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 (September 5 and 10)

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 $\boxed{T(n) = 2F_{n+1} - 1}$. In other words, computing F_n using this algorithm takes more than twice as many steps as just counting to F_n!

[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 this recursive algorithm takes only $O(n \log^2 n)$ time.

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

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. (See Homework Zero!)

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 $0 \le k < n$. 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:

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

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.

```
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 $[P]$ 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 Danger! Greed kills!

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 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!

Well... hardly ever. Later in the semester we'll see correct greedy algorithms for minimum spanning trees and shortest paths.

2.9 Optimal Binary Search Trees

You all remember that the cost of a successful search in a binary search tree is proportional to the depth of the target node plus one. As a result, the worst-case search time is proportional to the height of the tree. To minimize the worst-case search time, the height of the tree should be as small as possible; ideally, the tree is perfectly balanced.

In many applications of binary search trees, it is more important to minimize the total cost of several searches than to minimize the worst-case cost of a single search. If x is a more 'popular' search target than y, we can save time by building a tree where the depth of x is smaller than the depth of y, even if that means increasing the overall height of the tree. A perfectly balanced tree is *not* the best choice if some items are significantly more popular than others. In fact, a totally unbalanced tree of depth $\Omega(n)$ might actually be the best choice!

This situation suggests the following problem. Suppose we are given a sorted array of n keys $A[1..n]$ and an array of corresponding *access frequencies* $f[1..n]$. Over the lifetime of the search tree, we will search for the key $A[i]$ exactly $f[i]$ times. Our task is to build the binary search tree that minimizes the *total* search time.

Before we think about how to solve this problem, we should first come up with the right way to describe the function we are trying to optimize! Suppose we have a binary search tree T. Let $\text{depth}(T, i)$ denote the depth of the node in T that stores the key $A[i]$. Up to constant factors, the total search time $S(T)$ is given by the following expression:

$$S(T) = \sum_{i=1}^{n} \big(\text{depth}(T, i) + 1\big) \cdot f[i]$$

This expression is called the *weighted external path length* of T. We can trivially split this expression into two components:

$$S(T) = \sum_{i=1}^{n} f[i] + \sum_{i=1}^{n} \text{depth}(T, i) \cdot f[i].$$

The first term is the total number of searches, which doesn't depend on our choice of tree at all. The second term is called the weighted *internal* path length of T.

We can express $S(T)$ in terms of the recursive structure of T as follows. Suppose the root of T contains the key $A[r]$, so the left subtree stores the keys $A[1..r-1]$, and the right subtree stores the keys $A[r+1..n]$. We can actually define $\text{depth}(T, i)$ recursively as follows:

$$\text{depth}(T, i) = \begin{cases} \text{depth}(\text{left}(T), i) + 1 & \text{if } i < r \\ 0 & \text{if } i = r \\ \text{depth}(\text{right}(T), i) + 1 & \text{if } i > r \end{cases}$$

If we plug this recursive definition into our earlier expression for $S(T)$, we get the following:

$$S(T) = \sum_{i=1}^{n} f[i] \;+\; \sum_{i=1}^{r-1} \big(\text{depth}(\text{left}(T), i) + 1\big) \cdot f[i] \;+\; \sum_{i=1}^{r-1} \big(\text{depth}(\text{right}(T), i) + 1\big) \cdot f[i]$$

This looks complicated, until we realize that the second and third look exactly like our initial expression for $S(T)$!

$$\boxed{S(T) = \sum_{i=1}^{n} f[i] + S(\text{left}(T)) + S(\text{right}(T))}$$

Now our task is to compute the tree T_{opt} that minimizes the total search time $S(T)$. Suppose the root of T_{opt} stores key $A[r]$. The recursive definition of $S(T)$ immediately implies that the left subtree $\text{left}(T_{\text{opt}})$ must also be the optimal search tree for the keys $A[1..r-1]$ and access frequencies $f[1..r-1]$. Similarly, the right subtree $\text{right}(T_{\text{opt}})$ must also be the optimal search tree for the keys $A[r+1..n]$ and access frequencies $f[r+1..n]$. Thus, once we choose the correct key to store at the root, the recursion fairy will automatically construct the rest of the optimal tree for us!

More formally, let $S(i, j)$ denote the total search time for the *optimal* search tree containing the subarray $A[1..j]$; our task is to compute $S(1, n)$. To simplify notation a bit, let $F(i, j)$ denote the total frequency counts for all the keys in the subarray $A[i..j]$:

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

We now have the following recurrence:

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

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. We could use the base cases $S(i, i) = f[i]$ instead, but this would lead to extra special cases when $r = 1$ or $r = n$. Also, the $F(i, j)$ term is outside the max because it doesn't depend on the root index r.

Now, if we try to evaluate this recurrence directly using a recursive algorithm, the running time will have the following evil-looking recurrence:

$$T(n) = \Theta(n) + \sum_{k=1}^{n} \big(T(k-1) + T(n-k)\big)$$

The $\Theta(n)$ term comes from computing $F(1, n)$, which is the total number of searches. A few minutes of pain and suffering by a professional algorithm analyst gives us the solution $T(n) = \Theta(3^n)$. Once again, top-down recursion is not the way to go.

In fact, we're not even computing the access counts $F(i, j)$ as efficiently as we could. Even if we memoize the answers in an array $F[1..n][1..n]$, computing each value $F(i, j)$ using a separate for-loop requires a total of $O(n^3)$ 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 ← 1 to n
      F[i, i-1] ← 0
      for j ← i to n
          F[i, j] ← 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 recursion 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] ← ∞
  for r ← i to j
      tmp ← S[i, r − 1] + S[r + 1, j]
      if S[i, j] > tmp
          S[i, j] ← tmp
  S[i, j] ← S[i, j] + F[i, j]
```

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

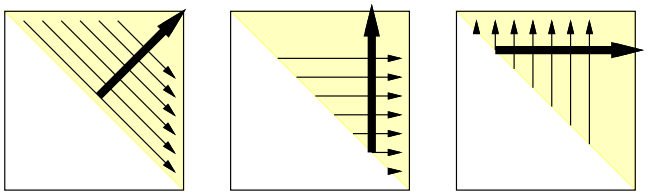
Each entry $S[i, j]$ depends on all entries $S[i, r-1]$ and $S[r+1, j]$ with $i \leq k \leq j$. In other words, every entry in the table depends on all the entries directly to the left or directly below. In order to fill the table efficiently, we must choose an order that computes all those entries before $S[i, j]$. 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 ← 1 to n
      S[i, i − 1] ← 0
  for d ← 0 to n − 1
      for i ← 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 ← n downto 1
      S[i, i − 1] ← 0
      for j ← i to n
          COMPUTES(i, j)
  return S[1, n]
```

```
OPTIMALSEARCHTREE3(f[1..n]):
  INITF(f[1..n])
  for j ← 0 to n
      S[j + 1, j] ← 0
      for i ← 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 $\Theta(n^3)$ time and uses $\Theta(n^2)$ 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 suggests a dynamic programming algorithm that uses $O(n^8)$ time and $O(n^5)$ space. This rule of thumb immediately usually gives us the right time bound to shoot for.

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 ← n downto 1
      S[i, i-1] ← 0
      R[i, i-1] ← i
      for j ← i to n
          ComputeSandR(i, j)
          return S[1, n]
```

```
ComputeSandR(f[1..n]):
  S[i, j] ← ∞
  for r ← R[i, j-1] to j
      tmp ← S[i, r-1] + S[r+1, j]
      if S[i, j] > tmp
          S[i, j] ← tmp
          R[i, j] ← r
  S[i, j] ← 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 $O(n^2)$.

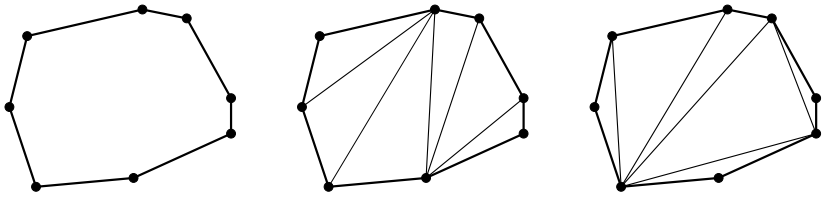
If we formulate the problem slightly differently, this algorithm can be improved even further. Suppose we require the optimum *external* binary tree, where the keys $A[1\,..\,n]$ are all stored at the leaves, and intermediate pivot values are stored at the internal nodes. An algorithm due to Te Ching Hu and Alan Tucker[4] computes the optimal binary search tree in this setting in only

[4]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.

$O(n \log n)$ time!

2.10 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.



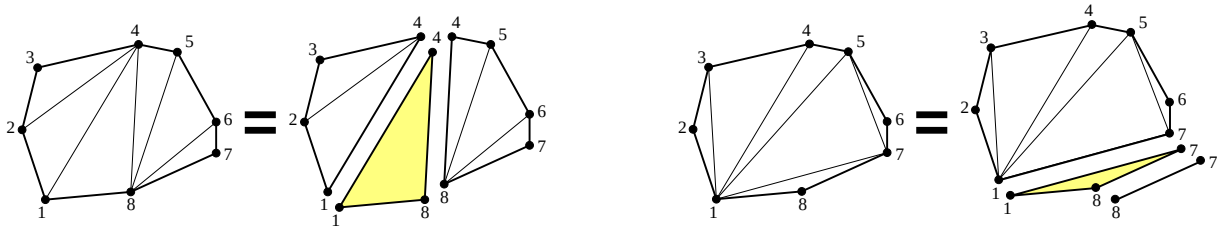
A convex polygon and two of its many possible triangulations.

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:

```
Δ(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)
```

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:

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

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] ← ∞
  for k ← i+1 to j-1
      tmp ← Δ(i,j,k) + M[i,k] + M[k,j]
      if M[i,j] > tmp
          M[i,j] ← 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 ← 1 to n-1
      M[i,i+1] ← 0
  for d ← 2 to n-1
      for i ← 1 to n-d
          COMPUTEM(i,i+d)
  return M[1,n]
```

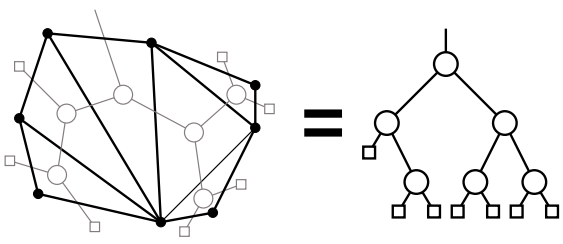
```
MINTRIANGULATION2:
  for i ← n downto 1
      M[i,i+1] ← 0
      for j ← i+2 to n
          COMPUTEM(i,j)
  return M[1,n]
```

```
MINTRIANGULATION3:
  for j ← 2 to n
      M[j-1,j] ← 0
      for i ← 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.

2.11 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:

- **$(AB)C$:** 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.
- **$A(BC)$:** 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 $D[0..n]$ 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 ← n downto 1
      M[i, i+1] ← 0
      for j ← i+2 to n
          ComputeM(i, j)
  return M[1, n]
```

```
ComputeM(i, j):
  M[i, j] ← ∞
  for k ← i+1 to j−1
      tmp ← (D[i] · D[j] · D[k]) + M[i, k] + M[k, j]
      if M[i, j] > tmp
          M[i, j] ← tmp
```

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