COM1009
Introduction to Algorithms and Data Structures

Topic 07: Dynamic Programming

Essential Reading:
Section 15.1

▶ 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 (less detail than in the book).

How to compute Fibonacci numbers?

- Fibonacci numbers:
  - $Fib(0) = Fib(1) = 1$
  - $Fib(k) = Fib(k-1) + Fib(k-2)$

Approximate value of Fib(n)

- It's possible to show (by induction) that

$$Fib(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}\right]$$

Since $\frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^{n+1}$ = $\left(\frac{1-\sqrt{5}}{2\sqrt{5}}\right)\left(\frac{1-\sqrt{5}}{2}\right)^{n} \approx (0.276) \times (-0.618)^{n}$
the second term tends to 0 as n grows ever larger, so

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

▶ A simple implementation

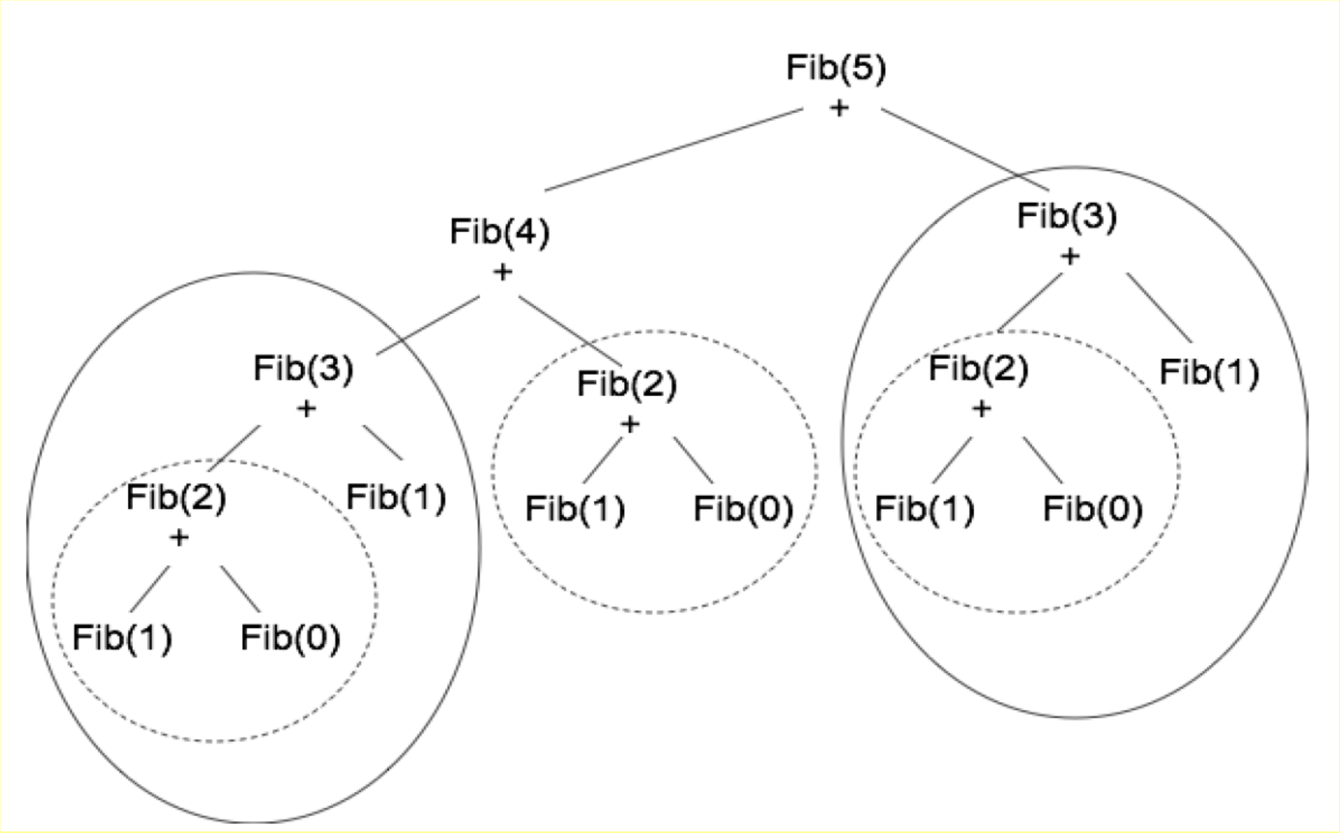
The formula on the last slide uses real numbers, but we can calculate it directly using integers, e.g. in Haskell:

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

What happens when we try to do this?

▶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: $T(0) = T(1) = 1$.
- Then $T(n) = T(n-1) + T(n-2) + 1$.
  - Ignore the "+1", then $T(n) = T(n-1) + T(n-2)$.
- Then $T = Fib$ so

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

- Note: $T(90) = Fib(90) = 4{,}660{,}046{,}610{,}375{,}530{,}309$.
  - 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 only ever refer to entries already calculated.)
- Time $O(n)$ instead of $\Omega\left(\left(\frac{\sqrt{5}+1}{2}\right)^n\right)$

▶Storing the results

```
fibs 0 = [1]

fibs 1 = [1,1]

fibs n = fibsSoFar ++ [fibn]

     where fibsSoFar = fibs (n-1)

           fibn = fibsSoFar[n-1] + fibsSoFar[n-2]


getFib n = (fibs n)[n]
```

Dynamic Programming

- **Use the same idea**:
  - solve **subproblems** of the original problem
  - **save the answers in a table**.
  - keep going 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 the subproblems typically "overlap" and you need to consider alternatives

Properties of Dynamic Programming

- Used for optimisation problems: find a solution with the best value.
- **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.

Example: Rod Cutting Problem

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

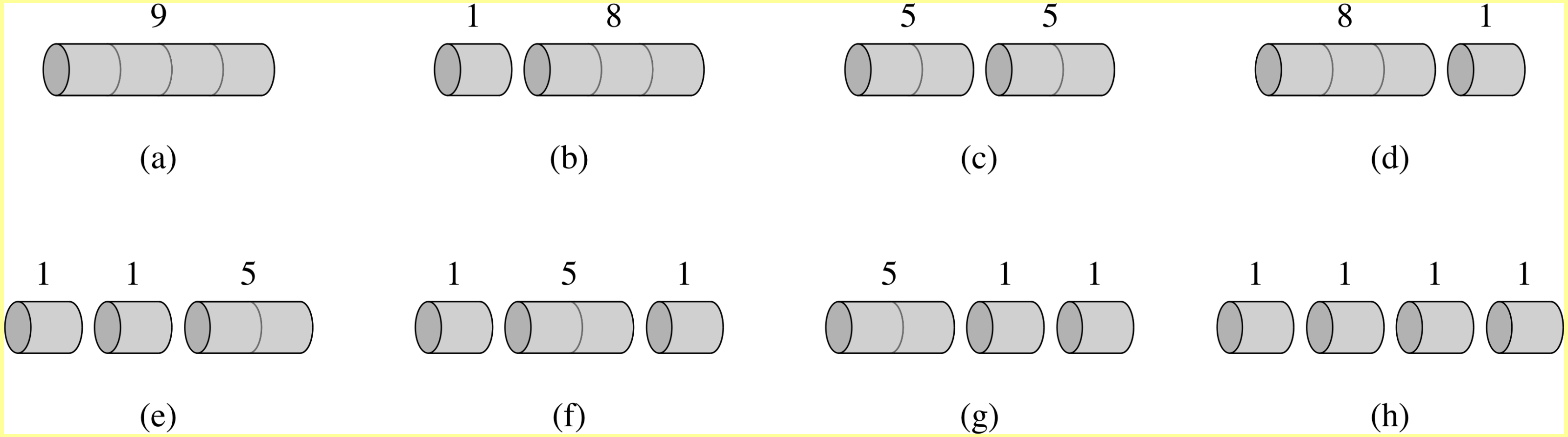
  - Each cut is free. Rod lengths are an integral number of cm.
  - Each rod length i has its own price p_i.
  - Output: maximum revenue obtainable from rods whose lengths sum to *n*, according to the price list.

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:



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.
- 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!

Bottom-up implementation

- Solve subproblems according to increasing size (smallest 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:

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

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

▶Implementation with Memoization

No, that's not a typo.

- Alternative to bottom-up:
  - solve problem top-down
  - **store solutions to subproblems in memory**
  - always access memory first, only compute a solution when it's not stored in memory.
- Similar idea to **caching** (cf. COM1006)
- Only solves problem sizes that are actually needed.
  - No better runtime for rod cutting, though.
- More details in the book for the curious.
- Let's stick with bottom-up as it's simpler.

Reconstructing a solution

- Bottom-up approach only tells 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*.

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.
- 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 $\Theta(n^2)$.