Greedy Algorithms (Ch. 16)

A greedy algorithm obtains an optimal solution to a problem by making a sequence of choices. At each decision point, the algorithm chooses the locally optimal solution. In other words, when we are considering which choice to make, we make the choice that looks best in the current situation, *without considering results from subproblems*.

Problems solved optimally by greedy algorithms share some features with those problems solved optimally by dynamic programming algorithms: optimal substructure, recursive solutions.

Today we look at two variations of the same problem, one with a greedy solution and one for which we can provide a counterexample to the existence of a greedy solution.

0-1 Knapsack problem

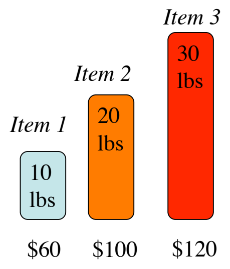
Given items T = $T_1, T_2, T_3, \ldots, T_n$ (items you might want to carry in your backpack) with associated weights $w_1, w_2, \ldots, w_n$ and benefit values $b_1, b_2, \ldots, b_n$, how can we maximize the total benefit considering that we are subject to an absolute weight limit W?

A brute-force solution to this problem is to enumerate all possible subsets of T and select the one with the highest total benefit from among all the subsets whose cumulative weight is ≤ W.

Unfortunately, the running time of this approach is:

0-1 Knapsack Problem

This is called the 0-1 knapsack problem because each item must be taken in its entirety. If we use a greedy strategy to solve this problem, we would take the objects in order of their benefit/weight value.

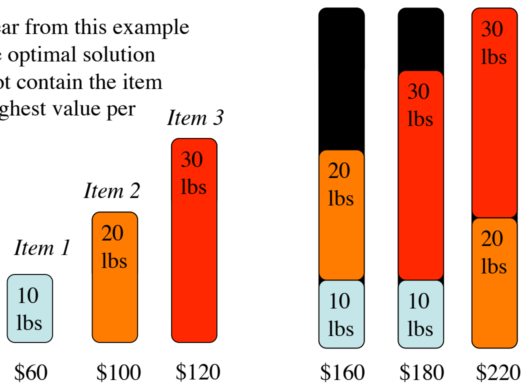


The cost per pound of:
Item 1 = \$60/10 lbs = \$6/lb
Item 2 = \$100/20 lb = \$5/lb
Item 3 = \$120/30 lb = \$4/lb

So how about if we add these items to our backpack in order of value? Suppose our backpack can only hold 50 lbs

0-1 Knapsack Problem

It is clear from this example that the optimal solution does not contain the item with highest value per pound.



Algorithm 0-1 Knapsack

Input: Set T of n items, such that item i has positive benefit b_i and positive integer weight w_i and maximum total weight W. (assumes there is a global array B of length W+1 and that each element of B has an associated list that is initially empty).

Output: For w = 0, …, W, maximum benefit B[w] of a subset of T with total weight w.

```
0-1Knapsack(T,W)
 1. for i = 0 to W do
 2.     B[i] = 0
 3. for k = 1 to |T| do
 4.     for w = W downto $w_k$ do
 5.         if B[w – $w_k$] + $b_k$ > B[w] then
 6.             B[w] = B[w – $w_k$] + $b_k$
 7.             list[w] = list [w – $w_k$] +k
```

Complexity?

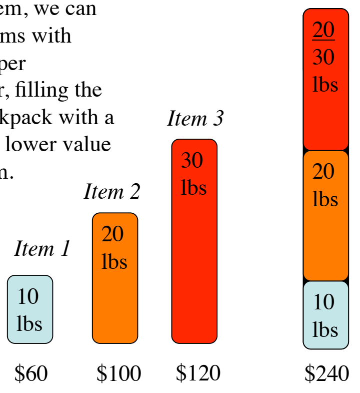
Fractional Knapsack problem

Given items T = $T_1, T_2, T_3, \ldots, T_n$ (items you might want to carry in your backpack) with associated weights $w_1, w_2, \ldots, w_n$ and benefit values $b_1, b_2, \ldots, b_n$, how can we maximize the total benefit considering that we are subject to an absolute weight limit W?

This is called the fractional knapsack problem because any fraction of each item can be used. Using a greedy strategy to solve this problem, we pack the items in order of their benefit/weight value.

Fractional Knapsack Problem

For this problem, we can include the items with highest value per pound in order, filling the rest of the backpack with a fraction of the lower value per pound item.



Notice that the overall benefit is higher than that achieved with the 0/1 version of the problem

Algorithm Fractional Knapsack

Input: Set T of n items, such that item i has positive benefit b_i and positive integer weight w_i and maximum total weight W. Assumes we have an array $v_1, v_2, \ldots, v_n$ to hold the benefit/weight ratio, and an array $x_1, x_2, \ldots, x_n$ to hold the amount of each item.

Output: Amount x_i of each item that maximizes the total benefit while not exceeding the maximum total weight W.

```
FractionalKnapsack(T,W)
1. for i = 1 to T do
2.     x_i = 0
3.     v_i = b_i/w_i      // value index of item}
4. w = 0
5. while w < W do
6.     remove from T item i with highest v_i
7.     a = min {w_i, W-w}
8.     x_i = a
9.     w = w + a
```

Complexity?