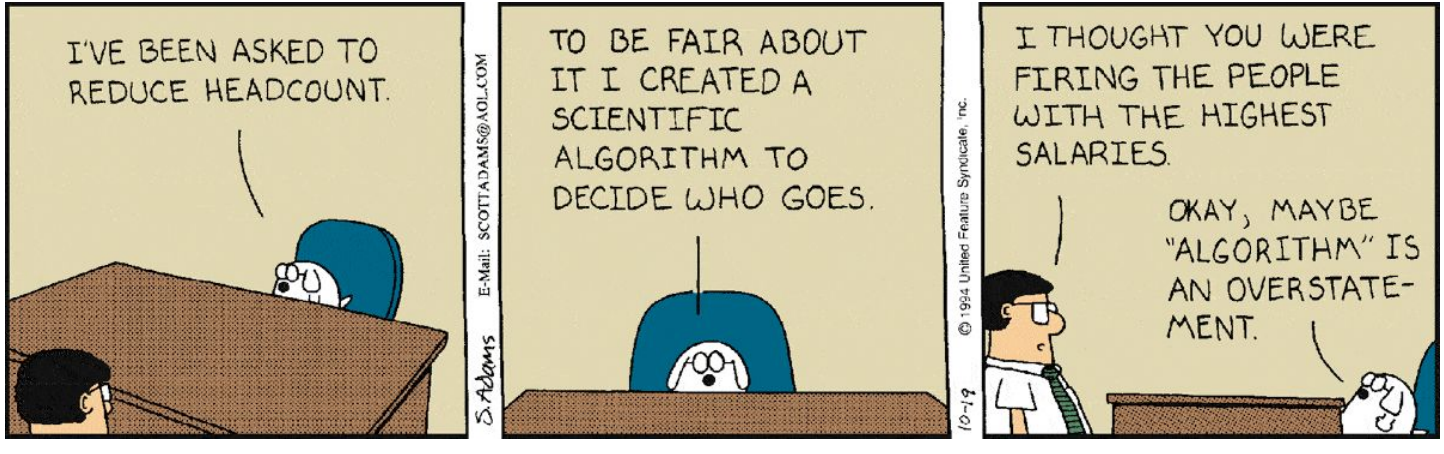
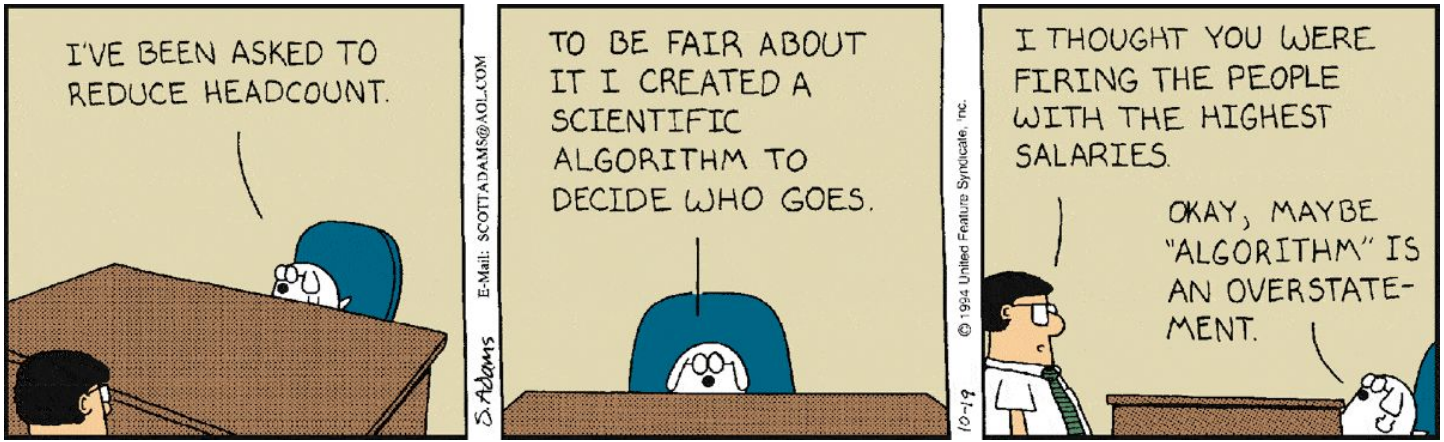
Announcements and reminders

- Homework 6 (written) is posted and is due ~~Friday 12 Oct~~ TODAY! at 12 PM Noon
- Homework 7 (Moodle) is posted and is due Friday 19 Oct at 12 PM Noon
- The CU final exam schedule is up. You must take your final exam during your scheduled final exam time.
  Tony's section: 7:30 - 10 PM, Sunday 16 Dec
  Rachel's section: 1:30 - 4 PM, Wednesday 19 Dec

Quizlet 7 due Monday 8a



Lecture 17: Algorithm Complexity

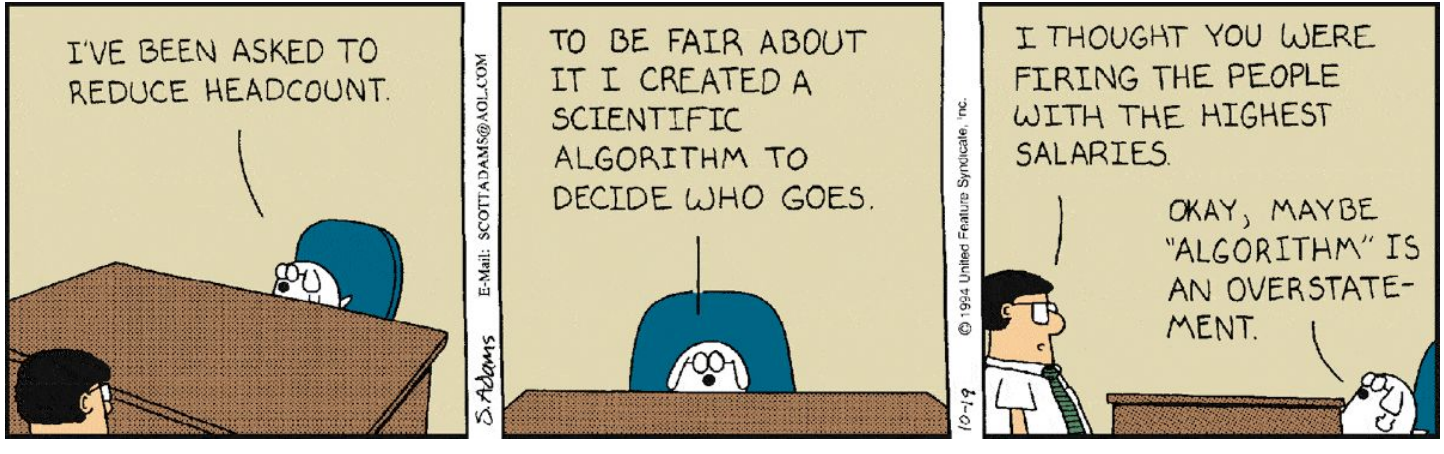


What did we do last time?

- We learned about a few ***algorithms***
  - We learned about **sorting**, **searching** and **greedy** algorithms
  - We started to think about algorithm **optimality** and **complexity**

Today:

- We dive deeper into ***algorithm complexity***
- This algorithm is nice and simple!



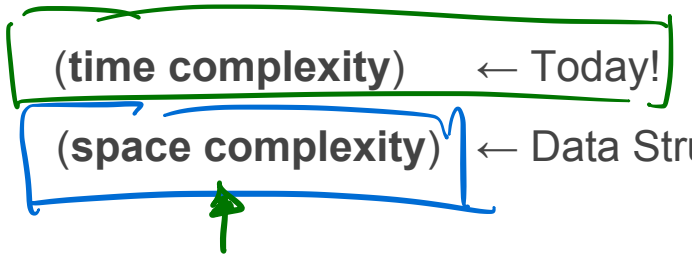
Algorithm complexity

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**) ← Today!
- How much memory does it require? (**space complexity**) ← Data Structures



"Time complexity" is a misleading phrase.

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

Algorithm complexity

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

- First: what are we going to count?
- **Typical strategy:** count the most common or most expensive operation
  - Search is mostly **comparisons**, so we count those.
- We'll start with a **worst-case** analysis. What is the worst case for a linear search?
  - **Worst-case**: *x* is not on the list.

Algorithm complexity

Example: What is the 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
```

Algorithm complexity

Example: What is the complexity of a linear search?

Assume len(a) = n

```
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
```

- Each time through the 'while' loop requires 2 comparisons:
  - ~~is~~ is $i \leq \text{len}(a)$
  - ~~is~~ is $x = a[i]$

Algorithm complexity

Example: What is the 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
```

- Each time through the 'while' loop requires 2 comparisons:
  - is $i \leq \text{len}(a)$
  - is $x = a[i]$

  } a times

- To exit the loop, we must make another comparison:
  - $i = \text{len}(a) < \text{len}(a)$

Algorithm complexity

Example: What is the 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
```

- Each time through the 'while' loop requires 2 comparisons:
  - is $i \leq \text{len}(a)$
  - is $x = a[i]$

) 2n
- To exit the loop, we must make another comparison:
  - $i = \text{len}(a) < \text{len}(a)$) + 1
- Then we must check if our final $i < \text{len}(a)$) + 1

Algorithm complexity

Example: What is the 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
```

- Each time through the 'while' loop requires 2 comparisons:
  - is $i \leq \text{len}(a)$
  - is $x = a[i]$
- To exit the loop, we must make another comparison:
  - $i = \text{len}(a) < \text{len}(a)$
- Then we must check if our final $i < \text{len}(a)$

$\Rightarrow n \times 2 + 1 + 1 = 2n + 2$ comparisons

$\Rightarrow$ **worst-case complexity is $2n + 2$**

Algorithm complexity

$n = 2^k$

$\log_2 n = \log_2(2^k) = k \log_2 2$

$\log_2 n = k$

$\log_2 2 = 1$

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

- Again, we'll count **comparisons**

[1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6, 0, 0]

log = ln

Simplifying assumption: Assume that the number of list items is a power of 2.

- Let $n = 2^k$ for some integer k and note that $k = \log_2 n$
- Obviously, not every list we would search is a power of 2 in size.
  But we could just add a bunch of 0s (for example) to the end to make it a power of 2.
  And since we're searching a longer list, we are still getting an estimate of **worst-case** complexity.

Algorithm complexity

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

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

Algorithm complexity

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

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

First time through the 'while' loop requires 2 comparisons:

- is *left* < *right*
- is *x* > *a*[*large_left*]

Algorithm complexity

$2^k \to 2^{k-1} \to 2^{k-2} \to \cdots \to 2^1 \to 1$

(1) (2) (k)

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

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

2k

First time through the 'while' loop requires 2 comparisons:

- is *left* < *right*
- is *x* > *a*[*large_left*]

After first step, repeat on list of size 2^{k-1}

- After ***k* steps** (2 comps each), we have just **one element left**
- **+1** comp for *left* < *right* to exit loop
- **+1** comp to test *x*==*a*[*left*] other order

$2k+2$ total comps.

Algorithm complexity

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

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

k times

First time through the 'while' loop requires 2 comparisons:

- is *left* < *right*
- is *x* > *a*[*large_left*]

After first step, repeat on list of size 2^{k-1}

- After ***k* steps** (2 comps each), we have just **one element left**
- **+1** comp for *left* < *right* to exit loop
- **+1** comp to test *x*==*a*[*left*]

⇒ $2k + 1 + 1 = 2k + 2$ comparisons

$= 2 \log_2 n + 2$

⇒ **worst-case complexity is $2 \log_2 n + 2$**

Algorithm complexity

Question: Which is more efficient:

LinearSearch @ $2n + 2$ or BinarySearch @ $2 \log_2 n + 2$?

➢ Well, the +2 and 2x are both on each, so they don't really matter:

LinearSearch @ n or BinarySearch @ $\log_2 n$?

➢ So we need to compare n and $\log_2 n$

Algorithm complexity

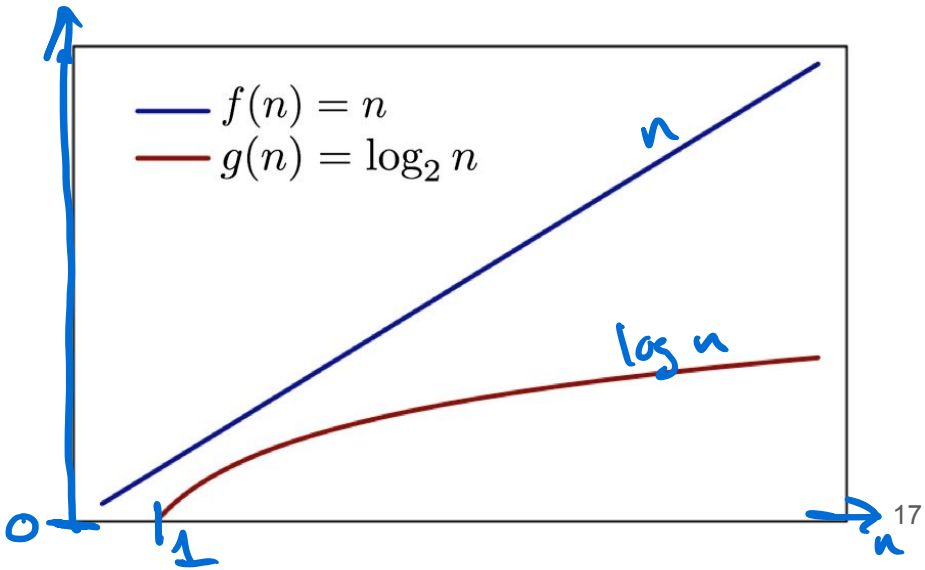
Question: Which is more efficient:

LinearSearch @ $2n + 2$ or BinarySearch @ $2 \log_2 n + 2$?

➢ Well, the +2 and 2x are both on each, so they don't really matter:

LinearSearch @ n or BinarySearch @ $\log_2 n$?

➢ So we need to compare n and $\log_2 n$



Algorithm complexity

Polynomials: n, n^2, n^3 ---

Question: Which is more efficient:

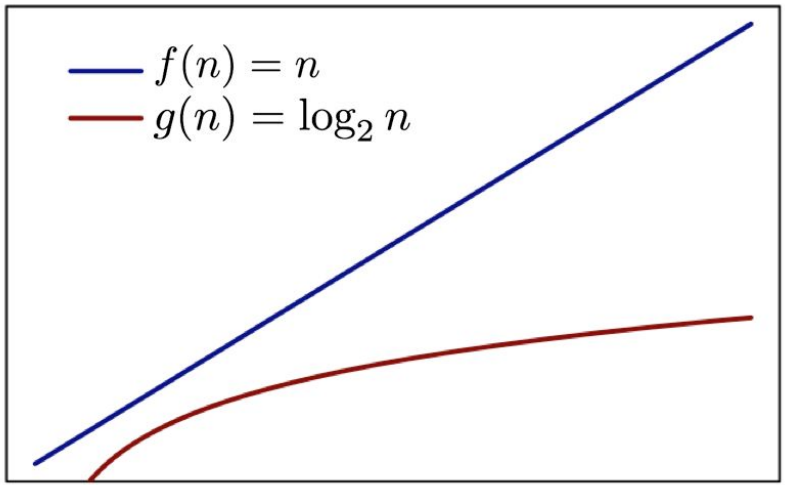
LinearSearch @ $2n + 2$ or BinarySearch @ $2 \log_2 n + 2$?

➢ Well, the +2 and 2x are both on each, so they don't really matter:

LinearSearch @ n or BinarySearch @ $\log_2 n$?

➢ So we need to compare n and $\log_2 n$

➢ **Rule:** Logarithms grow slower than all polynomials (including n)



Algorithm complexity

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

Algorithm complexity

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

len(a) = n

```
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
```

- Assume an equal probability that *x* is at any position in the list.
- Average the complexities for each possible position of *x*

  If *x* = *a*[1], need 3 comparisons…

  If *x* = *a*[2], need 5 comparisons…

  If *x* = *a*[3], need 7 comparisons…

  If *x* = *a*[*i*], need 2*i* + 1 comparisons…

Algorithm complexity

FOND MEMORY: $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

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

So the average of these *n* possibilities is then:

$$\frac{3+5+7+\cdots+(2n+1)}{n}$$

ops if a_1, a_2, a_3

$$= \frac{\overbrace{(1+2)}^{3}+\overbrace{(1+4)}^{5}+\overbrace{(1+6)}^{7}+\ldots+(1+2n)}{n}$$

$$= \frac{(1+1+1+\ldots+1)+(2+4+6+\ldots+2n)}{n}$$

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

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

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

Algorithm complexity

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

So the average of these *n* possibilities is then:

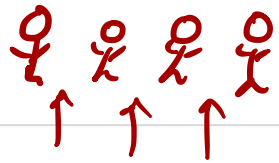
$$\frac{3+5+7+\cdots+(2n+1)}{n} = \frac{2(1+2+3+\cdots+n)+n}{n}$$

Fond memory: $$1+2+3+\cdots+n = \frac{n(n+1)}{2}$$

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

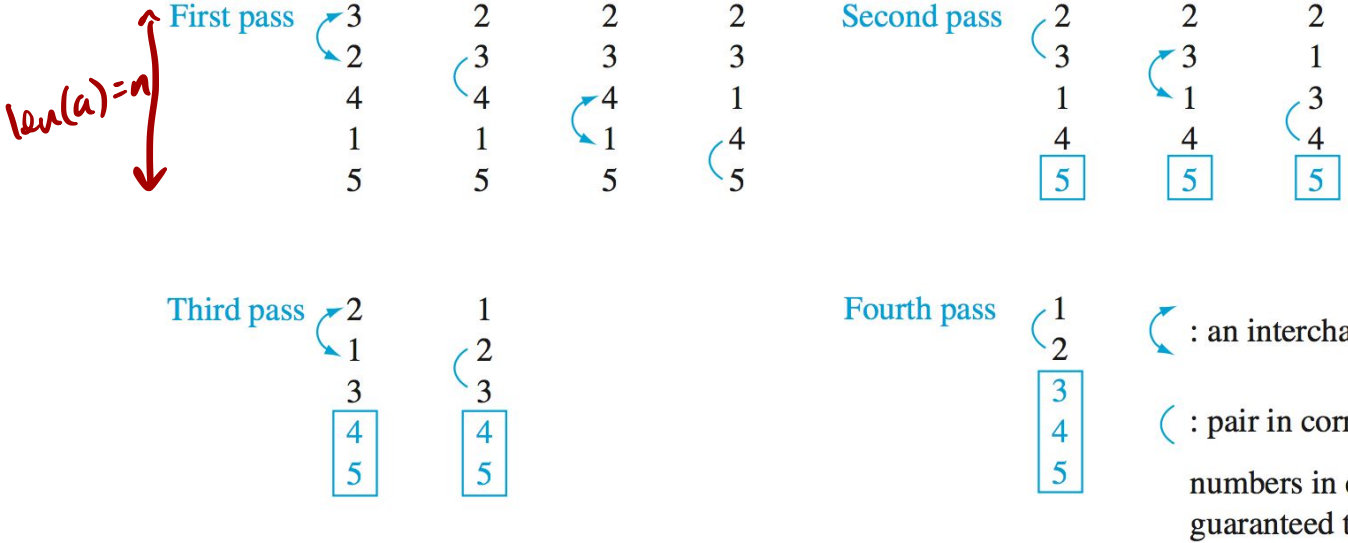
Which means the **average** time complexity of `LinearSearch` is ***n* + 2**

Algorithm complexity



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

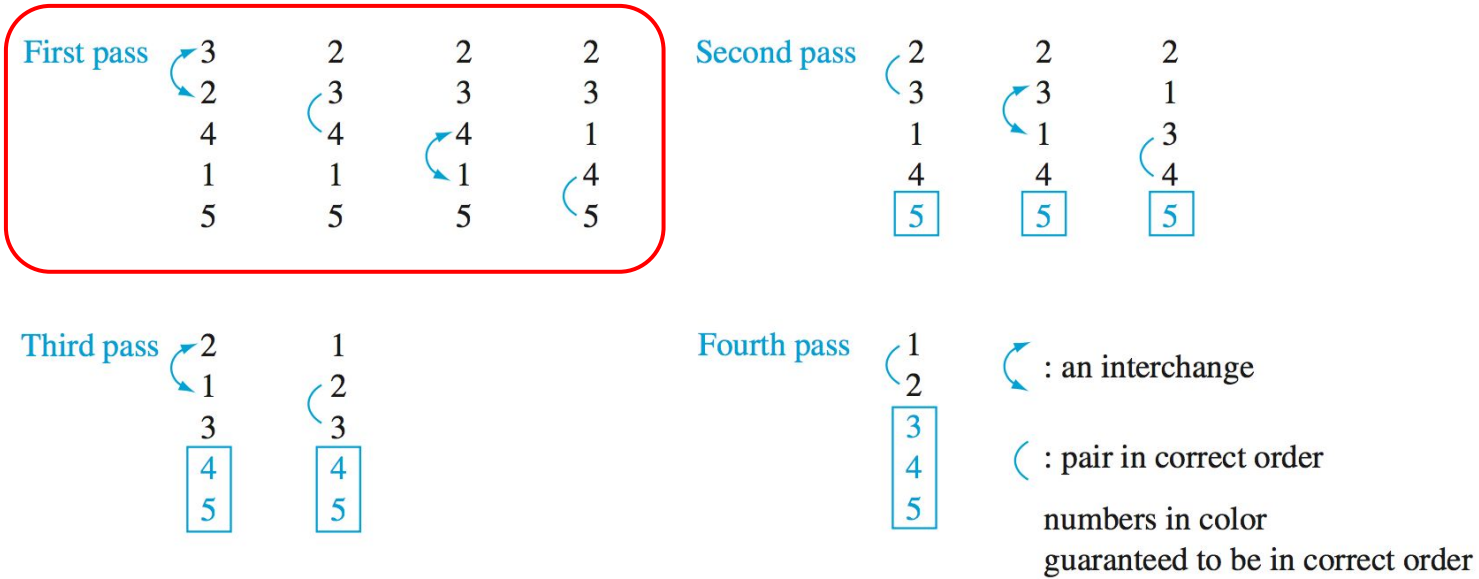
```
def bubbleSort (x, a):
    for i in range(0, N-1):
        for j in range(0, N-i):
            if (a[j] > a[j+1]): (then swap a[j] and a[j+1])
```



Algorithm complexity

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

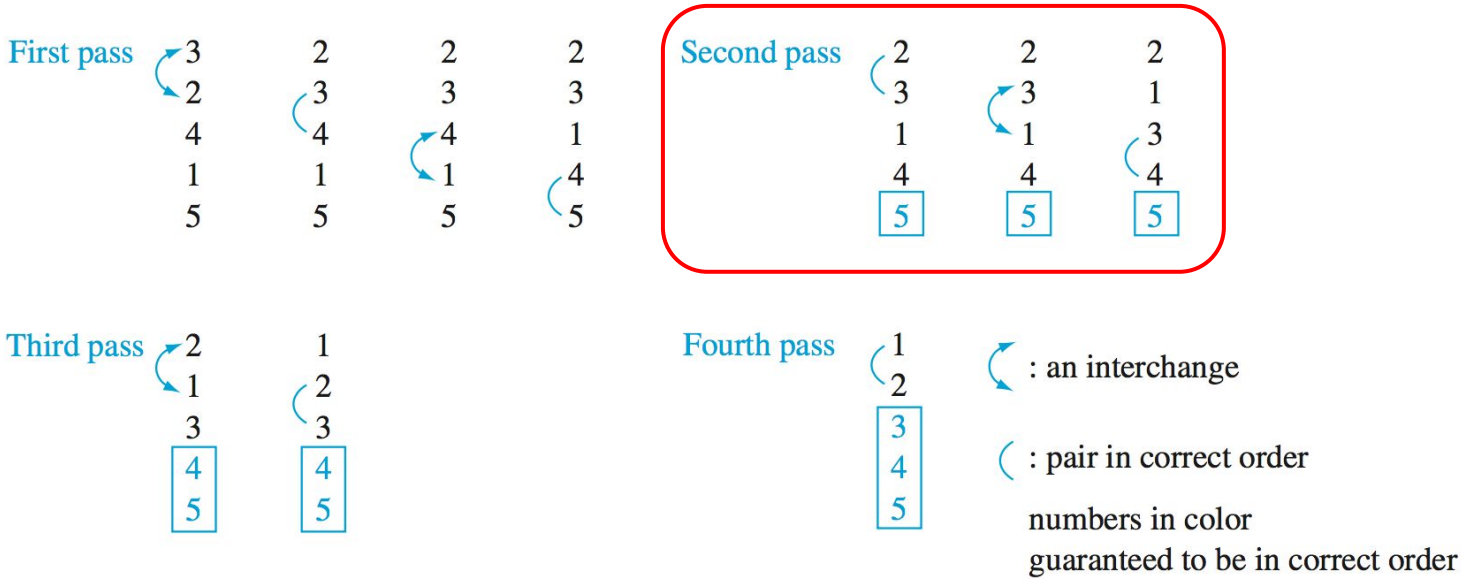
- Again, let's count comparisons.
- **Total:** $(n - 1)$
  First pass: $(n - 1)$ comparisons



Algorithm complexity

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

- Again, let's count comparisons.
- **Total:** $(n - 1) + (n - 2)$
  Second pass: $(n - 2)$ comparisons

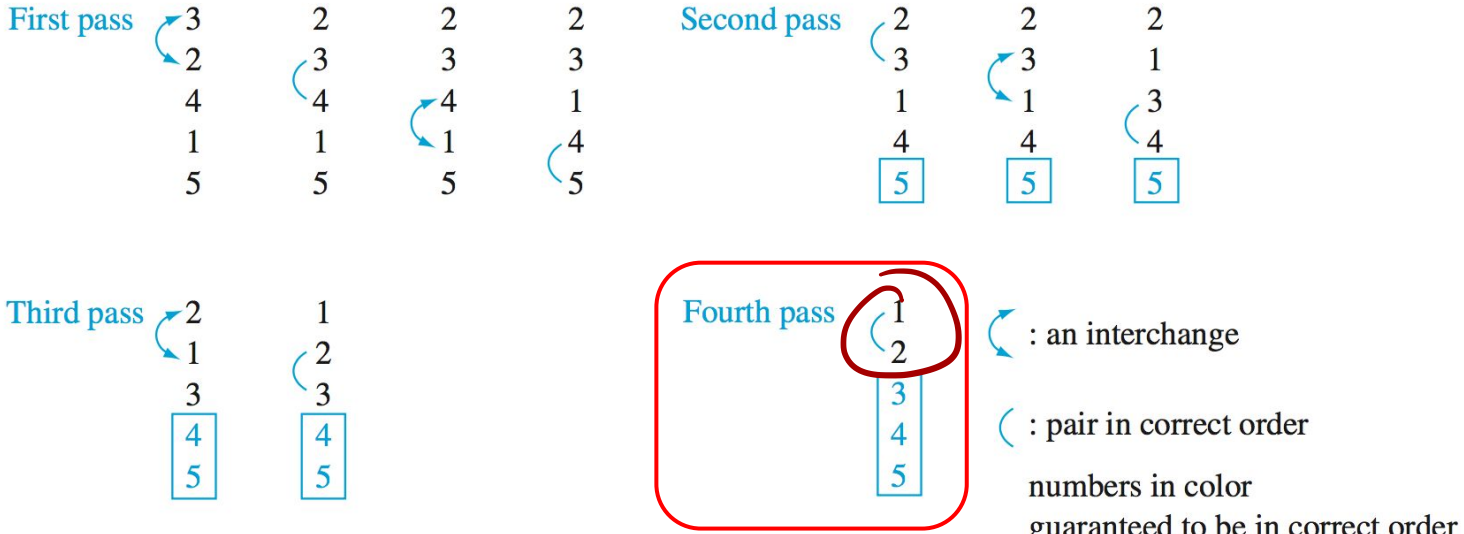


Algorithm complexity

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

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

- Again, let's count comparisons.
- **Total:** $(n - 1) + (n - 2) + \ldots + 2 + 1 = \sum_{k=1}^{n-1} k = \frac{(n-1)(n)}{2} = \frac{n(n-1)}{2}$
  Final pass: 1 comparison



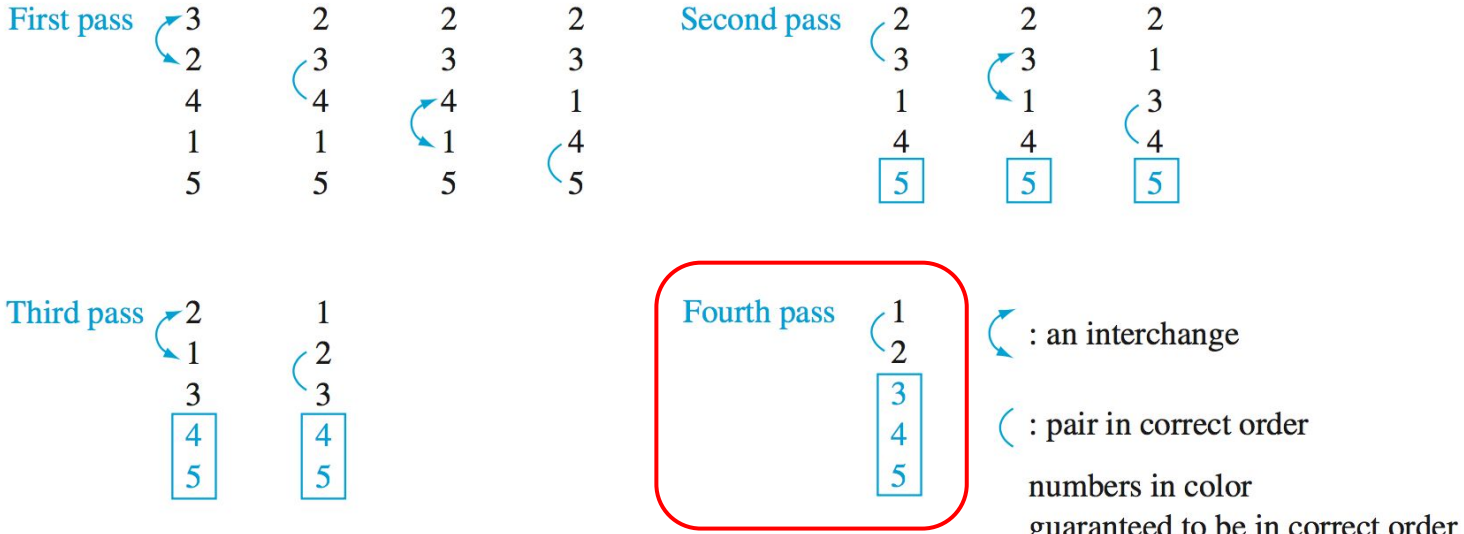
Algorithm complexity

Fond memory:

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

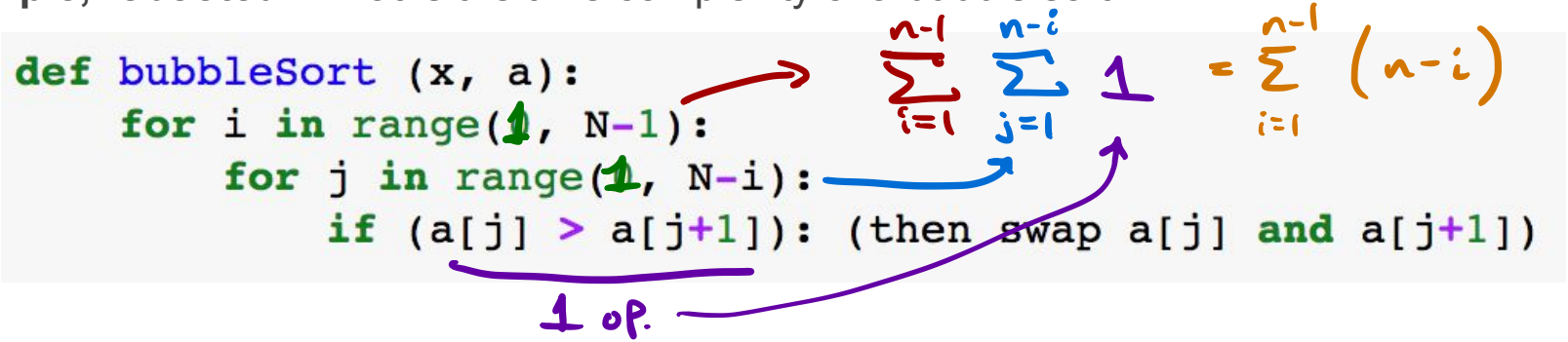
Example: What is the time complexity of a bubble sort?

- Again, let's count comparisons.
- **Total:** $(n - 1) + (n - 2) + \ldots + 2 + 1 = \frac{(n-1)n}{2}$



Algorithm complexity

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



Slick way to count if you have pseudocode:

- Count operations in inner loop(s)
- 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

Example, rebooted: 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])
```

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

$$= n(n-1) - \frac{(n-1)n}{2} = \frac{(n-1)n}{2}$$

Algorithm complexity

FYOG: Show that the worst-case complexity of the insert sort (when counting comparisons) is

$$\frac{n(n+1)}{2} - 1$$

Algorithm complexity

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

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

So **insert sort** requires *n* - 1 more comparisons than **bubble sort**

For large *n* though, the n^2 term dwarfs the *n* terms, and the ½ becomes irrelevant.

So we might say that they both use ***roughly*** n^2 comparisons.

… we'll define precisely what "roughly" means in a bit, but first…

leading order term: highest power of n: n^2

Algorithm complexity

Question: 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?

$T(n)$ = time it takes to sort a list of len = n

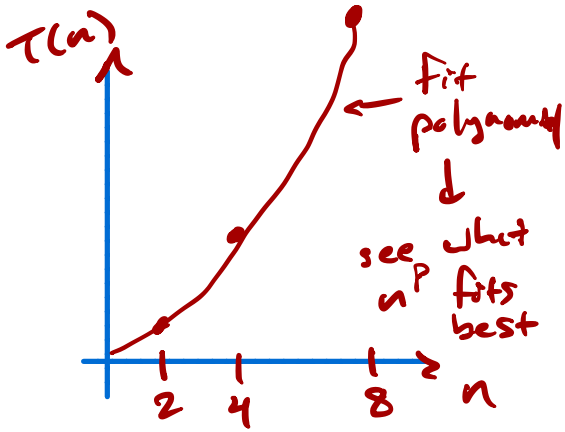
$$\frac{T(2n)}{T(n)} = \frac{(2n)^2}{n^2} = \frac{4n^2}{n^2} = 4$$

Algorithm complexity

Question: 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?

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



Rule: If complexity is n^2, then doubling size quadruples the time.

Question: What if complexity is n^3 ? → 8 times as long

$$\frac{(2n)^3}{n^3} = \frac{8n^3}{n^3} = 8$$

Algorithm complexity

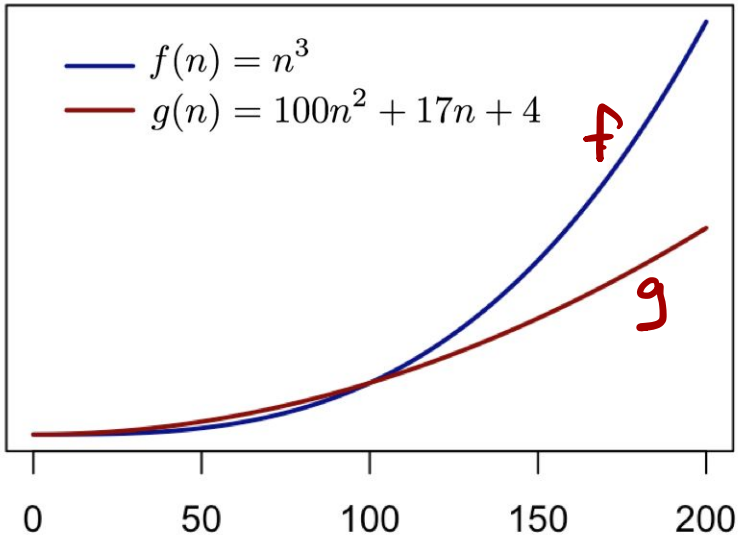
There are several conventions that allow us to talk about the efficiency of algorithms without writing out their precise operation counts.

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

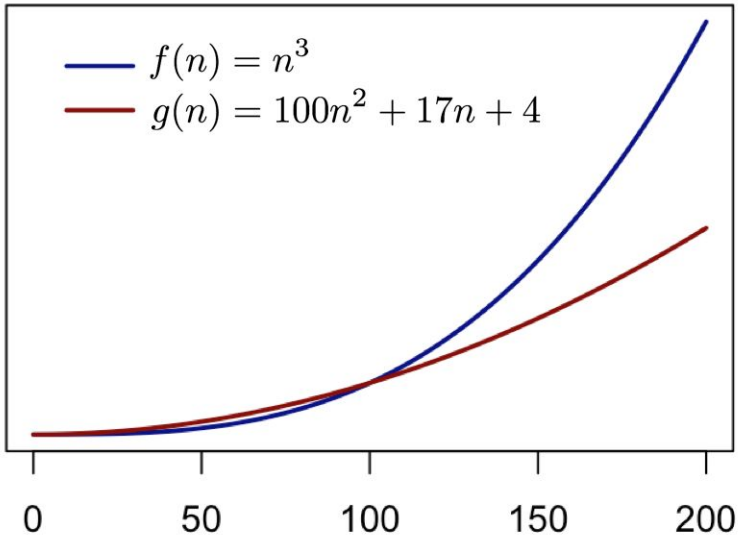
There are several conventions that allow us to talk about the efficiency of algorithms without writing out their precise operation counts.

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?



Answer:

- For small values of n, n^3 might be less than $100n^2 + 17n + 4$ …
- But n^3 becomes **much** larger than $100n^2 + 17n + 4$.

Algorithm complexity

algo.

fcn. you want complexity of

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

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

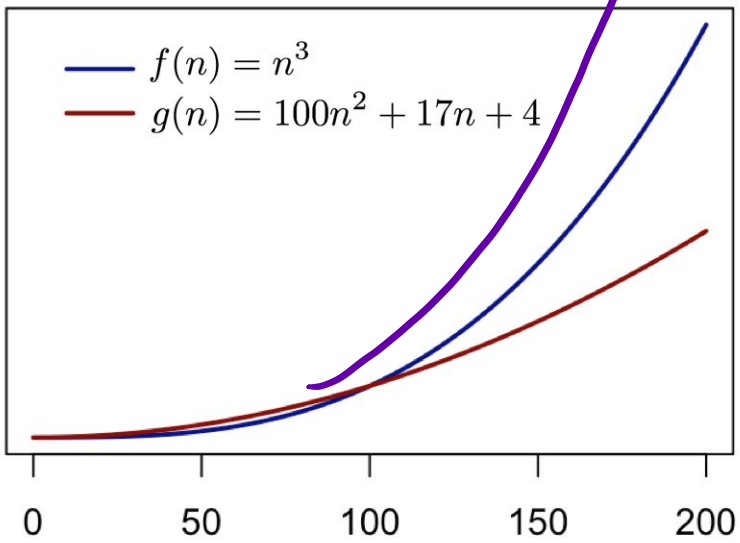
whenever $n > k$. [[*"f(n) is big-oh of g(n)"*]]

We call C & k the "witnesses"

↑ f is dominated by g after $n > k$

The big difference between the two is that one "is like" n^3 and the other "is like" n^2.

- We say that the complexity of Algorithm A, $100n^2 + 17n + 4$, is $\mathcal{O}(n^2)$
- And Algorithm B's complexity, n^3, is $\mathcal{O}(n^3)$



For $n \geq 100$, g is big-O of f

Algorithm complexity

$|f(n)| \leq Cn^2$ — HERE

$= f(n)$

Example: Show that $100n^2 + 17n + 4$ is $\mathcal{O}(n^2)$. → $O(n^2)$ => we want the lowest (tightest) upper bound

① ② ③

Strategy: get upper bound for each piece, then put back together.

in terms of n^2

① $100n^2 \leq 100n^2$ ✓

② $17n \leq 17n^2$ for $n \geq 1$ ✓

③ $4 \leq n^2$ for $n \geq 2$

how we find k: take the max. of the individual restrictions on n

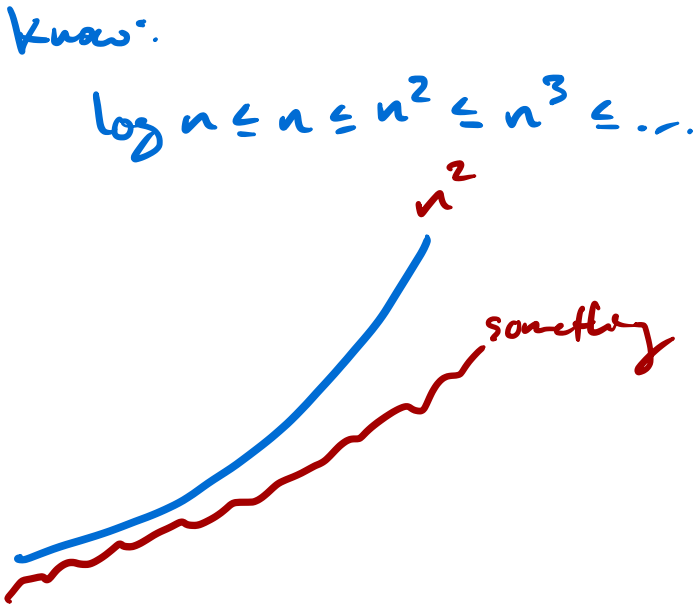
if $n \geq 2$

$\Rightarrow f(n) = 100n^2 + 17n + 4 \leq 100n^2 + 17n^2 + n^2 = 118n^2$

$\Rightarrow$ for $n \geq 2$, $|f(n)| \leq 118n^2$ → w/ $C = 118$ & $k = 2$. f is $O(n^2)$

Algorithm complexity

Example: Show that $100n^2 + 17n + 4$ is $\mathcal{O}(n^2)$.



Solution:
We know that for $n > 1$, it is true that $n \leq n^2$ and $4 \leq n^2$

So for $n > 1$, we have

$$100n^2 + 17n + 4 \leq 100n^2 + 17n^2 + n^2 = 118n^2$$

So $100n^2 + 17n + 4 \leq 118n^2$ when $n > 1$,

which means that $100n^2 + 17n + 4$ is $\mathcal{O}(n^2)$ with $C = 118$ and $k = 1$.

or $n \geq 3$ or 4 or 5....

Algorithm complexity

$n^2 \leq n^3$ for $n \geq 1$

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 $\mathcal{O}(n^{1000})$... ← n^2 is the tightest b.g-O bound
- 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

* $\log(n^5) \to 5(\log n)^5 + n^3$ ← highest power of n is good place to start

big O of n^3

Example: Give a big-O estimate of $h(n) = 5n^2 + n \log(n^3) - n$ = $5n^2 + 3n \log n - n$

① ② ③

① $5n^2 \leq 5n^2$ for any n

② $3n \log n \leq 3n \cdot n \leq 3n^2$

$\log n \leq n$ for $n \geq 1$

③ $-n \leq 0$ for $n \geq 0$

$k = \max(1, 0) = 1$

Put back together:

$h(n) = 5n^2 + 3n \log n - n \leq 5n^2 + 3n^2 + 0 = 8n^2$ for $n \geq 1$

$\therefore$ w/ $C = 8$ & $k = 1$, h is $O(n^2)$

Algorithm complexity

Example: Give a big-O estimate of $h(n) = 5n^2 + n \log(n^3) - n$

Solution: First note that $h(n) = 5n^2 + n \log(n^3) - n = h(n) = 5n^2 + 3n \log(n) - n$

Now note that for $n > 1$, $\log n \leq n$ and $-n < 0$

So, for $n > 1$, $h(n) = 5n^2 + 3n \log n - n \leq 5n^2 + 3n^2 = 8n^2$, is $\mathcal{O}(n^2)$

Theorem: S'pose that $f_1(n)$ is $\mathcal{O}(g_1(n))$ and $f_2(n)$ is $\mathcal{O}(g_2(n))$.
Then $(f_1 + f_2)(n)$ is $\mathcal{O}(\max(|g_1(n)|, |g_2(n)|))$

- **In words:** A sum of functions is big-O of the biggest function.

Theorem: S'pose that $f_1(n)$ is $\mathcal{O}(g_1(n))$ and $f_2(n)$ is $\mathcal{O}(g_2(n))$.
Then $(f_1 f_2)(n)$ is $\mathcal{O}(g_1(n) g_2(n))$

$f_1 = n^2 + 1$
$f_2 = n^3$
$f_1 f_2$ is $O(n^5)$

Algorithm complexity

- So $f(n)$ is $\mathcal{O}(g(n))$ 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*** $h(n)$

Algorithm complexity

- So $f(n)$ is $\mathcal{O}(g(n))$ 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: S'pose 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 $n > k$. [[*"f(n) is big-Omega of h(n)"*]]

Algorithm complexity

highest term is again good place to start

$= f(n)$

Example: Give a big-Ω estimate for $100n^2 + 17n + 4$ ← big Ω: want the tightest lower bound on f

$f(n) = 100n^2 + 17n + 4 \geq 100n^2$ if $n \geq 0$

if $n \geq 0$, $17n + 4 \geq 0$

how we get our k

∴ w/ $C = 100$ & $k = 0$, f is $\Omega(n^2)$

$C \cdot n^2$ in big Ω def.

Also true:

$f(n) \geq 17n$, so f is $\Omega(n)$ but that isn't the tightest Ω bound

Algorithm complexity

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

Solution:
We know that for $n > 0$, it is super true that $100n^2 + 17n + 4 \geq 100n^2$

$100n^2 + 17n + 4$ is $\Omega(n^2)$ with $C = 100$ and $k = 0$

Algorithm complexity

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

Solution:

We know that for $n > 0$, it is super true that $100n^2 + 17n + 4 \geq 100n^2$

$100n^2 + 17n + 4$ is $\Omega(n^2)$ with $C = 100$ and $k = 0$

for $n \geq \max(k_{big\,O}, k_{big\,\Omega})$, f is bounded by both.

Fun fact: We see that $100n^2 + 17n + 4$ is both $\mathcal{O}(n^2)$ and $\Omega(n^2)$

⇒ it grows no faster than a constant times n^2, and
no slower than some other constant times n^2

$100n^2 \leq f \leq 118n^2$

⇒ **Definition:** when this happens, we say that $100n^2 + 17n + 4$ is $\Theta(n^2)$

[[*"big-Theta of n^2"*]]

"order n^2"

Algorithm complexity

Big-Θ estimates of a function are the holy grail!

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

Algorithm complexity… *bonus!*

FYOG: Show that $1^k + 2^k + \ldots + n^k$ is $\mathcal{O}(n^{k+1})$

FYOG: Show that $\log(n^2 + 1)$ and $\log n$ are the same order

FYOG: Show that $\log_{10} n$ and $\log_2 n$ are the same order

FYOG: Give a big-Ω estimate for

$h(n) = 5n^2 + n \log(n^3) - n$

Algorithm complexity

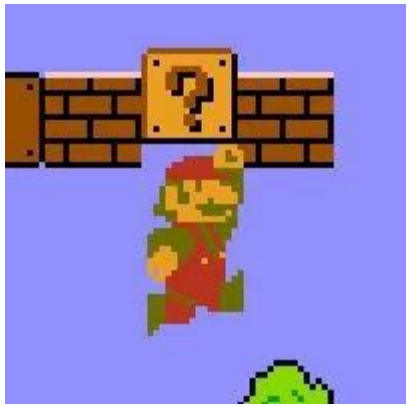
Recap:

- We learned about estimating the **complexity of algorithms**
- We learned about estimating the **growth rates of functions**
  - … because that's useful to see **how (in)efficient an algorithm is**, depending on the size of the input

Next time:

- A bit more about the complexity of algorithms, and
- Matrices!

Bonus material!



Algorithm complexity… *bonus!* -- hints!

- **FYOG:** Show that $1^k + 2^k + \ldots + n^k$ is $\mathcal{O}(n^{k+1})$
  - If $m < n$, how does m^k compare with n^k ?

- **FYOG:** Show that $\log(n^2 + 1)$ and $\log n$ are the same order
  - Work with $\log(n^2 + 1)$. Need to get $\log(n^2 + 1) \le$ [stuff] $\le C \log n$ for $n > k$
    and $\log(n^2 + 1) \ge$ [other stuff] $\ge D \log n$ for $n > m$

- **FYOG:** Show that $\log_{10} n$ and $\log_2 n$ are the same order
  - See if you can get ln n written in terms of each of those two. Then set them equal to one another.