EXPERIMENT NO – 4 **DATE –**

BACKTRACKING

7.1 The General Method

Backtracking is a powerful algorithmic technique used to solve problems that require searching for a set of solutions or an optimal solution among various feasible solutions. It is particularly effective for problems that involve constraint satisfaction, such as combinatorial optimization problems, puzzles, and decision-making tasks.

Origin and Development

- The term "backtrack" was first introduced by D. H. Lehmer in the 1950s.
- Further algorithmic studies and applications were contributed by R. J. Walker in 1960.
- S. Golomb and L. Baumert provided a comprehensive description and various applications in the same era.

Definition of Backtracking

Backtracking systematically searches for a solution to a problem by incrementally building a solution vector, and abandoning it ("backtracking") as soon as it is determined that the current partial solution cannot be extended to a valid complete solution.

The basic concept involves representing a potential solution as an n-tuple (x1, x2, ..., xn), where each xi is selected from a finite set Si. The objective is to find a vector that either maximizes or minimizes a given criterion function P(x1, x2, ..., xn) or satisfies a specific set of constraints.

Problem Formulation

- **Input:** A set of n-tuple values, where each xi is chosen from a set Si.
- **Objective:** Determine the tuple(s) that satisfy the given criterion function P.
- **Output:** A set of n-tuples that meet the specified constraints.

Constraints in Backtracking

Backtracking involves two types of constraints:

1. **Explicit Constraints:**
    - These are restrictions on the values of xi that are based on the problem definition. Examples include:
        - $xi \geq 0$ or $xi \in \{0, 1\}$
        - $li \leq xi \leq ui$
    - Explicit constraints define the solution space for the problem.
2. **Implicit Constraints:**

- These determine the relationship between elements in the solution space and are dependent on the criterion function. They specify how the elements in a tuple relate to each other to form a valid solution.

State Space Tree

- The state space tree is a conceptual representation of all potential solutions.
- Nodes in the tree represent partial solutions.
- Leaf nodes represent complete solutions.
- The objective is to traverse the tree systematically, evaluating each node based on the criterion function P.

Algorithms

- Backtracking algorithms can be implemented using recursive or iterative approaches.
- Both approaches involve systematically generating and evaluating potential solutions, using constraint functions to eliminate invalid paths early.

Algorithm 7.1 (Recursive Backtracking):

```
Algorithm Backtrack(k)
// This schema describes the backtracking process using
// recursion. On entering, the first k - 1 values
// x[1], x[2], ..., x[k - 1] of the solution vector
// x[1:n] have been assigned. x[ ] and n are global.
{
  for (each x[k] ∈ T(x[1], ..., x[k - 1]) do
  {
    if (B_k(x[1], x[2], ..., x[k]) ≠ 0) then
    {
      if (x[1], x[2], ..., x[k] is a path to an answer node)
       then write (x[1:k]);
      if (k < n) then Backtrack(k + 1);
    } }}
```

Algorithm 7.2 (Iterative Backtracking):

```
Algorithm Backtrack(n)
// This schema describes the backtracking process.
// All solutions are generated in x[1:n] and printed
// as soon as they are determined.
k := 1;
while (k ≠ 0) do
{  if (there remains an untried x[k] ∈ T(x[1], x[2], ..., x[k-1]) and B_k(x[1], ..., x[k]) is true) then
  {  if (x[1], ..., x[k] is a path to an answer node)
     then write (x[1:k]);
     k := k + 1; // Consider the next set.  }
  else k := k - 1; // Backtrack to the previous set.}
```

Backtracking is a versatile and systematic method for exploring potential solutions to complex problems involving constraints. By using state space trees and constraint checking, it reduces the search space and eliminates infeasible paths early in the search process.

DATE –

N QUEENS PROBLEM

AIM- Write a C program to solve n Queens problem using Backtracking

Problem Statement – Consider a N queens problem ,we have to solve and determine the number of solutions possible for that problem .

OUTPUT - A solution is displayed and total number of solutions are counted for each n .

ALGORITHM

1 Algorithm Place(k, i)

```
// Returns true if a queen can be placed in kth row and
// ith 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;
}
```

Recurrence Relation:

There is no true recurrence

Time Complexity:

I] **Best Case:** O(1)

- If the conflict is found in the very first comparison (e.g., same column or diagonal), the function returns early.

II] **Average Case:** O(k/2) ≈ O(k)

- On average, about half the previous rows are checked before a conflict is found or it is determined safe.

III] **Worst Case:** O(k)

- When no conflict is found, the loop runs through all k - 1 previous rows to confirm safety.

Space Complexity:

I] **Best / Average / Worst Case:** O(1)

- The function uses only constant extra space (loop variables and comparison logic).
- Global array x[] is accessed but not modified, and it's shared across calls.

II]Algorithm NQueens(k,n)

```
// Using backtracking, this procedure prints all
// possible placements of n queens on an n × 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);
      }
   }
}
```

Recurrence Relation:

Let T(k) be the time to place a queen in the kth row.

$T(k)= n \cdot T(k+1)+O(k)$

- For each row k, we try n columns.
- For each column, we call Place(k, i) which is O(k).
- Recursively goes deeper with T(k+1).
-

Time Complexity:

I] **Best Case:** O(n)

- If n = 1, or very early pruning happens via Place(), it terminates quickly. Only a few calls are made.

II] **Average Case:** Between O(n!) and O(n^n)

- Depends on how often conflicts are detected early and how much of the tree is pruned by Place(k, i).

III] **Worst Case:** O(n!)

- In the worst case (with no early pruning), all n! permutations of queen placements are explored.
- For each placement, Place() takes up to O(n) time → total cost up to O(n! * n).

Space Complexity:

I] **Best / Average / Worst Case:** O(n)

- One global array x[1..n] is used to store column positions of queens.
- Recursive call stack depth is at most n.

PROGRAM –

```
#include <stdio.h>
#include <math.h>
#include <time.h>

#define MAX_N 12
int x[MAX_N + 1];
int solution_count = 0;
int first_solution_shown = 0;

void displayBoard(int n) {
    printf("\n");
    for (int i = 1; i <= n; i++) {
        printf("|");
        for (int j = 1; j <= n; j++) {
            if (x[i] == j)
                printf(" Q ");
            else
                printf(" . ");
        }
        printf("|\n");
    }
    printf("\n");
}

int Place(int k, int i) {
    for (int j = 1; j < k; j++) {
        if (x[j] == i || abs(x[j] - i) == abs(j - k))
            return 0;
    }
    return 1;
}

void NQueens(int k, int n) {
    for (int i = 1; i <= n; i++) {
        if (Place(k, i)) {
            x[k] = i;
            if (k == n) {
                solution_count++;
                if (!first_solution_shown) {
                    displayBoard(n);
                    first_solution_shown = 1;
                }
            } else {
                NQueens(k + 1, n);
            }
        }
    }
}

void solveNQueens(int n) {
    if (n < 4 || n > MAX_N) {
        printf("Please enter a value between 4 and %d.\n", MAX_N);
        return;
    }

    solution_count = 0;
    first_solution_shown = 0;

    printf("\n%d-Queens Problem:\n", n);
    printf("One solution:");

    // Start timing
    clock_t start = clock();

    NQueens(1, n);

    // End timing
    clock_t end = clock();
    double time_taken = ((double)(end - start) / CLOCKS_PER_SEC) * 1000000; // Convert to microseconds

    printf("Total number of solutions for %d-Queens: %d\n", n, solution_count);
    printf("Time taken: %.2f microseconds\n", time_taken);
    printf("--------------------------------------------\n");
}

void displayMenu() {
    printf("**********************************************************\n");
    printf(" Roll number: 23B-CO-010\n");
    printf(" PR Number - 202311390\n");
    printf("**********************************************************\n\n");
    printf("\nN-Queens Problem Solver\n");
    printf("1. Solve for a specific N (4-%d)\n", MAX_N);
    printf("2. Solve for all N from 4 to %d\n", MAX_N);
    printf("3. Exit\n");
    printf("Enter your choice: ");
}

int main() {
    int choice, n;
```

```
    do {
        displayMenu();
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter the value of N (4-%d): ", MAX_N);
                scanf("%d", &n);
                solveNQueens(n);
                break;
            case 2:
                printf("Solving for all N from 4 to %d:\n", MAX_N);
                for (int i = 4; i <= MAX_N; i++) {
                    solveNQueens(i);
                }
                break;
            case 3:
                printf("Exiting program.\n");
                break;
            default:
                printf("Invalid choice. Please try again.\n");
        }
    } while (choice != 3);

    return 0;
}
```

OUTPUT -

```
******************************************************
 Roll number: 23B-CO-010
 PR Number - 202311390
******************************************************


N-Queens Problem Solver
1. Solve for a specific N (4-12)
2. Solve for all N from 4 to 12
3. Exit
Enter your choice: 2
Solving for all N from 4 to 12:

4-Queens Problem:
One solution:
| . Q . . |
| . . . Q |
| Q . . . |
| . . Q . |

Total number of solutions for 4-Queens: 2
Time taken: 1000.00 microseconds
----------------------------------------

5-Queens Problem:
One solution:
| Q . . . . |
| . . Q . . |
| . . . . Q |
| . Q . . . |
| . . . Q . |

Total number of solutions for 5-Queens: 10
Time taken: 3000.00 microseconds
----------------------------------------

6-Queens Problem:
One solution:
| . Q . . . . |
| . . . Q . . |
| . . . . . Q |
| Q . . . . . |
| . . Q . . . |
| . . . . Q . |

Total number of solutions for 6-Queens: 4
Time taken: 5000.00 microseconds
----------------------------------------

7-Queens Problem:
One solution:
| Q . . . . . . |
| . . Q . . . . |
| . . . . Q . . |
| . . . . . . Q |
| . Q . . . . . |
| . . . Q . . . |
| . . . . . Q . |

Total number of solutions for 7-Queens: 40
Time taken: 4000.00 microseconds
----------------------------------------

8-Queens Problem:
One solution:
| Q . . . . . . . |
| . . . . Q . . . |
| . . . . . . . Q |
| . . . . . Q . . |
| . . Q . . . . . |
| . . . . . . Q . |
| . Q . . . . . . |
| . . . Q . . . . |

Total number of solutions for 8-Queens: 92
Time taken: 10000.00 microseconds
----------------------------------------
```

```
9-Queens Problem:
One solution:
| Q  .  .  .  .  .  .  .  . |
| .  .  Q  .  .  .  .  .  . |
| .  .  .  .  .  Q  .  .  . |
| .  .  .  .  .  .  .  Q  . |
| .  Q  .  .  .  .  .  .  . |
| .  .  .  Q  .  .  .  .  . |
| .  .  .  .  .  .  .  .  Q |
| .  .  .  .  .  .  Q  .  . |
| .  .  .  .  Q  .  .  .  . |

Total number of solutions for 9-Queens: 352
Time taken: 16000.00 microseconds
----------------------------------------

10-Queens Problem:
One solution:
| Q  .  .  .  .  .  .  .  .  . |
| .  .  Q  .  .  .  .  .  .  . |
| .  .  .  .  .  Q  .  .  .  . |
| .  .  .  .  .  .  .  Q  .  . |
| .  .  .  .  .  .  .  .  .  Q |
| .  .  .  .  Q  .  .  .  .  . |
| .  .  .  .  .  .  .  .  Q  . |
| .  Q  .  .  .  .  .  .  .  . |
| .  .  .  Q  .  .  .  .  .  . |
| .  .  .  .  .  .  Q  .  .  . |

Total number of solutions for 10-Queens: 724
Time taken: 21000.00 microseconds
----------------------------------------

11-Queens Problem:
One solution:
| Q  .  .  .  .  .  .  .  .  .  . |
| .  .  Q  .  .  .  .  .  .  .  . |
| .  .  .  .  Q  .  .  .  .  .  . |
| .  .  .  .  .  .  Q  .  .  .  . |
| .  .  .  .  .  .  .  .  Q  .  . |
| .  .  .  .  .  .  .  .  .  .  Q |
| .  Q  .  .  .  .  .  .  .  .  . |
| .  .  .  Q  .  .  .  .  .  .  . |
| .  .  .  .  .  Q  .  .  .  .  . |
| .  .  .  .  .  .  .  Q  .  .  . |
| .  .  .  .  .  .  .  .  Q  .  . |

Total number of solutions for 11-Queens: 2680
Time taken: 51000.00 microseconds
----------------------------------------

12-Queens Problem:
One solution:
| Q  .  .  .  .  .  .  .  .  .  .  . |
| .  .  Q  .  .  .  .  .  .  .  .  . |
| .  .  .  .  Q  .  .  .  .  .  .  . |
| .  .  .  .  .  .  .  Q  .  .  .  . |
| .  .  .  .  .  .  .  .  .  Q  .  . |
| .  .  .  .  .  .  .  .  .  .  .  Q |
| .  .  .  .  .  Q  .  .  .  .  .  . |
| .  .  .  .  .  .  .  .  .  .  Q  . |
| .  Q  .  .  .  .  .  .  .  .  .  . |
| .  .  .  .  .  .  Q  .  .  .  .  . |
| .  .  .  .  .  .  .  .  Q  .  .  . |
| .  .  .  Q  .  .  .  .  .  .  .  . |

Total number of solutions for 12-Queens: 14200
Time taken: 204000.00 microseconds
----------------------------------------
```

CONCLUSION – N -Queens problem was successfully solved using backtracking method .

DATE –

SUM OF SUBSETS

AIM- Write a C program to calculate to a ideal sum from a set of elements using sum of subsets algorithm

Problem Statement – Given a ideal sum m and a set s of elements,we have to include only those elements from the set whose sum is equal to ideal sum .

Input – The set is inputed an

A = { 1,2,3,7,18} m =28

OUTPUT - A solution is displayed and total number of solutions are counted for each n .

ALGORITHM

Algorithm SumOfSub(s,k,r)

// Find all subsets of w[1:n] that sum to m. The values of x[j],

// 1≤j<k, have already been determined. s=∑_{j=1}^{k-1}w[j]*x[j]

// and r=∑_{j=k}^{n}w[j]. The w[j]'s are in nondecreasing order.

// It is assumed that w[1]≤m and ∑_{i=1}^{n}w[i]≥m.

```
x[k]:=1;
if (s+w[k]=m) then write (x[1:k]);
else if (s+w[k]+w[k+1]≤m)
then SumOfSub(s+w[k],k+1,r-w[k]);
if ((s+r-w[k]≥m) and (s+w[k+1]≤m)) then
{
   x[k]:=0;
   SumOfSub(s,k+1,r-w[k]);
}
```

Recurrence Relation:

Let T(k) be the time taken at level k:

T(k)= 2T(k+1)+O(1)

- At most, two recursive calls are made at each level (x[k] = 1 and x[k] = 0) depending on conditions.
- Pruning conditions (s + w[k] > m, s + r - w[k] < m, etc.) reduce the number of actual calls

.

Time Complexity:

I] **Best Case:** O(n)

- Only one valid subset is found early; heavy pruning eliminates all further recursion.

II] **Average Case:** O(2^n)

- Not all branches are explored due to pruning; number of recursive calls is reduced significantly compared to brute-force subset generation.

III] **Worst Case:** O(2^n)

- In worst case (e.g., no pruning), the recursion explores all possible subsets (similar to power set generation).
-

Space Complexity:

I] **Best / Average / Worst Case:** O(n)

- One array x[1..n] used for storing inclusion/exclusion of elements.
- Recursive call stack depth is at most n.

PROGRAM –

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <time.h>
#include <windows.h>

#define MAX_SIZE 100

typedef struct Node {
   int s;
   int k;
   int r;
   int x[MAX_SIZE];
   char status;
   struct Node *lchild;
   struct Node *rchild;
} Node;

int w[MAX_SIZE];
int m;
int n;
int solutionCount = 0;
int x[MAX_SIZE] = {0};
Node* root = NULL;

void printVector() {
   printf("(");
   for (int i = 1; i <= n; i++) {
      printf("%d", x[i]);
      if (i < n) printf(",");
   }
   printf(")\n");
}

Node* SumOfSub(int s, int k, int r)
{
   Node* tmp =
(Node*)malloc(sizeof(Node));
   tmp->s = s;
   tmp->k = k;
   tmp->r = r;
   tmp->lchild = NULL;
   tmp->rchild = NULL;
   tmp->status = 'N';

   for (int i = 1; i <= n; i++) {
      tmp->x[i] = x[i];
   }

   if (s == m) {
      tmp->status = 'S';
      solutionCount++;
      printf("Node(s=%d, k=%d,
r=%d) Solution\n", s, k, r);
      printf("Vector: ");
      printVector();
      return tmp;
   }

   if (k > n) return tmp;

   x[k] = 1;
   if (s + w[k] <= m) {
      if (s + w[k] == m) {
         tmp->status = 'S';
         solutionCount++;
         printf("Node(s=%d, k=%d,
r=%d) Solution\n", s + w[k], k + 1,
r - w[k]);
         printf("Vector: ");
         printVector();

         tmp->lchild =
(Node*)malloc(sizeof(Node));
         tmp->lchild->s = s + w[k];
         tmp->lchild->k = k + 1;
         tmp->lchild->r = r - w[k];
         tmp->lchild->status = 'S';
         tmp->lchild->lchild = NULL;
         tmp->lchild->rchild = NULL;

         for (int i = 1; i <= n; i++) {
            tmp->lchild->x[i] = x[i];
         }
      }
      else if (s + w[k] + r - w[k] >= m
&& k < n && s + w[k] + w[k+1] <=
m) {
         tmp->lchild = SumOfSub(s +
w[k], k + 1, r - w[k]);
      } else {
         printf("Node(s=%d, k=%d,
r=%d) Bounded\n", s + w[k], k + 1,
r - w[k]);
      }
   } else {
      printf("Node(s=%d, k=%d,
r=%d) Bounded\n", s + w[k], k + 1,
r - w[k]);
   }

   x[k] = 0;
   if (s + r - w[k] >= m && k < n &&
s + w[k+1] <= m) {
      tmp->rchild = SumOfSub(s, k
+ 1, r - w[k]);
   } else if (k <= n) {
```

```
        printf("Node(s=%d, k=%d, r=%d) Bounded\n", s, k + 1, r - w[k]);
    }

    return tmp;
}

void print_solutions(Node *root) {
    if (root == NULL) return;

    if (root->status == 'S' && root->s == m) {
        printf("Solution Node(s=%d, k=%d, r=%d): ", root->s, root->k, root->r);
        printf("(");
        for (int i = 1; i <= n; i++) {
            printf("%d", root->x[i]);
            if (i < n) printf(",");
        }
        printf(")\n");
    }

    print_solutions(root->lchild);
    print_solutions(root->rchild);
}

int calculateTotal(int w[], int n) {
    int total = 0;
    for (int i = 1; i <= n; i++) {
        total += w[i];
    }
    return total;
}

long long getMicrotime() {
    LARGE_INTEGER frequency;
    LARGE_INTEGER start;

    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&start);

    return (start.QuadPart * 1000000) / frequency.QuadPart;
}

void inputData() {
    printf("\nEnter the number of elements in set: ");
    scanf("%d", &n);

    printf("Enter the target sum (m): ");
    scanf("%d", &m);

    printf("Enter %d elements: ", n);
    for (int i = 1; i <= n; i++) {
        scanf("%d", &w[i]);
        x[i] = 0;
    }

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n - i; j++) {
            if (w[j] > w[j + 1]) {
                int temp = w[j];
                w[j] = w[j + 1];
                w[j + 1] = temp;
            }
        }
    }
}

void solveSubsetSum() {
    if (n == 0) {
        printf("\nPlease input data first (Option 1).\n");
        return;
    }

    int total = calculateTotal(w, n);

    if (w[1] > m || total < m) {
        printf("No solution exists.\n");
        return;
    }

    solutionCount = 0;
    printf("\nLEAF NODES: \n");

    long long startTime = getMicrotime();
    root = SumOfSub(0, 1, total);
    long long endTime = getMicrotime();
    long long executionTime = endTime - startTime;

    printf("\nTime taken: %lldµs\n", executionTime);

    if (solutionCount > 0) {
        printf("\nAll Solutions:\n");
        print_solutions(root);
        printf("\nTotal number of solutions: %d\n", solutionCount);
    } else {
```

```
    printf("\nNo solutions found.\n");
  }
}

int main() {
  printf("**********************************************\n");
  printf(" Roll number: 23B-CO-010\n");
  printf(" PR Number - 202311390\n");
  printf("**********************************************\n\n");
  int choice;

}
  while (1) {
    printf("\n=== Sum of Subsets Menu ===\n");
    printf("1. Input data\n");
    printf("2. Find subsets with given sum\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch(choice) {
      case 1:
        inputData();
        break;

      case 2:
        solveSubsetSum();
        break;

      case 3:
        printf("\nExiting program. Goodbye!\n");
        exit(0);

      default:
        printf("\nInvalid choice. Please try again.\n");
    }
  }

  return 0;
```

INPUT :

```
**************************************************
 Roll number: 23B-CO-010
 PR Number - 202311390
**************************************************


=== Sum of Subsets Menu ===
1. Input data
2. Find subsets with given sum
3. Exit
Enter your choice: 1

Enter the number of elements in set: 5
Enter the target sum (m): 28
Enter 5 elements: 1
2
3
7
18
```

OUTPUT –

```
=== Sum of Subsets Menu ===
1. Input data
2. Find subsets with given sum
3. Exit
Enter your choice: 2

LEAF NODES:
Node(s=13, k=5, r=18) Bounded
Node(s=6, k=5, r=18) Bounded
Node(s=28, k=6, r=0) Solution
Vector: (1,1,0,1,1)
Node(s=10, k=6, r=0) Bounded
Node(s=3, k=5, r=18) Bounded
Node(s=11, k=5, r=18) Bounded
Node(s=4, k=5, r=18) Bounded
Node(s=1, k=4, r=25) Bounded
Node(s=12, k=5, r=18) Bounded
Node(s=5, k=5, r=18) Bounded
Node(s=2, k=4, r=25) Bounded
Node(s=28, k=6, r=0) Solution
Vector: (0,0,1,1,1)
Node(s=10, k=6, r=0) Bounded
Node(s=3, k=5, r=18) Bounded
Node(s=0, k=4, r=25) Bounded
```

```
All Solutions:
Solution Node(s=28, k=6, r=0): (1,1,0,1,1)
Solution Node(s=28, k=6, r=0): (0,0,1,1,1)

Total number of solutions: 2
```

TIME TAKEN

```
Time taken: 2643μs
```

CONCLUSION : Sum of subsets method was successfully implemented to calculate the ideal sum .

DATE –

GRAPH COLOURING

AIM- Write a C program to colour a graph with m colours using m colouring algorithm

Problem Statement – Given a graph with n vertices ,we have to colour the graph in such a way that no adjacent vertices have the same colour

Input – The adjacency matrix is inputed with the number of colours to be used

OUTPUT – The matrix is printed which contains the order in the vertices must be coloured.

ALGORITHM

```
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;
      for j := 1 to n do
      {
         if ((G[k,j] ≠ 0) and (x[k] = x[j]))
         then break;
      }
      if (j = n + 1) then return; // New color found
   } until (false); // Otherwise try to find another color.
}
```

Recurrence Relation

The nextvalue function does not use recursion

Time Complexity:

I] Best Case: O(n)

- The first color tried is valid (i.e., it is not used by any adjacent vertex).
- Loop executes once, and adjacency is checked over all n vertices.

II] Average Case: O(m × n)

- Tries multiple colors (up to m total), and for each color, checks adjacency against n vertices.
- On average, may require trying m/2 colors.

III] Worst Case: O(m × n)

- All m colors are tried, and for each, the algorithm checks against n adjacent vertices.
- If none are valid, sets x[k] := 0.

Space Complexity:

I] Best Case: O(1)

- Only constant space is used for variables like x[k], j, and loop counters.

II] Average Case: O(1)

- No additional data structures used per call.

III] Worst Case: O(1)

- Same as above: no recursion, no auxiliary arrays used inside the function.

II]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,...,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);
}
```

Recurrence Relation

Let:

- T(n) = total time to color n vertices
- m = number of colors available

Recurrence Relation:

$T(n) = m \times T(n - 1) + O(n)$

Because for each of the n vertices, we try up to m colors and recursively color the next vertex. The O(n) term comes from checking adjacent colors.

Time Complexity

I] **Best Case:**

Time Complexity: O(n × m)

➞ Only a valid coloring is quickly found, and the algorithm stops early without exploring all combinations.

II] **Average Case:**

Time Complexity: O(m^n)

➞ Each vertex has up to m choices. The recursive tree explores several branches but often prunes invalid ones due to backtracking.

III] **Worst Case:**

Time Complexity: O(m^n)

➞ All possible combinations of m colors for n vertices are explored when no valid solution or all solutions are required (e.g., in complete graphs).

Space Complexity

I] **Best Case:**

Space Complexity: $O(n^2)$

➞ Due to adjacency matrix ($O(n^2)$), and recursion stack up to depth n (O(n)), but the matrix dominates.

II] **Average Case:**

Space Complexity: $O(n^2)$

➞ Recursion depth is still O(n), and adjacency matrix remains the major space consumer.

III] **Worst Case:**

Space Complexity: $O(n^2)$

➞ In the worst scenario, recursion goes all the way to n, and adjacency matrix is always $O(n^2)$.

PROGRAM –

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <windows.h>

#define MAX 20

int n, m;
int x[MAX], G[MAX][MAX];
int solution_count = 0;
int nodeCount = 0;
int solutions[MAX][MAX];

void print_current_state(int k, const char *state) {
    printf("\n%s ", state);
    for (int i = 1; i <= k; i++) {
        printf("x%d=%d ", i, x[i]);
    }
    printf("\n\n");
}

void NextValue(int k) {
    int j;
    while (1) {
        x[k] = (x[k] + 1) % (m + 1);
        if (x[k] == 0) {
            return;
        }
        for (j = 1; j <= n; j++) {
            if (G[k][j] && x[k] == x[j]) {
                print_current_state(k, "Bounded");
                break;
            }
        }
        if (j == n + 1) {
            return;
        }
    }
}

void mColoring(int k) {
    while (1) {
        NextValue(k);
        if (x[k] == 0) {
            return;
        }
        if (k == n) {
            solution_count++;
            printf("Solution %d: ", solution_count);
            for (int i = 1; i <= n; i++) {
                printf("x%d=%d ", i, x[i]);
                solutions[solution_count - 1][i - 1] = x[i];
            }
            printf("\n");
        } else {
            mColoring(k + 1);
        }
    }
}

void printSolutionMatrix() {
    printf("\nSolution Matrix:\n");
    for (int i = 0; i < solution_count; i++) {
        printf("%c: x[1..%d] = {", 'A' + i, n);
        for (int j = 0; j < n; j++) {
            printf("%d", solutions[i][j]);
            if (j < n - 1) {
                printf(", ");
            }
        }
        printf("}\n");
    }
}

void inputGraph() {
    printf("Enter number of vertices (n, max 20): ");
    scanf("%d", &n);

    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            G[i][j] = 0;
        }
    }

    printf("Enter the adjacency matrix (1 for edge, 0 for no edge):\n");
    for (int i = 1; i <= n; i++) {
        for (int j = 1; j <= n; j++) {
            printf("Edge between %d and %d (1/0): ", i, j);
            scanf("%d", &G[i][j]);
        }
    }
}

int main() {
    printf("**********************************************************\n");
    printf(" Roll number: 23B-CO-010\n");
    printf(" PR Number - 202311390\n");
```

```
    printf("**********
**********
*****\n\n");

    int choice;



    do {

        printf("\n=== GRAPH
COLORING ALGORITHM ===\n");

        printf("1. Input Graph\n");

        printf("2. Set Number of
Colors\n");

        printf("3. Display All m-
Colorings (DFS Order)\n");

        printf("4. Exit\n");

        printf("Enter choice: ");

        scanf("%d", &choice);



        switch (choice) {

            case 1:

                inputGraph();

                break;

            case 2:

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

                scanf("%d", &m);

                break;

            case 3:

                if (n == 0 || m == 0) {

                    printf("Please input
graph and set number of colors
first.\n");

                } else {

                    for (int i = 1; i <= n; i++)
x[i] = 0;

                    solution_count = 0;



                    LARGE_INTEGER
frequency, start, end;

                    QueryPerformanceFreq
uency(&frequency);

                    QueryPerformanceCou
nter(&start);



                    printf("ALGORITHM
):\n");

                    mColoring(1);



                    QueryPerformanceCou
nter(&end);

                    double elapsedTime =
(double)(end.QuadPart -
start.QuadPart) * 1000000.0 /
frequency.QuadPart;



                    if (solution_count == 0)
{

                        printf("No valid
colorings found.\n");

                    }

                    if (solution_count == 0)
{

                        printf("No solutions
available..\n");

                    } else {

                        printSolutionMatrix()
;

                    }



                    printf("\nExecution
Time: %.2f microseconds\n",
elapsedTime);

                }

                break;

            case 4:

                printf("Exiting...\n");

                break;

            default:

                printf("Invalid choice. Try
again.\n");

        }

    } while (choice != 5);



    return 0;

}
```

INPUT –

```
**************************************************
 Roll number: 23B-CO-010
 PR Number - 202311390
**************************************************


=== GRAPH COLORING ALGORITHM ===
1. Input Graph
2. Set Number of Colors
3. Display All m-Colorings (DFS Order)
4. Exit
Enter choice: 1
Enter number of vertices (n, max 20): 6
Enter the adjacency matrix (1 for edge, 0 for no edge):
Edge between 1 and 1 (1/0): 0
Edge between 1 and 2 (1/0): 1
Edge between 1 and 3 (1/0): 1
Edge between 1 and 4 (1/0): 0
Edge between 1 and 5 (1/0): 0
Edge between 1 and 6 (1/0): 0
Edge between 2 and 1 (1/0): 1
Edge between 2 and 2 (1/0): 0
Edge between 2 and 3 (1/0): 0
Edge between 2 and 4 (1/0): 1
Edge between 2 and 5 (1/0): 1
Edge between 2 and 6 (1/0): 0
Edge between 3 and 1 (1/0): 1
Edge between 3 and 2 (1/0): 0
Edge between 3 and 3 (1/0): 0
Edge between 3 and 4 (1/0): 1
Edge between 3 and 5 (1/0): 0
Edge between 3 and 6 (1/0): 0
Edge between 4 and 1 (1/0): 0
Edge between 4 and 2 (1/0): 1
Edge between 4 and 3 (1/0): 1
Edge between 4 and 4 (1/0): 0
Edge between 4 and 5 (1/0): 0
Edge between 4 and 6 (1/0): 1
Edge between 5 and 1 (1/0): 0
Edge between 5 and 2 (1/0): 1
Edge between 5 and 3 (1/0): 0
Edge between 5 and 4 (1/0): 0
Edge between 5 and 5 (1/0): 0
Edge between 5 and 6 (1/0): 1
Edge between 6 and 1 (1/0): 0
Edge between 6 and 2 (1/0): 0
Edge between 6 and 3 (1/0): 0
Edge between 6 and 4 (1/0): 1
Edge between 6 and 5 (1/0): 1
Edge between 6 and 6 (1/0): 0
```

```
=== GRAPH COLORING ALGORITHM ===
1. Input Graph
2. Set Number of Colors
3. Display All m-Colorings (DFS Order)
4. Exit
Enter choice: 2
Enter the number of colors (m): 2
```

OUTPUT –

```
=== GRAPH COLORING ALGORITHM ===
1. Input Graph
2. Set Number of Colors
3. Display All m-Colorings (DFS Order)
4. Exit
Enter choice: 2
Enter the number of colors (m): 2

=== GRAPH COLORING ALGORITHM ===
1. Input Graph
2. Set Number of Colors
3. Display All m-Colorings (DFS Order)
4. Exit
Enter choice: 3
ALGORITHM ):

Bounded x1=1 x2=1


Bounded x1=1 x2=2 x3=1


Bounded x1=1 x2=2 x3=2 x4=1 x5=1 x6=1

Solution 1: x1=1 x2=2 x3=2 x4=1 x5=1 x6=2

Bounded x1=1 x2=2 x3=2 x4=1 x5=2


Bounded x1=1 x2=2 x3=2 x4=2


Bounded x1=2 x2=1 x3=1 x4=1


Bounded x1=2 x2=1 x3=1 x4=2 x5=1

Solution 2: x1=2 x2=1 x3=1 x4=2 x5=2 x6=1

Bounded x1=2 x2=1 x3=1 x4=2 x5=2 x6=2


Bounded x1=2 x2=1 x3=2


Bounded x1=2 x2=2


Solution Matrix:
A: x[1..6] = {1, 2, 2, 1, 1, 2}
B: x[1..6] = {2, 1, 1, 2, 2, 1}
```

TIME TAKEN –

```
Execution Time: 9186.90 microseconds
```

CONCLUSION – The order of vertex colouring was successfully determined using m colouring algorithm.

DATE –

ATHARV GOVEKAR **23B-CO-010**

HAMILTONEAN CYCLE

AIM- Write a C program to determine hamiltonean cycle over a graph with n vertices using backtracking algorithm.

Problem Statement – Given a graph with n vertices ,we have to determine a path which forms a hamiltonean cycle over the graph.

Input – The adjacency matrix is inputed .

OUTPUT – The matrix is printed which contains the path that must be taken to form hamiltonean cycle over the graph.

ALGORITHM

I]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;
      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);  }
```

Recurrence Relation:

T(k) = O(n)

Because for each position k, we may iterate over up to n vertices and compare against all k-1 previous vertices for validity (distinctness + adjacency).

Time Complexity

I] **Best Case:**

Time Complexity: O(1)

A valid adjacent and distinct vertex is found in the first or second attempt.

II] **Average Case:**

Time Complexity: O(n)

On average, we may need to check multiple vertices (up to n) before finding a valid one.

III] **Worst Case:**

Time Complexity: O(n)

We scan through all n vertices, and possibly compare each with k-1 previous elements.

Space Complexity

I] **Best Case:**

Space Complexity: O(1)

Only constant space is used for variables like x[k], loop counters, and temp values.

II] **Average Case:**

Space Complexity: O(1)

Still constant space, as we don't store extra data structures apart from the array and adjacency matrix.

III] **Worst Case:**

Space Complexity: O(1)

Space does not grow with n; space use remains constant in this function.

II]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);
}
```

Recurrence Relation:

$T(k) = (n - k + 1) * T(k + 1)$

→ In the worst case, each level tries up to (n - k + 1) vertices.

Overall, the total time in worst case becomes:

T(1) = O(n!)

Time Complexity

I] **Best Case:**

Time Complexity: O(n)

A valid Hamiltonian cycle is found quickly without exploring all possibilities (very rare in practice).

II] **Average Case:**

Time Complexity: O(n!)

The algorithm backtracks through many permutations of vertices to find valid Hamiltonian cycles.

III] **Worst Case:**

Time Complexity: O(n!)

In the worst case, it explores all n! permutations (excluding rotations and reversals) to check for Hamiltonian cycles.

Space Complexity

I] **Best Case:**

Space Complexity: O(n)

Stores the path x[1:n] and uses recursion stack of depth n.

II] **Average Case:**

Space Complexity: O(n)

Even with average exploration, recursive depth and path array remain at most n.

III] **Worst Case:**

Space Complexity: O(n)

Maximum recursion depth is n, and path array holds n elements.

PROGRAM –

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define MAX 20

int x[MAX], G[MAX][MAX], n;
int solutionCount = 0;
int solutions[MAX][MAX];

long long getMicrotime() {
    LARGE_INTEGER frequency, counter;
    QueryPerformanceFrequency(&frequency);
    QueryPerformanceCounter(&counter);
    return (counter.QuadPart * 1000000) / frequency.QuadPart;
}

void printCurrentState(int k, const char *state) {
    printf("\n%s ", state);
    for (int i = 1; i <= k; i++) {
        printf("x%d=%d ", i, x[i]);
    }
    printf("\n");
}

void displaySolutionMatrix() {
    printf("\nSolution Matrix:\n");
    for (int i = 0; i < solutionCount; i++) {
        printf("%c = {", 'A' + i, n);
        for (int j = 0; j < n; j++) {
            printf("%d", solutions[i][j]);
            if (j < n - 1) {
                printf(", ");
            }
        }
        printf(", 1}\n");
    }
}

void NextValue(int k) {
    int j;
    while (1) {
        x[k] = (x[k] + 1) % (n + 1);
        if (x[k] == 0) {
            printCurrentState(k, "Bounded: No valid vertex available");
            return;
        }

        if (G[x[k - 1]][x[k]] != 0) {
            for (j = 1; j < k; j++) {
                if (x[j] == x[k]) break;
            }
            if (j == k) {
                if ((k < n) || (k == n && G[x[n]][x[1]] != 0)) {
                    return;
                } else if (k == n) {
                    printCurrentState(k, "Bounded: No edge back to starting vertex");
                }
            } else {
                printCurrentState(k, "Bounded: Vertex already in path");
            }
        } else {
        }
    }
}

void Hamiltonian(int k) {
    while (1) {
        NextValue(k);
        if (x[k] == 0) return;
        if (k == n) {
            solutionCount++;
            printf("\nSolution %d found: ", solutionCount);
            for (int i = 1; i <= n; i++) {
                printf("%d ", x[i]);
                solutions[solutionCount - 1][i - 1] = x[i];
            }
            printf("%d\n", x[1]);
        } else {
            Hamiltonian(k + 1);
        }
    }
}

void inputGraph() {
    printf("Enter number of vertices (n, max %d): ", MAX - 1);
    scanf("%d", &n);

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

```
    for (int j = 1; j <= n; j++) {
        G[i][j] = 0;
    }
  }

  printf("Enter the adjacency matrix (1 for edge, 0 for no edge):\n");
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      printf("Edge between %d and %d (1/0): ", i, j);
      scanf("%d", &G[i][j]);
    }
  }
}

void displayGraph() {
  printf("\nAdjacency Matrix:\n");
  for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
      printf("%d ", G[i][j]);
    }
    printf("\n");
  }
}
int main() {
  printf("**************************************************\n");
  printf(" Roll number: 23B-CO-010\n");
  printf(" PR Number - 202311390\n");
  printf("**************************************************\n\n");
  int choice;
  do {
    printf("\n=== HAMILTONIAN CYCLE FINDER ===\n");
    printf("1. Input Graph\n");
    printf("2. Display Graph\n");
    printf("3. Find All Hamiltonian Cycles\n");
    printf("4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch (choice) {
      case 1:
        inputGraph();
        for (int i = 1; i <= n; i++) x[i] = 0;
        x[1] = 1;
        solutionCount = 0;
        break;
      case 2:
        if (n == 0) {
          printf("Please input a graph first.\n");
        } else {
          displayGraph();
        }
        break;
      case 3:
        if (n == 0) {
          printf("Please input a graph first.\n");
        } else {
          printf("\nSearching for Hamiltonian cycles...\n");
          printf("Starting with vertex 1\n");
          solutionCount = 0;
          long long startTime = getMicrotime();
          Hamiltonian(2);
          long long endTime = getMicrotime();

          if (solutionCount == 0) {
            printf("No Hamiltonian cycles found in the graph.\n");
          } else {
            printf("Total Hamiltonian cycles found: %d\n", solutionCount);
          }
          printf("Time taken: %lld microseconds\n", endTime - startTime);
        }
        if (solutionCount == 0) {
          printf("No solutions available. Please find solutions first.\n");
        } else {
          displaySolutionMatrix();
        }
        break;
      case 4:
        printf("Exiting program. Goodbye!\n");
        break;
      default:
        printf("Invalid choice. Please try again.\n");
    }
  } while (choice != 5);

  return 0;
}
```

INPUT –

```
**************************************************
 Roll number: 23B-CO-010
 PR Number - 202311390
**************************************************


=== HAMILTONIAN CYCLE FINDER ===
1. Input Graph
2. Display Graph
3. Find All Hamiltonian Cycles
4. Exit
Enter your choice: 1
Enter number of vertices (n, max 19): 5
Enter the adjacency matrix (1 for edge, 0 for no edge):
Edge between 1 and 1 (1/0): 0
Edge between 1 and 2 (1/0): 1
Edge between 1 and 3 (1/0): 1
Edge between 1 and 4 (1/0): 1
Edge between 1 and 5 (1/0): 0
Edge between 2 and 1 (1/0): 1
Edge between 2 and 2 (1/0): 0
Edge between 2 and 3 (1/0): 1
Edge between 2 and 4 (1/0): 0
Edge between 2 and 5 (1/0): 1
Edge between 3 and 1 (1/0): 1
Edge between 3 and 2 (1/0): 1
Edge between 3 and 3 (1/0): 0
Edge between 3 and 4 (1/0): 0
Edge between 3 and 5 (1/0): 1
Edge between 4 and 1 (1/0): 1
Edge between 4 and 2 (1/0): 0
Edge between 4 and 3 (1/0): 0
Edge between 4 and 4 (1/0): 0
Edge between 4 and 5 (1/0): 1
Edge between 5 and 1 (1/0): 0
Edge between 5 and 2 (1/0): 1
Edge between 5 and 3 (1/0): 1
Edge between 5 and 4 (1/0): 1
Edge between 5 and 5 (1/0): 0
```

OUTPUT –

```
=== HAMILTONIAN CYCLE FINDER ===
1. Input Graph
2. Display Graph
3. Find All Hamiltonian Cycles
4. Exit
Enter your choice: 3

Searching for Hamiltonian cycles...
Starting with vertex 1

Bounded: Vertex already in path x1=1 x2=2 x3=1

Bounded: Vertex already in path x1=1 x2=2 x3=3 x4=1

Bounded: Vertex already in path x1=1 x2=2 x3=3 x4=2

Bounded: Vertex already in path x1=1 x2=2 x3=3 x4=5 x5=2

Bounded: Vertex already in path x1=1 x2=2 x3=3 x4=5 x5=3

Solution 1 found: 1 2 3 5 4 1

Bounded: No valid vertex available x1=1 x2=2 x3=3 x4=5 x5=0

Bounded: No valid vertex available x1=1 x2=2 x3=3 x4=0

Bounded: Vertex already in path x1=1 x2=2 x3=5 x4=2

Bounded: Vertex already in path x1=1 x2=2 x3=5 x4=3 x5=1

Bounded: Vertex already in path x1=1 x2=2 x3=5 x4=3 x5=2

Bounded: Vertex already in path x1=1 x2=2 x3=5 x4=3 x5=5

Bounded: No valid vertex available x1=1 x2=2 x3=5 x4=3 x5=0

Bounded: Vertex already in path x1=1 x2=2 x3=5 x4=4 x5=1

Bounded: Vertex already in path x1=1 x2=2 x3=5 x4=4 x5=5

Bounded: No valid vertex available x1=1 x2=2 x3=5 x4=4 x5=0

Bounded: No valid vertex available x1=1 x2=2 x3=5 x4=0

Bounded: No valid vertex available x1=1 x2=2 x3=0

Bounded: Vertex already in path x1=1 x2=3 x3=1

Bounded: Vertex already in path x1=1 x2=3 x3=2 x4=1

Bounded: Vertex already in path x1=1 x2=3 x3=2 x4=3

Bounded: Vertex already in path x1=1 x2=3 x3=2 x4=5 x5=2

Bounded: Vertex already in path x1=1 x2=3 x3=2 x4=5 x5=3

Solution 2 found: 1 3 2 5 4 1
```

```
Bounded: No valid vertex available x1=1 x2=3 x3=2 x4=5 x5=0

Bounded: No valid vertex available x1=1 x2=3 x3=2 x4=0

Bounded: Vertex already in path x1=1 x2=3 x3=5 x4=2 x5=1

Bounded: Vertex already in path x1=1 x2=3 x3=5 x4=2 x5=3

Bounded: Vertex already in path x1=1 x2=3 x3=5 x4=2 x5=5

Bounded: No valid vertex available x1=1 x2=3 x3=5 x4=2 x5=0

Bounded: Vertex already in path x1=1 x2=3 x3=5 x4=3

Bounded: Vertex already in path x1=1 x2=3 x3=5 x4=4 x5=1

Bounded: Vertex already in path x1=1 x2=3 x3=5 x4=4 x5=5

Bounded: No valid vertex available x1=1 x2=3 x3=5 x4=4 x5=0

Bounded: No valid vertex available x1=1 x2=3 x3=5 x4=0

Bounded: No valid vertex available x1=1 x2=3 x3=0

Bounded: Vertex already in path x1=1 x2=4 x3=1

Bounded: Vertex already in path x1=1 x2=4 x3=5 x4=2 x5=1

Solution 3 found: 1 4 5 2 3 1

Bounded: Vertex already in path x1=1 x2=4 x3=5 x4=2 x5=5

Bounded: No valid vertex available x1=1 x2=4 x3=5 x4=2 x5=0

Bounded: Vertex already in path x1=1 x2=4 x3=5 x4=3 x5=1

Solution 4 found: 1 4 5 3 2 1

Bounded: Vertex already in path x1=1 x2=4 x3=5 x4=3 x5=5

Bounded: No valid vertex available x1=1 x2=4 x3=5 x4=3 x5=0

Bounded: Vertex already in path x1=1 x2=4 x3=5 x4=4

Bounded: No valid vertex available x1=1 x2=4 x3=5 x4=0

Bounded: No valid vertex available x1=1 x2=4 x3=0

Bounded: No valid vertex available x1=1 x2=0
Total Hamiltonian cycles found: 4
Time taken: 50774 microseconds

Solution Matrix:
A = {1, 2, 3, 5, 4, 1}
B = {1, 3, 2, 5, 4, 1}
C = {1, 4, 5, 2, 3, 1}
D = {1, 4, 5, 3, 2, 1}
```

TIME TAKEN –

```
Time taken: 50774 microseconds
```

CONCLUSION – Backtracking was successfully used to determine hamiltonean cycles in the graph.

DATE –

KNAPSACK PROBLEM

AIM- Write a C program to implement 0/1 Knapsack problem using backtracking.

Problem statement – Consider we are given n objects and knapsack capacity of 'm' ,each object i has weight w_i and profit p_i .The objective is to fill the knapsack that maximizes the profits and since the capacity is m ,the toal weight must be less than or equal to m .

Input – $[p_1.....p_7]$ = { 17,14,20,18,22}

$[w_1....w_7]$ = 6,5,10,11,14} m = 21

Output - Generate the list of items that give maximum profit and has the total weight within the knapsack capacity .

ALGORITHM

I]Algorithm Bound(cp,cw,k)

```
// cp is the current profit total, cw is the current
// weight total; k is the index of the last removed
// item; and m is the knapsack size.
{
   b := cp; c := cw;
   for i := k + 1 to n do
   {
      c := c + w[i];
      if (c < m) then b := b + p[i];
      else return b + (1 - (c - m)/w[i]) * p[i];
   }
   return b;
}
```

Recurrence Relation

The Bound function does not use recursion

Time Complexity

I] **Best Case:**

Time Complexity: O(1)

➞ If the first remaining item exceeds the knapsack capacity, the loop exits early with a fractional profit calculation.

II] **Average Case:**

Time Complexity: O(n - k)

It checks and sums weights/profits for remaining items and may compute fractional item profit.

III] **Worst Case:**

Time Complexity: O(n - k) → O(n)

The loop runs from i = k+1 to n, iterating through all remaining items.

Space Complexity

I] **Best Case:**

Space Complexity: O(1)

➞ Only scalar variables b, c, i are used.

II] **Average Case:**

Space Complexity: O(1)

➞ No additional data structures used.

III] **Worst Case:**

Space Complexity: O(1)

➞ Constant space regardless of input size.

II]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]≥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] ≤ 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) ≥ 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];
      }
   }
}
```

Recurrence Relation

Let **T(k)** be the time to solve the problem for the k-th item.

$T(k)=2\cdot T(k+1)+O(n)$(in worst case)

- Because for each item, the algorithm branches into **two calls** (left and right child).
- The copy operation (for j := 1 to k) takes **O(n)** time.
- The Bound() function runs in **O(n - k)** (as previously explained), which is upper-bounded by **O(n)**.

Time Complexity

I] **Best Case:**

Time Complexity: O(n)

When **bounding** quickly prunes large parts of the tree, especially if all right branches are cut off early.

II] **Average Case:**

Time Complexity: O(2^k) where k ≤ n

Depends on pruning efficiency; not all branches may be explored due to bounding, but several partial paths might be explored.

III] **Worst Case:**

Time Complexity: O(2^n × n)

All 2^n subsets are explored (like brute-force), and each step has a loop of up to n due to Bound() and for loop.

Space Complexity

I] **Best Case:**

Space Complexity: O(n)

Only a single path of depth n stored in the recursion stack.

II] **Average Case:**

Space Complexity: O(n)

Recursion depth goes up to n; no extra structures apart from arrays of size n.

III] **Worst Case:**

Space Complexity: O(n)

Still only recursive depth of n, and arrays like x[], y[] of size n are reused.

PROGRAM –

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define MAX_ITEMS 100

int w[MAX_ITEMS];
int p[MAX_ITEMS];
int x[MAX_ITEMS];
int y[MAX_ITEMS];
int n, m;
int fw = 0, fp = 0;
int bound_nodes = 0;

typedef enum {
  NORMAL,
  BOUNDED,
  NON_FEASIBLE,
  SOLUTION
} State;

float Bound(int cp, int cw, int k) {
  float b = cp;
  float c = cw;

  for (int i = k; i < n; i++) {
    if (c + w[i] <= m) {
      b += p[i];
      c += w[i];
    } else {
      b += ((float)(m - c) / w[i]) * p[i];
      break;
    }
  }

  bound_nodes++;
  return b;
}

void printNode(int k, int cp, int cw, State state, float bound_value) {
  if (state == NORMAL) {
    return;
  }
  printf("Node: k = %d, cp = %d, cw = %d, Bound = %.2f, State = ", k+1, cp, cw, bound_value);
  switch (state) {
    case BOUNDED:
      printf("BOUNDED\n");
      break;
    case NON_FEASIBLE:
      printf("NON-FEASIBLE\n");
      break;
    case SOLUTION:
      printf("SOLUTION\n");
      break;
    default:
      break;
  }
}

void BKnap(int k, int cp, int cw) {
  float node_bound;

  if (cw + w[k - 1] <= m) {
    y[k - 1] = 1;
    if (k < n) {
      node_bound = Bound(cp + p[k - 1], cw + w[k - 1], k);
      if (node_bound > fp) {
        printNode(k, cp + p[k - 1], cw + w[k - 1], NORMAL, node_bound);
        BKnap(k + 1, cp + p[k - 1], cw + w[k - 1]);
      } else {
        printNode(k, cp + p[k - 1], cw + w[k - 1], BOUNDED, node_bound);
      }
    }

    if ((cp + p[k - 1] > fp) && (k == n)) {
      fp = cp + p[k - 1];
      fw = cw + w[k - 1];
      for (int j = 0; j < n; j++) {
        x[j] = y[j];
      }
      printNode(k, cp + p[k - 1], cw + w[k - 1], SOLUTION, (float)fp);
    }
  } else {
    printNode(k, cp, cw, NON_FEASIBLE, 0.0);
  }

  node_bound = Bound(cp, cw, k);
  if (node_bound > fp) {
    y[k - 1] = 0;

    if (k < n) {
```

```
        printNode(k, cp, cw, BOUNDED, node_bound);
        BKnap(k + 1, cp, cw);
    }

    if ((cp > fp) && (k == n)) {
        fp = cp;
        fw = cw;
        for (int j = 0; j < n; j++) {
            x[j] = y[j];
        }
        printNode(k, cp, cw, SOLUTION, (float)fp);
    }
  } else {
    printNode(k, cp, cw, BOUNDED, node_bound);
  }
}

long long current_timestamp() {
  clock_t t = clock();
  long long microseconds = (long long)(t * 1000000.0 / CLOCKS_PER_SEC);
  return microseconds;
}

int main() {

  printf("**********************************************************\n");
  printf(" Roll number: 23B-CO-010\n");
  printf(" PR Number - 202311390\n");
  printf("**********************************************************\n\n");
  int choice;

  do {
    printf("\n------- Knapsack Problem Using Backtracking -------\n");
    printf("1. Enter input data\n");
    printf("2. Solve Knapsack Problem\n");
    printf("3. Exit\n");
    printf("Enter your choice: ");
    scanf("%d", &choice);

    switch(choice) {
      case 1:
        printf("Enter number of items: ");
        scanf("%d", &n);

        printf("Enter knapsack capacity: ");
        scanf("%d", &m);

        printf("Enter weights of items:\n");
        for (int i = 0; i < n; i++) {
          printf("Weight of item %d: ", i + 1);
          scanf("%d", &w[i]);
        }

        printf("Enter profits of items:\n");
        for (int i = 0; i < n; i++) {
          printf("Profit of item %d: ", i + 1);
          scanf("%d", &p[i]);
        }
        break;

      case 2:
        bound_nodes = 0;
        fp = 0;
        fw = 0;

        for (int i = 0; i < n; i++) {
          x[i] = 0;
          y[i] = 0;
        }

        long long start_time = current_timestamp();
        BKnap(1, 0, 0);
        long long end_time = current_timestamp();

        printf("\n----- Solution -----\n");
        printf("\nSolution vector: ");
        for (int i = 0; i < n; i++) {
          printf("x%d = %d", i + 1, x[i]);
          if (i < n - 1) {
            printf(", ");
          }
        }
        printf("\n");

        printf("Items included in knapsack:\n");
        for (int i = 0; i < n; i++) {
          if (x[i] == 1) {
```

```
            printf("Item %d (Weight: %d, Profit: %d)\n", i + 1, w[i], p[i]);
        }
    }

    printf("\nMaximum profit: %d\n", fp);
    printf("Final weight: %d\n", fw);

    printf("Total bounded nodes visited: %d\n", bound_nodes);
    printf("Time taken: %lld microseconds\n", end_time - start_time);
    break;

case 3:
    printf("Exiting program. Goodbye!\n");
    break;
default:
    printf("Invalid choice. Please try again.\n");
}
} while (choice != 3);

return 0;
}
```

INPUT –

```
**********************************************************
 Roll number: 23B-CO-010
 PR Number - 202311390
**********************************************************


------- Knapsack Problem Using Backtracking -------
1. Enter input data
2. Solve Knapsack Problem
3. Exit
Enter your choice: 1
Enter number of items: 5
Enter knapsack capacity: 21
Enter weights of items:
Weight of item 1: 6
Weight of item 2: 5
Weight of item 3: 10
Weight of item 4: 11
Weight of item 5: 14
Enter profits of items:
Profit of item 1: 17
Profit of item 2: 14
Profit of item 3: 20
Profit of item 4: 18
Profit of item 5: 22
```

OUTPUT –

```
------- Knapsack Problem Using Backtracking -------
1. Enter input data
2. Solve Knapsack Problem
3. Exit
Enter your choice: 2
Node: k = 5, cp = 69, cw = 32, Bound = 0.00, State = NON-FEASIBLE
Node: k = 6, cp = 73, cw = 35, Bound = 0.00, State = NON-FEASIBLE
Node: k = 6, cp = 51, cw = 21, Bound = 51.00, State = SOLUTION
Node: k = 4, cp = 31, cw = 11, Bound = 47.36, State = BOUNDED
Node: k = 3, cp = 17, cw = 6, Bound = 45.18, State = BOUNDED
Node: k = 2, cp = 0, cw = 0, Bound = 43.82, State = BOUNDED

----- Solution -----

Solution vector: x1 = 1, x2 = 1, x3 = 1, x4 = 0, x5 = 0
Items included in knapsack:
Item 1 (Weight: 6, Profit: 17)
Item 2 (Weight: 5, Profit: 14)
Item 3 (Weight: 10, Profit: 20)

Maximum profit: 51
Final weight: 21
```

TIME TAKEN –

```
Time taken: 1000 microseconds
```

CONCLUSION – Knapsack problem was successfully solved using backtracking algorithm