DYNAMIC PROGRAMMING: ROD CUTTING

CS340

Rod Cutting

- How to cut steel rods into pieces in order to maximize the revenue you can get?
  - Each cut is free. Rod lengths are always an integral number of inches.
  - Input: A length n and table of prices p_i , for i = 1, 2, … , n.
  - Output: The maximum revenue obtainable for rods whose lengths sum to n, computed as the sum of the prices for the individual rods.

Rod Cutting

- Table of prices:

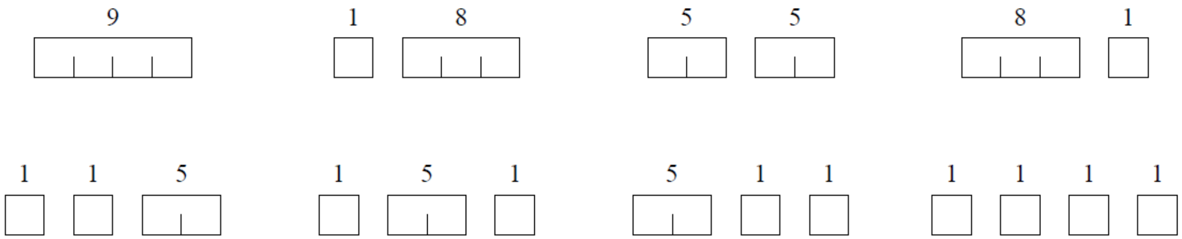
Length i	1	2	3	4	5	6	7	8
Price p_i	1	5	8	9	10	17	17	20

- Can cut up a rod in 2^{n-1} different ways, because can choose to cut or not cut after each of the first n - 1 inches.

Rod Cutting

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

- Table of prices:
- Here are all 8 ways to cut a rod of length 4, with the costs from the example:



- How many ways are repeats of other ways?
- Which way of cutting gives the most revenue?

Rod Cutting

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

Let r_i be the maximum revenue for a rod of length i. Can express a solution as a sum of individual rod lengths.

Can determine optimal revenues r_i for the example, by inspection:

i	r_i	optimal solution
1	1	1 (no cuts)
2	5	2 (no cuts)
3	8	3 (no cuts)
4	10	$2 + 2$
5	13	$2 + 3$
6	17	6 (no cuts)
7	18	$1 + 6$ or $2 + 2 + 3$
8	22	$2 + 6$

Rod Cutting

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

Can determine optimal revenue r_n by taking the maximum of

- p_n: the price we get by not making a cut,
- $r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,
- $r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches, ...
- $r_{n-1} + r_1$.

That is,

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

- To solve the original problem, solve problems on smaller sizes. It is recursive in nature.

Rod Cutting

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

Dynamic-programming solution

- The problem has an optimal substructure
- Instead of solving the same subproblems repeatedly, arrange to solve each subproblem just once.
- Save the solution to a subproblem in a table, and refer back to the table whenever we revisit the subproblem.
- "Store, don't recompute" → time-memory trade-off.
- Can turn an exponential-time solution into a polynomial-time solution.

Rod Cutting

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

Bottom-up

- Sort the subproblems by size and solve the smaller ones first. That way, when solving a subproblem, have already solved the smaller subproblems we need.

Rod Cutting

Length i	0	1	2	3	4	5	6	7	8
Price p_i	0	1	5	8	9	10	17	17	20
Revenue r_i	0								

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 $r[n]$

What is the time complexity?

Rod Cutting

Length i	0	1	2	3	4	5	6	7	8
Price p_i	0	1	5	8	9	10	17	17	20
Revenue r_i	0	1	5	8	10	13	17	18	22
Cuts s_i	0	1	2	3	2	2	6	1	2

Record optimal choices (locations of cuts) in addition to optimal revenues.

```
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 = -∞
      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
```

```
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]
```