UMass Lowell Computer Science 91.503

Analysis of Algorithms

Prof. Giampiero Pecelli

Fall, 2009

Dynamic Programming for Rod Cutting

(excerpts)

Example: Rod Cutting (text)

- 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.

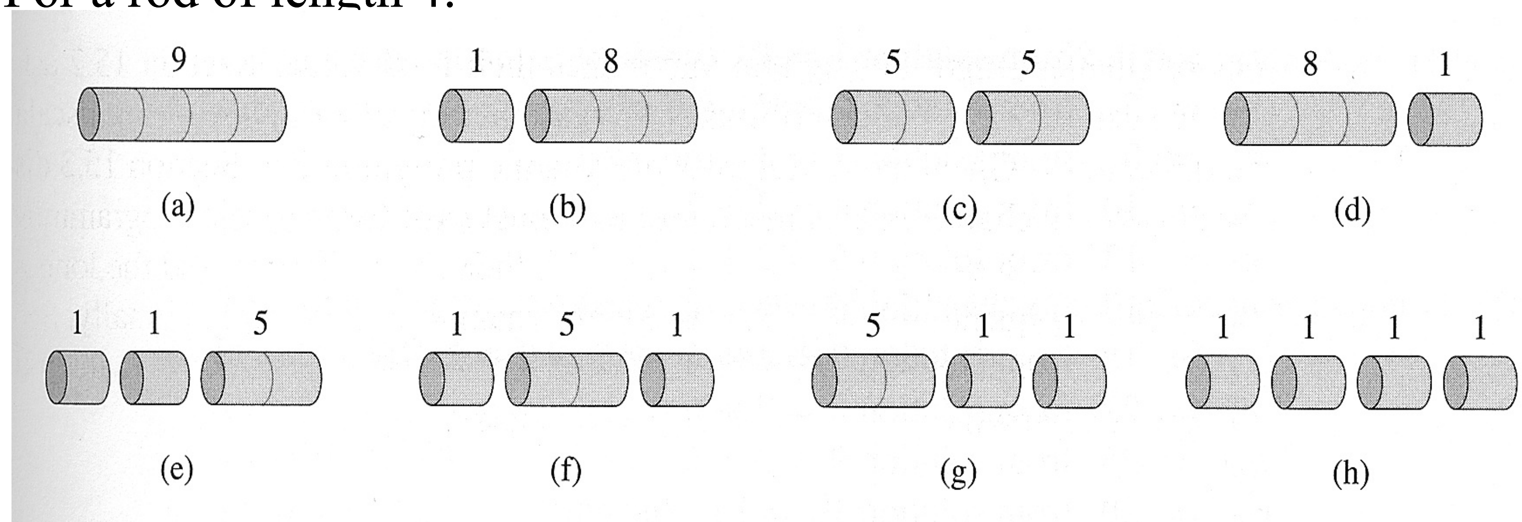
Length i	1	2	3	4	5	6	7	8	9	10
Price p_i	1	5	8	9	10	17	17	20	24	30

Example: Rod Cutting

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:



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

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 $0 \le k \le n-1$ 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 $i = 1, \ldots, 10$:

$r_1 = 1$ from sln. 1 = 1 (no cuts)
$r_2 = 5$ from sln. 2 = 2 (no cuts)
$r_3 = 8$ from sln. 3 = 3 (no cuts)
$r_4 = 10$ from sln. 4 = 2 + 2
$r_5 = 13$ from sln. 5 = 2 + 3
$r_6 = 17$ from sln. 6 = 6 (no cuts)
$r_7 = 18$ from sln. 7 = 1 + 6 or 7 = 2 + 2 + 3
$r_8 = 22$ from sln. 8 = 2 + 6
$r_9 = 25$ from sln. 9 = 3 + 6
$r_{10} = 30$ from sln. 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** (meaning?)
A slightly different way of stating the same recursion, which avoids repeating some computations, is

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

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

```
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
```

Example: Rod Cutting

Characterizing an Optimal Solution

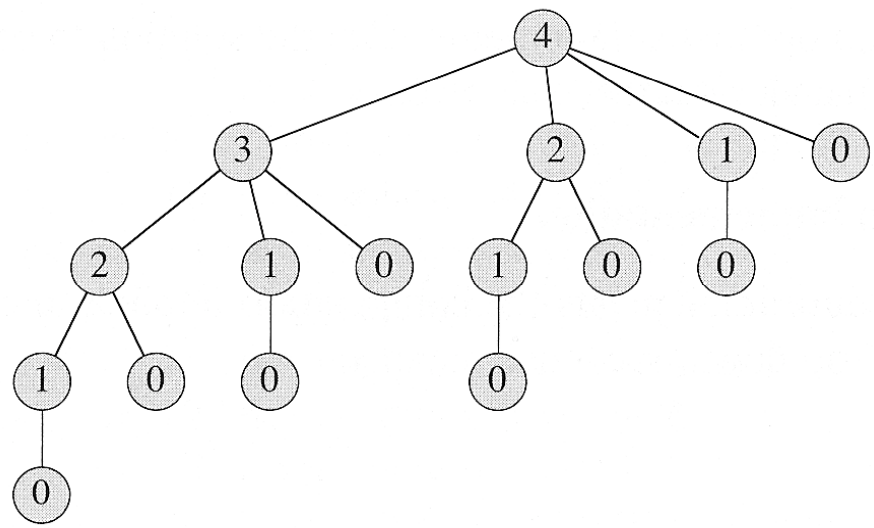
We can also notice that all the items we choose the maximum of are optimal in their own right: each substructure (max revenue for rods of lengths 1, …, n-1) is also optimal (again, **optimal substructure property**).

Nevertheless, we are still in trouble: computing the recursion leads to recomputing a number 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)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max(q, p[i] + CUT-ROD(p, n − i))
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 stack-frame activations.
• Note also that no more than $n + 1$ stack-frames 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] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

```
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
```

Example: Rod Cutting

More Sophisticated Caching

We now remove some unnecessary complications:

```
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]
```

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??