Experiment No. 4: Backtracking

Date:

Aim: Write a C program to implement the following program using Backtracking strategy

a. N-Queens
b. Sum Of Subsets
c. M-Coloring Graph
d. Hamiltonian Cycles
e. 0/1 Knapsack

THEORY:

- In the search for fundamental principles of algorithm design, backtracking represents one of the most general techniques. Many problems which deal with searching for a set of solutions or which ask for an optimal solution satisfying some constraints can be solved using the backtracking formulation.
- Backtracking is a systematic algorithmic approach used to solve problems by exploring all possible solutions through a depth-first search.
- The name backtrack was first coined by D. H. Lehmer in the 1950
- It is often used when the problem involves finding a solution among a large set of possibilities, and the brute force approach of checking all combinations is not feasible.
- The basic idea behind backtracking is to incrementally build a potential solution to the problem, while keeping track of whether the current partial solution can be extended to a valid complete solution. If it is determined that the current partial solution cannot be extended further to a valid solution, the algorithm backtracks and undoes the last decision, returning to a previous state and exploring alternative choices.
- Backtracking is widely used in various problem domains, such as combinatorial optimization, constraint satisfaction problems, puzzles, graph problems (e.g., graph coloring, Hamiltonian paths), and more. It provides an efficient way to explore the solution space by pruning branches that are guaranteed to lead to invalid solutions, reducing the overall search time.
- Advantages: It systematically explores all possible solutions, ensuring that a valid solution is found if one exists.
- Disadvantages: It can be computationally expensive and time-consuming for large problem spaces, as it may require exploring a significant number of branches and making many recursive calls.

a. N-Queens

Date:

The N-Queens problem involves placing N queens on an N×N chessboard such that no two queens threaten each other (i.e., no two queens share the same row, column, or diagonal). The goal is to find all possible solutions or a single valid configuration.

Problem Statement:

Write a c program to implement N-Queens Using Backtracking

Algorithm

```
Algorithm NQueens($k, n$)
// Using backtracking, this procedure prints all
// possible placements of $n$ queens on an $n \times n$
// chessboard so that they are nonattacking.
{
    for $i := 1$ to $n$ do
    {
        if Place($k, i$) then
        {
            $x[k] := i$;
            if ($k = n$) then write ($x[1 : n]$);
            else NQueens($k + 1, n$);
        }
    }
}
```

```
Algorithm Place($k, i$)
// Returns true if a queen can be placed in $k$th row and
// $i$th column. Otherwise it returns false. $x[\,]$ is a
// global array whose first $(k - 1)$ values have been set.
// Abs($r$) returns the absolute value of $r$.
{
    for $j := 1$ to $k - 1$ do
        if (($x[j] = i$) // Two in the same column
             or (Abs($x[j] - i$) = Abs($j - k$)))
                // or in the same diagonal
            then return false;
    return true;
}
```

Time Complexity & Space Complexity

The time complexity of the N-Queens problem using backtracking is O(N!) in the worst case, as the algorithm needs to explore all possible permutations of queen placements on the board.

The space complexity is O(N) for the recursive stack space, as the depth of recursion is limited to the number of queens placed on the board. Additionally, the space complexity includes the storage required for the chessboard representation and auxiliary data structures, which is O(N^2).

Code

```
#include<stdio.h>
#include <math.h>
int Board[16], count;
int place(int row, int column)
{
   int i;
   for(i=1; i<=row-1; ++i)
   {
      if(Board[i] == column){
         return 0;
      }
      else{
         if(abs(Board[i]-column)  ==  abs(i-
row))
         {
            return 0;
         }}}
   return 1;
}
void display(int n){
   int i,j;
   printf("\n\nSolution %d:\n\n", ++count);
   for(i=1; i<=n; ++i)
      printf("\t%d", i);
   for(i=1; i<=n; ++i) {
      printf("\n\n%d",i);
         for(j=1;j<=n; ++j) {
            if(Board[i] == j) {
               printf("\tQ");
            }
            else{
               printf("\t.");
            }
        }
}
   printf("\n\n");
}

void Queen(int row,int n)
{
   int col;
   for(col = 1; col<=n; ++col)
   {
      if(place(row, col))
      {
         Board[row]= col;
         if(row == n)
         {
            display(n);
         }
         else
         {
            Queen(row+1, n);
            }
        }
    }
}


int main()
{
    int n, i, j;
    void Queen(int row,int n);

    printf("\n\nEnter number of Queens:");
    scanf("%d",&n);
    Queen (1,n);
    return 0;
}
```

Output

```
Enter number of Queens:4



Solution 1:


        1       2       3       4


1       .       Q       .       .


2       .       .       .       Q


3       Q       .       .       .


4       .       .       Q       .
```

```
Solution 2:


        1       2       3       4


1       .       .       Q       .


2       Q       .       .       .


3       .       .       .       Q


4       .       Q       .       .
```

b. Sum Of Subsets

Date:

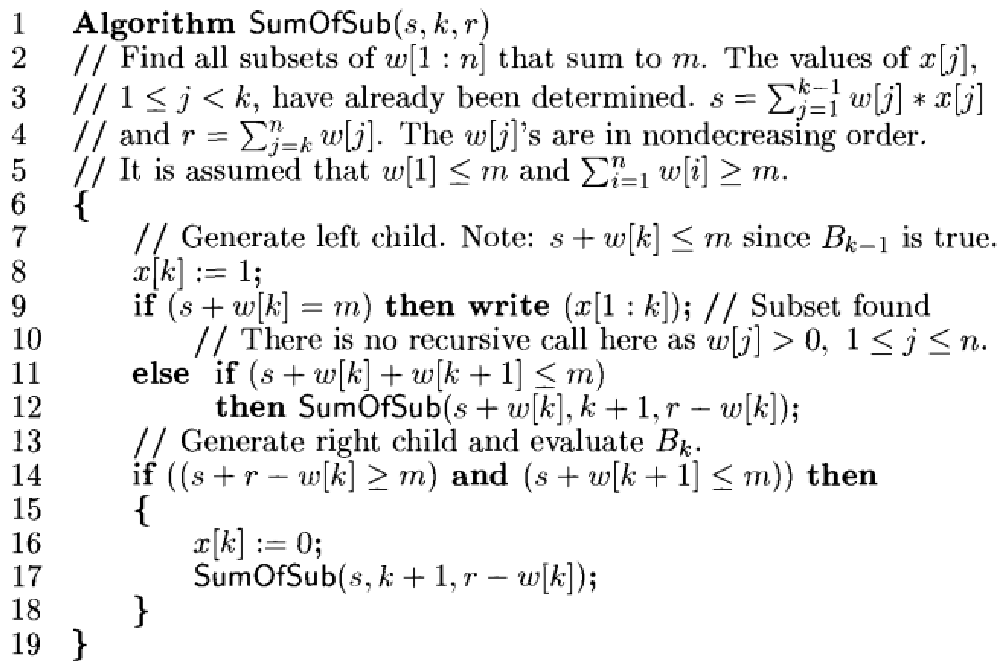
The **Sum of Subsets** problem involves finding all possible subsets of a given set whose elements sum up to a target value. The backtracking approach recursively explores different subsets, keeping track of the current sum and backtracking when the sum exceeds the target or all elements have been considered.

Problem Statement:

Write a c program to implement Sum of subsets using backtracking on the following :

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

Algorithm



```
Algorithm SumOfSub($s, k, r$)
// Find all subsets of $w[1:n]$ that sum to $m$. The values of $x[j]$,
// $1 \le j < k$, have already been determined. $s = \sum_{j=1}^{k-1} w[j] * x[j]$
// and $r = \sum_{j=k}^{n} w[j]$. The $w[j]$'s are in nondecreasing order.
// It is assumed that $w[1] \le m$ and $\sum_{i=1}^{n} w[i] \ge m$.
{
    // Generate left child. Note: $s + w[k] \le m$ since $B_{k-1}$ is true.
    $x[k] := 1$;
    if ($s + w[k] = m$) then write ($x[1:k]$); // Subset found
        // There is no recursive call here as $w[j] > 0,\ 1 \le j \le n$.
    else if ($s + w[k] + w[k+1] \le m$)
        then SumOfSub($s + w[k], k+1, r - w[k]$);
    // Generate right child and evaluate $B_k$.
    if (($s + r - w[k] \ge m$) and ($s + w[k+1] \le m$)) then
    {
        $x[k] := 0$;
        SumOfSub($s, k+1, r - w[k]$);
    }
}
```

Time Complexity and Space Complexity

The time complexity of the Sum of Subsets problem using backtracking is exponential, typically O(2^N), as the algorithm needs to explore all possible subsets of the given set.

The space complexity is O(N) for the recursive stack space, where N is the size of the input set. Additionally, the space complexity includes the storage required for the auxiliary data structures, which is proportional to the size of the input set.

Code

```
#include<stdio.h>

#define MAX 15

int x[6];

int w[MAX];

int w_sum;

int m;

int n;

void sumofsubset(int s,int k,int r);

void write(int x[MAX],int k);

int main()

{

   int s=0,k=0;

   int r;

   printf("Enter number of weights : ");

   scanf("%d",&n);

   printf("Enter Weights : ");

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

   {

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

      w_sum = w_sum+w[i];

   }

   r = w_sum;

   printf("Enter total sum : ");

   scanf("%d",&m);

   printf("\n\n");

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

   {

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

   }

   printf("\n");

   sumofsubset(s,k,r);

}

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

{

   //generate left child

   x[k]=1;

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

    write(x,k);

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

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


   //generate right child

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

   {

   x[k]=0;

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

   }

}
```

```
void write(int x[MAX],int k)
{
        if(k!=n)
        {
                for(int i=k+1;i<n;i++)
                x[i]=0;
        }
    for(int i=0;i<n;i++)
    {
       printf("%d\t",x[i]);
    }
    printf("\n");
}
```

Output:

```
Enter number of weights : 7

Enter Weights : 5 7 10 12 15 18 20

Enter total sum : 35



x1     x2     x3     x4     x5     x6     x7

1      0      1      0      0      0      1

1      0      0      1      0      1      0

0      1      1      0      0      1      0

0      0      0      0      1      0      1
```

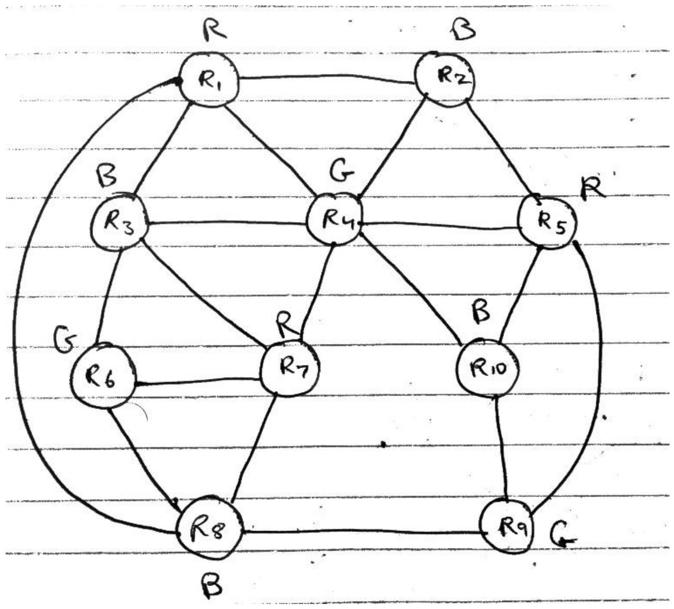
c. M-Coloring Graph

Date:

The **M Coloring Graph** problem involves coloring the vertices of a graph with M colors such that no two adjacent vertices share the same color. The backtracking approach recursively explores different color assignments for each vertex, checking for conflicts with adjacent vertices and backtracking when a conflict is detected.

Problem Statement:

Write a c program to implement m-coloring on following graph:



Algorithm

```
Algorithm mColoring($k$)
// This algorithm was formed using the recursive backtracking
// schema. The graph is represented by its boolean adjacency
// matrix $G[1:n, 1:n]$. All assignments of $1, 2, \ldots, m$ to the
// vertices of the graph such that adjacent vertices are
// assigned distinct integers are printed. $k$ is the index
// of the next vertex to color.
{
    repeat
    {// Generate all legal assignments for $x[k]$.
        NextValue($k$); // Assign to $x[k]$ a legal color.
        if ($x[k] = 0$) then return; // No new color possible
        if ($k = n$) then      // At most $m$ colors have been
                               // used to color the $n$ vertices.
            write ($x[1:n]$);
        else mColoring($k+1$);
    } until (false);
}
```

```
Algorithm NextValue(k)
// x[1], ..., x[k − 1] have been assigned integer values in
// the range [1, m] such that adjacent vertices have distinct
// integers. A value for x[k] is determined in the range
// [0, m]. x[k] is assigned the next highest numbered color
// while maintaining distinctness from the adjacent vertices
// of vertex k. If no such color exists, then x[k] is 0.
{
    repeat
    {
        x[k] := (x[k] + 1) mod (m + 1); // Next highest color.
        if (x[k] = 0) then return; // All colors have been used.
        for j := 1 to n do
        {   // Check if this color is
            // distinct from adjacent colors.
            if ((G[k, j] ≠ 0) and (x[k] = x[j]))
            // If (k, j) is and edge and if adj.
            // vertices have the same color.
                then break;
        }
        if (j = n + 1) then return; // New color found
    } until (false); // Otherwise try to find another color.
}
```

<u>Time Complexity and Space Complexity</u>

The time complexity of the M Coloring Graph problem using backtracking is typically O(M^N), where N is the number of vertices in the graph. This is because for each vertex, there are M possible color choices, resulting in a total of M^N combinations to explore.

The space complexity is O(N) for the recursive stack space, as the depth of recursion is limited to the number of vertices in the graph.

<u>Code</u>

```
#include <stdio.h>
#include <stdbool.h>
#define MAX_N 100
bool G[MAX_N][MAX_N];
int m, n, x[MAX_N];
void NextValue(int k){
   do{
        x[k] = (x[k] + 1) % (m + 1);
        if (x[k] == 0)
            return;
        int j;
        for (j = 1; j <= n; j++){
            if (G[k][j] && (x[k] == x[j]))
                break;}
        if (j == n + 1)
            return;
    } while (true);}
void mColoring(int k){
    do{
        NextValue(k);
        if (x[k] == 0)
            return;
        if (k == n){
            printf("Solution: ");
            for (int i = 1; i <= n; i++)
                printf("%d ", x[i]);
            printf("\n");
        }
        else
            mColoring(k + 1);
    } while (true);}
int main(){
    printf("Enter the number of vertices : ", MAX_N);
    scanf("%d", &n);
    printf("Enter the adjacency matrix:\n");
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            scanf("%d", &G[i][j]);
    printf("Enter the number of colors: ");
    scanf("%d", &m);
    for (int i = 1; i <= n; i++)
        x[i] = 0;
    mColoring(1);
    return 0;}
```

Output:

```
Enter the number of vertices : 10
Enter the adjacency matrix:
0 1 1 1 0 0 0 1 0 0
1 0 0 1 1 0 0 0 0 0
1 0 0 1 0 1 1 1 0 0
1 1 1 0 1 0 1 0 0 1
0 1 0 1 0 0 0 0 1 1
0 0 1 0 0 0 1 1 0 0
0 0 1 1 0 1 0 1 0 0
1 0 0 0 0 1 1 0 1 0
0 0 0 0 1 0 0 1 0 1
0 0 0 1 1 0 0 0 1 0
Enter the number of colors: 3
Solution: 1 2 2 3 1 3 1 2 3 2
Solution: 1 3 3 2 1 2 1 3 2 3
Solution: 2 1 1 3 2 3 2 1 3 1
Solution: 2 3 3 1 2 1 2 3 1 3
Solution: 3 1 1 2 3 2 3 1 2 1
Solution: 3 2 2 1 3 1 3 2 1 2
```

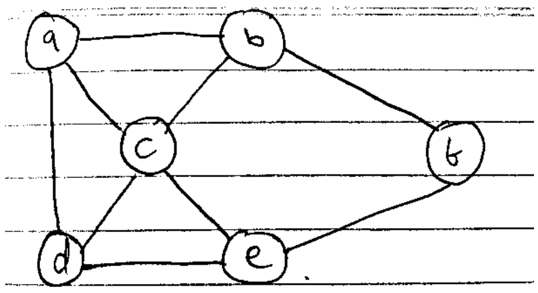
d. Hamiltonian Cycles

Date:

The **Hamiltonian Cycle** problem involves finding a cycle in a graph that visits every vertex exactly once. The backtracking approach recursively explores different paths in the graph, backtracking when a dead end is reached or when all vertices have been visited, checking for the existence of a Hamiltonian cycle.

Problem Statement:

Write a c program to implement Hamiltonian cycle using backtracking on the following graph :



Algorithm

```
Algorithm NextValue(k)
// x[1 : k − 1] is a path of k − 1 distinct vertices. If x[k] = 0, then
// no vertex has as yet been assigned to x[k]. After execution,
// x[k] is assigned to the next highest numbered vertex which
// does not already appear in x[1 : k − 1] and is connected by
// an edge to x[k − 1]. Otherwise x[k] = 0. If k = n, then
// in addition x[k] is connected to x[1].
{
    repeat
    {
        x[k] := (x[k] + 1) mod (n + 1); // Next vertex.
        if (x[k] = 0) then return;
        if (G[x[k − 1], x[k]] ≠ 0) then
        { // Is there an edge?
            for j := 1 to k − 1 do if (x[j] = x[k]) then break;
                          // Check for distinctness.
            if (j = k) then // If true, then the vertex is distinct.
                if ((k < n) or ((k = n) and G[x[n], x[1]] ≠ 0))
                    then return;
        }
    } until (false);
}
```

```
Algorithm Hamiltonian(k)
// This algorithm uses the recursive formulation of
// backtracking to find all the Hamiltonian cycles
// of a graph. The graph is stored as an adjacency
// matrix G[1 : n, 1 : n]. All cycles begin at node 1.
{
    repeat
    { // Generate values for x[k].
        NextValue(k); // Assign a legal next value to x[k].
        if (x[k] = 0) then return;
        if (k = n) then write (x[1 : n]);
        else Hamiltonian(k + 1);
    } until (false);
}
```

Time complexity and Space complexity

The time complexity of the Hamiltonian Cycle problem using backtracking is exponential, typically O(N!), where N is the number of vertices in the graph.

The space complexity is O(N) for the recursive stack space.

Code

```
#include <stdio.h>
#define MAX_NODES 20
int graph[MAX_NODES][MAX_NODES];
int count = 0;
void print(int x[MAX_NODES], int n)
{
printf("%d)\t", ++count);
for (int i = 1; i <= n; i++)
printf("%d ", x[i]);
printf("%d\n",x[1]);
}
void next_value(int x[MAX_NODES], int n, int k){
do {
x[k] = (x[k] + 1) % (n + 1);
if (!x[k]) return;
if (graph[x[k-1]][x[k]] != 0)
{
int j;
for (j = 1; j < k; j++)
if (x[j] == x[k]) break;
if (j == k && ((k < n) || (k == n && graph[x[n]][x[1]]))) return;
}
} while (1);
}
void hamiltonian(int x[MAX_NODES], int n, int k){
```

```
do{

next_value(x, n, k);

if (!x[k])

return;

if (k == n)

print(x, n);

else

hamiltonian(x, n, k + 1);

} while (1);

}

void create_graph(int n)

{

int max = n * n;

int origin, destination;

printf("Enter the edges of the graph (-1 -1 to
quit):\n");

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

{

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

if ((origin == -1) && (destination == -1))

break;

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

{

printf("Edge is Invalid\n");

i--;

}

else

{

graph[origin][destination] = 1;

graph[destination][origin] = 1;

}}}


int main()

{

int n, x[MAX_NODES];

printf("Enter the no of nodes : ");

scanf("%d", &n);


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

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

graph[i][j] = 0;


create_graph(n);

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

x[i] = 0;


printf("\nHAMILTONIAN CYCLES\n");


x[1] = 1;

hamiltonian(x,n,2);

if (count == 0)

printf("\nNo Hamiltonian cycles found\n");

else

printf("\nFound %d Hamiltonian cycles\n",
count);

return 0;
```

Output

```
Enter the no of nodes : 6

Enter the edges of the graph (-1 -1 to quit):

1 2

1 3

1 4

2 3

2 6

3 4

3 5

4 5

5 6

-1 -1


HAMILTONIAN CYCLES

1)      1 2 6 5 3 4 1

2)      1 2 6 5 4 3 1

3)      1 3 2 6 5 4 1

4)      1 3 4 5 6 2 1

5)      1 4 3 5 6 2 1

6)      1 4 5 6 2 3 1


Found 6 Hamiltonian cycles
```

e. 0/1 Knapsack

Date:

The **0/1 Knapsack** problem involves selecting a subset of items with maximum total value, given a weight constraint. The backtracking approach recursively explores different combinations of items, considering whether to include or exclude each item based on weight constraints, and backtracks when all items have been considered or the weight constraint is violated.

Problem Statement:

To implement 0/1 Knapsack Using Backtracking :

p = (30, 28, 20, 24)   m = 12

w = (5, 7, 4, 2)   n = 4

Algorithm

```
Algorithm BKnap($k, cp, cw$)
// $m$ is the size of the knapsack; $n$ is the number of weights
// and profits. $w[\,]$ and $p[\,]$ are the weights and profits.
// $p[i]/w[i] \geq p[i+1]/w[i+1]$. $fw$ is the final weight of
// knapsack; $fp$ is the final maximum profit. $x[k] = 0$ if $w[k]$
// is not in the knapsack; else $x[k] = 1$.
{
    // Generate left child.
    if ($cw + w[k] \leq m$) then
    {
        $y[k] := 1$;
        if ($k < n$) then BKnap($k + 1,\ cp + p[k],\ cw + w[k]$);
        if (($cp + p[k] > fp$) and ($k = n$)) then
        {
            $fp := cp + p[k]$; $fw := cw + w[k]$;
            for $j := 1$ to $k$ do $x[j] := y[j]$;
        }
    }
    // Generate right child.
    if (Bound($cp, cw, k$) $\geq fp$) then
    {
        $y[k] := 0$; if ($k < n$) then BKnap($k + 1,\ cp,\ cw$);
        if (($cp > fp$) and ($k = n$)) then
        {
            $fp := cp$; $fw := cw$;
            for $j := 1$ to $k$ do $x[j] := y[j]$;
        }
    }
}
```

Time complexity and Space complexity

The time complexity of the 0/1 Knapsack problem using backtracking is typically exponential, O(2^N), where N is the number of items. This is because the algorithm explores all possible combinations of including or excluding each item.

The space complexity is O(N) for the recursive stack space, as the depth of recursion is limited to the number of items.

Code

```
#include <stdio.h>

#define MAX_ITEMS 100

int maxProfit = 0;

int knapsackWeight = 0;

int finalSolution[MAX_ITEMS];

int weights[MAX_ITEMS];

int profits[MAX_ITEMS];

int bound(int currentProfit, int
currentWeight, int lastIndex, int
knapsackSize) {

int b = currentProfit;

int c = currentWeight;

for (int i = lastIndex + 1; i <
MAX_ITEMS; i++) {

c += weights[i];

if (c < knapsackSize) {

b += profits[i];

} else {

return b + ((1 - (c - knapsackSize) /
weights[i]) * profits[i]);

}}

return b;

}

void knapsack(int currentIndex, int
currentProfit, int currentWeight, int
knapsackSize) {

if (currentWeight <= knapsackSize &&
currentProfit > maxProfit) {

maxProfit = currentProfit;

knapsackWeight = currentWeight;

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

finalSolution[i] = 1;

}}

if (bound(currentProfit, currentWeight,
currentIndex, knapsackSize) > maxProfit)
{

finalSolution[currentIndex] = 1;

knapsack(currentIndex + 1, currentProfit +
profits[currentIndex], currentWeight +
weights[currentIndex], knapsackSize);

}

if (bound(currentProfit, currentWeight,
currentIndex, knapsackSize) > maxProfit)
{

finalSolution[currentIndex] = 0;

knapsack(currentIndex + 1, currentProfit,
currentWeight, knapsackSize);

}}

int main() {
```

```
int n, knapsackSize;

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

scanf("%d", &n);

printf("Enter the knapsack size: ");

scanf("%d", &knapsackSize);

printf("Enter the weights and profits of
each item:\n");

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

printf("Weight[%d]: ", i);

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

printf("Profit[%d]: ", i);

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

}

knapsack(0, 0, 0, knapsackSize);

printf("Maximum Profit: %d\n",
maxProfit);

printf("Knapsack Weight: %d\n",
knapsackWeight);

return 0;

}
```

Output

```
Enter the number of items: 4

Enter the knapsack size: 12

Enter the weights and profits of each item:

item[0]: 5 30

item[1]: 7 28

item[2]: 4 20

item[3]: 2 24

Maximum Profit: 58

Knapsack Weight: 12
```

CONCLUSION:

Backtracking was studied. The programs for (a) N-Queens , (b) Sum of subset (c) m-coloring graph (d) Hamiltonian cycle, and (e) 0/1 Knapsack using backtracking were studied and implemented successfully