**TASK 14**

**Branch & Bound - Knapsack algorithm**

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

A number grid with numbers

AI-generated content may be incorrect.

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