CSCI 2824: Discrete Structures

Lecture 17 & 18: Algorithms & Complexity

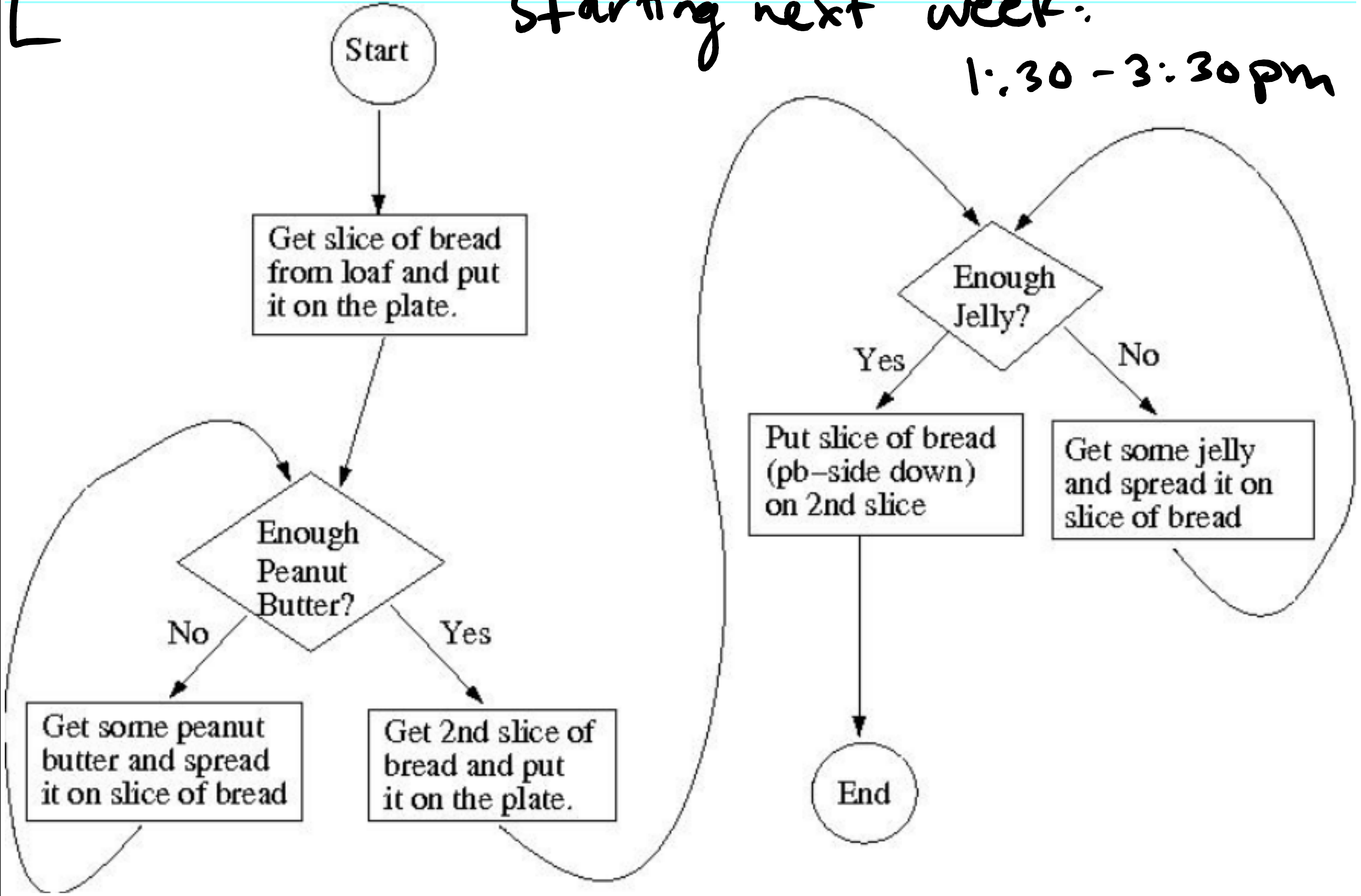
Rachel Cox

Department of Computer Science

HW6: Due Saturday at 6pm

Thurs Office Hours: cancelled tomorrow

Starting next week: 1:30 - 3:30pm



Algorithms

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

Algorithms will be described in "*pseudocode*"

- provides an intermediate step between an English language description of an algorithm and an implementation of this algorithm in a programming language

Algorithms – Maximum Element

Task: Find the largest element in a finite sequence

Example: Given the sequence $\{1, 3, 12, 8, 2, 47, 58, 54, 7\}$ return 58

procedure $\mathtt{max}(a_1, a_2, \ldots, a_n)$

$\mathtt{max} := a_1$

for $i := 2$ to n

  if $\mathtt{max} < a_i$ **then** $\mathtt{max} := a_i$

return max

1. Initialize temporary max to the 1st term in the sequence
2. Compare the 2nd term in sequence to max. If it's larger, set max to this integer
3. Repeat previous step if there are more terms in sequence
4. Stop when there are no more terms.

Algorithms – Properties

Input: An algorithm has inputs from a particular set.

Output: From each set of inputs, the algorithm produces outputs. The outputs are the solution to the problem.

Definiteness: The steps in the algorithm are defined precisely.

Correctness: The algorithm should produce the correct output for each set of inputs.

Finiteness: An algorithm should terminate in finite time.

Generality: An algorithm should be applicable for all problems of the desired form.

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…

Algorithms – Greedy Algorithms

Task: Find a solution that minimizes or maximizes some parameter.

Applications:

- Finding a route between cities that minimizes distance
- Encode a message using the fewest bits possible

Greedy Algorithms select the locally optimal choice at each step, the goal is global optimization.

Not guaranteed to find the overall optimal solution…. must check after a solution has been found.

Algorithms – Greedy Algorithms

Task: Consider making n cents change with quarters, dimes, nickels, and pennies, using the least total amount of coins.

Example: Suppose we want to make change for 67 cents.

67 cents: give 1 quarter

42 cents: give 1 quarter

17 cents: give 1 dime

7 cents: give 1 nickel

2 cents: 1 penny

1 penny

2 quarters,
1 dime,
1 nickel,
2 pennies

Algorithms – Greedy Algorithms

Task: Consider making n cents change with quarters, dimes, nickels, and pennies, using the least total amount of coins.

Let $c_1 > c_2 > \cdots > c_r$ denote coin denominations

procedure Change($c_1, c_2, \ldots, c_r$)

for $i := 1$ **to** r

$d_i := 0$ # d_i counts coins of denom. c_i

while $n \geq c_i$

$d_i := d_i + 1$ # add one coin of denom. c_i

$n := n - c_i$

Algorithms – Greedy Algorithms

Fact: The number of coins used to make n cents using quarters, dimes, nickels, and pennies in the previous algorithm is optimal.

Fact: If we only used quarters, dimes, and pennies (and no nickels) then the algorithm would not produce an optimal solution

Example: Suppose we wanted to make change for 30 cents using only quarters, dimes, and pennies

Our algorithm would use 1 quarter (25 cents) and 5 pennies (5 cents) for a total of 6 coins used

But a more optimal solution would be to use 3 dimes

Algorithms – Greedy Algorithms

Task: Given a number N, find the largest Decent Number with N digits. If no such number exists, return -1.

A **Decent Number** is a number that includes only 3's and 5's:

1. The number of 3's is divisible by 5
2. The number of 5's is divisible by 3

Examples:

- The largest 3-Digit Decent Number is 555
- The largest 5-Digit Decent Number is 33333
- The largest 8-Digit Decent Number is 55533333

Algorithms – Linear Search

Task: Find the location of an element in a list or determine that the desired element is not in the list.

Example: Find 34 in the sequence $\{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89\}$

procedure LinearSearch$(x, a_1, a_2, \ldots, a_n)$

$i := 1$

while $(i \leq n \text{ and } x \neq a_i)$

$\quad i := i + 1$

if $i \leq n$ **then** $location := i$

else $location := -1$

return $location$

1. Compare x with a_1. If $x = a_1$, the solution is the location of a_1 (namely 1)
2. When $x \neq a_1$, compare x with a_2.
3. Continue comparing x successively with each term on the list until a match is found.
4. If entire list is searched without locating x, return 0.

Algorithm Complexity – Linear Search

<u>Example</u>: 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)
```

Worst Case Scenario:

In the while loop

- i < len(a) +n
- x != a[i] +n

One final check to leave the while loop

- i < len(a) +1

If statement

- i < len(a) +1

Total Comparisons: n+n+1+1 = 2n+2

Algorithms – Binary Search

Task: Find the location of an element in a list or determine that the desired element is not in the list.

Example: Find 34 in the sequence $\{1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89\}$

procedure BinarySearch($x, a_1, a_2, \ldots, a_n$)

$i := 1$ # i is left endpoint of search interval

$j := n$ # j is right endpoint of search interval

while $i < j$

  $m := \lfloor (i+j)/2 \rfloor$ # m is index of largest in left list

  if $x > a_m$ **then** $i := m + 1$, **else** $j := m$

if $x = a_i$ **then** *location* $:= i$, **else** *location* $:= -1$

return *location*

1. Cut list in half.
2. Compare x with last term in first half of list.
3. If x is larger, x must be in second half of list (assuming it's in list).
4. Repeat
5. If entire list is searched without locating x, return 0.

Algorithms – Binary Search

Task: Find the location of an element in a list or determine that the desired element is not in the list.

Example: Find 34 in the sequence {1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89}

Index:	0	1	2	3	4	5	6	7	8	9	10
	1	1	2	3	5	8	13	21	34	55	89

i = 0

j = 10

m = (i+j)//2 = 5

Is 34 = 8? -no, 34>8

- reset i=m+1

$m = \frac{0+10}{2} = 5$

Index:	0	1	2	3	4	5	6	7	8	9	10
	1	1	2	3	5	8	13	21	34	55	89

i = 6

j = 10

m = (i+j)//2 = 8

Is 34 = 34? -yes

Algorithm Complexity – Binary Search

$\frac{2^k}{2^1} = 2^{k-1}$

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

Worst case scenario.

```
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 # elements $\boxed{n = 2^k}$ where k is some integer

Make the first cut
- ➢ two lists that are 2^{k-1} in length

$\frac{2^{k-1}}{2^1} = 2^{k-1-1}$

Make the second cut
- ➢ two lists that are 2^{k-2} in length

.
.
.

Make the k-th cut
- ➢ remaining "list" is $2^{k-k} = 2^0 = 1$ in length

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

After maximum of k steps, check that we should really leave the while loop +1

Check x == a[left] +1

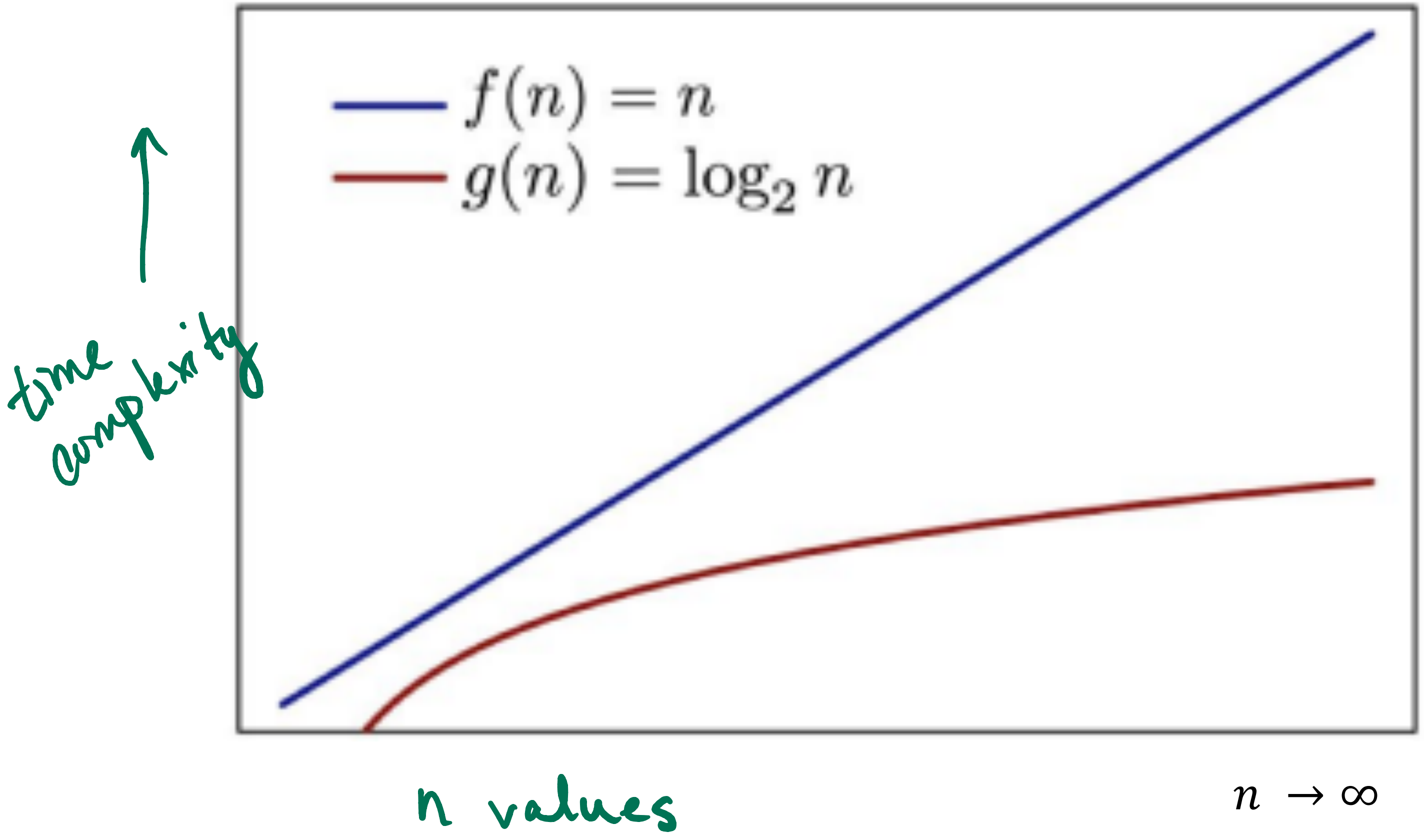
Total Comparisons: $\underbrace{k + 2} = \log_2 n + 2$

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 in the list

- 0 < len(a) +1
- x != a[0] +1
- i<len(a) +1

3 comparisons

If x is the second element in the list

- 0 < len(a) +1
- x != a[0] +1
- 1 < len(a) +1
- x != a[1] +1
- i<len(a) +1

5 comparisons

If x is the third element in the list

…

7 comparisons

.

.

.

If x is the n-th element in the list.

2n+1 comparisons

Algorithm Complexity

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

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

$$n$$

$$(1+1+1+\cdots+1)+(2+4+6+\cdots+2n)$$

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

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

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

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

$$= 1+n+1$$

1	100	101
2	99	101
3	98	101
4	97	101
5	96	101
6	95	101
⋮	⋮	
49	52	101
50	51	101

$50 \cdot 101$

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

$$= n+2$$

Algorithms – Sorting

Task: Given an unordered list of elements, organize them according to some notion of order.

e.g. Alphabetizing

- ➢ Ordering a list can be the goal in–and–of itself.
- ➢ Ordering a list can make other tasks easier.

Algorithms – Bubble Sort

Task: Given an unordered list of elements, organize them according to some notion of order.

Example: Sort the list $\{3, 2, 4, 1, 5\}$

procedure BubbleSort($a_1, a_2, \ldots, a_n$)

for $i := 1$ **to** $n - 1$

  for $j := 1$ **to** $n - i$

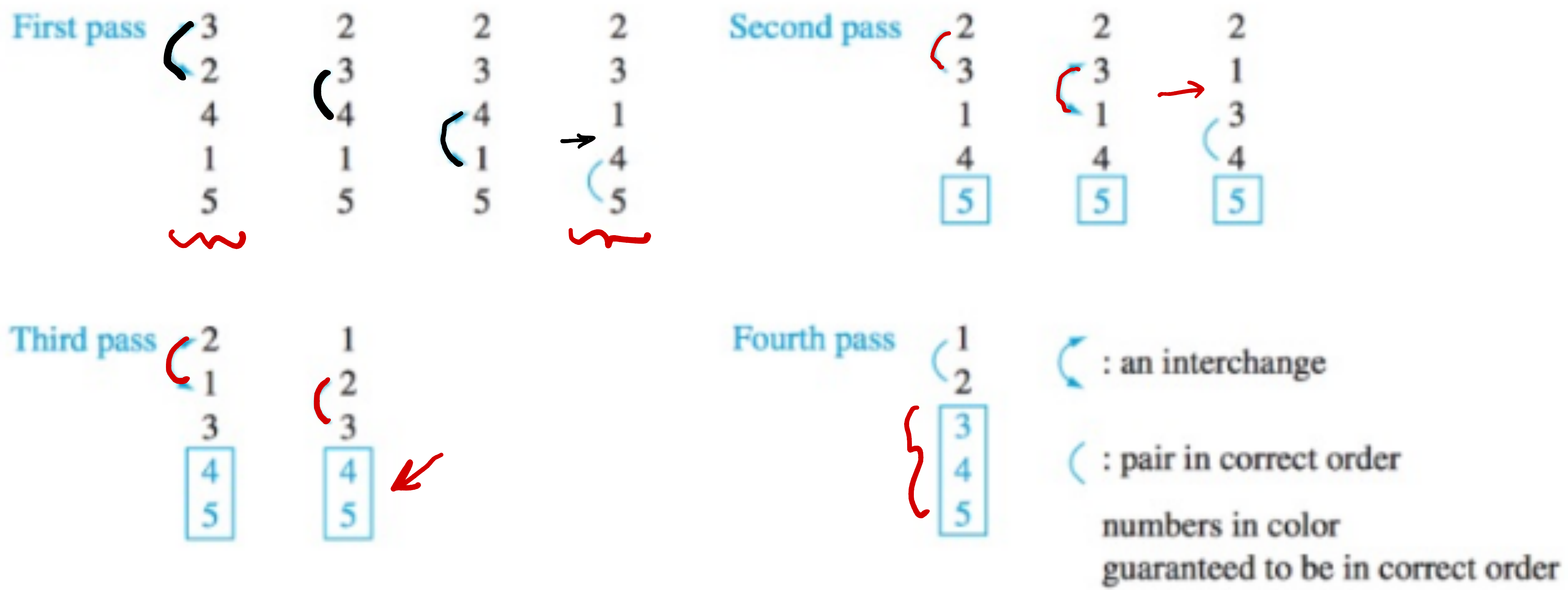
    if $a_j > a_{j+1}$ **then** interchange a_j and a_{j+1}

1. Make passes through the list, interchanging adjacent pairs that are in the wrong order.
2. Repeat until the list is sorted.
3. Large elements sink to bottom, small elements bubble to top.

Algorithms – Bubble Sort

Task: Given an unordered list of elements, organize them according to some notion of order.

Example: Sort the list {3, 2, 4, 1, 5}



Algorithms – Insertion Sort

Task: Given an unordered list of elements, organize them according to some notion of order.

Example: Sort the list $\{3, 2, 4, 1, 5\}$

procedure *insertion sort*($a_1, a_2, \ldots, a_n$: real numbers with $n \geq 2$)
for $j := 2$ **to** n
    $i := 1$
    while $a_j > a_i$
        $i := i + 1$
    $m := a_j$
    for $k := 0$ **to** $j - i - 1$
        $a_{j-k} := a_{j-k-1}$
    $a_i := m$
$\{a_1, \ldots, a_n$ is in increasing order$\}$

1. Make passes through the list, successively insert next unsorted element into the already sorted front end of the list.
2. Repeat until the list is sorted.

Algorithms – Insertion Sort

Task: Given an unordered list of elements, organize them according to some notion of order.

Example: Sort the list {3, 1, 5, 4, 2}

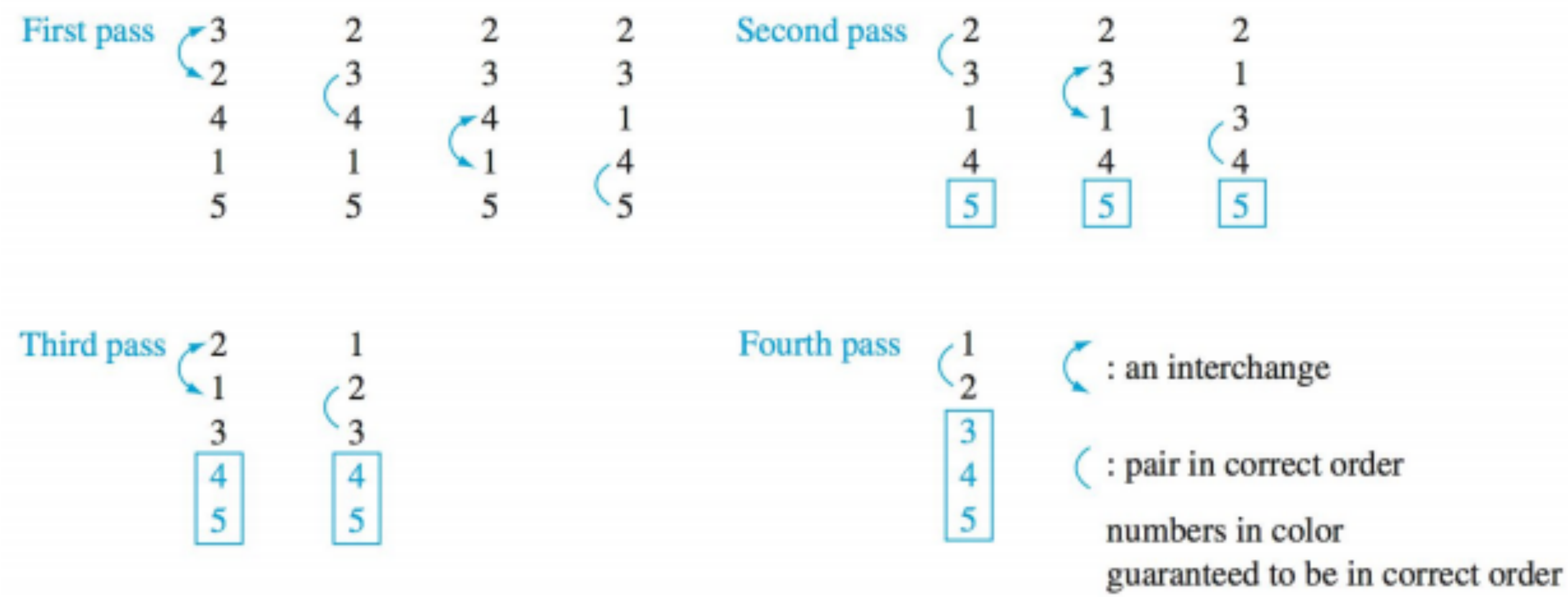
- $\{\mathbf{3}, \mathit{1}, 5, 4, 2\} \Rightarrow \{\mathbf{1}, \mathbf{3}, 5, 4, 2\}$
- $\{\mathbf{1}, \mathbf{3}, \mathit{5}, 4, 2\} \Rightarrow \{\mathbf{1}, \mathbf{3}, \mathbf{5}, 4, 2\}$
- $\{\mathbf{1}, \mathbf{3}, \mathbf{5}, \boxed{\mathit{4}}, 2\} \Rightarrow \{\mathbf{1}, \mathbf{3}, \mathbf{4}, \mathbf{5}, 2\}$
- $\{\mathbf{1}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathit{2}\} \Rightarrow \{\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}\}$

Algorithm Complexity

last time: $1+2+3+4+\dots+n = \frac{n(n+1)}{2}$

Example: 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])
```



n elements to be sorted:

first pass: n-1 comparisons

second pass: n-2 comparisons

third pass: n-3 comparisons

.

.

.

i-th pass: n-i comparisons

.

.

.

last pass: 1 comparison

Total Comparisons: $\underbrace{n-1+n-2+\ \dots+3+2+1}= \frac{(n-1)n}{2}$

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])
```

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)

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

Algorithm Complexity

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

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?

n is list length

(list length)^2

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

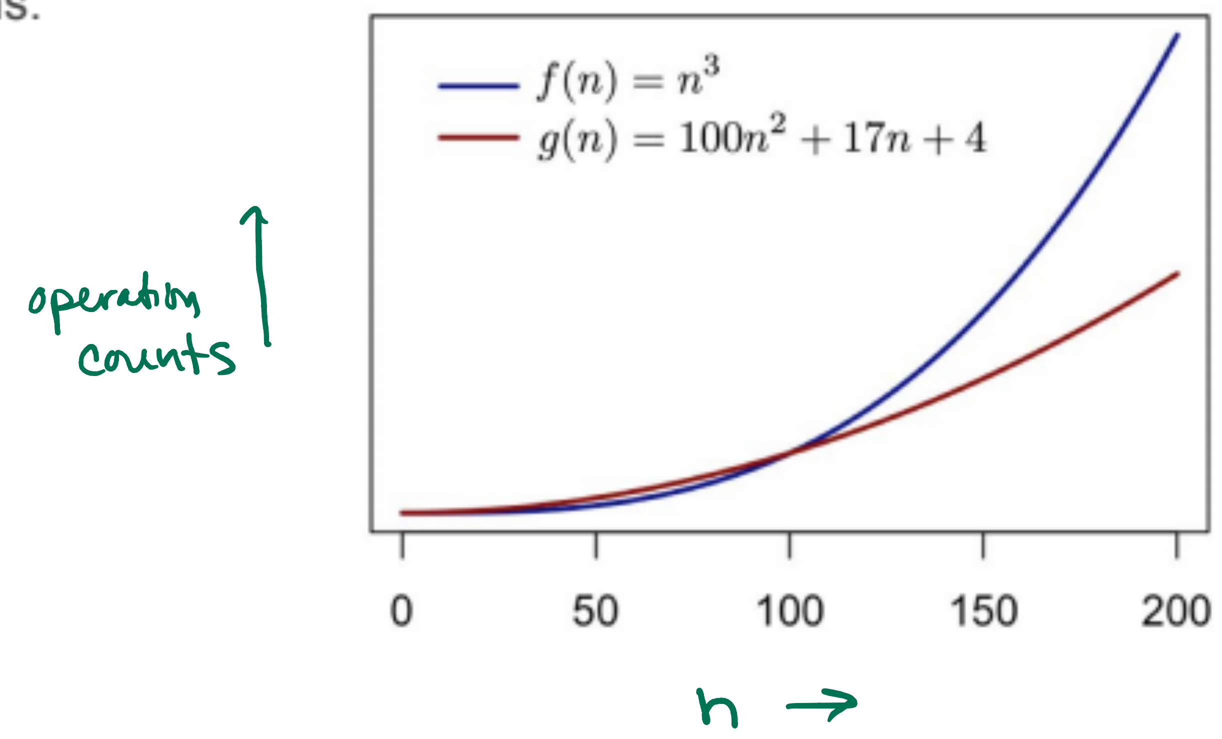
$$\text{Time (long list)} = (2n)^2 = 4n^2$$

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

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?



What we've done:

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

Next:

More Complexity and Matrices

$\log_2(n)$

$n + 3$

$n^2 + 7n$

n^3

As $n \to \infty$ $n^3 > n^2$

$n^3 > n^2 + 7n$

As $n \to \infty$ 10 100 1000

$n = 10$:

n^3: 1000

$n^2 + 7n$: 100 + 70

- - - - - - - - - - - - - -

$n = 100$ n^3: 1000000

$n^2 + 7n$: 10,000 + 700

- - - - - - - - - - - - - -

$n = 1000$

n^3: 1,000,000,000

$n^2 + 7n$: 1000000 + 7000