Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Dynamic programming

- Let's look at the problem of **finding the nth Fibonacci number**.

- We know the following recursion:

- $$F(n) = \begin{cases} 1 & n = 1 \text{ or } 2 \\ F(n-1) + F(n-2) & n > 2 \end{cases}$$

Dynamic programming

- So, we can use the following algorithm:

FIBONACCI(n)
1. **if** $n \leq 2$
2. **return** 1
3. **return** FIBONACCI(n-1)+ FIBONACCI(n-2)

Dynamic programming

FIBONACCI(n)
1. **if** $n \leq 2$
2. **return** 1
3. **return** FIBONACCI(n-1)+ FIBONACCI(n-2)

- **Question:** What is the time complexity?

Dynamic programming

FIBONACCI(n)
1. **if** $n \leq 2$
2. **return** 1
3. **return** FIBONACCI(n-1)+ FIBONACCI(n-2)

- **Answer:**

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

Dynamic programming

FIBONACCI(n)
1. **if** $n \le 2$
2.   **return** 1
3. **return** FIBONACCI(n-1)+ FIBONACCI(n-2)

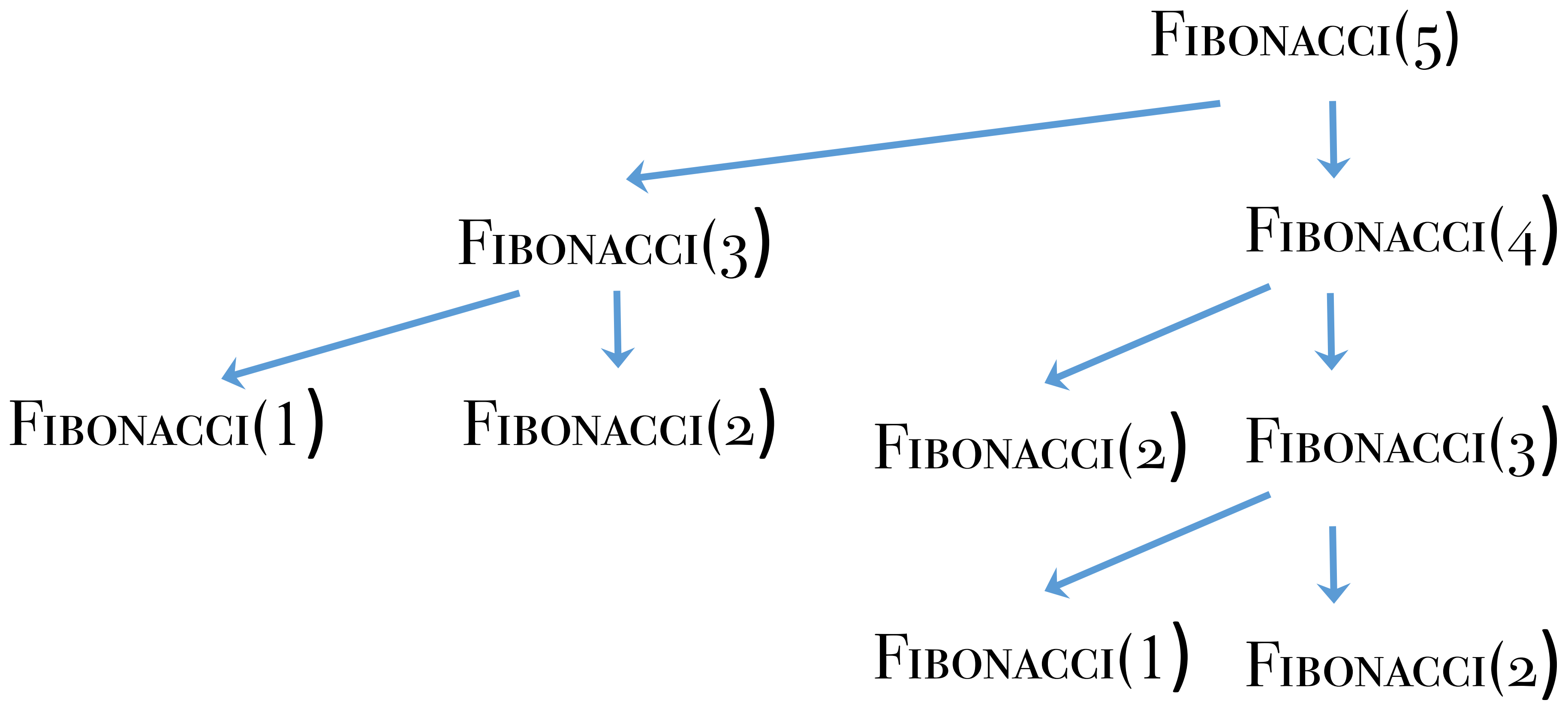
- **Answer:**

$$T(n) = T(n-1) + T(n-2) + c \geq 2T(n-2\,) + c$$

Using the recursion tree method we get $T(n) = \Omega\left(2^{n/2}\right)$ which is very inefficient!

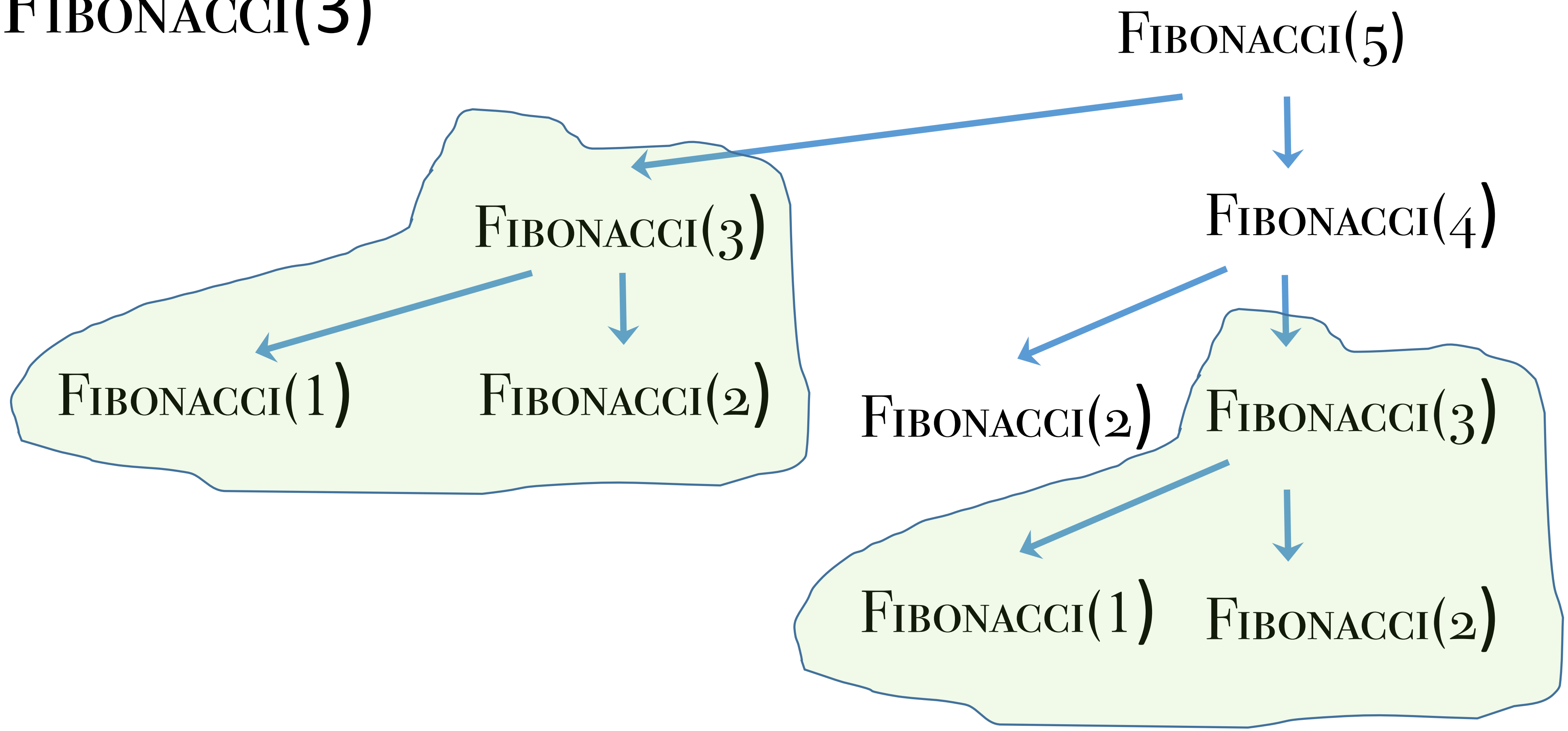
Dynamic programming

- Let's see what actually happens for FIBONACCI(5) as an example:



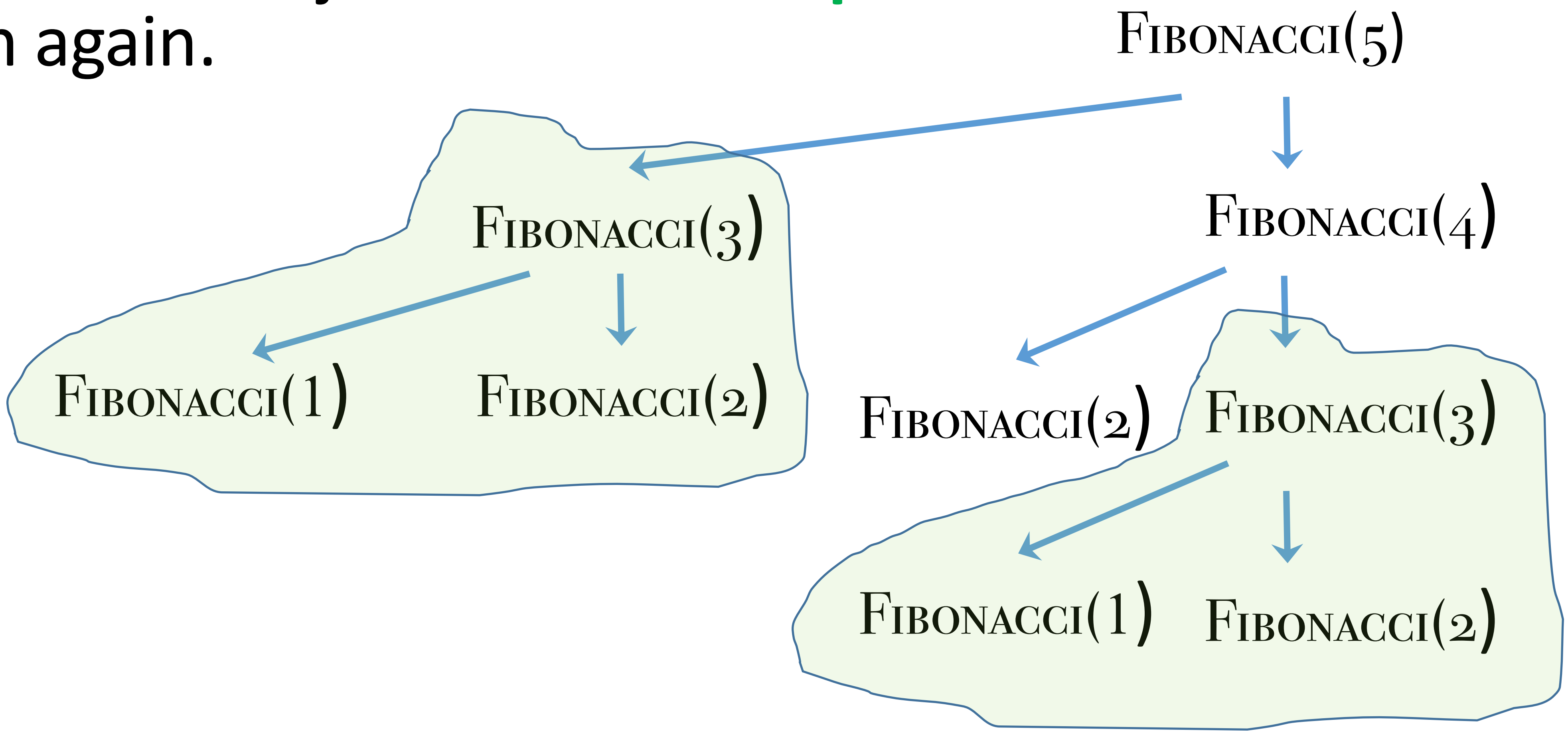
Dynamic programming

- We are solving some subproblems **more than once**. For example, FIBONACCI(3)



Dynamic programming

- So, it seems reasonable to **record the solution** to subproblems and just **look them up** if we need them again.



Dynamic programming

```
//visited is a Boolean array initialized with FALSE
//fib is an integer array to keep the actual answers
FIBONACCI(n)
if n ≤ 2
    return 1
if visited[n] == TRUE
    return fib[n]
visited[n] = TRUE
fib[n] = FIBONACCI(n-1)+FIBONACCI(n-2)
return fib[n]
```

Dynamic programming

```
//visited is a Boolean array initialized with FALSE
//fib is an integer array to keep the actual answers
FIBONACCI(n)
if n ≤ 2
    return 1
if visited[n] == TRUE
    return fib[n]
visited[n] = TRUE
fib[n] = FIBONACCI(n-1)+FIBONACCI(n-2)
return fib[n]
```

- **Question:** What is the time complexity now?

Dynamic programming

- **Answer:**
- Now, there are n **subproblems** that should be solved **exactly once**.

- The time complexity is $\boldsymbol{\Theta(n)}$ since any recursive call for a subproblem that has been solved before just needs $\boldsymbol{\Theta(1)}$ **time to return the recorded value**.

Dynamic programming

- **Dynamic programming** is a technique similar to ***divide-and-conquer*** that is used on problems that can be expressed as a number of **subproblems**.
- **Divide-and-conquer** is usually used when the **subproblems are brand new** problems. (e.g. sorting)
- **Dynamic programming** is used when many of the **subproblems overlap**.

Dynamic programming

- The term **"dynamic programming"** was coined in 1950s by Richard Bellman. (see wikipedia for the full story)

- **"Programming"** refers to a **tabular method**, not coding. In fact, is more similar to the term ***linear programming*** in mathematical optimization.

- **"Dynamic"** refers to the fact that we need multiple steps to build a solution.

Dynamic programming

- Elements of dynamic programming:

1. **Coming up with a recursive solution**

Dynamic programming

- Elements of dynamic programming:

1. Coming up with a recursive solution
2. **Convert the recursive solution to a recursive algorithm**

Dynamic programming

- Elements of dynamic programming:

1. Coming up with a recursive solution
2. Convert the recursive solution to a recursive algorithm
3. **Memoize the recursive algorithm**

✓ Don't confuse this with "memorized". **Memo-izing** a recursive algorithm means that the algorithm keeps a **memo to remember** the solutions it has computed before.

Dynamic programming

- Elements of dynamic programming:

1. Coming up with a recursive solution
2. Convert the recursive solution to a recursive algorithm
3. **Memoize the recursive algorithm**

✓ Don't confuse this with "memorized". **Memo-izing** a recursive algorithm means that the algorithm keeps a **memo to remember** the solutions it has computed before.

Dynamic programming

- There are two ways to implement a dynamic programming solution:

1. **Top-down**: It's the **recursive way** of implementing. Called top-down since the recursive calls are made from bigger problems (top) to smaller problems (bottom)

2. **Bottom-up:** It's the **non-recursive way using for loops**. It's called bottom-up since it builds the solutions to smaller problems first and then uses them to solve bigger problems.

Dynamic programming

- Here's the bottom-up implementation of Fibonacci number.

//*fib* is an integer array to keep the actual answers

FIBONACCI(n)

1. $fib[1] = fib[2] = 1$
2. **for** $i = 3$ to n
3. $\quad fib[i] = fib[i\text{-}1] + fib[i\text{-}2]$
4. **return** $fib[n]$

Dynamic programming

- You can think of the **bottom-up approach** as the **mathematical induction**.

//*fib* is an integer array to keep the actual answers

FIBONACCI(n)

1. $fib[1] = fib[2] = 1$ — **Base of induction**
2. **for** $i = 3$ to n
3. $\quad fib[i] = fib[i\text{-}1] + fib[i\text{-}2]$ — **Induction step**
4. **return** $fib[n]$

Rod-cutting

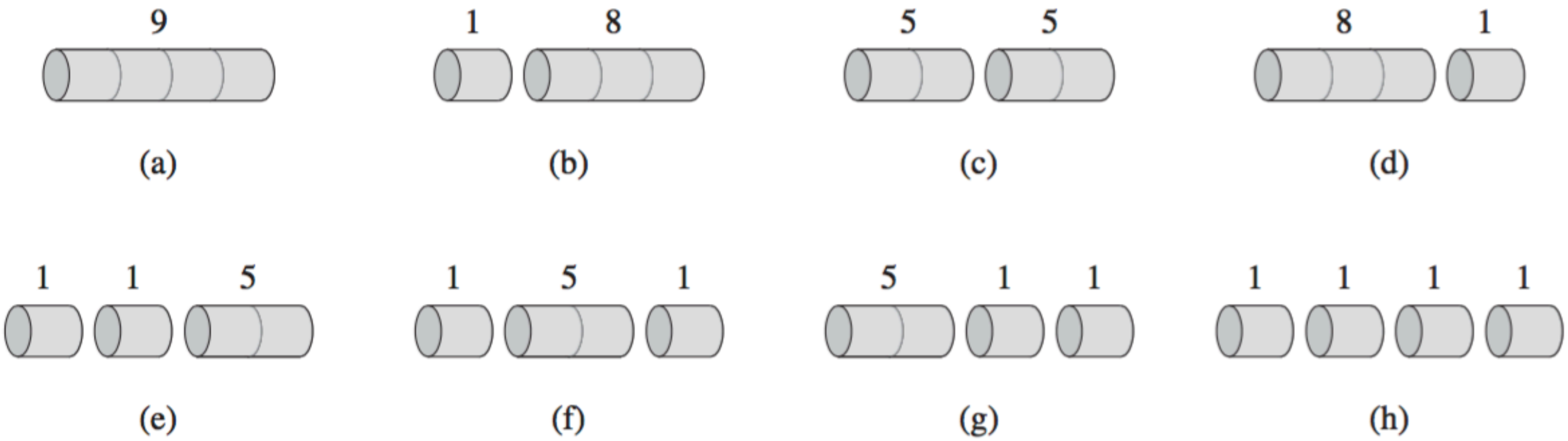
Rod-Cutting Problem

- Given a rod of length n, what is the ***maximum money*** we can earn by cutting the rod into different pieces?
- We can sell a rod of length k, for p_k dollars.
- The lengths and the prices are provided in a table, and the rod pieces can only have integer length.

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 8 possible ways to cut a rod of length 4:

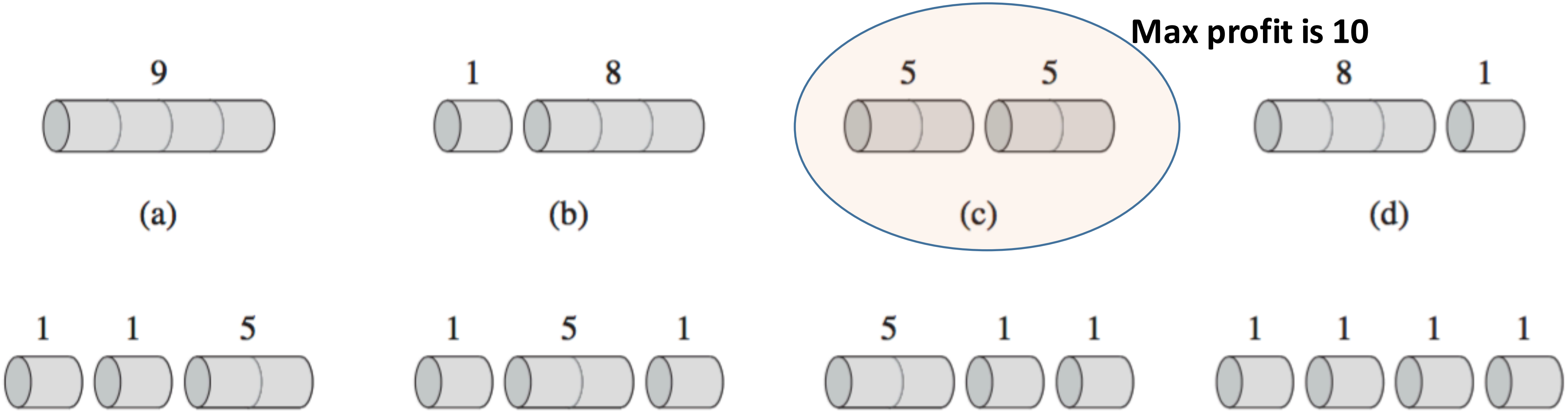


Example

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 8 possible ways to cut a rod of length 4:

Max profit is 10



(a) (b) (c) (d)

(e) (f) (g) (h)

Example

- Some of the ways that we can cut the rods are redundant for example getting two pieces of length 1 and 3 can happen in both scenario (b) and scenario (d) in the previous figure.

- However, we don’t care about this, since even if we optimize that we only make the algorithm 2 times faster at best which is not significant asymptotically.

Rod-cutting

- **Question:** but how can we formulate an optimal solution recursively for length n?

Rod-cutting

- **Question:** but how can we formulate an optimal solution recursively for length n?
- **Answer:** Consider all possible cases of **just one cut.** That one cut could be at length i where $1 \leq \ i < n$. One of these cuts must result in the optimal solution, so we gain the profit of p_i and also get a smaller subproblem of size $n - i$.
- We also have the option of not cutting at all. So, to generalize lets show that as a cut at length n, which gives us the profit of p_n. We can define the profit of a rod of length 0 to be 0 so we don't have to write an extra condition for not cutting.

Rod-cutting

- If r_n **is the max revenue** on a rod of **length n**, we have:

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

Rod-cutting

- If $\boldsymbol{r_n}$ **is the max revenue** on a rod of **length $\boldsymbol{n}$**, we have:

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

```
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

- The time complexity is

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$$

- We put 1 in the above equation for simplicity when we want to solve the recurrence, but in fact it is a constant.

Rod-cutting

- The time complexity is

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$$

- And we can use **induction** to show that $T(n) = 2^n$, we assume that $T(0) = 1$.

(Side note)

- There is also a simpler argument for **roughly showing** that the amount of work is exponential in n.
- The recursive algorithm is considering all possible ways of cutting the rod.
- However, for a rod of length n, there are $n-1$ places that you can cut the rod. For each of them either **you cut or you don't** so there are $2 \times 2 \times \cdots \times 2 = 2^{n-1}$ possible ways.

Rod-cutting

- The time complexity is

$$T(n) = 1 + \sum_{i=0}^{n-1} T(i)$$

- Since $T(n) = 2^n$, we **memoize** the recursive algorithm!

```
//r keeps the revenues and is initialized with −∞
//visited is a Boolean array initialized with FALSE
MEMOIZED-CUT-ROD(p, n) //p is the table that has the profit values
if n = 0
    return 0
if visited[n] == TRUE
    return r[n]
q = −∞
for i = 1 to n
    q = max(q, p[i] + MEMOIZED-CUT-ROD(p, n-i))
r[n] = q
visited[n] = TRUE
return r[n]
```

```
//r keeps the revenues and is initialized with −∞
//visited is a Boolean array initialized with FALSE
MEMOIZED-CUT-ROD(p, n) //p is the table that has the profit values
if n = 0
    return 0
if visited[n] == TRUE
    return r[n]
q = −∞
for i = 1 to n
    q = max(q, p[i] + MEMOIZED-CUT-ROD(p, n-i))
r[n] = q
visited[n] = TRUE
return r[n]
```

Question: What is the time complexity?

Analysis

Answer: As we mentioned before, in order to analyze the time complexity of a dynamic programming algorithm we have to see how many new subproblems we can have. Then, we should see how much time is required to compute the solution for each subproblem.

- Note that we don't try to come up with a recurrence for the analysis.

Analysis

Answer: There are n possible, one for each length and the lengths could be 1,2,…, or n.

On the subproblem for a rod of length i:

1. If it's first time that we see this subproblem we do a for loop from 1 to i and which takes $\boldsymbol{\Theta(i)}$
2. If it's not the first time we only spend $\boldsymbol{\Theta(1)}$ time.

- So, the overall complexity is

$$c\sum_{i=1}^{n} i = \Theta(n^2)$$

Rod-cutting

- The bottom-up implementation 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]
```

Rod-cutting

- The bottom-up implementation 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]
```

We have to solve the problem for a rod of **length j** before we can solve it for **length $j + 1$**

Rod-cutting

- The bottom-up implementation 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]
```

Considering all possible cuts for **length *j***

Rod-cutting

- But in reality we don’t just want the value of maximum revenue.

- We also want to know ***which choices*** lead to the maximum revenue.

- So, we have to come up with a way to ***keep the optimal choices***, as well.

Rod-cutting

- The idea is that **whenever we update** a the value for a rod of length j, we keep the length of the first piece that caused the update.

Rod-cutting

```
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
```

array *s* keeps the solutions

For length *j* this choice of *i* is the best so far.

Rod-cutting

- Now to print the optimal solution we **start at the original length which was n**, and print the length of first piece for n, i.e. $s[n]$. Then, we have to do the same thing for the **remaining part which has length $n - s[n]$.**

```
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]
```

Rod-cutting

- For example, if we call EXTENDED-BOTTOM-UP-CUT-ROD(p, 10) we get 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

- Let's say we want to the optimal choice for $n = 7$.

Rod-cutting

- First 1 is printed, then the remaining part will have the length of 7-1=6. So then, 6 will be printed and then the remaining part has a length of 6-6=0, and we are done. So, if your rod has **length 7 you have to cut to get rods of lengths 1 and 6 to gain the maximum profit**.

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