SI 335, Unit 4: Multiplication

Daniel S. Roche (roche@usna.edu)

Spring 2014

This unit covers algorithms for one of the most basic computations we could hope to perform: multiplication. We have already seen from the previous unit how multiplying huge numbers forms the backbone of RSA cryptography. We're going to look at how that actually works now, along with some new and exciting algorithmic ideas.

1 Representation of Numbers

We all know about *single-precision* integers, namely **int**s, which are stored in a single machine word of memory, as a sequence of bits corresponding to the base-2 (i.e., binary) representation of the integer.

Multiple-precision integers are those that are larger than 2^{32} (or perhaps 2^{64}) and therefore cannot be represented as a single **int**. These are instead stored as a *list of digits*, usually in an array indexed from the least-significant digit, where each digit is a single-precision integer.

(Notice that we are specifically assuming the integers we want to deal with are all greater than or equal to zero. If we want negative integers too, it's easy: just add a boolean flag to indicate negative-ness. For most of this unit, we'll quietly ignore this case because it doesn't really change the algorithms.)

From now on, we will say that each digit d satisfies $0 \leq d < B$, where the number B is the *base* and must be at least 2. In practice, B will usually correspond to the machine word size, like 2^{32} or 2^{64}.

For example, given the integer 4391354067575026 we could represent it in base $B = 10$ by the list

[6, 2, 0, 5, 7, 5, 7, 6, 0, 4, 5, 3, 1, 9, 3, 4]

or in base $B = 2^8 = 256$ by the list

[242, 224, 71, 203, 233, 153, 15]

Generally, if a number has n digits $[d_0, d_1, d_2, \ldots, d_{n-1}]$, in base B representation, then that number equals

$d_0 + d_1B + d_2B^2 + d_3B^3 + \cdots + d_{n-1}B^{n-1}$,

which is always between 0 and $B^n - 1$.

Does the choice of base matter? In practice, yes; the difference between say 1000 and 2000 words of memory can be huge. But as far as algorithms are concerned, *asymptotically*, it doesn't really matter. The difference in array lengths for any two bases will just be some constant factor (this is because of a basic property of logarithms). So we can always say the *size* of an integer x is $\Theta(\log x)$, no matter what base is used.

2 Basic Arithmetic with Integers

We've already been using big-integer arithmetic in the RSA algorithm. Now let's look at how it actually works. We'll start with the so-called "school" algorithms — like the ones you learned in grade school.

In presenting these algorithms, two important simplifications will be used throughout:

- The base B will always be 10. This is so that each digit in the way we normally think about numbers corresponds to one array entry in the computer representation. As discussed above, this doesn't make any *asymptotic* difference in the cost of any algorithm.
- We will always assume both integers to be added or multiplied have the same *size*. Why can we assume this? Well, if one integer is initially shorter than the other one, just add 0 digits to the end of the array until they have the same length. This process is called *zero padding* and obviously won't change the answer, but it does simplify the presentation. And it also doesn't change the *asymptotic* cost, since the size of the input will only change by (at most) a constant factor of two.

In practice, the base will be more like the machine word size, probably a power of two, and there will be special algorithm tweaks when the input integers are of different sizes. But the same general algorithmic principles will still apply.

2.1 Addition

Let's start with an example: add 40694 to 73593. Your first instinct is probably to stack these on top of each other, right-aligned, and start adding digits from right to left:

$$\begin{array}{cccccc} & 4 & 0 & 6 & 9 & 1 \\ + & 7 & 3 & 5 & 9 & 3 \\ \hline ? & ? & ? & ? & ? & 4 \end{array}$$

We see that the last digit of the sum is 4. What happens if we just do this all the way across the row? Remember that the individual digits are just separate numbers in an array, as far as the computer is concerned. So we get:

$$\begin{array}{cccccc} & 4 & 0 & 6 & 9 & 4 \\ + & 7 & 3 & 5 & 9 & 3 \\ \hline & 11 & 3 & 11 & 18 & 7 \end{array}$$

This isn't right because we ended up with individual digits in the sum that are larger than B. Of course you know that we just have to *carry* the overflowed 1's across to the next digit. But doing that might produce another carry, and another carry — will this go on forever?

Obviously not. We will start at the right hand side and at each step add together the two digits in that position, plus (possibly) the carry digit from the previous step. This produces the digit in that position of the sum, plus (possibly) a carry digit for the next step.

Next we need to ask, how big can the carry digit be? For base 10 at least, you can do some examples and notice that the carry is never more than 1. This is because, in the worst case, both digits in some position are equal to 9, and the carry from the previous position is 1, and $9 + 9 + 1 = 19$, which means the sum digit in that position is 9 and the carry is 1.

Base 10 is a simplification we made, but it turns out the carry can never be more than 1 in any base. In general, if we are in base B, the most any digit can be is $B - 1$, so we add $(B - 1) + (B - 1) + 1 = 2B - 1$. This is less than $2B$, so the carry digit (which corresponds to the quotient when dividing by B) is never greater than 1. Knowing this bound on how big the carry can be is important because it's how we know this method will finish and always produce a number that has at most $n + 1$ digits.

This leads us to the following algorithm. Certainly all of you knew this addition algorithm from grade school. But could you write it out as I have below? One thing to remember is that the right-hand side of the number when we write it down (the *least-significant digit*) is the digit that appears at the *beginning* of the array in the representation. Being able to transform our intuitive thoughts about a computation into an algorithm is an important skill! Don't worry, you'll get plenty of practice in this class...

Add

Input: Integers x and y, stored in arrays X and Y of exactly n digits each. Each digit is less than B.

Output: The integer $x + y$, stored in array Array A of length at most $n + 1$

```
def add(X, Y, B):
    n = len(X)
    carry = 0
    A = zero-filled array of length (n + 1)
    for i in range(0, n):
        carry, A[i] = divmod(X[i] + Y[i] + carry, B)
    A[n] = carry
    return A
```

(Note: **divmod** is a special built-in operation in Python that takes two numbers and returns both the quotient and the remainder of the division.)

Now from this it should be obvious that the worst-case cost of our algorithm is $\Theta(n)$, where n is the size of the input arrays. This is as good as it can get! Here n corresponds to the *size* of the input integers, which if you remember from the last unit is proportional to the logarithm of their *value*s. And it's really linear time because each of the operations is on small integers at most 2β.

Why is linear time the best possible? This might seem obvious, but at least in this case it's true that *we have to look at the entire input to get the right answer.* If even one digit in either input integer is changed, it changes the answer. So since it takes at least $\Omega(n)$ time to read through the input, this is a lower bound on any algorithm for this problem. Therefore the algorithm for integer addition that you learned in grade school is asymptotically optimal! Now go give your first grade teacher a big old kiss and thank them.

By the way, it should be mentioned that we can do subtraction in exactly the same way. With subtraction, however, the carries are *negative*, and we should be careful to only subtract bigger numbers from smaller ones, lest the whole result should be negative and we need a data structure more sophisticated than an array of digits to store it. (Not that much more sophisticated; we just need a flag to say if the integer is less than zero. But it's a complication not worth dealing with here.)

2.2 Multiplication

Certainly you remember long integer multiplication from elementary school. If you went to school in the United States, then the algorithm you taught probably looks something like this:

```
          7 4 0 7
      x   2 9 1 5
      -----------
        3 7 0 3 5
        7 4 0 7
    6 6 6 6 3
+ 1 4 8 1 4
-----------------
  2 1 5 9 1 4 0 5
```

To multiply x times y, we multiply x times each digit in y separately and then add up all these products, shifted appropriately by powers of the base B. Our algorithm to do this by computer will be almost the same, except that the sum at the bottom will be computed as we go along, rather than all at once at the end.

The multiplication of the first argument x by each digit of the second argument will be an inner loop in the algorithm. It will be accomplished similarly to the **add** algorithm above, with a carry digit to keep track of overflow as we go along. But once again, we should figure out how large this carry will be.

If you do some examples in base 10, you may notice that the carry is never more than 8. This is true even if we try a worst-case example like 99999 times 9. And once again, this generalizes to any base: the carry digit in multiplying by a single digit in base B is always at most $B-2$. To prove this we just have to work out the worst case:

$(B-1)\cdot(B-1)+(B-2) = B^2-2B+1+B-2 = (B-2)B+(B-1)$

Since the end of this equation looks like division with remainder by B, the quotient (which corresponds to the carry) is always at most $B-2$. This is nice because it means we can always store the carry in a single digit.

Here's the algorithm that we get:

standardMul

Input: Integers x and y, stored in arrays X and Y of exactly n digits each. Each digit is less than B.

Output: The integer xy, stored in array Array A of length at most $2n$

```
def smul(X, Y, B):
    n = len(X)
    A = zero-filled array of length (2*n)
    T = zero-filled array of length n
    for i in range(0, n):
        # set T = X * Y[i]
        carry = 0
        for j in range(0, n):
            carry, T[j] = divmod(X[j] * Y[i] + carry, B)
        # add T to A, the running sum
        A[i : i+n+1] = add(A[i : i+n], T[0 : n], B)
        A[i+n] = A[i+n] + carry
    return A
```

Analyzing this algorithm by now should be a piece of cake! The outer **for** loop runs exactly n times, and inside it we have another **for** loop that runs n times as well as a call to the **add** algorithm which has cost $\Theta(n)$. So the total cost of **standardMul** is $\Theta(n^2)$.

This leaves us with the question: *can we do better?* In the case of addition, because the standard algorithm is already linear-time, there's not much (any) room for improvement other than tweaks. But for multiplication, we have a *quadratic*-time algorithm, leaving lots of room to improve. But is this really possible? How could we avoid multiplying every digit of the first number by every digit of the second one?

3 Divide and Conquer Multiplication

The multiplication algorithm above is written iteratively, so we can easily analyze and implement it. But thinking *recursively* will be more helpful in reasoning about the algorithm and ultimately improving on it.

The way standard multiplication of x times y works is basically to write $y = y_1B + y_0$, where y_0 is the least significant digit of y, and then we multiply x times y_0 (the inner **for** loop) and add this to the result of the recursive call x times y_1. Writing out this recursion would give us the same running time as above, $\Theta(n^2)$.

But this reveals how we might improve on the algorithm. Since we know that MergeSort improves on the quadratic-time sorting algorithm by first dividing the input in half, why not try a similar idea for multiplication? The main difference is that here we have *two* arrays to split in half instead of just one.

Say we have input integers x and y to multiply, and their digits are stored in arrays X and Y, each of length n. Write $m = \lfloor n/2 \rfloor$, and then split the numbers in "half" as follows: Let $X_0 = X[0..m-1]$ and $X_1 = X[m..n-1]$. These are two arrays of digits, but they also represent (big) integers, so write x_0 and x_1 for the two integers that they represent. We can also define Y_0, Y_1, y_0, and y_1 in the same way. The mathematical relationship between the integers is:

- $x = x_0 + B^m x_1$
- $y = y_0 + B^m y_1$

So you see that although the arrays are split in half, the numbers aren't really divided by 2, but by B^m. This means for example that each of x_0 and x_1 are much closer to $\sqrt{x}$ than they are to $\frac{x}{2}$.

All these formulas are making me dizzy. Let's look at a concrete example. If we want to multiply 7407 by 2915, like in the example before, the splitting gives us all of the following:

```
  Integers    |  Array representation
--------------|------------------------
  x  = 7407   |   X = [7, 0, 4, 7]
  y  = 2915   |   Y = [5, 1, 9, 2]
x0 =    07    |  X0 = [7, 0]
x1 =    74    |  X1 = [4, 7]
y0 =    15    |  Y0 = [5, 1]
y1 =    29    |  Y1 = [9, 2]
```

So the question is, how can we multiply 7407 by 2915 using some recursive calls on the integers 7, 74, 15, and 29? It helps to write it out:

$7407 \cdot 2915 = (7 + 74 \cdot 100)(15 + 29 \cdot 100) = 7 \cdot 15 + 7 \cdot 29 \cdot 100 + 74 \cdot 15 \cdot 100 + 74 \cdot 29 \cdot 10000$

Now of course multiplying by any power of 10 (since we are using base $B = 10$) is easy (linear time). So all we really need to do is compute the four products

- $7 \cdot 15 = 105$
- $7 \cdot 29 = 203$
- $74 \cdot 15 = 1110$
- $74 \cdot 29 = 2146$

Now we add these up, shifted as appropriate by powers of 10:

```
              1 0 5
          2 0 3
        1 1 1 0
+   2 1 4 6
-------------------
    2 1 5 9 1 4 0 5
```

The same answer as before. Great! But it sure seems like a lot of work. Do we really save any time?

Well, more generally, the approach here is to multiply the four products x_0y_0, x_0y_1, x_1y_0, and x_1y_1 (four recursive calls), and then add them up (linear time). We can describe this with a recurrence:

- $T(n) = 1$ if $n = 1$
- $T(n) = n + 4T(n/2)$ if $n \geq 2$

We haven't seen a recurrence quite like this before, but let's try solving it with our standard technique:

$T(n) = n + 4T(n/2) = 3n + 16T(n/4) = 7n + 64T(n/8) = \cdots$

See the pattern? It's a little trickier, but we just have to recognize the three sequences 1,3,7,15,... and 4,16,64,256,... and 2,4,8,16,... Think about it a little and we get the general pattern:

$T(n) = (2^i - 1)n + 4^i T(n/2^i)$

Figuring out the base case is nothing new; we solve $n/2^i = 1$ for i to get $i = \lg n$. Now plug this back in and we have $T(n) = (n-1)n + 4^{\lg n}$. The second part of this looks a little odd, but we know that $4 = 2^2$, so we can rewrite $4^{\lg n} = 2^{2 \lg n} = n^2$. Therefore $T(n) = n(2n-1) \in \Theta(n^2)$.

Well this is a little upsetting. We went through all this effort with dividing the numbers in "half" and then making this cool recursive method and... we get exactly the same asymptotic cost as before! Are you ready to give up and declare that quadratic-time is the best possible?

3.1 Karatsuba Algorithm

Luckily someone wasn't ready to give up. And his name was... Gauss. Carl Friedrich Gauss, that is. Pretty famous for a mathematician — the Germans even put him on one of their bills! Gauss's observation was actually about multiplying complex numbers, but it's exactly what we need for this problem. It just amounts to basic algebra:

$$(x_0 + x_1 B^m)(y_0 + y_1 B^m) = x_0 y_0 + x_1 y_1 B^{2m} + ((x_0 + x_1)(y_0 + y_1) - x_0 y_0 - x_1 y_1) B^m$$

In the 1950s, a Russian guy named Karatsuba realized that this bunch of symbols is actually really useful — it means that we can multiply numbers faster!

Not buying it? Well, it's a three step process:

1. Compute two sums: $u = x_0 + x_1$ and $v = y_0 + y_1$.
2. Compute three m-digit products: $x_0 y_0$, $x_1 y_1$, and uv.
3. Sum them up and multiply by powers of B to get the answer: $xy = x_0 y_0 + x_1 y_1 B^{2m} + (uv - x_0 y_0 - x_1 y_1) B^m$

Let's go back to our favorite example of 7407 times 2915 to see how that works:

```
    x = 7407 = 7 + 74*100
    y = 2915 = 15 + 29*100
    u = x0 + x1 =  7 + 74 = 81
    v = y0 + y1 = 15 + 29 = 44
x0*y0 =  7*15 =  105
x1*x1 = 74*29 = 2146
 u*v  = 81*44 = 3564
 x*y  = 105 + 2146*10000 + (3564 - 105 - 2146)*100
      =          105
        +    1313
        + 2146
      =   21591405
```

The same answer again! Can you see why this is going to be a better algorithm? Let's write it out formally.

karatsubaMul

Input: Integers x and y, stored in arrays X and Y of exactly n digits each. Each digit is less than B.

Output: The integer xy, stored in array Array A of length at most $2n$

```
n = len(X)
if n <= 3:
    return smul(X, Y, B)
else:
    m = n // 2
    A = zero-filled array of length (2*n + 1)
    X0, X1 = X[0 : m], X[m : n]
    Y0, Y1 = Y[0 : m], Y[m : n]
    U = add(X1, X0, B)
    V = add(Y1, Y0, B)
    P0 = kmul(X0, Y0, B)
    P1 = kmul(X1, Y1, B)
    P2 = kmul(U, V, B)
    A[0 : 2*m] = P0
    A[2*m : 2*n] = P1
    A[m : 2*n+1] = add(A[m : 2*n], P2, B)
    A[m : 2*n+1] = sub(A[m : 2*n+1], P0, B)
    A[m : 2*n+1] = sub(A[m : 2*n+1], P1, B)
    return A[0 : 2*n]
```

This looks like a pretty long and complicated algorithm. But it's not too bad; really it's just the three steps that we had before, written out in their entirety.

For the analysis, look at the recursive case and think about the cost of each line. Other than the recursive calls, every line has running time $\Theta(n)$. And the three recursive calls are each on integers with about $n/2$ digits. So, with simplifications, the recurrence we need to solve looks like

- $T(n) = 1$ if $n = 1$
- $T(n) = n + 3T(n/2)$ if $n \geq 2$

Very similar to the one before, except *crucially* the coefficient of the $T(n/2)$ term is now 3 instead of 4. Think back to MergeSort; the recurrence for that had a very similar form, except the coefficient was 2 instead of 3. So when the coefficient is 4 we got $\Theta(n^2)$ and when it was 2 we got $\Theta(n \log n)$. What's left in between for the coefficient of 3 that we have here? Let's investigate:

$T(n) = n + 3T(n/2) = 5n/2 + 9T(n/4) = 19n/4 + 27T(n/8) = 65n/8 + 81T(n/16)$

The coefficients of n are starting to get pretty messy. There's a few different ways we could figure out the general pattern, including typing in the sequence of numerators to the online encyclopedia of integer sequences and seeing what pops out.

Instead, I'll tell you the trick: take $2n$ out every time. This gives

$T(n) = 3n - 2n + 3T(n/2) = 9n/2 - 2n + 9T(n/4) = 27n/4 - 2n + 27T(n/8)$

And voila! The pattern pops right out. It has to do with powers of 2 and powers of 3:

$T(n) = 3^i n/2^{i-1} - 2n + 3^i T(n/2^i)$

This is a little hairy looking, but it's nothing we can't handle. As usual, the first thing to do is figure out how many recursive calls to get to the base case. This looks like what we're used to: $n/2^i \leq 1$, so $i \geq \lg n$.

Plugging this in is going to give us some $3^{\lg n}$ coefficients popping up. (Remember before that we saw $4^{\lg n}$? It's not a coincidence!) We can simplify this as follows, using what we know about exponents and logarithms:

$3^{\lg n} = (2^{\lg 3})^{\lg n} = (2^{\lg n})^{\lg 3} = n^{\lg 3}$

Why is this simpler? Well, $\lg 3$ is just a constant! We are used to seeing integers in the exponent of n, corresponding to linear time, quadratic time, cubic, etc., and $\lg 3$ is just another constant wedged in between 1 and 2. If you must know, it's approximately 1.585. So it's still not quite as simple as we might want, but it looks a bit more sane. Now we can plug the $i = \lg n$ in everywhere to get

$T(n) = 3^{\lg n} n/2^{\lg n - 1} - 2n + 3^{\lg n} T(1) = 3n^{\lg 3} - 2n \in \Theta(n^{\lg 3})$

In case looking at "lg 3" makes your head spin, we can also say that T(n) is $O(n^{1.59})$, which is what you'll often see written.

This is exciting! We have a faster algorithm for multiplication, and its running time looks like nothing else we've seen, but fits somewhere between $\Theta(n \log n)$ and $\Theta(n^2)$. Karatsuba came up with this algorithm in 1958, and as it slowly trickled out from the Soviet press, computer scientists around the world were shocked that multiplication better than quadratic time was possible. Did it shock you too?

3.2 Even Better than Karatsuba

A running time like $\Theta(n^{\lg 3})$ should be unsatisfying. It's not a "clean" function like n or n^2 or even n log n. So it's hard to believe that Karatsuba's algorithm is really as good as we could possibly do.

In fact, it's not. A few years after Karatsuba shattered the quadratic barrier, another Russian named Toom came up with an algorithm along similar lines but that divides each input integer into three equal parts instead of two. In fact, he generalized to splitting into any number k of equal parts. Then a Ph.D. student at Harvard named Cook figured out how to analyze this crazy *set* of algorithms.

This is now referred to as the *Toom-Cook* algorithm and the running times for any k look like $\Theta(n^c)$ where c is some number between 1 and 2. In fact, as k increases (splitting into more parts), c decreases down towards 1 (but it never quite gets there!). In practice, the "overhead" of the algorithm as k gets larger makes it unusable except when k=3.

So is Toom-Cook the best we can do? Again, there is something better! There is an algorithm that you might have heard of called the *Fast Fourier Transform*, or FFT, which turns out to be one of the most important numerical algorithms ever. It's worst-case runtime is just like MergeSort, $\Theta(n \log n)$.

Less than a decade after Toom published his algorithm, two Germans named Schoenhage and Strassen came up with a way to use the FFT to do integer multiplication! Unfortunately, it's not quite as fast as the FFT, but it's really close: the worst case is $\Theta(n(\log n)(\log\log n))$. Now "log log n" is not a function we've seen before, but as you can imagine it grows really really slowly. In fact, for all practical purposes in this algorithm the "log log n" contributes a factor of 3 to the cost of the algorithm.

The really cool thing is that *all of these algorithms actually get used to multiply big integers*! You already know one (really important) application of multiple-precision arithmetic, namely the RSA algorithm. There are of course lots of other uses of big integers as well, and when they grow to more than a few hundred or thousand bits, algorithms like Karatsuba's start to become faster than the standard method. Actually, all three of the algorithms mentioned (Karatsuba, Toom-Cook, Schoenhage-Strassen) get used in some range of sizes for big integer multiplications.

There's even a faster one — in theory — developed a few years ago by Martin Furer, with worst-case cost $\Theta(n \log n 2^{\log^* n})$ that involves something called the *iterated logarithm* which we won't talk about. This is actually slightly better *asymptotically* than Schoenhage-Strassen's algorithm, but the hidden constant is too large to make it useful in practice.

Still no one has found an integer multiplication algorithm that costs n log n, although they are really really close. The main thing I want you to take away from this discussion is **there's a lot out there in algorithm development**! The world of sorting seems simple and solved, but most problems get messy and have lots of interesting opportunities for improvements, both in theory and in practice.

4 Master Method for Solving Recurrences

We've seen and solved a lot of recurrence relations by now. For all of them the base case is the same: $T(1) = 1$. The difference is of course in the recursive case. Let's see what kind of recurrences we have solved so far:

Algorithm	Recurrence	Asymptotic big-Θ
BinarySearch	$1 + T(n/2)$	log n
LinearSearch	$1 + T(n-1)$	n
MergeSort (space)	$n + T(n/2)$	n
MergeSort (time)	$n + 2T(n/2)$	n log n
KaratsubaMul	$n + 3T(n/2)$	$n^{\lg 3}$
SelectionSort	$n + T(n-1)$	n^2
StandardMul	$n + 4T(n/2)$	n^2

Our standard method to solve recurrences by writing them out, noticing the pattern, etc., is pretty useful, and it reveals something about the structure of the algorithm too. But after seeing the same patterns show up time and time again, this starts to get tedious. Could you generalize any of the patterns?

Well, the good news is that you don't have to; someone else has done it for you. These so-called "master methods" are now available for your use because you have *mastered* the standard way of solving recurrences. (Okay, that's not really why they're called master methods, but just go with it.)

The first one is a simplified version of what you will find in your book and online, and the second one is specially created by your generous instructor.

For both of these, we write the *non-recursive* part of the recurrence — that is, the part that doesn't involve T(...) — as $f(n) = n^c(\log n)^d$ for some non-negative constants c and d. Observe that in every case above, c is either 0 or 1, and d is always 0. But of course more complicated situations will arise in the future, and it's nice to be prepared.

Master Method A

Suppose $f(n) = n^c(\log n)^d$ for non-negative constants c and d, and $T(n) = aT(n/b) + f(n)$ for all $n \geq 2$, where a is a positive integer and $b > 1$.

Write $e = \log_b a = (\lg a)/(\lg b)$, which must be at least 0.

Then there are three cases to consider:

1) $\mathbf{c = e}$. Then $T(n) \in \Theta(f(n)\log n) = \Theta(n^c(\log n)^{d+1})$.
2) $\mathbf{c < e}$. Then $T(n) \in \Theta(n^e) = \Theta(n^{\log_b a})$.
3) $\mathbf{c > e}$. Then $T(n) \in \Theta(f(n)) = \Theta(n^c(\log n)^d)$.

We won't prove why this works, but the basic idea is that, looking at the recursion tree, either every level has the same cost (case 1, where we multiply by the number of levels, $\log n$), or the number of leaves at the bottom level dominates the cost (case 2), or the very first call dominates the whole cost (case 1).

This basically covers all the "divide-and-conquer" type algorithms that we have seen, and most of the ones that we ever will see. But it doesn't cover some of the other recursive algorithms, like SelectionSort above. For that we need the following, for recurrences that subtract from n rather than divide it in the recursive term:

Master Method B

Suppose $f(n) = n^c(\log n)^d$ for non-negative constants c and d, and $T(n) = aT(n-b) + f(n)$ for all $n \geq 2$, where a and b are both positive integers.

Then there are two cases to consider:

1) $\mathbf{a = 1}$. Then $T(n) \in \Theta(nf(n)) = \Theta(n^{c+1}(\log n)^d)$.
2) $\mathbf{a > 1}$. Then $T(n) \in \Theta(e^n)$, where e is the positive constant $a^{1/b}$, which must be greater than 1.

Things to notice here are first of all that the value of b doesn't matter in the asymptotic analysis, and secondly that if there is more than one recursive call, (i.e. $a > 1$), we end up with an exponential-time algorithm.

This should greatly simplify the process of going from a recurrence to an asymptotic growth rate. The process no longer requires any great intuition or "magic" steps. All we have to do is look at the recursive case, and try to match it to one of the cases of one of the master methods.

It is worth going through the examples in the table at the beginning of this section and seeing how to prove the big-Θ bound of each one using a master method.

5 Matrix Multiplication

Matrices are extremely important in computation. Every time you see an 3-D object move around on your screen, or Netflix recommends a movie for you to watch, the backbone of the operation is computations with (big) matrices.

Now it turns out that one of the most basic things we want to do with matrices is multiply them together. Let's review how this works.

The *dimensions* of a matrix are the number of rows and columns. So a 5x7 matrix has 5 rows and 7 columns, for a total of 35 entries. You should also remember that the *dot product* of any two vectors (which must have the same length) is the sum of the corresponding products in those vectors. We can use these to define the product of two matrices, which consists of all the dot products of any row in the first matrix with any column in the second matrix.

Here's an example. Say we want to multiply the following 4x3 and 3x2 matrices:

$$\underset{A}{\begin{bmatrix} 7 & 1 & 2 \\ 6 & 2 & 8 \\ 9 & 6 & 3 \\ 1 & 1 & 4 \end{bmatrix}} \quad \underset{B}{\begin{bmatrix} 2 & 0 \\ 6 & 3 \\ 4 & 3 \end{bmatrix}}$$

This matrix product will be a 4x2 matrix containing the 8 dot products of every row in A with every column in B:

$$\begin{bmatrix} 7*2+1*6+2*4 & 7*0+1*3+2*3 \\ 6*2+2*6+8*4 & 6*0+2*3+8*3 \\ 9*2+6*6+3*4 & 9*0+6*3+3*3 \\ 1*2+1*6+4*4 & 1*0+1*3+4*3 \end{bmatrix} = \begin{bmatrix} 28 & 9 \\ 56 & 30 \\ 66 & 27 \\ 24 & 15 \end{bmatrix}$$

In general, we might be multiplying a $m \times k$ matrix by a $k \times n$ matrix, to produce a $m \times n$ matrix product. Things to notice:

- The *middle dimensions* (3 in the example, k in general) must match up exactly, so that the dot products have matching lengths.
- Each entry in the product matrix requires exactly $\Theta(k)$ operations and exactly k multiplications in particular.

We could write out this algorithm formally, but I don't think we really need to. Since there are exactly mn entries in the product matrix, and each of them costs $\Theta(k)$ operations to compute, the total cost is $\Theta(mkn)$.

Here of course we are implicitly assuming that all the elements in either array are single-precision integers. If they are not, then the cost of the matrix product will just be multiplied by the cost of whatever integer multiplication algorithm we use. But the two don't really affect each other at all, so while we're concentrating on the matrix operations it's fine to assume single-precision entries.

Notice that mkn is also the *exact* number of multiplications required to compute the matrix product. This will be important later. Also notice that, if the dimensions are all n, i.e. we are multiplying square matrices, then the total cost is n^3 multiplications.

5.1 Strassen Algorithm

For a long time no one thought you could do better than n^3 operations for multiplying matrices. The multiplications seem to be involved in the very definition of the problem! But then Divide and Conquer strikes again!

Dividing a matrix "in half" is a little more involved than dividing an array or a polynomial, because matrices are two-dimensional objects. If we divide a matrix in half in one dimension only, then the resulting sub-problems won't have the same *shape* as the original one.

The solution is two-fold: First, we only multiply matrices that are *square*, meaning that the row and column dimensions are the same. And second, we assume both dimensions are even, so the matrices can be evenly divided in quarters. So we are just talking about the product of two $n \times n$ matrices, where n is evenly divisible by 2. Just like with integers, we can "pad" the matrices with zeroes to get the right dimensions.

$$\left[\begin{array}{c|c} S & T \\ \hline U & V \end{array}\right] \left[\begin{array}{c|c} W & X \\ \hline Y & Z \end{array}\right] = \left[\begin{array}{c|c} SW+TY & SX+TZ \\ \hline UW+VY & UX+VZ \end{array}\right]$$

Now remember that each of these submatrices $S, T, \ldots, Z$ is just a matrix with dimensions $\frac{n}{2} \times \frac{n}{2}$. So doing the product this way, you can count up to confirm that we will have to perform 8 multiplications and 4 additions of $\frac{n}{2} \times \frac{n}{2}$ matrices. Each matrix addition requires $n^2/4 = \Theta(n^2)$ time, so the total cost is given by the recurrence:

- $T(n) = 1$ when $n = 1$
- $T(n) = n^2 + 8T(n/2)$ when $n \geq 2$

Now we can plug this into Master Method A to conclude that $T(n) \in \Theta(n^3)$. Darn it! Once again, the straightforward divide-and-conquer approach didn't gain us anything!

But just like with Karatsuba's algorithm, there is a *very clever* way to add up some of the blocks before multiplying and then add and subtract the multiples so that the number of recursive calls is reduced by one — in this case from 8 to 7.

If you're interested in this stuff, I recommend reading the description in CLRS, Section 4.2, where they try to explain how Strassen came up with this algorithm. (Really, the answer is "Trying lots of stuff, seeing what works, learning from past mistakes, and being persistent". This is how nearly every difficult problem gets solved!)

The way Strassen's algorithm works is to first compute the **seven** $\frac{n}{2} \times \frac{n}{2}$ matrix products:

$$\begin{aligned}
P_1 &= S(X - Z) \\
P_2 &= (S + T)Z \\
P_3 &= (U + V)W \\
P_4 &= V(Y - W) \\
P_5 &= (S + V)(W + Z) \\
P_6 &= (T - V)(Y + Z) \\
P_7 &= (S - U)(W + X)
\end{aligned}$$

After computing these, the four blocks of the matrix product just require some additions and subtractions:

$$\left[\begin{array}{c|c} S & T \\ \hline U & V \end{array}\right] \left[\begin{array}{c|c} W & X \\ \hline Y & Z \end{array}\right] = \left[\begin{array}{c|c} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ \hline P_3 + P_4 & P_1 + P_5 - P_3 - P_7 \end{array}\right]$$

I'm not going to write out this algorithm. But hopefully you can see that it would work recursively, making each of the **seven** computations of the P_i's into recursive calls. All in all, each recursive call requires 10 additions/subtractions, then 7 (recursive) multiplications, and then 8 more additions/subtractions, all on $\frac{n}{2} \times \frac{n}{2}$ matrices. Since each addition/subtraction costs $\Theta(n^2)$, the total cost is given by the recursion

$$T(n) = \begin{cases} 1, & n < 1 \\ n^2 + 7T(n/2), & n \geq 2 \end{cases}$$

Applying the Master theorem here tells us that $T(n) \in \Theta(n^{\lg 7})$, which is $O(n^{2.81})$. Interestingly, when this algorithm was first presented, no one ever thought it would be used, because the "hidden constant" in the computation is so much larger than for the standard $\Theta(n^3)$ algorithm. But alas, the power of asymptotic analysis is that, as time goes on and computers get more powerful, problems get bigger. And when problems get bigger, eventually the better asymptotic algorithm wins. Today Strassen's algorithm is used in practice for very large matrix computations (think more than a million entries in each matrix).

But although the importance of this discovery — both in theory and in practice — cannot be overstated, it is again somewhat "unsatisfying" and doesn't seem to be the best possible algorithm.

In fact, asymptotically faster algorithms have been developed, and the fastest among them was invented by Coppersmith and Winograd a few decades ago; it has worst-case running time $O(n^{2.38})$. Actually, the running time of this algorithm has gone down twice in the last two years — even though there haven't really been any new *algorithmic* ideas! Instead, the analysis gets so difficult that people have been using computer programs to do the analysis for them, and coming up with slightly improved exponents on the worst-case cost. Yes, there are actually algorithms to do the analysis of other algorithms!

(And no, none of the asymptotic improvements to Strassen's algorithm get used in practice, yet.)

6 Computing Fibonacci Numbers

Remember our old pal the Fibonacci sequence? It's a little off-topic for this unit on "multiplication", but useful to introduce some techniques that will be useful to solve an interesting problem on matrix multiplication. Remember that the Fibonacci numbers are defined by the recurrence relation:

- $f_i = i$ when $0 \le i \le 1$
- $f_i = f_{i-1} + f_{i-2}$ when $i \ge 2$

Now here's a simple recursive algorithm to compute any Fibonacci number:

Fibonacci numbers, simple recursive version

Input: Non-negative integer n

Output: f_n

```
def fib(n):
    if n <= 1:
        return n
    else:
        return fib(n-1) + fib(n-2)
```

Seems simple enough. Now let's analyze it. The running time is given by the recurrence

$$T(n) = \begin{cases} 1, & n \le 1 \\ 1 + T(n-1) + T(n-2), & n \ge 2 \end{cases}$$

Does this look familiar? It's just like the actual Fibonacci recurrence except for that pesky "+ 1" part. Typing the first few values into the Online Encyclopedia of Integer Sequences will give you the general formula, which you can confirm with a proof by induction: $T(n) = 2f_{n+1} - 1$.

And this is bad! We already showed before that $f_n < 2^n$ for any value of n. And you can follow along the same steps of that proof to show that $f_n \ge 2^{n/2} - 1$ as well. Therefore the cost of this algorithm is actually exponential in the *value* of the input n.

If you coded this algorithm up, you probably wouldn't be able to get as far as computing f_{50}, even if you had the most powerful computer in the world! To see what the problem is, let's look at the tree of recursive calls for computing fib(6):

6.1 Memoizing Fibonacci

Remember that a key tool to improving algorithms is to look for repeated or unnecessary computations. There is certainly a lot of repetition here: just to compute fib(6), for example, there are 5 separate calls to fib(2), each of which *separately* calls fib(1) and fib(0).

The idea of *memoization* is just to remember the return value each time a function is called, storing them in a big table. The "table" could be a simple dynamic-sized array (like the vector class), or a hash table, or a red-black tree... you get the idea.

The memoized version of the Fibonacci function would look like the following, where T is the *globally-defined* table.

Fibonacci numbers, memoized version: fibmemo(n)

Input: Non-negative integer n

Output: f_n

```
fib_table = {} # an empty hash table
def fib_memo(n):
    if n not in fib_table:
        if n <= 1:
            return n
        else:
            fib_table[n] = fib_memo(n-1) + fib_memo(n-2)
    return fib_table[n]
```

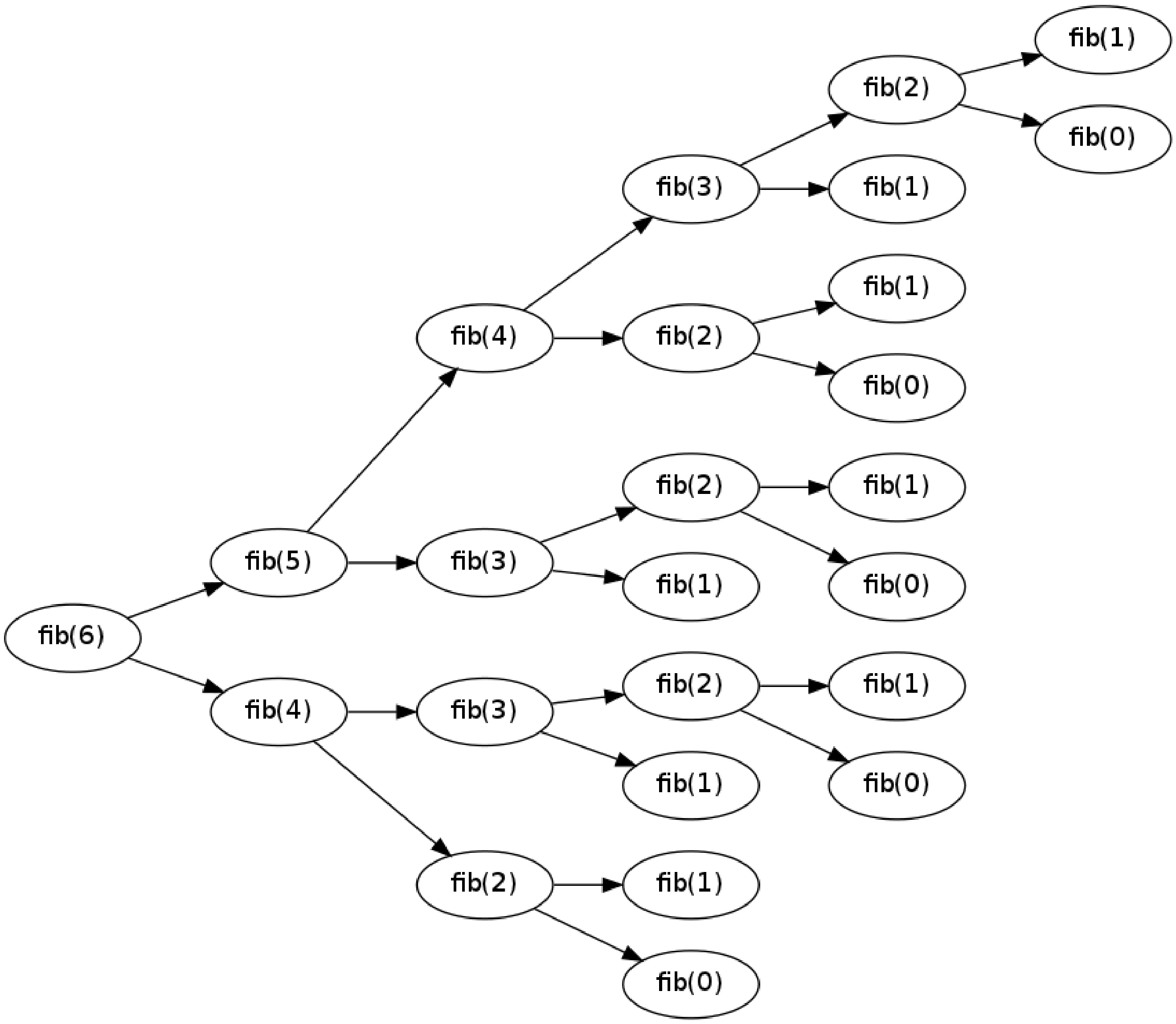

Figure 1: Recursion tree for fib(6)

Now look back and compare this to the non-memoized version. Notice that lines 2 and 3 correspond *exactly* to the original recursive function, and the rest is just about doing the table lookup. So we can apply memoization to any recursive function!

And at least for Fibonacci, this technique *greatly* reduces the running time of the function. Here is the tree of recursive calls for fibmemo(6):

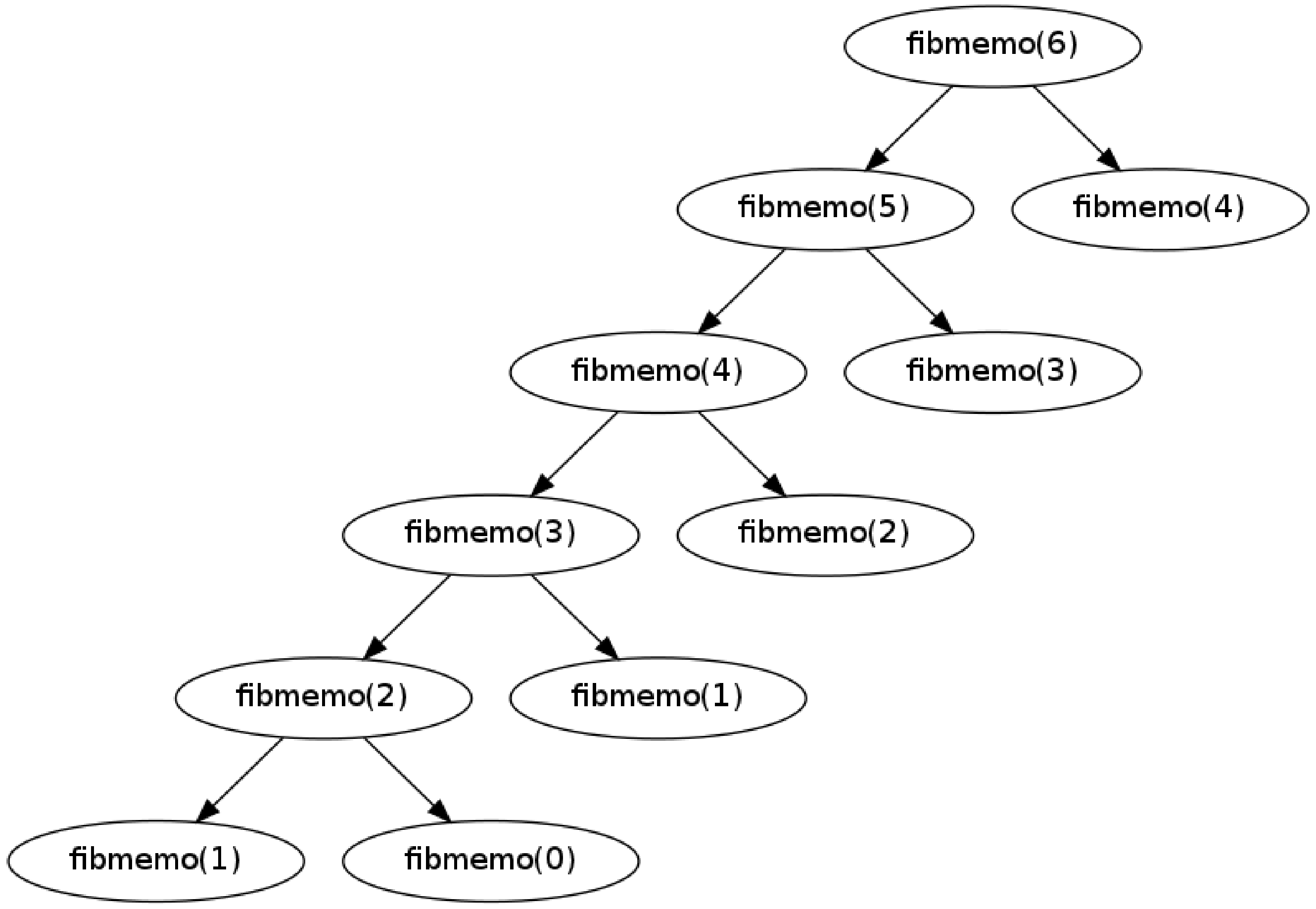

Figure 2: Recursion tree for fibmemo(6)

To analyze this is a little bit different than most analysis we've seen. But it's not too hard to follow. First, assume that table lookups take constant time. Then notice two facts:

1) The worst-case cost of fibmemo(n) is just a constant number of primitive operations, plus a constant number of recursive calls.
2) The only recursive calls required to compute fibmemo(n) are fibmemo(n−1), fibmemo(n−2), down to fibmemo(0).

Therefore, since each recursive call is only fully evaluated at most once, the total worst-case cost is $\Theta(n)$ calls each with cost $\Theta(1)$, for a total of $\Theta(n)$ primitive operations. Much improved from the original version!

Actually, there's a much simpler way to calculate Fibonacci numbers using a simple loop and keeping track of the previous two values in the sequence at every step, which also achieves $\Theta(n)$ time.

But the power of memoization is its *generality*. It can be applied to a wide variety of problems, not just this particular one. For any problem where we see the same recursive calls showing up again and again, memoization is an easy way to speed it up, often by a considerable margin. In fact, some programming languages even have memoization built in! (Not C++ or Java, unfortunately. Can you think of why it would be a bad idea to memoize every function call?)

7 Matrix Chain Multiplication

Now let's return to our theme of multiplication. We want to look at matrix multiplication, but this time suppose we have a whole list of matrices and want to multiply all n of them together:

$A_1 \cdot A_2 \cdot A_3 \cdots A_n$.

All we know so far is how to compute a product of two matrices. And, at least for now, we'll just use the standard algorithm to do it (not Strassen's). So the only thing to do really is to figure out how to break this big product of n matrices into $n-1$ products of 2 matrices each. This ordering of multiplications could be specified for example by putting parentheses everywhere.

Now there are two things to know about matrix multiplication. The first is, they're associative. So whether we do $(A_1A_2)A_3$ or $A_1(A_2A_3)$, the answer will be the same. Great!

But matrix multiplication is definitely *not* commutative. For example, $A_1A_2A_3$ will not (in general) be equal to $A_2A_1A_3$. This has to do with the rules of multiplication, that the "inner" dimensions must match up. Mixing up the order in this way would mix up the dimensions, and wouldn't make any sense at all. So when we say "ordering" of the products, we're just talking about which multiplications to do first, second, and so on (which we can mess around with safely), and *not* about which matrix comes first in the products (which we definitely can't mess around with).

To see what we're talking about, consider for example if we wanted to multiply XYZ where

- X is a 10x2 matrix.
- Y is a 2x8 matrix.
- Z is an 8x3 matrix.

Therefore the whole product is a 10x3 matrix. But we have two choices for the parenthesization:

1) **Compute X times Y first**. This corresponds to the parenthesization $(XY)Z$. The number of multiplications to compute X times Y is $10 * 2 * 8 = 160$, and the result XY will be a 10x8 matrix. Then the number of mults to compute XY times Z is $10 * 8 * 3 = 240$. So the total number of mults in this parenthesization is $160 + 240 = 400$.

2) **Compute Y times Z first**. This corresponds to the parenthesization $X(YZ)$. The number of multiplications to compute Y times Z is $2 * 8 * 3 = 48$, and the result XY will be a 2x3 matrix. Then the number of mults to compute X times YZ is $10 * 2 * 3 = 60$. So the total number of mults in this parenthesization is $48 + 60 = 108$.

What a difference! When we generalize this out to the product of n matrices, the difference in the cost between different parenthesizations can actually be exponential in n — so it could mean the difference between being able to do the computation, ever, on any computer, or being able to do it on your cell phone.

7.1 Computing the minimal mults

As a first step towards computing the best parenthesization, let's tackle the (hopefully) easier problem of computing what the least number of multiplications is to find the product of n matrices A_1 through A_n. What you should realize is that the contents of these matrices actually doesn't matter — all we need are their dimensions.

We'll say the dimensions are stored in an array D of size $n+1$. (We only need $n+1$ because all the "inner dimensions" must match up.) For each i from 1 to n, A_i will be a $D[i-1] \times D[i]$ matrix. In particular, the whole product is a $D[0] \times D[n]$ matrix.

There's an easy way to figure out the minimal mults if we use recursion: we just need to figure out what is the *last* multiplication that should be performed (corresponding to the *outermost* parentheses), and then let recursion handle the rest. Here's the algorithm:

Minimal mults, version 1: mm(D)

Input: Dimensions array D of length $n+1$

Output: The least number of element multiplications to compute the matrix chain product

```
def mm(D):
    n = len(D) - 1
    if n == 1:
        return 0
    else:
        fewest = infinity
        for i in range(1, n):
            t = mm(D[0 : i+1]) + D[0]*D[i]*D[n] + mm(D[i : n+1])
            if t < fewest:
                fewest = t
        return fewest
```

Now let's analyze it. It's pretty easy to see that the cost is always $\Theta(n)$ plus the cost of any recursive calls on line 5. So we just have to write down some summations for these recursive calls to get a recurrence for the whole thing:

$$T(n) = \begin{cases} 1, & n = 1 \\ n + \sum_{i=1}^{n-1}(T(i) + T(n-i)), & n \geq 2 \end{cases}$$

This obviously doesn't look like anything we've seen before. But let's try simplifying the recursive case:

$T(n) = n + \sum_{i=1}^{n-1}(T(i) + T(n-i)) = n + 2\sum_{i=1}^{n-1} T(i)$

This is just about re-ordering the second term in each entry of the summation. Now let's extract the $T(n-1)$ parts out of the above:

$T(n) = n + 2\sum_{i=1}^{n-1} T(i) = n + 2\sum_{i=1}^{n-2} T(i) + 2T(n-1)$

Now here's the clever part: From the last simplification, we know that $T(n-1) = n - 1 + 2\sum_{i=1}^{n-2} T(i)$. So now we can do

$T(n) = n + 2\sum)i = 1^{n-2}T(i) + 2T(n-1) = T(n-1) + 1 + 2T(n-1) = 1 + 3T(n-1)$.

Now that looks pretty nice. In fact, it's so nice that we can apply the Master Method B to it and conclude that the running time of this algorithm is $\Theta(3^n)$.

That's really, really slow! Just like with the original Fibonacci function, if we drew out the recursion tree, we would notice the same recursive calls getting computed over and over again. So you shouldn't be surprised at the first idea for an improvement:

7.2 Memoized Minimal Mults

Whenever we notice the same recursive calls being computed again and again, memoization is a general technique that can always be used to avoid some unnecessary computations. Let's try it for the matrix chain multiplication problem:

Minimal mults, memoized version: mmm(D)

Input: Dimensions array D of length $n+1$

Output: The least number of element multiplications to compute the matrix chain product

```
mm_table = {}
def mmm(D):
    n = len(D) - 1
    if D not in mm_table:
        if n == 1:
                mm_table[D] = 0
            else:
                fewest = infinity
                for i in range(1, n):
                    t = mmm(D[0 : i+1]) + D[0]*D[i]*D[n] + mmm(D[i : n+1])
                    if t < fewest:
                        fewest = t
                mm_table[D] = fewest
        return mm_table[D]
```

What's important to recognize is that memoizing this function is exactly the same as memoizing the Fibonacci function (or any other one, for that matter). We just have to create a table to hold all the saved values, add an "if" statement around the original code, and change all the original "return" statements to set the table entry.

The only real tricky part in implementing this is getting the table right. Notice now that the keys for looking up into the table T are now arrays rather than single integers. This is going to affect the choice of data structure for T (a hash table would still work great), as well as the cost of looking things up in the table (which should now be $\Theta(n)$).

Now for the analysis. Like with memoized Fibonacci, we want to ask what the worst-case cost is if any call *not* counting the recursive calls, and then ask what is the total number of recursive calls there could be.

Here the worst case cost of a single call (not counting recursion) isn't too difficult to figure it out. There is a for loop with $n-1$ steps, and two table lookups that each cost $\Theta(n)$. So the total cost is $\Theta(n)$, plus as many recursive calls.

But how many recursive calls with there be in total? Notice that every recursive call is on a contiguous subarray, or "chunk", of the original array D. Counting these up, we have

$D[0..1], D[0..2], D[0..3], \ldots, D[0..n], D[1..2], D[1..3], \ldots, D[n-1..n],$

which is the familiar arithmetic sequence $n + (n-1) + (n-2) + \cdots + 1$. Hopefully you remember (or can figure out) that this sums to $n(n+1)/2$. So there are $O(n^2)$ recursive calls in total.

Putting all this together, we have $O(n^2)$ recursive calls, each costing $O(n)$ for a total cost at most $O(n^3)$. Despite the cosmetic similarity, this is quite an improvement on the original $\Theta(3^n)$ cost!

7.3 Dynamic Programming Solution

Memoization is a fantastic solution to this problem, and it is a sufficiently general solution that it applies to many others as well. However, it has at least three drawbacks:

1) The choice of data structure for the table is going to have a big effect on performance. The best we can really do in general is to say "try a hash table and hope it works", but this requires a lot of trust and assumptions.
2) Our analysis was a bit tricky! There's some sophisticated stuff going on, and we have to unwind it a bit to do the analysis. It would be nice to have a clearer picture of the cost just by looking at the algorithm.
3) Clearly we have to use some memory to avoid the exponential running time of the original algorithm, but memoization can use *too much* extra memory, especially when there is a single global table T and there are a lot of calls to the original function.

Dynamic programming solves these issues by being more explicit about what the table looks like and how it gets filled in. It is more of a "bottom-up" approach to solving the problem, rather than the "top-down" approach of the memoized version. Generally speaking, dynamic programming solutions are harder to come up with than the memoized version, but they run faster and are easier to analyze.

From our analysis of the memoized version, notice that the recursive calls we need to compute mmm(D) all look like mmm(D[i..j]) for some integers i and j that satisfy $0 \le i < j \le n$. This leads to the idea of storing the saved values in a two-dimensional array A, where the entry in the i'th row and j'th column of A will specify the return value of mmm(D[i..j]).

Because of the way things are defined, A[i, j] will be undefined anywhere that $i \geq j$. This means that A will be a $(n+1) \times (n+1)$ matrix, where only the entries in the top-right half are filled in.

So far we are just formalizing the data structure for the memoized version. The tricky part for dynamic programming is getting the *order* right. Because we are working from the bottom-up, without using recursion, we have to be careful that we have the results of all the necessary subproblems "filled in" in the table before moving on.

To figure this out, consider what we need to fill in A[i, j]. This corresponds to mmm(D[i..j]), which will make further recursive calls to mmm(D[i..k]) and mmm(D[k..j]) for every k between i and j. These correspond to the table entries A[i, i+1] up to A[i, j−1] and A[j−1,j] up to A[i−1,j]. Labelling the diagonals with numbers starting with the main diagonal of the table, this corresponds to the following picture:

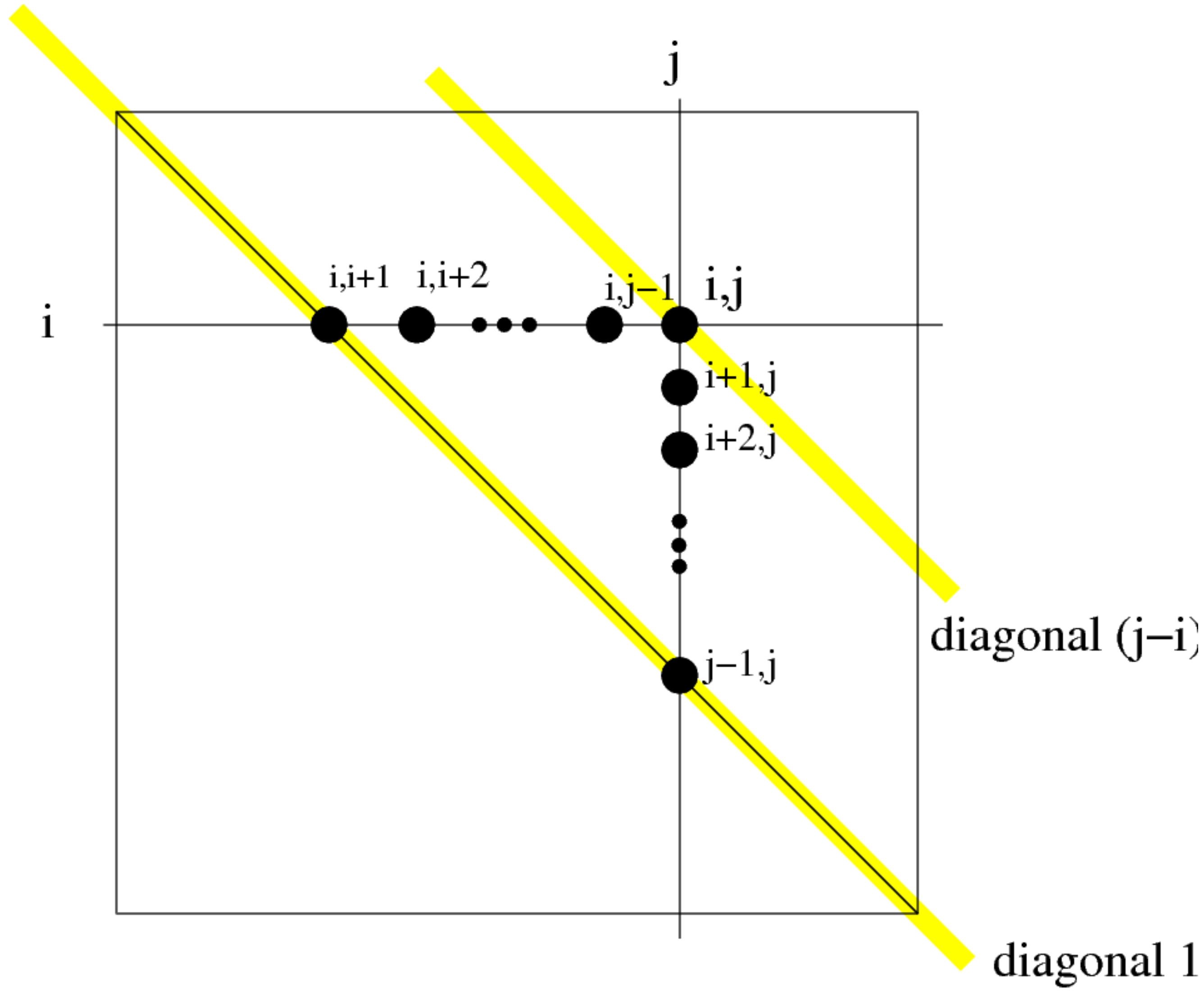

Figure 3: Dynamic programming for minimal mults

For any particular entry in the table, we need all the ones below it and to the left filled in already. More generally, for any entry in the i'th diagonal, we need every entry in every diagonal from 1 to $i-1$ filled in. This gives the ordering we'll use: fill in each diagonal, starting with number 1, until the whole table is filled in. Here's the algorithm:

Minimal mults, dynamic programming version

Input: Dimensions array D of length $n+1$

Output: The least number of element multiplications to compute the matrix chain product

```
def dmm(D):
    n = len(D) - 1
    A = (n+1) by (n+1) array
    for diag in range(1, n+1):
        for row in range(0, n-diag+1):
            col = diag + row
            # This part is just like the original!
            if diag == 1:
                A[row][col] = 0
            else:
                A[row][col] = infinity
                for i in range(row+1, col):
                    t = A[row][i] + D[row]*D[i]*D[col] + A[i][col]
                    if t < A[row][col]:
                        A[row][col] = t
    return A[0][n]
```

It's worth running through an example to see how this works. Try finding the minimal number of multiplications to compute $WXYZ$, where

- W is a 8x5 matrix
- X is a 5x3 matrix
- Y is a 3x4 matrix
- Z is a 4x1 matrix

Therefore $D = [8, 5, 3, 4, 1]$ is the matrix of dimensions, and the table A will be a 5x5 matrix holding all the intermediate values. See if you can follow the algorithm to fill it in. You should get:

	0	1	2	3	4
0		0	120	216	67
1			0	60	27
2				0	12
3					0
4					

This tells us that the minimal number of multiplications is 67, which we can see by unwinding the computation corresponds to the parenthesization $W(X(YZ))$. In practice this information on how to "unwind" the computation in order to figure out the actual ordering of the multiplications to perform would be stored in the table along side the minimal multiplication values. This just means saving the value of i in the inner for loop that corresponds to when the minimal multiple value is set.

Now let's compare the memoized and dynamic programming versions of this problem. The dynamic programming solution is more difficult to formulate, in large part because it requires us to specify the data structure more explicitly, and to choose the ordering carefully to fill in the table. But the advantages to this are that we get a more compact data structure, generally resulting in faster code, and it is easier to see how the algorithm actually works. For example, it is now just a familiar exercise in counting nested loops to see that the cost of this algorithm is $\Theta(n^3)$.