csce350 — Data Structures and Algorithms
Fall 2019 — Lecture Notes: Dynamic programming

This document contains slides from the lecture, formatted to be suitable for printing or individual reading, and with occasional supplemental explanations added. It is intended as a supplement to, rather than a replacement for, the lectures themselves — you should not expect the notes to be self-contained or complete on their own.

1 Introduction

8

Dynamic programming is a technique for solving problems with overlapping subproblems.

Key idea: Express the solution use a recurrence the connects the solution to the solutions of several subproblems.

"recursion without repetition"

2 Example: Fibonacci numbers

You have likely seen the **Fibonacci sequence** somewhere:

$$F(0) = 1$$
$$F(1) = 1$$
$$F(k) = F(k-1) + F(k-2)$$

Given given n, how can we compute $F(n)$?

3 Alternative: Fill in an array

Instead of the obvious recursive approach, we might fill in an array instead.

4 The process

1. Identify a **family of subproblems**, in which the final answer is a special case.
2. **Write a recurrence** showing how the solutions to those subproblems are related to each other.
3. Find **base cases** for the recurrence.
4. Form the algorithm.
   (a) Fill in a **table of recurrence values** in an order that obeys the dependencies.
   (b) Extract the **final solution** from the table.

5 Binomial coefficients

8.1

Suppose we want to compute **binomial coefficients**:

$$C(n,k) = \binom{n}{k}$$

Identities:

$$C(n,0) = 1$$
$$C(n,n) = 1$$
$$C(n,k) = C(n-1,k-1) + C(n-1,k)$$

6 Binomial coefficients: Table view

	0	1	2	$\cdots$	$k-1$	k
0	1					
1	1	1				
2	1	2	1			
3	1	3	3	1		
$\vdots$						
$n-1$					$C(n-1,k-1)$	$C(n-1,k)$
n						$C(n,k)$

7 Binomial coefficients: Algorithm

```
BINOMIALCOEF(n, k)
  for i ← 0, ..., n do
    for j ← 0, ..., min(i, k) do
      if j = 0 or j = i then
        C[i, j] ← 0
      else
        C[i, j] ← C[i − 1, j − 1] + C[i − 1, j]
      end if
    end for
  end for
  return C[n, k]
```

8 Knapsack problem revisited

8.2

Remember the **knapsack problem** from Chapter 3? **Input:**

- n items, each with a **weight** and a **value**.

$$\begin{array}{cccc} w_1 & w_2 & \cdots & w_n \\ v_1 & v_2 & \cdots & v_n \end{array}$$

- A knapsack with **capacity** W.

Output: A list of items to take that maximizes total value within the capacity constraint.

9 Step 1: Identify a family of subproblems

Define

$$V[i,j]$$

to be the best value that can be achieved for the instance with only the first i items and capacity j.

Note that the final answer will be $V[n,W]$.

10 Step 2: Write a recurrence

For a given i and j, how can we find $V[i, j]$?

Question: Do we take item i?

- If **yes**, then $V[i, j] = V[i - 1, j - w_i] + v_i$.
- If **no**, then $V[i, j] = V[i - 1, j]$.

11 Write a recurrence

$$V[i, j] = \begin{cases} \max(V[i-1, j-w_i] + v_i, V[i-1, j]) & \text{if } j \geq w_i \\ V[i-1, j] & \text{if } j < w_i \end{cases}$$

12 Step 3: Base cases

No capacity available:

$$V[i, 0] = 0$$

No items available:

$$V[0, j] = 0$$

13 Step 4: Form the algorithm

```
KNAPSACKDP(w_1, ..., w_n, v_1, ..., v_n, W)
  for i ← 0, ..., n do
    for j ← 0, ..., W do
      if i = 0 or j = 0 then
        V[i, j] ← 0
      else
        if j < w_i then
          V[i, j] ← V[i − 1, j]
        else
          V[i, j] ← max(V[i − 1, j], V[i − 1, j − w_i] + v_i)
        end if
      end if
    end for
  end for
  return V[n, W]
```

14 Knapsack DP example

Input:

	1	2	3	4	
w	2	1	3	2	$W = 5$
v	12	10	20	15	

Table:

V	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

15 Top-down dynamic programming

So far, we've looked at **bottom-up dynamic programming**, which starts with the base cases.

Alternative: Top-down dynamic programming ("memoization") solves only the subproblems that contribute to the final answer.

- Write a recursive algorithm directly from the Step 2 recurrence.
- Record each subproblem solution in the table.
- Before doing any real computation, check the table to see if that answer is already available.

16 Top-down Knapsack

Initialize: $V[i, j] = -1$ for all i and j

```
KnTD(i, j)
  if V[i, j] = −1 then
    if i = 0 or j = 0 then
      V[i, j] ← 0
    else
      if j < w_i then
        V[i, j] ← KnTD(i − 1, j)
      else
        V[i, j] ← max(KnTD(i − 1, j), KnTD(i − 1, j − w_i) + v_i)
      end if
    end if
  end if
  return V[i, j]
```

17 Knapsack DP top-down example

Input:

	1	2	3	4
w	2	1	3	2
v	12	10	20	15

$W = 5$

Table:

V	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	–	12	22	–	22
3	–	–	–	22	–	32
4	–	–	–	–	–	37

18 Top down versus Bottom up

Top-down advantages:

- Only compute values that are relevant to the final answer.
- No need to worry about the ordering.

Bottom-up advantages:

- No overhead for function calls.
- No overhead for 'Computed yet?' checks.
- Can optimize space by discarding entries that are no longer needed.