Computational Complexity
The Simplified Version

Prof. Darrell Long

CSE 13S

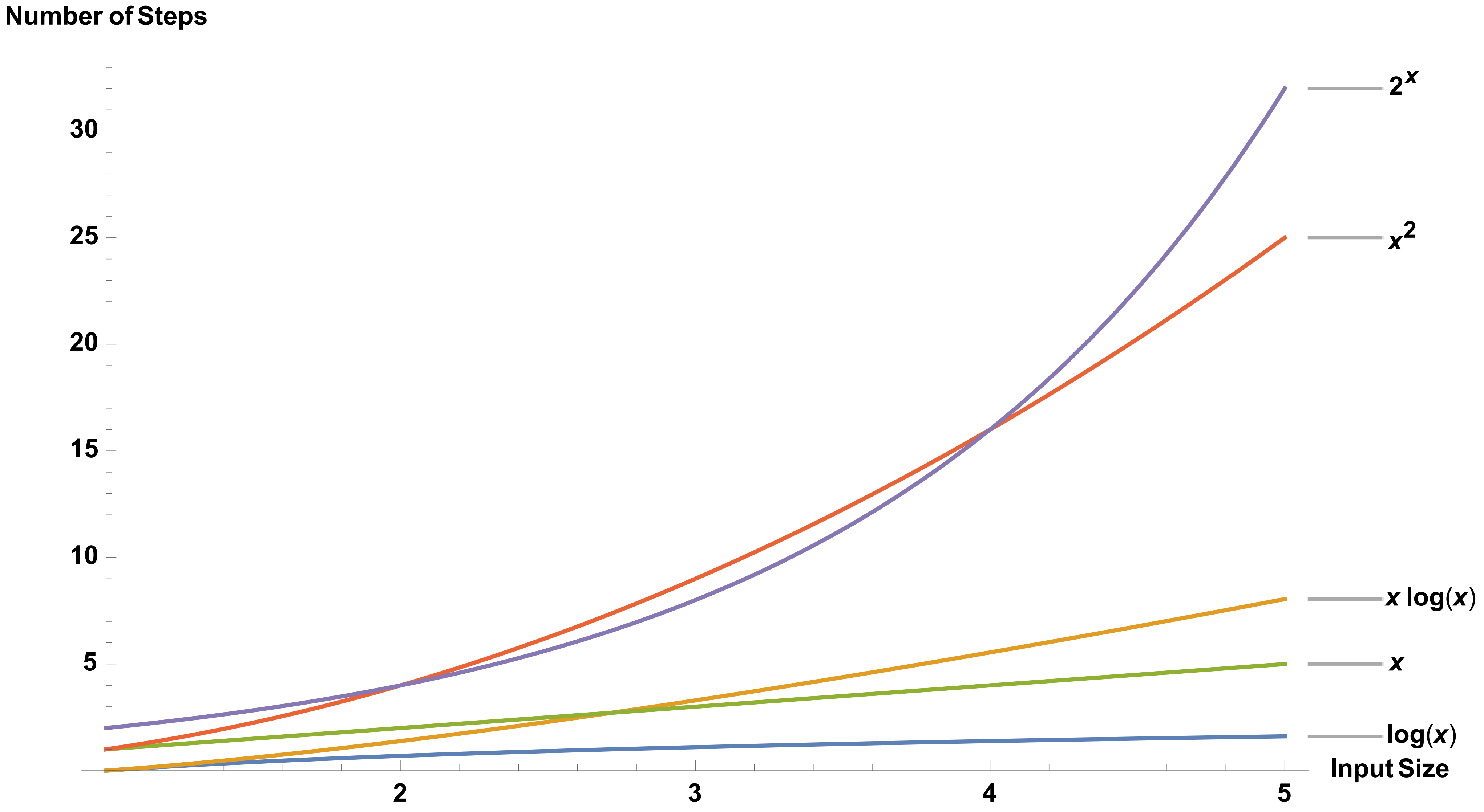
Who’s Counting?

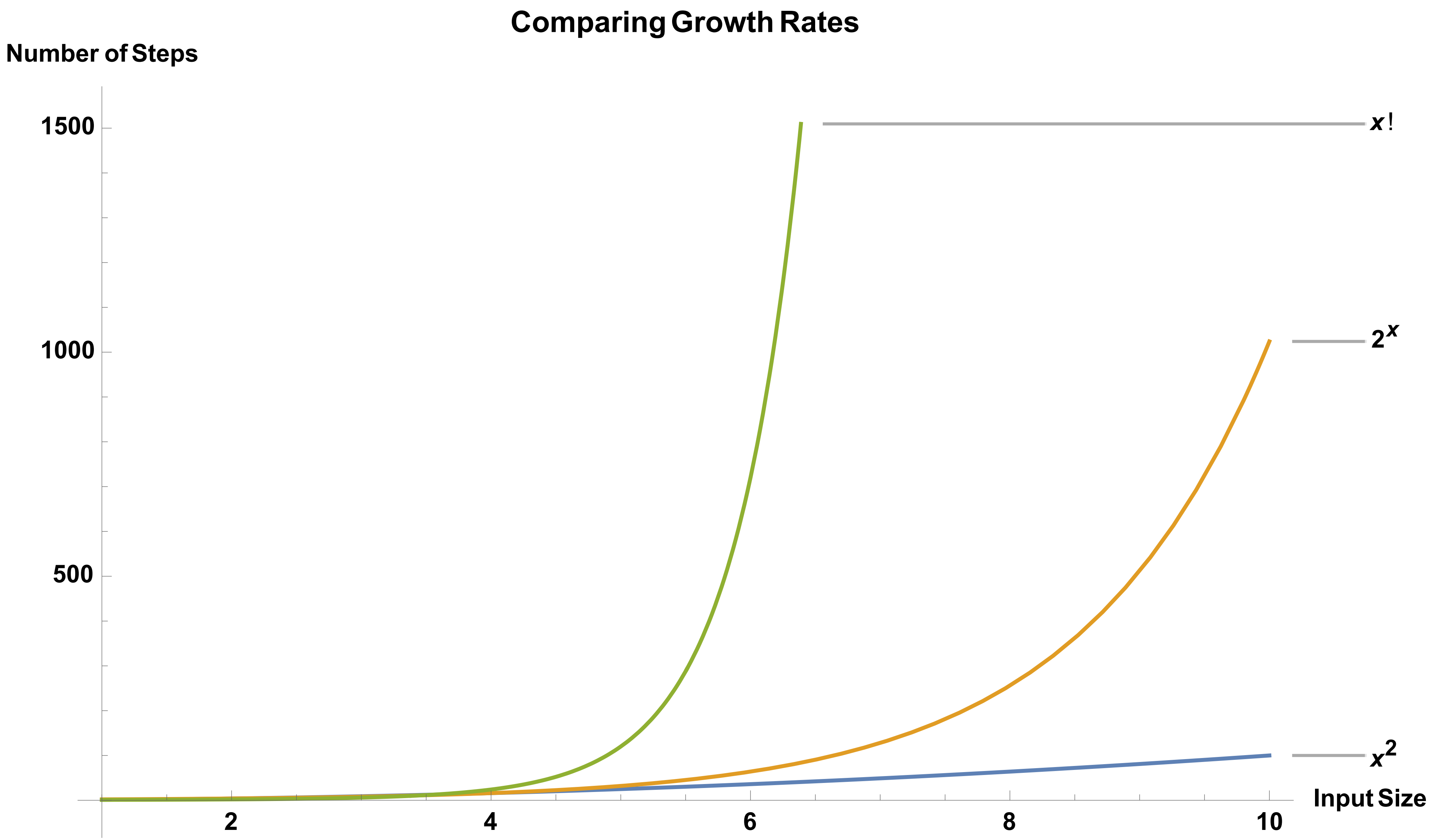
- We talk about the size of a problem:
  - How *long* is the string?
  - How *large* is the number?
  - How *many* items are in the set?
- We often talk about the *size of the input* (like the length of a string).
- We talk about the *cardinality* or *magnitude* of a number.
- It is the same for numbers and strings if we write the number in *unary* notation:
  - 1 = 🥕
  - 5 = 🥕🥕🥕🥕🥕
  - 10 = 🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕
  - 20 = 🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕🥕
- The *problem* and the *algorithm* that solves it define the complexity.

Some Common Functions

function	10	30
log(x)	2.30259	3.4012
x	10	30
x log(x)	23.0259	102.036
x^2	100	900
x^3	1000	27,000
2^x	1024	1,073,741,824
x!	3,628,800	265,252,859,812,191,058,636,308,480,000,000

Comparing Growth Rates





Comparing Growth Rates
Number of Steps
1500
1000
500
2
4
6
8
10
Input Size
$x!$
2^x
x^2

Exemplars

Order	Example
$O(\log(n))$	Binary Search
$O(n)$	Find Minimum
$O(n \log(n))$	Merge Sort
$O(n^2)$	Bubble Sort
$O(n^3)$	Matrix Multiply
$O(2^n)$	Enumerate Subsets
$O(n!)$	Enumerate Permutations

We are going to look at a little formal mathematics

$$f(n) \in O(g(n)) \text{ if and only if } \exists c, n_0 \text{ such that } \forall n > n_0, f(n) \leq c \cdot g(n)$$

Formalities

When we say things like “Bubble Sort is $O(n^2)$” or “Bubble Sort is order n^2” we’re not being very precise:

- Every Computer Scientist *should* know what we mean, but
- We must agree on what we mean.

Once x gets large enough, there is some constant c where $f(x)$ is always bounded by $c \cdot g(x)$:

- $f(n) \leq c \cdot g(n)$

$$O = \{f : \text{there exist } n_0 \text{ and } c \text{ such that } 0 \leq f(n) \leq c \cdot g(n) \text{ for all } n > n_0\}$$

Simple Linear Algorithm

$O(n)$

```
int minIndex(uint32_t a[], int first, int last)
{
  int min = first;
  for (int i = first; i < last; i += 1) {
    min = a[i] < a[min] ? i : min;
  }
  return min;
}
```

$O(n)$

Simple Quadratic Algorithm

$O(n^2)$

```
void insertionSort(uint32_t a[], int length) {
  for (int i = 1; i < length; i += 1) {
    int j = i;
    uint32_t tmp = a[i];
    while (j > 0 && a[j - 1] > tmp) {
      a[j] = a[j - 1];
      j -= 1;
    }
    a[j] = tmp;
  }
  return;
}
```

$O(n)$

$O(n)$

A Less Simple Algorithm

On *Average*

$O(n \log(n))$

Worst case

$O(n^2)$

```
int partition(uint32_t a[], int32_t low, int32_t high) {
  uint32_t pivotValue = a[(low + high) / 2];

  int32_t i = low - 1;
  int32_t j = high + 1;
  do {
    do {
      i += 1;
    } while (a[i] < pivotValue);
    do {
      j -= 1;
    } while (a[j] > pivotValue);
    if (i < j) {
      SWAP(a[i], a[j]);
    }
  } while (i < j);
  return j;
}

void quickSort(uint32_t a[], int32_t low, int32_t high) {
  if (low < high) {
    uint32_t p = partition(a, low, high);

    quickSort(a, low, p);
    quickSort(a, p + 1, high);
  }
  return;
}
```

What about these?

$O(n)$

It all depends on this!

$O(\log(n))$ or $O(n)$

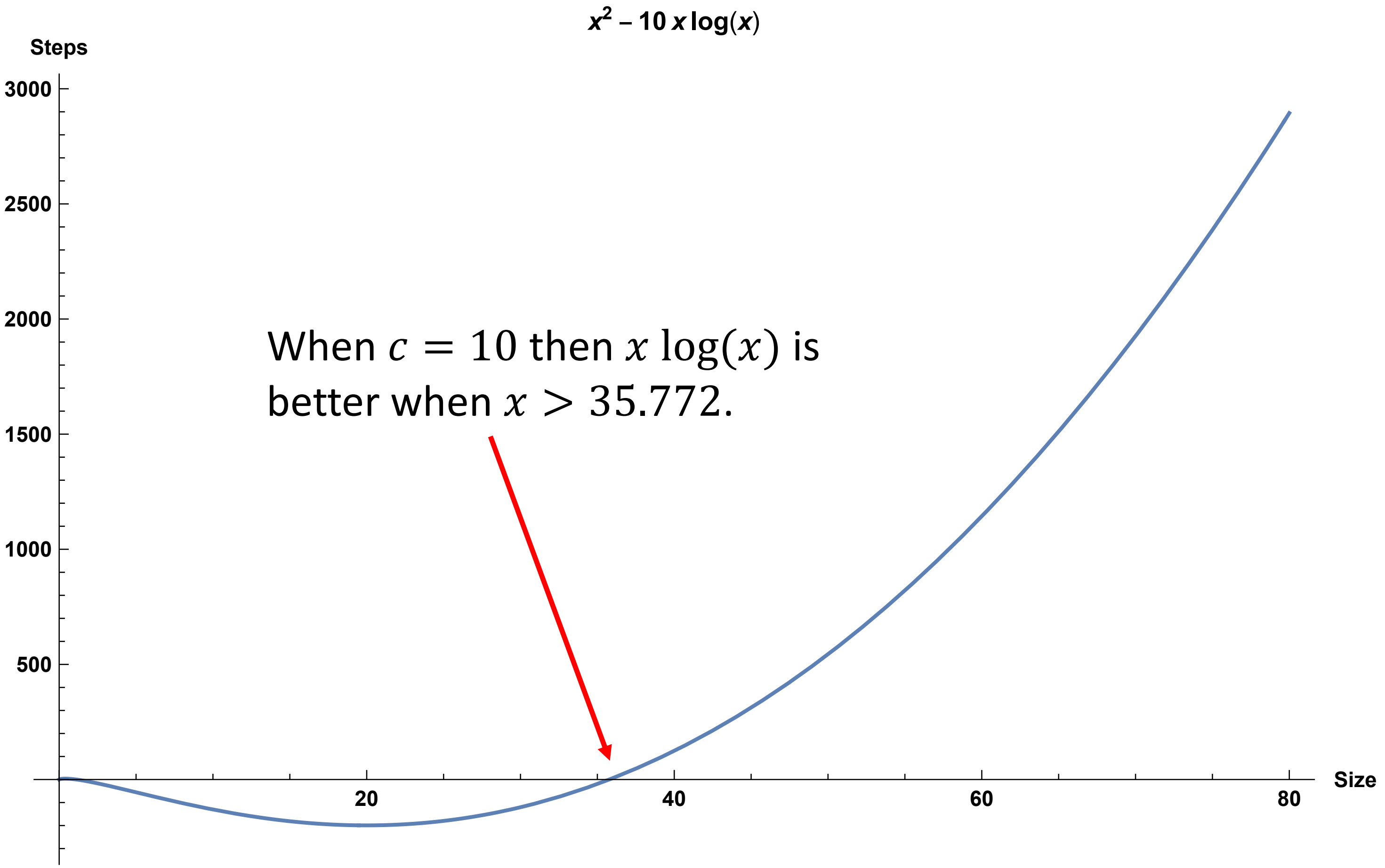
Estimating Complexity

- For now, we can adopt some simple rules for estimating complexity.
- We do it by looking at the loops and the recursion.
- We *add* the complexity of loops at the same level.
  - In general, this does not matter for larger inputs.
  - Why? It just affects the constant *c*.
- We *multiply* the complexity of loops nested inside of loops:
  - One loop usually contributes $O(n)$
  - So, two nested loops are likely $O(n^2)$
  - And three nested loops are likely $O(n^3)$, and so forth …

So what about c?

- You can think of c as the *overhead* that an algorithm requires.
  - For small *n*, Bubble Sort is faster than Quick Sort because of the overhead.
  - We say that Quick Sort has a "larger constant" or larger c.
- Optimizing your code will make *c* smaller.
  - It may make a big difference in the run-time of your program,
  - But it is *only a constant* speed-up.
- Changing to a better algorithm is much more impactful.
  - The existence of an efficient algorithm determines whether you can solve a particular problem.

Consider $x^2 - 10\,x\log(x)$



When $c = 10$ then $x\log(x)$ is better when $x > 35.772$.

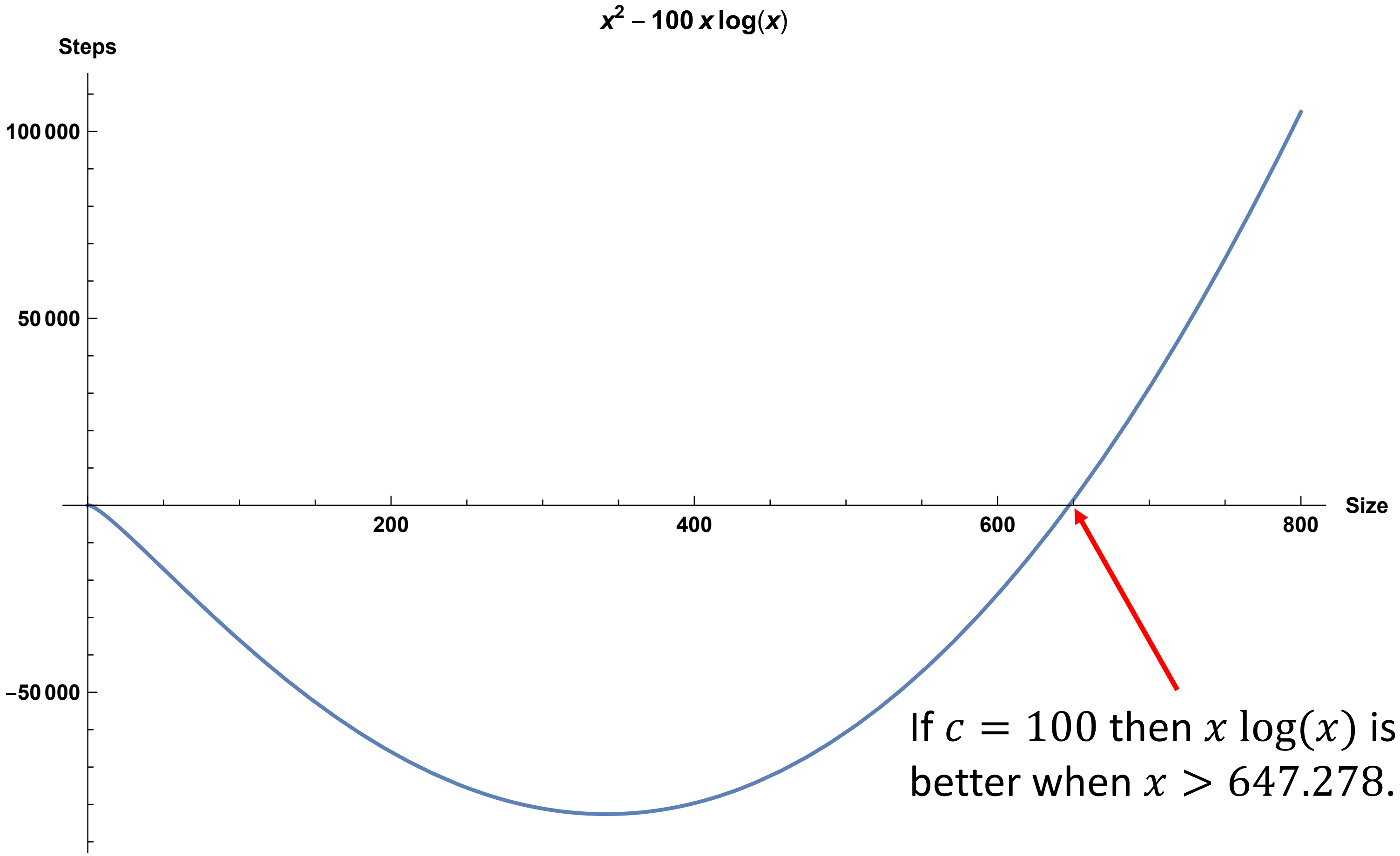
What does that tell me?

It’s probably better to use Bubble Sort for less than 50 items.

```c
#include "bubblesort.h"
#include <stdbool.h>
#include <stdint.h>

void bubbleSort(uint32_t a[], int length) {
  bool swapped;
  do {
    swapped = false;
    for (int i = 1; i < length; i += 1) {
      if (a[i - 1] > a[i]) {
        SWAP(a[i - 1], a[i]);
        swapped = true;
      }
    }
    length -= 1;
  } while (swapped);
  return;
}
```

Consider $x^2 - 100\,x\log(x)$



If $c = 100$ then $x\log(x)$ is better when $x > 647.278$.

So, what about recursion?

$O(n)$

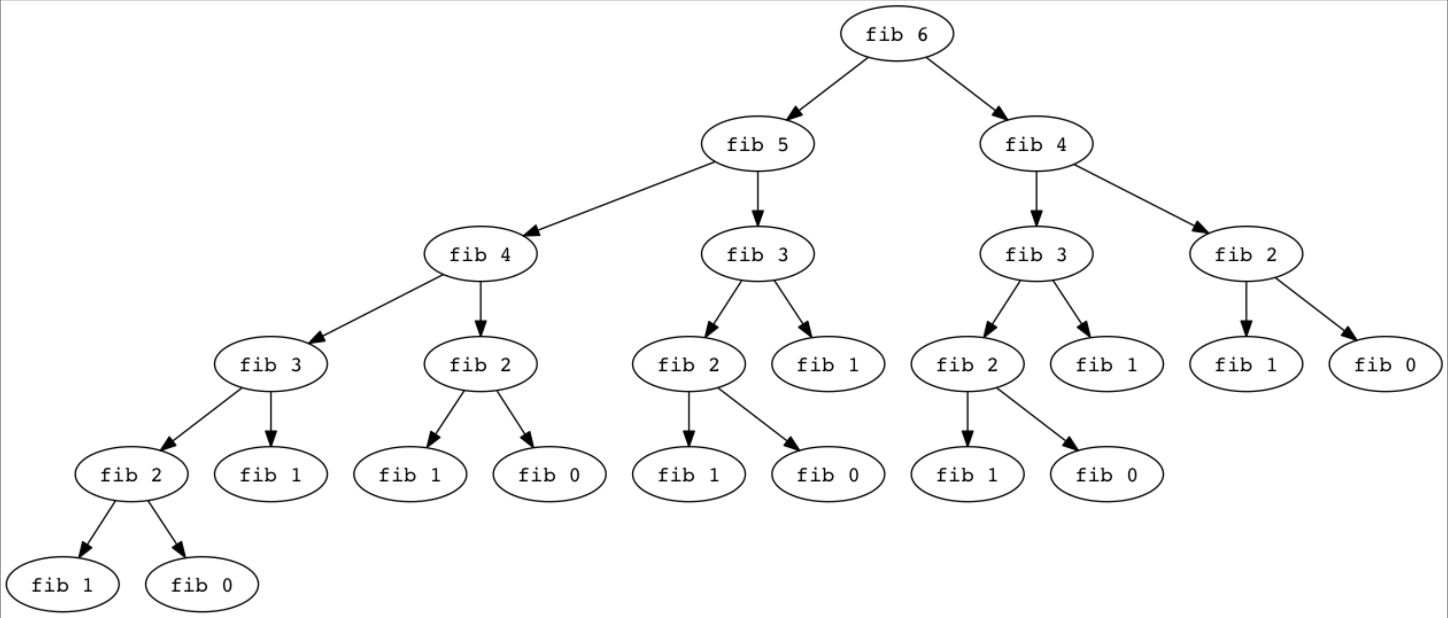
```c
int f(uint32_t n) {
  if (n == 0) {
    return 1;
  } else {
    return n * f(n - 1);
  }
}
```

What about this one?

```
int f(int n) {
  if (n == 0 || n == 1) {
    return n;
  } else {
    return f(n - 1) + f(n - 2);
  }
}
```

$$O\left(\left(\frac{1+\sqrt{5}}{2}\right)^n\right) < O(2^n)$$

Recursive Fibonacci



$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-2) + f(n-1)$$

It all started with Leonardo Pisano wondering about cute little bunnies!

Same Value — Different Algorithm

$O(n)$

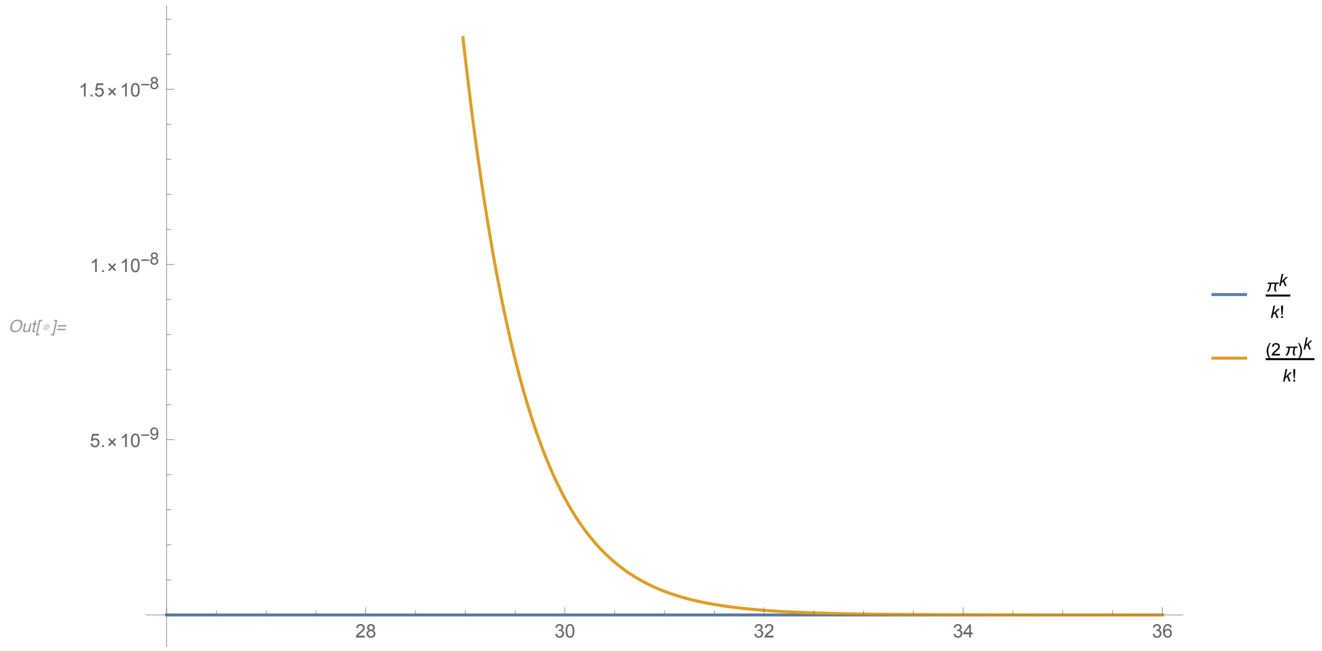
```
int fib(int n) {
  int a = 0, b = 1, sum = 0;
  if (n < 2) {
    return n;
  }
  for (int i = 2; i <= n; i++ 1) {
    sum = a + b;
    a = b;
    b = sum;
  }
  return sum;
}
```

What about this?

- The error term is $\frac{x^k}{k!}$
- How large does that have to be for it to be less than ε?
- The largest $x = 2\,\pi$, so we can figure it out!

```
double Sin(double) {
    x = modulus(x, 2 * M_PI); // Normalize to [-2π, 2π]
    double sgn = 1, val = x, trm = x;
    for (int k = 3; abs(trm) > epsilon; k += 2) {
        trm = trm * (x * x) / ((k - 1) * k);
        sgn = -sgn;
        val += sgn * trm;
    }
    return val;
}
```

Let's plot the ratio $(2\pi)^k/k!$



- For 10 digits of accuracy, we will need at most 35 iterations!
- We pick the largest x, in this case 2π.
- We find $k \ni \frac{x^k}{k!} < \varepsilon$.

```c
long double Sqrt(long double x) {
  long double f = 1.0;
  while (x > 1) {
    x /= 4.0;
    f *= 2.0;
  }
  long double m, l = 0.0, h = (x < 1) ? 1 : x;
  steps = 0;
  do {
    steps += 1;
    m = (l + h) / 2.0;
    if (m * m < x) {
      l = m;
    } else {
      h = m;
    }
  } while (abs(l - h) > epsilon);
  return f * m;
}
```

And what about this?

An Extreme Example

- Factor:

  $n =$
  3,712,368,003,163,152,165,726,917,914,396,003,989,286,
  393,900,857,974,804,551,541
- You could try every number 2, 3, 4, … and see if it divides *n*.
  - Your first success would be 1,821,640,449,726,328,871,578,165,744,201.
  - Which means you will finish in about 10^{13} years using your laptop.
- Is this the best we can do?
  - We do not know. We *believe* that integer factorization is *hard*.
  - We've known for 2300 years how to find all primes up to n in $O(n \log \log(n))$ time.
  - The best we know is the General number field sieve.

Summary

- Computer Scientists talk about the complexity of algorithms all the time.
- Complexity can measure *time*, *space* or both.
  - *Time* is "how many steps it takes" and
  - *Space* is "how much memory it uses".
- The algorithm that you choose has the largest impact on the performance of your program.
- For *small* problems, it may be better to use a worse algorithm if the better algorithm that a larger constant.
  - But you need to take the time and examine it carefully.