The best programs are written so that computing machines can perform them quickly and so that human beings can understand them clearly. A programmer is ideally an essayist who works with traditional aesthetic and literary forms as well as mathematical concepts, to communicate the way that an algorithm works and to convince a reader that the results will be correct. — **Donald E. Knuth**

Algorithms

Dynamic Programming

Problem :-

Unbounded Knapsack problem.
Given a set of **'n'** items having weights **{ W_1,W_2,W_3,.....,W_n }** & values **{ V_1,V_2,V_3,.....,V_n }** and a **Knapsack** of weight **W**, find the maximum value that can be accommodated using 1 or more instances of given weights.
So, the aim is to maximize the value of picked up items such that sum of the weights is less than or equal to **W** (Knapsack Weight).
Consider **3** items having weights **{ 2, 3, 5 }** & values **{ 50, 100, 140 }**.
We have a **Knapsack** of weight **17**. We can use one or more instances of any items to fill the knapsack such that the value that can be accommodated in the knapsack is maximized.
We see that, if we pick up 5 instances of weight **3** & 1 instance of weight **2**,
we can accommodate a total value of **550** which is the maximum value that can be collected using the given weights, keeping in mind the constraint that knapsack weight is **17**.

Solution :-

We will maintain an array **knapsack [W + 1]** where **' j '** index stores the maximum value that can be fitted in a knapsack of capacity **' j '**.
Then, **knapsack [W]** gives the maximum value that can fitted in the knapsack of given weight i.e **' W '**.
Let us find the structure of the optimal solution.
Let **knapsack (j)** denotes the maximum value that can be fitted in a knapsack of weight **' j '**.
While computing **knapsack (j)**, we need to decide whether to select or reject an instance of a weight **' i '**.
If we reject all the weights, then **knapsack (j) = knapsack (j - 1)** and
If we select an instance of weight **' i '**, then
knapsack (j) = maximum [knapsack (j - W (i)) + V (i) for i = 0, ... , n - 1]
Thus, the structure of the optimal solution is defined recursively as :
knapsack [j] = max (knapsack [j - 1] , { knapsack [j - w [i]] + v [i] for i = 0...n-1 })
Finally, **knapsack [W]** gives the required solution. In the implementation below, we maintain some additional information to print the **selected weights and their no. of instances** apart from the maximum value that can be accommodated in the given knapsack.

```
#include<iostream>
using namespace std;

// print out the actual weights and their no. of instances used
// to fill the knapsack
void instancesUsed(int items[],int weights[],int capacity,int size)
   int k = capacity;
   int instances[size];
   int i;
   for(i=0;i<size;i++)
      instances[i] = 0;
   // compute the no. of instances used for each selected item(weig
   while(k >= 0) {
      int x = items[k];
      if(x == -1) break;
      instances[x] += 1;
      k -= weights[items[k]];
   }
   cout<<"\nInstances used :-\n";
   for(i=0;i<size;i++)
```

```
21        cout<<weights[i]<<" "<<instances[i]<<endl;
22    }
23
24    /* Given n items, each having weight w(i) and value v(i) and a
25       knapsack of weight 'W', find the maximum value that can be
26       accommodated using 1 or more instances of given weights.
27       Suppose knapsack[j] -> max value that can be fitted in a knapsac
28                              of weight 'j'
29       Then, knapsack[W] -> required answer
30       Standard Recursive solution :-
31       knapsack[j] = max(knapsack[j-1], {knapsack[j-w(i)]+v(i) for i =
32    */
33    int findMaxValue(int weight[],int values[],int items[],int n,int ca
34       // temporary array where index 'j' denotes max value that can be
35       // in a knapsack of weight 'j'
36       int knapsack[capacity+1];
37       knapsack[0] = 0;
38       items[0] = -1;
39       int i,j;
40       for(j=1;j<=capacity;j++) {
41          items[j] = items[j-1];
42          // as per our recursive formula,
43          // iterate over all weights w(0)...w(n-1)
44          // and find max value that can be fitted in knapsack of weigh
45          int max = knapsack[j-1];
46          for(i=0;i<n;i++) {
47             int x = j-weight[i];
48             if(x >= 0 && (knapsack[x] + values[i]) > max) {
49                max = knapsack[x] + values[i];
50                items[j] = i;
51             }
52             knapsack[j] = max;
53          }
54       }
55       return knapsack[capacity];
56    }
57
58    // main
59    int main() {
60       int weight[] = {2,3,5};
61       int values[] = {50,100,140};
62       int capacity = 17; // capacity of the knapsack
63       int size = sizeof(weight)/sizeof(weight[0]);
64       // stores the items and the no. of instances of each item
65       // used to fill the knapsack
66       int items[capacity+1];
67       int val = findMaxValue(weight,values,items,size,capacity);
68       cout<<"\nMaximum value that can be fitted :: "<<val;
69       instancesUsed(items,weight,capacity,size);
70       cout<<endl;
71       return 0;
72    }
```

Compile & Run

Back | Next

All problems on Dynamic Programming

Privacy Policy

csegeek.com does not use cookies to track any user information. No personal information is collected. However, users can optionally register for the email newsletter to remain updated with the website content. This website displays ads via ad-networking mediums which might contain cookies. This website may contain links to other sites and is not responsible for the contents or privacy policies of such websites.