CS200

Computational Complexity, Orders of Magnitude

- Rosen Ch. 3.2: Growth of Functions
- Rosen Ch. 3.3: Complexity of Algorithms
- Prichard Ch. 10.1: Efficiency of Algorithms

Algorithm and Computational Complexity

- An ***algorithm*** is a finite sequence of precise instructions for performing a computation for solving a problem.

- ***Computational complexity*** measures the processing time and computer memory required by the algorithm to solve problems of a particular problem size.

Time Complexity of an Algorithm

How do we measure the **complexity** (time, space) of an algorithm? What is this a function of?

- The **size** of the problem: an integer n
  - # inputs (e.g., for sorting problem)
  - # digits of input (e.g., for the primality problem)
  - sometimes more than one integer
- We want to characterize the running time of an algorithm for increasing problem sizes by a function T(n)

Units of time

- 1 microsecond ?
- 1 machine instruction?
- # of code fragments that take constant time?

Units of time

- 1 microsecond ?

  no, too specific and machine dependent

- 1 machine instruction?

  no, still too specific and machine dependent

- # of code fragments that take constant time?

  yes

unit of space

- bit?

- int?

unit of space

- bit?

  very detailed but sometimes necessary

- int?

  nicer, but dangerous: we can code a whole program or array (or disk) in one **arbitrary** int, so we have to be careful with space analysis (take value ranges into account when needed). Better to think in terms of machine **words**

  - i.e. fixed size, 64 bit words

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, but can be an overestimate.
- Same for worst case space complexity

Why It Matters

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Measuring the efficiency of algorithms

- We have two algorithms: `alg1` and `alg2` that solve the same problem. Our application needs a fast running time.

- How do we choose between the algorithms?

Efficiency of algorithms

- Implement the two algorithms in Java and compare their running times?
- Issues with this approach:
  - How are the algorithms coded? We want to compare the algorithms, not the implementations.
  - What computer should we use? Choice of operations could favor one implementation over another.
  - What data should we use? Choice of data could favor one algorithm over another

Measuring the 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, constant time, operations the algorithm will perform for an input of given size

Example: array copy

- Copying an array with n elements requires ____ invocations of copy operations

How many steps?

How many instructions?

How many micro seconds?

Example: linear Search

```
private int linSearch(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?

Binary Search

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

What's the maximum number of steps binSearch takes ?

what's a step here?

for |A| = 31, 63, 1000

for |A| = n

Growth rates

A. Algorithm A requires $n^2 / 2$ steps to solve a problem of size n
B. Algorithm B requires $5n+10$ steps to solve a problem of size n

- Which one would you choose?

Growth rates

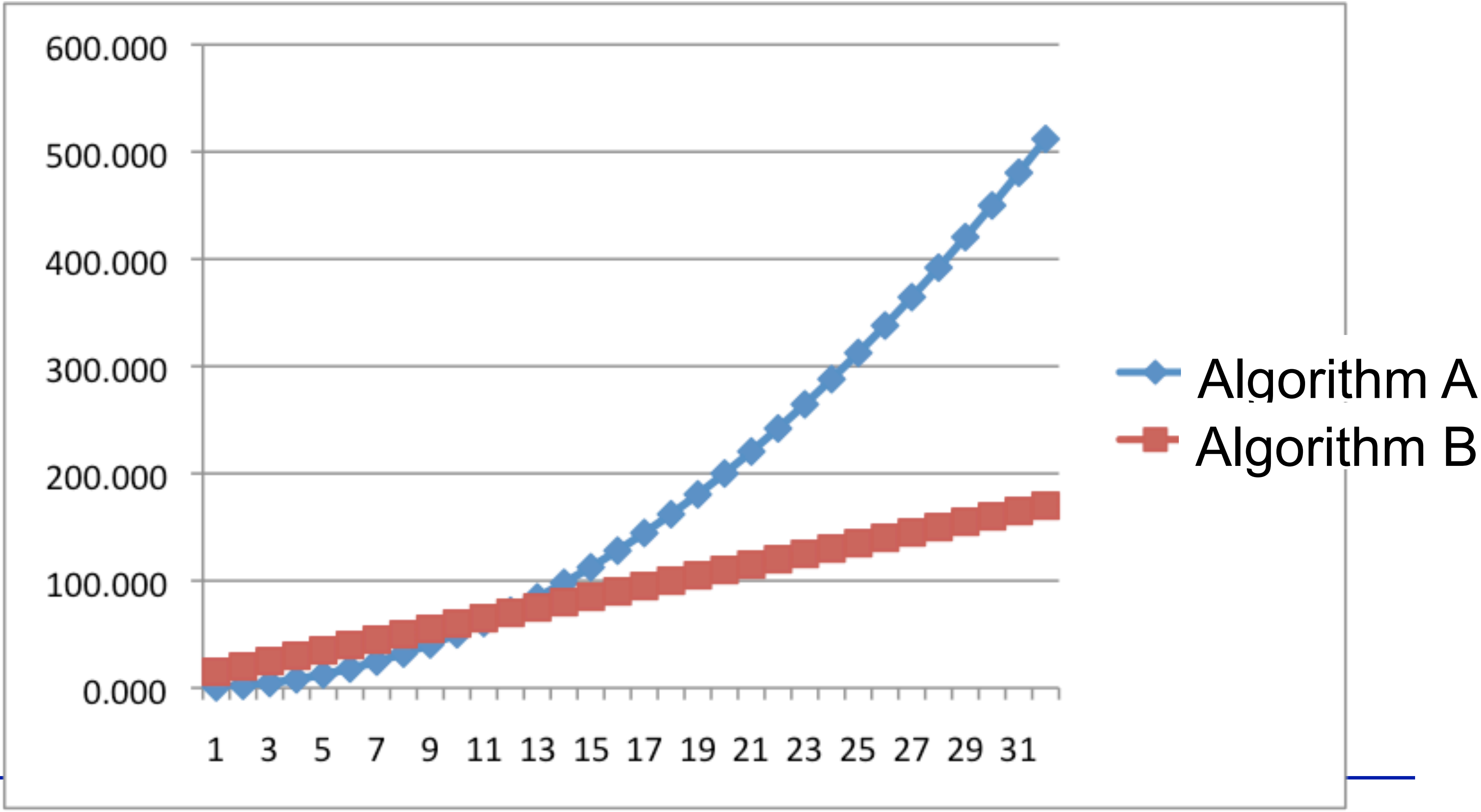
- When we increase the size of input n, how the **execution time grows** for these algorithms?

n	1	2	3	4	5	6	7	8
$n^2 / 2$	.5	2	4.5	8	12.5	18	24.5	32
5n+10	15	20	25	30	35	40	45	50

n	50	100	1,000	10,000	100,000
$n^2 / 2$	1250	5,000	500,000	50,000,000	5,000,000,000
5n+10	260	510	5,010	50,010	500,010

- We don't care about small input sizes

Growth Rates



Growth rates

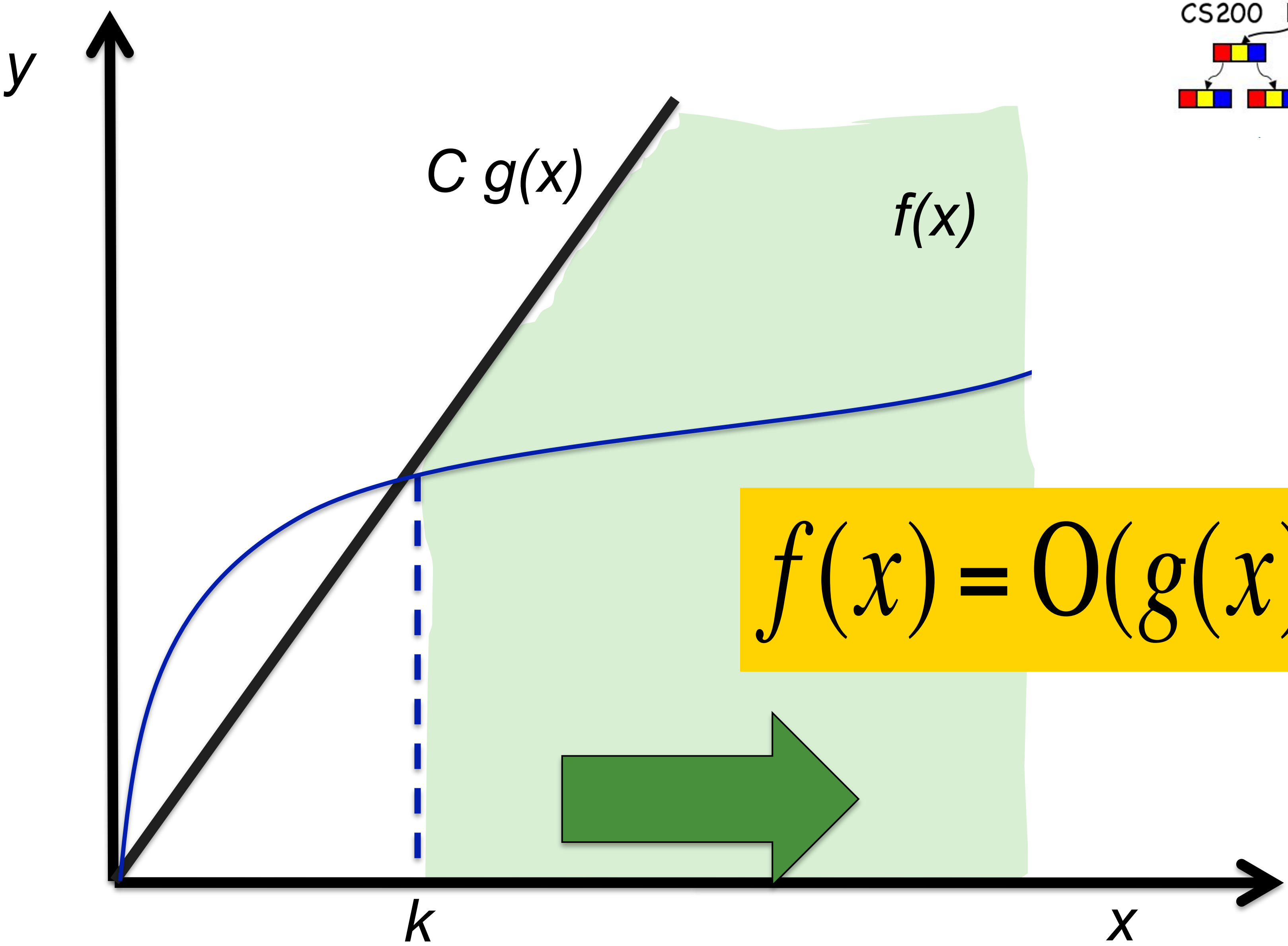
- Algorithm A requires $n^2 / 2 + 1$ operations to solve a problem of size n
- Algorithm B requires $5n + 10$ operations to solve a problem of size n
- For large enough problem size algorithm B is more efficient
- Important to know how quickly an algorithm's execution time grows as a function of program size
  - We focus on the growth rate:
    - 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 time requirement (for large n)*

Order of magnitude analysis

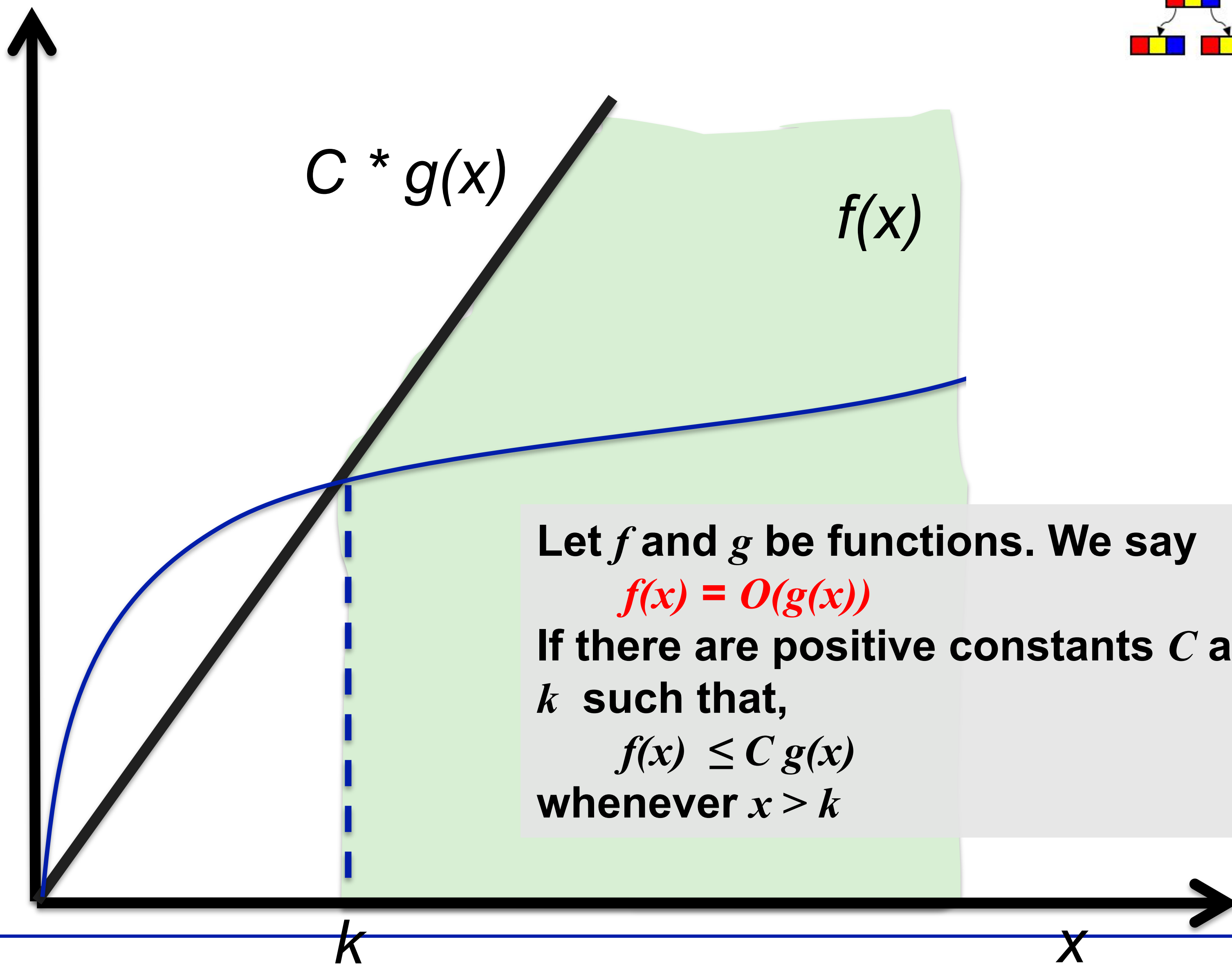
- *Big O notation: A function $f(x)$ is **$O(g(x))$** if there exist two positive constants, c and k, such that*

$$f(x) \leq c*g(x) \qquad \forall\, x > k$$

- Focus is on the shape of the function: g(x)
- Focus is on large x
- *C and k are called witnesses. There are infinitely many witness pairs (C,k)*



$$f(x) = O(g(x))$$



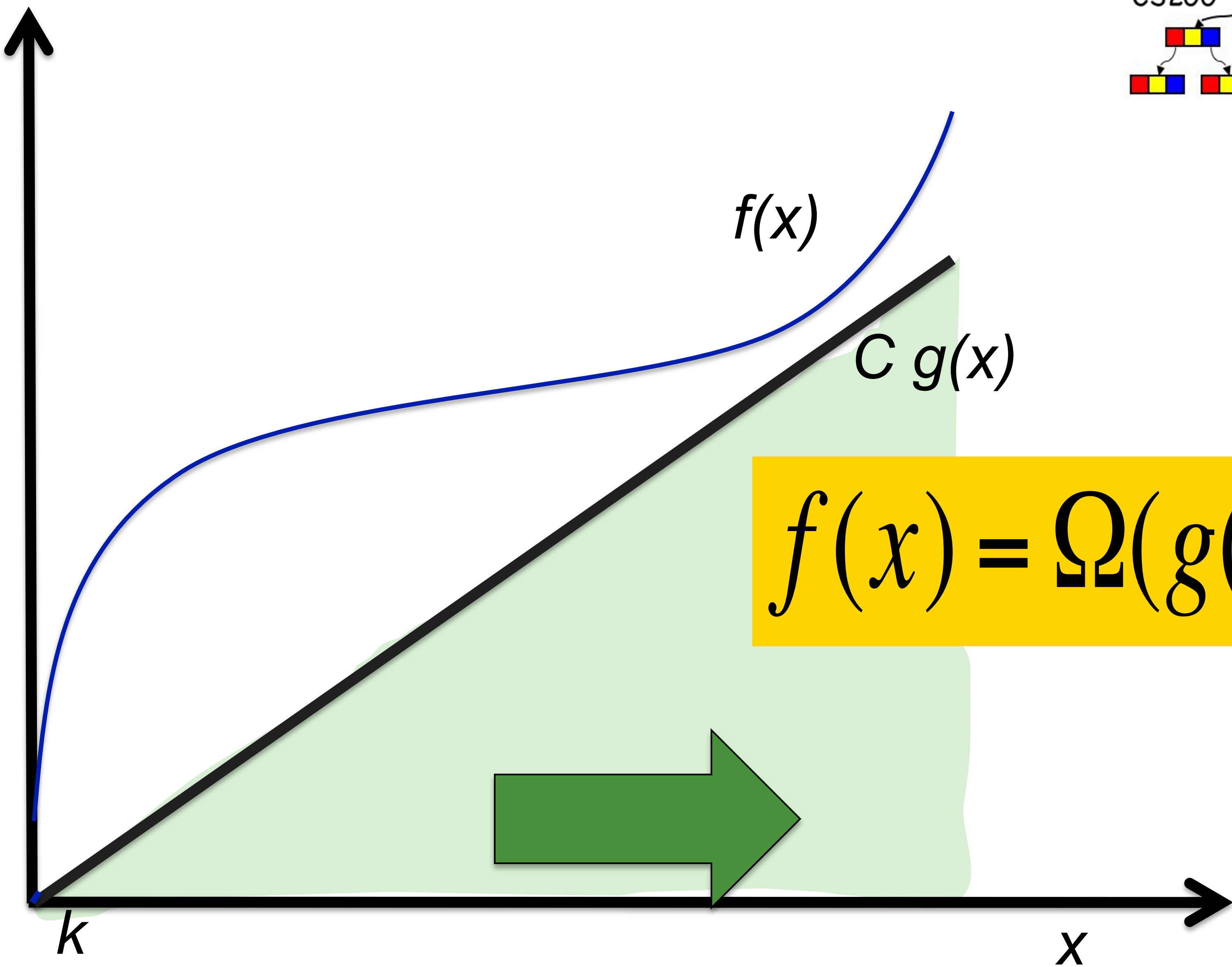
Let f and g be functions. We say

$$f(x) = O(g(x))$$

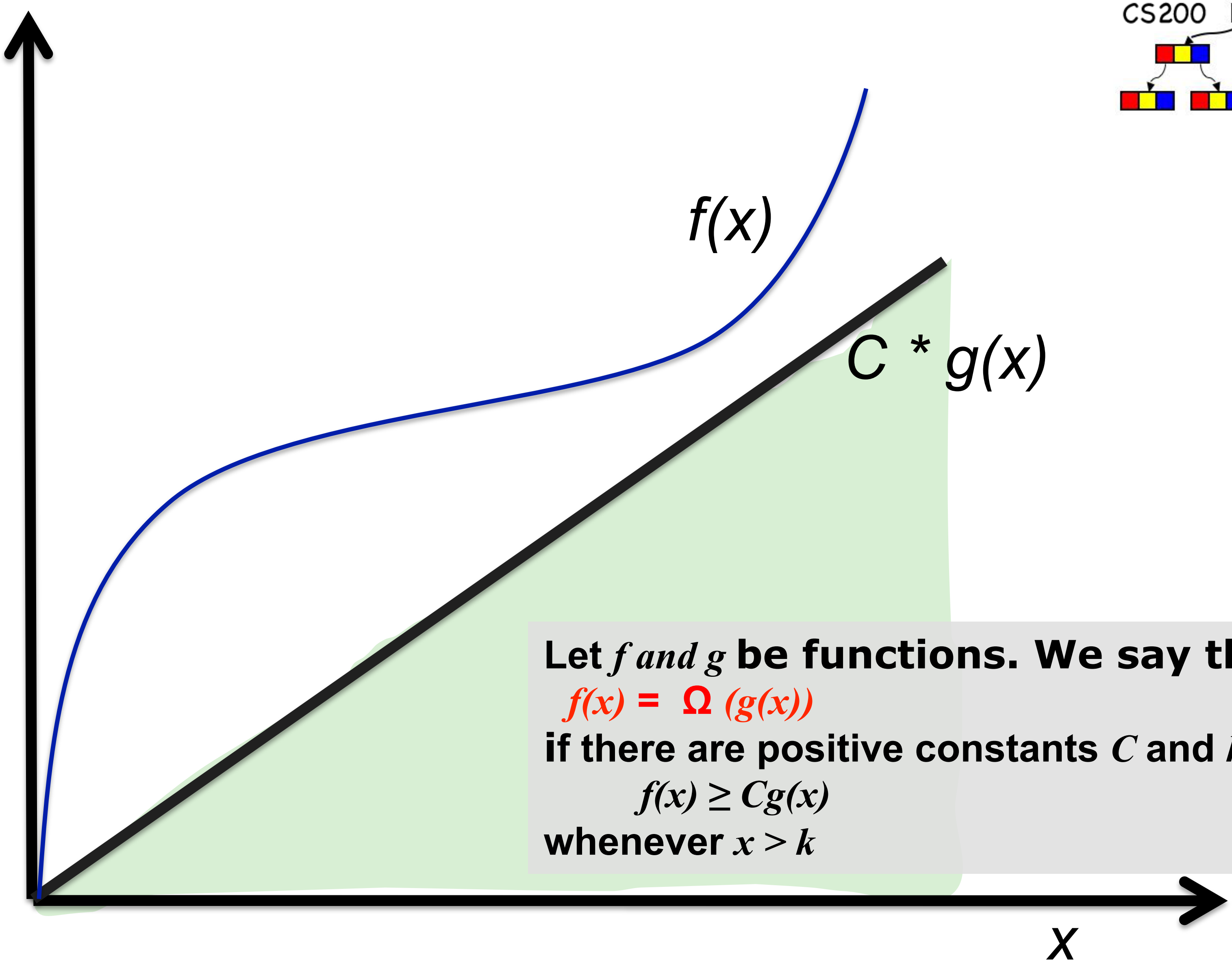
If there are positive constants C and k such that,

$$f(x) \le C\, g(x)$$

whenever $x > k$



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



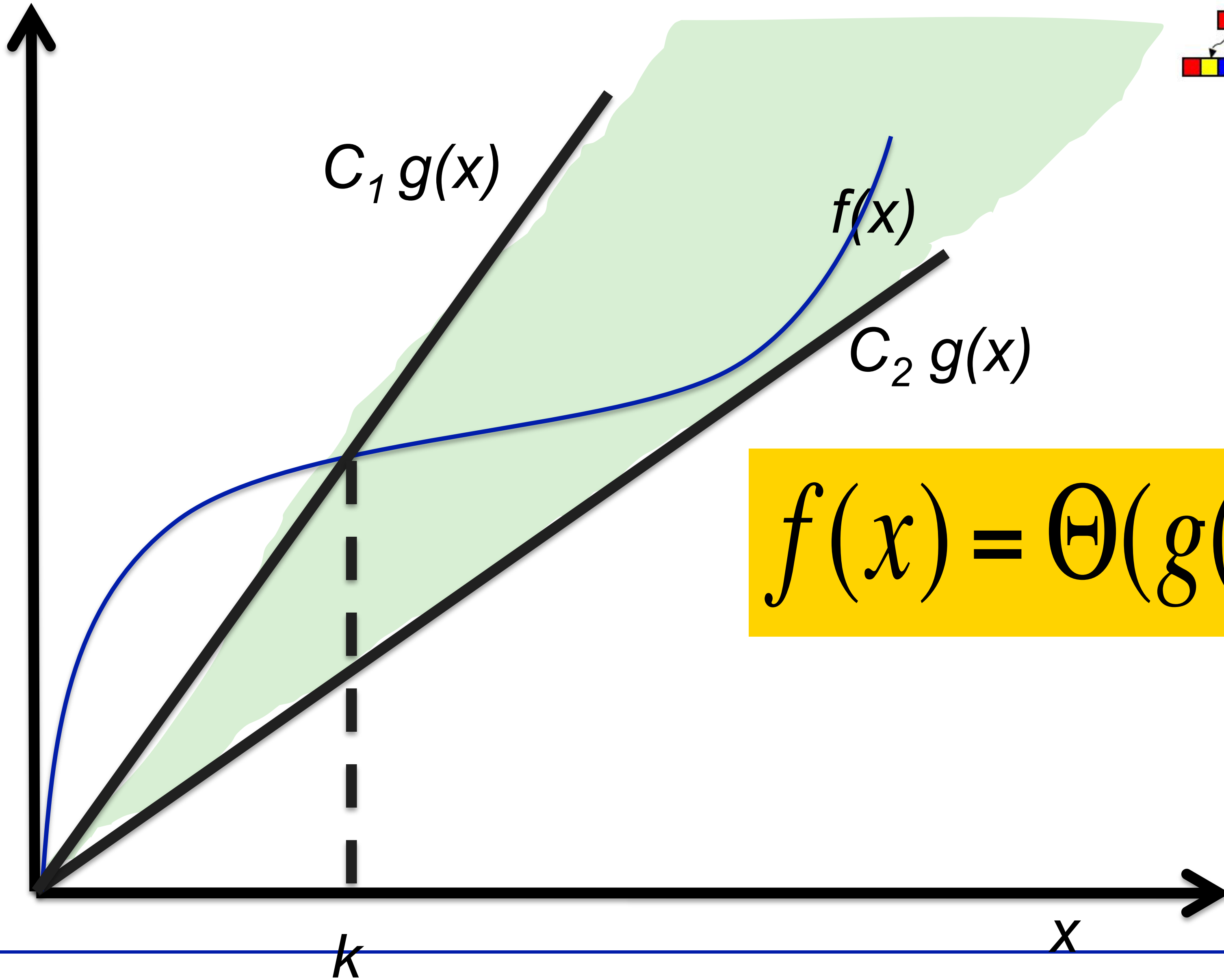
Let f and g be functions. We say that

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

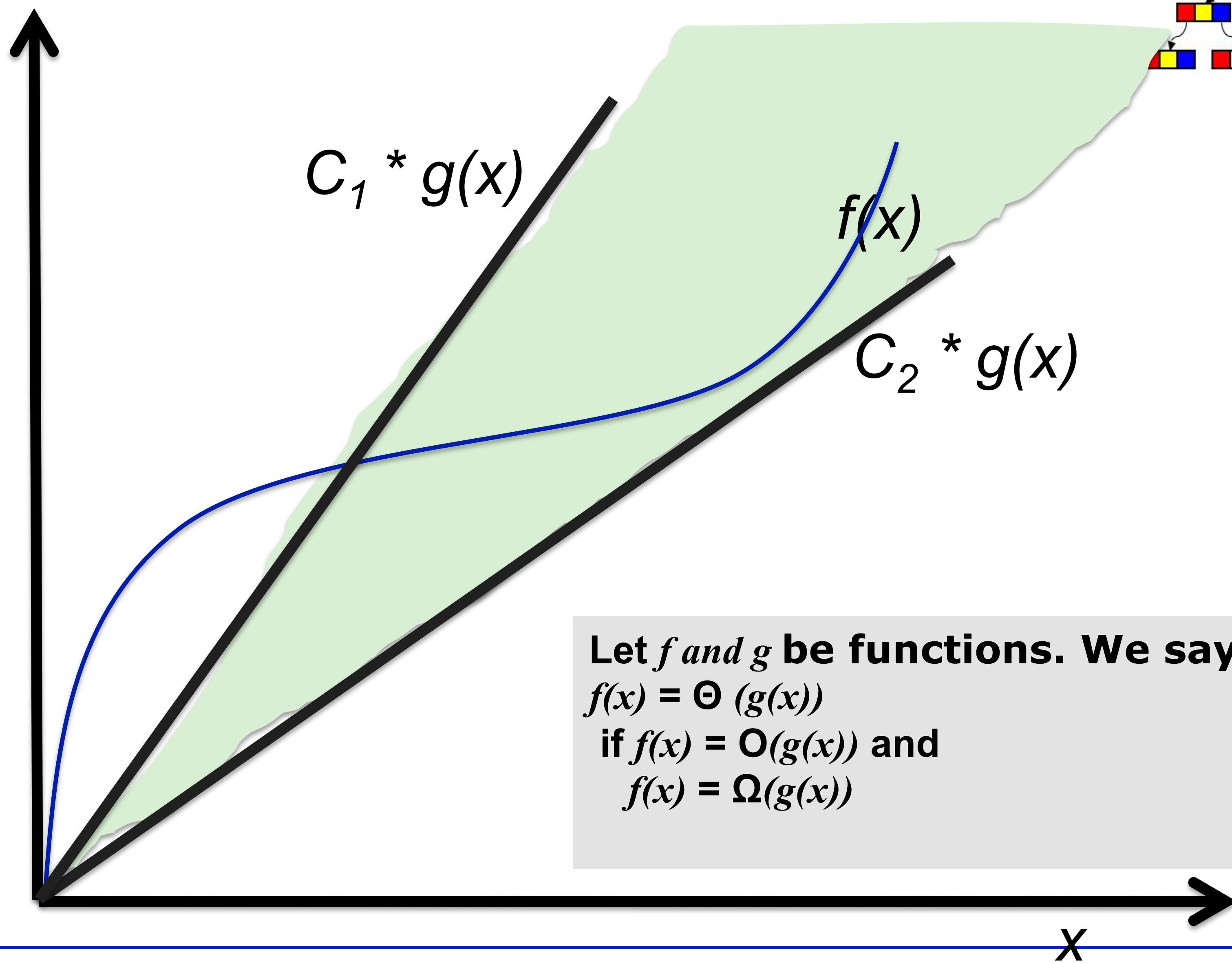
if there are positive constants C and k s.t,

$$f(x) \geq Cg(x)$$

whenever $x > k$



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



Let ***f and g*** **be functions. We say that**
f(x) **= Θ** ***(g(x))***
if ***f(x)*** **= O*(g(x))*** **and**
f(x) **= Ω*(g(x))***

Question

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

Is $f(n)$ $O(n^2)$

why?

Is $f(n)$ $\Omega(n^2)$

why?

Is $f(n)$ $\Theta(n^2)$

why?

Question

$f(x) = n + \log n$

Is $f(n)$ $O(n)$?

why?

Is $f(n)$ $\Omega(n)$?

why?

Is $f(n)$ $\Theta(n)$?

why?

Question

$f(n) = n\log n + 2n$

Is $f(n)$ $O(n)$?

why?

Is $f(n)$ $\Omega(n)$?

why?

Is $f(n)$ $\Theta(n)$?

why?

Question

$f(x) = n \log n + 2n$

Is $f(n)$ $O(n \log n)$

why?

Is $f(n)$ $\Omega(n \log n)$

why?

Is $f(n)$ $\Theta(n \log n)$

why?

Orders of Magnitude

- O (big O) is used for Upper Bounds in algorithm analysis: We use O in worst case analysis: this algorithm never takes more than this number of steps

We will **concentrate on worst case analysis**

cs320, cs420:

- Ω (big Omega) is used for lower bounds in problem characterization: how many steps does this problem at least take
- θ (big Theta) for tight bounds: a more precise characterization

Order of magnitude analysis

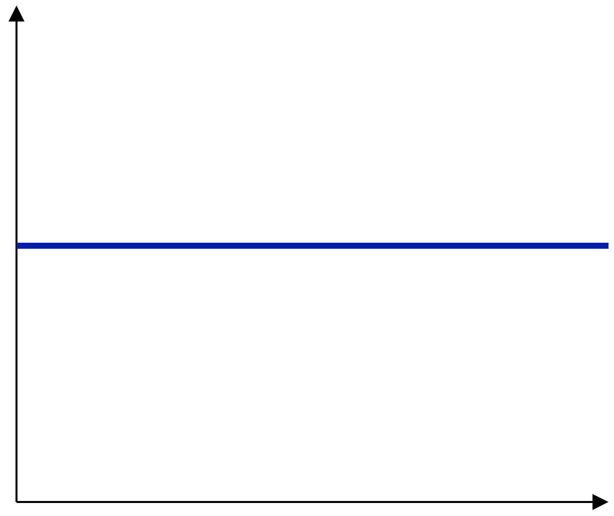
- *Big O notation: A function $f(x)$ is $O(g(x))$ if there exist two positive constants, c and k, such that*

$$f(x) \le c * g(x) \qquad \forall\ x > k$$

- c and k are witnesses to the relationship that $f(x)$ is $O(g(x))$
- If there is one pair of witnesses (c,k) then there are infinitely many (>c, >k).

Common Shapes: Constant

- *O(1)*



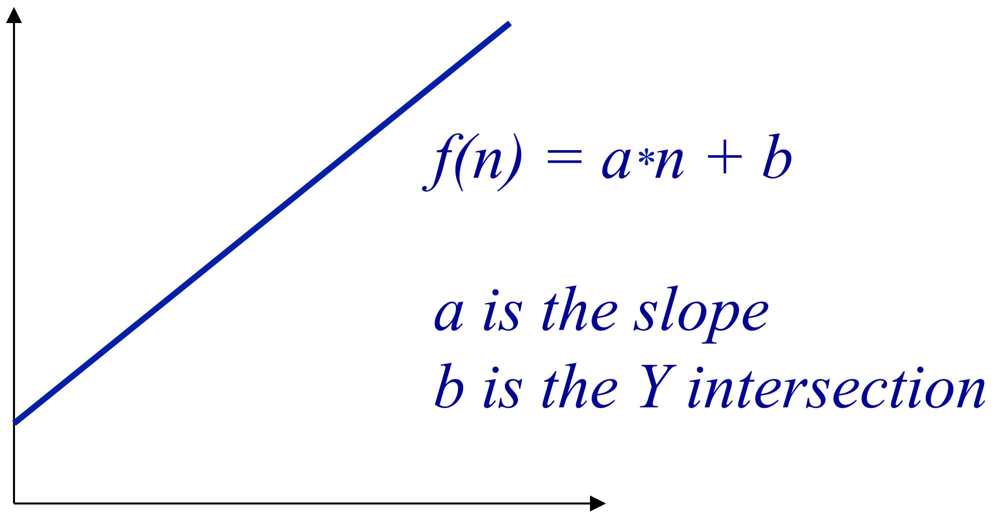
E.g.:
Any integer/double arithmetic / logic operation
Accessing a variable or an element in an array

Questions

- Which is an example of constant time operations?
    A. An integer/double arithmetic operation
    B. Accessing an element in an array
    C. Determining if a number is even or odd
    D. Sorting an array
    E. Finding a value in a sorted array

Common Shapes: Linear

- $O(n)$



Questions

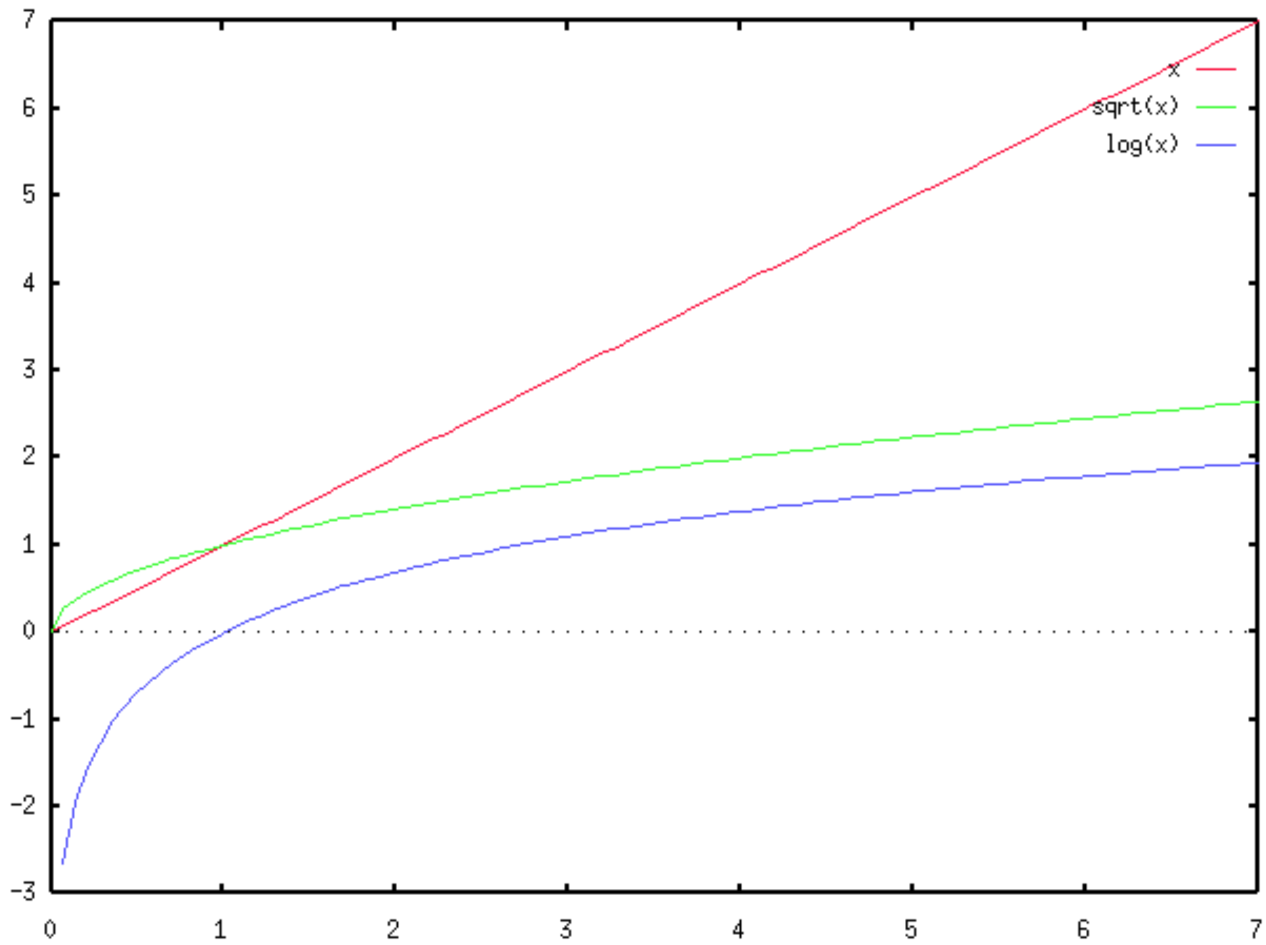
- Which is an example of a linear time operation?
  A. Summing n numbers
  B. `add(E element)` operation for Linked List
  C. Binary search
  D. `add(int index, E element)` operation for ArrayList
  E. Accessing A[i] in array A.

Linear

Example: copying an array

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

Other Shapes: Sublinear



Common Shapes: logarithm

- $log_b n$ is the number x such that $b^x = n$

  $2^3 = 8$ $\log_2 8 = 3$

  $2^4 = 16$ $\log_2 16 = 4$

- $log_b n$: (# of digits to represent n in base b) – 1
- We usually work with base 2
- $\log_2 n$: **number of times you can divide n by 2 until you get to 1**

$\log_2 n$ algorithm often break a problem in 2 halves and then solve 1 half, EXAMPLE?

Logarithms (cont.)

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

    notice that $log_b a$ *is a constant so*

    $$log_a n = O(log_b n) \text{ for any } a \text{ and } b$$

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

O(log n) in algorithms

O(log n) occurs in divide and conquer algorithms, when the problem size gets chopped in half (third, quarter,…) every step

(About) how many times do you need to divide

- 1,000 by 2 to get to 1 ?
- 1,000,000 ?
- 1,000,000,000 ?

Guessing game

I have a number between 0 and 63

How many questions do you need to find it?

is it >= 32 N

is it >= 16 Y

is it >= 24 N

is it >= 20 N

is it >= 18 Y

is it >= 19 Y

What’s the number?

Guessing game

I have a number between 0 and 63

How many questions do you need to find it?

is it >= 32	N	0
is it >= 16	Y	1
is it >= 24	N	0
is it >= 20	N	0
is it >= 18	Y	1
is it >= 19	Y	1

What's the number? 19 (010011 in binary)

Question

- Which is an example of a log time operation?
  A. Determining max value in an unsorted array
  B. Pushing an element onto a stack
  C. Binary search in a sorted array
  D. Sorting an array

Quadratic

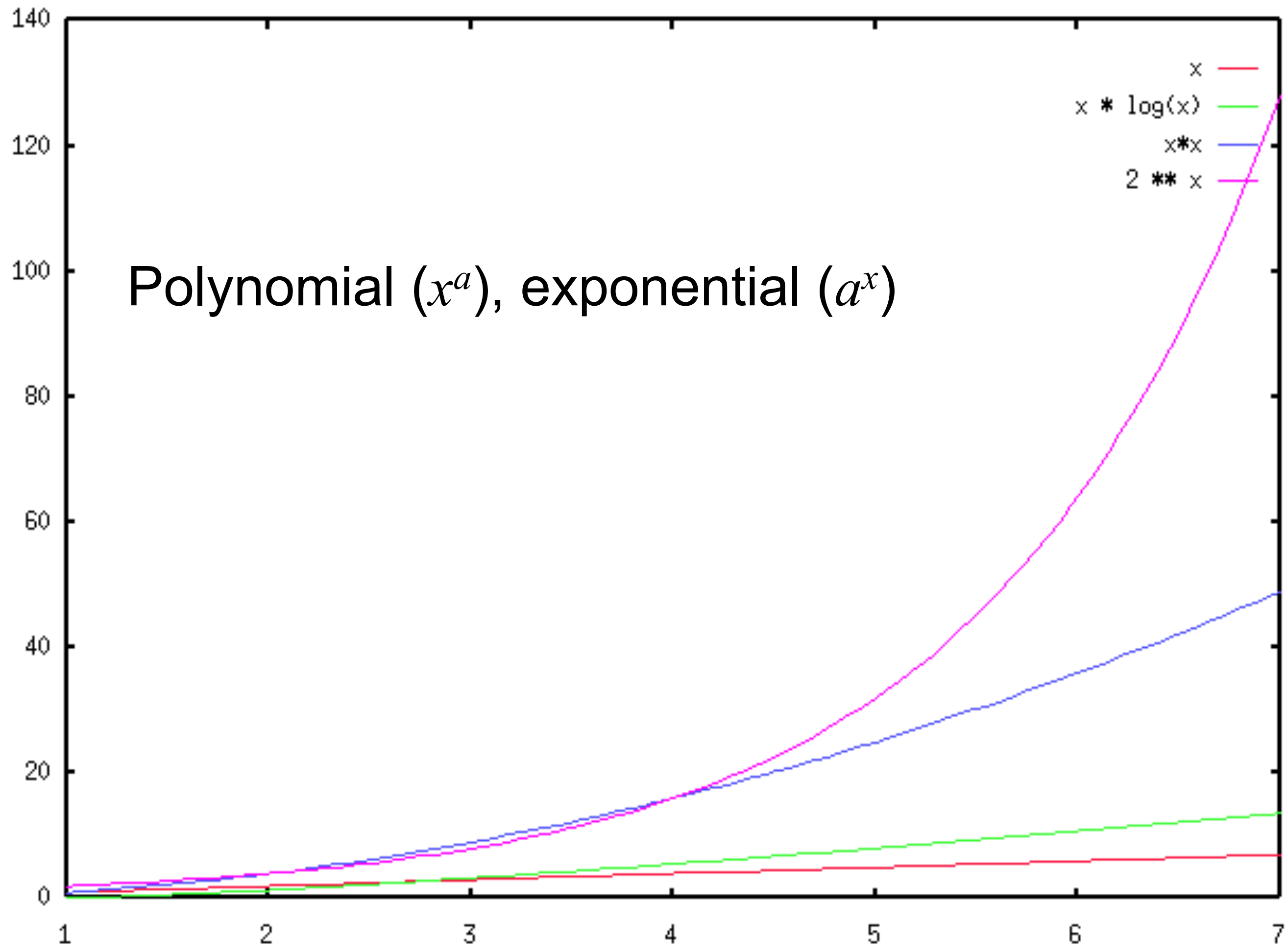
$O(n^2)$:

n times

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

n times

Other Shapes: Superlinear



Big-O for Polynomials

Theorem: Let

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

where $a_n, a_{n-1} \ldots, a_1, a_0$ are real numbers.
Then $f(x)$ is $O(x^n)$

Example: $x^2 + 5x$ is $O(x^2)$

Question

Give a Big O for the following growth function.

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

(a) $O(n)$

(b) $O(n^3)$

(c) $O(n^2)$

(d) $O(1)$

Is $f(n) = O(n^4)$?

Combinations of Functions

- Additive Theorem:

  Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$.
  Then $(f_1 + f_2)(x)$ is $O(\max(g_1(x), g_2(x)))$.

- Multiplicative Theorem:

  Suppose that $f_1(x)$ is $O(g_1(x))$ and $f_2(x)$ is $O(g_2(x))$.
  Then $(f_1 f_2)(x)$ is $O(g_1(x)g_2(x))$.

Practical Analysis – Code Combinations

- Sequential
  - Big-O bound: Steepest growth dominates
  - Example: copying of array, followed by binary search
    - n + log(n)   O(?)

- Embedded code
  - Big-O bound multiplicative
  - Example: a for loop with n iterations and a body taking O(log n)   O(?)

Worst and Average Case Time Complexity

- Worst case
  - Just how bad can it get: the maximal number of steps
  - Our focus in this course
- Average case
  - Amount of time expected “usually”
  - In this course we will hand wave when it comes to average case
- Best case
  - The smallest number of steps
  - Not very useful, e.g. sorting by repeatedly permuting the array and testing whether array is sorted: best case O(n), worst case O(n.n!)
- Example: searching for an item in an unsorted array

Question

```
1 public void insertElementAt(Object obj, int index) {
     …
2    for (i = elementCount; i > index; i--) {
3        elementData[i] = elementData[i-1];
     }
     ...
  }
```

How many times will line 3 repeat?

Practical Analysis – Dependent loops

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

i = 0:   inner-loop iters =0

i = 1:   inner-loop iters =1

.
.
.

i = n-1: inner-loop iters =n-1

Question

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

What is the Big O for this code?

A. O(n)
B. O(log n)
C. O(nlogn)
D. $O(n^2)$

Total = 0 + 1 + 2 + ... + (n-1)

f(n) = n*(n-1)/2

Loop Example

```
public int f7(int n){
        int s = n;
        int c = 0;
        while(s>1){
                s/=2;
                for(int i=0;i<n;i++)
                        for(int j=0;j<=i;j++)
                                c++;
        }
        return c;
}
```

How many outer (while) iterations?

How many inner
for i
    for j
iterations?

Big O complexity?

Practical Analysis - Recursion

- Number of operations depends on :
  - number of calls
  - work done in each call
- Examples:
  - factorial: how many recursive calls?
  - binary search?
- We will devote more time to analyzing recursive algorithms later in the course.

Example Recursive Code

```
public int divCo(int n){
        if(n<=1)
                return 1;
        else
                return 1 + divCo(n-1) + divCo(n-1);
}
```

How many recursive calls?
hint: draw the call tree

Big O complexity?

How much work per call?
What is the role of “return 1” and return 1+…” ?

Final Comments

- Order-of-magnitude analysis focuses on large problems
- If the problem size is always small, you can probably ignore an algorithm’s efficiency
- If a program responds faster than I can type, efficiency does not matter that much
- Weigh the trade-offs between an algorithm’s time requirements and its memory requirements, expense of programming/maintenance…