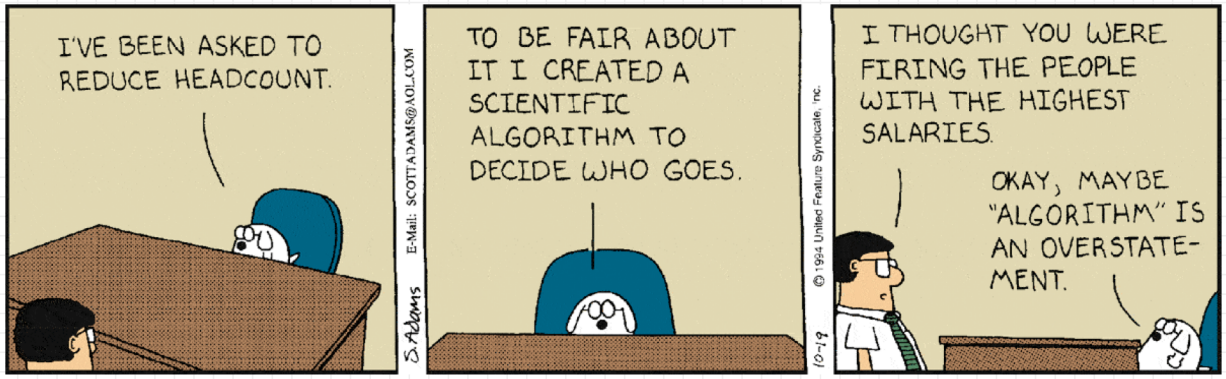


CSCI 2824: Discrete Structures

Lecture 19-20: Algorithms – Complexity.

Reminders

Submissions:

- Homework 7: **Fri 10/18 at noon –** Moodle (1 try)

Readings:

- Ch. 3 – Algorithms
  - 3.1 Greedy Algorithms
  - 3.2 Growth of Functions

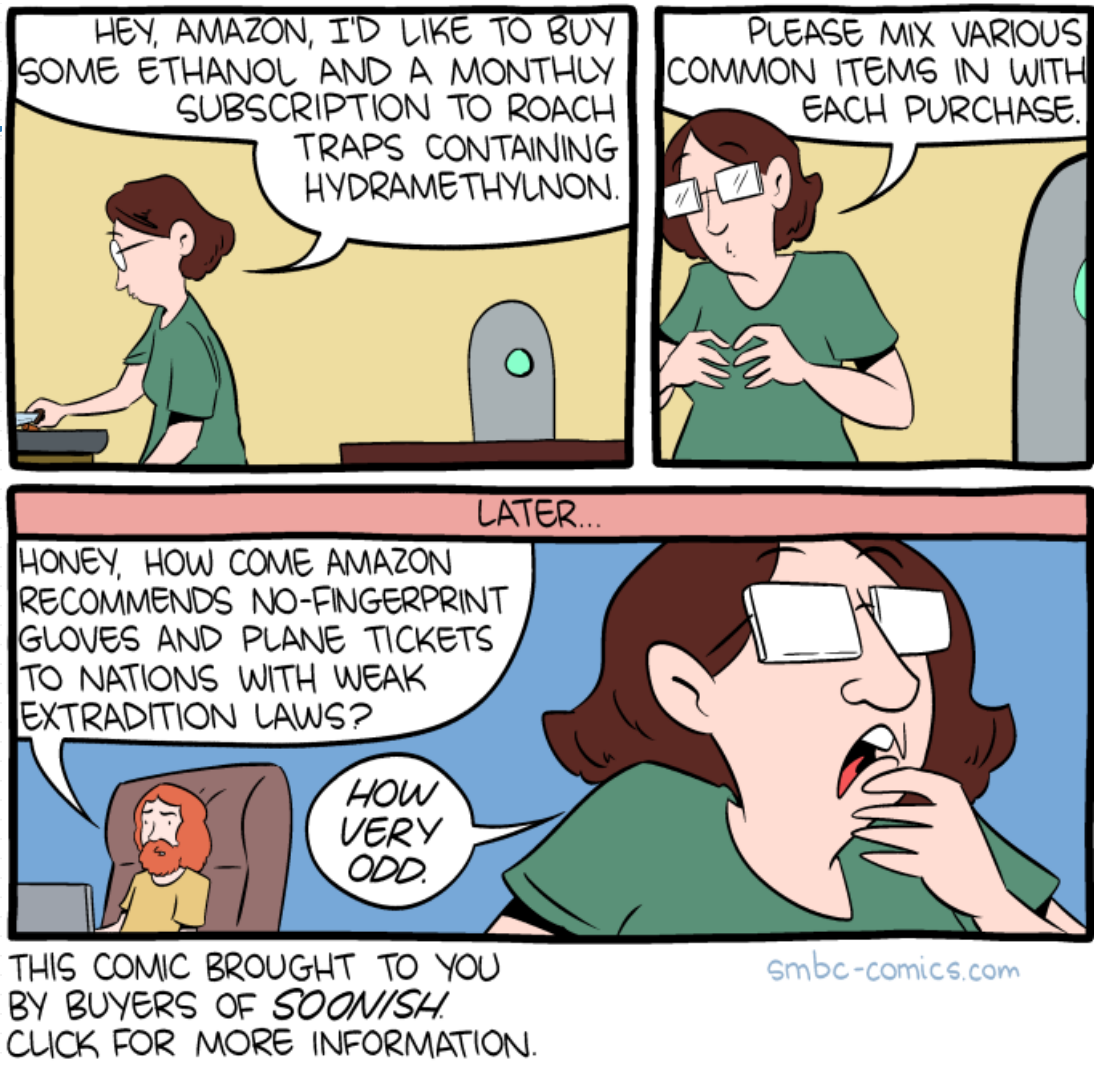
Last time

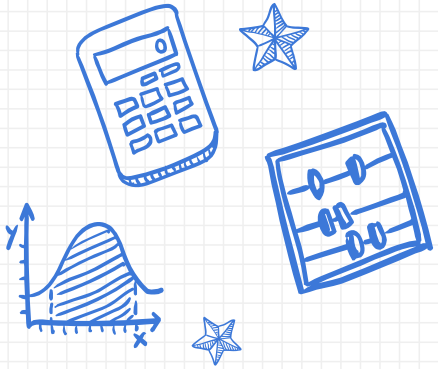
Algorithms…

- What are they?
- Special kinds: searching, sorting

Today:

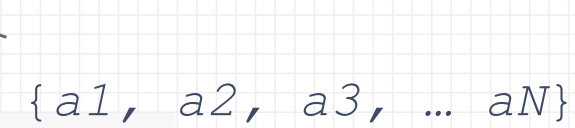
- algorithm *complexity*
- *Greedy* Algorithms
- Growth of Functions





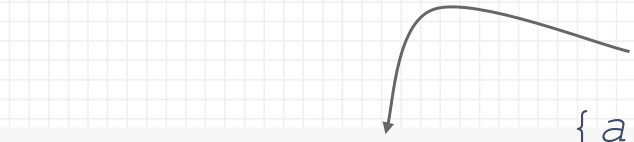
Algorithms

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
```

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
```

Searching algorithms

Super important question: Which of the two searching algorithms seems faster?

Caution! What do we really mean by “faster”?

Searching algorithms

Super important question: Which of the two searching algorithms seems faster?

Caution! What do we really mean by "faster"?

- Which is faster in the *best-case* scenario?
- Which is faster in the *worst-case* scenario?
- Which is faster *on average*?
- What *is* the best-/worst-case scenario?

We will tackle these questions today!

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?

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?

```
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 $x \mathrel{!}= a[i]$

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 $x \mathrel{!}= a[i]$

To exit the loop, we must make another comparison:

- $i > \text{len}(a)$ or $x = a[i]$

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 $x \mathrel{!}= a[i]$

To exit the loop, we must make another comparison:

- $i > \text{len}(a)$ or $x = a[i]$

Then we must check if found

- $i \leq \text{len}(a)$

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
```

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

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

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 found

- $i < \text{len}(a)$

Algorithm complexity

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

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

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
```

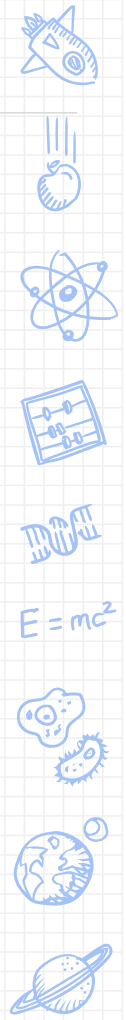
Algorithm complexity

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*[i_*large_left*]



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*[i_*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*]

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*[i_*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 we can instead look at:

LinearSearch @ n or

BinarySearch @ $\log_2 n$?

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

Algorithm complexity

Question: Which is more efficient:

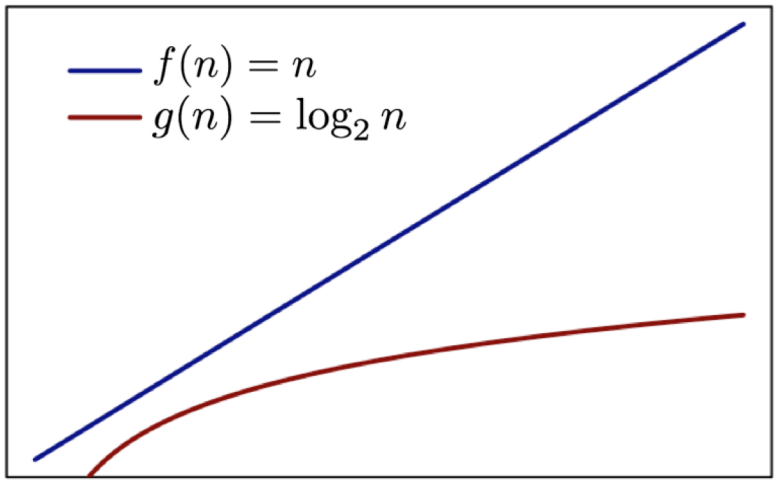
LinearSearch @ $2n + 2$ or BinarySearch @ $2 \log_2 n + 2$?

➢ Well, the +2 and 2x are both on each, so instead look at:

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 @ $2 \log_2 n + 2$?

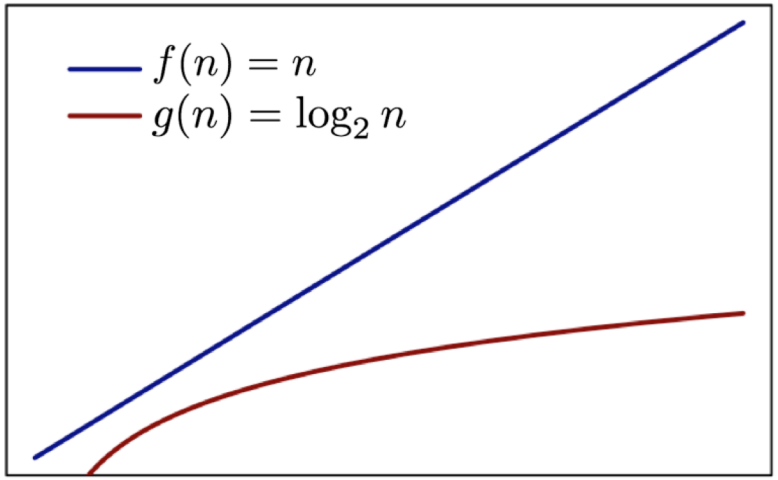
➢ Well, the +2 and 2x are both on each, so they don't really matter:

LinearSearch @ n or

BinarySearch @ $\log_2 n$?

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

➢ **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?

```
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 ……

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
```

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 $2i + 1$ comparisons…

Algorithm complexity

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

So the average of these *n* possibilities is then: $\frac{3+5+7+\cdots+(2n+1)}{n}$

Algorithm complexity

Example: What is the average time complexity of a linear search?
So the average of these *n* possibilities is then: $\frac{3+5+7+\cdots+(2n+1)}{n}$

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

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**

Sorting Algorithms

Task: Given some unordered list of elements, organize them according to some notion of "order" (e.g., increasing numbers, alphabetizing, etc…)

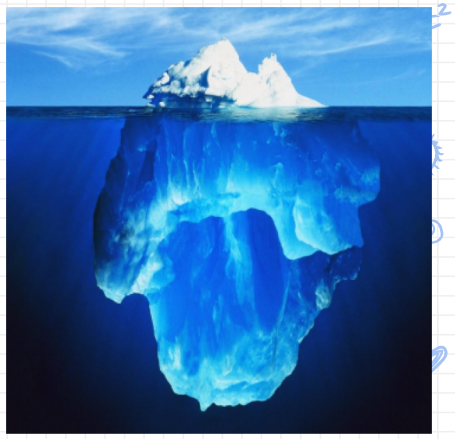
Applications: Sort mail by location along route.
Alphabetizing CSCI 2824 exams by student name.

Goals: Sometimes the whole point is just to sort the lists.

- Examples: to do a binary search, or find duplicates in a list

We will look at two algorithms: (1) ***bubble sort***
(2) ***insert sort***

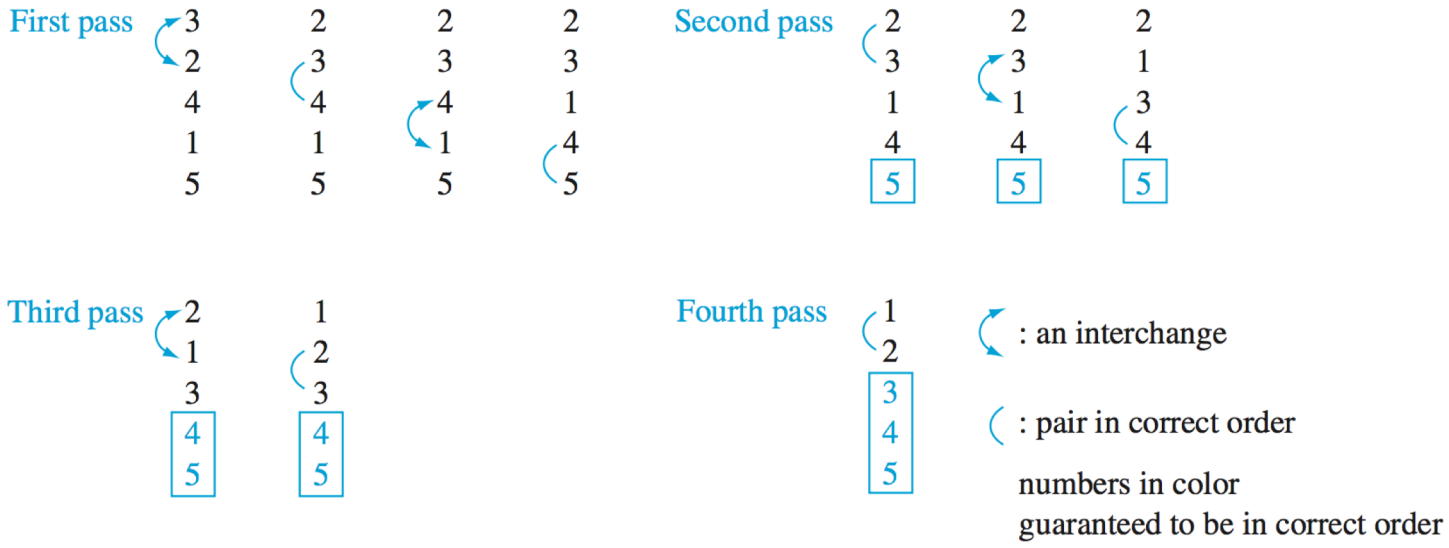
- But note that there is a LOT of effort/computational power spent on sorting, and this is just the tip of the iceberg.



Bubble Sort

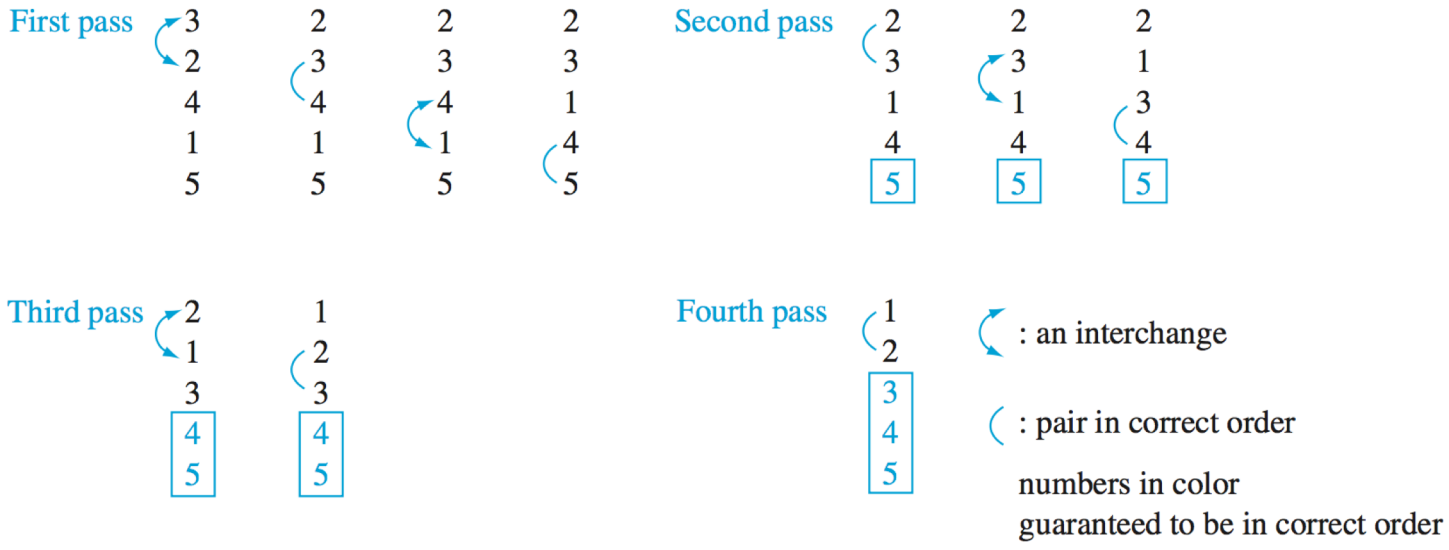
Make passes through the list; whenever you encounter two elements that are out of order, swap them. Repeat until the list is sorted.

Example: S'pose we want to sort the list {3, 2, 4, 1, 5}.



Bubble Sort

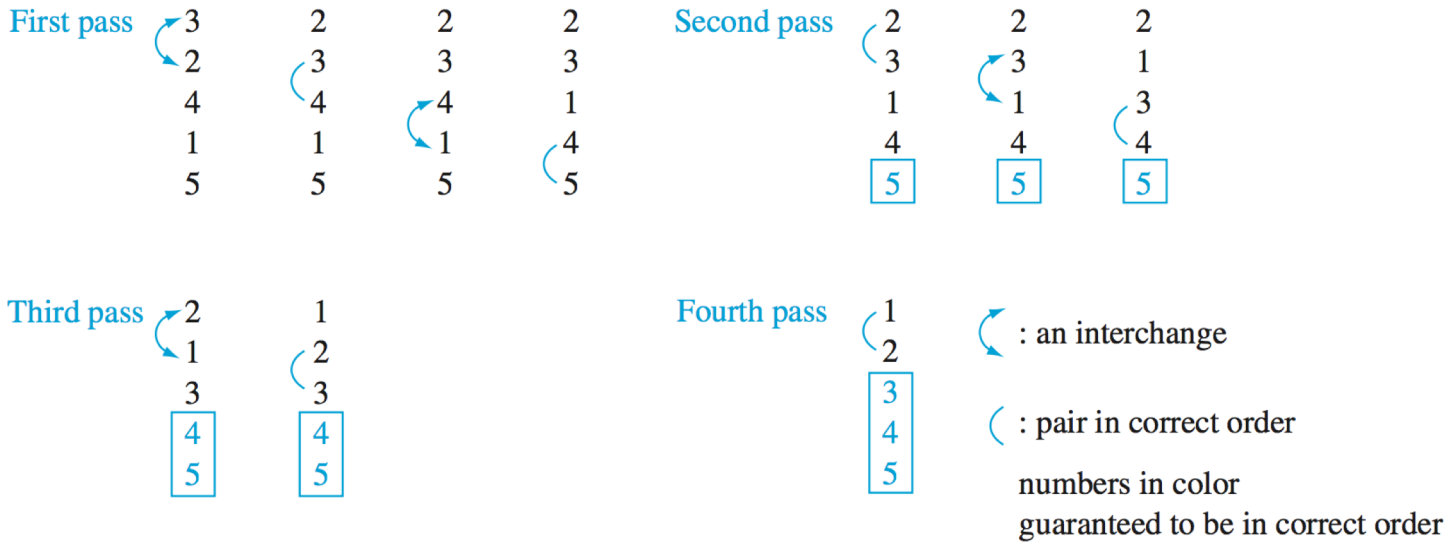
```
def bubbleSort (x, a):
    for i in range(0, N-1):
        for j in range(0, N-i):
            if (a[j] > a[j+1]): (then swap a[j] and a[j+1])
```



Algorithm complexity

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

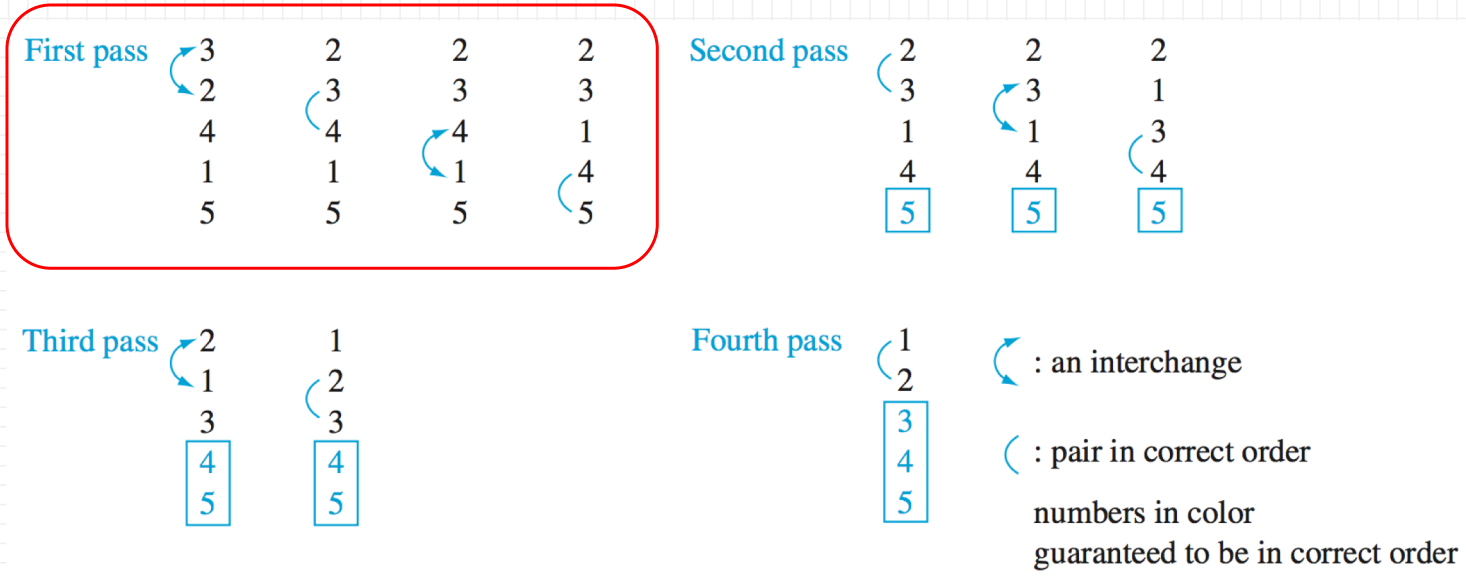
```
def bubbleSort (x, a):
    for i in range(0, N-1):
        for j in range(0, N-i):
            if (a[j] > a[j+1]): (then swap a[j] and a[j+1])
```



Algorithm complexity

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

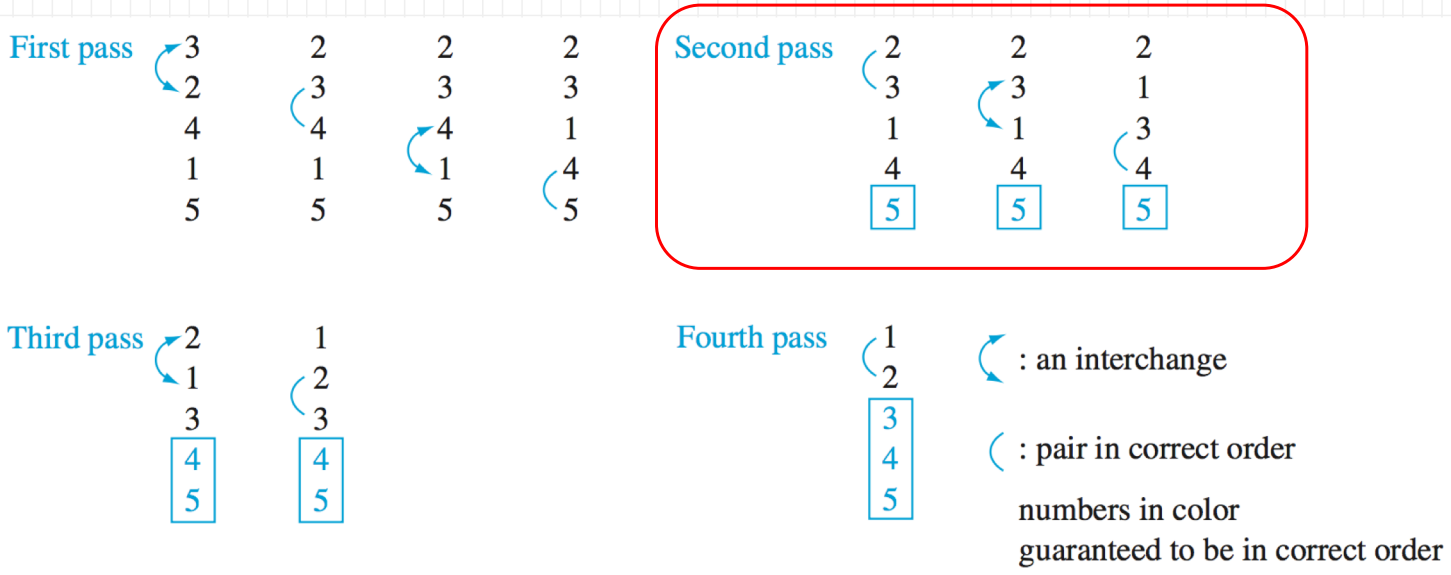
- Again, let's count comparisons.
- **Total:**
  First pass: comparisons



Algorithm complexity

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

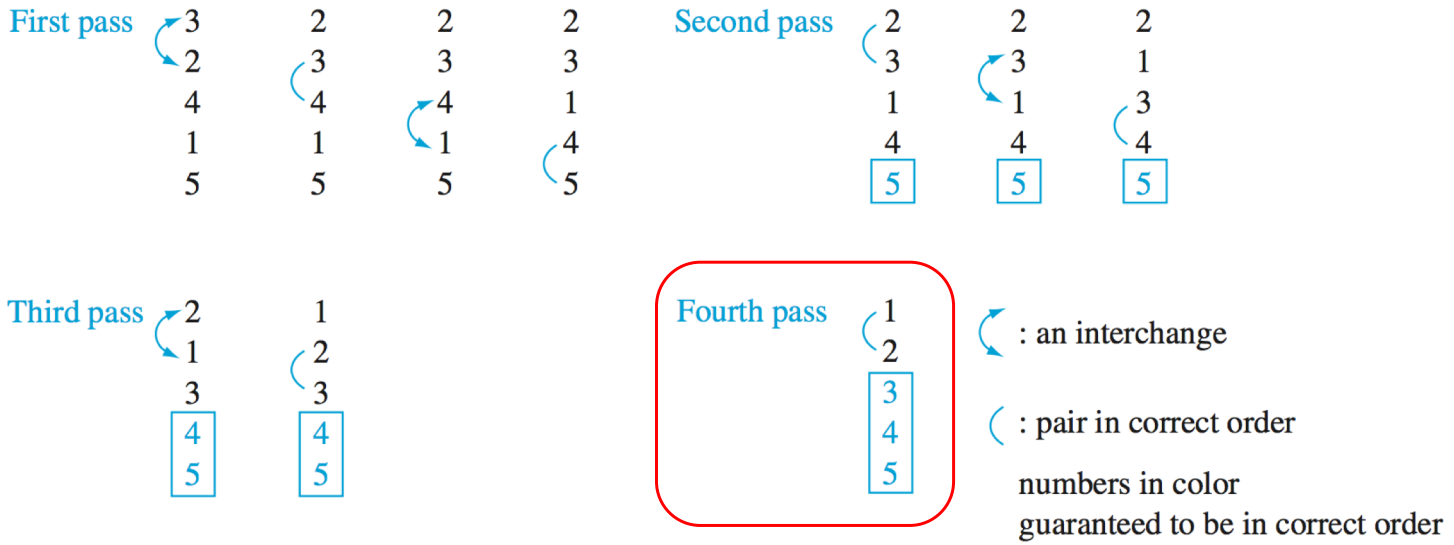
- Again, let's count comparisons.
- **Total:** $(n - 1) +$
  Second pass: comparisons



Algorithm complexity

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

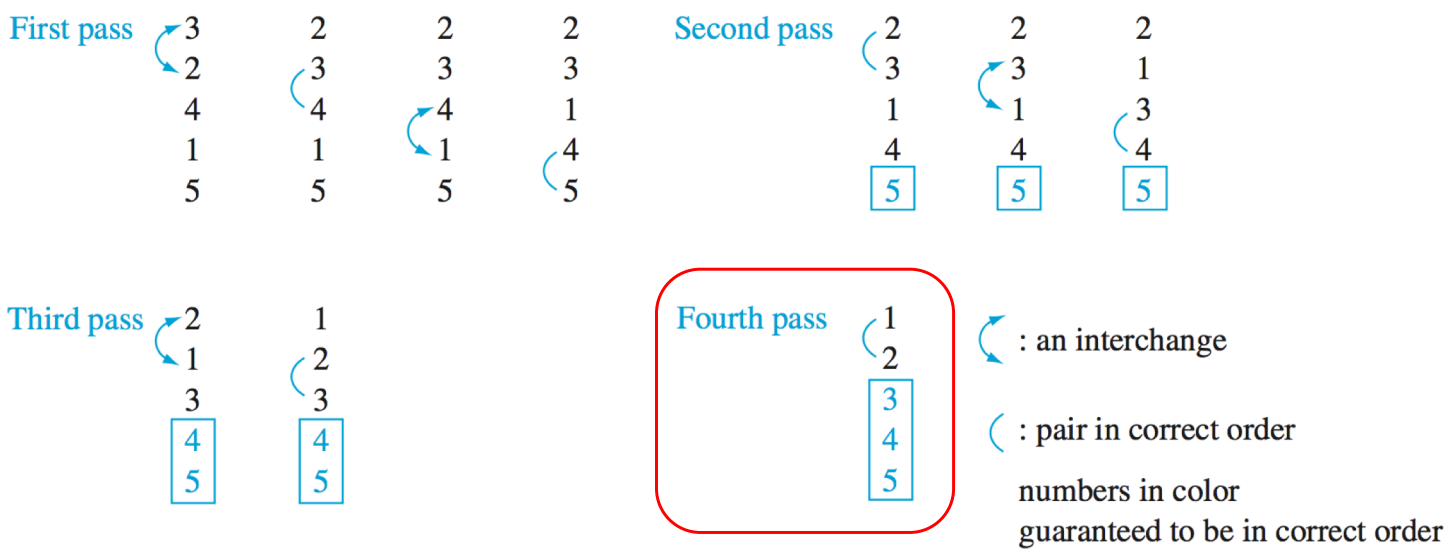
- Again, let's count comparisons.
- **Total:** $(n - 1) + (n-2) + \ldots +$
  Final pass: comparison



Algorithm complexity

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

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



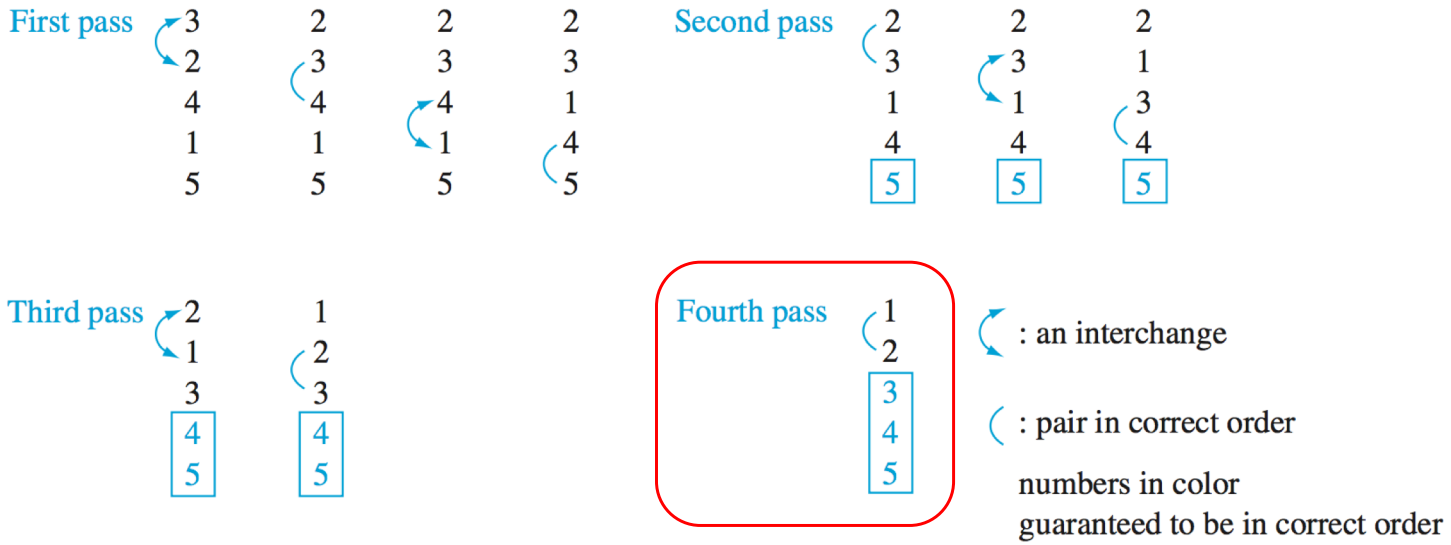
Algorithm complexity

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) + … + 1 = [(n-1) + 1] * (n-1) / 2 = $\frac{(n-1)n}{2}$



Algorithm complexity

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

```
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])
```

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

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])
```

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

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

Algorithm complexity

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])
```

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

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

Caveat: here (and elsewhere)

- we are **neglecting** the comparison needed to make sure we are still within the **for** loops. (Rosen, p. 221)
- we are not counting operations for swapping elements

Insertion Sort

Example: S'pose we want to sort the list {3, 2, 4, 1, 5}.

Insertion Sort

Example: S'pose we want to sort the list {3, 2, 4, 1, 5}.

- 1st element: {3} -- definitely in order -- proceed with {3}
- 2nd element: {3, 2} -- out of order -- put where it belongs -- proceed with {2, 3}
- 3rd element: {2, 3, 4} -- in order! -- proceed with {2, 3, 4}
- 4th element: {2, 3, 4, 1} -- out of order -- put where it belongs -- proceed with {1, 2, 3, 4}
- 5th element: {1, 2, 3, 4, 5} -- in order! -- proceed with {1, 2, 3, 4, 5}
- No elements left -- stop.

Algorithm complexity

FYOG: Show that the worst-case complexity of the insert sort (when counting comparisons) is $\frac{n(n+1)}{2} - 1$

Note: insertion sort algorithms can differ in implementations. This example uses the Rosen implementation page 222, where "in the worst case, j comparisons are required to insert the jth element into the correct position".

Homework 7 last problem: the algorithm is slightly different

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.

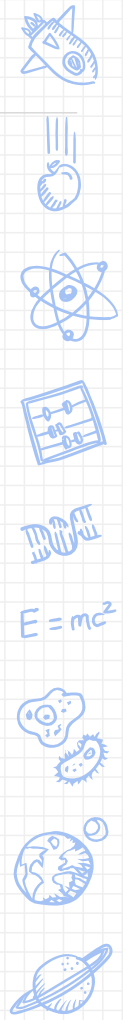
So we might say that they both use ***roughly*** n^2 comparisons.

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

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?



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?

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

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 ?

Algorithm complexity

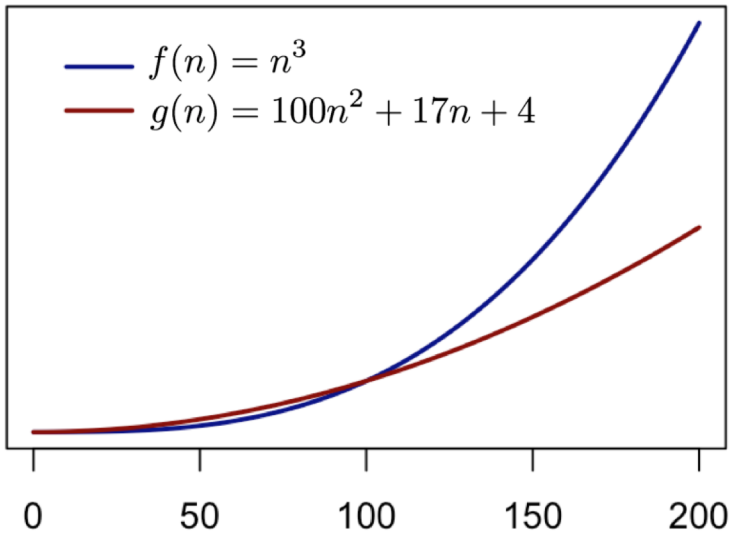
There are several conventions that allow us to talk about the efficiency of algorithms without writing out their precise operation counts.

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.

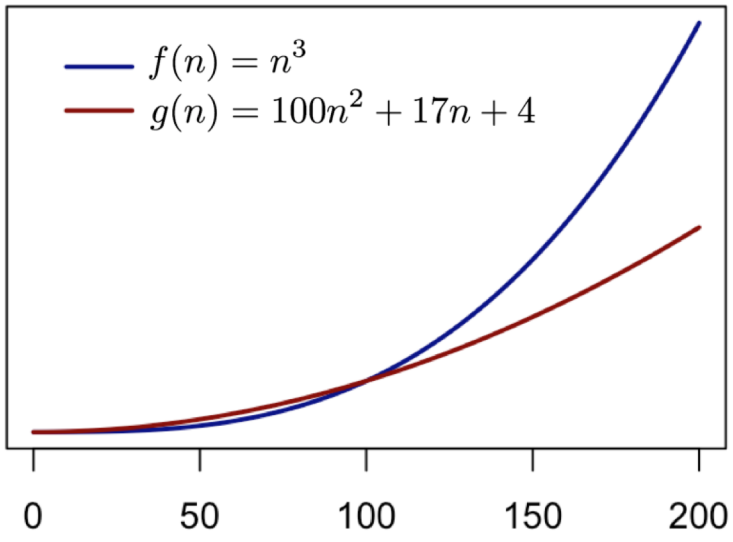
Algorithm B uses n^3 operations.

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 $100n^2 + 17n + 4$.



Greedy algorithms

Many algorithms are designed to solve ***optimization problems***.

Goal: To find a solution that maximizes or minimizes some ***objective function***.

Applications: Find a route between two cities that minimizes distance (or time travelled, or elevation gain, …).
Encode a message using the fewest bits possible.

Greedy algorithms select the best choice at each step.

Note: they are *not* guaranteed to find an optimal solution. You must check once a solution is found.

Greedy algorithms

Task: Consider making *n* cents change, using quarters, dimes, nickels and pennies, using the fewest total number of coins.
Greedy strategy: At each step, choose the largest denomination coin possible to add to the pile, so as not to exceed *n* cents.

Example: S'pose we want to make change for 67 cents.

1. Select a quarter (25 cents)
2. Select another quarter (50 cents)
3. Select a dime (60 cents)
4. Select a nickel (65 cents)
5. Select a penny (66 cents)
6. Select another penny (67 cents)

{c[0], c[1], c[2], …, c[N]} = coin denominations, eg...

```
def greedy (amt, c):
    total = amt
    d = []                  # initialize, to count coins of each denomination
    n_thiscoin = 0          # initialize, to count the current coin
    for thiscoin in range(0, len(c)):
        while (total ≥ c[thiscoin]):
            n_thiscoin = n_thiscoin + 1
            total = total - c[thiscoin]    # add another one of this coin
        d.append(n_thiscoin)
        n_thiscoin = 0                     # reset counter to 0
    return (d)
```

Greedy Algorithms

Fun Fact: If we have **quarters**, **dimes**, **nickels** and **pennies** available, then this algorithm is optimal.

Fun? Fact: If we only have **quarters**, **dimes** and **pennies** available (no nickels), then this algorithm is ***not*** optimal.

Example: If we wanted to make change for 30 cents, using no nickels, then we would:

1. take 1 quarter
2. take 5 pennies

… But it would be ***optimal*** to take 3 dimes instead.

Algorithms

Recap:

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

Next time:

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

(note though that you need to ***sort*** first!)