Complexity Analysis
(Orders and Notations)

PROF. NAVRATI SAXENA

Algorithm and Complexity

Algorithm: A finite sequence of precise instructions for performing a computation for solving a problem.

Computational complexity: Measure of the processing time required by the algorithm to solve problems of a particular problem size.

Time Complexity of Algorithm

How to measure the **complexity** (time) of an algorithm? What is this a function of?

The **size** of the problem: an integer n

1. # inputs (e.g., for sorting problem)
2. # digits of input (e.g., for the primality problem)
3. sometimes more than one integer

Objective: Characterize the running time of an algorithm for increasing problem sizes

Units of time

What is a good unit of time ?

- 1 microsecond ?
  - Possibly no. It’s too specific and machine dependent

- 1 machine instruction
  - No. Also quite specific and machine dependent

- # of code fragments that take constant time
  - Yes. Could be used

Worst-Case Analysis

- **Worst case** running time.

- **A bound on largest possible running time of algorithm on inputs of size n.**
  - Generally captures efficiency in practice.
  - Sometimes can be an overestimate.

Measuring Efficiency of Algorithms

- Two algorithms: Algo1 and Algo2 that solve the same problem.
- We need a fast running time.
- How do we choose between the algorithms?

Efficiency of Algorithms

Implement the two algorithms and compare their run-times?

Limitations with this approach

1. How are the algorithms coded?
   - We want to compare the algorithms, not the implementations.
2. What computer should we use?
   - Choice of operations could favor one implementation over another.
3. What data should we use?
   - Choice of data could favor one algorithm over another

Measuring Efficiency of Algorithms

Objective: Analyze algorithms independently of specific implementations, hardware or data.

Observation: An algorithm's execution time is related to the number of operations it executes

Solution: Count the number of **STEPS**: **significant** time and operations the algorithm will perform for an input of given size

An Example: Linear Search

```
int linearSearch (int k)
{
        for(int i = 0; i<A.length; i++ )
        {       if(A[i]==k)
                    return i;
        }
        return -1;
}
```

- What is the maximum number of steps linSearch takes?
- What’s a step here?
- for an Array of size 32? for an Array of size n?

Growth Rates (1/3)

- Algorithm A requires $n^2 / 2$ steps to solve a problem of size n
- Algorithm B requires $5n+10$ steps to solve a problem of size n
- Which one is better for us?

Growth Rates (2/3)

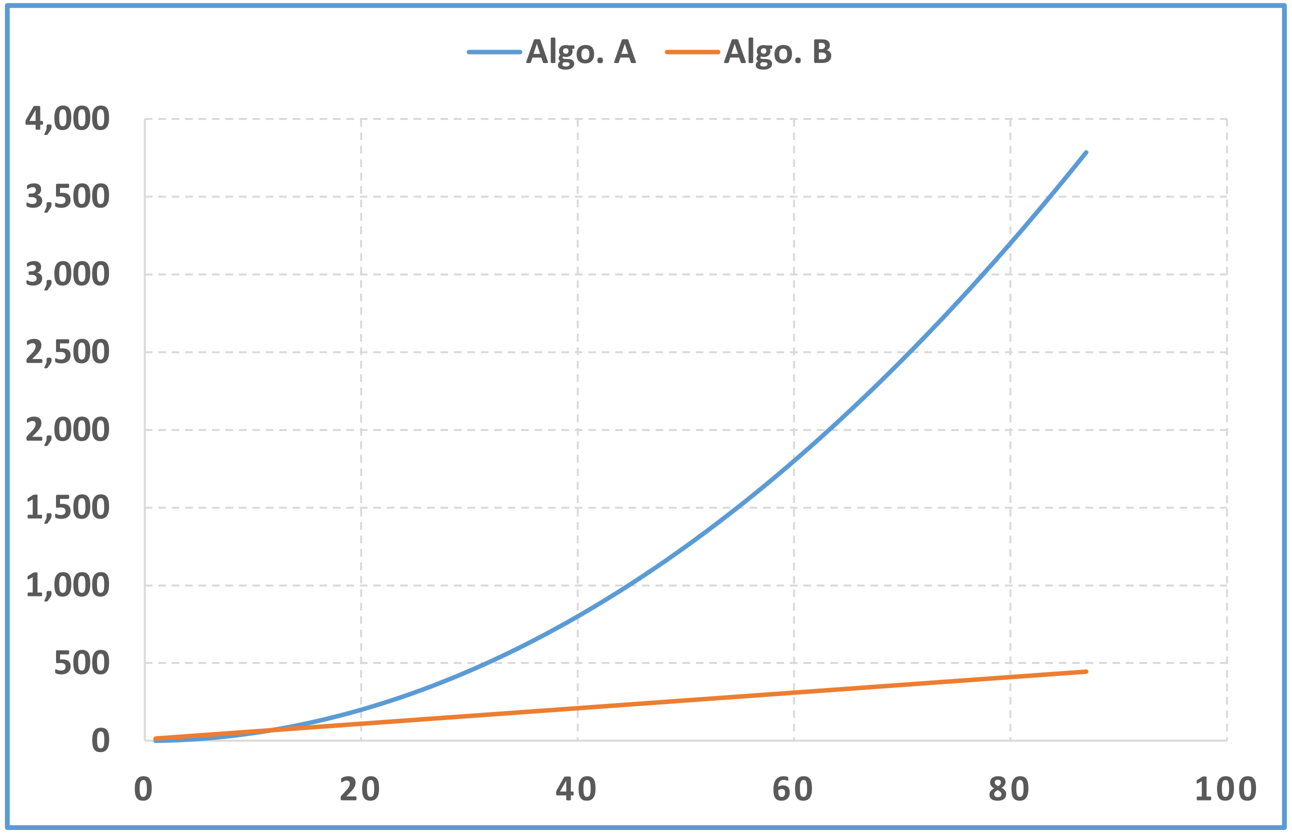
- Increase the size of input n,
- Check how the **execution time grows** for these algorithms?

n	1	2	5	8	10	20	50
$n^2 / 2$	0.5	2	12.5	32	50	200	1250
5n+10	15	20	35	50	60	110	260

n	100	1,000	10,000	100,000
$n^2 / 2$	5,000	500,000	50,000,000	5,000,000,000
5n+10	510	5,010	50,010	500,010

Care about large input sizes

Growth Rates (3/3)



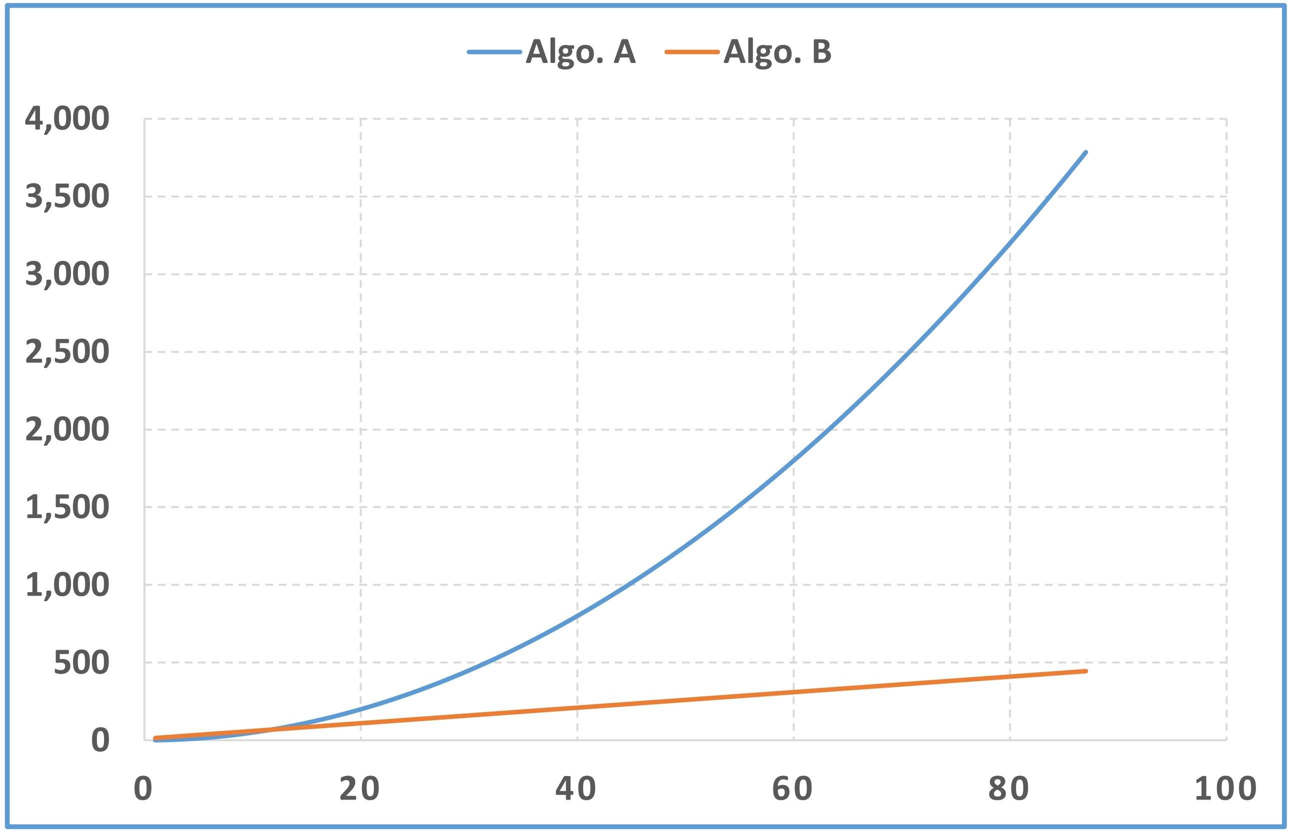
- **A**: $n^2 / 2+1$ operations (for size n)
- **B**: $5n + 10$ operations (for size n)
- For large problems B is more efficient
- **Important:** Growth of algorithm's execution time as a function of input.

<u>Conclusion</u>:

- **Algorithm A:** requires time proportional to n^2
- **Algorithm B:** requires time proportional to n
- ***B's time requirement grows more slowly*** *than A's*

Analyzing Order Complexity

***Big O notation**: A function $f(x)$ is $O(g(x))$, if there exist two positive constants, c and k, such that $f(x) \leq c \times g(x), \forall\ x > k$*

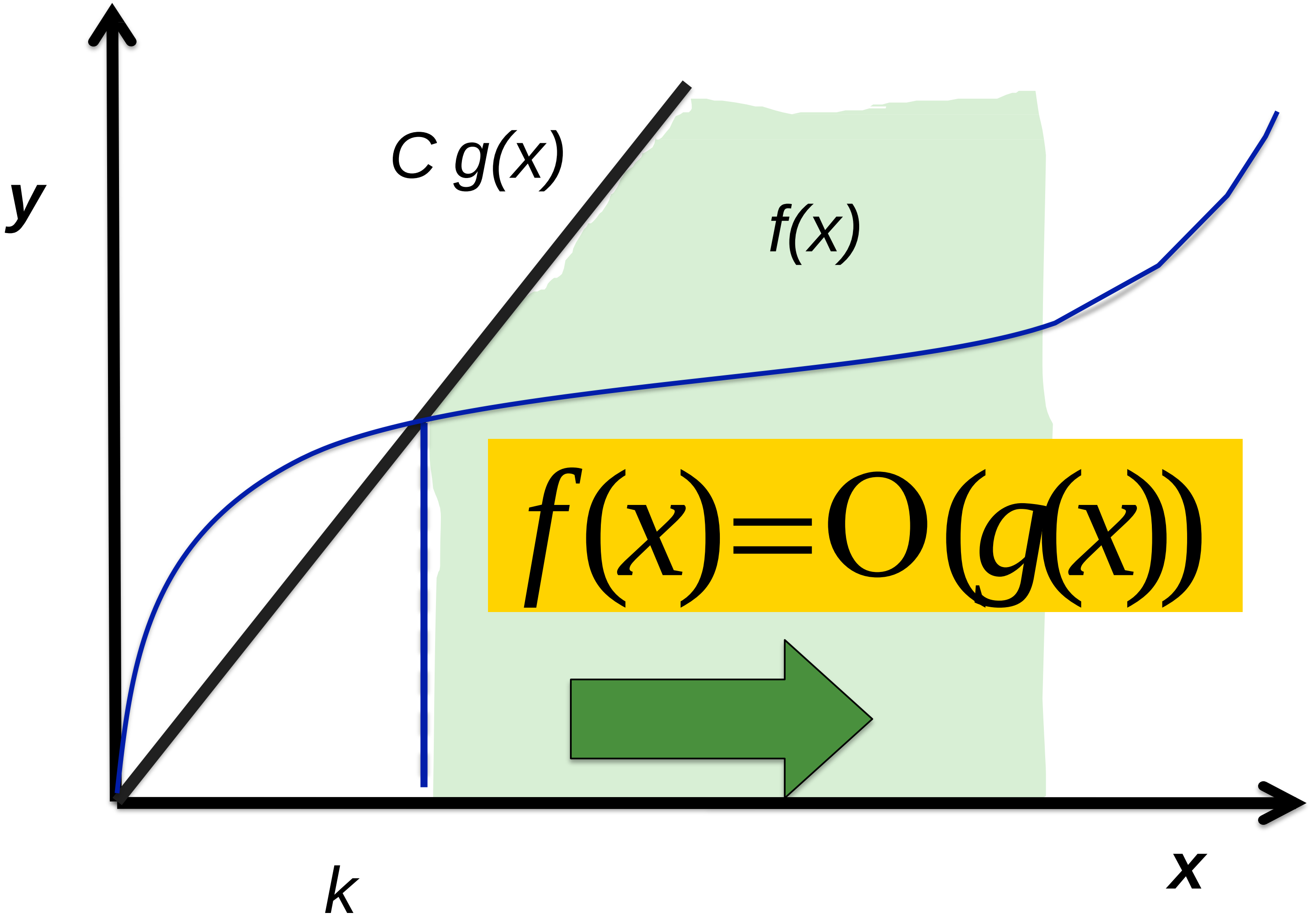


Focus on:

- **Shape of function g(x)**
- **Large x**

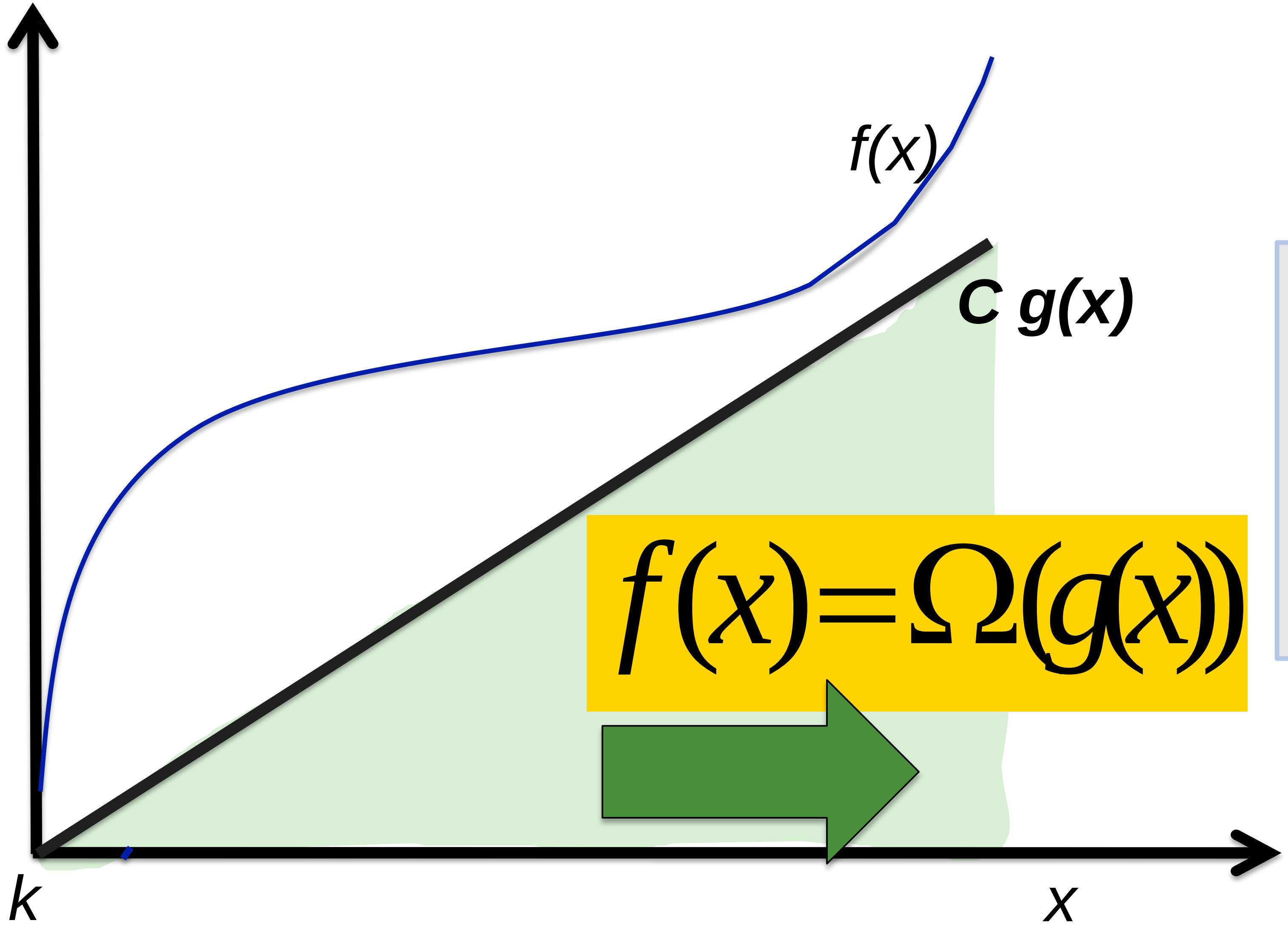
Infinite many pairs: (C, k)

Upper Bound: Notation Big "O"



- Let f and g be functions.
- $f(x) = O(g(x))$
- If there are positive constants c and k such that $f(x) \leq C\, g(x)$, whenever $x > k$

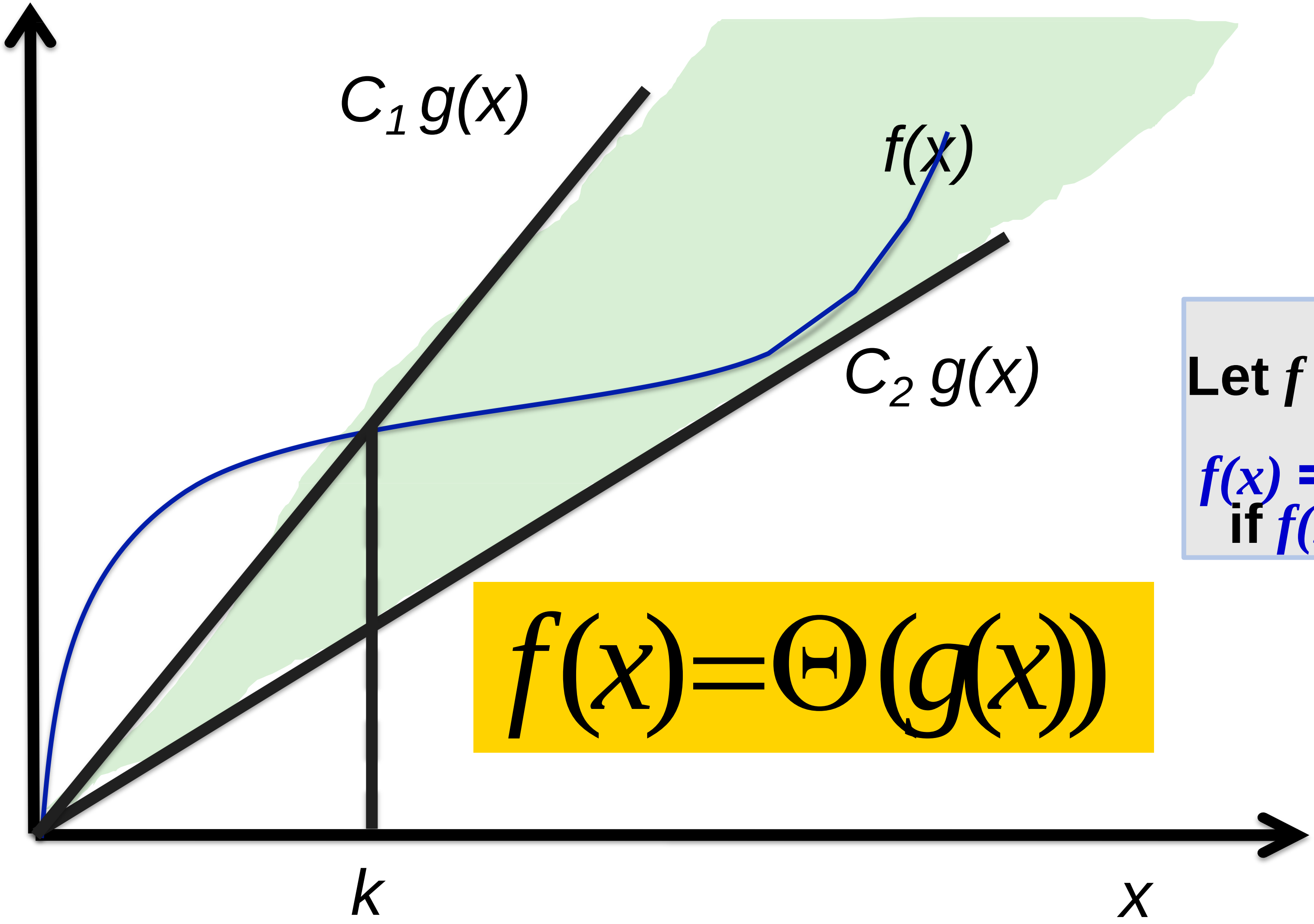
Lower Bound: Ω Notation



$$f(x) = \Omega(g(x))$$

- Let f and g be functions.
- $f(x) = \Omega(g(x))$
- If there are positive constants c and k such that $f(x) \geq C\, g(x)$, whenever $x > k$

A Tight Bound – Notation Θ



$$f(x) = \Theta(g(x))$$

Let f *and* g **be functions.**

$f(x) = \Theta\,(g(x))$ **if** $f(x) = O(g(x))$ **and** $f(x) = \Omega(g(x))$

Sample Questions

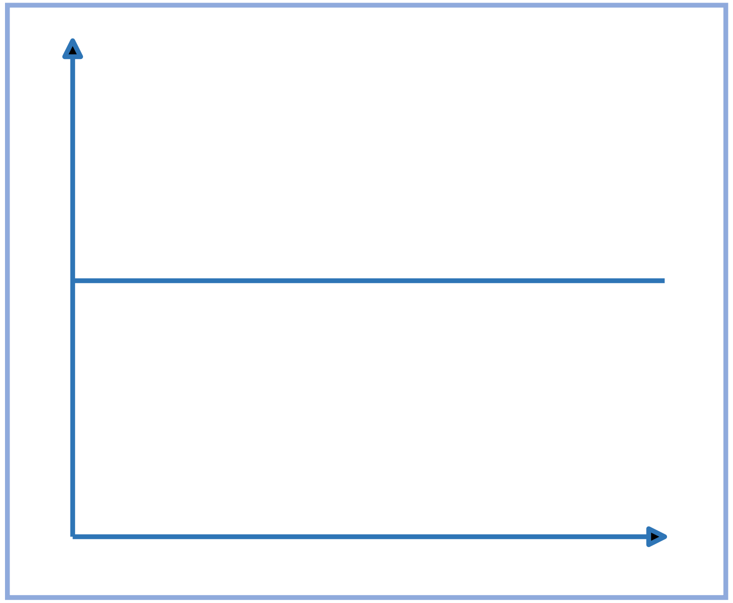
- If ***f(n) = n² + 3n***, can we say f(n) = ***O(n²)***?
- If ***f(n) = n + log(n)***, what can we say about Order Complexity **O**?
- If ***f(n) = 2n + n log(n)***, what is the order complexity **O?**

Orders of Magnitude

- **O (big O)** is used for Upper Bounds in algorithm analysis:
  - Used in worst case analysis
  - This algorithm never takes more than this number of steps
- We will concentrate on worst case analysis
- **Ω (big Omega)** is used for lower bounds in problem characterization:
  - Minimum number of steps this problem takes
- **θ (big Theta)** for tight bounds: a more precise characterization

Common Orders: Constant

O (1)



Shape

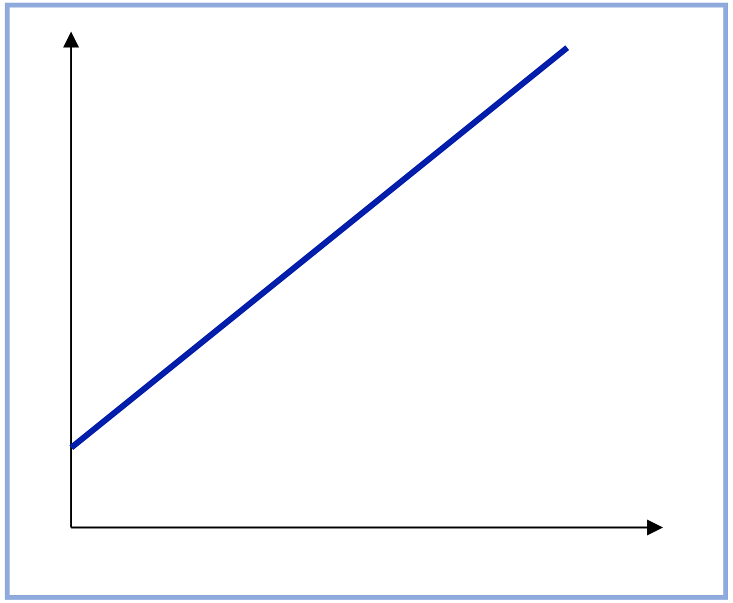
Examples:

Any integer/double arithmetic/logic operation

Accessing a variable or an element in an array

Common Orders: Linear

O (n)



Shape

$f(n) = a*n + b$

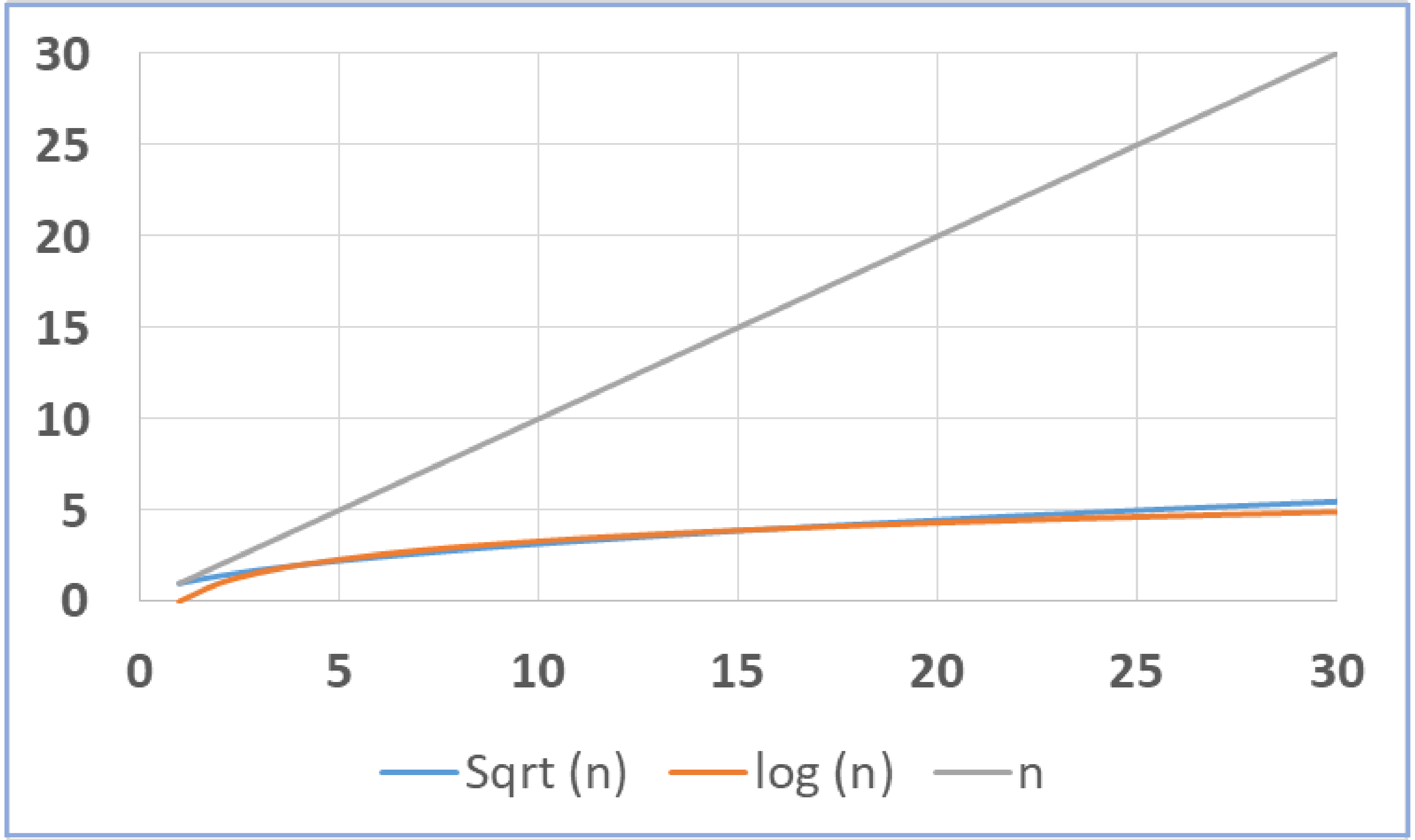
- *a is the slope*
- *b is the Y intersection*

Example of Algorithms with Linear Order

Example: copying an array

```
for (int i = 0; i < size; i++){
    a[i] <- b[i];
}
```

Other Shapes: Sublinear



Common Sub-Linear Order: Logarithm

- **$\log_b n$:** is the number x such that $b^x = n$
  - $2^3 = 8 \Rightarrow \log_2 8 = 3$
  - $2^5 = 32 \Rightarrow \log_2 32 = 5$
  - $2^{10} = 1024 \Rightarrow \log_2 1024 = 10$

- **$\log_b n$**: (# of digits to represent n in base b) – 1
- Most common base used: **2**

More on Logarithms

Common properties of logarithms

- $log(x\,y) = log\,x + log\,y$
- $log(x^a) = a\,log\,x$
- $log_a n = log_b n\,/\,log_b a$

Note: $\mathbf{log_b a}$ *is a* **constant** **=>** $\mathbf{log_a n = O(log_b n)}$ *for any a, b*

Logarithm is a **very** slow-growing function

Complexity: O(log n)

- Common for **"Divide and Conquer"** algorithms,
  - Problem size gets chopped into 1/2 (or 1/3, or ¼) in every step

- How many times 1,000 can be divided by 2 to get 1 ?
  - How about 1,000,000 ?

Example: Guess a Number

- Given a number between 0 and 100
- How many questions (only “Y/N”) needed to find it?

>=	50	N
>=	25	Y
>=	37	N
>=	31	N
>=	28	Y
>=	30	Y

What’s the number?

Another Example: Binary Search

```
private int binSearch (int Arr, int key, int lo, int hi) {
// pre: A array is already sorted
// post: if k in A[lo..hi] return its position, else return (-1)
   int r;
   if (lo>hi)
        r = -1;
   else
      {
        int mid = (lo+hi)/2;
        if (k==A[mid])
             r = mid;
        else if (k < A[mid])
             r = binSearch(Arr, key,lo,mid-1);
        else
              r = binSearch(Arr, key,mid+1,hi);
           }
        return  r;}
```

Not Found

Complexity?

Higher Order: Quadratic

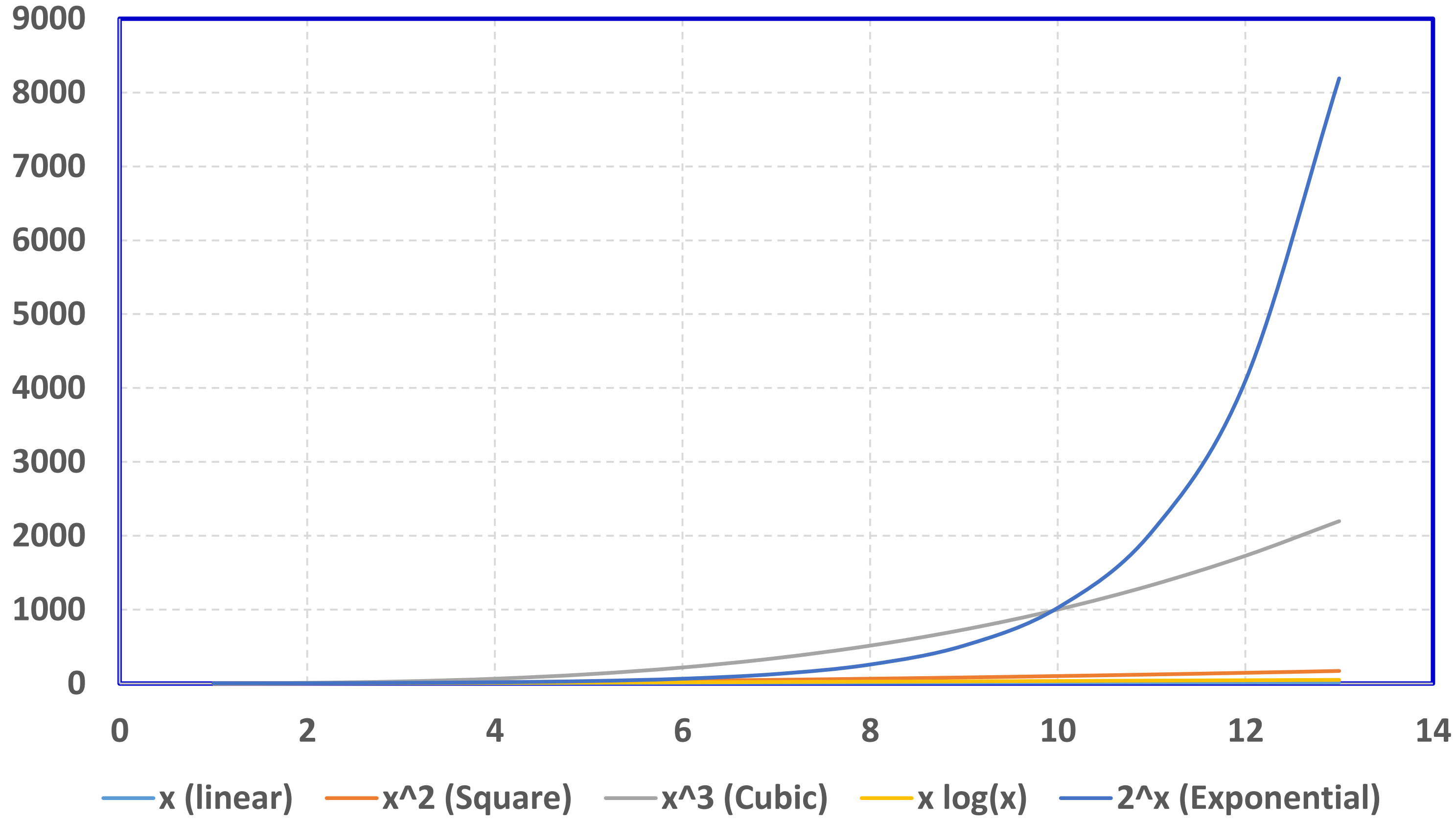
n times

n times

```
for(int i=0; i< n; i++){
      for (int j =0; j< n; j++
                  {
                     …
                  }
 }
```

O (n²)

Super-linear Complexities



Polynomial Order

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0$$

where $a_n, a_{n-1} \ldots, a_1, a_0$ are real numbers.

$\Rightarrow$ $\boldsymbol{f(x): O(x^n)}$

Example: Complexity of $x^2 + 5x$ is: $\boldsymbol{O(x^2)}$

Question

Complexity *for the following growth function.*

$$f(n) = (3n^2 + 8)(n + 1)$$

(a) O(n)

(b) $O(n^3)$

(c) $O(n^2)$

(d) O(1)

Complexity for Combination of Functions

- **Assumption**: $f_1(x)$: $O(g_1(x))$ and $f_2(x)$: $O(g_2(x))$
- **Additive Theorem**:

  $(f_1 + f_2)(x)$: $O(\max(g_1(x), g_2(x)))$.

- **Multiplicative Theorem**:

  $(f_1 f_2)(x)$ is $O(g_1(x) \times g_2(x))$.

Practical Examples

- **Additive**
  - Example: copying of array, followed by binary search

    $O(n) + O(\log(n)) \Rightarrow O(?)$

- **Multiplicative**
  - Example: a while loop with ***n*** iterations and the body in loop taking: $O(\log n) \Rightarrow O(?)$

Worst vs. Average Case Complexity

1. **Worst case**
   - Just how bad can it get: the maximal number of steps
   - Our focus in this topic

2. **Average case**
   - Amount of time expected “**usually**”
   - Not in scope of this course

3. Best case
   - The smallest number of steps
   - Generally not very useful.

Algorithm Analysis: Nested Loops

```
int nestedLoop1(int n){
    int result = 0;
    for (int i=0;i<n;i++){
        for (int j=0;j<n;j++){
            for (int k=0;k<n;k++)
                result++;
        }
    }
    return result;
}
```

Complexity = ?

In real life, this comes up in 3D imaging, video, etc., and it is **slow**!

Graphics cards are built with thousands of processors to tackle this problem!

What is Recursion?

- **Recursion**: A procedure or function calling itself
- Example: Factorial n (i.e. n!)

```
factorial (n)
{
   if (n = 0)
      return 1
   else
      return n * factorial(n-1)
}
```

Recursive Call

Practical Analysis – Recursion

- **Number of operations** in recursion depends on:
  - Number of recursive calls
  - Work done in each call

- **Examples**:
  - Factorial: how many recursive calls?
  - Fibonacci number?

Analysis – Factorial by Recursion

$T(n) = T(n-1) + c$

$= T(n-2) + 2c$ (Second recursive call)

$= T(n-3) + 3c$ (Third recursive call)

.

.

.

$= T(n-k) + kc$, until, $k = n$

After that,

$= T(n-n) + nc$

$= T(0) + nc$

$= 1 + nc$

$\Rightarrow$ $O(n)$

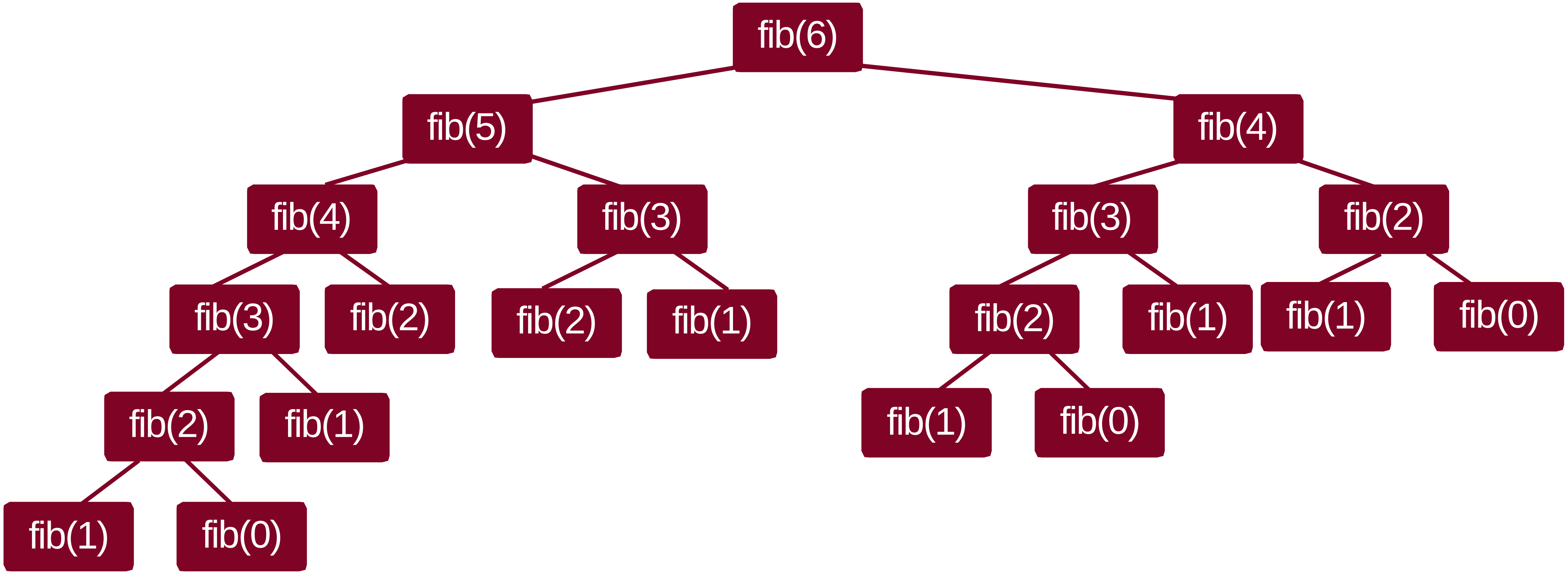
Analysis: Recursive Fibonacci (1/3)

```
fibo (n)
{
   if (n = 0)
      return 0
   else if (n = 1)
      return 1
   else
      return fibo(n-1) + fibo(n-2)
}
```

Recursive Call

- Looks nice, but has a serious flaw.
- Why?
- Let's look at the call tree for fibo(6):

Analysis: Recursive Fibonacci (2/3)



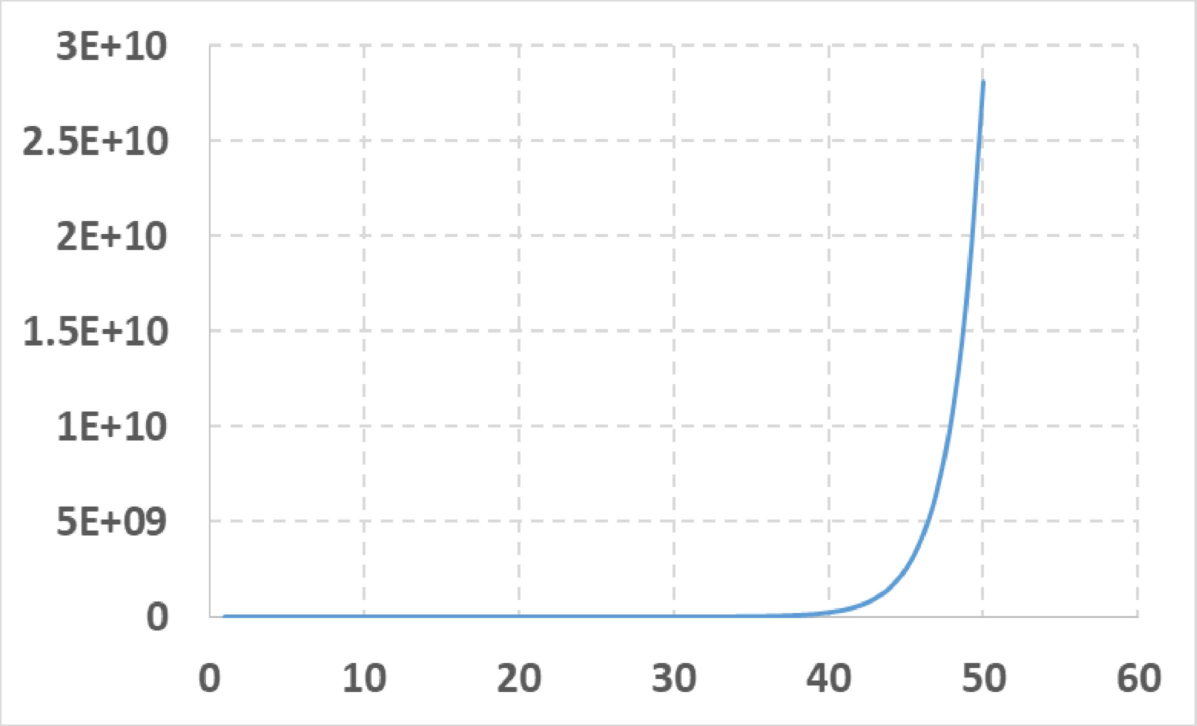
Look at all the functional duplication!
Each call is making two recursive calls, and many are duplicated!

Analysis: Recursive Fibonacci (3/3)

$T(n) = T(n-1) + T(n-2) + c$

- Need to solve this recurrence.
- Use characteristic equation: $x^2 = x + 1$
  - Roots: $(1 + \sqrt{5})/2$, $(1 - \sqrt{5})/2$

$T(n) = O((1 + \sqrt{5})/2)^n$
$= O(1.618)^n$



Exponential Order !

Differences in Complexity Matters

Assumption: An algorithm has 1000 elements, and the O(log n) version runs in 10 ns (nanoseconds)

Below is the comparison table of different order complexity

constant	*logarithmic*	*linear*	*n log n*	*quadratic*	*polynomial (k=3)*	*exponential (a=2)*
O(1)	***O(log n)***	***O(n)***	***O(n log n)***	***O(n^2)***	***O(n^k) (k≥1)***	***O(a^n) (a>1)***
1 ns	***10 ns***	***1 μs***	***10 μs***	***1 msec***	***1 sec***	***10^{292} years***

Thank you!