CS 457, Fall 2019

Drexel University, Department of Computer Science

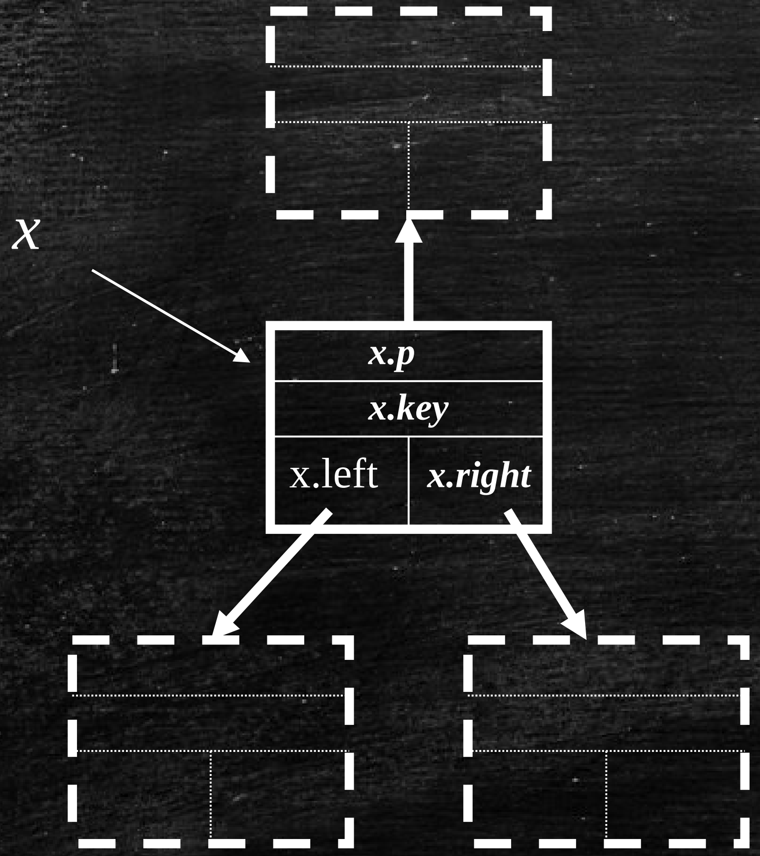
Lecture 13

Binary Trees and Induction

- Binary trees play a major role in computer science
  - E.g., heaps, binary search trees, red-black trees
  - It is important to have good intuition regarding their properties
- What is a complete (rooted) binary tree?
  - A binary tree in which all leaves have the same depth.
  - How many leaves does a complete binary tree of height h have? 2^h
  - How many nodes does a complete binary tree of height h have? $2^{h+1} - 1$
  - Can you prove this using induction?
- What is a full (rooted) binary tree?
  - A binary tree in which every node is either a leaf or has degree* exactly 2
  - How many leaves does a full binary tree with n nodes have?
  - Show that there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h in any n-node full binary tree

Binary Search Trees

- Each node x in a binary search tree (BST) contains:



- ***x.key*** - The value stored at x
- ***x.left*** - Pointer to left child of ***x***
- ***x.right*** - Pointer to right child of ***x***
- ***x.p*** - Pointer to parent of ***x***

Binary Search Tree Property

- Keys in BST satisfy the following properties:

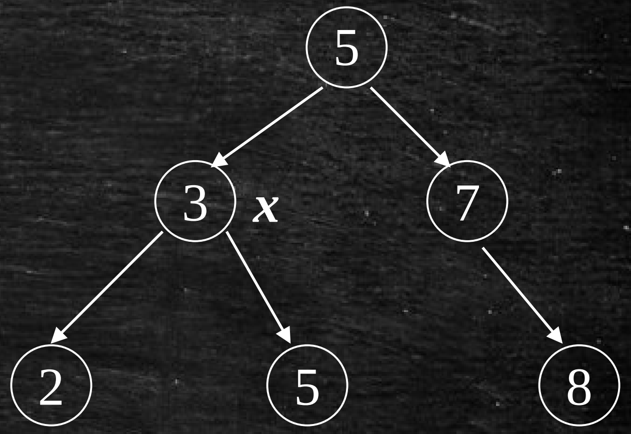
  Let ***x*** be a node in a BST:

  – If ***y*** is in the left subtree of ***x*** then:

  $$y.\boldsymbol{key} \le x.\boldsymbol{key}$$

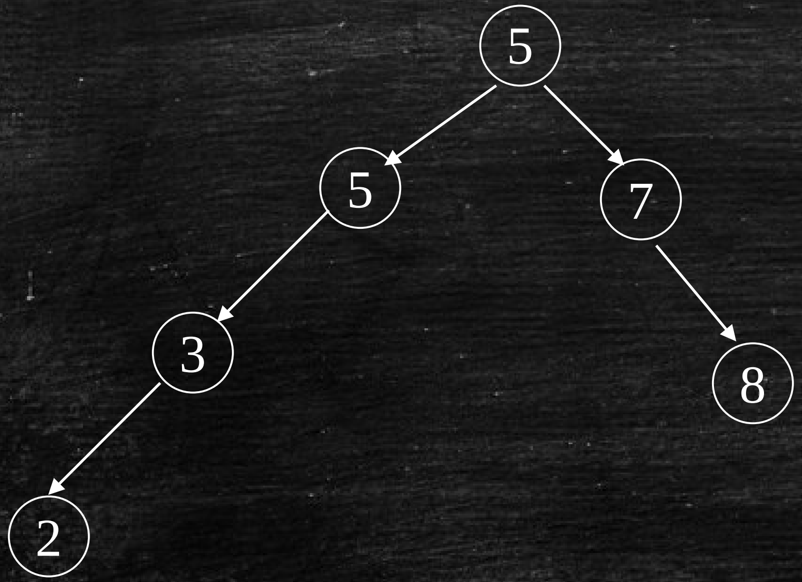
  – If ***y*** is in the right subtree of ***x*** then:

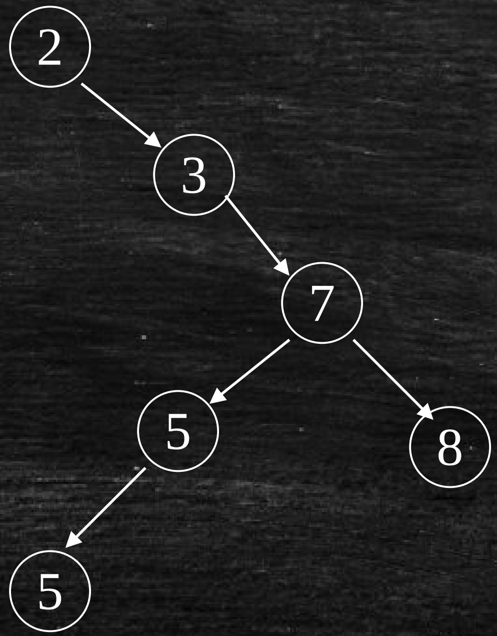
  $$y.\boldsymbol{key} > \boldsymbol{x.key}$$



Binary Search Tree Examples

- Two valid BST's for the keys: 2,3,5,5,7,8





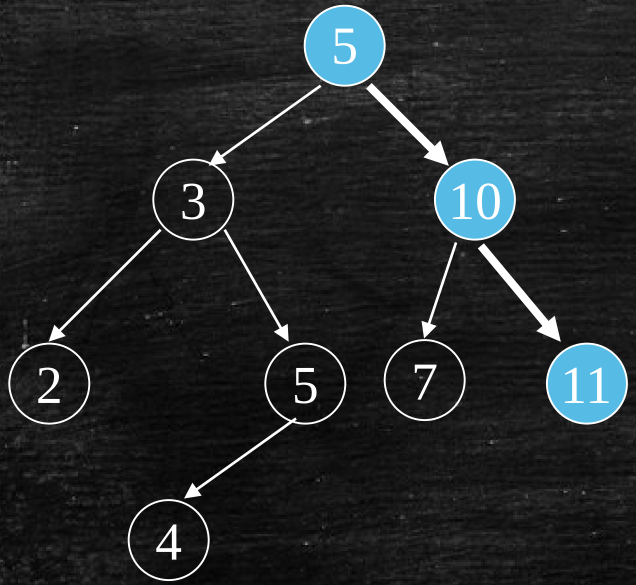
Searching in BST

- To find element with key k in tree T:
  - Compare k with the root
  - If k<**key**[**root**[T]] search for k in left subtree
  - Otherwise, search for k in right subtree

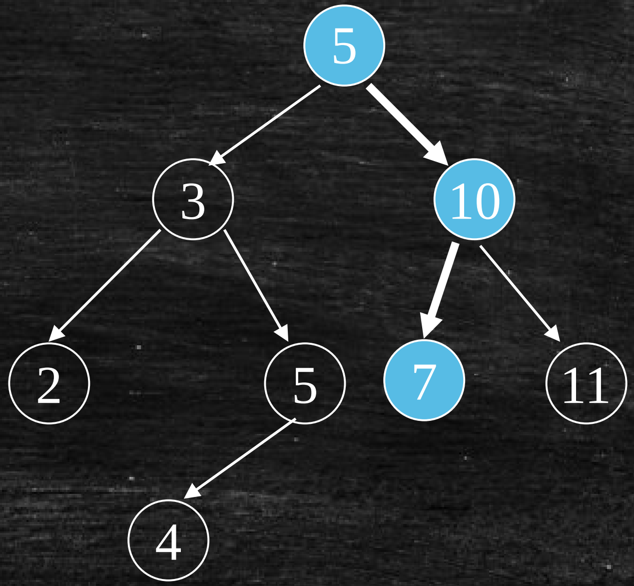
Search(T, k)
    x = root[T]
    if ***x*** **==** ***NIL***
        return("not found")
    if k == ***key***[x]
        return("found the key")
    if k < ***key***[x]
        Search(***left***[x], k)
    else
        Search(***right***[x], k)

Examples:

- Search(T, 11)

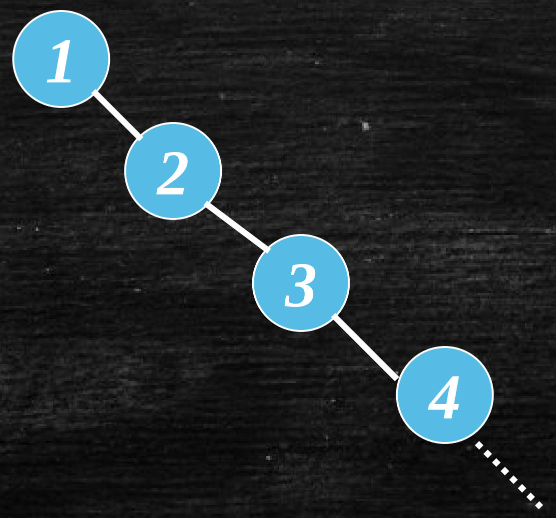


- Search(T, 6)



Analysis of Search

- Running time of search for a tree of height ***h*** is $O(h)$
- After insertion of n keys, worst case running time of search is $O(n)$
- We talked about expected height of binary search trees in recitation

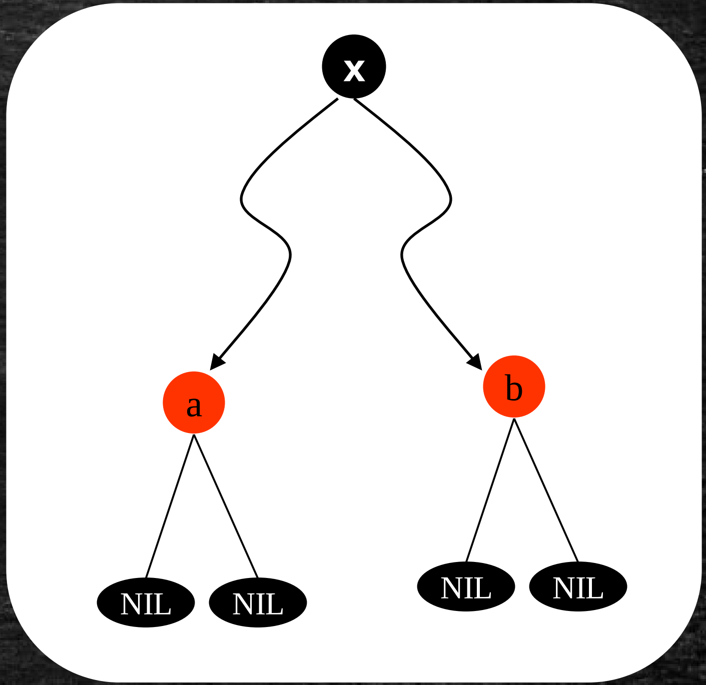


Red-black Trees

- They are balanced search trees (their height is $O(\log n)$)
- Most of the search and update operations on these trees take $O(\log n)$ time
- The structure is well balanced, i.e., each subtree is a balanced search tree.

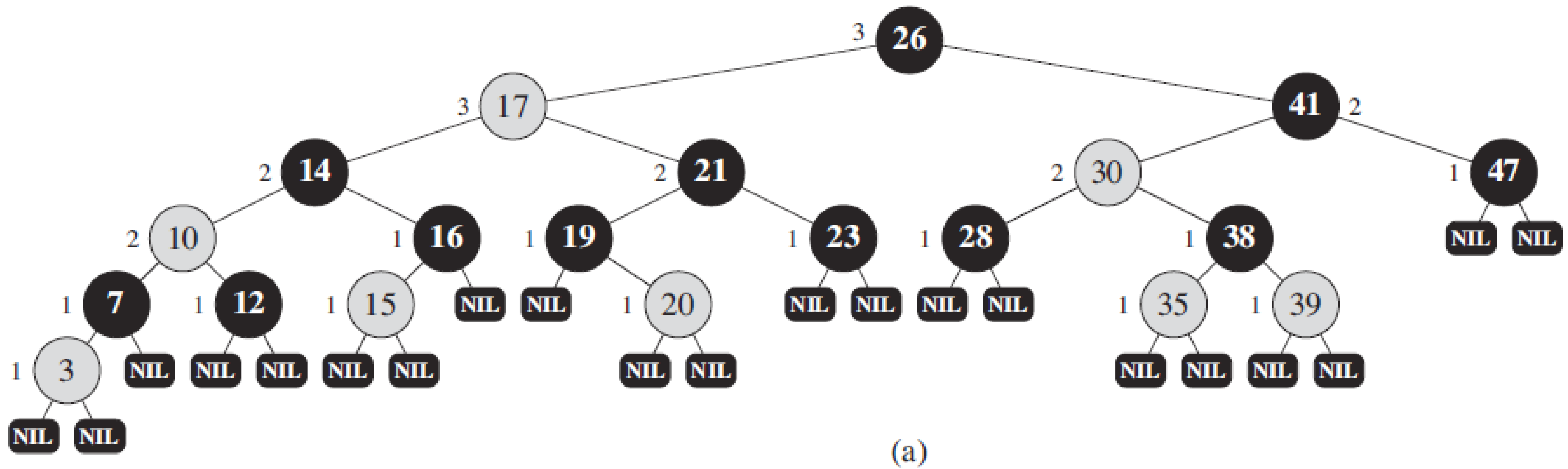
Red-black Trees

1. Every node is either red or black
2. The root is black
3. Every leaf (NIL) is black
4. If a node is red, both its children are black
5. All paths from a node x to a leaf have same number of black nodes (Black-Height($\boldsymbol{x}$))



Example

- A red-black tree with n keys has height at most $2\log(n+1)$
- Proof (Intuition): Merge the red nodes into their parents



(a)

Structural Induction Problem

Question: A rooted binary tree is a rooted tree in which each node has at most two children. A node of a tree is full if it contains a non-empty left child and a non-empty right child. Prove, using induction, that for any rooted binary tree, the number of full nodes is one less than the number of leaves.

Solution: For the base case, we consider the set of trees with 1 node. Clearly, there exists only one such tree that comprises of a single node and no edges. For this tree, the number of leaves is 1 and the number of full nodes is 0, so the number of full nodes is indeed exactly one less than the number of leaves in this case.

For the inductive step, we assume that for every rooted binary tree of size $n \geq 1$ the number of nodes with two children is exactly one less than the number of leaves, and our goal is to show that the same is true for *every* rooted binary tree of size $n+1$. To prove this step, consider *any* possible rooted binary tree T of size $n+1$, and remove one of its leaves, which reduces it to a tree T' of n nodes. If ℓ is the number of leaves of T', then we know, by our inductive hypothesis, that the number full nodes of T' is exactly $\ell - 1$. The leaf that we removed must have either been an only child of its parent, or one of two children. We consider both possibilities below.

- If the leaf that we removed was the only child of its parent, then its removal did not affect the number of full nodes and it did not affect the number of leaves either, since it removed a leaf and transformed its parent into one. Therefore, the number of leaves in T must have been ℓ as well, and the number of full nodes of T must have been $\ell - 1$, which is consistent with the statement that we are trying to prove.

- If, on the other hand, the leaf that we removed was the second child of its parent, then its removal decreased the number of leaves by one, but it also decreased the number of full nodes by one as well. Therefore, the number of leaves in the initial tree was $\ell + 1$ and the number of full nodes was ℓ, which is once again consistent with the statement that we are trying to prove.

Today’s Lecture

- Dynamic programming

Divide and Conquer

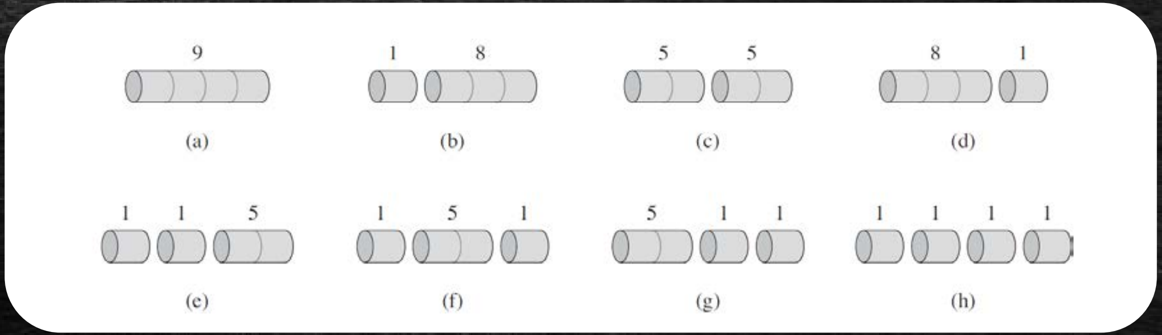
- **Divide and conquer** algorithm design approach
  - Divide the problem into smaller instances of same problem
  - Conquer the subproblems by solving them recursively until small enough
  - Combine solutions of the subproblems into solution for original problem

- We can analyze the running time using **recurrence equations**
  - E.g., for merge-sort: $T(n) = 2T(n/2) + n$ and $T(n) = 1$ for $n \leq 2$
  - We can solve such equations using master theorem, recursion-tree, or substitution method

- What about the recurrence equation for **Fibonacci numbers**?
  - $F_0 = 0$, $F_1 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$
  - $T(n) = T(n-1) + T(n-2) + 1$
  - This leads to $T(n)$=$\Theta(2^n)$. Can we do better than that?

Dynamic Programming

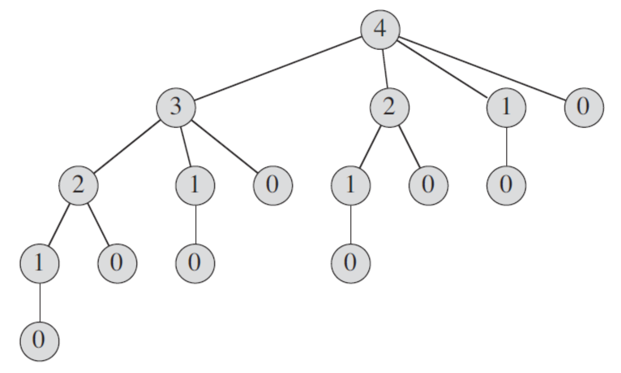
- Rod Cutting Problem
  - Given a rod of length n inches and a table of prices p_i for each $i = 1, \dots, n$ (price of a rod of i inches), determine the maximum revenue r_n obtainable by cutting up the rod and selling it in pieces.
  - E.g., say that the rod length is n=4 inches and the prices are p_1=1, p_2=5, p_3=8, and p_4=9



  - What would be a natural **greedy algorithm** for this problem?
    - Cutting a piece of size i with the largest p_i/i ratio and continue in the remaining rod of length $n-i$
    - Show that this algorithm does not always return the optimal solution
  - $r_n = \max(p_n,\ r_1 + r_{n-1}, \dots, r_{n-1} + r_1)$
  - $r_n = \max\limits_{1 \le i \le n} (p_i + r_{n-i})$

Dynamic Programming

- **Rod Cutting** Problem
  - What does the recursion tree look like for divide & conquer using equation $r_n = \max\limits_{1 \le i \le n}(p_i,\ r_{n-i})$?

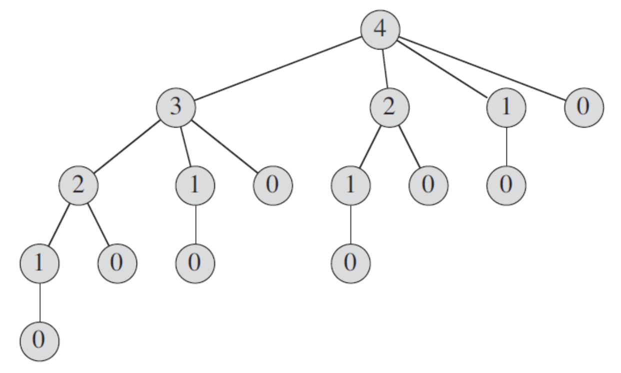



```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

  - E.g., say that the rod length is n=4 inches and the prices are p_1=1, p_2=5, p_3=8, and p_4=9

Dynamic Programming

- **Rod Cutting** Problem
  - What does the recursion tree look like for divide & conquer using equation $r_n = \max_{1 \le i \le n}(p_i,\ r_{n-i})$?

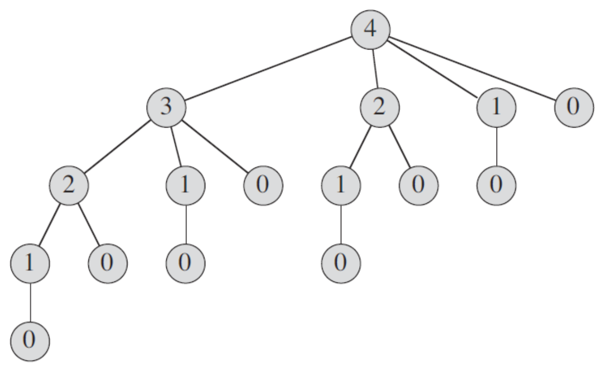



```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

  - No need to compute r_i again and again!

Dynamic Programming

- **Rod Cutting** Problem
  - What does the recursion tree look like for divide & conquer using equation $r_n = \max\limits_{1 \le i \le n}(p_i,\ r_{n-i})$?



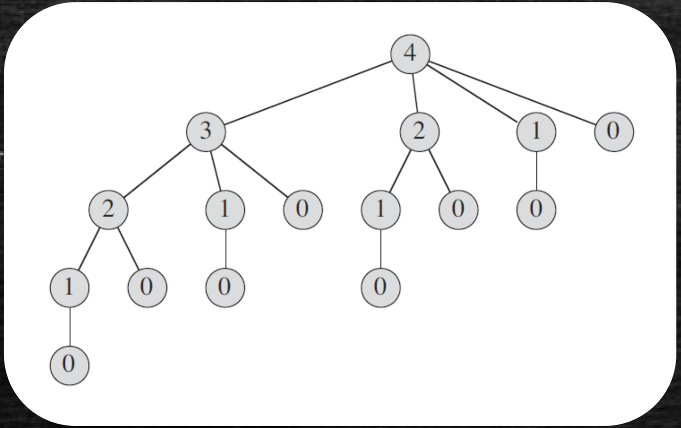
```
MEMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

Initially all the values in r are negative

  - No need to compute r_i again and again!
  - Approach 1: **top-down** with **memoization** (save as you go)

Dynamic Programming

- **Rod Cutting** Problem
  - What does the recursion tree look like for divide & conquer using equation $r_n = \max_{1 \le i \le n} (p_i,\ r_{n-i})$?

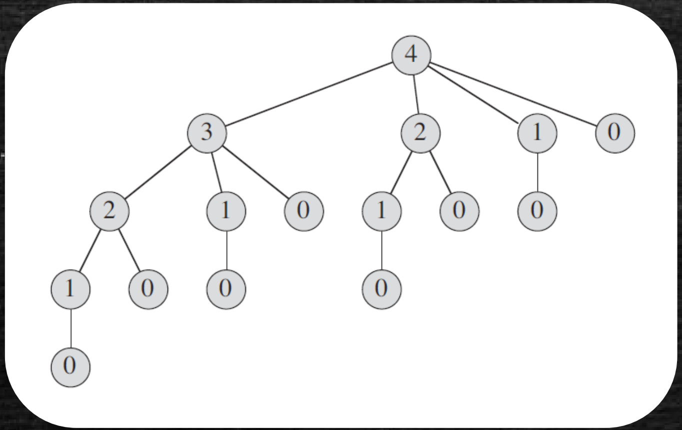


```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

  - No need to compute r_i again and again!
  - Approach 1: **top-down** with **memoization** (save as you go)
  - Approach 2: **bottom-up** (save from small to big subproblems)

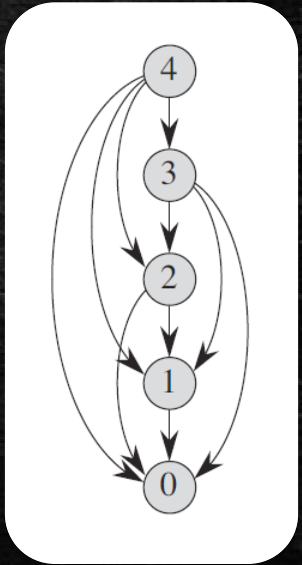
Dynamic Programming

- **Rod Cutting** Problem
  - What does the recursion tree look like for divide & conquer using equation $r_n = \max_{1 \le i \le n}(p_i,\ r_{n-i})$?



  - No need to compute r_i again and again!
  - Approach 1: **top-down** with **memoization** (save as you go)
  - Approach 2: **bottom-up** (save from small to big subproblems)

  - We can represent the dependence using the **subproblem graph**:
  - To reconstruct a solution, we can keep track of the optimal choice in each case



Maximum Subarray Problem

- You are given an array A of n numbers (both positive and negative)
  - Find a contiguous subarray with the maximum sum of numbers
  - In other words: find i, j such that $1 \leq i \leq j \leq n$ and maximize $\sum_{x=i}^{j} A[x]$
  - For example, consider the following array:

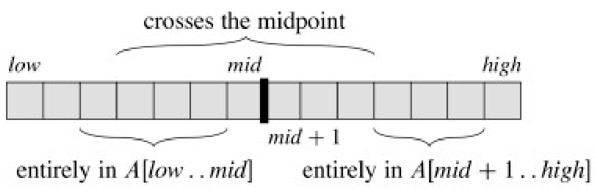
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

maximum subarray

  - What is the first, simple, algorithm that comes to mind?
  - What is the running time of this algorithm?
  - Can you come up with a divide & conquer algorithm?
    - How can we analyze the (worst case) running time of such algorithms?

Maximum Subarray Problem

```
FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
// Find a maximum subarray of the form A[i .. mid].
left-sum = -∞
sum = 0
for i = mid downto low
    sum = sum + A[i]
    if sum > left-sum
        left-sum = sum
        max-left = i
// Find a maximum subarray of the form A[mid + 1 .. j].
right-sum = -∞
sum = 0
for j = mid + 1 to high
    sum = sum + A[j]
    if sum > right-sum
        right-sum = sum
        max-right = j
// Return the indices and the sum of the two subarrays.
return (max-left, max-right, left-sum + right-sum)
```



(a)

Maximum Subarray Problem

Divide-and-conquer procedure for the maximum-subarray problem

```
FIND-MAXIMUM-SUBARRAY(A, low, high)
if high == low
    return (low, high, A[low])          // base case: only one element
else mid = ⌊(low + high)/2⌋
    (left-low, left-high, left-sum) =
        FIND-MAXIMUM-SUBARRAY(A, low, mid)
    (right-low, right-high, right-sum) =
        FIND-MAXIMUM-SUBARRAY(A, mid + 1, high)
    (cross-low, cross-high, cross-sum) =
        FIND-MAX-CROSSING-SUBARRAY(A, low, mid, high)
    if left-sum ≥ right-sum and left-sum ≥ cross-sum
        return (left-low, left-high, left-sum)
    elseif right-sum ≥ left-sum and right-sum ≥ cross-sum
        return (right-low, right-high, right-sum)
    else return (cross-low, cross-high, cross-sum)
```

Initial call: FIND-MAXIMUM-SUBARRAY$(A, 1, n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

Maximum Subarray Problem

- Can you provide a dynamic programming solution to this problem?
  - How would you break this problem into sub-problems?
  - You want the solutions to the subproblems to help you solve larger subproblems faster
- If you knew the best subarray ending at $A[i]$, could you find the best subarray ending at $A[i+1]$?
  - It is either the best subarray ending at $A[i]$ plus $A[i+1]$, or just $A[i+1]$
  - How quickly can you check which one is best?
  - How quickly can you compute the best subarray ending at $A[i]$ for every $i \in \{1, \ldots, n\}$?
  - Can you compute the optimal solution using this information?

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

maximum subarray