Design and Analysis of Algorithms

CSE 5311

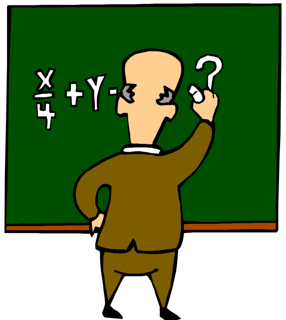
Lecture 13 Dynamic Programming

Junzhou Huang, Ph.D.

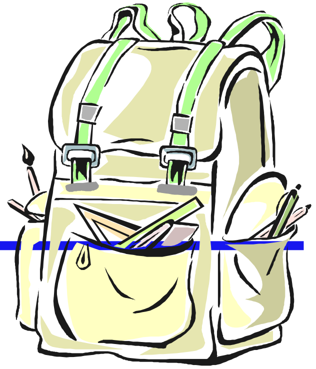
Department of Computer Science and Engineering

The General Dynamic Programming Technique

- Applies to a problem that at first seems to require a lot of time (possibly exponential), provided we have:
  - **Subproblem optimality:** the global optimum value can be defined in terms of optimal subproblems
  - **Subproblem overlap:** the subproblems are not independent, but instead they overlap (hence, should be constructed bottom-up).

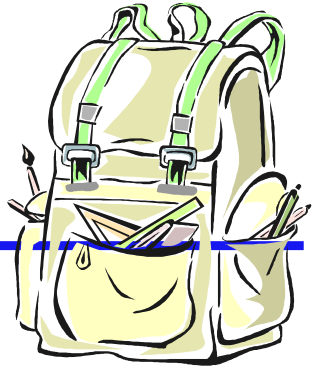


The 0/1 Knapsack Problem

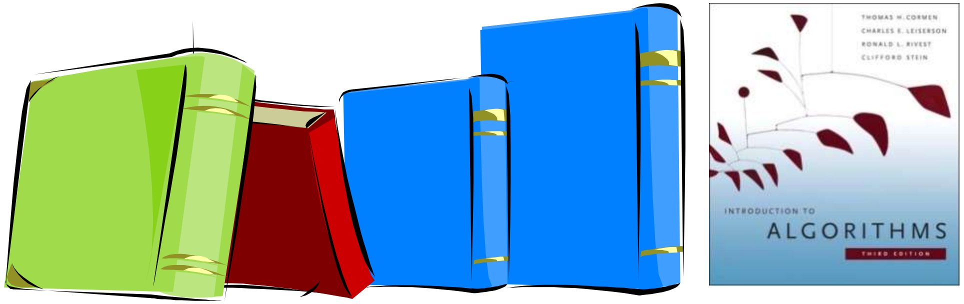


- Given: A set S of n items, with each item i having
  - w_i - a positive weight
  - b_i - a positive benefit
- Goal: Choose items with maximum total benefit but with weight at most W.
- If we are **not** allowed to take fractional amounts, then this is the **0/1 knapsack problem**.
  - In this case, we let T denote the set of items we take
  - Objective: maximize $$\sum_{i \in T} b_i$$
  - Constraint: $$\sum_{i \in T} w_i \leq W$$

Example

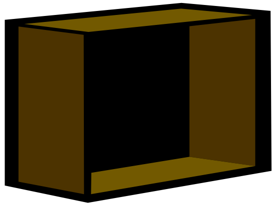


- Given: A set S of n items, with each item i having
  - b_i - a positive “benefit”
  - w_i - a positive “weight”
- Goal: Choose items with maximum total benefit but with weight at most W.



Items:	1	2	3	4	5
Weight:	4 in	2 in	2 in	6 in	2 in
Benefit:	$20	$3	$6	$25	$80

“knapsack”

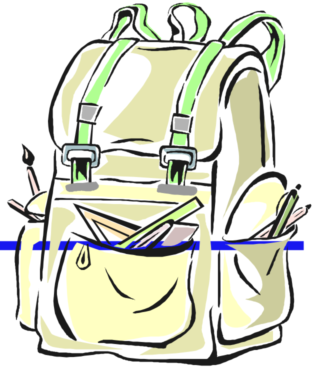


box of width 9 in

Solution:

- item 5 ($80, 2 in)
- item 3 ($6, 2in)
- item 1 ($20, 4in)

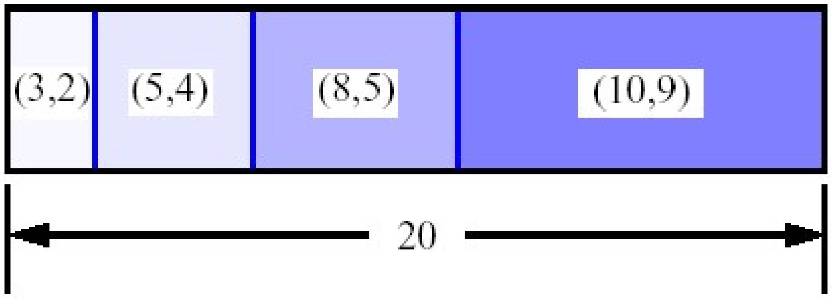
A 0/1 Knapsack Algorithm, First Attempt



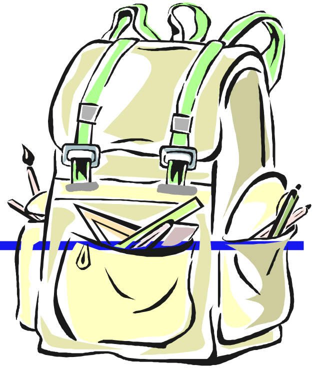
- S_k: Set of items numbered 1 to k.
- Define B[k] = best selection from S_k.
- Problem: does not have subproblem optimality:
  - Consider set S={(3,2),(5,4),(8,5),(4,3),(10,9)} of (benefit, weight) pairs and total weight W = 20

Best for S_4:

Best for S_5:

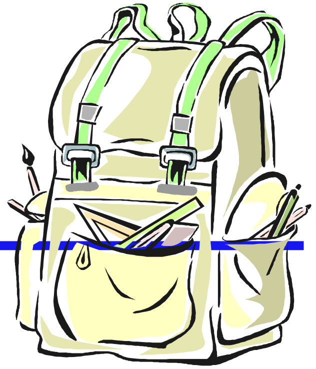


0-1 Knapsack problem: brute-force approach



- ***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 $O(2^n)$
- ***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}
- ***Define a subproblem***
  - 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 can't do that.

Define a Subproblem



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

Max weight:W = 20
For S_4: Total weight: 14;
Maximum benefit: 20

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

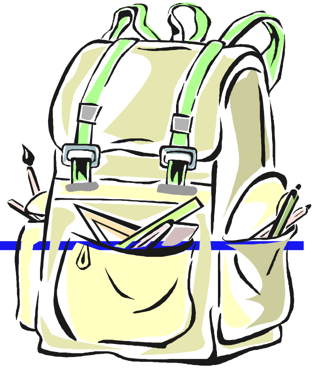
For S_5: Total weight: 20
Maximum benefit: 26

S_5 S_4

Item	Weight	Benefit
1	2	3
2	4	5
3	5	8
4	3	4
5	9	10

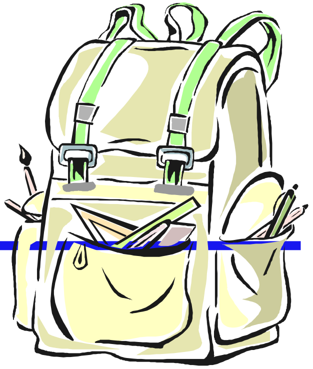
Solution for S_4 is not part of the solution for S_5!!!

Define a Subproblem



- 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 Formulation

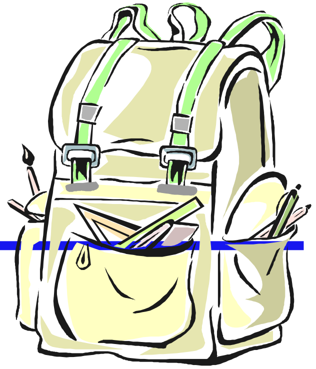


- S_k: Set of items numbered 1 to k.
- Define B[k,w] to be the best selection from S_k with weight at most w
- Good news: this does have subproblem optimality.

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

- I.e., the best subset of S_k with weight at most w is either
  - the best subset of S_{k-1} with weight at most w or
  - the best subset of S_{k-1} with weight at most $w-w_k$ plus item k
- Two cases (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: $w_k < w$. 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] // $w_i > w$
```

O(W)

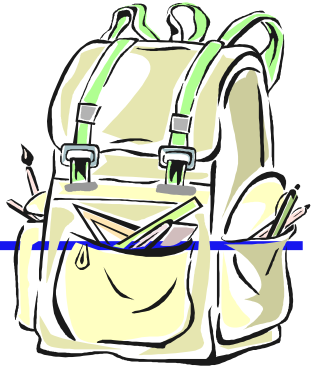
Repeat n times

O(W)

What is the running time of this algorithm? O(n*W)

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

Example

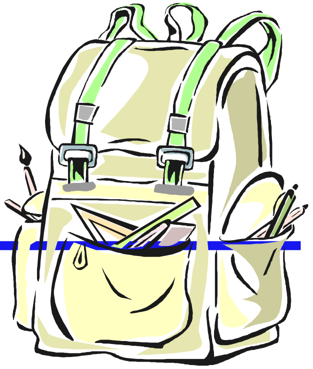


Let's run our algorithm on the following data:

n = 4 (# of elements)
W = 5 (max weight)

Elements (weight, benefit):
(2,3), (3,4), (4,5), (5,6)

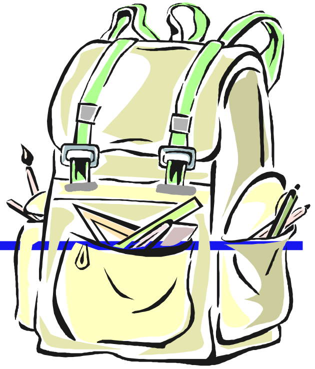
Example (2)



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
```

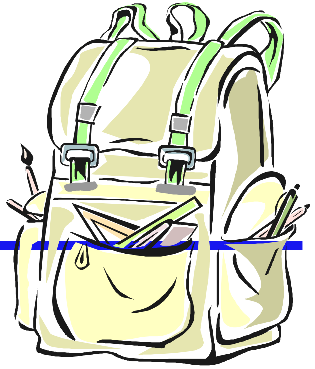
Example (3)



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)



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

Items:

$\boxed{\text{1: (2,3)}}$

2: (3,4)

3: (4,5)

4: (5,6)

```
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 (5)

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					

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i$ $=0$

Items:

1: (2,3)

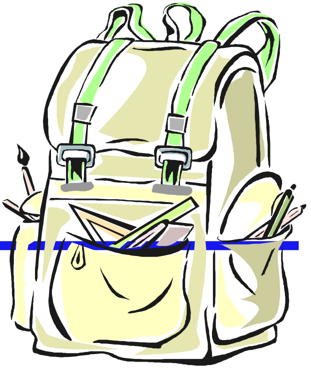
2: (3,4)

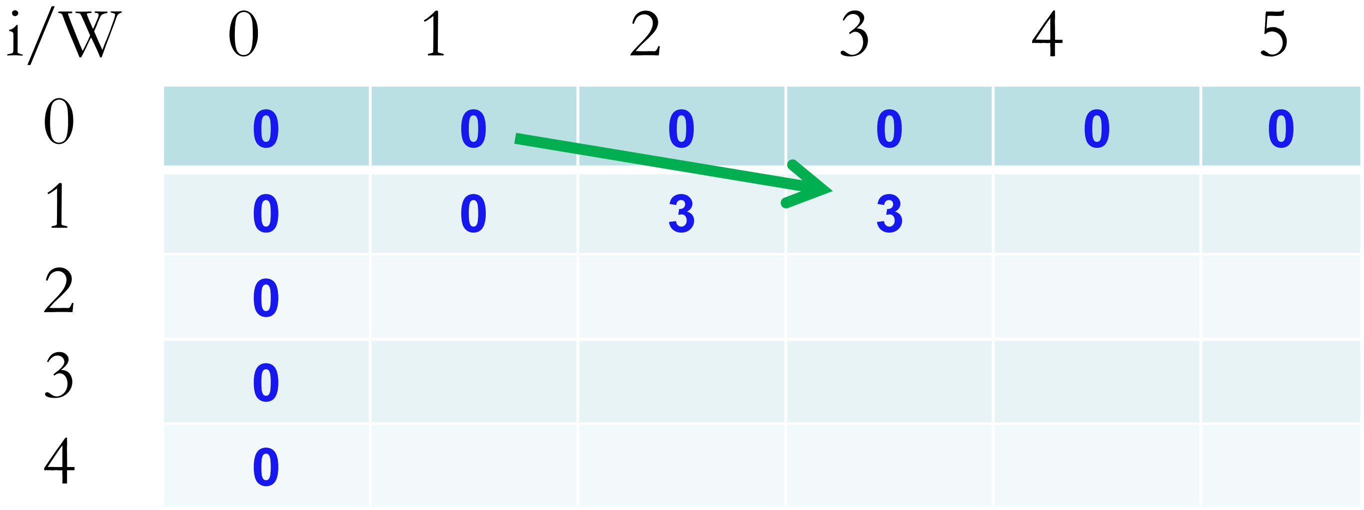
3: (4,5)

4: (5,6)

```
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 (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					

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i$
$=1$

Items:

1: (2,3)

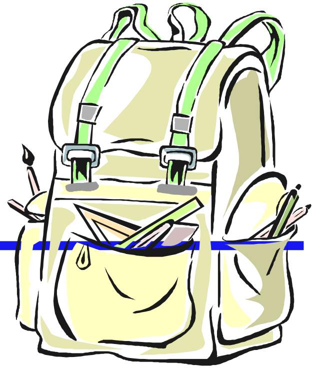
2: (3,4)

3: (4,5)

4: (5,6)

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

Example (7)



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					

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i$

$=2$

Items:

1: (2,3)

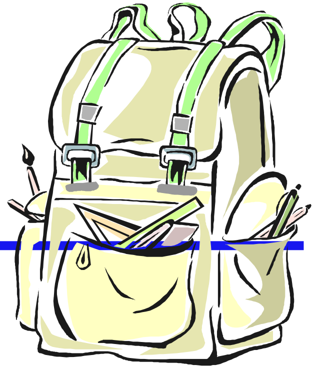
2: (3,4)

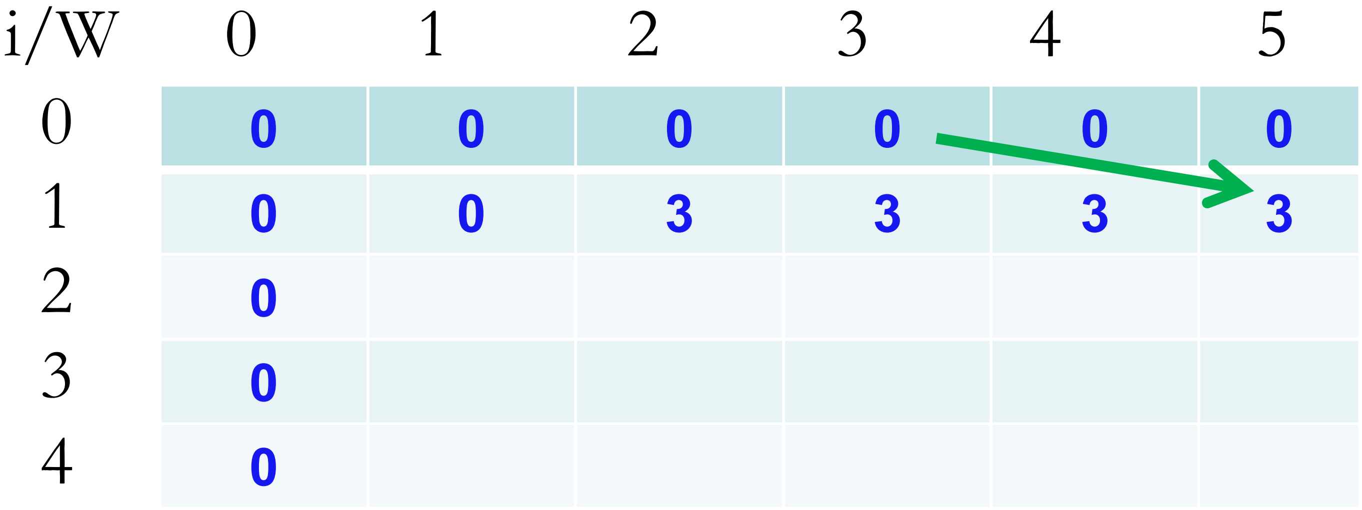
3: (4,5)

4: (5,6)

```
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 (8)





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

Items:

1: (2,3)

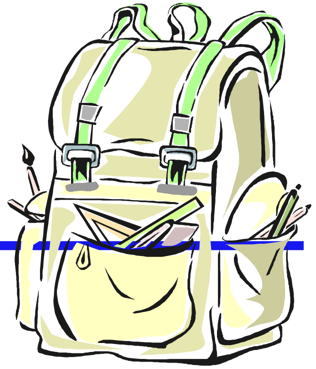
2: (3,4)

3: (4,5)

4: (5,6)

```
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)



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					

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i$

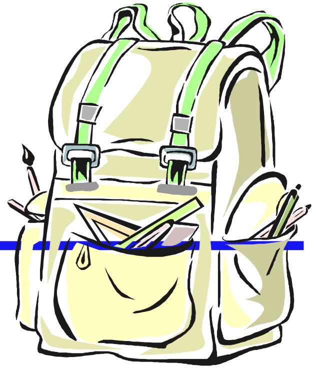
$=-2$

Items:

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

```
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)



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					

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i$
$=-1$

Items:

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

```
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)

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					

$i=2$

$b_i=4$

$w_i=3$

$w=3$

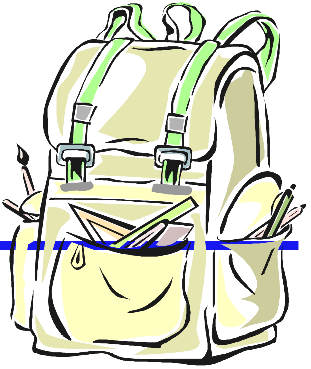
$w-w_i$
$=0$

Items:

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

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

Example (12)



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					

$i=2$

$b_i=4$

$w_i=3$

$w=4$

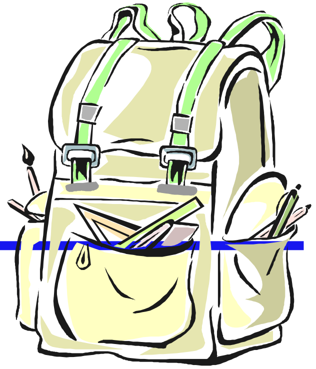
$w-w_i$
$=1$

Items:

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

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

Example (13)



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					

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i$

$=2$

Items:

1: (2,3)

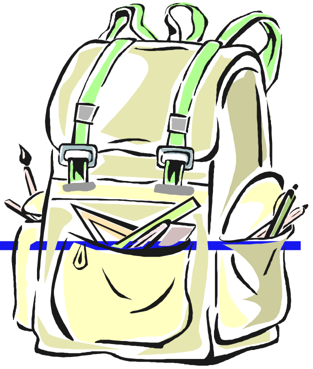
2: (3,4)

3: (4,5)

4: (5,6)

```
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)



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**				
4	**0**					

$i=3$

$b_i=5$

$w_i=4$

$w=1$

$w-w_i$

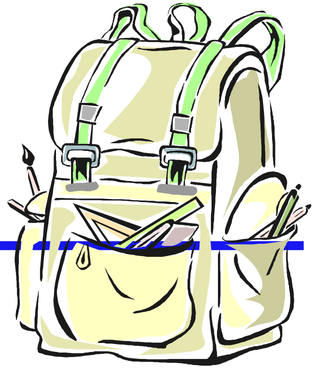
$=-3$

Items:

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

```
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)



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	0					

$i=3$

$b_i=5$

$w_i=4$

$w=2$

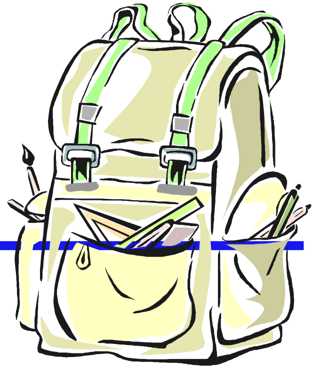
$w-w_i$
$=-2$

Items:

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

```
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)



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					

$i=3$

$b_i=5$

$w_i=4$

$w=3$

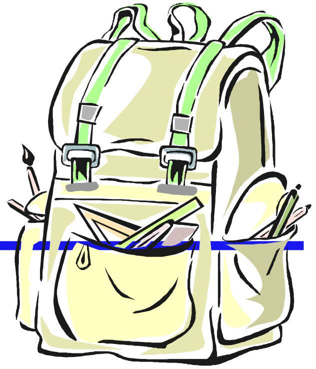
$w-w_i$
$=-1$

Items:

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

```
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)



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					

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w-w_i$

$=0$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

```
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)

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					

$i=3$

$b_i=5$

$w_i=4$

$w=5$

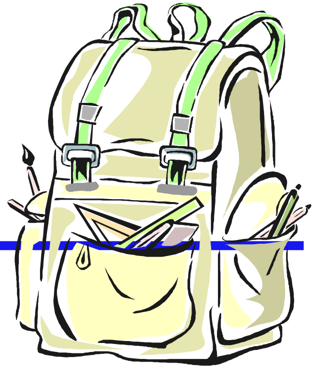
$w-w_i$
$=1$

Items:

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

```
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 (19)



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				

$i=4$

$b_i=6$

$w_i=5$

$w=1$

$w-w_i$

$=-4$

Items:

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

```
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 (20)

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			

i=4

b_i=6

w_i=5

w=2

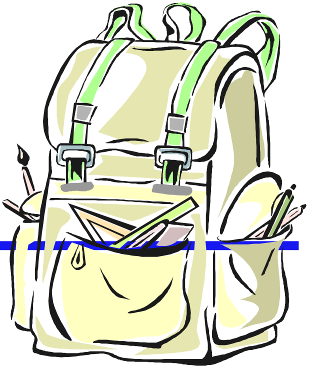
$w-w_i$ =-3

Items:

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

```
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 (21)



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		

$i=4$

$b_i=6$

$w_i=5$

$w=3$

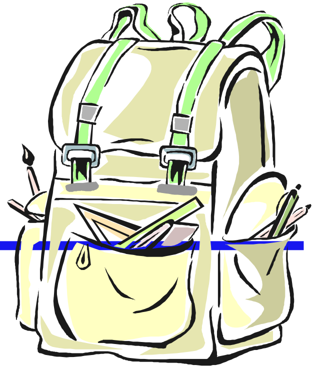
$w-w_i$
$=-2$

Items:

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

```
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 (22)



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	

$i=4$

$b_i=6$

$w_i=5$

$w=4$

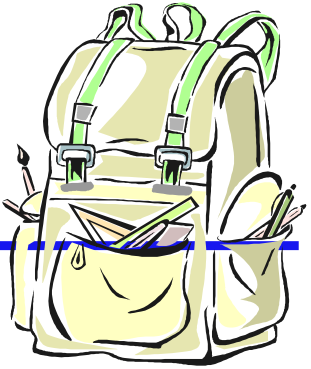
$w-w_i$ $=-1$

Items:

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

```
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 (23)



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

$i=4$

$b_i=6$

$w_i=5$

$w=5$

$w-w_i$
$=0$

Items:

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

```
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

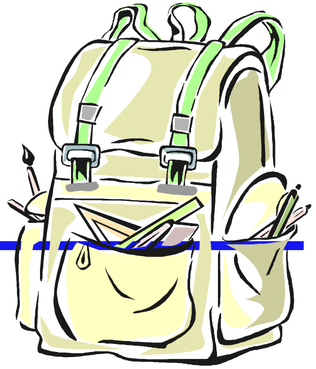
- This algorithm only finds the max possible value that can be carried in the knapsack
  - I.e., the value in B[n,W]
- To know the items that make this maximum value, an addition to this algorithm is necessary.

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 not in the knapsack
              // Could it be in the optimally packed knapsack?
```

Find the Items (1)



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

$i=4$

$k=5$

$b_i=6$

$w_i=5$

$B[i,k]=7$

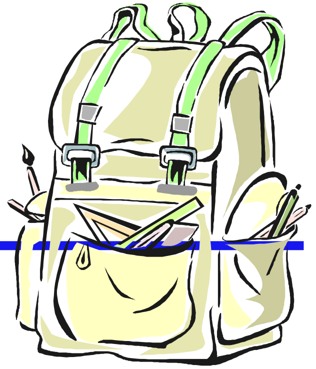
$B[i-1,k]=7$

Items:

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

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

Find the Items (2)



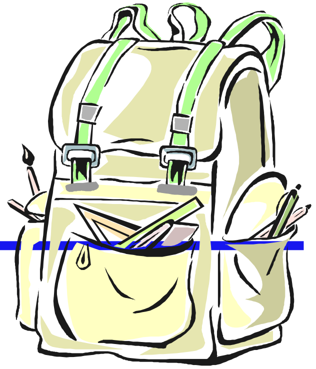
i/W	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7 ↑

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

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

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

Find the Items (3)



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

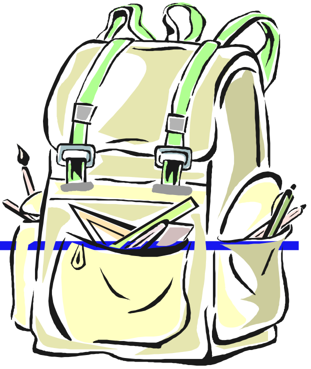
i=3
k=5
$b_i=5$
$w_i=4$
$B[i,k]=7$
$B[i-1,k]=7$

Items:

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

```
i=n, k=W
while i, k > 0
    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
        $i = i-1$
```

Find 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

$i=2$

$k=5$

$b_i=4$

$w_i=3$

$B[i,k]=7$

$B[i-1,k]=3$

$k-w_i=2$

Items:

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

```
i=n, k=W
while i, k > 0
    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
        $i = i-1$
```

Find 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

$b_i=3$

$w_i=2$

$B[i,k]=3$

$B[i-1,k]=0$

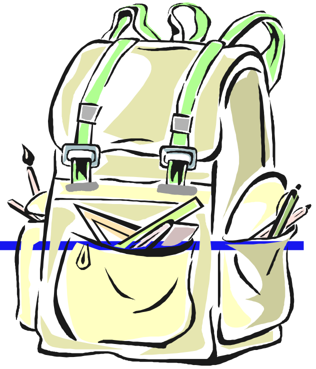
$k-w_i=0$

Items:

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

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

Find the Items (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

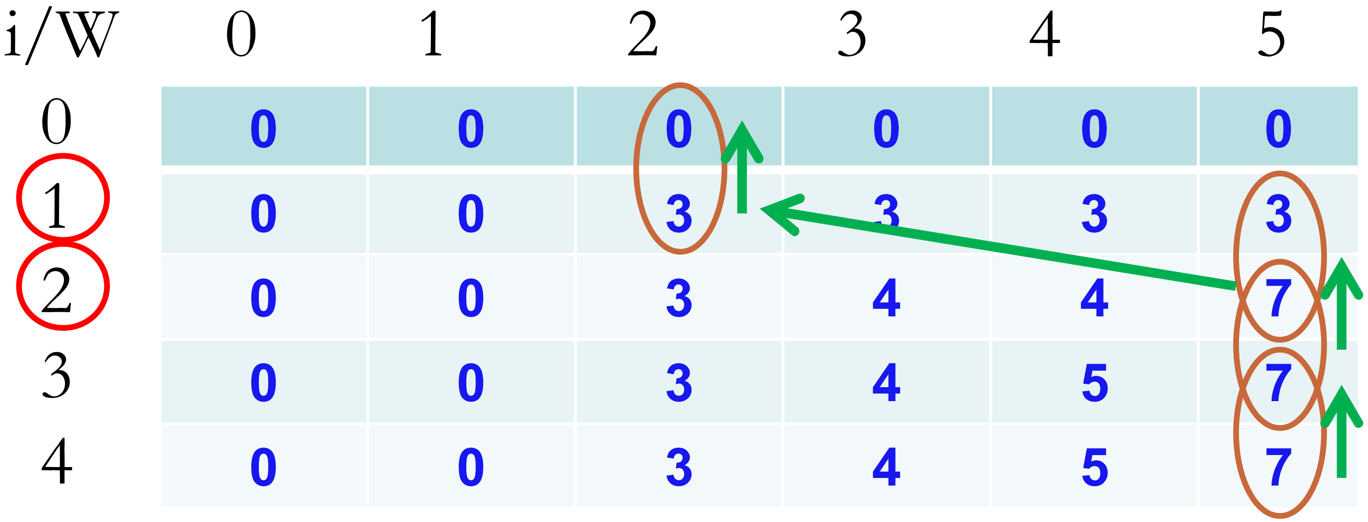
Items:

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

```
i=n, k=W
while i, k > 0
    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
        $i = i-1$
```

The optimal knapsack should contain {1, 2}

Find the Items (7)



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=n, k=W
while i, k > 0
    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
        $i = i-1$
```

The optimal knapsack should contain {1, 2}

Summary

- 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. $O(2^n)$