

# CSE 221: Algorithms

## Dynamic Programming

Mumit Khan  
Fatema Tuz Zohora

Computer Science and Engineering  
BRAC University

### References

- 1 Jon Kleinberg and Éva Tardos, *Algorithm Design*. Pearson Education, 2006.
- 2 T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.

Last modified: November 27, 2012



This work is licensed under the *Creative Commons Attribution-Noncommercial-Share Alike 3.0 Unported License*.

# Contents

## 1 Introduction

- Memoization
- Dynamic programming
- Weighted interval scheduling problem
- 0/1 Knapsack problem
- Coin changing problem
- What problems can be solved by DP?
- Conclusion

# Dynamic Programming (DP)

- Build up the solution by computing solutions to the subproblems.

# Dynamic Programming (DP)

- Build up the solution by computing solutions to the subproblems.
- Don't solve the same subproblem twice, but rather save the solution so it can be re-used later on.

# Dynamic Programming (DP)

- Build up the solution by computing solutions to the subproblems.
- Don't solve the same subproblem twice, but rather save the solution so it can be re-used later on.
- Often used for a large class to optimization problems.

# Dynamic Programming (DP)

- Build up the solution by computing solutions to the subproblems.
- Don't solve the same subproblem twice, but rather save the solution so it can be re-used later on.
- Often used for a large class to optimization problems.
- Unlike Greedy algorithms, implicitly solve all subproblems.

# Dynamic Programming (DP)

- Build up the solution by computing solutions to the subproblems.
- Don't solve the same subproblem twice, but rather save the solution so it can be re-used later on.
- Often used for a large class to optimization problems.
- Unlike Greedy algorithms, implicitly solve all subproblems.
- Motivating the case for DP with Memoization – a top-down technique, and then moving on to Dynamic Programming – a bottom-up technique.

# Dynamic Programming (DP)

- Build up the solution by computing solutions to the subproblems.
- Don't solve the same subproblem twice, but rather save the solution so it can be re-used later on.
- Often used for a large class to optimization problems.
- Unlike Greedy algorithms, implicitly solve all subproblems.
- Motivating the case for DP with Memoization – a top-down technique, and then moving on to Dynamic Programming – a bottom-up technique.

▷ *Greedy is evil, Dynamic Programming is good.* – Prof. Jeff Erickson, University of Illinois, Urbana-Champaign.



# Contents

## 1 Introduction

- Memoization
- Dynamic programming
- Weighted interval scheduling problem
- 0/1 Knapsack problem
- Coin changing problem
- What problems can be solved by DP?
- Conclusion

# Recursive solution to Fibonacci numbers

## Definition (Fibonacci numbers)

The Fibonacci numbers are given by the following sequence:

$$\langle 0, 1, 1, 2, 3, 5, 8, 21, 34, 55, 89, \dots \rangle$$

# Recursive solution to Fibonacci numbers

## Definition (Fibonacci numbers)

The Fibonacci numbers are given by the following sequence:

$$\langle 0, 1, 1, 2, 3, 5, 8, 21, 34, 55, 89, \dots \rangle$$

and described by the following recurrence.

$$FIB(n) = \begin{cases} n & \text{if } n = 0 \text{ or } 1 \\ FIB(n-1) + FIB(n-2) & \text{if } n \geq 2 \end{cases}$$

# Recursive solution to Fibonacci numbers

## Definition (Fibonacci numbers)

The Fibonacci numbers are given by the following sequence:

$$\langle 0, 1, 1, 2, 3, 5, 8, 21, 34, 55, 89, \dots \rangle$$

and described by the following recurrence.

$$\text{FIB}(n) = \begin{cases} n & \text{if } n = 0 \text{ or } 1 \\ \text{FIB}(n-1) + \text{FIB}(n-2) & \text{if } n \geq 2 \end{cases}$$

## Straightforward recursive algorithm

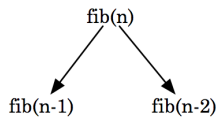
**FIBONACCI**( $n$ )  $\triangleright n \geq 0$

```
1  if  $n = 0$  or  $n = 1$ 
2    then return  $n$ 
3    else return FIBONACCI( $n - 1$ ) + FIBONACCI( $n - 2$ )
```

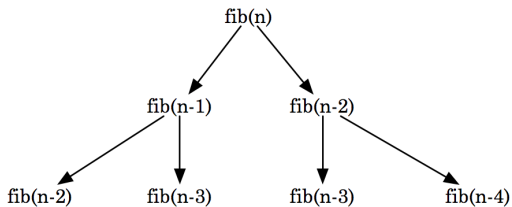
# Recursion tree

$\text{fib}(n)$

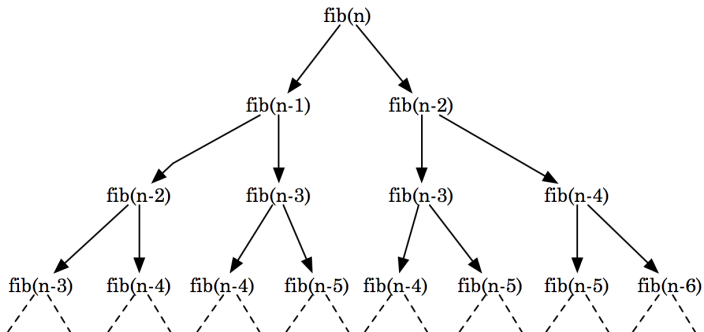
# Recursion tree



# Recursion tree

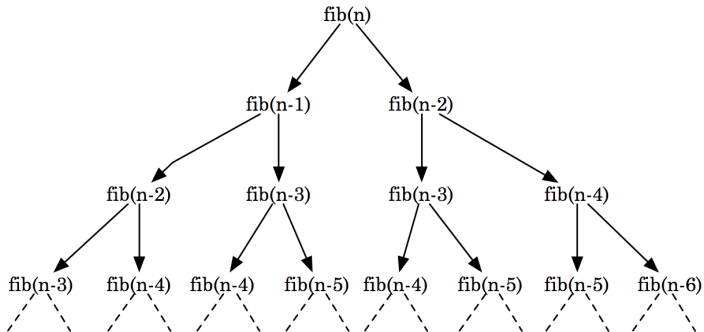


# Recursion tree





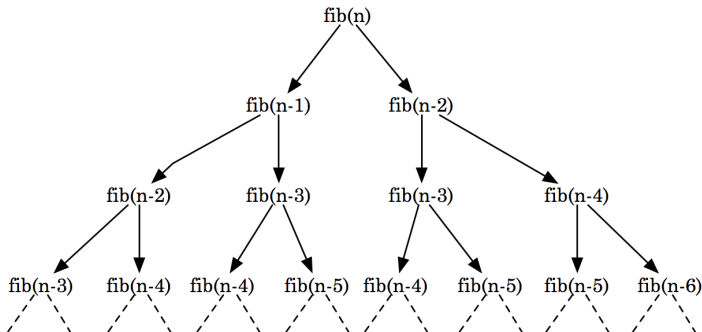
# Recursion tree



## Complexity

This recursive algorithm for Fibonacci numbers has **exponential** running time!

# Recursion tree

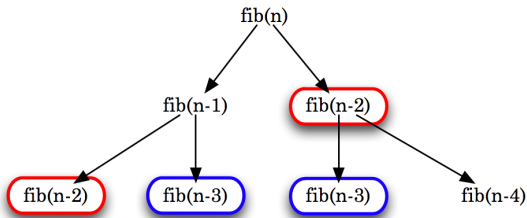


## Complexity

This recursive algorithm for Fibonacci numbers has **exponential** running time!

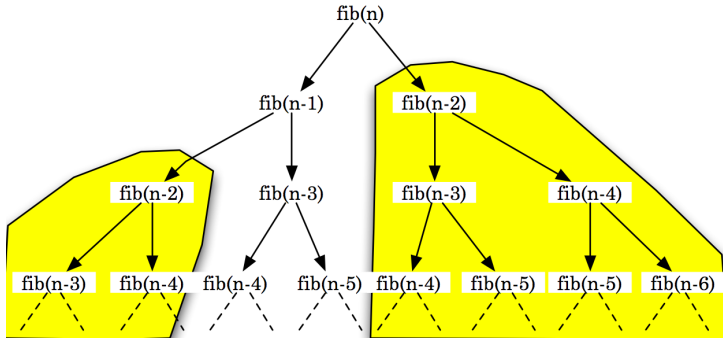
To be precise,  $T(n) = O(\varphi^n)$ , where  $\varphi = \frac{1+\sqrt{5}}{2}$  is the **golden ratio**.

# Redundant computations



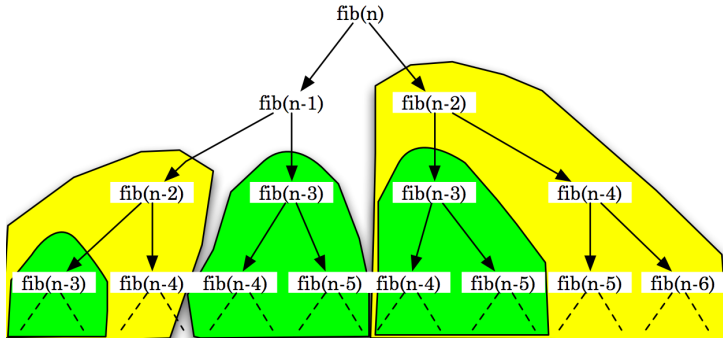
▷ Note how  $\text{FIB}(n-2)$  and  $\text{FIB}(n-3)$  are each being computed twice.

# Redundant computations



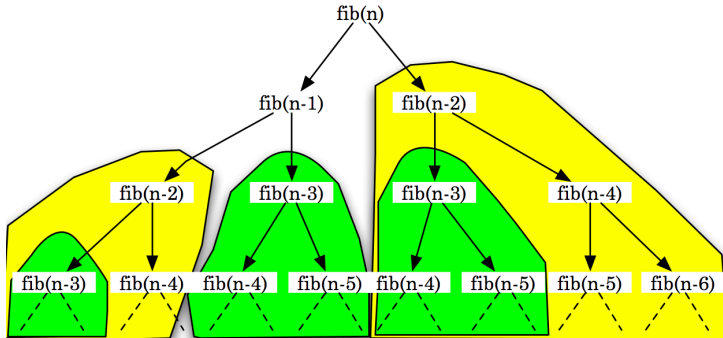
▷ In fact, computing  $\text{FIB}(n - 2)$  involves computing a whole subtree.

# Redundant computations



▷ Likewise for computing  $\text{FIB}(n - 3)$ .

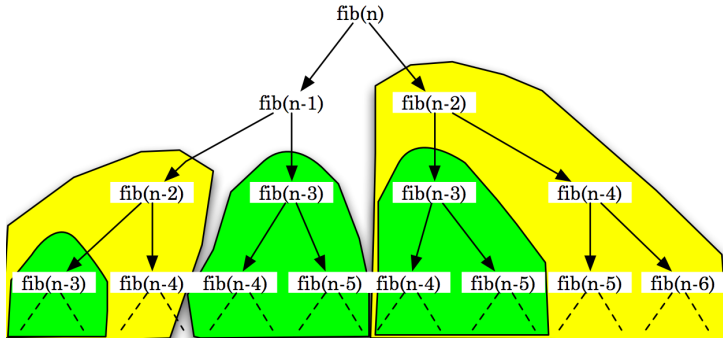
# Redundant computations



## Observations

- Spectacular redundancy in computation

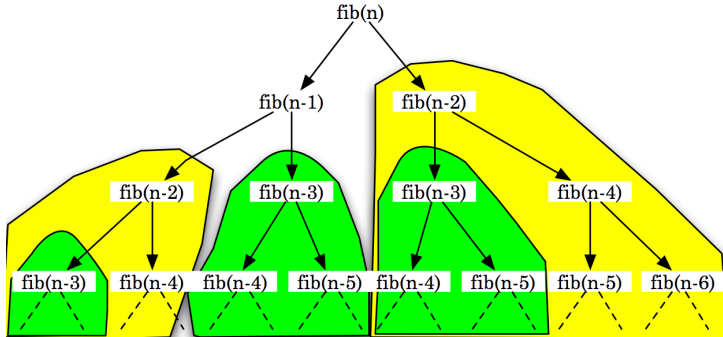
# Redundant computations



## Observations

- Spectacular redundancy in computation – how many times are we computing  $\text{FIB}(n - 2)$ ?

# Redundant computations

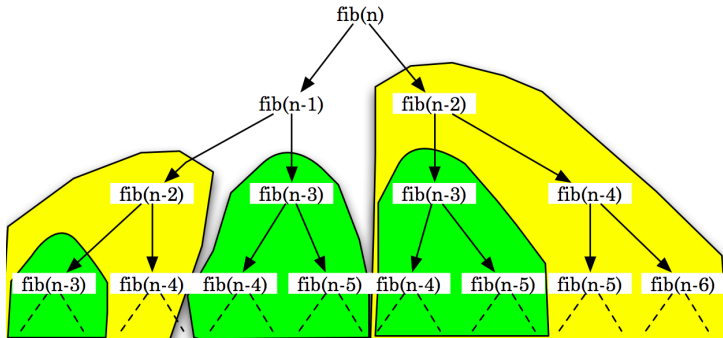


## Observations

- Spectacular redundancy in computation – how many times are we computing  $\text{FIB}(n-2)$ ?  $\text{FIB}(n-3)$ ?



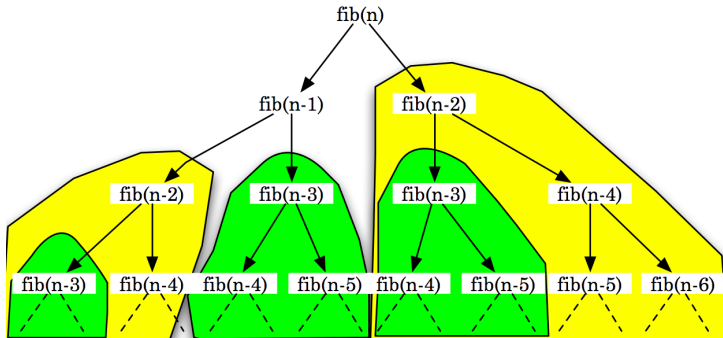
# Redundant computations



## Observations

- Spectacular redundancy in computation – how many times are we computing  $\text{FIB}(n-2)$ ?  $\text{FIB}(n-3)$ ?
- What if we compute and save the result of  $\text{FIB}(i)$  for  $i = \{2, 3, \dots, n\}$  the first time, and then re-use it each time afterward?

# Redundant computations



## Observations

- Spectacular redundancy in computation – how many times are we computing  $\text{FIB}(n-2)$ ?  $\text{FIB}(n-3)$ ?
- What if we compute and save the result of  $\text{FIB}(i)$  for  $i = \{2, 3, \dots, n\}$  the first time, and then re-use it each time afterward?
- Ah, we've just (re)discovered [Memo\(r\)ization](#)!

# Memoization

## Definition (Memoization)

The process of saving solutions to subproblems that can be re-used later without redundant computations.

# Memoization

## Definition (Memoization)

The process of saving solutions to subproblems that can be re-used later without redundant computations.

## Basic idea

Typically, the solutions to subproblems (i.e., the intermediate solutions) are saved in a global array, which are later looked up and re-used as needed.

# Memoization

## Definition (Memoization)

The process of saving solutions to subproblems that can be re-used later without redundant computations.

## Basic idea

Typically, the solutions to subproblems (i.e., the intermediate solutions) are saved in a global array, which are later looked up and re-used as needed.

- 1 At each step of computation, first see if the solution to the subproblem has already been found and saved.

# Memoization

## Definition (Memoization)

The process of saving solutions to subproblems that can be re-used later without redundant computations.

## Basic idea

Typically, the solutions to subproblems (i.e., the intermediate solutions) are saved in a global array, which are later looked up and re-used as needed.

- 1 At each step of computation, first see if the solution to the subproblem has already been found and saved.
- 2 If so, simply return the solution.

# Memoization

## Definition (Memoization)

The process of saving solutions to subproblems that can be re-used later without redundant computations.

## Basic idea

Typically, the solutions to subproblems (i.e., the intermediate solutions) are saved in a global array, which are later looked up and re-used as needed.

- 1 At each step of computation, first see if the solution to the subproblem has already been found and saved.
- 2 If so, simply return the solution.
- 3 If not, compute the solution, and save it before returning the solution.

# Memoized recursive algorithm for Fibonacci numbers

M-FIBONACCI( $n$ )      $\triangleright n \geq 0$ , global  $F = [0 \dots n]$

1    **if**  $n = 0$  or  $n = 1$

2        **then return**  $n$

$\triangleright$  Our base conditions.

3    **if**  $F[n]$  is empty

$\triangleright$  No saved solution found for  $n$ .

4        **then**  $F[n] \leftarrow \text{M-FIBONACCI}(n-1) + \text{M-FIBONACCI}(n-2)$

5    **return**  $F[n]$



# Memoized recursive algorithm for Fibonacci numbers

M-FIBONACCI( $n$ )     $\triangleright n \geq 0$ , global  $F = [0 \dots n]$

1    **if**  $n = 0$  or  $n = 1$

2        **then return**  $n$      $\triangleright$  Our base conditions.

3    **if**  $F[n]$  is empty     $\triangleright$  No saved solution found for  $n$ .

4        **then**  $F[n] \leftarrow \text{M-FIBONACCI}(n-1) + \text{M-FIBONACCI}(n-2)$

5    **return**  $F[n]$

## Questions

- What is this **global array**  $F$ ?

# Memoized recursive algorithm for Fibonacci numbers

M-FIBONACCI( $n$ )     $\triangleright n \geq 0$ , global  $F = [0 \dots n]$

1    **if**  $n = 0$  or  $n = 1$

2        **then return**  $n$      $\triangleright$  Our base conditions.

3    **if**  $F[n]$  is empty     $\triangleright$  No saved solution found for  $n$ .

4        **then**  $F[n] \leftarrow \text{M-FIBONACCI}(n-1) + \text{M-FIBONACCI}(n-2)$

5    **return**  $F[n]$

## Questions

- What is this **global array  $F$** ? It's used store the values of the intermediate results, and must be initialized by the caller to all empty.

# Memoized recursive algorithm for Fibonacci numbers

M-FIBONACCI( $n$ )     $\triangleright n \geq 0$ , global  $F = [0..n]$

1    **if**  $n = 0$  or  $n = 1$

2        **then return**  $n$      $\triangleright$  Our base conditions.

3    **if**  $F[n]$  is empty     $\triangleright$  No saved solution found for  $n$ .

4        **then**  $F[n] \leftarrow \text{M-FIBONACCI}(n-1) + \text{M-FIBONACCI}(n-2)$

5    **return**  $F[n]$

## Questions

- What is this **global array**  $F$ ? It's used store the values of the intermediate results, and must be initialized by the caller to all empty.
- What is an appropriate **sentinel** to indicate that  $F[i], 0 \leq i \leq n$  has not been solved yet (i.e., empty)?

# Memoized recursive algorithm for Fibonacci numbers

M-FIBONACCI( $n$ )     $\triangleright n \geq 0$ , global  $F = [0..n]$

```
1  if  $n = 0$  or  $n = 1$ 
2      then return  $n$                                  $\triangleright$  Our base conditions.
3  if  $F[n]$  is empty                                   $\triangleright$  No saved solution found for  $n$ .
4      then  $F[n] \leftarrow$  M-FIBONACCI( $n - 1$ ) + M-FIBONACCI( $n - 2$ )
5  return  $F[n]$ 
```

## Questions

- What is this **global array**  $F$ ? It's used to store the values of the intermediate results, and must be initialized by the caller to all empty.
- What is an appropriate **sentinel** to indicate that  $F[i], 0 \leq i \leq n$  has not been solved yet (i.e., empty)? Use  $-1$ , which is guaranteed to be an invalid value.

# Memoized ... Fibonacci numbers (continued)

FIBONACCI( $n$ )       $\triangleright n \geq 0$

$\triangleright$  Allocate an array  $F[0..n]$  to save results ( $\text{LENGTH}[F] = n + 1$ ).

```
1  for  $i \leftarrow 0$  to  $n$ 
2      do  $F[i] \leftarrow -1$        $\triangleright$  No solution computed for  $i$  yet (sentinel)
3  return M-FIBONACCI( $F, n$ )
```

# Memoized ... Fibonacci numbers (continued)

FIBONACCI( $n$ )       $\triangleright n \geq 0$

$\triangleright$  Allocate an array  $F[0..n]$  to save results ( $\text{LENGTH}[F] = n + 1$ ).

```
1  for  $i \leftarrow 0$  to  $n$ 
2      do  $F[i] \leftarrow -1$        $\triangleright$  No solution computed for  $i$  yet (sentinel)
3  return M-FIBONACCI( $F, n$ )
```

M-FIBONACCI( $F, n$ )       $\triangleright n \geq 0, F = [0..n]$

```
1  if  $n \leq 1$ 
2      then return  $n$ 
3  if  $F[n] = -1$        $\triangleright$  No saved solution found for  $n$ .
4      then  $F[n] \leftarrow \text{M-FIBONACCI}(F, n - 1) + \text{M-FIBONACCI}(F, n - 2)$ 
5  return  $F[n]$ 
```

# Memoized ... Fibonacci numbers (continued)

FIBONACCI( $n$ )       $\triangleright n \geq 0$

$\triangleright$  Allocate an array  $F[0..n]$  to save results ( $\text{LENGTH}[F] = n + 1$ ).

```
1  for  $i \leftarrow 0$  to  $n$ 
2      do  $F[i] \leftarrow -1$        $\triangleright$  No solution computed for  $i$  yet (sentinel)
3  return M-FIBONACCI( $F, n$ )
```

M-FIBONACCI( $F, n$ )       $\triangleright n \geq 0, F = [0..n]$

```
1  if  $n \leq 1$ 
2      then return  $n$ 
3  if  $F[n] = -1$        $\triangleright$  No saved solution found for  $n$ .
4      then  $F[n] \leftarrow$  M-FIBONACCI( $F, n - 1$ ) + M-FIBONACCI( $F, n - 2$ )
5  return  $F[n]$ 
```

## Running time

Each element  $F[2] \dots F[n]$  is filled in just once in  $\Theta(1)$  time, so

$$T(n) = \Theta(n).$$

# Memoization highlights

- Idea is to re-use saved solutions, trading off **space** for **time**.



# Memoization highlights

- Idea is to re-use saved solutions, trading off **space** for **time**.
- Any recursive algorithm can be **memoized**, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are **overlapping subproblems**).

# Memoization highlights

- Idea is to re-use saved solutions, trading off **space** for **time**.
- Any recursive algorithm can be **memoized**, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are **overlapping subproblems**).
- Any recursive algorithm where redundant solutions are computed, **Memoization** is an appropriate solution.

# Memoization highlights

- Idea is to re-use saved solutions, trading off **space** for **time**.
- Any recursive algorithm can be **memoized**, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are **overlapping subproblems**).
- Any recursive algorithm where redundant solutions are computed, **Memoization** is an appropriate solution.
- Often called **Top-down Dynamic Programming**.

# Memoization highlights

- Idea is to re-use saved solutions, trading off **space** for **time**.
- Any recursive algorithm can be **memoized**, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are **overlapping subproblems**).
- Any recursive algorithm where redundant solutions are computed, **Memoization** is an appropriate solution.
- Often called **Top-down Dynamic Programming**.

Questions to ask (and remember)

# Memoization highlights

- Idea is to re-use saved solutions, trading off **space** for **time**.
- Any recursive algorithm can be **memoized**, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are **overlapping subproblems**).
- Any recursive algorithm where redundant solutions are computed, **Memoization** is an appropriate solution.
- Often called **Top-down Dynamic Programming**.

## Questions to ask (and remember)

- What are the drawbacks, if any, of memoization?

# Memoization highlights

- Idea is to re-use saved solutions, trading off **space** for **time**.
- Any recursive algorithm can be **memoized**, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are **overlapping subproblems**).
- Any recursive algorithm where redundant solutions are computed, **Memoization** is an appropriate solution.
- Often called **Top-down Dynamic Programming**.

## Questions to ask (and remember)

- What are the drawbacks, if any, of memoization?
- Would all recursive algorithms benefit from memoization?

# Memoization highlights

- Idea is to re-use saved solutions, trading off **space** for **time**.
- Any recursive algorithm can be **memoized**, but only helps if there is redundancy in computing solutions to subproblems (in other words, if there are **overlapping subproblems**).
- Any recursive algorithm where redundant solutions are computed, **Memoization** is an appropriate solution.
- Often called **Top-down Dynamic Programming**.

## Questions to ask (and remember)

- What are the drawbacks, if any, of memoization?
- Would all recursive algorithms benefit from memoization?  
For example, would the recursive algorithm to compute the factorial of a number benefit from memoization?

# Contents

## 1 Introduction

- Memoization
- **Dynamic programming**
- Weighted interval scheduling problem
- 0/1 Knapsack problem
- Coin changing problem
- What problems can be solved by DP?
- Conclusion



# Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number  $n$  **top down** by computing (and saving) solutions for smaller values.

# Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number  $n$  **top down** by computing (and saving) solutions for smaller values.
- **Idea:** why not build up the solution bottom-up, starting from the base case(s) all the way to  $n$ ?

# Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number  $n$  **top down** by computing (and saving) solutions for smaller values.
- **Idea:** why not build up the solution bottom-up, starting from the base case(s) all the way to  $n$ ?
- This bottom up construction gives us the first **Dynamic Programming** algorithm.

# Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number  $n$  **top down** by computing (and saving) solutions for smaller values.
- **Idea**: why not build up the solution bottom-up, starting from the base case(s) all the way to  $n$ ?
- This bottom up construction gives us the first **Dynamic Programming** algorithm.

## Dynamic programming algorithm for fibonacci numbers

```
FIBONACCI( $n$ )           ▷  $n \geq 0$   
1   $F[0] \leftarrow 0$   
2   $F[1] \leftarrow 1$   
3  for  $i \leftarrow 2$  to  $n$   
4      do  $F[i] \leftarrow F[i - 1] + F[i - 2]$   
5  return  $F[n]$ 
```

# Dynamic programming

- Note how the recursive algorithm computes the Fibonacci number  $n$  **top down** by computing (and saving) solutions for smaller values.
- **Idea**: why not build up the solution bottom-up, starting from the base case(s) all the way to  $n$ ?
- This bottom up construction gives us the first **Dynamic Programming** algorithm.

## Dynamic programming algorithm for fibonacci numbers

```
FIBONACCI( $n$ )           ▷  $n \geq 0$   
1   $F[0] \leftarrow 0$   
2   $F[1] \leftarrow 1$   
3  for  $i \leftarrow 2$  to  $n$   
4      do  $F[i] \leftarrow F[i - 1] + F[i - 2]$   
5  return  $F[n]$ 
```

$$T(n) = \Theta(n)$$

# Dynamic programming (continued)

## The pattern

- 1 Formulate the problem recursively.

# Dynamic programming (continued)

## The pattern

- 1 **Formulate the problem recursively.** Write a formula for the whole problem as a simple combination of of the answers to smaller subproblems.

# Dynamic programming (continued)

## The pattern

- 1 **Formulate the problem recursively.** Write a formula for the whole problem as a simple combination of the answers to smaller subproblems.
- 2 **Build solutions to the recurrence from the bottom up.**



# Dynamic programming (continued)

## The pattern

- 1 **Formulate the problem recursively.** Write a formula for the whole problem as a simple combination of of the answers to smaller subproblems.
- 2 **Build solutions to the recurrence from the bottom up.** Write an algorithm that starts with the base case, and works its way up to the final solution by considering the subproblems in the correct order.

# Dynamic programming (continued)

## The pattern

- 1 **Formulate the problem recursively.** Write a formula for the whole problem as a simple combination of of the answers to smaller subproblems.
- 2 **Build solutions to the recurrence from the bottom up.** Write an algorithm that starts with the base case, and works its way up to the final solution by considering the subproblems in the correct order.

## Observations

- 1 Must ensure that the recurrence is correct of course!

# Dynamic programming (continued)

## The pattern

- 1 **Formulate the problem recursively.** Write a formula for the whole problem as a simple combination of the answers to smaller subproblems.
- 2 **Build solutions to the recurrence from the bottom up.** Write an algorithm that starts with the base case, and works its way up to the final solution by considering the subproblems in the correct order.

## Observations

- 1 Must ensure that the recurrence is correct of course!
- 2 Need a “place” to store the solutions to subproblems, and need to look these solutions up when needed.

# Dynamic programming (continued)

## The pattern

- 1 **Formulate the problem recursively.** Write a formula for the whole problem as a simple combination of the answers to smaller subproblems.
- 2 **Build solutions to the recurrence from the bottom up.** Write an algorithm that starts with the base case, and works its way up to the final solution by considering the subproblems in the correct order.

## Observations

- 1 Must ensure that the recurrence is correct of course!
- 2 Need a “place” to store the solutions to subproblems, and need to look these solutions up when needed. Typically, but not always, a multi-dimensional table is used as storage.

# Contents

## 1 Introduction

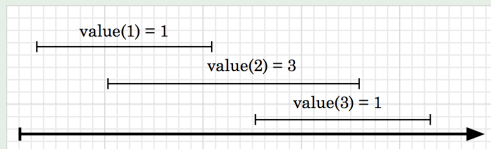
- Memoization
- Dynamic programming
- **Weighted interval scheduling problem**
- 0/1 Knapsack problem
- Coin changing problem
- What problems can be solved by DP?
- Conclusion

# Weighted interval scheduling problem

## Definition (Weighted interval scheduling problem)

Given a set of schedules  $I = \{I_i\}$ , with associated weights  $W = \{w_i\}$ , find  $A \subseteq I$  such that the members of  $A$  are **non-conflicting** and the total weight  $\sum_{i \in A} w_i$  is **maximized**.

## Example (an instance of weighted interval problem)



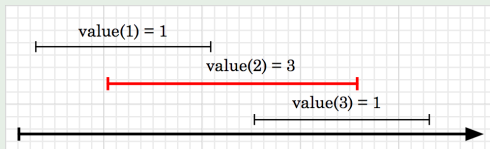
$$|A| = ???, \sum_{i \in A} w_i = ???.$$

# Weighted interval scheduling problem

## Definition (Weighted interval scheduling problem)

Given a set of schedules  $I = \{I_i\}$ , with associated weights  $W = \{w_i\}$ , find  $A \subseteq I$  such that the members of  $A$  are **non-conflicting** and the total weight  $\sum_{i \in A} w_i$  is **maximized**.

## Example (using an optimal strategy)



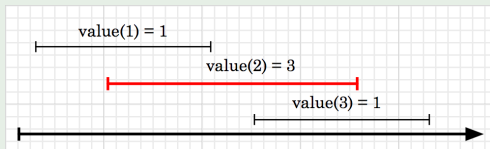
$$|A| = 1, \sum_{i \in A} w_i = 3.$$

# Weighted interval scheduling problem

## Definition (Weighted interval scheduling problem)

Given a set of schedules  $I = \{I_i\}$ , with associated weights  $W = \{w_i\}$ , find  $A \subseteq I$  such that the members of  $A$  are **non-conflicting** and the total weight  $\sum_{i \in A} w_i$  is **maximized**.

## Example (using an optimal strategy)



$$|A| = 1, \sum_{i \in A} w_i = 3.$$

## What now?

First step is to formulate a recursive solution, but first we need to figure out what the subproblems are.



# Developing a recursive solution

- Let  $W$  be an instance of a weighted interval problem.

# Developing a recursive solution

- Let  $W$  be an instance of a weighted interval problem.
- As in the greedy approach, we sort the intervals according to finish times such that  $f_i \leq f_j$  for  $i < j$  (“a natural order of the subproblems”).

# Developing a recursive solution

- Let  $W$  be an instance of a weighted interval problem.
- As in the greedy approach, we sort the intervals according to finish times such that  $f_i \leq f_j$  for  $i < j$  (“a natural order of the subproblems”).
- Let  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).

# Developing a recursive solution

- Let  $W$  be an instance of a weighted interval problem.
- As in the greedy approach, we sort the intervals according to finish times such that  $f_i \leq f_j$  for  $i < j$  (“a natural order of the subproblems”).
- Let  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).
- All we can say about  $\vartheta$  is the following: **interval  $n$  (the last interval) either belongs to  $\vartheta$ , or it doesn't.**

# Developing a recursive solution

- Let  $W$  be an instance of a weighted interval problem.
- As in the greedy approach, we sort the intervals according to finish times such that  $f_i \leq f_j$  for  $i < j$  (“a natural order of the subproblems”).
- Let  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).
- All we can say about  $\vartheta$  is the following: **interval  $n$  (the last interval) either belongs to  $\vartheta$ , or it doesn't.**

If  $n \in \vartheta$  Then clearly all intervals that conflict with  $n$  are not members of  $\vartheta$ .  $\vartheta$  then contains  $n$ , plus an optimal solution to all intervals that do not conflict with  $n$ . We now need to have a quick way of computing list of conflicting intervals for  $n$ .

# Developing a recursive solution

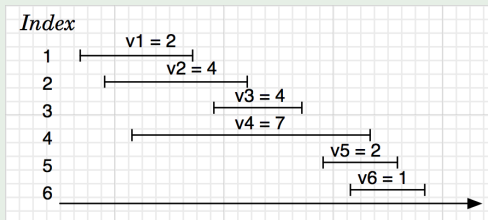
- Let  $W$  be an instance of a weighted interval problem.
- As in the greedy approach, we sort the intervals according to finish times such that  $f_i \leq f_j$  for  $i < j$  (“a natural order of the subproblems”).
- Let  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).
- All we can say about  $\vartheta$  is the following: **interval  $n$  (the last interval) either belongs to  $\vartheta$ , or it doesn't.**

**If  $n \in \vartheta$**  Then clearly all intervals that conflict with  $n$  are not members of  $\vartheta$ .  $\vartheta$  then contains  $n$ , plus an optimal solution to all intervals that do not conflict with  $n$ . We now need to have a quick way of computing list of conflicting intervals for  $n$ .

**If  $n \notin \vartheta$**  Then  $\vartheta$  contains an optimal solution for the intervals  $\{i_1, i_2, \dots, i_{n-1}\}$ .

# Developing a recursive solution (continued)

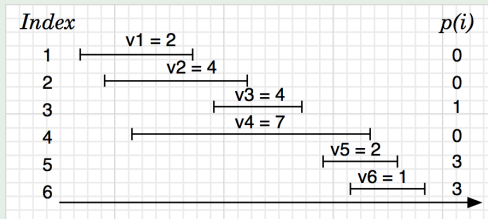
## Example (an instance of a weighted interval problem)



► For each interval  $i$ , compute  $p(i)$ , the rightmost interval among the non-conflicting preceding intervals of  $i$ . Define  $p(j) = 0$  if no request  $i < j$  is disjoint from  $j$ .

# Developing a recursive solution (continued)

## Example (an instance of a weighted interval problem)

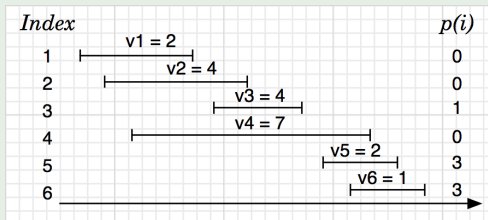


► For a given interval  $i$ ,  $p(i)$  means that intervals  $\{p(i) + 1, p(i) + 2, \dots, i - 1\}$  overlap with it. For example,  $p(6) = 3$ , which means that intervals  $\{4, 5\}$  overlap interval 6.



# Developing a recursive solution (continued)

## Example (an instance of a weighted interval problem)



► Alternatively, intervals  $\{1, 2, \dots, p(i)\}$  *do not* overlap interval  $i$ . For example,  $p(6) = 3$  means that intervals  $\{1, 2, 3\}$  do not overlap interval 6.

# Developing a recursive solution (continued)

- If  $n \in \vartheta$ , then  $\vartheta$  must include, in addition to interval  $n$ , an optimal solution to the subproblem consisting of intervals  $\{1, 2, \dots, p(n)\}$ .

# Developing a recursive solution (continued)

- If  $n \in \vartheta$ , then  $\vartheta$  must include, in addition to interval  $n$ , an optimal solution to the subproblem consisting of intervals  $\{1, 2, \dots, p(n)\}$ . If  $\vartheta(n)$  is an optimal solution to the subproblem for intervals  $\{1, 2, \dots, n\}$ , then:  
▷  $\vartheta(n) = w_n + \vartheta(p(n))$

# Developing a recursive solution (continued)

- If  $n \in \vartheta$ , then  $\vartheta$  must include, in addition to interval  $n$ , an optimal solution to the subproblem consisting of intervals  $\{1, 2, \dots, p(n)\}$ . If  $\vartheta(n)$  is an optimal solution to the subproblem for intervals  $\{1, 2, \dots, n\}$ , then:  
▷  $\vartheta(n) = w_n + \vartheta(p(n))$
- If  $n \notin \vartheta$ , then  $\vartheta$  simply contains an optimal solution to the subproblem consisting of the intervals  $\{1, 2, \dots, n-1\}$ .

# Developing a recursive solution (continued)

- If  $n \in \vartheta$ , then  $\vartheta$  must include, in addition to interval  $n$ , an optimal solution to the subproblem consisting of intervals  $\{1, 2, \dots, p(n)\}$ . If  $\vartheta(n)$  is an optimal solution to the subproblem for intervals  $\{1, 2, \dots, n\}$ , then:
  - ▷  $\vartheta(n) = w_n + \vartheta(p(n))$
- If  $n \notin \vartheta$ , then  $\vartheta$  simply contains an optimal solution to the subproblem consisting of the intervals  $\{1, 2, \dots, n-1\}$ .
  - ▷  $\vartheta(n) = \vartheta(n-1)$

# Developing a recursive solution (continued)

- If  $n \in \vartheta$ , then  $\vartheta$  must include, in addition to interval  $n$ , an optimal solution to the subproblem consisting of intervals  $\{1, 2, \dots, p(n)\}$ . If  $\vartheta(n)$  is an optimal solution to the subproblem for intervals  $\{1, 2, \dots, n\}$ , then:  
▷  $\vartheta(n) = w_n + \vartheta(p(n))$
- If  $n \notin \vartheta$ , then  $\vartheta$  simply contains an optimal solution to the subproblem consisting of the intervals  $\{1, 2, \dots, n-1\}$ .  
▷  $\vartheta(n) = \vartheta(n-1)$
- Since an optimal solution must maximize the sum of the weights in the intervals it contains, we accept the larger of the two.

# Developing a recursive solution (continued)

- If  $n \in \vartheta$ , then  $\vartheta$  must include, in addition to interval  $n$ , an optimal solution to the subproblem consisting of intervals  $\{1, 2, \dots, p(n)\}$ . If  $\vartheta(n)$  is an optimal solution to the subproblem for intervals  $\{1, 2, \dots, n\}$ , then:
  - ▷  $\vartheta(n) = w_n + \vartheta(p(n))$
- If  $n \notin \vartheta$ , then  $\vartheta$  simply contains an optimal solution to the subproblem consisting of the intervals  $\{1, 2, \dots, n-1\}$ .
  - ▷  $\vartheta(n) = \vartheta(n-1)$
- Since an optimal solution must maximize the sum of the weights in the intervals it contains, we accept the larger of the two.
  - ▷  $\vartheta(n) = \text{MAX}(w_n + \vartheta(p(n)), \vartheta(n-1))$

# Developing a recursive solution (continued)

## Recursive algorithm for an optimal value

If  $OPT(j)$  is an optimal solution to the subproblem for intervals  $\{1, 2, \dots, j\}$ , for any  $j \in \{1, 2, \dots, n\}$ , then:

$$OPT(j) = \text{MAX}(w_j + OPT(p(j)), OPT(j - 1))$$



# Developing a recursive solution (continued)

## Recursive algorithm for an optimal value

If  $OPT(j)$  is an optimal solution to the subproblem for intervals  $\{1, 2, \dots, j\}$ , for any  $j \in \{1, 2, \dots, n\}$ , then:

$$OPT(j) = \text{MAX}(w_j + OPT(p(j)), OPT(j - 1))$$

## Extracting the intervals in an optimal solution

The interval  $j$  is in an optimal solution  $OPT(j)$  **if and only if** the first of the two options is larger than the second.

# Developing a recursive solution (continued)

## Recursive algorithm for an optimal value

If  $OPT(j)$  is an optimal solution to the subproblem for intervals  $\{1, 2, \dots, j\}$ , for any  $j \in \{1, 2, \dots, n\}$ , then:

$$OPT(j) = \text{MAX}(w_j + OPT(p(j)), OPT(j - 1))$$

## Extracting the intervals in an optimal solution

The interval  $j$  is in an optimal solution  $OPT(j)$  **if and only if** the first of the two options is larger than the second.

*Interval  $j$  belongs to an optimal solution on the set  $\{1, 2, \dots, j\}$  **if and only if***

$$w_j + OPT(p(j)) \geq OPT(j - 1)$$

# A recursive algorithm

$\text{WIS}(j)$

```
1  if  $j = 0$   
2    then return 0  
3    else return  $\text{MAX}(w_j + \text{WIS}(p(j)),$   
                   $\text{WIS}(j - 1))$ 
```

# A recursive algorithm

$\text{WIS}(j)$

```
1  if  $j = 0$   
2    then return 0  
3    else return  $\text{MAX}(w_j + \text{WIS}(p(j)),$   
                   $\text{WIS}(j - 1))$ 
```

- The initial call is  $\text{WIS}(n)$  for intervals  $\{1, 2, \dots, n\}$  sorted in non-decreasing order of the finishing times.

# A recursive algorithm

$\text{WIS}(j)$

```
1  if  $j = 0$ 
2      then return 0
3      else return  $\text{MAX}(w_j + \text{WIS}(p(j)),$   
                     $\text{WIS}(j - 1))$ 
```

- The initial call is  $\text{WIS}(n)$  for intervals  $\{1, 2, \dots, n\}$  sorted in non-decreasing order of the finishing times.
- The tree grows very rapidly, leading to **exponential** running time. The tree when  $p(j) = j - 2$  for all  $j$  shows how quickly it grows.

# A recursive algorithm

$\text{WIS}(j)$

```
1  if  $j = 0$ 
2    then return 0
3    else return  $\text{MAX}(w_j + \text{WIS}(p(j)),$   
                   $\text{WIS}(j - 1))$ 
```

- The initial call is  $\text{WIS}(n)$  for intervals  $\{1, 2, \dots, n\}$  sorted in non-decreasing order of the finishing times.
- The tree grows very rapidly, leading to **exponential** running time. The tree when  $p(j) = j - 2$  for all  $j$  shows how quickly it grows.
- There are many **overlapping subproblems**, so the obvious choice is to **memoize** the recursion.

# Memoizing the recursion

M-WIS( $j$ )

```
1  if  $j = 0$ 
2      then return 0
3  elseif  $M[j]$  is empty
4      then  $M[j] \leftarrow \text{MAX}(w_j + \text{M-WIS}(p(j)),$   

                                      $\text{M-WIS}(j - 1))$ 
5  return  $M[j]$ 
```

# Memoizing the recursion

M-WIS( $j$ )

```
1  if  $j = 0$ 
2    then return 0
3  elseif  $M[j]$  is empty
4    then  $M[j] \leftarrow \text{MAX}(w_j + \text{M-WIS}(p(j)),$   
                                $\text{M-WIS}(j - 1))$ 
5  return  $M[j]$ 
```

- Each entry in  $M[j]$  gets filled in only once at  $\Theta(1)$  time, and there are  $n + 1$  entries, so M-WIS( $n$ ) takes  $\Theta(n)$  time.



# Memoizing the recursion

M-WIS( $j$ )

```
1  if  $j = 0$ 
2    then return 0
3  elseif  $M[j]$  is empty
4    then  $M[j] \leftarrow \text{MAX}(w_j + \text{M-WIS}(p(j)),$   
                                $\text{M-WIS}(j - 1))$ 
5  return  $M[j]$ 
```

- Each entry in  $M[j]$  gets filled in only once at  $\Theta(1)$  time, and there are  $n + 1$  entries, so M-WIS( $n$ ) takes  $\Theta(n)$  time.
- Of course, sorting the intervals by the finish times takes  $\Theta(n \lg n)$  time.

# Memoizing the recursion

M-WIS( $j$ )

```

1  if  $j = 0$ 
2      then return 0
3  elseif  $M[j]$  is empty
4      then  $M[j] \leftarrow \text{MAX}(w_j + \text{M-WIS}(p(j)),$ 
                            $\text{M-WIS}(j - 1))$ 
5  return  $M[j]$ 
  
```

- Each entry in  $M[j]$  gets filled in only once at  $\Theta(1)$  time, and there are  $n + 1$  entries, so M-WIS( $n$ ) takes  $\Theta(n)$  time.
- Of course, sorting the intervals by the finish times takes  $\Theta(n \lg n)$  time.
- This memoized algorithm *plus* sorting the intervals takes  $\Theta(n \lg n) + \Theta(n) = \Theta(n \lg n)$  time.

# Computing a solution in addition to its values

- The memoized algorithm only computes the optimal value, but does not extract the intervals that make up the solution.
- The key to extracting the solution is to note that item  $j$  is in  $\vartheta$  if and only if  $w_j + M[p(j)] \geq M[j - 1]$ . This provides two ways of extracting the intervals in the optimal solution:
  - 1 Trace back from  $M[n]$  and extract the solution by checking which choice was made –  $j - 1$  or  $p(j)$  – when  $M[j]$  was included in the optimal set of intervals.
  - 2 Whenever a choice is made between two options, save in  $pred[j]$ , the predecessor pointer, the choice that was made between  $j - 1$  and  $p(j)$ .

# Computing a solution in addition to its values (continued)

- The first way recursively extracts an optimal set of intervals for a problem size of  $1 \leq j \leq n$ .
- Calling `WIS-FIND-SOLUTION( $n$ )` extracts all the intervals in the optimal solution.

# Computing a solution in addition to its values (continued)

- The first way recursively extracts an optimal set of intervals for a problem size of  $1 \leq j \leq n$ .
- Calling WIS-FIND-SOLUTION( $n$ ) extracts all the intervals in the optimal solution.

WIS-FIND-SOLUTION( $j$ )

```
1  if  $j = 0$ 
2      then Output nothing
3      else
4          if  $w_j + M[p(j)] \geq M[j - 1]$ 
5              then Output  $j$ 
6                  WIS-FIND-SOLUTION( $p(j)$ )
7          else WIS-FIND-SOLUTION( $j - 1$ )
```

# Computing a solution in addition to its values (continued)

- The second way requires that M-WIS use an auxiliary array  $pred[0..n]$  to save the predecessor of each interval in the solution.
- Initialize  $pred[j] = 0$  for all  $0 \leq j \leq n$ .

# Computing a solution in addition to its values (continued)

- The second way requires that M-WIS use an auxiliary array  $pred[0..n]$  to save the predecessor of each interval in the solution.
- Initialize  $pred[j] = 0$  for all  $0 \leq j \leq n$ .

M-WIS( $j$ )

```

1  if  $j = 0$ 
2      then return 0
3  elseif  $M[j]$  is empty
4      then if  $w_j + \text{M-WIS}(p(j)) > \text{M-WIS}(j - 1)$ 
5          then  $M[j] \leftarrow w_j + \text{M-WIS}(p(j))$ 
6               $pred[j] \leftarrow p(j)$ 
7          else  $M[j] \leftarrow \text{M-WIS}(j - 1)$ 
8               $pred[j] \leftarrow j - 1$ 
9  return  $M[j]$ 
  
```

# Computing a solution in addition to its values (continued)

Now that we have  $pred[j]$  filled in, we start from  $M[n]$  and work backwards.

- 1 If  $pred[j] = p(j)$ , then we did add the  $j^{th}$  interval in the final solution, and we continue with  $pred[j] \leftarrow p(j)$ .
- 2 if  $pred[j] \neq p(j)$ , then we did not add the  $j^{th}$  interval in the final solution, and we continue with  $pred[j] \leftarrow j - 1$ .



# Computing a solution in addition to its values (continued)

Now that we have  $pred[j]$  filled in, we start from  $M[n]$  and work backwards.

- ① If  $pred[j] = p(j)$ , then we did add the  $j^{th}$  interval in the final solution, and we continue with  $pred[j] \leftarrow p(j)$ .
- ② if  $pred[j] \neq p(j)$ , then we did not add the  $j^{th}$  interval in the final solution, and we continue with  $pred[j] \leftarrow j - 1$ .

WIS-FIND-SOLUTION( $j$ )

```
1  if  $j = 0$ 
2      then Output nothing
3      else
4          if  $pred[j] = p(j)$ 
5              then Output  $j$ 
6                  WIS-FIND-SOLUTION( $p(j)$ )
7          else WIS-FIND-SOLUTION( $j - 1$ )
```

# Computing a solution in addition to its values (continued)

Now that we have  $pred[j]$  filled in, we start from  $M[n]$  and work backwards.

- 1 If  $pred[j] = p(j)$ , then we did add the  $j^{th}$  interval in the final solution, and we continue with  $pred[j] \leftarrow p(j)$ .
- 2 if  $pred[j] \neq p(j)$ , then we did not add the  $j^{th}$  interval in the final solution, and we continue with  $pred[j] \leftarrow j - 1$ .

WIS-FIND-SOLUTION( $j$ )

```
1  if  $j = 0$ 
2      then Output nothing
3      else
4          if  $pred[j] = p(j)$ 
5              then Output  $j$ 
6                  WIS-FIND-SOLUTION( $p(j)$ )
7              else WIS-FIND-SOLUTION( $j - 1$ )
```

Can you come up with an iterative version?

# Developing a Dynamic Programming algorithm

- The value of an optimal solution  $OPT(j)$  for any  $j \in \{1, 2, 3, \dots, n\}$  depends on the values of  $OPT(p(j))$  and  $OPT(j - 1)$ .

# Developing a Dynamic Programming algorithm

- The value of an optimal solution  $OPT(j)$  for any  $j \in \{1, 2, 3, \dots, n\}$  depends on the values of  $OPT(p(j))$  and  $OPT(j - 1)$ .
- We can build the table  $M[j]$  bottom-up, starting from the base case of  $j = 0$ , up to  $n$  by using the memoized recursive formulation:  $M[j] = \text{MAX}(w_j + M[p(j)], M[j - 1])$ .

# Developing a Dynamic Programming algorithm

- The value of an optimal solution  $OPT(j)$  for any  $j \in \{1, 2, 3, \dots, n\}$  depends on the values of  $OPT(p(j))$  and  $OPT(j - 1)$ .
- We can build the table  $M[j]$  bottom-up, starting from the base case of  $j = 0$ , up to  $n$  by using the memoized recursive formulation:  $M[j] = \text{MAX}(w_j + M[p(j)], M[j - 1])$ .

## Dynamic programming algorithm

WIS( $n$ )

1  $M[0] \leftarrow 0$

2 **for**  $j \leftarrow 1$  **to**  $n$

3     **do**  $M[j] = \text{MAX}(w_j + M[p(j)], M[j - 1])$

4 **return**  $M[n]$

# Developing a Dynamic Programming algorithm

- The value of an optimal solution  $OPT(j)$  for any  $j \in \{1, 2, 3, \dots, n\}$  depends on the values of  $OPT(p(j))$  and  $OPT(j - 1)$ .
- We can build the table  $M[j]$  bottom-up, starting from the base case of  $j = 0$ , up to  $n$  by using the memoized recursive formulation:  $M[j] = \text{MAX}(w_j + M[p(j)], M[j - 1])$ .

## Dynamic programming algorithm

WIS( $n$ )

1  $M[0] \leftarrow 0$

2 **for**  $j \leftarrow 1$  **to**  $n$

3     **do**  $M[j] = \text{MAX}(w_j + M[p(j)], M[j - 1])$

4 **return**  $M[n]$

$$T(n) = \Theta(n)$$

# Computing a solution in addition to its values

WIS( $n$ )

```
1   $M[0] \leftarrow 0$ 
2  for  $j \leftarrow 1$  to  $n$ 
3      do if  $w_j + M[p(j)] > M[j - 1]$ 
4          then  $M[j] = w_j + M[p(j)]$ 
5               $pred[j] = p(j)$ 
6          else  $M[j] = M[j - 1]$ 
7               $pred[j] = j - 1$ 
8  return  $M[n]$ 
```

# Computing a solution in addition to its values

WIS( $n$ )

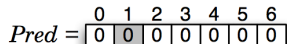
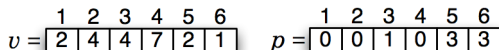
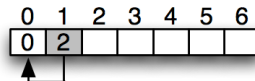
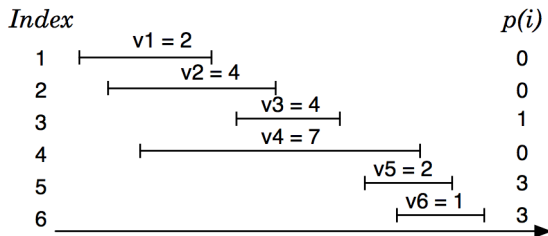
```
1   $M[0] \leftarrow 0$ 
2  for  $j \leftarrow 1$  to  $n$ 
3      do if  $w_j + M[p(j)] > M[j - 1]$ 
4          then  $M[j] = w_j + M[p(j)]$ 
5               $pred[j] = p(j)$ 
6          else  $M[j] = M[j - 1]$ 
7               $pred[j] = j - 1$ 
8  return  $M[n]$ 
```

WIS-FIND-SOLUTION( $j$ )

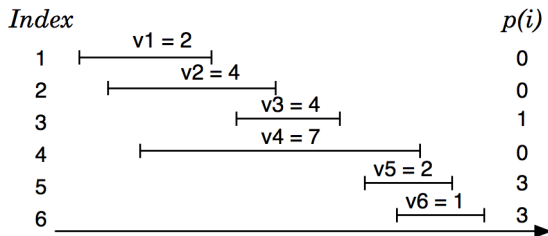
```
1   $j \leftarrow n$ 
2  while  $j > 0$ 
3      do if  $pred[j] = p(j)$ 
4          then Output  $j$ 
5           $j \leftarrow pred[j]$ 
```



# Weighted Interval Scheduling DP algorithm in action



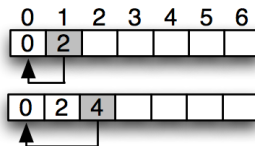
# Weighted Interval Scheduling DP algorithm in action



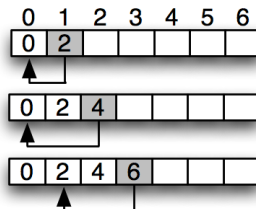
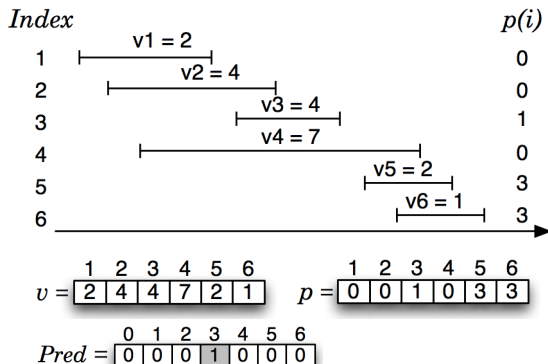
$v =$	1	2	3	4	5	6
	2	4	4	7	2	1

$p =$	1	2	3	4	5	6
	0	0	1	0	3	3

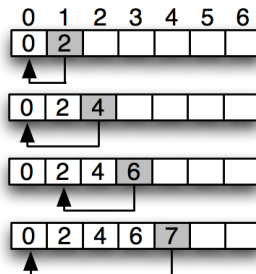
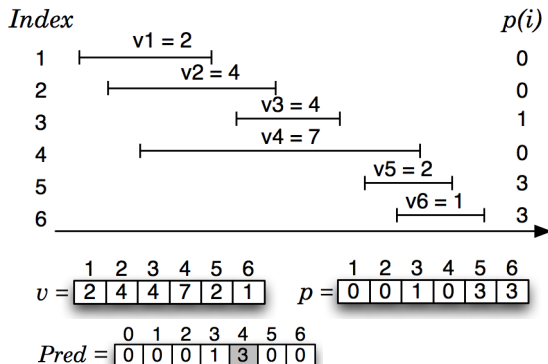
$Pred =$	0	1	2	3	4	5	6
	0	0	1	0	0	0	0



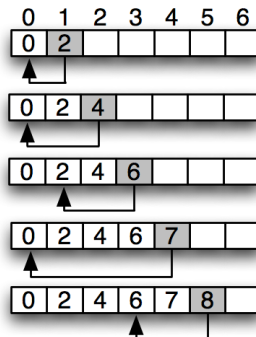
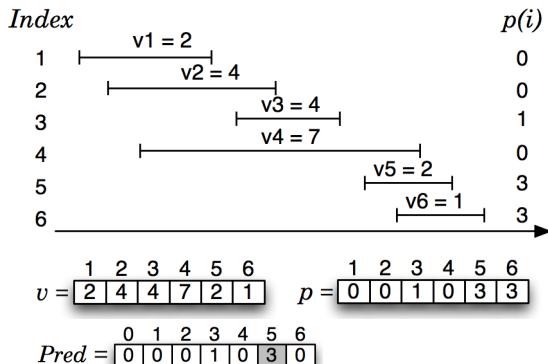
# Weighted Interval Scheduling DP algorithm in action



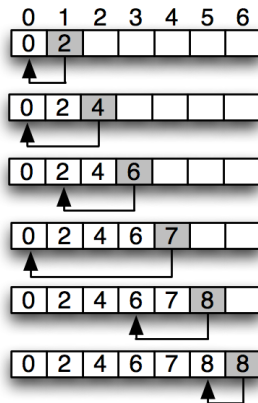
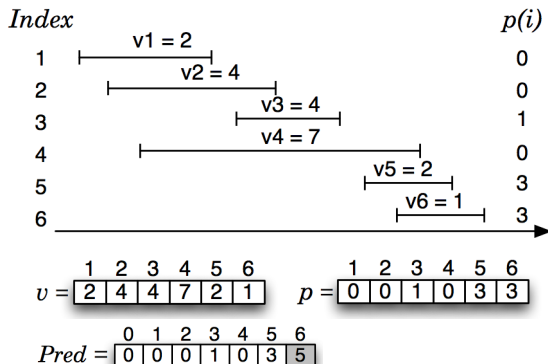
# Weighted Interval Scheduling DP algorithm in action



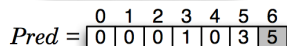
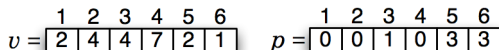
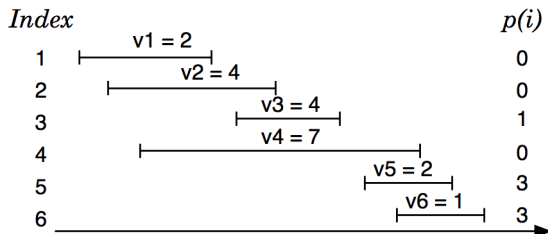
# Weighted Interval Scheduling DP algorithm in action



# Weighted Interval Scheduling DP algorithm in action

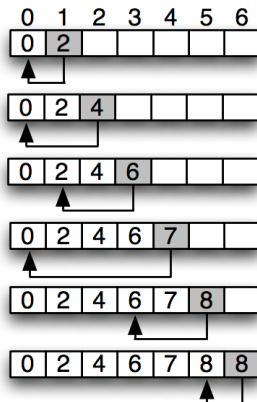


# Weighted Interval Scheduling DP algorithm in action

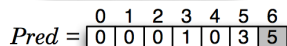
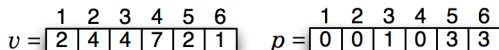
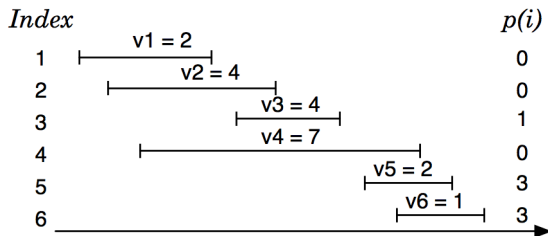


Optimal value: **8**

Optimal solution: **{5, 3, 1}**

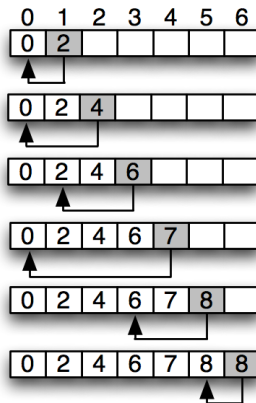


# Weighted Interval Scheduling DP algorithm in action



Optimal value: 8

Optimal solution: {1, 3, 5}





# So, you think you understand Dynamic Programming now?

## Answer the following questions

- 1 Instead of sorting the intervals by **finish time**, what if we sorted the requests by **start time**?

# So, you think you understand Dynamic Programming now?

## Answer the following questions

- 1 Instead of sorting the intervals by **finish time**, what if we sorted the requests by **start time**?
- 2 What if we didn't sort the requests at all? Would it still work?

# So, you think you understand Dynamic Programming now?

## Answer the following questions

- 1 Instead of sorting the intervals by **finish time**, what if we sorted the requests by **start time**?
- 2 What if we didn't sort the requests at all? Would it still work?
- 3 If all the *weights* are the same, what does this problem become?

# So, you think you understand Dynamic Programming now?

## Answer the following questions

- 1 Instead of sorting the intervals by **finish time**, what if we sorted the requests by **start time**?
- 2 What if we didn't sort the requests at all? Would it still work?
- 3 If all the *weights* are the same, what does this problem become? Can you solve it using DP?

# Contents

## 1 Introduction

- Memoization
- Dynamic programming
- Weighted interval scheduling problem
- 0/1 Knapsack problem
- Coin changing problem
- What problems can be solved by DP?
- Conclusion

# 0/1 knapsack problem

## Definition (0/1 knapsack problem)

Given a set  $S$  of  $n$  items, such that each item  $i$  has a positive benefit  $v_i$  and a positive weight  $w_i$ , the goal is to find the maximum-benefit subset that does not exceed a given weight  $W$ .

# 0/1 knapsack problem

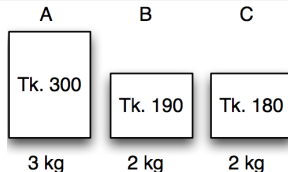
## Definition (0/1 knapsack problem)

Given a set  $S$  of  $n$  items, such that each item  $i$  has a positive benefit  $v_i$  and a positive weight  $w_i$ , the goal is to find the maximum-benefit subset that does not exceed a given weight  $W$ . Formally, we wish to determine a subset of  $S$  that maximizes  $\sum_{i \in S} v_i$ , subject to  $\sum_{i \in S} w_i \leq W$ .

# 0/1 knapsack problem

## Definition (0/1 knapsack problem)

Given a set  $S$  of  $n$  items, such that each item  $i$  has a positive benefit  $v_i$  and a positive weight  $w_i$ , the goal is to find the maximum-benefit subset that does not exceed a given weight  $W$ . Formally, we wish to determine a subset of  $S$  that maximizes  $\sum_{i \in S} v_i$ , subject to  $\sum_{i \in S} w_i \leq W$ .



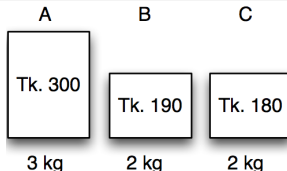
Maximum weight:  $W = 4 \text{ kg}$



# 0/1 knapsack problem

## Definition (0/1 knapsack problem)

Given a set  $S$  of  $n$  items, such that each item  $i$  has a positive benefit  $v_i$  and a positive weight  $w_i$ , the goal is to find the maximum-benefit subset that does not exceed a given weight  $W$ . Formally, we wish to determine a subset of  $S$  that maximizes  $\sum_{i \in S} v_i$ , subject to  $\sum_{i \in S} w_i \leq W$ .



Maximum weight:  $W = 4$  kg

Optimal solution: items  $B$  and  $C$

Benefit: 370

# Developing a recursive solution

- Let  $S$  be an instance of a 0/1 Knapsack problem, and  $v^*$  be an optimal solution (even if we have no idea what it is yet).

# Developing a recursive solution

- Let  $S$  be an instance of a 0/1 Knapsack problem, and  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item  $i$  in  $\vartheta$  does not preclude any other item  $j \neq i$  in  $\vartheta$ .

# Developing a recursive solution

- Let  $S$  be an instance of a 0/1 Knapsack problem, and  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item  $i$  in  $\vartheta$  does not preclude any other item  $j \neq i$  in  $\vartheta$ .
- If **item  $n$**  weighs more than the maximum allowed weight, it will not be in  $\vartheta$ .

# Developing a recursive solution

- Let  $S$  be an instance of a 0/1 Knapsack problem, and  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item  $i$  in  $\vartheta$  does not preclude any other item  $j \neq i$  in  $\vartheta$ .
- If item  $n$  weighs more than the maximum allowed weight, it will not be in  $\vartheta$ .
- Otherwise, all we can say about  $\vartheta$  is the following: item  $n$  (the last one) either belongs to  $\vartheta$ , or it doesn't.

# Developing a recursive solution

- Let  $S$  be an instance of a 0/1 Knapsack problem, and  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item  $i$  in  $\vartheta$  does not preclude any other item  $j \neq i$  in  $\vartheta$ .
- If item  $n$  weighs more than the maximum allowed weight, it will not be in  $\vartheta$ .
- Otherwise, all we can say about  $\vartheta$  is the following: item  $n$  (the last one) either belongs to  $\vartheta$ , or it doesn't.
  - If  $n \in \vartheta$  Then the optimal solution contains  $n$ , plus an optimal solution for the other  $n - 1$  items, but with a reduced maximum weight of  $W - w_n$ .

# Developing a recursive solution

- Let  $S$  be an instance of a 0/1 Knapsack problem, and  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item  $i$  in  $\vartheta$  does not preclude any other item  $j \neq i$  in  $\vartheta$ .
- If item  $n$  weighs more than the maximum allowed weight, it will not be in  $\vartheta$ .
- Otherwise, all we can say about  $\vartheta$  is the following: item  $n$  (the last one) either belongs to  $\vartheta$ , or it doesn't.
  - If  $n \in \vartheta$  Then the optimal solution contains  $n$ , plus an optimal solution for the other  $n - 1$  items, but with a reduced maximum weight of  $W - w_n$ .
  - If  $n \notin \vartheta$  Then  $\vartheta$  simply contains an optimal solution for the first  $n - 1$  items, with the maximum allowed weight  $W$  remaining unchanged.

# Developing a recursive solution

- Let  $S$  be an instance of a 0/1 Knapsack problem, and  $\vartheta$  be an optimal solution (even if we have no idea what it is yet).
- Note that the presence of an item  $i$  in  $\vartheta$  does not preclude any other item  $j \neq i$  in  $\vartheta$ .
- If item  $n$  weighs more than the maximum allowed weight, it will not be in  $\vartheta$ .
- Otherwise, all we can say about  $\vartheta$  is the following: item  $n$  (the last one) either belongs to  $\vartheta$ , or it doesn't.
  - If  $n \in \vartheta$  Then the optimal solution contains  $n$ , plus an optimal solution for the other  $n - 1$  items, but with a reduced maximum weight of  $W - w_n$ .
  - If  $n \notin \vartheta$  Then  $\vartheta$  simply contains an optimal solution for the first  $n - 1$  items, with the maximum allowed weight  $W$  remaining unchanged.
- We have two parameters for each subproblem – the items  $S$ , and the maximum allowed weight  $W$ .



# Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$ .  
▷  $\vartheta(n, W) = \vartheta(n-1, W)$

# Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$ .
  - ▷  $\vartheta(n, W) = \vartheta(n-1, W)$
- Otherwise,  $n$  is either  $\in \vartheta$  or  $\notin \vartheta$ .
  - If  $n \in \vartheta$ , then  $\vartheta(n, W)$  is an optimal solution to the subproblem for items  $\{1, 2, \dots, n\}$ :
    - ▷  $\vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$

# Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$ .
  - ▷  $\vartheta(n, W) = \vartheta(n-1, W)$
- Otherwise,  $n$  is either  $\in \vartheta$  or  $\notin \vartheta$ .
  - If  $n \in \vartheta$ , then  $\vartheta(n, W)$  is an optimal solution to the subproblem for items  $\{1, 2, \dots, n\}$ :
    - ▷  $\vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$
  - If  $n \notin \vartheta$ , then  $\vartheta(n, W)$  simply contains an optimal solution to the subproblem consisting of the intervals  $\{1, 2, \dots, n-1\}$ :
    - ▷  $\vartheta(n, W) = \vartheta(n-1, W)$

# Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$ .
  - ▷  $\vartheta(n, W) = \vartheta(n-1, W)$
- Otherwise,  $n$  is either  $\in \vartheta$  or  $\notin \vartheta$ .
  - If  $n \in \vartheta$ , then  $\vartheta(n, W)$  is an optimal solution to the subproblem for items  $\{1, 2, \dots, n\}$ :
    - ▷  $\vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$
  - If  $n \notin \vartheta$ , then  $\vartheta(n, W)$  simply contains an optimal solution to the subproblem consisting of the intervals  $\{1, 2, \dots, n-1\}$ :
    - ▷  $\vartheta(n, W) = \vartheta(n-1, W)$
  - Since an optimal solution must maximize the sum of the weights in the intervals it contains, we accept the larger of the two.

# Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$ .
  - ▷  $\vartheta(n, W) = \vartheta(n-1, W)$
- Otherwise,  $n$  is either  $\in \vartheta$  or  $\notin \vartheta$ .
  - If  $n \in \vartheta$ , then  $\vartheta(n, W)$  is an optimal solution to the subproblem for items  $\{1, 2, \dots, n\}$ :
    - ▷  $\vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$
  - If  $n \notin \vartheta$ , then  $\vartheta(n, W)$  simply contains an optimal solution to the subproblem consisting of the intervals  $\{1, 2, \dots, n-1\}$ :
    - ▷  $\vartheta(n, W) = \vartheta(n-1, W)$
  - Since an optimal solution must maximize the sum of the weights in the intervals it contains, we accept the larger of the two.
    - ▷  $\vartheta(n, W) = \text{MAX}(v_n + \vartheta(n-1, W - w_n), \vartheta(n-1, W))$

# Developing a recursive solution (continued)

## Recursive algorithm for an optimal value

If  $OPT(j, w)$  is an optimal solution to the subproblem for items  $\{1, 2, \dots, j\}$ , for any  $j \in \{1, 2, \dots, n\}$ , and with a maximum allowed weight of  $w$ , then:

$$OPT(j, w) = \begin{cases} OPT(j-1, w) & \text{if } w_j > w, \\ \text{MAX}(v_j + OPT(j-1, w - w_j), \\ \quad OPT(j-1, w)) & \text{otherwise.} \end{cases}$$

# Developing a recursive solution (continued)

## Recursive algorithm for an optimal value

If  $OPT(j, w)$  is an optimal solution to the subproblem for items  $\{1, 2, \dots, j\}$ , for any  $j \in \{1, 2, \dots, n\}$ , and with a maximum allowed weight of  $w$ , then:

$$OPT(j, w) = \begin{cases} OPT(j-1, w) & \text{if } w_j > w, \\ \text{MAX}(v_j + OPT(j-1, w - w_j), \\ \quad OPT(j-1, w)) & \text{otherwise.} \end{cases}$$

## Extracting the items in an optimal solution

The item  $j$  is in an optimal solution  $OPT(j, w)$  **if and only if** the first of the two options is larger than the second.

$$v_j + OPT(j-1, w - w_j) \geq OPT(j-1, w)$$

# A recursive algorithm

KNAPSACK( $j, w$ )

```
1  if  $j = 0$  or  $w = 0$ 
2      then return 0
3  elseif  $w_j > w$ 
4      then return KNAPSACK( $j - 1, w$ )
5  else return MAX( $v_j + \text{KNAPSACK}(j - 1, w - w_j)$ ,
                  KNAPSACK( $j - 1, w$ ))
```



# A recursive algorithm

$\text{KNAPSACK}(j, w)$

```
1  if  $j = 0$  or  $w = 0$ 
2    then return 0
3  elseif  $w_j > w$ 
4    then return  $\text{KNAPSACK}(j - 1, w)$ 
5  else return  $\max(v_j + \text{KNAPSACK}(j - 1, w - w_j),$   
                $\text{KNAPSACK}(j - 1, w))$ 
```

- The initial call is  $\text{KNAPSACK}(n, W)$ .

# A recursive algorithm

KNAPSACK( $j, w$ )

```
1  if  $j = 0$  or  $w = 0$ 
2      then return 0
3  elseif  $w_j > w$ 
4      then return KNAPSACK( $j - 1, w$ )
5  else return MAX( $v_j + \text{KNAPSACK}(j - 1, w - w_j)$ ,
                  KNAPSACK( $j - 1, w$ ))
```

- The initial call is KNAPSACK( $n, W$ ).
- The tree grows very rapidly, leading to **exponential** running time.

# A recursive algorithm

KNAPSACK( $j, w$ )

```
1  if  $j = 0$  or  $w = 0$ 
2      then return 0
3  elseif  $w_j > w$ 
4      then return KNAPSACK( $j - 1, w$ )
5  else return MAX( $v_j + \text{KNAPSACK}(j - 1, w - w_j)$ ,
                  KNAPSACK( $j - 1, w$ ))
```

- The initial call is KNAPSACK( $n, W$ ).
- The tree grows very rapidly, leading to **exponential** running time.
- There are many **overlapping subproblems**, so the obvious choice is to **memoize** the recursion.

## Memoizing the recursion

$$\text{M-KNAPSACK}(j, w)$$

1 **if**  $j = 0$  or  $w = 0$

```
2    then return 0
```

```
3 elseif  $M[j, w]$  is empty
```

```

4   then  $M[j, w] \leftarrow \text{MAX}(v_j + \text{M-KNAPSACK}(j - 1, w - w_j),$   

 $\text{M-KNAPSACK}(j - 1, w))$ 

```

```

5  return  $M[j, w]$ 

```

# Memoizing the recursion

M-KNAPSACK( $j, w$ )

```
1  if  $j = 0$  or  $w = 0$ 
2    then return 0
3  elseif  $M[j, w]$  is empty
4    then  $M[j, w] \leftarrow \text{MAX}(v_j + \text{M-KNAPSACK}(j - 1, w - w_j),$   

                                $\text{M-KNAPSACK}(j - 1, w))$ 
5  return  $M[j, w]$ 
```

- Each entry in  $M[j, w]$  gets filled in only once at  $\Theta(1)$  time, and there are  $n + 1 \times W + 1$  entries, so M-KNAPSACK( $n, W$ ) takes  $\Theta(nW)$  time.

# Memoizing the recursion

M-KNAPSACK( $j, w$ )

```
1  if  $j = 0$  or  $w = 0$ 
2    then return 0
3  elseif  $M[j, w]$  is empty
4    then  $M[j, w] \leftarrow \text{MAX}(v_j + \text{M-KNAPSACK}(j - 1, w - w_j),$   

                                $\text{M-KNAPSACK}(j - 1, w))$ 
5  return  $M[j, w]$ 
```

- Each entry in  $M[j, w]$  gets filled in only once at  $\Theta(1)$  time, and there are  $n + 1 \times W + 1$  entries, so M-KNAPSACK( $n, W$ ) takes  $\Theta(nW)$  time.
- Is this a linear-time algorithm?

# Memoizing the recursion

M-KNAPSACK( $j, w$ )

```
1  if  $j = 0$  or  $w = 0$ 
2    then return 0
3  elseif  $M[j, w]$  is empty
4    then  $M[j, w] \leftarrow \text{MAX}(v_j + \text{M-KNAPSACK}(j - 1, w - w_j),$   

                                    $\text{M-KNAPSACK}(j - 1, w))$ 
5  return  $M[j, w]$ 
```

- Each entry in  $M[j, w]$  gets filled in only once at  $\Theta(1)$  time, and there are  $n + 1 \times W + 1$  entries, so M-KNAPSACK( $n, W$ ) takes  $\Theta(nW)$  time.
- Is this a linear-time algorithm?
- This is an example of a pseudo-polynomial problem, since it depends on another parameter  $W$  that is independent of the problem size.

# Developing a Dynamic Programming algorithm

KNAPSACK( $n, W$ )

```
1  for  $i \leftarrow 0$  to  $n$       ▷ no remaining capacity
2      do  $M[i, 0] \leftarrow 0$ 
3  for  $w \leftarrow 0$  to  $W$     ▷ no item to choose from
4      do  $M[0, w] \leftarrow 0$ 
5  for  $j \leftarrow 1$  to  $n$ 
6      do for  $w \leftarrow 1$  to  $W$ 
7          do if  $w_j > w$  //we cannot take object  $j$ 
8              then  $M[j, w] = M[j - 1, w]$ 
9              else  $M[j, w] \leftarrow \text{MAX}(v_j + M[j - 1, w - w_j],$   

                                      $M[j - 1, w])$ 
10 return  $M[n, W]$ 
```



# 0/1 Knapsack recursive algorithm in action

Given the following (from M. H. Alsuwaiyel, ex. 7.6):

$$W = 9$$

$$w_i = \{2, 3, 4, 5\}$$

$$v_i = \{3, 4, 5, 7\}$$

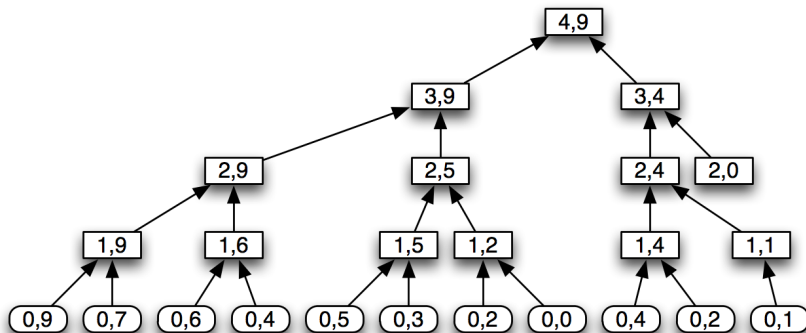
# 0/1 Knapsack recursive algorithm in action

Given the following (from M. H. Alsuwaiyel, ex. 7.6):

$$W = 9$$

$$w_i = \{2, 3, 4, 5\}$$

$$v_i = \{3, 4, 5, 7\}$$



# 0/1 Knapsack DP algorithm in action

Given the following (from M. H. Alsuwaiyel, ex. 7.6):

$$W = 9$$

$$w_i = \{2, 3, 4, 5\}$$

$$v_i = \{3, 4, 5, 7\}$$

	0	1	2	3	4	5	6	7	8	9
4	-	-	-	-	-	-	-	-	-	-
3	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-
1	-	-	-	-	-	-	-	-	-	-
0	-	-	-	-	-	-	-	-	-	-

# 0/1 Knapsack DP algorithm in action

Given the following (from M. H. Alsuwaiyel, ex. 7.6):

$$W = 9$$

$$w_i = \{2, 3, 4, 5\}$$

$$v_i = \{3, 4, 5, 7\}$$

	0	1	2	3	4	5	6	7	8	9
4	0	0	3	4	5	7	8	10	11	12
3	0	0	3	4	4	7	8	9	9	12
2	0	0	3	4	4	7	7	7	7	7
1	0	0	3	3	3	3	3	3	3	3
0	0	0	0	0	0	0	0	0	0	0

## Related problem: Subset Sums problem

### Definition (Subset Sums problem)

Given a set  $S$  of  $n$  items, such that each item  $i$  has a positive weight  $w_i$ , the goal is to find the maximum-weight subset that does not exceed a given weight  $W$ .

# Related problem: Subset Sums problem

## Definition (Subset Sums problem)

Given a set  $S$  of  $n$  items, such that each item  $i$  has a positive weight  $w_i$ , the goal is to find the maximum-weight subset that does not exceed a given weight  $W$ .

Formally, we wish to determine a subset of  $S$  that maximizes  $\sum_{i \in S} w_i$ , subject to  $\sum_{i \in S} w_i \leq W$ .

## Related problem: Subset Sums problem

### Definition (Subset Sums problem)

Given a set  $S$  of  $n$  items, such that each item  $i$  has a positive weight  $w_i$ , the goal is to find the maximum-weight subset that does not exceed a given weight  $W$ .

Formally, we wish to determine a subset of  $S$  that maximizes  $\sum_{i \in S} w_i$ , subject to  $\sum_{i \in S} w_i \leq W$ .

- How is this similar to the 0/1 Knapsack problem?

## Related problem: Subset Sums problem

### Definition (Subset Sums problem)

Given a set  $S$  of  $n$  items, such that each item  $i$  has a positive weight  $w_i$ , the goal is to find the maximum-weight subset that does not exceed a given weight  $W$ .

Formally, we wish to determine a subset of  $S$  that maximizes  $\sum_{i \in S} w_i$ , subject to  $\sum_{i \in S} w_i \leq W$ .

- How is this similar to the 0/1 Knapsack problem?
- Can you solve this using the same algorithm?



# Contents

## 1 Introduction

- Memoization
- Dynamic programming
- Weighted interval scheduling problem
- 0/1 Knapsack problem
- **Coin changing problem**
- What problems can be solved by DP?
- Conclusion

# Coin changing problem

## Definition

Given coin denominations in  $C = \{c_i\}$ , make change for a given amount  $A$  with the minimum number of coins.

# Coin changing problem

## Definition

Given coin denominations in  $C = \{c_i\}$ , make change for a given amount  $A$  with the minimum number of coins.

## Example

Coin denominations,  $C = \{12, 5, 1\}$       Amount to change,  $A = 15$

# Coin changing problem

## Definition

Given coin denominations in  $C = \{c_i\}$ , make change for a given amount  $A$  with the minimum number of coins.

## Example

Coin denominations,  $C = \{12, 5, 1\}$       Amount to change,  $A = 15$

- 1 Choose 0 12 coins, so remaining is 15

# Coin changing problem

## Definition

Given coin denominations in  $C = \{c_i\}$ , make change for a given amount  $A$  with the minimum number of coins.

## Example

Coin denominations,  $C = \{12, 5, 1\}$       Amount to change,  $A = 15$

- 1 Choose 0 12 coins, so remaining is 15
- 2 Choose 3 5 coins, so remaining is  $15 - 3 * 5 = 0$

# Coin changing problem

## Definition

Given coin denominations in  $C = \{c_i\}$ , make change for a given amount  $A$  with the minimum number of coins.

## Example

Coin denominations,  $C = \{12, 5, 1\}$       Amount to change,  $A = 15$

- 1 Choose 0 12 coins, so remaining is 15
- 2 Choose 3 5 coins, so remaining is  $15 - 3 * 5 = 0$

Solution: 3 coins.

# Coin changing problem

## Definition

Given coin denominations in  $C = \{c_i\}$ , make change for a given amount  $A$  with the minimum number of coins.

## Example

Coin denominations,  $C = \{12, 5, 1\}$       Amount to change,  $A = 15$

- 1 Choose 0 12 coins, so remaining is 15
- 2 Choose 3 5 coins, so remaining is  $15 - 3 * 5 = 0$

Solution: 3 coins.

## Questions

What is the natural search space? Does this problem have a Dynamic Programming solution? If so, how do we develop it?

# Developing a recursive solution

Coin denominations,  $C = \{12, 5, 1\}$       Amount to change,  $A = 15$



# Developing a recursive solution

Coin denominations,  $C = \{12, 5, 1\}$       Amount to change,  $A = 15$

- The best combination of coins for 15 paisa must be one of the following:

# Developing a recursive solution

Coin denominations,  $C = \{12, 5, 1\}$     Amount to change,  $A = 15$

- The best combination of coins for 15 paisa must be one of the following:
  - 1 Best combination for  $15 - 12 = 3$  paisa, plus a 12 paisa coin.

# Developing a recursive solution

Coin denominations,  $C = \{12, 5, 1\}$     Amount to change,  $A = 15$

- The best combination of coins for 15 paisa must be one of the following:
  - 1 Best combination for  $15 - 12 = 3$  paisa, plus a 12 paisa coin.
  - 2 Best combination for  $15 - 5 = 10$  paisa, plus a 5 paisa coin.

# Developing a recursive solution

Coin denominations,  $C = \{12, 5, 1\}$       Amount to change,  $A = 15$

- The best combination of coins for 15 paisa must be one of the following:
  - 1 Best combination for  $15 - 12 = 3$  paisa, plus a 12 paisa coin.
  - 2 Best combination for  $15 - 5 = 10$  paisa, plus a 5 paisa coin.
  - 3 Best combination for  $15 - 1 = 14$  paisa, plus a 1 paisa coin.

# Developing a recursive solution

Coin denominations,  $C = \{12, 5, 1\}$     Amount to change,  $A = 15$

- The best combination of coins for 15 paisa must be one of the following:
  - 1 Best combination for  $15 - 12 = 3$  paisa, plus a 12 paisa coin.
  - 2 Best combination for  $15 - 5 = 10$  paisa, plus a 5 paisa coin.
  - 3 Best combination for  $15 - 1 = 14$  paisa, plus a 1 paisa coin.
- Since we're minimizing the number of coins, the best combination would be the minimum of these three choices.

# Developing a recursive solution

Coin denominations,  $C = \{12, 5, 1\}$     Amount to change,  $A = 15$

- The best combination of coins for 15 paisa must be one of the following:
  - 1 Best combination for  $15 - 12 = 3$  paisa, plus a 12 paisa coin.
  - 2 Best combination for  $15 - 5 = 10$  paisa, plus a 5 paisa coin.
  - 3 Best combination for  $15 - 1 = 14$  paisa, plus a 1 paisa coin.
- Since we're minimizing the number of coins, the best combination would be the minimum of these three choices.
- By recursively solving for the best combination, this can be generalized to  $|C|$  denominations to make change for any amount  $A$ .

# Developing a recursive solution

Coin denominations,  $C = \{12, 5, 1\}$     Amount to change,  $A = 15$

- The best combination of coins for 15 paisa must be one of the following:
  - 1 Best combination for  $15 - 12 = 3$  paisa, plus a 12 paisa coin.
  - 2 Best combination for  $15 - 5 = 10$  paisa, plus a 5 paisa coin.
  - 3 Best combination for  $15 - 1 = 14$  paisa, plus a 1 paisa coin.
- Since we're minimizing the number of coins, the best combination would be the minimum of these three choices.
- By recursively solving for the best combination, this can be generalized to  $|C|$  denominations to make change for any amount  $A$ .
- What are the subproblems?

# Developing a recursive solution (continued)

If  $OPT(p)$  is the minimum number of coins needed to make change for amount  $p$  with denominations  $C = \{c_1, c_2, \dots, c_k\}$ , then:



# Developing a recursive solution (continued)

If  $OPT(p)$  is the minimum number of coins needed to make change for amount  $p$  with denominations  $C = \{c_1, c_2, \dots, c_k\}$ , then:

- The coin  $c_i$  chosen at any step must be smaller than  $p$ , the amount left at that point.

# Developing a recursive solution (continued)

If  $OPT(p)$  is the minimum number of coins needed to make change for amount  $p$  with denominations  $C = \{c_1, c_2, \dots, c_k\}$ , then:

- The coin  $c_i$  chosen at any step must be smaller than  $p$ , the amount left at that point.
- Once we choose  $c_i \leq p$ ,  $OPT(p) = 1 + OPT(p - c_i)$ , since we have to find the best combination for the remaining amount (picking a coin smaller than the amount at each step).

# Developing a recursive solution (continued)

If  $OPT(p)$  is the minimum number of coins needed to make change for amount  $p$  with denominations  $C = \{c_1, c_2, \dots, c_k\}$ , then:

- The coin  $c_i$  chosen at any step must be smaller than  $p$ , the amount left at that point.
- Once we choose  $c_i \leq p$ ,  $OPT(p) = 1 + OPT(p - c_i)$ , since we have to find the best combination for the remaining amount (picking a coin smaller than the amount at each step).
- Since we don't know which coin would be chosen, we have to search all  $|C|$  denominations and find the minimum.

# Developing a recursive solution (continued)

If  $OPT(p)$  is the minimum number of coins needed to make change for amount  $p$  with denominations  $C = \{c_1, c_2, \dots, c_k\}$ , then:

- The coin  $c_i$  chosen at any step must be smaller than  $p$ , the amount left at that point.
- Once we choose  $c_i \leq p$ ,  $OPT(p) = 1 + OPT(p - c_i)$ , since we have to find the best combination for the remaining amount (picking a coin smaller than the amount at each step).
- Since we don't know which coin would be chosen, we have to search all  $|C|$  denominations and find the minimum.
- The number of coins for 0 amount is 0.

# Developing a recursive solution (continued)

If  $OPT(p)$  is the minimum number of coins needed to make change for amount  $p$  with denominations  $C = \{c_1, c_2, \dots, c_k\}$ , then:

- The coin  $c_i$  chosen at any step must be smaller than  $p$ , the amount left at that point.
- Once we choose  $c_i \leq p$ ,  $OPT(p) = 1 + OPT(p - c_i)$ , since we have to find the best combination for the remaining amount (picking a coin smaller than the amount at each step).
- Since we don't know which coin would be chosen, we have to search all  $|C|$  denominations and find the minimum.
- The number of coins for 0 amount is 0.

## Recurrence

$$OPT(p) = \begin{cases} 0 & \text{if } p = 0 \\ \min_{i: c_i \leq p} \{1 + OPT(p - c_i)\} & \text{if } p > 0 \end{cases}$$

# A recursive algorithm

CHANGE( $n, C$ )

```
1  if  $n = 0$ 
2    then return 0
3    else  $min \leftarrow \infty$ 
4        for  $i \leftarrow 1$  to  $|C|$ 
5            do if  $c_i \leq n$  and  $1 + \text{CHANGE}(n - c_i, C) < min$ 
6                then  $min \leftarrow 1 + \text{CHANGE}(n - c_i, C)$ 
```

# A recursive algorithm

CHANGE( $n, C$ )

```
1  if  $n = 0$ 
2      then return 0
3      else  $min \leftarrow \infty$ 
4          for  $i \leftarrow 1$  to  $|C|$ 
5              do if  $c_i \leq n$  and  $1 + \text{CHANGE}(n - c_i, C) < min$ 
6                  then  $min \leftarrow 1 + \text{CHANGE}(n - c_i, C)$ 
```

- The initial call is CHANGE( $A, C$ ).

# A recursive algorithm

CHANGE( $n, C$ )

```
1  if  $n = 0$ 
2    then return 0
3    else  $min \leftarrow \infty$ 
4        for  $i \leftarrow 1$  to  $|C|$ 
5            do if  $c_i \leq n$  and  $1 + \text{CHANGE}(n - c_i, C) < min$ 
6                then  $min \leftarrow 1 + \text{CHANGE}(n - c_i, C)$ 
```

- The initial call is CHANGE( $A, C$ ).
- The tree grows very rapidly, leading to **exponential** running time.



# A recursive algorithm

CHANGE( $n, C$ )

```
1  if  $n = 0$ 
2    then return 0
3    else  $min \leftarrow \infty$ 
4          for  $i \leftarrow 1$  to  $|C|$ 
5              do if  $c_i \leq n$  and  $1 + \text{CHANGE}(n - c_i, C) < min$ 
6                  then  $min \leftarrow 1 + \text{CHANGE}(n - c_i, C)$ 
```

- The initial call is CHANGE( $A, C$ ).
- The tree grows very rapidly, leading to **exponential** running time.
- There are many **overlapping subproblems**, so the obvious choice is to **memoize** the recursion.

# Memoizing the recursion

M-CHANGE( $n, C$ )

```
1  if  $n = 0$ 
2      then return 0
3      else if  $M[n]$  is empty
4          then  $min \leftarrow \infty$ 
5              for  $i \leftarrow 1$  to  $|C|$ 
6                  do if  $c_i \leq n$  and
9                       $1 + \text{M-CHANGE}(n - c_i, C) < min$ 
7                          then  $min \leftarrow 1 + \text{M-CHANGE}(n - c_i, C)$ 
8                       $M[n] \leftarrow min$ 
9      return  $M[n]$ 
```

# Memoizing the recursion

M-CHANGE( $n, C$ )

```

1  if  $n = 0$ 
2      then return 0
3      else if  $M[n]$  is empty
4          then  $min \leftarrow \infty$ 
5              for  $i \leftarrow 1$  to  $|C|$ 
6                  do if  $c_i \leq n$  and
9                       $1 + \text{M-CHANGE}(n - c_i, C) < min$ 
7                          then  $min \leftarrow 1 + \text{M-CHANGE}(n - c_i, C)$ 
8                       $M[n] \leftarrow min$ 
9      return  $M[n]$ 
  
```

- Each entry in  $M[n]$  gets filled in only once at  $\Theta(|C|)$  time, and there are  $n + 1$  entries, so M-CHANGE( $n$ ) takes

$\Theta(n|C|)$  time.

# Memoizing the recursion

M-CHANGE( $n, C$ )

```

1  if  $n = 0$ 
2      then return 0
3      else if  $M[n]$  is empty
4          then  $min \leftarrow \infty$ 
5              for  $i \leftarrow 1$  to  $|C|$ 
6                  do if  $c_i \leq n$  and
9                       $1 + \text{M-CHANGE}(n - c_i, C) < min$ 
7                          then  $min \leftarrow 1 + \text{M-CHANGE}(n - c_i, C)$ 
8                       $M[n] \leftarrow min$ 
9      return  $M[n]$ 
  
```

- Each entry in  $M[n]$  gets filled in only once at  $\Theta(|C|)$  time, and there are  $n + 1$  entries, so M-CHANGE( $n$ ) takes  $\Theta(n|C|)$  time.
- Another **pseudo-polynomial** problem!

# Developing a Dynamic Programming algorithm

CHANGE( $n, C$ )

▷  $M = [0 \dots n], S = [0 \dots n]$

1  $M[0] \leftarrow 0$       no amount to change

2 **for**  $p \leftarrow 1$  **to**  $n$

3     **do**  $min \leftarrow \infty$

4         **for**  $i \leftarrow 1$  **to**  $|C|$

5             **do if**  $c_i \leq p$  and  $1 + M[p - c_i] < min$

6                 **then**  $min \leftarrow 1 + M[p - c_i]$

7                      $coin \leftarrow i$

8          $M[p] \leftarrow min$

9          $S[p] \leftarrow coin$

10 **return**  $M$  and  $S$

# Developing a Dynamic Programming algorithm

CHANGE( $n, C$ )

▷  $M = [0 \dots n], S = [0 \dots n]$

```
1   $M[0] \leftarrow 0$       no amount to change
2  for  $p \leftarrow 1$  to  $n$ 
3      do  $min \leftarrow \infty$ 
4          for  $i \leftarrow 1$  to  $|C|$ 
5              do if  $c_i \leq p$  and  $1 + M[p - c_i] < min$ 
6                  then  $min \leftarrow 1 + M[p - c_i]$ 
7                       $coin \leftarrow i$ 
8           $M[p] \leftarrow min$ 
9           $S[p] \leftarrow coin$ 
10 return  $M$  and  $S$ 
```

- $M[p]$  for all  $0 \leq p \leq n$  – minimum number of coins needed to change for  $p$  paisa.
- $S[p]$  for all  $0 \leq p \leq n$  – the first coin chosen in computing an optimal solution for making change for  $p$  paisa.

# Computing a solution in addition to its values

- The  $S$  array in the algorithm “remembers” the first coin we use when computing an optimal value for a given amount.
- We go backwards using  $S[n]$  until  $n = 0$  and find the coin that was added at each step.

# Computing a solution in addition to its values

- The  $S$  array in the algorithm “remembers” the first coin we use when computing an optimal value for a given amount.
- We go backwards using  $S[n]$  until  $n = 0$  and find the coin that was added at each step.

COINS( $S, C, n$ )

```
1  while  $n > 0$ 
2      do Output  $S[n]$ 
3       $n \leftarrow n - C_{S[n]}$ 
```



# Contents

## 1 Introduction

- Memoization
- Dynamic programming
- Weighted interval scheduling problem
- 0/1 Knapsack problem
- Coin changing problem
- What problems can be solved by DP?
- Conclusion

# Problem types solved by Dynamic Programming

- The most important part of DP is to set up the subproblem structure.

# Problem types solved by Dynamic Programming

- The most important part of DP is to set up the subproblem structure.
- DP is not applicable to all optimization problems.

# Problem types solved by Dynamic Programming

- The most important part of DP is to set up the subproblem structure.
- DP is not applicable to all optimization problems.
- If a problem has the following properties, then it's likely to have a dynamic programming solution.

# Problem types solved by Dynamic Programming

- The most important part of DP is to set up the subproblem structure.
- DP is not applicable to all optimization problems.
- If a problem has the following properties, then it's likely to have a dynamic programming solution.

**Polynomially many subproblems** The total number of subproblems should be a polynomial, or else DP may not provide an efficient solution.

# Problem types solved by Dynamic Programming

- The most important part of DP is to set up the subproblem structure.
- DP is not applicable to all optimization problems.
- If a problem has the following properties, then it's likely to have a dynamic programming solution.

**Polynomially many subproblems** The total number of subproblems should be a polynomial, or else DP may not provide an efficient solution.

**Subproblem optimality** If the optimal solution to the entire problem contain optimal solution to the subproblems, then it has the subproblem optimality property. Also called the *principle of optimality*.

# Dynamic Programming highlights

- Dynamic Programming, just like Memoization, avoids computing solutions to overlapping subproblems by saving intermediate results, and thus both require space for the “table”.

# Dynamic Programming highlights

- Dynamic Programming, just like Memoization, avoids computing solutions to overlapping subproblems by saving intermediate results, and thus both require space for the “table”.
- Dynamic Programming is a bottom-up techniques, and finds the solution by starting from the base case(s) and works its way upwards.



# Dynamic Programming highlights

- Dynamic Programming, just like Memoization, avoids computing solutions to overlapping subproblems by saving intermediate results, and thus both require space for the “table”.
- Dynamic Programming is a bottom-up techniques, and finds the solution by starting from the base case(s) and works its way upwards.
- Developing a Dynamic Programming solution often requires some thought into the subproblems, especially how to find the natural order in which to solve the subproblems.

# Dynamic Programming highlights

- Dynamic Programming, just like Memoization, avoids computing solutions to overlapping subproblems by saving intermediate results, and thus both require space for the “table”.
- Dynamic Programming is a bottom-up techniques, and finds the solution by starting from the base case(s) and works its way upwards.
- Developing a Dynamic Programming solution often requires some thought into the subproblems, especially how to find the natural order in which to solve the subproblems.
- Unlike Memoization, which solves only the needed subproblems, DP solves all the subproblems, because it does it bottom-up.

# Dynamic Programming highlights

- Dynamic Programming, just like Memoization, avoids computing solutions to overlapping subproblems by saving intermediate results, and thus both require space for the “table”.
- Dynamic Programming is a bottom-up techniques, and finds the solution by starting from the base case(s) and works its way upwards.
- Developing a Dynamic Programming solution often requires some thought into the subproblems, especially how to find the natural order in which to solve the subproblems.
- Unlike Memoization, which solves only the needed subproblems, DP solves all the subproblems, because it does it bottom-up.
- Dynamic Programming on the other hand may be much more efficient because its iterative, whereas Memoization must pay for the (often significant) overhead due to recursion.

# Conclusion

- Memoization is the top-down technique, and dynamic programming is a bottom-up technique.
- The key to Dynamic programming is in “intelligent” recursion (the hard part), not in filling up the table (the easy part).
- Dynamic Programming has the potential to transform exponential-time brute-force solutions into polynomial-time algorithms.
- Greed does not pay, Dynamic Programming does!