Knapsack DP

Problem Statement

Given weights and values of n items, put these items in a knapsack of capacity W to get the maximum total value in the knapsack.

0-1 Knapsack Problem

value[] = {60, 100, 120};
weight[] = {10, 20, 30};
W = 50;

Solution: 220

Weight = 10; Value = 60;
Weight = 20; Value = 100;
Weight = 30; Value = 120;
Weight = (20+10); Value = (100+60);
Weight = (30+10); Value = (120+60);
Weight = (30+20); Value = (120+100);
Weight = (30+20+10) > 50

Types of Knapsack

- 0-1 knapsack problem
  - Items are indivisible; you either take an item or not
- Fractional knapsack problem
  - Items are divisible: you can take any fraction of an item

Fractional knapsack

Items as (value, weight) pairs
arr[] = {{60, 10}, {100, 20}, {120, 30}}
Knapsack Capacity, W = 50;

Maximum possible value = 240
by taking items of weight 10 and 20 kg and 2/3 fraction
of 30 kg. Hence total price will be 60+100+(2/3)(120) = 240

Fractional Knapsack Solution

- An efficient solution is to use **Greedy approach**. The basic idea of the greedy approach is to calculate the ratio value/weight for each item and sort the item on basis of this ratio. Then take the item with the highest ratio and add them until we can't add the next item as a whole and at the end add the next item as much as we can. Which will always be the optimal solution to this problem.
- Complexity: ??

Code of Fractional Knapsack

```
ll fracknapsack()
{
    ll n,sum=0,capacity;
    cin>>n>>capacity;
    ll weight[n+5],value[n+5];
    vector< pair<double,ll> >v;
    for(int i=0;i<n;i++)
        cin>>weight[i];
    for(int i=0;i<n;i++)
    {
        cin>>value[i];
        v.push_back( make_pair((value[i]*1.0)/weight[i],i));
    }
    //sort ascending order of value/weight ratio
    sort(v.rbegin(),v.rend());
    for(int i=0;i<n;i++)
    {
        //pick whole item if its total weight is less than current capacity
        if(weight[v[i].second]<=capacity){
            capacity-=weight[v[i].second];
            sum+=value[v[i].second];
        }
        else
        {
             sum+=(capacity*1.0/weight[v[i].second])*value[v[i].second];
             // Break since capacity become 0
             break;
        }
    }
    cout<<sum<<"\n";
    return sum;
```

0-1 Knapsack

Example 1:

val[] = {60, 100, 120};

wt[] = {10, 20, 30};

W = 50;

Output : 220 //maximum value that can be obtained 30 20 weights 20 and 30 are included.

Example 2:

val[] = {40, 100, 50, 60};

wt[] = {20, 10, 40, 30};

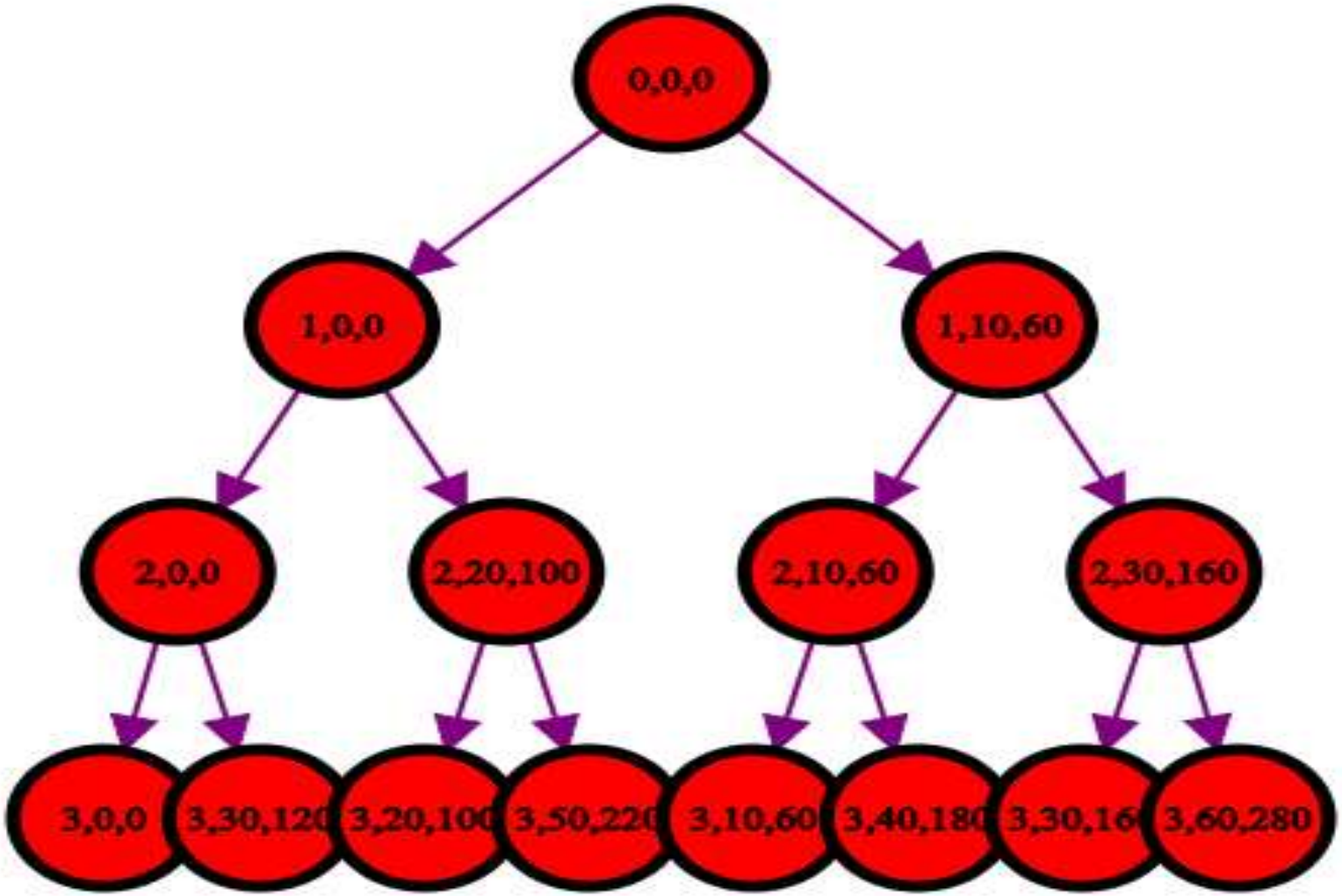
W = 60;

Output : 200

// Pick item with weights 30,20,10

0-1 Knapsack Solution

- Recursion
  - A simple solution is to consider all subsets of items and calculate the total weight and value of all subsets.
  - Complexity : ???
- Dynamic Programming
  - We will make 2 state DP . The state DP[i][j] will denote maximum value that can achieve using knapsack capacity ‘j-weight’ considering all values from ‘1 to ith’. So for every ith element we have two possibilities that can take place either
    - Choose the ith item
    - Or not choose the ith item
  - Complexity : ???


0,0,0
1,0,0
1,10,60
2,0,0
2,20,100
2,10,60
2,30,160
3,0,0
3,30,120
3,20,100
3,50,220
3,10,60
3,40,180
3,30,16
3,60,280

Code of 0-1 Knapsack

```
ll dp[105][1005],weight[105],value[105];

ll dfs(ll n,ll capacity,ll x){
    if(n==x)
        return 0;
    if(dp[n][capacity]!=-1)
        return dp[n][capacity];
    //capacity is more than weight of item then we have two possibilites pick element or not pick element
    if(capacity>=weight[n])
        return dp[n][capacity]=max(value[n]+dfs(n+1,capacity-weight[n],x),dfs(n+1,capacity,x));
    return dp[n][capacity]=dfs(n+1,capacity,x);
}

ll knapsackDP()
{
    ll n,capacity;
    cin>>n>>capacity;
    memset(dp,-1,sizeof(dp));
    for(int i=0;i<n;i++)
        cin>>weight[i];
    for(int i=0;i<n;i++)
        cin>>value[i];
    ll ans=dfs(0,capacity,n);
    cout<<ans<<"\n";
    return ans;
}
```

Thank You

Made by Gaurav Katare