CS217 - Data Structures & Algorithm Analysis (DSAA)

Lecture #11

▶Dynamic Programming

Prof. Pietro S. Oliveto

Department of Computer Science and Engineering

Southern University of Science and Technology (SUSTech)

olivetop@sustech.edu.cn

https://faculty.sustech.edu.cn/olivetop

Reading: Chapter 14.1

Aims of this lecture

- To discuss the dynamic programming paradigm for solving optimisation problems.
- To work through an example of a problem solved efficiently with dynamic programming.
- To discuss properties of problems where dynamic programming is efficient.
- To discuss how to implement dynamic programming algorithms.

How to compute Fibonacci numbers?

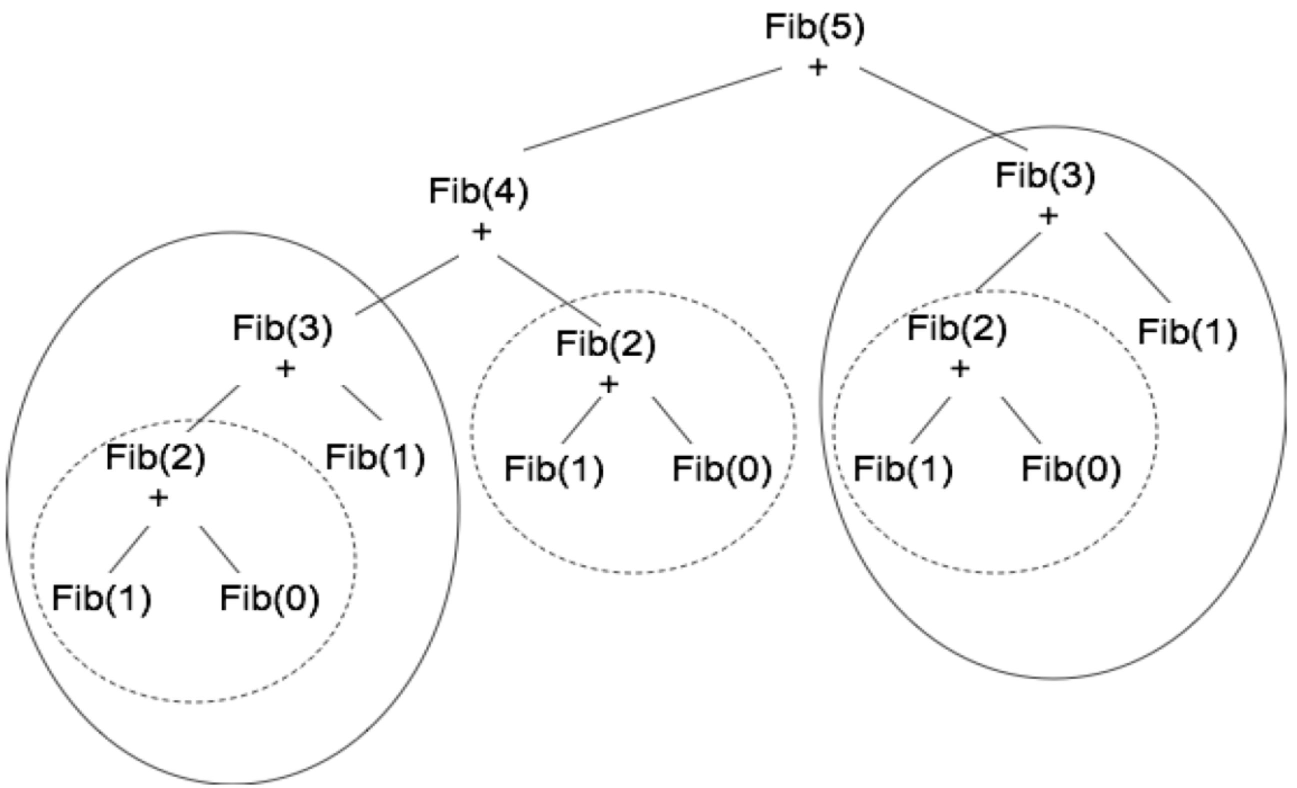
- Fibonacci numbers:
  - $Fib(0) = Fib(1) = 1$
  - $Fib(k) = Fib(k-1) + Fib(k-2)$
  - Handy closed form lower bound:

$$\mathrm{Fib}(k) \geq \frac{1}{\sqrt{5}}\left[\left(\frac{\sqrt{5}+1}{2}\right)^{k+1} - 1\right]$$

- Let's try to compute $Fib(n)$ exactly using the recursive definition.

What happened??

- The same values are **computed from scratch many times!**



What happened??

- Let's call $T(n)$ the time to compute $Fib(n)$.
- Let's ignore constants for simplicity so that

$$T(0) = T(1) = 1.$$

- Then $T(n) = T(n-1) + T(n-2) + 1$.
- Let's ignore the "+1" and take

$$T(n) = T(n-1) + T(n-2).$$

- Then $T = Fib$! And from closed formula *Fib(n)* = $\Omega\left(\left(\frac{\sqrt{5}+1}{2}\right)^n\right)$
- $T(90) = Fib(90) = 4660046610375530309$.
- Larger than the age of the Universe in seconds.

A smarter way

- Compute Fibonacci numbers **bottom-up in a table**.
- Refer to table instead of re-calculating!
- (Bottom-up ensures we refer to entries already calculated.)
- Time $O(n)$ instead of

$$\Omega\left(\left(\frac{\sqrt{5}+1}{2}\right)^n\right)$$

Dynamic Programming

- A **general algorithm design method** that can be used when the solution to a problem may be viewed as the result of a **sequence of decisions**.
  - Developed back in the day when "**programming**" meant "**tabular method**".
- **Idea**: solve **subproblems** of the original problem and **save the answers in a table**. Solve subproblems of increasing size until we can solve the original problem.
  - Avoids the work of recomputing the answer every time it solves a subproblem.
  - Solving subproblems is similar to divide and conquer, but for Dynamic Programming **subproblems typically overlap**.
- Optimisation problems: find a solution with the optimal value.

Properties of Dynamic Programming

- **Optimal substructure**: The solutions to the subproblems used within the optimal solution must themselves be optimal.
  - Often: making a first decision in an optimal way, and then being left with a smaller problem that needs to be solved optimally.
- Dynamic Programming is usually efficient if the problem has optimal substructure and the space of subproblems is small.

Rod Cutting Problem

- How to cut a steel rod of length *n* into pieces in order to maximise the revenue from selling all pieces?

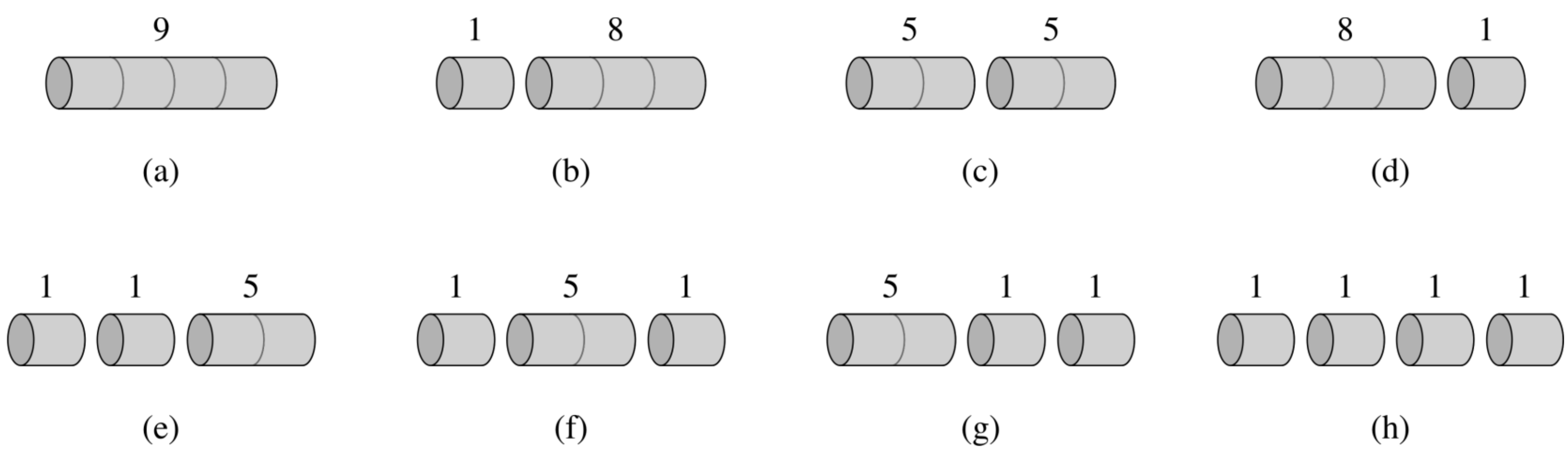
  - Each cut is free. Rod lengths are an integral number of cm.
  - Each rod length *i* has its own price p_i.
  - Output: maximum revenue obtainable from rods whose lengths sum to *n*, according to the price list.

Rod Cutting Problem: Example

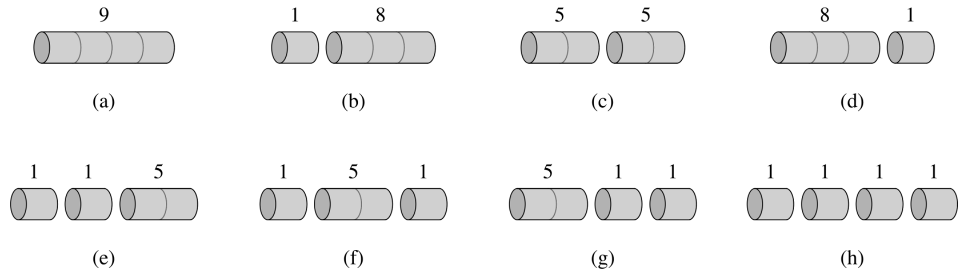
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

There are 2^{n-1} different ways to cut up a rod, because we can choose to cut or not cut after each of the first $n-1$ cm.

Here are all $2^{4-1}=8$ ways to cut a rod of length 4, with above prices:



Rod Cutting Problem: One way



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

- Let r_i be the maximum revenue for a rod of length *i*

$$r_n = \max\{p_n, p_1 + r_{n-1}, r_2 + r_{n-2}, r_3 + r_{n-3}, \ldots, r_{n-1} + p_1\}$$

(Bellman equation)

- If we knew the solutions of the smaller r_i values we would be done, because the optimal solution incorporates the optimal solutions to the smaller subproblems (max rev. of the two pieces) **(optimal substructure)**

- These subproblems may be solved independently of the original (larger) problem

The journey of 1000 miles begins with one step

- The rod cutting of *n* cm begins with one cut.
- Let r_i be the maximum revenue for a rod of length *i*.
  - Boundary case: $r_0 = 0$ (no rod to sell).
- If we make a first cut of length *i*, **the revenue from the first piece is p_i** and we are left with a rod of length *n-i*.
- **Optimal substructure**: we get an optimal revenue if
  - we make an optimal decision for the first cut length *i* and
  - we get optimal revenue for the remaining rod of length *n-i.*
- Leads to the following **Bellman equation**:

$$r_n = \max\{p_i + r_{n-i} \mid 1 \leq i \leq n\}$$

Bellman equations

$$r_n = \max\{p_i + r_{n-i} \mid 1 \leq i \leq n\}$$

- The **Bellman equation** tells us **how an optimal solution for a problem depends on solutions to smaller subproblems.**
  - It captures an **optimal decision** (e.g. which cut length *i* for 1st cut?)
  - The precise equation depends on the problem being solved. Different problems have different Bellman equations.
  - Named after **Richard Bellman**, the inventor of dynamic programming.
  - (Strangely, the book refuses to mention the term "Bellman equation".)
- The Bellman equation is **at the heart of a dynamic programming algorithm**.
  - Working it out can be hard work; implementation is usually straightforward once you have worked out the Bellman equation!

Same mistake again...

$$r_j = \max\{p_i + r_{j-i} \mid 1 \le i \le j\}$$

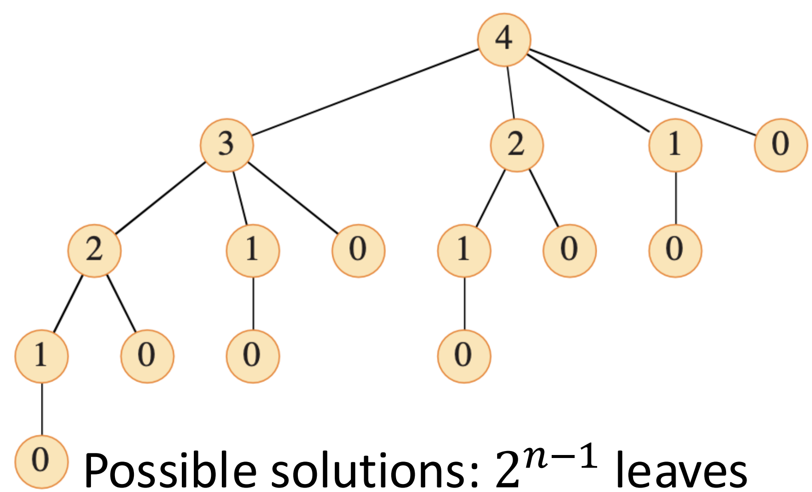
```
CUT-ROD(p, n)
if n == 0
    return 0
q = −∞
for i = 1 to n
    q = max {q, p[i] + CUT-ROD(p, n − i)}
return q
```

p[1,..n] : array of prices; *n* length

Revenue = 0
Minimum revenue is negative

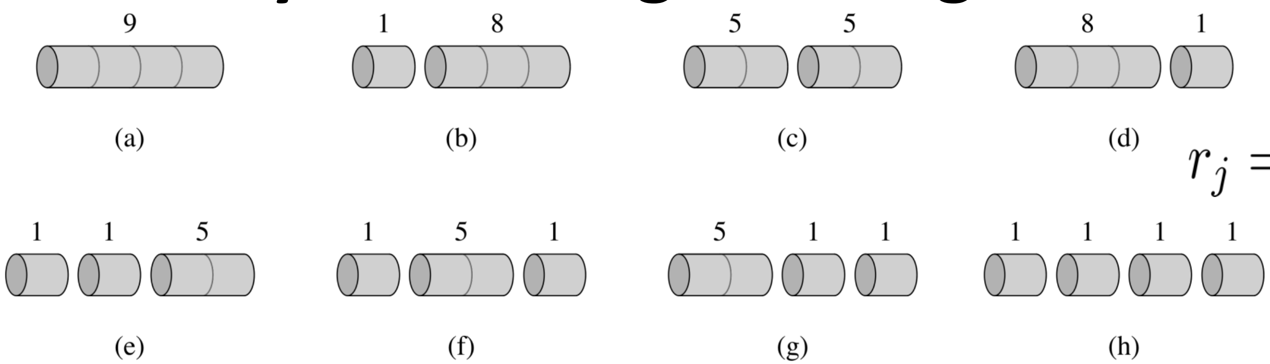
Recursively calculates Bellman eq.
Returns max

- Correctness: simple induction.
- Runtime?
  (T(n): n. of times Cut-Rod is called for size n)
- T(0) = 1
- $T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 2^n$
  (again by induction)



Possible solutions: 2^{n-1} leaves
(A path from root one of the possible ways to cut the rod)

Dynamic Programming



$$r_j = \max\{p_i + r_{j-i} \mid 1 \le i \le j\}$$

- Arrange for the problems to be **solved only once**
- If the rod had length n=1, what would be the optimal solution? $\boldsymbol{r_1 = p_1}$
- If the rod had length n=2? $\boldsymbol{r_2 = \max(p_2, r_1 + r_1) = \max(p_i + r_{2-i} \mid 1 \le i \le 2)}$
- Sort the subproblems by size, solve the smaller ones first, and **store the solutions**
  - That way, when solving a subproblem, we have already solved (and tabulated) the smaller subproblems we need.
- How many subproblems do we have when $n = 4$?

Bottom-up implementation

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

```
BOTTOM-UP-CUT-ROD(p, n)
Let r[0...n] be a new array
r[0] = 0
for j = 1 to n do
    q = −∞
    for i = 1 to j do
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Outer loop solves problem of rod length *j*

Inner loop computes Bellman equation:

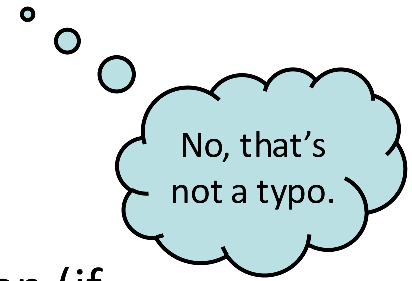
$$r_j = \max\{p_i + r_{j-i} \mid 1 \leq i \leq j\}$$

Correctness? Same as before

Runtime? Runtime is $\Theta(n^2)$.

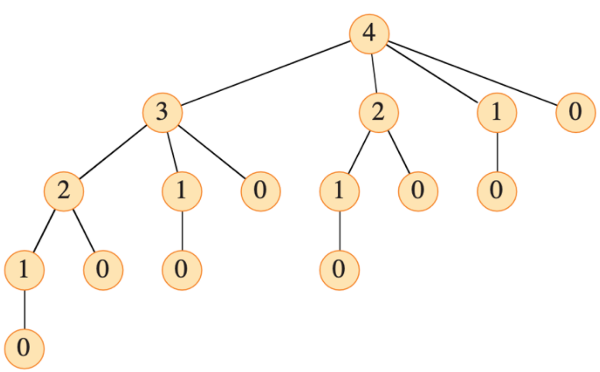
Top down Implementation with Memoization

- Alternative to bottom-up:
  - Write the recursive procedure naturally, but save the subproblem solutions somewhere
  - Recursive procedure first checks if it knows the solution (if so returns it); Otherwise proceeds and saves it



Runtime?

Same arithmetic series: $\Theta(n^2)$




```
MEMOIZED-CUT-ROD(p, n)
let r[0 : n] be a new array          // will remember solution values in r
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                  // already have a solution for length n?
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n           // i is the position of the first cut
        q = max {q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
r[n] = q                     // remember the solution value for length n
return q
```

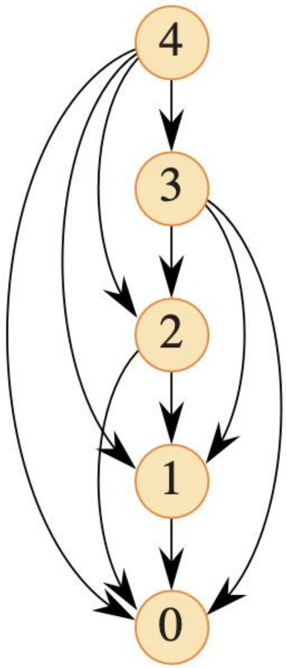
Top down Implementation with Memoization

No, that’s not a typo.

- Advantage:
  - Only solves problem sizes that are actually needed.
  - No better runtime for rod cutting, though.
- Disadvantage:
  - Bottom-up has better constant factors (lower overhead for recursive procedure calls)

Dynamic Programming: when to use

- The problem has Optimal Substructure.
- Runs in polynomial time when the number of **distinct** subproblems involved is polynomial in the input size **and** you can solve each subproblem in polynomial time.
- A subproblem graph indicates the subproblems that need to be solved before the larger problem can.
- **Top-Down:** arrows indicate the recursive calls
- **Bottom-Up:** solves the nodes "pointed at" before those "pointing to"
- Time to compute subproblem is proportional to degree of its node.
- Usually the runtime of dynamic programming is **linear in the number of vertices and edges**



Reconstructing a solution

- The algorithms only tell us the **value of the optimal revenue**, it **doesn't reveal how to cut**!

- **Solution**: if we know how to compute the optimal value, we can **record additional information** about how we got there (that is, **recording decisions** made in Bellman equations).

```
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 do
    q = −∞
    for i = 1 to j do
        if q < p[i] + r[j − i] then
            q = p[i] + r[j − i]
            s[j] = i
    r[j] = q
return r and s
```

Current best solution cuts at *i*
Store this information in *s*.

```
PRINT-CUT-ROD-SOLUTION(p, n)
(r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
while n > 0
    print s[n]          // cut location for length n
    n = n − s[n]        // length of the remainder of the rod
```

Summary

- Dynamic Programming is a **general design paradigm** that breaks down a problem into **smaller subproblems**; these are solved first and the solutions are usually tabulated.
- Works for optimisation problems with **optimal substructure**: the optimal solution is composed of optimal solutions for subproblems.
- The **Bellman equation** describes how an optimal solution is derived from optimal solutions for subproblems.
- Bottom-up approach solves subproblems of increasing size; Top-down solves recursively asking when needed
- The solution can be reconstructed by **recording decisions** made in applying Bellman equations across subproblems.
- The rod cutting problem can be solved this way in time $\Theta(n^2)$ reducing the runtime from exponential to a small polynomial.