INTRODUCTION

**An Algorithm** is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time.

Every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

**Time Complexity:** The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

**Space Complexity:** The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components: • Space needed by constants and simple variables in program. • Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

**Best case, Worst Case and Average case Efficiencies**

**Best-case efficiency:** Efficiency (number of basic operation will be executed) for best case input of size n i.e. the algorithm runs fastest among all possible inputs.

**Worst case efficiency**: Efficiency (number of basic operation will be executed) for worst case input or size n i.e. the algorithm runs longest among all possible inputs.

LITERATURE SURVEY

**Find Minimum Cost Spanning Tree of a given undirected graph using Kruskal's algorithm.**

A greedy algorithm makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. The choice made at each step must be:

* + Feasible: Satisfy the problem’s constraints
  + locally optimal: Be the best local choice among all feasible choices
  + Irrevocable: Once made, the choice can’t be changed on subsequent steps.

**Spanning tree** of a connected graph *G*: a connected acyclic sub graph (tree) of *G* that includes all of *G*’s vertices.

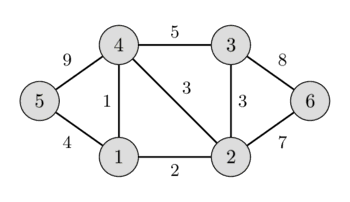
**Minimum Spanning Tree** of a weighted, connected graph *G*: a spanning tree of *G* of minimum total weight.

* Kruskal’s algorithm finds the minimum spanning tree for a weighted connected graph G=(V,E) to get an acyclic subgraph with |V|-1 edges for which the sum of edge weights is the smallest.
* Consequently the algorithm constructs the minimum spanning tree as an expanding sequence of subgraphs, which are always acyclic but are not necessarily connected on the intermediate stages of algorithm.
* The algorithm begins by sorting the graph’s edges in non-decreasing order of their weights. Then starting with the empty subgraph, it scans the sorted list adding the next edge on the list to the current sub graph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

**Formulation of the Problem:**

**Let directed weight graph G = (V, E), where V is the set of n vertices; E is the set of m edges and W is the set of weights associated from vi to vj of the graph. Let’s assume, eij = the edge from vertices vi to vj. wij = The weight of the edge eij. From the following rule the weight matrix W of the G is constructed: If there is an edge from vi to vj presented in G then Set, W[i,j] = wij else Set, W[i,j] = 0 Figure 1 shows the directed weight graph G1**

**Example:**

****

**Complexity Analysis of the Algorithm:**

**The nxn weight matrix W is the input to the algorithm, where n is the number of the vertices in the graph. It selects n-1 elements to produce T from n2 elements. Basically it searches n2-1 elements to produce T. Hence the complexity of this algorithm is O(n2).**

**Best Case Analysis:**

**The situation for the best case is, when, only the elements in first row or first column are available for mark. Other rows or columns are marked as 0. In this situation the complexity for this algorithm will be O(n). When constructing the Directed Minimum Cost Spanning tree with this type of element the algorithm will consume O(n) time. Hence the best case will be O(n) and instead of n, if m replaced, the best case time complexity will be in order of O(m), where m = n-1**

**Worst Case Analysis:**

**The situation for the worst case is, when all the elements in matrix W is consider for searching and marking suitable edges. In this situation the complexity will be O(n2). When constructing the Directed Minimum Cost Spanning tree with consider all the elements of matrix W, the algorithm will consume O(n2) time. Hence the best case will be O(n2) and instead of n, if m replaced, the worst**

**case time complexity will be in order of O(m2), where m = n-1. If n is large, the complexity O(m) and O(m2) are better than O(n) and O(n2), where m = n-1. Include conclusion**

**Program:**

**#include<stdio.h.h>**

**#include<conio.h>**

**#include<time.h>**

**#include<dos.h>**

**int n,a,b,u,v,node=1;**

**int visited[10],min,totcost=0,cost[10][10],parent[10];**

**void main()**

**{**

**clrscr();**

**clock\_t start,end;**

**start=clock();**

**cout<<"Enter the number of nodes:"<<endl;**

**cin>>n;**

**cout<<"Enter the adjacency matrix:"<<endl;**

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

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

**{**

**cin>>cost[i][j];**

**if(cost[i][j]==0)**

**cost[i][j]=999;**

**}**

**while(node<n)**

**{**

**min=999;**

**for(int x=1;x<=n;x++)**

**for(int y=1;y<=n;y++)**

**if(cost[x][y]<min)**

**{**

**min=cost[x][y];**

**a=u=x;**

**b=v=y;**

**}**

**while(parent[u])**

**u=parent[u];**

**while(parent[v])**

**v=parent[v];**

**if(u!=v)**

**{**

**node++;**

**cout<<"edge"<<a<<" -> "<<b<<" = "<<min;**

**totcost+=min;**

**parent[v]=u;**

**}**

**cost[a][b]=cost[b][a]=999;**

**}**

**cout<<"\nTotal cost ="<<totcost;**

**delay(5);**

**end=clock();**

**cout<<"\nTime taken for execution:"<<(end-start)/CLK\_TCK;**

**getch();**

**}**

**Find the Binomial Co-efficient using Dynamic Programming.**

**Theorem Statement:**

In [elementary algebra](https://en.wikipedia.org/wiki/Elementary_algebra), the **binomial theorem** (or **binomial expansion**) describes the algebraic expansion of [powers](https://en.wikipedia.org/wiki/Exponentiation) of a [binomial](https://en.wikipedia.org/wiki/Binomial_(polynomial)). According to the theorem, it is possible to expand the polynomial (*x* + *y*)*n* into a [sum](https://en.wikipedia.org/wiki/Summation) involving terms of the form *a xb* *yc*, where the exponents *b* and *c* are [nonnegative integers](https://en.wikipedia.org/wiki/Nonnegative_integer) with *b* + *c* = *n*, and the [coefficient](https://en.wikipedia.org/wiki/Coefficient) *a* of each term is a specific [positive integer](https://en.wikipedia.org/wiki/Positive_integer) depending on *n* and *b*. For example (for *n* = 4),

(x+y)4=x4+4x3y+6x2y2+4xy3+y4

FORMULA:

The formal expression of the Binomial Theorem is as follows:

(a + b)^n = sum[k=0,n][(n over k)a^(n-k)b^k]

The parenthetical bit above has these equivalents:

(n over k) = nCk = n!/[(n-k)!k!]

Example:

**a+b** is a binomial (the two terms are **a** and **b**)

Let us multiply **a+b** by itself using [Polynomial Multiplication](https://www.mathsisfun.com/algebra/polynomials-multiplying.html) :

(a+b)(a+b) = **a2 + 2ab + b2**

Now take that result and multiply by **a+b** again:

(a2 + 2ab + b2)(a+b) = **a3 + 3a2b + 3ab2 + b3**

And again:

(a3 + 3a2b + 3ab2 + b3)(a+b) = **a4 + 4a3b + 6a2b2 + 4ab3 + b4**

**ALGORITHM Binomial(*n*,*k*)**

Computes *C*(*n*, *k*) by the dynamic programming algorithm

//Input: A pair of nonnegative integers n ≥ k ≥ 0

//Output: The value of *C*(*n* ,*k*)

for *i🡨*0 to n do

for *j🡨*⇓0 to min (*i* ,*k*) do

if *j* = 0 or *j* = *k*

*C* [*i* , *j*]🡨 1

else *C* [*i* , *j*] 🡨 *C*[*i*-1, *j*-1] + *C*[*i*-1, *j*]

return C [n, k]

Number of Additions=A(n,k)

Program:

#include <iostream.h>

#include <conio.h>

#define MAX 10

int bicoef(int n,int k)

{

int c[MAX][MAX];

for (int i=0;i<=n;i++)

for (int j=0;j<=i,j<=k);j++)

if(j==0||j==i)

c[i][j]=1;

else

c[i][j]=c[i-1][j-1]+c[i-1][j];

return( c[n][k]);

}

void main()

{

int n,k,ncr;

cout<<"Enter Positive value of n : "<<endl ;

cin>>n;

cout<<"Enter Positive value of k : "<<endl ;

cin>>k;

if(k<=n)

{

ncr=bicoef(n,k);

cout<<"c("<<n<<","<<k<<")="<<ncr<<endl;

}

else

printf("\n !!! Can not compute Binomial...n must be less than k !!!");

getch();

}

OUPUT: