**Experiment No.5: Branch and Bound Algorithm**

**Date:**

**Aim:** To implement the following Algorithms using the Branch and Bound technique:

a) 0/1 Knapsack using Branch and Bound

**Theory:**

* The Branch and Bound algorithm is used to solve optimization problems, particularly combinatorial optimization problems.
* It involves dividing the problem into smaller subproblems and constructing a search tree.
* The algorithm starts with an initial solution and initializes upper and lower bounds.
* The search process begins by exploring the search tree using a depth-first search (DFS) or breadth-first search (BFS) strategy.
* There are primarily three strategies for branch and bound : FIFO ( First In First Out), LIFO ( Last In First Out), and LC (Least Cost).
* At each node of the search tree, the algorithm evaluates the bounds to determine if further exploration is warranted.
* If the lower bound of a node is higher than the current best solution, the subtree rooted at that node is pruned.
* If the upper bound of a node is lower than the current best solution, the algorithm backtracks to the parent node.
* At each node, a decision is made to include or exclude a particular element or constraint from the solution.
* The current solution and the bounds are updated accordingly.
* The algorithm continues exploring the search tree until all nodes have been visited or until the best solution is found.
* The final solution obtained is the optimal solution to the optimization problem.

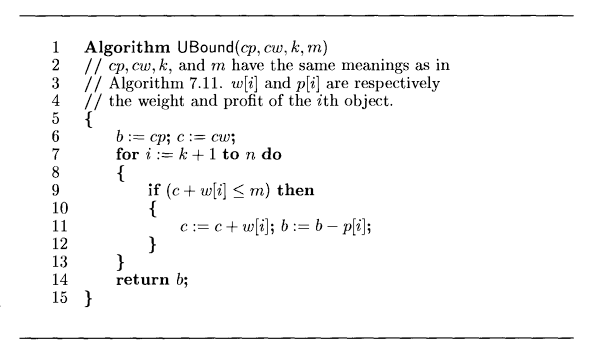
**a)0/1 Knapsack**

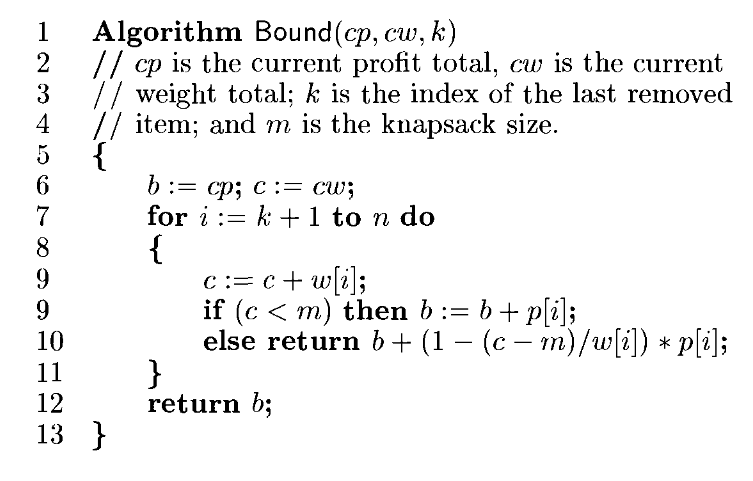
**Date:**

**Problem Statement:** Solve the 0/1 knapsack problem for the following knapsack instance

M = 12, N = 4, P(1 .. 4) = {30,28,20,24}, W(1 .. 4) = {5,7,4,2}

**Algorithm:**





Algorithm BNBKnap(int k, int cp, int cw)

{

c = Bound(k, cp, cw);

u = UBound(k, cp, cw);

if (u < upper && cw <= m)

upper = u; //Update global upper bound

// Left child

if (c <= upper)

{

y[k] = 1;

if (k < n - 1)

{

if (cw <= m)

BNBKnap(k + 1, cp + K[k].p, cw + K[k].w); //Recursively go to left child

}

if (k == n - 1)

{

if (cw + K[k].w <= m)

{

fp = cp + K[k].p;

fw = cw + K[k].w;

Print final profit;

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

x[j] = y[j];

x[k] = 1;

Print updated array;

}

}

}

// Right child

if (c <= upper)

{

y[k] = 0;

if (k < n - 1)

{

BNBKnap(k + 1, cp, cw);

}

if (k == n - 1)

{

//Calculate new bounds for right child

c = Bound(k, cp - K[k].p, cw - K[k].w);

u = UBound(k, cp - K[k].p, cw - K[k].w);

if (cw <= m && c <= upper)

{

fp = cp;

fw = cw;

Print final profit;

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

x[j] = y[j];

x[k] = 0;

Print updated array;

}

}

}

}

**Time complexity:**

O(2n + n2)

**Space complexity:**

O(n)

**Code:**

#include <stdio.h>

#include <math.h>

#define MAX 100

struct knap

{

int p;

int w;

float rat;

};

int m, n, x[MAX], y[MAX], z[MAX], fp, fw, upper = \_\_INT32\_MAX\_\_;

int Bound(int k, int cp, int cw, struct knap K[])

{

int b = cp, c = cw;

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

{

c += K[i].w;

if (c < m)

b += K[i].p;

else

{

return (-1) \* (b + (int)ceil((1 - (c - m) / (float)K[i].w) \* K[i].p));

}

}

return -b;

}

int UBound(int k, int cp, int cw, struct knap K[])

{

int b = cp, c = cw;

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

{

if (c + K[i].w <= m)

{

c += K[i].w;

b += K[i].p;

}

}

return -b;

}

void BNBKnap(int k, int cp, int cw, struct knap K[])

{

int c = Bound(k, cp, cw, K);

int u = UBound(k, cp, cw, K);

printf("%d %d \n", c, u);

if (u < upper && cw <= m)

upper = u;

// Left child

if (c <= upper)

{

y[k] = 1;

if (k < n - 1)

{

printf("Left Child of K:%d CP:%d CW:%d\n", k + 1, cp, cw);

if (cw <= m)

BNBKnap(k + 1, cp + K[k].p, cw + K[k].w, K);

}

if (k == n - 1)

{

printf("Left Child of K:%d P:%d W:%d\n", k + 1, cp, cw);

printf("%d %d \n", c, u);

if (cw + K[k].w <= m)

{

fp = cp + K[k].p;

fw = cw + K[k].w;

printf("New cost is %d.\n", fp);

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

x[j] = y[j];

x[k] = 1;

printf("Array updated.\n");

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

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

printf("\n");

}

}

}

// Right child

if (c <= upper)

{

y[k] = 0;

if (k < n - 1)

{

printf("Right Child of K:%d P:%d W:%d.\n", k + 1, cp, cw);

BNBKnap(k + 1, cp, cw, K);

}

if (k == n - 1)

{

printf("Right Child of K:%d P:%d W:%d\n", k + 1, cp, cw);

int c = Bound(k, cp - K[k].p, cw - K[k].w, K);

int u = UBound(k, cp - K[k].p, cw - K[k].w, K);

printf("%d %d \n", c, u);

if (cw <= m && c <= upper)

{

fp = cp;

fw = cw;

printf("New cost is %d.\n", fp);

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

x[j] = y[j];

x[k] = 0;

printf("Array updated.\n");

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

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

printf("\n");

}

}

}

}

void sort(struct knap K[])

{

struct knap key;

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

{

key = K[i];

int j = i - 1;

while (j >= 0 && K[j].rat < key.rat)

{

K[j + 1] = K[j];

j = j - 1;

}

K[j + 1] = key;

}

}

int main()

{

printf("Enter number of knapsack values and size of knapsack.\n");

scanf("%d%d", &n, &m);

struct knap K[n + 1];

int p\_values[MAX], w\_values[MAX];

printf("Enter profits of knapsack.\n");

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

scanf("%d", &p\_values[i]);

printf("Enter weights of knapsack.\n");

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

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

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

{

K[i].p = p\_values[i];

K[i].w = w\_values[i];

K[i].rat = (float)K[i].p / K[i].w;

}

sort(K);

fp = -1;

BNBKnap(0, 0, 0, K);

printf("The final array is:\n");

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

{

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

{

if (p\_values[i] == K[j].p && w\_values[i] == K[j].w)

{

if (x[j])

z[i] = 1;

else

z[i] = 0;

}

}

}

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

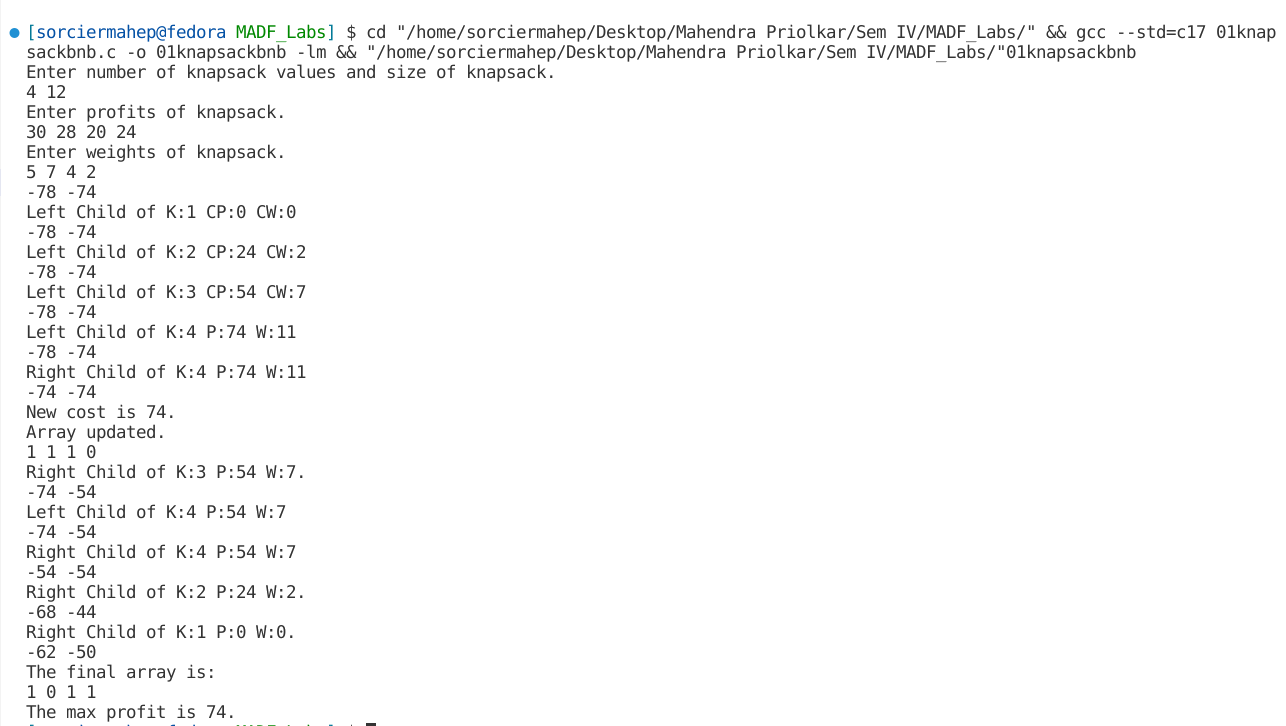
printf("%d ", z[i]);

printf("\nThe max profit is %d.\n", fp);

return 0;

}

**Output:**



**Conclusion:**

Optimization problems were studied and implemented using the Branch and Bound Programming Algorithm. 0/1 Knapsack Problem was implemented using the Branch and Bound Approach.