Algorithms: Recall Binary Search Trees and a Dynamic Programming

Theophile Thiery

EPFL School of Computer and Communication Sciences

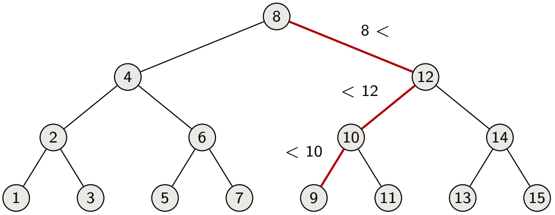
Lecture 9, 18.03.2025

RECALL BINARY SEARCH TREES

Binary Search Trees

Key property:

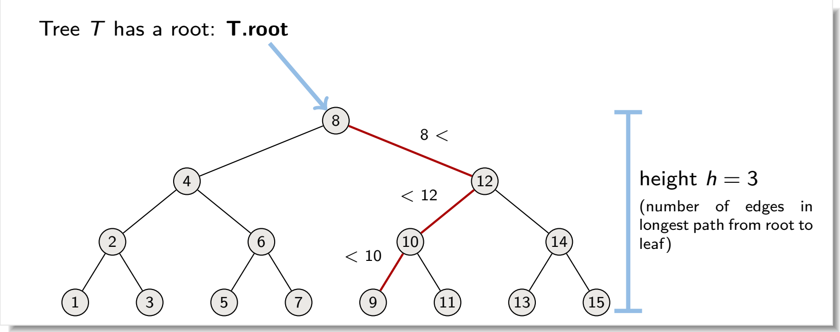
- If y is in the left subtree of x then $y.key < x.key$
- If y is in the right subtree of x then $y.key \geq x.key$



Binary Search Trees

Key property:

- If y is in the left subtree of x then $y.key < x.key$
- If y is in the right subtree of x then $y.key \geq x.key$

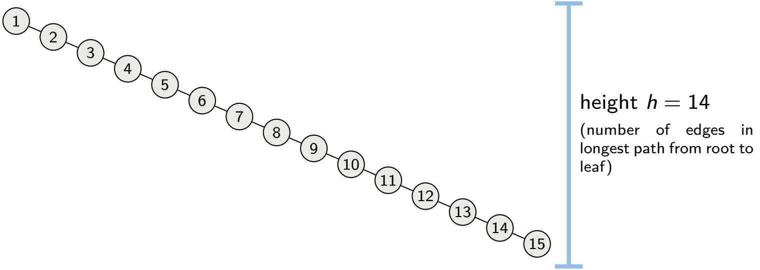


Binary Search Trees

Encodes a strategy whatever number we look for

Key property:

- If y is in the left subtree of x then $y.key < x.key$
- If y is in the right subtree of x then $y.key \geq x.key$

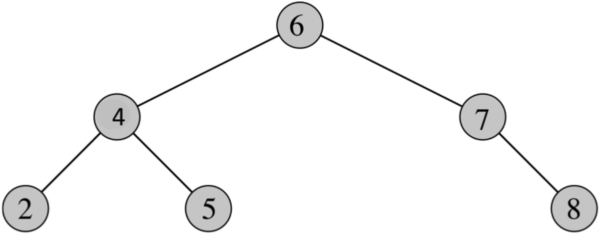


Basic operations take time proportional to height: $O(h)$

QUERYING A BINARY SEARCH TREE

(Searching, Minimum, Maximum, Successor, Predecessor)

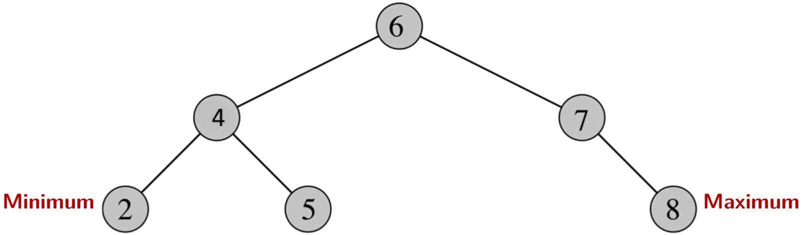
Searching



What is the running time? $O(h)$

```
TREE-SEARCH(x, k)
  if x == NIL or k == key[x]
      return x
  if k < x.key
      return TREE-SEARCH(x.left, k)
  else return TREE-SEARCH(x.right, k)
```

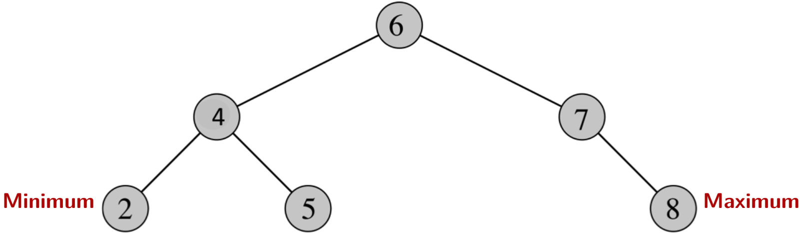
Minimum and Maximum



By key property:

- Minimum is located in leftmost node
- Maximum is located in rightmost node

Minimum and Maximum

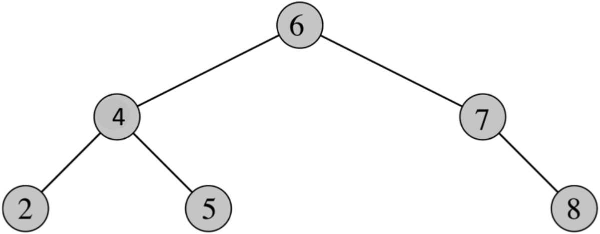


What is the running time? $O(h)$

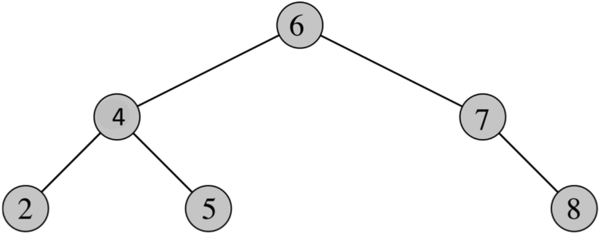
```
TREE-MINIMUM(x)
  while x.left ≠ NIL
      x = x.left
  return x
```

```
TREE-MAXIMUM(x)
  while x.right ≠ NIL
      x = x.right
  return x
```

Successor



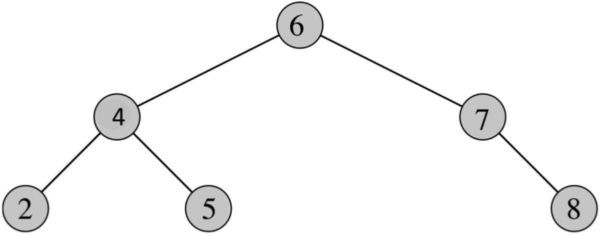
Successor



Successor of a node x is the node y such that $y.key$ is the

$$\text{"smallest key"} > x.key$$

Successor

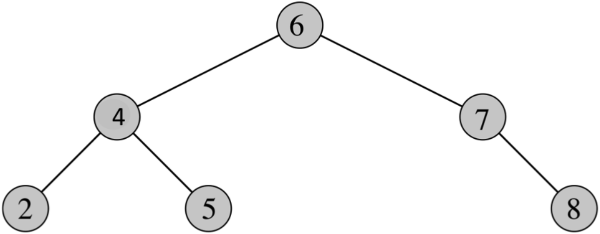


Successor of a node x is the node y such that $y.key$ is the

$$\text{“smallest key”} > x.key$$

- What is the successor of 6?

Successor



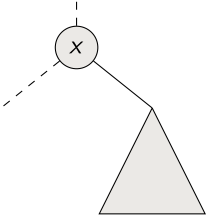
Successor of a node x is the node y such that $y.key$ is the

$$\text{“smallest key”} > x.key$$

- What is the successor of 6?
- What is the successor of 5?

Two cases when finding successor of x:

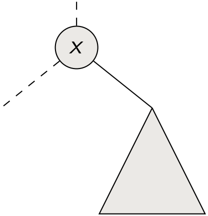
Case 1: x has a non-empty right subtree



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

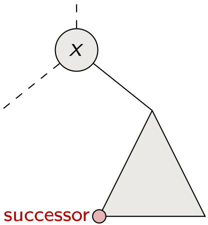
x's successor is the minimum in the right subtree



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

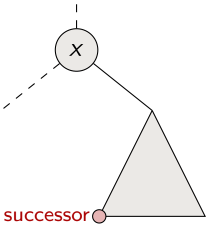
x's successor is the minimum in the right subtree



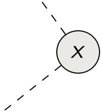
Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

x's successor is the minimum in the right subtree



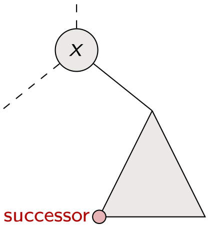
Case 2: x has an empty right subtree



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

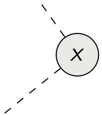
x's successor is the minimum in the right subtree



Case 2: x has an empty right subtree

As long as we go to the left up the tree we're visiting smaller keys

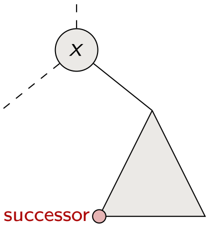
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

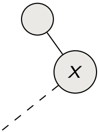
x's successor is the minimum in the right subtree



Case 2: x has an empty right subtree

As long as we go to the left up the tree we're visiting smaller keys

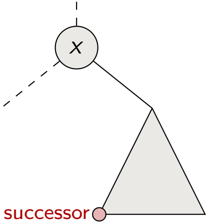
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

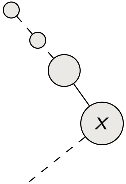
x's successor is the minimum in the right subtree



Case 2: x has an empty right subtree

As long as we go to the left up the tree we're visiting smaller keys

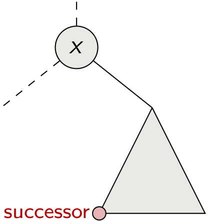
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

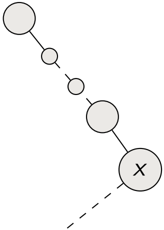
x's successor is the minimum in the right subtree



Case 2: x has an empty right subtree

As long as we go to the left up the tree we're visiting smaller keys

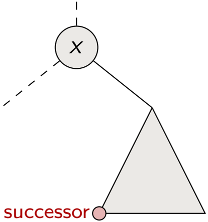
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Two cases when finding successor of x:

Case 1: x has a non-empty right subtree

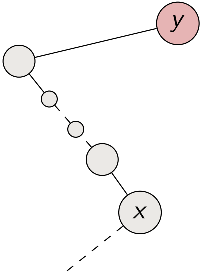
x's successor is the minimum in the right subtree



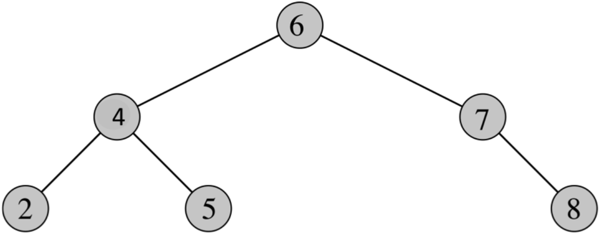
Case 2: x has an empty right subtree

As long as we go to the left up the tree we're visiting smaller keys

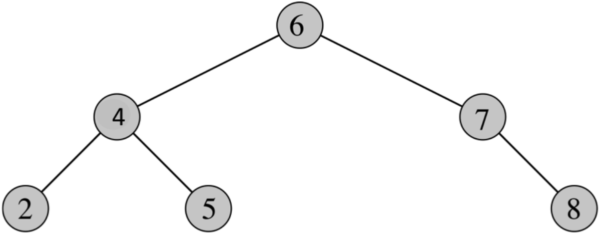
x's successor is y is the node that x is the predecessor of (x is the maximum in y's left subtree)



Successor (Predecessor is symmetric)

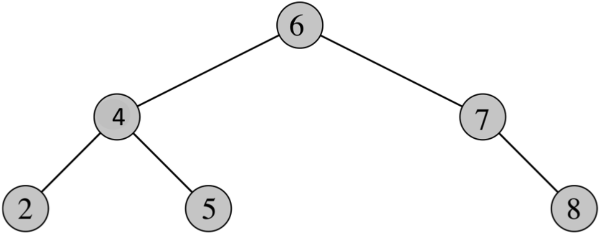


Successor (Predecessor is symmetric)




```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

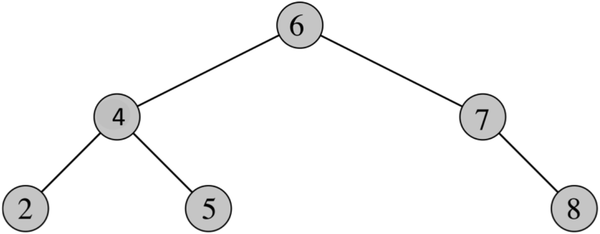
Successor (Predecessor is symmetric)



What is the running time?

```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

Successor (Predecessor is symmetric)



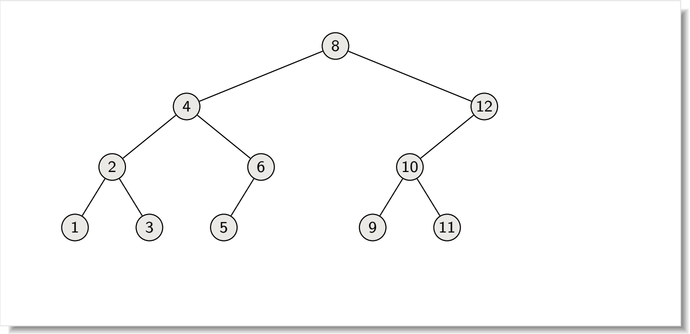
What is the running time? $O(h)$

```
TREE-SUCCESSOR(x)
  if x.right ≠ NIL
      return TREE-MINIMUM(x.right)
  y = x.p
  while y ≠ NIL and x == y.right
      x = y
      y = y.p
  return y
```

PRINTING A BINARY SEARCH TREE

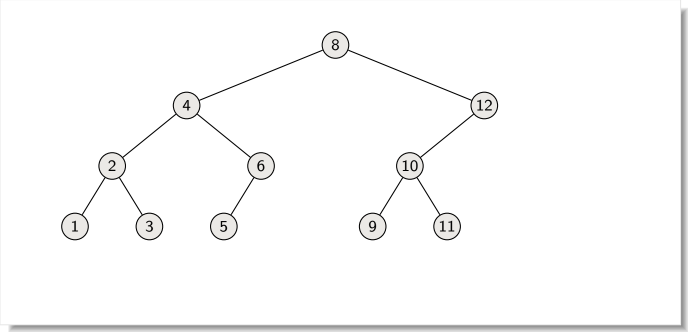
(Inorder, Preorder, Postorder)

Printing Inorder (Idea)



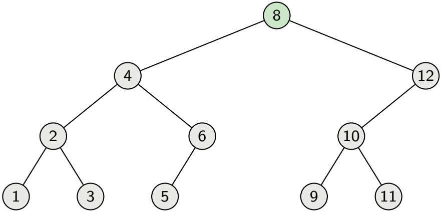
Printing Inorder (Idea)

- Print left subtree recursively
- Print root
- Print right subtree recursively



Printing Inorder (Idea)

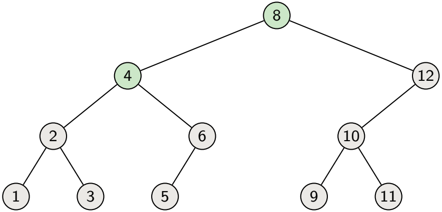
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output:

Printing Inorder (Idea)

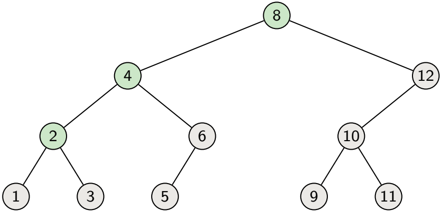
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output:

Printing Inorder (Idea)

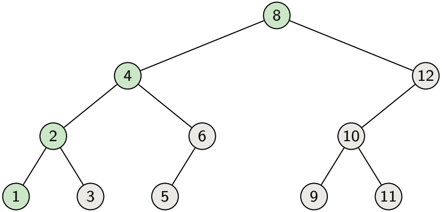
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output:

Printing Inorder (Idea)

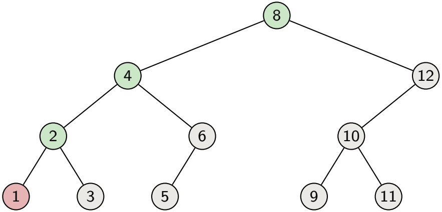
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output:

Printing Inorder (Idea)

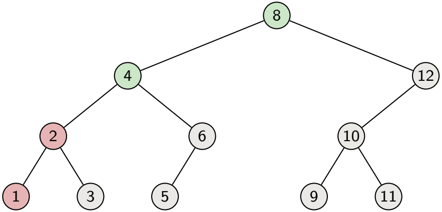
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,

Printing Inorder (Idea)

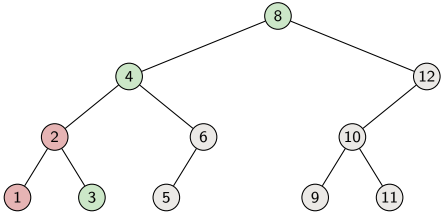
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,

Printing Inorder (Idea)

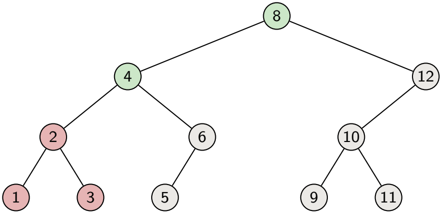
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,

Printing Inorder (Idea)

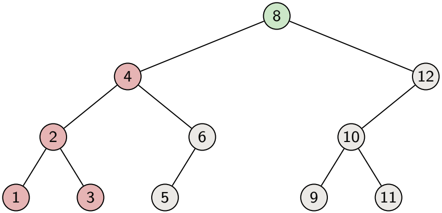
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,

Printing Inorder (Idea)

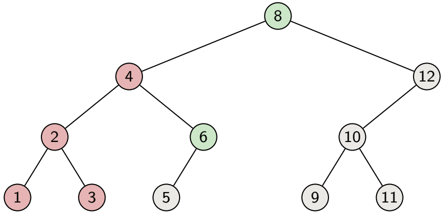
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,

Printing Inorder (Idea)

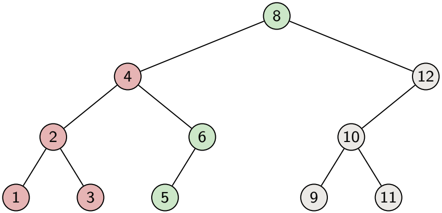
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,

Printing Inorder (Idea)

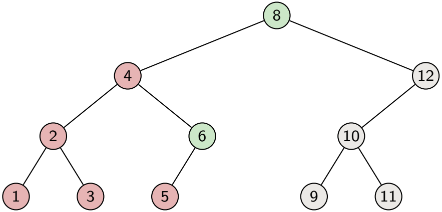
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,

Printing Inorder (Idea)

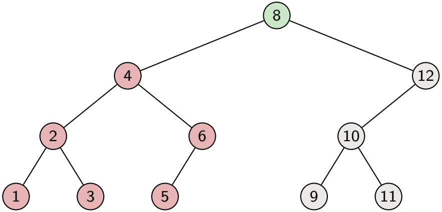
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,

Printing Inorder (Idea)

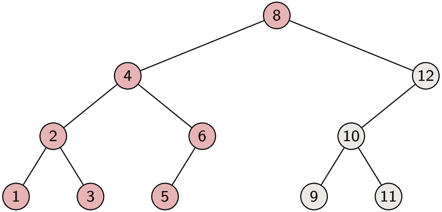
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,

Printing Inorder (Idea)

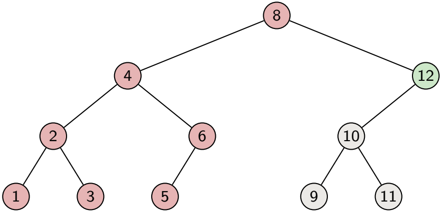
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,

Printing Inorder (Idea)

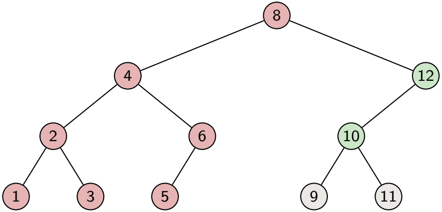
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,

Printing Inorder (Idea)

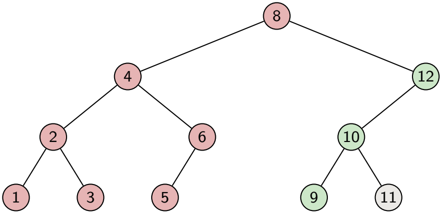
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,

Printing Inorder (Idea)

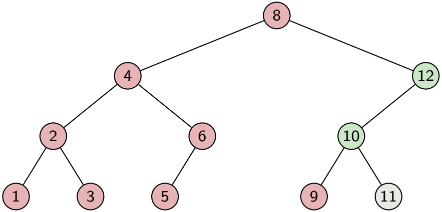
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,

Printing Inorder (Idea)

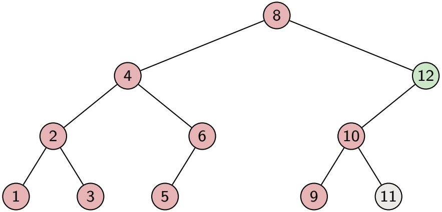
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,

Printing Inorder (Idea)

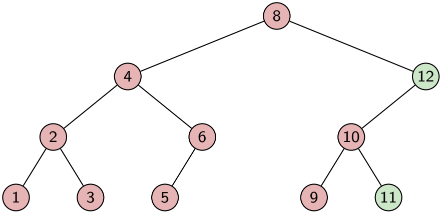
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,10,

Printing Inorder (Idea)

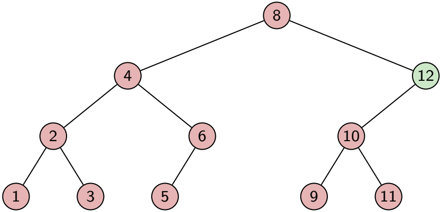
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,10,

Printing Inorder (Idea)

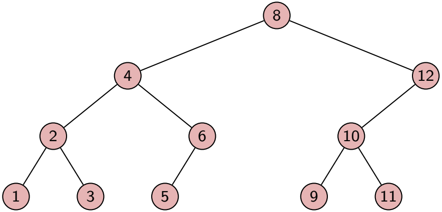
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,10,11,

Printing Inorder (Idea)

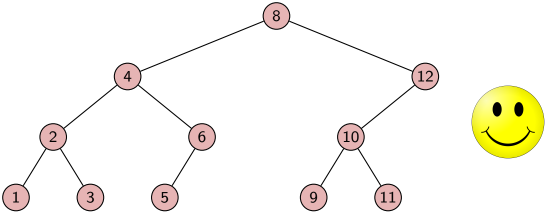
- Print left subtree recursively
- Print root
- Print right subtree recursively



Output: 1,2,3,4,5,6,8,9,10,11,12

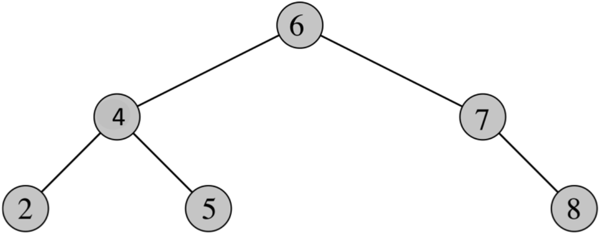
Printing Inorder (Idea)

- Print left subtree recursively
- Print root
- Print right subtree recursively



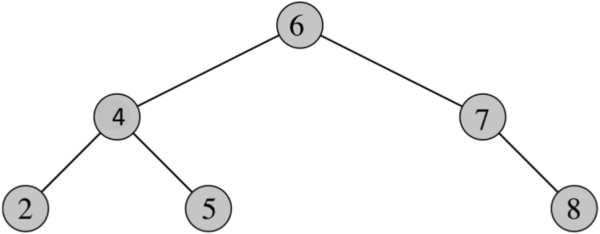
Output: 1,2,3,4,5,6,8,9,10,11,12

Inorder tree walk




```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

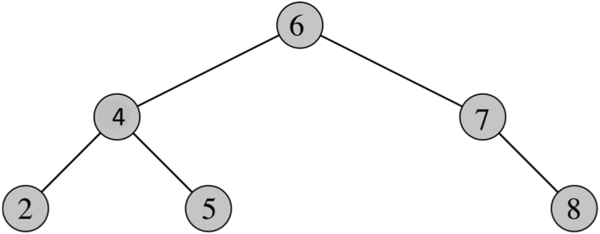
Inorder tree walk



What is the running time?

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

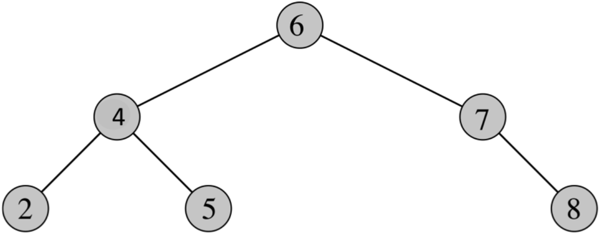
Inorder tree walk



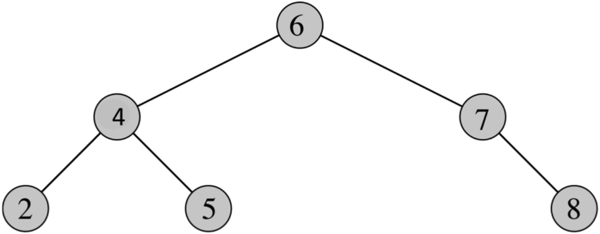
What is the running time? $\Theta(n)$

```
INORDER-TREE-WALK(x)
  if x ≠ NIL
      INORDER-TREE-WALK(x.left)
      print key[x]
      INORDER-TREE-WALK(x.right)
```

Printing Preorder and Postorder



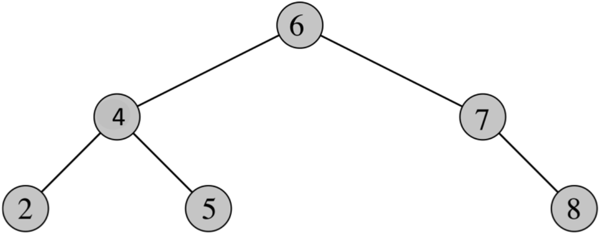
Printing Preorder and Postorder



PREORDER-TREE-WALK(x)

1. **if** $x \neq NIL$
2. print $key[x]$
3. PREORDER-TREE-WALK($x.left$)
4. PREORDER-TREE-WALK($x.right$)

Printing Preorder and Postorder



PREORDER-TREE-WALK(x)

1. **if** $x \neq NIL$
2. print $key[x]$
3. PREORDER-TREE-WALK($x.left$)
4. PREORDER-TREE-WALK($x.right$)

POSTORDER-TREE-WALK(x)

1. **if** $x \neq NIL$
2. POSTORDER-TREE-WALK($x.left$)
3. POSTORDER-TREE-WALK($x.right$)
4. print $key[x]$

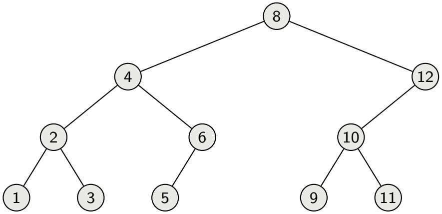
MODIFYING A BINARY SEARCH TREE

(Insertion and Deletion)

Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

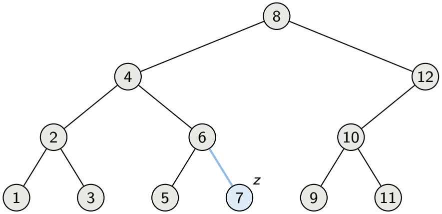
Ex: insert z with key 7



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

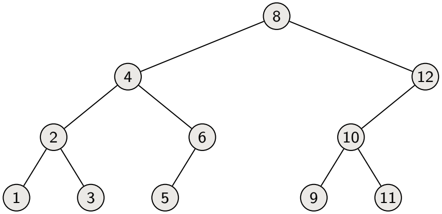
Ex: insert z with key 7



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

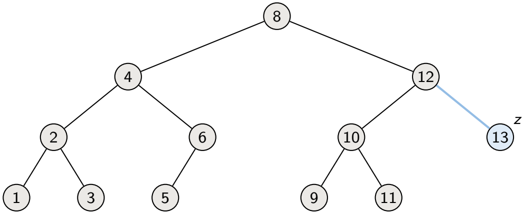
Ex: insert z with key 13



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

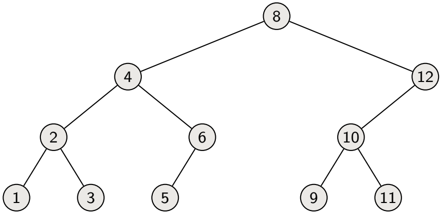
Ex: insert z with key 13



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

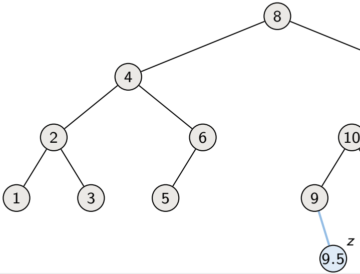
Ex: insert z with key 9.5



Idea of inserting z

- Search for $z.key$
- When arrived at *nil* insert z at that position

Ex: insert z with key 9.5



Insertion

"search" phase

"insert" phase

TREE-INSERT(T, z)

```
y = NIL
x = T.root
while x ≠ NIL
      y = x
      if z.key < x.key
            x = x.left
      else x = x.right
z.p = y
if y == NIL
      T.root = z          // tree T was empty
elseif z.key < y.key
      y.left = z
else y.right = z
```

Insertion

"search" phase

"insert" phase

```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z          // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

What is the running time?

Insertion

"search" phase

"insert" phase

```
TREE-INSERT(T, z)
  y = NIL
  x = T.root
  while x ≠ NIL
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y
  if y == NIL
      T.root = z          // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
```

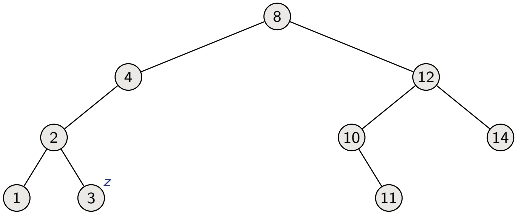
What is the running time? $O(h)$

Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it

Ex: Delete z

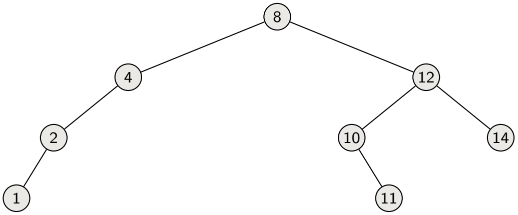


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it

Ex: Delete z

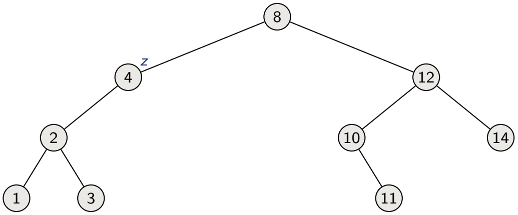


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z's position in the tree

Ex: Delete z

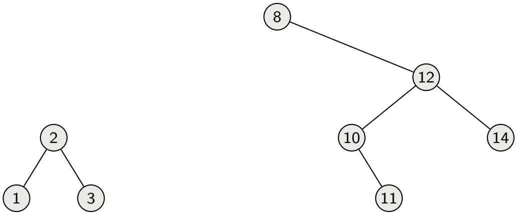


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z's position in the tree

Ex: Delete z

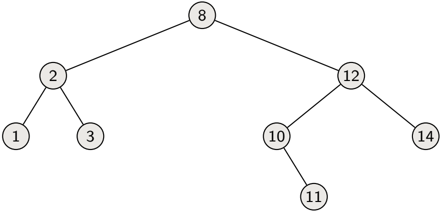


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z's position in the tree

Ex: Delete z

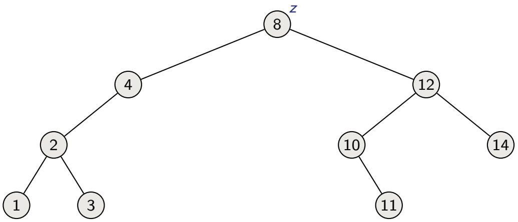


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z's position in the tree
- If z has two children, then find its successor y and replace z by y

Ex: Delete z

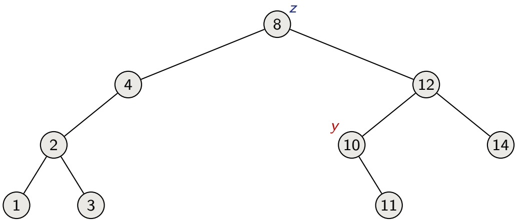


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z's position in the tree
- If z has two children, then find its successor y and replace z by y

Ex: Delete z

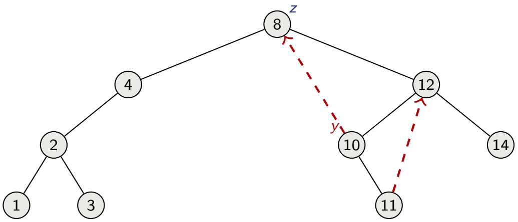


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z's position in the tree
- If z has two children, then find its successor y and replace z by y

Ex: Delete z

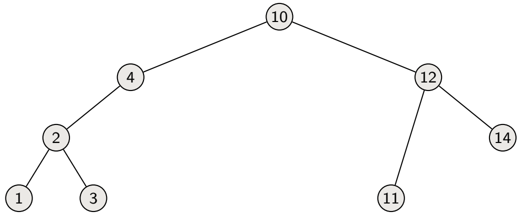


Idea of deletion

Conceptually 3 cases:

- If z has no children, remove it
- If z has one child, then make that child take z's position in the tree
- If z has two children, then find its successor y and replace z by y

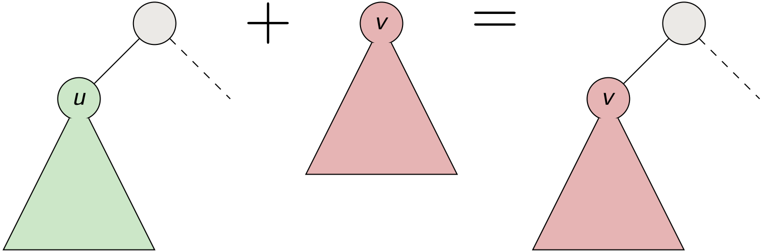
Ex: Delete z



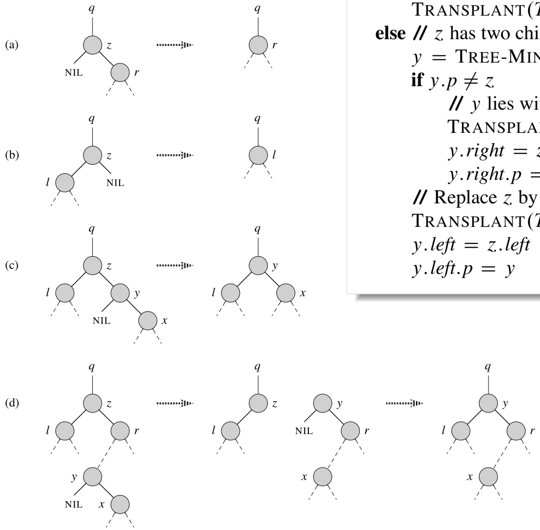
Deletion Implementation: Transplant

TRANSPLANT(T, u, v)
if $u.p$ == NIL
    $T.root = v$
elseif $u == u.p.left$
    $u.p.left = v$
else $u.p.right = v$
if $v \neq$ NIL
    $v.p = u.p$

TRANSPLANT(T, u, v) replaces subtree rooted at u with that rooted at v



Deletion Procedure




```
TREE-DELETE(T, z)
  if z.left == NIL
      TRANSPLANT(T, z, z.right)          // z has no left child
  elseif z.right == NIL
      TRANSPLANT(T, z, z.left)           // z has just a left child
  else // z has two children.
      y = TREE-MINIMUM(z.right)          // y is z's successor
      if y.p ≠ z
          // y lies within z's right subtree but is not the root of this
          TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      // Replace z by y.
      TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
```

Summary of Binary Search Trees

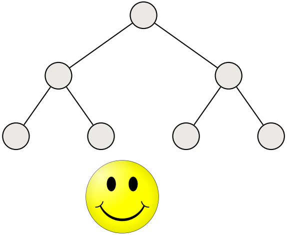
Query operations: Search, Max, Min, Predecessor, Successor: **O(h)** time

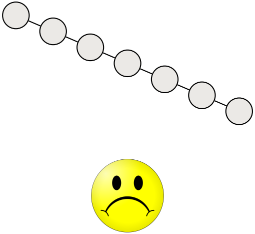
Modifying operations: Insertion, Deletion: **O(h)** time

Summary of Binary Search Trees

Query operations: Search, Max, Min, Predecessor, Successor: **O(h)** time

Modifying operations: Insertion, Deletion: **O(h)** time



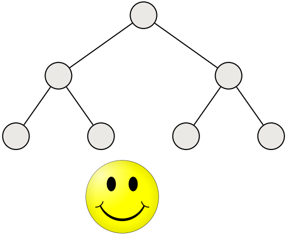


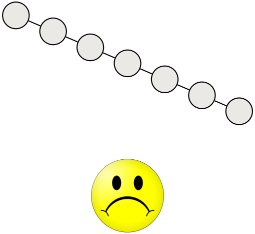
Summary of Binary Search Trees

Query operations: Search, Max, Min, Predecessor, Successor: **O(h)** time

Modifying operations: Insertion, Deletion: **O(h)** time

Exist efficient procedures to keep tree balanced (AVL trees, red-black trees, etc.)





Comparison of Data Structures

Stacks: Last-in-first-out, Insertion and deletion $O(1)$ time,
Array implementation: fixed capacity

Comparison of Data Structures

Stacks: Last-in-first-out, Insertion and deletion $O(1)$ time,
Array implementation: fixed capacity

Queues: First-in-first-out, Insertion and deletion $O(1)$ time,
Array implementation: fixed capacity

Comparison of Data Structures

Stacks: Last-in-first-out, Insertion and deletion $O(1)$ time,
Array implementation: fixed capacity

Queues: First-in-first-out, Insertion and deletion $O(1)$ time,
Array implementation: fixed capacity

Linked List: No fixed capacity, Insertion and deletion $O(1)$ time, supports search but $O(n)$ time

Comparison of Data Structures

Stacks: Last-in-first-out, Insertion and deletion $O(1)$ time,
Array implementation: fixed capacity

Queues: First-in-first-out, Insertion and deletion $O(1)$ time,
Array implementation: fixed capacity

Linked List: No fixed capacity, Insertion and deletion $O(1)$ time, supports search but $O(n)$ time

Binary Search Trees: No fixed capacity, supports most operations (insertion, deletion, search, max, min, ...) in time $O(\text{height of tree})$

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of "programming")

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of "programming")

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

DYNAMIC PROGRAMMING

(An algorithmic paradigm not a way of “programming”)

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

What is $2^5 + 3 - \sqrt{16}$?

Dynamic Programming (DP)

Main idea:

- Remember calculations already made
- Saves enormous amounts of computation

Allows to solve many optimization problems

- Always at least one question in google code jam needs DP

First application: Fibonacci numbers

Sequence of numbers defined 1000 years ago:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

First application: Fibonacci numbers

Sequence of numbers defined 1000 years ago:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

$1, 1$

First application: Fibonacci numbers

Sequence of numbers defined 1000 years ago:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

$1, 1, 2$

First application: Fibonacci numbers

Sequence of numbers defined 1000 years ago:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

$1, 1, 2, 3$

First application: Fibonacci numbers

Sequence of numbers defined 1000 years ago:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

$1, 1, 2, 3, 5$

First application: Fibonacci numbers

Sequence of numbers defined 1000 years ago:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

$1, 1, 2, 3, 5, 8$

First application: Fibonacci numbers

Sequence of numbers defined 1000 years ago:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

$1, 1, 2, 3, 5, 8, 13$

First application: Fibonacci numbers

Sequence of numbers defined 1000 years ago:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

$1, 1, 2, 3, 5, 8, 13, 21$

First application: Fibonacci numbers

Sequence of numbers defined 1000 years ago:

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

1, 1, 2, 3, 5, 8, 13, 21, ?

Calculating the *n*-th Fibonacci number

Calculating the n-th Fibonacci number

First idea:

FIB(n)

1. **if** $n = 0$ or $n = 1$
2. **return** 1
3. **else**
4. **return** FIB($n - 1$) + FIB($n - 2$)

Calculating the n-th Fibonacci number

First idea:

$\textsc{Fib}(n)$

1. **if** $n = 0$ or $n = 1$
2. **return** 1
3. **else**
4. **return** $\textsc{Fib}(n-1) + \textsc{Fib}(n-2)$

What is the problem?

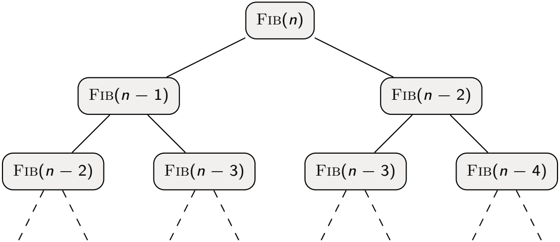
Calculating the n-th Fibonacci number

First idea:

```
FIB(n)

if n = 0 or n = 1
    return 1
else
    return FIB(n − 1) + FIB(n − 2)
```

What is the problem?



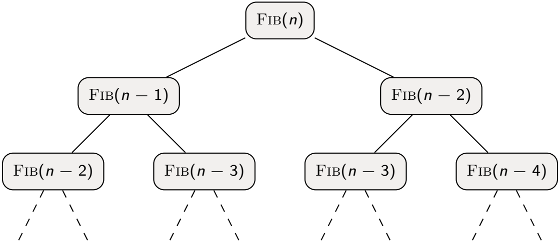
Calculating the n-th Fibonacci number

First idea:

$\text{Fib}(n)$

1. **if** $n = 0$ or $n = 1$
2. **return** 1
3. **else**
4. **return** $\text{Fib}(n-1) + \text{Fib}(n-2)$

What is the problem? Same calculations again and again $\Rightarrow$ exponential time!



The solution

Remember what we have done

The solution

Remember what we have done

Two different ways:

1. Top-down with memoization
    - Solve recursively but store each result in a table
    - Memoizing is remembering what we have computed previously

The solution

Remember what we have done

Two different ways:

1. Top-down with memoization
   - Solve recursively but store each result in a table
   - Memoizing is remembering what we have computed previously
2. Bottom-up
   - Sort the subproblems and solve the smaller ones first
   - That way, when solving a subproblem, have already solved the smaller subproblems we need

Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

1. Let $r = [0 \ldots n]$ be a new array
2. **for** $i = 0$ **to** n
3. $\quad r[i] \leftarrow -\infty$
4. **return** MEMOIZED-FIB-AUX(n, r)

Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

1. Let $r = [0 \ldots n]$ be a new array
2. **for** $i = 0$ **to** n
3. $\quad r[i] \leftarrow -\infty$
4. **return** MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)

1. **if** $r[n] \geq 0$
2. $\quad$ **return** $r[n]$
3. **if** $n = 0$ or $n = 1$
4. $\quad ans \leftarrow 1$
5. **else**
6. $\quad ans \leftarrow$ MEMOIZED-FIB-AUX($n-1, r$)+ MEMOIZED-FIB-AUX($n-2, r$)
7. $r[n] \leftarrow ans$
8. **return** $r[n]$

Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

1. Let $r = [0 \ldots n]$ be a new array
2. **for** $i = 0$ **to** n
3. $\quad r[i] \leftarrow -\infty$
4. **return** MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)

1. **if** $r[n] \geq 0$
2. $\quad$ **return** $r[n]$
3. **if** $n = 0$ or $n = 1$
4. $\quad ans \leftarrow 1$
5. **else**
6. $\quad ans \leftarrow$ MEMOIZED-FIB-AUX($n-1, r$)+ MEMOIZED-FIB-AUX($n-2, r$)
7. $r[n] \leftarrow ans$
8. **return** $r[n]$

M-F-A(n, r)

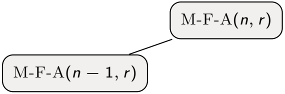
Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

```
Let r = [0 . . . n] be a new array
for i = 0 to n
    r[i] ← −∞
return MEMOIZED-FIB-AUX(n, r)
```

MEMOIZED-FIB-AUX(n, r)

```
if r[n] ≥ 0
    return r[n]
if n = 0 or n = 1
    ans ← 1
else
    ans ← MEMOIZED-FIB-AUX(n − 1, r)+
              MEMOIZED-FIB-AUX(n − 2, r)
r[n] ← ans
return r[n]
```



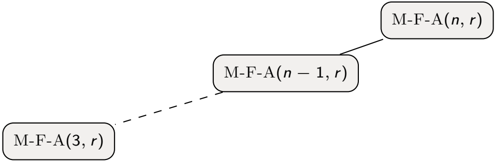
Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

Let $r = [0 \ldots n]$ be a new array
for $i = 0$ **to** n
    $r[i] \leftarrow -\infty$
return MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)

if $r[n] \geq 0$
    return $r[n]$
if $n = 0$ or $n = 1$
    $ans \leftarrow 1$
else
    $ans \leftarrow$ MEMOIZED-FIB-AUX($n-1, r$)+
        MEMOIZED-FIB-AUX($n-2, r$)
$r[n] \leftarrow ans$
return $r[n]$



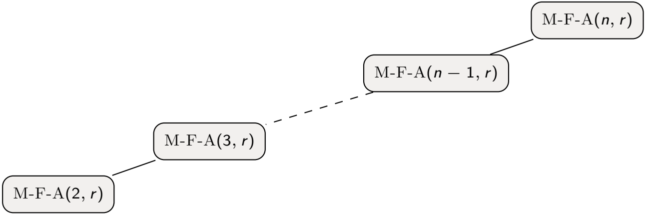
Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

```
Let r = [0 ... n] be a new array
for i = 0 to n
    r[i] ← −∞
return MEMOIZED-FIB-AUX(n, r)
```

MEMOIZED-FIB-AUX(n, r)

```
if r[n] ≥ 0
    return r[n]
if n = 0 or n = 1
    ans ← 1
else
    ans ← MEMOIZED-FIB-AUX(n − 1, r)+
              MEMOIZED-FIB-AUX(n − 2, r)
r[n] ← ans
return r[n]
```



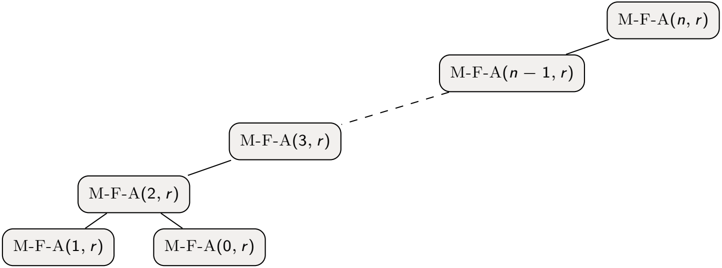
Top-down with memoization: Fibonacci numbers

```
MEMOIZED-FIB-AUX(n, r)

if r[n] ≥ 0
    return r[n]
if n = 0 or n = 1
    ans ← 1
else
    ans ← MEMOIZED-FIB-AUX(n − 1, r)+
            MEMOIZED-FIB-AUX(n − 2, r)
r[n] ← ans
return r[n]
```

```
MEMOIZED-FIB(n)

Let r = [0 . . . n] be a new array
for i = 0 to n
    r[i] ← −∞
return MEMOIZED-FIB-AUX(n, r)
```



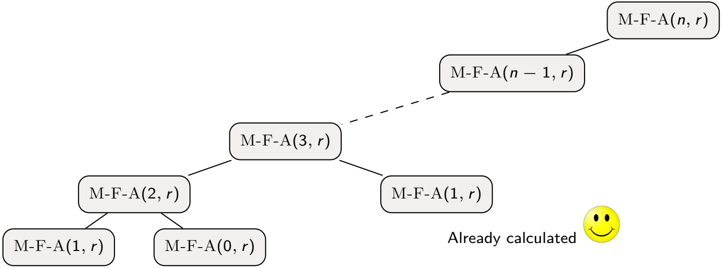
Top-down with memoization: Fibonacci numbers

```
MEMOIZED-FIB(n)

Let r = [0 ... n] be a new array
for i = 0 to n
    r[i] ← −∞
return MEMOIZED-FIB-AUX(n, r)
```

```
MEMOIZED-FIB-AUX(n, r)

if r[n] ≥ 0
    return r[n]
if n = 0 or n = 1
    ans ← 1
else
    ans ← MEMOIZED-FIB-AUX(n − 1, r)+
              MEMOIZED-FIB-AUX(n − 2, r)
r[n] ← ans
return r[n]
```



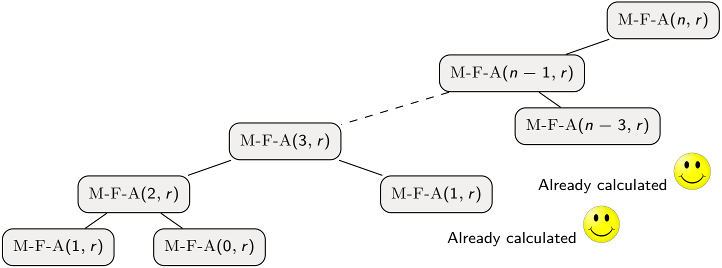
Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

Let $r = [0 \ldots n]$ be a new array
for $i = 0$ **to** n
    $r[i] \leftarrow -\infty$
return MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)

if $r[n] \geq 0$
    return $r[n]$
if $n = 0$ or $n = 1$
    $ans \leftarrow 1$
else
    $ans \leftarrow$ MEMOIZED-FIB-AUX($n-1, r$)+
            MEMOIZED-FIB-AUX($n-2, r$)
$r[n] \leftarrow ans$
return $r[n]$



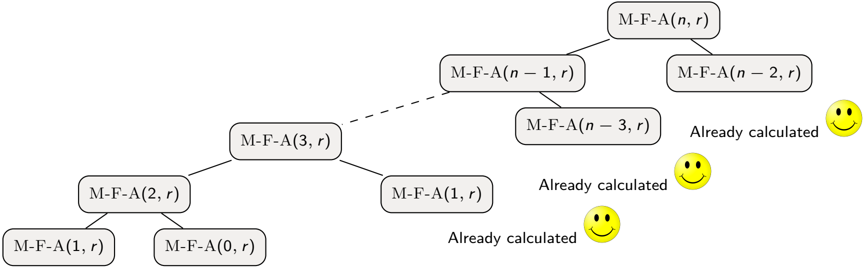
Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

```
Let r = [0 ... n] be a new array
for i = 0 to n
    r[i] ← −∞
return MEMOIZED-FIB-AUX(n, r)
```

MEMOIZED-FIB-AUX(n, r)

```
if r[n] ≥ 0
    return r[n]
if n = 0 or n = 1
    ans ← 1
else
    ans ← MEMOIZED-FIB-AUX(n − 1, r)+
              MEMOIZED-FIB-AUX(n − 2, r)
r[n] ← ans
return r[n]
```



Top-down with memoization: Fibonacci numbers

```
MEMOIZED-FIB(n)

Let r = [0 ... n] be a new array
for i = 0 to n
    r[i] ← −∞
return MEMOIZED-FIB-AUX(n, r)
```

```
MEMOIZED-FIB-AUX(n, r)

if r[n] ≥ 0
    return r[n]
if n = 0 or n = 1
    ans ← 1
else
    ans ← MEMOIZED-FIB-AUX(n − 1, r)+
          MEMOIZED-FIB-AUX(n − 2, r)
r[n] ← ans
return r[n]
```

Time analysis:

Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

1. Let $r = [0 \ldots n]$ be a new array
2. **for** $i = 0$ **to** n
3. $\quad r[i] \leftarrow -\infty$
4. **return** MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)

1. **if** $r[n] \geq 0$
2. $\quad$ **return** $r[n]$
3. **if** $n = 0$ or $n = 1$
4. $\quad ans \leftarrow 1$
5. **else**
6. $\quad ans \leftarrow$ MEMOIZED-FIB-AUX($n-1, r$)+ MEMOIZED-FIB-AUX($n-2, r$)
7. $r[n] \leftarrow ans$
8. **return** $r[n]$

Time analysis:

- Steps 1-3 in MEMOIZED-FIB take time $\Theta(n)$

Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

1. Let $r = [0 \ldots n]$ be a new array
2. **for** $i = 0$ **to** n
3. $\quad r[i] \leftarrow -\infty$
4. **return** MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)

1. **if** $r[n] \geq 0$
2. $\quad$ **return** $r[n]$
3. **if** $n = 0$ or $n = 1$
4. $\quad$ $ans \leftarrow 1$
5. **else**
6. $\quad$ $ans \leftarrow$ MEMOIZED-FIB-AUX($n-1, r$)+ MEMOIZED-FIB-AUX($n-2, r$)
7. $r[n] \leftarrow ans$
8. **return** $r[n]$

Time analysis:

- Steps 1-3 in MEMOIZED-FIB take time $\Theta(n)$
- Each call to MEMOIZED-FIB-AUX takes time $\Theta(1)$

Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

1. Let $r = [0 \dots n]$ be a new array
2. **for** $i = 0$ **to** n
3. $\quad r[i] \leftarrow -\infty$
4. **return** MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)

1. **if** $r[n] \geq 0$
2. $\quad$ **return** $r[n]$
3. **if** $n = 0$ or $n = 1$
4. $\quad ans \leftarrow 1$
5. **else**
6. $\quad ans \leftarrow$ MEMOIZED-FIB-AUX($n - 1, r$)+ MEMOIZED-FIB-AUX($n - 2, r$)
7. $r[n] \leftarrow ans$
8. **return** $r[n]$

Time analysis:

- Steps 1-3 in MEMOIZED-FIB take time $\Theta(n)$
- Each call to MEMOIZED-FIB-AUX takes time $\Theta(1)$
- Number of calls to MEMOIZED-FIB-AUX is $\Theta(n)$

Top-down with memoization: Fibonacci numbers

MEMOIZED-FIB(n)

1. Let $r = [0 \dots n]$ be a new array
2. **for** $i = 0$ **to** n
3. $\quad r[i] \leftarrow -\infty$
4. **return** MEMOIZED-FIB-AUX(n, r)

MEMOIZED-FIB-AUX(n, r)

1. **if** $r[n] \geq 0$
2. $\quad$ **return** $r[n]$
3. **if** $n = 0$ or $n = 1$
4. $\quad ans \leftarrow 1$
5. **else**
6. $\quad ans \leftarrow$ MEMOIZED-FIB-AUX($n - 1, r$)+ MEMOIZED-FIB-AUX($n - 2, r$)
7. $r[n] \leftarrow ans$
8. **return** $r[n]$

Time analysis:

- Steps 1-3 in MEMOIZED-FIB take time $\Theta(n)$
- Each call to MEMOIZED-FIB-AUX takes time $\Theta(1)$
- Number of calls to MEMOIZED-FIB-AUX is $\Theta(n)$
- Total time is thus $\Theta(n)$

Bottom-up: Fibonacci numbers

```
BOTTOM-UP-FIB(n)

Let r = [0 . . . n] be a new array
r[0] ← 1
r[1] ← 1
for i = 2 to n
    r[i] ← r[i − 1] + r[i − 2]
return r[n]
```

Example $n = 8$:

$r =$

1							

Bottom-up: Fibonacci numbers

```
BOTTOM-UP-FIB(n)

Let r = [0 . . . n] be a new array
r[0] ← 1
r[1] ← 1
for i = 2 to n
    r[i] ← r[i − 1] + r[i − 2]
return r[n]
```

Example $n = 8$:

$r =$

1	1						

Bottom-up: Fibonacci numbers

BOTTOM-UP-FIB(n)

Let $r = [0 \ldots n]$ be a new array
$r[0] \leftarrow 1$
$r[1] \leftarrow 1$
for $i = 2$ **to** n
  $r[i] \leftarrow r[i-1] + r[i-2]$
return $r[n]$

Example $n = 8$:

$r =$

1	1	2					

Bottom-up: Fibonacci numbers

Bottom-Up-Fib(n)

Let $r = [0 \ldots n]$ be a new array
$r[0] \leftarrow 1$
$r[1] \leftarrow 1$
for $i = 2$ **to** n
    $r[i] \leftarrow r[i-1] + r[i-2]$
return $r[n]$

Example $n = 8$:

$r =$

1	1	2	3				

Bottom-up: Fibonacci numbers

BOTTOM-UP-FIB(n)

1. Let $r = [0 \ldots n]$ be a new array
2. $r[0] \leftarrow 1$
3. $r[1] \leftarrow 1$
3. **for** $i = 2$ **to** n
4. $\quad r[i] \leftarrow r[i-1] + r[i-2]$
5. **return** $r[n]$

Example $n = 8$:

$r =$

1	1	2	3	5			

Bottom-up: Fibonacci numbers

```
BOTTOM-UP-FIB(n)

Let r = [0 . . . n] be a new array
r[0] ← 1
r[1] ← 1
for i = 2 to n
    r[i] ← r[i − 1] + r[i − 2]
return r[n]
```

Example $n = 8$:

$r =$

1	1	2	3	5	8		

Bottom-up: Fibonacci numbers

Bottom-Up-Fib(n)

Let $r = [0 \ldots n]$ be a new array
$r[0] \leftarrow 1$
$r[1] \leftarrow 1$
for $i = 2$ **to** n
    $r[i] \leftarrow r[i-1] + r[i-2]$
return $r[n]$

Example $n = 8$:

$r =$

1	1	2	3	5	8	13	

Bottom-up: Fibonacci numbers

BOTTOM-UP-FIB(n)

1. Let $r = [0 \ldots n]$ be a new array
2. $r[0] \leftarrow 1$
3. $r[1] \leftarrow 1$
3. **for** $i = 2$ **to** n
4. $\quad r[i] \leftarrow r[i-1] + r[i-2]$
5. **return** $r[n]$

Example $n = 8$:

$r =$

1	1	2	3	5	8	13	21

Bottom-up: Fibonacci numbers

Bottom-Up-Fib(n)

1. Let $r = [0 \ldots n]$ be a new array
2. $r[0] \leftarrow 1$
3. $r[1] \leftarrow 1$
3. **for** $i = 2$ **to** n
4. $\quad r[i] \leftarrow r[i-1] + r[i-2]$
5. **return** $r[n]$

Example $n = 8$:

$r =$

1	1	2	3	5	8	13	21

Time?

Bottom-up: Fibonacci numbers

```
BOTTOM-UP-FIB(n)

Let r = [0 . . . n] be a new array
r[0] ← 1
r[1] ← 1
for i = 2 to n
    r[i] ← r[i − 1] + r[i − 2]
return r[n]
```

Example $n = 8$:

$r =$

1	1	2	3	5	8	13	21

Time? $\Theta(n)$

Summary

- We had a recursive formulation of our problem

$$F_0 = 1$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

- Introduced memory (array r)
- Filled in table"top-down with memoization" or with "bottom-up"

Key elements in designing a DP-algorithm

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or several subproblems to solve

Key elements in designing a DP-algorithm

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or several subproblems to solve and the optimal solution solves the subproblems optimally

Key elements in designing a DP-algorithm

Optimal substructure

- Show that a solution to a problem consists of making a choice, which leaves one or several subproblems to solve and the optimal solution solves the subproblems optimally

Overlapping subproblems

- A naive recursive algorithm may revisit the same (sub)problem over and over.
- Top-down with memoization

  Solve recursively but store each result in a table
- Bottom-up

  Sort the subproblems and solve the smaller ones first; that way, when solving a subproblem, have already solved the smaller subproblems we need

ROD CUTTING

Rod cutting

Instance:

- A length n of a metal rod.
- A table of prices p_i for rods of lengths $i = 1, \ldots, n$.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod cutting

Instance:

- A length n of a metal rod.
- A table of prices p_i for rods of lengths $i = 1, \ldots, n$.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Objective: Decide how to cut the rod into pieces and maximize the price.

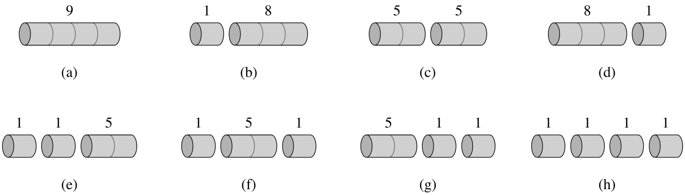
Rod cutting

Instance:

- A length n of a metal rod.
- A table of prices p_i for rods of lengths $i = 1, \ldots, n$.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Objective: Decide how to cut the rod into pieces and maximize the price.



Size of the Problem

- There 2^{n-1} possible solutions—either cut or do not cut after every length unit.

Size of the Problem

- There 2^{n-1} possible solutions—either cut or do not cut after every length unit.
- Need structure for an efficient algorithm.

Size of the Problem

- There 2^{n-1} possible solutions—either cut or do not cut after every length unit.
- Need structure for an efficient algorithm.

Theorem

If:

- *the leftmost cut in an optimal solution is after i units.*
- *an optimal way to cut a solution of size $n - i$ is into rods of sizes: $s_1, s_2, \ldots, s_k$.*

Then, an optimal way to cut our rod is into rods of sizes: $i, s_1, s_2, \ldots, s_k$.

Proof of Structural Theorem

Theorem

If:

- *the leftmost cut in an optimal solution is after i units.*
- *an optimal way to cut a solution of size $n - i$ is into rods of sizes: $s_1, s_2, \ldots, s_k$.*

Then, an optimal way to cut our rod is into rods of sizes: $i, s_1, s_2, \ldots, s_k$.

Proof

Proof of Structural Theorem

Theorem

If:

- *the leftmost cut in an optimal solution is after* i *units.*
- *an optimal way to cut a solution of size* $n - i$ *is into rods of sizes:* $s_1, s_2, \ldots, s_k$.

Then, an optimal way to cut our rod is into rods of sizes: $i, s_1, s_2, \ldots, s_k$.

Proof

Feasibility: Since $s_1, s_2, \ldots, s_k$ is a feasible solution for an instance of size $n - i$:

$$\sum_{j=1}^{k} s_j = n - i \ .$$

Proof of Structural Theorem

Theorem

If:

- *the leftmost cut in an optimal solution is after i units.*
- *an optimal way to cut a solution of size $n - i$ is into rods of sizes: $s_1, s_2, \ldots, s_k$.*

Then, an optimal way to cut our rod is into rods of sizes: $i, s_1, s_2, \ldots, s_k$.

Proof

Feasibility: Since $s_1, s_2, \ldots, s_k$ is a feasible solution for an instance of size $n - i$:

$$\sum_{j=1}^{k} s_j = n - i \ .$$

Hence, $i + \sum_{j=1}^{k} s_j = n$.

Proof of Structural Theorem

Theorem

If:

- *the leftmost cut in an optimal solution is after i units.*
- *an optimal way to cut a solution of size $n - i$ is into rods of sizes: $s_1, s_2, \ldots, s_k$.*

Then, an optimal way to cut our rod is into rods of sizes: $i, s_1, s_2, \ldots, s_k$.

Proof

Optimality: Let $i, o_1, o_2, \ldots, o_\ell$ be an optimal solution—exists by assumption.

Proof of Structural Theorem

Theorem

If:

- *the leftmost cut in an optimal solution is after i units.*
- *an optimal way to cut a solution of size $n - i$ is into rods of sizes: $s_1, s_2, \ldots, s_k$.*

Then, an optimal way to cut our rod is into rods of sizes: $i, s_1, s_2, \ldots, s_k$.

Proof

Optimality: Let $i, o_1, o_2, \ldots, o_\ell$ be an optimal solution—exists by assumption. Recall that $s_1, s_2, \ldots, s_k$ is an optimal way to cut a rod of size $n - i$, thus,

$$\sum_{j=1}^{k} p_{s_j} \geq \sum_{j=1}^{\ell} p_{o_j} \quad .$$

Proof of Structural Theorem

Theorem

If:

- *the leftmost cut in an optimal solution is after i units.*
- *an optimal way to cut a solution of size $n - i$ is into rods of sizes: $s_1, s_2, \ldots, s_k$.*

Then, an optimal way to cut our rod is into rods of sizes: $i, s_1, s_2, \ldots, s_k$.

Proof

Optimality: Let $i, o_1, o_2, \ldots, o_\ell$ be an optimal solution—exists by assumption. Recall that $s_1, s_2, \ldots, s_k$ is an optimal way to cut a rod of size $n - i$, thus,

$$\sum_{j=1}^{k} p_{s_j} \geq \sum_{j=1}^{\ell} p_{o_j} \quad .$$

Hence, $p_i + \sum_{j=1}^{k} p_{s_j} \geq p_i + \sum_{j=1}^{\ell} p_{o_j}$.

First Algorithm

If we let $r(n)$ be the optimal revenue from a rod of length n, then, by the structural theorem, we can express $r(n)$ recursively as follows

$$r(n) = \begin{cases} 0 & \text{if } n = 0 \ , \\ \max_{1 \leq i \leq n} \{p_i + r(n-i)\} & \text{otherwise if } n \geq 1 \ . \end{cases}$$

First Algorithm

If we let $r(n)$ be the optimal revenue from a rod of length n, then, by the structural theorem, we can express $r(n)$ recursively as follows

$$r(n) = \begin{cases} 0 & \text{if } n = 0 \ , \\ \max_{1 \leq i \leq n} \{p_i + r(n-i)\} & \text{otherwise if } n \geq 1 \ . \end{cases}$$

```
CUT-ROD(p, n)
  if n == 0
      return 0
  q = −∞
  for i = 1 to n
      q = max(q, p[i] + CUT-ROD(p, n − i))
  return q
```

Problem

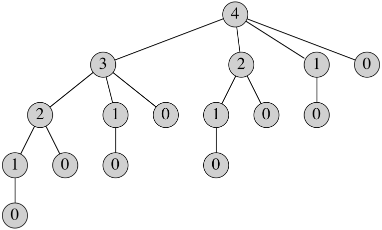
- The procedure is extremely inefficient—in fact exponential.

Problem

- The procedure is extremely inefficient—in fact exponential.
- What went wrong?

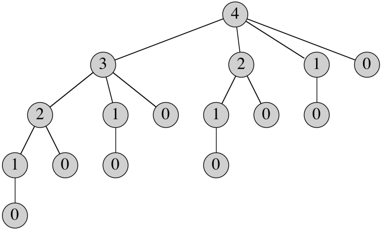
Problem

- The procedure is extremely inefficient—in fact exponential.
- What went wrong?



Problem

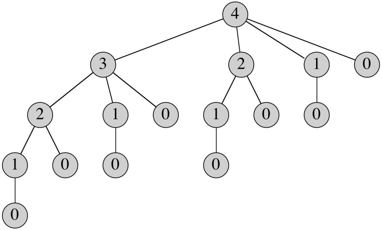
- The procedure is extremely inefficient—in fact exponential.
- What went wrong?



- The procedure repeatedly calculates the same profits.

Problem

- The procedure is extremely inefficient—in fact exponential.
- What went wrong?



- The procedure repeatedly calculates the same profits.
- Dynamic programming can save the extra calculations.

Top-Down Dynamic Programming

General Approach

Top-Down Dynamic Programming

General Approach

- Keep the recursive structure of the pseudocode.

Top-Down Dynamic Programming

General Approach

- Keep the recursive structure of the pseudocode.
- Memoize (store) the result of every recursive call.

Top-Down Dynamic Programming

General Approach

- Keep the recursive structure of the pseudocode.
- Memoize (store) the result of every recursive call.
- At each recursive call, try to avoid work using memoized results.

Top-Down Dynamic Programming

General Approach

- Keep the recursive structure of the pseudocode.
- Memoize (store) the result of every recursive call.
- At each recursive call, try to avoid work using memoized results.

Pseudocode

```
MEMOIZED-CUT-ROD(p, n)
  let r[0..n] be a new array
  for i = 0 to n
      r[i] = −∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Top-Down Dynamic Programming

General Approach

- Keep the recursive structure of the pseudocode.
- Memoize (store) the result of every recursive call.
- At each recursive call, try to avoid work using memoized results.

Pseudocode

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else q = −∞
      for i = 1 to n
          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
  r[n] = q
  return q
```

```
MEMOIZED-CUT-ROD(p, n)
  let r[0..n] be a new array
  for i = 0 to n
      r[i] = −∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

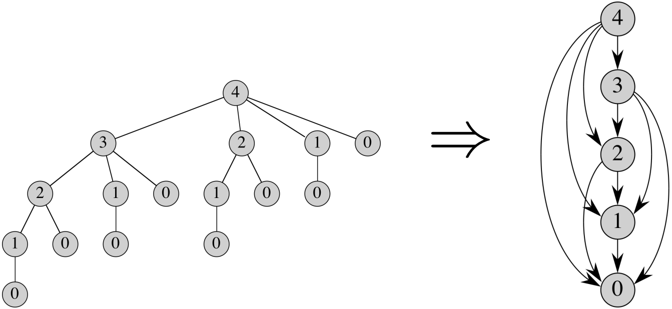
What did we gain?

What did we gain?

Memoization helps us avoid recalculations.

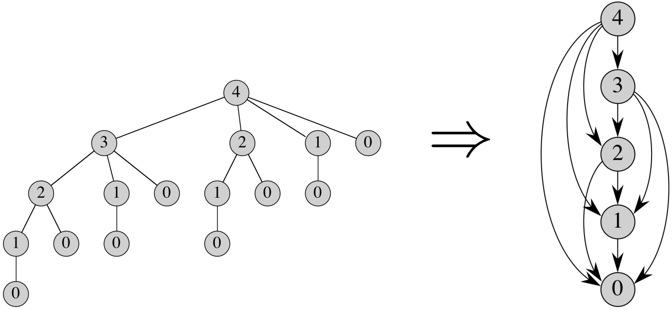
What did we gain?

Memoization helps us avoid recalculations.



What did we gain?

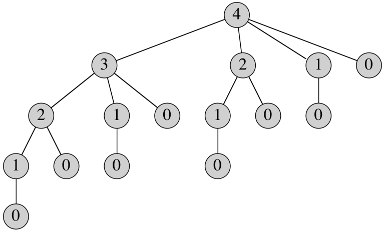
Memoization helps us avoid recalculations.



One can think of all the recursive calls using a memoized value as additional parents of the call generating this value.

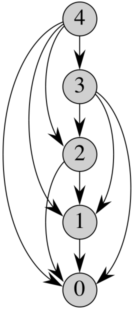
What did we gain?

Memoization helps us avoid recalculations.



$\Rightarrow$

Subproblem Graph



One can think of all the recursive calls using a memoized value as additional parents of the call generating this value.

Time Complexity

Time Complexity

- The initialization takes $O(n)$ time.

```
MEMOIZED-CUT-ROD(p, n)
  let r[0..n] be a new array
  for i = 0 to n
      r[i] = -∞
  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

Time Complexity

- The initialization takes $O(n)$ time.
- Processing each sub-problem takes linear time in the number of sub-problems it evokes.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else q = −∞
      for i = 1 to n
          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
  r[n] = q
  return q
```

Time Complexity

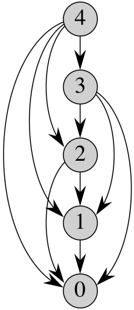
- The initialization takes $O(n)$ time.
- Processing each sub-problem takes linear time in the number of sub-problems it evokes.

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
  if r[n] ≥ 0
      return r[n]
  if n == 0
      q = 0
  else q = −∞
      for i = 1 to n
          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
  r[n] = q
  return q
```

- The time complexity is proportional to the number of nodes and edges in the subproblem graph.

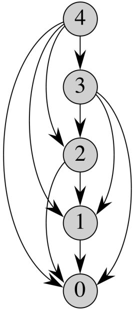
Time Complexity

- The initialization takes $O(n)$ time.
- Processing each sub-problem takes linear time in the number of sub-problems it evokes.
- The time complexity is proportional to the number of nodes and edges in the subproblem graph.



Time Complexity

- The initialization takes $O(n)$ time.
- Processing each sub-problem takes linear time in the number of sub-problems it evokes.
- The time complexity is proportional to the number of nodes and edges in the subproblem graph.



Time Complexity
$O(n^2)$

Bottom-Up Dynamic Programming

General Approach

Bottom-Up Dynamic Programming

General Approach

- Sort the sub-problems by size.

Bottom-Up Dynamic Programming

General Approach

- Sort the sub-problems by size.
- Solve the smaller ones first.

Bottom-Up Dynamic Programming

General Approach

- Sort the sub-problems by size.
- Solve the smaller ones first.
- When reaching a sub-problem, the smaller ones are already solved.

Bottom-Up Dynamic Programming

General Approach

- Sort the sub-problems by size.
- Solve the smaller ones first.
- When reaching a sub-problem, the smaller ones are already solved.

Pseudocode

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0 . . n] be a new array
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          q = max(q, p[i] + r[j − i])
      r[j] = q
  return r[n]
```

Bottom-Up Dynamic Programming

General Approach

- Sort the sub-problems by size.
- Solve the smaller ones first.
- When reaching a sub-problem, the smaller ones are already solved.

Pseudocode

```
BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] be a new array
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          q = max(q, p[i] + r[j − i])
      r[j] = q
  return r[n]
```

Time Complexity
$O(n^2)$

Reconstructing an Optimal Solution

- The above algorithms only return the optimal profit.
- Sometimes one needs also to find an optimal solution.

Reconstructing an Optimal Solution

- The above algorithms only return the optimal profit.
- Sometimes one needs also to find an optimal solution.

Approach

- Each cell of the memoization table corresponds to a decision: the location of the left most cut.

Reconstructing an Optimal Solution

- The above algorithms only return the optimal profit.
- Sometimes one needs also to find an optimal solution.

Approach

- Each cell of the memoization table corresponds to a decision: the location of the left most cut.
- Store the decision corresponding to every cell in a separate table.

Reconstructing an Optimal Solution (cont.)

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] and s[0..n] be new arrays
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          if q < p[i] + r[j − i]
              q = p[i] + r[j − i]
              s[j] = i
      r[j] = q
  return r and s
```

Reconstructing an Optimal Solution (cont.)

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
  let r[0..n] and s[0..n] be new arrays
  r[0] = 0
  for j = 1 to n
      q = −∞
      for i = 1 to j
          if q < p[i] + r[j − i]
              q = p[i] + r[j − i]
              s[j] = i
      r[j] = q
  return r and s
```

Output

i	0	1	2	3	4	5	6	7	8
r[i]	0	1	5	8	10	13	17	18	22
s[i]	0	1	2	3	2	2	6	1	2

Summary

- We had a recursive formulation for the optimal value for our problem

$$r(n) = \begin{cases} 0 & \text{if } n = 0 \ , \\ \max_{1 \leq i \leq n} \{p_i + r(n-i)\} & \text{otherwise if } n \geq 1 \ . \end{cases}$$

- Speed up the calculations by filling in a table either "top-down with memoization" or with "bottom-up".
- Recovered an optimal solution using an additional table.

Problem Solving: the Change-Making Problem

- How can a given amount of money be made with the least number of coins of given denominations?

Problem Solving: the Change-Making Problem

- How can a given amount of money be made with the least number of coins of given denominations?

Formally:

Input: n distinct coin denominators (integers) $0 < w_1 < w_2 < \ldots < w_n$ and an amount W (the change) which is also a positive integer.

Output: The minimum number of coins needed in order to make the change:

$$\min \left\{ \sum_{j=1}^{n} x_j : \sum_{j=1}^{n} w_j x_j = W \text{ and } x_j\text{'s are integers} \right\}.$$

Problem Solving: the Change-Making Problem

- How can a given amount of money be made with the least number of coins of given denominations?

Formally:

Input: n distinct coin denominators (integers) $0 < w_1 < w_2 < \ldots < w_n$ and an amount W (the change) which is also a positive integer.

Output: The minimum number of coins needed in order to make the change:

$$\min\left\{\sum_{j=1}^{n} x_j : \sum_{j=1}^{n} w_j x_j = W \text{ and } x_j\text{'s are integers}\right\}.$$

Example: On input $w_1 = 1, w_2 = 2, w_3 = 5$ and $W = 8$, the output should be 3 since the best way of giving 8 is $x_1 = x_2 = x_3 = 1$.

Summary

- Identify choices and optimal substructure
- Write optimal solution recursively as a function of smaller subproblems
- Use top-down with memoization or bottom-up to solve the recursion efficiently (without repeatedly solving the same subproblems)