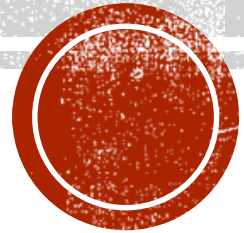
Lect04. The Greedy Approach



Greedy Algorithm

- The **greedy algorithm** grabs data items in sequence, each time taking the item that is deemed to be "best," without regard to the choices it has made before or to the choices that it might make in the future. Despite the negative impression, they often lead to simple and efficient solutions.

- A greedy algorithm arrives at a solution by making a sequence of choices, each of which simply looks best at the moment. The Change problem discussed in the text shows that a greedy algorithm **does not guarantee an optimal solution**.

Greedy Algorithm

- A greedy algorithm **starts with an empty set** and **adds items** to the set in sequence **until the set represents a solution** to an instance of a problem. Each iteration in the greedy algorithm consists of the following components:
  - A **selection procedure**, chooses the next item to add to the set. The selection is preformed 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.

Coin Change Problem

- Finding the number of ways of making changes for a particular amount of cents, n, using a given set of denominations $C = \{c_1, \ldots, c_d\}$ (e.g the US coin system:{1, 5, 10, 25, 50, 100})
  - An example : $n = 4, C = \{1,2,3\}$, solutions: $\{1,1,1,1\}, \{1,1,2\}, \{2,2\}, \{1,3\}$.
- Minimizing the number of coins returned for a particular quantity of change(available coins {1, 5, 10, 25})
- i.e. n=25a + 10b + 5c +d, what are the a, b, c, and d, minimizing(a+b+c+d)
  - 30 cents(solution:25+2, two coins)
  - 67 cents ?
- 17 cents given denominations ={1,2,3,4}?

Coin changing problem

- $n = 25a + 10b + 5c + d$, what are the a, b, c, and d, minimizing (a+b +c+d)
- How to do it in brute-force?
  - At most we use n pennies
  - Try all the combinations where $a \leq n, b \leq n, c \leq n, d \leq n$
  - Choose all the combinations that $n = 25d + 10b + 5c + d$
  - Choose the combination with smallest (a+b+c+d)

Coin changing problem

- n = 25a + 10b + 5c + d, what are the a, b, c, and d, minimizing (a+b+c+d)
- How to do it by a greedy algorithm?

```
coinGreedy( n ){
    if(n>=25) s = coinGreedy(n-25); s.a++;
    else if(n>=10) s = coinGreedy(n-10); s.b++;
    else if(n>=5) s = coinGreedy(n-5); s.c++;
    else s=(a=0, b=0, c=0, d=n, sum=n);
    s.sum++;
    return  s;
```

Greedy choice

Always choose the possible largest coin

Time complexity? $T(n)=\Theta(n)$

It that greedy algorithm correct?

Coin changing problem

- Optimal substructure
  - After the greedy choice, assuming the greedy choice is correct, can we get the optimal solu tion from sub optimal result?
    - 38 cents
    - Assuming we have to choose 25
    - Is a quarter + optimal **coin**(38-25) the optimal solution of 38 cents?
- Greedy choice property
  - If we do not choose the largest coin, is there a better solution?

Find the Fewest Coins: Casher's algorithm

- Given 30 cents, and coins {1, 5, 10, 25}
- Here is what a casher will do : always go with coins of highest value first
  - Choose the coin with highest value 25
    - 1 quarter
  - Now we have 5 cents left
    - 1 nickel
  - The solution is 2(one quarter + one nickel)


Greedy Algorithm Does not Always Give Optimal Solution to Coin Change Problem

- Coins ={1, 3, 4, 5}
- 7 cents = ?

- Greedy solution:
  - 3 coins : one 5 + two 1

- Optimal solution:
  - 2 coins: one 3 + one 4

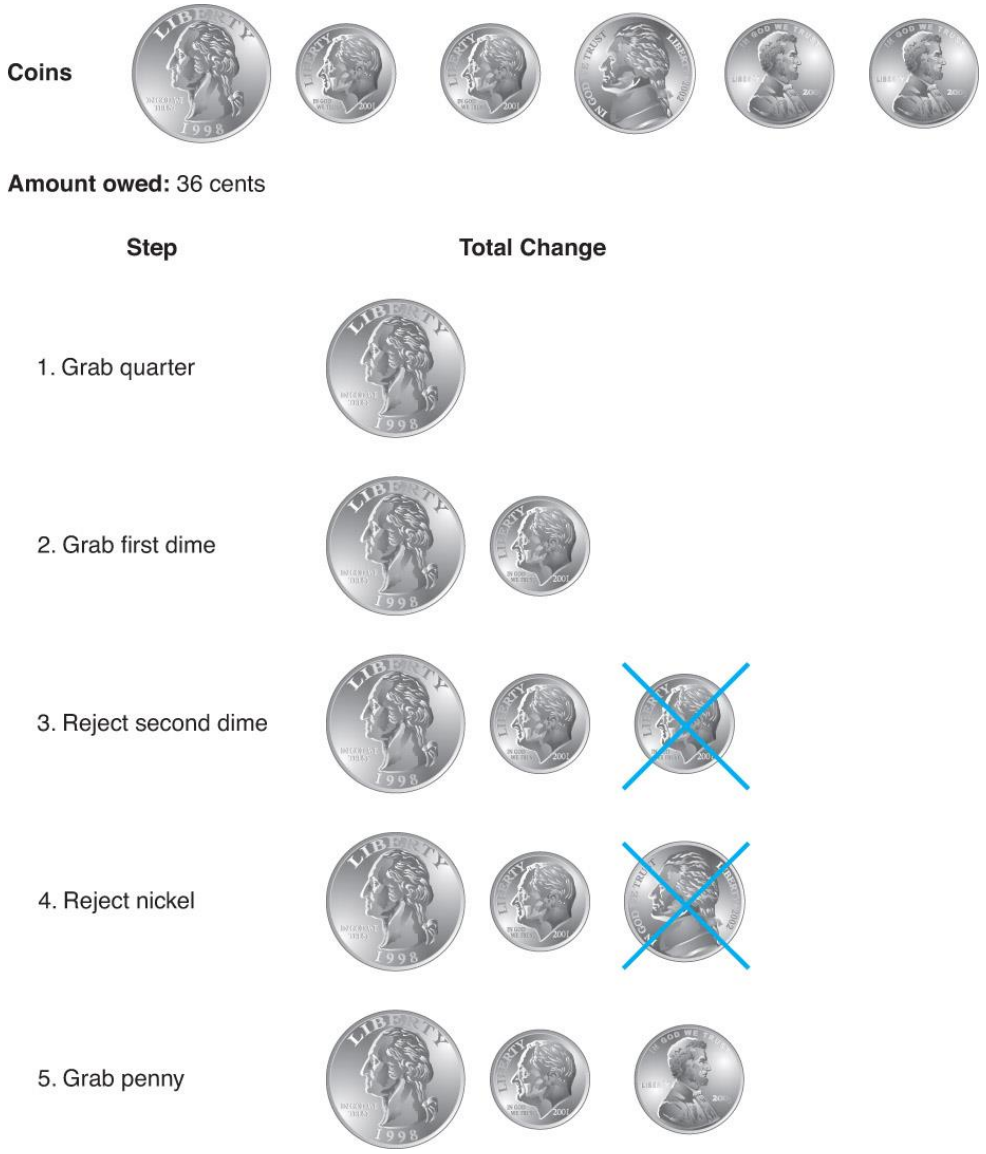


Figure 4.1: A greedy algorithm for giving change

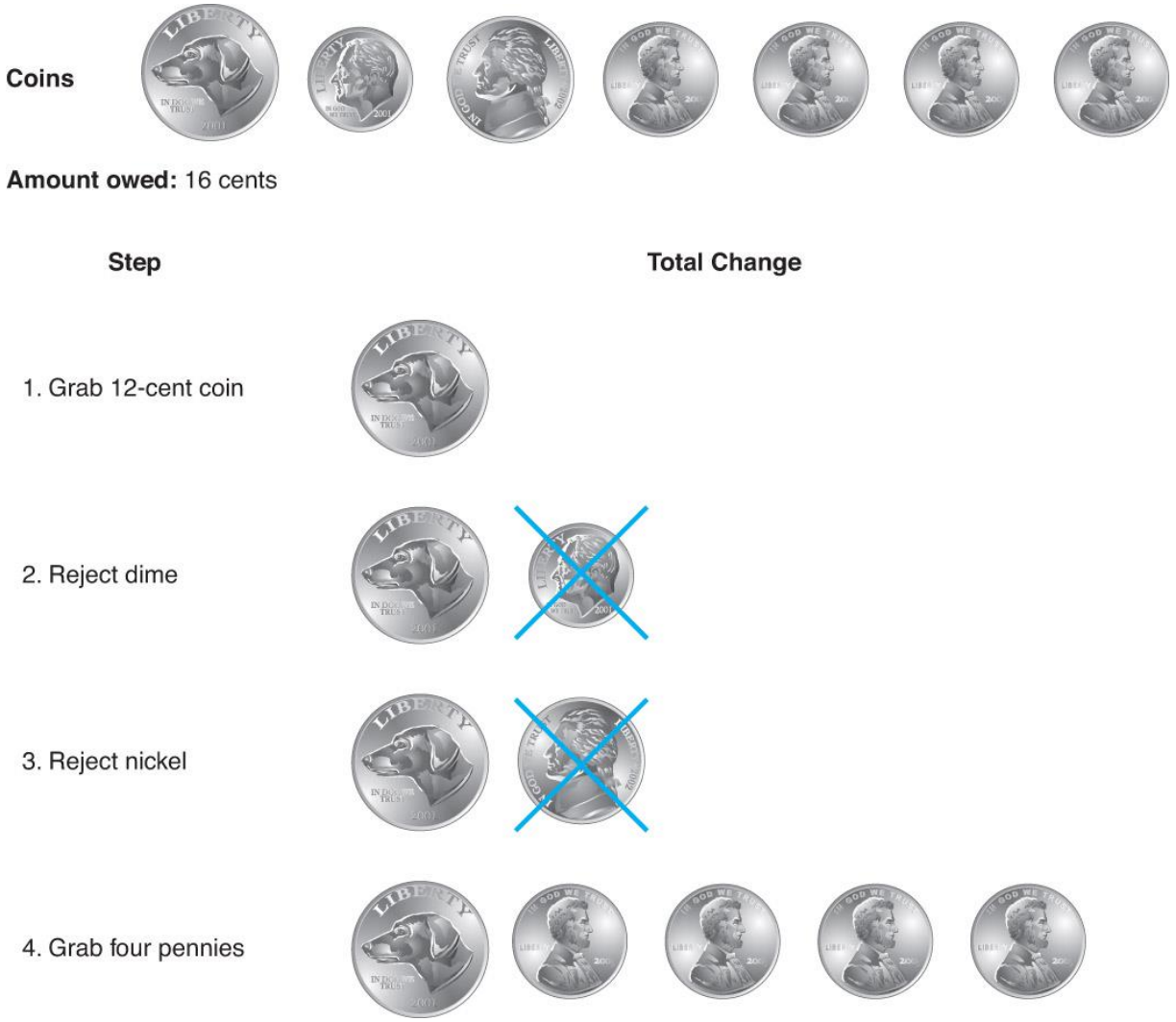


Figure 4.2: The greedy algorithm is not optimal if a 12-cent coin is included

Huffman Code

Huffman Code

- Even though the capacity of secondary storage devices keeps getting larger, and their cost keeps getting smaller, the devices continue to fill up due to increased storage demands.
- Given a data file, it would therefore be desirable to find a way to store the file as efficiently as possible.
- The problem of **data compression** is to find an efficient method for encoding a data file.
- Next, we discuss the encoding method, called **Huffman code,** and a greedy algorithm for encoding a given file.

Huffman Code

- A common way to represent a file is to use a **binary code**. In such a code, each character is represented by a unique binary string, called the **codeword.**

Huffman Code

- A **fixed-length** binary code represents each character using the same number of bits.
- If our character set is {*a,b,c*}, we could use two bits to code each character, as follows:

a: 00 b: 01 c: 11 (Code 4.1)

Given this code, if our file is

ababcbbbc,

our encoding is

a b a b c b b b c

000100011101010111.

- Needs 18 bits.

Huffman Code

- We can obtain a more efficient encoding using **variable-length binary code**. Such a code can represent different characters using different numbers of bits.

The code

a: 10 b: 0 c: 11 (Code 4.2)

Would be encoded as

10010011000 11.

a b ab c bbb c

- With this encoding, it takes only **13 bits** to represent the file, where it took **18 bits** to represent the previous file.

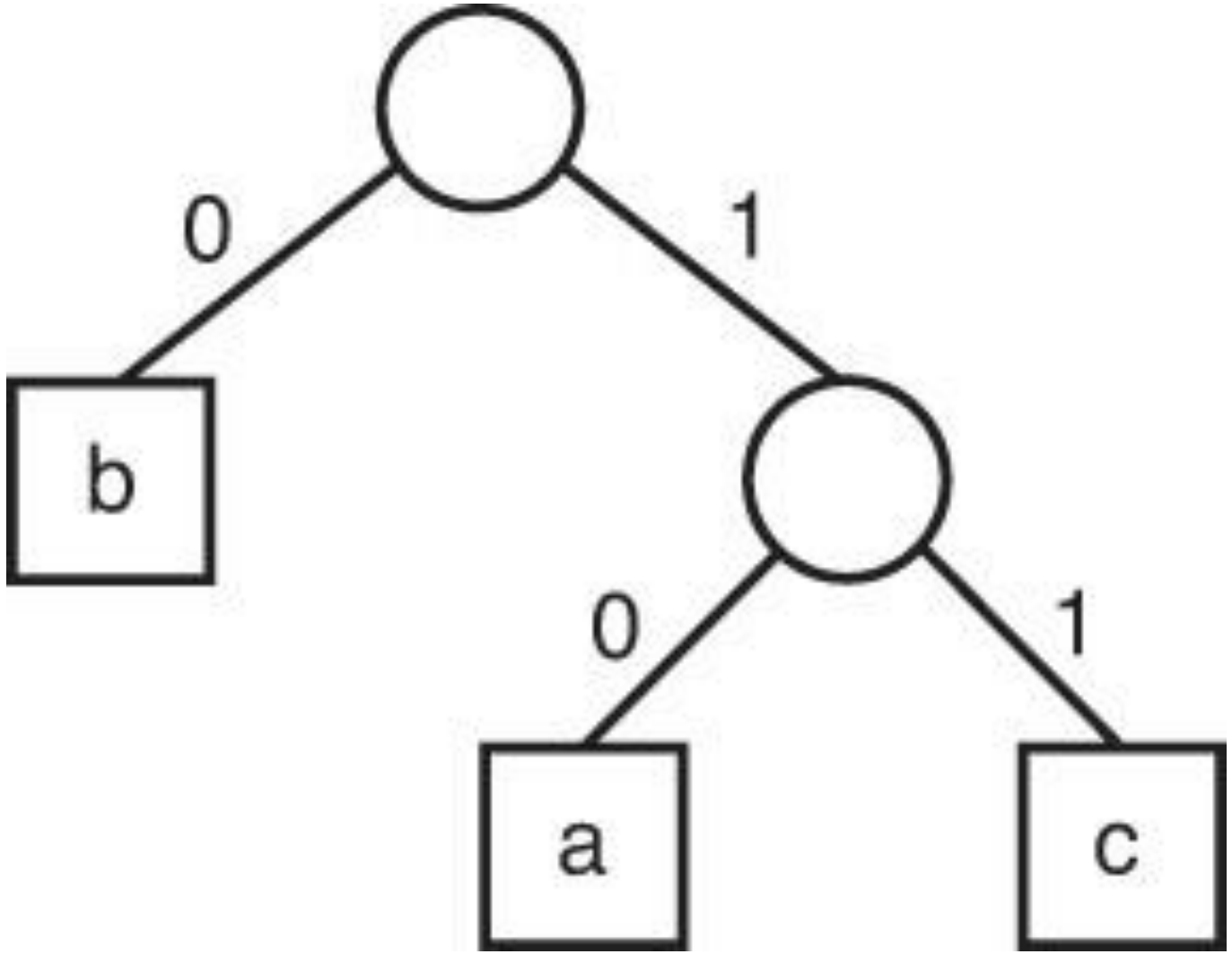


Figure 4.9: Binary tree corresponding to Code 4.2

Huffman Code

- Given a file, the **Optimal Binary Code problem** is **to find a binary code** for the characters in the file, which represents the file in **the least number of bits**.
- First, we will discuss prefix codes, and then we will develop Huffman's algorithm for solving this problem.

Prefix Codes

- One particular type of **variable-length code** is a **prefix code**.
- In a prefix code, no codeword for one character constitutes the beginning of the codeword for another character.
  - Ex. 01 : code word for ‘a’, 011 could not be the codeward for ‘b’
- A fixed-length code is also a prefix code.
- Every **prefix code** can be represented by a **binary tree** whose **leaves are the characters** that are to be encoded.
- The **advantage of a prefix code** is that we **need not look ahead** when parsing the file. To parse, we start at the first bit on the left in the file and the root of the tree.
- We sequence through the bits, and go left or right down the tree depending on whether a 0 or 1 is encountered.
- When we **reach a leaf**, we **obtain the character at that leaf**; then we **return to the root** and **repeat the procedure starting with the next bit in sequence**.

Example

- Character set : {a, b, c, d, e, f}
- Character appears

character	Frequency	C1(fixed length)	C2	C2(Huffman)
a	16	000	10	00
b	5	001	11110	1110
c	12	010	1110	110
d	17	011	110	01
e	10	100	11111	1111
f	25	101	0	10

- # of bits for each encoding:
  - Bits(C1)=16(3)+5(3)+12(3)+17(3)+10(3)+25(3)=255
  - Bits(C2)=16(2)+5(5)+12(4)+17(3)+10(5)+25(1)=231
  - Bits(C3)=16(2)+5(4)+12(3)+17(2)+10(4)+25(2)=212

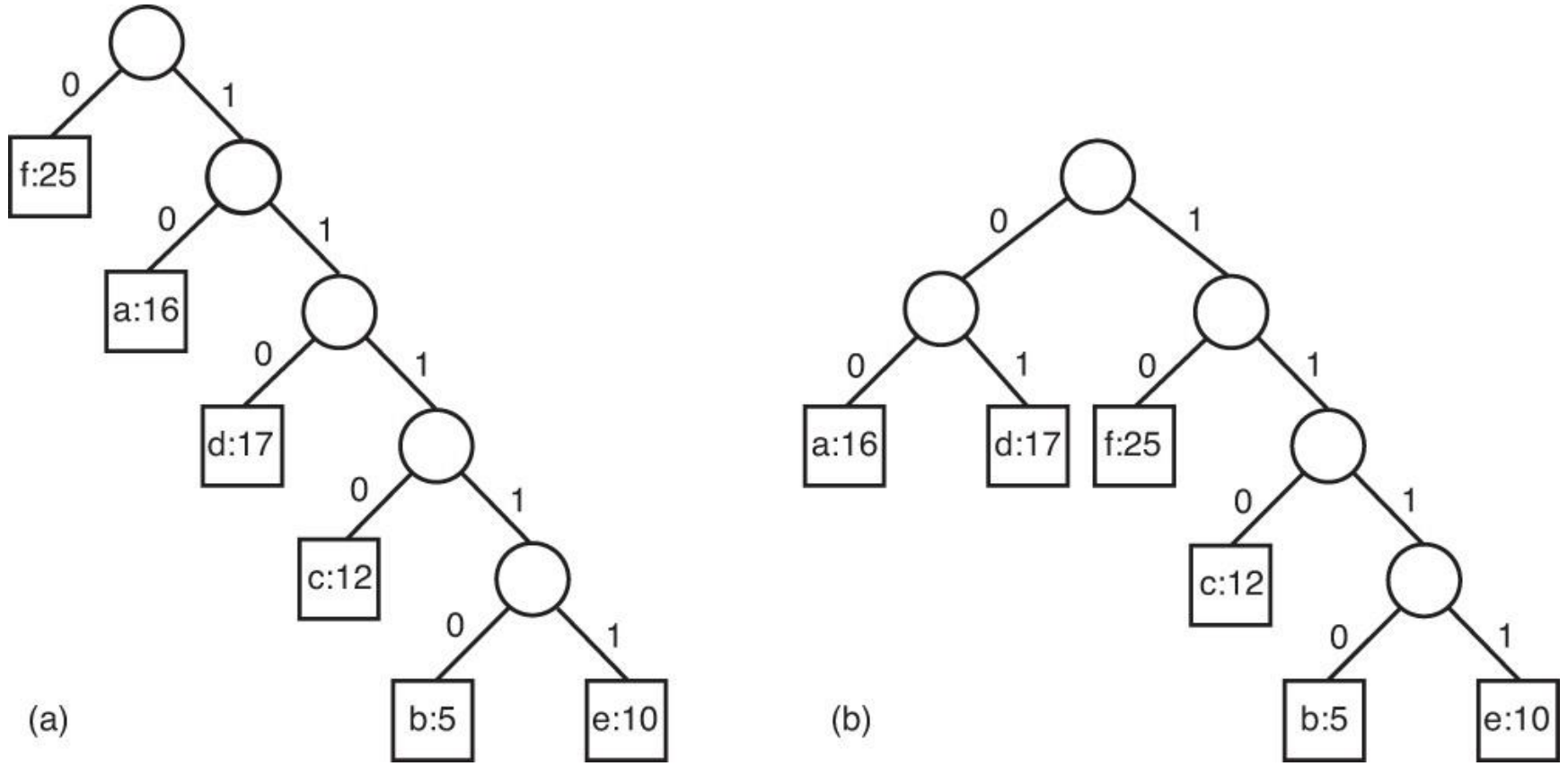


Figure 4.10: The binary character code for Code C2 in Example 4.7 appears in (a), while the one for Code C3 (Huffman) appears in (b).

Huffman's Algorithm

- Huffman developed a greedy algorithm that produces an optimal binary character code by constructing a binary tree corresponding to an optimal code. A code produced by this algorithm is called a **Huffman code**.
- We represent a high-level version of the algorithm. Because it involves constructing a tree, we need to be more detailed in our other high-level algorithms. We have the following type declaration:

Huffman's Algorithm

- Type declaration

```
struct nodetype
{
    char symbol; //The value of a character
    int frequency; // The # of times the character is in the file

    nodetype* left;
    nodetype* right;
}
```

- We also need to use a **priority queue**. In a priority queue, the element with the highest priority is always removed next. In this case, the element with the **highest priority** is the character with the **lowest frequency in the file**. The priority queue can be implemented as a linked list, but more efficiently as a heap.

Huffman's Algorithm

- Inputs : file, n : # of characters in the file
- Output : prefix code tree(heap for Huffman code)
- Algorithm

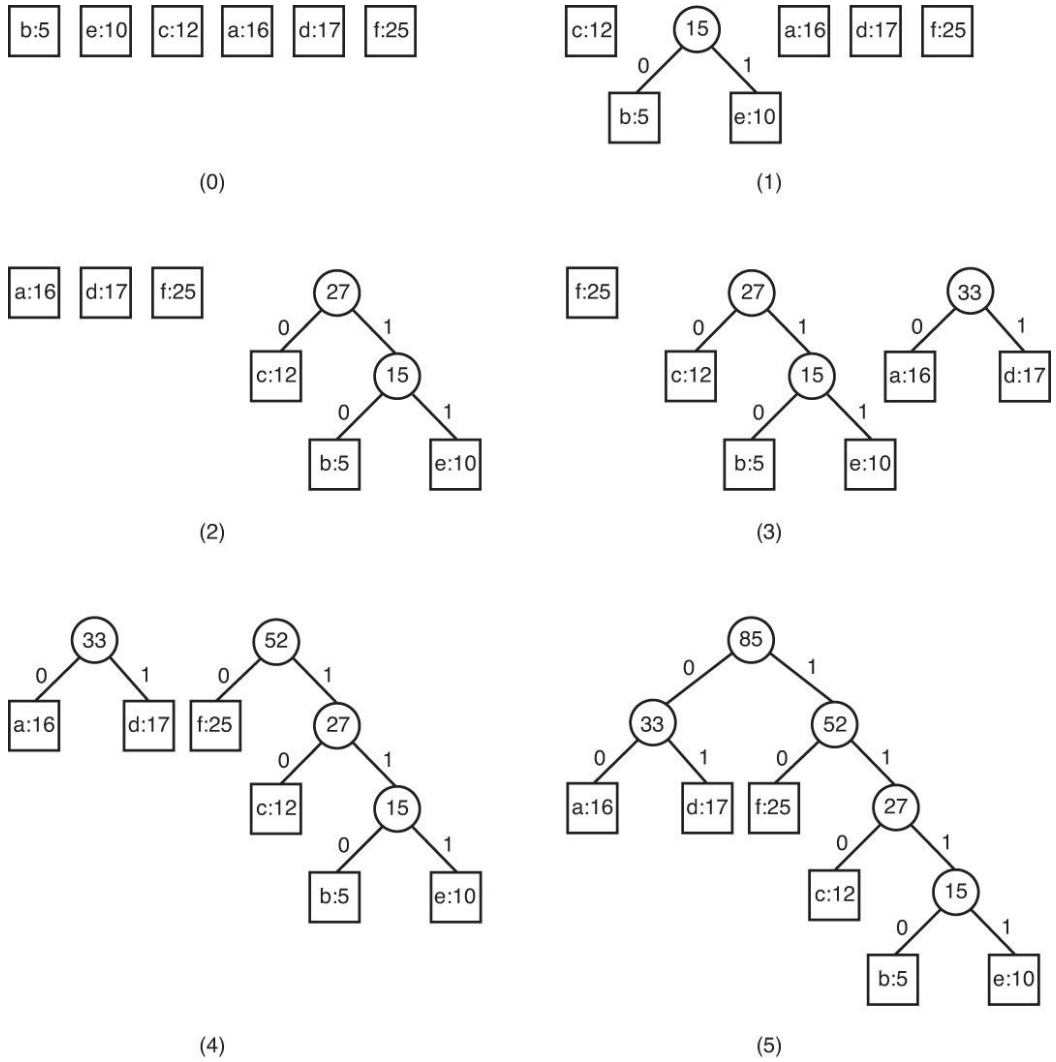
1. Arrange n pointers to nodetype records in a priority queue PQ as follows

```
for each pointer p in PQ
    p->symbol = a distinct character in the file;
    p->frequency = the frequency of that character in the file;
    p->left=p->right=NULL;
```

2.
```
for(i=1; i<= n-1 ; i++){
    remove(PQ, p);
    remove(PQ, q);
    r= new nodetype;
    r->left = p;
    r->right = q;
    r->frequency = p->frequency + q->frequency;
    insert(PQ, r);
}
remove(PQ, r);
return r;
```

Huffman's Algorithm: example

character	Frequency
a	16
b	5
c	12
d	17
e	10
f	25



The Knapsack Problem

The Greedy Approach versus Dynamic Programming: The Knapsack Problem

- The **greedy approach** and **dynamic programming** are two ways **to solve optimization problems**. A problem can be solved using either approach.
- It is usually **more difficult to determine** whether a **greedy algorithm always produces an optimal solution**.
- A proof is needed to show that a particularly greedy algorithm always produces an optimal solution, whereas a counterexample is needed to show that it is not.

0/1 Knapsack Problem

- Given a knapsack with weight $W > 0$.

- *A* set S of n items with weights $w_i > 0$ and profits $p_i > 0$ for $i = 1,\ldots,n$.

- $S = \{ (item_1, w_1, p_1), (item_2, w_2, p_2), \ldots, (item_n, w_n, p_n) \}$

- Find a **subset** of the items which **does not exceed the weight** ***W*** of the knapsack and **maximizes the profit**.

0/1 Knapsack problem

Determine a subset A of $\{ 1, 2, \ldots, n \}$ that satisfies the following:

$$\max \sum_{i \in A} p_i \text{ where } \sum_{i \in A} w_i \leq W$$

In 0/1 knapsack a specific item is either selected or not

Variations of the Knapsack problem

- Fractions are allowed. This applies to items such as:
  - bread, for which taking half a loaf makes sense
  - gold dust
- No fractions.
  - 0/1 (1 brown pants, 1 green shirt…)
  - Allows putting many items of same type in knapsack
    - 5 pairs of socks
    - 10 gold bricks
  - More than one knapsack, etc.
- First 0/1 *knapsack* problem will be covered then the Fractional *knapsack* problem.

A Greedy Approach to the 0-1 Knapsack Problem

- In this example, a thief carrying a knapsack breaks into a jewelry store. The knapsack will break if the total weight exceeds the maximum weight, W. Each item in the knapsack has value and weight. The thief's dilemma is to maximize the total number of items while not making the total weight exceed W.
- The obviously **greedy strategy** is to steal the items with the **largest profit first**, steal them in nonincreasing order according to profit. This strategy does not work if the most valuable item has a large weight in comparison to its profit.
- Another **greedy strategy** is to steal the **lightest items first**. This strategy fails if the lightest items have smaller profits.
- The most efficient strategy is for the thief to steal the items that have the **largest profit per unit weight first**. So, we order the items in nonincreasing order according to profit per unit weight and select them in sequence. An item is put in the knapsack if it does not bring the total weight above W.

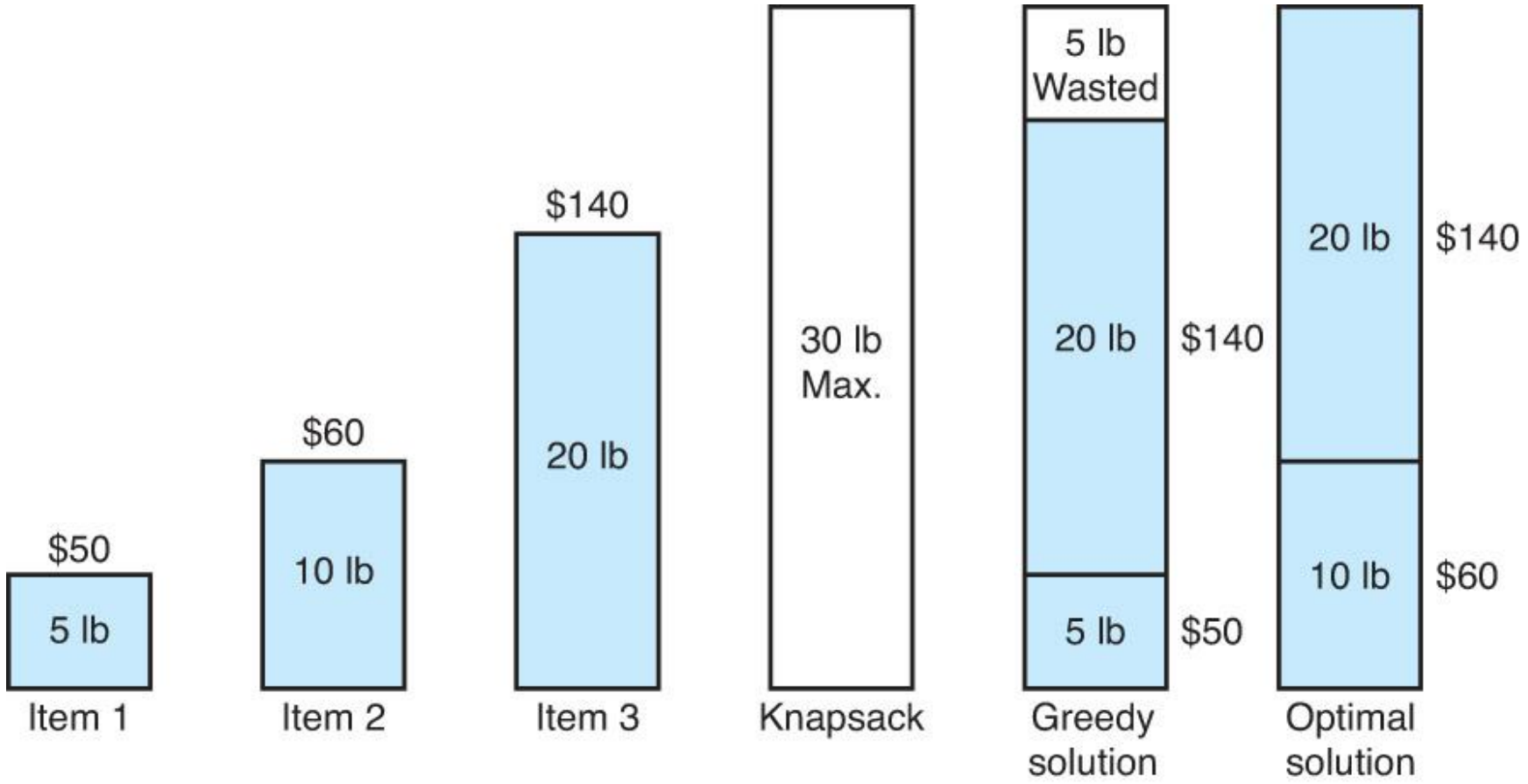


Figure 4.13: A greedy solution and an optimal solution to the 0-1 Knapsack problem.

A Greedy Approach to the Fractional Knapsack Problem

- In the Fractional Knapsack problem, the thief does not have to steal all of an item, but rather can take any fraction of the item.

- The greedy approach to the fractional knapsack problem **yields the optimal solution**:

  $50 + $140 + (5/10) *60 = $220 (The thief takes 5/10 of item 2)

  5 is the remaining capacity of the knapsack

- The greedy algorithm never wastes any capacity in the Fractional Knapsack problem as it does in the 0/1 Knapsack problem. It always leads to an optimal solution.

Brute force!

- Generate all 2^n subsets

- Discard all subsets whose sum of the weights exceed *W (not feasible)*
- Select the maximum total profit of the remaining (feasible) subsets

Example with "brute force"

S = { ($item_1$, 5, \$70), ($item_2$,10, \$90), ($item_3$, 25, \$140) } , W=25

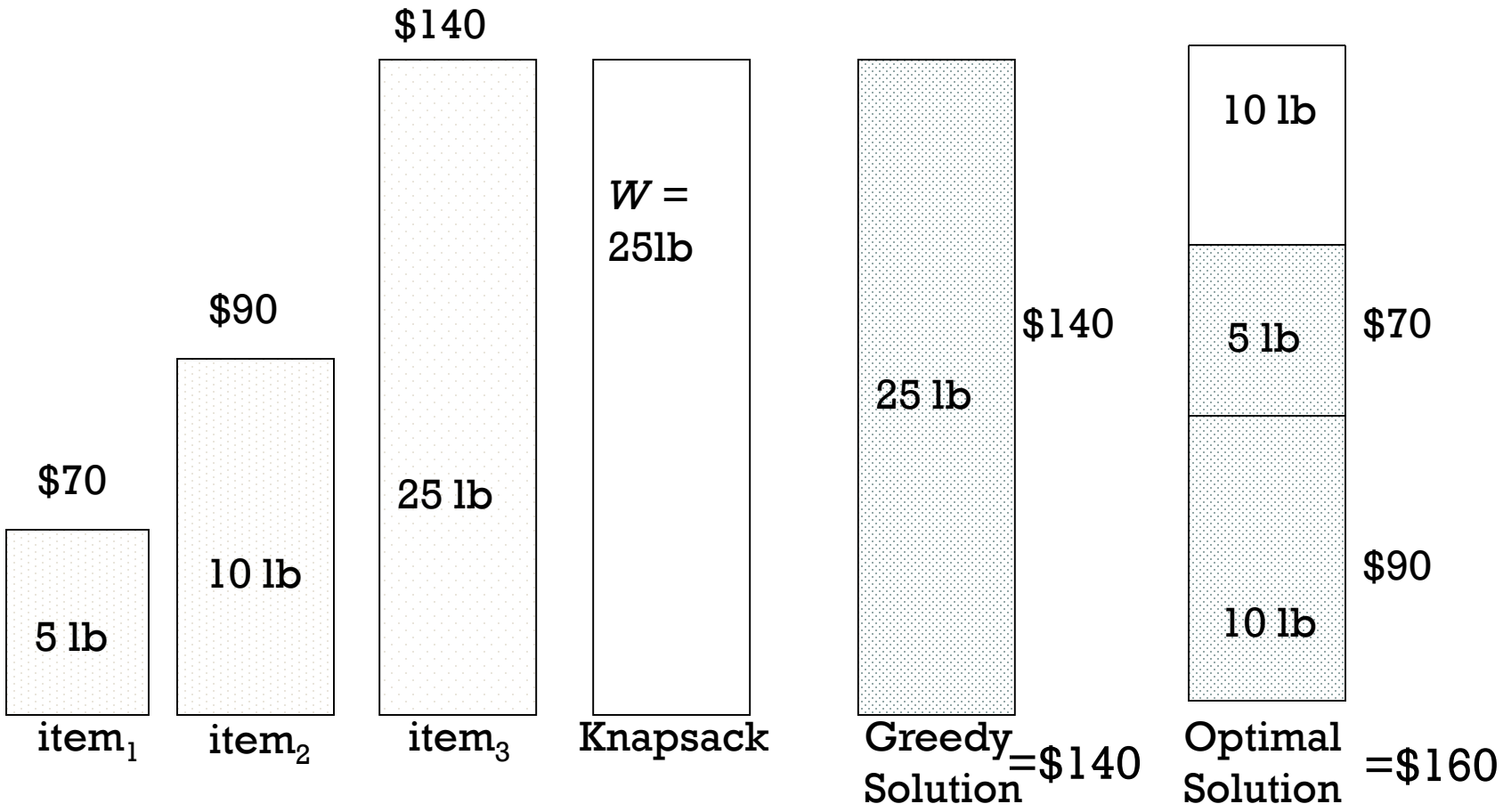
- Subsets:

1. {}
2. { ($item_1$, 5, \$70) } Profit=\$70
3. { ($item_2$,10, \$90) } Profit=\$90
4. { ($item_3$, 25, \$140) } Profit=\$140
5. { ($item_1$, 5, \$70), ($item_2$,10, \$90)}. Profit=\$160 ****
6. { ($item_2$,10, \$90), ($item_3$, 25, \$140) } exceeds W
7. { ($item_1$, 5, \$70), ($item_3$, 25, \$140) } exceeds W
8. { ($item_1$, 5, \$70), ($item_2$,10, \$90), ($item_3$, 25, \$140) } exceeds W

Greedy 1: Selection criteria: *Maximum beneficial* item.

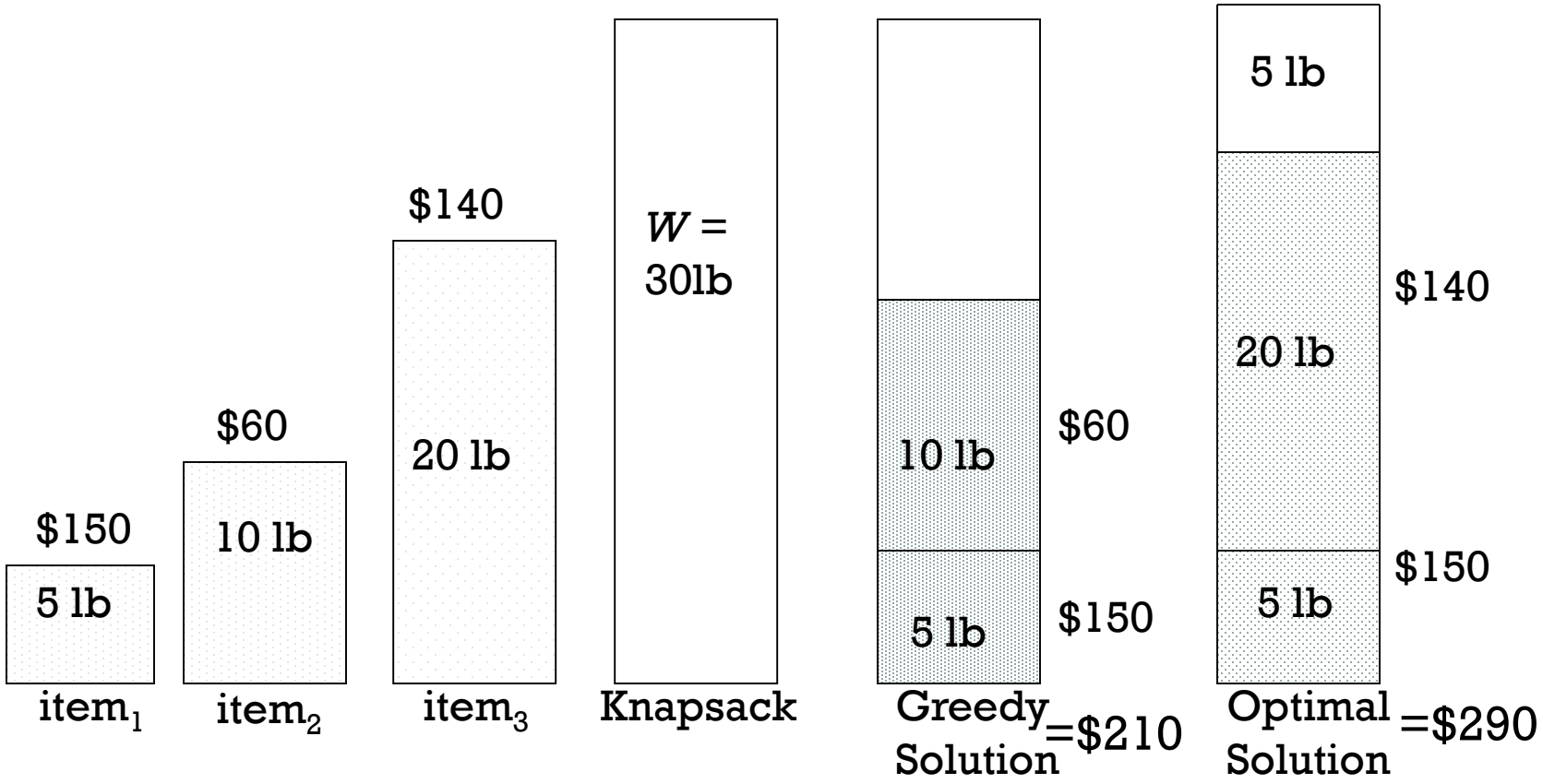
Counter Example:

$S = \{ (item_1, 5, \$70), (item_2, 10, \$90), (item_3, 25, \$140) \}$



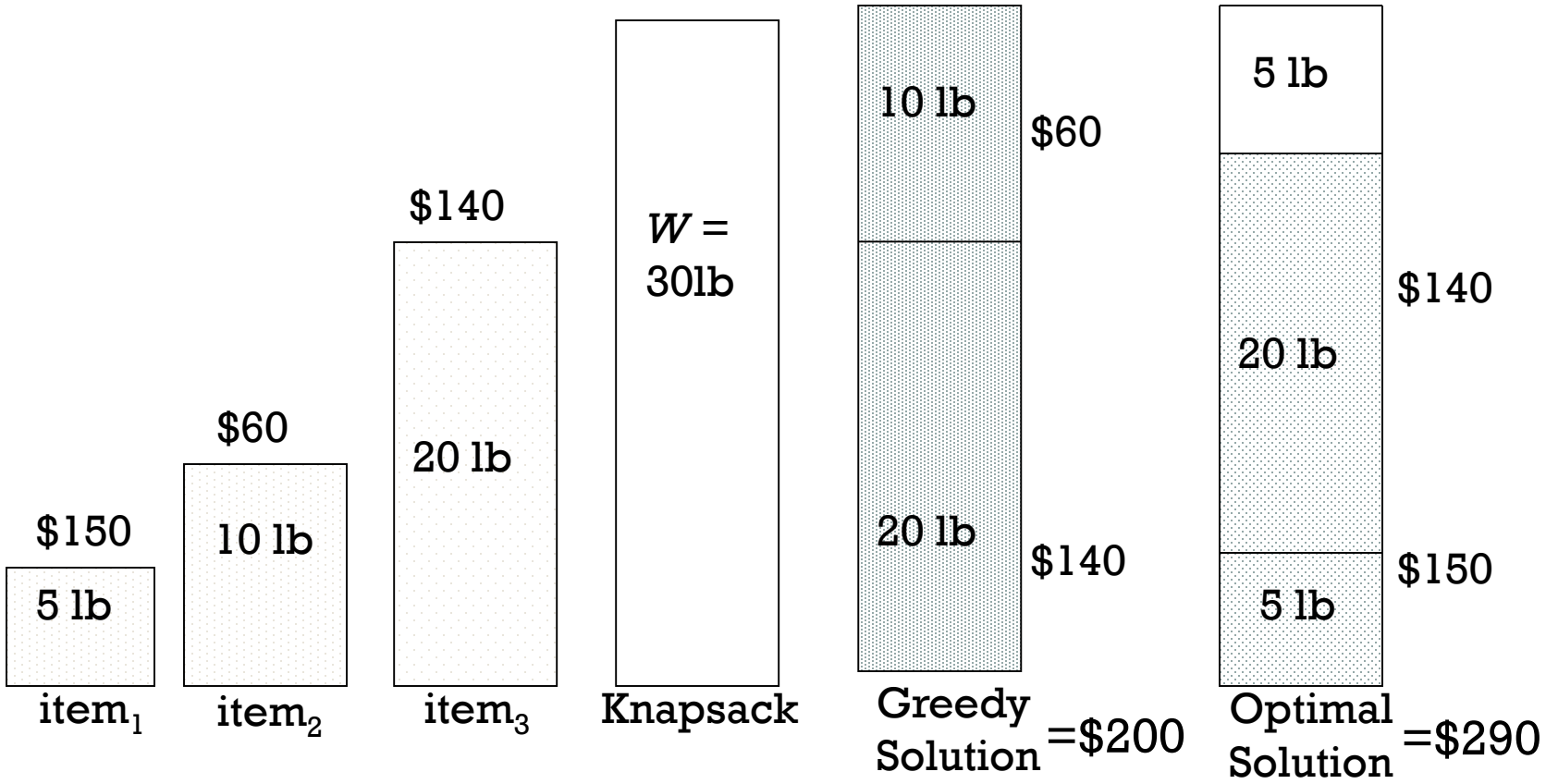
Greedy 2: Selection criteria: *Minimum weight* item
Counter Example:

$S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \}$



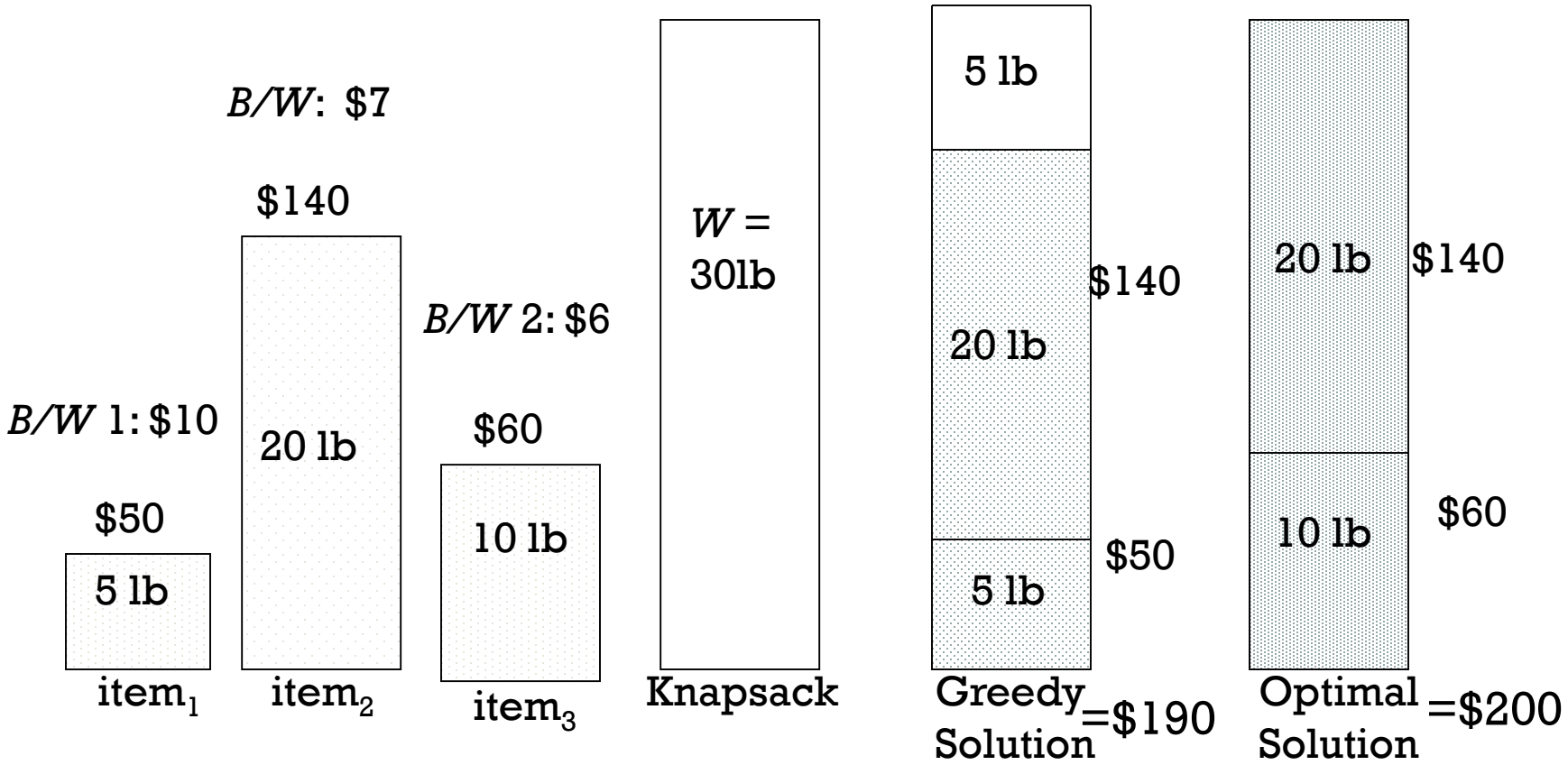
Greedy 3: Selection criteria: *Maximum weight* item
Counter Example:

$S = \{ (item_1, 5, \$150), (item_2, 10, \$60), (item_3, 20, \$140) \}$



Greedy 4: Selection criteria: *Maximum benefit per unit* item
Counter Example

$S = \{ (item_1, 5, \$50), (item_2, 20, \$140) (item_3, 10, \$60), \}$



An Optimal Greedy Algorithm for Knapsack with Fractions (KWF)

In this problem a fraction of any item may be chosen

The following algorithm provides the optimal benefit:

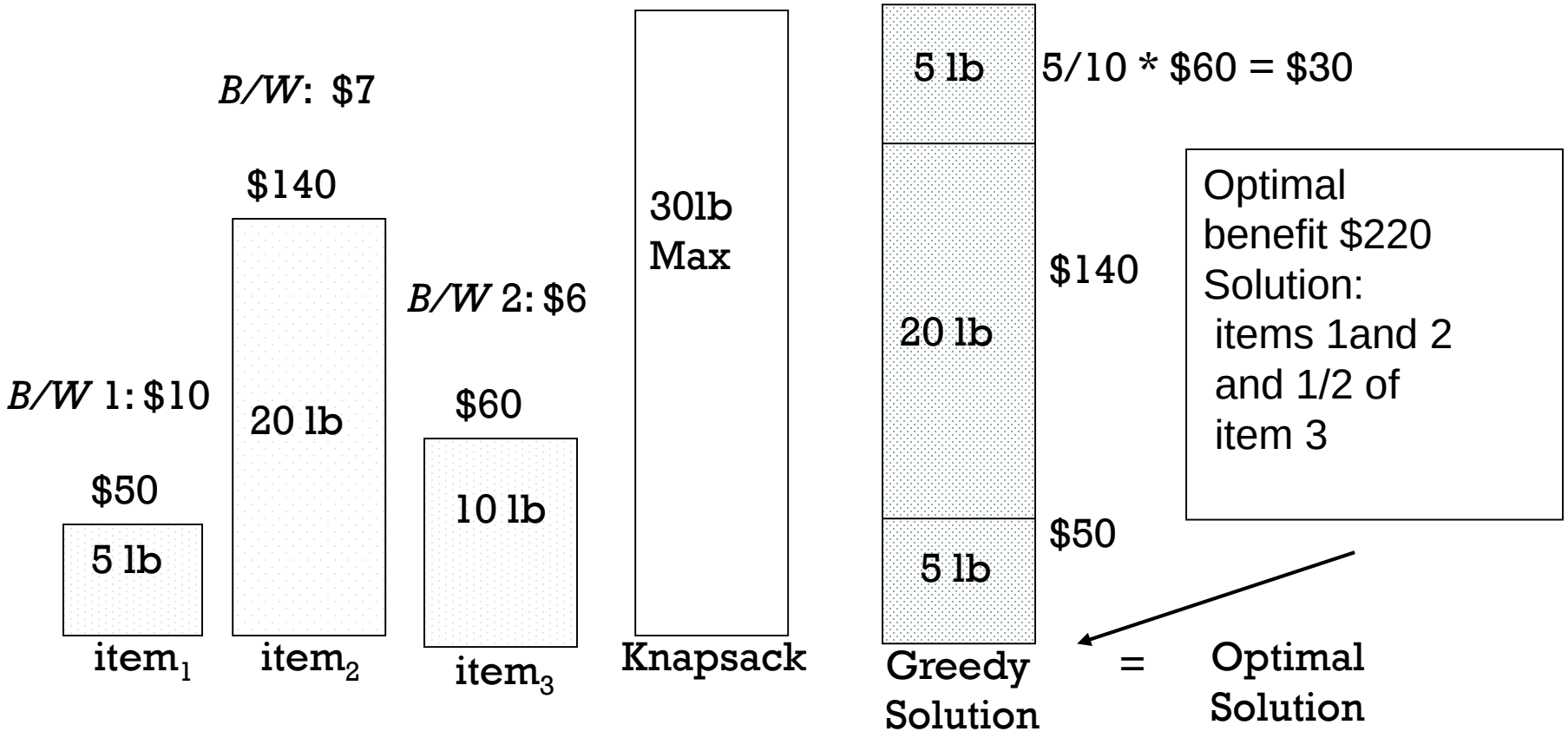
- The greedy algorithm uses the *maximum profit per unit* selection criteria
  1. Sort items in decreasing p_i / w_i.

  2. *Add items to knapsack (starting at the first) until there are no more items*, or the next item to be added exceeds W.

  3. If knapsack is not yet full, fill knapsack with a fraction of next unselected item.

Example of applying the optimal greedy algorithm for Fractional Knapsack Problem

- $S = \{ (item_1, 5, \$50), (item_2, 20, \$140) (item_3, 10, \$60), \}$



Example of applying the optimal greedy algorithm for Fractional Knapsack Problem

W=30
S = { (item1, 5, $50), (item2, 20, $140), (item3, 10, $60) }
Note: items are already sorted by benefit/weight

Applying the algorithm:
Current weight in knapsack=0, Current benefit=0.
Can item 1 fit? 0+5<30 so select it. Current benefit=0+50
Can item 2 fit? 5+20<30, so select. Current benefit =50+140=190
Can item 3 fit? 25+10>30. No.
We can add 5 to knapsack (30-25).
So select 5/10=0.5 of item 3.
Current benefit=190+30=220

Greedy Algorithm for Knapsack with fractions

- To show that the greedy algorithm finds the optimal profit for the fractional Knapsack problem you need to prove there is no solution with a higher profit

- Notice there may be more than one optimal solution

Dynamic Programming Approach

- Given a knapsack problem with n items and knapsack weight of W.
- We will first compute the maximum benefit, and then determine the subset.
- To use dynamic programming we solve smaller problems and use the optimal solutions of these problems to find the solution to larger ones.

Dynamic Programming Approach

- What are the smaller problem?
  - Assume a subproblem in which the set of items is restricted to $\{1,\ldots, i\}$ where $i \leq n$, and the weight of the knapsack is w, where $0 \leq w \leq W$.
  - Let $P[i][w]$ denote the maximum benefit achieved for this problem.
  - Our goal is to compute the **maximum benefit** of the original problem $P[n][W]$
  - We solve the original problem by computing $P[i][w]$ for $i = 0, 1,\ldots, n$ and for $w = 0,1,\ldots,W$.
  - We need to specify the solution to a larger problem in terms of a smaller one

Recursive formula for the "smaller" 0/1Knapsack Problem
Using only $item_1$ to $item_j$ and knapsack weight at most w

3 cases:

1. There are no items in the knapsack, or the weight of the knapsack is 0 - the benefit is 0
2. The weight of $item_i$ exceeds the weight w of the knapsack - $item_i$ cannot be included in the knapsack and the maximum profit is $P[i\text{-}1][w]$
3. Otherwise, the profit is the maximum achieved by either not including $item_i$ (i.e., $P[i\text{-}1][w]$), or by including $item_i$ (i.e., $P[i\text{-}1][w\text{-}w_i]+p_i$)

$$P[i][w] = \begin{cases} 0 & \text{for } i = 0 \text{ or } w = 0 \\ P[i-1][w] & \text{if } w_i > w \\ \max\{P[i-1][w], P[i-1][w-w_i] + p_i\} & \text{otherwise} \end{cases}$$

Pseudo-code:0/1 Knapsack
$(n+1)*(W+1)$ Matrix

Input: $\{w_1, w_2, \cdots w_n\}, W, \{b_1, b_2, \cdots b_n\}$

Output: $P[n][W]$,

for $w \leftarrow 0$ to W **do** *// row* 0
  $P[0][w] \leftarrow 0$
for $k \leftarrow 1$ to n **do** // rows 1 to n
  $P[k][0] \leftarrow 0$ // element in column 0
  for $w \leftarrow 1$ to W **do** // elements in columns 1 to W

if $(w_k \leq w)$ **and** $(P[k\text{-}1][w\text{-}w_k] + b_k > P[k\text{-}1][w])$
then $P[k][w] \leftarrow P[k\text{-}1][w\text{-}w_k] + p_k$
else $P[k][w] \leftarrow P[k\text{-}1][w]$

Knapsack 0-1 Problem

- Let's run our algorithm on the following data:
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight, value):
    (2,3), (3,4), (4,5), (5,6)

Knapsack 0-1 Example

(2,3), (3,4), (4,5), (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

```
// Initialize the base cases
for w = 0 to W
        B[0,w] = 0

for i = 1 to n
        B[i,0] = 0
```

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

i = 1
$v_i = 3$
$w_i = 2$
w = 1
$w-w_i = -1$

if w_i <= w //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = v_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

i = 1

$v_i = 3$

$w_i = 2$

w = 2

$w\text{-}w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

i = 1

$v_i = 3$

$w_i = 2$

w = 3

$w-w_i = 1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$w = 4$
$w\text{-}w_i = 2$

if $w_i <= w$ //item i can be in the solution
  if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
    $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
  else
    $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

i = 1
$v_i = 3$
$w_i = 2$
w = 5
$w\text{-}w_i = 3$

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
        $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
    else
        $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
w = 1
$w\text{-}w_i = -2$

if $w_i <= w$ //item i can be in the solution
  if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
    $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
  else
    $B[i,w] = B[i\text{-}1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

i = 2
$v_i = 4$
$w_i = 3$
w = 2
$w\text{-}w_i = -1$

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
        $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
    else
        $B[i,w] = B[i\text{-}1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$w = 3$
$w\text{-}w_i = 0$

if $w_i <= w$ //item i can be in the solution
  if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
    $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
  else
    $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$w = 4$
$w\text{-}w_i = 1$

if $w_i <= w$ //item i can be in the solution
  if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
    $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
  else
    $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$w = 5$
$w\text{-}w_i = 2$

if $w_i <= w$ //item i can be in the solution
  if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
    $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
  else
    $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

i = 3
$v_i = 5$
$w_i = 4$
w = 1..3
$w\text{-}w_i = -3..-1$

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = v_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
w = 4
$w\text{-}w_i = 0$

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
        $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
    else
        $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
w = 5
$w\text{-}w_i = 1$

if $w_i <= w$ //item i can be in the solution
  if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
    $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
  else
    B[i,w] = B[i-1,w]
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

i = 4
$v_i = 6$
$w_i = 5$
w = 1..4
$w\text{-}w_i = -4..-1$

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = v_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 4$
$v_i = 6$
$w_i = 5$
$w = 5$
$w\text{-}w_i = 0$

if $w_i <= w$ //item i can be in the solution
  if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$
    $B[i,w] = v_i + B[i\text{-}1,w\text{-}w_i]$
  else
    $B[i,w] = B[i\text{-}1,w]$
else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	**7**

We're DONE!!

The max possible value that can be carried in this knapsack is ***$7***

Analysis

- It is straightforward to fill in the array using the expression on the previous slide. SO What is the size of the array??
- The array is the (number of items+1) * (W+1).
- So the algorithm will run in $\Theta(nW)$. It appears to be linear BUT the weight is not a function of only the number of items. What if $W = n!$? Then this algorithm is worst than the brute force method. One can show algorithms for 0/1 knapsack with worst case time complexity of $O(min(2^n, nW))$ (N169)
- No one has ever found a 0/1 knapsack algorithm whose worst case time is better than exponential AND no one has proven that such an algorithm is not possible.

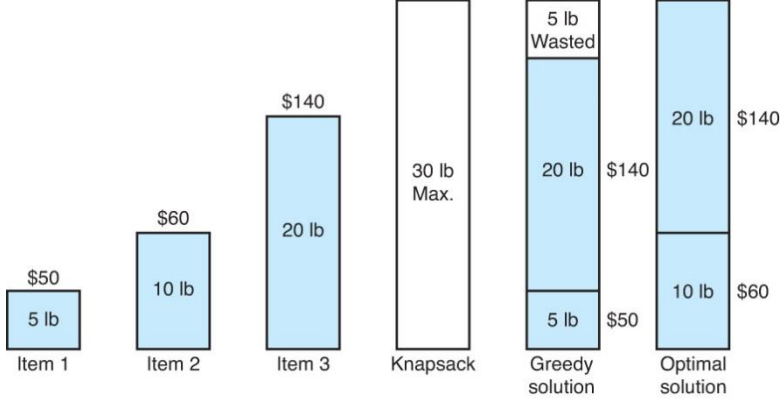
A Dynamic Programming Approach to the 0/1 Knapsack Problem

- If we can show that the principle of optimality applies, we can use dynamic programming to solve the 0 /1 Knapsack problem.
- Let *A* be an optimal subset of *n* items. There are two cases:
  - If *A* contains item *n*, then the total profit of items in *A* is equal to *pn* plus the optimal profit obtained from the first *n* –1 items, where the total weight cannot exceed *W* – *wn*.
  - If *A* does not contain item n, then the total profit of items in *A* is equal to the optimal subset of the first *n* –*1* items.

A Refinement of the Dynamic Programming Algorithm for the 0/1 Knapsack Problem

- The fact that the previous expression for the number of array entries computed is linear in *n* can mislead one into thinking that the algorithm is efficient for all instances containing *n* items. That is not the case. There is no relationship between *n* and *W*. When W is extremely large in comparison to n, this algorithm is worse than the brute-force algorithm that simply considers all subsets. See the text for more details.
-
- As is the case for the Traveling Salesperson problem, no one has ever found an algorithm for the 0/1 knapsack problem whose worst-case time complexity is better than exponential, yet no one has proven that such an algorithm is not possible.

Example

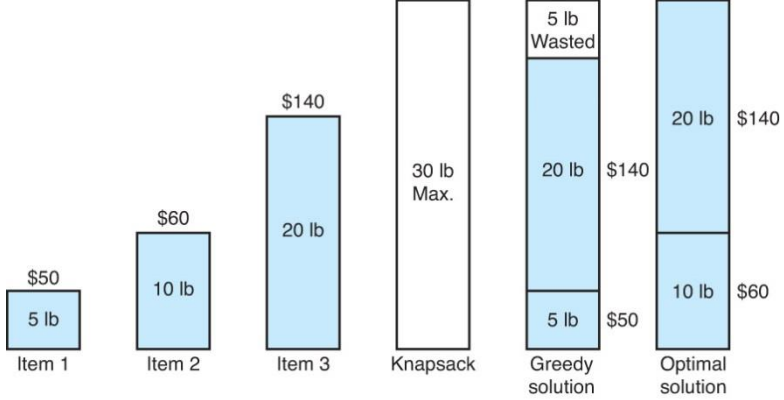


- W=30
  - Determine entries needed in row 3;
    P[3][W]=P[3][30]
  - Determine entries needed in row 2;
    To compute P[3][30],
    P[3-1][30]=P[2][30], P[3-1][30-w_3]=P[2][10]
  - Determine entries needed in row 1;
    To compute P[2][30],
    P[2-1][30]=P[1][30], P[2-1][30-w_2]=P[1][20]
    To compute P[2][10],
    P[2-1][10]=P[1][10], P[2-1][10-w_2]=P[1][0]
  - Next we do the computations.
  - Compute row 1:

$$P[1][w] = \begin{cases} \max(P[0][w], \$50 + P[0][w-5]) & \text{if } w_1 = 5 \le w \\ P[0][w] & \text{if } w_1 = 5 > w \end{cases}$$

$$= \begin{cases} \$50 & \text{if } w_1 = 5 \le w \\ \$50 & \text{if } w_1 = 5 > w \end{cases}$$

Example



- Therefore ,

  P[1][0]=$0

  P[1][10]=$50

  P[1][20]=$50

  P[1][30]=$50.

- Compute row 2:

$$P[2][10] = \begin{cases} \max(P[1][10], \$60 + P[1][0]) & \text{if } w_2 = 10 \leq 10 \\ P[1][10] & \text{if } w_2 = 10 > 10 \end{cases}$$

$$= \$60.$$

$$P[2][30] = \begin{cases} \max(P[1][30], \$60 + P[1][20]) & \text{if } w_2 = 10 \leq 30 \\ P[1][10] & \text{if } w_2 = 10 > 30 \end{cases}$$

$$= \$60 + \$50 = \$110.$$

- Compute row 3:

$$P[3][30] = \begin{cases} \max(P[2][30], \$140 + P[2][10]) & \text{if } w_3 = 20 \leq 30 \\ P[2][30] & \text{if } w_3 = 20 > 30 \end{cases}$$

$$= \$140 + \$60 = \$200.$$

Discussion

- Computes only 7 entries, whereas the original version would have computed (3)(30)-90 entries.
- Let's determine how efficient this version is in the worst case. Notice that we compute at most 2^i entries in the (n-i)th row. Therefore, at most the total number of entries computed is

$$1 + 2 + 2^2 + \ldots + 2^{n-1} = 2^n - 1.$$

- It is left as an exercise to show that the following is an instance for which about 2^n entries are computed (the profits can have any values):

$$w_i = 2^{i-1} \text{ for } 1 \leq i \leq n \text{ and } W = 2^n - 2.$$

- Combining these two results, we can conclude that the worst-case number of entries computed is in

$$\Theta(2^n).$$

- The previous bound is in terms of only n. Let's also obtain a bound in terms of n and W combined. We know that the number of entries computed is in $O(nW)$, but perhaps this version avoids ever reaching this bound. This is not the case.

Discussion(2)

- If $n = W + 1$ and $w_i = 1$ for all i, then the total number of entries computed is about

$$1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2} = \frac{(W+1)(n+1)}{2}.$$

- The second equality drives from the fact that $n = W + 1$ in this instance. Therefore, this bound is reached for arbitrarily large values of n and W, which means the worst-case # of entries computed is in $\Theta(nW)$.
- Combining our two results, the worst-case # of entries computed is in $O(minimum(2^n, nW))$.

Discussion(3)

- We do not need to create the entire array to implement the algorithm. Instead, we can store just the entries that are needed. The entire array exists only implicitly. If the algorithm is implemented in this manner. The worst-case memory usage has these same bounds.
- We could write a divide-and-conquer algorithm using the expression for P[i][w] that was used to develop the dynamic programming algorithm. For this algorithm the worst-case number of entries computed is also $\Theta(2^n)$.
- The main advantage of the dynamic programming algorithm is the additional bound. Indeed, this bound is obtained of the fundamental difference between dynamic programming and divide-and-conquer. That is, dynamic programming does not process the same instance more than once. The bound in terms of nW is very significant when W is not large in comparison with n.