Example 4: Computing $F(n)$ is $O(\lg n)$

We have seen that Fibonacci numbers $F(n)$ grow asymptotically as $O(2^n)$ and $\Omega((\frac{3}{2})^2)$. Moreover, we can compute all the Fibonacci numbers of up n in $O(n)$ time, just by computing all the $F(k)$ iteratively for $k = 0$ up to $k = n$.

Let's ask a slightly different question. How much time do we need to compute $F(n)$ for *some particular* n, say $n = 3421$. Surprisingly, the time we need is $O(\lg n)$.

Here is how we do it. Define a 2×2 matrix for general n:

$$\begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix}$$

Since $F(0) = 0$, $F(1) = 1$, $F(2) = 1$, the matrix is

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

when $n = 1$. Next, verify for yourself that

$$\begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F(n+2) & F(n+1) \\ F(n+1) & F(n) \end{bmatrix}$$

and then prove (by induction) that

$$\begin{bmatrix} F(n+1) & F(n) \\ F(n) & F(n-1) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n, \quad n \geq 1$$

Thus we can compute $F(n)$ in the time it takes us to compute the matrix power to the n. But now note that we can compute powers in $O(\lg n)$ time using recursion:

Algorithm: Power(A, n)
Input: *Square matrix* A *and exponent* $n > 0$
Output: $A^n = A \cdot A \cdot A \cdot \cdots A$

```
if n = 1 then
  return A
else
  B ← Power(A, ⌊n/2⌋)
  if (n%2) = 1 then
    return B · B · A
  else
    return B · B
  end if
end if
```

The recurrence relation is:

$$t(n) = O(1) + t(\lfloor n/2 \rfloor)$$

and the base $t(n) = 1$ if $n = 1$. This recurrence relation is basically the same as what we saw last class for Decimal to Binary conversion. Thus, $t(n)$ is $O(\lg n)$. Wow !

The $O(1)$ represents a worst case (constant) the time it takes to multiply three square matrices (i.e. $B \cdot B \cdot A$) plus any other overhead. Note that, although several multiplications are required to do a matrix product, the time it takes is constant (for a fixed size matrix e.g. 2×2).

Example 2: Mergesort (see GT textbook pages 493-4):

I discussed the Mergesort algorithm at the end of lecture 11. Let's now return to that algorithm. The idea of mergesort is simple. If there is just one number to sort ($n = 1$), then do nothing. Otherwise, partition the n numbers into two sets of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$, sort each of these two sets, and then merge the two sorted sets.

Algorithm: Mergesort(S)
Input: List S
Output: Sorted list

```
if (S.length = 1) then
  return S
else
  mid ← S.length / 2
  S1 ← S.getElements(0,mid-1)
  S2 ← S.getElements(mid,S.length)
  Mergesort(S1)
  Mergesort(S2)
  return Merge(S2,S2,S)
end if
```

Algorithm: Merge($S1, S2, S$)
Input: *Sorted sequences* $S1$ *and* $S2$
Output: *Sorted sequence* S *containing the elements from S1 and S2*

```
while S1 is not empty & S2 is not empty do
  if S1.first < S2.first then
    S.addlast( S1.remove(S1.first))
  else
    S.addlast( S2.remove(S2.first))
  end if
end while
while S1 is not empty do
  S.addlast( S1.remove(S1.first))
end while
while S2 is not empty do
  S.addlast( S2.remove(S2.first))
end while
```

So, for example, suppose we have a list

$$3, 6, 1, 7, 2, 5, 4.$$

We define two lists

$$3, 6, 1 \qquad\qquad 7, 2, 5, 4$$

and sort these

$$1, 3, 6 \qquad\qquad 2, 4, 5, 7$$

and then merge them

$$1, 2, 3, 4, 5, 6, 7.$$

Mergesort is an example of a *divide and conquer* algorithm, namely we divide the problem into pieces, solve the pieces, and then combine the solutions.

The recurrence equation for mergesort is:

$$t(n) = t(\lfloor n/2 \rfloor) + t(\lceil n/2 \rceil) + n$$

or, in the case that n is a power of 2,

$$t(n) = 2t(n/2) + n.$$

It turns out the latter gives the same $O(\ )$ bound as the former, so let's just look at the latter, i.e. we assume $n = 2^k$ or equivalently $k = \lg n$.

By backwards substitution, you can see:

$$\begin{aligned} t(n) &= 2t(n/2) + n \\ &= 2(\ 2t(n/4 + n/2)\ ) + n \\ &= 4t(n/4) + n + n \\ &= 4(2t(n/8) + n/4) + n + n \\ &= 8t(n/8) + n + n + n \\ &= \ldots \\ &= n\ t(1) + n \lg n \end{aligned}$$

which is $O(n\ \lg n)$.

This might not seem so impressive at first glance. But compare it to insertion sort which we saw earlier was $O(n^2)$. To make the comparison, note that $2^{10} = 1024 \approx 1000$ and so $\lg 1000 \approx 10$. Consider the following table:

n	$\lg n$	$n \lg n$	n^2
$10^3 \approx 2^{10}$	10	10^4	10^6
$10^6 \approx 2^{20}$	20	20×10^6	10^{12}
$10^9 \approx 2^{30}$	30	30×10^9	10^{18}
...	...	...	...

For large values of n, there is an astronomical difference between $n \lg n$ and n^2.