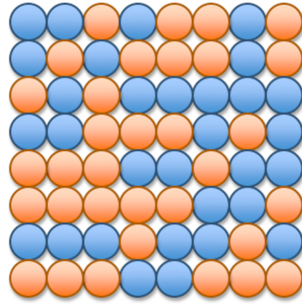
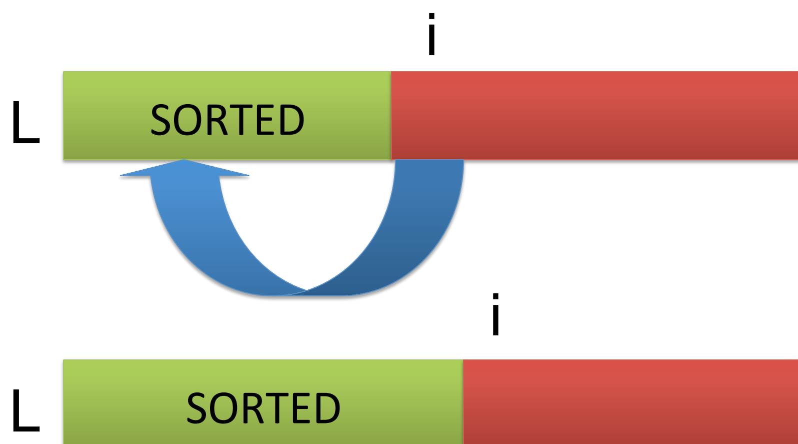


UNIT 4C
Iteration:
Correctness and Efficiency

General Idea: Any one Iteration



Look Closer at Insertion Sort

L[0..i) means:
List L from index 0
up to but not including i

Given a list L of length n, n > 0.

1. Set i = 1.
2. While i is not equal to n, do the following:

   Precondition for each iteration: L[0..i) is sorted

   a. Insert L[i] into its correct position in L between index 0 and index i inclusive.

   b. Add 1 to i.

   Postcondition for each iteration: L[0..i) is sorted
3. Return the list L which will now be sorted.

Look Closer at Insertion Sort

Given a list L of length n, n > 0.

1. Set i = 1.
2. While i is not equal to n, do the following:

   Loop invariant: L[0..i) is sorted

   a. Insert L[i] into its correct position in L between index 0 and index i inclusive.

   b. Add 1 to i.
3. Return the list L which will now be sorted.

A <u>loop invariant</u> is a condition that is true at the start and end of each iteration of a loop.

Reasoning with the Loop Invariant

The loop invariant is true at the end of each iteration, including the last iteration. After the last iteration, when we go to step 3:

L[0..i) is sorted (from the last iteration)

AND

i is equal to n (due to the while loop terminating)

These 2 conditions imply that L[0..n) is sorted, but this range is the entire list, so the list must always be sorted when we return our final answer!

Counting Operations

- We measure time efficiency by counting the number of operations performed by the algorithm.
- But what is an operation?
  - assignment statements
  - comparisons
  - return statements
  - ...

Linear Search: Worst Case

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


Counting Operations

- 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 and <u>count whatever operation depends on the amount or size of the data</u> we're processing.
  - We don't consider what machine we're using, what compiler we use, what language we use, etc.
- For linear search, we would <u>count the number of times we compare elements in the list</u> to the key.

Linear Search: Worst Case Simplified

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

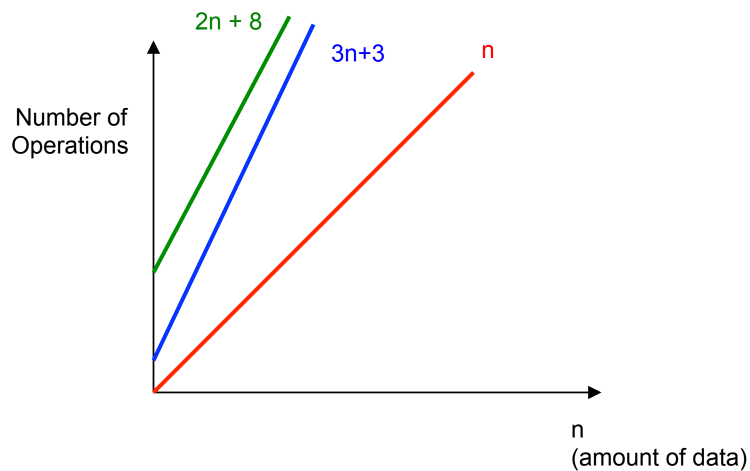

Order of Complexity

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

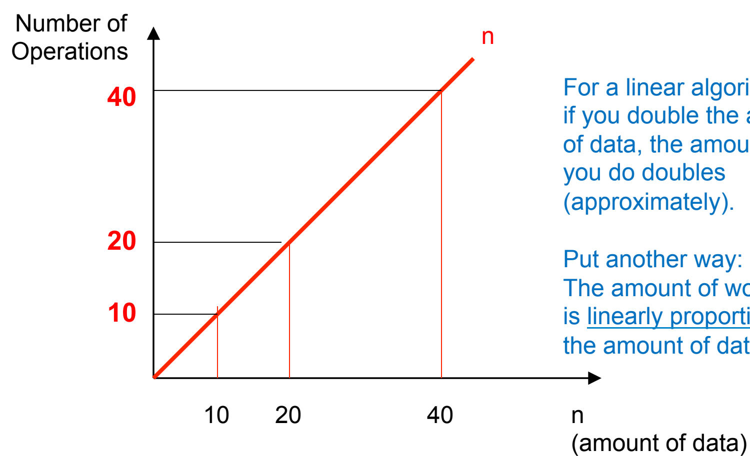
Number of operations	Order of Complexity
n	O(n)
3n+3	O(n)
2n+8	O(n)

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).

Put another way:
The amount of work done is linearly proportional to the amount of data.

Linear Search: Best Case

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

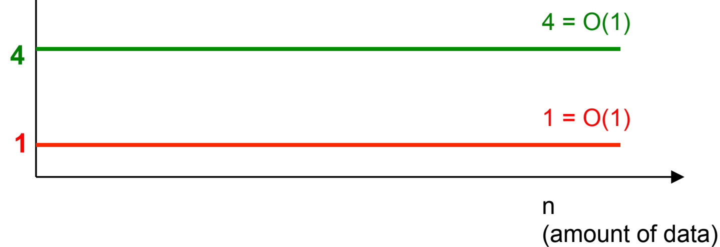

Linear Search: Best Case Simplified

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

O(1) (“Constant-Time”)

Number of Operations

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

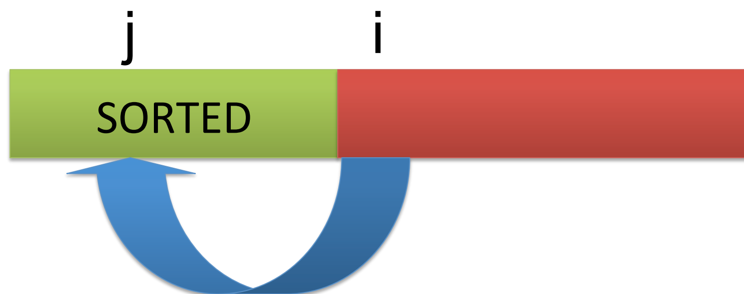


n
(amount of data)

Linear Search

- Worst Case: O(n)
- Best Case: O(1)
- Average Case: ____________

Insertion Sort: Worst Case



- On iteration i, we need to examine j elements and then shift i-j elements to the right, so we have to do j + (i-j) = i units of work.

Insertion Sort: Worst Case

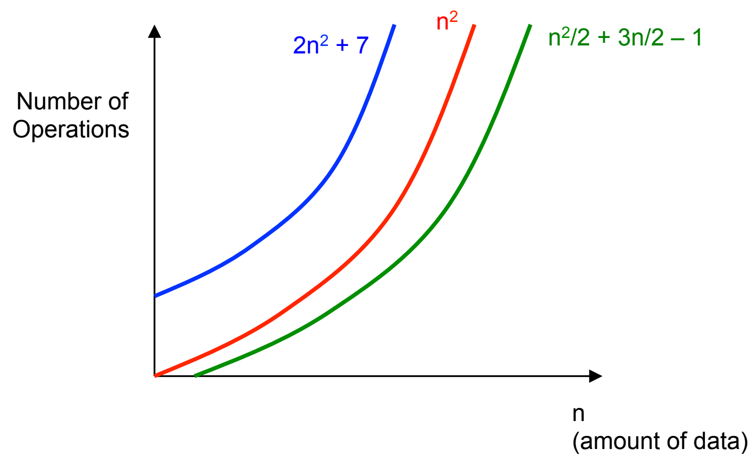
- When i = 1, we have 1 unit of work.
- When i = 2, we have 2 units of work.
- ...
- When i = n-1, we have n-1 units of work.
- The total amount of work done is:
  1 + 2 + ... + (n-1)
  = n(n-1)/2
  = $(n^2 - n)/2$ (a quadratic function)
  = $O(n^2)$

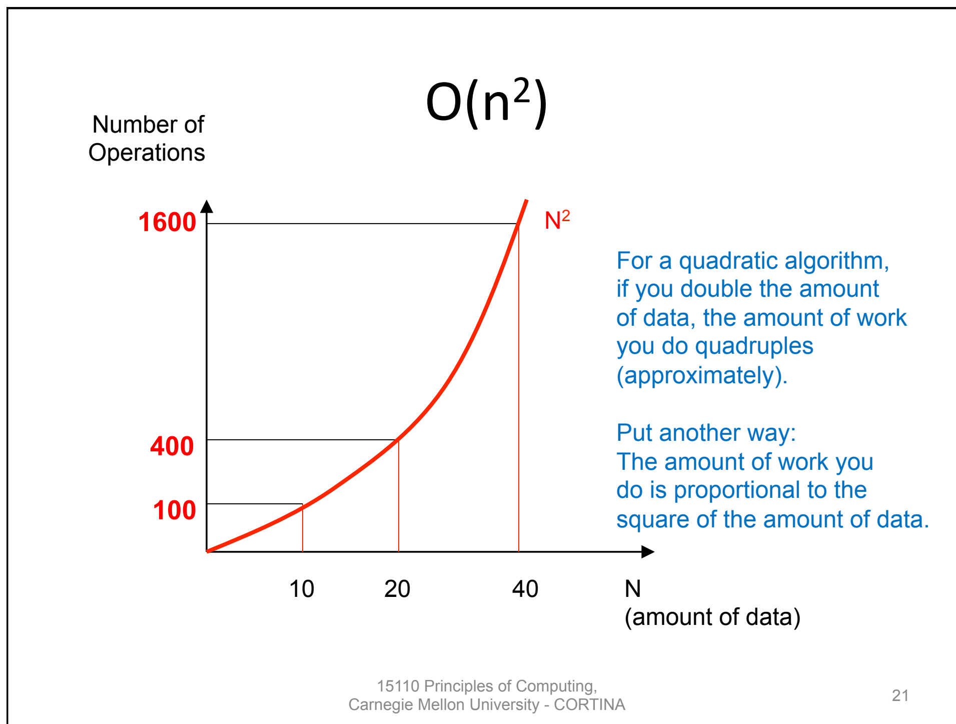
Order of Complexity

Number of operations	Order of Complexity
n^2	$O(n^2)$
$n^2/2 + 3n/2 - 1$	$O(n^2)$
$2n^2 + 7$	$O(n^2)$

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

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





Our Insertion Sort

- Worst Case: $O(n^2)$
- Best Case: In our insertion sort implementation, the worst case and best case are the same!
  It doesn't matter where we do the inserts.
  - If we insert near the front, we have fewer elements to compare, but more shifts.
  - If we insert near the end, we have more elements to compare, but fewer shifts.
  - But we can get the best case to be O(n). (See PS4!)