Greedy Algorithms

- Optimization problem: there can be many possible solution. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value
- Greedy algorithm: an algorithmic technique to solve optimization problems
  - Always makes the choice that looks best at the moment.
  - Makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

When to be Greedy

- Greedy algorithms apply to problems with:
  - The greedy choice property: an optimal solution can be obtain by making the greedy choice at each step.
  - Optimal substructures: optimal solutions contain optimal subsolutions.
  - Many optimal solutions, but we only need one such solution.
- Unlike dynamic programming, we do not need to know the solutions to the sub-problems to make choices.
  - Hence, greedy algorithms are often more efficient than dynamic programming.
- Possible drawback:
  - Actions with a small short-term cost may lead to a situation, where further large costs are unavoidable.

Knapsack Problems

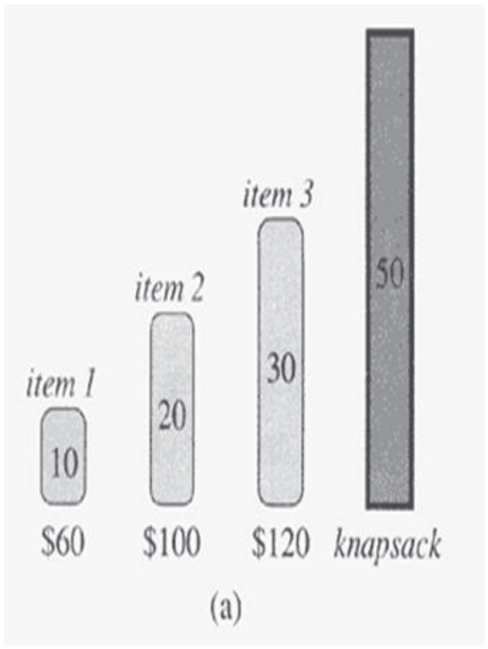
Knapsack Problems

- You want to pack n items in your knapsack
  - The ith item is worth v_i and weighs w_i
  - Take a valuable a load as possible, but cannot exceed weight W.
  - v_i , w_i, W are integers.
- 0-1 knapsack → each item is taken or not taken
- Fractional knapsack → fractions of items can be taken
- Both exhibit the optimal-substructure property
  - 0-1: consider a optimal solution. If item j is removed from the load, the remaining load must be the most valuable load weighing at most W-w_j
  - Fractional: If w of item j is removed from the optimal load, the remaining load must be the most valuable load weighing at most W-w that can be taken from other n-1 items plus $w_j - w$ of item j

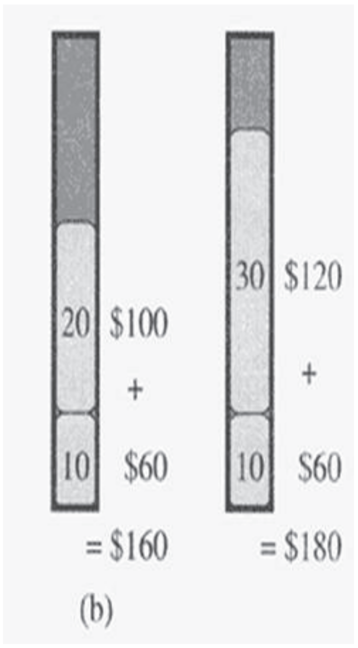
Greedy 0-1 Knapsack Alg?

- 3 items:
  - item 1 weighs 10 lbs, worth $60 ($6/lb)
  - item 2 weighs 20 lbs, worth $100 ($5/lb)
  - item 3 weighs 30 lbs, worth $120 ($4/lb)
- knapsack can hold 50 lbs
- greedy strategy:
  - take item 1
  - take item 2
  - no room for item 3

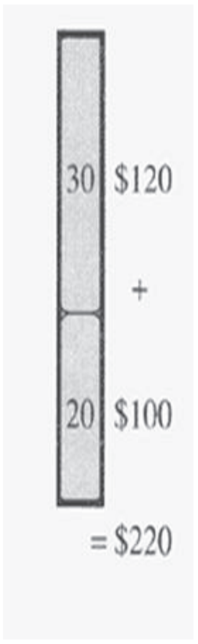
Greedy strategy does not work for 0-1 Knapsack



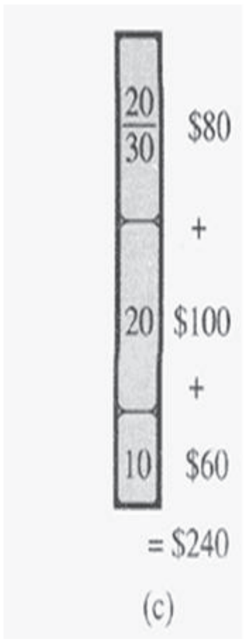
$6/lb $5/lb $4/lb



Taking item 1 is a big mistake globally although looks good locally



Optimal sol. for 0-1 Knapsack



Optimal sol. for fractional Knapsack

Fractional Knapsack Problem

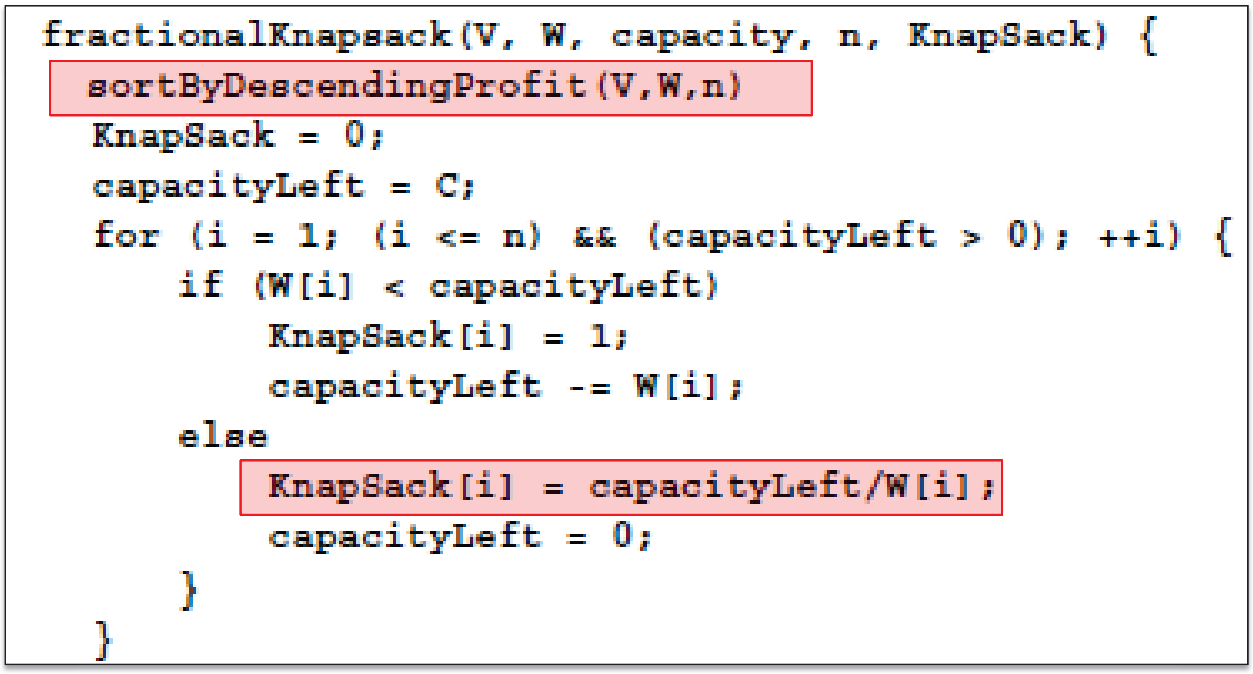
- Given a set of n objects where object i has value v_i and weight w_i and a knapsack capacity C, determine the fractional amount f_i of each object i to be included in the knapsack such that the profit is maximized while the weight of the included objects does not exceed the knapsack capacity.

  Maximize $\sum v_i f_i$
  - such that $\sum w_i f_i \leq C$
  - where $0 \leq f_i \leq 1$

Greedy Strategy

- Fractional knapsack can be solvable by the greedy strategy
  - Compute the value per pound v_i/w_i for each item
  - Obeying a greedy strategy, we take as much as possible of the item with the greatest value per unit wt.
  - If the supply of that item is exhausted and we can still carry more, we take as much as possible of the item with the next value per unit wt, and so forth until we cannot carry any more
  - O(n *lg* n) (we need to sort the items by value per pound)
  - Algorithm ?

Greedy Algorithm for Fractional Knapsack



```
fractionalKnapsack(V, W, capacity, n, KnapSack) {
  sortByDescendingProfit(V,W,n)
  KnapSack = 0;
  capacityLeft = C;
  for (i = 1; (i <= n) && (capacityLeft > 0); ++i) {
      if (W[i] < capacityLeft)
          KnapSack[i] = 1;
          capacityLeft -= W[i];
      else
          KnapSack[i] = capacityLeft/W[i];
          capacityLeft = 0;
      }
  }
```

Example

i	1	2	3	4	5	6	7	8
v_i	12	4	5	3	8	8	12	1
w_i	4	3	5	6	1	4	10	4
p_i	3	1.33	1	0.5	8	2	1.2	.25

C=15

Calculate $p_i=v_i/w_i$ for all given items

```
fractionalKnapsack(V, W, capacity, n, KnapSack) {
  sortByDescendingProfit(V,W,n)
  KnapSack = 0;
  capacityLeft = C;
  for (i = 1; (i <= n) && (capacityLeft > 0); ++i) {
      if (W[i] < capacityLeft)
          KnapSack[i] = 1;
          capacityLeft -= W[i];
      else
          KnapSack[i] = capacityLeft/W[i];
          capacityLeft = 0;
      }
  }
```

i	5	1	6	2	7	3	4	8
v_i	8	12	8	4	12	5	3	1
w_i	1	4	4	3	10	5	6	4
p_i	8	3	2	1.33	1.2	1	0.5	.25

After sorting the items w.r.t, $p_i=v_i/w_i$

i	6	1	6	2	7	3	4	8
v_i	8	12	8	4	12	5	3	1
w_i	1	4	4	3	3/10	5	6	4
p_i	8	3	2	1.33	1.2	1	0.5	.25

$\sum v_i f_i = 8\times1+12\times1+8\times1+4\times1+12\times3/10$
$=35.6$

Kruskal’s and Prim’s

- Make greedy decision about the next edge to add

```
KRUSKAL($G$)
1  for all $v \in V$
2      MAKESET($v$)
3  $T \leftarrow \{\}$
4  sort the edges of $E$ by weight
5  for all edges $(u, v) \in E$ in increasing order of weight
6      if FIND-SET($u$) $\neq$ FIND-SET($v$)
7          add edge to $T$
8          UNION(FIND-SET($u$),FIND-SET($v$))
```

```
PRIM($G, r$)
1  for all $v \in V$
2      $key[v] \leftarrow \infty$
3      $prev[v] \leftarrow null$
4  $key[r] \leftarrow 0$
5  $H \leftarrow$ MAKEHEAP($key$)
6  while !$Empty(H)$
7      $u \leftarrow$ EXTRACT-MIN($H$)
8      $visited[u] \leftarrow true$
9      for each edge $(u, v) \in E$
10         if !$visited[v]$ and $w(u, v) < key(v)$
11             DECREASE-KEY($v, w(u, v)$)
12             $prev[v] \leftarrow u$
```