

### Experiment 3:

Write a program to solve a 0-1 Knapsack problem using dynamic programming or branch and bound strategy.

#### Code:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <queue>

using namespace std;

class Item
{
public:
    int value;
    int weight;
    double ratio;

    Item(int value, int weight)
    {
        this->value = value;
        this->weight = weight;
        this->ratio = (double)value / weight;
    }
};

class KnapsackNode
{
public:
    vector<int> items;
    int value;
    int weight;

    KnapsackNode(vector<int> items, int value, int weight)
    {
        this->items = items;
        this->value = value;
        this->weight = weight;
    }
};
```

```

class Knapsack
{
public:
    int maxWeight;
    vector<Item> items;

    Knapsack(int maxWeight, vector<Item> items)
    {
        this->maxWeight = maxWeight;
        this->items = items;
    }

    int solve()
    {
        sort(this->items.begin(), this->items.end(), [](const Item& a, const Item& b)
        {
            return a.ratio > b.ratio;
        });

        int bestValue = 0;
        queue<KnapsackNode> q;
        q.push(KnapsackNode({}, 0, 0));

        while (!q.empty())
        {
            KnapsackNode node = q.front();
            q.pop();
            int i = node.items.size();

            if (i == this->items.size())
            {
                bestValue = max(bestValue, node.value);
            }
            else
            {
                Item item = this->items[i];
                KnapsackNode withItem(node.items, node.value + item.value,
node.weight + item.weight);
                if (isPromising(withItem, this->maxWeight, bestValue))
                {
                    q.push(withItem);
                }
                KnapsackNode withoutItem(node.items, node.value,
node.weight);
            }
        }
    }
}

```

```

        if (isPromising(withoutItem, this->maxWeight, bestValue))
        {
            q.push(withoutItem);
        }
    }
}

return bestValue;
}

bool isPromising(KnapsackNode node, int maxWeight, int bestValue)
{
    return node.weight <= maxWeight && node.value + getBound(node) >
bestValue;
}

int getBound(KnapsackNode node)
{
    int remainingWeight = this->maxWeight - node.weight;
    int bound = node.value;

    for (int i = node.items.size(); i < this->items.size(); i++)
    {
        Item item = this->items[i];

        if (remainingWeight >= item.weight)
        {
            bound += item.value;
            remainingWeight -= item.weight;
        }
        else
        {
            bound += remainingWeight * item.ratio;
            break;
        }
    }

    return bound;
}

};

```

### Output:

Best value: 220