Greedy vs Dynamic Programming Approach

Outline

- Compare the methods
- Knapsack problem
    - Greedy algorithms for 0/1 knapsack
    - An approximation algorithm for 0/1 knapsack
    - Optimal greedy algorithm for knapsack with fractions
    - A dynamic programming algorithm for 0/1 knapsack

Greedy Approach VS Dynamic Programming (DP)

- Greedy and Dynamic Programming are methods for solving optimization problems
- Greedy algorithms are usually more efficient than DP solutions.
- However, often you need to use dynamic programming since the optimal solution cannot be guaranteed by a greedy algorithm.
- DP provides efficient solutions for *some* problems for which a brute force approach would be very slow.
- To use Dynamic Programming we need to show that the principle of optimality applies to the problem.

The 0/1 Knapsack problem

- A knapsack with weight capacity $W > 0$ is given.
- *A* set S of n items with weights $w_i > 0$ and benefits $b_i > 0$ for $i = 1,\ldots,n$.
- $S = \{ (item_1, w_1, b_1), (item_2, w_2, b_2), \ldots, (item_n, w_n, b_n) \}$
- Find a subset of the items that does not exceed the weight W of the knapsack and maximizes the benefit.

0/1 Knapsack problem

- Determine a subset *A* of { 1, 2, …, *n* } that satisfies the following:

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

- 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 are allowed
  - 0/1 (1 brown pant, 1 green shirt, 1 tablet…)
- Allows putting many items of same type in knapsack
  - 5 pairs of socks
  - 10 gold bricks
- More than one knapsack, etc.

0/1 vs. Fractional Knapsack

- Totally different problems
- 0/1 knapsack: very hard problem
  - The notion of “hard” will be formally defined in theory of NP
- Fractional knapsack: not hard
- Let’s first discuss 0/1 *knapsack* problem and then fractional *knapsack* problem

0/1 Knapsack Problem

Brute force approach

- Generate all 2^n subsets
  - Discard all subsets whose sum of the weights exceed *W (not feasible)*
  - Select the maximum total benefit of the remaining (feasible) subsets
- What is the run time?
  - $\Omega(2^n)$

Example with “brute force”

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

- Subsets:

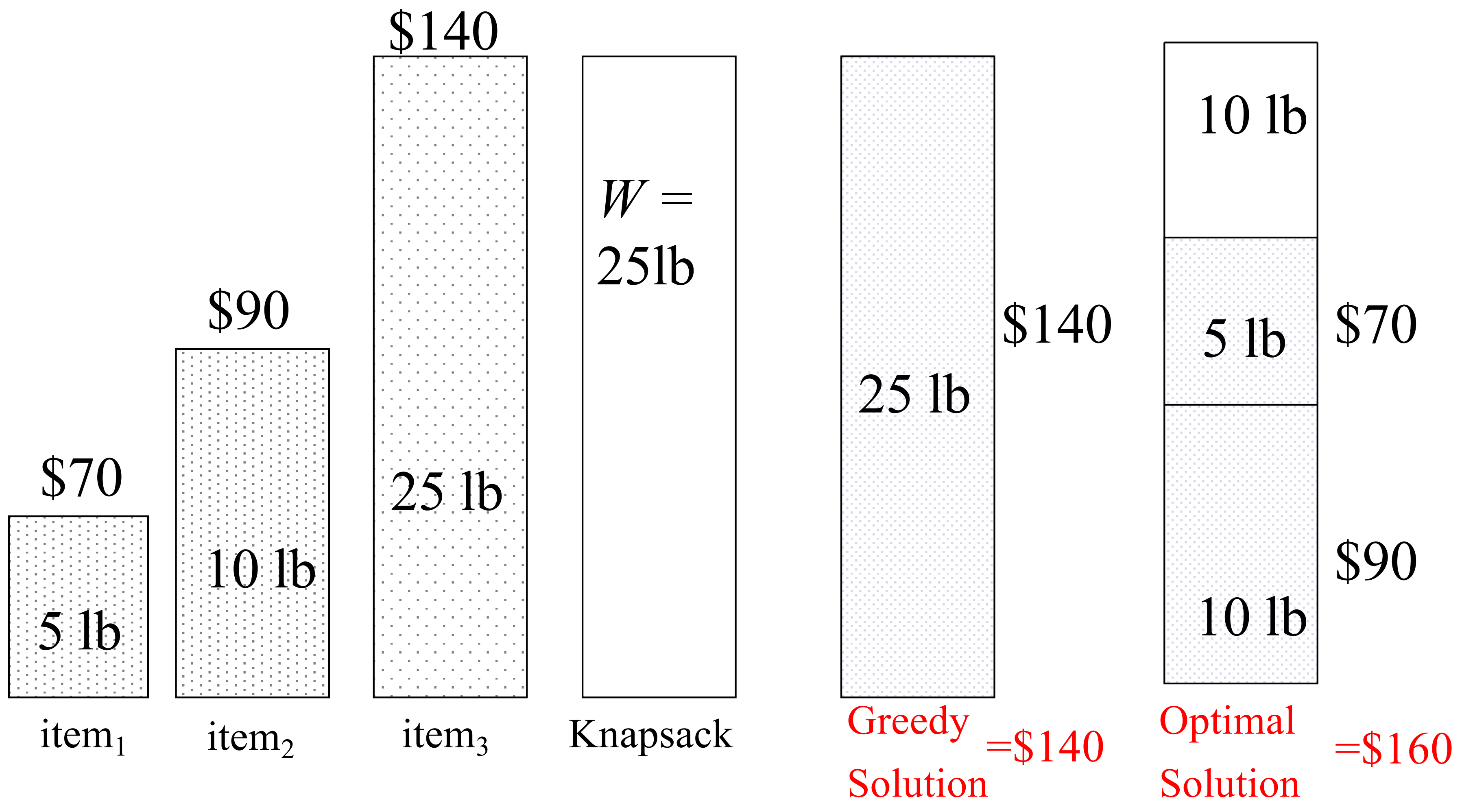
1. {}
2. { ($item_1$, 5, \$70) } Profit=\$70
3. { ($item_2$,10, \$90) } Profit=\$90
4. { ($item_3$, 25, \$140) } Profit=\$140
5. { ($item_1$, 5, \$70), ($item_2$,10, \$90)}. Profit=\$160 ****
6. { ($item_2$,10, \$90), ($item_3$, 25, \$140) } exceeds W
7. { ($item_1$, 5, \$70), ($item_3$, 25, \$140) }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

Greedy approach for 0/1 Knapsack?

- You may not get an optimal solution!
  - 0/1 knapsack is a hard problem!
- See examples in the following slides.

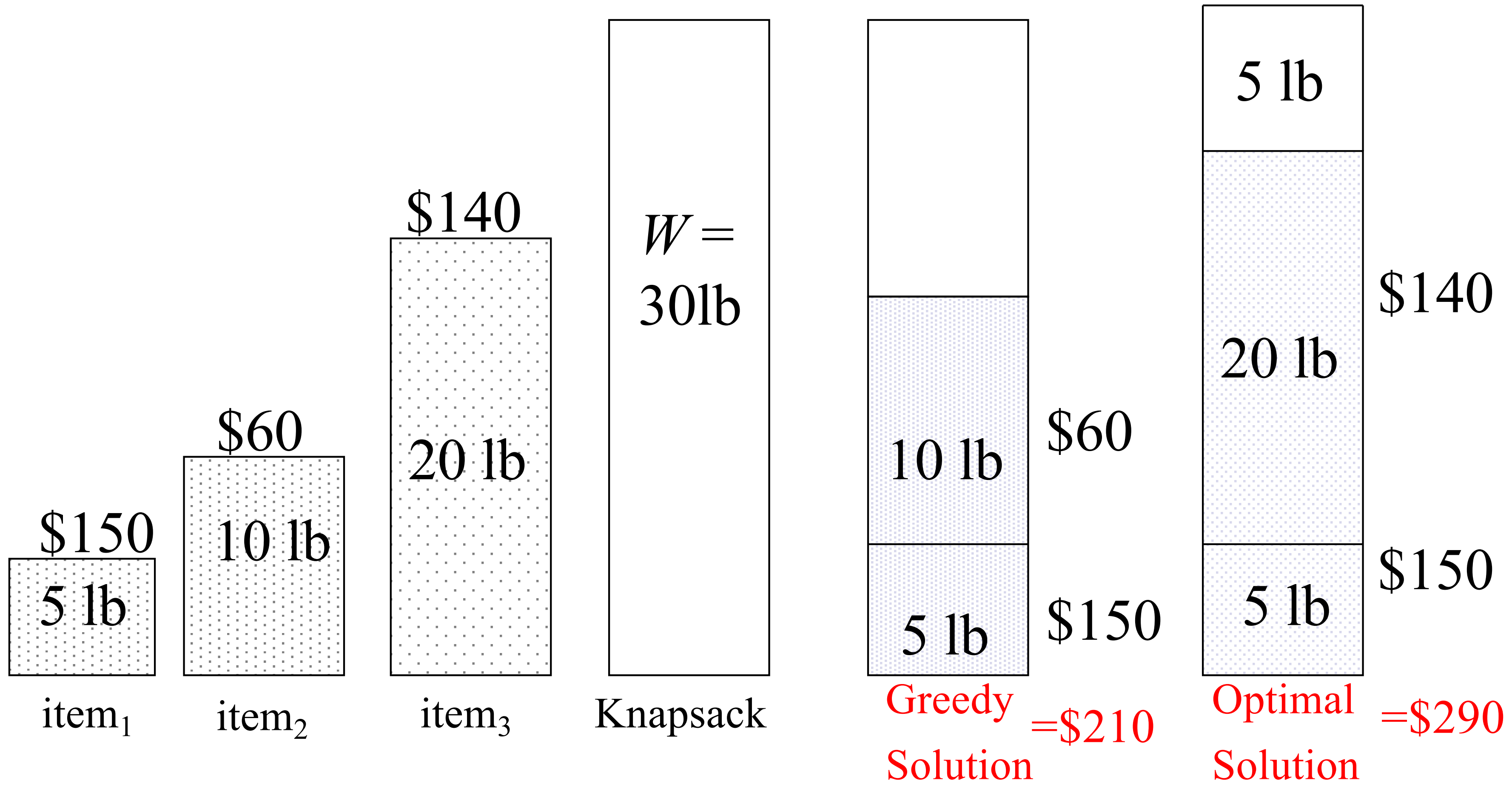
Greedy 1: Max benefit first – Counter example

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



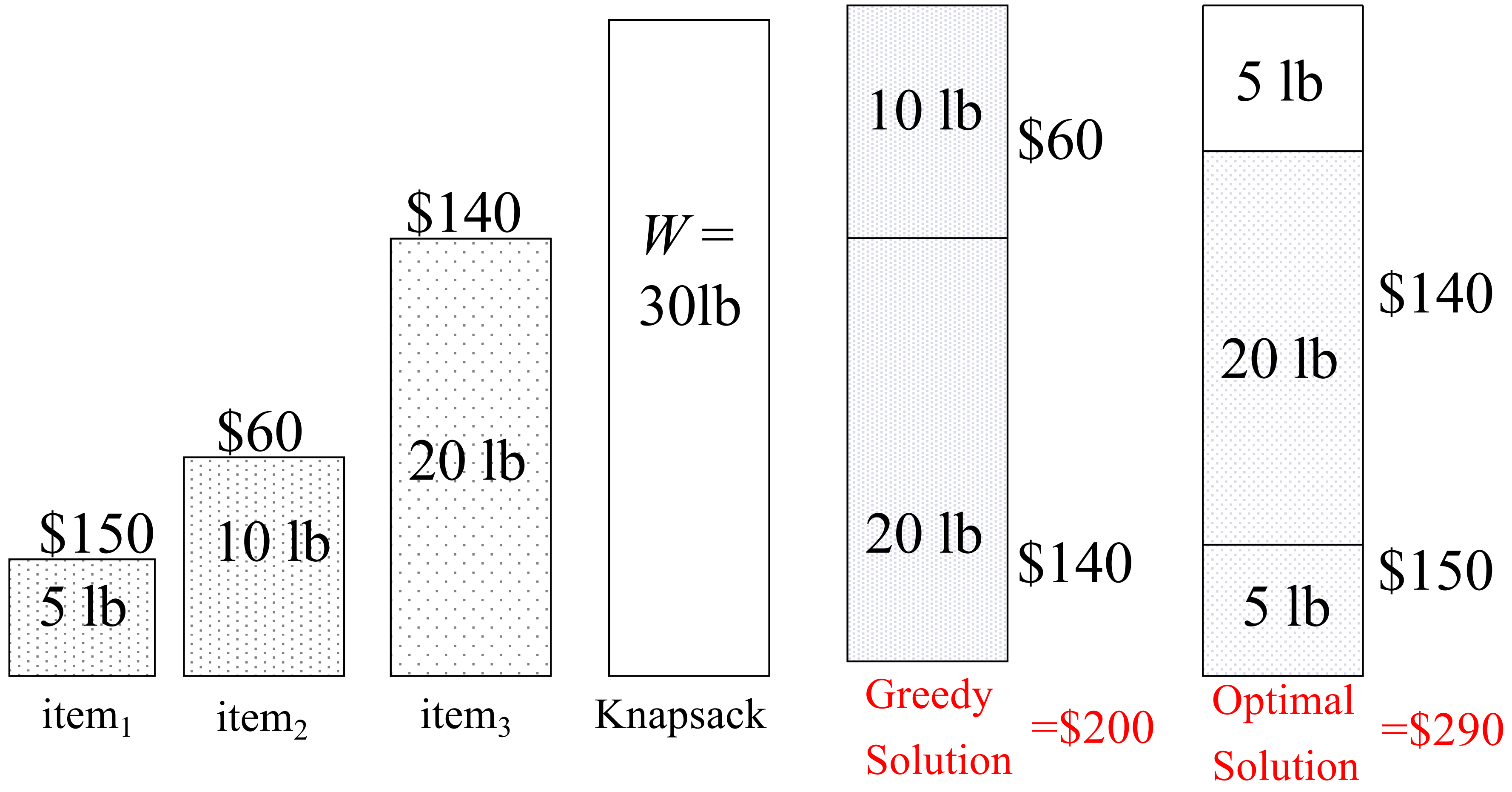
Greedy 2: Minimum weight first – Counter example

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



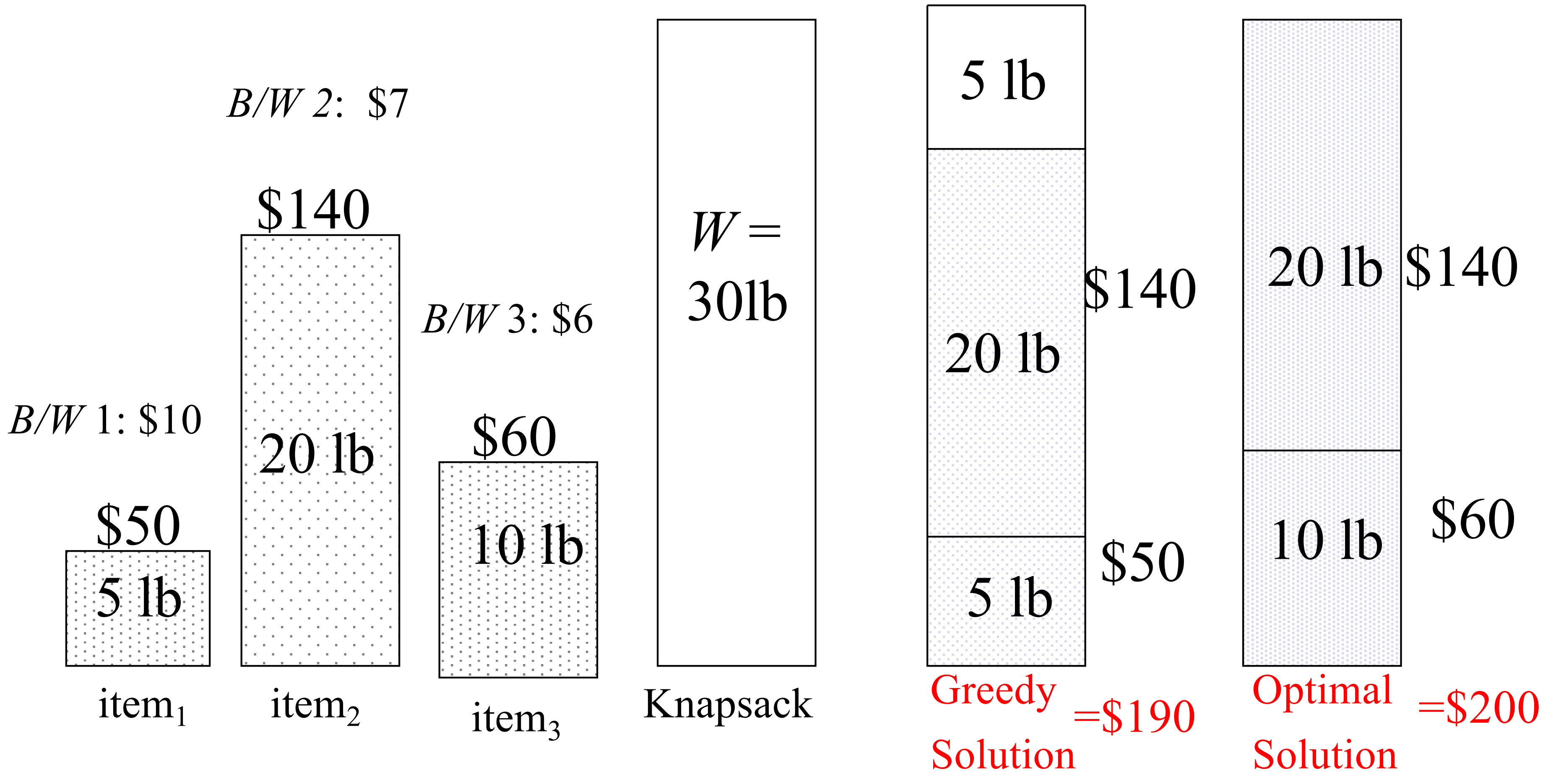
Greedy 3: Max weight first – Counter Example

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



Greedy 4: *Maximum benefit per unit weight* -- Counter Example

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



Approximation algorithms

- Approximation algorithms are not guaranteed to provide an optimal solution, but yields one that is reasonably close to an optimal solution.

- Let AppAlg represent a solution provided by an approximate algorithm. How far is the approximate solution away from the optimum OPT in the worst case?

- Many criteria are used. We use OPT/AppAlg for maximization, and attempt to establish OPT/AppAlg $\leq$ K where K is a constant (AppAlg/OPT for minimization)

Approximation algorithms

- Greedy 4 does not satisfy OPT/AppAlg ≤ K
  - Often greedy4 gives an optimal solution, but for some problem instances the ratio can be very large
- A small modification of greedy4, however, guarantees that OPT/AppAlg ≤ 2
  - This means the approx. alg. produces at least half the optimal benefit
  - This is a big improvement
  - There are better approximation algorithms for knapsack

When Greedy4 fails hopelessly?

- Example: Greedy4, which selects the item with maximum benefit per unit first, provides *boundlessly poorer solution compared to an optimal solution*:
  - Assume a 0/1 knapsack problem with $n = 2$
  - Very large W
  - S = {(item1, 1, \$2), (item 2, W, \$1.5W) }
- The solution to greedy4 has a benefit of \$2
- An optimal solution has a benefit of \$1.5W
- Suppose W=10,000
  - The first solution only produces a profit of \$2
  - The 2nd solution generates a profit of \$15,000!

Approximation Continued

- Let ***BOpt*** denote the optimal benefit for the 0/1 knapsack problem

- Let ***BGreedy4*** be the benefit calculated by greedy4.
  - For last example ***BOpt*** / ***BGreedy4*** = \$1.5*W* / 2
  - Note: *W* can be arbitrarily large

We would like to find a better algorithm ***Alg*** such that ***BOpt*** / ***Alg*** ≤ ***K*** where *K* is a small constant and is independent of the problem instance.

A Better Approximation Algorithm

- Let *maxB* = *max*{ b_i| *i* = 1, …, *n* }
- The approximation algorithm selects, either the solution to *Greedy4*, or only the *item* with benefit *MaxB* depending on *max*{*BGreedy4*, *maxB* }.
- Let *APP* = *max*{*BGreedy4*, *maxB* }
- What is the asymptotic runtime of this algorithm?
- It can be shown that with this modification, the ratio ***BOpt*/ *APP*** $\leq$ 2 (App produces at least half the optimal benefit)

Proof for half of optimal benefit

- Let x* be an optimum solution for the 0/1 Knapsack instance, x be the solution from Greedy4(includes $b_1, b_2, \ldots, b_{k-1}$) and z* is the respective optimum solution for Knapsack with Fraction.
- Let val(x), val(x*) and val(z*) be the benefit for the solution x, x* and z*, and $\alpha \in [0, 1)$.
- Bopt = val(x*) $\leq$ val(z*) = val(x) $+\alpha b_k$ $\leq$ val(x) + b_k $\leq$val(x) +maxB $\leq$ 2max{val(x),maxB} = *2max{BGreedy4, maxB } =2APP.*

Knapsack with fractions

An Optimal Greedy Algorithm for Knapsack with Fractions (KWF)

If a fraction of any item can be chosen, the following algorithm provides the optimal benefit:

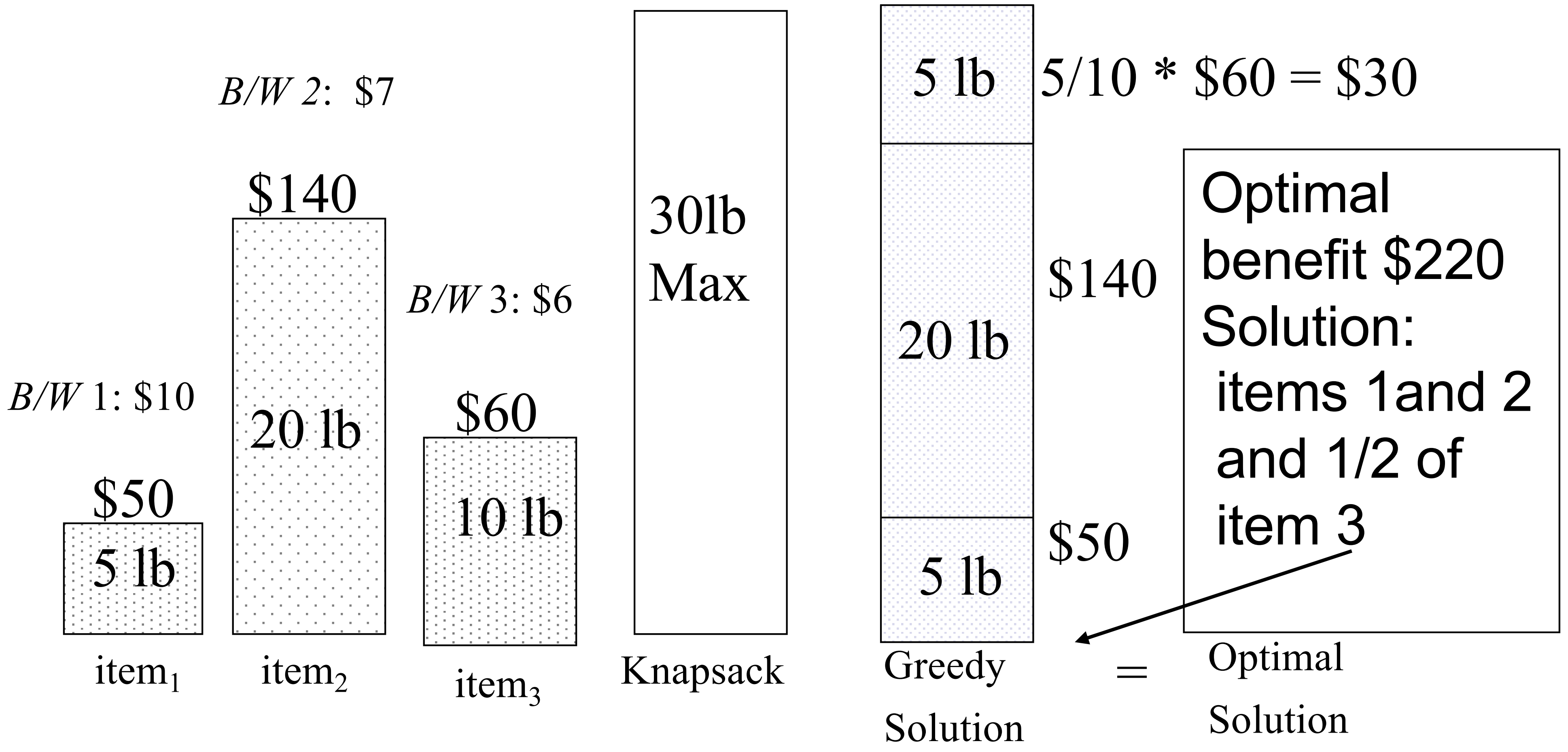
- The greedy algorithm uses the *maximum benefit per unit* selection criteria
  1. Sort items in non-ascending b_i / w_i order
  2. Add items to knapsack in the sorted order until there is no more item or the next item to be added exceeds W
  3. If knapsack is not full yet, fill knapsack with a fraction of next unselected item

KWF

- Let *k* be the index of the last item included in the knapsack. We may be able to include the whole or only a fraction of item *k*
- *Without item k totweight =* $\sum_{i=1}^{k-1} w_i$
- *profitKWF =* $\sum_{i=1}^{k-1} p_i$ + min{(***W*** - *totweight),w*$_k$ } X (*p*$_k$ / *w*$_k$)
- min{(***W*** - *totweight),w*$_k$ } means that we either take the entire item k when the knapsack can include the item without violating the constraint. Or, we fill the knapsack by a fraction of item k.

Example of applying the optimal greedy algorithm to Fractional Knapsack Problem

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



Optimal benefit \$220
Solution:
items 1and 2
and 1/2 of
item 3

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
  – Proof by contradiction

- Notice there may be more than one optimal solution

Proof by contradiction

- Assume that an optimal solution for a fractional knapsack problem chooses a less dense item that does not have max(benefit/weight) ratio. You can then replace the item with the densest item among the ones not included in the knapsack. This will increase the total benefit, contradicting to the assumption that the given solution was optimal.

- Exercise: Also prove that an optimal solution has a full knapsack.

Dynamic programming approach for the *0/1 Knapsack* problem

- Greedy solutions fail to guarantee an optimal solution
- Show the principle of optimality holds
- Discuss the algorithm

Principle of Optimality for 0/1 Knapsack problem

- **Theorem**: 0/1 knapsack satisfies the principle of optimality
- **Proof by contradiction**
  - Assume that item_i is in the **most beneficial subset** that weighs at most W. If we remove item_i from the subset, the remaining subset must be **the most beneficial subset** weighing at most $W - w_i$ of the n -1 remaining items after excluding item_i
  - If the remaining subset after excluding item_i was not the most beneficial one weighing at most $W - w_i$ of the n -1 remaining items, we could find a better solution for this problem and improve the optimal solution. This contradicts to the underlined assumption above.

Dynamic Programming Approach

- A knapsack problem with n items and knapsack weight of *W* is given
- 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 is the smaller problem?
  - Assume a subproblem in which the set of items is restricted to $\{1,\ldots, i\}$ where $i \le n$, and the total weight for the knapsack can not exceed w, where $0 \le w \le W$
  - Let $B[i, w]$ denote the maximum benefit achieved for this problem
  - Our goal is to compute the maximum benefit of the original problem $B[n, W]$
  - We solve the original problem by computing $B[i, w]$ for $i = 0, 1,\ldots, n$ and $\boldsymbol{w = 0,1,\ldots,W}$
  - We need to specify the solution to a larger problem in terms of a smaller one

Recursive formula for the "smaller" 0/1Knapsack Problem only using *item*$_1$ to *item*$_i$ and knapsack weight at most w

1. If there is no item in the knapsack or w is 0, then the benefit is 0
2. If the weight of item$_i$ exceeds the weight w of the knapsack then item$_i$ cannot be included in the knapsack and the maximum benefit is B[i-1, w]
3. Otherwise, the benefit is the maximum achieved by either not including item$_i$ (i.e., B[i-1, w]) or by including item$_i$ (i.e., B[i-1, w-w$_i$] + b$_i$)

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

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

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

Output: $B[n+1, W+1]$

```
for $w \leftarrow 0$ to $W$  // row 0 (empty knapsack)
    $B[0,w] \leftarrow 0$

for $k \leftarrow 1$ to $n$  // rows 1 to n
{
    $B[k, 0] \leftarrow 0$          // element in column 0 (no profit for w = 0)
    for $w \leftarrow 1$ to $W$   // elements in columns 1 to $W$
            if ($w_k \leq w$) and ($B[k-1, w - w_k] + b_k > B[k-1, w]$)
                    then  $B[k, w] \leftarrow B[k-1, w - w_k] + b_k$
                    else  $B[k, w] \leftarrow B[k-1, w]$
}
```

Example:

***W* = 30, *S* = { (i_1 , 5, \$50), (i_2 ,10, \$60), (i_3, 20, \$140) }**

Weight:	0	1	2	3	…	30
MaxProfit { }	0	0	0	0	…	0

Weight:	0	1	2	3	4	5	…	30
MaxProfit { }	0	0	0	0	0	0	…	0
MaxProfit{i_1}	0	0	0	0	0	50	…	50

Example continued
W = 30, S = { (i_1 , 5, \$50), (i_2 ,10, \$60), (i_3, 20, \$140) }

Weight:	0	...	4	5	…	9	10	…	14	15	...	30
MaxProfit { }	0	...	0	0	…	0	0	…	0	0	…	0
MaxProfit{i_1}	0	...	0	50	…	50	50	...	50	50	…	50
MaxProfit{i_1, i_2}	0	…	0	50	…	50	60	…	60	110	…	110

- B[2,10] = max { B[1,10], B[1,10-10] + b_2 }
  = 60
- B[2,15] = max { B[1,15], B[1,15-10] + b_2 }
  = max {50, 50+60}
  = 110

Example continued
W = 30, S = { (i_1 , 5, \$50), (i_2 ,10, \$60), (i_3, 20, \$140) }

Wt:	0...4	5 … 9	10…14	15… 19	20… 24	25…29	30
MaxP{ }	0...0	0 … 0	0 …0	0 … 0	0… 0	0… 0	0
MaxP{i_1}	0...0	50…50	50…50	50… 50	50… 50	50… 50	50
MaxP{i_1, i_2}	0…0	50…50	60…60	110...110	110…	110...	110
110							
MaxP{i_1,i_2,i_3}	0…0	50…50	60…60	110...110	140…140	190…190	200

- B[3,20] = max { B[2,20], B[2,20-20] + b_3 }
  = 140
- B[3,25] = max { B[2,25], B[2,25-20] + 140 }
  = max {110, 50+140}
  = 190
- B[3,30] = max { B[2,30], B[2,30-20] + 140 }
  = 200

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 runs in $\Theta(n W)$. It appears to be linear. But, note that the weight can be arbitrarily large. What if $W = n!$? Then, this algorithm is worse even than the brute force method.
- The algorithm can be improved so that the worst-case number of entries computed is in $\Theta(2^n)$.

A refinement of Dynamic programming approach for the *0/1 Knapsack* problem

- it is not necessary to determine the entries in the *i*th row for every *w* between 1 and *W*.
- In the *n*th row we need only determine *B*[*n*,*W*]. The only entries needed in the (*n*− 1)st row are the ones needed to compute *B*[*n*,*W*].
- The only entries needed in the (*n* − 1)st row are B[n-1,W] and B[n-1,W-w_n].
- After we determine which entries are needed in the *i*th row, we determine which entries are needed in the (*i* − 1)st row using the fact B[i,w] is computed from B[i-1,w] and B[i-1, w-w_i].
- We stop when *n* = 1 or *w* ≤ 0. After determining the entries needed, we do the computations starting with the first row.

Example of the refinement algorithm

- $W = 30$, $S = \{ (i_1 , 5, \$50), (i_2 ,10, \$60), (i_3, 20, \$140) \}$
- Determine entries needed in row 3, we need B[3,30].
- Determine entries needed in row 2 (w_3=20): B[2,30] and B[2,10].
- Determine entries needed in row 1(w_2=10).
  - To compute B[2,30], we need B[1,30] and B[1,20].
  - To compute B[2,10], we need B[1,10] and B[1,0].
- Next we do the computations.
  - Compute row 1 for B[1,0], B[1,10], B[1,20] and B[1,30] using the following formula.
  - $B[1,w] = \begin{cases} maximum(B[0,w], \$50 + B[0,w-5]) & (w_1 = 5) \leq w \\ B[0,w] & (w_1 = 5) > w \end{cases}$

Example of the refinement algorithm

- B[1,0] = \$0, B[1,10]= \$50, B[1,20]= \$50 and B[1,30] =\$50
- Compute row 2:
  - $B[2,10] = \begin{cases} maximum(B[1,10], \$60 + B[1,0]) & (w_2 = 10) \leq 10 \\ B[1,10] & (w_2 = 10) > 10 \end{cases}$

    =\$60
  - $B[2,30] = \begin{cases} maximum(B[1,30], \$60 + B[1,20]) & (w_2 = 10) \leq 30 \\ B[1,30] & (w_2 = 10) > 30 \end{cases}$

    =\$60+\$50=\$110
- Compute row 3:
  - $B[3,30] = \begin{cases} maximum(B[2,30], \$140 + B[2,20]) & (w3 = 20) \leq 30 \\ B[2,30] & (w3 = 20) > 30 \end{cases}$

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

Analysis of refinement algorithm

- This version of the refinement algorithm computes only 7 entries, whereas the original version would have computed 3× 30 = 90 entries.
- The worst case time efficiency.
  - We compute at most 2^j entries in the (n− i)th row.
  - At most the total number of entries computed is

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

  - We can also prove that for the following instance about 2^n entries are computed.

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

  - The worst-case number of entries computed is in $\theta(2^n)$

Analysis of refinement algorithm

- We know that the number of entries computed is in *O* (*nW*), but perhaps this refinement algorithm avoids ever reaching this bound.
- For *n*=W+1 and w_i =1, the total number of entries computed is $1+2+3+\cdots+n=\frac{n(n+1)}{2}=\frac{(W+1)(n+1)}{2}.$
- This means the worst-case time efficiency is in $\theta(\mathrm{nW})$
- the worst-case number of entries computed is in

$$O\left(minimum\left(2^n, nW\right)\right).$$

- *No one has ever found a 0/1 knapsack algorithm whose worst case time is better than exponential. No one has proven that developing such an algorithm is impossible either.*

Construct an Optimal Solution

- We find the maximum profit that can be achieved.
- We need another procedure to identify a set of items that are included in the knapsack to achieve the maximum profit.
- We now show such a procedure to construct an optimal solution from the table *B*.

The Basic Idea

- We start from $B[n, C]$ and work backward to determine when an item is added to the knapsack to achieve maximum profit.
- From the recurrence, we can see that only when $B[i, w] = B[i-1, w-w_i] + b_i$ or when $B[i, w] \neq B[i-1, w]$, another item (item$_i$) is added to the knapsack.

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

Algorithm

```
Print-Op-Items($B$) // $B = B[0 .. n, 0 .. W]$
    IS = $\phi$ // IS will store an optimal item set, empty initially
    $i = n$
    $w = W$
    while $i > 0$ and $w > 0$
        if $B[i, w] \neq B[i - 1, w]$
            IS = IS $\cup$ { item$_i$ }
            $w = w - w_i$
            $i = i - 1$
        else
            $i = i - 1$
    return IS
```

Example (1)

- Item 4 ($i = 4$): $(w_4, b_4) = (5, 6)$
- $w = 8$

i \ w	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9
4	0	0	3	4	5	7	8	9	10 ← $B[4, 8]$

- Since $B[4, 8] = 10 \neq B[3, 8] = 9$, add item 4 to knapsack.
- $w = w - w_4 = 3$ and $i = i - 1 = 3$.

Example (2)

- Item 3 ($i = 3$): $(w_3, b_3) = (4, 5)$
- $w = 3$

i \ w	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9
4	0	0	3	4	5	7	8	9	10

- Since $B[3, 3] = 4 = B[2, 3]$, do not add item 3 to knapsack.
- w remains 3, $i = i - 1 = 2$.

Example (3)

- Item 2 (i = 2): (w_2, b_2) = (3, 4)
- w = 3

i \ w	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	3	3	3	3	3	3	3
2	0	0	3	4	4	7	7	7	7
3	0	0	3	4	5	7	8	9	9
4	0	0	3	4	5	7	8	9	10

- Since $B[2, 3] = 4 \neq B[1, 3] = 3$, add item 2 to knapsack.
- $w = w - w_2 = 0$ and $i = i - 1 = 1$. The procedure stops.
- Final result: IS = {item 2, item 4}.