# Data Structures and Algorithms
# Dynamic Programming

Dr. Chung-Wen Albert Tsao

# Overview

- Introductory Example
- Shortest Path Problem
- Dynamic Programming Concept
- Longest Common Subsequence
- General Alignment
- Knapsack
- Scheduling

# A motivating example: Fibonacci numbers

- 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, …

$$\begin{cases} F(0) = 1; \\ F(1) = 1; \\ F(n) = F(n-1) + F(n-2) \end{cases}$$

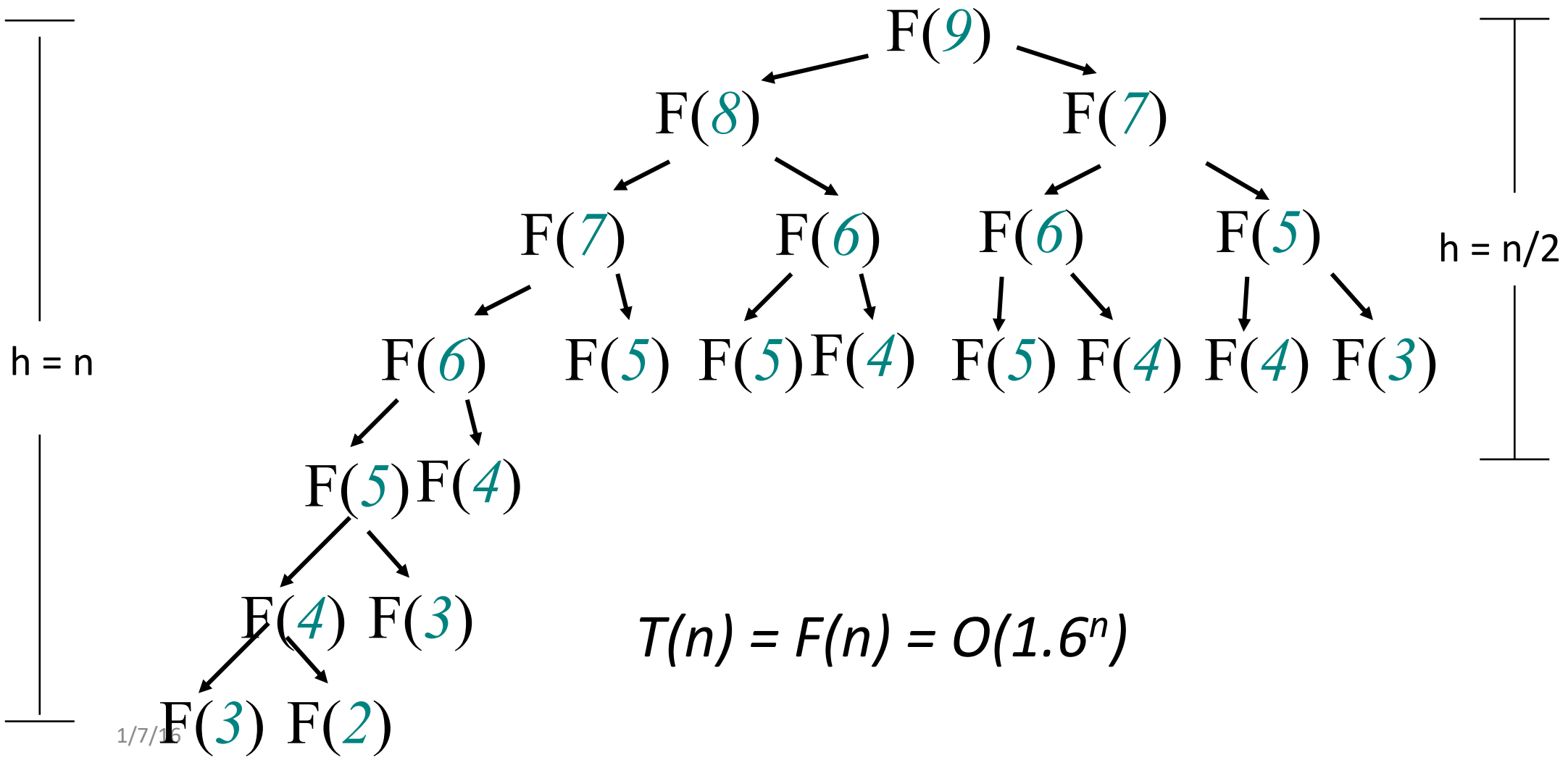
- How to compute F(n)?

# A Top-Down Recursive Algorithm

```
function fib(n)
        if (n == 0 or n == 1)  return 1;
        else  return fib(n-1) + fib(n-2);
```

What's the time complexity?

- Time complexity between $2^{n/2} = (1.414)^n$ and $2^n$




$T(n) = F(n) = O(1.6^n)$

# An Bottom-Up **iterative** algorithm

```
function fib(n)
        F[0] = 1;       F[1] = 1;
         for i = 2 to n
                 F[i] = F[i-1] + F[i-2];
        Return F[n];
```

Time complexity?

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Fib | **1** | **1** | **2** | **3** | **5** | **8** | | | | | | | | |

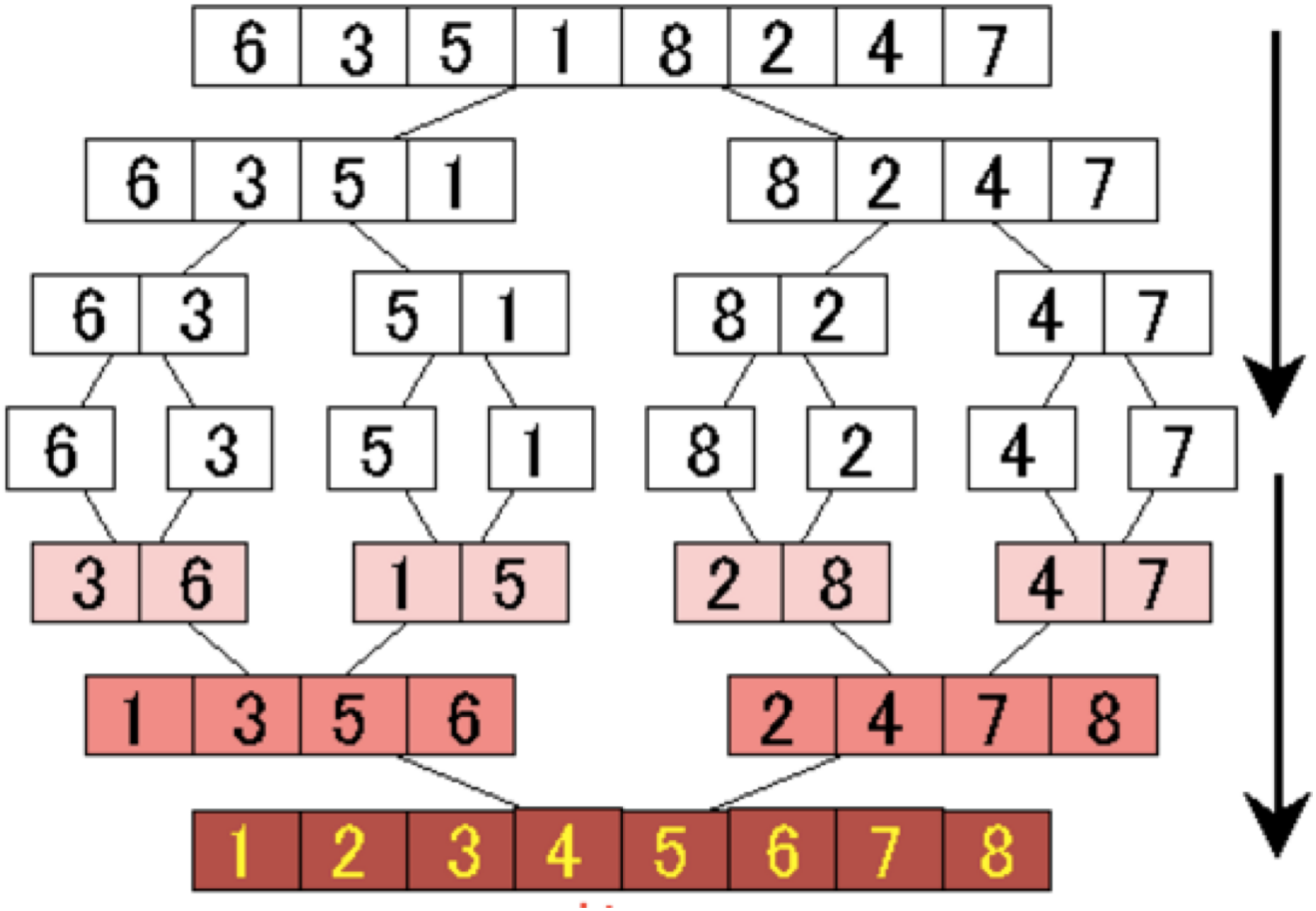
# Top-Down Recursive Algorithm

- Problem with the recursive Fib algorithm:
  - **Same subproblem was solved for many times!**
- Solution: avoid solving the same subproblem more than once

  (1) Bottom-up iterative : pre-compute all subproblems that may be needed later

  (2) Top-Down bottom-up: Compute on demand, but memorize the solution to avoid recomputing

# Merge Sort




- Divide-and-conquer
- Recursively sort ½ the set
- Can you always speedup a recursive algorithm by making it a bottom-up iterative algorithm?
  - E.g., merge sort
  - No. since there is no overlap between the two sub-problems

# Maximum-Subarray problem (Ex 4.1.5, p75)

Find the contiguous subarray with the largest sum.

- For example, given the array A= [ −2, 1, −3,4,−1,2,1,−5,4],

➔The contiguous subarray [4,−1,2,1] has the largest sum = 6

- Brute-force solution (Exhaustive Enumberations)?

# Example: House Robbery

- You are a professional robber planning to rob houses along a street.
- Each house has a certain amount of money stashed
- **You will trigger alarm if you break into any two adjacent houses.**
- Example:
  - Money in 7 houses: M=[$5, $5, $6, $4, $0, $1, $5]
  - Houses robbed : [ 1, 0, 1, 0, 0, 0, 1]
  - Money robbed : $16

- Solution:
  - Brute Force: generate all possible ($2^n$) scenarios, and pick the best.
  - Can we do better?

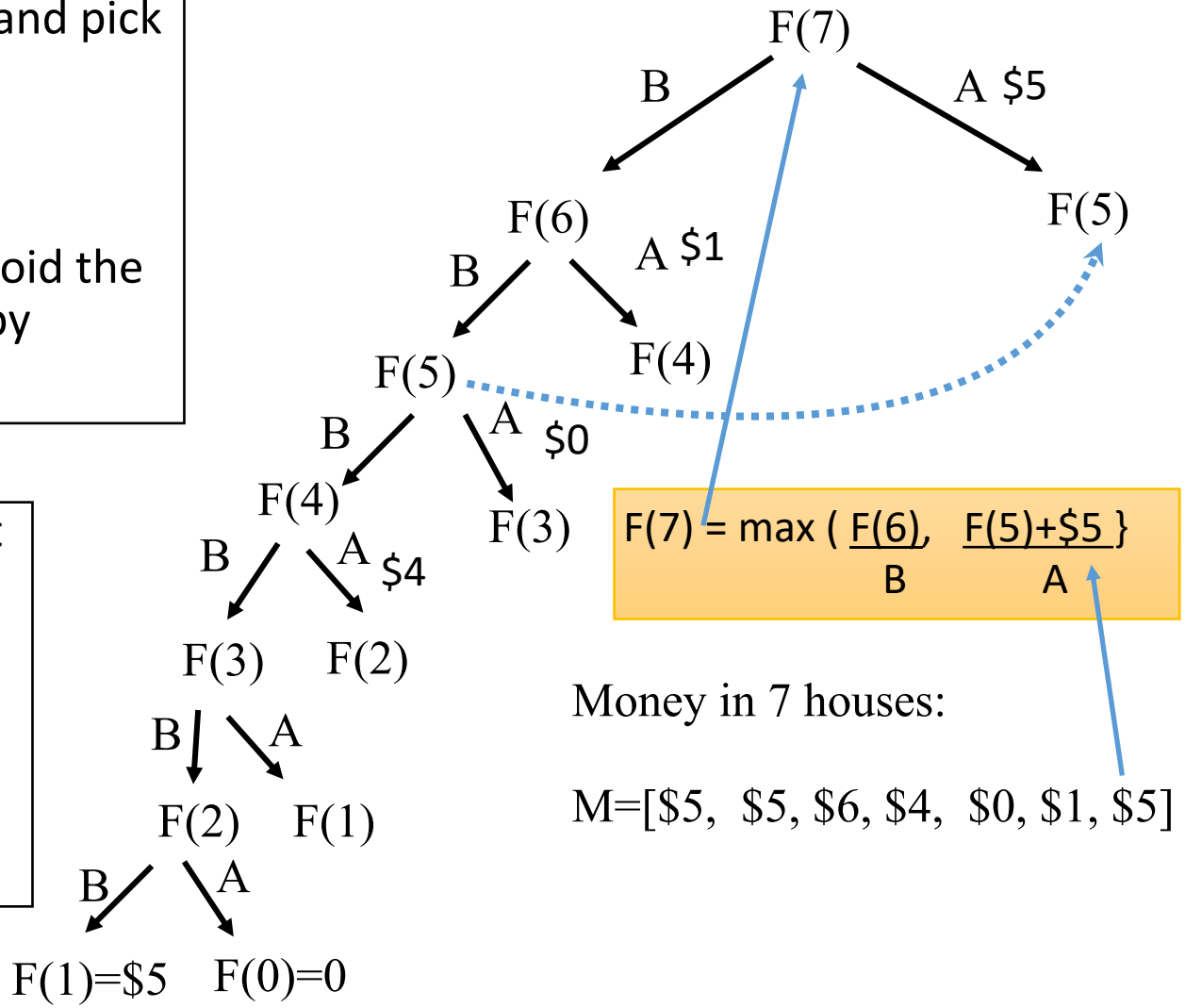
# Dynamic Programming for House Robber

- Brute Force: generate all possible (2^n) scenarios, and pick the best.
  - Can we do better? (with less enumerations)
  - Yes. **Dynamic Programming**
- Break problem to overlapping subproblems, but avoid the repeated computation for the same subproblems by "memorization"

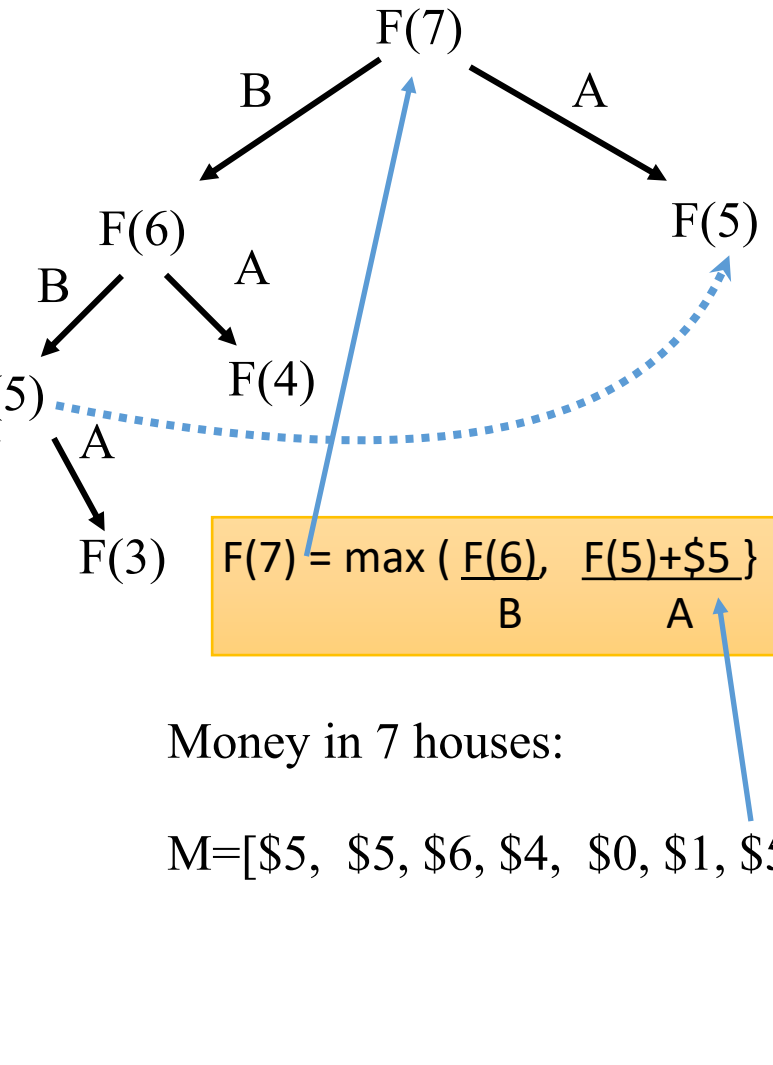
Define F[k] = The most money robbed from the first k houses.

F[k] = max { M[k]+F[k-2], Case A: rob house k
F[k-1], Case B: skip house k
}, k>=2

T(N)=?



| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| M | $5 | $5 | $6 | $4 | $0 | $1 | $5 |
| F | $5 | $5 | $11 | | | | |
| case | A | A | A | | | | |
| robbed | | | | | | | |

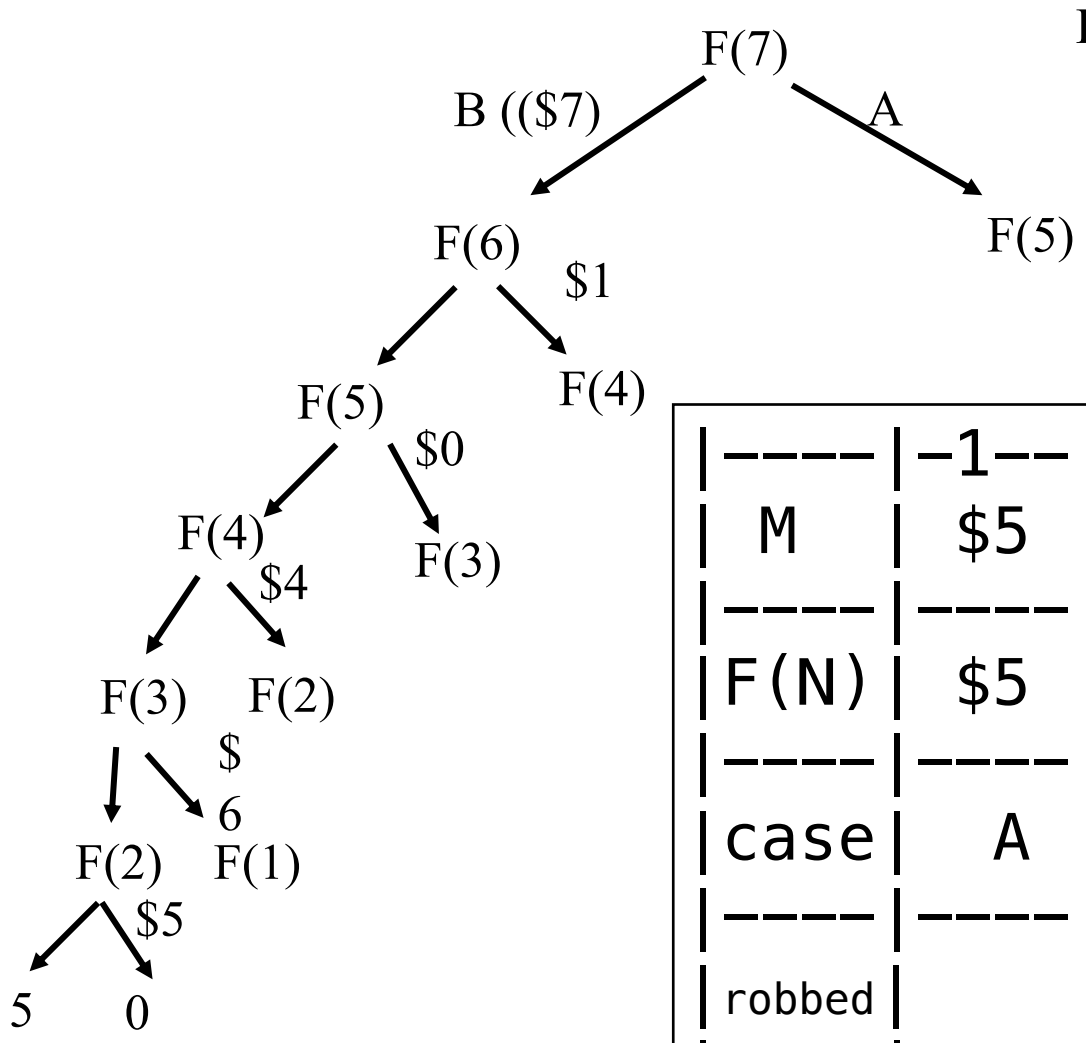



F(7) = max ( F(6), F(5)+$5 }
B A

Money in 7 houses:

M=[$5, $5, $6, $4, $0, $1, $5]

# LeetCode 198. House Robber



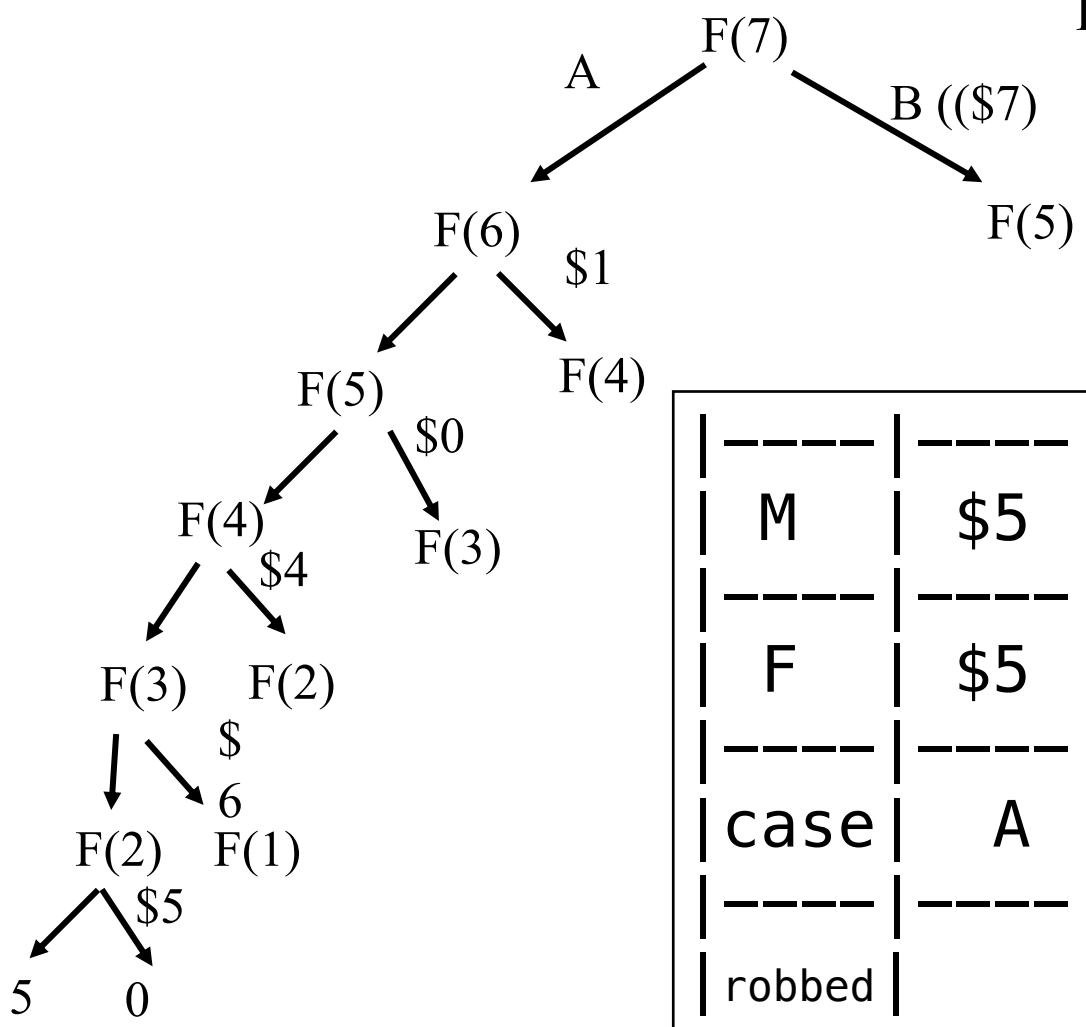

F(7) = max (F(6), F(5)+$5 }
A B

F(4) = max (F(3)=$11, F(2)+$4=$9 }
B A

| | -1- | -2- | --3- | --4- | -5- | -6- | -7- |
|---|---|---|---|---|---|---|---|
| M | $5 | $5 | $6 | $4 | $0 | $1 | $5 |
| F(N) | $5 | $5 | $11 | $11 | | | |
| case | A | A/B | A | B | | | |
| robbed | | | | | | | |

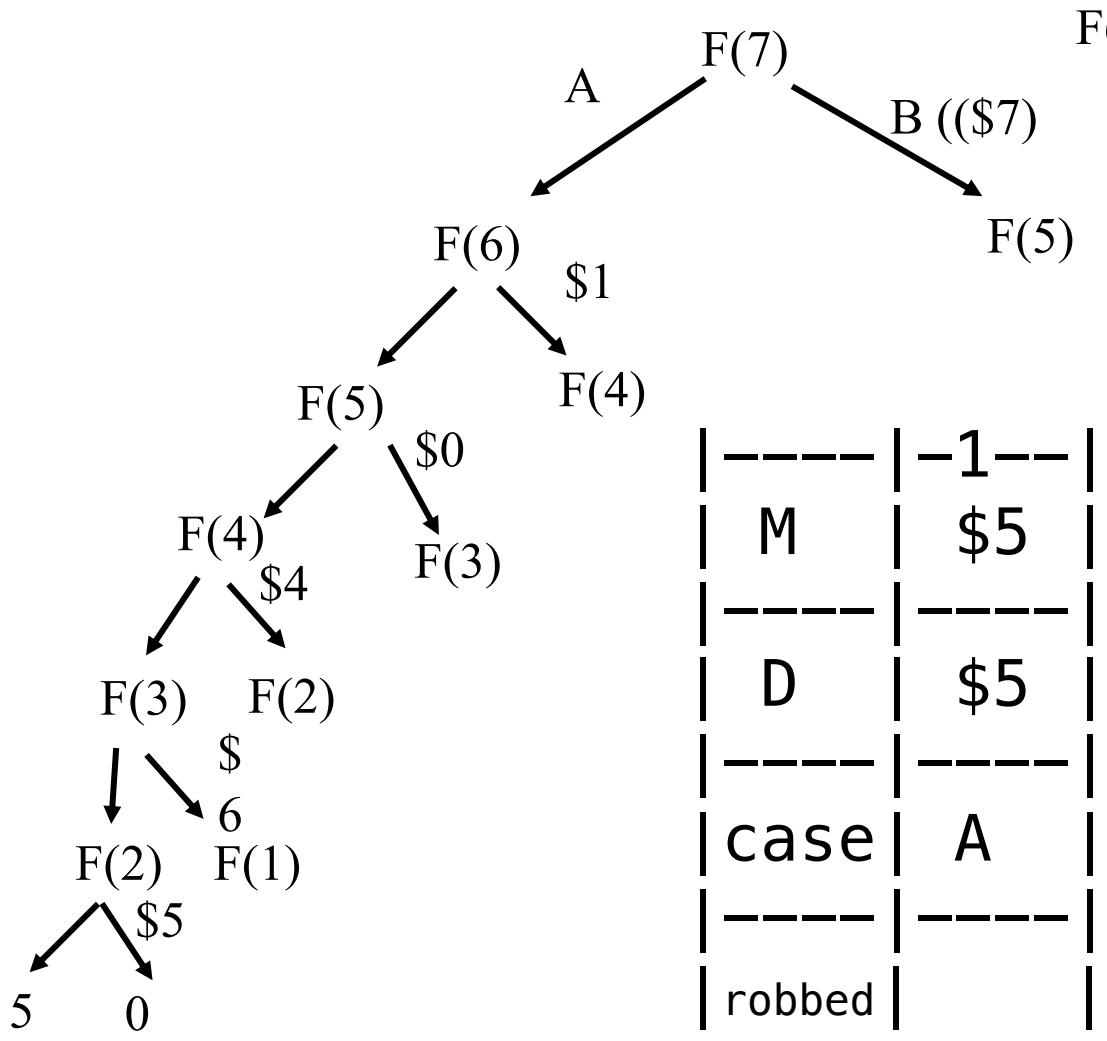
# LeetCode 198. House Robber

F(7) = max (F(6), F(5)+$5 }
A A B




| | | | | -4- | -5- | -6- | |
|---|---|---|---|---|---|---|---|
| M | $5 | $5 | $6 | $4 | $0 | $1 | $5 |
| F | $5 | $5 | $11 | $11 | $11 | | |
| case | A | A/B | A | B | A/B | | |
| robbed | | | | | | | |

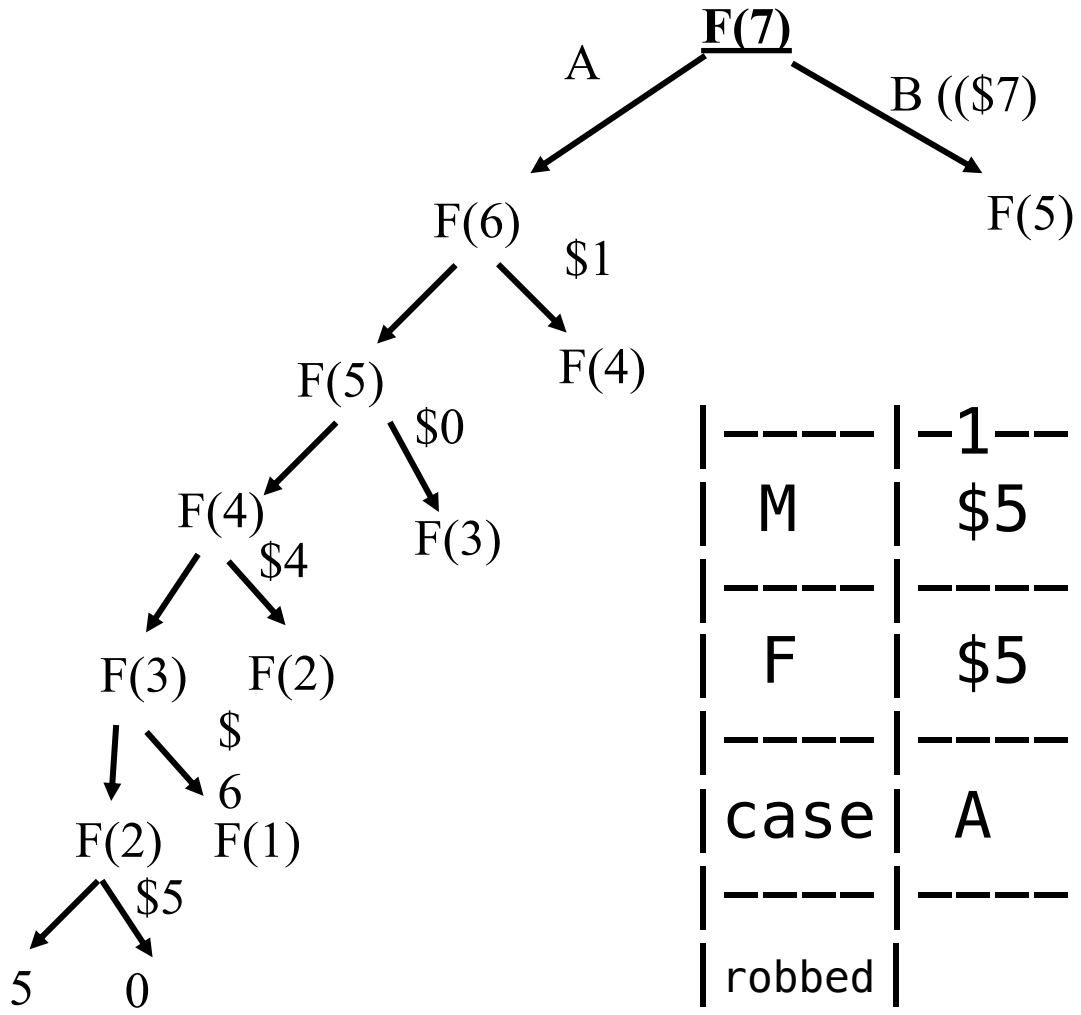
# LeetCode 198. House Robber




$$F(7) = \max(\underbrace{F(6)}_{A}, \underbrace{F(5)+\$5}_{B})$$

| | -1- | -2- | --3- | --4- | -5-- | -6-- | -7-- |
|---|---|---|---|---|---|---|---|
| M | $5 | $5 | $6 | $4 | $0 | $1 | $5 |
| D | $5 | $5 | $11 | $11 | $11 | $12 | |
| case | A | A | A | B | B | A | |
| robbed | | | | | | | |

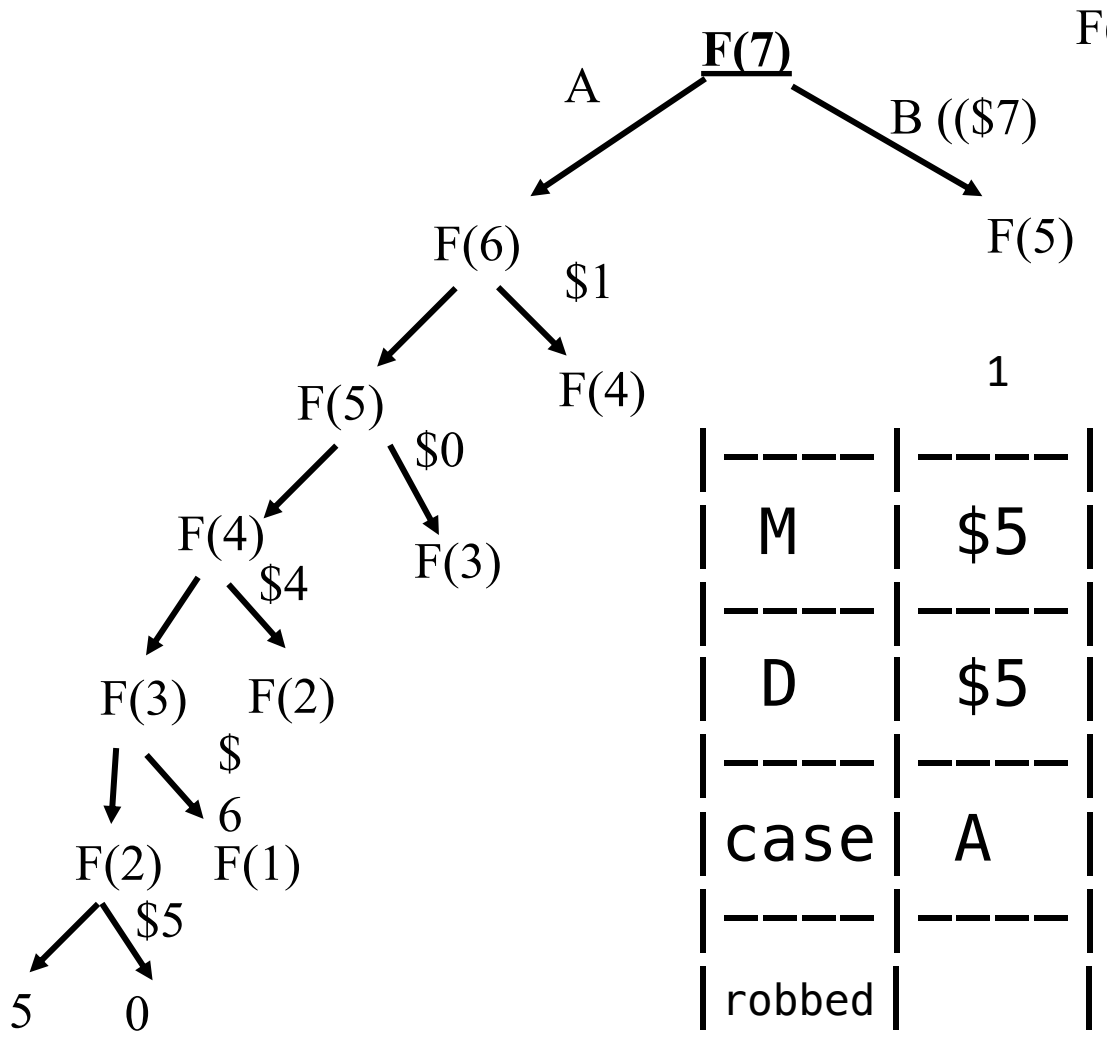
# LeetCode 198. House Robber




$$F(7) = \max(\underbrace{F(6)}_{A}, \underbrace{F(5)+\$5}_{B})$$

| | -1- | -2- | --3- | --4- | -5-- | -6-- | -7-- |
|---|---|---|---|---|---|---|---|
| M | $5 | $5 | $6 | $4 | $0 | $1 | $5 |
| F | $5 | $5 | $11 | $11 | $11 | $12 | $16 |
| case | A | A | A | B | B | A | A |
| robbed | | | | | | X | 1 |

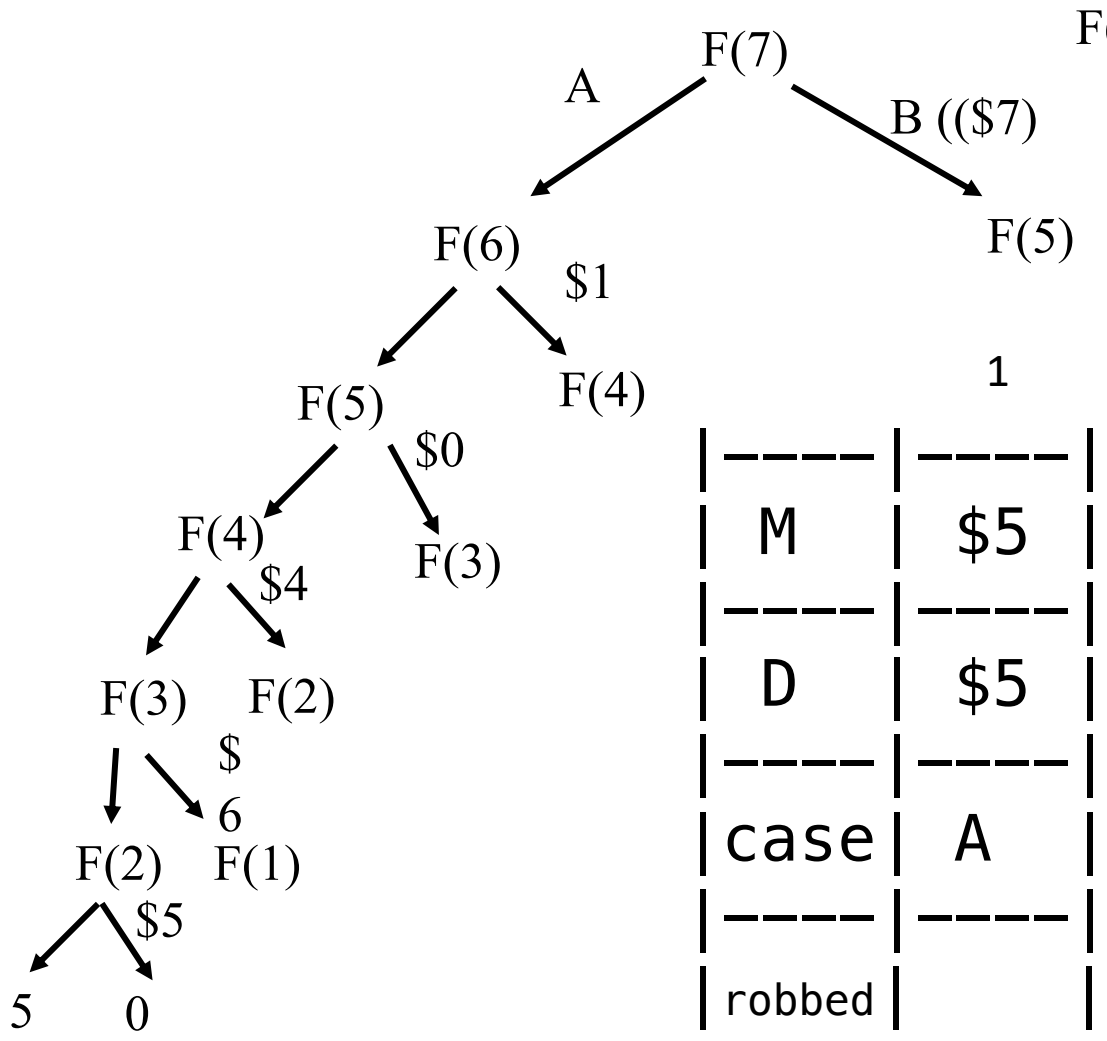
# LeetCode 198. House Robber




F(7) = max (F(6), F(5)+$5 }
A B

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| M | $5 | $5 | $6 | $4 | $0 | $1 | $5 |
| D | $5 | $5 | $11 | $11 | $11 | $12 | $**16** |
| case | A | A | A | B | B | A | A |
| robbed | | | | 0 | 0 | 0 | 1 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

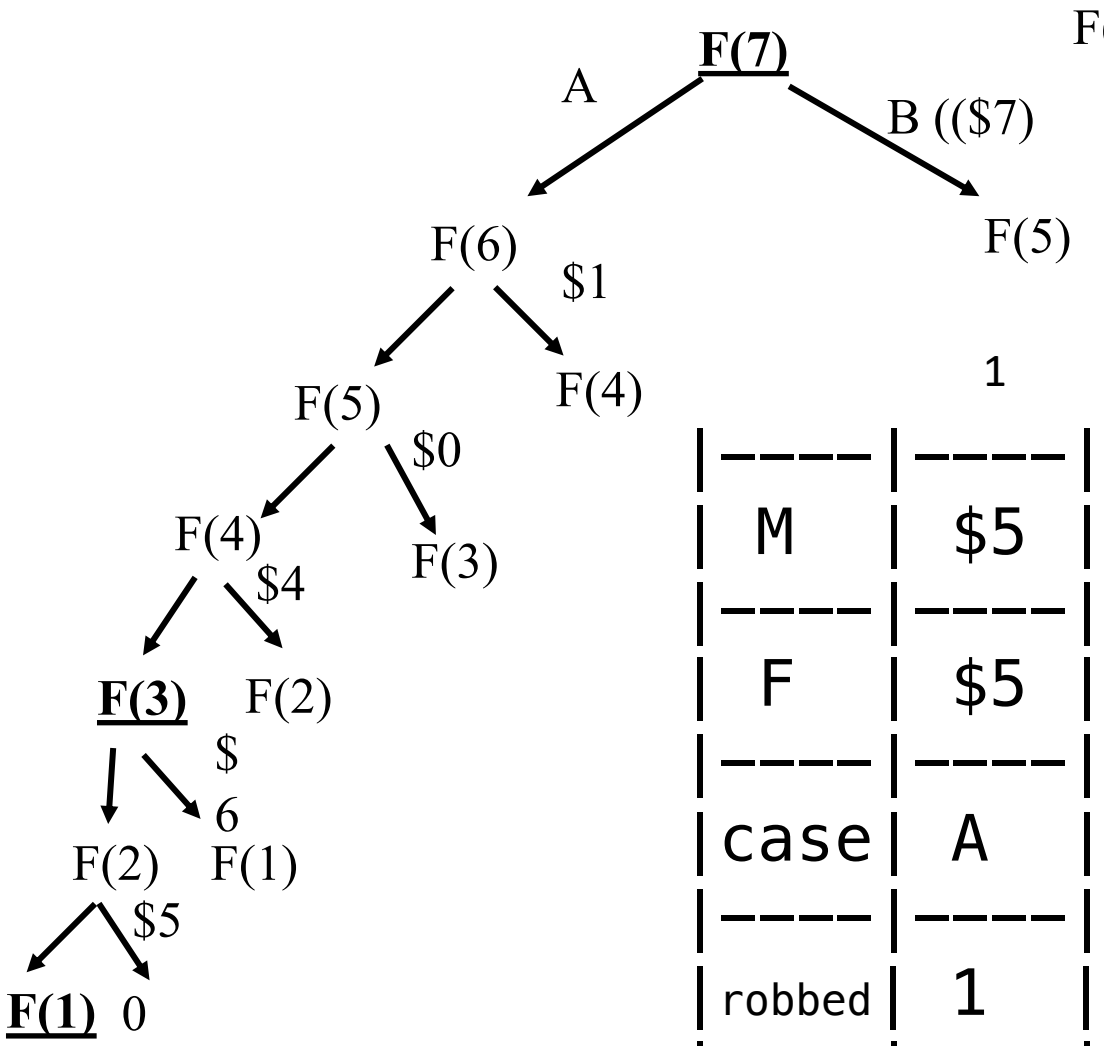
# LeetCode 198. House Robber




$$F(7) = \max(\underbrace{F(6)}_{A}, \underbrace{F(5)+\$5}_{B})$$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| M | $5 | $5 | $6 | $4 | $0 | $1 | $5 |
| D | $5 | $5 | $11 | $11 | $11 | $12 | $16 |
| case | A | A | A | B | B | A | A |
| robbed | | X | 1 | 0 | 0 | 0 | 1 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# LeetCode 198. House Robber




$$F(7) = \max(\underbrace{F(6)}_{A}, \underbrace{F(5)+\$5}_{B})$$

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| M | $5 | $5 | $6 | $4 | $0 | $1 | $5 |
| F | $5 | $5 | $11 | $11 | $11 | $12 | $16 |
| case | A | A | A | B | B | A | A |
| robbed | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

# LeetCode 198. House Robber

- How about
  - You will trigger alarm if you break into any two houses that have less than ***n*** houses between them, where ***n***=1,2,3, …

- Define F[k] = The most money you can rob from the first k houses.

  F[k] = max {

  F[k-n-1]+M[k], Case A: rob house k

  F[k-1], Case B: not rob house k (Same as the previous solution)

  }, k>=n+1

  - F[ k ]= max {M[1], …, M[n]}, k <=n

# Elements of dynamic programming

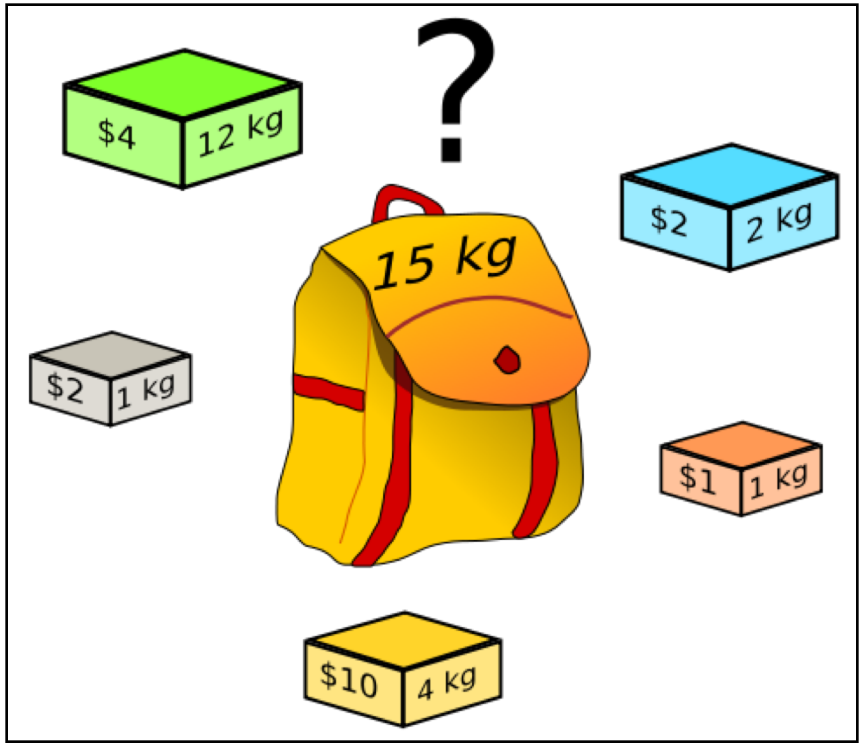
- Optimal sub-structures
  - Optimal solutions to the original problem contains optimal solutions to sub-problems
- Overlapping sub-problems
  - Some sub-problems appear repeatedly
  - Memorization and reuse
  - Carefully choose the order that sub-problems are solved, such that each sub-problem will be solved for at most once and the solution can be reused

# Two steps to dynamic programming

- Formulate the solution as a recurrence relation of solutions to subproblems.
- Specify an order to solve the subproblems so you always have what you need.
  - Bottom-up
    - Tabulate the solutions to all subproblems before they are used
  - Top-down
    - Compute when needed
    - Remember the ones you've computed

# Knapsack problem

- Each item has a **value** and a **weight**
- Objective: maximize value
- Constraint: knapsack has a **weight limitation**




Three versions:

0-1 knapsack problem: take each item or leave it

Unbounded knapsack problem: unlimited supplies of each item

Fractional knapsack problem: items are divisible

Which one is easiest to solve?

Which one is most difficult to solve?

# Fractional Knapsack

# Greedy

- Consider following items (weight, value):

**#1(10oz, $5)**
**#2(15oz, $6)**
**#3(20,oz $5)**
**#4(18oz, $6)**

- Weight limit **35oz**

What will be the most money?
What items to be picked up?

Sort items by their ratio $/oz

item #**1**: $5/10oz=**$0.5/oz**
item #**2**: $6/15oz=$**0.4**/oz
item #**4**: $6/**18**oz=$**.333**/oz

item #3: $5/20oz=$.25/oz

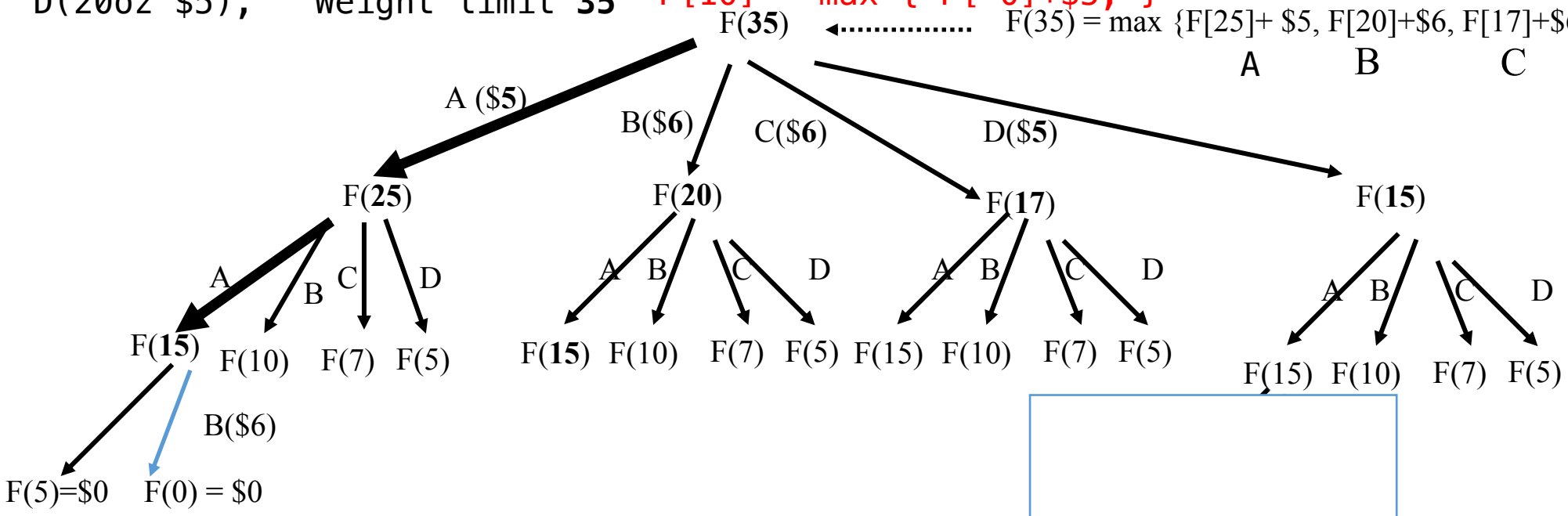
#**1**(**10oz**) ==> $**5**
#**2**(**15oz**) ==> $**6**
#**4** (**10oz** only) ==> $**3.33**
total $**14.333**

# **Unbounded** Knapsack

## Top-down Recursion

Consider following items
(weight, value):
A(**10oz**, $**5**),
B(**15oz**, $**6**),
C(18oz, $6),
D(20oz $5),  Weight limit **35**

F(35) = max {F[25]+ $5, F[20]+$6, F[17]+$6,  F[15]+$ }
F[25] = max { F[15]+$5, F[10]+$6, ~~F[7]~~ + $6, ~~F[5]~~+$5 }
F[20] = max { F[10]+$5, ~~F[5 ]~~+$6, ~~F[2]~~ + $6, ~~F[0]~~+$5 }
F[17] = max { ~~F[ 7]~~+$5, ~~F[2 ]~~+$6, }
F[15] = max { ~~F[ 5]~~+$5, ~~F[0 ]~~+$6, }
F[10] = max { ~~F[ 0]~~+$5, }




What will be the most money? $16
What items to pick? ___items A,A,B,____

# Number of Coin Change

- http://www.geeksforgeeks.org/dynamic-programming-set-7-coin-change/
- Given some denomination of coins (e.g., $1,$2,$5), Find the total number of coins to make change for a value (e.g, $11)

- Unbounded knapsack: Unlimited number of coins for each denomination
  - Coins = [$1, $2, $5], amount = $11 ⇒ return 3 ($11 = $5 + $5 + $1)
- 0-1 Knapsack problem: Use each denomination at most once
  - Coins = [$1, $2, $5], amount = $11 ⇒ no solution

# Unbounded Knapsack

## LeetCode#322. Fewest Coin Change

- Define D[i] = fewest coins to make up $j.
- D[i][j] = min{ D[ j - $i ] + 1 },  } for all  possible coin with value $i
- Example:
  - Coins = [$2, $5,$7,$10], amount = $13, fewest coins=? Answer = 4(1 pt)  (using one $7 and three $2)

```
F[13] = 1 + min {F[11],F[8],F[6].F[3] }  = 1 + 3 = 4
F[11] = 1 + min {F[ 9],F[6],F[4].F[1] } = 1 + 2 = 3
F[ 9] = 1 + min {F[ 7],F[4],F[2] } = 2
F[ 8] = 1 + min {F[ 6],F[3],F[1] = 4
F[ 7] = 1
F[ 6] = 1 + min {F[ 4], F[1] }  = 3
F[ 4] = 1 + min {F[ 2] } = 2
```

# Formal definition (0-1 problem)

- Knapsack has weight limit $W$
- Items labeled $1, 2, \ldots, n$ (arbitrarily)
- Items have weights $w_1, w_2, \ldots, w_n$
  - Assume all weights are integers
  - For practical reason, only consider $w_i < W$
- Items have values $v_1, v_2, \ldots, v_n$
- Objective: find a subset of items, $S$, such that $\sum_{i \in S} w_i \leq W$ and $\sum_{i \in S} v_i$ is maximal among all such (*feasible*) subsets

# Naïve algorithms

- Enumerate all subsets.
  - Optimal. But exponential time
- Greedy 1: take the item with the largest value
  - Not optimal
  - Give an example
- Greedy 2: take the item with the largest value/weight ratio
  - Not optimal
  - Give an example

# 0-1 Knapsack

## Brute force!

- 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? $O(n\ 2^n)$
- Lets try the obvious greedy strategy .

# 0-1 Knapsack

## Example with "**brute force**"

*S* = { ( *A* , 5, $70 ), (*B* ,10, $90 ), ( *C*, 25, $140 ) } , W=25

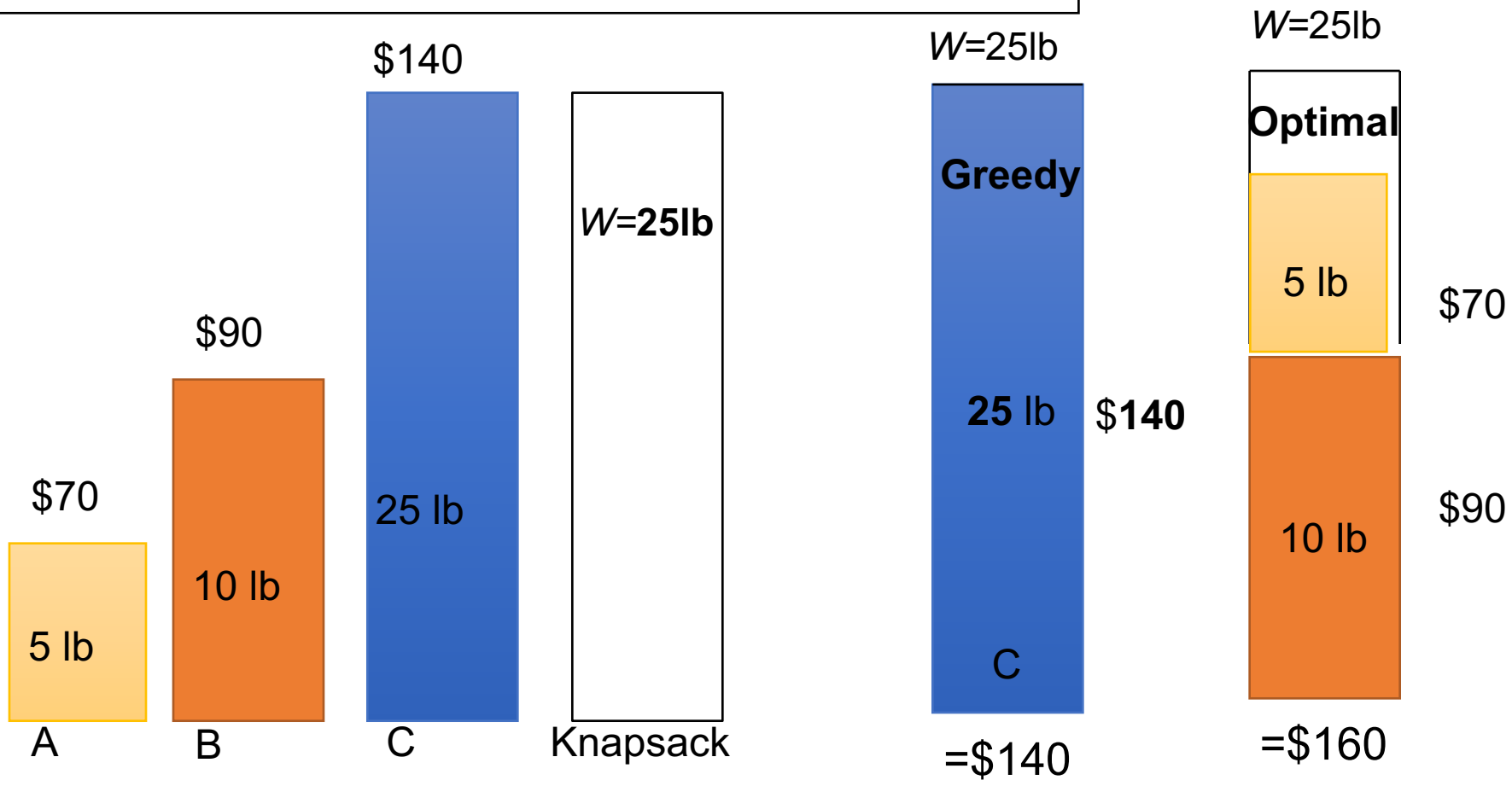
Subsets:

| | |
|---|---|
| 1. {} | Profit=$0 |
| 2. (*A* , 5, $70 ) | Profit=$70 |
| 3. (*B* ,10, $90 ) | Profit=$90 |
| 4. (*C*, 25, $140 ) | Profit=$140 |
| **5. (*A* , 5, $70 ), (*B* ,10, $90 )** | **Profit=$160 ****** |
| 6. (*B* ,10, $90 ), ( *C*, 25, $140 ) | exceeds W |
| 7. ( *A* , 5, $70 ), ( *C*, 25, $140 ) | exceeds W |
| 8. ( *A* , 5, $70 ), (*B* ,10, $90 ), ( *C*, 25, $140 ) | exceeds W |

# 0-1 Knapsack

Greedy 1: ***Maximum beneficial*** item.
Counter Example:

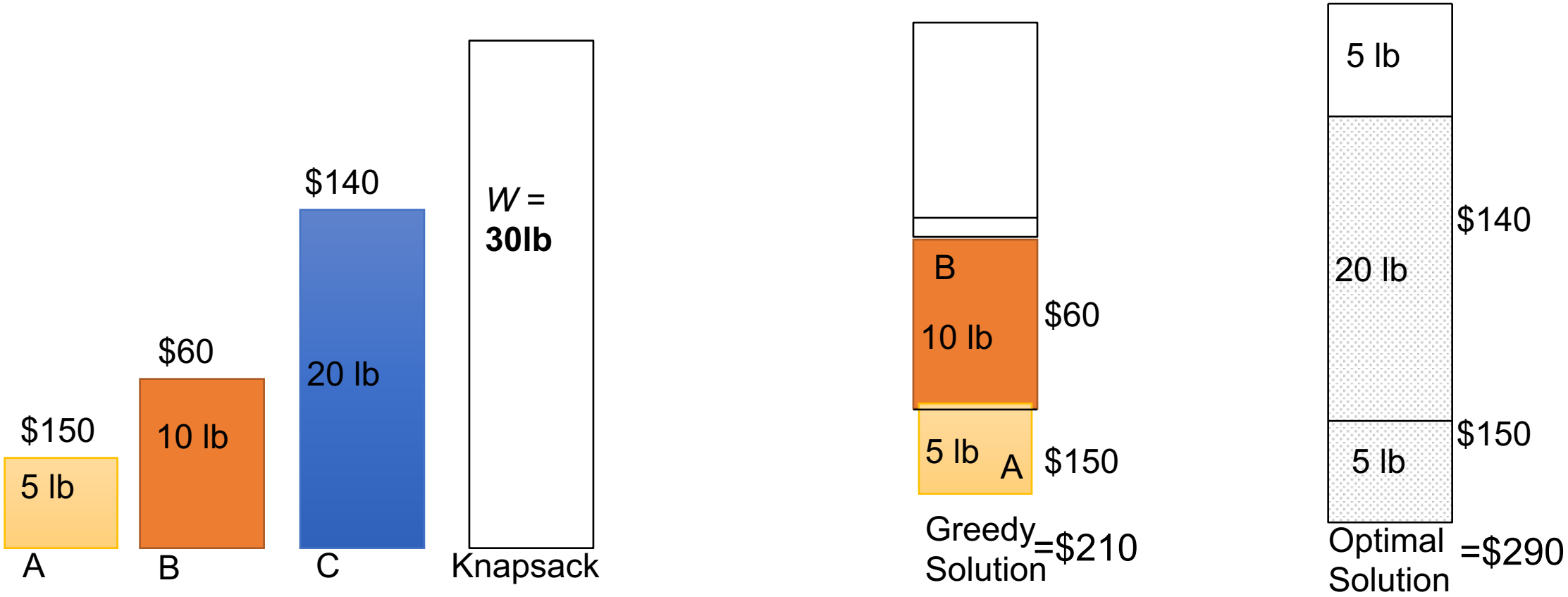
Counter Example:( *A* , 5, $70 ), ( *B* ,10, $90 ), (*C*, **25**, $140 )



# 0-1 Knapsack

## Greedy 2: ***Minimum weight*** item

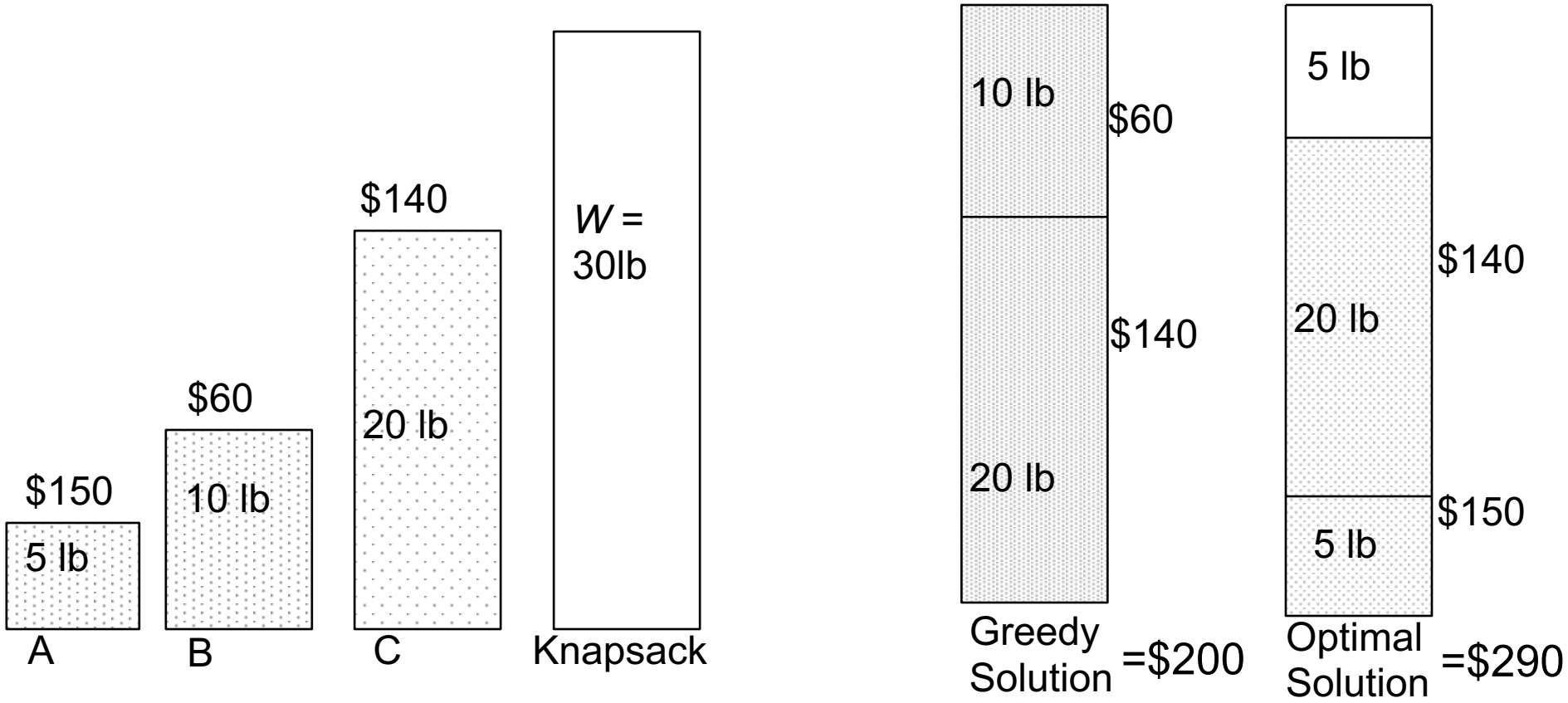
**Counter Example: *S* = { ( *A* , 5lb, $150 ), (*B* ,10, $60 ), ( *C*, 20, $140 ) }**



# 0-1 Knapsack

## Greedy 3: ***Maximum weight*** item

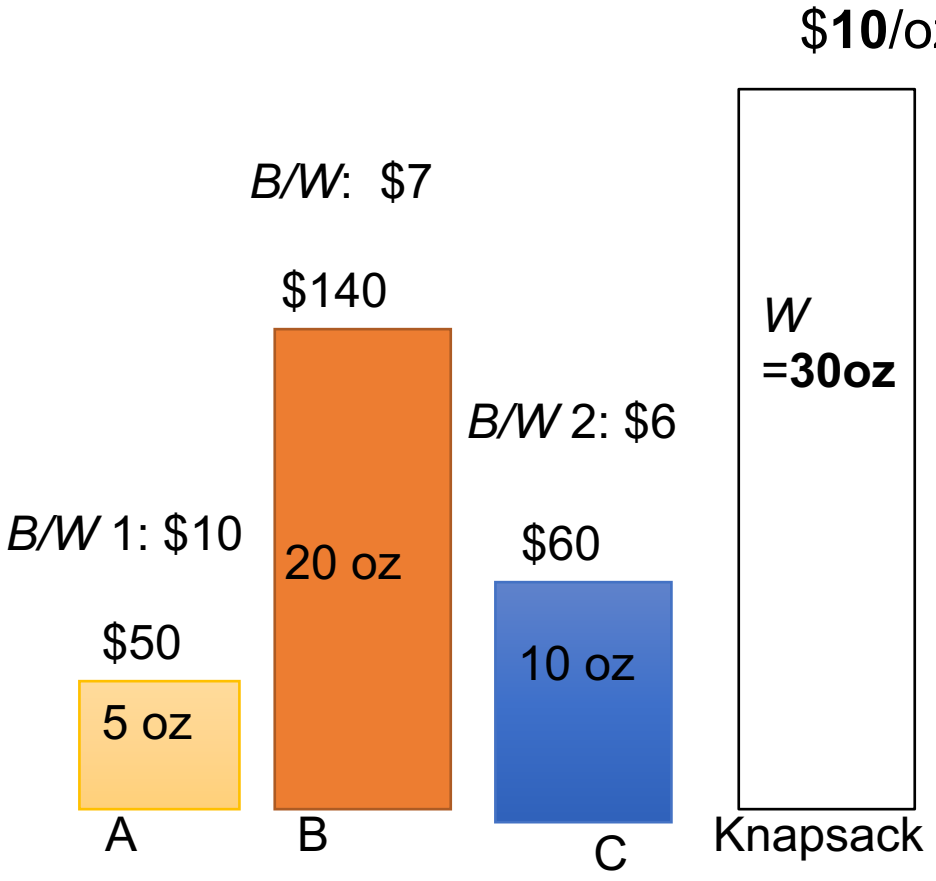
Counter Example: $S = \{ (A, 5, \$150), (B, 10, \$60), (C, 20, \$140) \}$

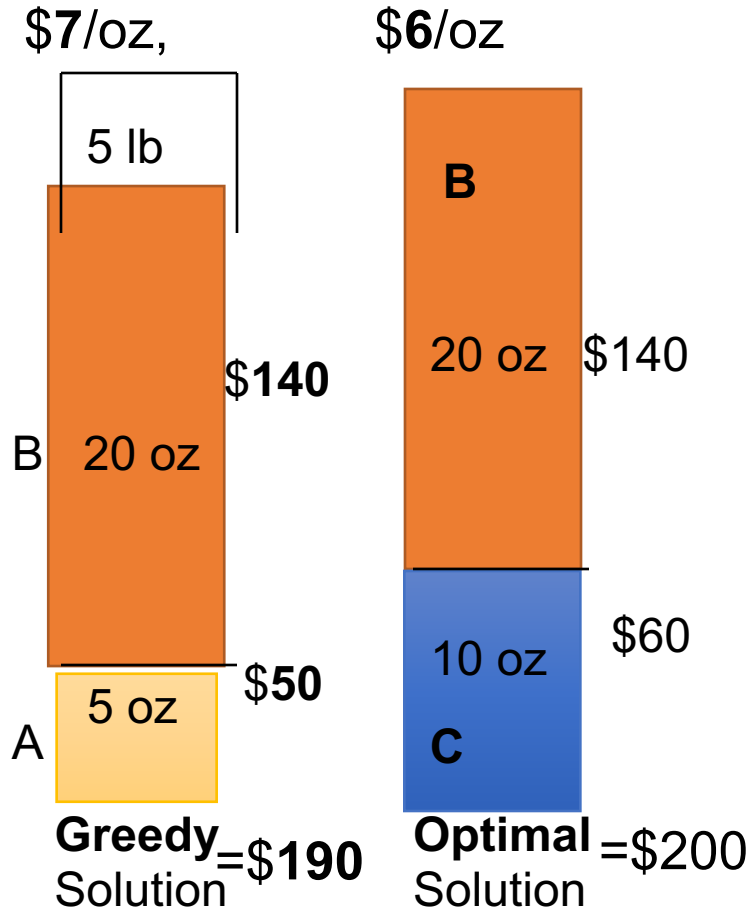


# **0-1** Knapsack

Greedy 4: ***Maximum benefit per unit*** item

Counter Example $S$ = { (***A*** , 5oz, $50 ), ( ***B***, 20oz, $140 ) (***C*** ,10oz, $60 ), }



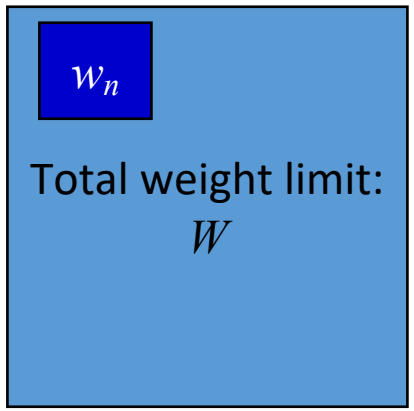




runtime?

# A DP algorithm

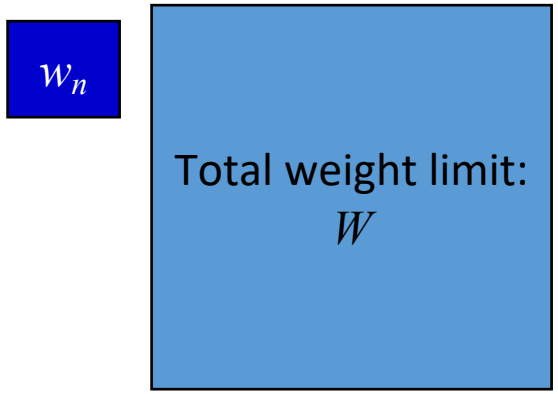
Suppose you've find the optimal solution S

Case 1: item $n$ is included




Find an optimal solution using items $1, 2, \ldots, n\text{-}1$ with weight limit $W - w_n$

Case 2: item $n$ is not included



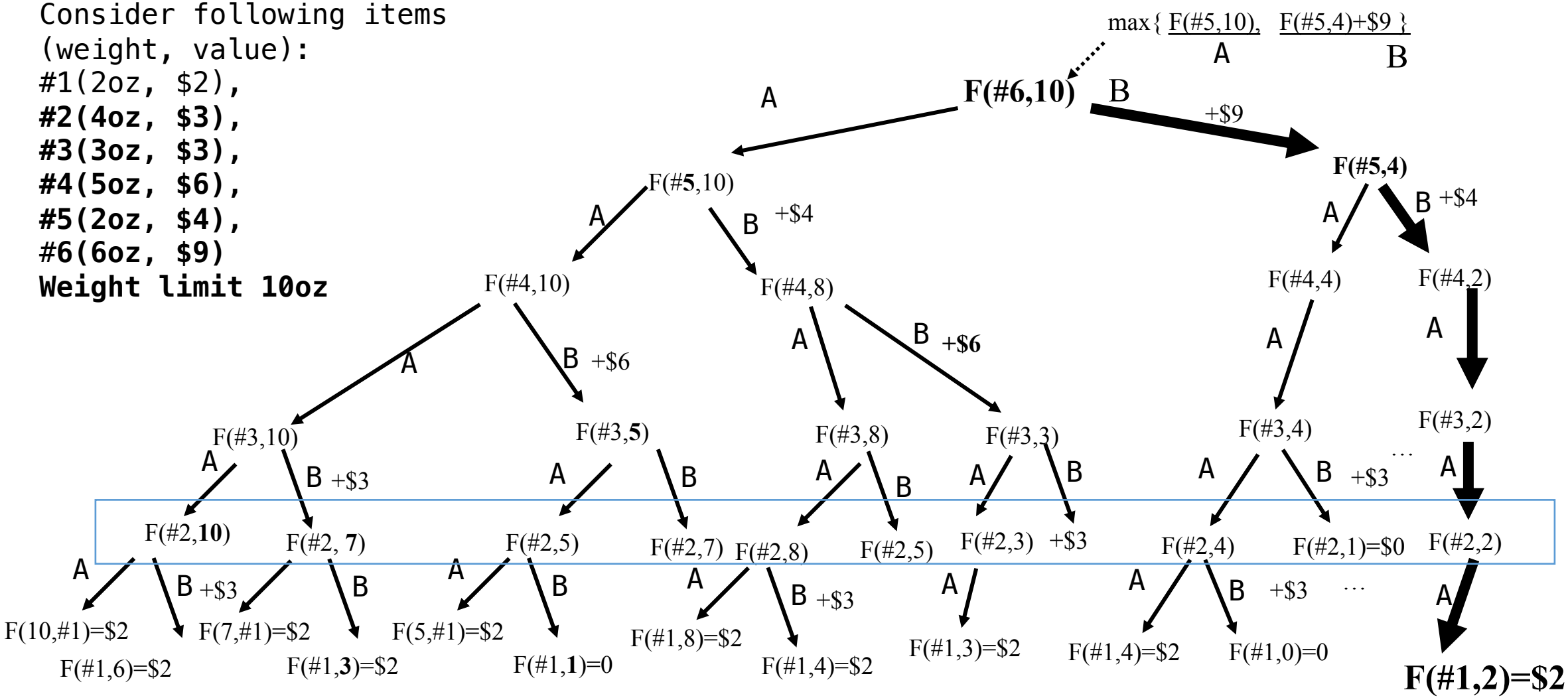

Find an optimal solution using items $1, 2, \ldots, n\text{-}1$ with weight limit $W$

$$F[i, w] = \max \begin{cases} F[\mathbf{i\text{-}1}, w\text{-}w_i] + v_i & \text{Case I: item } i \text{ is taken, and } w >= w_i \\ F[i\text{-}1, w] & \textit{Case II:} \text{ item } i \text{ not taken} \end{cases}$$

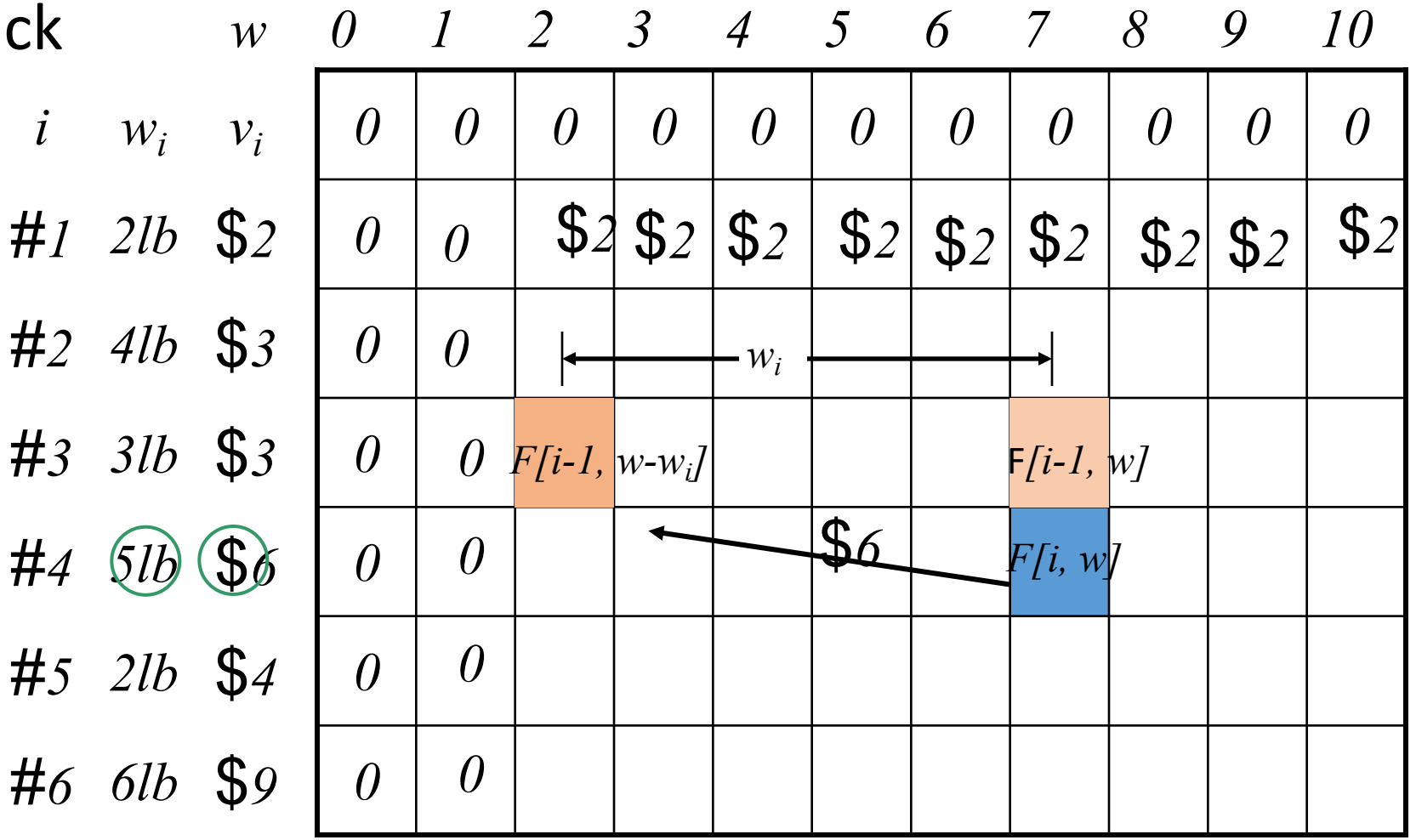
# 0-1 Knapsack

$$F[i, w] = \max \begin{cases} F[i-1, w-w_i] + v_i & \text{Case I: item } i \text{ is taken, and } w >= w_i \\ F[i-1, w] & \textit{Case II:} \text{ item } i \text{ not taken} \end{cases}$$

Consider following items (weight, value):
#1(2oz, $2),
**#2(4oz, $3),**
**#3(3oz, $3),**
**#4(5oz, $6),**
**#5(2oz, $4),**
**#6(6oz, $9)**
**Weight limit 10oz**



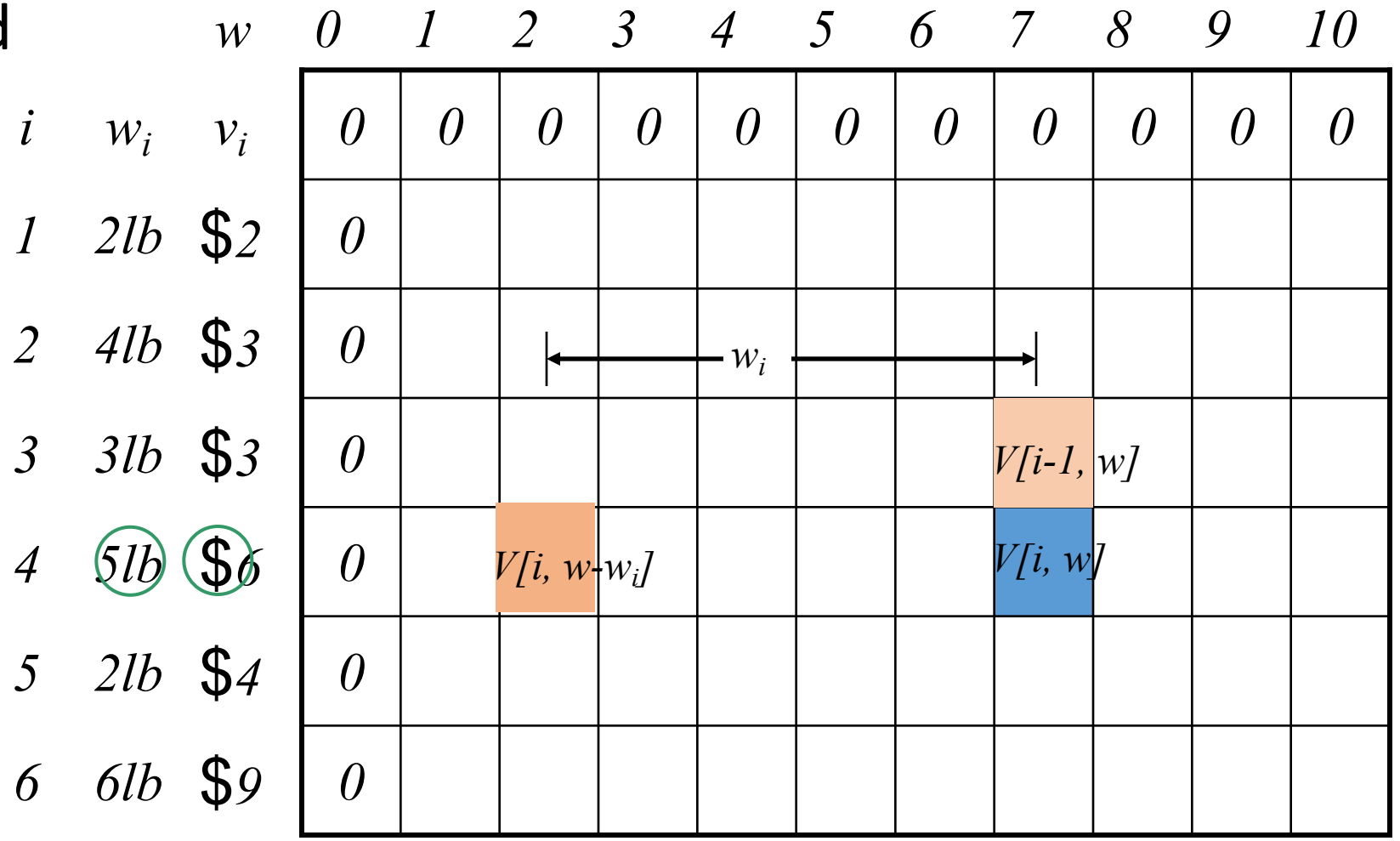
# 0-1 Knapsack




- *V[i, w]* = optimal total value when only considering items *1, 2, …, i* with weight limit *w*

$$F[i, w] = \max \begin{cases} F[\mathbf{i-1}, w\text{-}w_i] + v_i & \text{Case I: item } i \text{ is taken, and } w >= w_i \\ F[i\text{-}1, w] & \textit{Case II:} \text{ item } i \text{ not taken} \end{cases}$$

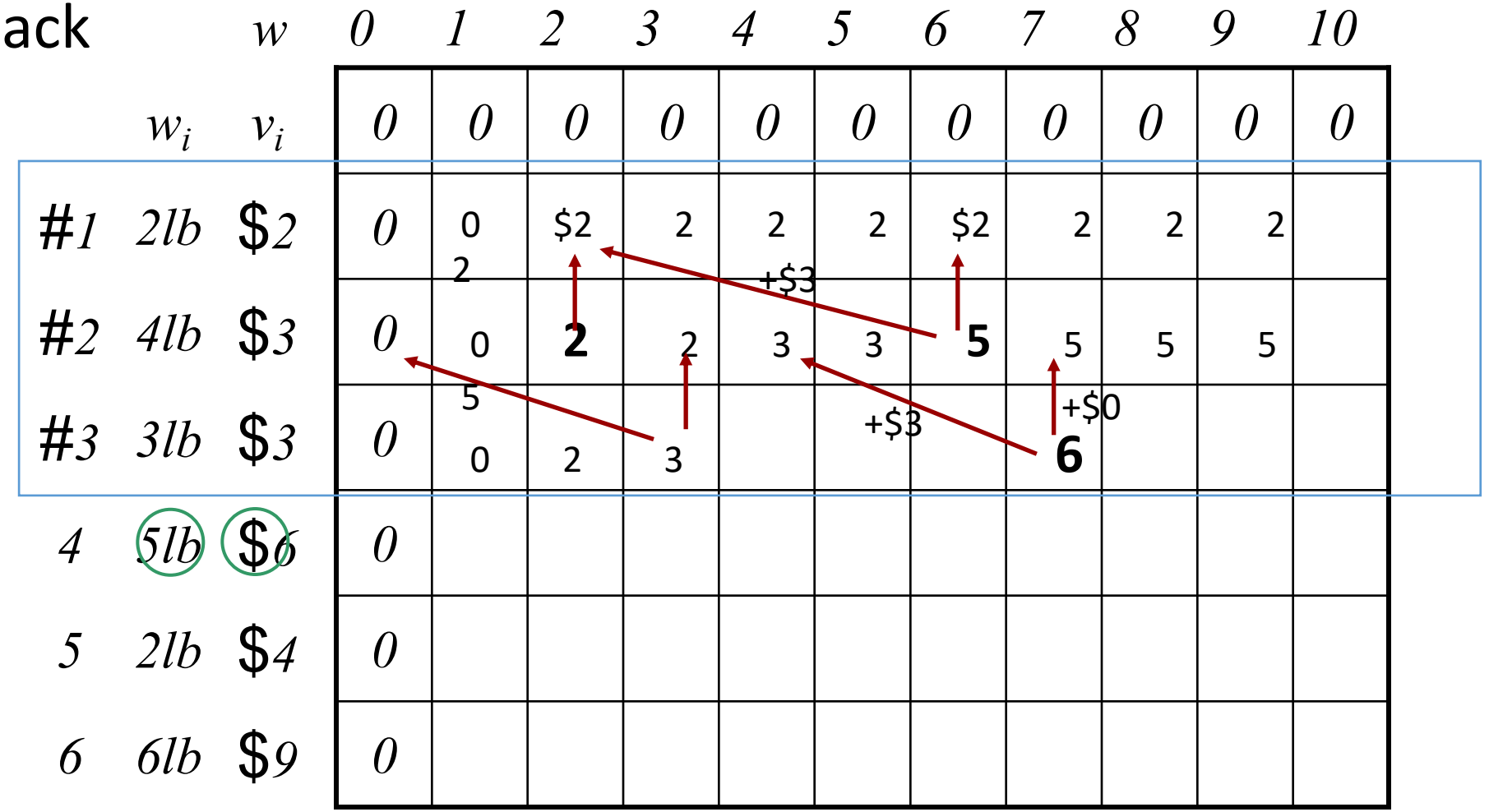
# Unbounded Knapsack




- $V[i, w]$ = optimal total value when only considering items $1, 2, \ldots, i$ with weight limit $w$

$$V[i, w] = \max \begin{cases} V[\mathbf{i}, w\text{-}w_i] + v_i & \text{Case I: item } i \text{ is taken, and } w >= w_i \\ V[i\text{-}1, w] & \textit{Case II:} \text{ item } i \text{ not taken any more} \end{cases}$$

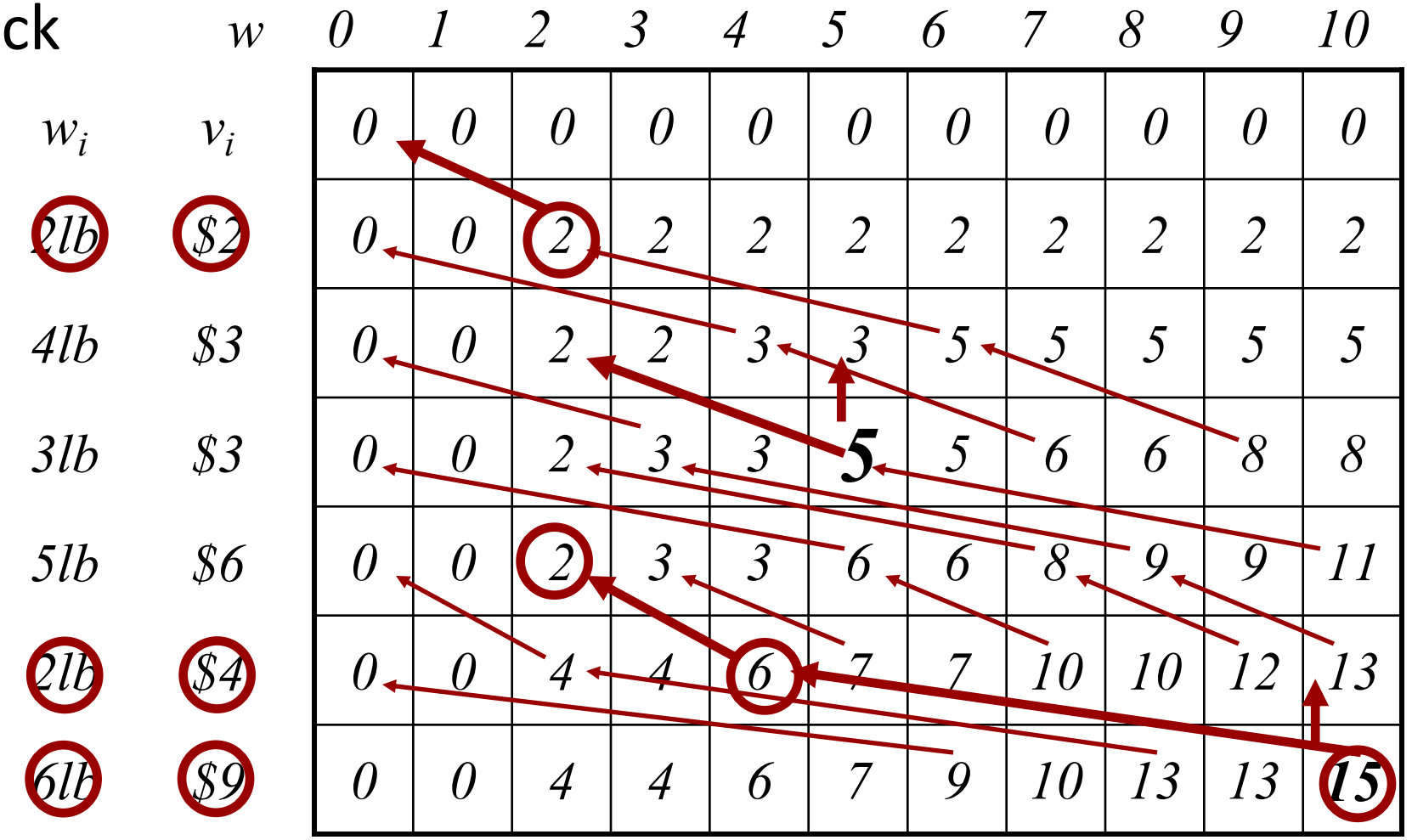
# 0-1 Knapsack




- $V[i, w]$ = optimal total value when only considering items $1, 2, \ldots, i$ with weight limit $w$

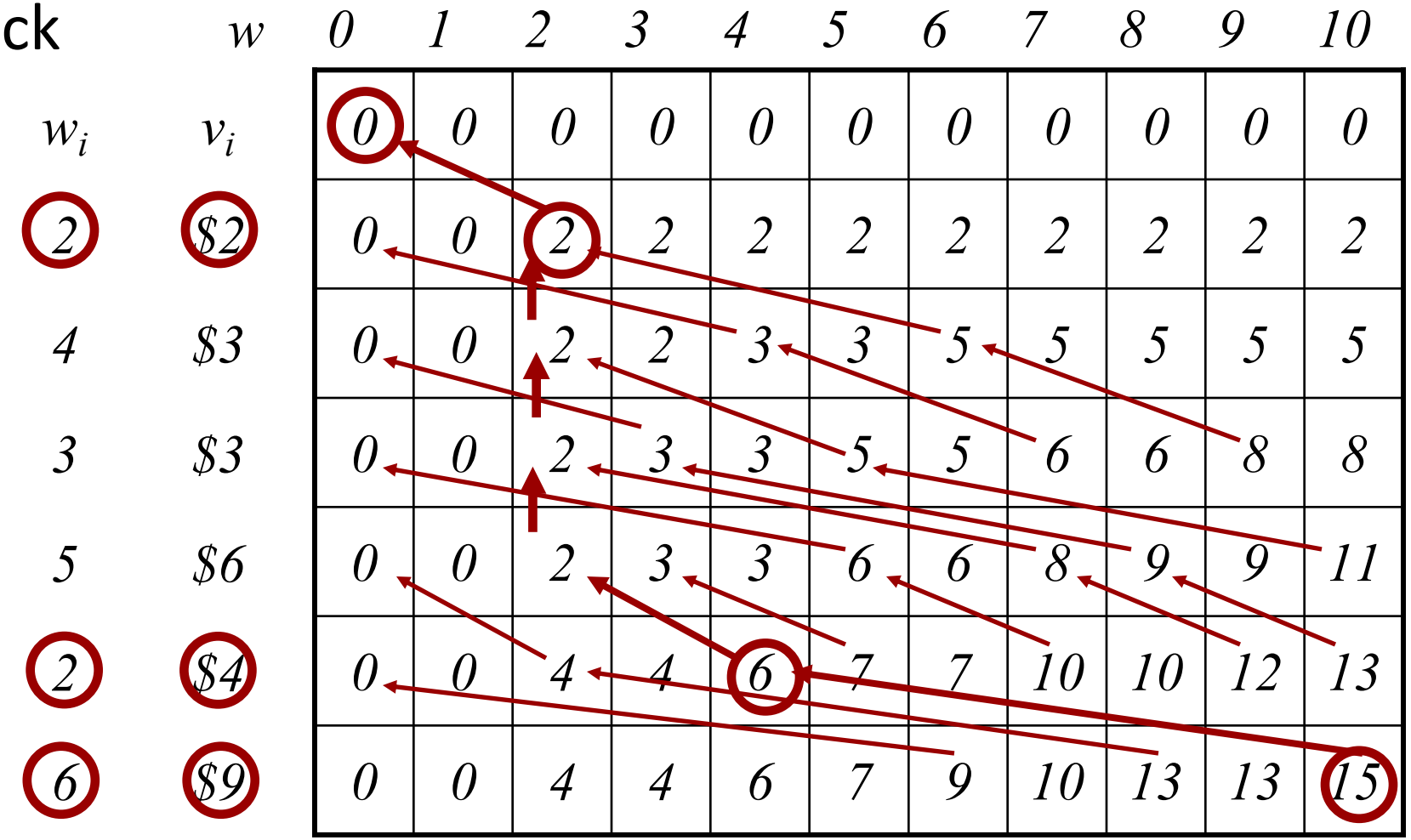
$$V[i, w] = max \begin{cases} V[i\text{-}1, w\text{-}w_i] + v_i & \text{item } i \text{ is taken, and } w >= w_i \\ V[i\text{-}1, w] & \text{item } i \text{ not taken} \end{cases}$$

# 0-1 Knapsack




$$V[i, w] = max \begin{cases} V[i\text{-}1, w\text{-}w_i] + v_i & \text{item } i \text{ is taken, and } w >= w_i \\ V[i\text{-}1, w] & \text{item } i \text{ not taken} \end{cases}$$

# 0-1 Knapsack




Item: 6, 5, 1

Weight: 6 + 2 + 2 = 10

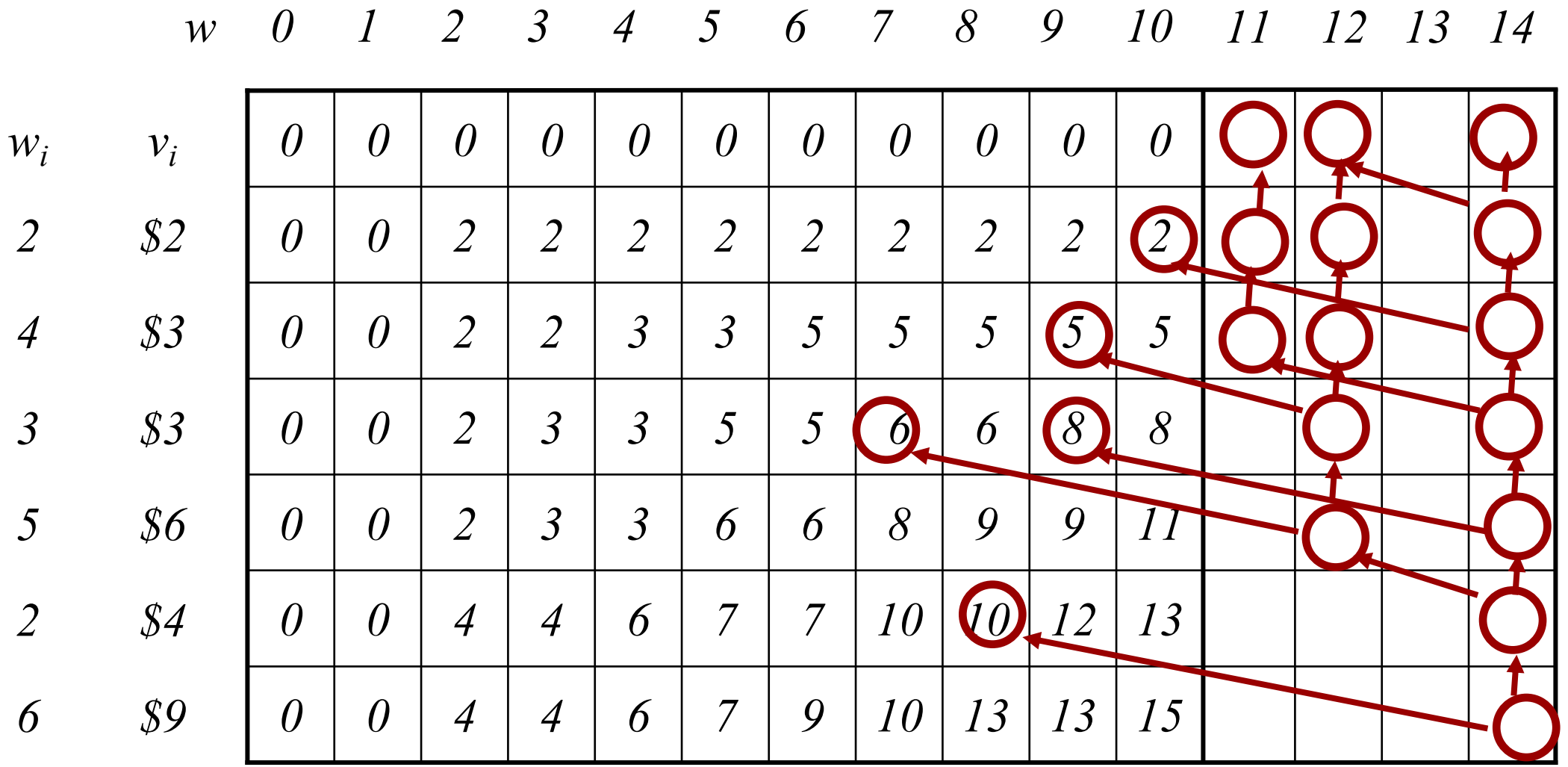
Value: 9 + 4 + 2 = 15

Optimal value: 15

# 0-1 Knapsack

| | $w$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w_i$ | $v_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | |
| 2 | \$2 | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | | | | |
| 4 | \$3 | 0 | 0 | 2 | 2 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | | | | |
| 3 | \$3 | 0 | 0 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 8 | 8 | | | | |
| 5 | \$6 | 0 | 0 | 2 | 3 | 3 | 6 | 6 | 8 | 9 | 9 | 11 | | | | |
| 2 | \$4 | 0 | 0 | 4 | 4 | 6 | 7 | 7 | 10 | 10 | 12 | 13 | | | | |
| 6 | \$9 | 0 | 0 | 4 | 4 | 6 | 7 | 9 | 10 | 13 | 13 | 15 | | | | |

# 0-1 Knapsack



| | | $w$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w_i$ | $v_i$ | | | | | | | | | | | | | | | | |
| | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | | | | |
| 2 | \$2 | | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | | | | |
| 4 | \$3 | | 0 | 0 | 2 | 2 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | | | | |
| 3 | \$3 | | 0 | 0 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 8 | 8 | | | | |
| 5 | \$6 | | 0 | 0 | 2 | 3 | 3 | 6 | 6 | 8 | 9 | 9 | 11 | | | | |
| 2 | \$4 | | 0 | 0 | 4 | 4 | 6 | 7 | 7 | 10 | 10 | 12 | 13 | | | | |
| 6 | \$9 | | 0 | 0 | 4 | 4 | 6 | 7 | 9 | 10 | 13 | 13 | 15 | | | | |

# 0-1 Knapsack

| | | $w$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $w_i$ | $v_i$ | | | | | | | | | | | | | | | | |
| | | | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | \$2 | | 0 | 0 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 | 2 |
| 4 | \$3 | | 0 | 0 | 2 | 2 | 3 | 3 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 | 5 |
| 3 | \$3 | | 0 | 0 | 2 | 3 | 3 | 5 | 5 | 6 | 6 | 8 | 8 | 8 | 8 | 8 | 8 |
| 5 | \$6 | | 0 | 0 | 2 | 3 | 3 | 6 | 6 | 8 | 9 | 9 | 11 | 11 | 12 | 12 | 14 |
| 2 | \$4 | | 0 | 0 | 4 | 4 | 6 | 7 | 7 | 10 | 10 | 12 | 13 | 13 | 15 | 15 | 16 |
| 6 | \$9 | | 0 | 0 | 4 | 4 | 6 | 7 | 9 | 10 | 13 | 13 | 15 | 16 | 16 | 19 | 19 |

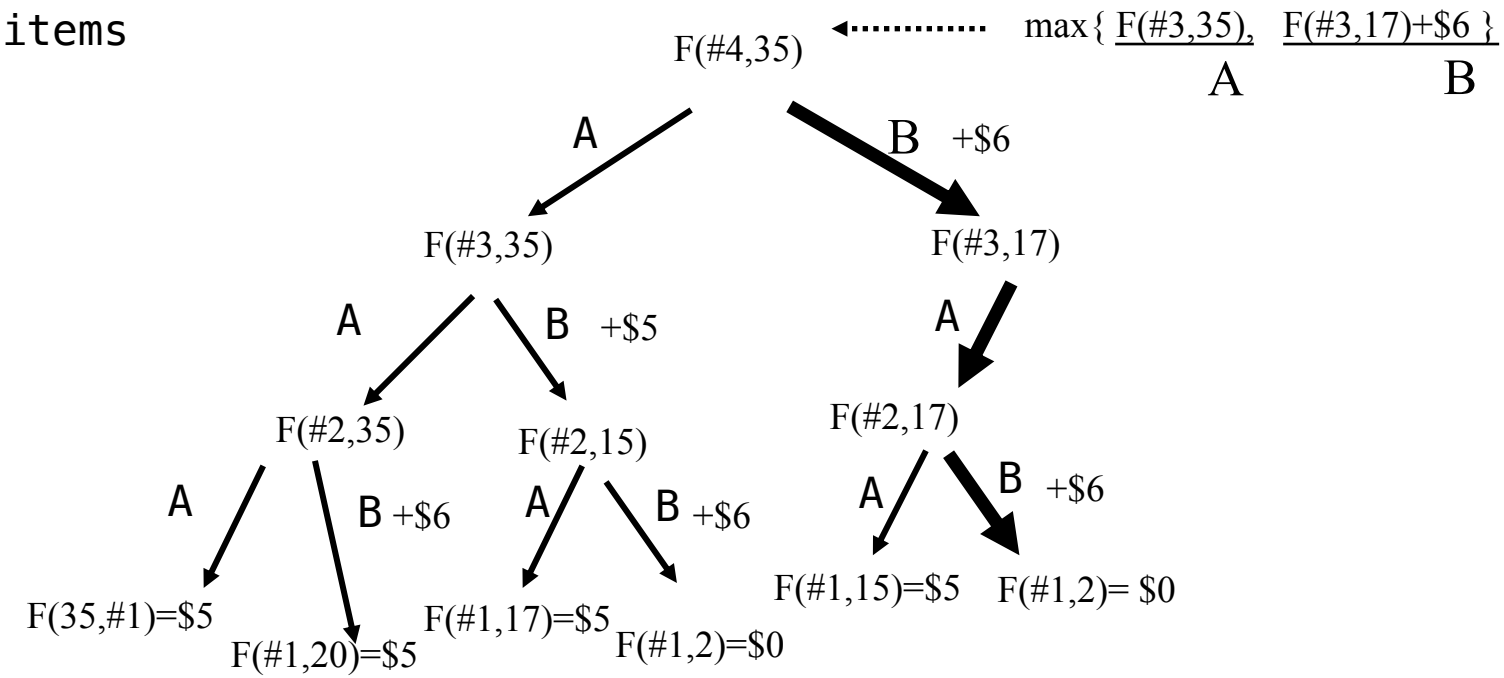
# Time complexity

- Θ (nW)
- Polynomial?
  - Pseudo-polynomial
  - Works well if W is small
- Consider following items (weight, value):

  (10oz, $5), (15oz, $6), (20,oz $5), (18oz, $6)
- Weight limit 35
  - Brute-Force: 2^4 = 16 subsets
  - Dynamic programming: fill up a #4 x 35lb = 140 table entries
- What's the problem?
  - Many entries are unused: no such weight combination
  - **Top-down may be better**

# 0-1 Knapsack: Top-down

Consider following items
(weight, value):
#1(10oz, $5),
#2(15oz, $6),
#3(20oz, $5),
#4(18oz, $6),
Weight limit 35




| W | V | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 | 33 | 34 | 35 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 10 | $5 | | 0 | 0 | 0 | 0 | | | | | $5 | | | | | 5 | | 5 | | | 5 | | | | | 5 | | | | | | | | | | | **5** |
| 15 | $6 | | 0 | | | **0** | | | | | | | | | | 6 | | 6 | | | **6** | | | | | | | | | | | | | | | **6** |
| 20 | $5 | | | | | | | | | | | | | | | **6** | | 6 | | | | | | | | | | | | | | | | | | **11** |
| 18 | $6 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | **12** |