2 Day 2: *Dynamic Programming*

2.1 Computing Fibonacci Numbers

The Fibonacci numbers are 1, 1, 2, 3, 5, 8, 13, 21,... where you get the next number in the sequence by adding the previous two. Mathematically, $F_0 = F_1 = 1$, and for $n \geq 2$, the n^{th} Fibonacci number F_n satisfies $F_n = F_{n-1} + F_{n-2}$.

What do you think of these two functions for computing the n^{th} Fibonacci number? What are some good things or bad things about them?

Algorithm 5 Recursive algorithm for computing the n^{th} Fibonacci number

```
function FIBREC(n)                          ▷ Returns the n^th Fibonacci number
    if n = 0 or n = 1 then return 1
    elsereturn FIBREC(n-1) + FIBREC(n-2)
end function
```

Algorithm 6 Bottom-up algorithm for computing the n^{th} Fibonacci number

```
function FIBBU(n)                           ▷ Returns the n^th Fibonacci number
    Create an array FibNumbers of length n+1
    Set FibNumbers[0] = 1
    Set FibNumbers[1] = 1
    for i = 2 to n do
        Set FibNumbers[i] = FibNumbers[i-1] + FibNumbers[i-2]
    return FibNumbers[n]
end function
```

The first algorithm does a lot of extra work: it solves the same subproblems many times. The second algorithm only solves each subproblem once. This is an example of *dynamic programming*.

Of course, for Fibonacci numbers we can come up with another solution that uses much less space:

Algorithm 7 Space-efficient algorithm for computing the n^{th} Fibonacci number

```
function FIBSPACE(n)                          ▷ Returns the n^th Fibonacci number
    if n = 0 or n = 1 then return 1
    prev = 1
    current = 1
    for i = 2 to n do
        next = prev + current
        prev = current
        current = next
    return current
end function
```

Bonus exercise: How can you compute the n^{th} Fibonacci number even faster than this?

2.2 Dynamic Programming

Dynamic programming is a general approach to solving problems when there's lots of overlapping subproblems. Problems for which dynamic programming gives good solutions are extremely broad and varied, and the best way to learn about dynamic programming is to try to solve problems!

General Principles:

1. Formulate the problem recursively. Can you write the solution to your problem in terms of solutions to subproblems? What do these subproblems look like, and what are your base cases (smallest subproblems)?
2. Give a dynamic programming algorithm. What order do you want to solve the subproblems in? How do you keep track of intermediate results?

Note: Dynamic programming can often work when greedy algorithms don't!

2.3 Longest Increasing Subsequence

Given a list A of n integers, we want to find the longest increasing subsequence, that is, a sequence of indices $1 \leq i_1 < i_2 < \ldots < i_k \leq n$ such that $A[i_1] < A[i_2] < \ldots \leq A[i_k]$.

Example: What's the longest increasing subsequence of:

$$5, 3, 8, 11, 4, 6, 9, 8, 14, 12$$

$$5, 2, 0, 14, 32, 56, 34, 5, 89, 19, 3, 20, 31, 28, 41$$

How might you write code to solve this problem? (always try the easy solutions first!)

- Look at all subseqences and see if they're increasing: Very slow!
- Greedily add entries to your subsequence if they're larger than the previous entry in the subseqeunce: Can give wrong answer!
- Dynamic Programming

List some ways you might break this problem up into subproblems.

How you might use answers to subproblems to answer the original problem?

The right subproblem to consider here is $L[i] =$ the length of the longest increasing subsequence that has $A[i]$ as it's last element. Using this, how might you write the solution of one problem in terms of its subproblems?

Consider the following dynamic programming algorithm.

Algorithm 8 Dynamic Program for Longest Increasing Subsequence

```
function LIS(Array A of length n)
    Create Array L of length n initially all ones
    for j = 2 to n do
        for i = 1 to j − 1 do
            if A[i] < A[j] AND L[i] + 1 > L[j] then
                L[j] = L[i] + 1
    return L(n)
end function
```

Exercise: Show how to modify the above dynamic programming algorithm to also return what the longest increasing subsequence is, not just how long it is.

Exercise: You are considering opening a series of ice cream shops along a long, straight beach. There are n potential locations where you could open a shop, and the distance of these locations from the start of the beach are, in meters and in increasing order, $d_1, d_2, d_3, ..., d_n$; you may assume all of the d_i are integers. The constraints are as follows:

- At each location you may open at most one ice cream shop.
- The expected profit from opening an ice cream shop at location i is p_i, where $p_i > 0$ and $i = 1, 2, 3, ..., n$.
- Any two ice cream shops should be at least x meters apart, where x is a positive integer.

Give an efficient algorithm to compute the maximum expected total profit subject to these constraints.

Exercise: Implement one of the above algorithms in python.

2.4 Edit Distance

The *edit distance* between two strings X and Y is the minimum number of insertions, deletions, or substitutions needed to change one string into another.

Insertion: ABCDE → AZBCDE

Deletion: ABCDE → ABDE

Substition: ABCDE → ABCXE

Example: What's the edit distance between SUNNY and SNOWY?

Example: What's the edit distance between GERRYMANDERING and RECOMBINATION?

How might you break the problem up into subproblems?

How might you use solutions to subproblems to find a solution to the original problem?

The right subproblem to consider here is $E(i, j) =$ the edit distance between the first i characters of X and the first j characters of Y. How do you write $E(i, j)$ in terms of solutions to smaller subproblems? What are the base cases/smallest subproblems you consider?

Consider the following dynamic programming algorithm for Edit Distance:

Algorithm 9 Dynamic Program for Edit Distance

function EDITDIST(String X of length m and string Y of length n)
  Create an $m+1 \times n+1$ array E
  for $i = 0$ to m **do** $E(i, 0) = i$
  for $j = 1$ to n **do** $E(0, j) = j$
  for $i = 1$ to m **do**
    for $j = 2$ to n **do**
      $E(i, j) = \min\{E(i-1, j) + 1, E(i, j-1) + 1, E(i-1, j-1) + \delta_{X[i],Y[j]}\}$
      ▷ $\delta_{a,b} = 1$ if $a = b$, $\delta_{a,b} = 0$ if $a \neq b$
  return $C(m, n)$
end function

Exercise: Show how to modify the above dynamic programming algorithm to also return how to change string X into string Y, not just what the edit distance is.

Exercise: A DNA sequence is a string consisting of only the characters G, C, T, and A. Suppose you are given two different DNA sequences X (of length n) and Y (of length m), and you want to determine how many times string X appears as a substring of Y (the characters of X don't have to appear consecutively in Y, but they do have to appear in the same order as in X). Give an efficient algorithm that takes X and Y as input and outputs the number of times X appears as a substring of Y.

Exercise: Suppose you are given two strings, X of length n and Y of length m, and you want to find the length of their longest common subsequence. For example, the longest common subsequence of $ALGORITHMS$ and $GEORGIATECH$ is $GORITH$, or length 6; another common subsequence that is not as long is ATH. (It's merely a coincidence that the letters of $GORITH$ appear consecutively in $ALGORITHMS$, in general the subsequences we consider do not have to be consecutive). Give a dynamic programming algorithm for this problem.

Exercise: You have n items, labelled $1, 2, \ldots, n$, each of which has a weight w_i and a value v_i. You are given a knapsack which can hold a total weight W of items. Assume all w_i are positive integers and W is a positive integer. Give a dynamic programming algorithm that will figure out the most total value you can fit in your knapsack (you can't include partial items).

Bonus Exercise: What is the running time of your dynamic programming algorithm for the knapsack problem in the previous exercise? Does this contradict the fact that the knapsack problem is NP-hard, meaning there is not believed to an algorithm for solving it that runs in time polynomial in the size of its input? Why or why not?

Exercise: Implement one of the above algorithms in python.

2.5 Other types of Subproblems

We saw just two ways of breaking problems up into subproblems; there are many more! I like the helpful graphic on page 178 of Dasgupta, Papdimitriou, and Vazirani's Algorithms book, available on one author's website: https://people.eecs.berkeley.edu/~vazirani/algorithms/chap6.pdf

Some exercises using a different kind of substructure:

Exercise: A certain string-processing language offers a primitive operation which splits a string into two pieces. Since this operation involves copying the original string, it takes n units of time for a string of length n, regardless of the location of the cut. Suppose, now, that you want to break a string into many pieces. The order in which the breaks are made can affect the total running time. For example, if you want to cut a 20-character string at positions 3 and 10, then making the first cut at position 3 incurs a total cost of 20+17 = 37, while doing position 10 first has a better cost of 20+10=30.

Give a dynamic programming algorithm that, given the locations c_1, c_2, ... , c_m of the m cuts in increasing order in a string of length n, finds the minimum cost of breaking the string into $m + 1$ pieces at the m cut locations.

Exercise: (Matrix Chain Multiplication) You can only multiply matrices when their indices line up: multiplying an $a \times b$ dimensional matrix with a $c \times d$ dimensional matrix is only possibly when $b = c$. The resulting matrix has dimension $a \times d$.

You can multiply an $x \times y$ dimensional matrix with a $y \times z$ dimensional matrix in $x \cdot y \cdot z$ steps. When multiplying a chain of matrices, you get the same result no matter how you group the multiplications: $A(BC) = (AB)C$. But, the time it takes to do these multiplications is different depending on the parentheses. If matrix A is 4×7, matrix B is 7×3, and C is 3×2, calculating $A(BC)$ takes $7 \cdot 3 \cdot 2 + 4 \cdot 7 \cdot 2 = 98$ steps and multiplying $(AB)C$ takes $4 \cdot 7 \cdot 3 + 4 \cdot 3 \cdot 2 = 108$ step. Write a program that, given a list of the dimensions of matrices to multiply, calculates the way to put the parentheses around the matrix product so that the multiplication is as fast as possible.

Note: This matrix multiplication problem gives a hint to the solution to the bonus exercise above, about how to calculate Fibonacci numbers faster.

Exercise: Given a tree with n nodes, root r, and an integer weight on every vertex, give an algorithm that computes the largest sum along a path from a leaf to the root. Make sure your algorithm only visits each node once.

Exercise: Given a tree with n nodes, give a dynamic programming algorithm that figures out which node you should designate as the root to get a tree with the largest possible height (the height of a tree is the longest distance from the root to a leaf).

2.6 Memoization

Another way to do dynamic programming is to use Memoization: still do recursion, but write down answers to subproblems you've already solved so you don't solve them multiple times. This can be useful when you think you won't need to solve all subproblems, but you don't know ahead of time which ones you'll have to solve.

As an example, Algorithm 10 is a memoization algorithm for Fibonacci numbers.

Algorithm 10 Memoization algorithm for computing the n^{th} Fibonacci number

```
function FibMemMain(n)                          ▷ Computes the n^th Fibonacci number
    Create an array FibNumbers of length n+1
    for i = 0 to n do FibNumbers[i] = -1
    return FibMem(n, FibNumbers)                ▷ Pass FibNumbers by reference
end function

function FibMem(i,FibNumbers)
    if FibNumbers[i] ≠ -1 then return FibNumbers[i]
    if i = 0 or i = 1 then
        FibNumbers[i] = 1
    else
        FibNumbers[i] = FibMem(i-1,FibNumbers) + FibMem(i-2, FibNumbers)
    return FibNumbers[i]
end function
```

2.7 More examples

Examples Relevant to VRDI: In the implementation of the recombination chain, calculating where to split a spanning tree into two parts with equal population is done with dynamic programming.

In Justin's recent work on finding cycles in series-parallel graphs, this is done via dynamic programming.

Shortest Paths We'll see more about shortest paths when we look at Linear and Integer Programming tomorrow, but you can also compute shortest paths in a graph using dynamic programming. The Bellman-Ford algorithm (for computing shortest paths from a given vertex) and the Floyd-Warshall algorithm (for simultaneously computing the shortest paths between all pairs of vertices) are both examples of dynamic programs.

More resources: For a long list of practice dynamic programming problems, see the exercises at the end of Chapter 3 of Jeff Erickson's Algorithms textbook, available free on the author's website: `http://jeffe.cs.illinois.edu/teaching/algorithms/book/03-dynprog.pdf`