Those who cannot remember the past are doomed to repeat it.

— George Santayana, *The Life of Reason, Book I: Introduction and Reason in Common Sense* (1905)

The "Dynamic-Tension®" bodybuilding program takes only 15 minutes a day in the privacy of your room.

— Charles Atlas

2 Dynamic Programming (January 23 and 25)

2.1 Exponentiation (Again)

Last time we saw a "divide and conquer" algorithm for computing the expression a^n, given two integers a and n as input: first compute $a^{\lfloor n/2 \rfloor}$, then $a^{\lceil n/2 \rceil}$, then multiply. If we computed both factors $a^{\lfloor n/2 \rfloor}$ and $a^{\lceil n/2 \rceil}$ recursively, the number of multiplications would be given by the recurrence

$$T(1) = 0, \qquad T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1.$$

The solution is $T(n) = n - 1$, so the naïve recursive algorithm uses *exactly* the same number of multiplications as the naïve iterative algorithm.[1]

In this case, it's obvious how to speed up the algorithm. Once we've computed $a^{\lfloor n/2 \rfloor}$, we we don't need to start over from scratch to compute $a^{\lfloor n/2 \rfloor}$; we can do it in at most one more multiplication. This same simple idea—**don't solve the same subproblem more than once**—can be applied to lots of recursive algorithms to speed them up, often (as in this case) by an exponential amount. The technical name for this technique is *dynamic programming*.

2.2 Fibonacci Numbers

The Fibonacci numbers F_n, named after Leonardo Fibonacci Pisano[2], are defined as follows: $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. The recursive definition of Fibonacci numbers immediately gives us a recursive algorithm for computing them:

```
RecFibo(n):
  if (n < 2)
      return n
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

How long does this algorithm take? Except for the recursive calls, the entire algorithm requires only a constant number of steps: one comparison and possibly one addition. If $T(n)$ represents the number of recursive calls to RecFibo, we have the recurrence

$$T(0) = 1, \quad T(1) = 1, \quad T(n) = T(n-1) + T(n-2) + 1.$$

This looks an awful lot like the recurrence for Fibonacci numbers! In fact, it's fairly easy to show by induction that $T(n) = 2F_{n+1} - 1$. In other words, computing F_n using this algorithm takes more than twice as many steps as just counting to F_n!

[1] But less time. If we assume that multiplying two n-digit numbers takes $O(n \log n)$ time, then the iterative algorithm takes $O(n^2 \log n)$ time, but the recursive algorithm takes only $O(n \log^2 n)$ time.

[2] Literally, "Leonardo, son of Bonacci, of Pisa".

2.3 Aside: The Annihilator Method

Just how slow is that? We can get a good asymptotic estimate for $T(n)$ by applying the annihilator method, described in the 'solving recurrences' handout:

$$\begin{aligned}
\langle T(n+2)\rangle &= \langle T(n+1)\rangle + \langle T(n)\rangle + \langle 1\rangle \\
\langle T(n+2) - T(n+1) - T(n)\rangle &= \langle 1\rangle \\
(E^2 - E - 1)\langle T(n)\rangle &= \langle 1\rangle \\
(E^2 - E - 1)(E-1)\langle T(n)\rangle &= \langle 0\rangle
\end{aligned}$$

The characteristic polynomial of this recurrence is $(r^2 - r - 1)(r-1)$, which has three roots: $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$, $\hat{\phi} = \frac{1-\sqrt{5}}{2} \approx -0.618$, and 1. Thus, the generic solution is

$$T(n) = \alpha\phi^n + \beta\hat{\phi}^n + \gamma.$$

Now we plug in a few base cases:

$$\begin{aligned}
T(0) &= 1 = \alpha + \beta + \gamma \\
T(1) &= 1 = \alpha\phi + \beta\hat{\phi} + \gamma \\
T(2) &= 3 = \alpha\phi^2 + \beta\hat{\phi}^2 + \gamma
\end{aligned}$$

Solving this system of linear equations gives us

$$\alpha = 1 + \frac{1}{\sqrt{5}}, \quad \beta = 1 - \frac{1}{\sqrt{5}}, \quad \gamma = -1,$$

so our final solution is

$$T(n) = \left(1 + \frac{1}{\sqrt{5}}\right)\phi^n + \left(1 - \frac{1}{\sqrt{5}}\right)\hat{\phi}^n - 1 = \Theta(\phi^n).$$

Actually, if we only want an asymptotic bound, we only need to show that $\alpha \neq 0$, which is much easier than solving the whole system of equations. Since ϕ is the largest characteristic root with non-zero coefficient, we immediately have $T(n) = \Theta(\phi^n)$.

2.4 Memo(r)ization and Dynamic Programming

The obvious reason for the recursive algorithm's lack of speed is that it computes the same Fibonacci numbers over and over and over. A single call to RecursiveFibo(n) results in one recursive call to RecursiveFibo$(n-1)$, two recursive calls to RecursiveFibo$(n-2)$, three recursive calls to RecursiveFibo$(n-3)$, five recursive calls to RecursiveFibo$(n-4)$, and in general, F_{k-1} recursive calls to RecursiveFibo$(n-k)$, for any $0 \leq k < n$. For each call, we're recomputing some Fibonacci number from scratch.

We can speed up the algorithm considerably just by writing down the results of our recursive calls and looking them up again if we need them later. This process is called *memoization*.[3]

[3] "My name is Elmer J. Fudd, millionaire. I own a mansion and a yacht."

```
MEMFIBO(n):
  if (n < 2)
      return n
  else
      if F[n] is undefined
          F[n] ← MEMFIBO(n − 1) + MEMFIBO(n − 2)
      return F[n]
```

If we actually trace through the recursive calls made by MEMFIBO, we find that the array F[] gets filled from the bottom up: first F[2], then F[3], and so on, up to F[n]. Once we realize this, we can replace the recursion with a simple for-loop that just fills up the array in that order, instead of relying on the complicated recursion to do it for us. This gives us our first explicit *dynamic programming* algorithm.

```
ITERFIBO(n):
  F[0] ← 0
  F[1] ← 1
  for i ← 2 to n
      F[i] ← F[i − 1] + F[i − 2]
  return F[n]
```

ITERFIBO clearly takes only $O(n)$ time and $O(n)$ space to compute F_n, an exponential speedup over our original recursive algorithm. We can reduce the space to $O(1)$ by noticing that we never need more than the last two elements of the array:

```
ITERFIBO2(n):
  prev ← 1
  curr ← 0
  for i ← 1 to n
      next ← curr + prev
      prev ← curr
      curr ← next
  return curr
```

(This algorithm uses the non-standard but perfectly consistent base case $F_{-1} = 1$.)

But even this isn't the fastest algorithm for computing Fibonacci numbers. There's a faster algorithm defined in terms of matrix multiplication, using the following wonderful fact:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} y \\ x + y \end{bmatrix}$$

In other words, multiplying a two-dimensional vector by the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$ does exactly the same thing as one iteration of the inner loop of ITERFIBO2. This might lead us to believe that multiplying by the matrix n times is the same as iterating the loop n times:

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} F_{n-1} \\ F_n \end{bmatrix}.$$

A quick inductive argument proves this. So if we want to compute the nth Fibonacci number, all we have to do is compute the nth power of the matrix $\left[\begin{smallmatrix} 0 & 1 \\ 1 & 1 \end{smallmatrix}\right]$.

We saw in the previous lecture, and the beginning of this lecture, that computing the nth power of something requires only $O(\log n)$ multiplications. In this case, that means $O(\log n)$ 2×2 matrix multiplications, but one matrix multiplications can be done with only a constant number of integer multiplications and additions. By applying our earlier dynamic programming algorithm for computing exponents, we can compute F_n in only $O(\log n)$ steps.

This is an exponential speedup over the standard iterative algorithm, which was already an exponential speedup over our original recursive algorithm. Right?

2.5 Uh... wait a minute.

Well, not exactly. Fibonacci numbers grow exponentially fast. The nth Fibonacci number is approximately $n \log_{10} \phi \approx n/5$ decimal digits long, or $n \log_2 \phi \approx 2n/3$ bits. So we can't possibly compute F_n in logarithmic time — we need $\Omega(n)$ time just to write down the answer!

I've been cheating by assuming we can do arbitrary-precision arithmetic in constant time. As we discussed last time, multiplying two n-digit numbers takes $O(n \log n)$ time. That means that the matrix-based algorithm's actual running time is given by the recurrence

$$T(n) = T(\lfloor n/2 \rfloor) + O(n \log n),$$

which solves to $T(n) = O(n \log n)$ by the Master Theorem.

Is this slower than our "linear-time" iterative algorithm? No! Addition isn't free, either. Adding two n-digit numbers takes $O(n)$ time, so the running time of the iterative algorithm is $O(n^2)$. (Do you see why?) So our matrix algorithm really is faster than our iterative algorithm, but not exponentially faster.

Incidentally, in the recursive algorithm, the extra cost of arbitrary-precision arithmetic is overwhelmed by the huge number of recursive calls. The correct recurrence is

$$T(n) = T(n-1) + T(n-2) + O(n),$$

which still has the solution $O(\phi^n)$ by the annihilator method.

2.6 The Pattern

Dynamic programming is essentially *recursion without repetition*. Developing a dynamic programming algorithm generally involves two separate steps.

1. **Formulate the problem recursively.** Write down a formula for the whole problem as a simple combination of the answers to smaller subproblems.

2. **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution by considering the intermediate subproblems in the correct order.

Of course, you have to prove that each of these steps is correct. If your recurrence is wrong, or if you try to build up answers in the wrong order, your algorithm won't work!

Dynamic programming algorithms need to store the results of intermediate subproblems. This is often *but not always* done with some kind of table.

2.7 Edit Distance

The *edit distance* between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word into another. For example, the edit distance between FOOD and MONEY is at most four:

$$\underline{\mathtt{F}}\mathtt{OOD} \rightarrow \mathtt{MO}\underline{\mathtt{O}}\mathtt{D} \rightarrow \mathtt{MON}\underline{\wedge}\mathtt{D} \rightarrow \mathtt{MONE}\underline{\mathtt{D}} \rightarrow \mathtt{MONEY}$$

A better way to display this editing process is to place the words one above the other, with a gap in the first word for every insertion, and a gap in the second word for every deletion. Columns with two *different* characters correspond to substitutions. Thus, the number of editing steps is just the number of columns that don't contain the same character twice.

```
F  O  O     D
M  O  N  E  Y
```

It's fairly obvious that you can't get from FOOD to MONEY in three steps, so their edit distance is exactly four. Unfortunately, this is not so easy in general. Here's a longer example, showing that the distance between ALGORITHM and ALTRUISTIC is at most six. Is this optimal?

```
A  L  G  O  R     I     T  H  M
A  L     T  R  U  I  S  T  I  C
```

To develop a dynamic programming algorithm to compute the edit distance between two strings, we first need to develop a recursive definition. Let's say we have an m-character string A and an n-character string B. Then define $E(i,j)$ to be the edit distance between the first i characters of A and the first j characters of B. The edit distance between the entire strings A and B is $E(m,n)$.

This gap representation for edit sequences has a crucial "optimal substructure" property. Suppose we have the gap representation for the shortest edit sequence for two strings. **If we remove the last column, the remaining columns must represent the shortest edit sequence for the remaining substrings.** We can easily prove this by contradiction. If the substrings had a shorter edit sequence, we could just glue the last column back on and get a shorter edit sequence for the original strings.

There are a couple of obvious base cases. The only way to convert the empty string into a string of j characters is by doing j insertions, and the only way to convert a string of i characters into the empty string is with i deletions:

$$E(i,0) = i, \qquad E(0,j) = j.$$

In general, there are three possibilities for the last column in the shortest possible edit sequence:

- **Insertion:** The last entry in the bottom row is empty. In this case, $E(i,j) = E(i-1,j) + 1$.
- **Deletion:** The last entry in the top row is empty. In this case, $E(i,j) = E(i,j-1) + 1$.
- **Substitution:** Both rows have characters in the last column. If the characters are the same, we don't actually have to pay for the substitution, so $E(i,j) = E(i-1,j-1)$. If the characters are different, then $E(i,j) = E(i-1,j-1) + 1$.

To summarize, the edit distance $E(i,j)$ is the smallest of these three possibilities:

$$E(i,j) = \min \left\{ \begin{array}{l} E(i-1,j)+1 \\ E(i,j-1)+1 \\ E(i-1,j-1) + \big[A[i] \neq B[j]\big] \end{array} \right\}$$

[The bracket notation $\big[P\big]$ denotes the *indicator variable* for the logical proposition P. Its value is 1 if P is true and 0 if P is false. This is the same as the C/C++/Java expression P ? 1 : 0.]

If we turned this recurrence directly into a recursive algorithm, we would have the following horrible double recurrence for the running time:

$$T(m,0) = T(0,n) = O(1), \qquad T(m,n) = T(m,n-1) + T(m-1,n) + T(n-1,m-1) + O(1).$$

Yuck!! The solution for this recurrence is an exponential mess! I don't know a general closed form, but $T(n,n) = \Theta((1+\sqrt{2})^n)$. Obviously a recursive algorithm is not the way to go here.

Instead, let's build an $m \times n$ table of all possible values of $E(i,j)$. We can start by filling in the base cases, the entries in the 0th row and 0th column, each in constant time. To fill in any other entry, we need to know the values directly above it, directly to the left, and both above and to the left. If we fill in our table in the standard way—row by row from top down, each row from left to right—then whenever we reach an entry in the matrix, the entries it depends on are already available.

```
EditDistance(A[1..m], B[1..n]):
  for i ← 1 to m
      Edit[i,0] ← i
  for j ← 1 to n
      Edit[0,j] ← j

  for i ← 1 to m
      for j ← 1 to n
          if A[i] = B[j]
              Edit[i,j] ← min { Edit[i-1,j] + 1,
                                Edit[i,j-1] + 1,
                                Edit[i-1,j-1] }
          else
              Edit[i,j] ← min { Edit[i-1,j] + 1,
                                Edit[i,j-1] + 1,
                                Edit[i-1,j-1] + 1 }

  return Edit[m,n]
```

Since there are $\Theta(n^2)$ entries in the table, and each entry takes $\Theta(1)$ time once we know its predecessors, the total running time is $\Theta(n^2)$.

Here's the resulting table for `ALGORITHM` → `ALTRUISTIC`. Bold numbers indicate places where characters in the two strings are equal. The arrows represent the predecessor(s) that actually define each entry. Each direction of arrow corresponds to a different edit operation: horizontal=deletion, vertical=insertion, and diagonal=substitution. Bold diagonal arrows indicate "free" substitutions of a letter for itself. A path of arrows from the top left corner to the bottom right corner of this table represents an optimal edit sequence between the two strings. There can be many such paths.

		A	L	G	O	R	I	T	H	M
	0	1	2	3	4	5	6	7	8	9
A	1	**0**	1	2	3	4	5	6	7	8
L	2	1	**0**	1	2	3	4	5	6	7
T	3	2	1	1	2	3	4	**4**	5	6
R	4	3	2	2	2	**2**	3	4	5	6
U	5	4	3	3	3	3	3	4	5	6
I	6	5	4	4	4	4	**3**	4	5	6
S	7	6	5	5	5	5	4	4	5	6
T	8	7	6	6	6	6	5	**4**	5	6
I	9	8	7	7	7	7	6	5	5	6
C	10	9	8	8	8	8	7	6	6	6

The edit distance between ALGORITHM and ALTRUISTIC is indeed six. There are three paths through this table from the top left to the bottom right, so there are three optimal edit sequences:

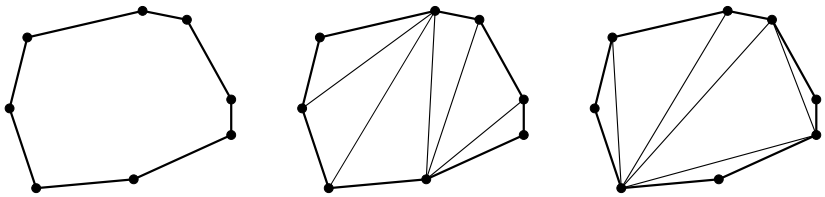
```
A  L  G  O  R  I     T  H  M
A  L  T  R  U  I  S  T  I  C

A  L  G  O  R     I     T  H  M
A  L     T  R  U  I  S  T  I  C

A  L  G  O  R     I     T  H  M
A  L  T     R  U  I  S  T  I  C
```

2.8 Optimal Triangulations of Convex Polygons

A *convex polygon* is a circular chain of line segments, arranged so none of the corners point inwards—imagine a rubber band stretched around a bunch of nails. (This is technically not the best definition, but it'll do for now.) A *diagonal* is a line segment that cuts across the interior of the polygon from one corner to another. A simple induction argument (hint, hint) implies that any n-sided convex polygon can be split into $n-2$ triangles by cutting along $n-3$ different diagonals. This collection of triangles is called a *triangulation* of the polygon. Triangulations are incredibly useful in computer graphics—most graphics hardware is built to draw triangles incredibly quickly, but to draw anything more complicated, you usually have to break it into triangles first.



A convex polygon and two of its many possible triangulations.