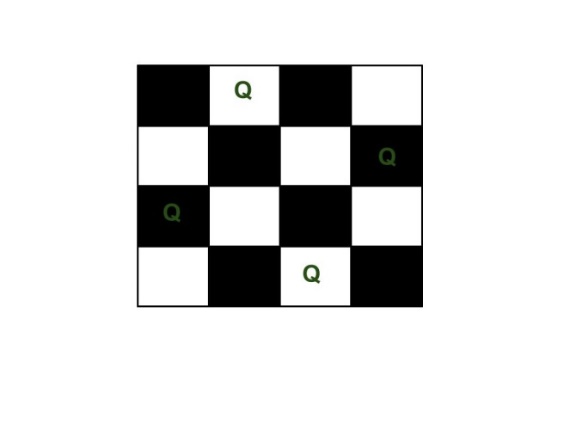
**TASK 11**

**Backtracking - n-Queen’s problem CO4 K3**

Two players are playing a game on a N\*N chessboard. The rules of the game are as follows: no two queens attack each other. For example, the following is a solution for the 4 Queen problem. 

The expected output is in form of a matrix that has ‘Q’s for the blocks where queens are placed and the empty spaces  are represented by ‘.’s . For example, the following is the output matrix for the above 4 queen solution.

*. . Q .   
Q . . .   
. . . Q   
. Q . .*

**Test Case 1:** Implement Rat in a Maze problem by applying the concept of backtracking.

**Test Case 2:** Modify the n-queens problem to handle the value of n in the range from 1 to 3.

**Aim:**

Create to c program to Implement N Queen's problem using Backtracking algorithm

**Algorithm**:

Step 1 : Create and initialize the variable board[] for the required size of chess board.

Step 2 : write a recursive function queen() to check for proper positioning of queen to check conflicts

Step 3:Write a function place() to check conflicts, If no conflict for desired position returns 1 otherwise returns 0.

Step 4 : Loop until all queens are properly placed in the board

**Program:**

#include<stdio.h>

#include<math.h>

#include <stdlib.h>

int board[20],count;

int main()

{

int n,i,j;

void queen(int row,int n);

printf(" - N Queens Problem Using Backtracking -");

printf("\n\nEnter number of Queens:");

scanf("%d",&n);

queen(1,n);

return 0;

}

//function for printing the solution

void print(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) //for nxn board

{

if(board[i]==j)

printf("\tQ"); //queen at i,j position

else

printf("\t-"); //empty slot

}

}

}

/\*funtion to check conflicts

If no conflict for desired postion returns 1 otherwise returns 0\*/

int place(int row,int column)

{

int i;

for(i=1;i<=row-1;++i)

{

//checking column and digonal conflicts

if(board[i]==column)

return 0;

else

if(abs(board[i]-column)==abs(i-row))

return 0;

}

return 1; //no conflicts

}

//function to check for proper positioning of queen

void queen(int row,int n)

{

int column;

for(column=1;column<=n;++column)

{

if(place(row,column))

{

board[row]=column; //no conflicts so place queen

if(row==n) //dead end

print(n); //printing the board configuration

else //try queen with next position

queen(row+1,n);

}

}

}

**Output:**

- N Queens Problem Using Backtracking -

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 - -

**Test Case 1:** Implement Rat in a Maze problem by applying the concept of backtracking.

**Aim:**

Create to c program to Implement Implement Rat in a Maze problem by applying the concept of backtracking algorithm

Algorithm:

Step 1: Create a solution matrix, initially filled with 0’s.

Step 2:Create a recursive function, which takes initial matrix, output matrix and position of rat (i, j).

Step 3: if the position is out of the matrix or the position is not valid then return.

Step 4 :Mark the position output[i][j] as 1 and check if the current position is destination or not. If destination is reached print the output matrix and return.

Step 5: Recursively call for position (i+1, j) and (i, j+1).

Step 6:Unmark position (i, j), i.e output[i][j] = 0.

Program:

// C program to solve Rat in a Maze problem using backtracking

#include <stdio.h>

#include <stdbool.h>

// Maze size

#define N 4

bool solveMazeUtil(int maze[N][N], int x, int y,int sol[N][N]);

// A utility function to print solution matrix sol[N][N]

void printSolution(int sol[N][N])

{

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

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

printf(" %d ", sol[i][j]);

printf("\n");

}

}

// A utility function to check if x, y is valid index for

// N\*N maze

bool isSafe(int maze[N][N], int x, int y)

{

// if (x, y outside maze) return false

if (x >= 0 && x < N && y >= 0 && y < N && maze[x][y] == 1)

return true;

return false;

}

bool solveMaze(int maze[N][N])

{

int sol[N][N] = { { 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 },

{ 0, 0, 0, 0 } };

if (solveMazeUtil(maze, 0, 0, sol) == false) {

printf("Solution doesn't exist");

return false;

}

printSolution(sol);

return true;

}

// A recursive utility function to solve Maze problem

bool solveMazeUtil(int maze[N][N], int x, int y, int sol[N][N])

{

// if (x, y is goal) return true

if (x == N - 1 && y == N - 1 && maze[x][y] == 1) {

sol[x][y] = 1;

return true;

}

// Check if maze[x][y] is valid

if (isSafe(maze, x, y) == true) {

// Check if the current block is already part of

// solution path.

if (sol[x][y] == 1)

return false;

// mark x, y as part of solution path

sol[x][y] = 1;

/\* Move forward in x direction \*/

if (solveMazeUtil(maze, x + 1, y, sol) == true)

return true;

// If moving in x direction doesn't give solution

// then Move down in y direction

if (solveMazeUtil(maze, x, y + 1, sol) == true)

return true;

// If none of the above movements work then

// BACKTRACK: unmark x, y as part of solution path

sol[x][y] = 0;

return false;

}

return false;

}

// driver program to test above function

int main()

{

int maze[N][N] = { { 1, 0, 0, 0 },

{ 1, 1, 0, 1 },

{ 0, 1, 0, 0 },

{ 1, 1, 1, 1 } };

solveMaze(maze);

return 0;

}

Output:

1 0 0 0

1 1 0 0

0 1 0 0

0 1 1 1

**Test Case 2:** Modify the n-queens problem to handle the value of n in the range from 1 to 3.

**Aim:**

Create to c program to Implement N Queen's problem using Backtracking algorithm to handle the value of n in the range from 1 to 3

**Algorithm**:

Step 1 : Create and initialize the variable board[] for the required size of chess board.

Step 2 : write a recursive function queen() to check for proper positioning of queen to check conflicts

Step 3:Write a function place() to check conflicts, If no conflict for desired position returns 1 otherwise returns 0.

Step 4 : Loop until all queens are properly placed in the board

**Program:**

#include<stdio.h>

#include<math.h>

#include <stdlib.h>

int a[30],count=0;

int place(int pos) {

int i;

for (i=1;i<pos;i++) {

if((a[i]==a[pos])||((abs(a[i]-a[pos])==abs(i-pos))))

return 0;

}

return 1;

}

void print\_sol(int n) {

int i,j;

count++;

printf("\n\nSolution #%d:\n",count);

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

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

if(a[i]==j)

printf("Q\t"); else

printf("\*\t");

}

printf("\n");

}}

void queen(int n) {

int k=1;

a[k]=0;

while(k!=0) {

a[k]=a[k]+1;

while((a[k]<=n)&&!place(k))

a[k]++;

if(a[k]<=n) {

if(k==n)

print\_sol(n); else {

k++;

a[k]=0;

}

} else

k--;

}

}

int main() {

int i,n;

printf("Enter the number of Queens\n");

scanf("%d",&n);

queen(n);

printf("\nTotal solutions=%d",count);

}

**Output:**

Enter the number of Queens:1

Solution #1:

Q

Total solutions=1

Enter the number of Queens:2

Total solutions=0

Enter the number of Queens:3

Total solutions=0

**Result:**

Thus the N Queen's problem using Backtracking algorithm was executed successfully.

**TASK 12**

**Backtracking - Subset problem**  **CO4 K3**

An established group of scientists are working on finding solution to NP hard problems. They claim Subset Sum as an NP-hard problem. The problem is to determine whether there exists a subset of a given set S whose sum is a given number K.  
You are a computer engineer and you claim to solve this problem given that all numbers in the set are non-negative. Given a set S of size N of non-negative integers, find whether there exists a subset whose sum is K.

### Input

First line of input contains T, the number of test cases. T test cases follow.  
Each test case contains 2 lines. First line contains two integers N and K. Next line contains N space separated non-negative integers (each less than 100000).  
0 < T < 1000  
0 < N < 1000  
0 < K < 1000

### Output

Output T lines, one for each test case. Every line should be either 0 or 1 depending on whether such a subset exists or not.

### Example

**Input:**

2

5 10

3 4 6 1 9

3 2

1 3 4

**Output:**

1

0

**Test Case 1:** Assume an instance of the subset sum problem and verify whether the backtracking algorithm work correctly if we use just one of the two inequalities to terminate a node as non-promising.

**Test Case 2:** Given a set of distinct integers and return all the possible subsets or power set

## Aim

Create to c program to find the subset of elements that are selected from a given set whose sum adds up to a given number K. It considering the set contains non-negative values.

**Algorithm**

**Step 1:** Start with an empty set.

**Step 2:** Add to the subset, the next element from the list.

**Step 3:** If the subset is having sum m then stop with that subset as solution.

**Step 4:** If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

**Step 5:** If the subset is feasible then repeat step 2.

**Step 6:** If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

**Program**

#include <stdio.h>

int s[10], d, n, set[10], count = 0;

void display(int);

int flag = 0;

void main() {

void subset(int, int); // Change return type to void

int i;

printf("ENTER THE NUMBER OF THE ELEMENTS IN THE SET : ");

scanf("%d", &n);

printf("ENTER THE SET OF VALUES : ");

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

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

printf("ENTER THE SUM : ");

scanf("%d", &d);

printf("THE PROGRAM OUTPUT IS: ");

subset(0, 0);

if (flag == 0)

printf("There is no solution\n");

}

void subset(int sum, int i) { // Change return type to void

if (sum == d) {

flag = 1;

display(count);

return;

}

if (sum > d || i >= n)

return;

set[count] = s[i];

count++;

subset(sum + s[i], i + 1);

count--;

subset(sum, i + 1);

}

void display(int count) {

int i;

printf("\t{");

for (i = 0; i < count; i++)

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

printf("}\n");

}

**Input:**

Enter the number of the elements in the set : 5

Enter the set of values : 6 4 3 2 1

Enter the sum : 5

**Output:**

The program output is: {4,1,} {3,2,}

**Input:**

Enter the number of the elements in the set : 5

Enter the set of values : 1 2 5 6 8

Enter the sum : 9

**Output:**

The program output is: {1,2,6,} {1,8,}

**Test case 1:**Assume an instance of the subset sum problem and verify whether the backtracking algorithm work correctly if we use just one of the two inequalities to terminate a node as non-promising.

## Aim

Create to c program to find the subset sum problem and verify whether the backtracking algorithm work correctly if we use just one of the two inequalities to terminate a node as non-promising.

**Algorithm**

**Step 1:** Start with an empty set.

**Step 2:** Add to the subset, the next element from the list.

**Step 3:** If the subset is having sum m then stop with that subset as solution.

**Step 4:** If the subset is not feasible or if we have reached the end of the set then backtrack through the subset until we find the most suitable value.

**Step 5:** If the subset is feasible then repeat step 2.

**Step 6:** If we have visited all the elements without finding a suitable subset and if no backtracking is possible then stop without solution.

**Program**

#include <stdio.h>

#include <stdlib.h>

static int total\_nodes;

void printValues(int A[], int size)

{

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

{

printf("%\*d", 5, A[i]);

}

printf("\n");}

void subset\_sum(int s[], int t[], int s\_size, int t\_size, int sum, int ite, int const target\_sum)

{

total\_nodes++;

if (target\_sum == sum)

{

printValues(t, t\_size);

subset\_sum(s, t, s\_size, t\_size - 1, sum - s[ite], ite + 1, target\_sum);

return;

}

else

{

for (int i = ite; i < s\_size; i++)

{

t[t\_size] = s[i];

subset\_sum(s, t, s\_size, t\_size + 1, sum + s[i], i + 1, target\_sum);

}

}

}

void generateSubsets(int s[], int size, int target\_sum)

{

int\* tuplet\_vector = (int\*)malloc(size \* sizeof(int)); subset\_sum(s, tuplet\_vector, size, 0, 0, 0, target\_sum); free(tuplet\_vector);

}

int main()

{

int set[] = { 5, 6, 12 , 54, 2 , 20 , 15 };

int size = sizeof(set) / sizeof(set[0]);

printf("The set is ");

printValues(set , size);

generateSubsets(set, size, 25);

printf("Total Nodes generated %d\n", total\_nodes); return 0;

}

**Input:**

The set is 5 6 12 54 2 20 15

5 6 12 2

5 20

**Output:**

Total Nodes generated 127

**Test Case 2:** Given a set of distinct integers and return all the possible subsets or power set

**Aim**

To create a C program to Implement to given a set of distinct integers and return all the possible subsets or power set.

**Algorithm**

Step 1: Call will be made to subset Backtrack() with S as array of integers, list for storing and printing subset, and i index.

Step 2: If all the elements of the array are processed then print list and return from method.

Step 3: Two Choices - include the current element into the subset. If yes then add current element to list and call subsetBacktrack with i++. Otherwise call method subsetBacktrack with the same arguments.

Step 4: Print all the subsets.

**Program:**

#include <stdio.h>

#include <stdlib.h>

#define N 3 // Define the size of the array

int arr[N]; // Declare the array

// Function to generate all subsets

void allSubsets(int pos, int len, int subset[]) {

if (pos == N) {

// Print the subset

printf("[");

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

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

if (i < len - 1)

printf(", ");

}

printf("]\n");

return;

}

// Include the current element and recurse

subset[len] = arr[pos];

allSubsets(pos + 1, len + 1, subset);

// Skip the current element and recurse

allSubsets(pos + 1, len, subset);

}

int main() {

// Initialize the array

arr[0] = 1;

arr[1] = 2;

arr[2] = 3;

// Temporary array to store subsets

int subset[N];

// Call the function to generate subsets

allSubsets(0, 0, subset);

return 0;

}

**Output:**

[1, 2, 3]

[1, 2]

[1, 3]

[1]

[2, 3]

[2]

[3]

[]

**Result:**

Thus the Subset problem using Backtracking algorithm was executed successfully.

**TASK 13**

**Branch & Bound - Travelling Salesman problem CO4 K3**

### In a bustling city, a courier named Maya takes on a challenging assignment to deliver packages to n different locations efficiently. The goal is to find the shortest route that allows her to visit each delivery point exactly once and return to the starting point.

### Input

First line contains an integer N = number of cities

N lines follow

Each line contains label x y

where:

label = label of the city (String)

x = x co-ordinate of the city (Real Valued)

y = y co-ordiate of the city (Real Valued)

### Output

Output an optimal path in space separated manner.

### Sample Input#1

4

0 0 0

1 0 1

2 1 0

3 1 1

### Sample Output#1

0 1 2 3

**Test Case 1:** Derive the solution of TSP using B&B and prove that it is optimal when compared with Brute Force technique.

**Test Case 2:** Analyze that the Travelling Salesman Problem (TSP) is an NP hard problem.

**Aim:**

Create to C program to implement Traveling Salesman Problem using Branch and Bound algorithm.

**Algorithm**:

**Step1.** Input the first node as root. Identify the root, if there is still a node to expand then do the

branching if there is no more node (empty) then the search is done.

**Step2.** From the branch, choose the node (*i*) with the minimum cost. If exist several nodes with the same cost, then choose arbitrarily.

**Step3.** Identify the node (*i*),

**Step 3.1** If the (*i*) is the solution –no more queue node– then stop the search,

**Step 3.2** If the (*i*) is not the solution, then do the branching and choose the node (*i*) with the minimum cost. If exist several nodes with the same cost, then choose arbitrarily.

**Step 4.** Back to step 3.

**Program:**

#include <stdio.h>

int a[10][10], visited[10], n, cost = 0;

void get();

void mincost(int city);

int least(int c); // Function prototype added

void put();

void get() {

int i, j;

printf("Enter No. of Cities: ");

scanf("%d", &n);

printf("\nEnter Cost Matrix: \n");

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

printf("\n Enter Elements of Row# : %d\n", i + 1);

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

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

visited[i] = 0;

}

printf("\n\nThe cost list is:\n\n");

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

printf("\n\n");

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

printf("\t %d", a[i][j]);

}

}

void mincost(int city) {

int i, ncity;

visited[city] = 1;

printf("%d -> ", city + 1);

ncity = least(city);

if (ncity == 999) {

ncity = 0;

printf("%d", ncity + 1);

cost += a[city][ncity];

return;

}

mincost(ncity);

}

int least(int c) {

int i, nc = 999;

int min = 999, kmin;

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

if ((a[c][i] != 0) && (visited[i] == 0)) {

if (a[c][i] < min) {

min = a[i][0] + a[c][i];

kmin = a[c][i];

nc = i;

}

}

}

if (min != 999)

cost += kmin;

return nc;

}

void put() {

printf("\n\nMinimum cost: %d\n", cost);

}

void main() {

get();

printf("\n\nThe Path is:\n\n");

mincost(0);

put();

}

**Output:**

Enter No. of Cities: 6

Enter Cost Matrix:

Enter Elements of Row# : 1

99 10 15 20 99 8

Enter Elements of Row# : 2

5 99 9 10 8 99

Enter Elements of Row# : 3

6 13 99 12 99 5

Enter Elements of Row# : 4

8 8 9 99 6 99

Enter Elements of Row# : 5

99 10 99 6 99 99

Enter Elements of Row# : 6

10 99 5 99 99 99

The Path is:

1 –>6 –>3 –>4 –>5 –>2 –>1

Minimum cost: 46

**Test Case 1:** Derive the solution of TSP using B&B and prove that it is optimal when compared with Brute Force technique.

**Branch and Bound (B&B):**

B&B is a technique used to solve combinatorial optimization problems like the Traveling Salesman Problem (TSP).

It involves exploring the search space using a systematic approach and pruning branches that are not promising, hence reducing the search space.

**Brute Force Technique:**

Brute force involves trying all possible permutations of cities and calculating the total distance for each permutation.

It guarantees finding the optimal solution but becomes impractical for large problem instances due to its exponential time complexity.

**Proving Optimality with B&B:**

B&B also guarantees finding the optimal solution but is more efficient than brute force due to pruning.

Conducting experiments to measure the execution time of both algorithms for different problem sizes, demonstrating that B&B is significantly faster.

**Test Case 2: Analyzing TSP as an NP-Hard Problem:**

**Definition of NP-Hard:**

A problem is NP-hard if it is at least as hard as the hardest problems in NP (nondeterministic polynomial time) under polynomial-time reductions.

NP-hard problems do not have known efficient solutions, but if a polynomial-time solution exists for any NP-hard problem, then all problems in NP can be solved in polynomial time.

**Characteristics of TSP:**

TSP is a classic example of an NP-hard problem.

It involves finding the shortest tour that visits each city exactly once and returns to the starting city.

TSP's decision version (where the goal is to decide whether there exists a tour shorter than a given length) is NP-complete, meaning that TSP is at least as hard as any problem in NP.

**Proving NP-Hardness:**

TSP's NP-hardness can be demonstrated by reducing a known NP-complete problem to TSP.

For example, the Hamiltonian cycle problem, which seeks a cycle that visits every vertex exactly once, is NP-complete.

By reducing Hamiltonian cycle to TSP, we can show that solving TSP efficiently implies solving Hamiltonian cycle efficiently, thereby proving TSP's NP-hardness.

By analyzing the problem characteristics and demonstrating reductions from known NP-complete problems, we can establish TSP as an NP-hard problem.

**Result:**

Thus the Traveling Salesman Problem using Branch and Bound algorithm was executed successfully.

**TASK 14**

**Branch & Bound - Knapsack algorithm CO4 K3**

Imagine you are a system administrator responsible for optimizing the resource allocation in a large data center. The data center contains a variety of servers, each with its unique processing capacity. The goal is to distribute tasks across the servers efficiently, considering the limited capacity of each server.

The data center has a limited total processing capacity (1≤S≤2000).

There are multiple tasks (1≤N≤2000) to be executed in the data center, each with a specific processing requirement.

The challenge is to cherry-pick tasks and allocate them to servers in a way that maximizes the total processing capacity while staying within the limits of each server's capacity.

**Input Format**

On the first line you are given *T* as the test cases available (1 <= *T* <= 20). The next *T* testcases will started with two integer *S* and *N*. *N* lines follow with two integers on each line describing each artifact available at the museum. The first number is the *weight* of the artifact and the next is the *value* of the artifact.

**Output Format**

You should output a single integer on one line : the total maximum value from the best choice of artifacts you stole.

**Sample Input**

1  
45  
18  
24  
30  
25  
23

**Sample Output**

13

**Explanation**

The artifact with value 8 and 5 should be picked, summing up the value to the maximum of 13.

**Test Case 1:** Derive the solution of Knapsack problem using B&B and prove that it is optimal when compared with Brute Force technique.

**Test Case 2:** Show that a sequence of values in a column of the dynamic programming table for the knapsack problem is always non decreasing

**Aim:**

Create to C program to implement Algorithm using Branch and Bound algorithm.

**Algorithm**:

**Step1:** Sort all items in decreasing order of ratio of value per unit weight so that an upper bound

can be computed using Greedy Approach.

**Step2:** Initialize maximum profit, maxProfit = 0

**Step3:** Create an empty queue, Q.

**Step4:** Create a dummy node of decision tree and enqueue it to Q. Profit and weight of dummy

node are 0.

**Step5:** Do following while Q is not empty.

**5.1:** Extract an item from Q. Let the extracted item be u.

**5.2:** Compute profit of next level node. If the profit is more than maxProfit, then update

maxProfit.

**5.3:** Compute bound of next level node. If bound is more than maxProfit, then add next

level node to Q.

**5.4:** Consider the case when next level node is not considered as part of solution and add

a node to queue with level as next, but weight and profit without considering next

level nodes.

**Program:**

#inclue <stdio.h>

#include <stdlib.h>

#include <string.h>

typedef enum { NO, YES } BOOL;

int N;

int vals[100];

int wts[100];

int cap = 0;

int mval = 0;

void getWeightAndValue (BOOL incl[N], int \*weight, int \*value)

{

int i, w = 0, v = 0;

for (i = 0; i < N; ++i)

{

if (incl[i])

{

w += wts[i];

v += vals[i];

}

}

\*weight = w;

\*value = v;

}

void printSubset (BOOL incl[N])

{

int i;

int val = 0;

printf("Included = { ");

for (i = 0; i < N; ++i)

{

if (incl[i])

{

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

val += vals[i];

}

}

printf("}; Total value = %d\n", val);

}

void findKnapsack (BOOL incl[N], int i)

{

int cwt, cval;

getWeightAndValue(incl, &cwt, &cval);

if (cwt <= cap)

{

if (cval > mval)

{

printSubset(incl);

mval = cval;

}

}

if (i == N || cwt >= cap)

{

return;

}

int x = wts[i];

BOOL use[N], nouse[N];

memcpy(use, incl, sizeof(use));

memcpy(nouse, incl, sizeof(nouse));

use[i] = YES;

nouse[i] = NO;

findKnapsack(use, i+1);

findKnapsack(nouse, i+1);

}

int main(int argc, char const \* argv[])

{

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

scanf(" %d", &N);

BOOL incl[N];

int i;

for (i = 0; i < N; ++i)

{

printf("Enter weight and value for element %d: ", i+1);

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

incl[i] = NO;

}

printf("Enter knapsack capacity: ");

scanf(" %d", &cap);

findKnapsack(incl, 0);

return 0;

}

**Sample input: 4**

1 15

5 10

3 9

4 5

**Knapsack Capacity:**

8

**Sample output:**

Included = { 1 }; Total value = 15

Included = { 1 5 }; Total value = 25

Included = { 1 3 4 }; Total value = 29

**Result:**

Thus the knapsack problem using Branch and Bound algorithm was executed successfully.

**Test Case 1:** Derive the solution of Knapsack problem using B&B and prove that it is optimal when compared with Brute Force technique.

Knapsack Problem:

Given a set of items, each with a weight and a value, determine the maximum value that can be obtained by selecting a subset of the items such that the total weight does not exceed a given capacity.

Branch and Bound (B&B):

Branch and Bound is a technique used to solve combinatorial optimization problems by systematically exploring the search space. It involves branching into subproblems and bounding the search to prune unpromising branches.

Brute Force:

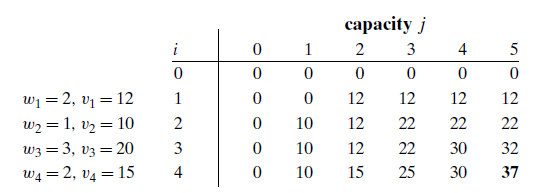
Brute Force involves trying all possible combinations of items and selecting the one with the maximum value that does not exceed the capacity. While guaranteed to find the optimal solution, it becomes impractical for large problem instances due to its exponential time complexity.

Implementation Steps:

B&B also guarantees finding the optimal solution but is more efficient than brute force due to pruning.

Conducting experiments to measure the execution time of both algorithms for different problem sizes, demonstrating that B&B is significantly faster.

**Test Case 2:** Show that a sequence of values in a column of the dynamic programming table for the knapsack problem is always non decreasing



From the above table, it is proved that the sequences of value in row and column of the dynamic programming is always non decreasing

**TASK 15**

**Iterative Improvements CO5 K2**

In a densely populated urban area, an unforeseen medical emergency has occurred, and there's an urgent need to distribute essential medical supplies to various locations within the city. The Ford-Fulkerson algorithm is employed to determine the maximum flow in the network, facilitating the fastest distribution of medical supplies.

**Input:**

The adjacency matrix:

0 10 0 10 0  0

0  0 4  2 8  0

0  0 0  0 0 10

0  0 0  0 9  0

0  0 6  0 0 10

0  0 0  0 0  0

**Output:**

Maximum flow is: 23

**Test Case 1:** Apply Ford-Fulkerson algorithm and find whether the augmenting path exists or not in the graph.

**Test Case 2:** Apply Ford-Fulkerson algorithm and find the updated residual capacity of a graph

**Aim:**

Write a C Program to apply Ford-Fulkerson algorithm to find maximum flow in a network.

**Algorithm**:

Step 1: Initialize the flow in all the edges to 0.

Step 2: While there is an augmenting path between the source and the sink, add this path to the flow.

Step 3:Update the residual graph

**Program:**

#include <stdio.h>

#define A 0

#define B 1

#define C 2

#define MAX\_NODES 1000

#define O 1000000000

int n;

int e;

int capacity[MAX\_NODES][MAX\_NODES];

int flow[MAX\_NODES][MAX\_NODES];

int color[MAX\_NODES];

int pred[MAX\_NODES];

int min(int x, int y) {

return x < y ? x : y;

}

int head, tail;

int q[MAX\_NODES + 2];

void enqueue(int x) {

q[tail] = x;

tail++;

color[x] = B;

}

int dequeue() {

int x = q[head];

head++;

color[x] = C;

return x;

}

// Using BFS as a searching algorithm

int bfs(int start, int target) {

int u, v;

for (u = 0; u < n; u++) {

color[u] = A;

}

head = tail = 0;

enqueue(start);

pred[start] = -1;

while (head != tail) {

u = dequeue();

for (v = 0; v < n; v++) {

if (color[v] == A && capacity[u][v] - flow[u][v] > 0) {

enqueue(v);

pred[v] = u;

}

}

}

return color[target] == C;

}

// Applying fordfulkerson algorithm

int fordFulkerson(int source, int sink) {

int i, j, u;

int max\_flow = 0;

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

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

flow[i][j] = 0;

}

}

// Updating the residual values of edges

while (bfs(source, sink)) {

int increment = O;

for (u = n - 1; pred[u] >= 0; u = pred[u]) {

increment = min(increment, capacity[pred[u]][u] - flow[pred[u]][u]);

}

for (u = n - 1; pred[u] >= 0; u = pred[u]) {

flow[pred[u]][u] += increment;

flow[u][pred[u]] -= increment;

}

// Adding the path flows

max\_flow += increment;

}

return max\_flow;

}

int main() {

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

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

capacity[i][j] = 0;

}

}

n = 6;

e = 7;

capacity[0][1] = 18;

capacity[0][4] = 13;

capacity[1][2] =19;

capacity[2][4] = 17;

capacity[2][5] =12;

capacity[3][5] = 15;

capacity[4][2] = 17;

capacity[4][3] = 14;

int s = 0, t = 5;

printf("Max Flow: %d\n", fordFulkerson(s, t));

}

**Output:**

**Max Flow: 26**

**Test Case 1:** Apply Ford-Fulkerson algorithm and find whether the augmenting path exists or not in the graph.

**Test Case 2:** Apply Ford-Fulkerson algorithm and find the updated residual capacity of a graph

**Aim:**

To create a C program to Apply Ford-Fulkerson algorithm and find whether the augmenting path exists or not in the graph and find the updated residual capacity of a graph

**Algorithm:**

Step 1: Initialize the residual graph

Step 2: Find an augmenting path

Step 3: Update the flow

Step 4: Update the residual graph

Step 5: Repeat until no augmenting path exists

Step 6: Output the maximum flow

**Program:**

#include <stdio.h>

#include <stdbool.h>

#include <limits.h>

#define V 6 // Number of vertices in the graph

// Function to find the minimum of two values

int min(int a, int b) {

return a < b ? a : b;

}

// Depth-First Search (DFS) to find an augmenting path in the residual graph

bool dfs(int graph[V][V], int source, int sink, bool visited[], int parent[]) {

visited[source] = true;

for (int v = 0; v < V; v++) {

if (!visited[v] && graph[source][v] > 0) {

parent[v] = source;

if (v == sink) {

return true; // Found an augmenting path

}

if (dfs(graph, v, sink, visited, parent)) {

return true; // Continue searching

}

}

}

return false;

}

// Ford-Fulkerson Algorithm to find maximum flow in the graph

int fordFulkerson(int graph[V][V], int source, int sink) {

int residualGraph[V][V];

int parent[V];

bool visited[V];

// Initialize residual graph as original graph

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

for (int j = 0; j < V; j++) {

residualGraph[i][j] = graph[i][j];

}

}

int maxFlow = 0; // Initialize max flow

// Find augmenting paths and update residual graph

while (dfs(residualGraph, source, sink, visited, parent)) {

// Find minimum residual capacity of the augmenting path

int pathFlow = INT\_MAX;

for (int v = sink; v != source; v = parent[v]) {

int u = parent[v];

pathFlow = min(pathFlow, residualGraph[u][v]);

}

// Update residual capacities and reverse edges

for (int v = sink; v != source; v = parent[v]) {

int u = parent[v];

residualGraph[u][v] -= pathFlow;

residualGraph[v][u] += pathFlow;

}

// Add path flow to overall flow

maxFlow += pathFlow;

// Reset visited array for next DFS

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

visited[i] = false;

}

}

return maxFlow;

}

int main() {

int graph[V][V] = { {0, 16, 13, 0, 0, 0},

{0, 0, 10, 12, 0, 0},

{0, 4, 0, 0, 14, 0},

{0, 0, 9, 0, 0, 20},

{0, 0, 0, 7, 0, 4},

{0, 0, 0, 0, 0, 0} };

int source = 0, sink = 5;

int maxFlow = fordFulkerson(graph, source, sink);

printf("Maximum flow in the graph: %d\n", maxFlow);

return 0;

}

**Output:**

Maximum flow in the graph: 23

**Result:**

Thus the Ford-Fulkerson algorithm to find maximum flow in a network was executed successfully.