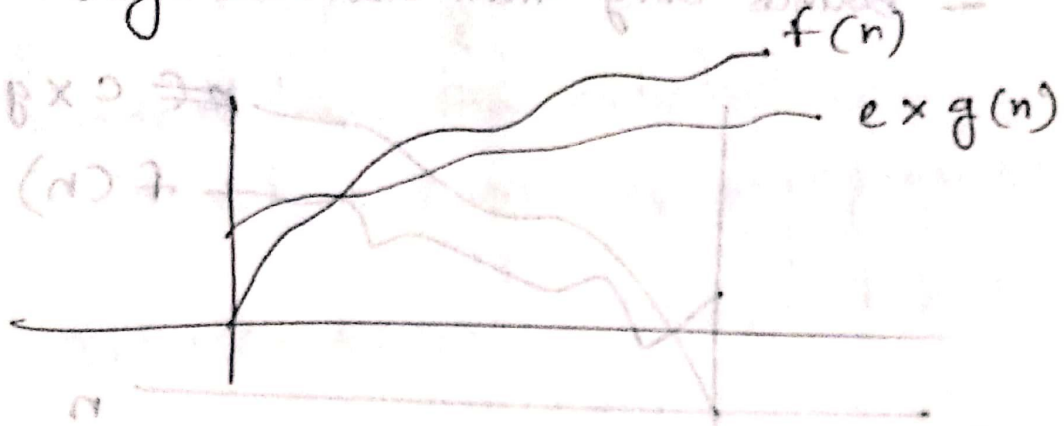
= { $f(n)$: there exist positive constants c and n_0 such that

$$0 <= f(n) <= c \times g(n)$$

for all $n > n_0$ }

Ω notation:

— asymtotic lower bound



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

$\Omega(g(n)) = \{ f(n) :$ there exists positive constants c & n_0 such that

$0 <= c \times g(n) <= f(n)$

for all $n > n_0$ }

~~Recurrences~~ ~~Relation Solving~~

Analyzing Loops

for(i = 1; i <= n; i *= c) {}

for (i = n; i > 0 ; i /= c) {}

$\Rightarrow O(\log_c n)$

** Solving Recurrance Relations

gfg → analysic of algorithm - set 4

for merge sort

$$T(n) = 2T\left(\frac{n}{2}\right) + cn$$

Substitution method -

guess a bound and prove it by induction

Master Method For Solving Recurrance Relations:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$[a >= 1, \quad b > 1]$$

$$= aT\left(\frac{n}{b}\right) + \theta(n^d)$$

$d < \log_b a \quad T(n) = \theta(n^{\log_b a})$

$d = \log_b a \quad T(n) = \theta(n^d \log n)$

$d > \log_b a \quad T(n) = \theta(f(n)) = \theta(n^d)$

$f(n) = n^{\log_b a - \epsilon} \Rightarrow T(n) = \theta(n^{\log_b a})$

Merge Sort

$$T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$$

$$= 2T\left(\frac{n}{2}\right) + \theta(n^1)$$

Comparing with,

$$T(n) = aT\left(\frac{n}{b}\right) + \theta(n^c)$$

$a = 2 \quad b = 2 \quad c = 1$

$\therefore \log_b a = \log_2 2 = 1, \quad c = 1$

$\therefore c = \log_b a$

$\therefore$ This process is $\theta(n^1 \log n)$

$= \theta(n \log n)$

Divide & Conquer Approach // gfg

// See gfg article

```
DAC (P)
    if small(P) {
        Solve(P);
    }
    else {
        divide P into P1, P2, P3 ... Pn
        Apply DAC(P1), DAC(P2) ... DAC(Pn)
        Combine DAC(P1), DAC(P2) ... DAC(Pn)
    }
```

Divide and Conquer Algorithms example —

1. Binary search
2. Finding maximum and minimum
3. Merge sort
4. Quick sort
5. Strassen's Matrix Multiplication

Decision Tree Model

- classification of predictions

A decision tree model is a flow-chart-like tree structure, where each internal node denotes a test on an attribute, each branch represents outcome of the test, and each leaf node holds a class label.

A tree can be learnt by splitting the source set into subsets based on the attribute value test.

// See GFG article

Lower Bound on Sorting

A decision tree is a full binary tree that represents the comparisons between the elements that are performed by a particular sorting algorithm operating on an input of a given size.

$\therefore n! <= 2^x$

$\Rightarrow \log_2(n!) <= \log_2 2^x$

$\Rightarrow \log_2(n!) <= x$

$\log_2(n!) = \Theta(n \log_2 n)$

Hence we can say

$x = \Omega(n \log_2 n)$

n log n এর কমে sort করা সম্ভব না।

// See GFG article

Sorting in linear time
- Radix Sort
- Bucket Sort
- Counting Sort

Radix Sort

GFG Article + Video

Counting Sort - যেভাবে বাস্তবায়িত

Bucket Sort — GFG

procedure DFS (G, source)

u ← source
time ← time + 1;
d[u] ← time
color[u] ← grey

for all edges from u to v in G.adjacentedges(v)

if (color[v] = white)

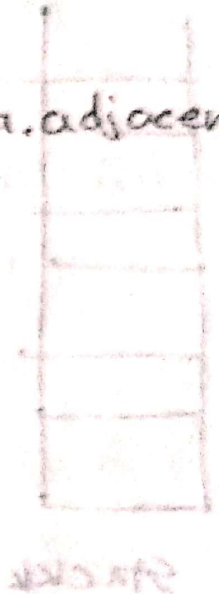
DFS (G, v)

end if

end for

color[u] ← black
time ← time+1
f[u] ← time

return



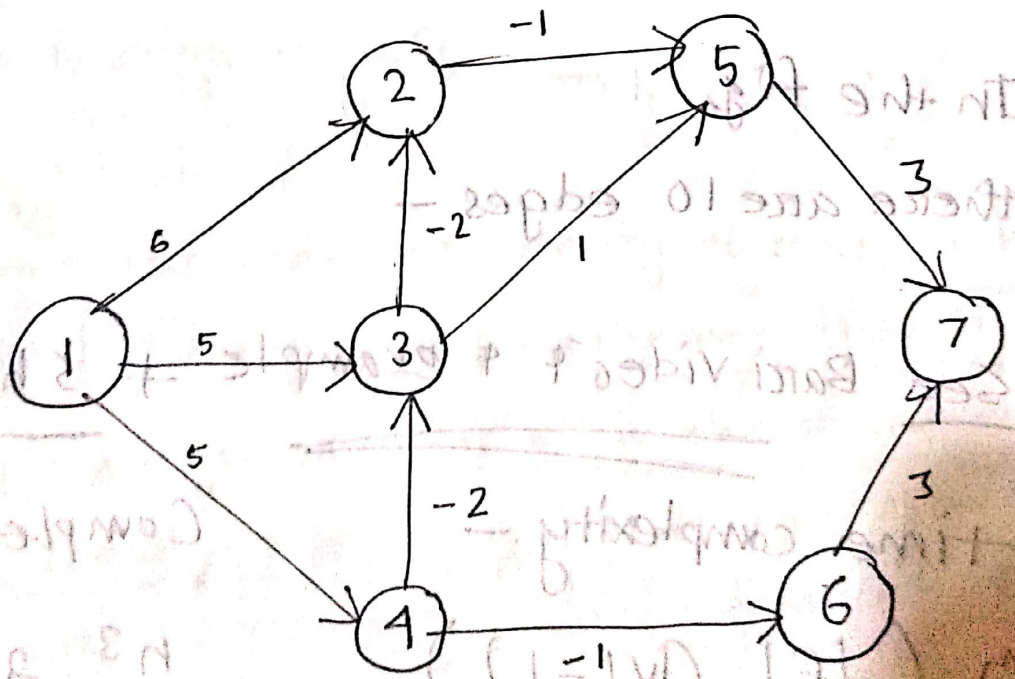
Bellman Ford Algorithm

- Single Source shortest Path Problem

Problem

In a edge weighted directed graph we have to select one of the vertices as source vertex and find out the shortest path to all other vertices.

As dijkstra algorithm does not solve negetive edged graph, bellman ford came with solution.



Dynamic programming says that you pick out all possible solutions and pick up the best solution

* go on relaxing for all of the edges for (n-1) times. n = |V|

* Here, n = 7
⇒ n-1 = 6

Relaxation :

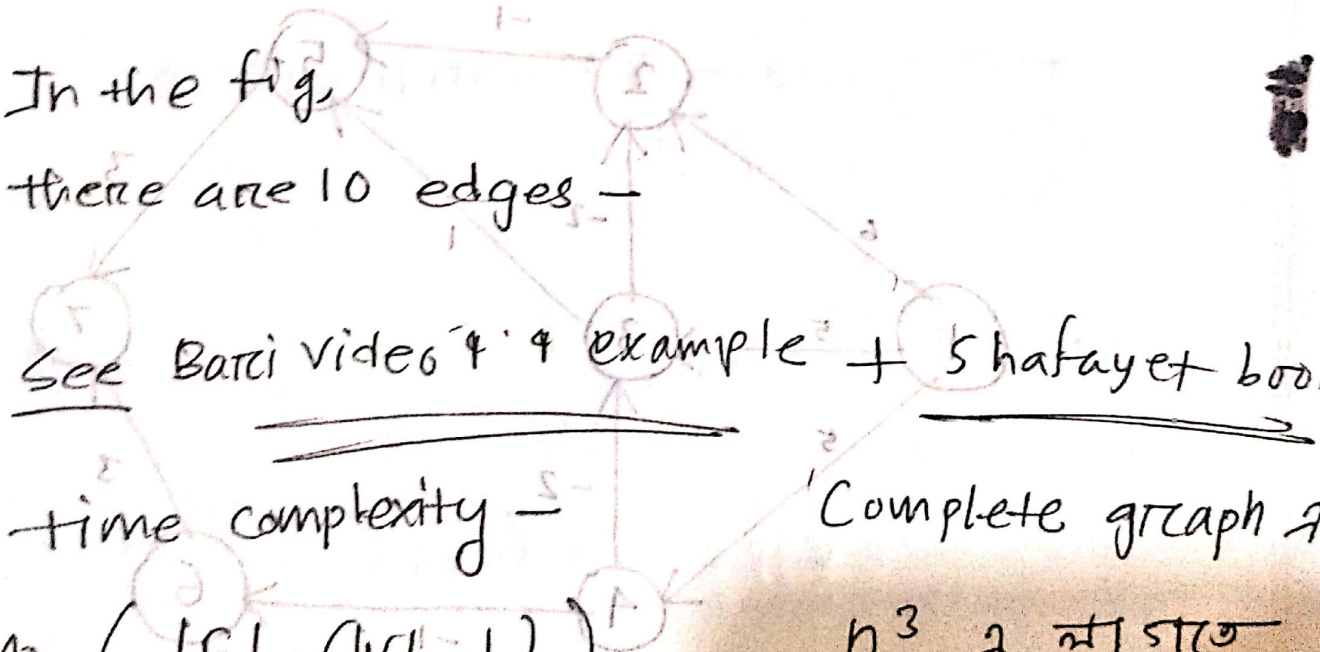
(u, v)

if (dis[u] + cost[u][v] < dis[v])
    dis[v] = dis[u] + cost[u][v]

x | Drawback?
নেগেটিভ cycle থাকলে কাজ করবে না।
Graph cannot be solved

Pseudo

In the fig,
there are 10 edges -



See Barri video & example + Shafayet book

time complexity -

O (|E| (|V| - 1))
= O (|V| |E|)
= O (n^2)

Complete graph এ n^3 এ লাগতে পারে।

Binary Search Tree #GFG - Set 1, 2

AVL Tree #GFG

Red black tree

- every node has a color either red or black
- root is always black
- there are no two adjacent red nodes
- every path from a node (any leaf) to any of its descendant NULL node has the same number of

See GFG.

Set Cover — GFG * — Greedy

Bin Packing — GFG *

In computational complexity theory, a numeric algorithm runs in pseudopolinomial time if its running time is polinomial in the # numeric value of input (the largest number present in input) - but not necessarily in the length of the input (the number of bits req. to represent it)

Polinomial algorithm's running time is upper bounded by any polinomial expression in the size of the input

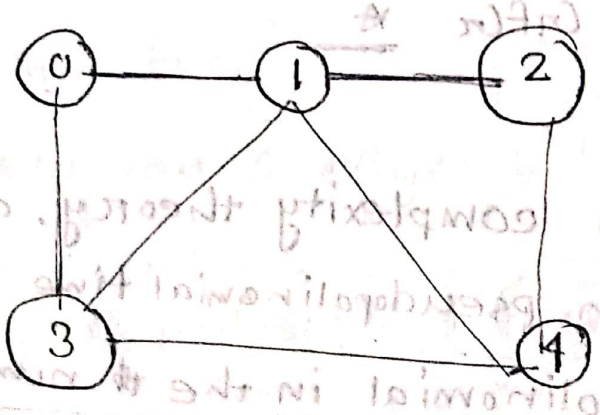
Hamilton Cycle

- NP Hard Problem (means exponential time taking prob)

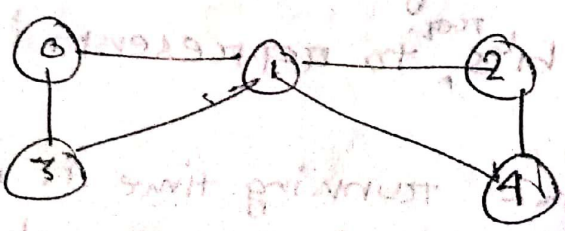
Hamilton path in an undirected graph is a path that visits each vertex exactly once.

A hamilton cycle or hamilton circuit is a hamilton path such that there is an edge (in graph) from the last vertex to the first vertex of the hamilton path.

Determining if a graph contains hamilton path or not, if contains print the path —



[contains hamilton cycle]



[does not contain hamilton cycle]

Naive algorithm –

Generate all possible configurations of vertices and print a configuration that satisfies the given constraints. There will be n! configurations.

```
while there are untried configurations {
        generate next configuration
        if ( there are edges between two consecutive
             vertices of this configuration and there is
             an edge from the last vertex to the first )
        {
            print this configuration ;
            break;
        }
}
```

Backtracking Algorithm for Hamilton Cycle

Create an empty path array and add vertex 0 to it.
Add other vertices, starting from vertex 1.
Before adding —

~~check~~ confirm —

- ~~whether~~ it's adjacent to the previously added vertex
- not added previously

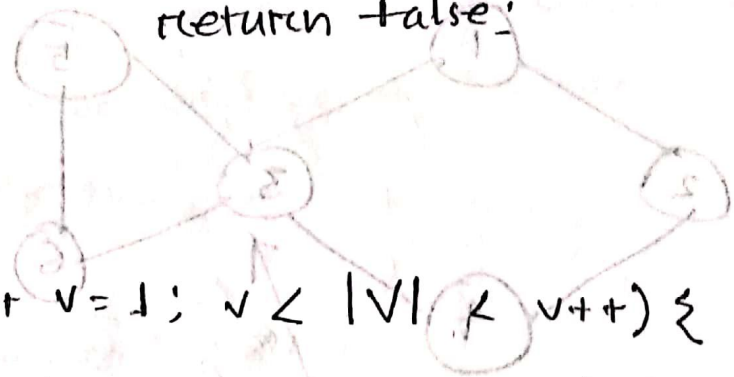
if we find such, we add the vertex to soln

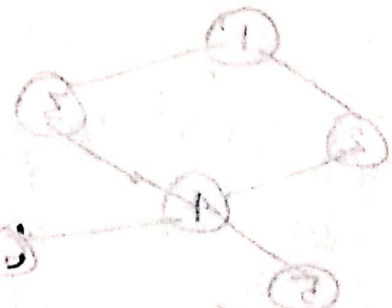
else return false

// See GFG implementation

```
bool isSafe(int v, bool graph[V][V],
            int path[], int pos){
    if (graph[path[pos-1]][v] == 0)
        return false;
    for (int i=0; i<pos; i++){
        if (path[i] == v) return false;}
    return true;
```

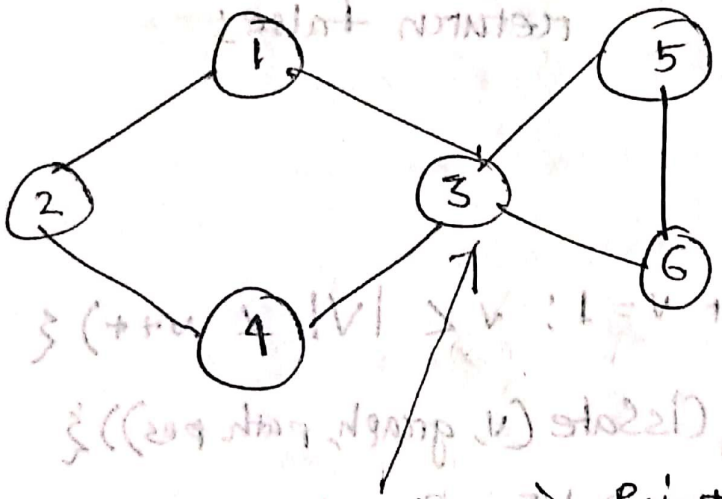
```
bool hamiltonUtil(bool graph[V][V], int path[], int pos){
    if (pos == V){
        if (graph[path[pos-1]][0] == 1)
            return true;
        else
            return false;
    }

    for(int v = 1; v < |V| ; v++){
        if (isSafe(v, graph, path, pos)){
            path[pos] = v;
            if (hamiltonUtil
                (graph, path, pos+1)
                == true))
                return true;
            else path[pos] = -1;
        }
    }

    return false;
}
```



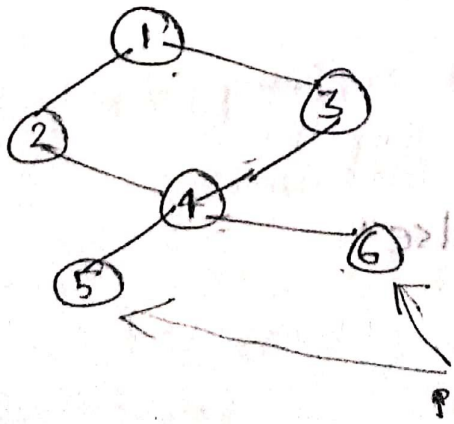


// Abdul Barii See 6.4 Hamiltonian ...

Articulation Point ⟶ Vertex — Junction for two graph



If there is articulation point in a graph, then hamiltonian cycle is not existing in the graph.



Independent set (Graph Theory)

- In graph theory, an independent set, stable set, coclique or anticlique is a set of vertices in a graph, no two of which are adjacent. That is, it is a set S of vertices such that for every two vertices in S, there is no edge connecting the two.

// tutorialspoint article, see

Independent sets are represented in cells in which

- there should not be any edges adjacent to each other. There should not be any common vertex between two edges

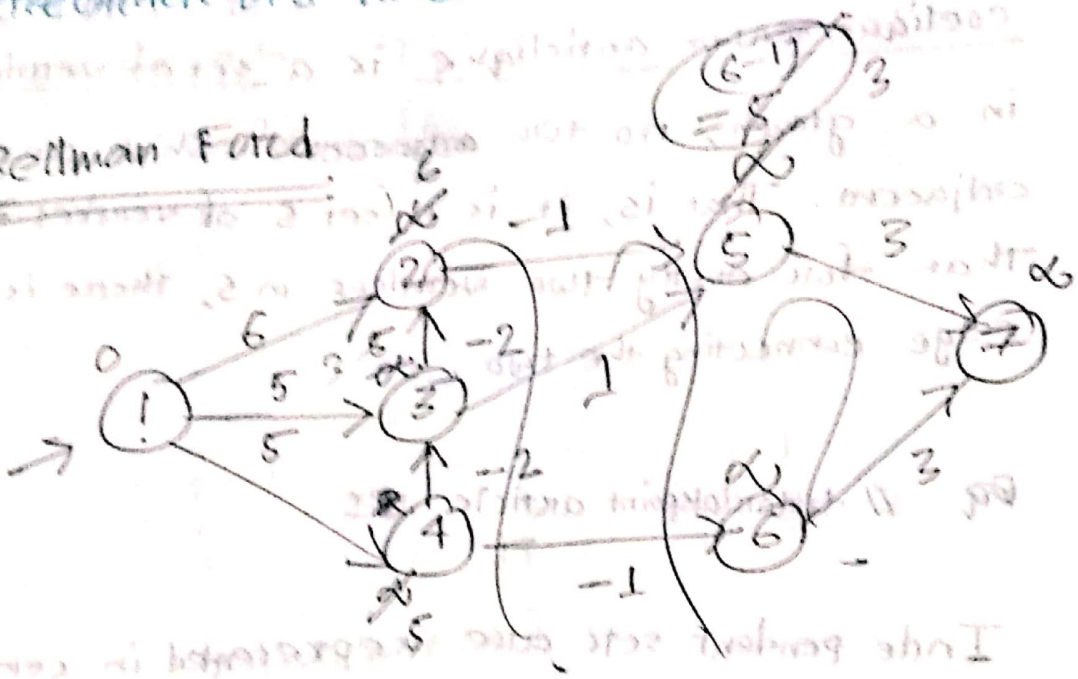
- there should not be any vertices adjacent to each other. There should not be any common edge between any two vertices.

// See tutorialspoint article
// See GFG Article - Largest Independent Set DP 26

Depth First Search

Pre Order DFS Time — see Rani video [1:00]

Bellman Ford:



// list of edges —

(1,2) (1,3) (1,4) (2,5)

(3,4) (3,5) (4,3) (4,6)

(5,7) (6,7)

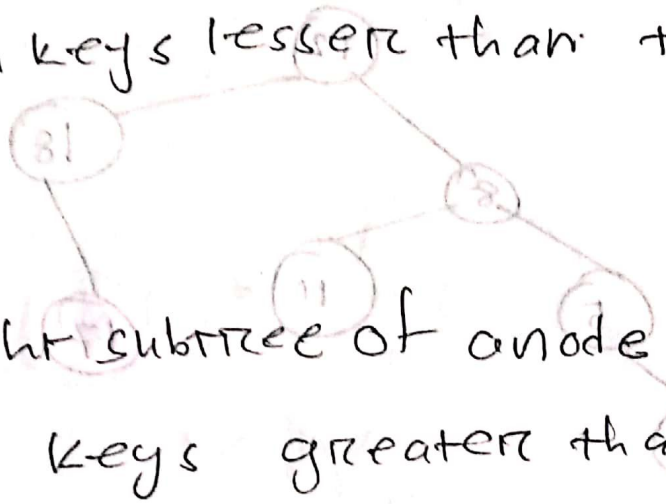
// relax the edges —

// relax (|V| - 1) times

// Abdul
// Bari
// Video
// A·A

Binary search tree is a node-based binary tree data structure which has the following properties —

- The left subtree of a node contains only nodes with keys lesser than the node's key.

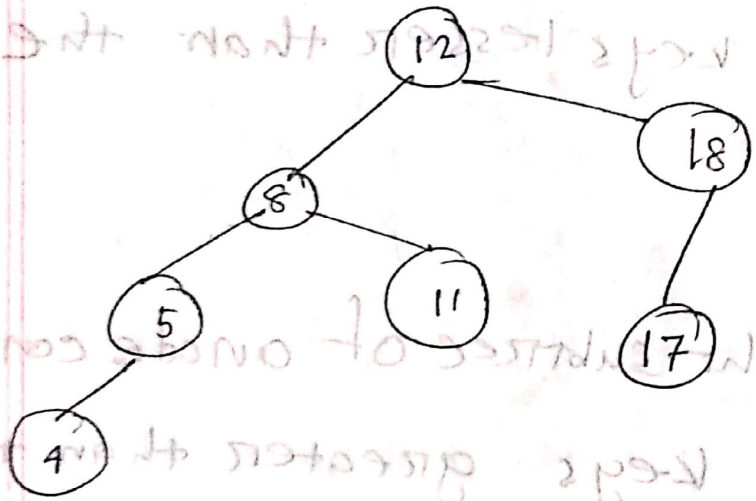


- The right subtree of a node contains only nodes with keys greater than the node's key.

- The left & right subtree each must be a binary search tree.

AVL Tree is a self-balanced BST where the difference between heights of ~~the~~ left and right subtree cannot be more than one for all nodes.

#



Tushar

Insertion

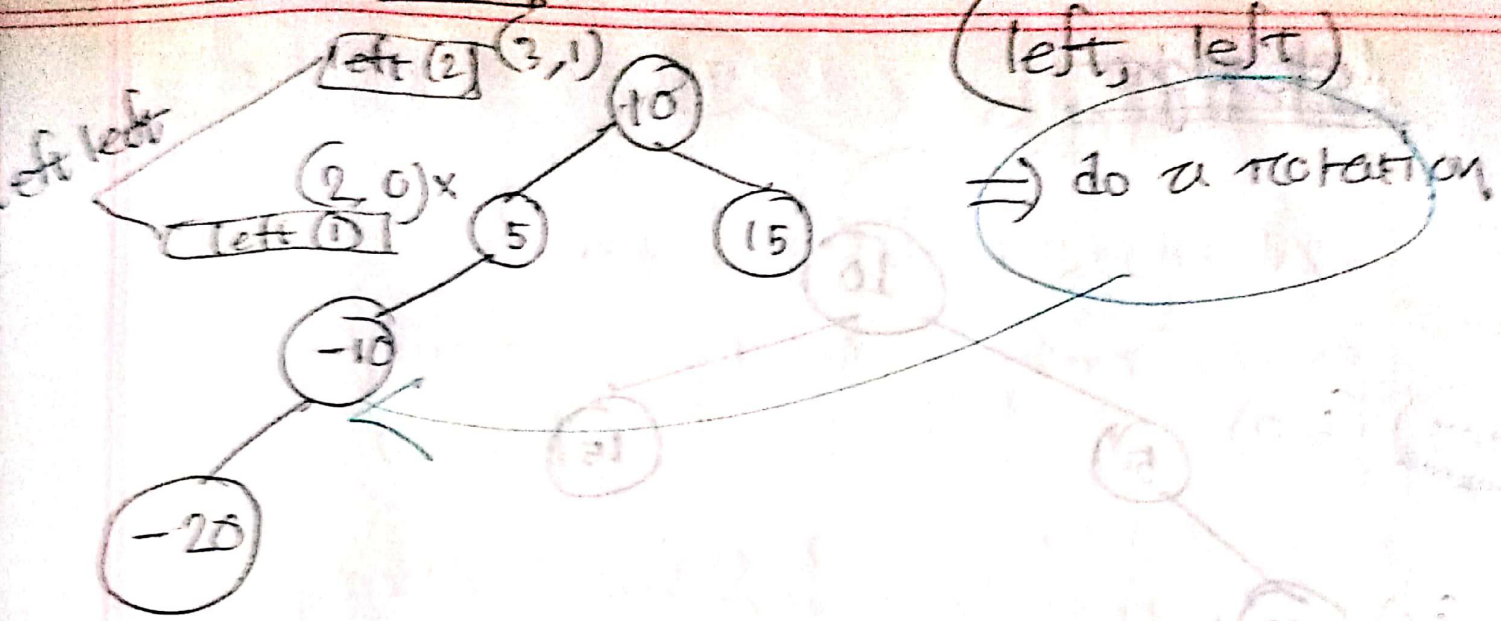
left left

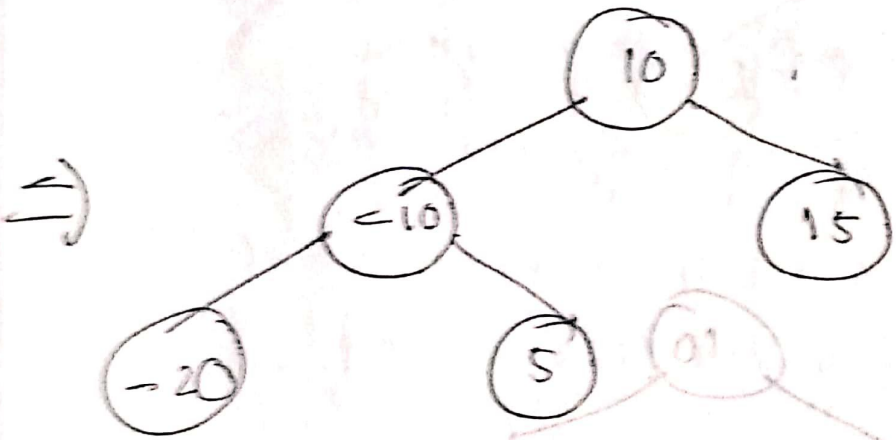
left right

right right

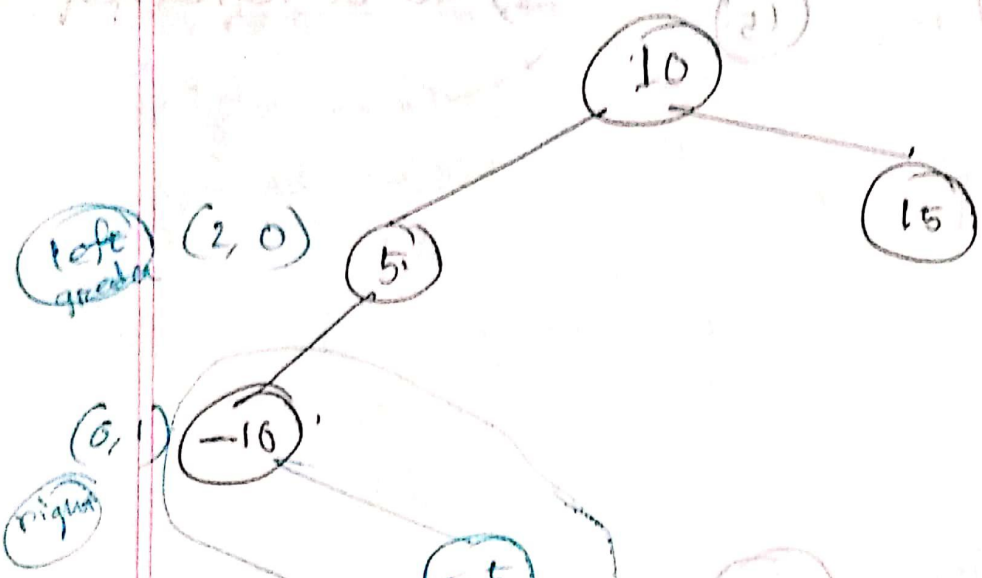
right left

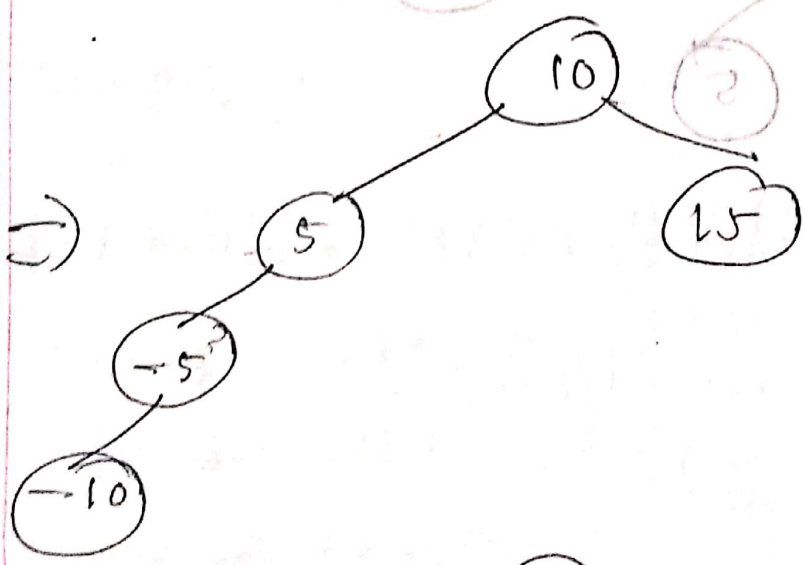
Insert -20

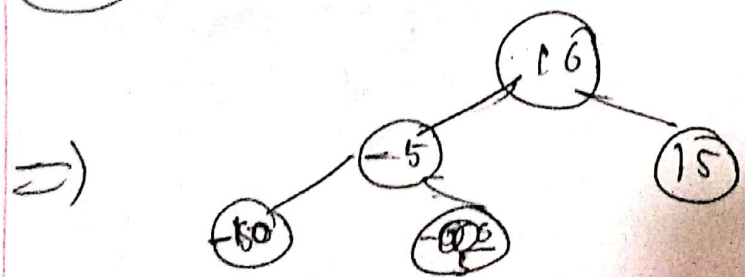

left (2)
(3,1)
10
left left
(2,0)x
left (1)
5
15
-10
-20
(left, left)
⇒ do a rotation

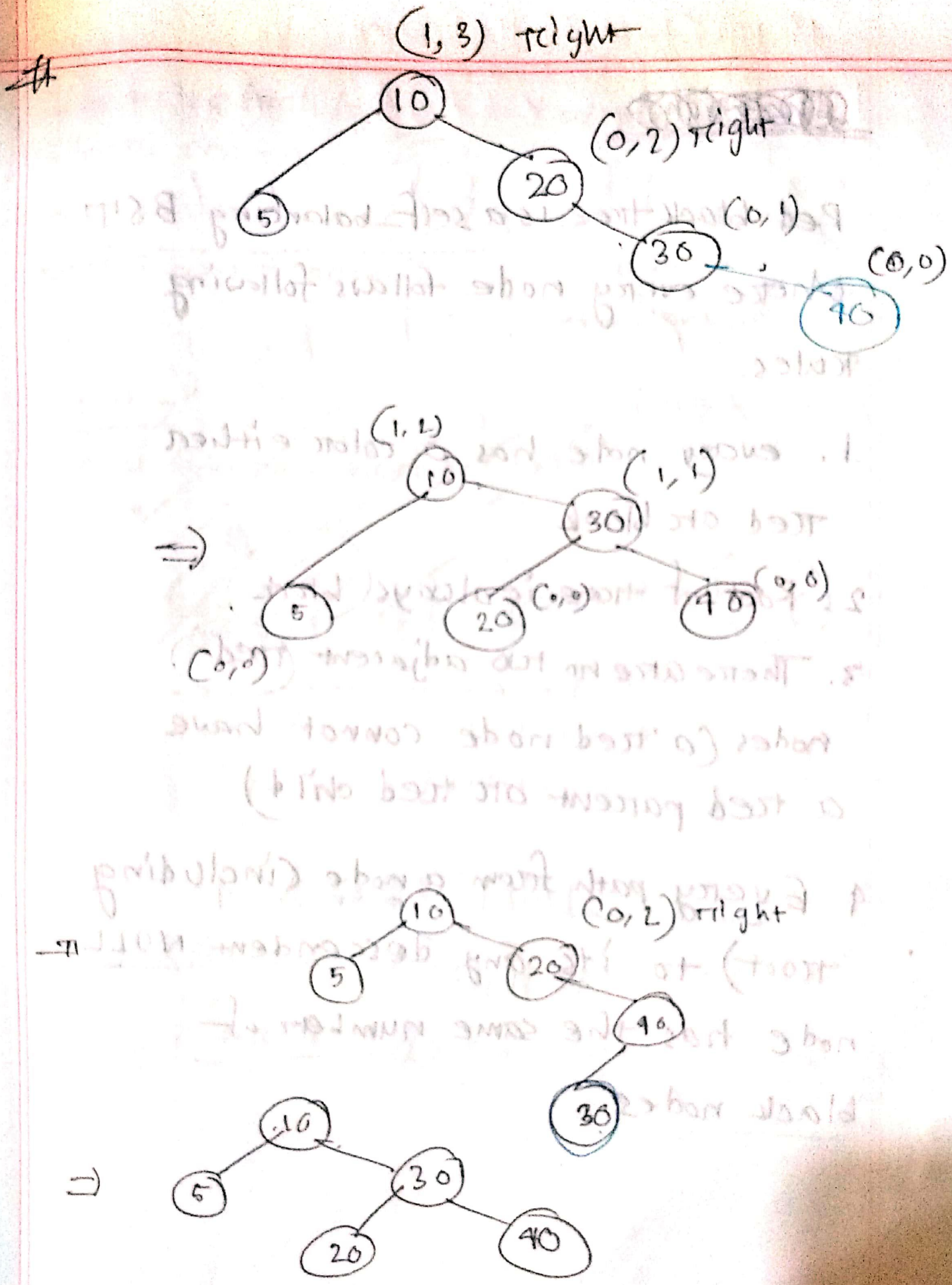

⇒
10
-10
15
-20
5

left, right





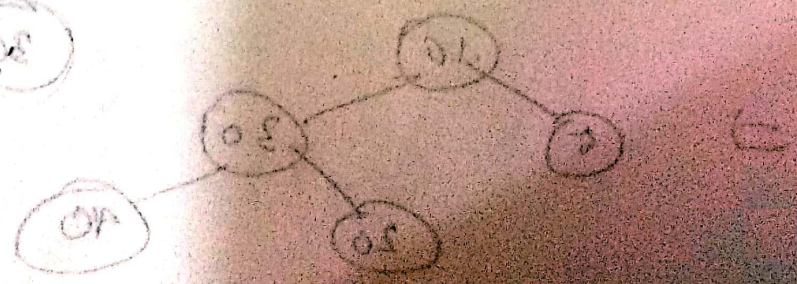



(1, 3) right
10
5
20
(0, 2) right
30
(0, 1)
40
(0, 0)
(1, 2)
10
5
(0, 0)
(1, 1)
30
20
(0, 0)
40
(0, 0)
10
5
(0, 2) right
20
40
30
10
5
30
20
40

Introduction

Red black tree is a self-balancing BST where every node follows following rules.

1. every node has a color either red or black
2. Root of tree is always black
3. There are no two adjacent red nodes (a red node cannot have a red parent or red child)
4. Every path from a node (including root) to its any descendent NULL node has the same number of black nodes



linear search — n

binary search — $\log n$

merge sort — $n \log n$

matrix multiplication — n^3

0/1 Knapsack

TSP

Sum of subsets

Graph Color

Hamilton Cycle

2^n

~~Dijkstra~~

~~# dijkstra — proof of correctness~~

~~#~~

~~Bard —~~