DYNAMIC PROGRAMMING

Analysis of Algorithm

Punjab University College of Information Technology (PUCIT)
University of the Punjab, Lahore, Pakistan.

Credit

- These notes contain material from Chapter 15 of Cormen, Leiserson, Rivest, and Stein (3rd Edition).

Dynamic Programming

- Dynamic programming, like the divide-and-conquer method
- Divide and conquer is used for disjoint subproblems however dynamic programming is for overlap subproblems
- Here “Programming” refers to a tabular method, not to writing computer code.
- A dynamic-programming algorithm solves each subsubproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time it solves each subsubproblem

Dynamic Programming (Cont.)

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

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, typically in a bottom-up fashion.
4. Construct an optimal solution from computed information.

Rod Cutting Problem

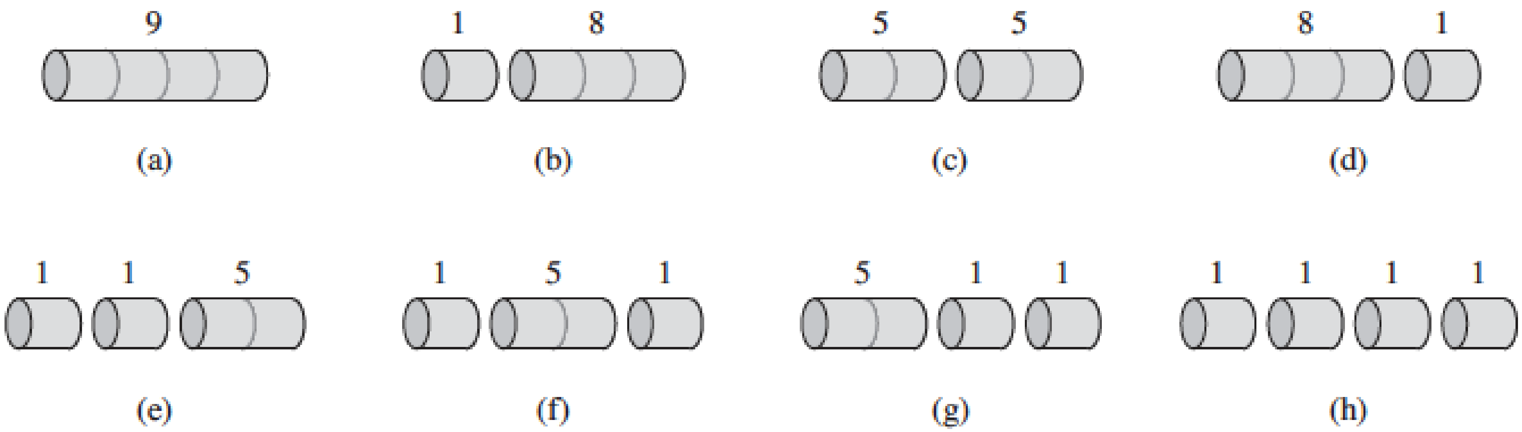
Given a rod of length n inches and a table of prices p_i for $i = 1,2,\ldots.n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces.

Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

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

Rod Cutting Problem (Cont.)

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



If an optimal solution cuts the rod into k pieces, for some 1<= k <=n, then an optimal decomposition:

$$n = i_1 + i_2 + \cdots + i_k$$

provides maximum corresponding revenue

$$r_n = p_{i_1} + p_{i_2} + \cdots + p_{i_k} \ .$$

Rod Cutting Problem (Cont.)

For our sample problem, we can determine the optimal revenue figures r_i, for $i = 1, 2, \ldots, 10$, by inspection, with the corresponding optimal decompositions

$$
\begin{aligned}
r_1 &= 1 && \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 && \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 && \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 && \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 && \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 && \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 && \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 && \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 && \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 && \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

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

More generally, we can frame the values r_n for $n \geq 1$ in terms of optimal revenues from shorter rods:

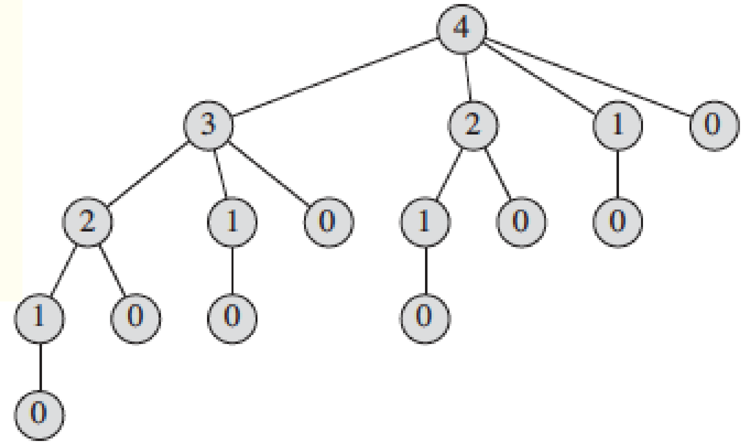
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) . \tag{15.1}$$

simpler version of the above equation: $r_n = \max\limits_{1 \leq i \leq n} (p_i + r_{n-i})$.

Rod Cutting Problem (Cont.)

Recursive top-down implementation

```
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
```



In general, this recursion tree has **2**^{n} nodes and **2**$^{n-1}$ leaves which gives us intuition:

$$T(n) = 2^n$$

length	**1**	**2**	**3**	**4**
p[length]	5	3	2	1

Dynamic programming solution: Top down approach

```
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
```

Remember traditional method

```
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
```

Dynamic programming solution: Bottom-up approach

```
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]
```

Remember traditional method

```
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
```

Lets dry run for n =4 using the following price table

length	1	2	3	4
p[length]	5	3	2	1

Dynamic programming solution: Bottom-up approach also prints cut

```
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]
```

In our rod-cutting example, the call EXTENDED-BOTTOM-UP-CUT-ROD$(p, 10)$ would return the following arrays:

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

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