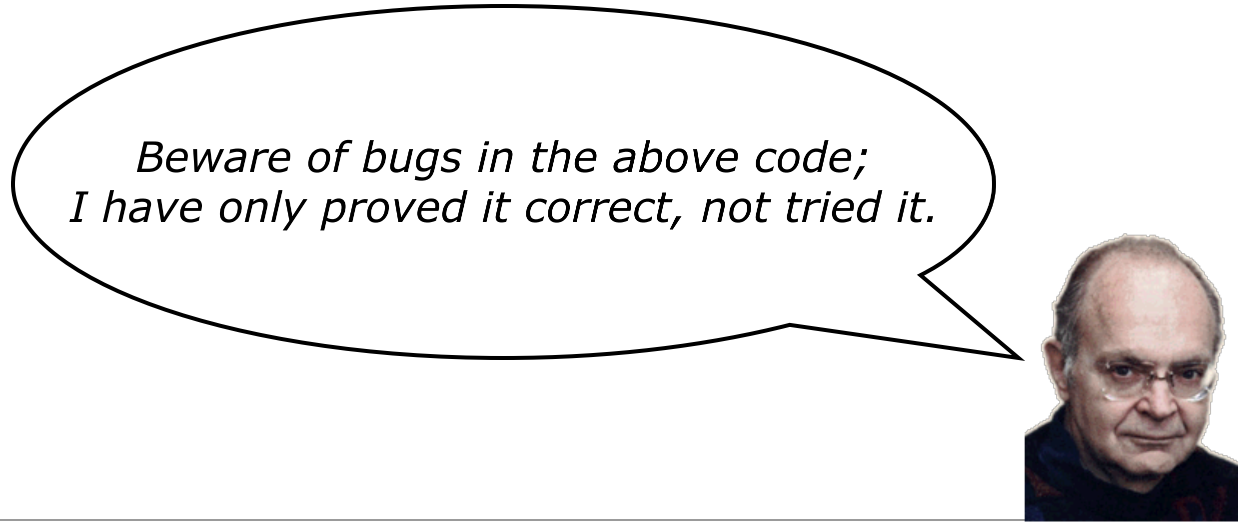

Beware of bugs in the above code;
I have only proved it correct, not tried it.

Theory

Computer Science Theory

- Unlike many other areas, **Computer Science has its own theory.**
- **Theory of Computation**
  What can be computed?
- **Algorithm Complexity Theory**
  How can things be computed more efficiently?
- **Other Theory Examples**
  Artificial Intelligence (search, probability, machine learning)
  Systems (queueing theory)
  Languages (type theory)
  Security (cryptanalysis)
  Databases (logic and relations)

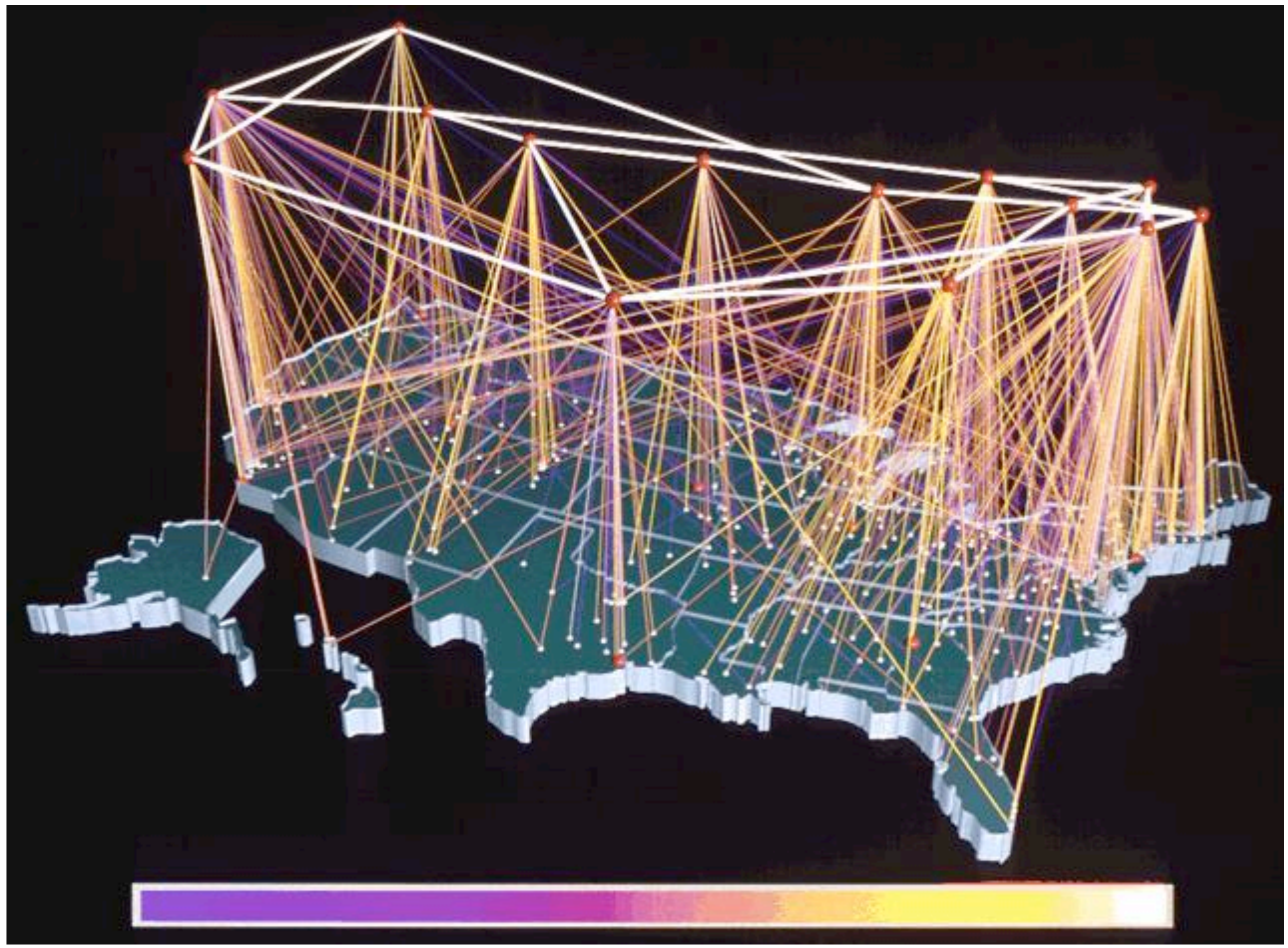
Algorithmic Complexity Theory

- Describe an algorithm.
- Show that this algorithm requires *at least* certain resources to operate (memory, time).
- Show that this algorithm requires *more* or *less* resources than another algorithm.
- **Why is this useful?**

Ma Bell

- Question: how should AT&T route a *minimum length* cable connecting the following cities?

  Des Moines
  Chicago
  Kansas City
  Sioux City
  Denver
  Omaha
  Madison
  Ann Arbor
  Indianapolis
  Springfield



Ma Bell

- You could search all possible routings to find the shortest:

  10! = 3628800

- How about 100 cities?

  100! = a big number

- **AT&T had this problem *all the time.*** Cable length cost them billions of dollars. The US has a lot more than 100 cities.

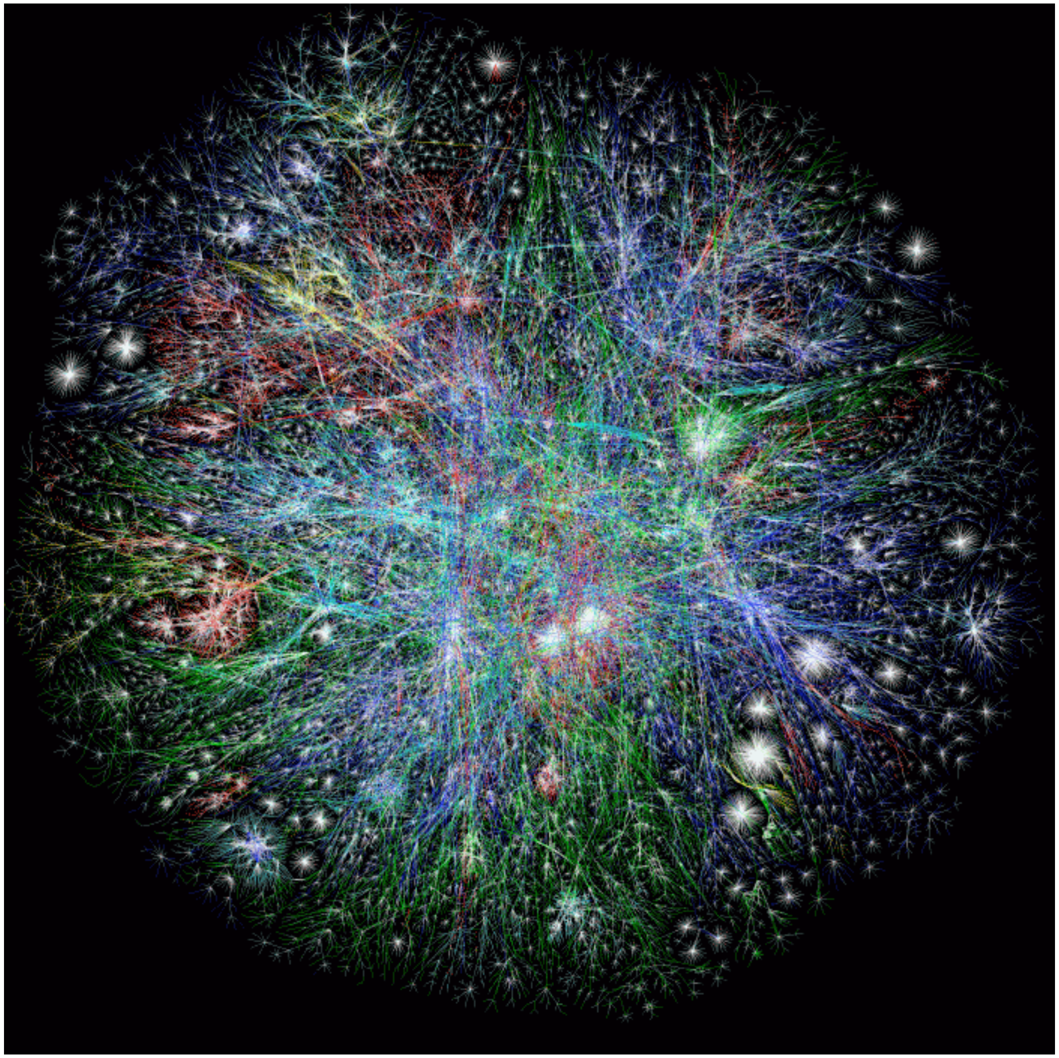
- It was helpful to spend think-time on this.

The Internet

- This problem is getting uglier.

- Network routing
  Polygon reduction (graphics)
  Database algorithms
  Robot swarms
  Cryptography

- Basically **everything in computer science** desperately needs a provably better algorithm.



Fibonacci Sequence

- A basic example.

- Fibonacci(0) = 1

- Fibonacci(1) = 1

- Fibonacci(n) = Fibonacci(n-1) + Fibonacci(n-2)

- How could we write this function?

Fibonacci Sequence

- Like this?

```
Procedure Fibonacci(n):
   if (n<=1):
         return 1                              // here's a computation
   else:
         a := Fibonacci(n-1)                   // a call
         b := Fibonacci(n-2)                   // a call
         return a + b                          // here's a computation
```

- Every time we call this function, *one Fibonacci computation is done,* plus either zero or two other Fibonacci calls (which do more computations!).

Counting

- So to compute Fibonacci(4), we do one computation, and two calls to other Fibonacci functions, which do additional computations each. How many total Fibonacci computations are performed?

```
Procedure Fibonacci(n):
   if (n<=1):
         return 1                                  // here's a computation
   else:
         a := Fibonacci(n-1)                       // a call
         b := Fibonacci(n-2)                       // a call
         return a + b                              // here's a computation
```

Counting

- So to compute Fibonacci(4), we do one computation, and two calls to other Fibonacci functions, which do additional computations each. How many total Fibonacci computations are performed?

  A. Fibonacci(4) *calls* **B.** Fibonacci(3) *and* **C.** Fibonacci(2), then computes B+C
  B. Fibonacci(3) *calls* **D.** Fibonacci(2) *and* **E.** Fibonacci(1), then computes D+E
  C. Fibonacci(2) *calls* **F.** Fibonacci(1) *and* **G.** Fibonacci(0), then computes F+G
  D. Fibonacci(2) *calls* **H.** Fibonacci(1) *and* **I.** Fibonacci(0), then computes H+I
  E. Fibonacci(1) computes to 1
  F. Fibonacci(1) computes to 1
  G. Fibonacci(0) computes to 1
  H. Fibonacci(1) computes to 1
  I. Fibonacci(0) computes to 1

- 9

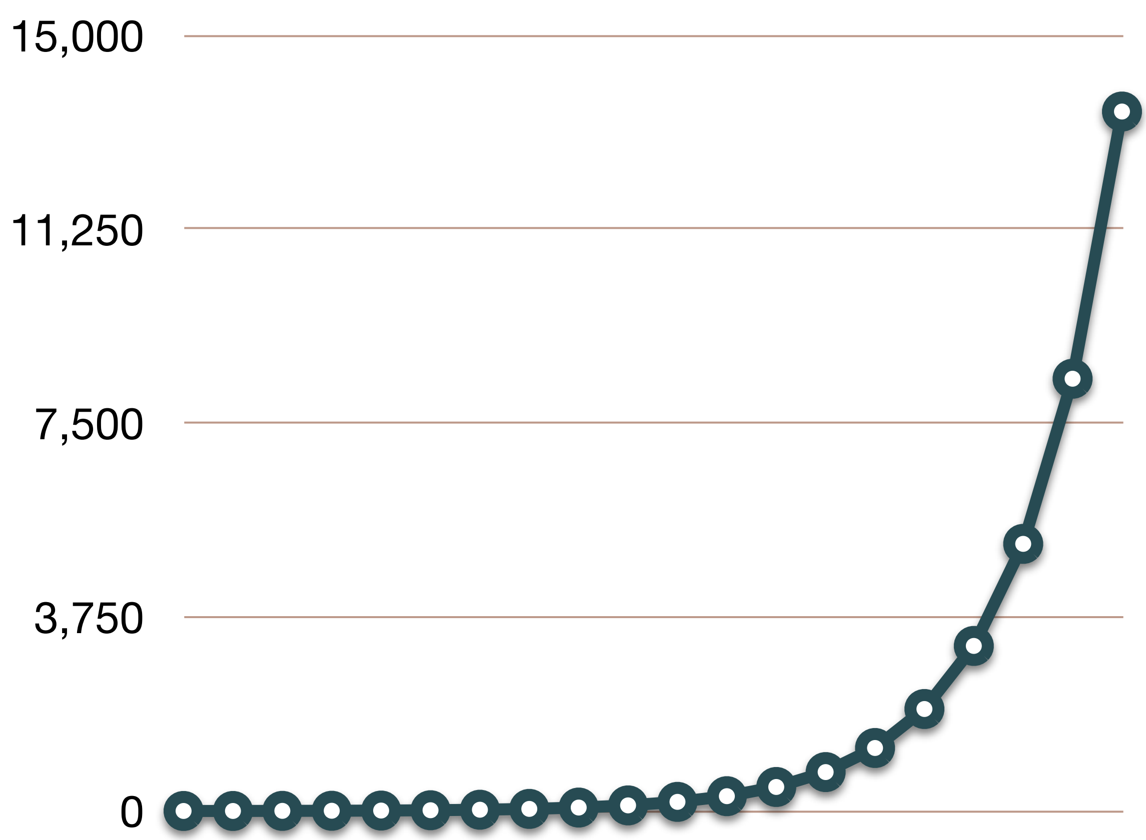
Counting

- In general, how many computations does it take to do Fibonacci(*n*)?

- Fibonacci(<= 1): = 1
- Fibonacci(2): 1 + Fibonacci(1) + Fibonacci(0) = 3
- Fibonacci(3): 1 + Fibonacci(2) + Fibonacci(1) = 5
- Fibonacci(4): 1 + Fibonacci(3) + Fibonacci(2) = 9
- Fibonacci(5): 1 + Fibonacci(4) + Fibonacci(3) = 15
- Fibonacci(6): 1 + Fibonacci(5) + Fibonacci(4) = 25
- Fibonacci(7): 1 + Fibonacci(6) + Fibonacci(5) = 41
- Fibonacci(9): = 67
- Fibonacci(10): = 109
- Fibonacci(11): = 177
- Fibonacci(12): = 287
- Fibonacci(13): = 465
- Fibonacci(14): = 753

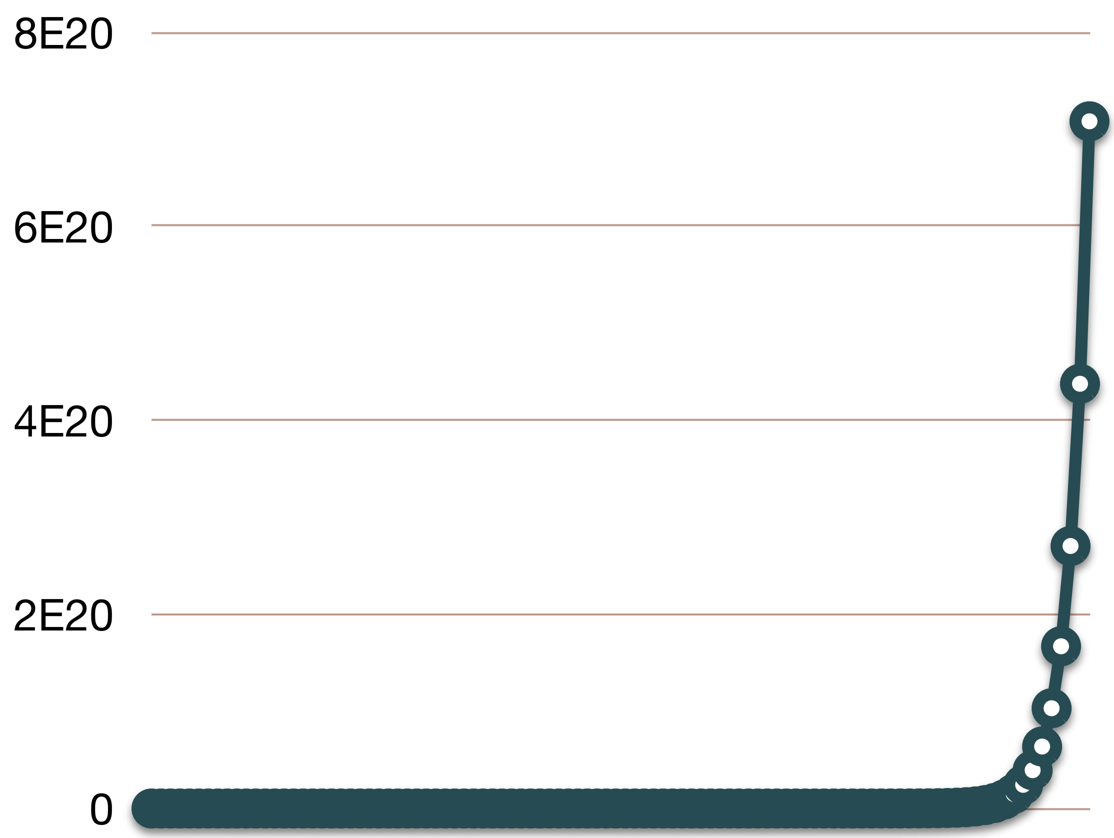
Not Looking Good.

- Number of computations, for Fibonacci(0) ... Fibonacci(20)



Bad. Very Bad.

- Number of computations, for Fibonacci(0) ... Fibonacci(100)



This Algorithm Stinks

- Algorithmic complexity theory helps us **figure out just how much it stinks.**
  - How fast is it growing?
  - Can it be improved?
  - Is all hope lost?
- So... is all hope lost?

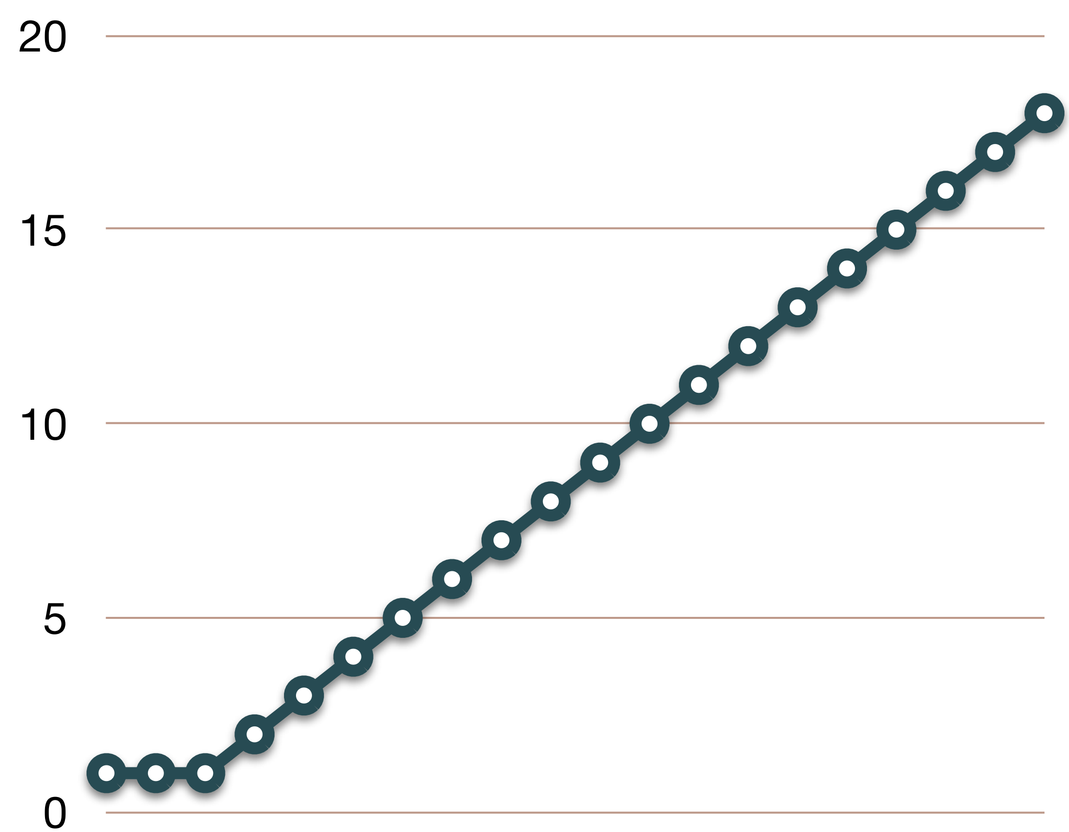
Fibonacci Sequence (Again)

- How about this?

```
Procedure Fibonacci(n):
   if n == 0 or n==1:
         return 1                                    // here's a computation
   else:
         minusone := 1
         minustwo := 1
         current := 0
         for i := 2 to n do:                         // we do this n-1 times
               current := minusone + minustwo       // here's a computation
               minustwo := minusone
               minusone := current
         return current
```

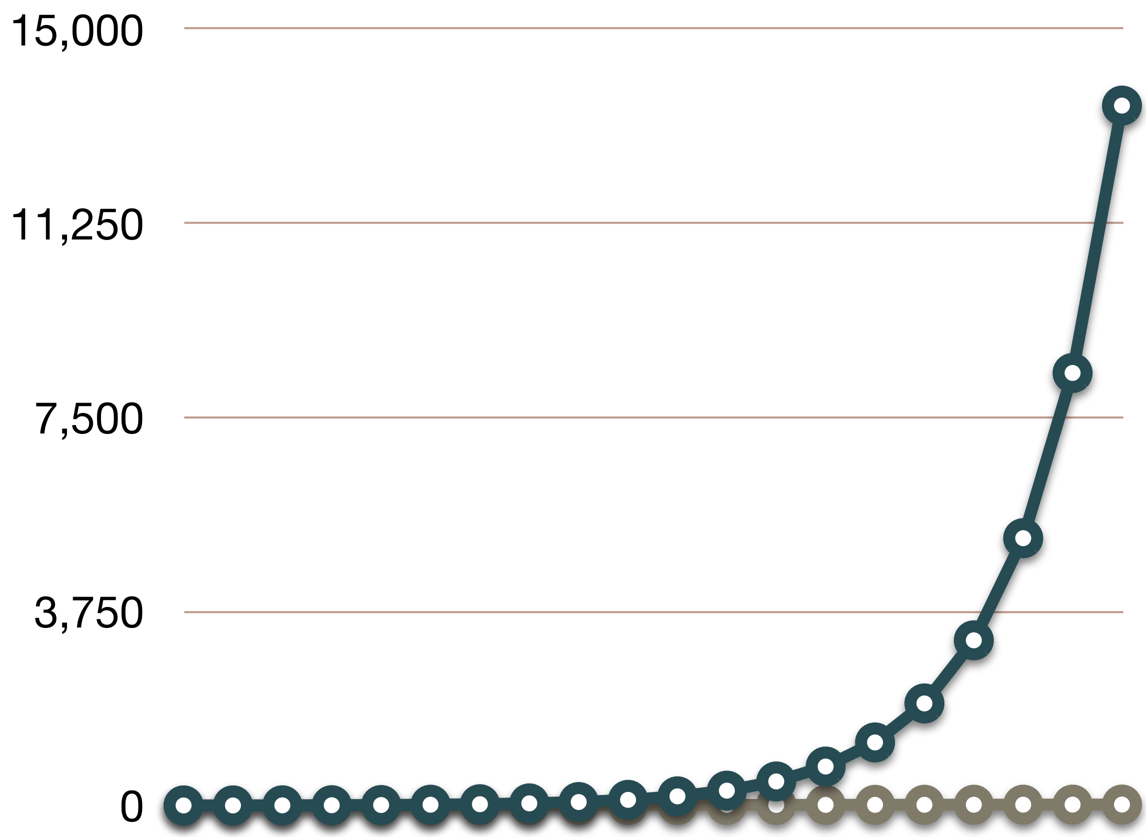
Counting

- If you call Fibonacci(0) you get 1 computation
- If you call Fibonacci(1) you get 1 computation
- If you call Fibonacci(n > 1) you get about n-1 computations
- **Happy.**



Which Would You Prefer?

- It'd sure be nice to be able to figure this out *without running first!* AT&T would have liked to know.



Theory of Computation

- Describe an automaton.
- Show that it is (or is not) capable of computing various things.
- Show that it can computer *at least as many* things than another automaton
  How could you do this?
- Show that it is *exactly equivalent* to another automaton.
  How could you do this?
- **Why is this useful?**

The Halting Problem

- Prove that there exists something easily described, and would be useful to have, but which **cannot be computed.**
- The **Halt(*program, input*)** function tells us this
  - “Does a given *program*, if fed *input* as its input data, eventually halt and return a value, or does it go into an infinite loop?”
- That’d be useful! For example, if we’re about to run a program on a very costly supercomputer, we’d sure like to first know if it will eventually give us an answer or if we’re just wasting cycles!
- Can’t be done.

Describing the Halting Problem

- All programs can be expressed as **unique integers.**

1. Convert your program into machine language for, say, the Intel Pentium.

2. This program now is a string of 1's and 0's.

3. That string is a number. There you go.

   A program.
   1011011011110101101011010100101101011010111111001010100010000
   1001001001000101010101000000001111010100101001010001010101011011
   1101010100101000010110111011111001001010101010001010101011101000
   1010101010101010110100000101010100010101001010010111010101111010
   0101000101010101010101001001010100100101000101010001010011101010

Describing the Halting Problem

- All **pairs of integers** ⟨*x*,*y*⟩ can be expressed as a **unique integer** *z*. *One scheme:*

1. Look up the value for ⟨*x*,*y*⟩ in the table at right.
2. That's the value for *z*.

- **Can you write the function f(*x*,*y*) which returns *z* without enumerating all the values?**

- **Is it difficult to write g(*z*) which returns *x* and *y*?**

x

y	**0**	**1**	**-1**	**2**	**-2**	**3**	**-3**	**4**	**...**
0	0	1	3	6	10	15	21	28	
1	2	4	7	11	16	22	29		
-1	5	8	12	17	23	30			
2	9	13	18	24	31				
-2	14	19	25	32					
3	20	26	33						
-3	27	34							
4	35								
...									

etc.

Describing the Halting Problem

- All **lists of integers** ⟨*a,b,c,d,e,f,...*⟩ can be expressed by a unique integer.

1. Convert the first two integers *a, b* into an integer α
2. Take α and the next integer *c* and convert them into an integer β
3. Take β and the next integer *d* and convert them into an integer γ
4. And so on... repeat this until the whole list has been converted into an integer (let's call it ω)
5. Let λ be the length of the list. Convert λ and ω into a unique integer ξ. That's your unique integer.

- **Given ξ, is it straightforward to *de-convert* back to the list?**

Describing the Halting Problem

- Since all lists of integers can be expressed as a single unique integer, any particular string of data we might want to feed to a program can be expressed as a single unique integer.

- **Halt(*program, data*)** takes two integers:

    - ***program*** the unique integer representing the program to test

    - ***data*** the unique integer representing the string of data fed to it

- **Halt** will return an integer: **1** if the program represented by the integer ***program*** halted and returned a value when fed ***data*** as input, and **0** otherwise.

Proving the Halting Problem

- Our proof will go like this:

1. Suppose you could write the procedure **Halt**

2. Then it would be easy to write a procedure called **Bad**

3. If you could write **Bad**, it'd result in a logical contradiction which we'll show.

   (and thus the universe would cease to exist or something)

4. Thus **Halt** can't be written.

"proof by contradiction"

The Bad Function

- **Bad(*program*)** works like this:
  - If ***program*** halts and returns a value when fed *its own integer* as input, then Bad will go into an infinite loop.
  - If ***program*** goes into an infinite loop when fed *its own integer* as input, then Bad will halt and return a 1.
- Example code:

```
procedure Bad(program):
  if Halt(program, program) == 1 then:
      while(1==1):
          print “Ha ha ha! I’m in an infinite loop!”
  else:
      return 1
```

- **Piece of cake.**

Proof

- What does **Halt(*Bad, Bad*)** do?

- Suppose it returns a 1. Then **Bad (*Bad*)** must halt.

- So let's run **Bad(*Bad*).** It calls **Halt (*Bad, Bad*),** which returns a 1, which then causes **Bad(*Bad*)** to go into an infinite loop!

- So if **Halt(*Bad, Bad*)** returns a 1, then **Halt(*Bad, Bad*)** must return a 0.
  Not good.

- procedure Bad(program):

```
if Halt(program, program) == 1 then:
    while(1==1):
        print "Infinite Loop!"
else:
    return 1
```

Proof

- Let's suppose it returns a 0. Then **Bad(*Bad*)** must go into an infinite loop.

- So let's run **Bad(*Bad*).** It calls **Halt(*Bad, Bad*),** which returns a 0, which then causes **Bad(*Bad*)** to halt and return a 1!

- So if **Halt(*Bad, Bad*)** returns a 0, then **Halt(*Bad, Bad*)** must return a 1.

- ```
  procedure Bad(program):
      if Halt(program, program) == 1 then:
          while(1==1):
              print "Infinite Loop!"
      else:
          return 1
  ```
  ```

Proof

- So... if **Halt(*Bad, Bad*)** returns a 0, then **Halt(*Bad, Bad*)** must return a 1.
  And... if **Halt(*Bad, Bad*)** returns a 1, then **Halt(*Bad, Bad*)** must return a 0.
- Halt can't return either a 0 or a 1.
  But those are the only things Halt can do!
- A contradiction. So Halt can't exist.
- Q.E.D.