CSCI 2824: Discrete Structures

Lecture 18: Algorithm Complexity

HW6 is due today! (by noon) ECOT 732

Moodle Quiz - due Monday by 8am

Rachel Cox

Department of Computer Science

Algorithm Complexity

Why do we care? – – Algorithms perform computations and / or solve problems.

- We would like to know how efficiently they can solve these problems.

Different measures of efficiency:

- How long does it take to run? (time complexity)
- How much memory does it require? (space complexity)

"Time Complexity"

- Different computers run at different speeds.
- Instead we focus on the number of operations needed.
  - e.g. comparisons, additions, multiplications, etc…

Algorithm Complexity

worst case scenario

Example: What is the time complexity of a linear search?

Typical Strategy: Count the most common or expensive operation.

```
def LinearSearch (x, a):
    i = 0
    while (i<len(a) and x!=a[i]):
        i = i+1
    if (i < len(a)):
        location = i
    else:
        location = -1
    return (location)
```

- in the while loop.
  - $i < len(a)$ n
  - $x != a[i]$ n
- one final check to leave the while loop
  - $i < len(a)$ +1
- $i < len(a)$ +1

total comparisons = $n + n + 1 + 1$

$= 2n + 2$

Algorithm Complexity

n = # elements in list

$[\quad O(\quad) \quad]$

Example: What is the time complexity of a binary search?

```
def BinarySearch (x, a):
    location = -1
    left = 1; right = N
    while (left < right):
        large_left = ⌊(left+right)/2⌋
        if (x > a[large_left]):
            left = large_left + 1
        else:
            right = large_left
    if (x==a[left]): location = left
    return (location)
```

assume that $n = 2^K$ where K is some integer

- make first cut
  two lists that 2^{K-1} in length
- make second cut
  2^{K-2}
  ⋮
  $2^{K-K} = 2^0$

K steps

- After K steps check that we should really leave while loop +1
- check x == a[left] +1

comparisons $K+2$

$K = \log_2(n)$

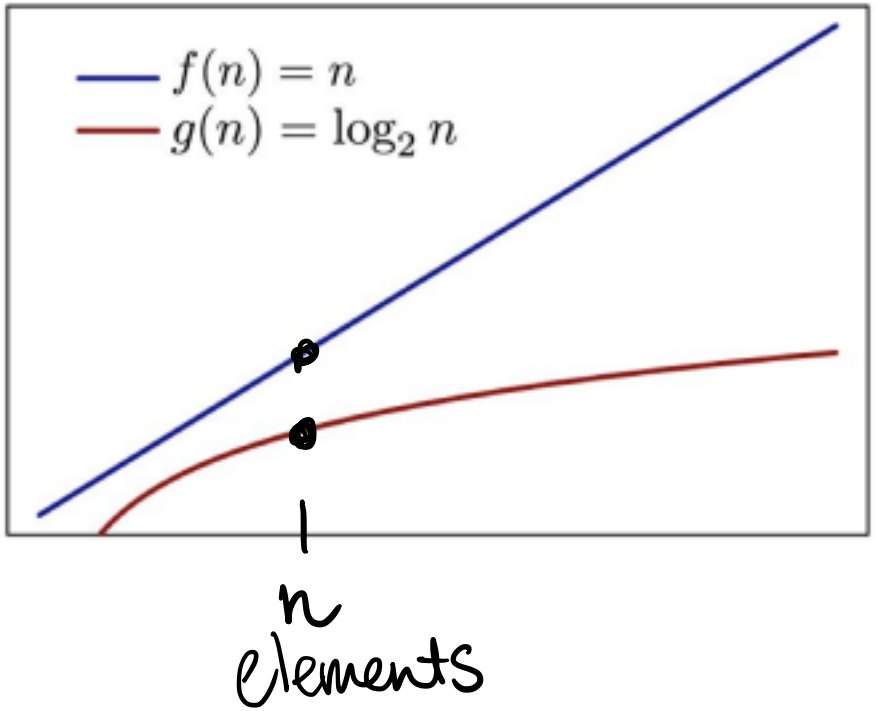
$\lceil \log_2(n) + 2 \rceil$

Algorithm Complexity

Question: Which is more efficient: Linear Search at $2n + 2$

or

Binary Search at $2 \log_2 n + 2$?



Algorithm Complexity

- Both of the previous complexity counts were for the **worst-case scenario**.
- There's a good chance that in practice, they would finish much faster.
- Instead, we can try to calculate the **average-case complexity**.

Algorithm Complexity

Example: What is the average time complexity of a linear search?

```
def LinearSearch (x, a):
    i = 0
    while (i<len(a) and x!=a[i]):
        i = i+1
    if (i < len(a)):
        location = i
    else:
        location = -1
    return (location)
```

If x is the first element
- 0 < len(a)
- x != a[0]
- i < len(a) +1

} 3

If x is the second element
- 0 < len(a)
- x != a[0]
- 1 < len(a)
- x != a[1]
- i < len(a)

} 5

3rd element

} 7

nth element

2n+(

Algorithm Complexity

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Example (continued): What is the average time complexity of a linear search?

$$\frac{3+5+7+\cdots+2n+1}{n} = \frac{1+2+1+4+1+6+\cdots+1+2n}{n}$$

$$= \frac{n+(2+4+6+8+\cdots+2n)}{n}$$

$$= \frac{n+2(1+2+3+4+\cdots+n)}{n}$$

$$= \frac{n+2\left(\frac{n(n+1)}{2}\right)}{n}$$

$$= 1+n+1 = n+2$$

Algorithm Complexity

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Example: What is the time complexity of a bubble sort?

n elements

```
def bubbleSort (x, a):
    for i in range(1, n-1):
        for j in range(1, n-i):
            if (a[j] > a[j+1]): (then swap a[j] and a[j+1])
```

1st pass n-1 comp.

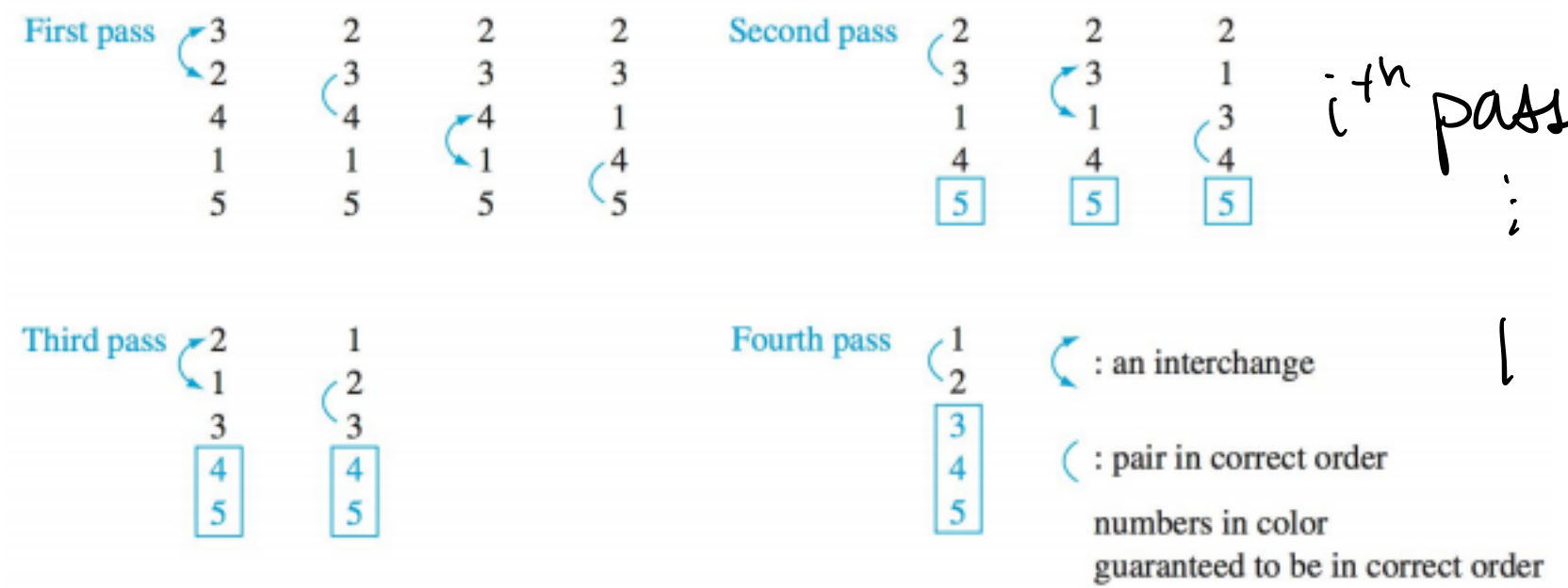
2nd pass n-2 comp.

3rd pass n-3 comp.

⋮

i^{th} pass: n-i comp.

⋮

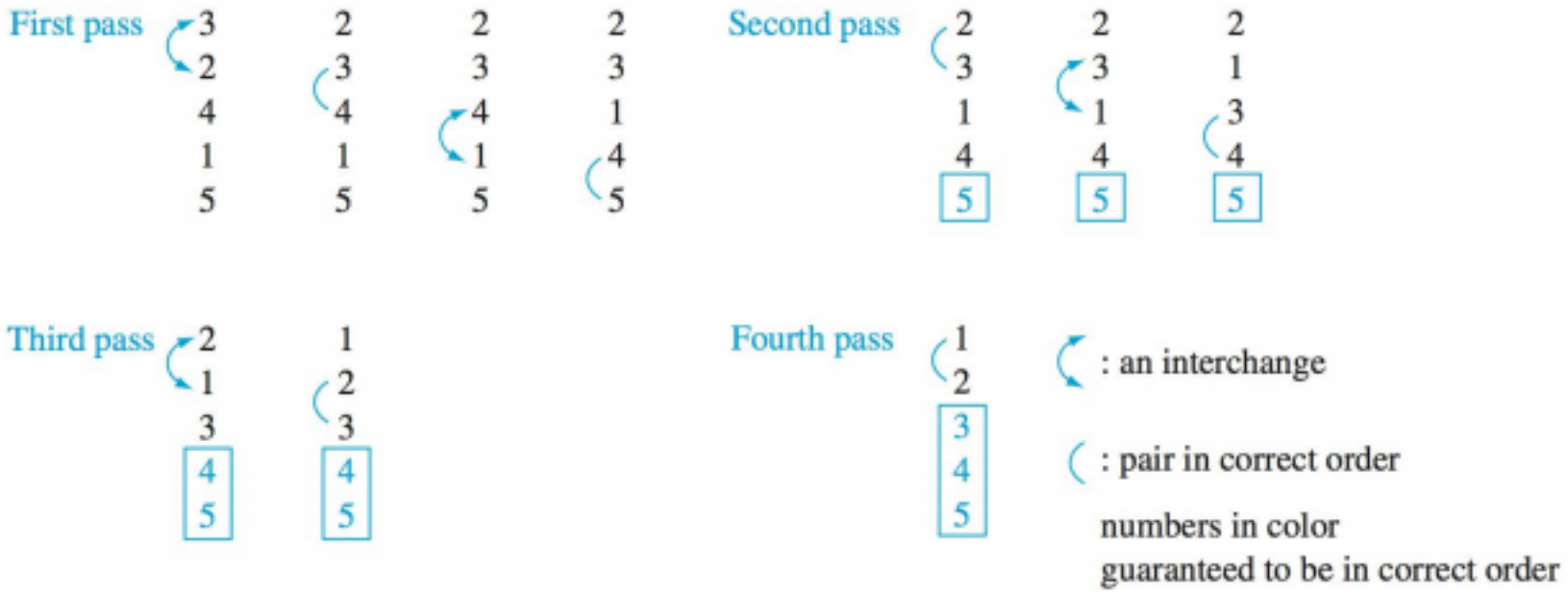


1 Comparison

total = n-1 + n-2 + ---- +3+2+1 $= (n-1)n/2 \left[= \frac{1}{2}n^2 - \frac{1}{2}n\right]$

Algorithm Complexity

Example (continued): What is the time complexity of a bubble sort?



Algorithm Complexity

Example (alternate way): What is the time complexity of a bubble sort?

```
def bubbleSort (x, a):
    for i in range(1, n-1):
        for j in range(1, n-i):
            if (a[j] > a[j+1]): (then swap a[j] and a[j+1])
```

$$\sum_{i=1}^{n-1}\left(\sum_{i=1}^{n-i} 1\right) = \sum_{i=1}^{n-1} n-i$$

A way to count if you have pseudocode:

- Count operations in inner-most loop.
- Turn loops into summations.

Caveat: here (and elsewhere), we are neglecting the comparison needed to make sure we are still within the for loops (Rosen, p. 221)

Algorithm Complexity

$$\sum_{i=1}^{n-1} n-i = (n-1)+(n-2)+(n-3)+\cdots+(n-(n-1))$$

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Example (alternate way): What is the time complexity of a bubble sort?

```
def bubbleSort (x, a):
    for i in range(1, n-1):
        for j in range(1, n-i):
            if (a[j] > a[j+1]): (then swap a[j] and a[j+1])
```

$$\sum_{i=1}^{n-1}\sum_{j=1}^{n-i} 1 = \sum_{i=1}^{n-1}\left(\sum_{j=1}^{n-i} 1\right) = \sum_{i=1}^{n-1}(n-i) = \sum_{i=1}^{n-1} n - \sum_{i=1}^{n-1} i = n(n-1) - \frac{(n-1)n}{2} = \frac{(n-1)n}{2}$$

$$= \underbrace{n+n+n+n\cdots+n}_{n-1 \text{ times}} - (1+2+3+\cdots+n-1)$$

Algorithm Complexity

$$\lim_{n \to \infty} \frac{3^n}{n^n}$$

Bubble sort: $\frac{(n-1)n}{2}$, which can be rewritten as $\frac{1}{2}(n^2 - n)$

Insert sort: $\frac{n(n+1)}{2} - 1$, which can be rewritten as $\frac{1}{2}(n^2 - n) + n - 1$

As $n \to \infty$

e.g. $n = 100$

$n = 1{,}000{,}000$

look at $n^2 - n$

n	$n^2 - n$
100	10,000 − 100
1,000,000	1,000,000,000,000 − 1,000,000

Algorithm Complexity

Example: What can you say about the performance of an algorithm with n^2 complexity, as n grows? More specifically, if I sort a list, and then sort a list that is twice as long, how do the two times compare?

$$\text{Time(short list)} = n^2$$

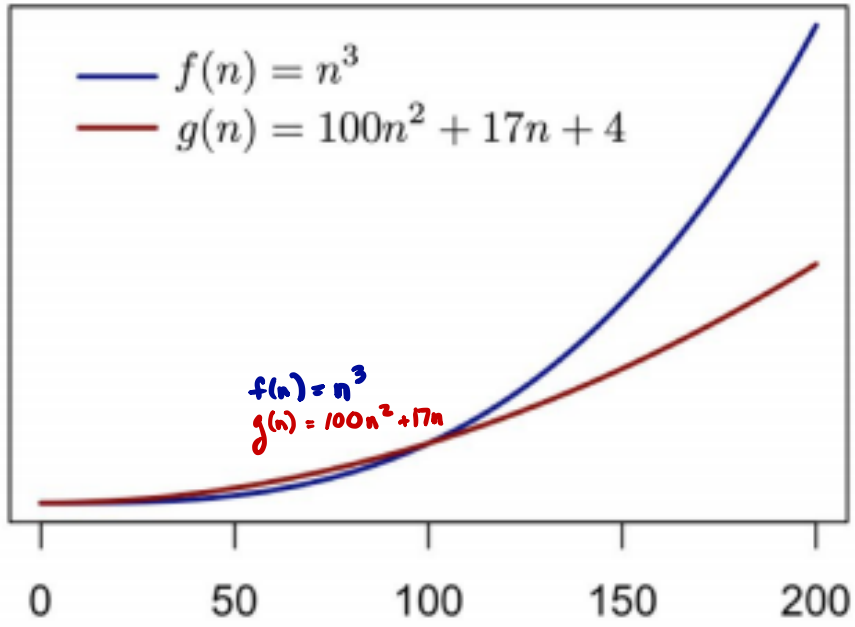
$$\text{Time(long list)} = (2n)^2 = 4n^2$$

$$\frac{\text{Time(long list)}}{\text{Time(short list)}} = \frac{4n^2}{n^2} = 4$$

Algorithm Complexity

- There are several conventions that allow us to talk about the efficiency of algorithms without writing out their precise operation counts.

- **Question:** S'pose we have two algorithms that solve the same problem.
  Algorithm A uses $100n^2 + 17n + 4$ operations.
  Algorithm B uses n^3 operations.
  Which should you use?



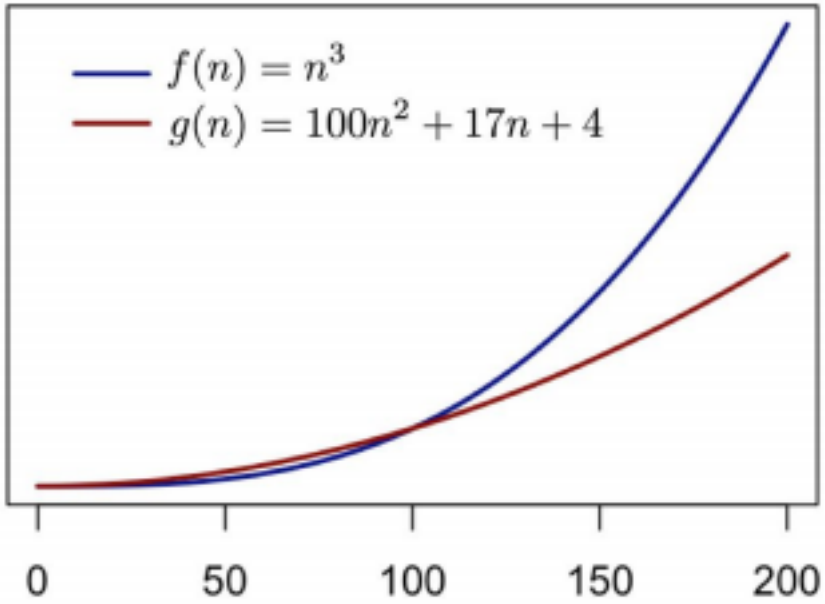
Algorithm Complexity

Definition: Let f and g be functions from the set of integers. We say that $f(n)$ is $\vartheta(g(n))$ if there are constants C and k such that

$$|f(n)| \leq C|g(n)|$$

whenever $n > k$.

big-O of $g(n)$



Algorithm Complexity

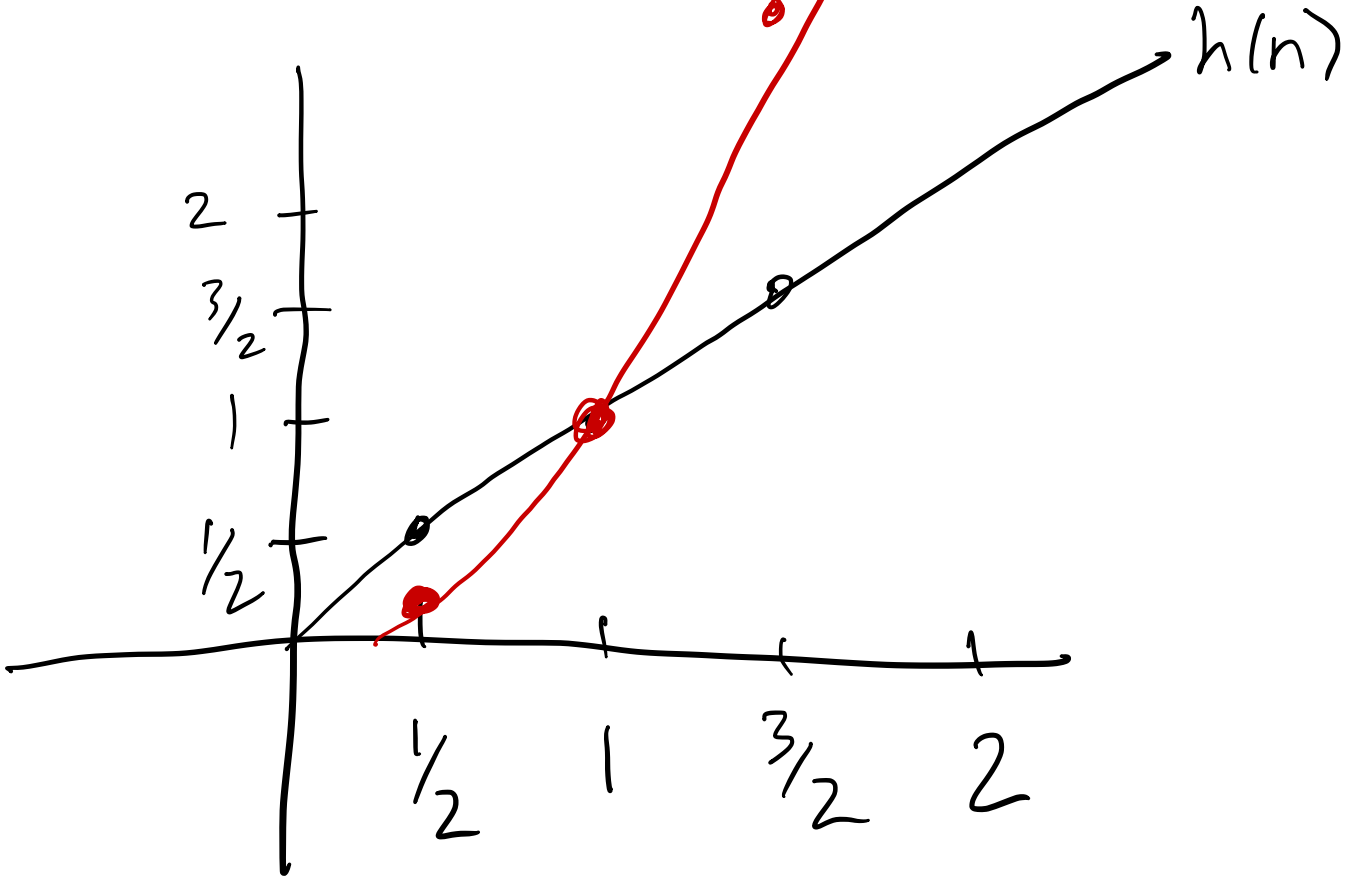
"big-O of n^2"

Example: Show that $100n^2 + 17n + 4$ is $\vartheta(n^2)$.

$h(n) = n$

$g(n) = n^2$

examine n versus n^2



We know that for $n > 1$

$n^2 > n$

Therefore (for $n > 1$)

$100n^2 + 17n + 4$

$\leq 100n^2 + 17n^2 + 4$

$\leq 100n^2 + 17n^2 + 4n^2$

$= 118n^2$

So,

Triangle Inequality: $|a+b| \leq |a| + |b|$

$$|f(n)| = |100n^2 + 17n + 4|$$

Bounding on previous page

$$\leq |100n^2 + 17n^2 + 4n^2|$$

Triangle Inequality

$$\leq |100n^2| + |17n^2| + |4n^2|$$

$$\leq |121n^2|$$

$$\leq |121|\,|n^2|$$

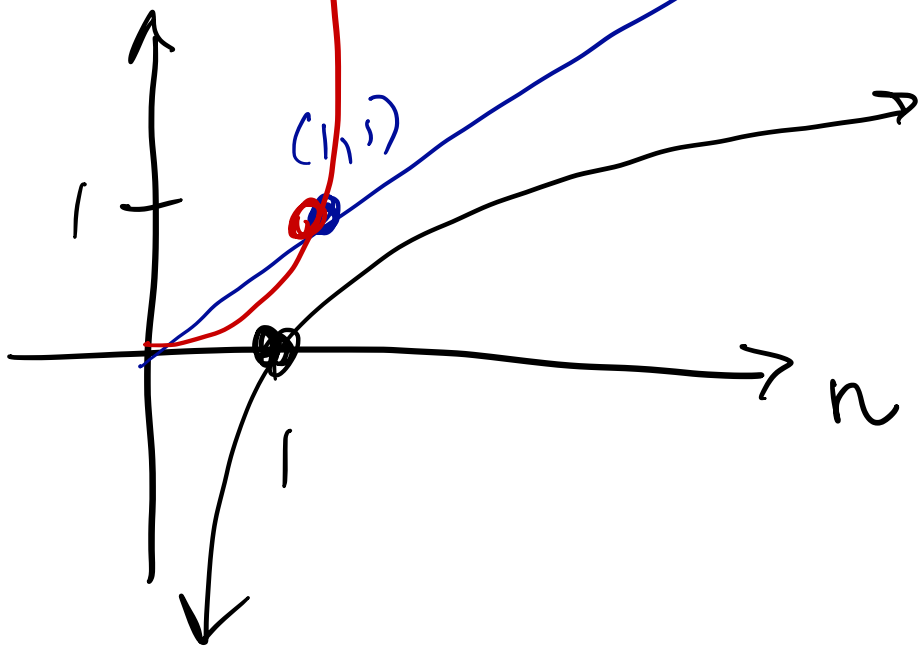
$\Rightarrow$ $f(n)$ is $\Theta(n^2)$ with $K > 1$ and $C = |121|$

Algorithm Complexity

- **Note:** $f(n)$ is $\mathcal{O}(g(n))$ means that for some C and k, $f(n)$ is bounded above by $C \cdot g(n)$
- This definition also means that $100n^2 + 17n + 4$ is $\mathcal{O}(n^3)$ and $\mathcal{O}(n^{1000})$...
- This is slightly unsatisfactory, because saying that $100n^2 + 17n + 4$ is $\mathcal{O}(n^{1000})$ is in the realm of information that is true, but not very useful.

Algorithm Complexity

Example: Give the big-O estimate of $h(n) = 2n^2 + 5\log(n)$



$$|h(n)| = |2n^2 + 5\log(n)|$$

for $n \geq 1$ $\quad n > \log(n)$

$$\begin{aligned} |h(n)| &= |2n^2 + 5\log(n)| \\ &\leq |2n^2 + 5n| \\ &\leq |2n^2 + 5n^2| \\ &= 7|n^2| \end{aligned}$$

So $h(n)$ is $O(n^2)$ for $k=1$, $C=7$

Algorithm Complexity

Theorem: Suppose that $f_1(n)$ is $\vartheta(g_1(n))$ and $f_2(n)$ is $\vartheta(g_2(n))$.
Then $(f_1 + f_2)(n)$ is $\vartheta(\max(|g_1(n)|, |g_2(n)|))$.

Theorem: Suppose that $f_1(n)$ is $\vartheta(g_1(n))$ and $f_2(n)$ is $\vartheta(g_2(n))$.
Then $(f_1 f_2)(n)$ is $\vartheta(g_1(n)g_2(n))$.

e.g. $f_1(n) = n^2$
$f_2(n) = n^3$

$f_1 f_2(n)$ would be $\mathcal{O}(n^5)$

Algorithm Complexity

So $f(n)$ is $\Theta(g(n))$ tells us that f grows no faster than g

That's a nice upper bound on the growth of f

But it would be ***really*** nice to also have a **lower bound**… to say, for example, that $f(n)$ grows ***at least as fast as*** $h(n)$

Definition: Suppose that $f(n)$ and $h(n)$ are functions from the set of integers. We say that $f(n)$ is $\Omega(h(n))$ if there are constants C and k such that

$$|f(n)| \geq C|h(n)|$$

whenever $n > k$.

big - Omega

Algorithm Complexity

Example: Give a big-Ω estimate for $100n^2 + 17n + 4$.

$100n^2 + 17n + 4 \geq 100n^2 + 17n$

Because $4 \geq 0$

$\geq 100n^2$

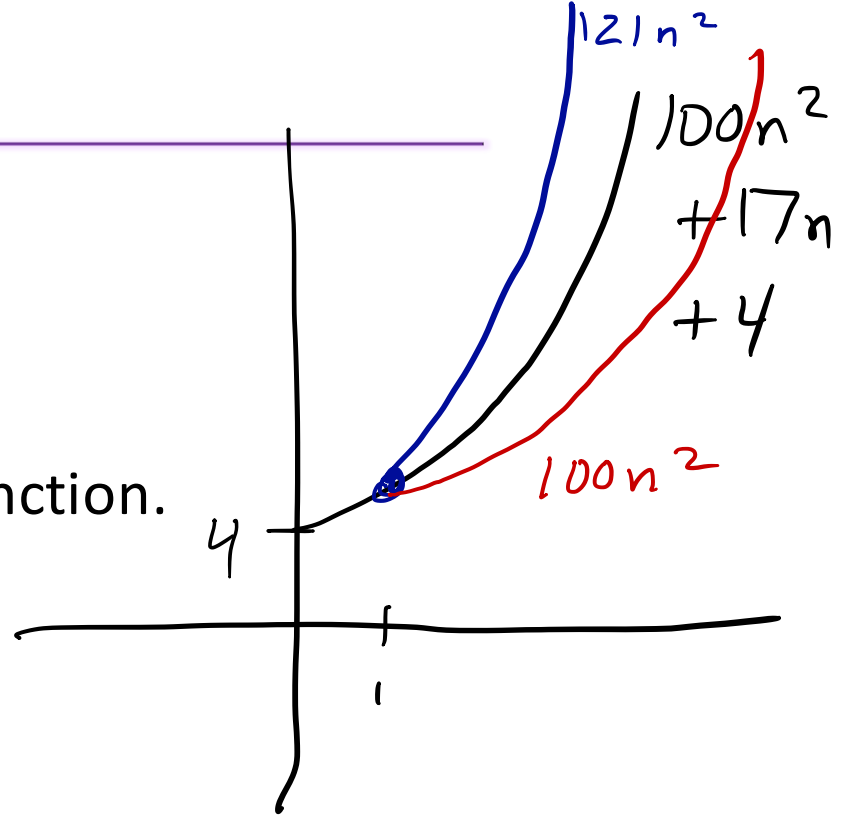
Assuming $n \geq 0$

$\Rightarrow 100n^2 + 17n + 4$ is $\Omega(n^2)$

with $k = 0$

$C = 100$

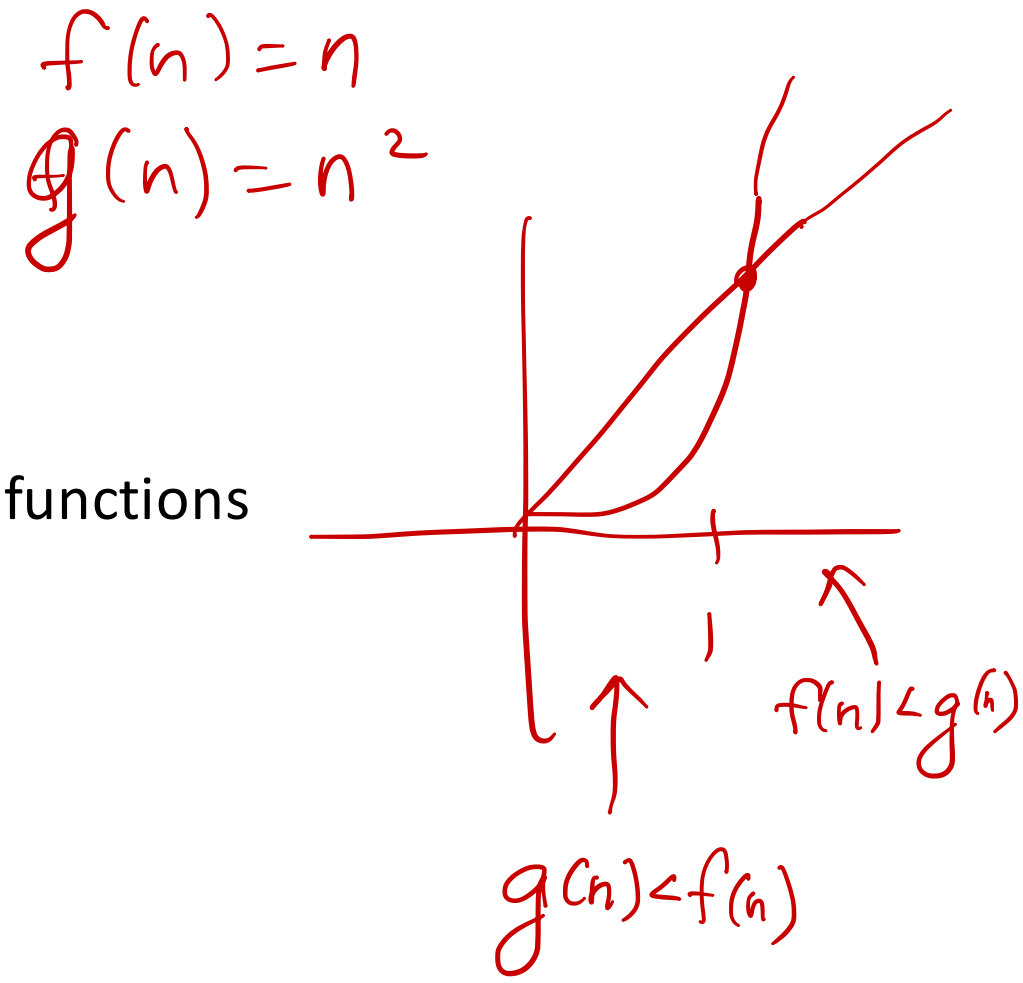
Algorithm Complexity



Big-Θ estimates of a function are the best of both worlds.

- They say we know everything about the growth of a function.
- It's smaller than "this", but larger than "that".

Definition: If $f(n)$ and $g(n)$ are both $\Theta(h(n))$, then we say that f and g have order $h(n)$, and that f and g are the same order.



What we've done:

- ❖ Complexity of Algorithms
- ❖ Estimating growth rates of functions
- ❖ Greedy Algorithms
- ❖ PB&J algorithms

Next:

Complexity and Matrices