Knapsack

0-1 knapsack

Given a set of items with a weight and value. Determine the max number of items to include in the knapsack, so that the total weight≤ given capacity.

The values can be 0 or 1 NOT fractional in 0-1 Knapsack. The items I'm carrying cannot be divisible or breakable.

There are 2 cases

1. **Case 1:** The item is included in the optimal subset.
2. **Case 2:** The item is not included in the optimal set.

Therefore, the maximum value that can be obtained from ‘n’ items is the max of the following two values.

1. Maximum value obtained by n-1 items and W weight (excluding nth item). **a[i-1][j]**
2. Value of nth item plus maximum value obtained by n-1 items and W minus the weight of the nth item (including nth item). **a[i-1][j-price[i-1]]+profit[i-1]**

If the weight of ‘nth’ item is greater than ‘W’, then the nth item cannot be included and **Case 1** is the only possibility.

- **Time Complexity:** O(N*W). where ‘N’ is the number of weight element and ‘W’ is capacity. As for every weight element we traverse through all weight capacities 1<=w<=W.
- **Auxiliary Space:** O(N*W). The use of 2-D array of size ‘N*W’.

```
package knapsack;
import java.util.Scanner;
public class dp {
public static void main(String[] args) {
  Scanner sc=new Scanner(System.in);
  int budget=sc.nextInt();
  int nbofitems=sc.nextInt();
  int p[]=new int [nbofitems];
  int of[]=new int[nbofitems];
```

```
  for(int i=0;i<of.length;i++)
    of[i]=sc.nextInt();
  for(int i=0;i<p.length;i++)
    p[i]=sc.nextInt();
  System.out.println(ks(budget,nbofitems,p,of));
}

static int ks(int w, int n, int[]price,int[]profit) {
  int [][]a=new int[n+1][w+1];
  for(int i=0; i<=n; i++)
    a[i][0]=0;
  for(int i=0;i<=w;i++)
    a[0][i]=0;
  for(int i=1;i<=n;i++) {
    for(int j=1;j<=w;j++) {
      if(price[i-1]<=j)
        a[i][j]=Math.max(a[i-1][j-price[i-1]]+profit[i-1], a[i-1][j]);
      else
        a[i][j]=a[i-1][j];
    }
  }
  return a[n][w];
  }
}
```

Fractional Knapsack

YouTube: https://www.youtube.com/watch?v=oTTzNMHM05I

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

In the 0-1 Knapsack we are not allowed to break items. We either take the whole item or don’t take it.

In **Fractional Knapsack**, we can break items for maximizing the total value of knapsack. This problem in which we can break an item is also called the fractional knapsack problem.

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.

Time complexity: O(nlogn)

Example:

objects:	1	2	3	4	5	6	7
profit:	10	5	15	7	6	18	3
weight:	2	3	5	7	1	4	1
p/w:	5	1.3	3	1	6	4.5	3

number of items, n=7

budget, capacity, m=15

constraint: $\Sigma w_i x_i \leq m$

Objective: max=$\Sigma p_i x_i$

Since we want the maximum value of the objects in a knapsack, then it is an optimization problem. Since the constraints is weight being less than or equal to the total capacity, it can be solved using the greedy approach.

We can take the objects with the most profit first, or we can take the objects with smaller weight as to gain more items, therefore more profit.

It is better to take the item with that makes the **MOST PROFIT PER WEIGHT UNIT. p/w.**

Steps:

Step-01:

For each item, compute its value / weight ratio.

Step-02:

Arrange all the items in decreasing order of their value / weight ratio.

Step-03:

Start putting the items into the knapsack beginning from the item with the highest ratio.

Put as many items as you can into the knapsack.

CODE:

```
package knapsack;
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;
public class FractionalKnapSack {
```

```
public static void main(String[] args) {
Scanner sc=new Scanner(System.in);
int n=sc.nextInt();
int w=sc.nextInt();
int []wt=new int[n];
for(int i=0;i<n;i++)
wt[i]=sc.nextInt();
int[] val=new int[n];
for(int i=0;i<n;i++)
val[i]=sc.nextInt();
double maxValue = getMaxValue(wt, val, w);
System.out.println(maxValue);

}

private static double getMaxValue(int[] wt,int[] val, int capacity) {
    ItemValue[] value = new ItemValue[wt.length];

    for(int i = 0; i < wt.length; i++)
        value[i] = new ItemValue(wt[i], val[i], i);

    Arrays.sort(value, new Comparator<ItemValue>() {
        public int compare(ItemValue o1, ItemValue o2)  {
            return o2.cost.compareTo(o1.cost) ;
        }
    });

    double totalValue = 0.0;

    for(ItemValue i: value){
      int curWt = (int) i.wt;
        int curVal = (int) i.val;
        if (capacity - curWt >= 0) {
            capacity = capacity-curWt;
            totalValue += curVal;
            }
        else{
          double fraction = ((double)capacity/(double)curWt);
            totalValue += (curVal*fraction);
            capacity = (int)(capacity - (curWt*fraction));
            break;
        }
        }
    return totalValue;
}

static class ItemValue{
    Double cost;
    double wt, val, ind;

    public ItemValue(int wt, int val, int ind) {
      this.wt = wt;
        this.val = val;
        this.ind = ind;
        cost=new Double(val/wt);
    }
}
}
```