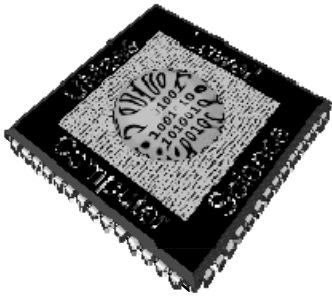


UMass Lowell Computer Science 91.503



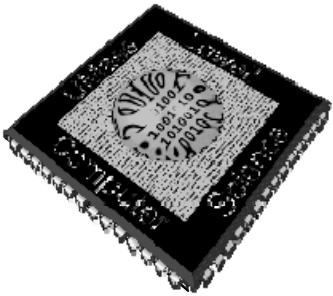
Analysis of Algorithms

Prof. Giampiero Pecelli

Fall, 2009

Dynamic Programming for Rod Cutting

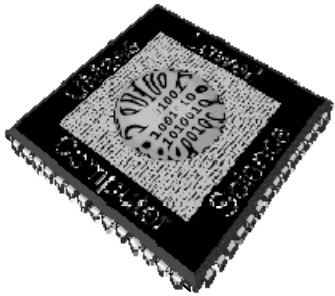
(excerpts)



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

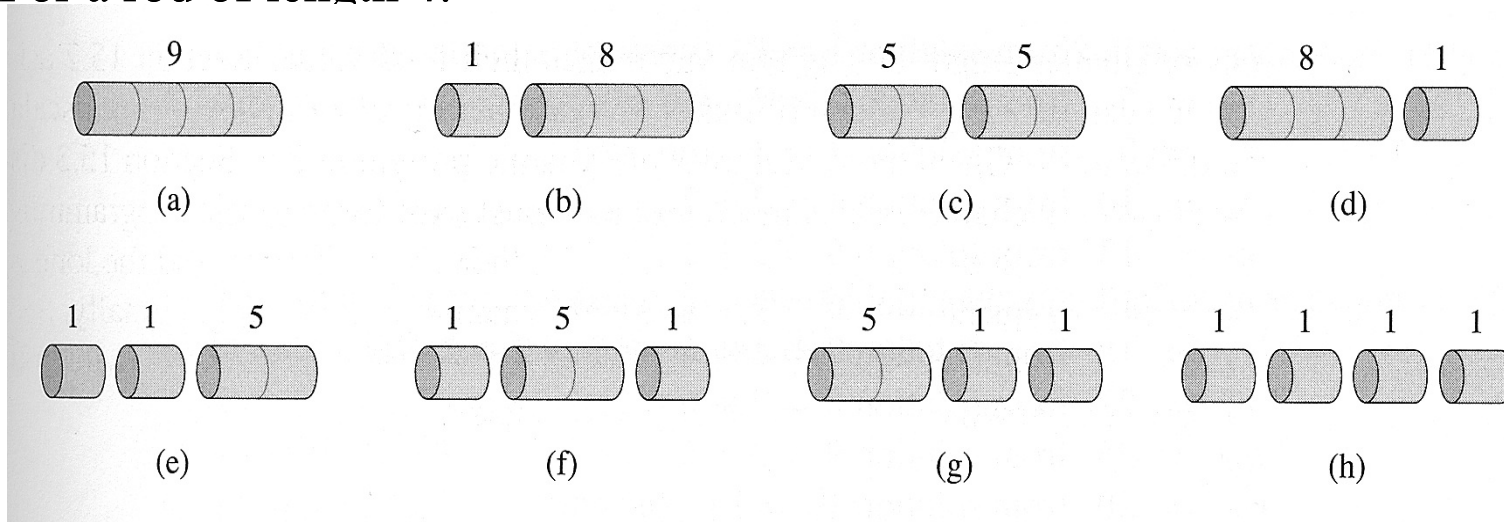


Example: Rod Cutting

Step 1: Characterizing an Optimal Solution

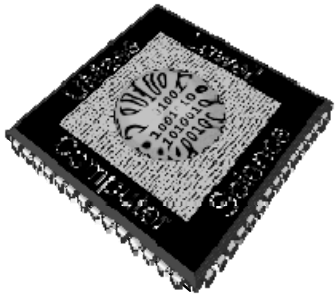
Question: in how many different ways can we cut a rod of length n ?

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 ?

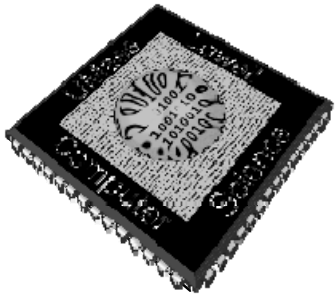
Proof Details: a rod of length n can have exactly $n-1$ possible cut positions – choose $0 \leq k \leq n-1$ actual cuts. We can choose the k cuts (without repetition) anywhere we want, so that for each such k the number of different choices is

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

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, \dots, i_k ,

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

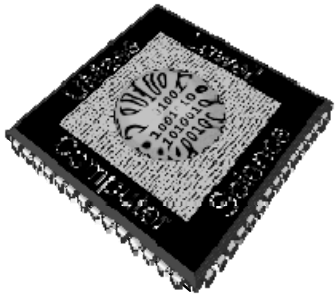
with revenue

$$r_n = p_{i_1} + \dots + p_{i_k}$$

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 $i = 1, \dots, 10$:

$r_1 = 1$ 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. $5 = 2 + 3$

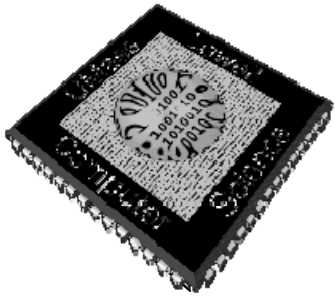
$r_6 = 17$ from sln. $6 = 6$ (no cuts)

$r_7 = 18$ 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

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

exhibiting **optimal substructure** (meaning?)

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

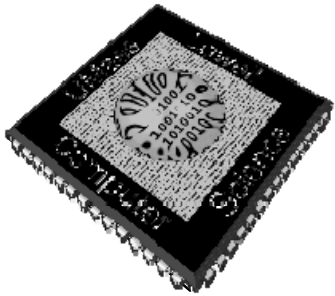
$$r_n = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

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

procedure:

CUT-ROD(p, n)

```
1  if  $n == 0$ 
2      return 0
3   $q = -\infty$ 
4  for  $i = 1$  to  $n$ 
5       $q = \max(q, p[i] + \text{CUT-ROD}(p, n - i))$ 
6  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 $1, \dots, n-1$) is also optimal (again, **optimal substructure property**).

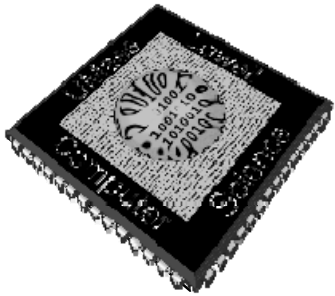
Nevertheless, we are still in trouble: computing the recursion leads to recomputing a number of values – how many?


$$\text{CUT-ROD}(p, n)$$

```
graph TD; 4((4)) --- 3((3)); 4 --- 2((2)); 4 --- 1((1)); 4 --- 0_0((0)); 3 --- 2_1((2)); 3 --- 1_1((1)); 3 --- 0_1((0)); 2_1 --- 1_2((1)); 2_1 --- 0_2((0)); 1_1 --- 0_3((0)); 2_2((2)) --- 1_3((1)); 2_2 --- 0_4((0)); 1_2((1)) --- 0_5((0)); 0_2((0)) --- 0_6((0)); 0_4((0)) --- 0_7((0)); 0_5((0)) --- 0_8((0)); 0_6((0)) --- 0_9((0)); 0_7((0)) --- 0_10((0)); 0_8((0)) --- 0_11((0)); 0_9((0)) --- 0_12((0)); 0_10((0)) --- 0_13((0)); 0_11((0)) --- 0_14((0)); 0_12((0)) --- 0_15((0)); 0_13((0)) --- 0_16((0)); 0_14((0)) --- 0_17((0)); 0_15((0)) --- 0_18((0)); 0_16((0)) --- 0_19((0)); 0_17((0)) --- 0_20((0)); 0_18((0)) --- 0_21((0)); 0_19((0)) --- 0_22((0)); 0_20((0)) --- 0_23((0)); 0_21((0)) --- 0_24((0)); 0_22((0)) --- 0_25((0)); 0_23((0)) --- 0_26((0)); 0_24((0)) --- 0_27((0)); 0_25((0)) --- 0_28((0)); 0_26((0)) --- 0_29((0)); 0_27((0)) --- 0_30((0)); 0_28((0)) --- 0_31((0)); 0_29((0)) --- 0_32((0)); 0_30((0)) --- 0_33((0)); 0_31((0)) --- 0_34((0)); 0_32((0)) --- 0_35((0)); 0_33((0)) --- 0_36((0)); 0_34((0)) --- 0_37((0)); 0_35((0)) --- 0_38((0)); 0_36((0)) --- 0_39((0)); 0_37((0)) --- 0_40((0)); 0_38((0)) --- 0_41((0)); 0_39((0)) --- 0_42((0)); 0_40((0)) --- 0_43((0)); 0_41((0)) --- 0_44((0)); 0_42((0)) --- 0_45((0)); 0_43((0)) --- 0_46((0)); 0_44((0)) --- 0_47((0)); 0_45((0)) --- 0_48((0)); 0_46((0)) --- 0_49((0)); 0_47((0)) --- 0_50((0)); 0_48((0)) --- 0_51((0)); 0_49((0)) --- 0_52((0)); 0_50((0)) --- 0_53((0)); 0_51((0)) --- 0_54((0)); 0_52((0)) --- 0_55((0)); 0_53((0)) --- 0_56((0)); 0_54((0)) --- 0_57((0)); 0_55((0)) --- 0_58((0)); 0_56((0)) --- 0_59((0)); 0_57((0)) --- 0_60((0)); 0_58((0)) --- 0_61((0)); 0_59((0)) --- 0_62((0)); 0_60((0)) --- 0_63((0)); 0_61((0)) --- 0_64((0)); 0_62((0)) --- 0_65((0)); 0_63((0)) --- 0_66((0)); 0_64((0)) --- 0_67((0)); 0_65((0)) --- 0_68((0)); 0_66((0)) --- 0_69((0)); 0_67((0)) --- 0_70((0)); 0_68((0)) --- 0_71((0)); 0_69((0)) --- 0_72((0)); 0_70((0)) --- 0_73((0)); 0_71((0)) --- 0_74((0)); 0_72((0)) --- 0_75((0)); 0_73((0)) --- 0_76((0)); 0_74((0)) --- 0_77((0)); 0_75((0)) --- 0_78((0)); 0_76((0)) --- 0_79((0)); 0_77((0)) --- 0_80((0)); 0_78((0)) --- 0_81((0)); 0_79((0)) --- 0_82((0)); 0_80((0)) --- 0_83((0)); 0_81((0)) --- 0_84((0)); 0_82((0)) --- 0_85((0)); 0_83((0)) --- 0_86((0)); 0_84((0)) --- 0_87((0)); 0_85((0)) --- 0_88((0)); 0_86((0)) --- 0_89((0)); 0_87((0)) --- 0_90((0)); 0_88((0)) --- 0_91((0)); 0_89((0)) --- 0_92((0)); 0_90((0)) --- 0_93((0)); 0_91((0)) --- 0_94((0)); 0_92((0)) --- 0_95((0)); 0_93((0)) --- 0_96((0)); 0_94((0)) --- 0_97((0)); 0_95((0)) --- 0_98((0)); 0_96((0)) --- 0_99((0)); 0_97((0)) --- 0_100((0)); 0_98((0)) --- 0_101((0)); 0_99((0)) --- 0_102((0)); 0_100((0)) --- 0_103((0)); 0_101((0)) --- 0_104((0)); 0_102((0)) --- 0_105((0)); 0_103((0)) --- 0_106((0)); 0_104((0)) --- 0_107((0)); 0_105((0)) --- 0_108((0)); 0_106((0)) --- 0_109((0)); 0_107((0)) --- 0_110((0)); 0_108((0)) --- 0_111((0)); 0_109((0)) --- 0_112((0)); 0_110((0)) --- 0_113((0)); 0_111((0)) --- 0_114((0)); 0_112((0)) --- 0_115((0)); 0_113((0)) --- 0_116((0)); 0_114((0)) --- 0_117((0)); 0_115((0)) --- 0_118((0)); 0_116((0)) --- 0_119((0)); 0_117((0)) --- 0_120((0)); 0_118((0)) --- 0_121((0)); 0_119((0)) --- 0_122((0)); 0_120((0)) --- 0_123((0)); 0_121((0)) --- 0_124((0)); 0_122((0)) --- 0_125((0)); 0_123((0)) --- 0_126((0)); 0_124((0)) --- 0_127((0)); 0_125((0)) --- 0_128((0)); 0_126((0)) --- 0_129((0)); 0_127((0)) --- 0_130((0)); 0_128((0)) --- 0_131((0)); 0_129((0)) --- 0_132((0)); 0_130((0)) --- 0_133((0)); 0_131((0)) --- 0_134((0)); 0_132((0)) --- 0_135((0)); 0_133((0)) --- 0_136((0)); 0_134((0)) --- 0_137((0)); 0_135((0)) --- 0_138((0)); 0_136((0)) --- 0_139((0)); 0_137((0)) --- 0_140((0)); 0_138((0)) --- 0_141((0)); 0_139((0)) --- 0_142((0)); 0_140((0)) --- 0_143((0)); 0_141((0)) --- 0_144((0)); 0_142((0)) --- 0_145((0)); 0_143((0)) --- 0_146((0)); 0_144((0)) --- 0_147((0)); 0_145((0)) --- 0_148((0)); 0_146((0)) --- 0_149((0)); 0_147((0)) --- 0_150((0)); 0_148((0)) --- 0_151((0)); 0_149((0)) --- 0_152((0)); 0_150((0)) --- 0_153((0)); 0_151((0)) --- 0_154((0)); 0_152((0)) --- 0_155((0)); 0_153((0)) --- 0_156((0)); 0_154((0)) --- 0_157((0)); 0_155((0)) --- 0_158((0)); 0_156((0)) --- 0_159((0)); 0_157((0)) --- 0_160((0)); 0_158((0)) --- 0_161((0)); 0_159((0)) --- 0_162((0)); 0_160((0)) --- 0_163((0)); 0_161((0)) --- 0_164((0)); 0_162((0)) --- 0_165((0)); 0_163((0)) --- 0_166((0)); 0_164((0)) --- 0_167((0)); 0_165((0)) --- 0_168((0)); 0_166((0)) --- 0_169((0)); 0_167((0)) --- 0_170((0)); 0_168((0)) --- 0_171((0)); 0_169((0)) --- 0_172((0)); 0_170((0)) --- 0_173((0)); 0_171((0)) --- 0_174((0)); 0_172((0)) --- 0_175((0)); 0_173((0)) --- 0_176((0)); 0_174((0)) --- 0_177((0)); 0_175((0)) --- 0_178((0)); 0_176((0)) --- 0_179((0)); 0_177((0)) --- 0_180((0)); 0_178((0)) --- 0_181((0)); 0_179((0)) --- 0_182((0)); 0_180((0)) --- 0_183((0)); 0_181((0)) --- 0_184((0)); 0_182((0)) --- 0_185((0)); 0_183((0)) --- 0_186((0)); 0_184((0)) --- 0_187((0)); 0_185((0)) --- 0_188((0)); 0_186((0)) --- 0_189((0)); 0_187((0)) --- 0_190((0)); 0_188((0)) --- 0_191((0)); 0_189((0)) --- 0_192((0)); 0_190((0)) --- 0_193((0)); 0_191((0)) --- 0_194((0)); 0_192((0)) --- 0_195((0)); 0_193((0)) --- 0_196((0)); 0_194((0)) --- 0_197((0)); 0_195((0)) --- 0_198((0)); 0_196((0)) --- 0_199((0)); 0_197((0)) --- 0_200((0)); 0_198((0)) --- 0_201((0)); 0_199((0)) --- 0_202((0)); 0_200((0)) --- 0_203((0)); 0_201((0)) --- 0_204((0)); 0_202((0)) --- 0_205((0)); 0_203((0)) --- 0_206((0)); 0_204((0)) --- 0_207((0)); 0_205((0)) --- 0_208((0)); 0_206((0)) --- 0_209((0)); 0_207((0)) --- 0_210((0)); 0_208((0)) --- 0_211((0)); 0_209((0)) --- 0_212((0)); 0_210((0)) --- 0_213((0)); 0_211((0)) --- 0_214((0)); 0_212((0)) --- 0_215((0)); 0_213((0)) --- 0_216((0)); 0_214((0)) --- 0_217((0)); 0_215((0)) --- 
```

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

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



Example: Rod Cutting

Beyond Naïve Time Complexity

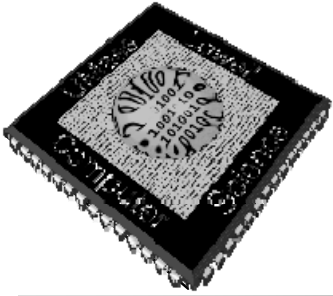
We have a problem: “reasonable size” problems are not solvable in “reasonable time” (but, in this case, they are solvable in “reasonable space”).

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

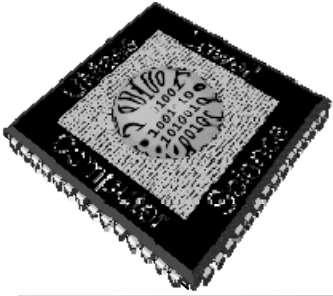
We introduce two procedures:

MEMOIZED-CUT-ROD(p, n)

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

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

```
1  if  $r[n] \geq 0$ 
2      return  $r[n]$ 
3  if  $n == 0$ 
4       $q = 0$ 
5  else  $q = -\infty$ 
6      for  $i = 1$  to  $n$ 
7           $q = \max(q, p[i] + \text{MEMOIZED-CUT-ROD-AUX}(p, n - i, r))$ 
8   $r[n] = q$ 
9  return  $q$ 
```



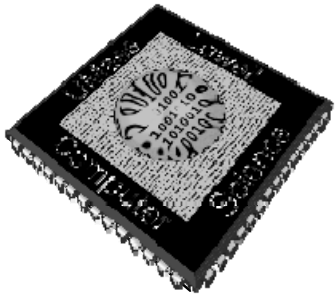
Example: Rod Cutting

More Sophisticated Caching

We now remove some unnecessary complications:

BOTTOM-UP-CUT-ROD(p, n)

```
1  let  $r[0..n]$  be a new array
2   $r[0] = 0$ 
3  for  $j = 1$  to  $n$ 
4       $q = -\infty$ 
5      for  $i = 1$  to  $j$ 
6           $q = \max(q, p[i] + r[j - i])$ 
7       $r[j] = q$ 
8  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??