Rod-Cutting Problem

Kuan-Yu Chen (陳冠宇)

2019/04/16 @ TR-310-1, NTUST

Review

- We can categorize that
  - Comparison Sorts
    - The sorted order they determine is based only on comparisons between the input elements
    - Insertion Sort, Merge Sort, Quick Sort
  - Non-comparison Sorts
    - Counting Sort, Radix Sort, Bucket Sort

Algorithm	Worst-case running time	Average-case/expected running time	Best-case running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$	$O(n)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \log_2 n)$
Heapsort	$O(n \lg n)$	$O(n \lg n)$	$O(n)$
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)	$\Theta(n \log_2 n)$
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$	$\Theta(d(k + n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)	$\Theta(n)$

Dynamic Programming

- Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems

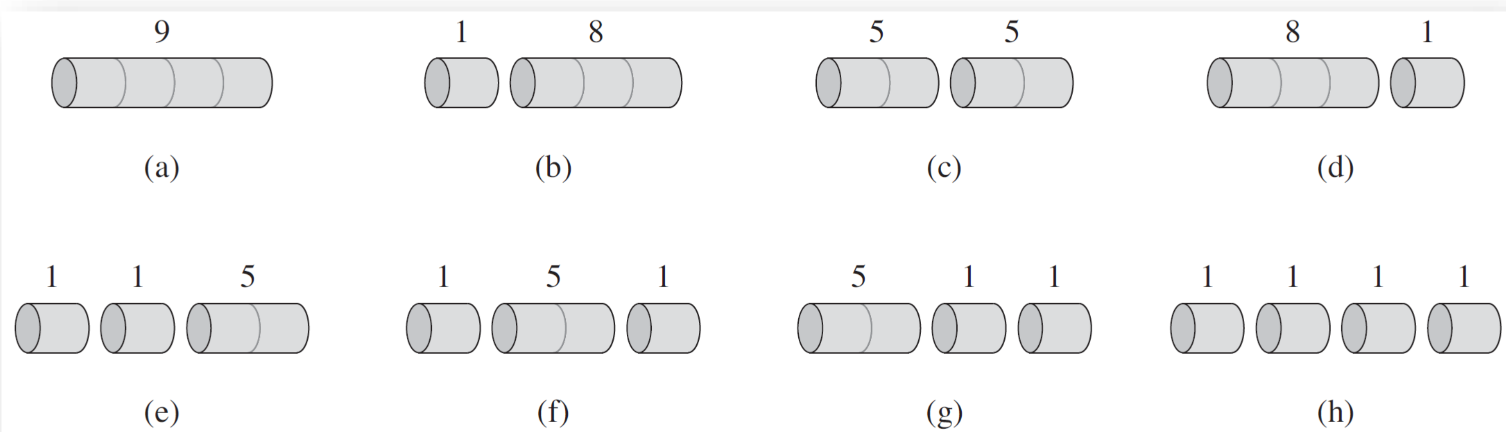
- We typically apply dynamic programming to ***optimization problems***
  - Such problems can have many possible solutions
  - Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value

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

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

  - Consider the case when $n = 4$
    - We can cut up a rod of length n in 2^{n-1} different ways
    - Cutting a 4-inch rod into two 2-inch pieces produces revenue $p_2 + p_2 = 5 + 5 = 10$ is optimal



Rod-Cutting Problem..

- Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \dots, n$, 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

  - If an optimal solution cuts the rod into k pieces, $1 \le k \le n$, then an optimal decomposition is $\{i_1, i_2, \dots, i_k\}$

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

- The maximum revenue is

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

- A general form is

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

maximum revenue

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

Rod-Cutting Problem…

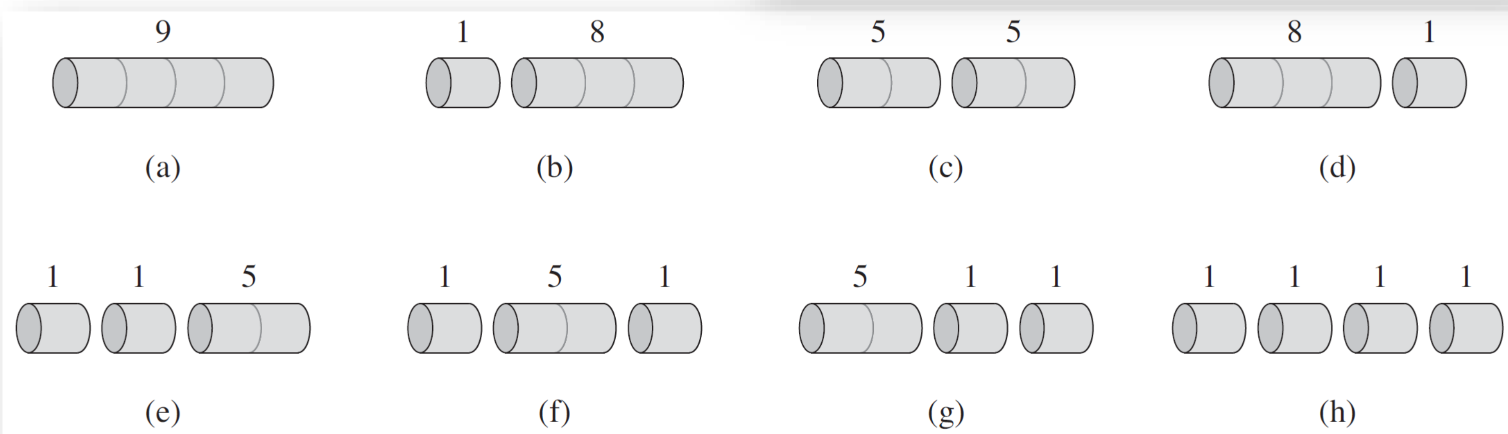
– For a given rod of length n inches, the general form of maximum revenue is

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

– A simpler equation is

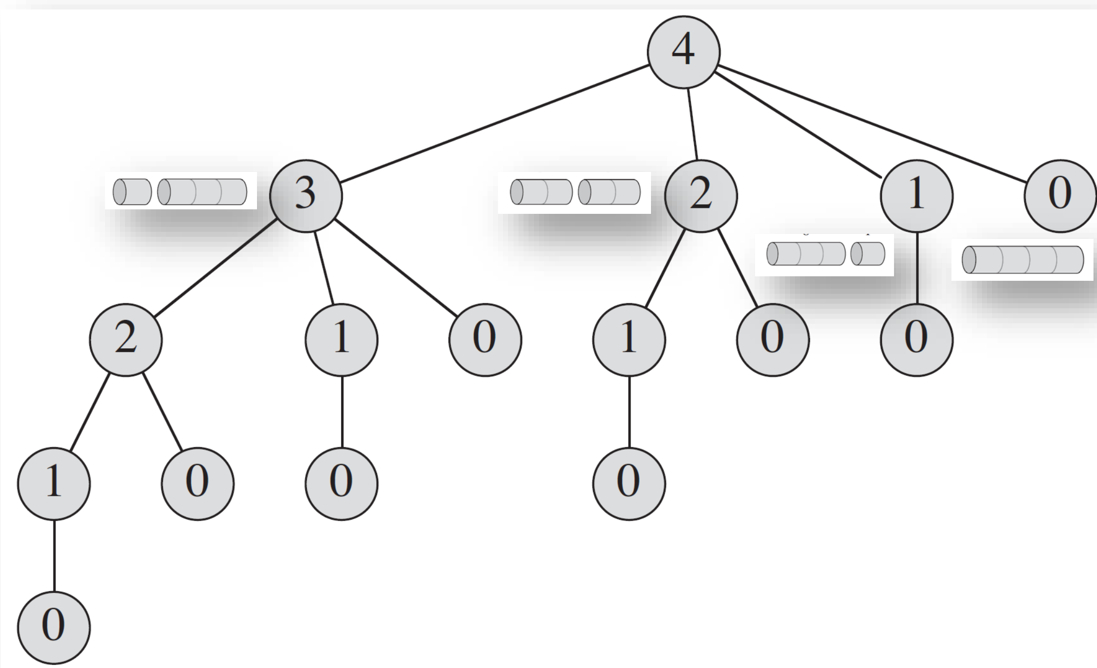
$$r_n = \max_{1 \le i \le n}(p_i + r_{n-i})$$

```
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
```



Rod-Cutting Problem….

- The Cut-Rod function is very inefficient!
  - For $n = 40$, you would find that your program takes at least several minutes, and most likely more than an hour
  - **The problem is that Cut-Rod calls itself recursively over and over again with the same parameter values**




```
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
```

DP for Rod-Cutting Problem

- The naive recursive solution is inefficient because it solves the same subproblems repeatedly, thus DP solves each subproblem only once
  - If we need to refer to this subproblem's solution again later, we can just look it up, rather than recompute it
  - Dynamic programming thus uses additional memory to save computation time
    - ***time-memory trade-off***
  - There are usually two equivalent ways to implement a dynamic-programming approach
    - ***top-down with memorization***
    - ***bottom-up method***

Top-down with Memorization

- The pseudocode for the top-down Cut-Rod procedure with memorization
  - The procedure Memoized-Cut-Rod-Aux is just the memoized version of Cut-Rod procedure

```
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
```

```
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)
```

```
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
```

Bottom-up Strategy

- The bottom-up version is even simpler

```
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]
```

  - For the bottom-up dynamic-programming approach, Bottom-Up-Cut-Rod, solves the problem from “smaller” subproblems
    - The procedure solves subproblems of sizes $j = 0,1,\cdots,n$, in that order

- The bottom-up and top-down versions have the same asymptotic running time $\Theta(n^2)$

Reconstructing a Solution.

- Here is an extended version of Bottom-Up-Cut-Rod that computes, for each rod size j , not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off
  - Our dynamic-programming solutions do not return an actual solution, i.e., a list of piece sizes
  - It updates $s[j]$ in line 8 to hold the optimal size i of the first piece to cut off when solving a subproblem of size j

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1   let r[0..n] and s[0..n] be new arrays
2   r[0] = 0
3   for j = 1 to n
4       q = -∞
5       for i = 1 to j
6           if q < p[i] + r[j - i]
7               q = p[i] + r[j - i]
8               s[j] = i
9       r[j] = q
10  return r and s
```

Reconstructing a Solution..

- The following procedure takes a price table p and a rod size n

```
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]
```

```
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
```

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

  - For 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

Thinking!

- What's the major difference between dynamic programing and divide-and-conquer strategies?

Questions?

kychen@mail.ntust.edu.tw