دانشگاه کردستان
University of Kurdistan

Dynamic Programming

Sadoon Azizi

s.azizi@uok.ac.ir

Department of Computer Engineering and IT

Spring 2019

Techniques for the design of Algorithms

- Divide and Conquer
- **Dynamic Programming**
- Greedy Algorithms
- Backtracking Algorithms
- Branch and Bound Algorithms

The main idea of dynamic programming

- Dynamic Programming (DP), like the Divide-and-Conquer (D&C) method, solves problems by combining the solutions to subproblems.

- Since there are a lot of common subproblems, therefore by using divide and conquer approach we have to solve all of them and this cause the exponential time complexity.

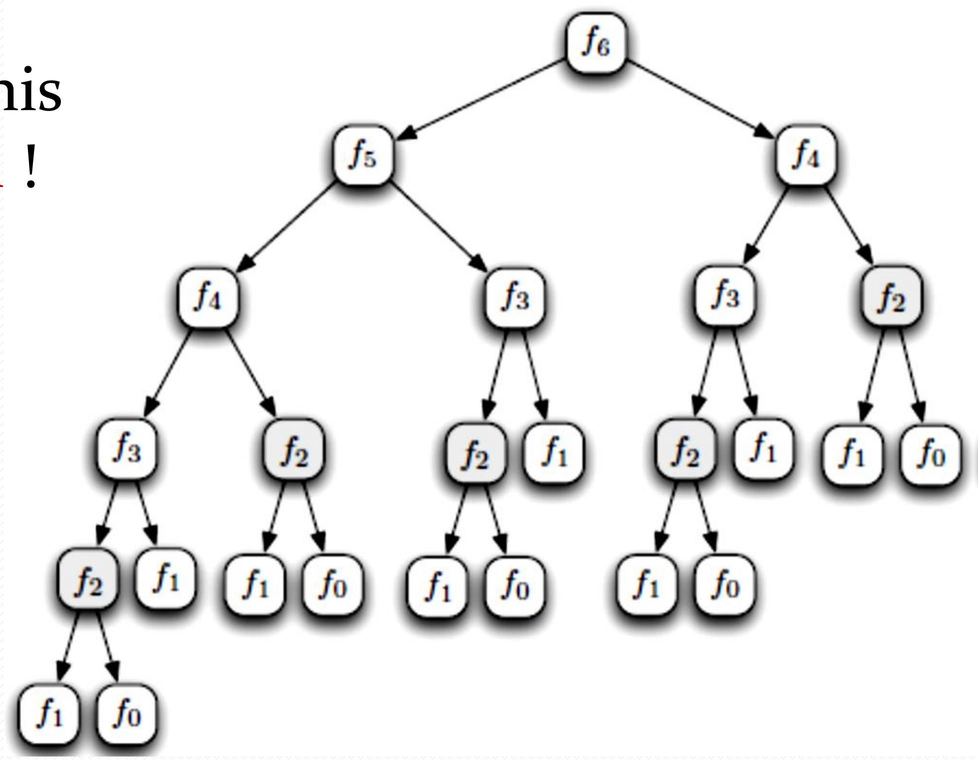
- Instead, we can solve each subproblem **exactly once** and **save** its answer for the future usage.

- We typically apply dynamic programming to **optimization problems**.

Fibonacci Sequence (D&C)

$$f(n) = \begin{cases} 0 & if\ n = 0 \\ 1 & if\ n = 1 \\ f(n-1) + f(n-2) & if\ n \geq 2 \end{cases}$$

❖The time complexity of this approach is **exponential** !



Fibonacci Sequence (DP)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610

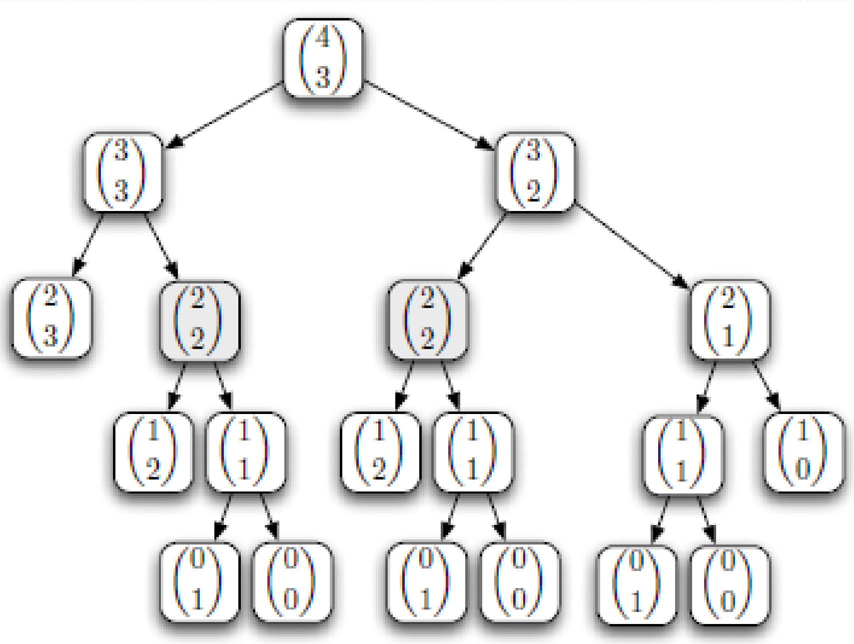
```
Fibo(n) {
int A[0..n], i;
A[0]=0;
A[1]=1;
for (i=2; i<=n; i++)
        A[i]=A[i-1]+A[i-2];
return A[n];
}
```

Time complexity: O(n)
Space complexity: O(n)

Q: Can we reduce Space complexity?

Choosing k objects among n objects (D&C)

$$\binom{n}{k} = \begin{cases} 0 & \text{if } n < k \\ 1 & \text{if } k = 0 \text{ or } k = n \\ \binom{n-1}{k} + \binom{n-1}{k-1} & \text{otherwise} \end{cases}$$



Choosing k objects among n objects (DP)

$$B[i][j] = \begin{cases} 0 & \text{if } j < i \\ 1 & \text{if } j = 0 \text{ or } j = i \\ B[i-1][j] + B[i-1][j-1] & 0 < \mathrm{j} < \mathrm{i} \end{cases}$$

j

		0	1	2	3	4
	0	1	0	0	0	0
	1	1	1	0	0	0
	2	1	2	1	0	0
	3	1	3	3	1	0
i	4	1	4	6	4	1
	5	1	5	10	10	5
	6	1	6	15	20	15
	7	1	7	21	35	35
	8	1	8	28	56	70

Choosing k objects among n objects (DP)

$$B[i][j] = \begin{cases} 0 & \text{if } j < i \\ 1 & \text{if } j = 0 \text{ or } j = i \\ B[i-1][j] + B[i-1][j-1] & 0 < \mathrm{j} < \mathrm{i} \end{cases}$$

```
Choose(k,n) {
  int i,j,B[0..n][0..k];
  for(i=0; i<=n; i++)
    for(j=0; j<=min(i,k); j++)
      if(j==0 || j==i)
        B[i][j]=1;
      else
        B[i][j]=B[i-1][j]+B[i-1][j-1];
  return B[n][k]; }
```

Time complexity: O(nk)
Space complexity: O(nk)

Q: Can we reduce Space complexity?

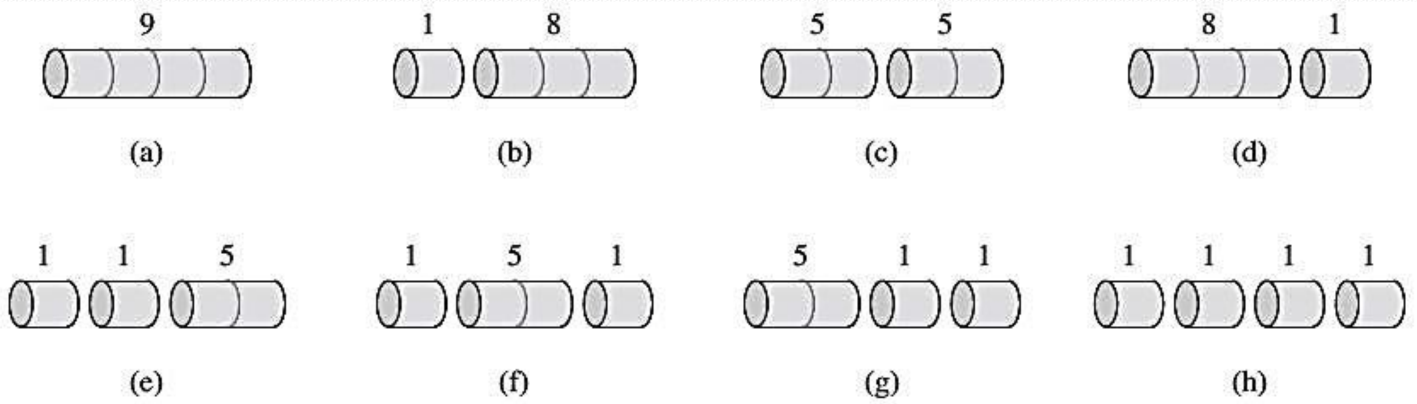
The rod-cutting problem

- The ***rod-cutting problem*** is the following:
- Given a rod of length n inches and a table of prices p_i for i=1,2, …, n .
- Determine the **maximum revenue r_n** obtainable by cutting up the rod and selling the pieces.

The rod-cutting problem

- **Example:**

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



The rod-cutting problem

- Q: How many ways are there to cut up a rod of length n?
- **A:** 2^{n-1} (why?)

The rod-cutting problem

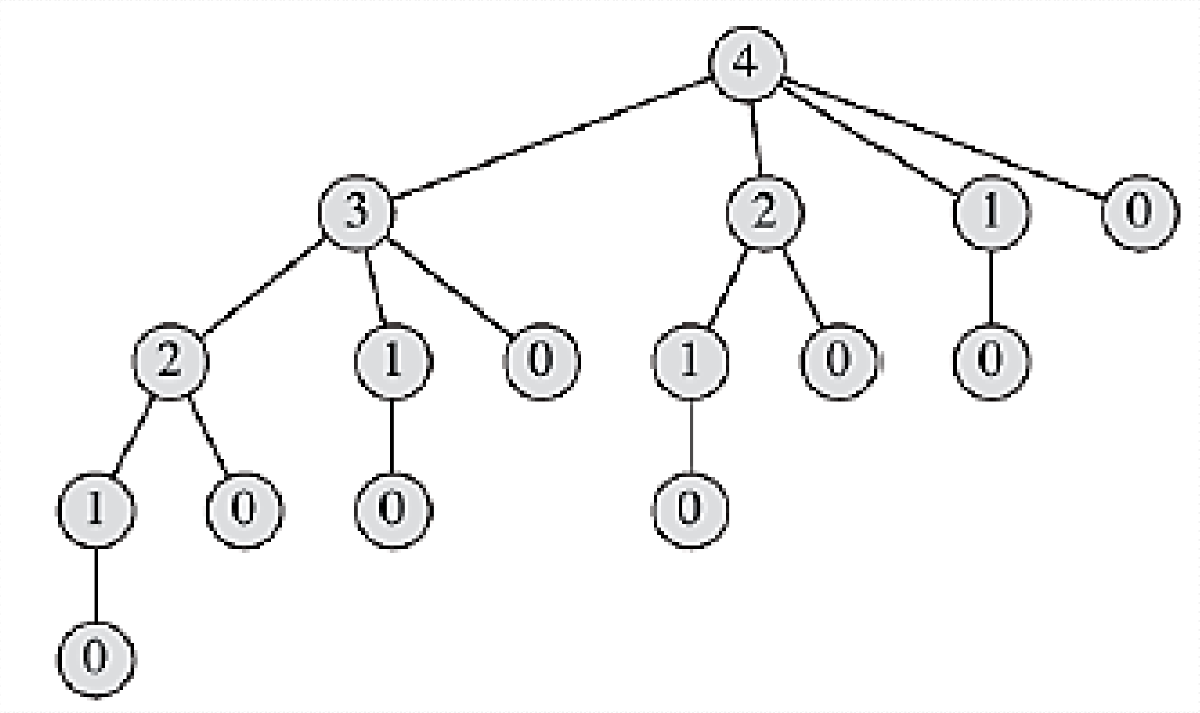
- Provide a recursive relationship to the problem

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

- Set $r_0 = 0$
- Cut a piece of length i , with remainder of length n-i
- Only the remainder may be further divided

The rod-cutting problem

- Example (n=4):



The rod-cutting problem

- After solving a subproblem, store the solution
  - Next time you encounter same subproblem, lookup the solution, instead of solving it again
  - Uses space to save time
- Two main methodologies: top-down and bottom-up
  - Corresponding algorithms have the same asymptotic cost, but bottom-up is usually faster in practice
- Main idea of bottom-up
  - Don't wait until subproblem is encountered.
  - Sort the subproblems by size; solve smallest subproblems first
  - Combine solutions of small subproblems to solve larger ones

The rod-cutting problem (top-down)

```
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
```

The rod-cutting problem (bottom-up)

```
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]
```

مسئله برش میله

- The time complexity of MEMOIZED-CUT-ROD (top-down)

$$\theta(n^2)$$

- The time complexity of BOTTOM-UP-CUT-ROD

$$\theta(n^2)$$

The rod-cutting problem (Reconstructing a solution)

```
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
```

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

* for n=10, would print just 10

* for n=7, would print 1 and 6

```
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]
```