CS472 Module 5 Part A - Dynamic Programming - Overview

Athens State University

Outline

Contents

1 What kind of programming is "dynamic programming"?

Divide and Conquer

- Top-down approach to problem solving
- Blindly divide problem into smaller instances and solve the smaller instances
- Best used with problems where the smaller instances are unrelated
  - Leads to inefficient solution to problems where smaller instances are related
  - Compare and contrast the recursive and iterative algorithms for computing the Fibonacci sequence

Dynamic Programming?

- *Dynamic programming*: a meta-heuristic for solving a given problem by breaking into component parts, solving and remembering the solution to those component parts, and combining the solutions to the component parts into a solution for the original problem
  - An archaic use of the word "programming", more in the sense of "planning"
  - *Memoization*: store the results of an algorithm for a specific set of inputs so that we do not need repeat the computation in future calls

Dynamic Programming: A Thought Question

Is the "divide and conquer" meta-heuristic the same as the "dynamic programming" meta-heuristic?

Contrast to Divide and Conquer

- Bottom-up approach to problem solving
- Instances of problem are divided into smaller instances
- Smaller instances are solved first and stored for later use by solution to solve larger instances
- Table-driven approach: Look it up instead of re-compute

Dynamic Programming: Key Problem Attributes

- *Optimal substructure*: solution to a problem can be obtained by combining together optimal solutions to its sub-problems
  - First step in applying this meta-heuristic is to check whether a problem exhibits this behavior
  - Such optimal substructures are usually described by means of recursion (yes... recursion)
  - Example: given a graph $G = (V, E)$, the shortest path p from a vertex u to vertex v exhibits optimal substructure
    - Pick an intermediate vertex w on the shortest path p.
    - Then we can split p into a path p_1 from u to w and path p_2 from w to v
    - The paths p_1 and p_2 are the shortest paths between the corresponding vertices

Dynamic Programming: Key Problem Attributes

- *Overlapping sub-problems*: the space of sub-problems must be small; that is any recursive algorithm solving the problem should solve the same sub-problems over and over rather than generating new subproblems
  - Example: Consider the recurrence relation for generating the Fibonacci sequence:

$$F_i = F_{i-1} + F_{i-2}$$

    with base case of $F_1 = F_2 = 1$
    - Note that $F_{43} = F_{42} + F_{41}$ and $F_{42} = F_{41} + F_{40}$
    - Note how F_{41} is involved in the computation of both F_{43} and F_{42}
    - But the naive recursive algorithm computes these quantities over and over*

Dynamic Programming: General approaches

- *Top-down approach*
  - Direct consequence of the recursive formulation of a problem
  - If we can express a problem in a recurrence relation, and the sub-problems in the relation overlap, then we can easily store the solutions to sub-problems in a table
  - When we attempt to solve a sub-problem, we check the table
    - If the value exists, then use that value
    - Otherwise, solve the sub-problem and add the result to the table

Dynamic Programming: General approaches

- *Bottom-up approach*
  - Try solving the sub-problems first and store the results
  - Use the solutions in the table to build solutions to the bigger sub-problems

Dynamic Programming: Top-down Example

- OK... the recursive algorithm for Fibonacci sequence is doing lots of extra work
- Let's do some "memoization": Suppose we keep track of the intermediate values already computed

---

Algorithm 1: Computing Fibonacci numbers: a DP algorithm: top-down

---

```
Input: A positive integer n
Output: The n-th Fibonacci number
map(0) ← 0;
map(1) ← 1;
if n is not in map then
    map(n) ← fib(n-1) + fib(n-2);
return map(n);
```

Dynamic Programming: Bottom-up Example

Algorithm 2: Computing Fibonacci numbers: a DP algorithm: bottom-up

---

```
Input: A positive integer n
Output: The n-th Fibonacci number
if n = 0 then
    Return 0;
else
    previous ← 0;
    current ← 1;
    repeat
        new ← previous + current;
        previous ← current;
        current ← new;
    until n-1;
return current;
```

---

2 Example: Coins in a row

Example: Coins in a row

You have a row of n coins that have positive values c_1, c_2, ..., c_n where the values may not be distinct. What is best selection of coins to pick up such that you get most money while not picking up two adjacent coins?

Let $F(n)$ be the maximum amount that one can pick up from the row of n coins. We can divide the coins into two groups:

- Those without the last coin - the max amount is ?
- Those with the last coin - the max amount is ?

Example: Coins in a row

Thus, we have the recurrence relation:

$$F(0) = 0$$
$$F(1) = c_1$$
$$F(n) = \max(c_n + F(n-2), F(n-1)), n > 1$$

Example: Coins in a row

$$F(n) = \max(c_n + F(n-2), F(n-1)), n > 1$$

	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F(1), F(2)	0	5					
F(3)	0	5	5				
F(4)	0	5	5	7			
F(5)	0	5	5	7	15		
F(6)	0	5	5	7	15	15	
F(7)	0	5	5	7	15	15	17

3 Example: Change-making problem

Example: Change-making problem

- What coins should be returned for amount n if the denominations are $d_1 < d_2 < d_3 < \ldots < d_m$? Assume that we have an unlimited quantity of each denomination of coin available for distribution.
- Let $F(n)$ be the minimum number of coins whose values add up to n and assume that $F(0) = 0$. The amount n can be only be obtained by adding one coin of denomination d_j to amount $n - d_j$ for $j = 1, 2, \ldots, m$ such that $n \geq d_j$. As the penny (i.e., 1) is constant, we can find the smallest $F(n - d_j)$ first and add 1 to it.

Example: Change-making problem

Thus, we have the recurrence relation

$$F(n) = \min_{j:n \geq d_j} (F(n - d_j) + 1), n > 0$$
$$F(0) = 0$$

Example: Change-making problem

Algorithm 3: Use dynamic programming to make change

Input: Positive integer n and array D of increasing positive integers indicating coin denominations, with D[1]=1

Output: The minimum number of coins that equal to n

```
F[0] ← 0;
for i ← [1..n] do
    temp ← ∞;
    while j ≤ m AND i ≥ D[j] do
        time ← min F[i − D[j]], temp;
        j ← j+1;
    F[i] ← temp + 1;
return F[n];
```

4 Key Points

Key Points

- What is dynamic programming?
- Relationship between recursion and dynamic programming
  - Optimal substructure
  - Overlapping sub-problems
- Approaches
  - Top-down
  - Bottom-up