Greedy Approach

V. Balasubramanian

The idea

- Ebenezer Scrooge’s approach-**Ebenezer Scrooge** is the principal character in Charles Dickens' 1843 novel, *A Christmas Carol*.
- Grab data items in sequence, each time taking the “best” one without regard for the choices made before or in the future
- Often used to solve optimization problems
- Must determine that the globally optimal solution can be obtained by a sequence of locally optimal solution

Greedy Method

```
SolType Greedy(Type a[], int n)
// a[1:n] contains the n input
{
    SolType solution = Empty; // Initialise the solution.
    For (int i =1; i <=n; i++) {
        Type x = Select (a);
        If ( Feasible (Solution, x)
            Solution = Union (Solution, x);
        } // for loop
        Return solution;
}
```

An example: Making changes

Quarter 25p

Dime =10p

Nickel = 5p

Penny = 1p

36 Paise

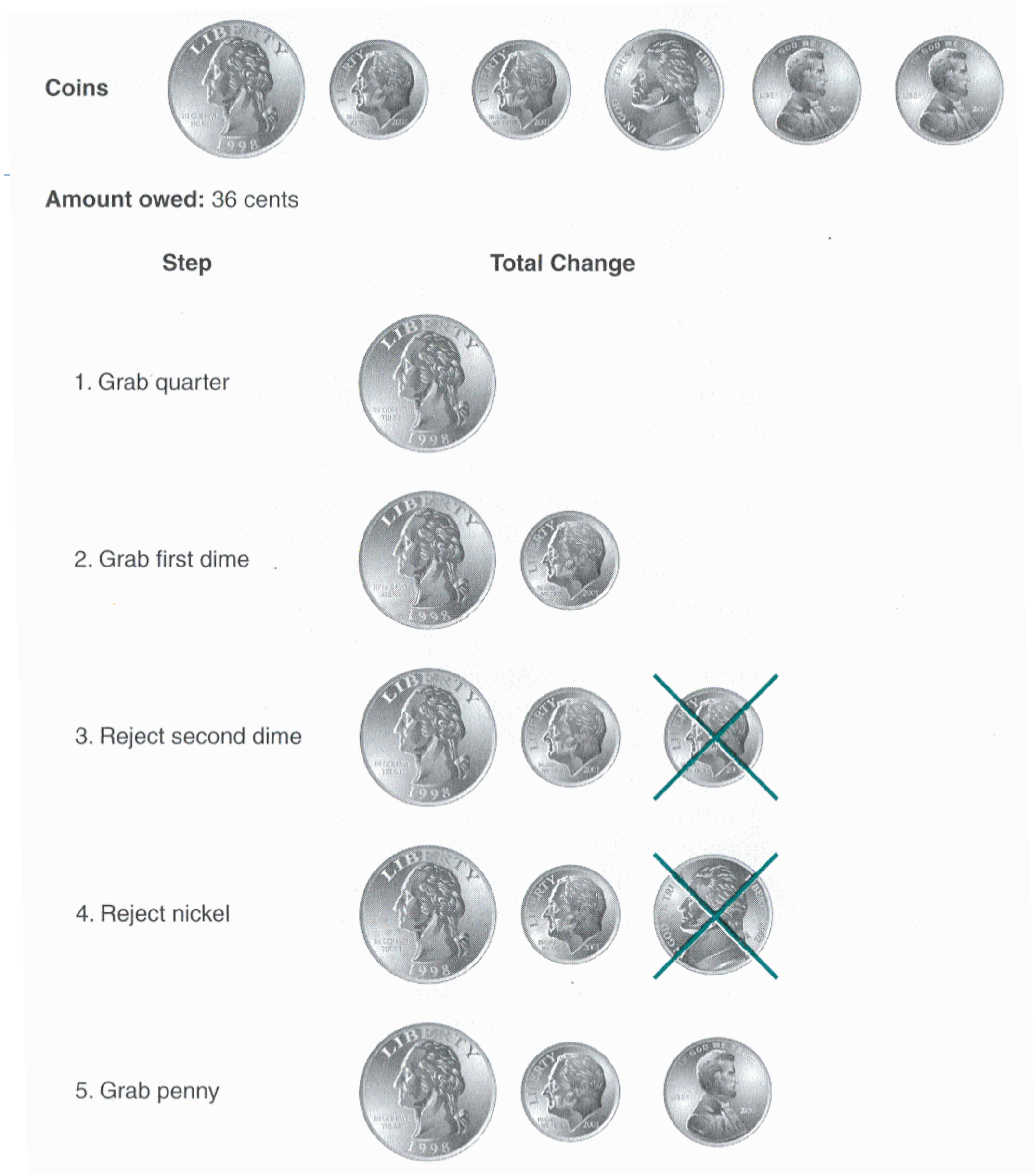


Figure 4.1 ● A greedy algorithm for giving change.

The algorithm

```
while ( there are more coins and the instance is not solved){

    grab the largest remaining coin;
                                                // selection procedure

    If (adding the coin makes the change exceed the amount owed)
            reject the coin;                    // feasibility check
    else
            add the coin to the change;

    If (the total value of the change equals the amount owed)
                                                // solution check
            the instance is solved;
}
```

When the greedy approach does not work

Quarter 25p
d2=12p
Dime =10p
Nickel = 5p
Penny = 1p

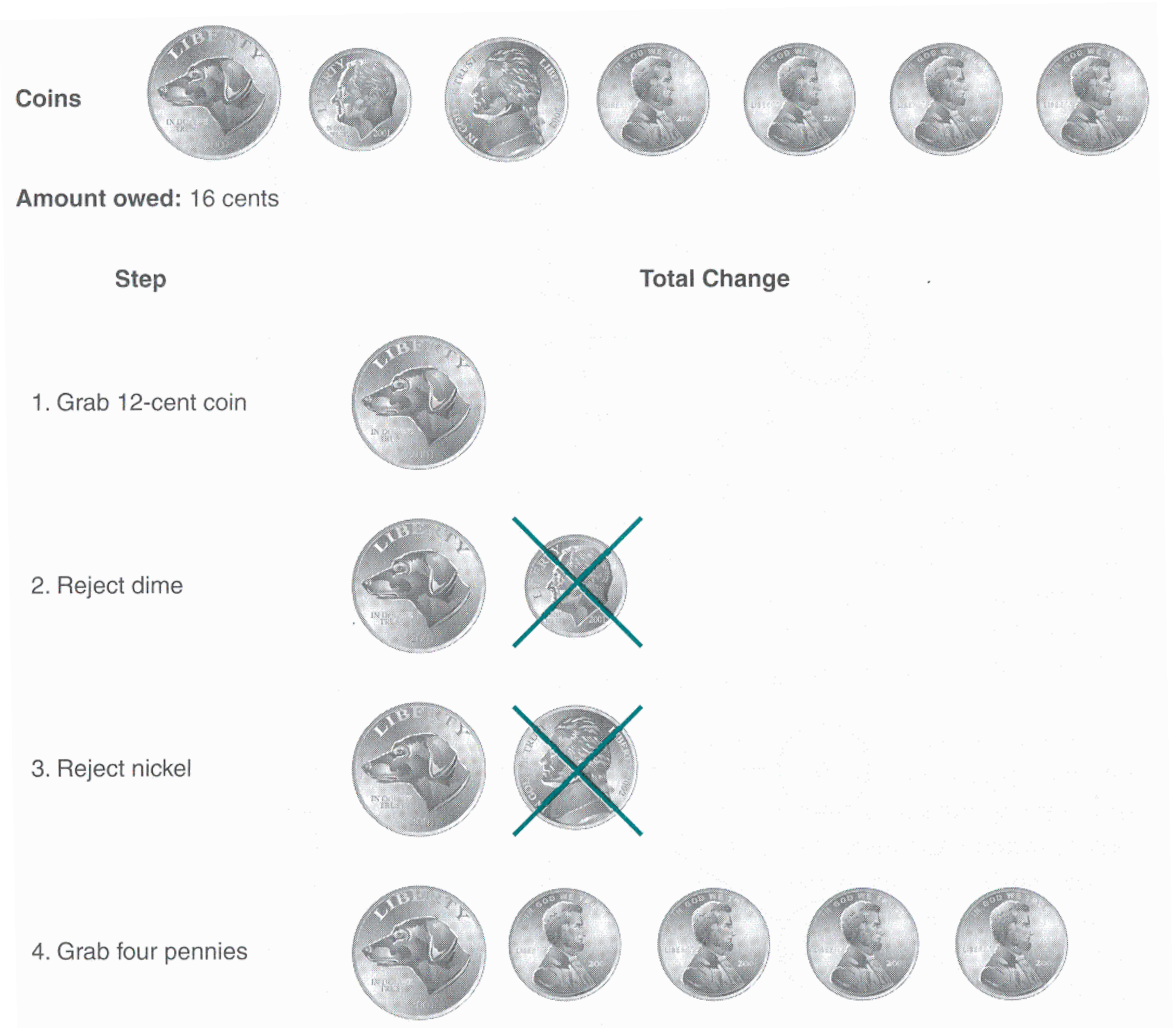


Figure 4.2 ● The greedy algorithm is not optimal if a 12-cent coin is included.

Example 3

- D1= 25, d2 = 10, d3 = 5, d4=1
- N = 67
- First D1 is chosen, second D1 is chosen
- Third D1 exceeds N, hence choose D2, then D3, and 2 D4.

Basic components in Greedy approach

- A ***selection procedure*** chooses the next item to add to the set. The selection is performed according to a greedy criterion that satisfies some locally optimal consideration at the time.

- A ***feasibility check*** determines if the new set is feasible by checking whether it is possible to complete this set in such a way as to give a solution to the instance.

- A ***solution check*** determines whether the new set constitutes a solution to the instance.

Machine Scheduling

- We have n tasks and an infinite supply of machines on which these tasks will be performed.
- Task has a start time s_i & finish time $f_{i,}$ $s_i < f_i$
- $[s_i , f_i]$ processing interval

- Feasible task-to-machine assignment is an assignment in which no machine is assigned two overlapping tasks.
- Optimal assignment is a feasible assignments that utilises the fewest number of machines.

Example

Task	A	B	C	D	E	F	g
Start	0	3	4	9	7	1	6
Finish	2	7	7	11	10	5	8

Task = 7, simple solution is use 7 machines, but we need to find optimal solution

Greedy solution

- Assign task in stages, sort the task in non-decreasing start time.
- Call a machine old, if a task is assigned to it.
- If a machine is not old, it is new.
- If an old machine becomes available by the start time of the task to be assigned, assign the task to this machine, if not assign it to a new machine.

solution

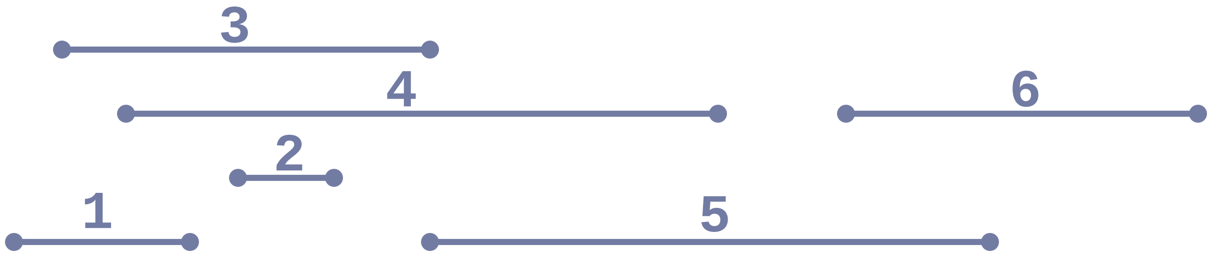
- Stage 1: No old machines: so task a is assigned to M1, it will be busy up to time 2.
- Stage 2: task f (task in increasing order of start time), is considered, since M1 is busy, assign it to M2.
- Stage 3: task b is considered, M1 is free, so assign it to M1.
- Stage 4: task c is considered, M1(f_b=7) & M2(f_f=5) are busy, assign it to M3.
- Stage 5: task g, M2 is available, assign task g.
- Stage 6: task e, M1, M3 is free, assign to M1.
- Stage 7: task d, M2, M3 is free, assign to M2.

Activity-Selection

- Formally:
  - Given a set S of n activities

    s_i = start time of activity i

    f_i = finish time of activity i
  - Find max-size subset A of compatible activities



- Assume (wlog) that $f_1 \le f_2 \le \ldots \le f_n$

Algorithm

- So actual algorithm is simple:
  - Sort the activities by finish time
  - Schedule the first activity
  - Then schedule the next activity in sorted list which starts after previous activity finishes
  - Repeat until no more activities
- Complexity:
  - O(n log n) for sorting the tasks, heap sort can be used.

Container loading

- A large ship is to be loaded with cargo.
- Cargo is containerized. All containers are of same size.
- Different containers may have different weights.
- Let w_i is the weight of *i*th container, $1 \leq i \leq n$.
- Cargo capacity of the ship is c.
- Problem is to load the ship with maximum number of containers.

Container loading

- Let x_i is the variable whose value can be either 0 or 1.
- If xi =0, container is not loaded. If 1, then container is loaded.

$$\sum_{i=1}^{n} w_i x_i \le c \qquad x_i \in \{0,1\}, 1 \le i \le n.$$

Every set of Xi that satisfies the constraints is a feasible solution.

$$\sum_{i=1}^{n} x_i = optimalsol\ ution$$

Contd..

- Ship is loaded in stages.
- As each stage we need to select a container to load.
- Use greedy criterion: from the remaining containers, select the one with least weight. This order of selection will keep the total weight of the selected containers minimum, and hence maximum containers can be loaded.

Example

- N =8, [w1…..w8] = [100, 200,50,90,150,50,20,80], c=400
- The containers considered for loading are in increasing order of weight.
- Containers considered are 7,3,6,8,4,1,5,2.
- Containers 7,3,6,8,4,1 together weigh 390 units and are loaded. Remaining 10 unit is inadequate to load container 5 & 2.
- [x1….x8] = [1,0,1,1,0,1,1,1] and $$\sum_{i=1}^{n} x_i = 6$$

Contd..

- Complexity O(n log n) for sorting the weights in ascending order
- Remainder of the algorithm takes O(n).
- Overall complexity is O(n log n)

knapsack problem

- The famous *knapsack problem*:
  - A thief breaks into a museum. Fabulous paintings, sculptures, and jewels are everywhere. The thief has a good eye for the value of these objects, and knows that each will fetch hundreds or thousands of dollars on the clandestine art collector’s market. But, the thief has only brought a single knapsack to the scene of the robbery, and can take away only what he can carry. What items should the thief take to maximize the haul?

Contd…

- More formally, the *0-1 knapsack problem*:
  - The thief must choose among *n* items, where the *i*th item worth v_i dollars and weighs w_i pounds
  - Carrying at most *W* pounds, maximize value
    - Note: assume v_i, w_i, and *W* are all integers
    - "0-1" b/c each item must be taken or left in entirety
- A variation, the *fractional knapsack problem*:
  - Thief can take fractions of items
  - Think of items in 0-1 problem as gold ingots, in fractional problem as buckets of gold dust

Knapsack problem

- We have n objects and a knapsack or bag.
- Object i has a weight w_i, and the knapsack has a capacity m.
- If a fraction x_i, $0 \le x_i \le 1$, of i is placed into knapsack, then a profit of $p_i x_i$ is earned.
- Objective is maximise the total profit earned.
- Since capacity is m,

Maximise $$\sum_{i=1}^{n} p_i x_i$$

$$\sum_{i=1}^{n} w_i x_i \le m$$

$$0 \le x_i \le 1,$$

$$1 \le i \le n$$

solution

- Feasible solution is any set (x1,….xn) satisfying second equation.
- Optimal solution is a feasible solution for which equ 1 is maximised.

Example

- Consider no of objects n =3, knapsack capacity m =20, profit of the objects is (p1, p2, p3) = (25,24,15), and weight of the object (w1, w2, w3) = (18,15,10).
- Feasible solutions are

(x1,x2,x3)	wixi	pixi
½,1/3,1/4	16.5	24.25
1,2/15,0	20	18.2
0,2/3,1	20	31
0,1,1/2	20	31.5. among 4 feasible solution, last one is optimal.

Lemma

- In the case the sum of all weights is ≤ m, then xi=1, 1 ≤ i≤n is an optimal solution.
- Lemma 2: let us assume that the sum of weights exceeds m, then xi's cannot be 1.
- All optimal solutions will fill the knapsack exactly.

Contd…

- 1. we try to fill the knapsack by including object of higher profit.
- If an object under consideration does not fit , then a fraction is chosen.
- For example, if we are left with 2 units of space and 2 objects with (pi =4, wi=4) & (pj=3,wj=2). It is better to include j than half of i.

Contd…

- Consider no of objects n =3, knapsack capacity m =20, profit of the objects is (p1, p2, p3) = (25,24,15), and weight of the object (w1, w2, w3) = (18,15,10).
- Sol:
- Largest profit p1 is placed in knapsack. X1=1.
- Object 2 is next largest profit, p2=24. w2=15, it does not fit in knapsack, b/c total weight 18+15 exceeds 20.
- So a fraction is chosen x2=2/15, profit =28.2.
- It is not optimal solution.

Next strategy

- Theorem: $p1/w1 \geq p2/w2 \geq \ldots . \geq pn/wn$, then greedy algorithm generates an optimal solution.
- 25/18, 24/15, 15/10 = 1.388, 1.6, 1.5
- Choose p2 weight is 15, remaining 5 unit , choose 0.5 unit of p3.
- (0,1,0.5)

Review: The Knapsack Problem And Optimal Substructure

- Both variations exhibit optimal substructure
- To show this for the 0-1 problem, consider the most valuable load weighing at most *W* pounds
  - *If we remove item j from the load, what do we know about the remaining load?*
  - A: remainder must be the most valuable load weighing at most W - w_j that thief could take from museum, excluding item j

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - *How?*
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - Greedy strategy: take in order of dollars/pound
  - Example: 3 items weighing 10, 20, and 30 pounds, knapsack can hold 50 pounds
    - *Suppose item 2 is worth $100. Assign values to the other items so that the greedy strategy will fail*

The Knapsack Problem: Greedy Vs. Dynamic

- The fractional problem can be solved greedily
- The 0-1 problem cannot be solved with a greedy approach
  - As you have seen, however, it can be solved with dynamic programming

The greedy approach versus dynamic programming: The knapsack problem

- Efficiency:
  - Greedy approach is often simpler and more efficient
- Proof:
  - Dynamic programming: principle of optimality
  - Greedy approach: a proof that is usually more complex

A greedy approach to the 0-1 knapsack problem

- The 0-1knapsack problem
  - Let
    - S = {*item*$_1$, *item*$_2$, …, *item*$_n$}
    - w_i = weight of $item_i$
    - p_i = profit of $item_i$
    - W = maximum weight the knapsack can hold
  - Determine a subset *A* and *S* such that

$$\sum_{item_i \in A} p_i \quad is \quad \max imized \quad subject \quad to \quad \sum_{item_i \in A} w_i \leq W$$

  - Brute force algorithm: $O(2^n)$ - there are 2^n *sub-sets*

Greedy approach fails

- Approach 1: steal the items with the largest profit first
- Steal the lightest items first
- Steal the items with the largest profit per unit weight first

Greedy approach fails (cont'd)

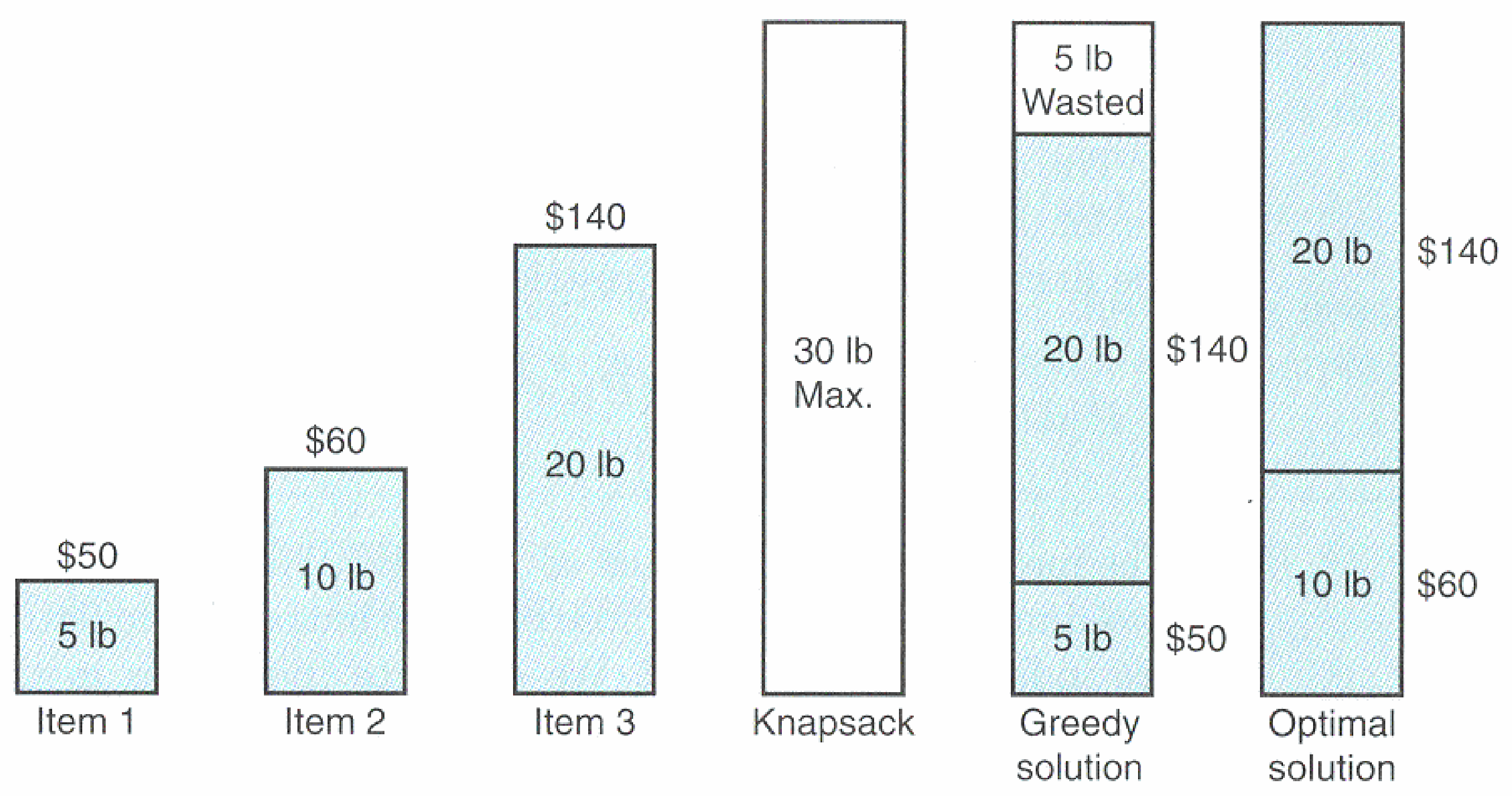


Figure 4.13 ● A greedy solution and an optimal solution to the 0-1 Knapsack problem.

A greedy approach to the fractional knapsack problem

- Total profit in the previous example

  \$50 + \$140 + (5/10)(\$60) = \$220

A dynamic programming approach to the 0-1 knapsack problem

- The algorithm

$$P\left[i\right]\left[w\right] = \begin{cases} maximum\left(P\left[i-1\right]\left[w\right], p_i + P\left[i-1\right]\left[w-w_i\right]\right) & \text{if } w_i \leq w \\ P\left[i-1\right]\left[w\right] & \text{if } w_i > w. \end{cases}$$

- The maximum profit = *P*[*n*][*W*]
- Using array *P*[0-*n*][0-*W*]

  Set *P*[0][*W*] and *P*[*i*][0] to 0
- Time complexity:

  The number of entries computed is *nW* $\Theta(nW)$

A refinement of the dynamic programming algorithm for the 0-1 knapsack problem

- Going back to determine what entries are needed. Because

$$P[n][W] = \begin{cases} maximum\left(P[n-1][W], p_n + P[n-1][W-w_n]\right) & \text{if } w_n \leq W \\ P[n-1][W] & \text{if } w_n > W, \end{cases}$$

- Entries needed in the (n-1)st row are $P[n\text{-}1][W]$ and $P[n\text{-}1][W\text{-}w_n]$
- In general, we can use the fact that $P[i][w]$ is computed from $P[i\text{-}1][w]$ and $P[i\text{-}1][w\text{-}w_i]$
- Until n = 1 or $w \leq 0$

An example

- W = 30

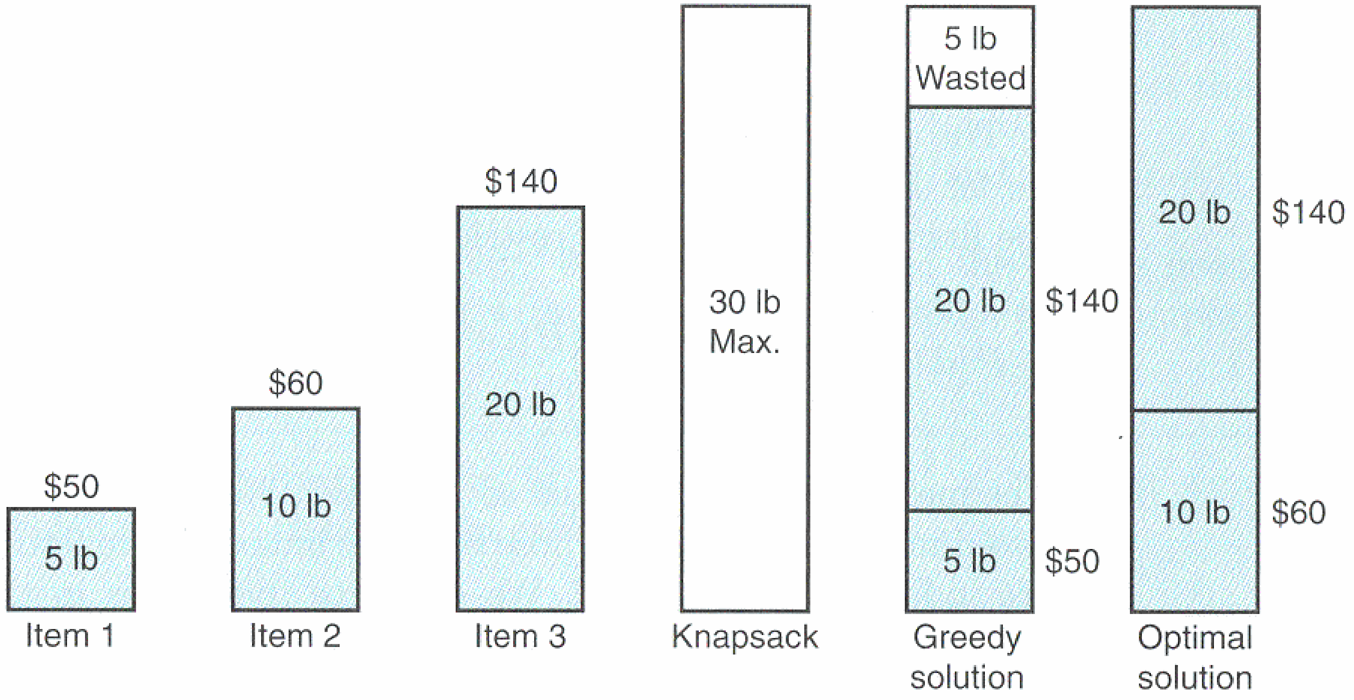


Figure 4.13 ● A greedy solution and an optimal solution to the 0-1 Knapsack problem.

Time complexity

- We compute at most 2^i entries in the $(n\text{-}i)$th row.
- The total number of entries computed is at most
  $1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n - 1 = \Theta(2^n)$