Dynamic Programming

Rod Cutting

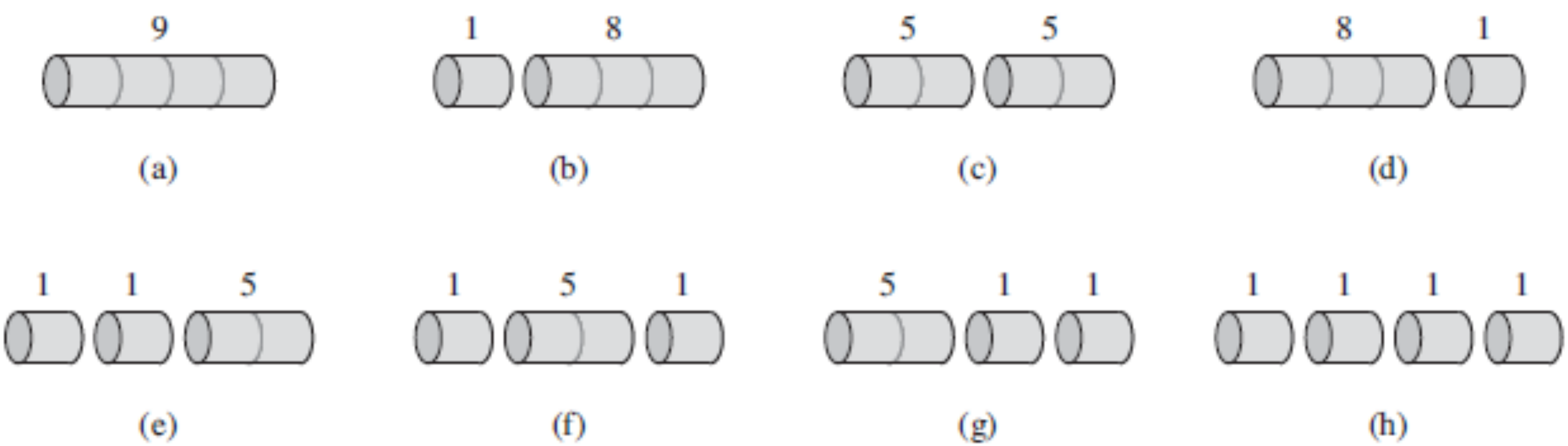
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Problem: We are given a rod of length n, and want to maximize revenue, by cutting up the rod into pieces and selling each of the pieces.

Example: we are given a 4 inches rod. Best solution to cut up?

Example n=4

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



Brute force

In general, rod of length n can be cut in 2^{n-1} different ways, since we can choose cutting, or not cutting, at all distances i $(1 \leq i \leq n-1)$ from the left end

Notation:

Given a rod of length n.

If an optimal solution cuts the rod into k pieces, for some $1 \leq k \leq n$, then an optimal decomposition

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

of the rod into pieces of lengths $i_1, i_2, \ldots i_k$ provides maximum corresponding revenue

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

In simple words, our goal is to compute r_n.

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

$r_1 = 1$

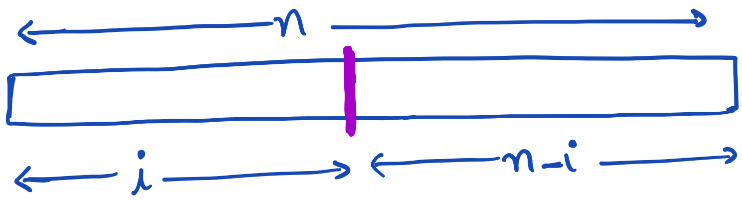
$r_2 = 5$ (no cuts)

$r_3 = 8$ (no cuts)

$r_4 = 10$ $(4 = 2 + 2)$

$r_5 = 13$ $(5 = 3 + 2)$

$\vdots \quad \vdots \quad \vdots$



$$r_n = r_i + r_{n-i}$$

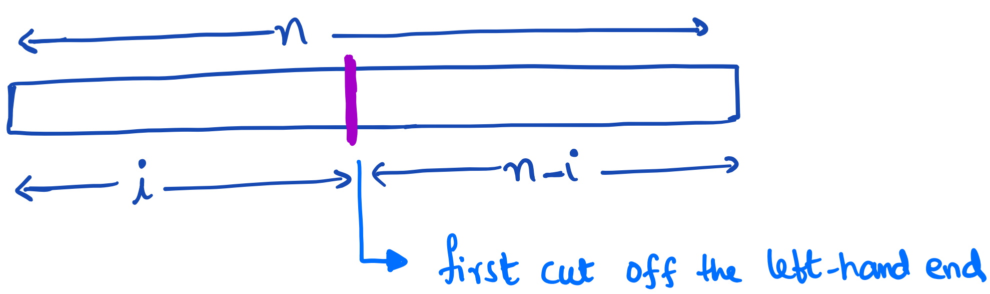
Since we don't know ahead of time which value of i optimizes revenue. We try all possibilities.

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

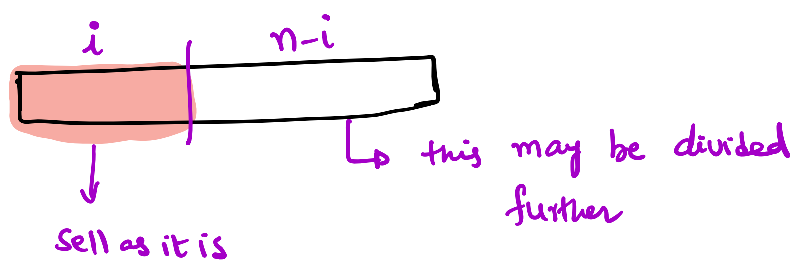
no cuts at all

Note that to solve the original problem of size n, we solve smaller problems of the same type, but of smaller sizes. Once we make the first cut, we may consider the two pieces as independent instances of the rod-cutting problem. The overall optimal solution incorporates optimal solutions to the two related subproblems, maximizing revenue from each of those two pieces. We say that the rod-cutting problem exhibits ***optimal substructure***: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.

Another approach



that is only right-hand piece may be further divided, not the first piece.



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

Recursive Top-down implementation

Cut-Rod(p, n)

```
if n = 0 then
    return 0;
end
q = −∞;
for i = 1 to n do
    q = max(q, p[i] + Cut-Rod(p, n − i));
end
return q;
```

Algorithm Time

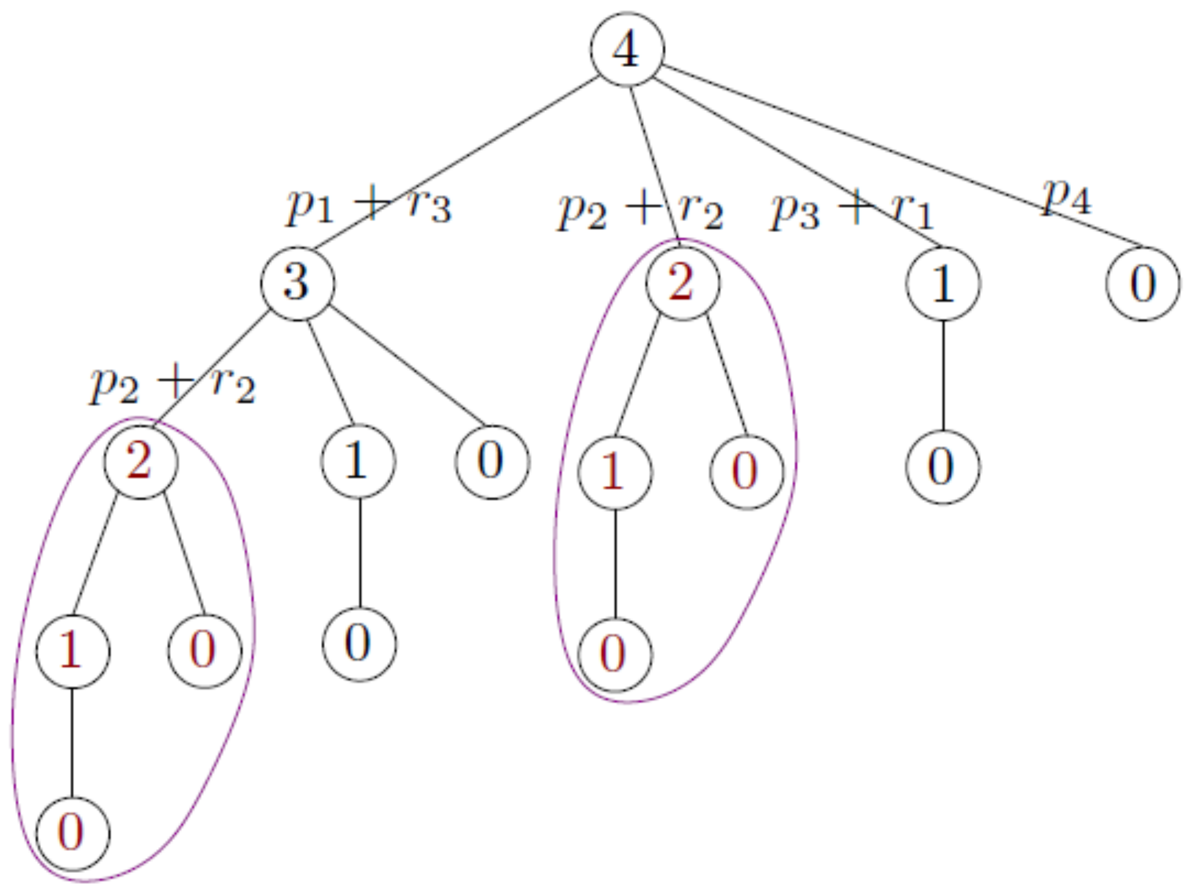
- $T(n)$: the total number of calls made to Cut-Rod when called with rod length n

$$T(n) = \begin{cases} 1 + \sum_{0 \le j \le n-1} T(j), & \text{if } n > 0, \\ 1, & \text{if } n = 0. \end{cases}$$

- Induction $\Rightarrow T(n) = 2^n$

Why exponential?

- Algorithm calls same subproblem many times



DP- solution

DP-solution

- p_i are the problem inputs.
- r_i is max profit from cutting rod of length i.
- Goal is to calculate r_n
- r_i defined by
  - $r_1 = p1$ and $r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$
- Iteratively fill in r_i table by calculating $r_1, r_2, r_3, \ldots$
- r_n **is final solution**

i	1	2	3	4	... ...	n
r_i	p_1				... ...	

Bottom-up implementation

Bottom-Up-Cut-Rod(p, n)

```
r[0] = 0; // Array r[0...n] stores the computed optimal values
for j = 1 to n do
    // Consider problems in increasing order of size
    q = −∞;
    for i = 1 to j do
        // To solve a problem of size j, we need to consider all
           decompositions into i and j − i
        q = max(q, p[i] + r[j − i]);
    end
    r[j] = q;
end
return r[n];
```

- Cost: $O(n^2)$
  - The outer loop computes $r[1], r[2], \ldots, r[n]$ in this order
  - To compute $r[j]$, the inner loop uses all values $r[0], r[1], \ldots, r[j-1]$ (i.e., $r[j-i]$ for $1 \leq i \leq j$)

Output the cutting

Output the cutting

- Algorithm only *computes* r_i. It does not output the cutting.
- Easy fix
  - When calculating $r_j = \max_{1 \le i \le j}(p_i + r_{j-i})$
    store value of i that achieved this max in new array $s[j]$.
  - This i is the size of last piece in the optimal cutting.
- After algorithm is finished, can reconstruct optimal cutting by unrolling the s_j.

Extension

Extended-Bottom-Up-Cut-Rod(p, n)

```
// Array s[0...n] stores the optimal size of the first piece to
   cut off
r[0] = 0; // Array r[0...n] stores the computed optimal values
for j = 1 to n do
    q = −∞;
    for i = 1 to j do
        // Solve problem of size j
        if q < p[i] + r[j − i] then
            q = p[i] + r[j − i];
            s[j] = i; // Store the size of the first piece
        end
    end
    r[j] = q;
end
while n > 0 do
    // Print sizes of pieces
    Print s[n];
    n = n − s[n];
end
```

Top-down with memoization

Recall:

We say that a recursive procedure has been memoized: if it remembers what results it has computed previously.

Memoized-version

```
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
```

Longest Common Subsequence

A Subsequence of a given sequence is just the given sequence with zero or more elements left out.

Example: $X = \langle A, B, C, B, D, A, B \rangle$

$Z_1 = \langle B, C, D, B \rangle$
$Z_2 = \langle B, B, B \rangle$
Subsequences of X

$Z_3 = \langle A, A, D \rangle$
Not a Subsequence of X

Given two sequences X & Y, we say that a sequence Z is a common subsequence of X & Y if Z is a subsequence of both X & Y

Ex: $X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

$\langle B, C, A \rangle$ is a common subsequence of X & Y.

Not a longest common subsequence of X & Y

$\langle B, C, B, A \rangle$ is a LCS of X & Y

Longest-Common-Subsequence Problem

Input: Two sequences $X = \langle x_1, x_2, \ldots x_m \rangle$

$Y = \langle y_1, y_2, \ldots y_n \rangle$

Output: Find a Maximum-length common subsequence of X and Y.

Brute-Force solution

Notation:

If $X = \langle x_1, x_2, .. x_m \rangle$

i^{th} prefix of X, $X_i = \langle x_1, x_2 .. x_i \rangle$

Eg $X = \langle A, B, C, B, D, A, B \rangle$ then

$X_4 = \langle A, B, C, B \rangle$

X_0 = empty sequence

Given $X = \langle x_1 \ldots x_m \rangle$

$Y = \langle y_1 \ldots y_n \rangle$

If $x_m = y_n$ then ??

If $x_m \neq y_n$ then ??

let

$c[i,j]$ be the length of an LCS of the sequences X_i and Y_j.

- If $i=0$ or $j=0$ then $c[i,j]=0$

$$c[i,j] = \begin{cases} 0 & \text{if } i=0 \text{ or } j=0 \\ c[i-1,j-1]+1 & \text{if } i,j>0 \ \& \ x_i = y_j \\ \max(c[i-1,j], c[i,j-1]), & \text{if } i,j>0 \ \& \\ & x_i \neq y_j \end{cases}$$

Dynamic Programming – Bottom Up

Given $X = \langle x_1, \ldots x_m \rangle$ and

$Y = \langle y_1, \ldots y_n \rangle$

- We store $c[i,j]$ values in a table $c[0 \ldots m, 0 \ldots n]$

- We fill the table row wise (left to right)

LCS-LENGTH(X, Y)

```
m = X.length
n = Y.length
let b[1..m, 1..n] and c[0..m, 0..n] be new tables
for i = 1 to m
    c[i, 0] = 0
for j = 0 to n
    c[0, j] = 0
for i = 1 to m
    for j = 1 to n
        if x_i == y_j
            c[i, j] = c[i - 1, j - 1] + 1
            b[i, j] = "↖"
        elseif c[i - 1, j] ≥ c[i, j - 1]
            c[i, j] = c[i - 1, j]
            b[i, j] = "↑"
        else c[i, j] = c[i, j - 1]
            b[i, j] = "←"
return c and b
```

Example

$$X = \langle A, B, C, B, D, A, B \rangle$$

$$Y = \langle B, D, C, A, B, A \rangle$$

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Constructing an LCS

PRINT-LCS(b, X, i, j)

if $i == 0$ or $j == 0$
    return
if $b[i, j] ==$ "$\nwarrow$"
    PRINT-LCS$(b, X, i-1, j-1)$
    print x_i
elseif $b[i, j] ==$ "$\uparrow$"
    PRINT-LCS$(b, X, i-1, j)$
else PRINT-LCS$(b, X, i, j-1)$