

- **Floyd warshall APSP**

```
int d[ELE][ELE], costing[ELE][ELE];
```

```
void apsp(int n)
```

```
{
```

```
    int temp;
```

```
    for(int i=1;i<=n;i++)
```

```
        for(int j=1;j<=n;j++)
```

```
        {
```

```
            if(costing[i][j]==-1)
```

```
                d[i][j]=MAX;
```

```
            else
```

```
                d[i][j]=costing[i][j];
```

```
        }
```

```
    for(int k=1;k<=n;k++)
```

```
        for(int i=1;i<=n;i++)
```

```
            for(int j=1;j<=n;j++)
```

```
            {
```

```
                temp=d[i][k]+d[k][j];
```

```
                if(temp<d[i][j])
```

```
                    d[i][j]=temp;
```

```
            }
```

```
}
```

- **BFS with queue**

```
void bfs (int n,int src)
```

```
{
    queue<int>Q;
    Q.push (src);
    int taken [100]={0},distance [100];
    taken [src]=1;
    distance [src]=0;
    while (!Q.empty ())
    {
        int u=Q.front ();
        for (int i=0;i<G [u].size ();i++)
        {
            int v=G [u][i];
            if(!taken [v])
            {
                distance [v]=distance [u]+1;
                taken [v]=1;
                Q.push (v);
            }
        }
        Q.pop ();
    }

    for (int i=1;i<=n;i++) printf("%d to %d distance %d\n",src,i,distance [i]);
}
```

- **BitMask**

```
bool check(int N,int pos){return (bool)(N & (1<<pos));}
int reset(int N,int pos){return N= N & ~(1<<pos);}
int Set(int N,int pos){return N=N|(1<<pos);}
```

- **Coin Change**

```
int minCoin[100000],process[10000],maxM;
```

```
vector<int>coins;
```

```
int coinNum;
```

```
m
```

```
int minCoinChange(int amount)
```

```
{
```

```
// cout<<amount<<endl;
```

```
if(minCoin[amount]!=-1)
```

```
    return minCoin[amount];
```

```
if(amount==0)
```

```
    return minCoin[amount]=0;
```

```
int temp,low=maxM;
```

```
for(int i=0;i<coinNum;i++)
```

```
{
```

```
    if(coins[i]>amount)
```

```
        break;
```

```
    temp=minCoinChange(amount-coins[i]);
```

```
    if(temp<low)
```

```
        low=temp;
```

```

}

minCoin[amount]=low+1;

return minCoin[amount];
}

int totalProcess(int amount)
{
    int processAmount[10000][10];

    for(int i=0;i<coinNum; i++)
        processAmount[0][i]=1;

    if(amount==0)//if there no amount needed then no coins needed!!
        return 1;
    // if(coinIndex<0)// no coins left but still amount to be filled
    //    return 0;

    for(int i=1;i<=amount;i++)
        for(int j=0;j<coinNum;j++)
        {
            int x,y;

            int temp=i-coins[j];

            if(temp<0)//coins[i] is bigger than amount needed

                x=0;

            else

                x=processAmount[temp][j]; //number of process after taking coins[i]

            if(j<1)//no more coins left after this

```

```

        y=0;

    else

        y=processAmount[i][j-1]; //number of process without taking coins[i]

        processAmount[i][j]=x+y;

    }

    return processAmount[amount][coinNum-1];

}

```

//Following is a simplified version of method 2. The auxiliary space required here is  $O(n)$  only.

**int count( int S[], int m, int n )**

```

{
    // table[i] will be storing the number of solutions for
    // value i. We need n+1 rows as the table is constructed
    // in bottom up manner using the base case (n = 0)

    int table[n+1];

    // Initialize all table values as 0
    memset(table, 0, sizeof(table));

    // Base case (If given value is 0)
    table[0] = 1;

    // Pick all coins one by one and update the table[] values
    // after the index greater than or equal to the value of the
    // picked coin

    for(int i=0; i<m; i++)
        for(int j=S[i]; j<=n; j++)
            table[j] += table[j-S[i]];
}

```

```

    return table[n];
}

int processCount(int amount)
{
    int coins[]={1,5,10,25,50};
    coinNum=5;
    memset(process,0, sizeof process);
    process[0]=1;

    for(int i=0;i<coinNum;i++)
    {
        for(int j=1;j<=amount;j++)
        {
            if(j+coins[i]>amount)
                break;
            if(process[j]!=0) //once it has been made
                process[j+coins[i]]+=process[j];
        }
    }
    return process[amount];
}

```

- **DFS**

```
int dfs(int index)
```

```
{
    if(visited[index]) return 0;
    int counting=1;
    visited[index]=true;
    for(int i=0;i<dependent[index].size(); i++)
    {
        if(!visited[dependent[index][i]])
            counting+=dfs(dependent[index][i]);
    }
    return counting;
}
```

- **Havel Hakimi**

```
vector<int> degree;
```

```
bool func(int a,int b)
```

```
{
    return a>b;
}
```

```
bool hakimi(int item)
```

```
{
    sort(degree.begin(),degree.end(),func);
    bool flag=true;
    int nonZero=0;
```

```

for(int i=0;i<item;i++)
    if(degree[i])
        nonZero++;
// item=test;
while(degree[0])
{
    if(degree[0]>=nonZero)
    {
        flag=false;
        break;
    }
    for(int i=1;i<=degree[0];i++)
    {
        degree[i]--;
        if(!degree[i])
            nonZero--;
    }
    degree[0]=0;
    nonZero--;
    sort(degree.begin(),degree.end(),func);
}

return flag;
}

```



- **Knpasack**

```
int profit[32][1005],w[32],price[32],mark[32],counting;
```

```
int knapsack(int totalItem,int totalWeight)
```

```
{
    int with,without;
    for(int i=0;i<=totalItem;i++)
        profit[i][0]=0;
    for(int j=0;j<=totalWeight;j++)
        profit[0][j]=0;
    for(int i=1;i<=totalItem;i++)
        for(int j=1;j<=totalWeight;j++)
        {
            if(w[i-1]<=j)
                with=profit[i-1][j-w[i-1]]+price[i-1];
            else
                with=0;
            without=profit[i-1][j];
            profit[i][j]=max(with,without);
        }
}
```

```
return profit[totalItem][totalWeight];
```

```
}
```

```
void backTrack(int item,int weight)
```

```
{
```

```
    if(!item || !weight)
```

```
        return;
```

```
    if((profit[item][weight]-price[item-1])==profit[item-1][weight-w[item-1]])
```

```
    {
```

```
        mark[item-1]=1;
```

```
        counting++;
```

```
        backTrack(item-1, weight-w[item-1]);
```

```
    }
```

```
    else if(profit[item-1][weight]>profit[item][weight-1])
```

```
        backTrack(item-1,weight);
```

```
    else
```

```
        backTrack(item,weight-1);
```

```
}
```

### • LCS

```
class lcs_bottom_up{
```

```
public:
```

```
    int bottom_up();
```

```
    string X,Y;
```

```
    int lcs[200][200];
```

```

int lenx,leny;

void find_len();

void print_em_all();

int maxi(int, int);

void traceback(int i, int j);

};

void lcs_bottom_up::find_len()
{
    lenx=X.size();
    leny=Y.size();
}

int lcs_bottom_up::bottom_up()
{
    find_len();
    for(int i=0; i<=lenx; i++)
        lcs[i][0]=0;
    for(int i=0; i<=leny; i++)
        lcs[0][i]=0;
    for(int i=1; i<=lenx; i++)
        for(int j=1; j<=leny; j++)
        {
            if(X[i-1]==Y[j-1])//i is the ith character of X which means i-1 index same goes for j with Y
                lcs[i][j]=1+lcs[i-1][j-1];
            else
                lcs[i][j]=maxi(lcs[i-1][j], lcs[i][j-1]);
        }
    }

```

```
//      cout<<"lcs("<<i<<","<<j<<")= "<<lcs[i][j]<<endl;

    }

    return lcs[lenx][leny];
}
```

```
void lcs_bottom_up::print_em_all()
```

```
{
    cout<<"\t";
    for(int i=0; i<=Y.size(); i++)
        cout<<i<<"\t";
    cout<<endl;
    for(int i=0; i<=Y.size()+1; i++)
        cout<<"---"<<"\t";
    cout<<endl;
    for(int i=0; i<=X.size();i++)
    {
        cout<<i<<"|\t";
        for(int j=0; j<=Y.size(); j++)
            cout<<lcs[i][j]<<"\t";
        cout<<endl;
    }
}
```

```
void lcs_bottom_up::traceback(int i, int j){
```

```
    cout<<i<<"\t"<<j<<endl;

    if(i == 0 || j == 0)return;
```

```

if(X[i-1] == Y[j-1]) {
    traceback(i-1, j-1);
    cout << X[i-1];
}
else if(lcs[i-1][j] > lcs[i][j-1])
    traceback(i-1, j);
else traceback(i, j-1);
}

```

- **LIS**

**vector<int> numbers,highLength;//all of them are 1 indexed array**

**int totalItem;**

**int lis(int index)**

```

{
    if(highLength[index]!=-1)
        return highLength[index];
    int max=0,temp;
    for(int v=index+1;v<=totalItem;v++)
        if(numbers[v]<numbers[index])
        {
            temp=lis(v);
            if(temp>max)
                max=temp;
        }
    return highLength[index]=max+1;
}

```

```
}
```

## LIS NlogN

```
void binary_search(int start, int end, int key)
```

```
{
    int mid;
    while(start<=end)
    {
        mid=(start+end)/2;
        if(tailTable[mid] == key)
            return;
        else if(tailTable[mid] > key)
            end=mid-1;
        else
            start=mid+1;
    }
    if(tailTable[start]<key) // no need; just for safety!!
        start++;
    tailTable[start]=key;
}
```

```
int lis(int elements)
```

```
{
    int i,n,cur,num,set=1;
    cur=1;
    tailTable[0]=numbers[0];
    for(int i=1;i<elements;i++)
```

```

{
    num=numbers[i];
    if(num>tailTable[cur-1])
        tailTable[cur++]=num;
    else if(num<tailTable[cur-1])
        binary_search(0,cur-1,num);
}
return cur;
}

```

### ***LDS $N\log N$***

```

void binary_search(int start, int end, int key)
{
    int mid;
    while(start<=end)
    {
        mid=(start+end)/2;
        if(tailTable[mid] == key)
            return;
        else if(tailTable[mid] < key)
            end=mid-1;
        else
            start=mid+1;
    }
    if(tailTable[start]>key) // no need; just for safety!!

```

```

    start++;
    tailTable[start]=key;
}

int lis(int elements)
{
    int i,n,cur,num,set=1;

    cur=1;
    tailTable[0]=numbers[0];
    for(int i=1;i<elements;i++)
    {
        num=numbers[i];
        if(num<tailTable[cur-1])
            tailTable[cur++]=num;
        else if(num>tailTable[cur-1])
            binary_search(0,cur-1,num);
    }
    return cur;
}

```

- **MST Cruchkal**

```

class MST{
public:
    vector<int>parent,usedEdges;

    int elements,costing,second_best,totalEdge;

    vector<pair<int,pair<int,int> > > v;

```



```

void input(int cost,int node1, int node2);

int rooting(int a);

bool makeUnion(int a,int b);

void initialize(int total);

int findFirstMst();

int findSecondMst();

};

```

```

void MST::input(int cost,int node1, int node2){
    v.push_back(make_pair(cost,make_pair(node1,node2)));
}

```

```

int MST::rooting(int a)
{
    if(parent[a]==a)
        return a;

    return (parent[a]=rooting (parent[a]));
}

```

```

bool MST::makeUnion(int a,int b)
{
    int p,q;

    p=rooting(a);

    q=rooting(b);

    if(p==q)
        return false;

    parent[p]=parent[q];
}

```

```

    return true;
}

void MST::initialize(int total)
{
    for(int i=0;i<=total;i++)
        parent.push_back(i);
}

int MST::findFirstMst()
{
    sort(v.begin(),v.end());
    totalEdge=v.size();
    costing=0;
    int edgeUsed=0;
    initialize(elements);
    for(int i=0;i<totalEdge;i++)
    {
        if(makeUnion(v[i].second.first,v[i].second.second))
        {
            usedEdges.push_back(i);
            costing+=v[i].first;
            edgeUsed++;

            if(edgeUsed==(elements-1))
                return costing;
        }
    }
}

```

```

}

int MST::findSecondMst()
{
    int edgeUsed=0,fin;
    second_best=1<<15;
    for(int j=0;j<elements-1;j++)
    {
        int costNow=0;
        edgeUsed=0;
        parent.clear();
        initialize(elements);
        for(int i=0;i<totalEdge;i++)
        {
            if(i==usedEdges[j])
                continue;
            if(makeUnion(v[i].second.first,v[i].second.second))
            {
                costNow+=v[i].first;
                edgeUsed++;
                if(edgeUsed==elements || costNow>=second_best)
                    break;
            }
        }
    }
    if(costNow<second_best && edgeUsed==(elements-1))
    {

```

```

        second_best=costNow;
//        if(second_best==costing)
//            return second_best;
    }
}
return second_best;
}

```

- **N-Queen**

```

int x[9],result=1,col[9];
bool place(int k,int i)
{
    for(int j=1;j<=i-1;j++)
    {
        if(col[j]==k)
            return false;
        if(abs(col[j]-k)==abs(j-i))
            return false;
    }
    return true;}

```

### **Segment tree**

```

class SegmentTree{
public:
    int ary[ELE],save[3*ELE],maxM;
    SegmentTree();
    int makeTree(int leftLim,int rightLim, int node);

```

```

int query(int rangeStart,int rangeFin,int givenLeft,int givenRight,int node);

int updateTree(int leftLim,int rightLim, int node, int x, int value);

void print(int );

};

```

**SegmentTree::SegmentTree()**

```

{
    maxM=1<<15;
}

```

**int SegmentTree::makeTree(int leftLim,int rightLim, int node)**

```

{
    if(leftLim==rightLim)
        return save[node]=ary[leftLim];

    int x,y;
    x=makeTree(leftLim,(leftLim+rightLim)/2,2*node);
    y=makeTree((leftLim+rightLim)/2+1,rightLim,2*node+1);
    return save[node]=min(x,y);}

```

**int SegmentTree::query(int rangeStart,int rangeFin,int givenLeft,int givenRight,int node)**

```

{
    if(rangeStart>givenRight || rangeFin<givenLeft)
        return maxM;

    if(rangeStart>=givenLeft && rangeFin<=givenRight)

```

```
return save[node];
```

```
int x,y;
```

```
x=query(rangeStart,(rangeStart+rangeFin)/2,givenLeft,givenRight,node*2);
```

```
y=query((rangeStart+rangeFin)/2+1,rangeFin,givenLeft,givenRight,node*2+1);
```

```
return min(x,y);
```

```
}
```

```
int SegmentTree::updateTree(int leftLim,int rightLim, int node, int x,int value)
```

```
{
```

```
if(leftLim==rightLim && leftLim==x)
```

```
    return save[node]=value;
```

```
if(leftLim>x || rightLim<x)
```

```
    return save[node];
```

```
int a,b;
```

```
a=updateTree(leftLim,(leftLim+rightLim)/2, 2*node,x,value);
```

```
b=updateTree((leftLim+rightLim)/2+1, rightLim, 2*node+1, x, value);
```

```
return save[node]=min(a,b);
```

```
}
```

## Segment Tree Sum

```
int ary[ELE],save[3*ELE];
```

```
int makeTree(int leftLim,int rightLim, int node)
```

```
{
    if(leftLim==rightLim)
        return save[node]=ary[leftLim];

    int x,y;
    x=makeTree(leftLim,(leftLim+rightLim)/2,2*node);
    y=makeTree((leftLim+rightLim)/2+1,rightLim,2*node+1);
    return save[node]=x+y;
}
```

```
int query(int rangeStart,int rangeFin,int givenLeft,int givenRight,int node)
```

```
{
    if(rangeStart>givenRight || rangeFin<givenLeft)
        return 0;

    if(rangeStart>=givenLeft && rangeFin<=givenRight)
        return save[node];

    int x,y;
    x=query(rangeStart,(rangeStart+rangeFin)/2,givenLeft,givenRight,node*2);
    y=query((rangeStart+rangeFin)/2+1,rangeFin,givenLeft,givenRight,node*2+1);
    return x+y;
}
```

```
int updateTree(int leftLim,int rightLim, int node, int x,int value)
```

```
{
    if(leftLim==rightLim && leftLim==x)
        return save[node]=value;

    if(leftLim>x || rightLim<x)
        return save[node];

    int a,b;
    a=updateTree(leftLim,(leftLim+rightLim)/2, 2*node,x,value);
    b=updateTree((leftLim+rightLim)/2+1, rightLim, 2*node+1, x, value);
    return save[node]=a+b;
}
```

### ***Lazy Propagation***

```
//not tested yet
```

```
int ary[ELE],save[3*ELE],prop[3*ELE];
```

```
//memset(prop,0,sizeof prop);
```

```
int makeTree(int node,int leftLim,int rightLim)
```

```
{
    if(leftLim==rightLim)
        return save[node]=ary[leftLim];
}
```



```

int x,y;

x=makeTree(2*node,leftLim,(leftLim+rightLim)/2);

y=makeTree(2*node+1,(leftLim+rightLim)/2+1,rightLim);

return save[node]=x+y;

}

void updateTree(int node,int leftLim,int rightLim, int givenLeft, int givenRight,int value)
{
    if(leftLim>givenRight || rightLim<givenLeft)return;

    if(leftLim>=givenLeft && rightLim<=givenRight)
    {
        save[node]+=(rightLim-leftLim+1)*value;
        prop[node]+=value;
        return;
    }

    int left,right,mid;
    left=node*2;
    right=node+1;
    mid=(leftLim+rightLim)/2;

    updateTree(2*node, leftLim,mid,givenLeft,givenRight,value);
    updateTree(2*node+1,mid+1,rightLim,givenLeft,givenRight,value);

```

save[node]=save[2\*node]+save[2\*node+1]+(rightLim-leftLim+1)\*prop[node];//if this save [node] was previously upgraded then prop[node] will be non-zero

}

**int query(int node, int leftLim, int rightLim, int givenLeft, int givenRight, int carry)**

{

if(leftLim>givenRight || rightLim<givenLeft) return 0;

if(leftLim>=givenLeft && rightLim<=givenRight)

return save[node]+(rightLim-leftLim+1)\*carry;

int p,q,mid;

mid=(leftLim+rightLim)/2;

p=query(2\*node,leftLim,mid,givenLeft,givenRight,carry+prop[node]);

q=query(2\*node+1,mid+1,rightLim,givenLeft,givenRight,carry+prop[node]);

return p+q;}

### • **Sum of Subsets**

**class non\_Sos{**

public:

int high,totalItem,maxM;

vector<int> item,temp,ans;

void backtrack(int nowSum,int index);

void printAns();

non\_Sos();

};

```
non_Sos::non_Sos()
```

```
{
    high=-1<<15;
}
```

```
void non_Sos::backtrack(int nowSum,int index)
```

```
{
    if(nowSum>maxM)//as the sum never can be greater than the maxM value
        return;

    if(nowSum>high)//this is the optimal solution till now
    {
        //changing the value of high & answer vector
        high=nowSum;
        ans=temp;
    }

    if(index==item.size())
        return;

    temp.push_back(item[index]);

    backtrack(nowSum+item[index],index+1);// Including x indexed data

    temp.pop_back();

    backtrack(nowSum,index+1);// Excluding x indexed data
```

```

void non_Sos::printAns()
{
    int totalSolution=ans.size();
    for(int i=0;i<totalSolution;i++)
        printf("%d ",ans[i]);
    printf("sum:%d\n",high);
}

```

- **Topological sort**

```

class topSort{
public:
    vector<int>dependent[100];
    int indegree [100],taken[100],elements,result[100];
    stack<int>resultStack;
    void input(int, int);
    void topologicalSort2D();
    void topSortStack(int);
    topSort(int);
};

```

```

topSort::topSort(int x)
{
    memset(indegree,0,sizeof(indegree));

```

```

memset(taken,0,sizeof(taken));

elements=x;
}

```

```

void topSort::input(int x,int y)

```

```

{
    dependent[x].push_back(y);
    indegree[y]++;
}

```

```

void topSort::topologicalSort2D()

```

```

{
    int index=0;
    for(int i=0;i<=elements;i++)
    {
        if(!indegree[i] && !taken[i])
        {
            taken[i]=1;
            result[index++]=i;
            for(int j=0;j<dependent[i].size(); j++)
            {
                int temp=dependent[i][j];
                indegree[temp]--;
            }
        }
        if(index==elements)

```

```

        return;

        i=-1;

    }

}

}

```

- **Travelling Salesman**

```
int costing[CITY_NO][CITY_NO],ending[CITY_NO],totalCity,dp[CITY_NO][TURNS],E;
```

```
int salesman(int city,int turnsLeft)
```

```

{
    if(dp[city][turnsLeft])
        return dp[city][turnsLeft];
    if(turnsLeft==0)
        return dp[city][0]=0;
    int high=-1<<31,temp;
    if(turnsLeft==1)
    {
        for(int i=1;i<=E;i++)
        {
            temp=costing[city][ending[i]];
            high=max(temp,high);
        }
        return dp[city][1]=high;
    }
    for(int i=1;i<=totalCity;i++)
    {

```

```
temp=costing[city][i]+salesman(i,turnsLeft-1);  
high=max(high,temp);  
}  
return dp[city][turnsLeft]=high;}
```

KUET DOUR