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 1950s were not good years for mathematical research. We had a very interesting gentleman in Washington named Wilson. He was secretary of Defense, and he actually had a pathological fear and hatred of the word 'research'*. I'm not using the term lightly; I'm using it precisely. His face would suffuse, he would turn red, and he would get violent if people used the term* 'research' *in his presence. You can imagine how he felt, then, about the term* 'mathematical'*. The RAND Corporation was employed by the Air Force, and the Air Force had Wilson as its boss, essentially. Hence, I felt I had to do something to shield Wilson and the Air Force from the fact that I was really doing mathematics inside the RAND Corporation. What title, what name, could I choose?*

— Richard Bellman, on the origin of his term 'dynamic programming' (1984)

If we all listened to the professor, we may be all looking for professor jobs.

— Pittsburgh Steeler's head coach Bill Cowher, responding to
David Romer's dynamic-programming analysis of football strategy (2003)

4 Dynamic Programming

4.1 Fibonacci Numbers

The Fibonacci numbers F_n, named after Leonardo Fibonacci Pisano[1], the mathematician who popularized 'algorism' in Europe in the 13th century, 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! The annihilator method gives us an asymptotic bound of $\Theta(\phi^n)$, where $\phi = (\sqrt{5}+1)/2 \approx 1.61803398875$, the so-called *golden ratio*, is the largest root of the polynomial $r^2 - r - 1$. But it's fairly easy to prove (hint, hint) the exact solution $\boxed{T(n) = 2F_{n+1} - 1}$. In other words, computing F_n using this algorithm takes more than twice as many steps as just counting to F_n!

Another way to see this is that the RECFIBO is building a big binary tree of additions, with nothing but zeros and ones at the leaves. Since the eventual output is F_n, our algorithm must call RECRIBO(1) (which returns 1) exactly F_n times. A quick inductive argument implies that RECFIBO(0) is called exactly F_{n-1} times. Thus, the recursion tree has $F_n + F_{n-1} = F_{n+1}$ leaves, and therefore, because it's a full binary tree, it must have $2F_{n+1} - 1$ nodes.

[1]literally, "Leonardo, son of Bonacci, of Pisa"

4.2 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*.[2]

```
MEMFIBO($n$):
  if ($n < 2$)
      return $n$
  else
      if $F[n]$ is undefined
          $F[n] \leftarrow$ 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($n$):
  $F[0] \leftarrow 0$
  $F[1] \leftarrow 1$
  for $i \leftarrow 2$ to $n$
      $F[i] \leftarrow F[i-1] + F[i-2]$
  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 $\leftarrow 1$
  curr $\leftarrow 0$
  for $i \leftarrow 1$ to $n$
      next $\leftarrow$ curr + prev
      prev $\leftarrow$ curr
      curr $\leftarrow$ 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:

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

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

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 $\left[\begin{smallmatrix}0&1\\1&1\end{smallmatrix}\right]$.

If we use repeated squaring, 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. Thus, we can compute F_n in only $O(\log n)$ integer arithmetic operations.

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

4.3 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 original 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.

4.4 The Pattern: Smart Recursion

In a nutshell, dynamic programming is *recursion without repetition*. Developing a dynamic programming algorithm almost always requires two distinct 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. This is usually easier than the first step.

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 store the solutions of intermediate subproblems, often *but not always* done with some kind of array or table. One common mistake that lots of students make is to be distracted by the table (because tables are easy and familiar) and miss the *much* more important (and difficult) task of finding a correct recurrence. ***Dynamic programming isn't about filling in tables; it's about smart recursion.*** As long as we memoize the correct recurrence, an explicit table isn't necessary, but if the recursion is incorrect, nothing works.

4.5 The Warning: Greed is Stupid

If we're very very very very lucky, we can bypass all the recurrences and tables and so forth, and solve the problem using a *greedy* algorithm. The general greedy strategy is look for the best first step, take it, and then continue. For example, a greedy algorithm for the edit distance problem might look for the longest common substring of the two strings, match up those substrings (since those substitutions dont cost anything), and then recursively look for the edit distances between the left halves and right halves of the strings. If there is no common substring—that is, if the two strings have no characters in common—the edit distance is clearly the length of the larger string.

If this sounds like a stupid hack to you, pat yourself on the back. It isn't even *close* to the correct solution. Nevertheless, for many problems involving dynamic programming, many student's first intuition is to apply a greedy strategy. This almost never works; problems that can be solved correctly by a greedy algorithm are *very* rare. Everyone should tattoo the following sentence on the back of their hands, right under all the rules about logarithms and big-Oh notation:

Greedy algorithms never work!
Use dynamic programming instead!

What, never? No, never! What, *never*? Well. . . hardly ever.[3]

A different lecture note describes the effort required to prove that greedy algorithms are correct, in the rare instances when they are. **You will not receive *any* credit for *any* greedy algorithm for *any* problem in this class without a *formal* proof of correctness.** We'll push through the formal proofs for two specific problems—minimum spanning trees and shortest paths—but those will be the only greedy algorithms we will consider this semester.

4.6 Edit Distance

The *edit distance* between two words—sometimes also called the *Levenshtein distance*—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:

$$\texttt{\underline{F}OOD} \longrightarrow \texttt{MO\underline{O}D} \longrightarrow \texttt{MON}_{\wedge}\texttt{D} \longrightarrow \texttt{MONE\underline{D}} \longrightarrow \texttt{MONEY}$$

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.

[3] He's hardly ever sick at sea! Then give three cheers, and one cheer more, for the hardy Captain of the *Pinafore*! Then give three cheers, and one cheer more, for the Captain of the *Pinafore*!

```
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. Once we figure out what should go in the last column, the Recursion Fairy will magically give us the rest of the optimal gap representation.

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

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

If neither string is empty, there are three possibilities for the last column in the shortest 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:[4]

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

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

$$T(m,n) = \begin{cases} O(1) & \text{if } n=0 \text{ or } m=0, \\ T(m,n-1)+T(m-1,n)+T(n-1,m-1)+O(1) & \text{otherwise.} \end{cases}$$

[4]Once again, I'm using Iverson's bracket notation $[P]$ to denote the *indicator variable* for the logical proposition P, which has value 1 if P is true and 0 if P is false.

I don't know of a general closed-form solution for this mess, but we can derive an upper bound by defining a new function

$$T'(N) = \max_{n+m=N} T(n,m) = \begin{cases} O(1) & \text{if } N = 0, \\ 2T(N-1) + T(N-2) + O(1) & \text{otherwise.} \end{cases}$$

The annihilator method implies that $T'(N) = O((1+\sqrt{2})^N)$. Thus, the running time of our recursive edit-distance algorithm is at most $T'(n+m) = O((1+\sqrt{2})^{n+m})$.

We can bring the running time of this algorithm down to a polynomial by building an $m \times n$ table of all possible values of $E(i,j)$. We begin 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.

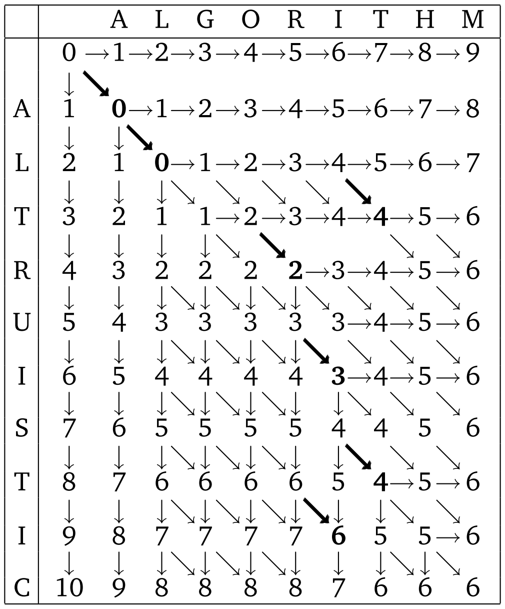
```
EditDistance($A[1..m], B[1..n]$):
    for $i \leftarrow 1$ to $m$
        Edit$[i,0] \leftarrow i$
    for $j \leftarrow 1$ to $n$
        Edit$[0,j] \leftarrow j$

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

    return Edit$[m,n]$
```

Since there are $\Theta(mn)$ entries in the table, and each entry takes $\Theta(1)$ time once we know its predecessors, the total running time is $\Theta(mn)$. The algorithm uses $O(mn)$ space.

Here's the resulting table for `ALGORITHM` $\rightarrow$ `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. Any 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.) Moreover, since we can compute these arrows in a postprocessing phase from the values stored in the table, we can reconstruct the actual optimal editing sequence in $O(n+m)$ additional time.



		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
```

*4.7 Saving Space: Divide and Conquer

Just as we did for the Fibonacci recurrence, we can reduce the space complexity of our algorithm to $O(m)$ by only storing the current and previous rows of the memoization table. However, if we throw away most of the rows in the table, we no longer have enough information to reconstruct the actual editing sequence. Now what?

Fortunately for memory-misers, in 1975 Dan Hirshberg discovered a simple divide-and-conquer strategy that allows us to compute the optimal editing sequence in $O(mn)$ time, using just $O(m)$ space. The trick is to record not just the edit distance for each pair of prefixes, but also a single position in the middle of the editing sequence for that prefix. Specifically, the optimal editing sequence that transforms $A[1..m]$ into $B[1..n]$ can be split into two smaller editing sequences, one transforming $A[1..m/2]$ into $B[1..h]$ for some integer h, the other transforming $A[m/2+1..m]$ into $B[h+1..n]$. To compute this breakpoint h, we define a second function $\text{Half}(i,j)$ as follows:

$$\text{Half}(i,j) = \begin{cases} \infty & \text{if } i < m/2 \\ j & \text{if } i = m/2 \\ \text{Half}(i-1,j) & \text{if } i > m/2 \text{ and } \text{Edit}(i,j) = \text{Edit}(i-1,j)+1 \\ \text{Half}(i,j-1) & \text{if } i > m/2 \text{ and } \text{Edit}(i,j) = \text{Edit}(i,j-1)+1 \\ \text{Half}(i-1,j-1) & \text{otherwise} \end{cases}$$

A simple inductive argument implies that Half(m,n) is the correct value of h. We can easily modify our earlier algorithm so that it computes Half(m,n) at the same time as the edit distance Edit(m,n), all in $O(mn)$ time, using only $O(m)$ space.

Now, to compute the optimal editing sequence that transforms A into B, we recursively compute the optimal subsequences. The recursion bottoms out when one string has only constant length, in which case we can determine the optimal editing sequence by our old dynamic programming algorithm. Overall the running time of our recursive algorithm satisfies the following recurrence:

$$T(m,n) = \begin{cases} O(n) & \text{if } m \le 1 \\ O(m) & \text{if } n \le 1 \\ O(mn) + T(m/2, h) + T(m/2, n-h) & \text{otherwise} \end{cases}$$

It's easy to prove inductively that $T(m,n) = O(mn)$, no matter what the value of h is. Specifically, the entire algorithm's running time is at most twice the time for the initial dynamic programming phase.

$$\begin{aligned} T(m,n) &\le \alpha mn + T(m/2,h) + T(m/2,n-h) \\ &\le \alpha mn + 2\alpha mh/2 + 2\alpha m(n-h)/2 && \text{[inductive hypothesis]} \\ &= 2\alpha mn \end{aligned}$$

A similar inductive argument implies that the algorithm uses only $O(n+m)$ space.

4.8 Optimal Binary Search Trees

A few lectures ago we developed a recursive algorithm for the optimal binary search tree problem. We are given a sorted array $A[1\,..\,n]$ of search keys and an array $f[1\,..\,n]$ of frequency counts, where $f[i]$ is the number of searches to $A[i]$. Our task is to construct a binary search tree for that set such that the total cost of all the searches is as small as possible. We developed the following recurrence for this problem:

$$\textit{OptCost}(f[1\,..\,n]) = \min_{1\le r\le n} \left\{ \textit{OptCost}(f[1\,..\,r-1]) \;+\; \sum_{i=1}^{n} f[i] \;+\; \textit{OptCost}(f[r+1\,..\,n]) \right\}$$

To put this recurrence in more standard form, fix the frequency array f, and let $S(i,j)$ denote the total search time in the optimal search tree for the subarray $A[i\,..\,j]$. To simplify notation a bit, let $F(i,j)$ denote the total frequency count for all the keys in the interval $A[i\,..\,j]$:

$$F(i,j) = \sum_{k=i}^{j} f[k]$$

We can now write

$$\boxed{S(i,j) = \begin{cases} 0 & \text{if } j < i \\ F(i,j) + \displaystyle\min_{i\le r\le j} \big(S(i,r-1) + S(r+1,j)\big) & \text{otherwise} \end{cases}}$$

The base case might look a little weird, but all it means is that the total cost for searching an empty set of keys is zero.

The algorithm will be somewhat simpler and more efficient if we precompute all possible values of $F(i,j)$ and store them in an array. Computing each value $F(i,j)$ using a separate for-loop would $O(n^3)$ time. A better approach is to turn the recurrence

$$F(i,j) = \begin{cases} f[i] & \text{if } i = j \\ F(i,j-1) + f[j] & \text{otherwise} \end{cases}$$

into the following $O(n^2)$-time dynamic programming algorithm:

```
InitF($f[1..n]$):
    for $i \leftarrow 1$ to $n$
        $F[i,i-1] \leftarrow 0$
        for $j \leftarrow i$ to $n$
            $F[i,j] \leftarrow F[i,j-1] + f[i]$
```

This will be used as an initialization subroutine in our final algorithm.

So now let's compute the optimal search tree cost $S(1,n)$ from the bottom up. We can store all intermediate results in a table $S[1..n, 0..n]$. Only the entries $S[i,j]$ with $j \geq i-1$ will actually be used. The base case of the recurrence tells us that any entry of the form $S[i,i-1]$ can immediately be set to 0. For any other entry $S[i,j]$, we can use the following algorithm fragment, which comes directly from the recurrence:

```
ComputeS($i,j$):
    $S[i,j] \leftarrow \infty$
    for $r \leftarrow i$ to $j$
        $tmp \leftarrow S[i,r-1] + S[r+1,j]$
        if $S[i,j] > tmp$
            $S[i,j] \leftarrow tmp$
    $S[i,j] \leftarrow S[i,j] + F[i,j]$
```

The only question left is what order to fill in the table.

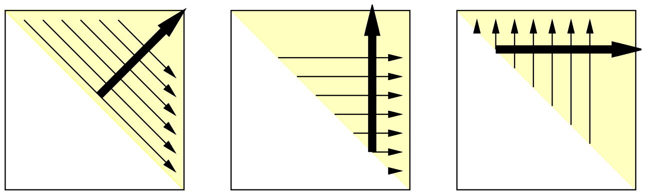
Each entry $S[i,j]$ depends on all entries $S[i,r-1]$ and $S[r+1,j]$ with $i \leq k \leq j$. In other words, every entry in the table depends on all the entries directly to the left or directly below. In order to fill the table efficiently, we must choose an order that computes all those entries before $S[i,j]$. There are at least three different orders that satisfy this constraint. The one that occurs to most people first is to scan through the table one diagonal at a time, starting with the trivial base cases $S[i,i-1]$. The complete algorithm looks like this:

```
OptimalSearchTree($f[1..n]$):
    InitF($f[1..n]$)
    for $i \leftarrow 1$ to $n$
        $S[i,i-1] \leftarrow 0$
    for $d \leftarrow 0$ to $n-1$
        for $i \leftarrow 1$ to $n-d$
            ComputeS($i,i+d$)
    return $S[1,n]$
```

We could also traverse the array row by row from the bottom up, traversing each row from left to right, or column by column from left to right, traversing each columns from the bottom up. These two orders give us the following algorithms:

```
OptimalSearchTree2($f[1..n]$):
    InitF($f[1..n]$)
    for $i \leftarrow n$ downto 1
        $S[i, i-1] \leftarrow 0$
        for $j \leftarrow i$ to $n$
            ComputeS($i, j$)
    return $S[1, n]$
```

```
OptimalSearchTree3($f[1..n]$):
    InitF($f[1..n]$)
    for $j \leftarrow 0$ to $n$
        $S[j+1, j] \leftarrow 0$
        for $i \leftarrow j$ downto 1
            ComputeS($i, j$)
    return $S[1, n]$
```



Three different orders to fill in the table $S[i, j]$.

No matter which of these three orders we actually use, the resulting algorithm runs in $\boxed{\Theta(n^3) \text{ time}}$ and uses $\boxed{\Theta(n^2) \text{ space}}$.

We could have predicted this from the original recursive formulation.

$$S(i,j) = \begin{cases} 0 & \text{if } j = i - i \\ F(i,j) + \min\limits_{i \le r \le j} \big(S(i, r-1) + S(r+1, j)\big) & \text{otherwise} \end{cases}$$

First, the function has two arguments, each of which can take on any value between 1 and n, so we probably need a table of size $O(n^2)$. Next, there are *three* variables in the recurrence (i, j, and r), each of which can take any value between 1 and n, so it should take us $O(n^3)$ time to fill the table.

In general, you can get an easy estimate of the time and space bounds for any dynamic programming algorithm by looking at the recurrence. The time bound is determined by how many values *all* the variables can have, and the space bound is determined by how many values the parameters of the function can have. For example, the (completely made up) recurrence

$$F(i,j,k,l,m) = \min_{0 \le p \le i} \max_{0 \le q \le j} \sum_{r=1}^{k-m} F(i-p, j-q, r, l-1, m-r)$$

should immediately suggest a dynamic programming algorithm to compute $F(n,n,n,n,n)$ in $O(n^8)$ time and $O(n^5)$ space. This simple rule of thumb usually gives us the right time bound to shoot for.

*4.9 Montonicity Helps

But not always! In fact, the algorithm I've described is *not* the most efficient algorithm for computing optimal binary search trees. Let $R[i, j]$ denote the root of the optimal search tree for $A[i..j]$. Donald Knuth proved the following nice monotonicity property for optimal subtrees: if we move either end of the subarray, the optimal root moves in the same direction or not at all, or more formally:

$$\boxed{R[i, j-1] \le R[i,j] \le R[i+1, j] \text{ for all } i \text{ and } j.}$$

This (nontrivial!) observation leads to the following more efficient algorithm:

```
FASTEROPTIMALSEARCHTREE($f[1..n]$):
  INITF($f[1..n]$)
  for $i \leftarrow n$ downto 1
      $S[i, i-1] \leftarrow 0$
      $R[i, i-1] \leftarrow i$
      for $j \leftarrow i$ to $n$
          COMPUTESANDR($i, j$)
          return $S[1, n]$
```

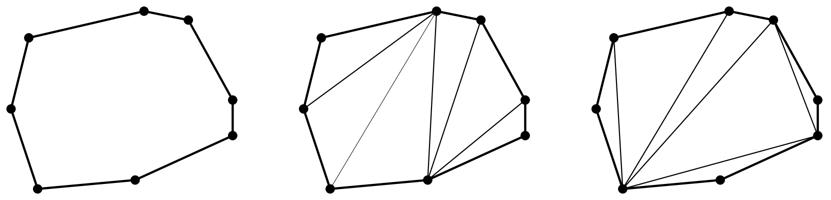
```
COMPUTESANDR($f[1..n]$):
  $S[i, j] \leftarrow \infty$
  for $r \leftarrow R[i, j-1]$ to $j$
      $tmp \leftarrow S[i, r-1] + S[r+1, j]$
      if $S[i, j] > tmp$
          $S[i, j] \leftarrow tmp$
          $R[i, j] \leftarrow r$
  $S[i, j] \leftarrow S[i, j] + F[i, j]$
```

It's not hard to see the r increases monotonically from i to n during each iteration of the *outermost* for loop. Consequently, the innermost for loop iterates at most n times during a single iteration of the outermost loop, so the total running time of the algorithm is $O(n^2)$.

If we formulate the problem slightly differently, this algorithm can be improved even further. Suppose we require the optimum *external* binary tree, where the keys $A[1..n]$ are all stored at the leaves, and intermediate pivot values are stored at the internal nodes. An algorithm due to Te Ching Hu and Alan Tucker[5] computes the optimal binary search tree in this setting in only $O(n \log n)$ time!

4.10 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.

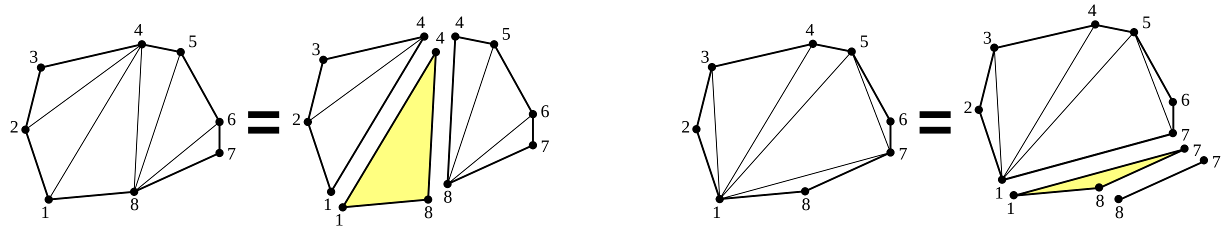
There are several different ways to triangulate any convex polygon. Suppose we want to find the triangulation that requires the least amount of ink to draw, or in other words, the triangulation where the total perimeter of the triangles is as small as possible. To make things concrete, let's label the corners of the polygon from 1 to n, starting at the bottom of the polygon and going clockwise. We'll need the following subroutines to compute the perimeter of a triangle joining three corners using their x- and y-coordinates:

```
$\Delta(i, j, k)$ :
  return DIST($i, j$) + DIST($j, k$) + DIST($i, k$)
```

```
DIST($i, j$) :
  return $\sqrt{(x[i] - x[j])^2 + (y[i] - y[j])^2}$
```

[5]T. C. Hu and A. C. Tucker, Optimal computer search trees and variable length alphabetic codes, *SIAM J. Applied Math.* 21:514–532, 1971. For a slightly simpler algorithm with the same running time, see A. M. Garsia and M. L. Wachs, A new algorithms for minimal binary search trees, *SIAM J. Comput.* 6:622–642, 1977. The original correctness proofs for both algorithms are rather intricate; for simpler proofs, see Marek Karpinski, Lawrence L. Larmore, and Wojciech Rytter, Correctness of constructing optimal alphabetic trees revisited, *Theoretical Computer Science*, 180:309-324, 1997.

In order to get a dynamic programming algorithm, we first need a recursive formulation of the minimum-length triangulation. To do that, we really need some kind of recursive definition of a *triangulation*! Notice that in any triangulation, exactly one triangle uses both the first corner and the last corner of the polygon. If we remove that triangle, what's left over is two smaller triangulations. The base case of this recursive definition is a 'polygon' with just two corners. Notice that at any point in the recursion, we have a polygon joining a contiguous subset of the original corners.



Two examples of the recursive definition of a triangulation.

Building on this recursive definition, we can now recursively define the total length of the minimum-length triangulation. In the best triangulation, if we remove the 'base' triangle, what remains must be the optimal triangulation of the two smaller polygons. So we just have choose the best triangle to attach to the first and last corners, and let the recursion fairy take care of the rest:

$$M(i,j) = \begin{cases} 0 & \text{if } j = i+1 \\ \displaystyle\min_{i<k<j} \big(\Delta(i,j,k) + M(i,k) + M(k,j)\big) & \text{otherwise} \end{cases}$$

What we're looking for is $M(1,n)$.

If you think this looks similar to the recurrence for $S(i,j)$, the cost of an optimal binary search tree, you're absolutely right. We can build up intermediate results in a two-dimensional table, starting with the base cases $M[i,i+1] = 0$ and working our way up. We can use the following algorithm fragment to compute a generic entry $M[i,j]$:

```
COMPUTEM($i, j$):
  $M[i,j] \leftarrow \infty$
  for $k \leftarrow i+1$ to $j-1$
      $tmp \leftarrow \Delta(i,j,k) + M[i,k] + M[k,j]$
      if $M[i,j] > tmp$
          $M[i,j] \leftarrow tmp$
```

As in the optimal search tree problem, each table entry $M[i,j]$ depends on all the entries directly to the left or directly below, so we can use any of the orders described earlier to fill the table.

```
MINTRIANGULATION:
  for $i \leftarrow 1$ to $n-1$
      $M[i,i+1] \leftarrow 0$
  for $d \leftarrow 2$ to $n-1$
      for $i \leftarrow 1$ to $n-d$
          COMPUTEM($i, i+d$)
  return $M[1,n]$
```

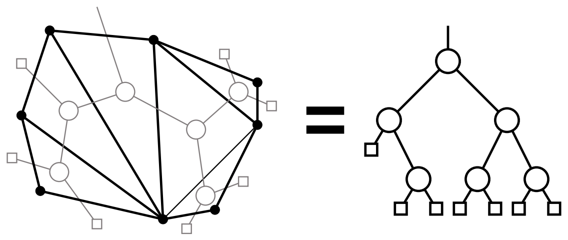
```
MINTRIANGULATION2:
  for $i \leftarrow n$ downto 1
      $M[i,i+1] \leftarrow 0$
      for $j \leftarrow i+2$ to $n$
          COMPUTEM($i, j$)
  return $M[1,n]$
```

```
MINTRIANGULATION3:
  for $j \leftarrow 2$ to $n$
      $M[j-1,j] \leftarrow 0$
      for $i \leftarrow j-1$ downto 1
          COMPUTEM($i, j$)
  return $M[1,n]$
```

In all three cases, the algorithm runs in $\Theta(n^3)$ time and uses $\Theta(n^2)$ space, just as we should have guessed from the recurrence.

4.11 It's the same problem!

Actually, the last two problems are both special cases of the same meta-problem: computing optimal *Catalan* structures. There is a straightforward one-to-one correspondence between the set of triangulations of a convex n-gon and the set of binary trees with $n-2$ nodes. In effect, these two problems differ only in the cost function for a single node/triangle.


A polygon triangulation and the corresponding binary tree. (Squares represent null pointers.)

A third problem that fits into the same mold is the infamous matrix chain multiplication problem. Using the standard algorithm, we can multiply a $p \times q$ matrix by a $q \times r$ matrix using $O(pqr)$ arithmetic operations; the result is a $p \times r$ matrix. If we have three matrices to multiply, the cost depends on which pair we multiply first. For example, suppose A and C are 1000×2 matrices and B is a 2×1000 matrix. There are two different ways to compute the threefold product ABC:

- $\boldsymbol{(AB)C}$**:** Computing AB takes $1000 \cdot 2 \cdot 1000 = 2\,000\,000$ operations and produces a 1000×1000 matrix. Multiplying this matrix by C takes $1000 \cdot 1000 \cdot 2 = 2\,000\,000$ additional operations. So the total cost of $(AB)C$ is $4\,000\,000$ operations.
- $\boldsymbol{A(BC)}$**:** Computing BC takes $2 \cdot 1000 \cdot 2 = 4000$ operations and produces a 2×2 matrix. Multiplying A by this matrix takes $1000 \cdot 2 \cdot 2 = 4\,000$ additional operations. So the total cost of $A(BC)$ is only 8000 operations.

Now suppose we are given an array $D[0\mathinner{..}n]$ as input, indicating that each matrix M_i has $D[i-1]$ rows and $D[i]$ columns. We have an exponential number of possible ways to compute the n-fold product $\prod_{i=1}^{n} M_i$. The following dynamic programming algorithm computes the number of arithmetic operations for the best possible parenthesization:

```
MATRIXCHAINMULT:
  for $i \leftarrow n$ downto $1$
      $M[i, i+1] \leftarrow 0$
      for $j \leftarrow i+2$ to $n$
          COMPUTEM$(i, j)$
  return $M[1, n]$
```

```
COMPUTEM$(i, j)$:
  $M[i, j] \leftarrow \infty$
  for $k \leftarrow i+1$ to $j-1$
      $tmp \leftarrow (D[i] \cdot D[j] \cdot D[k]) + M[i,k] + M[k,j]$
      if $M[i,j] > tmp$
          $M[i,j] \leftarrow tmp$
```

The derivation of this algorithm is left as a simple exercise.

4.12 More Examples

We've already seen two other examples of recursive algorithms that we can significantly speed up via dynamic programming.

4.12.1 Subset Sum

Recall from the very first lecture that the *Subset Sum* problem asks, given a set X of positive integers (represented as an array $X[1..n]$ and an integer T, whether any subset of X sums to T. In that lecture, we developed a recursive algorithm which can be reformulated as follows. Fix the original input array $X[1..n]$ and the original target sum T, and define the boolean function

$$S(i,t) = \text{some subset of } X[i..n] \text{ sums to } t.$$

Our goal is to compute $S(1,T)$, using the recurrence

$$S(i,t) = \begin{cases} \text{TRUE} & \text{if } t = 0, \\ \text{FALSE} & \text{if } t < 0 \text{ or } i > n, \\ S(i+1,t) \vee S(i+1,t-X[i]) & \text{otherwise.} \end{cases}$$

Observe that there are only nT possible values for the input parameters that lead to the interesting case of this recurrence, so storing the results of all such subproblems requires $\boxed{O(mn) \text{ space}}$. If $S(i+1,t)$ and $S(i+1,t-X[i])$ are already known, we can compute $S(i,t)$ in constant time, so memoizing this recurrence gives us and algorithm that runs in $\boxed{O(nT) \text{ time}}$.[6] To turn this into an explicit dynamic programming algorithm, we only need to consider the subproblems $S(i,t)$ in the proper order:

```
SUBSETSUM($X[1..n], T$):
    $S[n+1,0] \leftarrow$ TRUE
    for $t \leftarrow 1$ to $T$
        $S[n+1,t] \leftarrow$ FALSE

    for $i \leftarrow n$ downto $1$
        $S[i,0] =$ TRUE
        for $t \leftarrow 1$ to $X[i]-1$
            $S[i,t] \leftarrow S[i+1,t]$        ⟨⟨Avoid the case $t < 0$⟩⟩
        for $t \leftarrow X[i]$ to $T$
            $S[i,t] \leftarrow S[i+1,t] \vee S[i+1,t-X[i]]$

    return $S[1,T]$
```

This direct algorithm clearly always uses $\boxed{O(nT) \text{ time and space}}$. In particular, if T is significantly larger than 2^n, this algorithm is actually slower than our naïve recursive algorithm. Dynamic programming isn't *always* an improvement!

4.12.2 Longest Increasing Subsequence

We also developed a recurrence for the longest increasing subsequence problem. Fix the original input array $A[1..n]$ with a sentinel value $A[0] = -\infty$. Let $L(i,j)$ denote the length of the longest increasing subsequence of $A[j..n]$ with all elements larger than $A[i]$. Our goal is to compute $L(0,1)-1$. (The -1 removes the sentinel $-\infty$.) For any $i < j$, our recurrence can be stated as follows:

$$L(i,j) = \begin{cases} 0 & \text{if } j > n \\ L(i,j+1) & \text{if } A[i] \geq A[j] \\ \max\{L(i,j+1),\ 1 + L(j,j+1)\} & \text{otherwise} \end{cases}$$

[6]This does not contradict our earlier upper bound of $O(2^n)$. Both upper bounds are correct. Which bound is actually better depends on the size of T.

The recurrence suggests that our algorithm should use $O(n^2)$ time and space, since the input parameters i and j each can take n different values. To get an explicit dynamic programming algorithm, we only need to ensure that both $L(i, j+1)$ and $L(j, j+1)$ are considered before $L(i, j)$, for all i and j.

```
LIS($A[1..n]$):
    $A[0] \gets -\infty$                  ⟨⟨Add a sentinel⟩⟩
    for $i \gets 0$ to $n$                 ⟨⟨Base cases⟩⟩
        $L[i, n+1] \gets 0$
    for $j \gets n$ downto $1$
        for $i \gets 0$ to $j-1$
            if $A[i] \geq A[j]$
                $L[i,j] \gets L[i, j+1]$
            else
                $L[i,j] \gets \max\{L[i,j+1],\ 1 + L[j,j+1]\}$
    return $L[0,1] - 1$              ⟨⟨Don't count the sentinel⟩⟩
```

As predicted, this algorithm clearly uses $\boxed{O(n^2) \text{ time and space}}$. We can reduce the space to $O(n)$ by only maintaining the two most recent columns of the table, $L[\cdot, j]$ and $L[\cdot, j+1]$.

This is not the only recursive strategy we could use for computing longest increasing subsequences. Here is another recurrence that gives us the $O(n)$ space bound for free. Let $L'(i)$ denote the length of the longest increasing subsequence of $A[i..n]$ that starts with $A[i]$. Our goal is to compute $L'(0) - 1$. To define $L'(i)$ recursively, we only need to specify the *second* element in subsequence; the Recursion Fairy will do the rest.

$$L'(i) = 1 + \max\left\{L'(j) \mid j > i \text{ and } A[j] > A[i]\right\}$$

Here, I'm assuming that $\max \varnothing = 0$, so that the base case is $L'(n) = 1$ falls out of the recurrence automatically. Memoizing this recurrence requires $O(n)$ space, and the resulting algorithm runs in $O(n^2)$ time. To transform this into a dynamic programming algorithm, we only need to guarantee that $L'(j)$ is computed before $L'(i)$ whenever $i < j$.

```
LIS2($A[1..n]$):
    $A[0] = -\infty$                     ⟨⟨Add a sentinel⟩⟩
    for $i \gets n$ downto $0$
        $L'[i] \gets 1$
        for $j \gets i+1$ to $n$
            if $A[j] > A[i]$ and $1 + L'[j] > L'[i]$
                $L'[i] \gets 1 + L'[j]$
    return $L'[0] - 1$               ⟨⟨Don't count the sentinel⟩⟩
```