CSE 221: Algorithms

Dynamic Programming

Mumit Khan

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: December 30, 2010

Contents

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

Recursive solution to Fibonacci numbers

Definition (Fibonacci numbers)

The Fibonacci numbers are given by the following sequence:

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

Recursive solution to Fibonacci numbers

Definition (Fibonacci numbers)

The Fibonacci numbers are given by the following sequence:

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

and described by the following recurrence.

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

Recursive solution to Fibonacci numbers

Definition (Fibonacci numbers)

The Fibonacci numbers are given by the following sequence:

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

and described by the following recurrence.

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

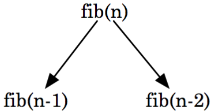
Straightforward recursive algorithm

```
FIBONACCI(n)                ▷ n ≥ 0
if n = 0 or n = 1
   then return n
   else return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

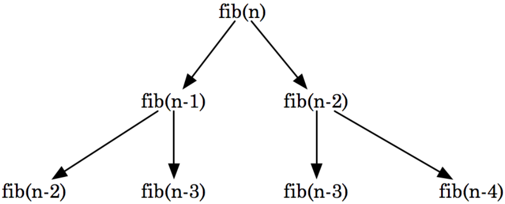
Recursion tree

fib(n)

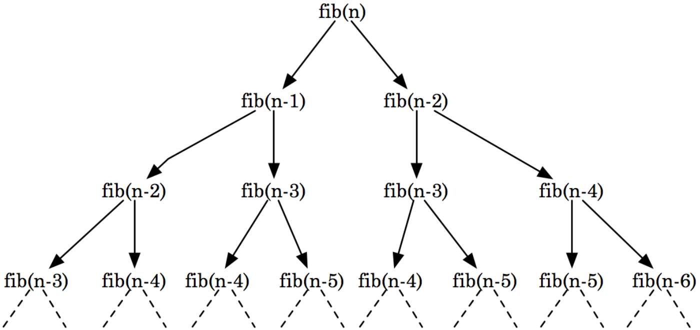
Recursion tree



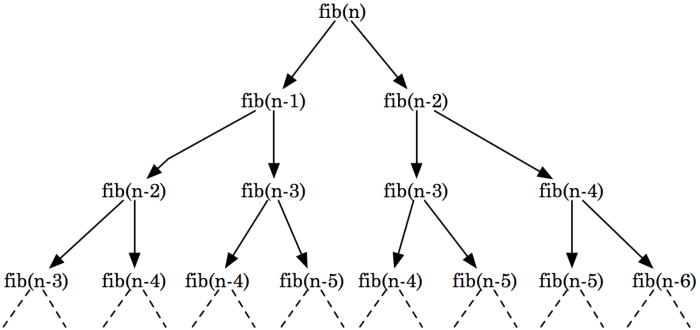
Recursion tree



Recursion tree



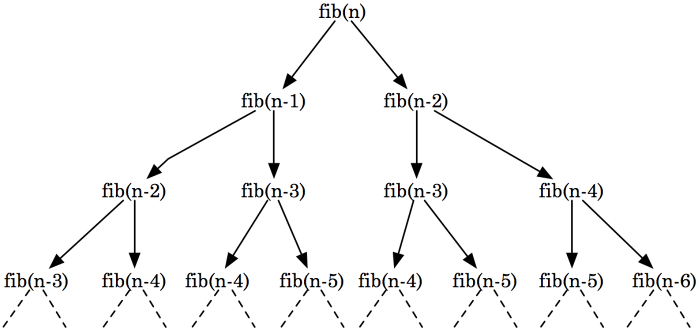
Recursion tree



Complexity

This recursive algorithm for Fibonacci numbers has exponential running time!

Recursion tree

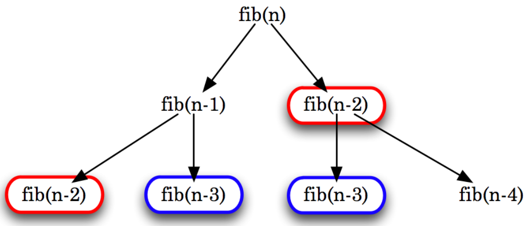


Complexity

This recursive algorithm for Fibonacci numbers has exponential running time!

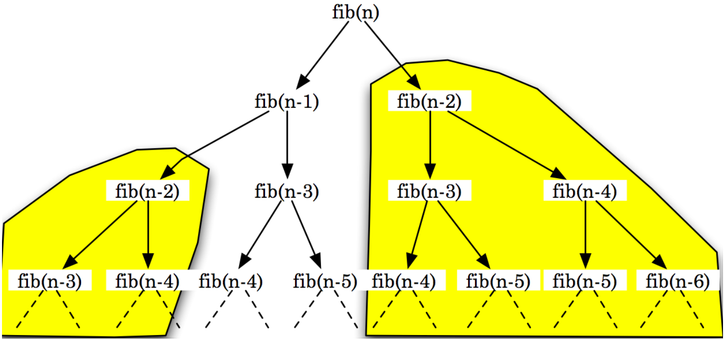
To be precise, $\boxed{T(n) = O(\varphi^n)}$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.

Redundant computations



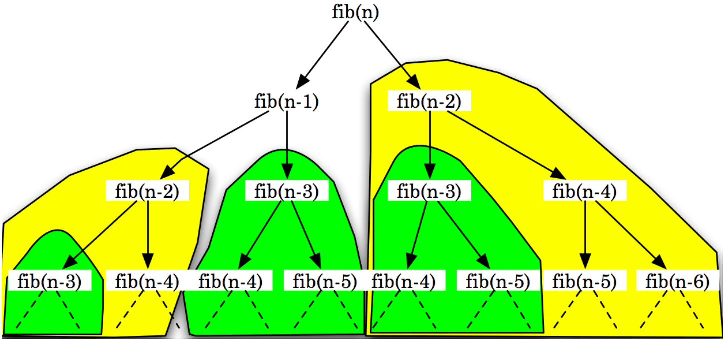
▷ Note how $\textsc{Fib}(n-2)$ and $\textsc{Fib}(n-3)$ are each being computed twice.

Redundant computations



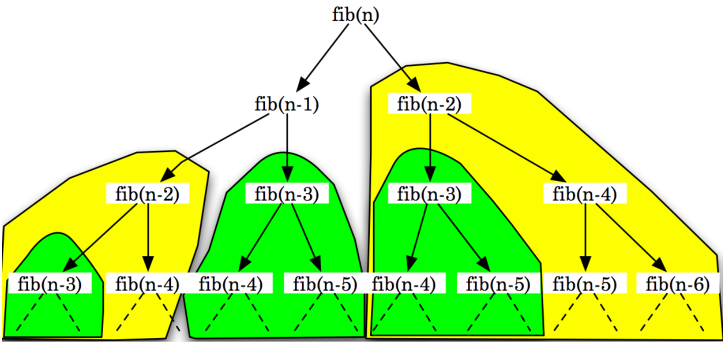
▷ In fact, computing $\text{FIB}(n-2)$ involves computing a whole subtree.

Redundant computations



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

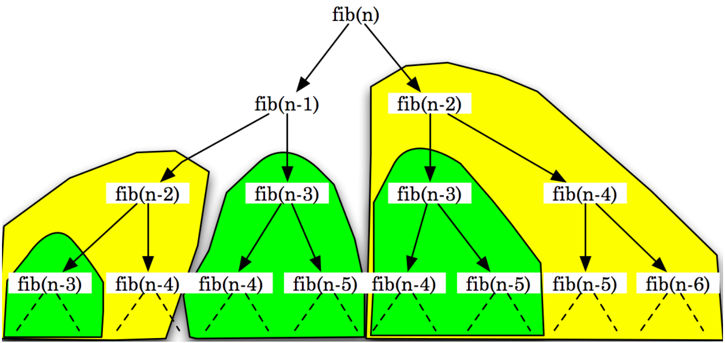
Redundant computations



Observations

- Spectacular redundancy in computation

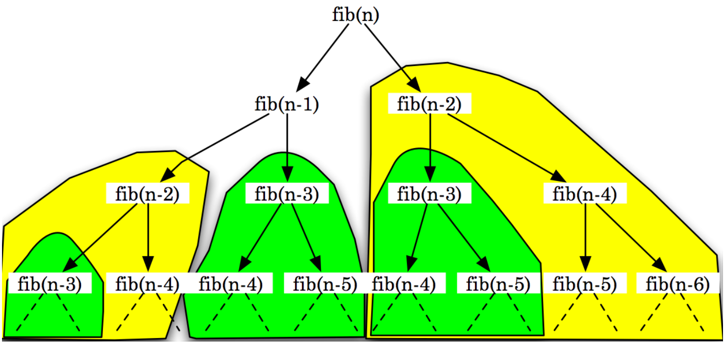
Redundant computations



Observations

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

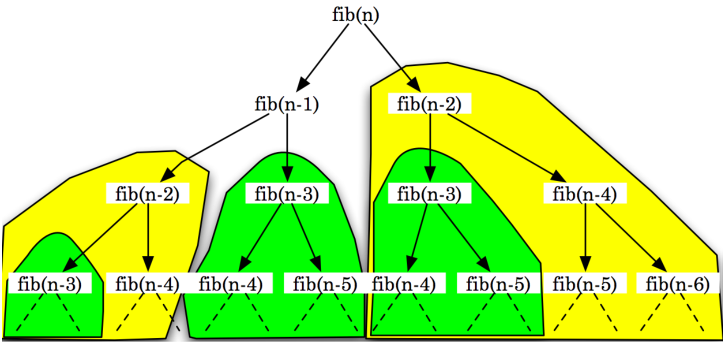
Redundant computations



Observations

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

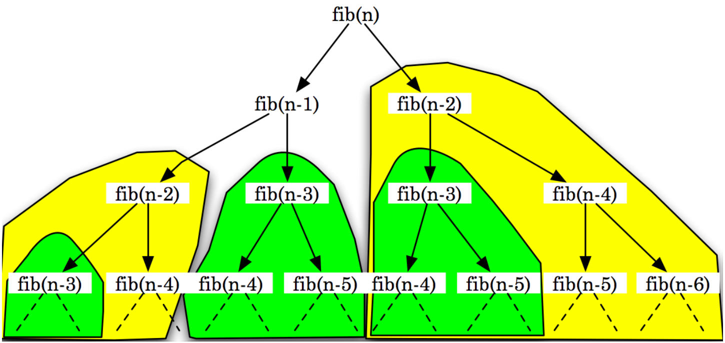
Redundant computations



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 $i = \{2, 3, \ldots, n\}$ 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 $i = \{2, 3, \ldots, n\}$ 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) $\quad \triangleright$ $n \geq 0$, global $F = [0 \,..\, n]$

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

Memoized recursive algorithm for Fibonacci numbers

$\textsc{M-Fibonacci}(n)$ $\triangleright$ $n \geq 0$, global $F = [0\,..\,n]$

```
if n = 0 or n = 1
    then return n                                   ▷ Our base conditions.
if F[n] is empty                          ▷ No saved solution found for n.
    then F[n] ← M-FIBONACCI(n − 1) + M-FIBONACCI(n − 2)
return F[n]
```

Questions

- What is this global array F?

Memoized recursive algorithm for Fibonacci numbers

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

```
if n = 0 or n = 1
    then return n                                ▷ Our base conditions.
if F[n] is empty                                 ▷ No saved solution found for n.
    then F[n] ← M-FIBONACCI(n − 1) + M-FIBONACCI(n − 2)
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)        ▷ n ≥ 0, global F = [0 . . n]
if n = 0 or n = 1
    then return n                                ▷ Our base conditions.
if F[n] is empty                       ▷ No saved solution found for n.
    then F[n] ← M-FIBONACCI(n − 1) + M-FIBONACCI(n − 2)
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) ▷ $n \geq 0$, global $F = [0 \mathinner{..} n]$

```
if n = 0 or n = 1
    then return n                          ▷ Our base conditions.
if F[n] is empty                           ▷ No saved solution found for n.
    then F[n] ← M-FIBONACCI(n − 1) + M-FIBONACCI(n − 2)
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)? Use -1, which is guaranteed to be an invalid value.

Memoized . . . Fibonacci numbers (continued)

```
FIBONACCI(n)            ▷ n ≥ 0
   ▷ Allocate an array F[0 . . n] to save results (LENGTH[F] = n + 1).
   for i ← 0 to n
        do F[i] ← −1            ▷ No solution computed for i yet (sentinel)
   return M-FIBONACCI(F, n)
```

Memoized . . . Fibonacci numbers (continued)

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

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

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

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

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

Memoized . . . Fibonacci numbers (continued)

$\textsc{Fibonacci}(n)$ $\qquad \triangleright\ n \geq 0$

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

for $i \leftarrow 0$ **to** n

    do $F[i] \leftarrow -1$ $\qquad \triangleright$ No solution computed for i yet (sentinel)

return $\textsc{m-fibonacci}(F, n)$

$\textsc{M-Fibonacci}(F, n)$ $\qquad \triangleright\ n \geq 0, F = [0\,..\,n]$

if $n \leq 1$

  then return n

if $F[n] = -1$ $\qquad \triangleright$ No saved solution found for n.

  then $F[n] \leftarrow \textsc{m-fibonacci}(F, n-1) + \textsc{m-fibonacci}(F, n-2)$

return $F[n]$

Running time

Each element $F[2] \ldots F[n]$ is filled in just once in $\Theta(1)$ time, so $\boxed{T(n) = \Theta(n)}$.

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

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) $\triangleright$ $n \geq 0$

$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n
  do $F[i] \leftarrow F[i-1] + F[i-2]$
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

$\textsc{Fibonacci}(n)$ $\quad\quad \triangleright\ n \geq 0$

$F[0] \leftarrow 0$
$F[1] \leftarrow 1$
for $i \leftarrow 2$ **to** n
 do $F[i] \leftarrow F[i-1] + F[i-2]$
return $F[n]$

$$\boxed{T(n) = \Theta(n)}$$

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 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 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 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.

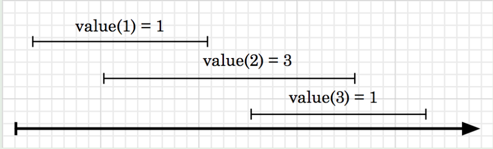
Contents

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)



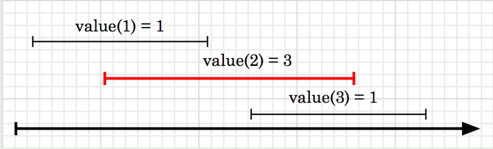
$$|A| = ???, \sum_{i \in A} w_i = ???.$$

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)



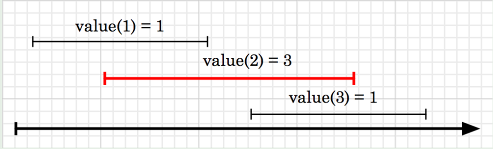
$|A| = 1$, $\sum_{i \in A} w_i = 3$.

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 ϑ 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 ϑ be an optimal solution (even if we have no idea what it is yet).
- All we can say about ϑ is the following: interval n (the last interval) either belongs to ϑ, 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 ϑ be an optimal solution (even if we have no idea what it is yet).
- All we can say about ϑ is the following: interval n (the last interval) either belongs to ϑ, or it doesn't.

  If $n \in \vartheta$ Then clearly all intervals that conflict with n are not members of ϑ. ϑ 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

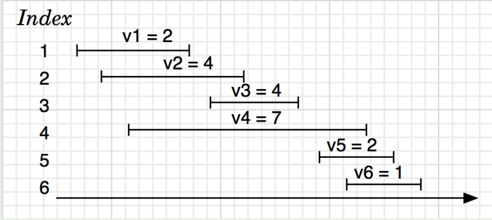
- Let W be an instance of a weighted interval problem.
- As in the greedy approach, we sort the intervals according to finish times such that $f_i \leq f_j$ for $i < j$ ("a natural order of the subproblems").
- Let ϑ be an optimal solution (even if we have no idea what it is yet).
- All we can say about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

  If $n \in \vartheta$ Then clearly all intervals that conflict with n are not members of ϑ. ϑ 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 ϑ contains an optimal solution for the intervals $\{i_1, i_2, \ldots, i_{n-1}\}$.

Developing a recursive solution (continued)

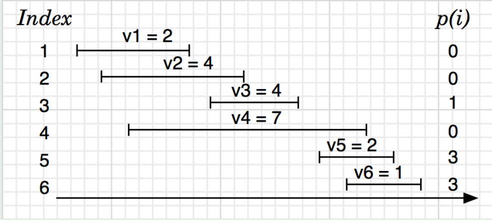
Example (an instance of a weighted interval problem)



▷ For each interval i, compute $p(i)$, the leftmost interval that does not conflict with i. Define $p(j) = 0$ if not request $i < j$ is disjoint from j.

Developing a recursive solution (continued)

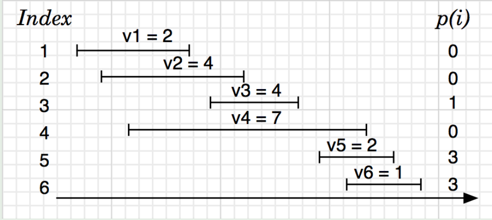
Example (an instance of a weighted interval problem)



▷ For a given interval i, $p(i)$ means that intervals $\{p(i)+1, p(i)+2, \ldots, i-1\}$ 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 $\{1, 2, .., p(i)\}$ *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 ϑ must include, in addition to interval n, an optimal solution to the subproblem consisting of intervals $\{1, 2, \ldots, p(n)\}$.

Developing a recursive solution (continued)

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

Developing a recursive solution (continued)

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

Developing a recursive solution (continued)

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

Developing a recursive solution (continued)

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

Developing a recursive solution (continued)

Recursive algorithm for an optimal value

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

$$\boxed{OPT(j) = \textsc{max}(w_j + OPT(p(j)), OPT(j-1))}$$

Developing a recursive solution (continued)

Recursive algorithm for an optimal value

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

$$\boxed{OPT(j) = \textsc{max}(w_j + OPT(p(j)), OPT(j-1))}$$

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 $\{1, 2, \ldots, j\}$, for any $j \in \{1, 2, \ldots, n\}$, then:

$$\boxed{OPT(j) = \textsc{max}(w_j + OPT(p(j)), OPT(j-1))}$$

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 $\{1, 2, \ldots, j\}$ ***if and only if***

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

A recursive algorithm

$\mathrm{WIS}(j)$

if $j = 0$

  then return 0

  else return $\textsc{max}(w_j + \mathrm{WIS}(p(j)),$
$\qquad\qquad\mathrm{WIS}(j-1))$

A recursive algorithm

$\text{WIS}(j)$

```
if j = 0
    then return 0
    else return MAX(w_j + WIS(p(j)),
                    WIS(j − 1))
```

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

A recursive algorithm

$\text{WIS}(j)$

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

- The initial call is $\text{WIS}(n)$ for intervals $\{1, 2, \ldots, n\}$ 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

$\text{WIS}(j)$

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

- The initial call is $\text{WIS}(n)$ for intervals $\{1, 2, \ldots, n\}$ 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)
if j = 0
    then return 0
elseif M[j] is empty
    then M[j] ← MAX(w_j + M-WIS(p(j)),
                      M-WIS(j − 1))
return M[j]
```

Memoizing the recursion

```
M-WIS(j)
if j = 0
    then return 0
elseif M[j] is empty
    then M[j] ← MAX(w_j + M-WIS(p(j)),
                    M-WIS(j − 1))
return M[j]
```

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

Memoizing the recursion

```
M-WIS(j)
if j = 0
    then return 0
elseif M[j] is empty
    then M[j] ← MAX(w_j + M-WIS(p(j)),
                        M-WIS(j − 1))
return M[j]
```

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

Memoizing the recursion

M-WIS(j)

if $j = 0$
  then return 0
elseif $M[j]$ is empty
  then $M[j] \leftarrow \textsc{max}(w_j + \text{M-WIS}(p(j)),$
$\text{M-WIS}(j-1))$
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 $\boxed{\Theta(n)}$ 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 ϑ 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 $\text{WIS-FIND-SOLUTION}(n)$ 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 $1 \le j \le n$.
- Calling $\text{WIS-FIND-SOLUTION}(n)$ extracts all the intervals in the optimal solution.

$\text{WIS-FIND-SOLUTION}(j)$

if $j = 0$
  then Output nothing
  else
    if $w_j + M[p(j)] \ge M[j-1]$
      then Output j
        $\text{WIS-FIND-SOLUTION}(p(j))$
      else $\text{WIS-FIND-SOLUTION}(j-1)$

Computing a solution in addition to its values (continued)

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

$\textsc{M-WIS}(j)$

if $j = 0$
    then return 0
elseif $M[j]$ is empty
    then if $w_j + \textsc{M-WIS}(p(j)) > \textsc{M-WIS}(j-1))$
        then $M[j] \leftarrow w_j + \textsc{M-WIS}(p(j)$
            $pred[j] \leftarrow p(j)$
        else $M[j] \leftarrow \textsc{M-WIS}(j-1)$
            $pred[j] \leftarrow j-1$
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)

```
if j = 0
    then Output nothing
    else
        if pred[j] = p(j)
            then Output j
                 WIS-FIND-SOLUTION(p(j))
            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)

```
if j = 0
    then Output nothing
    else
        if pred[j] = p(j)
            then Output j
                 WIS-FIND-SOLUTION(p(j))
            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 $j \in \{1, 2, 3, \ldots, n\}$ 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 $j \in \{1, 2, 3, \ldots, n\}$ 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 $j \in \{1, 2, 3, \ldots, n\}$ 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: $M[j] = \textsc{max}(w_j + M[p(j)], M[j-1])$.

Dynamic programming algorithm

$\textrm{WIS}(n)$

$M[0] \leftarrow 0$
for $j \leftarrow 1$ **to** n
 do $M[j] = \textsc{max}(w_j + M[p(j)], M[j-1])$
return $M[n]$

Developing a Dynamic Programming algorithm

- The value of an optimal solution $OPT(j)$ for any $j \in \{1, 2, 3, \ldots, n\}$ 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)

```
M[0] ← 0
for j ← 1 to n
        do M[j] = MAX(w_j + M[p(j)], M[j − 1])
return M[n]
```

$\boxed{T(n) = \Theta(n)}$

Computing a solution in addition to its values

WIS(n)

```
M[0] ← 0
for j ← 1 to n
        do if w_j + M[p(j) > M[j − 1]
              then M[j] = w_j + M[p(j)]
                   pred[j] = p(j)
              else M[j] = M[j − 1]
                   pred[j] = j − 1
return M[n]
```


Computing a solution in addition to its values

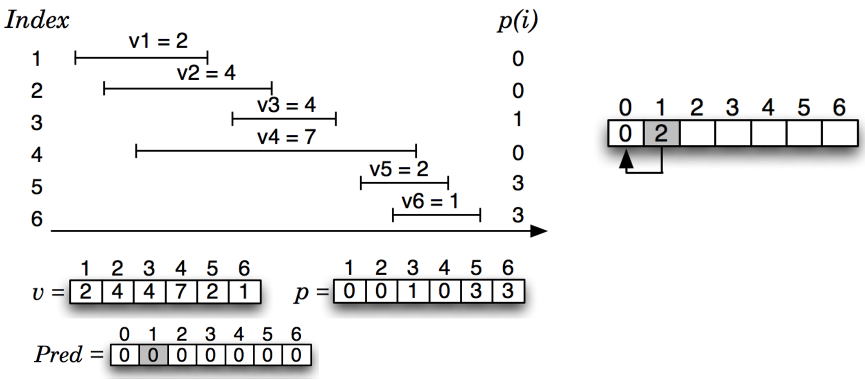
WIS(n)

```
M[0] ← 0
for j ← 1 to n
      do if w_j + M[p(j) > M[j − 1]
            then M[j] = w_j + M[p(j)]
                 pred[j] = p(j)
            else M[j] = M[j − 1]
                 pred[j] = j − 1
return M[n]
```

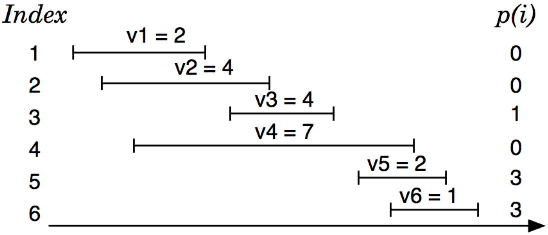
WIS-FIND-SOLUTION(j)

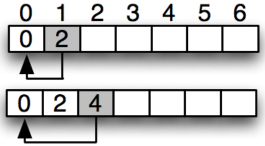
```
j ← n
while j > 0
      do if pred[j] = p(j)
            then Output j
         j ← pred[j]
```

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

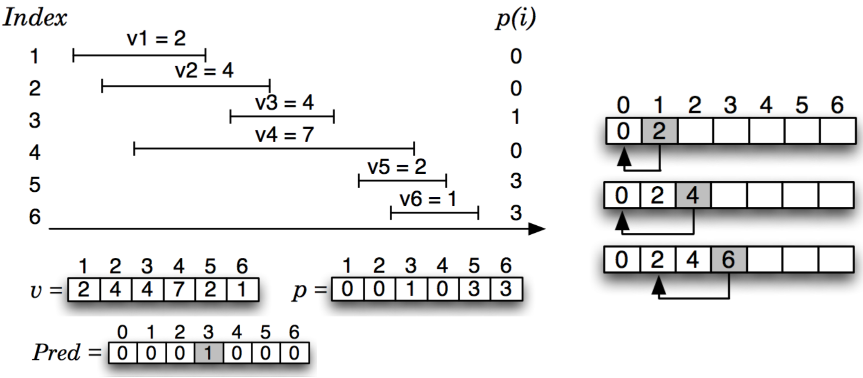
$p =$

1	2	3	4	5	6
0	0	1	0	3	3

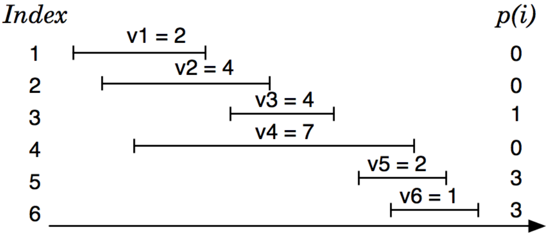
$Pred =$

0	1	2	3	4	5	6
0	0	0	0	0	0	0

Weighted Interval Scheduling DP algorithm in action



Weighted Interval Scheduling DP algorithm in action



$v =$

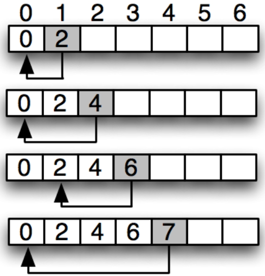
1	2	3	4	5	6
2	4	4	7	2	1

$p =$

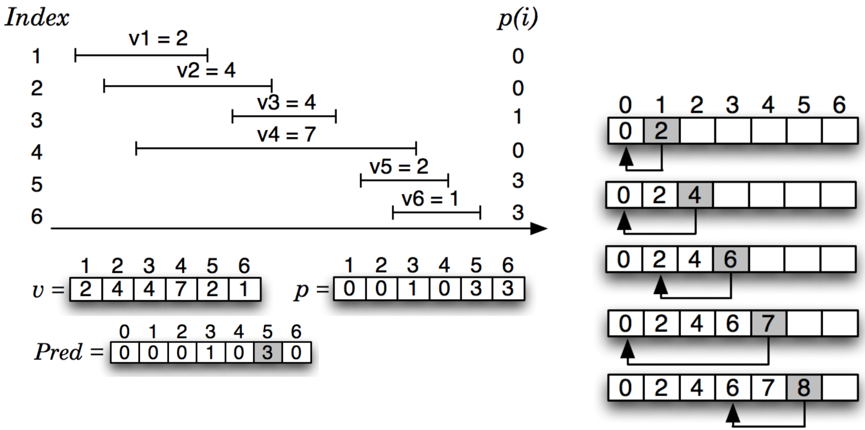
1	2	3	4	5	6
0	0	1	0	3	3

$Pred =$

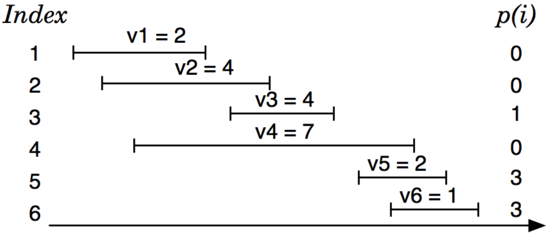
0	1	2	3	4	5	6
0	0	0	1	0	0	0



Weighted Interval Scheduling DP algorithm in action



Weighted Interval Scheduling DP algorithm in action



$v =$

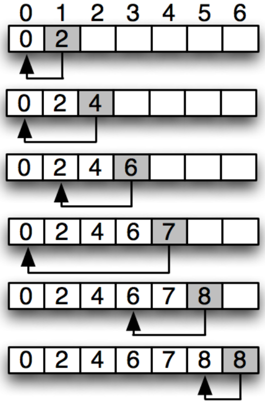
1	2	3	4	5	6
2	4	4	7	2	1

$p =$

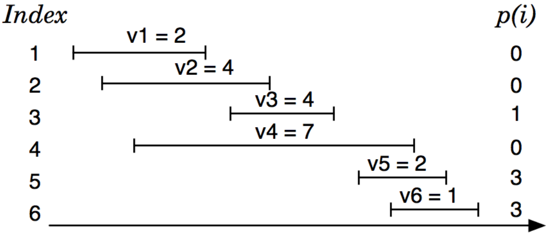
1	2	3	4	5	6
0	0	1	0	3	3

$Pred =$

0	1	2	3	4	5	6
0	0	0	1	0	3	5



Weighted Interval Scheduling DP algorithm in action



$v =$

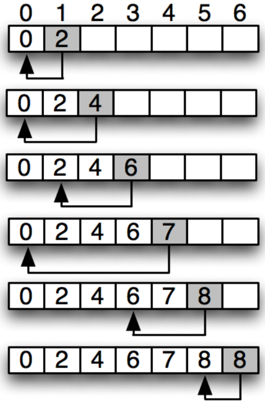
1	2	3	4	5	6
2	4	4	7	2	1

$p =$

1	2	3	4	5	6
0	0	1	0	3	3

$Pred =$

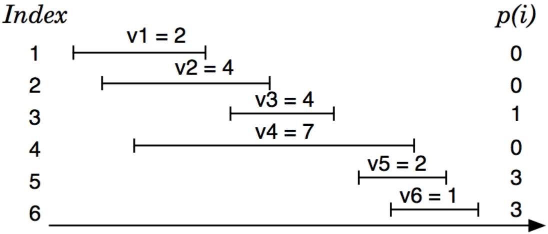
0	1	2	3	4	5	6
0	0	0	1	0	3	5



Optimal value: $\boxed{8}$

Optimal solution: $\boxed{\{5, 3, 1\}}$

Weighted Interval Scheduling DP algorithm in action



$v =$

1	2	3	4	5	6
2	4	4	7	2	1

$p =$

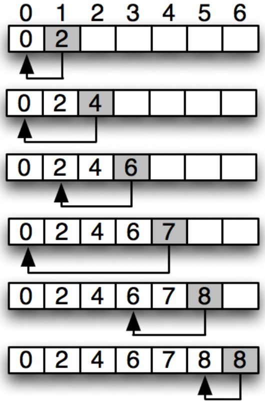
1	2	3	4	5	6
0	0	1	0	3	3

$Pred =$

0	1	2	3	4	5	6
0	0	0	1	0	3	5

Optimal value: $\boxed{8}$

Optimal solution: $\boxed{\{1, 3, 5\}}$



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

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

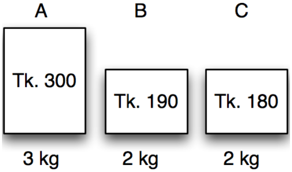
Definition (0/1 knapsack problem)

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 $\sum_{i \in S} w_i \leq W$.

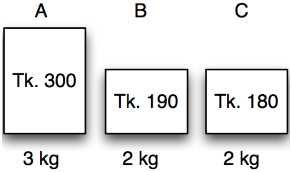


Maximum weight: $\boxed{W = 4 \text{ kg}}$

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: $\boxed{W = 4 \text{ kg}}$

Optimal solution: items B and C Benefit: $\boxed{370}$

Developing a recursive solution

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

Developing a recursive solution

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

Developing a recursive solution

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

Developing a recursive solution

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

- $w_n > W \implies n \notin \vartheta$.
  $\triangleright$ $\vartheta(n, W) = \vartheta(n-1, W)$
- 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 $\{1, 2, \ldots, n\}$:
    $\triangleright$ $\vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$

Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$.
  $\triangleright$ $\vartheta(n, W) = \vartheta(n-1, W)$
- 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 $\{1, 2, \ldots, n\}$:
    $\triangleright$ $\vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$
  - If $n \notin \vartheta$, then $\vartheta(n, W)$ simply contains an optimal solution to the subproblem consisting of the intervals $\{1, 2, \ldots, n-1\}$:
    $\triangleright$ $\vartheta(n, W) = \vartheta(n-1, W)$

Developing a recursive solution (continued)

- $w_n > W \implies n \notin \vartheta$.
  $\triangleright\ \vartheta(n, W) = \vartheta(n-1, W)$
- 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 $\{1, 2, \ldots, n\}$:
    $\triangleright\ \vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$
  - If $n \notin \vartheta$, then $\vartheta(n, W)$ simply contains an optimal solution to the subproblem consisting of the intervals $\{1, 2, \ldots, n-1\}$:
    $\triangleright\ \vartheta(n, W) = \vartheta(n-1, W)$
  - 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)

- $w_n > W \implies n \notin \vartheta$.
  $\triangleright$ $\vartheta(n, W) = \vartheta(n-1, W)$
- 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 $\{1, 2, \ldots, n\}$:
    $\triangleright$ $\vartheta(n, W) = v_n + \vartheta(n-1, W - w_n)$
  - If $n \notin \vartheta$, then $\vartheta(n, W)$ simply contains an optimal solution to the subproblem consisting of the intervals $\{1, 2, \ldots, n-1\}$:
    $\triangleright$ $\vartheta(n, W) = \vartheta(n-1, W)$
  - Since an optimal solution must maximize the sum of the weights in the intervals it contains, we accept the larger of the two.
    $\triangleright$ $\boxed{\vartheta(n, W) = \textsc{max}(v_n + \vartheta(n-1, W - w_n), \vartheta(n-1, W))}$

Developing a recursive solution (continued)

Recursive algorithm for an optimal value

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

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

Developing a recursive solution (continued)

Recursive algorithm for an optimal value

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

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

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.

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

A recursive algorithm

$\textsc{Knapsack}(j, w)$

```
if j = 0 or w = 0
    then return 0
elseif w_j > w
    then return KNAPSACK(j − 1, w))
else return MAX(v_j + KNAPSACK(j − 1, w − w_j),
                     KNAPSACK(j − 1, w))
```

A recursive algorithm

$\textsc{Knapsack}(j, w)$

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

- The initial call is $\textsc{Knapsack}(n, W)$.

A recursive algorithm

$\textsc{Knapsack}(j, w)$

if $j = 0$ or $w = 0$
  then return 0
elseif $w_j > w$
  then return $\textsc{Knapsack}(j - 1, w))$
else return $\textsc{max}(v_j + \textsc{Knapsack}(j - 1, w - w_j),$
  $\textsc{Knapsack}(j - 1, w))$

- The initial call is $\textsc{Knapsack}(n, W)$.
- The tree grows very rapidly, leading to exponential running time.

A recursive algorithm

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

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

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

M-KNAPSACK(j, w)

if $j = 0$ or $w = 0$
    then return 0
elseif $M[j, w]$ is empty
    then $M[j, w] \leftarrow$ MAX(v_j + M-KNAPSACK($j - 1, w - w_j$),
        M-KNAPSACK($j - 1, w$))
return $M[j, w]$

Memoizing the recursion

```
M-KNAPSACK(j, w)
if j = 0 or w = 0
    then return 0
elseif M[j, w] is empty
    then M[j, w] ← MAX(v_j + M-KNAPSACK(j − 1, w − w_j),
                       M-KNAPSACK(j − 1, w))
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 $\textsc{M-Knapsack}(n, W)$ takes $\boxed{\Theta(nW)}$ time.

Memoizing the recursion

```
M-KNAPSACK(j, w)
if j = 0 or w = 0
    then return 0
elseif M[j, w] is empty
    then M[j, w] ← MAX(v_j + M-KNAPSACK(j − 1, w − w_j),
                       M-KNAPSACK(j − 1, w))
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 $\textsc{M-Knapsack}(n, W)$ takes $\boxed{\Theta(nW)}$ time.
- Is this a linear-time algorithm?

Memoizing the recursion

$\textsc{M-Knapsack}(j, w)$

if $j = 0$ or $w = 0$
  then return 0
elseif $M[j, w]$ is empty
  then $M[j, w] \leftarrow \textsc{max}(v_j + \textsc{M-Knapsack}(j-1, w-w_j),$
    $\textsc{M-Knapsack}(j-1, w))$
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 $\textsc{M-Knapsack}(n, W)$ takes $\boxed{\Theta(nW)}$ 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)
for i ← 0 to n                ▷ no remaining capacity
      do M[i, 0] ← 0
for w ← 0 to W                ▷ no item to choose from
      do M[0, w] ← 0
for j ← 1 to n
      do for w ← 1 to W
            do if w_j > w
                  then M[j,w] = M[j − 1, w]
                  else  M[j, w] ← MAX(v_j + M[j − 1, w − w_j],
                                      M[j − 1, w])
return M[n, W]
```

0/1 Knapsack recursive algorithm in action

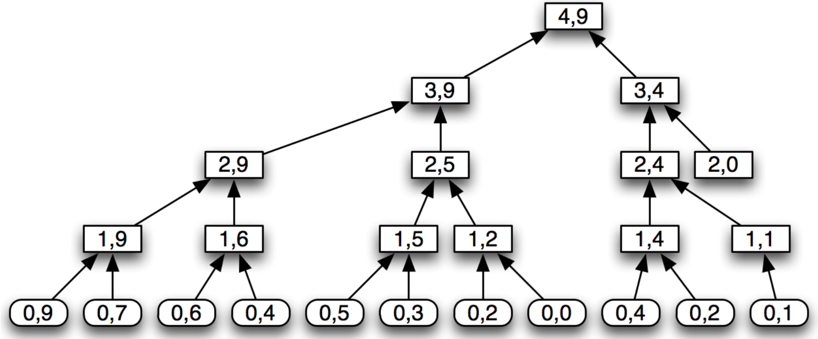
Given the following (from M. H. Alsuwaiyel, ex. 7.6):

$$W = 9$$
$$w_i = \{2, 3, 4, 5\}$$
$$v_i = \{3, 4, 5, 7\}$$

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	5	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

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 $C = \{c_1, c_2, \ldots, c_k\}$, then:

Developing a recursive solution (continued)

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

$\textsc{Change}(n, C)$

if $n = 0$
  then return 0
  else $min \leftarrow \infty$
    for $i \leftarrow 1$ **to** $|C|$
      do if $c_i \leq n$ and $1 + \textsc{Change}(n - c_i, C) < min$
        then $min \leftarrow 1 + \textsc{Change}(n - c_i, C)$

A recursive algorithm

$\textsc{Change}(n, C)$

if $n = 0$
  then return 0
  else $min \leftarrow \infty$
    for $i \leftarrow 1$ **to** $|C|$
      do if $c_i \leq n$ and $1 + \textsc{Change}(n - c_i, C) < min$
        then $min \leftarrow 1 + \textsc{Change}(n - c_i, C)$

- The initial call is $\textsc{Change}(A, C)$.

A recursive algorithm

$\textsc{Change}(n, C)$

if $n = 0$
  then return 0
  else $min \leftarrow \infty$
    for $i \leftarrow 1$ **to** $|C|$
      do if $c_i \leq n$ and $1 + \textsc{Change}(n - c_i, C) < min$
        then $min \leftarrow 1 + \textsc{Change}(n - c_i, C)$

- The initial call is $\textsc{Change}(A, C)$.
- The tree grows very rapidly, leading to exponential running time.

A recursive algorithm

$\textsc{Change}(n, C)$

if $n = 0$
  then return 0
  else $min \leftarrow \infty$
    for $i \leftarrow 1$ **to** $|C|$
      do if $c_i \leq n$ and $1 + \textsc{Change}(n - c_i, C) < min$
        then $min \leftarrow 1 + \textsc{Change}(n - c_i, C)$

- The initial call is $\textsc{Change}(A, C)$.
- 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)

if n = 0
   then return 0
   else  if M[n] is empty
            then min ← ∞
                 for i ← 1 to |C|
                     do if c_i ≤ n and
                           1 + M-Change(n − c_i, C) < min
                          then min ← 1 + M-Change(n − c_i, C)
                 M[n] ← min
         return M[n]
```

Memoizing the recursion

```
M-CHANGE(n, C)
if n = 0
   then return 0
   else if M[n] is empty
           then min ← ∞
                for i ← 1 to |C|
                    do if c_i ≤ n and
                          1 + M-CHANGE(n − c_i, C) < min
                         then min ← 1 + M-CHANGE(n − c_i, C)
                M[n] ← min
        return M[n]
```

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

Memoizing the recursion

M-CHANGE(n, C)

```
if n = 0
   then return 0
   else if M[n] is empty
           then min ← ∞
                for i ← 1 to |C|
                    do if c_i ≤ n and
                          1 + M-CHANGE(n − c_i, C) < min
                         then min ← 1 + M-CHANGE(n − c_i, C)
                M[n] ← min
        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 $\boxed{\Theta(n|C|)}$ time.
- Another pseudo-polynomial problem!

Developing a Dynamic Programming algorithm

```
CHANGE(n, C)
   ▷ M = [0 . . n], S = [0 . . n]
   M[0] ← 0          no amount to change
   for p ← 1 to n
       do min ← ∞
          for i ← 1 to |C|
              do if c_i ≤ p and 1 + M[p − c_i] < min
                    then min ← 1 + M[p − c_i]
                         coin ← i
          M[p] ← min
          S[p] ← coin
   return M and S
```


Developing a Dynamic Programming algorithm

```
CHANGE(n, C)
    ▷ M = [0 . . n], S = [0 . . n]
    M[0] ← 0        no amount to change
    for p ← 1 to n
        do min ← ∞
           for i ← 1 to |C|
               do if c_i ≤ p and 1 + M[p − c_i] < min
                     then min ← 1 + M[p − c_i]
                          coin ← i
           M[p] ← min
           S[p] ← coin
    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 paise.

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)

```
while n > 0
      do Output C[S[n]]
         n ← n − C_{S[n]}
```

Contents

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!