UNIT 4C
Iteration - Sorting
Iteration: Scalability & Big O Notation

Last Time

- Insertion Sort Algorithm

Writing the Python code

```
def isort(items):
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

insert a[i] into a[0..i] in its correct sorted position

Moving left: examples

76:

```
a = [26, 53, 76, 30, 14, 91, 68, 42]
```

Searching from right to left starting with 53, the first element less than 76 is 53. Insert 76 to the right of 53 (where it was before).

14:

```
a = [26, 30, 53, 76, 14, 91, 68, 42]
```

Searching from right to left starting with 76, all elements left of 14 are greater than 14. Insert 14 into position 0.

68:

```
a = [14, 26, 30, 53, 76, 91, 68, 42]
```

Searching from right to left starting with 91, the first element less than 68 is 53. Insert 68 to the right of 53.

The `move_left` algorithm

Given a list *a* of length *n*, *n* > 0 and a value at index *i* to be "moved left" in the list.

1. Remove *a*[*i*] from the list and store in *x*.
2. Set *j* = *i-1.*
3. While *j* >= 0 and *a*[*j*] > *x*, do the following:
   a. Subtract 1 from *j*.
4. (**At this point, what do we know? Either *j* is ..., or *a*[*j*] is ...**) Reinsert *x* into position *a*[*j*+1].

From algorithm to code

- Our algorithm says to ***remove*** and ***insert*** elements of a list.

How do we do that?

- There are built –in Python operations for that

Removing a list element: pop

```
>>> a = ["Wednesday", "Monday", "Tuesday"]

>>> day = a.pop(1)
>>> a
['Wednesday', 'Tuesday']
>>> day
'Monday'

>>> day = a.pop(0)
>>> day
'Wednesday‘
>>> a
['Tuesday']
```

Inserting an element: insert

```
a = [10, 20, 30]  → [10, 20, 30]

a.insert(0,"foo") → ["foo", 10, 20, 30]

a.insert(2,"bar") → ["foo", 10, "bar", 20, 30]

a.insert(5,"baz") → ["foo", 10, "bar", 20, 30, "baz"]
```

move_left in Python

```
def move_left(items, i):
    x = items.pop(i)
    j = i - 1
    while j >= 0 AND items[j] > x:
        j = j – 1
    items.insert(j + 1, x)
```

remove the item at position i in list a and store it in x

logical operator AND: both conditions must be true for the loop to continue

insert x at position j+1 of list a, shifting all elements from j+1 and beyond over one position

Insertion sort with a bug

```
def move_left(items, i):
    # Insert the element at items[i] into its place
    x = items.pop(i)
    j = i - 1
    while j > 0 and items[j] > x:
        j = j - 1
    items.insert(j + 1, x)

def isort(items):
   # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

Why should we believe our code works?

- We can test it:

```
>>> data = [13, 78, 18, 25, 100, 89, 12]

>>> isort(data)
[13, 12, 18, 25, 78, 89, 100]

>>>
```

- Hmmmm. What went wrong?

Using assert to debug

- What do we know has to be true for move_left to do the right thing?

- We have a loop that decreases j and checks for an element at index j smaller than or equal to x. When should it stop looping?
  - When the value of j is -1,
  - or when the item at index j is <= x

```
j == -1 or items[j] <= x
```

So, insert assertions

```
def move_left(items, i):
    # Insert the element at items[i] into its place
    x = items.pop(i)
    j = i - 1
    while j > 0 and items[j] > x:
        j = j – 1
    assert(j == -1 or items[j] <= x)
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

Run the same test again

```
>>> data = [13, 78, 18, 25, 100, 89, 12]
>>> isort(data)
[13,  12, 18, 25, 78, 89, 100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "isort.py", line 16, in isort
    move_left(items, i)
  File "isort.py", line 7, in move_left
    assert(j == -1 or items[j] <= x)
AssertionError
```

This tells us we did something wrong with the loop!

Where's the bug?

```
def move_left(items, i):
    # Insert the element at items[i] into its place
    x = items.pop(i)
    j = i - 1
    while j > 0 and items[j] > x:
        j = j – 1
    assert(j == -1 or items[j] <= x)
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

FALSE!
Why?????

The fix

```
def move_left(items, i):
    # Insert the element at items[i] into its place
    x = items.pop(i)
    j = i - 1
    while j >= 0 and items[j] > x:
        j = j – 1
    assert(j == -1 or items[j] <= x)
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

Run the same test again

```
>>> data = [13, 78, 18, 25, 100, 89, 12]
>>> isort(data)
[12, 13, 18, 25, 78, 89, 100]
```

Hurray!

Do we know for sure that the program will always do the right thing now?

This Lecture

- Now it is time to think about our programs and do some analyses like a computer scientist

Efficiency

- A computer program should be **correct**, but it should also
  - execute as quickly as possible (**time-efficiency**)
  - use memory wisely (**storage-efficiency**)

- How do we compare programs (or algorithms in general) with respect to execution time?
  - various computers run at different speeds due to different processors
  - compilers optimize code before execution
  - the same algorithm can be written differently depending on the programming paradigm

Counting Operations

- We measure time efficiency by considering **"work" done**
  – Counting the **number of operations** performed by the algorithm.

- But what is an "**operation**"?
  – assignment statements
  – comparisons
  – function calls
  – return statements

Think of it in a machine-independent way

- **We think of an operation as <u>any computation</u> that is independent of the size of our input.**

Linear Search

```
# let n = the length of list.
def search(list, key):
    index = 0
    while index < len(list):
        if list[index] == key:
            return index
        index = index + 1
    return None
```

Best case: the key is the first element in the list

Linear Search: Best Case

```
# let n = the length of list.
def search(list, key):
  index = 0                                   1
  while index < len(list):                    1
     if list[index] == key:                   1
          return index                        1
     index = index + 1
  return None
                              Total:          4
```

Linear Search: Worst Case

```
# let n = the length of list.
def search(list, key):
  index = 0
  while index < len(list):
      if list[index] == key:
           return index
      index = index + 1
  return None
```

Worst case: the key is not an element in the list

Linear Search: Worst Case

```
# let n = the length of list.
def search(list, key):
  index = 0                                 1
  while index < len(list):                  n+1
     if list[index] == key:                 n
          return index
     index = index + 1                      n
  return None                               1
                                  Total:    3n+3
```

Asymptotic Analysis

- How do we know that each operation we count takes the same amount of time?
  - We don't.

- So generally, we look at the process more abstractly
  - We care about the behavior of a program in the long run (on large input sizes)
  - We don't care about constant factors (we care about how many iterations we make, not how many operations we have to do in each iteration)

What Do We Gain?

- Show important characteristics in terms of resource requirements
- Suppress tedious details
- Matches the outcomes in practice quite well
- As long as operations are faster than some constant (1 ns? 1 μs? 1 year?), it does not matter

Linear Search: Best Case Simplified

```
# let n = the length of list.
def search(list, key):
  index = 0
  while index < len(list):     1 iteration
     if list[index] == key:
          return index
     index = index + 1
  return None
```

Linear Search: Worst Case Simplified

```
# let n = the length of list.
def search(list, key):
  index = 0
  while index < len(list):    n iterations
     if list[index] == key:
          return index
     index = index + 1
  return None
```

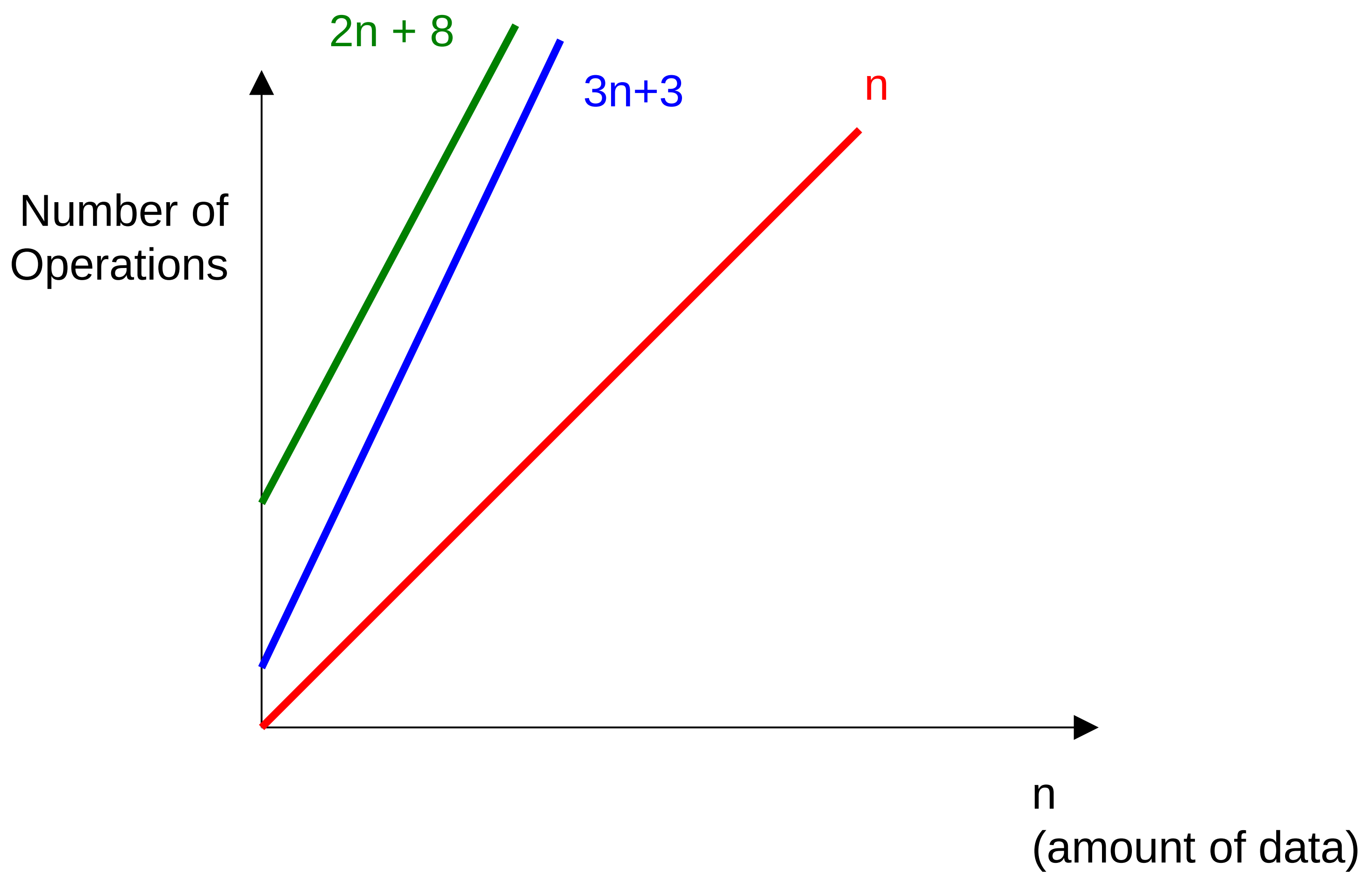
Order of Complexity

- For very large n, we express the number of operations as the (time) order of complexity.
- For asymptotic upper bound, order of complexity is often expressed using Big-O notation:

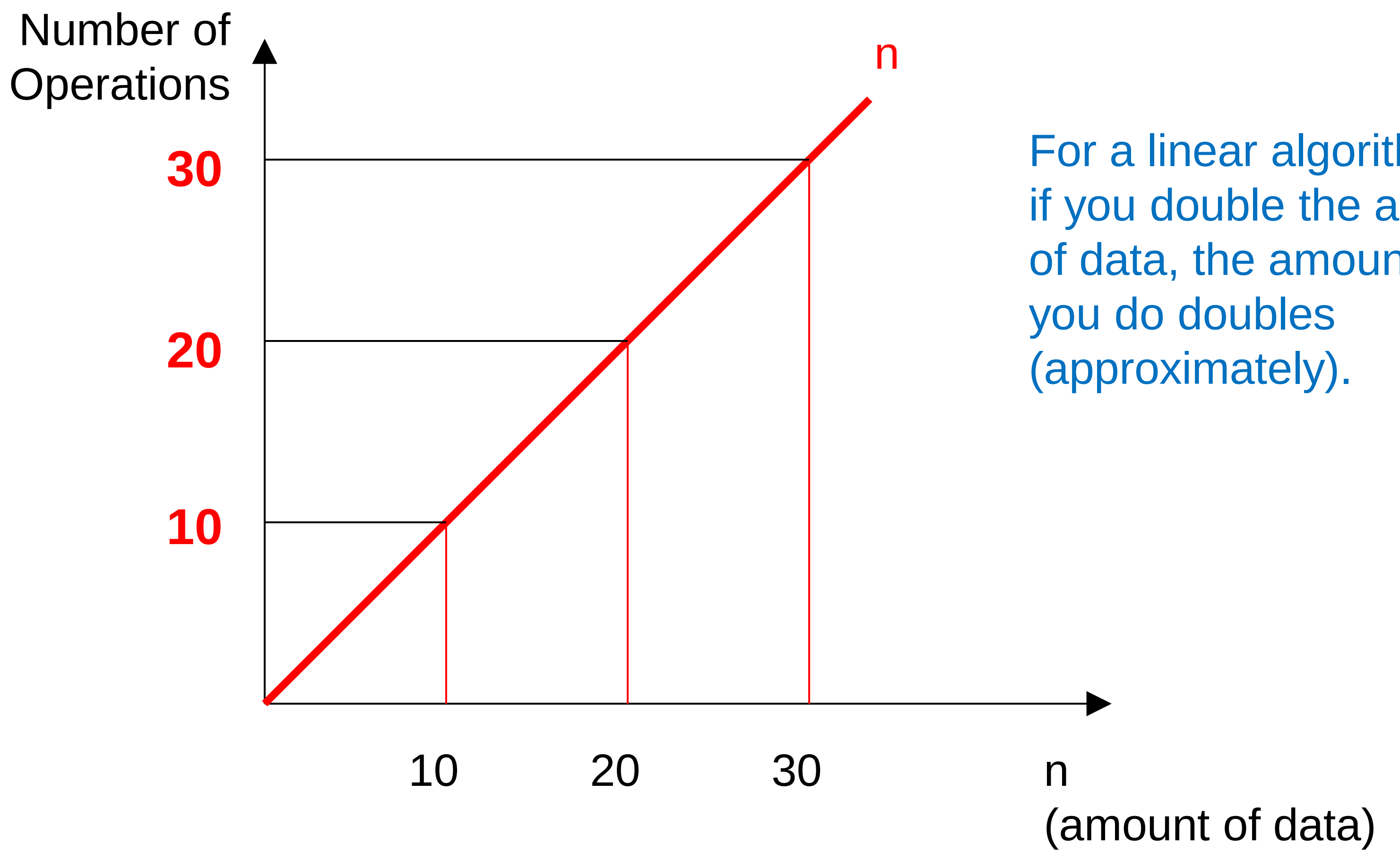
Number of operations	Order of Complexity
n	O(n)
3n+3	O(n)
2n+8	O(n)

Usually doesn't matter what the constants are... we are only concerned about the highest power of n.

O(n) ("Linear")

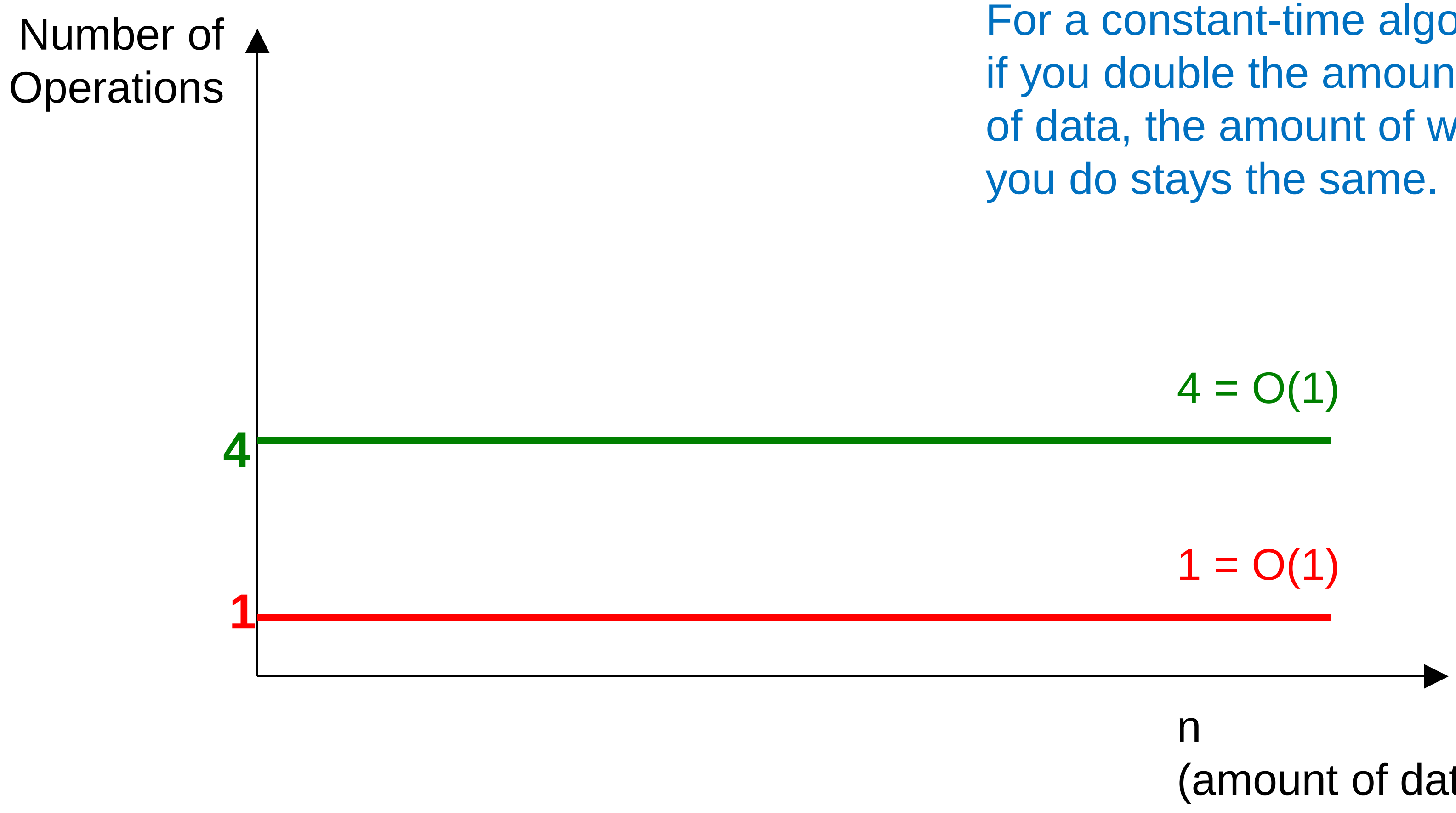


O(n)



For a linear algorithm, if you double the amount of data, the amount of work you do doubles (approximately).

O(1) ("Constant-Time")



For a constant-time algorithm, if you double the amount of data, the amount of work you do stays the same.

Linear Search

- Best Case: O(1)
- Worst Case: O(n)
- Average Case: ?
  - Depends on the distribution of queries
  - But can't be worse than O(n)

Insertion Sort

```
# let n = the length of list.
def isort(list):
    i = 1
    while i != len(list):  n-1 iterations
        move_left(list, i)
        i = i + 1
    return list
```

Insertion Sort: Worst Case

- When i = 1, move_left shifts at most 1 element.
- When i = 2, move_left shifts at most 2 elements.
- ...
- When i = n-1, move_left shifts at most n-1 elements.
- The maximum number of elements shifted, S, approximates the total amount of work done in the worst case.
- $S = 1 + 2 + ... + (n-1) = n(n-1)/2 = O(n^2)$

In general by calculating shifts

move_left

```
# let n = the length of list.
def move_left(a, i):                           at most
      x = a.pop(i)
      j = i - 1
      while j >= 0 and a[j] > x: i iterations
            j = j – 1
      a.insert(j + 1, x)
```

but how long do pop and insert take?

Calculating in Detail

Measuring pop and insert

2 million elements in list, 1000 inserts:0.7548720836639404 seconds
4 million elements in list, 1000 inserts:1.6343820095062256 seconds
8 million elements in list, 1000 inserts:3.327040195465088 seconds

8 million elements in list, 1000 pops:2.031071901321411 seconds
16 million elements in list, 1000 pops:4.033380031585693 seconds
32 million elements in list, 1000 pops:8.06456995010376 seconds

Doubling the size of the list doubles the cost (time) of insert or pop. These functions take **linear time.**

move_left

```
# let n = the length of list.
def move_left(a, i):
    x = a.pop(i)                   n iterations
    j = i - 1
    while j >= 0 and a[j] > x: i iterations
        j = j – 1
    a.insert(j + 1, x)             n iterations
```

Insertion Sort: what is the cost of move_left?

```
# let n = the length of list.
def move_left(a, i):
    x = a.pop(i)                    n iterations
    j = i - 1
    while j >= 0 and a[j] > x:      i iterations
        j = j – 1
    a.insert(j + 1, x)              n iterations
```

Total cost (at most): **n + i + n**

But what is **i** ? To find out, look at isort, which calls move_left, supplying a value for **i**

Insertion Sort: what is the cost of the whole thing?

```
# let n = the length of list.
def isort(list):
      i = 1
      while i != len(list):    #n-1 iterations
            move_left(list,i) #i goes from 1 to n-1
            i = i + 1
      return list
```

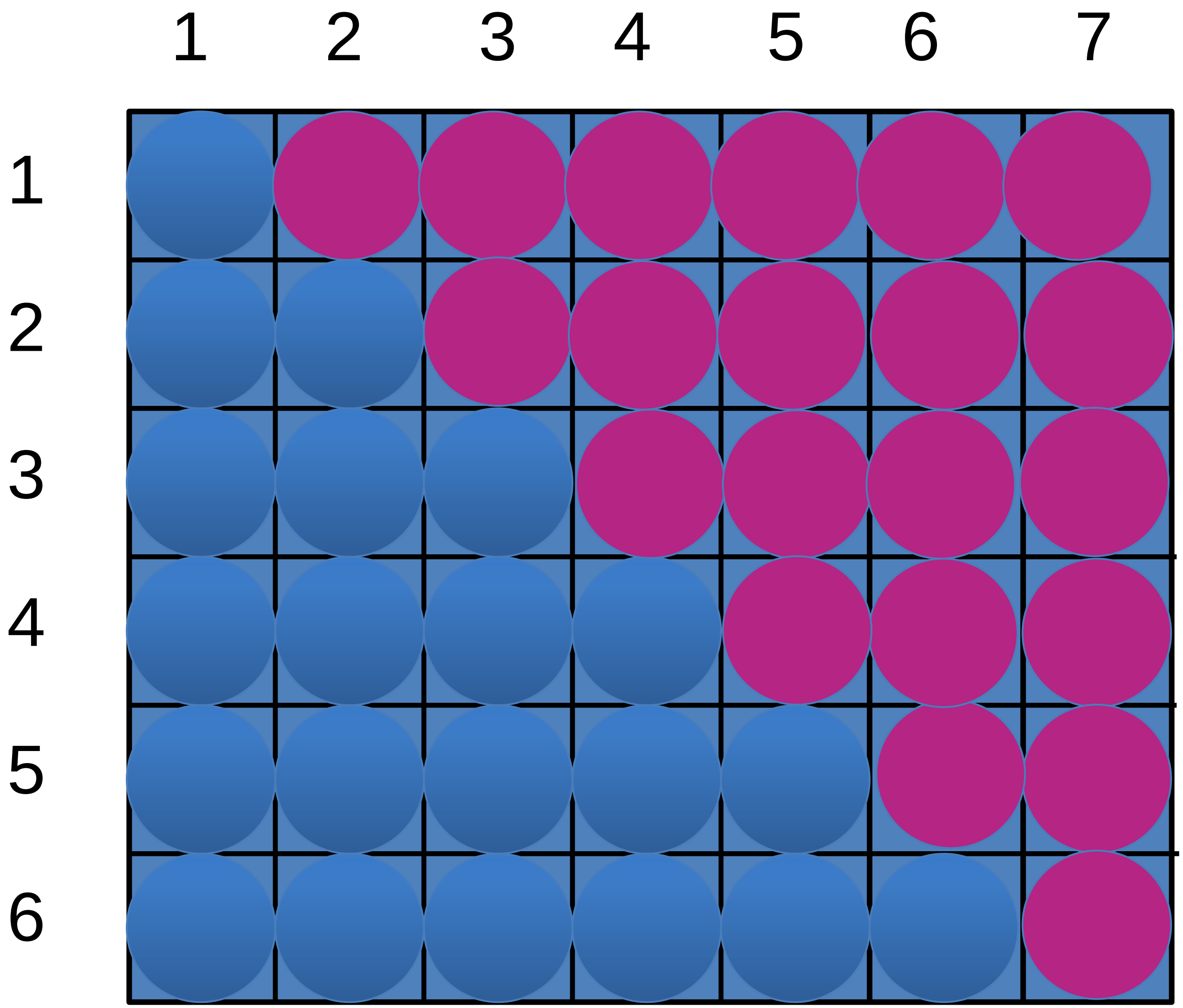
```
Total cost: cost of move_left as i goes from 1 to n-1

Cost of all the move_lefts:  n + 1 + n
                           + n + 2 + n
                           + n + 3 + n
                             ...
                           + n + n-1 + n
```

Figuring out the sum

- `n + 1 + n`
- `+ n + 2 + n`
- `+ n + 3 + n`
- `...`
- `+ n + n-1 + n`

```
(n-1)*2n
+ 1
+ 2
+ 3
...
+ n-1
```

Adding 1 through n-1



(6 * 7) / 2
blue circles

Adding 1 through n-1

- We saw 1 + 2 + ... + 6 = (6 * 7) / 2
- Generalizing, 1 + 2 + ... + n-1 = (n-1)(n) / 2
- So our whole cost is:

```
  (n-1)*2n + 1 + 2 + 3 ... + n-1
= (n-1)*2n + (n-1)(n) / 2
```

$= 2n^2 - 2n + (n^2 - n) / 2$

$= (5n^2 - 5n) / 2 = (5/2)n^2 - (5/2)n$

Observe that the highest-order term is n^2

Order of Complexity

Number of operations	Order of Complexity
n^2	$O(n^2)$
$(5/2)n^2 - (1/2)n$	$O(n^2)$
$2n^2 + 7$	$O(n^2)$

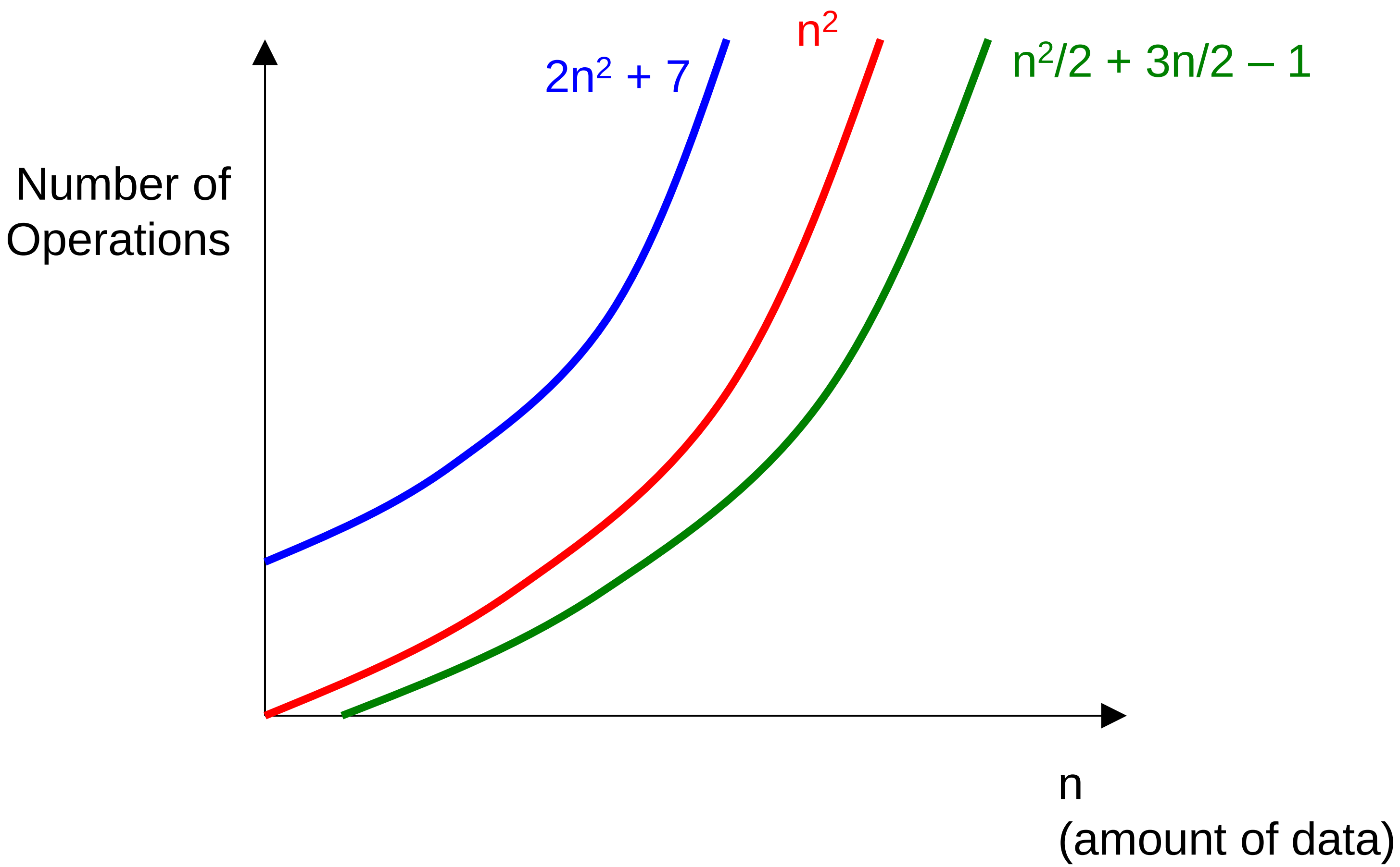
Usually doesn't matter what the constants are… we are only concerned about the highest power of n.

$f(n)$ is $O(g(n))$ means $f(n) < g(n) \cdot k$ for some positive k

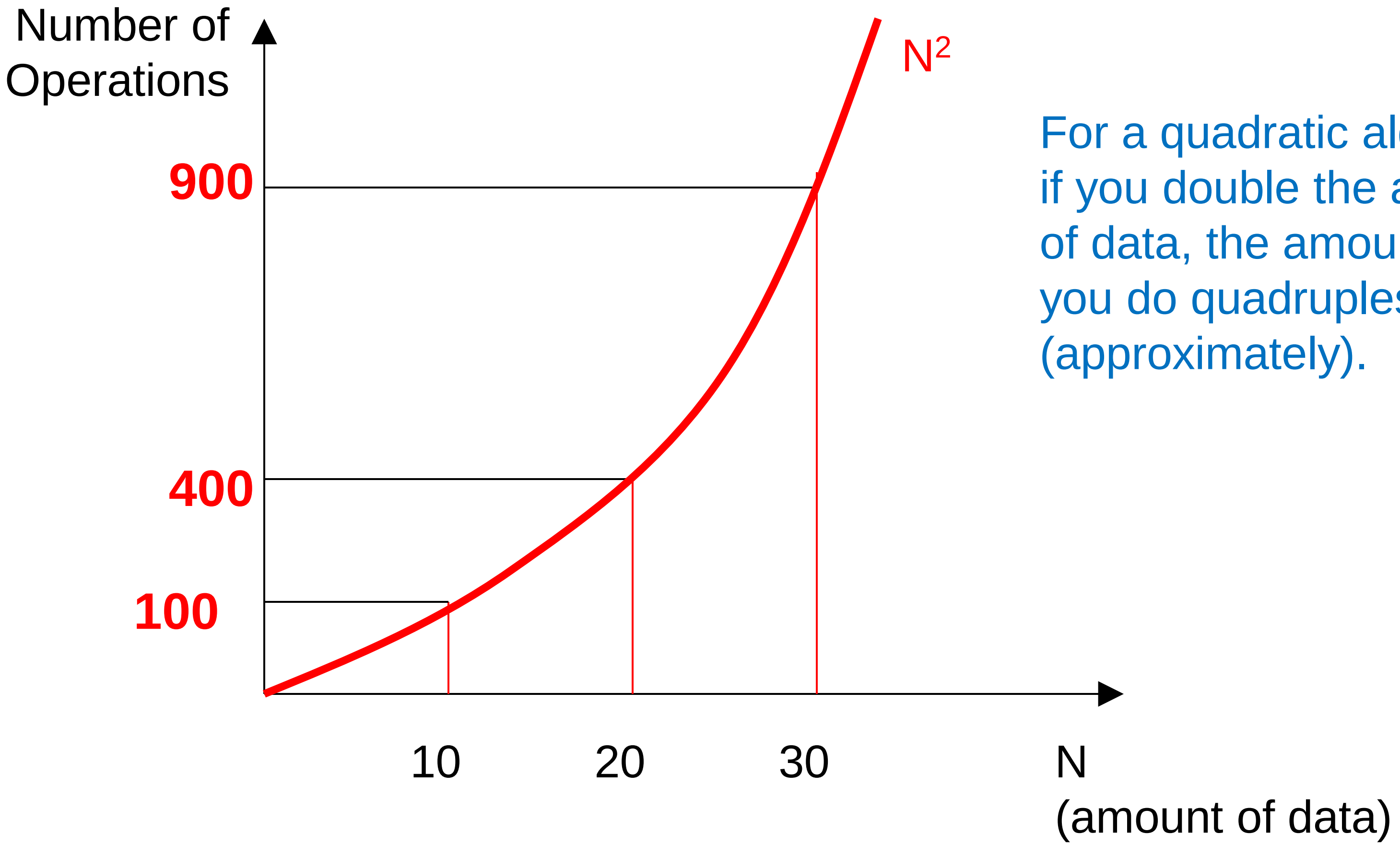
Keep It Simple

- "Big O" notation expresses an upper bound:
  $f(n)$ is O($g(n)$) means $f(n) < g(n) \cdot k$
  (whenever n is large enough)
- So if $f(x)$ is $O(n^2)$, then $f(x)$ is $O(n^3)$ too!
- But we always use the smallest possible function, and the simplest possible.
- We say $3n^2 + 4n + 1$ is $O(n^2)$, not $O(n^3)$
- We say $3n^2 + 4n + 1$ is $O(n^2)$, not $O(3n^2 + 4n)$
- ...even though all of the above are true

$O(n^2)$ ("Quadratic")



$O(n^2)$



For a quadratic algorithm, if you double the amount of data, the amount of work you do quadruples (approximately).

Tomorrow

- A new technique called recursion
- More sorting and searching using recursion
- Do the online module on recursion as a preparation for the next lecture

Now - Review