**TASK 12**

**Backtracking - Subset problem**

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.

**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.