Dynamic Programming

Rod Cutting

Rod cutting

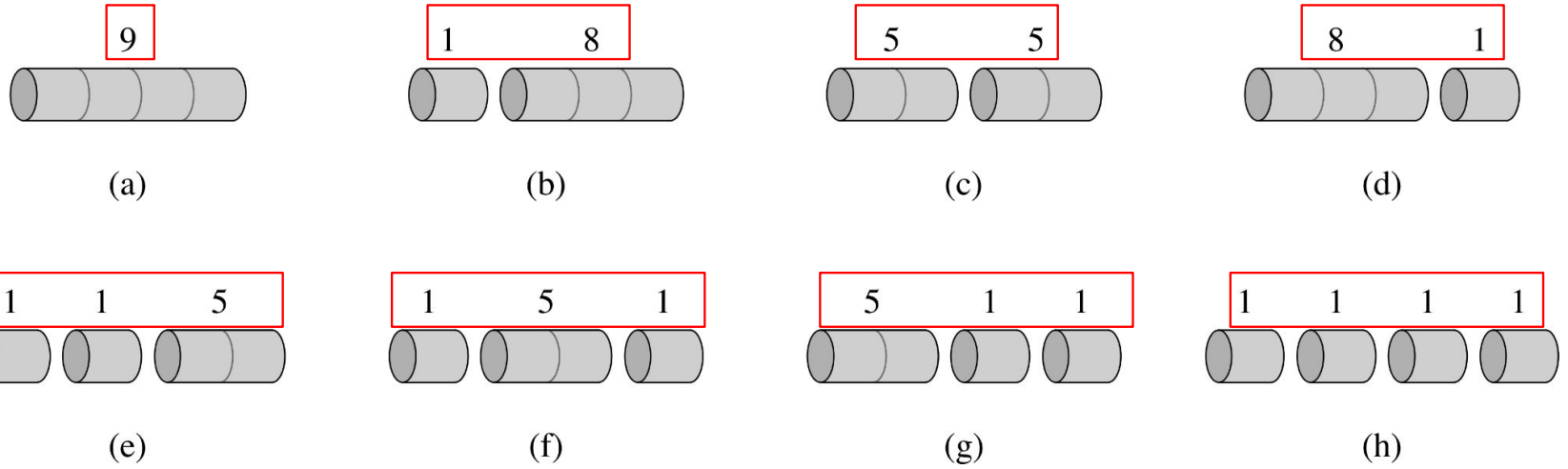
- **Input:**
  - a rod of length n inches
  - A table of prices p
- **Output**:
  - maximum revenue by cutting up the rod and selling the pieces
  - If p_i is large for a certain length, no cutting is needed

LENGTH	PRICE
1	1
2	5
3	8
4	9
5	10
6	17
7	17
8	20
9	24
10	30

Rod cutting: Example

LENGTH	PRICE
1	1
2	5
3	8
4	9

- The are 8 possible ways of cutting up a rod of **length 4**
  - Above each piece is the value of that piece
  - The revenue for each strategy is sum of the values of pieces
- Optimal strategy is (c)



Rod cutting: Brute Force

- The rod has length n
- At distance i from left, there are two options:
  - Cut at distance i
  - No cut at distance i
- Runtime: There are 2^{n-1} possibilities
  - Runtime: $O(2^n)$

Rod cutting

- Can we solve it using divide-and-conquer?

Rod cutting: Optimal Substructure

- Rod cutting problem has optimal substructure property
  - Can we express the solution to the problem in terms of the solution to the subproblems (rods with smaller length)?
  - The optimal solution for a rod of length n ($\mathbf{r_n}$) contains the optimal solution to $\mathbf{r_{n-i}}$
    - If $\mathbf{r_{n-i}}$ is not optimal, we can replace it with the optimal value, and get a better value for $\mathbf{r_n}$
  - **Each piece of rod is independent of the other piece after a cut: subproblems are independent**

Therefore, it can be solved using dynamic programming

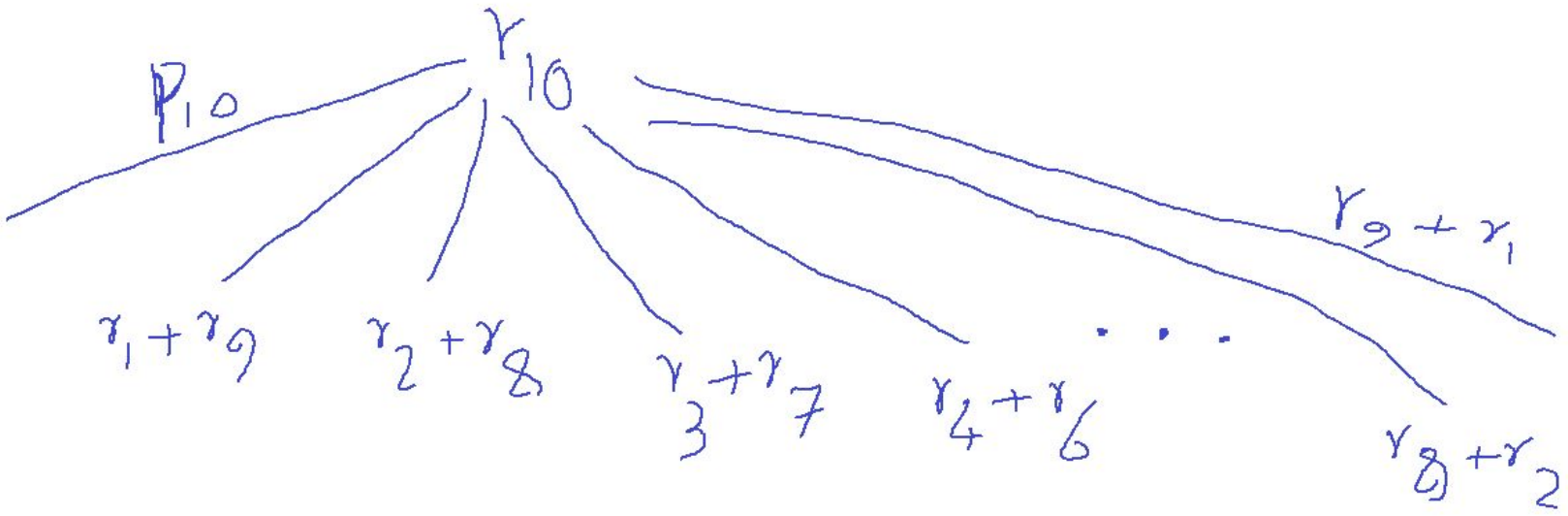
Rod cutting: Finding Recursive Formula

- $\mathbf{r_n}$: The maximum revenue of a rod with length n
- To solve the problem recursively:
  - First cut the rod at some length i
  - Solve the problem recursively for each part
  - Pick the length i so that $r_i + r_{n-i}$ is maximum
- What should be the value of i? We don't know, So we try all possible cases
- r_0 is when the rod is not cut = p_n

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

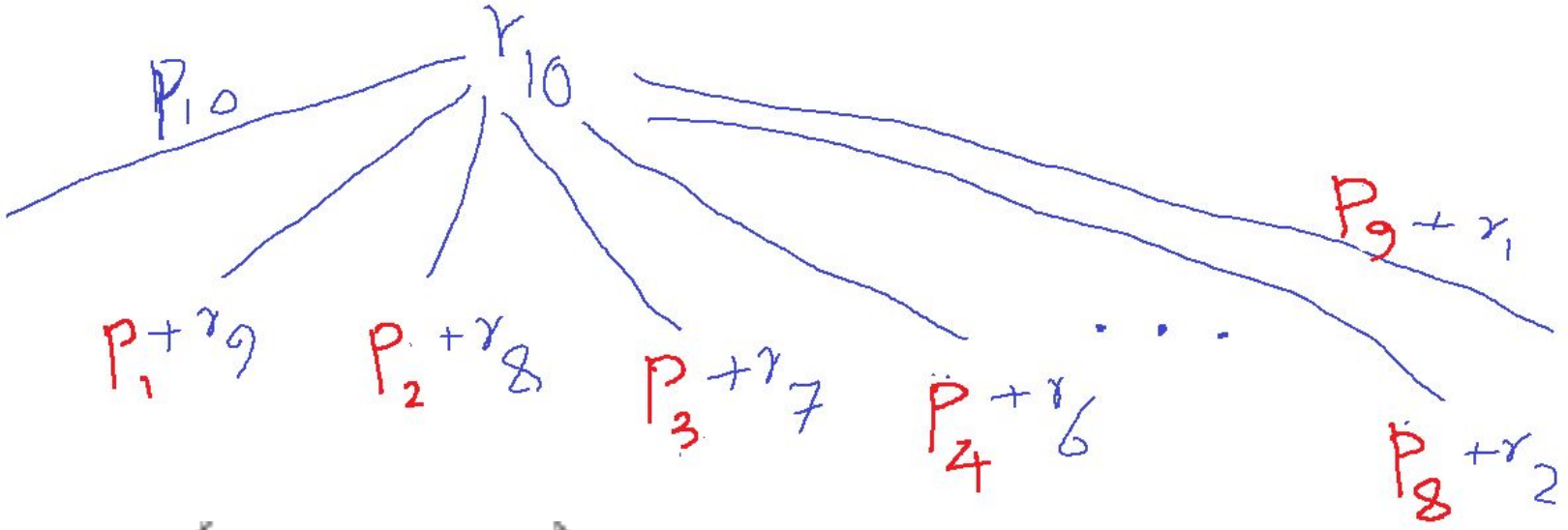
Rod cutting: Finding Recursive Formula

- We have repetition here: $r_2 + r_8$ and $r_8 + r_2$
- We just cut at length i from the left and solve recursively for the right part



Rod cutting: Finding Recursive Formula

- A better recursive relation
- Try cutting a piece at length i, and combining it with the optimal way to cut a rod of length n - i
- We are trying all the possible length and picking up the best (maximum revenue)
- i=n is the case where there is no cut at all



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

Rod cutting: Proof of correctness

- **Theoreme:** cut-rod(p, n) returns the maximum revenue for a rod of length n
- **Base-case:** n==0, the rod is of length 0 and therefore no revenue. The function also returns 0
- **Strong Induction hypothesis:** cut-rod(p, k) returns the maximum revenue for all rod of length k where k < n
- **Induction step:** having IH, we want to prove that cut-rod(p, k) returns the maximum revenue for a rod of length k
  - Maximum revenue for length k = $\max_{1 \le i \le k}(p[i] + \text{cut-rod}(p, k - i))$
  - The above equation is checking all possibilities and taking the maximum. In each one it is using solution to the smaller subproblem (which is maximum by induction hypothesis), therefore it is generating the optimal solution.

Rod cutting: Implementation

- Recursive implementation:
  - Runtime: Exponential
- Recursive implementation with memoization: Dynamic Programming
  - Top-down dynamic programming
  - Runtime: Polynomial
- Iterative implementation: Dynamic Programming
  - Runtime: Polynomial

Rod cutting: Recursive Implementation

Runtime:

T(n) = T(n-1) + T(n-2) + T(n-3) + T(n-4) + … + T(1) + T(0)

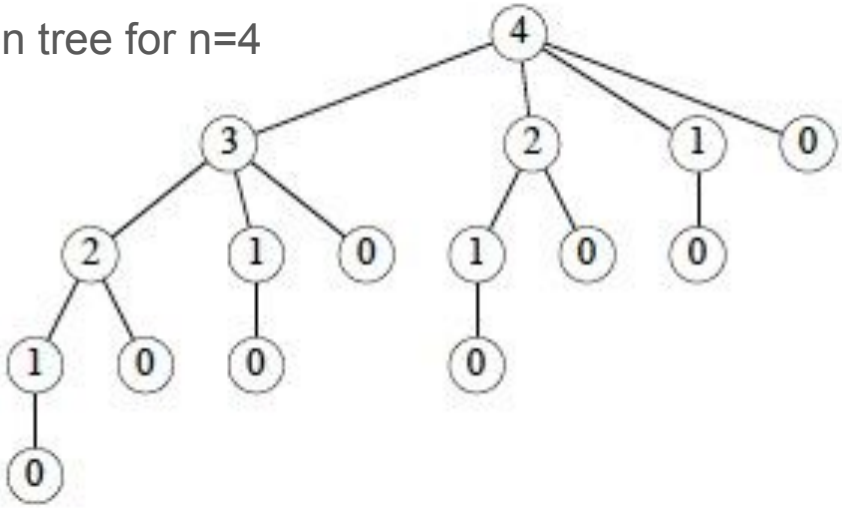
T(0) = 1

```
def cut-rod(p, n)
    If n==0:
        return 0
    q = negative infinity
    for i=1 to n:
        q = max (q, p[i]+ cut-rod(p, n-i))
    return q
```

It is exponential

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

The recursion tree for n=4



Rod cutting: Recursive Implementation: Runtime Analysis

- Calculating T(n):Recurrence relation with a full history

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

- Using substitution method
  - Finding a guess by repeatedly plugging the recursion into itself

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

  - Proving the guess using induction

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

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

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

$$T(n+1) = 2 * 2T(n-1) = 2 * 2 * 2T(n-2) = 2 * 2 * 2 * 2T(n-3) = 2^n T(1) = 2^n$$

$$Guess : T(n) = 2^n$$

Rod cutting: Recursive Implementation: Runtime Analysis

- Our guess: T(n) = 2^n
- **Proof by induction:**

$$Guess: T(n) = 2^n$$

$$\text{Base case: } T(0) = 2^0 = 1 \text{ whcih is true}$$

$$\text{Induction hypothesis: if } T(k) = 2^k \text{ then } T(K+1) = 2^{k+1}$$

$$\text{Induction step: } T(k+1) = 1 + \sum_{i=0}^{k} T(i) = 1 + T(k) + \sum_{i=0}^{k-1} T(i) = 1 + T(k) + T(k) - 1 = 2T(k) = 2.2^k = 2^{k+1}$$

Rod cutting: DP: top-down with memoization

- Just modify the previous algorithm to save the result of each subproblem
- **Memoized**: It remembers the results computed previously
- You can define r global variable or as a local variable and pass it as an argument to each call of the function (check CLR for that)

```
r = [None, None, None, None, … ,None]
#r has n+1 elements
def cr-memoized(p, n)
    if r[n] >= 0:
        return r[n]
    If n==0:
        q = 0
    else q = negative infinity
    for i=1 to n:
        q = max (q, p[i]+ cr-memoized(p, n-i))
    r[n] = q
    return q
```

Rod cutting: DP: top-down with memoization

- **Runtime: $\Theta(n^2)$**
  - To solve a problem of size n the for loop iterates n times
  - Each problem is solved only once
  - **Amortized Analysis:**
    - Recursive call is done only once in each loop, amortized cost is O(1) for each iteration.
    - Each recursive call has a for loop that repeats n times
    - Therefore enough to count the number of iteration of the for loop for each problem:
    - $1+2+...+n = n(n+1)/2 = \Theta(n^2)$

Rod cutting: DP: bottom-up

LENGTH	PRICE
1	1
2	5
3	8
4	9

- subproblem j needs all subproblems smaller than j
- fill the table by filling the smaller indices first
- Runtime: $\Theta(n^2)$

Cut-rod-bottom-up (p, 4)

	r_0	r_1	r_2	r_3	r_4
r[i]	0	1	5	8	10
s[i]		1	2	3	2

```
def cut-rod-bottom-up(p, n)
    r = [None, None, None, None, … ,None]
    #r has n+1 elements
    r[0] = 0
    for j=1 to n
        q = negative infinity
        for i=1 to j: #i is the position of the first
cut
            q = max (q, p[i]+ r[j-i])
        r[j] = q
    return r[n]
```

Rod cutting: Reconstructing a Solution

- Return the optimal solution in addition to the value of the optimal solution
- Arrays s says where to cut the array (at what length)
- To find the complete solution, after printing s[n] we need to look at the position n-s[n] in array s

```
def extended-cut-rod-bottom-up(p, n)
     r = [None, None, None, None, … ,None]
     s = [None, None, None, None, … ,None]
     #r has n+1 elements
     r[0] = 0
     for j=1 to n
          q = negative infinity
          for i=1 to j:
               #q = max (q, p[i]+ r[j-i])
               If q < p[i] + r[j-i]
                    q = p[i] + r[j-i]
                    s[j] = i
          r[j] = q
     return r, s

print_solution(p, n)
     r,s = extended-cut-rod-bottom-up(p, n)
     while n > 0
          print s[n]
          n = n - s[n]
```

Rod cutting: Dynamic Programming: Summary

- Runtime: $\Theta(n^2)$
- Space: $\Theta(n)$

Recipe for writing a dynamic programming

1. Check to see if the problem has the optimal substructure property and identify the optimal substructure
2. Find a recursive formulation for solving the problem, i.e., the value of the optimal solution
3. Implement the recursive formulation using memoized top-down or bottom-up approach
    a. Which one is preferred?
        i. Top-down version is harder to analyze its runtime
        ii. Overhead of recursive approach takes more time
4. If needed, modify and extend the algorithm so that it returns the actual solution as well as the value of the solution