Today:
- Matrix Subarray (Divide & Conquer)
- Intro to Dynamic Programming
  (Rod cutting)

COSC 581, Algorithms

January 21, 2014

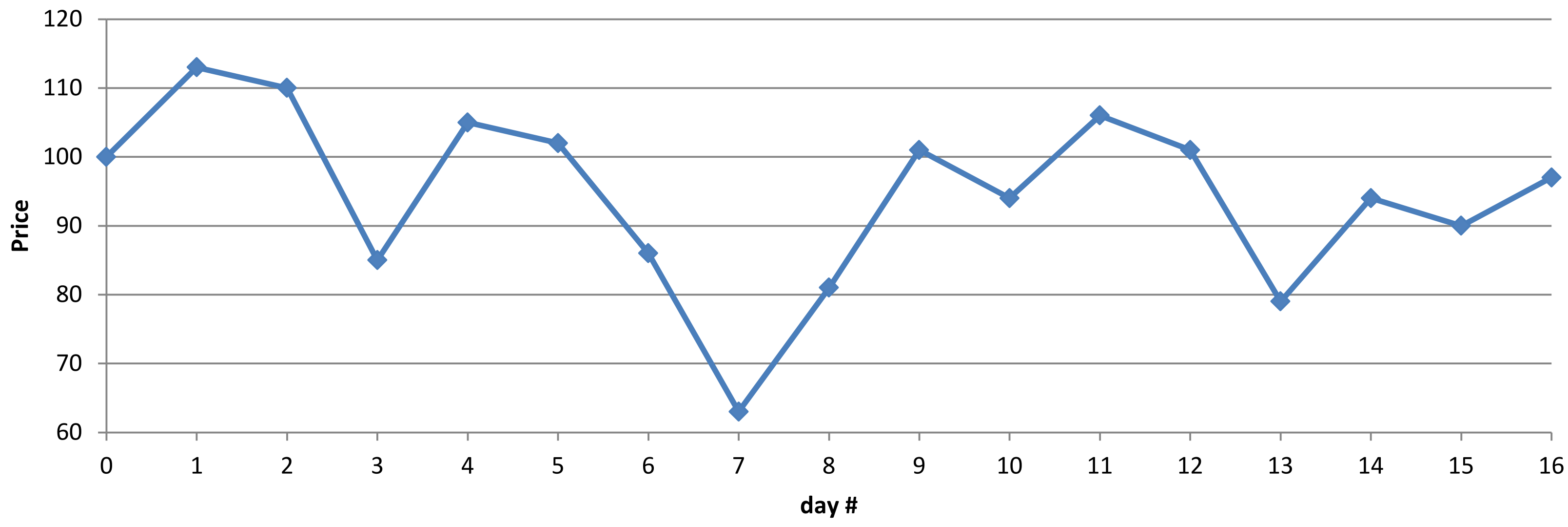
Reading Assignments

- Today's class:
  - Chapter 4.1, 15.1

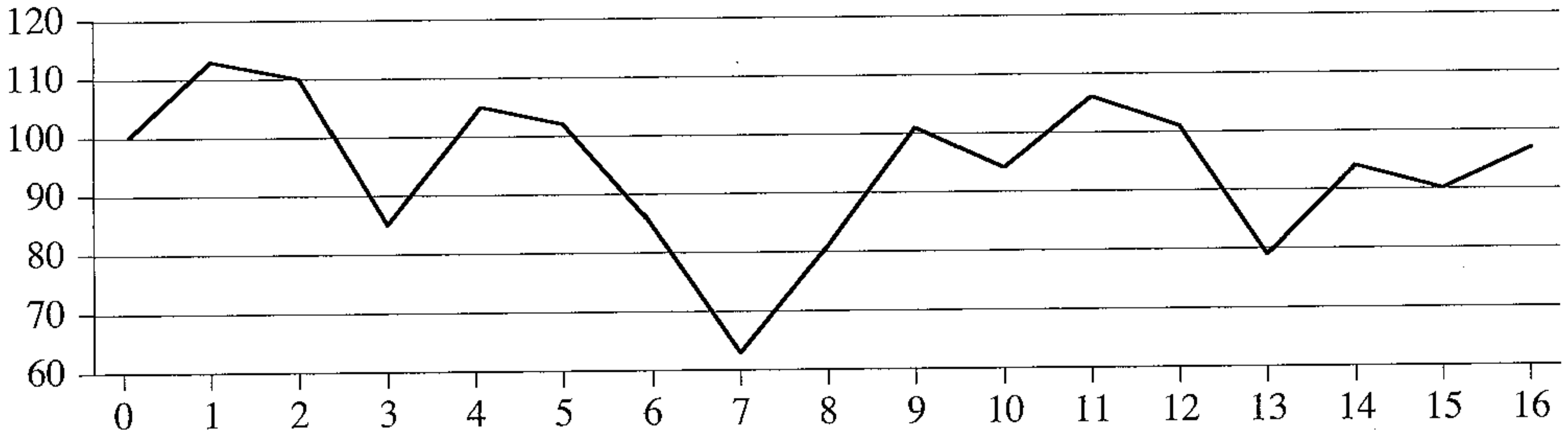
- Reading assignment for next class:
  - Chapter 15.2

Maximum-subarray problem (Another Divide & Conquer problem)

- If you know the price of certain stock from day *i* to day *j*;
- You can only buy and sell one share once
- How to maximize your profit?



Maximum-Subarray Example

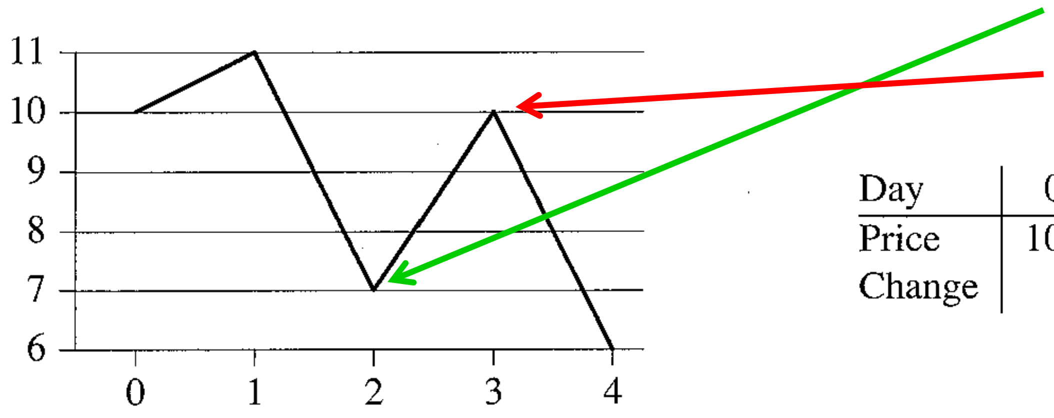


Day	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Price	100	113	110	85	105	102	86	63	81	101	94	106	101	79	94	90	97
Change		13	−3	−25	20	−3	−16	−23	18	20	−7	12	−5	−22	15	−4	7

Figure 4.1 Information about the price of stock in the Volatile Chemical Corporation after the close of trading over a period of 17 days. The horizontal axis of the chart indicates the day, and the vertical axis shows the price. The bottom row of the table gives the change in price from the previous day.

Maximum-Subarray Example

Buying low and selling high doesn't always work



But doesn't follow "buy low, sell high" rule

Day	0	1	2	3	4
Price	10	11	7	10	6
Change		1	−4	3	−4

Figure 4.2 An example showing that the maximum profit does not always start at the lowest price or end at the highest price. Again, the horizontal axis indicates the day, and the vertical axis shows the price. Here, the maximum profit of \$3 per share would be earned by buying after day 2 and selling after day 3. The price of \$7 after day 2 is not the lowest price overall, and the price of \$10 after day 3 is not the highest price overall.

Maximum-subarray problem

- What is the **brute-force** solution?

```
max = -infinity;
for each day pair p {
   if(p.priceDifference>max)
         max=p.priceDifference;
}
```

Time complexity?

Maximum-subarray problem

- What is the **brute-force** solution?

```
max = -infinity;
for each day pair p {
   if(p.priceDifference>max)
         max=p.priceDifference;
}
```

Time complexity? $\binom{n}{2}$ pairs, so $O(n^2)$

How to solve more efficiently?

- If we know the price difference of each 2 contiguous days
- The solution can be found from the **maximum-subarray**
- **Maximum-subarray** of array A is:
  - A subarray of A
  - Nonempty
  - Contiguous
  - Whose values have the largest sum

Examine subarrays

Day	0	1	2	3	4
Price	10	11	7	10	6
Difference		1	-4	3	-4

Remember best solution: Buy on day 2, sell on day 3

Examine the differences across subarrays (some examples):

Sub-array	0-1	0-2	0-3	0-4	2-2	2-3	2-4	3-3	3-4	4-4
Difference	1	-3	0	-4	-4	-1	-5	3	-1	-4

Divide-and-Conquer Approach

- How to divide?
  - Divide into 2 arrays
- What is the base case?
- How to combine the subproblem solutions?

Divide-and-Conquer Approach

- Note where solution must lie:

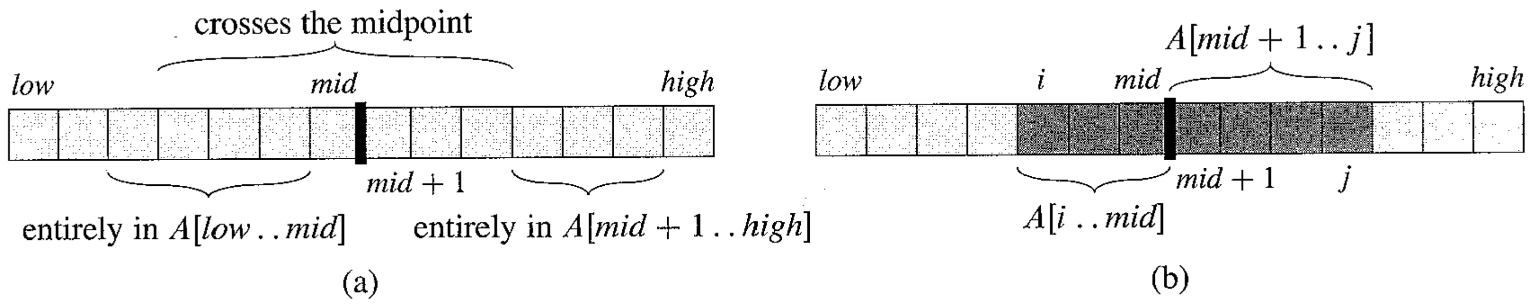


Figure 4.4 **(a)** Possible locations of subarrays of $A[low\,..\,high]$: entirely in $A[low\,..\,mid]$, entirely in $A[mid+1\,..\,high]$, or crossing the midpoint *mid*. **(b)** Any subarray of $A[low\,..\,high]$ crossing the midpoint comprises two subarrays $A[i\,..\,mid]$ and $A[mid+1\,..\,j]$, where $low \le i \le mid$ and $mid < j \le high$.

- 3 choices:
    - A[i, …, mid] // best is in the left array
    - A[mid+1, …, j] // best is in the right array
    - A[…, mid, mid+1….] // best is in the array across the midpoint
  - The maximum subarray for A[i,…,j] is the best of these 3 choices

Maximum-subarray problem – divide-and-conquer algorithm

Input: array A[i, …, j]

Ouput: sum of maximum-subarray, start point of maximum-subarray, end point of maximum-subarray

```
FindMaxSubarray:
1.  if(j<=i) return (A[i], i, j);
2.  mid = floor(i+j);
3.  (sumCross, startCross, endCross) =
    FindMaxCrossingSubarray(A, i, j, mid);
4.  (sumLeft, startLeft, endLeft) = FindMaxSubarray(A, i, mid);
5.  (sumRight, startRight, endRight) = FindMaxSubarray(A, mid+1, j);
6.  Return the largest of these 3
```

Maximum-subarray problem – divide-and-conquer algorithm

Input: array A[i, …, j]

Ouput: sum of maximum-subarray, start point of maximum-subarray, end point of maximum-subarray

FindMaxSubarray:

```
1. if(j<=i) return (A[i], i, j);
2. mid = floor(i+j);
3. (sumCross, startCross, endCross) =
   FindMaxCrossingSubarray(A, i, j, mid);
4. (sumLeft, startLeft, endLeft) = FindMaxSubarray(A, i, mid);
5. (sumRight, startRight, endRight) = FindMaxSubarray(A, mid+1, j);
6. Return the largest of these 3
```

Time complexity?

Maximum-subarray problem – divide-and-conquer algorithm

Input: array A[i, …, j]

Ouput: sum of maximum-subarray, start point of maximum-subarray, end point of maximum-subarray

```
FindMaxSubarray:
1.  if(j<=i) return (A[i], i, j);
2.  mid = floor(i+j);
3.  (sumCross, startCross, endCross) =
    FindMaxCrossingSubarray(A, i, j, mid);
4.  (sumLeft, startLeft, endLeft) = FindMaxSubarray(A, i, mid);
5.  (sumRight, startRight, endRight) = FindMaxSubarray(A, mid+1, j);
6.  Return the largest of these 3
```

Time complexity? $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

Maximum-subarray problem – divide-and-conquer algorithm

Input: array A[i, …, j]

Ouput: sum of maximum-subarray, start point of maximum-subarray, end point of maximum-subarray

```
FindMaxSubarray:
1.  if(j<=i) return (A[i], i, j);
2.  mid = floor(i+j);
3.  (sumCross, startCross, endCross) =
    FindMaxCrossingSubarray(A, i, j, mid);
4.  (sumLeft, startLeft, endLeft) = FindMaxSubarray(A, i, mid);
5.  (sumRight, startRight, endRight) = FindMaxSubarray(A, mid+1, j);
6.  Return the largest of these 3
```

Time complexity? $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$ = $\Theta(n \lg n)$

In-Class Exercise for Divide & Conquer

- Suppose you are given a complete binary tree of height *h* with $n = 2^h$ leaves. [Here, we'll assume that the tree is completely filled in at all levels, including the deepest level.]
- Each node and each leaf, *x*, in the tree has an associated "value" *v*(*x*), which is an arbitrary real number.
- If *x* is a leaf, we denote by *A*(*x*) the set of ancestors of *x* (including *x* as one of its own ancestors). That is, *A*(*x*) consists of *x, x*'s parent, grandparent, etc., up to the root of the tree.
- Let *f*(*x*) be the sum of the values of *A*(*x*) – that is, $f(x) = \sum_{y \in A(x)} v(y)$.
- Presume we have the functions *left*(*x*), *right*(*x*), and *parent*(*x*), which return pointers to the left child, right child, and parent of node *x*, respectively. These functions return *nil* when no such node exists.

Give an efficient algorithm that finds the maximum value of *f*(*x*) across all leaves *x* of the tree. Note that we do not need to know which set of ancestors, *A*(*x*), sums to this maximum total; we only need to know its value.

Dynamic Programming

- ***Dynamic programming*** is typically applied to optimization problems. In such problem there can be ***many solutions***. Each solution has a value, and we wish to find *a* ***solution*** with the optimal value.

Optimization

This, generally, refers to classes of problems that possess multiple solutions at one level, and where we have a real-valued function defined on the solutions.

Problem: find a solution that minimizes or maximizes the value of this function.

Note: there is no guarantee that such a solution will be unique and, moreover, there is no guarantee that you will find it (local maxima) unless the search is over a small enough search space or the function is restricted enough.

Optimization

Question: are there classes of problems for which you can guarantee an optimizing solution can be found?

Optimization

Question: are there classes of problems for which you can guarantee an optimizing solution can be found?

Answer: yes. BUT you also need to find such a solution in a "reasonable" amount of time.

We are going to look at two classes of problems, and the techniques that will succeed in constructing their solutions in a "reasonable" (i.e., low degree polynomial in the size of the initial data) amount of time.

Optimization

We begin with a rough comparison that contrasts a method you are familiar with (divide and conquer) and the method (still unspecified) of Dynamic Programming (developed by Richard Bellman in the late 1940's and early 1950's).

Two Algorithmic Models:

	Divide & Conquer	Dynamic Programming
View problem as collection of subproblems	✓	✓
"Recursive" in nature	✓	✓
Independent subproblems	✓	
Overlapping subproblems		✓
Number of subproblems	depends on partitioning factors	typically small
Preprocessing characteristic running time	Typically log function of n	depends on number and difficulty of subproblems
Primarily for optimization problems		✓
Optimal substructure: ***optimal solution to problem contains within it optimal solutions to subproblems***		✓

The Primary Steps of Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom up fashion.
4. Construct an optimal solution from computed information.

Example: Rod Cutting

- You are given a rod of length $n \geq 0$ (n in inches)
- A rod of length i inches will be sold for p_i dollars
- Cutting is free (simplifying assumption)
- **Problem**: given a table of prices p_i determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

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

Example: Rod Cutting

We can see immediately (from the values in the table) that

$$n \le p_n \le 3n.$$

This is not very useful because:

- The range of potential revenues is very large
- Our finding quick upper and lower bounds depends on finding quickly the minimum and maximum p_i/i ratios (one pass through the table), but then we are back to the point above….

Example: Rod Cutting

Step 1: Characterizing an Optimal Solution

Question: in how many different ways can we cut a rod of length *n*?

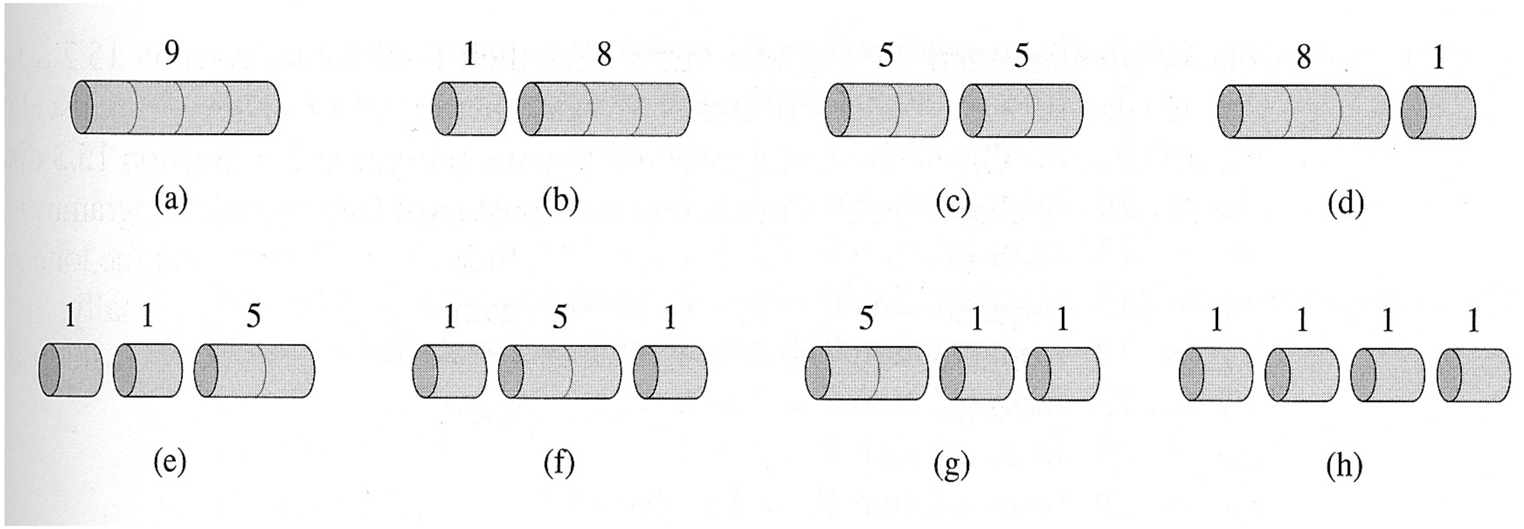
For a rod of length 4:

Example: Rod Cutting

Step 1: Characterizing an Optimal Solution

Question: in how many different ways can we cut a rod of length *n*?

For a rod of length 4:



Options: $2^{4-1} = 2^3 = 8$

For a rod of length *n*: 2^{n-1}. **Exponential**: we cannot try all possibilities for *n* "large". The obvious exhaustive approach won't work.

Example: Rod Cutting

Step 1: Characterizing an Optimal Solution

Question: in how many different ways can we cut a rod of length *n*?

Proof Details: a rod of length *n* can have exactly *n*-1 possible cut positions – choose 0 ≤ *k* ≤ *n*-1 actual cuts. We can choose the *k* cuts (without repetition) anywhere we want, so that for each such *k* the number of different choices is

$$\binom{n-1}{k}$$

When we sum up over all possibilities (*k* = 0 to *k* = *n*-1):

$$\sum_{k=0}^{n-1}\binom{n-1}{k}=\sum_{k=0}^{n-1}\frac{(n-1)!}{k!(n-1-k)!}=(1+1)^{n-1}=2^{n-1}.$$

For a rod of length *n*: 2^{n-1}.

Example: Rod Cutting

Characterizing an Optimal Solution

Let us find a way to solve the problem recursively (we might be able to modify the solution so that the maximum can be actually computed):

Assume we have cut a rod of length n into $0 \leq k \leq n$ pieces of length $i_1, \ldots, i_k$,

$$n = i_1 + \ldots + i_k,$$

with revenue:

$$r_n = p_{i1} + \ldots + p_{ik}$$

Assume further that this solution is optimal.

How can we construct it?

Advice: when you don't know what to do next, start with a simple example and hope something will occur to you… ☺

Example: Rod Cutting

Characterizing an Optimal Solution

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

We begin by constructing (by hand) the optimal solutions for i = 1, …, 10:

r_1 = 1	from soln. 1 = 1 (no cuts)
r_2 = 5	from soln. 2 = 2 (no cuts)
r_3 = 8	from soln. 3 = 3 (no cuts)
r_4 = 10	from soln. 4 = 2 + 2
r_5 = 13	from soln. 5 = 2 + 3
r_6 = 17	from soln. 6 = 6 (no cuts)
r_7 = 18	from soln. 7 = 1 + 6 or 7 = 2 + 2 + 3
r_8 = 22	from soln. 8 = 2 + 6
r_9 = 25	from soln. 9 = 3 + 6
r_{10} = 30	from soln. 10 = 10 (no cuts)

Example: Rod Cutting

Characterizing an Optimal Solution

Notice that in some cases $r_n = p_n$, while in other cases the optimal revenue r_n is obtained by cutting the rod into smaller pieces.

In ALL cases we have the recursion

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

exhibiting **optimal substructure**

A slightly different way of stating the same recursion, which avoids repeating some computations, is

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

This latter relation can be implemented as a simple top-down recursive procedure:

```
CUT-ROD($p, n$)
if $n == 0$
    return 0
$q = -\infty$
for $i = 1$ to $n$
    $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$
return $q$
```

Example: Rod Cutting
Characterizing an Optimal Solution

Time Out: How to justify the step from:

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$$

to

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

Note: every optimal partitioning of a rod of length n has a first cut – a segment of, say, length i. The optimal revenue, r_n, must satisfy $r_n = p_i + r_{n-i}$, where r_{n-i} is the optimal revenue for a rod of length $n – i$. If the latter were not the case, there would be a better partitioning for a rod of length $n – i$, giving a revenue $r'_{n–i} > r_{n-i}$ and a total revenue $r'_n = p_n + r'_{n-i} > p_i + r_{n-i} = r_n$.

Since we do not know which one of the leftmost cut positions provides the largest revenue, we just maximize over all the possible first cut positions.

Example: Rod Cutting
Characterizing an Optimal Solution

We can also notice that all the items we choose the maximum of are optimal in their own right: each substructure (max revenue for rods of lengths 1, …, *n*-1) is also optimal (again, **optimal substructure property**).

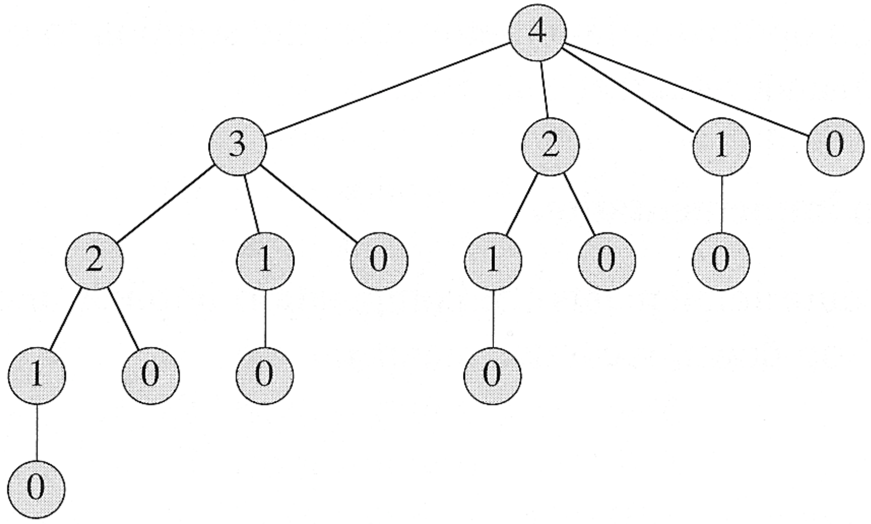
Nevertheless, we are still in trouble: computing the recursion leads to recomputing a number (= overlapping subproblems) of values – how many?

Example: Rod Cutting

Characterizing an Optimal Solution

Let’s call Cut-Rod(p, 4), to see the effects on a simple case:

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



The number of nodes for a tree corresponding to a rod of size *n* is:

$$T(0)=1, \quad T(n)=1+\sum_{j=0}^{n-1}T(j)=2^n,\ n\geq 1.$$

Example: Rod Cutting
Beyond Naïve Time Complexity

We have a problem: “reasonable size” problems are not solvable in “reasonable time” (but, in this case, they are solvable in “reasonable space”).

Specifically:
• Note that navigating the whole tree requires 2^n work.
• Note also that no more than $n + 1$ subproblems are active at any one time and that no more than $n + 1$ different values need to be computed or used.

Can we exploit these observations?
A standard solution method involves saving the values associated with each $T(j)$, so that we compute each value only once (called “**memoizing**” = writing yourself a memo).

Example: Rod Cutting
Naïve Caching

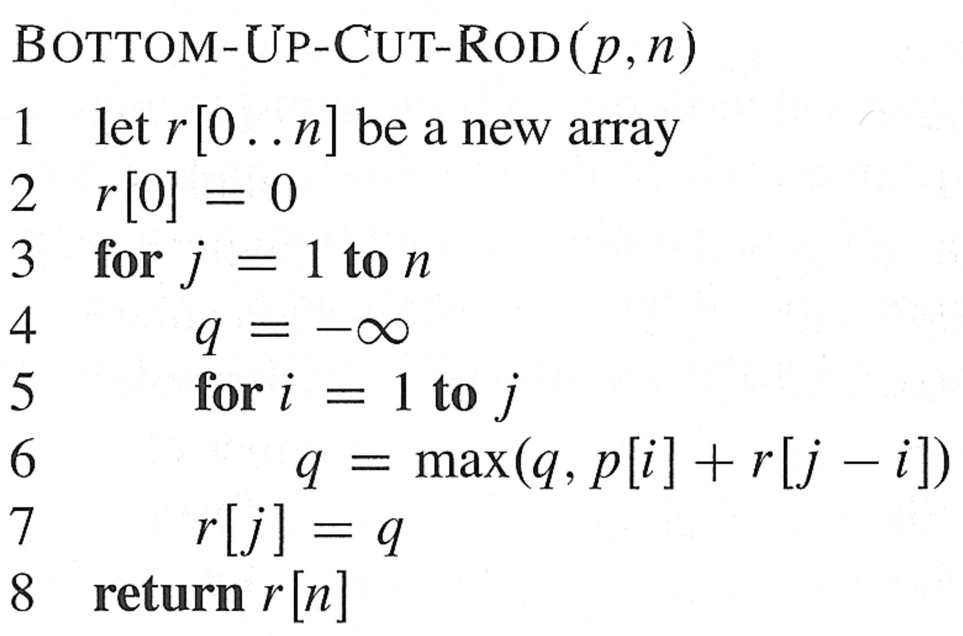
We introduce two procedures:

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

```
MEMOIZED-CUT-ROD-AUX$(p, n, r)$
if $r[n] \geq 0$
    return $r[n]$
if $n == 0$
    $q = 0$
else $q = -\infty$
    for $i = 1$ to $n$
        $q = \max(q, p[i] + $ MEMOIZED-CUT-ROD-AUX$(p, n - i, r))$
$r[n] = q$
return $q$
```

Example: Rod Cutting

More Sophisticated Caching By Solving Bottom-Up



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

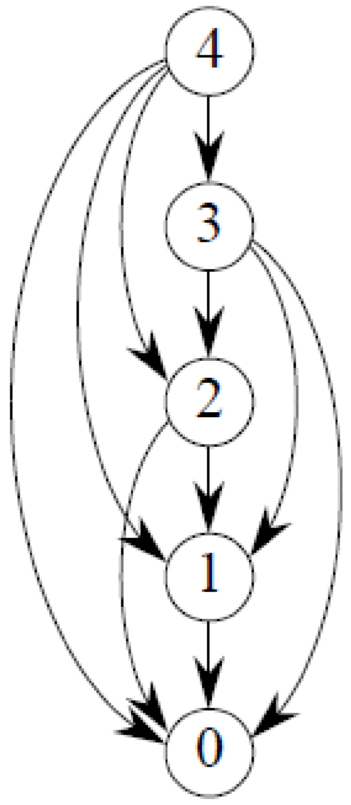
Example: Rod Cutting
Time Complexity

Whether we solve the problem in a top-down or bottom-up manner the asymptotic time is $\Theta(n^2)$, the major difference being recursive calls as compared to loop iterations.

Why??

Handy Tool: Subproblem graphs

- For rod-cutting problem with n = 4
  - Subprogram graph is a directed graph
    - One vertex for each distinct subproblem.
    - Has a directed edge (x, y) if computing an optimal solution to subproblem x *directly* requires knowing an optimal solution to subproblem y.



Subproblem graphs

- Can think of the subproblem graph as a collapsed version of the tree of recursive calls, where all nodes for the same subproblem are collapsed into a single vertex, and all edges go from parent to child.
- Subproblem graph can help determine running time. Because we solve each subproblem just once, running time is sum of times needed to solve each subproblem.
  - Time to compute solution to a subproblem is typically linear in the out-degree (number of outgoing edges) of its vertex.
  - Number of subproblems equals number of vertices.
- When these conditions hold, running time is linear in number of vertices and edges.

Reconstructing a solution

- So far, have focused on computing the value of an optimal solution, rather than the *choices* that produced an optimal solution.
- Extend the bottom-up approach to record not just optimal values, but optimal choices. Save the optimal choices in a separate table. Then use a separate procedure to print the optimal choices.

Reconstructing a solution

```
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 $n$
    $q = -\infty$
    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 $s$
```

Saves the first cut made in an optimal solution for a problem of size i in $s[i]$.

Reconstructing a solution

To print out the cuts made in an optimal solution:

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]
    n = n − s[n]
```

Example: For the example, EXTENDED-BOTTOM-UP-CUT-ROD returns

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

A call to PRINT-CUT-ROD-SOLUTION$(p, 8)$ calls EXTENDED-BOTTOM-UP-CUT-ROD to compute the above r and s tables. Then it prints 2, sets n to 6, prints 6, and finishes (because n becomes 0).

In-Class Exercise

- Draw the recursion tree for the MERGE-SORT procedure on an array of 16 elements. (Each node of the recursion tree should simply indicate which elements of the array are being solved at that node.)
- Explain why memoization is ineffective in speeding up a good divide-and-conquer algorithm such as MERGE-SORT.

The Primary Steps of Dynamic Programming

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom up fashion.
4. Construct an optimal solution from computed information.

Reading Assignments

- Today's class:
  - Chapter 4.1, 15.1

- Reading assignment for next class:
  - Chapter 15.2 (Matrix chain multiplication)