

Dynamic Programming

Course Code: CSC 2211

Course Title: Algorithms



Dept. of Computer Science
Faculty of Science and Technology

Lecturer No:		Week No:	08	Semester :	Summer 2019-202
Lecturer:	<i>Name & email</i>				

Lecture Outline

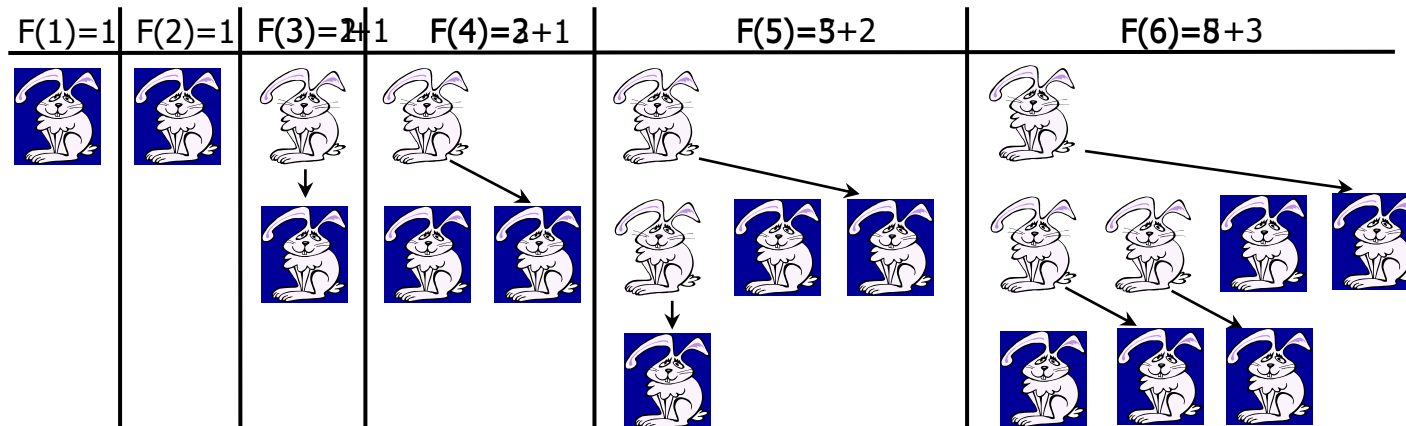


1. Introduction to Dynamic Programming
2. Elements of Dynamic Programming
3. Designing a Dynamic Programming Algorithm
4. 0/1 Knapsack Problem

Fibonacci Numbers

➤ *Leonardo Fibonacci (1202):*

- A rabbit starts producing offspring during the second year after its birth and produces one child each generation
- How many rabbits will there be after n generations?



...Fibonacci Numbers

➤ $F(n) = F(n-1) + F(n-2)$

➤ $F(0) = 0, F(1) = 1$

➤ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...

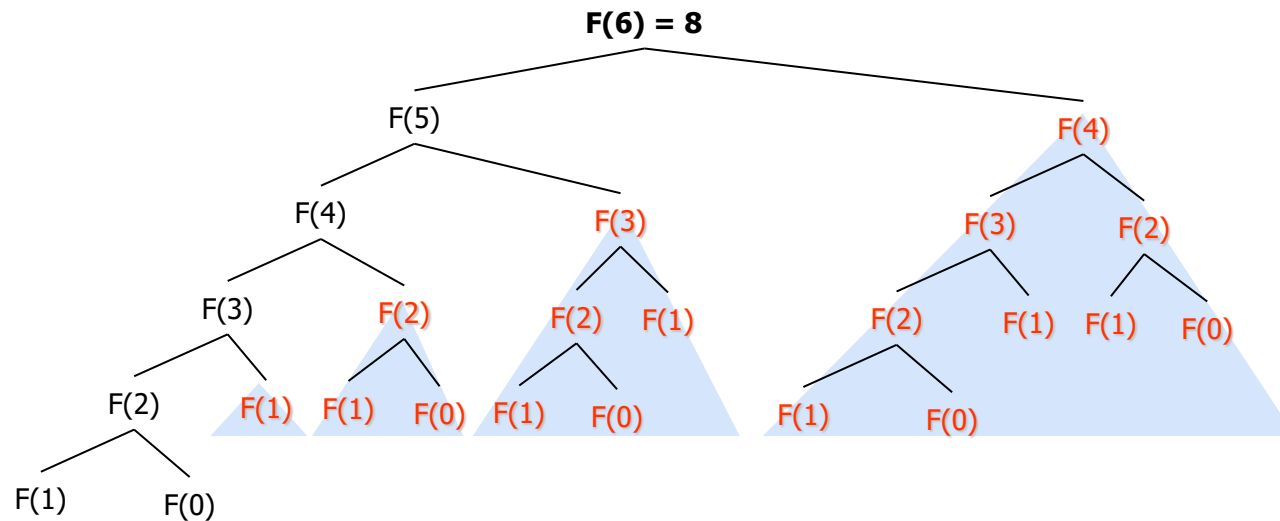
```
FibonacciR(n)
```

```
01 if n ≤ 1 then return n
```

```
02 else return FibonacciR(n-1) + FibonacciR(n-2)
```

➤ Straightforward recursive procedure is slow!

...Fibonacci Numbers



- We keep calculating the same value over and over!
- Subproblems are overlapping – they share sub-subproblems



...Fibonacci Numbers

➤ How many summations are there in $F(n)$?

➤ $F(n) = F(n - 1) + F(n - 2) + 1$

➤ $F(n) \geq 2F(n - 2) + 1$ and $F(1) = F(0) = 0$

➤ Solving the recurrence we get

$$F(n) \geq 2^{n/2} - 1 \approx 1.4^n$$

➤ Running time is *exponential!*

...Fibonacci Numbers

- We can calculate $F(n)$ in *linear* time by remembering solutions to the solved sub-problems (= *dynamic programming*).
- Compute solution in a bottom-up fashion
- Trade space for time!

```
Fibonacci (n)
```

```
01  F[0] ← 0
```

```
02  F[1] ← 1
```

```
03  for i ← 2 to n do
```

```
04      F[i] ← F[i-1] + F[i-2]
```

```
05  return F[n]
```

...Fibonacci Numbers

- In fact, only two values need to be remembered at any time!

```
FibonacciImproved(n)
01 if n ≤ 1 then return n
02 Fim2 ← 0
03 Fim1 ← 1
04 for i ← 2 to n do
05     Fi ← Fim1 + Fim2
06     Fim2 ← Fim1
07     Fim1 ← Fi
05 return Fi
```




History

- Dynamic programming
 - Invented in the 1957 by *Richard Bellman* as a general method for optimizing multistage decision processes
 - The term “programming” refers to a *tabular method*.
 - Often used for *optimization* problems.

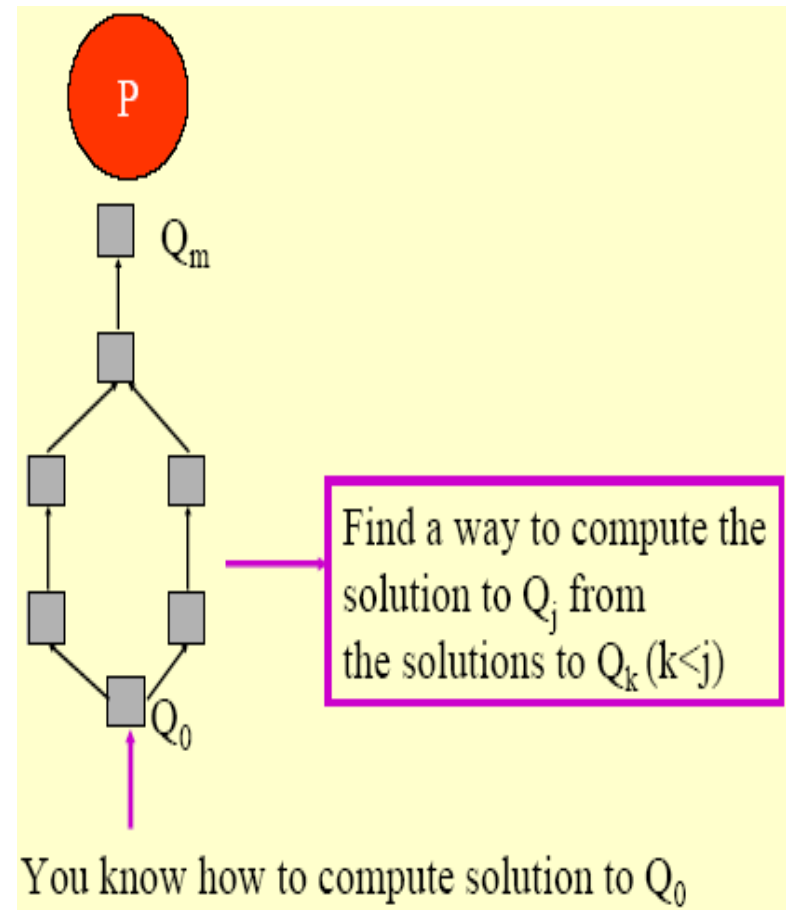


Dynamic Programming

- Solves problems by **combining** the solutions to sub problems that contain common sub-sub-problems.
- Difference between DP and Divide-and-Conquer
 - Using ***Divide and Conquer*** to solve these problems is **inefficient** as the same common sub-sub-problems have to be solved **many times**.
 - DP will solve each of them **once** and their **answers are stored in a table** for future reference.

Intuitive Explanation

- Given a problem P , obtain a sequence of problems Q_0, Q_1, \dots, Q_m , where:
 - You have a solution to Q_0
 - The solution to a problem Q_j , $j > 0$, can be obtained from solutions to problems $Q_0 \dots Q_k$, $k < j$, that appear earlier in the “sequence”.

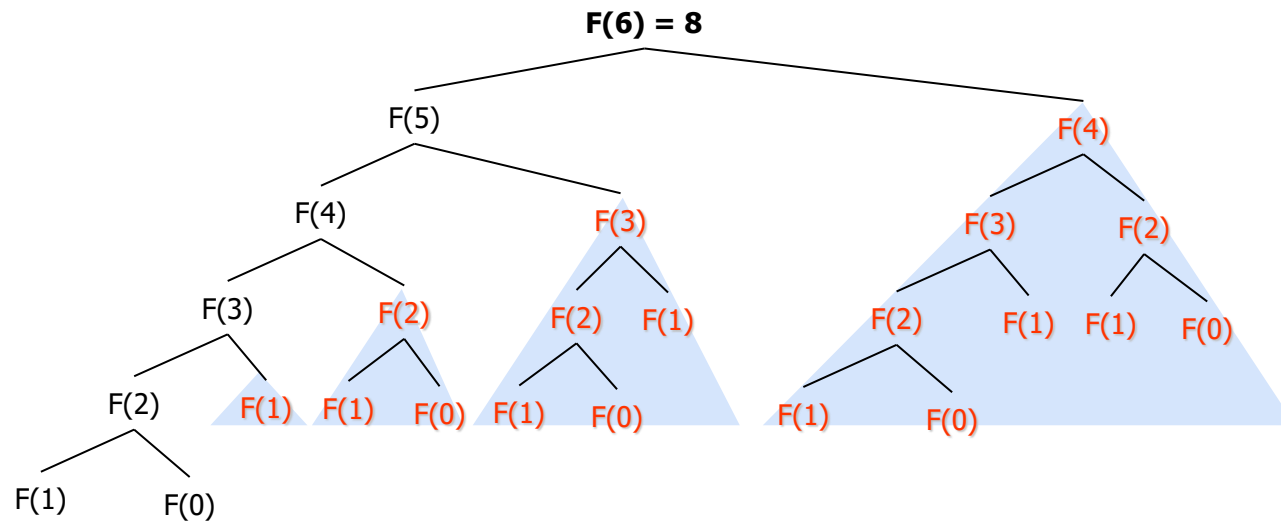




Elements of Dynamic Programming

- **DP** is used to solve problems with the following characteristics:
 - **Optimal sub-structure** (Principle of Optimality)
 - an optimal solution to the problem contains within it optimal solutions to sub-problems.
 - **Overlapping sub problems**
 - there exist some places where we solve the same sub problem more than once

Example: Fibonacci Numbers



- We keep calculating the same value over and over!
- Subproblems are overlapping – they share sub-subproblems



Steps to Designing a Dynamic Programming Algorithm

1. Characterize *optimal sub-structure*
2. *Recursively* define the value of an optimal solution
3. Compute the value *bottom up*
4. (if needed) *Construct* an optimal solution



1. Optimal Sub-Structure

- An optimal solution to the problem contains optimal solutions to sub-problems
- Solution to a problem:
 - Making a **choice** out of a number of **possibilities** (look what possible choices there can be)
 - **Solving** one or more **sub-problems** that are the **result of a choice** (characterize the space of sub-problems)
- Show that solutions to sub-problems must themselves be optimal for the whole solution to be optimal.



2. Recursive Solution

- Write a recursive solution for the value of an optimal solution.

$$M_{opt} = \underset{\text{over all } k \text{ choices}}{\text{Optimal}} \left[\begin{array}{l} \text{Combination of } M_{opt} \text{ for all subproblems resulting from choice } k \\ + \text{ The cost associated with making the choice } k \end{array} \right]$$

- Show that the number of different instances of sub-problems is bounded by a polynomial.



3. Bottom Up

- **Compute** the value of an optimal solution in a bottom-up fashion, so that you always have the necessary **sub-results pre-computed** (or use memoization)
- Check if it is possible to **reduce** the **space** requirements, by “**forgetting**” solutions to sub-problems that will not be used any more



4. Construct Optimal Solution

- Construct an optimal solution from **computed information** (which records a sequence of choices made that lead to an optimal solution)



Knapsack Problem



The Knapsack Problem

➤ The famous *knapsack problem*:

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



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?



Item	A	B	C	D
Profit	24	18	18	10
Weight	24	10	10	7

Knapsack Size = 25

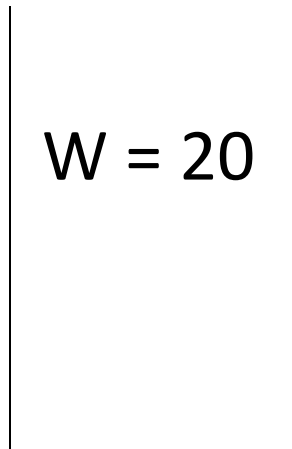
Item	A	B	C
Price	100	280	120
Weight	10	40	20
Ratio	10	7	6





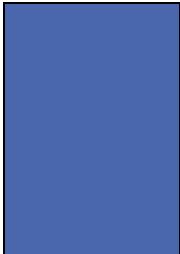
Knapsack size = 60



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



The Knapsack Problem

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



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.
- Just another version of this problem is the “*Fractional Knapsack Problem*”, where 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 the most total value and with total weight less or equal to W
- Running time will be $O(2^n)$



0-1 Knapsack problem: brute-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 valid 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. Explanation follows....



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;

total benefit: 20

$w_1=2$ $b_1=3$	$w_2=4$ $b_2=5$	$w_3=5$ $b_3=8$	$w_4=9$ $b_4=10$
--------------------	--------------------	--------------------	---------------------

For S_5 :

Total weight: 20

total 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 !!!



Defining a Subproblem (continued)

- 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

- 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 one of the two:

- 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: $w_k \leq w$. Then the item k can be in the solution, and we choose the case with greater value



The Knapsack Problem And Optimal Substructure

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



Solving The Knapsack Problem

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

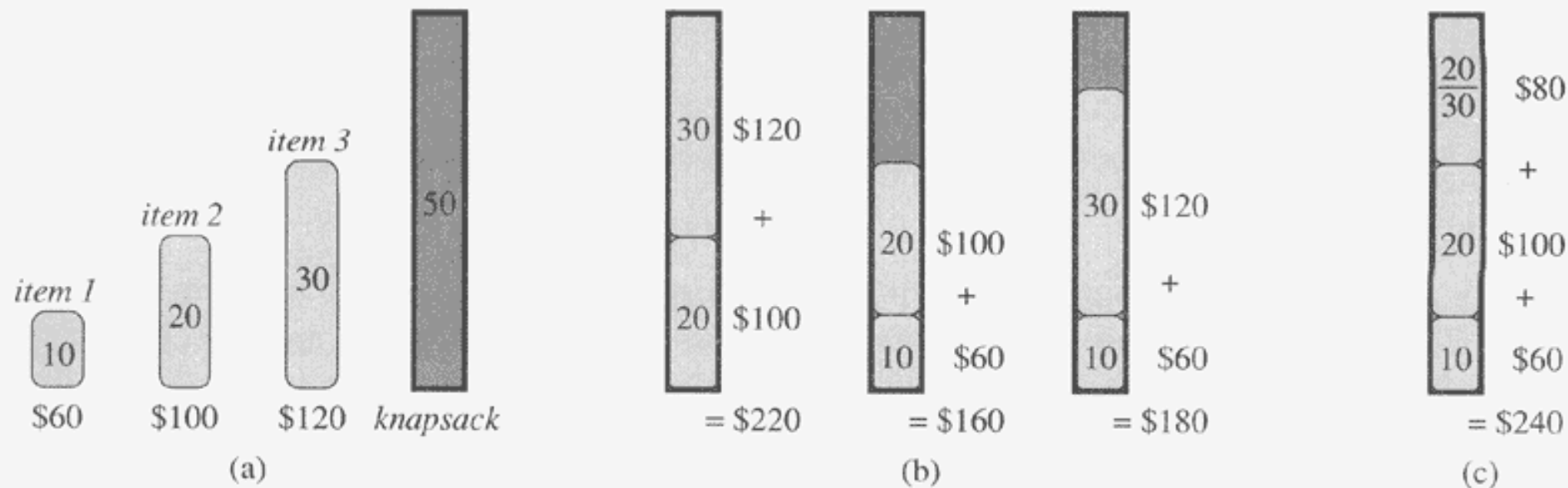


Figure 16.2 The greedy strategy does not work for the 0-1 knapsack problem. (a) The thief must select a subset of the three items shown whose weight must not exceed 50 pounds. (b) The optimal subset includes items 2 and 3. Any solution with item 1 is suboptimal, even though item 1 has the greatest value per pound. (c) For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.

The Knapsack Problem: Greedy Vs. Dynamic



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



Fractional-knapsack

➤ **Greedy-fractional-knapsack (w, v, W)**

➤ FOR $i = 1$ to n
 do $x[i] = 0$
weight = 0
while weight < W
 do $i =$ best remaining item
 IF weight + $w[i] \leq W$
 then $x[i] = 1$
 weight = weight + $w[i]$
 else
 $x[i] = (w - \text{weight}) / w[i]$
 weight = W
return x



0-1 Knapsack Algorithm

```
for w = 0 to W
    B[0,w] = 0
for i = 0 to n
    B[i,0] = 0
    for w = 0 to W
        if  $w_i \leq w$  // 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 $i = 0$ to n

Repeat n times

$B[i,0] = 0$

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	0	1	2	3	4
W						
0		0				
1		0				
2		0				
3		0				
4		0				
5		0				

for $w = 0$ to W

$$B[0,w] = 0$$

Example (3)

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

for $i = 0$ to n
 $B[i,0] = 0$

Example (4)

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

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=1$

$w-w_i=-1$

if $w_i \leq w$ // 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	0	1	2	3	4
W						
0		0	0	0	0	0
1		0	0			
2		0	3			
3		0				
4		0				
5		0				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=2$

$w-w_i=0$

if $w_i \leq w$ // 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	0	1	2	3	4
W						
0		0	0	0	0	0
1		0	0			
2		0	3			
3		0	3			
4		0				
5		0				

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=3$

$w-w_i=1$

if $w_i \leq w$ // 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 (7)

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

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=4$

$w-w_i=2$

if $w_i \leq w$ // 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	0	1	2	3	4
w						
0		0	0	0	0	0
1		0	0			
2		0	3			
3		0	3			
4		0	3			
5		0	3			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=1$

$b_i=3$

$w_i=2$

$w=5$

$w-w_i=2$

if $w_i \leq w$ // 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	0	1	2	3	4
w						
0		0	0	0	0	0
1		0	0	→ 0		
2		0	3			
3		0	3			
4		0	3			
5		0	3			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=1$

$w-w_i=-2$

if $w_i \leq w$ // 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	0	1	2	3	4
w						
0		0	0	0	0	0
1		0	0	0		
2		0	3	→ 3		
3		0	3			
4		0	3			
5		0	3			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=2$

$w-w_i=-1$

if $w_i \leq w$ // 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	0	1	2	3	4
w						
0		0	0	0	0	0
1		0	0	0		
2		0	3	3		
3		0	3	4		
4		0	3			
5		0	3			

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=3$

$w-w_i=0$

if $w_i \leq w$ // 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)

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

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=4$

$w-w_i=1$

if $w_i \leq w$ // 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)

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

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=2$

$b_i=4$

$w_i=3$

$w=5$

$w-w_i=2$

if $w_i \leq w$ // 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	0	1	2	3	4
w						
0		0	0	0	0	0
1		0	0	0 → 0		
2		0	3	3 → 3		
3		0	3	4 → 4		
4		0	3	4		
5		0	3	7		

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=1..3$

if $w_i \leq w$ // 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	0	1	2	3	4
w						
0		0	0	0	0	0
1		0	0	0	0	
2		0	3	3	3	
3		0	3	4	4	
4		0	3	4	5	
5		0	3	7		

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=4$

$w - w_i = 0$

if $w_i \leq w$ // 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	0	1	2	3	4
W						
0		0	0	0	0	0
1		0	0	0	0	
2		0	3	3	3	
3		0	3	4	4	
4		0	3	4	5	
5		0	3	7	→ 7	

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=5$

$w - w_i = 1$

if $w_i \leq w$ // 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	0	1	2	3	4
w						
0		0	0	0	0	0
1		0	0	0	0 →	0
2		0	3	3	3 →	3
3		0	3	4	4 →	4
4		0	3	4	5 →	5
5		0	3	7	7	

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

$i=3$

$b_i=5$

$w_i=4$

$w=1..4$

if $w_i \leq w$ // 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	0	1	2	3	4
w						
0		0	0	0	0	0
1		0	0	0	0	0
2		0	3	3	3	3
3		0	3	4	4	4
4		0	3	4	5	5
5		0	3	7	7	7

$i=3$

$b_i=5$

$w_i=4$

$w=5$

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

if $w_i \leq w$ // 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
- To know the items that make this maximum value, an addition to this algorithm is necessary
- Please see LCS algorithm lecture for the example how to extract this data from the table we built



Books

1. ***Introduction to Algorithms, Third Edition, Thomas H. Cormen, Charle E. Leiserson, Ronald L. Rivest, Clifford Stein (CLRS).***
2. ***Fundamental of Computer Algorithms, Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran (HSR)***



References

- https://algorithmist.com/wiki/Dynamic_programming
- <https://www.topcoder.com/community/competitive-programming/tutorials/dynamic-programming-from-novice-to-advanced/>
- CLRS: 15.3
- HSR: 5.1, 5.5