

# CPSC 510: Algorithms

## Unit 4: Dynamic Programming

Reading: Ch. 6

# Introduction to Dynamic Programming

- **Dynamic programming** is an algorithm design technique that is often used for optimization.
- It is used when problems where a greedy algorithm does not work.
- Dynamic programming explores the space of all possible solutions by decomposing the problem into a series of subproblems.

# Outline

- **Example: Weighted Interval Scheduling**
- Principles of Dynamic Programming
- Additional Examples

# Example: Weighted Interval Scheduling

Assume we have a single conference room and  $n$  meetings that would like to use the conference room. Each meeting  $i$  has a starting time  $s(i)$  and finishing time  $f(i)$  with  $s(i) < f(i)$ . In addition, each meeting has a value  $v(i) > 0$ . Develop an algorithm that schedules the meeting such that the value  $\sum_{i \in S} v(i)$  is maximized.

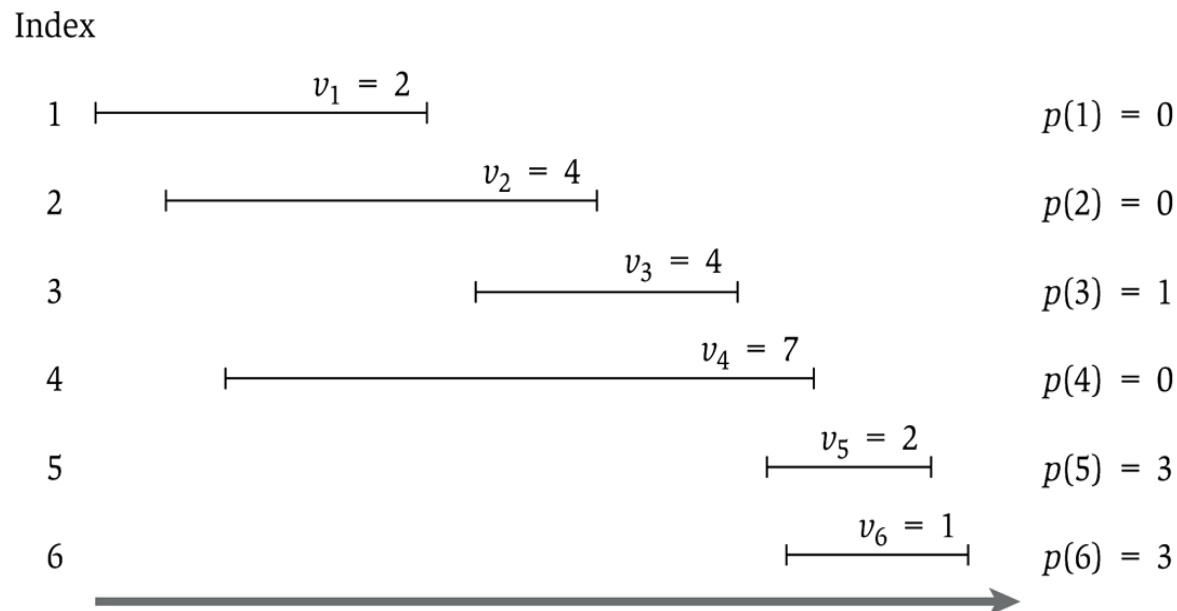
This problem is similar to the (non-weighted) interval scheduling example discussed last unit but with one key difference:

**No greedy algorithm exists for this problem!**

# Example: Weighted Interval Scheduling

Assume that the meetings are sorted by their finishing time (earliest first):  $f(1) \leq f(2) \leq \dots \leq f(n)$

Define  $p(i)$  to be the meeting number that has the latest finishing time that does not overlap with meeting  $i$  or 0 if there is no such meeting.



# Example: Weighted Interval Scheduling

- The basic idea is to start with meeting  $n$  and determine whether it should be in the solution or not. However, this is dependent on the solutions to subproblems.
- Consider two cases:
  - Meeting  $n$  is in the optimal solution.
  - Meeting  $n$  is not in the optimal solution.

# Example: Weighted Interval Scheduling

Let  $\text{OPT}(n)$  be the value of the optimal solution.

- Case 1: Meeting  $n$  is in the solution.

$$\text{OPT}(n) = v(n) + \text{OPT}(p(n))$$

- Case 2: Meeting  $n$  is not in the solution.

$$\text{OPT}(n) = \text{OPT}(n-1)$$

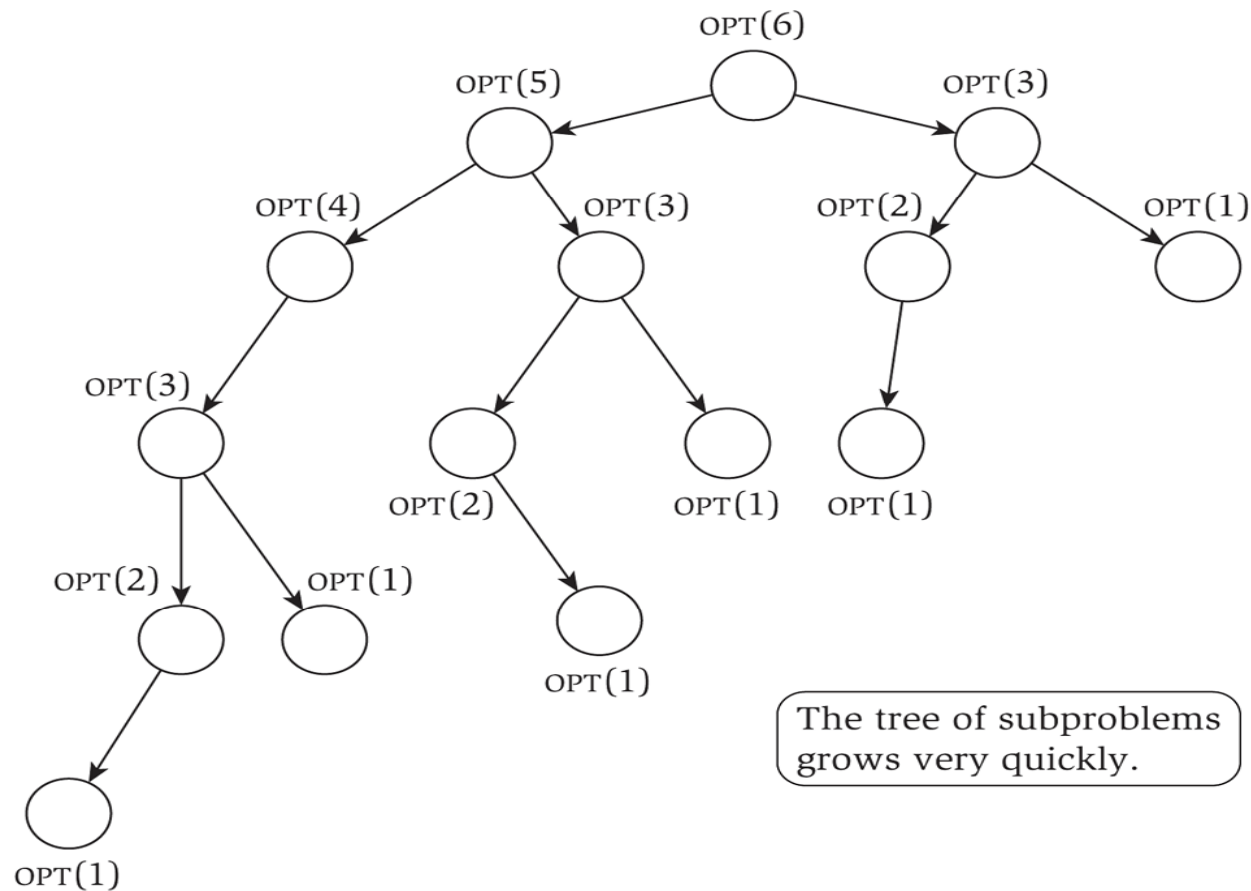
# Example: Recurrence

$n$  is only in the solution if  $v(n) + \text{OPT}(p(n)) \geq \text{OPT}(n-1)$

$$\text{OPT}(i) = \begin{cases} \max(v(i) + \text{OPT}(p(i)), \text{OPT}(i-1)) & i > 0 \\ 0 & i = 0 \end{cases}$$



# Example: Recursive Trace of Function Calls



# Memoization

- The recursive algorithm is exponential, not polynomial. However, many of the calls are redundant.
- **memoization:** Saving the value from a function call for future use.
- Memoization can be implemented for the weighted interval scheduling problem as follows:
  - Create an array  $M$  of size  $n$ .<sup>\*</sup>
  - All slots in the array are initialized to “empty”.
  - Once  $\text{COMPUTE-OPT}(i)$  has been computed, the result is stored in  $M[i]$ .
  - Before calling  $\text{COMPUTE-OPT}(i)$ , check  $M[i]$ :
    - If empty, call the function as normal.
    - If not empty, use the value  $M[i]$  as the answer and continue normally.

<sup>\*</sup> May want to add additional entries for 0, -1, ... if appropriate.

# Example: Algorithm with Memoization

COMPUTE-OPT( $i$ ):

```
1  if  $i == 0$ 
2    return 0
3  else if  $M[i]$  is not empty
4    return  $M[i]$ 
5  else
6     $M[i] = \max(v(i) + \text{COMPUTE-OPT}(p(i)), \text{COMPUTE-OPT}(i - 1))$ 
7    return  $M[i]$ 
```

# Example: Running Time

What is the running time of the memoized version of COMPUTE-OPT( $n$ )?

— Running time is  $O(n)$ . Assumes that  $p(i)$  is created, it also assumes the meetings are sorted.

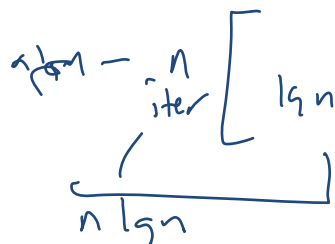
— If not the case, it takes  $O(n \lg n)$  to sort and to compute  $p(i)$ :

$n \lg n$  1. Sort the finishing times

$n \lg n$  2. Sort the starting times

3. For each meeting  $i$ :

4. Use binary search in the starting time array to compute  $p(i)$ .



# Example: Bottom-Up Algorithm

Another way to accomplish this task is in a bottom-up fashion where we iteratively determine the solution for meetings from 1 to  $n$ :

COMPUTE\_OPT( $n$ ):

- 1 let  $M$  be an array with  $n+1$  elements indexed from 0 to  $n$ .
- 2  $M[0] = 0$
- 3 for  $i = 1$  to  $n$
- 4      $M[i] = \max(v[i] + M[p[i]], M[i-1])$
- 5 return  $M[n]$

Running time is  $O(n)$  under the same assumptions.

Faster in practice - no function calls.

# Example: Computing the Optimal Solution

This algorithm only computes the maximum value. The problem asks us to return the set of meetings that make up the solution. Once we have filled in the array  $M$  (using either of the two previous algorithms), we can use this recursive algorithm:

WEIGHTED-SCHEDULE( $i$ ):

```
1  if  $i = 0$ 
2    return  $\emptyset$ 
3  else if  $v(i) + M[p(i)] \geq M[i - 1]$ 
4    return  $\{i\} \cup \text{WEIGHTED-SCHEDULE}(p(i))$ 
5  else
6    return  $\text{WEIGHTED-SCHEDULE}(i - 1)$ 
```

# Outline

- Example: Weighted Interval Scheduling
- **Principles of Dynamic Programming**
- Additional Examples

# Dynamic Programming vs. Greedy Algorithms

Dynamic programming is similar to greedy algorithms in several ways:

- Applies to optimization problems
- Uses recursion.
- Exhibits optimal substructure: An optimal solution to the problem contains within it optimal solutions to subproblems.

# Dynamic Programming vs. Greedy Algorithms

- Greedy algorithms do not exist for many problems that can be solved using dynamic programming.
- Greedy algorithms also exhibit a greedy choice property.
  - Greedy algorithms: Simply pick the greedy choice
  - Dynamic programming: Consider two or more alternatives.
- Greedy algorithms are faster in practice.
  - A greedy algorithm does not always have a better asymptotic running time when compared to a dynamic programming algorithm.

# Dynamic Programming Guidelines

When to use dynamic programming? Here are a few guidelines...

- There are only a polynomial number of subproblems.
- The solutions to the original problem can easily be computed using solutions from the subproblems.
- There is a natural ordering of subproblems with an easy-to-compute recurrence.

# Dynamic Programming vs. Divide and Conquer

Divide and conquer uses recursion. Isn't it the same technique as dynamic programming? No.

- Divide and conquer divides the problem into one or more subproblems. Except for the stopping case, you always divide. In dynamic programming, you consider several options and don't always divide.
- In dynamic programming, there are a polynomial number of subproblems.
  - Subproblems of the same size will always return the same answer.
  - The exact same identical subproblems are encountered again and again in the recursion tree (**overlapping subproblem property**).
- In divide and conquer, there are too many subproblems to consider (often exponential or infinite).
  - Consider merge sort: each subproblem that sorts a list of size 10 is likely sorting a different list.
  - However, only a few of these subproblems are actually necessary.

# Overlapping Subproblems

The overlapping subproblems property is important:

- Allows answers to subproblems to be memoized.
  - With a polynomial number of subproblems, a dynamic programming algorithm will have polynomial running-time.
  - Failing to memoize answer will often result in an exponential time algorithm (like the first weighted interval scheduling ~~problem~~ algorithm).
- If a problem fails to have this property, dynamic programming cannot be used.
  - It may mean the ~~program~~ problem is NP-complete where an exponential brute-force search is required.

# Developing Dynamic Programming Algorithms

When developing a dynamic programming algorithm:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Appropriately define the stopping cases for “trivial” subproblem sizes.
4. Compute the value of an optimal solution using memoization.
  - Could be top-down using recursive calls or bottom-up using iteration.
5. Construct an optimal solution from computed information.

# Outline

- Example: Weighted Interval Scheduling
- Principles of Dynamic Programming
- **Additional Examples**

# Example 1: Coin Changing

Create a dynamic programming algorithm for the change giving problem when the coin denominations are \$1, 50¢, 30¢, 15¢, 5¢, 1¢.

Input:  $x = 62$

Output:  $\{30, 30, 1, 1\}$

$$\text{OPT}(i) = \begin{cases} 1 + \min(\text{OPT}(i-1), \text{OPT}(i-5), \text{OPT}(i-15), \text{OPT}(i-30), \text{OPT}(i-50), \text{OPT}(i-100)) & i \geq 1 \\ 0 & i = 0 \\ \infty & i < 0 \end{cases}$$

# Example 1: Computing Optimal Value

OPT-COIN-VALUE( $x$ ):

```
1  let M be an array of  $x + 100$  elements indexed from -99 to  $x$ .
2  for  $i = -99$  to -1
3       $M[i] = \infty$ 
4   $M[0] = 0$ 
5  for  $i = 1$  to  $x$ 
6       $M[i] = 1 + \min(M[i - 1], M[i - 5], M[i - 15], M[i - 30], M[i - 50], M[i - 100])$ 
7  return  $M[x]$ 
```

# Example 1: Computing Optimal Set

OPT-COIN-SET( $M, x$ ):

1 let  $A$  be an empty list

2 while  $x \neq 0$

3 if  $M[x - 100] < M[x]$  *equiv.  $M[x - 100] + 1 = M[x]$*

4     INSERT(100,  $A$ )

5      $x = x - 100$

6 else if  $M[x - 50] < M[x]$

7     INSERT(50,  $A$ )

8      $x = x - 50$

9 else if  $M[x - 30] < M[x]$

10     INSERT(30,  $A$ )

11      $x = x - 30$

12 else if  $M[x - 15] < M[x]$

13     INSERT(15,  $A$ )

14      $x = x - 15$

15 else if  $M[x - 5] < M[x]$

16     INSERT(5,  $A$ )

17      $x = x - 5$

18 else

19     INSERT(1,  $A$ )

20      $x = x - 1$

21 return  $A$

## Example 2: Knapsack Problem

Assume you are a thief robbing a store. You have a knapsack to store items you are stealing. There are  $n$  items in the store and each item  $i$  in the store has a value  $v_i$  (in dollars) and a weight  $w_i$  (in pounds). You can only carry at most  $W$  pounds in your knapsack. Which items should you put in your knapsack such that total value is maximized and the weight limit is not exceeded? Assume the weight of each item is an integer greater than zero.

# Example 2: Computing Optimal Value

KNAPSACK-VALUE( $n, W$ ):

```
1  let M be a two-dimensional array with dimensions [0...n][0...W]
2  for  $j = 0$  to  $n$ 
3     $M[j][0] = 0$ 
4  for  $w = 1$  to  $W$ 
5     $M[0][w] = 0$ 
6  for  $j = 1$  to  $n$ 
7    for  $w = 1$  to  $W$ 
8      if  $w_j > w$ 
9         $M[j][w] = M[j - 1][w]$ 
10     else
11        $M[j][w] = \max(M[j - 1][w], v_j + M[j - 1][w - w_j])$ 
12 return  $M[n][W]$ 
```

$$O(nW)$$

# Example 2: Computing Optimal Set

KNAPSACK-SET( $M, n, W$ ):

```
1  let  $K$  be an empty set  $M$ 
2  let  $j = n$ 
3  let  $w = W$ 
4  while  $j > 0$  and  $w > 0$ 
5      if  $M[j][w] > M[j - 1][w]$ 
6          add  $j$  to  $K$ 
7           $w = w - w_j$ 
8       $j = j - 1$ 
9  return  $K$ 
```