19. Dynamic Programming

CPSC 535

Kevin A. Wortman

CALIFORNIA STATE UNIVERSITY FULLERTON

Dynamic Programming

- **Dynamic programming:** pattern for solving problems with a divide-and-conquer structure *and overlapping subproblems*
- Note: "programming" does not refer to coding
- Similar to divide-and-conquer
  - Recall: merge sort, closest pair
- Only applies to a narrow category of problems
- **But** offers huge speed-ups in those rare cases
  - often exponential $\longrightarrow$ fast polynomial

Designing Dynamic Programming Algorithms

Suggested process to design a dynamic programming algorithm:

1. Characterize the structure of an optimal solution (i.e. data type)
2. Recursively define the value of an optimal solution (like divide-and-conquer)
3. Design pseudocode that computes the **value** of an optimal solution
   - Either *bottom-up*, or with *memoization*
4. (If desired, next class) Design pseudocode that constructs an optimal solution based on information computed in step 3.

Rod Cutting Problem

rod cutting problem
input: integer rod length $n \geq 0$ and a table of prices $p_1, \ldots, p_n$
output: maximum revenue obtainable by cutting the rod into pieces of length $\leq n$

(Above computes a *value* of a solution, to compute an actual *solution* change the **output** to:)

output: a list of rod-lengths in $[1, n]$ that add up to exactly n and maximize revenue

Greedy Doesn't Work

- Tempting to try a greedy heuristic
  - e.g. pick the length with the best unit price p_i/i
- **But** greedy algorithms for this problem are **not correct**
- Problem definition makes **no guarantee** that
  - prices p_i obey common-sense properties
  - e.g. larger pieces are more valuable than smaller ones
  - e.g. buying in bulk is a better deal
  - e.g. small scraps like p_1 are nearly worthless
- In general, problems that benefit from dynamic programming cannot be solved correctly by greedy methods
- If you design a greedy alg., onus is on you to prove correctness
- *Tip:* if a problem is framed as dynamic programming, don't even bother with greedy approaches

Baseline: Exhaustive Search

- Baseline idea: try every way of dividing length n into smaller pieces
- $\approx 2^n$ candidates
- $O(2^n)$ time
- extremely slow

1. Characterize Solution

- recall: p_i = price of a rod of length i
- a solution is a sequence of rod-lengths $S = \langle i_1, i_2, \ldots, i_k \rangle$ such that $\sum_j i_j = n$ and the total revenue

  $$\sum_j p_{i_j}$$

  is maximized
- define

  $r_i =$ the **maximum revenue obtainable** from a rod of length i
- the **optimal value** is r_n

2. Recursive Definition of an Optimal Solution

- Each piece must have length ≥ 1, so each $i_j \in [1, n]$
- Given original length n, we could make one cut to form: $1 + (n-1), 2 + (n-2), 3 + (n-3), \ldots, n + 0$ (last entry means to keep the rod whole; no cut)
- These produce revenue $p_1 + r_{n-1}, p_2 + r_{n-2}, p_3 + r_{n-3}, \ldots, p_n + r_0$
- Goal is to **maximize** revenue
- So $r_n = \max(p_1 + r_{n-1}, p_2 + r_{n-2}, p_3 + r_{n-3}, \ldots, p_n + r_0)$ i.e.

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

Aside: Recursive Top-Down Algorithm

```
function CUT-ROD(p[0..n], n)
    if n = 0 then
        return 0
    end if
    q = −∞
    for i = 1 to n do
        q = max(q, p[i] + CUT-ROD(p, n − i))
    end for
    return q
end function
```

Overlapping subproblems: ex. CUT-ROD$(p, 2)$ will be computed many times

Slow; $O(2^n)$ time

Enter Dynamic Programming

- Problem: top-down recursion recomputes the same values over and over
- Solution: use a **table** (array) to cache these solutions
- For each x, evaluate

$$\text{CUT-ROD}(p, x)$$

  only once
- Time/space trade-off
  - table takes extra space
  - saves a **lot** of time
  - exponential $\longrightarrow$ polynomial

Memoization and Bottom-Up

- Two fine approaches for caching subsolutions
- **Memoization:** use an array or hash table T, where

  $T[i] =$ solution for input i, or dummy value if undefined
- CUT-ROD is still recursive, but has a base case to reuse $T[i]$ instead of evaluating the function body
- **Bottom-Up:** solve subproblems from smallest to largest
- Initialize $T[0], T[1], \ldots, T[n]$ in an interative loop
- Mostly a matter of preference
- Some programming languages support automatic memoization (ex. Racket)

3. Pseudocode for **Memoized** Dynamic Programming Alg.

```
function MEMOIZED-CUT-ROD(p[0..n], n)
    Create empty hash map H
    return CUT-ROD-REC(p, n, H)
end function
function CUT-ROD-REC(p[0..n], n, H)
    if H.CONTAINS-KEY(n) then
        return H[n]
    end if
    if n = 0 then
        return 0
    end if
    q = −∞
    for i = 1 to n do
        q = max(q, p[i] + CUT-ROD-REC(p, n − i, H))
    end for
    H[n] = q
    return q
end function
```

3. Pseudocode for **Bottom-Up** Dyn. Prog. Alg.

```
function BOTTOM-UP-CUT-ROD(p[0..n], n)
    Create new array r[0..n]
    r[0] = 0                                   ▷ base case
    for j = 1 to n do                          ▷ general cases bottom-up
        q = −∞
        for i = 1 to j do    ▷ only references initialized elements
            q = max(q, p[i] + r[j − 1])
        end for
        r[j] = q
    end for
    return r[n]
end function
```

Analysis

- BOTTOM-UP-CUT-ROD: clearly $\Theta(n^2)$ b/c nested for loops
- MEMOIZED-CUT-ROD:
  - Less clear, but $\Theta(n^2)$ expected time
  - Cache hit (H contains n): $\Theta(1)$ expected time
  - Cache miss: $\Theta(n)$ expected time due to for loop
  - Observe: each miss happens *at most once*
  - Total time at most

$$\sum_{i=0}^{n} \Theta(i) \in \Theta(n^2)$$

- Same $\Theta(n^2)$ efficiency; memoized version is expected
- **Huge** speedup: $O(2^n) \longrightarrow \Theta(n^2)$
- Modest $\Theta(n)$ space complexity for table

What's Next

- Computing a solution (list of rod-lengths)
- Maximizing over two indices
- Longest common subsequence problem