# CSCI-C311 Programming Languages

## Dynamic Programming

Dr. Hang Dinh

1

## Outline and Reading

- After this lecture, you will learn
  - What dynamic programming is
  - How to apply dynamic programming
  - Where to apply dynamic programming
- Reference
  - Cormen et al., *Introduction to Algorithms*, 3rd Edition - Section 15.1

2

# Divide and Conquer

- It’s a method for solving problems by
  - dividing the problem into *independent* subproblems
  - conquering  each subproblem.
- Example: the binary search algorithm
  - Search a sorted array by dividing the array into two halves and reducing the search to within one half

| **Lower half** | **middle** | **Upper half** |
|---|---|---|
| a[i]<=a[middle] | a[middle] | a[i]>=a[middle] |

Search here if key<a[middle]

Search here if key>a[middle]

- Easy to be implemented using recursion

3

# Dynamic Programming

- Like the *divide-and-conquer* method
  - Solves problems by combining solutions to subproblems.
  - “Programming” in this context refers to a tabular method, not to writing code
- Unlike the divide-and-conquer method
  - Dynamic programming applies when the subproblems overlap, i.e., when subproblems share subsubproblems
  - In this context, a divide-and-conquer algorithm does more work than necessary, repeatedly solving the common subsubproblems
  - Dynamic programing solves each subsubproblem just once and saves its answer in a table.

4

# Dynamic Programming

- Typically apply to ***optimization problems***
  - Such problems can have many possible solutions, each has a value
  - Wish to find a solution with the optimal (minimum or maximum) value.
  - There may be many such *optimal solutions* to the same problem.
- Steps to develop a dynamic programming algorithm:
  1. Characterize the structure of an optimal solution
  2. **Recursively** define the value of an optimal solution
  3. Compute the value of an optimal solution, typically in a bottom-up fashion
  4. Construct an optimal solution from computed information

5

# Optimization Problem: Rod Cutting

- Background:
  - Serling Enterprises buys long steel rods and cuts them into shorter rods to sell
  - The management wants to know the best way to cut up the rods.
- Assumptions
  - Rod lengths are always an integral number of inches
  - Serling Enterprises charges $p_i$ dollars for a rod of length $i$ inches.

| **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 |

A sample price table for rods

6

## Optimization Problem: Rod Cutting

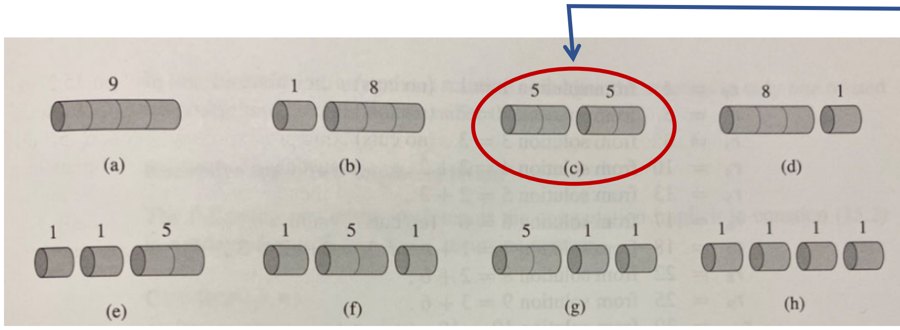
- The ***rod-cutting problem*** definition:
  - **Input**: a rod of length $n$ inches and a table of price $p_i$ for $i = 1, 2, \dots, n$.
  - **Output**: the maximum revenue $r_n$ obtainable by cutting up the rod and selling the pieces
- Note: if the price $p_n$ for a rod of length $n$ is large enough, an optimal solution may require no cutting at all.
- We can cut a rod of length $n$ in $2^{n-1}$ different ways
  - Since we have independent of option of cutting or not cutting at distance $i$ inches from the left end, for each $i = 1, 2, \dots, n-1$


7

## Optimization Problem: Rod Cutting

- Example: when $n = 4$
  - There are 8 different ways of cutting the rod

| Length $i$ | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Price $p_i$ | 1 | 5 | 8 | 9 |



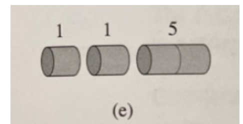

Optimal solution

8

# Representing Solutions to Rod Cutting

- Each solution to cutting a rod of length $n$ that results in $k$ pieces of lengths $i_1, i_2, \dots, i_k$ will be denoted using additive notation as:

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




  - Example: 4 = 1 + 1+ 2 represents solution (e) on the right:
  - If $n = i_1 + i_2 + \cdots + i_k$ is an optimal solution, then the maximum revenue, denoted $r_n$, is

$$r_n = p_{i_1} + p_{i_2} + \ \dots . + p_{i_k}$$

- Subproblem: For each positive integer $m \le n$
  - Let $r_m$ be the maximum revenue for the problem of cutting a rod of length $m$ using the same price table of the original problem (cutting rod of length $n$)

9

# Determine Optimal Values of Subproblems for Rod Cutting

- Example $n = 10$:

| 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 |

  - $r_1 = 1$ from solution 1 = 1 (no cut)
  - $r_2 = 5$ from solution 2 = 2 (no cut)
    - No cut → revenue \$5
    - Cut into two 1-inch pieces → revenue \$2
  - $r_3 = 8$ from solution 3 = 3 (no cut)
    - No cut → revenue \$8
    - First cut at length 1 → max revenue = $r_1 + r_2$ = \$6
    - First cut at length 2 → max revenue = $r_2 + r_1$ = \$6

10

# Determine Optimal Values of Subproblems for Rod Cutting

- Example $n = 10$:

| 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 |

  - $r_1 = 1$ from solution 1 = 1 (no cut)
  - $r_2 = 5$ from solution 2 = 2 (no cut)
  - $r_3 = 8$ from solution 3 = 3 (no cut)
  - $r_4 = 10$ from solution 4 =2+2
    - No cut → revenue \$9
    - First cut at length 1 → max revenue = $r_1 + r_3$ = \$9
    - First cut at length 2 → max revenue = $r_2 + r_2$ = \$10
    - First cut at length 3 → max revenue = $r_3 + r_1$ = \$9

11

# Determine Optimal Values of Subproblems for Rod Cutting

- Example $n = 10$:

| 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 |

  - $r_1 = 1$ from solution 1 = 1 (no cut)
  - $r_2 = 5$ from solution 2 = 2 (no cut)
  - $r_3 = 8$ from solution 3 = 3 (no cut)
  - $r_4 = 10$ from solution 4 =2+2
  - $r_5 = 13$ from solution 5 = 2+3 or 5=3+2
    - No cut → revenue \$10
    - First cut at length 1 → max revenue = $r_1 + r_4$ = \$11
    - First cut at length 2 → max revenue = $r_2 + r_3$ = \$13
    - First cut at length 3 → max revenue = $r_3 + r_2$ = \$13
    - First cut at length 4 → max revenue = $r_4 + r_1$ = \$11

12

## Determine Optimal Values of Subproblems for Rod Cutting

- Example $n = 10$:

| 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 |

  - $r_1 = 1$ from solution 1 = 1 (no cut)
  - $r_2 = 5$ from solution 2 = 2 (no cut)
  - $r_3 = 8$ from solution 3 = 3 (no cut)
  - $r_4 = 10$ from solution 4 =2+2
  - $r_5 = 13$ from solution 5 = 2+3
  - $r_6 = 17$ from solution 6 = 6 (no cut)
  - $r_7 = 18$ from solution 7 = 1+6 or 7=2+2+3
  - $r_8 = 22$ from solution 8 = 2+6
  - $r_9 = 25$ from solution 9 = 3+6
  - $r_{10} = 30$ from solution 10 = 10 (no cut)

13

## Recursively Define Optimal Values for Rod Cutting

- Generally, we can frame the values $r_n$ for $n \geq 1$ in terms of optimal revenue from shorter rods:

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

  - The first argument of function max corresponds to no cuts at all
  - The argument $r_i + r_{n-i}$ for $i = 1, 2, \dots, n-1$ corresponds to the optimal solution among strategies of making the first cut of the rod at length $i$ inches.
- The rod-cutting problem exhibits ***optimal substructure***:
  - Optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently

14

# A Simpler Recursive Structure for Rod Cutting

- View a decomposition of a rod of length $n$ as consisting of
  - a first piece of length $i$ from the left end
  - followed a decomposition of right-hand remainder of length $n-i$.
- The no-cut solution corresponds to the decomposition with the first piece of length $n$. By convention, $r_0 = 0$.
- Simpler version of recursive equation for $r_n$:

$$r_n = \max_{1 \le i \le n} \underbrace{(p_i + r_{n-i})}_{\text{maximal revenue of all solutions whose first piece from the left end has length } i}$$

15

# Naïve Recursive Implementation in C/C++ for Rod Cutting

```
int CutRod(int p[], int n){
     if (n==0)
          return 0;
     int q=-1;
     for(int i=1; i<=n; i++)
          q=max(q, p[i] + CutRod(p, n-i));
     return q;
}
```

For Java, change this to `int[] p`

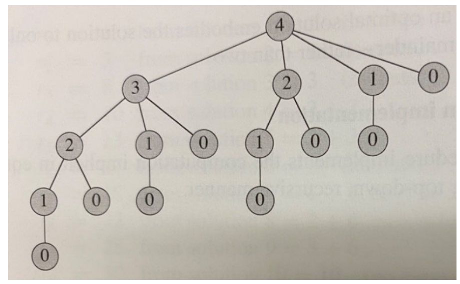
$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

This CutRod function is very SLOW!!!

16

## Why is Naïve Recursion Inefficient?

- It solves the same subproblems repeatedly
  - `CutRod(p,n)` calls `CutRod(p,n-i)` for all `i=1,2..,  n`
  - Equivalently, `CutRod(p,n)` calls `CutRod(p,j)` for all `j=0, 1..,n-1`
- Recursive tree for `CutRod(p,4)`
  - `CutRod(p,3)` is called 1 time
  - `CutRod(p,2)` is called 2 times
  - `CutRod(p,1)` is called 4 times
  - `CutRod(p,0)` is called 8 times



- Can prove by strong induction:
  - The number of nodes in the recursive tree for `CutRod(p,n)` is $2^n$

17

## Using Dynamic Programming For Rod Cutting

- Arrange for each subproblem to be solved only **once**, saving its solution
  - 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 time.
- Two equivalent ways to implement dynamic programing
  - ***Top-down with memorization***: write the procedure recursively in a natural manner, but modified to save the result of each subproblem.
  - ***Bottom-up method***: sort the subproblems by size and solve them in size order, smallest first.

18

# Top-down with Memorization for Rod Cutting

```
int MemorizedCutRod_Aux(int p[], int n, int r[]){
    if (r[n]>=0) return r[n];
    int q=0;
    if (n>0){
        q=-1;
        for(int i=1; i<=n; i++)
            q=max(q, p[i]+ MemorizedCutRod_Aux(p, n-i, r));
    }
    r[n] = q;
    return q;
}
```

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

19

# Top-down with Memorization for Rod Cutting

```
int MemorizedCutRod(int p[], int n){
    int* r = new int[n+1];
    for(int i=0; i<=n; i++)
        r[i]=-1;
    int q= MemorizedCutRod_Aux(p, n, r);
    delete r;
    r=NULL;
    return q;
}
```

20

# Bottom-Up Method for Rod Cutting

```
int ButtomUpCutRod(int p[], int n){
	int* r = new int[n+1];
	r[0]=0;
	for(int j=1; j<=n; j++) {
		int q=-1;
		for(int i=1; i<=j; i++)
			if(q<p[i]+r[j-i])
				q=p[i]+r[j-i];
		r[j]=q;
	}
	return r[n];
}
```

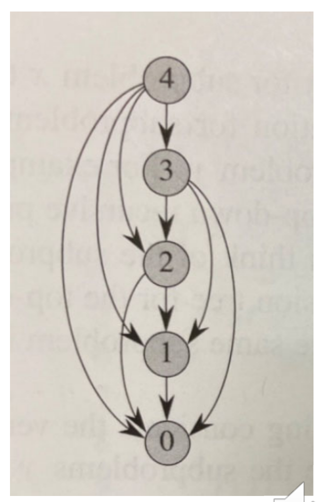
$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i})$$

$$r_j = \max_{1 \le i \le j} (p_i + r_{j-i})$$

21

# Subproblem Graphs

- The ***subproblem graph*** for a dynamic programming problem shows how subproblems depend on one another.
  - It's a directed graph
  - Each vertex corresponds to a subproblem
  - Each directed edge (x, y) indicates solving subproblem x involves an optimal solution of subproblem y
- Example: Subproblem graph for rod cutting with n=4




22

# Reconstructing a Solution for Rod Cutting

- Extend the dynamic programming approach to also record a choice that led to the optimal value.

s[j] saves the optimal length of the first piece to cut off from a rod of length j.

```
int Ext_ButtomUpCutRod(int p[], int n){
    int* r = new int[n+1];
    int* s = new int[n+1];
    r[0]=0;
    for(int j=1; j<=n; j++) {
        int q=-1;
        for(int i=1; i<=j; i++)
            if(q<p[i]+r[j-i]) {
                q=p[i]+r[j-i];
                s[j] = i;
            }
        r[j]=q;
    }
    PrintSolution(n, s);
    return r[n];
}
```

23

# Print Optimal Solutions to Rod Cutting

- Example
  - Input: n=10
  - Output:

| 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 |

| $j$ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| r[j] | 0 | 1 | 5 | 8 | 10 | 13 | 17 | 18 | 22 | 25 | 30 |
| s[j] | 0 | 1 | 2 | 3 | 2 | 2 | 6 | 1 | 2 | 3 | 10 |

```
void PrintSolution(int n, int s[]){
    while(n>0){
        printf("%d ", s[n]);
        n = n - s[n];
    }
}
```

If array s has values as above,

- `PrintSolution(5,s)` prints **2 3**
- `PrintSolution(6,s)` prints **6**
- `PrintSolution(7,s)` prints **1 6**
- `PrintSolution(9,s)` prints **3 6**

24