GREEDY ALGORITHM

Greedy algorithm is a group of algorithms that have one common characteristic i.e

Local Optimality :Making the best choice locally at each step without considering future plans.

Thus, the essence of greedy algorithm is a choice function: given a set of options, choose the current best option.

It is applicable for Optimization Problem

Components of Greedy Algorithm:

Objective Function :Maximize or Minimize based on given Problem

Generating Multiple candidate solution :A candidate may have N inputs or candidate solutions .All possible solutions may not be optimum

Selection Procedure :Selection must be based on some greedy local optimal criteria.

Feasibility Check :To check if the selected item is feasible as per the constraint

Solution Check: This checks whether the partial solutions together constitute a global solution for the given problem.

Example Problems :

Fractional Knapsack Algorithm

Coin Exchange Problem(Greedy may not give optimal solution)

Scheduling Problem

Huffman Encoding for Data Compression

Minimum spanning Tree

GREEDY ALGORITHM

```
Algorithm Greedy(a, n)
// a[1 : n] contains the n inputs.
{
    solution := ∅; // Initialize the solution.
    for i := 1 to n do
    {
        x := Select(a);
        if Feasible(solution, x) then
            solution := Union(solution, x);
    }
    return solution;
}
```

GREEDY –FRACTIONAL/CONTINUOUS KNAPSACK PROBLEM

The fractional knapsack problem is defined as:

- Given a list of n objects say $\{I_1, I_2 \dots \dots, I_n\}$ and a Knapsack (or bag).
- Capacity of Knapsack is M.
- Each object I_i has a weight w_i and a profit of p_i .
- If a fraction x_i (where $x_i \in \{0, \dots .,1\}$) of an object I_i is placed into a knapsack then a profit of $p_i x_i$ is earned.

The **problem** (or Objective) is to fill a knapsack (up to its maximum capacity M) which maximizes the total profit earned.

$$Maximize\ (the\ profit) \sum_{i=1} p_i x_i \ ; \ subjected\ to\ the\ constraints$$

$$\sum_{i=1}^{n} w_i x_i \le \mathrm{M\ and}\ \ x_i \in \{0, \dots ,1\}, 1 \le \mathrm{i} \le \mathrm{n}$$

Example 4.1 Consider the following instance of the knapsack problem: $n = 3, m = 20, (p_1, p_2, p_3) = (25, 24, 15)$, and $(w_1, w_2, w_3) = (18, 15, 10)$. Four feasible solutions are:

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
1.	(1/2, 1/3, 1/4)	16.5	24.25
2.	(1, 2/15, 0)	20	28.2
3.	(0, 2/3, 1)	20	31
4.	(0, 1, 1/2)	20	31.5

$$\text{maximize} \sum_{1 \le i \le n} p_i x_i$$

$$\text{subject to} \sum_{1 \le i \le n} w_i x_i \le m$$

$$\text{and } 0 < x_i < 1, \quad 1 < i < n$$

FRACTIONAL KNAPSACK PROBLEM-EXAMPLE

$Number\ of\ objects;\ n = 3$
Capacity of Knapsack; M=20
$(p_1, p_2, p_3) = (25,24,15)$
$(w_1, w_2, w_3) = (18,15,10)$

To solve this problem, Greedy method may apply any one of the following strategies:

- From the remaining objects, select the object with maximum profit that fit into the knapsack.
- From the remaining objects, select the object that has minimum weight and also fits into knapsack.
- From the remaining objects, select the object with maximum p_i/w_i that fits into the knapsack.

FRACTIONAL KNAPSACK PROBLEM-EXAMPLE

Approach	(x_1, x_2, x_3)	$\sum_{i=1}^{3} w_i x_i$	$\sum_{i=1}^{3} p_i x_i$
1	$(1, \frac{2}{15}, 0)$	18+2+0=20	28.2
2	$< 0, \frac{2}{3}, 1 >$	0+10+10=20	31.0
3	$< 0, 1, \frac{1}{2} >$	0+15+5=20	31.5

FRACTIONAL KNAPSACK PROBLEM-ALGORITHM

```
Greedy Fractional-Knapsack (P[1..n], W[1..n], X [1..n], M)
        /* P[1..n] and W[1..n] contains the profit and weight of the n-objects ordered such
        that
        X[1..n] is a solution set and M is the capacity of KnapSack*/
{
        For i ← 1 to n do
                X[i] ← 0
                profit ← 0      //Total profit of item filled in Knapsack
                weight ← 0      // Total weight of items packed in KnapSack
                i←1
        While (Weight < M) // M is the Knapsack Capacity
        {
                if (weight + W[i] ≤ M)
                        X[i] = 1
                        weight = weight + W[i]
                else
                        X[i] = (M-wright)/w[i]
                        weight = M
        Profit = profit = profit + p [i]*X[i]
        i++;
}//end of while
}//end of Algorithm
```

FRACTIONAL KNAPSACK PROBLEM-ALGORITHM

Running time of Knapsack (fractional) problem:

Sorting of n items (or objects) in decreasing order of the ratio p_i/w_i takes $O(nlogn)$ $time$. Since this is the lower bound for any comparison based sorting algorithm. Line 6 of ***Greedy Fractional-Knapsack*** takes $O(n)$ time. Therefore, the total time including sort is $O(nlogn)$.

FRACTIONAL KNAPSACK PROBLEM-EXAMPLE

Example: 1: Find an optimal solution for the knapsack instance n=7 and M=15 ,

$(p_1, p_2 \dots\dots, p_7) = (10, 5, 15, 7, 6, 18, 3)$
$(w_1, w_2, \dots\dots, w_7) = (2, 3, 5, 7, 1, 4, 1)$

$$\left(\frac{p_1}{w_1}, \frac{p_2}{w_2}, \dots, \frac{p_7}{w_7}\right) = (5,1.67,3,1,6,4.5,3).$$

Approach	$(x_1, x_2, x_3, x_4, x_5, x_6, x_7)$	$\sum_{i=1}^{7} w_i x_i$	$\sum_{i=1}^{7} p_i x_i$
Selection of object in decreasing order of the ratio p_i/w_i	$(1, \frac{2}{3}, 1,0,1,1,1)$	1+2+4+5+1+2 =15	6+10+18+15+3+3.33 =55.33

DYNAMIC PROGRAMMING

- » A metatechnique, not an algorithm (like divide & conquer)
- » The word "programming" is historical and predates computer programming

◆ Use when problem breaks down into recurring small subproblems

Dynamic Programming

- It is used when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*).
- Algorithm finds solutions to subproblems and stores them in memory for later use.
- More efficient than "*brute-force methods*", which solve the same subproblems over and over again.
- Basic idea:
  - » Optimal substructure: optimal solution to problem consists of optimal solutions to subproblems
  - » Overlapping subproblems: few subproblems in total, many recurring instances of each
  - » Solve bottom-up, building a table of solved subproblems that are used to solve larger ones
- Variations:
  - » "Table" could be 3-dimensional, triangular, a tree, etc.

Knapsack problem

There are two versions of the problem:

1. "0-1 knapsack problem" and
2. "Fractional knapsack problem"

1. Items are indivisible; you either take an item or not. Solved with *dynamic programming*

2. Items are divisible: you can take any fraction of an item. Solved with a *greedy algorithm*.
    - We have already seen this version

0-1 Knapsack problem

- Given a knapsack with maximum capacity W, and a set S consisting of n items
- Each item i has some weight w_i and benefit value b_i (all w_i , b_i and W are integer values)
- Problem: How to pack the knapsack to achieve maximum total value of packed items?

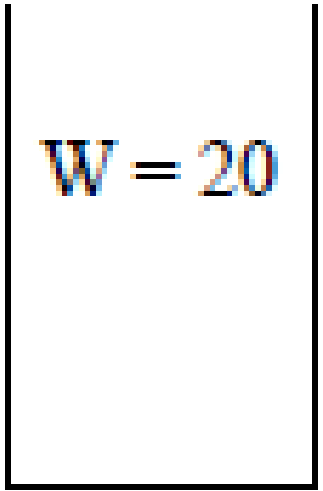
apsack problem (Review)

Given some items, pack the knapsack to get the maximum total value. Each item has some weight and some value. Total weight that we can carry is no more than some fixed number W. So we must consider weights of items as well as their value.

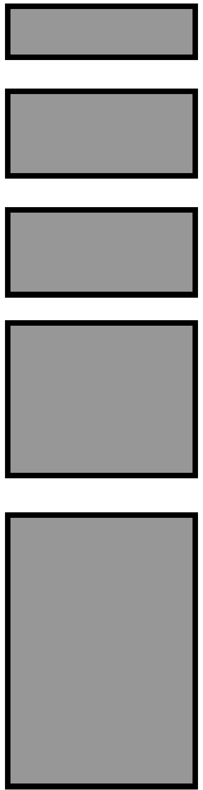
Item #	Weight	Value
1	1	8
2	3	6
3	5	5

0-1 Knapsack problem: a picture

This is a knapsack
Max weight: W = 20



Items



Weight w_i	Benefit value b_i
2	3
3	4
4	5
5	8
9	10

0-1 Knapsack problem

- Problem, in other words, is to find

$$\max \sum_{i \in T} b_i \text{ subject to } \sum_{i \in T} w_i \leq W$$

- The problem is called a "*0-1*" problem, because each item must be entirely accepted or rejected.
- In the "*Fractional Knapsack Problem,*" we can take fractions of items.

0-1 Knapsack problem: brute-force approach

Let's first solve this problem with a straightforward algorithm

- Since there are n items, there are 2^n possible combinations of items.
- We go through all combinations and find the one with maximum value and with total weight less or equal to W
- Running time will be $O(2^n)$

Knapsack problem: e-force approach

- Can we do better?
- Yes, with an algorithm based on dynamic programming
- We need to carefully identify the subproblems

Let's try this:
If items are labeled $1..n$, then a subproblem would be to find an optimal solution for $S_k = \{items\ labeled\ 1, 2, .. k\}$

Defining a Subproblem

If items are labeled *1..n*, then a subproblem would be to find an optimal solution for S_k = *{items labeled 1, 2, .. k}*

- This is a reasonable subproblem definition.
- The question is: can we describe the final solution (S_n) in terms of subproblems (S_k)?
- Unfortunately, we <u>can't</u> do that.

Defining a Subproblem

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=3$ $b_4=4$?	

Max weight: $W = 20$

For S_4:

Total weight: 14;

Maximum benefit: 20

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_5=9$ $b_5=10$	

For S_5:

Total weight: 20

Maximum benefit: 26

Item #	Weight w_i	Benefit b_i
1	2	3
2	3	4
3	4	5
4	5	8
5	9	10

Solution for S_4 is not part of the solution for S_5!!!

- As we have seen, the solution for S_4 is not part of the solution for S_5
- So our definition of a subproblem is flawed and we need another one!
- Let's add another parameter: w, which will represent the exact weight for each subset of items
- The subproblem then will be to compute $B[k,w]$

Recursive formula for subproblems:

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w],\ B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

It means, that the best subset of S_k that has total weight w is:

1) the best subset of S_{k-1} that has total weight w, **or**
2) the best subset of S_{k-1} that has total weight w-w_k plus the item k

Recursive Formula

$$B[k,w]=\begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.
- First case: $w_k > w$. Item k can't be part of the solution, since if it was, the total weight would be $> w$, which is unacceptable.
- Second case: $w_k \le w$. Then the item k can be in the solution, and we choose the case with greater value.

0-1 Knapsack Algorithm

```
for w = 0 to W
   B[0,w] = 0
for i = 1 to n
   B[i,0] = 0
for i = 1 to n
   for w = 0 to W
        if w_i <= w // item i can be part of the solution
              if b_i + B[i-1,w-w_i] > B[i-1,w]
                    B[i,w] = b_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
```

Running time

```
for w = 0 to W                    O(W)
   B[0,w] = 0
for i = 1 to n
   B[i,0] = 0
for i = 1 to n                    Repeat n times
   for w = 0 to W                 O(W)
      < the rest of the code >
```

What is the running time of this algorithm?

$O(n*W)$

Remember that the brute-force algorithm takes $O(2^n)$

Example

Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)
Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

Example (2)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1						
2						
3						
4						

```
for w = 0 to W
    B[0,w] = 0
```

Example (3)

i\W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

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

Example (4)

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					

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

```
if wi <= w // item i can be part of the solution
    if bi + B[i-1,w-wi] > B[i-1,w]
        B[i,w] = bi + B[i-1,w- wi]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```


Example (5)

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					

$i=1$
$b_i=3$
$w_i=2$
$w=2$
$w-w_i=0$

```
if wi <= w // item i can be part of the solution
    if bi + B[i-1,w-wi] > B[i-1,w]
        B[i,w] = bi + B[i-1,w- wi]
    else
        B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```

Example (6)

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
b_i=3
w_i=2
w=3
w-w_i =1

```
if wi <= w // item i can be part of the solution
    if bi + B[i-1,w-wi] > B[i-1,w]
      B[i,w] = bi + B[i-1,w- wi]
    else
      B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```

Example (7)

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
b_i=3
w_i=2
w=4
w-w_i =2

```
if wi <= w // item i can be part of the solution
    if bi + B[i-1,w-wi] > B[i-1,w]
      B[i,w] = bi + B[i-1,w- wi]
    else
      B[i,w] = B[i-1,w]
else B[i,w] = B[i-1,w]  // wi > w
```

Example (8)

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
b_i=3
w_i=2
w=5
w-w_i =3

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_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
```

Example (9)

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
b_i=4
w_i=3
w=1
w-w_i =-2

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_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
```

Example (10)

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					

$i=2$
$b_i=4$
$w_i=3$
$w=2$
$w-w_i=-1$

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
      B[i,w] = b_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

Example (11)

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					

$i=2$
$b_i=4$
$w_i=3$
$w=3$
$w-w_i=0$

if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
      B[i,w] = b_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

Example (12)

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					

$i=2$
$b_i=4$
$w_i=3$
$w=4$
$w-w_i=1$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_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

Example (13)

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					

$i=2$
$b_i=4$
$w_i=3$
$w=5$
$w-w_i=2$

if w_i <= w // item i can be part of the solution
  if b_i + B[i-1,w-w_i] > B[i-1,w]
    B[i,w] = b_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

Example (14)

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					

$i=3$
$b_i=5$
$w_i=4$
$w= 1..3$

if w_i <= w // item i can be part of the solution
 if b_i + B[i-1,w-w_i] > B[i-1,w]
  B[i,w] = b_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

Example (15)

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					

$i=3$
$b_i=5$
$w_i=4$
$w= 4$
$w- w_i=0$

if w_i <= w // item i can be part of the solution
 if b_i + B[i-1,w-w_i] > B[i-1,w]
  B[i,w] = b_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

Example (16)

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					

$i=3$

$b_i=5$

$w_i=4$

$w= 5$

$w- w_i=1$

if w_i <= w // item i can be part of the solution
    if $b_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = b_i + B[i-1,w- w_i]$
    else
        B[i,w] = B[i-1,w]
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Example (17)

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	

$i=4$

$b_i=6$

$w_i=5$

$w= 1..4$

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

Example (18)

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

$i=4$
$b_i=6$
$w_i=5$
$w= 5$
$w- w_i=0$

```
if w_i <= w // item i can be part of the solution
    if b_i + B[i-1,w-w_i] > B[i-1,w]
        B[i,w] = b_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
```

Comments

- This algorithm only finds the max possible value that can be carried in the knapsack
  - » I.e., the value in B[n,W]
- To know the items that make this maximum value, an addition to this algorithm is necessary.

How to find actual Knapsack Items

- All of the information we need is in the table.
- $B[n,W]$ is the maximal value of items that can be placed in the Knapsack.
- Let i=n and k=W

  if $B[i,k] \neq B[i-1,k]$ then

  mark the i^{th} item as in the knapsack

  $i = i-1$, $k = k-w_i$

  else

  $i = i-1$ // Assume the i^{th} item is <u>not</u> in the knapsack

  // Could it be in the optimally packed knapsack?

Finding the Items

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
k= 5
b_i=6
w_i=5
$B[i,k] = 7$
$B[i-1,k] = 7$

```
i=n, k=W
while i,k > 0
    if B[i,k] ≠ B[i–1,k] then
        mark the ith item as in the knapsack
        i = i–1, k = k-wi
    else
        i = i–1
```

Finding the Items (2)

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
k= 5
b_i=6
w_i=5
$B[i,k] = 7$
$B[i-1,k] = 7$

```
i=n, k=W
while i,k > 0
    if B[i,k] ≠ B[i−1,k] then
        mark the ith item as in the knapsack
        i = i−1, k = k-w_i
    else
        i = i−1
```

Finding the Items (3)

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=3
k= 5
b_i=6
w_i=4
$B[i,k] = 7$
$B[i-1,k] = 7$

```
i=n, k=W
while i,k > 0
    if B[i,k] ≠ B[i−1,k] then
        mark the ith item as in the knapsack
        i = i−1, k = k-w_i
    else
        i = i−1
```

Items:

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

Finding the Items (4)

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=2
k= 5
b_i=4
w_i=3
$B[i,k] = 7$
$B[i-1,k] = 3$
$k - w_i = 2$

```
i=n, k=W
while i,k > 0
    if B[i,k] ≠ B[i–1,k] then
        mark the i^th item as in the knapsack
        i = i–1, k = k-w_i
    else
        i = i–1
```

Items:

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

Finding the Items (5)

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=1
k= 2
b_i=3
w_i=2
$B[i,k] = 3$
$B[i-1,k] = 0$
$k - w_i = 0$


```
i=n, k=W
while i,k > 0
    if B[i,k] ≠ B[i–1,k] then
        mark the i^th item as in the knapsack
        i = i–1, k = k-w_i
    else
        i = i–1
```

Finding the Items (6)

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=0
k= 0

The optimal knapsack should contain {1, 2}

```
i=n, k=W
while i,k > 0
    if B[i,k] ≠ B[i−1,k] then
        mark the nth item as in the knapsack
        i = i−1, k = k-w_i
    else
        i = i−1
```

Finding the Items (7)

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

The optimal knapsack should contain {1, 2}

```
i=n, k=W
while i,k > 0
    if B[i,k] ≠ B[i−1,k] then
        mark the nth item as in the knapsack
        i = i−1, k = k-w_i
    else
        i = i−1
```

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, excluding item j

Solving The Knapsack Problem

- The optimal solution to the fractional knapsack problem can be found with a greedy algorithm
  - » *Do you recall how?*
  - » Greedy strategy: take in order of dollars/pound
- The optimal solution to the 0-1 problem cannot be found with the same greedy strategy
  - » 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 can be solved with a dynamic programming approach

Memoization

- *Memoization* is another way to deal with overlapping subproblems in dynamic programming
  - » After computing the solution to a subproblem, store it in a table
  - » Subsequent calls just do a table lookup
- With memoization, we implement the algorithm recursively:
  - » If we encounter a subproblem we have seen, we look up the answer
  - » If not, compute the solution and add it to the list of subproblems we have seen.
- Must useful when the algorithm is easiest to implement recursively
  - » Especially if we do not need solutions to all subproblems.

Conclusion

- Dynamic programming is a useful technique of solving certain kind of problems
- When the solution can be recursively described in terms of partial solutions, we can store these partial solutions and re-use them as necessary (memoization)
- Running time of dynamic programming algorithm vs. naïve algorithm:
  - 0-1 Knapsack problem: **$O(W*n)$** vs. **$O(2^n)$**