



CS5800: Algorithms

Week 5 – Greedy Algorithm

Dr. Ryan Rad

Spring 2025



Northeastern
University

Greedy Algorithms

- Quick Review of Dynamic Programming
- What is Greedy Algorithm?
- Greedy vs Dynamic Programming
- Activity-Selection Problem
- Correctness for Optimal Greedy Choice
- Knapsack Problem
- House Robber
- Course Survey Response





Homework 3

Preview

HW3 – Question 1

Problem #1 – Longest Subsequence Problem

The *Longest Subsequence Problem* is a well-studied problem in Computer Science, where given a sequence of distinct positive integers, the goal is to output the longest subsequence whose elements appear from smallest to largest, or from largest to smallest.

For example, consider the sequence $S = [9, 7, 4, 10, 6, 8, 2, 1, 3, 5]$. The longest increasing subsequence of S has length three ($[4, 6, 8]$ or $[2, 3, 5]$), and the longest decreasing subsequence of S has length five ($[9, 7, 4, 2, 1]$ or $[9, 7, 6, 2, 1]$).

And if we have the sequence $S = [531, 339, 298, 247, 246, 195, 104, 73, 52, 31]$, then the length of the longest increasing subsequence is 1 and the length of the longest decreasing subsequence is 10.

- (a) Find a sequence with nine distinct integers for which the length of the longest increasing subsequence is 3, and the length of the longest decreasing subsequence is 3. Briefly explain how you constructed your sequence.
- (b) Let S be a sequence with ten distinct integers. Prove that there *must* exist an increasing subsequence of length 4 (or more) or a decreasing subsequence of length 4 (or more).

Hint: for each integer k in the sequence you found in part (a), define the ordered pair $(x(k), y(k))$, where $x(k)$ is the length of the longest increasing subsequence beginning with k , and $y(k)$ is the length of the longest decreasing subsequence beginning with k . You should notice that each of your ordered pairs is different. Explain why this is not a coincidence, i.e., why it is impossible for two *different* numbers in your sequence to be represented by the *same* ordered pair $(x(k), y(k))$.

HW3 – Question 3



What does "for such a matrix" mean?

- It refers to **any arbitrary $m \times n$ matrix** that follows the problem definition (cells containing awake cats, sleeping cats, or empty spaces).
- The example in the assignment illustrates one possible case, but the actual matrix **can be different in size and structure**.
- The **top-left corner is not guaranteed** to contain an awake cat.

Is there always at least one awake cat in the matrix?

- **No**. The matrix is **randomly assigned**, meaning it may contain **only sleeping cats** at time 0.
- If no cats are awake initially, **no sleeping cat can wake up**, and the correct output should be -1 (indicating it will never wake up).
- Your algorithm must **check for this case explicitly**.

Do we know the number of empty cells in advance?

- **No**. You must **determine** the number of empty cells as part of your algorithm.

How many cats are in the matrix?

- The number of cats is **unknown in advance** and can vary. Your algorithm must **count the number of sleeping and awake cats**.

HW3 – Question 3

Could a sleeping cat be surrounded by empty cells?

- **Yes.** If a sleeping cat is completely surrounded by empty cells (with no awake cats nearby), it **will never wake up**.
- In such cases, your algorithm should return -1 if querying the wake-up time of that cat.

What if there is only one sleeping cat and no awake cats?

- If there are **only sleeping cats and no awake cats in the matrix**, no wake-up propagation is possible.
- Your algorithm should handle this case and return -1.

How do I determine the starting point for the wake-up process?

- Your algorithm **must find all initially awake cats** and use them as the starting points for the wake-up process.
- The wake-up propagation follows the **adjacent (up, down, left, right) rule**.

Does the cat go back to sleep after it's awake?

- **No.** Once a cat wakes up, it remains awake **permanently**.

Too-Many-Coins Problem



30 Minutes

Problem Statement:

You are given an array representing different coin denominations and a total amount. The task is to find the minimum number of coins needed to make up that amount. Assume an infinite supply of each coin denomination.

Example:

- Input:
 - Coin denominations: [1, 2, 5]
 - Total amount to make: 11
- Output: 3 (using two 5 coins and one 1 coin)



2 Dollar Coin
"Toonie"



1 Dollar Coin
"Loonie"



50 Cent Coin



25 Cent Coin
"Quarter"



10 Cent Coin
"Dime"



5 Cent Coin
"Nickel"



1 Cent Coin
"Penny"

Divide and Conquer (DAC) Approach:




Time Complexity?

- **Time Complexity:** Exponential, $O(2^n)$ - where n is the total amount. (This is due to the repeated computation of subproblems.)
- **Space Complexity:** $O(n)$ - The recursion depth can go up to n .

1. Divide and Conquer (DAC) Approach:

python

 Copy code

```
def minCoins_DAC(coins, total):  
    if total == 0:  
        return 0  
  
    result = float('inf')  
  
    for coin in coins:  
        if coin <= total:  
            subproblem_result = minCoins_DAC(coins, total - coin)  
            result = min(result, subproblem_result + 1)  
  
    return result  
  
# Example Usage:  
coins = [1, 2, 5]  
total_amount = 11  
result_DAC = minCoins_DAC(coins, total_amount)  
print(f"Minimum number of coins (DAC): {result_DAC}")
```


Memoized DP Approach:



Time Complexity?

- **Time Complexity:** $O(\text{coins} * \text{total})$ - where coins is the number of coin denominations. (Each subproblem is solved only once, and memoization avoids redundant computations.)
- **Space Complexity:** $O(\text{total})$ - The memoization table stores results for each subproblem.

2. Memoization Approach:

python

Copy code

```
def minCoins_Memo(coins, total, memo):  
    if total == 0:  
        return 0  
  
    if memo[total] != -1:  
        return memo[total]  
  
    result = float('inf')  
  
    for coin in coins:  
        if coin <= total:  
            subproblem_result = minCoins_Memo(coins, total - coin, memo)  
            result = min(result, subproblem_result + 1)  
  
    memo[total] = result  
    return result  
  
# Example Usage:  
coins = [1, 2, 5]  
total_amount = 11  
memo = [-1] * (total_amount + 1)  
result_Memo = minCoins_Memo(coins, total_amount, memo)  
print(f"Minimum number of coins (Memoization): {result_Memo}")
```

Bottom-Up DP Approach:



Time Complexity?

- **Time Complexity:** $O(\text{coins} * \text{total})$ - where coins is the number of coin denominations. (Similar to the memoization approach, each subproblem is solved only once.)
- **Space Complexity:** $O(\text{total})$ - The DP table stores results for each subproblem.

3. Bottom-Up Dynamic Programming Approach:

python

Copy code

```
def minCoins_BottomUp(coins, total):  
    dp = [float('inf')] * (total + 1)  
    dp[0] = 0  
  
    for i in range(1, total + 1):  
        for coin in coins:  
            if coin <= i:  
                dp[i] = min(dp[i], dp[i - coin] + 1)  
  
    return dp[total]  
  
# Example Usage:  
coins = [1, 2, 5]  
total_amount = 11  
result_BottomUp = minCoins_BottomUp(coins, total_amount)  
print(f"Minimum number of coins (Bottom-Up DP): {result_BottomUp}")
```

Greedy Approach:



Time Complexity?

- **Time Complexity:** $O(N)$, where N is the total amount. Sorting the coin denominations has a time complexity of $O(M \log M)$, where M is the number of coin denominations, but this is usually dominated by the linear iteration through the sorted denominations.
- **Space Complexity:**

python

Copy code

```
def minCoinsGreedy(coins, total):  
    coins.sort(reverse=True) # Sort denominations in descending order  
    num_coins = 0  
  
    for coin in coins:  
        while total >= coin:  
            total -= coin  
            num_coins += 1  
  
    return num_coins  
  
# Example Usage:  
coin_denominations = [1, 2, 5]  
total_amount = 11  
result = minCoinsGreedy(coin_denominations, total_amount)  
print(f"Minimum number of coins (Greedy): {result}")
```

Is **Greedy algorithm**
optimal for the Coin
Change problem ?



Greedy Approach:



- The greedy algorithm is optimal for the Coin Change problem under the assumption that the coin system is a **canonical coin system**,

Canonical Denomination:

- refers to a set of coin values where each coin's value is a multiple of the denomination of the smaller coin

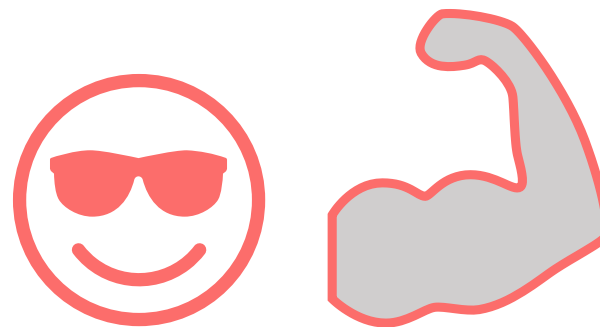
Example:

Coin denominations: [1, 3, 4], and a total amount of 6

- **Greedy:** 3 Coins
- **Optimal:** 2 Coins



322. Coin Change



You just solved a medium Leetcode question on DP!

Greedy vs Dynamic

Items	Greedy	Dynamic
Decision-Making	Opportunistic	Realistic
Optimality	Sometimes	Always
Complexity	Less Complex	More Complex

The activity-selection Problem

In this problem, you have a list of proposed activities $1, \dots, n$.

Activity i starts at time s_i and finishes at time f_i . You want to do as many activities as possible while avoiding time conflicts.

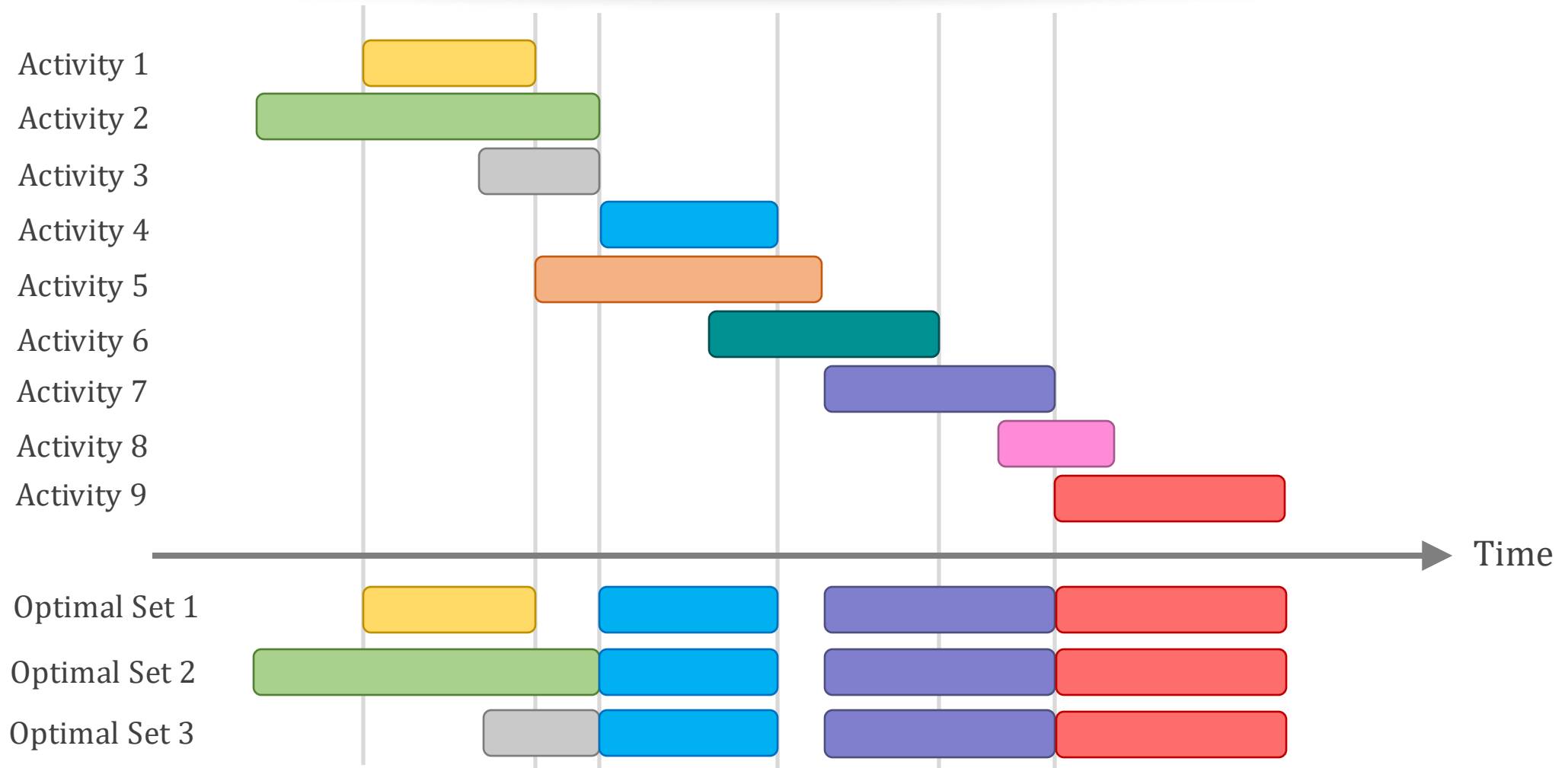
Formally, we want to find a maximum-size subset of mutually compatible activities. Activities i and j are compatible if the intervals $[s_i, f_i)$ and $[s_j, f_j)$ do not overlap.

i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

Important:

- We assume that the activities are sorted in monotonically increasing order of finish time, i.e. $f_1 \leq f_2 \leq \dots \leq f_n$

The activity-selection Problem



The activity-selection Problem

Let's assume a_k is in an optimal solution, then we are left with two subproblems:

- Finding mutually compatible activities in the set S_{ik} (activities that start after activity a_i finishes and that finish before activity a_k starts) and
- Finding mutually compatible activities in the set S_{kj} (activities that start after activity a_k finishes and that finish before activity a_j starts).

This way of characterizing optimal substructure suggests that you can solve the activity-selection problem by dynamic programming.

Let's denote the size of an optimal solution for the set S_{ij} by $c[i, j]$. Then, the dynamic-programming approach gives the recurrence: $c[i, j] = c[i, k] + c[k, j] + 1$

We need to consider all possible cases for a_k

$$c[i, j] = \begin{cases} 0 & \text{if } S_{ij} = \emptyset, \\ \max \{c[i, k] + c[k, j] + 1 : a_k \in S_{ij}\} & \text{if } S_{ij} \neq \emptyset. \end{cases}$$

The activity-selection Problem (Greedy choice)

What is the greedy choice for the activity-selection problem?

Intuition suggests that you should **choose an activity** that **leaves the maximum resources available** for as many other activities as possible.

Since the activities are sorted in monotonically increasing order by finish time, the greedy choice is activity **a_1** .

Choosing the **first activity to finish** is one way to think of making a greedy choice for this problem.

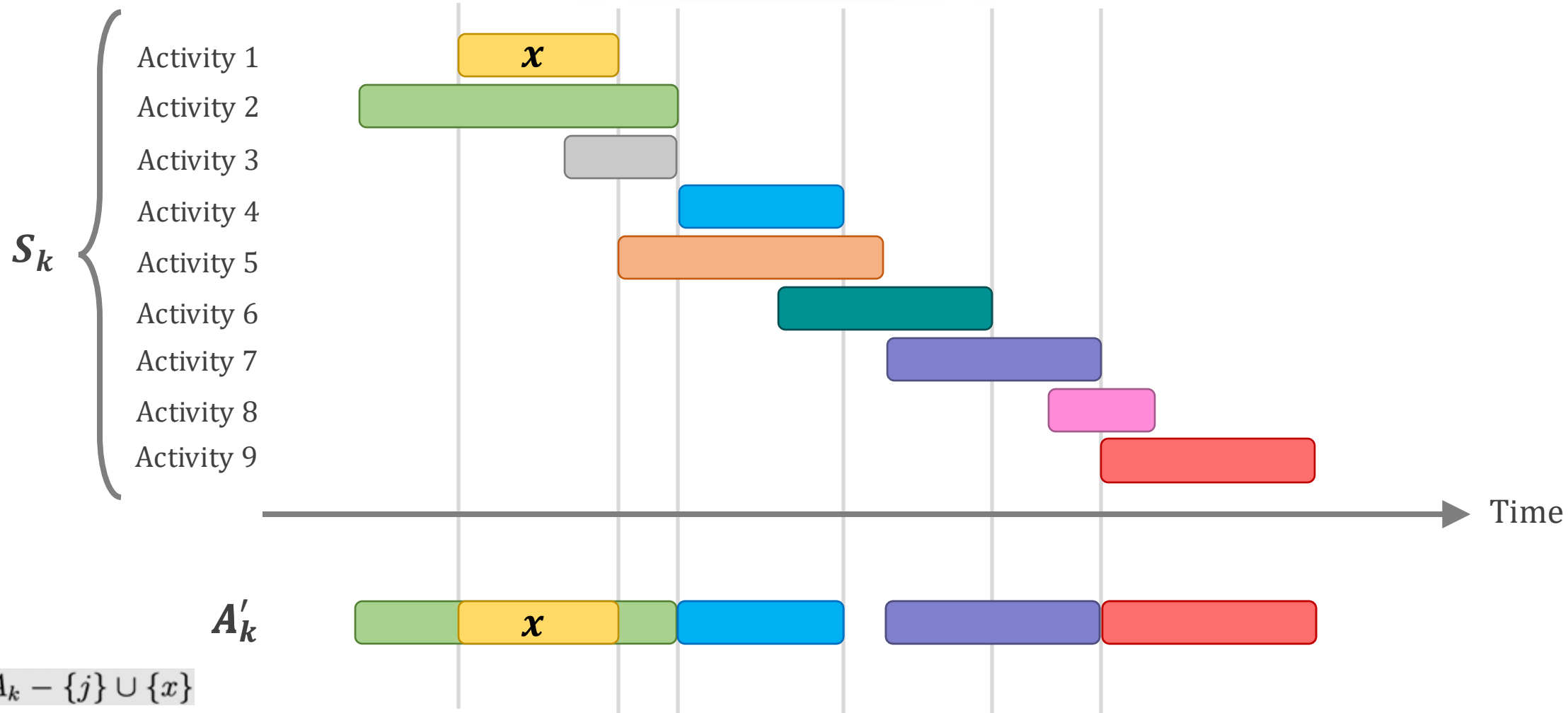
Correctness Proof (Greedy choice for activity-selection problem)

Theorem: Consider any nonempty subproblem S_k , and let x be an activity in S_k with the earliest finish time. Then x is included in some maximum-size subset of mutually compatible activities in S_k .

Proof: Suppose we have an “optimal subset” of activities A_k (note that there may be more than one optimal subset). Let j be the activity in A_k with the earliest finishing time. If $j = x$, then we are done, since we have shown that x is some maximum size subset of mutually compatible activities of S_k .

If $j \neq x$, then consider the set of activities $A'_k = A_k - \{j\} \cup \{x\}$, where we swap out the first activity in the “optimal subset” with the first activity in the entire subproblem. The activities in A'_k are disjoint, which follows because the activities in A_k are disjoint, j is the first activity in A_k to finish, and $f_x \leq f_j$. Since $|A'_k| = |A_k|$, we conclude that A'_k is a maximum size subset of mutually compatible activities of S_k , and it includes activity x .

Correctness Proof (Greedy choice for activity-selection problem)



The Greedy choice

Warning!

Just because there is a greedy algorithm that leads to an optimal solution does not mean that all greedy choices lead to an optimal solution!

- Picking the activity with the **shortest duration** can lead to a non-optimal solution
- Picking the activity with the **longest duration** can lead to a non-optimal solution
- Picking the activity with the **earliest start time** can lead to a non-optimal solution
- Picking the activity with the **latest finish time** can lead to a non-optimal solution

Identical to the “Earliest Finish time”!

How about:

- Picking the activity with the **latest start time**

Quick Question

Question

- Is Greedy Top-Down or Bottom-Up?

Answer

Top-Down (Recursive)

Bottom-Up (Iterative)

Greedy is not about the order Top-Down or Bottom-Up!
It's all about the **strategy**, the **Greedy choice**!

The activity-selection Problem (Recursive Greedy)

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

```
1   $m = k + 1$ 
2  while  $m \leq n$  and  $s[m] < f[k]$       // find the first activity in  $S_k$  to finish
3       $m = m + 1$ 
4  if  $m \leq n$ 
5      return  $\{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6  else return  $\emptyset$ 
```

Question

- Why this highlighted line needed?

The activity-selection Problem

Recursive Greedy

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

```
1   $m = k + 1$ 
2  while  $m \leq n$  and  $s[m] < f[k]$     // find the first activity in  $S_k$  to finish
3       $m = m + 1$ 
4  if  $m \leq n$ 
5      return  $\{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6  else return  $\emptyset$ 
```

The initial call to solve the entire problem:

Recursive-Activity-Selector($s, f, 0, 11$)

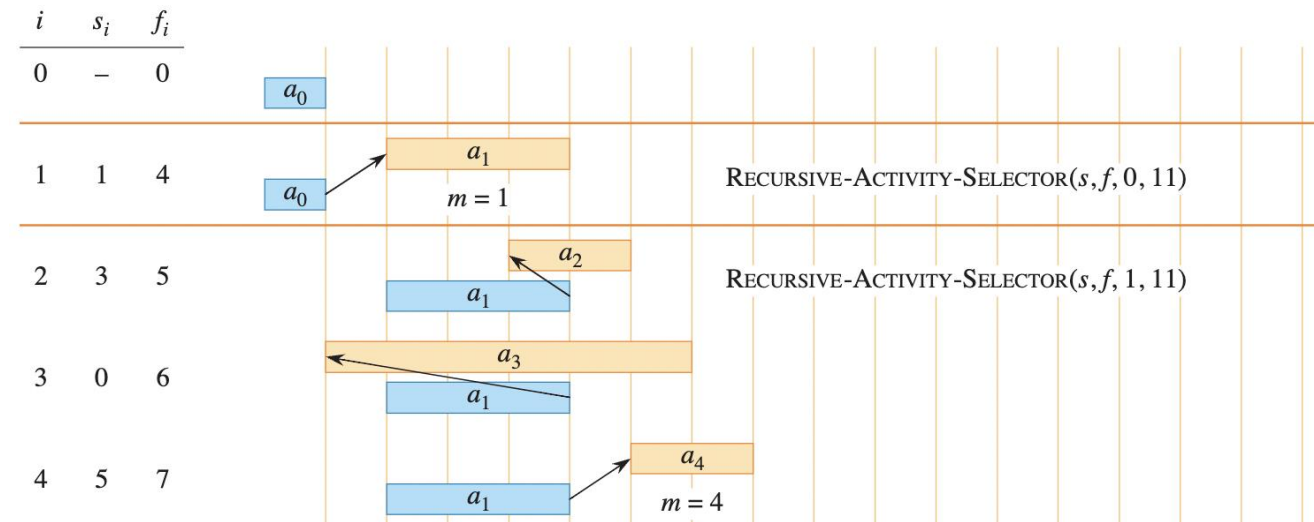
i	1	2	3	4	5	6	7	8	9	10	11
s_i	1	3	0	5	3	5	6	7	8	2	12
f_i	4	5	6	7	9	9	10	11	12	14	16

i	0	1	2	3	4	5	6	7	8	9	10	11
s_i	-1	1	3	0	5	3	5	6	7	8	2	12
f_i	0	4	5	6	7	9	9	10	11	12	14	16



The activity-selection Problem

i	0	1	2	3	4	5	6	7	8	9	10	11
s_i	-1	1	3	0	5	3	5	6	7	8	2	12
f_i	0	4	5	6	7	9	9	10	11	12	14	16



The next recursive call:
 Recursive-Activity-Selector ($s, f, 1, 11$)

The activity-selection Problem

i	0	1	2	3	4	5	6	7	8	9	10	11
s_i	-1	1	3	0	5	3	5	6	7	8	2	12
f_i	0	4	5	6	7	9	9	10	11	12	14	16

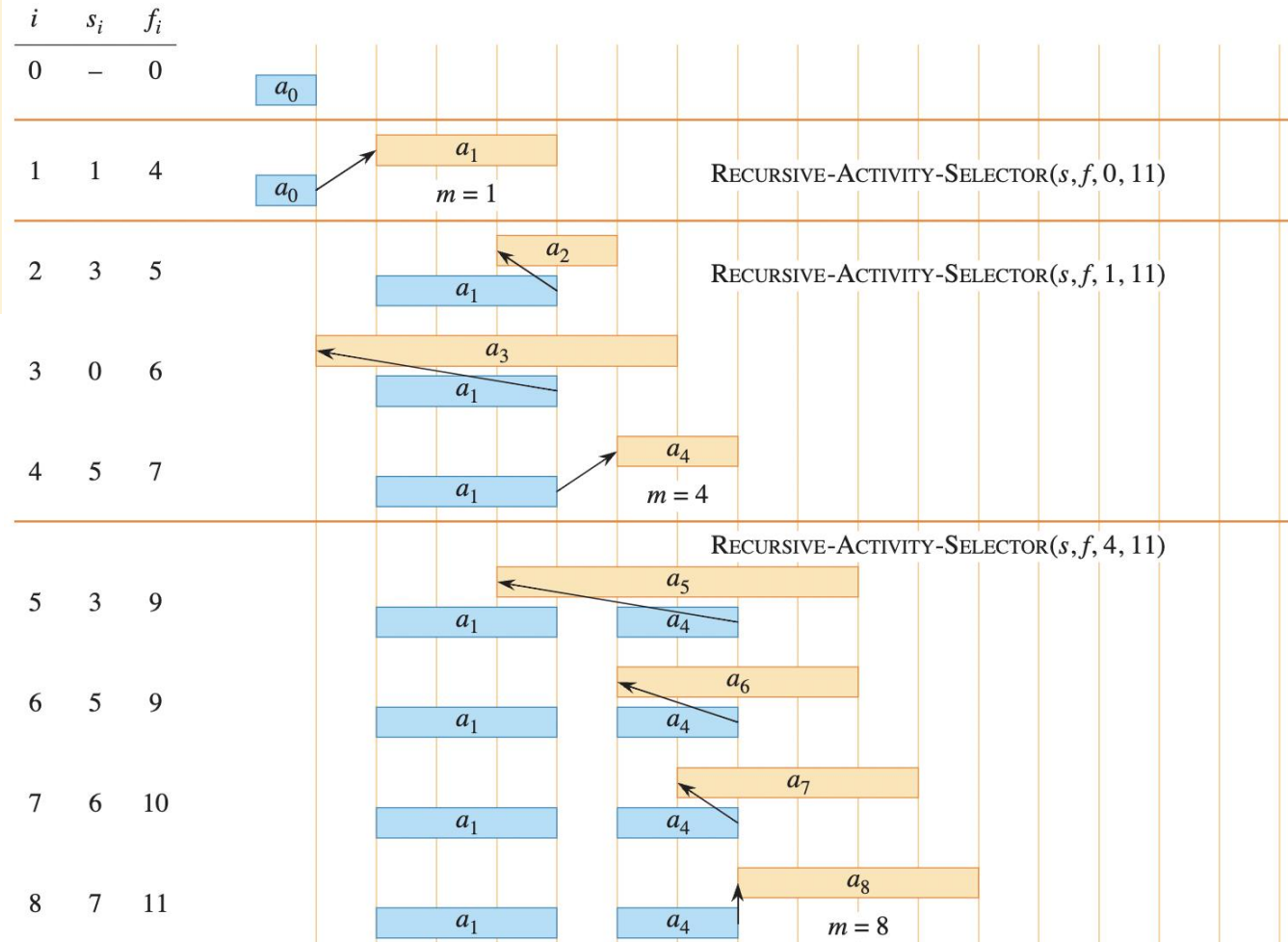
RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

```

1   $m = k + 1$ 
2  while  $m \leq n$  and  $s[m] < f[k]$     // find the first activity in  $S_k$  to finish
3       $m = m + 1$ 
4  if  $m \leq n$ 
5      return  $\{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6  else return  $\emptyset$ 
    
```

The next recursive call:

Recursive-Activity-Selector($s, f, 4, 11$)



The activity-selection Problem

RECURSIVE-ACTIVITY-SELECTOR(s, f, k, n)

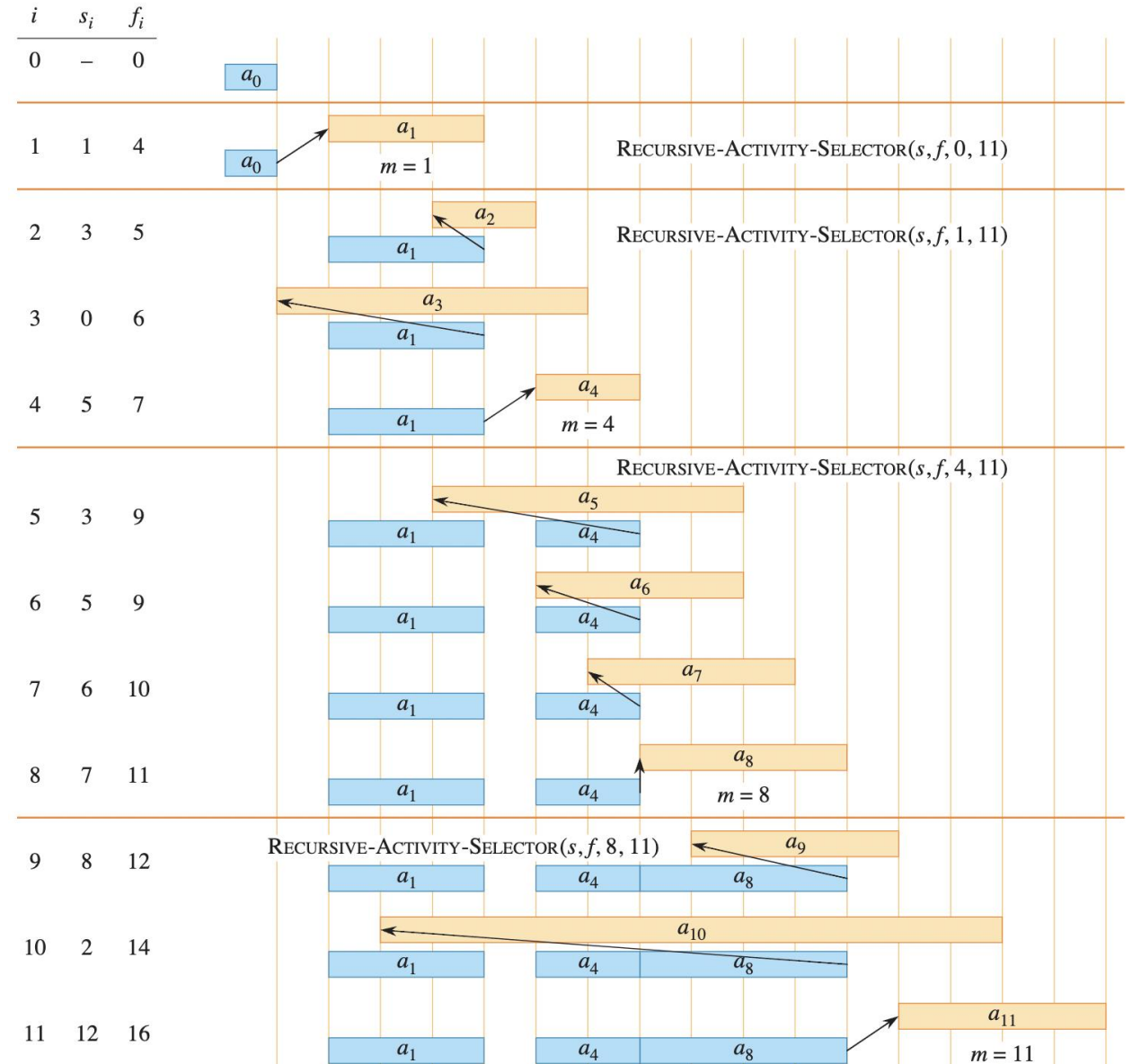
```

1   $m = k + 1$ 
2  while  $m \leq n$  and  $s[m] < f[k]$     // find the first activity in  $S_k$  to finish
3       $m = m + 1$ 
4  if  $m \leq n$ 
5      return  $\{a_m\} \cup \text{RECURSIVE-ACTIVITY-SELECTOR}(s, f, m, n)$ 
6  else return  $\emptyset$ 
    
```

The next recursive call:

Recursive-Activity-Selector ($s, f, 8, 11$)

i	0	1	2	3	4	5	6	7	8	9	10	11
s_i	-1	1	3	0	5	3	5	6	7	8	2	12
f_i	0	4	5	6	7	9	9	10	11	12	14	16



The activity-selection Problem (Iterative Greedy)

GREEDY-ACTIVITY-SELECTOR(s, f, n)

```
1   $A = \{a_1\}$ 
2   $k = 1$ 
3  for  $m = 2$  to  $n$ 
4      if  $s[m] \geq f[k]$            // is  $a_m$  in  $S_k$ ?
5           $A = A \cup \{a_m\}$      // yes, so choose it
6           $k = m$                  // and continue from there
7  return  $A$ 
```

Quiz – Q1

Determine the choice of event organizers that will enable Ryan to maximize his total revenue.

- A. [21,25]
- B. [8,14]
- C. [20,30]
- D. [24,29]
- E. [1,9]
- F. [2,6]
- G. [16,22]
- H. [13,17]



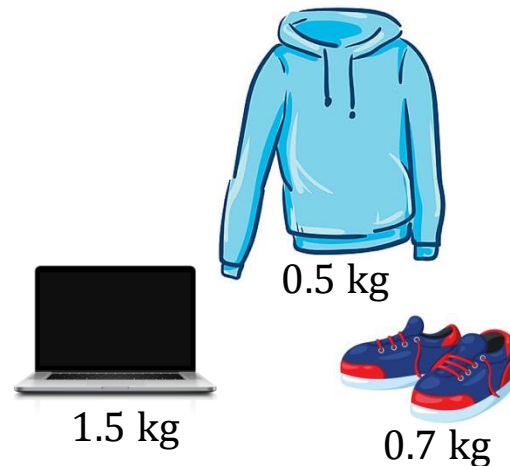
- F. [2,6]
- E. [1,9]
- B. [8,14]
- H. [13,17]
- G. [16,22]
- A. [21,25]
- D. [24,29]
- C. [20,30]

Knapsack Problem (Binary – 0/1)

Suppose we have n items $\{1, \dots, n\}$ and we want to decide which ones to take to the pawn shop.

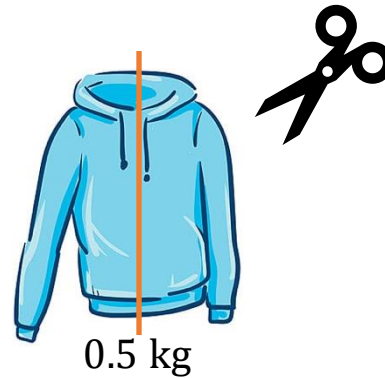
Each item i has a value v_i (which represents how much we could sell it for at the pawn shop), and a weight $w_i > 0$.

Our knapsack can only hold a total weight of W , which constrains which items we can bring.



Knapsack Problem (Fractional)

Two key variations of the Knapsack Problem: **Binary** and **Fractional**



Knapsack Problem (Greedy solution)

Question What is the greedy choice for the activity-selection problem?

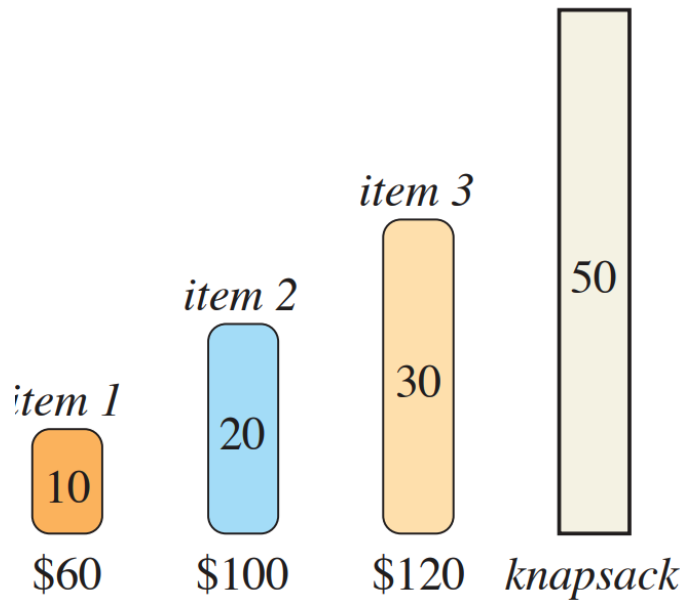
Answer Take the item with greatest value per pound ($\frac{v_i}{w_i}$)

Following a greedy strategy, the thief begins by taking as much as possible of the item with the greatest value per pound.

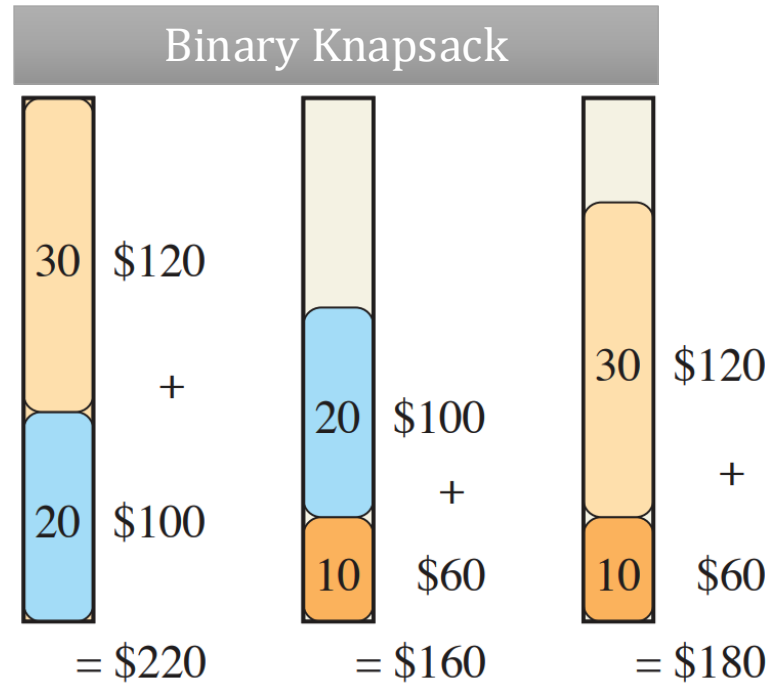
If the supply of that item is exhausted and the thief can still carry more, then the thief takes as much as possible of the item with the next greatest value per pound, and so forth, until reaching the weight limit W .

Knapsack Problem (Greedy solution)

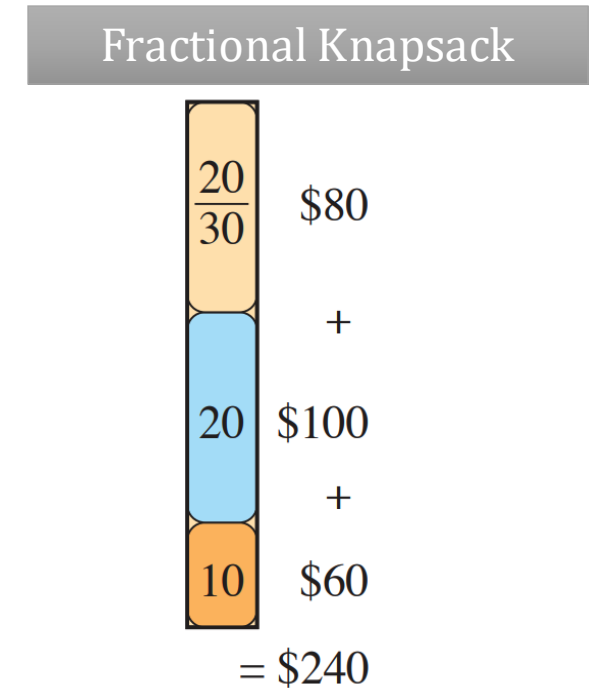
An example showing that the greedy strategy does not work for the 0-1 knapsack problem



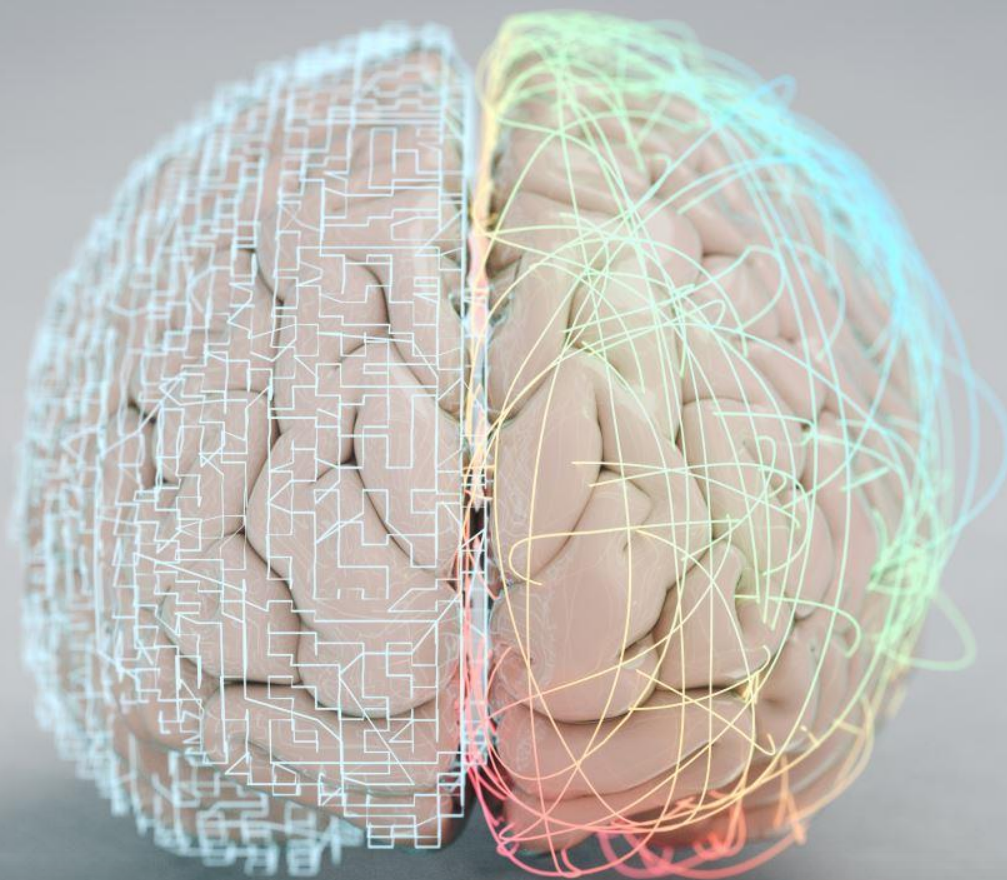
We must select a subset of the three items shown whose weight must not exceed 50 pounds.



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.



For the fractional knapsack problem, taking the items in order of greatest value per pound yields an optimal solution.



Brain Break:

5+2 minutes

Knapsack Problem (Brute Force)



```
1 ✓ def solve_knapsack(profits, weights, capacity):  
2   | return knapsack_recursive(profits, weights, capacity, 0)  
3  
4  
5 ✓ def knapsack_recursive(profits, weights, capacity, currentIndex):  
6   # base checks  
7 ✓ if capacity <= 0 or currentIndex >= len(profits):  
8   | return 0  
9  
10  # recursive call after choosing the element at the currentIndex  
11  # if the weight of the element at currentIndex exceeds the capacity, we shouldn't process this  
12  profit1 = 0  
13 ✓ if weights[currentIndex] <= capacity:  
14 ✓ | profit1 = profits[currentIndex] + knapsack_recursive(  
15 | | profits, weights, capacity - weights[currentIndex], currentIndex + 1)  
16  
17  # recursive call after excluding the element at the currentIndex  
18  profit2 = knapsack_recursive(profits, weights, capacity, currentIndex + 1)  
19  
20  return max(profit1, profit2)
```

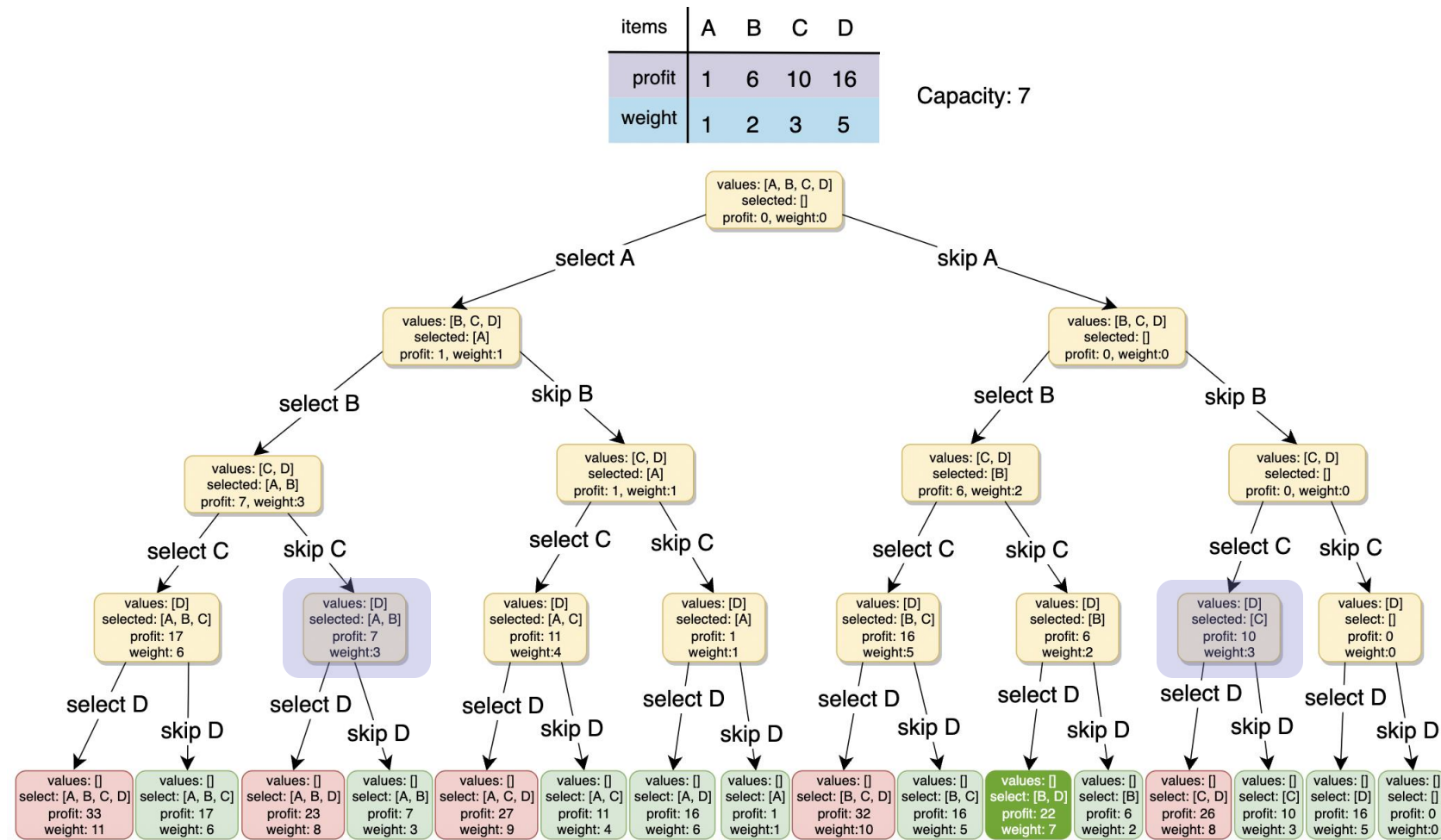


Knapsack Problem (Brute Force)

Time Complexity?

Number of recursive calls: $O(2^n)$

Identical Subproblems:



Knapsack Problem (DP - Memoization)

```
1 def solve_knapsack(profits, weights, capacity):
2     # create a two dimensional array for Memoization, each element is initialized to '-1'
3     dp = [[-1 for x in range(capacity+1)] for y in range(len(profits))]
4     return knapsack_recursive(dp, profits, weights, capacity, 0)
5
6
7 def knapsack_recursive(dp, profits, weights, capacity, currentIndex):
8
9     # base checks
10    if capacity <= 0 or currentIndex >= len(profits):
11        return 0
12
13    # if we have already solved a similar problem, return the result from memory
14    if dp[currentIndex][capacity] != -1:
15        return dp[currentIndex][capacity]
16
17    # recursive call after choosing the element at the currentIndex
18    # if the weight of the element at currentIndex exceeds the capacity, we
19    # shouldn't process this
20    profit1 = 0
21    if weights[currentIndex] <= capacity:
22        profit1 = profits[currentIndex] + knapsack_recursive(
23            dp, profits, weights, capacity - weights[currentIndex], currentIndex + 1)
24
25    # recursive call after excluding the element at the currentIndex
26    profit2 = knapsack_recursive(
27        dp, profits, weights, capacity, currentIndex + 1)
28
29    dp[currentIndex][capacity] = max(profit1, profit2)
30    return dp[currentIndex][capacity]
```

Knapsack Problem (DP with Memoization)

- What is the **time** & **space** complexity of the above solution?

Time complexity:

$O(N * C)$.

Space complexity:

$O(N * C + N)$, which is asymptotically equivalent to $O(N * C)$.

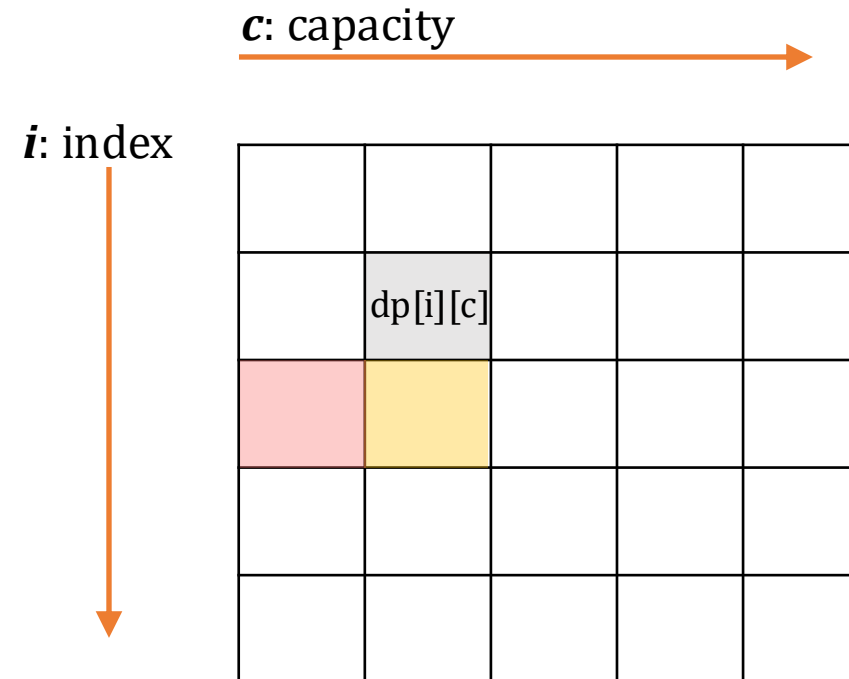
Knapsack Problem

Let's form a DP – Bottom-Up solution!

How to eliminate the need for recursive calls?
i.e., How to reorder the sub-problems?

```
21 if weights[currentIndex] <= capacity:
22     profit1 = profits[currentIndex] + knapsack_recursive(
23         dp, profits, weights, capacity - weights[currentIndex], currentIndex + 1)
24
25 # recursive call after excluding the element at the currentIndex
26 profit2 = knapsack_recursive(
27     dp, profits, weights, capacity, currentIndex + 1)
28
29 dp[currentIndex][capacity] = max(profit1, profit2)
```

dp [i] [c]

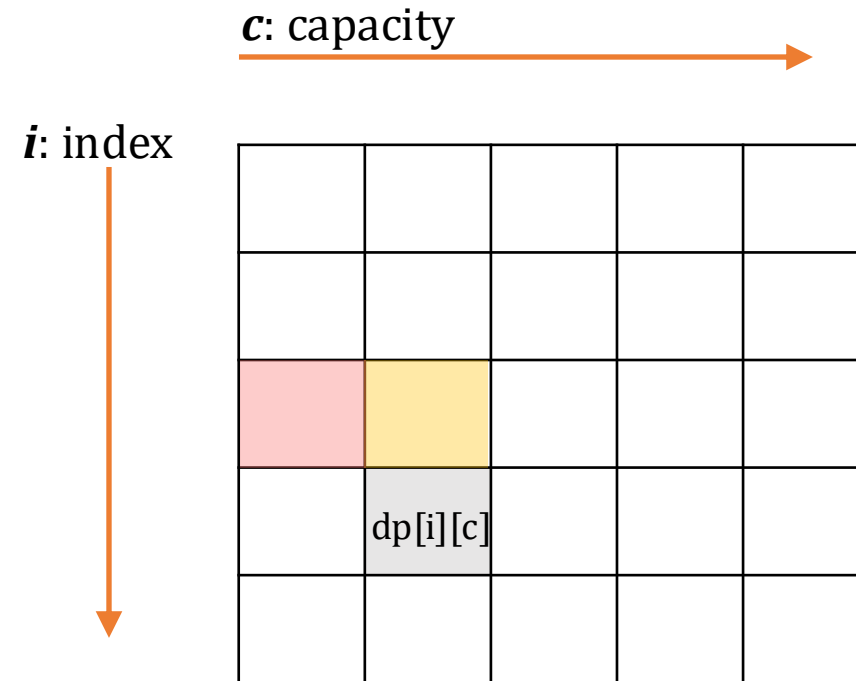


Knapsack Problem

Let's form a DP – Bottom-Up solution!

How to eliminate the need for recursive calls?
i.e., How to reorder the sub-problems?

```
profit1 = profits[i] + dp[i - 1][c - weights[i]]  
profit2 = dp[i - 1][c]
```



Knapsack Problem (DP – Bottom Up)

```
1 def solve_knapsack(profits, weights, capacity):
2     # basic checks
3     n = len(profits)
4     if capacity <= 0 or n == 0 or len(weights) != n:
5         return 0
6
7     dp = [[0 for x in range(capacity+1)] for y in range(n)]
8
9     # populate the capacity = 0 columns, with '0' capacity we have '0' profit
10    for i in range(0, n):
11        dp[i][0] = 0
12
13    # if we have only one weight, we will take it if it is not more than the capacity
14    for c in range(0, capacity+1):
15        if weights[0] <= c:
16            dp[0][c] = profits[0]
17
18    # process all sub-arrays for all the capacities
19    for i in range(1, n):
20        for c in range(1, capacity+1):
21            profit1, profit2 = 0, 0
22            # include the item, if it is not more than the capacity
23            if weights[i] <= c:
24                profit1 = profits[i] + dp[i - 1][c - weights[i]]
25            # exclude the item
26            profit2 = dp[i - 1][c]
27            # take maximum
28            dp[i][c] = max(profit1, profit2)
29
30    # maximum profit will be at the bottom-right corner.
31    return dp[n - 1][capacity]
```


Knapsack Problem (DP – Bottom Up)

Step 0

	0	1	2	3	4	5
	0	0	0	0	0	0
item 1 (v: 4 w: 2)	0	0	4	4	4	4
item 2 (v: 3 w: 1)	0	0	0	0	0	0
item 3 (v: 5 w: 3)	0	0	0	0	0	0
item 4 (v: 10 w: 5)	0	0	0	0	0	0

Knapsack Problem (DP – Bottom Up)

Step 1

	0	1	2	3	4	5
	0	0	0	0	0	0
item 1 (v: 4 w: 2)	0	0	4	4	4	4
item 2 (v: 3 w: 1)	0	3	0	0	0	0
item 3 (v: 5 w: 3)	0	0	0	0	0	0
item 4 (v: 10 w: 5)	0	0	0	0	0	0

Knapsack Problem (DP – Bottom Up)

Step 2

	0	1	2	3	4	5
	0	0	0	0	0	0
item 1 (v: 4 w: 2)	0	0 ⁺³	4	4	4	4
item 2 (v: 3 w: 1)	0	3	4	0	0	0
item 3 (v: 5 w: 3)	0	0	0	0	0	0
item 4 (v: 10 w: 5)	0	0	0	0	0	0

Knapsack Problem (DP – Bottom Up)

Step 2

	0	1	2	3	4	5
	0	0	0	0	0	0
item 1 (v: 4 w: 2)	0	0	4	4	4	4
item 2 (v: 3 w: 1)	0	3	4	7	0	0
item 3 (v: 5 w: 3)	0	0	0	0	0	0
item 4 (v: 10 w: 5)	0	0	0	0	0	0

Knapsack Problem (DP – Bottom Up)

profit []	weight []	index	capacity -->							
			0	1	2	3	4	5	6	7
1	1	0	0							
6	2	1	0							
10	3	2	0							
16	5	3	0							

With '0' capacity, maximum profit we can have for every subarray is '0'



Questions?