CS161 - Dynamic Programming

David Kauchak

- Dynamic programming is a method for solving problems where the optimal solution can be defined in terms of optimal solutions to sub-problems.
- Fibonacci numbers

  $1, 1, 2, 3, 5, 8, 13, 21, 34, ...$

  What is the n^{th} Fibonacci number?

  $F(n) = F(n-1) + F(n-2)$

  The solution to the answer for n is defined with respect to the solution for $n-1$ and $n-2$. Top-down approach:

  FIBONACCI(n)
  1 **if** $n = 1$ or $n = 2$
  2 **return** 1
  3 **else**
  4 **return** FIBONACCI($n-1$) + FIBONACCI($n-2$)

  - Is it correct?
  - Runtime? $O(2^n)$
  - Memory?

    Each call uses $\Theta(1)$ space. However, because we must recurse down to $n = 1$ and we keep track of this on the stack, we will still use $\Theta(n)$ space.

  Can we do any better?

Let's look at the call sequences:

FIBONACCI(n)
  FIBONACCI($n-1$)
    FIBONACCI($n-2$)
      FIBONACCI($n-3$)
      FIBONACCI($n-4$)
    FIBONACCI($n-3$)
      FIBONACCI($n-4$)
      FIBONACCI($n-5$)
  FIBONACCI($n-2$)
    FIBONACCI($n-3$)
      FIBONACCI($n-4$)
      FIBONACCI($n-5$)
    FIBONACCI($n-4$)
      FIBONACCI($n-5$)
      FIBONACCI($n-6$)

Many of calls are redundant, since we already know the answer. Two main ideas for dynamic programming:

1. Identify a solution to the problem with respect to solutions of supbroblems
2. Bottom-up: Start with solutions to the smallest subproblems and build solutions to larger problems

FIBONACCI-DP(n)
```
1  fib[1] ← 1
2  fib[2] ← 1
3  for i ← 3 to n
4          fib[i] ← fib[i − 1] + fib[i − 2]
5  return fib[n]
```

– Is it correct?

- Runtime? - $O(n)$
- Memory - $O(n)$

- Binary search trees

  How many unique binary search trees can be created using the numbers 1 through n?

  Step 1: What is the subproblem?

  Consider each of the 1 to n numbers as the root of the tree. For each configuration, how many possibilities are there? If we sum up all of these possibilities then we'll have our answer.

  Let's look at a simple example. Take $n = 10$ and consider 4 as the root. How many binary search trees can be created?

  On the left, will be the numbers $(1, 2, 3)$ and on the right $(5, 6, 7, 8, 9, 10)$. Let N_L be the number of possible trees in the left tree and N_R the number of possible trees in the right tree, these could be calculated with calls to our function with values 3 and 6 respectively. Given N_L and N_R how many trees have 4 as the root?

  $$N_L * N_R$$

  every combination of tree in the left child with every combination on the right child.

  We can generalize this and sum over all roots

  BST-COUNT(n)

```
1  if n = 0
2          return 1
3  else
4          sum = 0
5          for i ← 1 to n
6                  sum ← sum + BST-COUNT(i − 1) * BST-COUNT(n − i)
7  return sum
```

- Is it correct?
  Tries all roots 1 to n and sums up the values.
- Runtime?

Step 2: Generate solution from the bottom-up

As with FIBONACCI we can start at the bottom and work our way up

BST-COUNT-DP(n)

```
1  c[0] = 1
2  c[1] = 1
3  for k ← 2 to n
4          c[k] ← 0
5          for i ← 1 to k
6                  c[k] ← c[k] + c[i − 1] ∗ c[k − i]
7  return c[n]
```

- Is it correct?
- Runtime?
  $O(n^2)$

- Longest common sequence

  Given a sequence $X = x_1, x_2, x_3, ..., x_n$ a *subsequence* is defined by a set of increasing indices $(i_1, i_2, ..., i_k)$ where $1 \leq i_1 < i_2 < .. < i_k \leq n$ which define a subset of the sequence X, $x_{i_1}, x_{i_2}, x_{i_3}, ..., x_k$.

  For example, given $X = ABACDABAB$, the following are all subsequences:

$$ABA$$
$$ADB$$
$$BCDB$$
$$ABAABAB$$

  Given two sequences X and Y a *common subsequence* is a subsequence that occurs both in X and Y.

Given two sequences $X = x_1, x_2, ..., x_n$ and $Y = y_1, y_2, .., y_m$, what is the length of the longest common subsequence?

Step 1: Define the problem with respect to subproblems

$$LCS(X,Y) = \begin{cases} 0 & \text{if } n = 0 \text{ or } m = 0 \\ 1 + LCS(X_{1...n-1}, Y_{1...m-1}) & \text{if } x_n = y_m \\ \max(LCS(X_{1...n-1}, Y), LCS(X, Y_{1...m-1}) & \text{otherwise} \end{cases}$$

The optimal LCS falls into one of two cases. If the last characters are the same, then the last character is part of the LCS and the entire LCS is that character plus the LCS of the remaining characters of the two sequences. If the last characters are not the same, then the LCS is either the LCS of X with the last character of Y removed or the LCS of Y with the last character of X removed.

Proof?

Step 2: Build the solution from the bottom-up

LCS-LENGTH(X, Y)

```
 1  m ← length[X]
 2  n ← length[Y]
 3  c[0,0] ← 0
 4  for i ← 1 to m
 5          c[i,0] ← 0
 6  for j ← 1 to n
 7          c[0,j] ← 0
 8  for i ← 1 to m
 9          for j ← 1 to n
10                  if x_i = y_i
11                          c[i,j] ← 1 + c[i-1,j-1]
12                  elseif c[i-1,j] > c[i,j-1]
13                          c[i,j] ← c[i-1,j]
14                  else
15                          c[i,j] ← c[i,j-1]
16  return c[m,n]
```

An example $X = ABCBDAB$ and $Y = BDCABA$

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0	0	0	1	1	1
2	B	0	1	1	1	1	2	2
3	C	0	1	1	2	2	2	2
4	B	0	1	1	2	2	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

(adapted from Figure 15.6, pg. 354 of [1])

  - Is it correct?
  - Runtime - We fill in an m by n matrix where each entry takes $\Theta(1)$ time - $\Theta(mn)$

- Keeping track of the solution

  So far, we have only asked the question of how many, but we may also want to know what the actual solution is. Generally, this means keeping track each decision that was made to fill in a particular entry.

  In the LCS problem, this is one of three options. Then, given the final solution, we can trace back the actual solution.

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	0u	0u	0u	1d	1l	1d
2	B	0	1d	1l	1l	1u	2d	2l
3	C	0	1u	1u	2d	2l	2u	2u
4	B	0	1d	1u	2u	2u	3d	3l
5	D	0	1u	2d	2u	2u	3u	3u
6	A	0	1u	2u	2u	3d	3u	4d
7	B	0	1d	2u	2u	3u	4d	4u

(adapted from Figure 15.6, pg. 354 of [1])

- Longest increasing subsequence

```
LIS(X)
 1  n ← LENGTH(X)
 2  create array lis with n entries
 3  for i ← n to 1
 4          max ← 1
 5          for j ← i + 1 to n
 6                  if X[j] > X[i]
 7                          if 1 + lis[j] > max
 8                                  max ← 1 + lis[j]
 9          lis[i] ← max
10  max ← 0
11  for i ← 1 to n
12          if lis[i] > max
13                  max ← lis[i]
14  return max
```

- Memoization

```
FIBONACCI-MEMOIZED(n)
1  fib[1] ← 1
2  fib[2] ← 1
3  for i ← 3 to n
4          fib[i] ← ∞
5  return FIB-LOOKUP(n)
```

```
FIB-LOOKUP(n)
1  if fib[n] < ∞
2          return fib[n]
3  x ← FIB-LOOKUP(n − 1) + FIB-LOOKUP(n − 2)
4  if x < fib[n]
5          fib[n] ← x
6  return fib[n]
```

- Other dynamic programming problems
  - 0-1 knapsack problem
  - matrix multiplication bracketing
  - edit distance between two sequence (LCS with substitution)

– longest increasing subsequence

– ...

These notes are adapted from material found in chapter 15 of [1].

References

[1] Thomas H. Cormen, Charles E. Leiserson Ronald L. Rivest and Clifford Stein. 2007. Introduction to Algorithms, 2nd ed. MIT Press.