CS 457, Fall 2019

Drexel University, Department of Computer Science

Lecture 14

Divide and Conquer

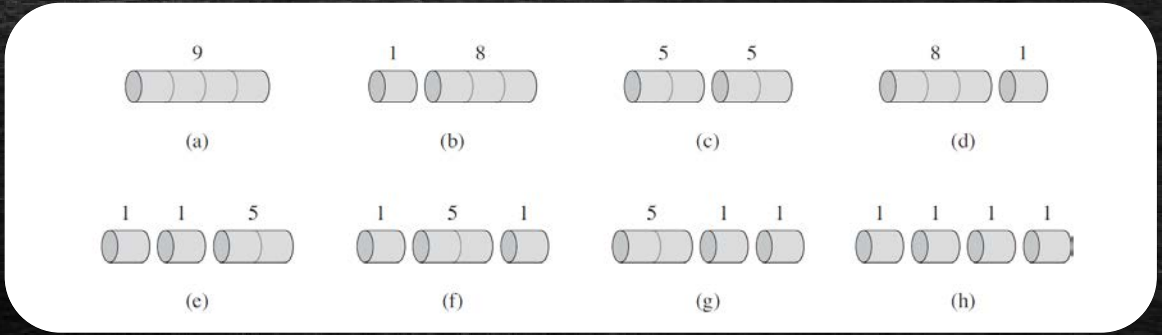
- **Divide and conquer** algorithm design approach
  - Divide the problem into smaller instances of same problem
  - Conquer the subproblems by solving them recursively until small enough
  - Combine solutions of the subproblems into solution for original problem

- 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

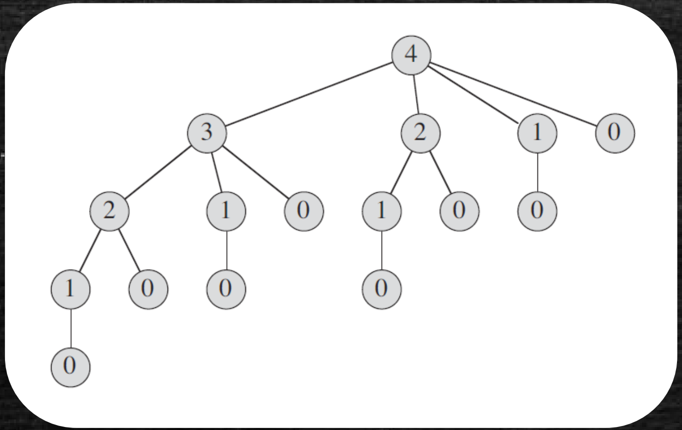
- **Rod Cutting** Problem
  - Given a rod of length n inches and a table of prices p_i for each $i = 1, \ldots, n$ (price of a rod of i inches), determine the maximum revenue r_n obtainable by cutting up the rod and selling it in pieces.
  - E.g., say that the rod length is n=4 inches and the prices 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
  - $r_n = \max(p_n,\ r_1 + r_{n-1}, \ldots, r_{n-1} + r_1)$
  - $r_n = \max\limits_{1 \le i \le n} (p_i + r_{n-i})$

Dynamic Programming

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

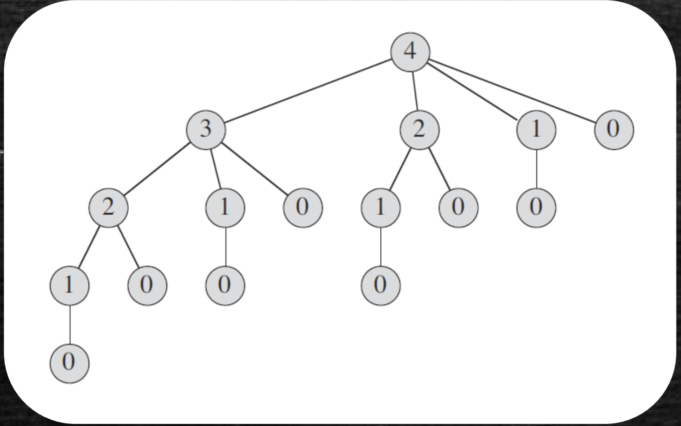



```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
return q
```

  - 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 $r_n = \max_{1 \le i \le n}(p_i,\ r_{n-i})$?



MEMOIZED-CUT-ROD-AUX(p, n, r)

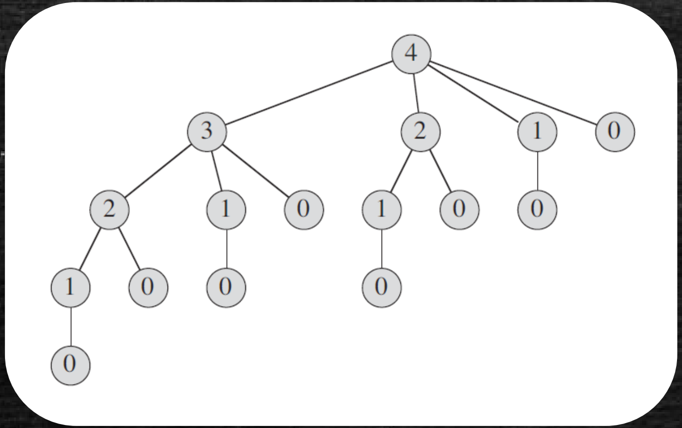
```
if r[n] ≥ 0
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n
        q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
r[n] = q
return q
```

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 $r_n = \max_{1 \le i \le n}(p_i, r_{n-i})$?

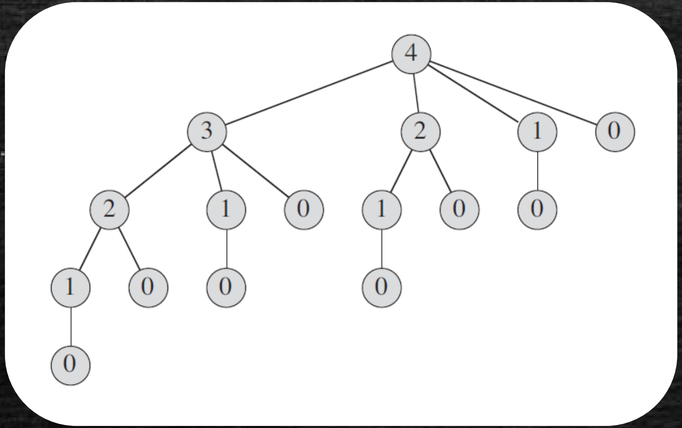


```
BOTTOM-UP-CUT-ROD(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n
    q = −∞
    for i = 1 to j
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

  - 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)

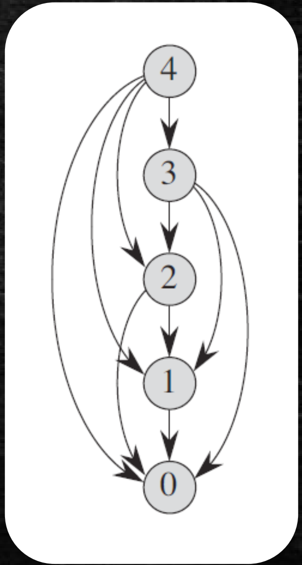
Dynamic Programming

- **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 $1 \le i \le j \le n$ and maximize $\sum_{x=i}^{j} A[x]$
  - For example, consider the following array:

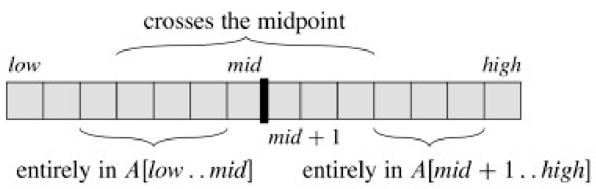
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	13	–3	–25	20	–3	–16	–23	18	20	–7	12	–5	–22	15	–4	7

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 $A[i+1]$?
  - Let's create a new array B and store the sum of the best subbary ending at $A[i]$ in $B[i]$
  - The best ending at $A[i+1]$ is either the best ending at $A[i]$ plus $A[i+1]$, or just $A[i+1]$
  - This means that $B[i+1] = \max\{B[i] + A[i+1],\ A[i+1]\}$
  - How quickly can you check which one is best?
  - How quickly can you compute the best subarray ending at $A[i]$ for every $i \in \{1, \dots, n\}$?
  - 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

Matrix-Chain Multiplication

- Given sequence $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices, compute the product $A_1 A_2 \cdots A_n$
  - Matrix multiplication is associative so all parenthesizations yield the same product
  - But, do they all take the same amount of time?
  - E.g., say that n=3 and the dimensions are 10x100, 100x5, and 5x50

```
MATRIX-MULTIPLY(A, B)
1  if A.columns ≠ B.rows
2      error "incompatible dimensions"
3  else let C be a new A.rows × B.columns matrix
4      for i = 1 to A.rows
5          for j = 1 to B.columns
6              c_ij = 0
7              for k = 1 to A.columns
8                  c_ij = c_ij + a_ik · b_kj
9      return C
```

If A is p x q and B is q x r, then step 8 is executed pqr times

- Find the optimal multiplication order using dynamic programming

Matrix-Chain Multiplication

- The minimum cost of parenthesizing the product $A_i A_{i+1} \cdots A_j$ (where A_i is $p_{i-1} \times p_i$) is

$$m[i,j] = \begin{cases} 0 & \text{if } i = j , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j . \end{cases}$$

- Running the recursive algorithm would require exponential time
- But, how many sub-problems have we defined?
  - One for each pair (i, j) so $\Theta(n^2)$
- Input is $p = \langle p_0, p_1, \ldots, p_n \rangle$ where p_{i-1}x p_i are the dimensions of matrix A_i
- Output is table m and table s
  - Table m stores cost of each $m[i,j]$
  - Table s records index of k that achieved that cost

```
MATRIX-CHAIN-ORDER(p)
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n
    m[i, i] = 0
for l = 2 to n              // l is the chain length
    for i = 1 to n − l + 1
        j = i + l − 1
        m[i, j] = ∞
        for k = i to j − 1
            q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
            if q < m[i, j]
                m[i, j] = q
                s[i, j] = k
return m and s
```