Greedy Algorithms

Fractional Knapsack

wt[]: An array containing weight against each item.
value[]: An array containing the market value of each item.
items[]: An array containing the item numbers.
N: Total number of available items.
M: Maximum capacity of Knapsack.
Goal: maximize the profit.
Difference between binary (0/1) and fractional Knapsack
M = 10kg
W[] = {5, 3, 4}
V[] = {50, 40, 20}
In binary knapsack you have only two options i.e., either to pick the item completely or drop the item, whereas in fractional Knapsack if you are unable to pick the complete item then you can include the fractional part of that item as well.
e.g., Considering the above sample example, if we include the items having weights 5kg and 3kg then we are left with the available capacity of 2Kg. So, according to binary Knapsack you are left with no other option but to drop the item having 4kg weight, but fractional Knapsack provides the freedom to include the fractional part of the item having 4Kg weight.
Q: What fraction we can include?
Ans: (available capacity / weight of item) => M/w
Q: how much profit we can earn by including this fractional part?
Ans: (available capacity/weight of item) * value of item => (M/w)*v

M = 15Kg
W[]={10, 5, 2, 7, 4}
V[]={50, 12, 30, 14, 8}
In greedy algorithms you need a greedy choice for decision making.
Greedy Choices:
1) Choose the items by prioritizing those with maximum value.
Greedy Sol:
Include the items having value $50 and $30 completely and fractional part of the item having $14 value.
Explanation: After including the above-mentioned two items completely we are left with 3Kg weight. so profit of fractional part will be (3/7)*14 = $6
50+30+6 = $86
Optimal Sol:
If we include the fractional part of the item having $12 we could have maximize the profit. (3/5)*12 = $7.2
50+30+7.2 = $87.2 (Since we get a maximum score without following the greedy approach So this greedy choice will fail to provide an optimal solution.

2) Choose the items by prioritizing those with minimum weight.
Provide a counter example to show that this greedy choice will also fail in providing the optimal solution. You can change the data values of the array.

Optimal Greedy Choices:
To maximize the profit we need to make sure that both the above mentioned greedy choices are fulfilled at the same time. i.e., item must have max value and min weight so we need to calculate **val to wt ratio**.
a) Choose the items by prioritizing those with maximum value to weight ratio.
b) Choose the items by prioritizing those with minimum weight to value ratio.

```
FKSP (items[], wt[], value[], N, M)
{
//create a new array "V_W_Ratio" of type double to store the ratio of (value/wt)
   V_W_Ratio[1...N]
   For(i=1 to N)
   {
      V_W_Ratio[i] = value[i]/wt[i]
   }
//Now sort the data of all the arrays in descending order on the base of (value/wt) ratio.
//Why? So that the items having maximum ratios are shifted towards the start of array and we can select the items easily without any search operation.
   Sort(items, wt, value, V_W_Ratio, N) // sort in descending order
//Now calculate the profit and keep track of selected items providing maximum profit.
   profit = 0
   vector<int> Selected_Items
   For(i=1 to N)
   {
      if(wt[i] <= M) {
//if the available capacity "M" is greater than the current item's weight "wt[i]" then select the current item completely.
         M = M - wt[i]   //update remaining capacity
         profit += value[i]  //update profit
         Selected_Items.push_back(items[i])  //keep the tract of Selected Item number
      }
      else
      {
//if the available capacity "M" is less than the current item's weight "wt[i]" then select the fractional part of current item.
         profit += (M/wt[i]) *value[i];
//After selecting the fractional part of this item, the remaining capacity "M" will become zero so there is no need to iterate further.
         break;
      }
   }
   return (profit, Selected_Items)
}
```

//Time Complexity: O(N*logN) //since we are sorting the arrays.
for optimal choice (b) i.e., "Choose the items by prioritizing those with minimum weight to value ratio" you need to calculate w/v ratio and then sort all the arrays in the ascending order on the base of v/w ratio. All the remaining code will be the same as mentioned above.