**Experiment no: 4 Backtracking Algorithm**

**Date:**

**Aim:** To implement the following Algorithms using the Backtracking Algorithm

i) N-Queens Problem

ii) Sum of Subset Problem

iii) M-Coloring Problem

iv) Hamiltonian Cycles

v)0/1 Knapsack Problem

**Theory:**

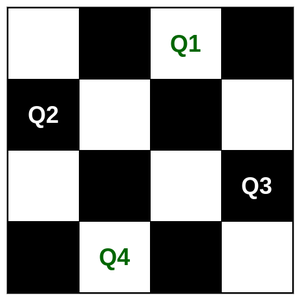
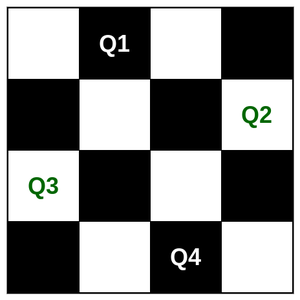
* Backtracking is a systematic approach to problem-solving that involves trying different solutions and undoing choices when they are found to be incorrect.
* It is commonly used for problems that require finding all possible solutions or a specific solution within a large search space.
* The algorithm explores the search space incrementally by building a solution one step at a time and backtracking when it reaches an invalid or unsatisfactory state.
* Recursion is often used to handle the search process, with each recursive call representing a step or decision in the problem-solving process.
* The algorithm maintains a state or configuration that represents the progress made so far in the search.
* At each step, the algorithm makes choices or selects options available and evaluates their validity or desirability.
* Bounding functions are used to determine if a partial solution can possibly lead to a valid solution or if it should be abandoned to save computational resources.
* If a choice leads to a valid solution, it is accepted and the algorithm continues to explore further.
* If a choice leads to an invalid or unsatisfactory state, the algorithm backtracks, undoing the choice and trying an alternative option.
* Backtracking continues until either a solution is found or all possible options have been exhausted.
* The algorithm often utilizes pruning techniques to optimize the search process by eliminating unnecessary or redundant choices.
* Backtracking is applicable to problems with "optimal substructure," meaning that a solution can be constructed from solutions to smaller subproblems.
* It is commonly used in various problem domains such as combinatorial optimization, constraint satisfaction, puzzles, graph problems, and more.

**a)N-Queens Problem**

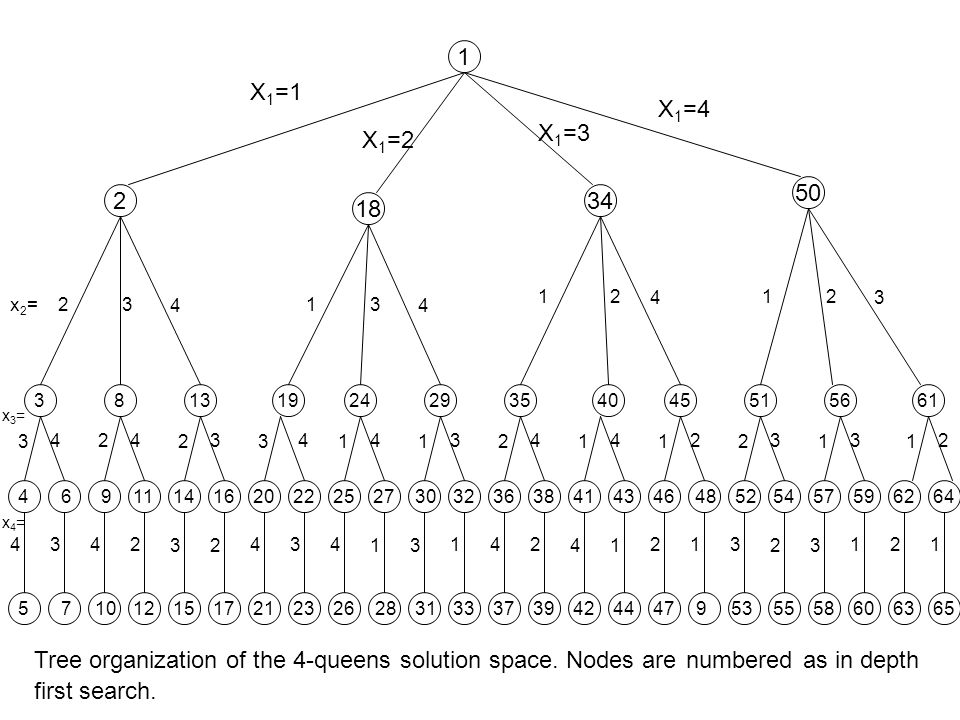
**Date:**

**Problem Statement:** Write a C program to find all the possible placements of N queens on an NxN board, such that they don't attack each other, thereby solving the N-Queens Problem.

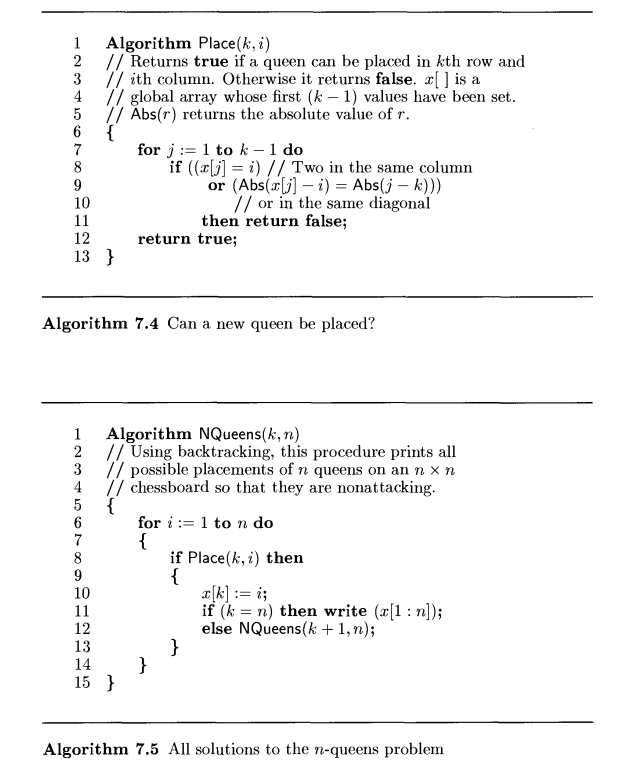
(n = 4)



**State Space tree for 4-Queens**



**Algorithm:**



**Time Complexity:**

O(n!) (where n = number of queens)

**Space Complexity:**

O(n2)

**Code:**

#include <stdio.h>

#include <stdlib.h>

#define MAX 100

int M[MAX];

int Place(int k, int i)

{

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

{

if (M[j] == i)

return 0;

else if (abs(i - M[j]) == abs(k - j))

return 0;

}

return 1;

}

void write(int n)

{

printf("\n\n");

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

{

printf("\t%d", i + 1);

}

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

{

printf("\n\n%d", i + 1);

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

{

if (M[i] == j)

printf("\tQ");

else

printf("\tX");

}

};

printf("\n\n");

}

void NQueen(int k, int n)

{

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

{

if (Place(k, i))

{

M[k] = i;

if (k == n - 1)

{

write(n);

return;

}

else

NQueen(k + 1, n);

}

}

}

int main()

{

int n;

printf("Enter value of n:");

scanf("%d", &n);

NQueen(0, n);

return 0;

}

**Output:**



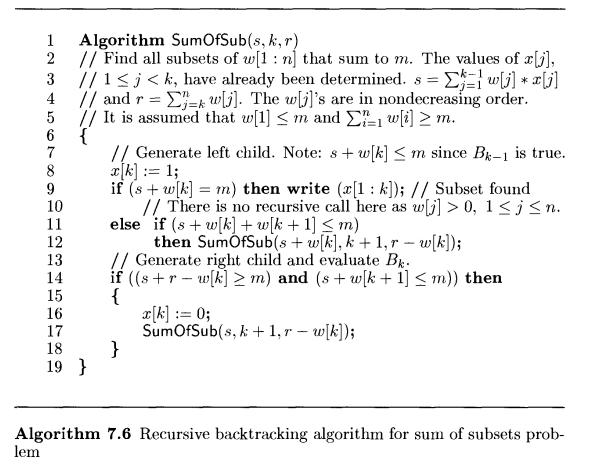
**b)Sum of Subsets (using Backtracking)**

**Date:**

**Problem Statement:** Given ‘n’ weights, find the combinations of these weights such that the total weight of the subset equal to ‘m’ (where n and m are given)

W = {5, 7, 10, 12, 15, 18, 20} m = 35

**Algorithm:**



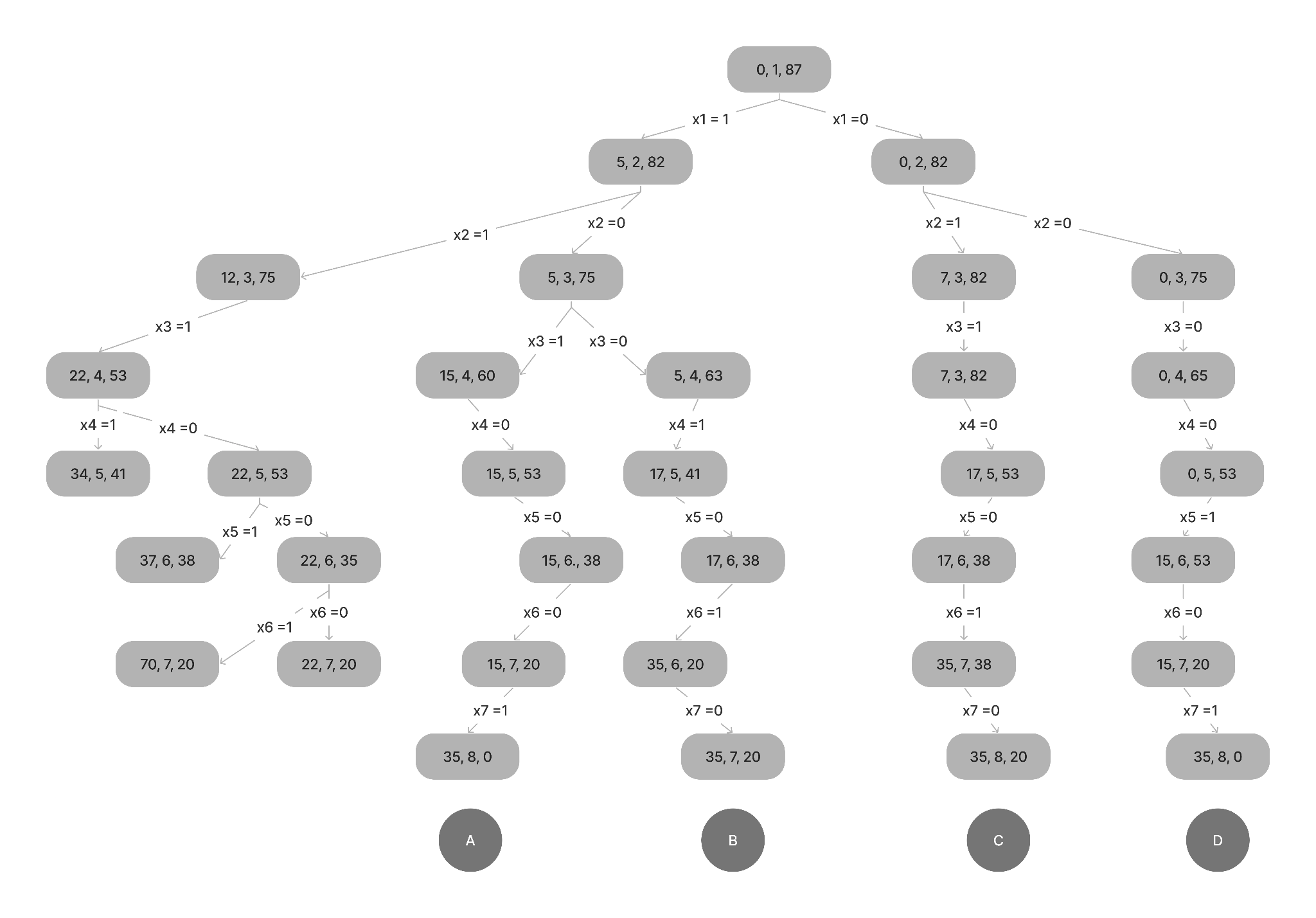
**Time Complexity:**

O(mn)

**Space Complexity:**

O(mn)

**State Space tree:**



**Code:**

#include <stdio.h>

#define MAX 100

int w[MAX], x[MAX];

int m, n;

void write(int k)

{

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

{

printf("%d ", x[i]);

}

printf(" i.e weights : ");

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

{

if (x[i] == 1)

printf("%d ", w[i]);

}

printf("\n");

}

void SumofSubsets(int s, int k, int r)

{

if (k > n)

return;

x[k] = 1;

if (s + w[k] == m)

{

write(k);

}

else if (s + w[k] + w[k + 1] <= m)

{

SumofSubsets(s + w[k], k + 1, r);

}

if ((s + r - w[k] >= m) && (s + w[k + 1] <= m))

{

x[k] = 0;

SumofSubsets(s, k + 1, r);

}

}

int main()

{

int r = 0;

printf("Enter number of elements.");

scanf("%d", &n);

printf("Enter the elements of the set.\n");

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

{

x[i] = 0;

scanf("%d", &w[i]);

r += w[i];

}

printf("Enter sum.\n");

scanf("%d", &m);

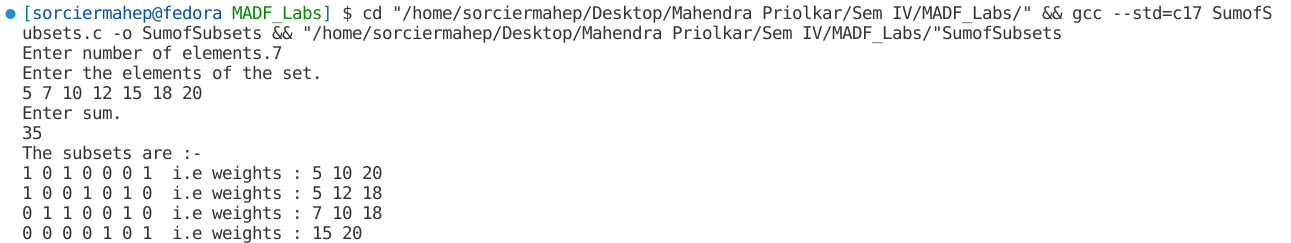
printf("The subsets are :-\n");

SumofSubsets(0, 0, r);

return 0;

}

**Output:**

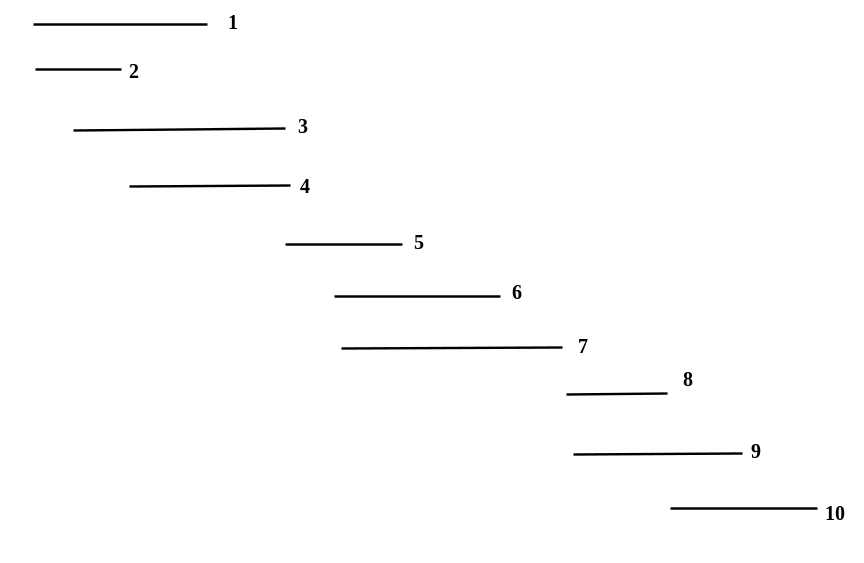


**c)M-Coloring**

**Date:**

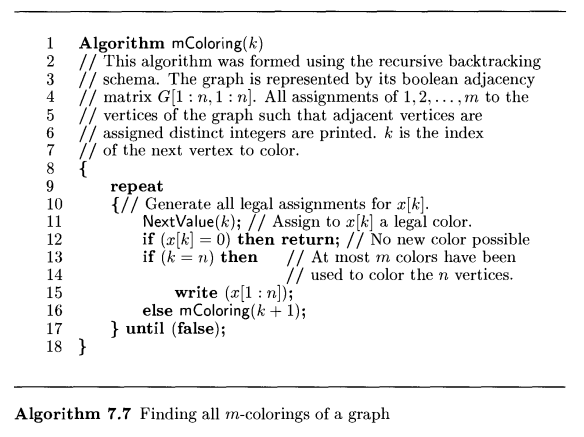
**Problem Statement:** Solve the following M-colorability problems

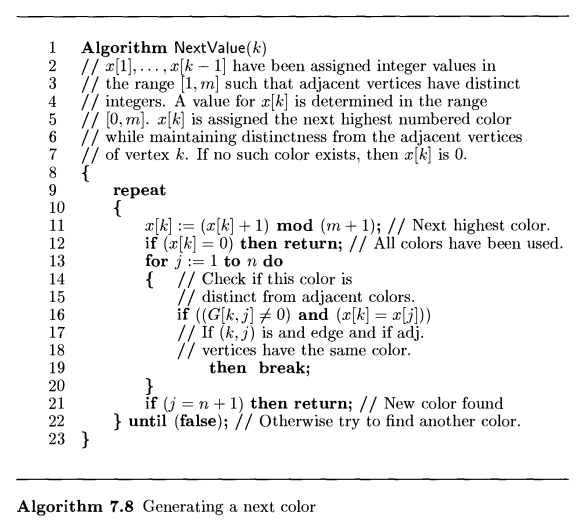
1)Consider the following active routes vs time:



Calculate the minimum amount of taxis required for operating on the routes, and all the possible permutations.

**Algorithm:**





**Time Complexity:**

O(nmn)

**Space Complexity:**

O(mn)

**Code:**

#include <stdio.h>

#define N 20

int G[N][N], count = 0;

char C[8] = {'X', 'V', 'I', 'B', 'G', 'Y', 'O', 'R'};

void accept\_graph(int n)

{

int max\_edges = n \* n;

int origin, destin;

printf("Enter the edges of the graph.\n");

printf("Enter the edge;( 0 0 ) to quit :\n");

for (int i = 0; i < max\_edges; i++)

{

scanf("%d %d", &origin, &destin);

if ((origin == 0) && (destin == 0))

break;

if (origin > n || destin > n || origin <= 0 || destin <= 0)

{

printf("Invalid edge.\n");

i--;

}

else

{

G[origin - 1][destin - 1] = 1;

G[destin - 1][origin - 1] = 1;

}

}

}

void show(int x[N], int nodes)

{

printf("%d.\t", ++count);

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

printf("%c ", C[x[i]]);

printf("\t");

}

void nextvalue(int x[N], int nodes, int colours, int k)

{

do

{

x[k] = (x[k] + 1) % (colours + 1);

if (!x[k])

return;

int j;

for (j = 0; j < nodes; j++)

{

if ((G[k][j] != 0) && (x[k] == x[j]))

break;

}

if (j == nodes)

return;

} while (1);

}

void colour(int x[N], int nodes, int colours, int k)

{

do

{

nextvalue(x, nodes, colours, k);

if (!x[k])

return;

if (k == nodes - 1)

show(x, nodes);

else

colour(x, nodes, colours, k + 1);

} while (1);

}

int main()

{

int nodes, colours, x[N];

printf("Enter the number of nodes:");

scanf("%d", &nodes);

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

for (int j = 0; j < nodes; j++)

G[i][j] = 0;

accept\_graph(nodes);

printf("Enter the number of colors:");

scanf("%d", &colours);

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

x[i] = 0;

printf("\nGraph colouring possibilities are:\n");

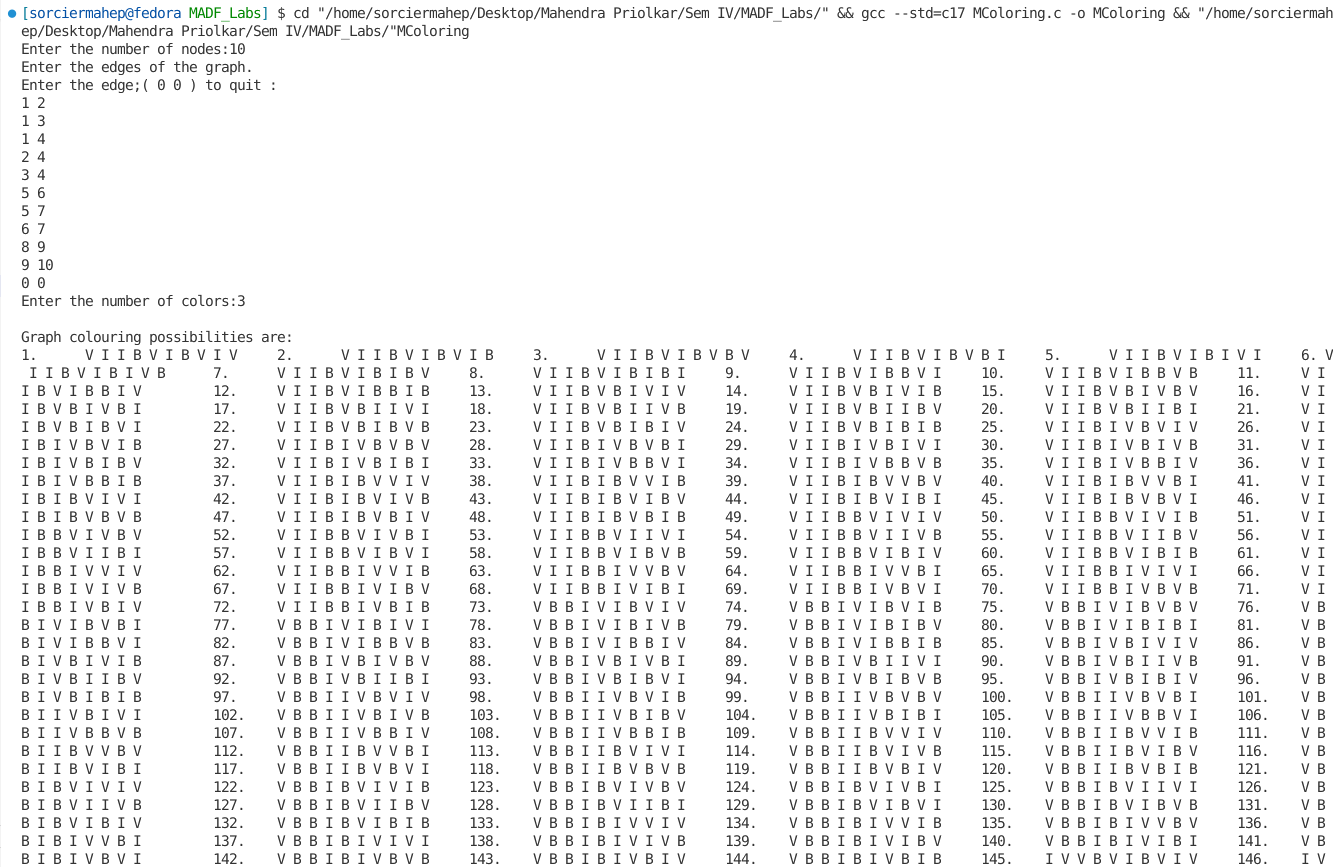
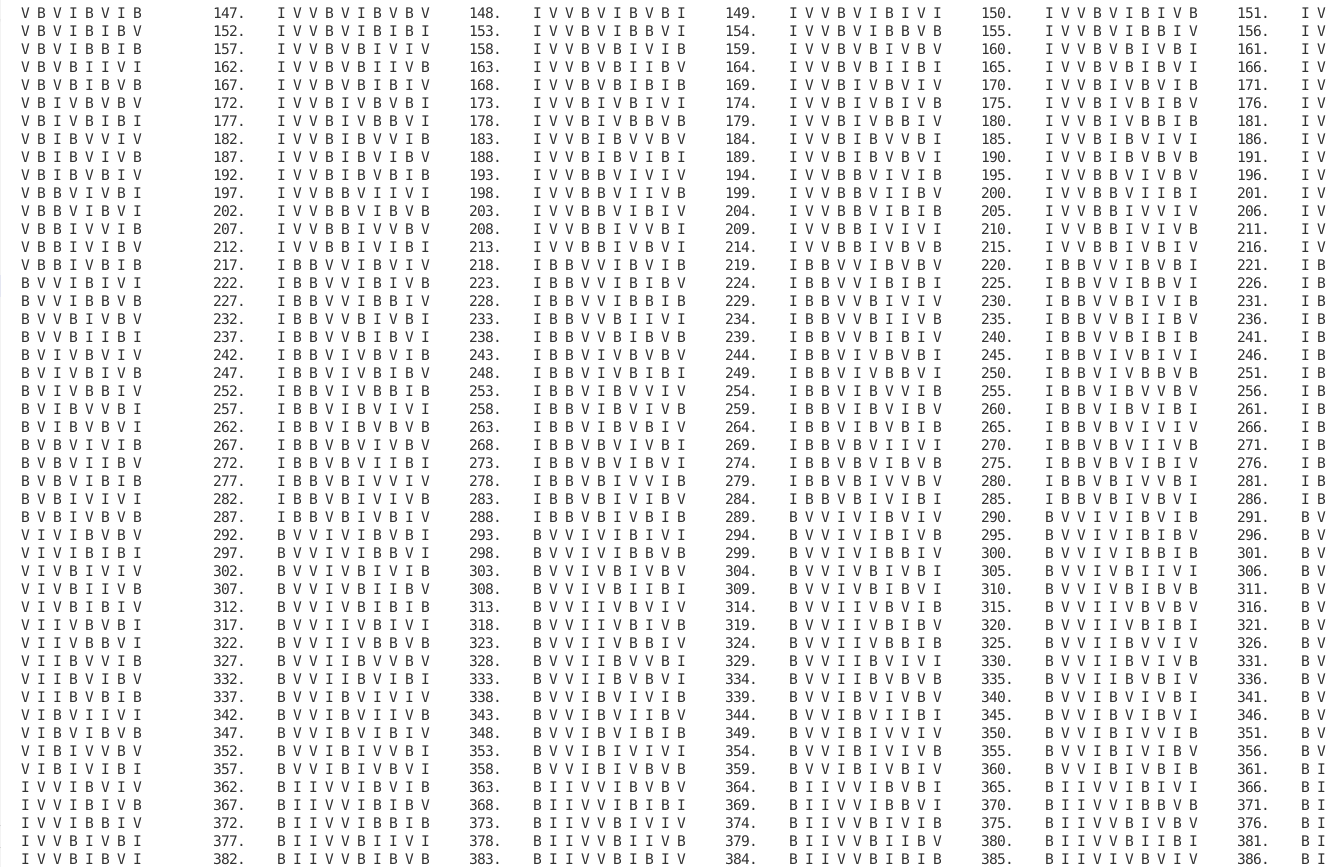
colour(x, nodes, colours, 0);

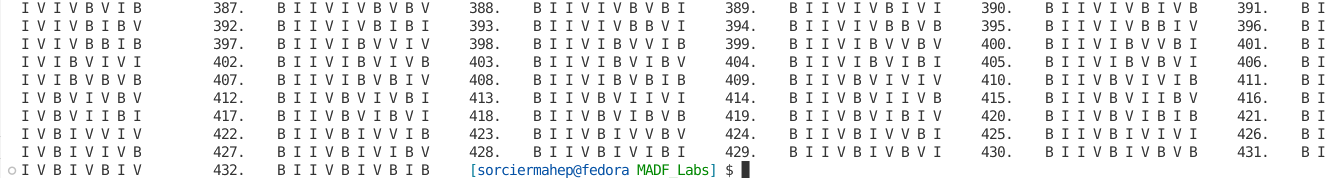
return 0;

}

**Output:**

1)

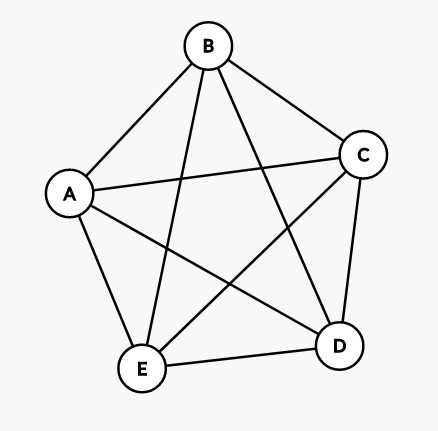




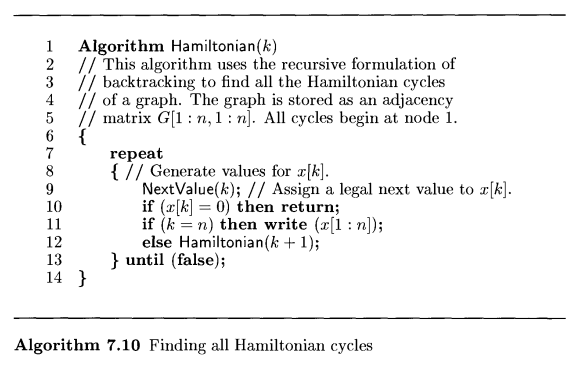
**d)Hamiltonian Cycle**

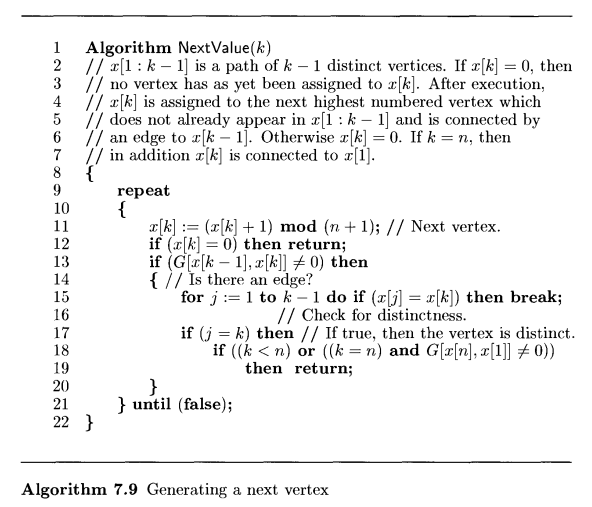
**Date:**

**Problem Statement:** Find the Hamiltonian cycles in the given graphs using the backtracking approach



**Algorithm:**





**Time Complexity:**

O(cN), where c is the common edge cost

**Space Complexity:**

O(N)

**Code:**

#include <stdio.h>

#define N 20

char G[N][N], count = 0;

void accept\_graph(int n)

{

int max\_edges = n \* n;

char origin, destin;

printf("Enter the edges of the graph.\n");

printf("Enter the edge;( ~ ~ ) to quit :\n");

for (int i = 0; i < max\_edges; i++)

{

scanf(" %c %c", &origin, &destin);

if ((origin == '~') && (destin == '~'))

break;

if (origin > 'A' + n || destin > 'A' + n || origin < 'A' || destin < 'A')

{

printf("Invalid edge.\n");

i--;

}

else

{

G[(int)origin - 64][(int)destin - 64] = 1;

G[(int)destin - 64][(int)origin - 64] = 1;

}

}

}

void show(char x[N], int nodes)

{

printf("%d.\t", ++count);

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

printf("%c - ", x[i + 1] + 64);

printf(" %c", x[1] + 64);

printf("\n");

}

void nextvalue(char x[N], int nodes, int k)

{

do

{

x[k] = (x[k] + 1) % (nodes + 1);

if (!x[k])

return;

if (G[x[k - 1]][x[k]] != 0)

{

int j;

for (j = 1; j <= k; j++)

if (x[j] == x[k])

break;

if (j == k)

if ((k < nodes) || ((k == nodes && G[x[nodes]][x[1]] != 0)))

return;

}

} while (1);

}

void Hamiltonian(char x[N], int nodes, int k)

{

do

{

nextvalue(x, nodes, k);

if (!x[k])

return;

if (k == nodes)

show(x, nodes);

else

Hamiltonian(x, nodes, k + 1);

} while (1);

}

int main()

{

int nodes;

char x[N];

printf("Enter the number of nodes: ");

scanf("%d", &nodes);

for (int i = 1; i <= nodes; i++)

for (int j = 1; j <= nodes; j++)

G[i][j] = 0;

accept\_graph(nodes);

x[1] = (int)'A' - 64;

for (int i = 2; i <= nodes; i++)

x[i] = 0;

printf("\nHamiltonian Cycles starting from node %d are : \n", x[1]);

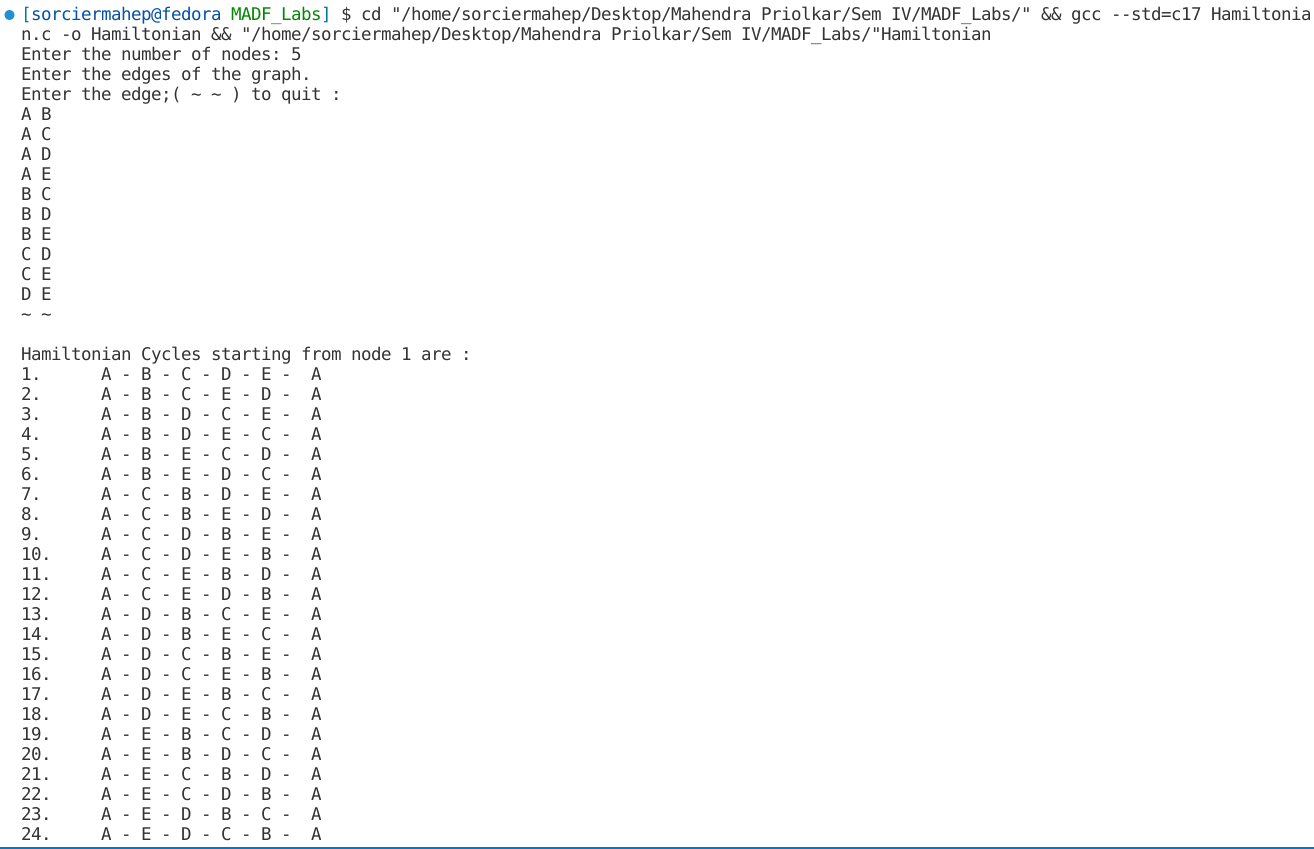
Hamiltonian(x, nodes, x[1] + 1);

return 0;

}

**Output:**

1)



**e)0/1 Knapsack**

**Date:**

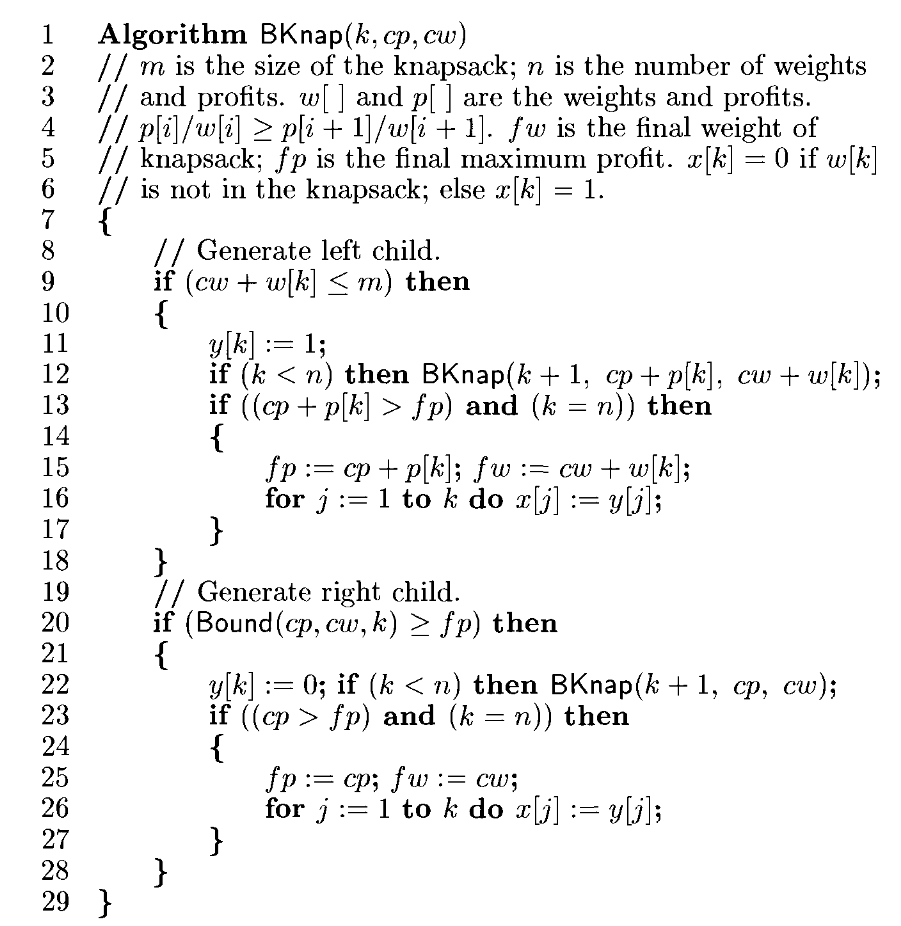
**Problem Statement:**

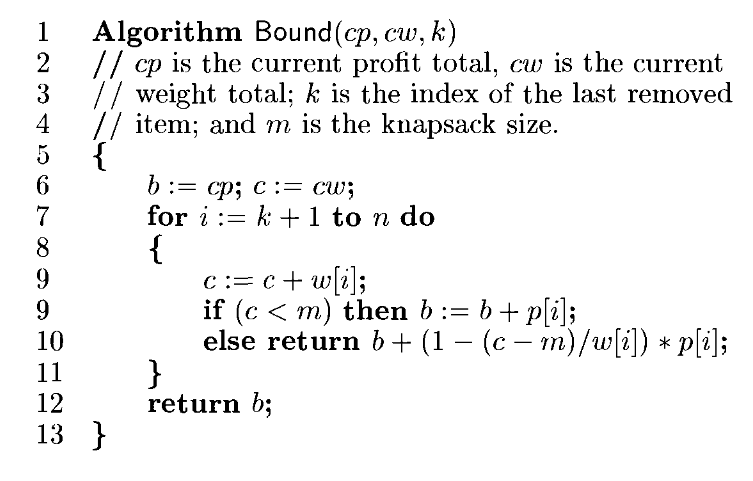
Solve the 0/1 knapsack problem for the knapsack instance n = 4, m = 8 using backtracking schema.

(w1 . . . w4) = (2,3,4,5)

(p1 . . . p7) = (3,5,6,10)

**Algorithm:**





**Time and Space Complexity:**

In the worst case, the algorithm has to consider 2^N combinations, where N is the number of items, resulting in an exponential time complexity of O(2^N).

The space complexity can be expressed as O(N), where N is the number of items. This is because the backtracking algorithm typically uses an array or a stack to store the current combination of items, and the maximum depth of recursion in the backtracking algorithm is bounded by the number of items.

**Code:**

#include <stdio.h>

#include <math.h>

#define MAX 100

struct knap

{

int p;

int w;

float rat;

};

int m, n, x[MAX], y[MAX], z[MAX], fp, fw;

int Bound(int k, int cp, int cw, struct knap K[])

{

int b = cp, c = cw;

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

{

c += K[i].w;

if (c < m)

b += K[i].p;

else

{

return b + (int)ceil((1 - (c - m) / (float)K[i].w) \* K[i].p);

}

}

return b;

}

void BKnap(int k, int cp, int cw, struct knap K[])

{

// Left child

if (cw + K[k].w <= m)

{

y[k] = 1;

if (k < n - 1)

{

printf("Left Child of K:%d CP:%d CW:%d\n", k + 1, cp, cw);

BKnap(k + 1, cp + K[k].p, cw + K[k].w, K);

}

if ((cp + K[k].p > fp) && (k == n - 1))

{

printf("Left Child of K:%d P:%d W:%d\n", k + 1, cp, cw);

fp = cp + K[k].p;

fw = cw + K[k].w;

printf("New cost is %d.\n", fp);

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

x[j] = y[j];

x[k] = 1;

printf("Array updated.\n");

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

printf("%d ", x[i]);

printf("\n");

}

}

// Right child

if (Bound(k, cp, cw, K) >= fp)

{

y[k] = 0;

if (k < n - 1)

{

printf("Right Child of K:%d P:%d W:%d.\n", k + 1, cp, cw);

BKnap(k + 1, cp, cw, K);

}

if ((cp > fp) && (k == n - 1))

{

printf("Right Child of K:%d P:%d W:%d\n", k + 1, cp, cw);

fp = cp;

fw = cw;

printf("New cost is %d.\n", fp);

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

x[j] = y[j];

x[k] = 0;

printf("Array updated.\n");

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

printf("%d ", x[i]);

printf("\n");

}

}

}

void sort(struct knap K[])

{

struct knap key;

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

{

key = K[i];

int j = i - 1;

while (j >= 0 && K[j].rat < key.rat)

{

K[j + 1] = K[j];

j = j - 1;

}

K[j + 1] = key;

}

}

int main()

{

printf("Enter number of knapsack values and size of knapsack.\n");

scanf("%d%d", &n, &m);

struct knap K[n + 1];

int p\_values[MAX], w\_values[MAX];

printf("Enter profits of knapsack.\n");

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

scanf("%d", &p\_values[i]);

printf("Enter weights of knapsack.\n");

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

scanf("%d", &w\_values[i]);

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

{

K[i].p = p\_values[i];

K[i].w = w\_values[i];

K[i].rat = (float)K[i].p / K[i].w;

}

sort(K);

fp = -1;

BKnap(0, 0, 0, K);

printf("The final array is:\n");

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

{

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

{

if (p\_values[i] == K[j].p && w\_values[i] == K[j].w)

{

if (x[j])

z[i] = 1;

else

z[i] = 0;

}

}

}

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

printf("%d ", z[i]);

printf("\nThe max profit is %d.\n", fp);

return 0;

}

**Output:**



**Conclusion:** Several Optimization problems were studied and implemented using the Backtracking Programming Algorithm. N-Queens, Sum of Subset, M-Coloring, Hamiltonian cycles, 0/1 Knapsack were implemented using the Backtracking Approach.