

# 11. Dynamic Programming Introduction

## CPSC 535

Kevin A. Wortman



This work is licensed under a Creative Commons Attribution 4.0 International License.

# Dynamic Programming

- ▶ pattern for designing algorithms
- ▶ *programming*:
  - ▶ optimize subject to constraints
  - ▶ (same as Linear Programming)
  - ▶ **not** writing programs
- ▶ *dynamic*: curious buzzword
- ▶ specialized tool
  - ▶ dynamic programming only applies to problems with *overlapping subproblems*
  - ▶ rare
  - ▶ huge speedup over naïve algorithms for such problems

## Big Ideas

- ▶ important algorithm design approach in its own right
- ▶ problem solving to view a problem in a different way
- ▶ **time-space trade-off**
  - ▶ speedup costs space
- ▶ **efficiency-complexity trade-off**
  - ▶ top-down, bottom-up variants
  - ▶ top-down is simpler to design and implement
  - ▶ bottom-up has faster constant factors

## Optimization, Value, Solution

- ▶ dynamic programming usually applies to **optimization** problems
  - ▶ correct output “minimizes” or “maximizes” something
- ▶ **value**: quality of the solution
  - ▶ quantity to minimize/maximize
- ▶ designing a dynamic programming algorithm to...
  - ▶ ...calculate optimal **value** is simpler
  - ▶ ...calculate optimal **solution** is more complicated
- ▶  $\therefore$  some examples and exercises only involves **values**
- ▶ algo's for **solutions** are more practical but difficult

## Example: Vertex Cover

*vertex cover problem*

**input:** an undirected graph  $G = (V, E)$

**output:** a vertex cover  $C$  of minimum size

- ▶ solution = a set of vertices  $C$
- ▶ value = size of  $C$
- ▶ optimal = minimize  $|C|$

*vertex cover value problem*

**input:** an undirected graph  $G = (V, E)$

**output:** the minimum size of a vertex cover of  $G$

- ▶ note: output data type is an integer, not a set

## Example: Bipartite Matching

### **bipartite maximum matching problem**

*input:* an undirected bipartite graph  $G = (V, E)$  with parts  $V = L \cup R$

*output:* a matching  $M \subseteq E$  where the number of matched vertices is maximum

- ▶ solution = a set of edges  $M$
- ▶ value = size of  $M$

### **bipartite maximum matching value problem**

*input:* an undirected bipartite graph  $G = (V, E)$  with parts  $V = L \cup R$

*output:* the maximum number of edges in a matching of  $G$

- ▶ note: output data type is an integer, not a set

# Ties

- ▶ we say **an** optimal solution
- ▶ not **the** optimal solution
- ▶ multiple solutions may have same value
- ▶ any of these are correct
- ▶ examples:
  - ▶ vertex cover: “a vertex cover  $C$  of minimum size”
  - ▶ bipartite matching: “a matching  $M \subseteq E$  where the number of matched vertices is maximum”
- ▶ not worrying about ties simplifies dynamic programming algorithms

## The Main Idea

- ▶ dynamic programming works on a problem where...
  - ▶ a solution has a **recursive structure**
  - ▶ so we *could* design a naïve divide-and-conquer algorithm
  - ▶ **but**, subproblems **overlap**
  - ▶ so divide-and-conquer would do the same work repeatedly
  - ▶ would be slow (often exponential time)
- ▶ idea: store subproblem solutions in a **table** (array or hash dictionary)
- ▶ only solve subproblems not already in table
- ▶  $\therefore$  each subproblem is solved **only once**
- ▶ fast polynomial time, often  $\Theta(n)$  or  $\Theta(n^2)$



## Top-Down versus Bottom Up

- ▶ two ways to write the pseudocode
- ▶ **top-down**
  - ▶ improvement to divide-and-conquer pseudocode
  - ▶ add a base case that checks for a solution in the table
  - ▶ simple to derive from divide-and-conquer algorithm
  - ▶ usually depends on a hash dictionary data structure, so expected time
- ▶ **bottom-up**
  - ▶ clean-sheet redesign
  - ▶ nested loops explicitly solve problems in sorted order
  - ▶ base case, larger subproblems, ..., full problem
  - ▶ store subproblems in array (not hash table)
  - ▶ no recursion or hash table  $\Rightarrow$  faster constant factors

## Design Process

1. Identify the problem's **solution** and **value**, and note which is our **goal**.
2. Derive a **recurrence** for an optimal value.
3. Design a divide-and-conquer algorithm that computes an **optimal value**.
4. Design a dynamic programming algorithm that computes an **optimal value**.
  - 4.1 **top-down** alternative: add table base case (**memoization**)
  - 4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

## Rod Cutting Problem

story:

- ▶ have a rod of metal  $n$  inches long
- ▶ can chop it into pieces of size  $1, 2, \dots, n$
- ▶ total length of all pieces  $= n$
- ▶ market price of a  $i$ -inch piece is  $p_i$
- ▶ market price of a 0-inch piece is 0
- ▶ goal: maximize total price of the pieces

*rod cutting value problem*

**input:** an array of non-negative prices  $P = \langle p_1, \dots, p_n \rangle$

**output:** the maximum total price that can be achieved by cutting an  $n$ -inch rod into pieces

## Example with $n = 4$

$i$	1	2	3	4
$p_i$	3	7	8	11

Ways of cutting  $\square\square\square\square$ :

1.  $\square\square\square\square : p_4 = \$11$
2.  $\square \mid \square\square\square : p_1 + p_3 = \$3 + \$8 = \$11$
3.  $\square\square \mid \square\square : p_2 + p_2 = \$7 + \$7 = \$14$
4.  $\square\square\square \mid \square : p_3 + p_1 = \$8 + \$3 = \$11$
5.  $\square \mid \square \mid \square\square : p_1 + p_1 + p_2 = \$3 + \$3 + \$7 = \$13$
6.  $\square \mid \square\square \mid \square : p_1 + p_2 + p_1 = \$3 + \$7 + \$3 = \$13$
7.  $\square\square \mid \square \mid \square : p_2 + p_1 + p_1 = \$7 + \$3 + \$3 = \$13$
8.  $\square \mid \square \mid \square \mid \square : p_1 + p_1 + p_1 + p_1 = \$3 + \$3 + \$3 + \$3 = \$12$

## Greedy Fails

- ▶ greedy heuristics are **not correct** for this problem
- ▶ note that there is no requirement that prices  $p_i$  obey “common sense” market dynamics
- ▶ e.g. it is allowed for  $p_4 > p_5$
- ▶ example of an **incorrect** greedy heuristic: find length  $i$  with highest unit price  $p_i/i$ , then make  $\lceil n/i \rceil$  pieces of length  $i$
- ▶ fails when the leftover  $n - \lceil n/i \rceil$  inches could be used better
- ▶ Recall: the designer of a greedy algorithm has the burden of **proving** their heuristic is correct
- ▶ **Tip:** if you are told to design a dynamic programming algorithm, don't waste time with greedy algorithms

## Rod Cutting Step 1

1. Identify the problem's **solution** and **value**, and note which is our **goal**.

*rod cutting value problem*

**input:** an array of non-negative prices  $P = \langle p_1, \dots, p_n \rangle$

**output:** the maximum total price that can be achieved by cutting an  $n$ -inch rod into pieces

- ▶ **solution:** list of piece lengths e.g.  $\langle 2, 2 \rangle$
- ▶ **value:** total price e.g. \$14
- ▶ **goal:** **value**

## Rod Cutting Step 2

2. Derive a **recurrence** for an optimal value.

- ▶ define  $r_i$  = the maximum total price starting from  $i$  inches
- ▶ base case:  $r_0 = 0$
- ▶ general case:
  - ▶ **think** divide-and-conquer; define  $r_i$  in terms of  $r_{<i}$
  - ▶ make the problem **one piece** smaller
  - ▶ try to make one cut, then recursively use the remaining inches
  - ▶ try every option and keep the optimal one

$$r_i = \max_{1 \leq j \leq n} (p_j + r_{n-j})$$

## Rod Cutting Step 3

3. Design a divide-and-conquer algorithm that computes an **optimal value**.

```
1: function CUT-ROD-DC(P, n)
2:   if n == 0 then
3:     return 0
4:   end if
5:    $q = -\infty$ 
6:   for i from 1 to n do
7:      $q = \max(q, P[i] + \text{CUT-ROD-DC}(P, n-i))$ 
8:   end for
9:   return q
10: end function
```



## Sidebar: Analysis of CUT-ROD-DC

- ▶ CUT-ROD-DC corresponds directly to the  $r_i$  definition
- ▶ **but** it is very slow
- ▶ fundamental problem: CUT-ROD-DC calls itself many times
  - ▶ each iteration of the **for** loop is a recursive call
  - ▶ each of those has a **for** loop with recursive calls. . .
- ▶ recall: fast divide-and-conquer algorithms usually call themselves 1–2 times
- ▶ *Claim:* The time complexity of CUT-ROD-DC is  $O(2^{n-1})$ .
- ▶ dynamic programming will circumvent all this recursion

## Rod Cutting Step 4.a

4. Design a dynamic programming algorithm that computes an **optimal value**.

4.1 **top-down** alternative: add table base case (**memoization**)

- ▶ **memoization**: use a hash dictionary to make a “memo” of pre-calculated solutions
- ▶ use  $i$  as key in table  $T$  (same API as hash tables in deck 4)
- ▶ after we compute an  $r_i$ , set  $r_i.key = i$  and insert  $r_i$  into  $T$
- ▶ if  $T$  does not contain key  $i$ , then we haven't computed  $r_i$  yet
- ▶ need two functions
  - ▶ public non-recursive function to create  $T$  and start recursion
  - ▶ private recursive function that expects  $T$  to exist

## Rod Cutting Step 4.a

```
1: function CUT-ROD-MEMOIZED( $P, n$ )
2:   HASH-TABLE-CREATE( $T$ )
3:   return CUT-ROD-MEMO-REC( $T, P, n$ )
4: end function
5: function CUT-ROD-MEMO-REC( $T, P, n$ )
6:    $q$  = HASH-TABLE-SEARCH( $T, n$ )
7:   if  $q \neq \text{NIL}$  then
8:     return  $q$ 
9:   end if
10:  if  $n == 0$  then
11:     $q = 0$ 
12:  else
13:     $q = -\infty$ 
14:    for  $i$  from 1 to  $n$  do
15:       $q = \max(q, P[i] + \text{CUT-ROD-MEMO-REC}(T, P, n - i))$ 
16:    end for
17:  end if
18:   $q.\text{key} = n$ 
19:  HASH-TABLE-INSERT( $q$ )
20:  return  $q$ 
21: end function
```

## Rod Cutting Step 4.b

4. Design a dynamic programming algorithm that computes an **optimal value**.
  - 4.1 **top-down** alternative: add table base case (**memoization**)
  - 4.2 **bottom-up** alternative: rewrite to use bottom-up loops instead of recursion
- ▶ observe: in CUT-ROD-MEMOIZED, keys are inserted into  $T$  in order  $0, 1, \dots, n$
- ▶ **bottom-up**: write an explicit **for** loop that computes and stores every general case  $r_i$  in order  $r_1, \dots, r_n$
- ▶ base case is computed and stored before the loop
- ▶ convenient to use an array instead of hash table
- ▶ define  $R[i] = r_i$
- ▶ no more recursion, just loops

## Rod Cutting Step 4.b

```
1: function CUT-ROD-BU( $P[1..n]$ )
2:   Create array  $R[0..n]$ 
3:    $R[0] = 0$ 
4:   for  $j$  from 1 to  $n$  do
5:      $q = -\infty$ 
6:     for  $i$  from 1 to  $j$  do
7:        $q = \max(q, P[i] + R[j - i])$ 
8:     end for
9:      $R[j] = q$ 
10:  end for
11:  return  $R[n]$ 
12: end function
```

## Bottom-Up Analysis

- ▶ CUT-ROD-BU is clearly  $\Theta(n^2)$  time
- ▶ (Note: easy analysis)

## Top-Down Analysis

- ▶ trickier analysis
- ▶ observe: hash **if** statement guarantees that each subproblem is solved exactly once
- ▶ solving subproblem  $i$ , not counting recursion:  $\Theta(i)$  time due to **for** loop
- ▶ total of all subproblems is  $\sum_{i=1}^n i \in \Theta(n^2)$
- ▶ hash operations add “expected” qualifier
- ▶  $\therefore$  CUT-ROD-MEMOIZED takes  $\Theta(n^2)$  expected time
- ▶ with effort we could replace the hash table with an array for  $\Theta(n^2)$  worst-case time
- ▶ CUT-ROD-MEMOIZED has worse constant factors due to the overhead of recursive function calls

## Trade-Offs

Factor	Naïve	TDDP	BUDP
Ease of design	easiest	difficult	very difficult
Ease of analysis	medium	difficult	easy
Time efficiency	$O(2^{n-1})$	$\Theta(n^2)$ exp.	$\Theta(n^2)$ w/ fast const.
Space overhead	n/a	$O(n)$ hashtable	$O(n)$ array

- ▶ according to principles, **bottom up dynamic programming** is superior
- ▶ but top-down dynamic programming is a close second



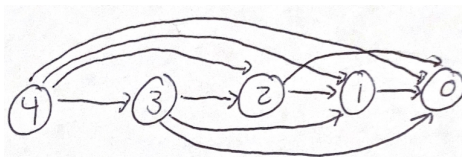
## Subproblem Graphs

- ▶ solutions have a recursive structure

$$r_i = \max_{1 \leq i \leq n} (p_i + r_{n-i})$$

- ▶ a general-case solution depends on other solution(s)
- ▶ algorithm must compute solutions in an order that satisfies dependencies
- ▶ memoization automates this, with overhead
- ▶ bottom-up loops must be designed carefully to iterate in satisfactory order
- ▶ visualize dependencies in a **subproblem graph**

## Subproblem Graphs



- ▶ vertex  $i$  = subproblem  $i$
- ▶ directed edge  $(i, j)$  = computing  $i$  requires solution to  $j$
- ▶ subproblem  $i$  must wait until all outgoing neighbors have been computed
- ▶ top-down manages with hashtable
- ▶ bottom up manages with loop iteration order

## Rod Cutting Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

*rod cutting value problem*

**input:** an array of non-negative prices  $P = \langle p_1, \dots, p_n \rangle$

**output:** the maximum total price that can be achieved by cutting an  $n$ -inch rod into pieces

*rod cutting problem*

**input:** an array of non-negative prices  $P = \langle p_1, \dots, p_n \rangle$

**output:** the list of cut-lengths of maximum total price for an  $n$ -inch rod

## Storing All Subproblem Solutions Is Expensive

- ▶ solution to rod cutting problem: a list of cut-lengths;  $O(n)$  space each
- ▶ our algorithms compute  $n + 1$  solutions
- ▶ storing all subproblem solutions would takes  $O((n + 1) \times n) = O(n^2)$  space, **expensive**
- ▶ instead, store only  $O(n)$  information

## Backtracking

- ▶ algorithm computes optimal value, and **logs (records) how it made each decision**
- ▶ after all optimal values have been computed, follow a “trail” to create solution object
- ▶ trail ends at the optimal solution
- ▶ each log entry says how to go one step backwards
- ▶ follow them until we get to the start (a base case)
- ▶ traverses solution in backwards order; reverse it if order matters
- ▶ backtracking is usually only  $O(n)$  time, and  $O(n)$  space overhead

## Rod Cutting Step 5

5. (if goal is a solution algo.) Design a dynamic programming algorithm that computes an **optimal solution**.

- ▶ bottom-up algo. makes optimal choices with

$$q = \max(q, P[i] + R[j - i])$$

step

- ▶ i.e. it chooses how many inches to cut right now
- ▶ **log** these choices in another array
- ▶ recall  $R[j]$  = maximum total price starting from  $j$  inches
- ▶ define  $S[j]$  = size of the first optimal cut starting from  $j$  inches
- ▶ need to update pseudocode to
  - ▶ create  $S$
  - ▶ update  $S$  inside the loops
  - ▶ at the end, backtrack  $S$  to compute a list of lengths

## Rod Cutting Step 5 – Pseudocode

```
1: function CUT-ROD-SOLUTION( $P[1..n]$ )
2:   Create arrays  $R[0..n]$  and  $S[0..n]$ 
3:    $R[0] = 0$ 
4:   for  $j$  from 1 to  $n$  do
5:      $q = -\infty$ 
6:     for  $i$  from 1 to  $j$  do
7:       if  $q < (P[i] + R[j - i])$  then
8:          $q = P[i] + R[j - i]$ 
9:          $S[j] = i$ 
10:      end if
11:    end for
12:     $R[j] = q$ 
13:  end for
14:  soln = empty list
15:   $j = n$ 
16:  while  $j > 0$  do
17:    soln.add( $S[j]$ )
18:     $j = j - S[j]$ 
19:  end while
20:  reverse soln
21:  return soln
22: end function
```

## Analysis

- ▶ CUT-ROD-SOLUTION solves the *rod cutting problem*
  - ▶ it returns a list of cut-lengths, not a price
- ▶ analysis is actually straightforward
- ▶ time efficiency:
  - ▶ nested **for** loops:  $\Theta(n^2)$
  - ▶ backtracking: **while** loop iterates at most  $n$  times  $\Rightarrow \Theta(n)$  time
  - ▶ reverse soln:  $\Theta(n)$
  - ▶ total  $\Theta(n^2 + n + n) = \Theta(n^2)$  time
- ▶ space efficiency:  $R$  and  $S$  take  $\Theta(n + n) = \Theta(n)$  space
- ▶ (same as the step-4 algorithms)