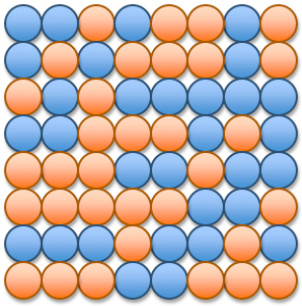


UNIT 4C
Iteration: Scalability & Big O

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 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
```


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
```

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 count whatever operation depends on the amount or size of the data we're processing.
- For linear search, we would count the number of times we compare elements in the list to the key.

Linear Search: Worst Case Simplified

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

Linear Search: Best Case Simplified

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

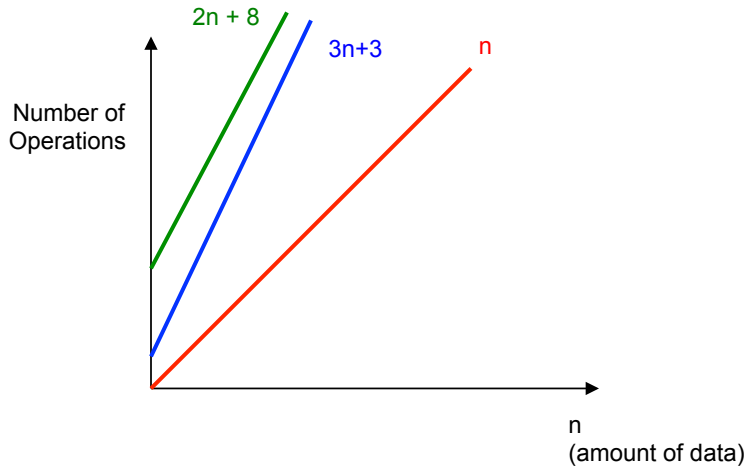

Order of Complexity

- 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:

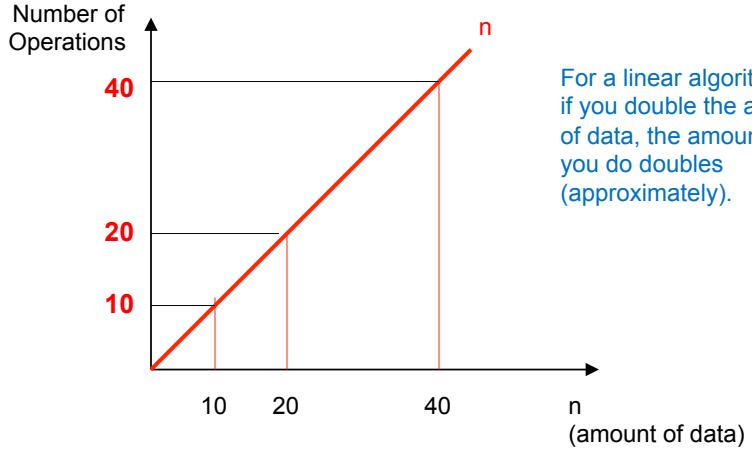
Number of operations	Order of Complexity
n	O(n)
3n+3	O(n)
2n+8	O(n)

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

O(n) ("Linear")

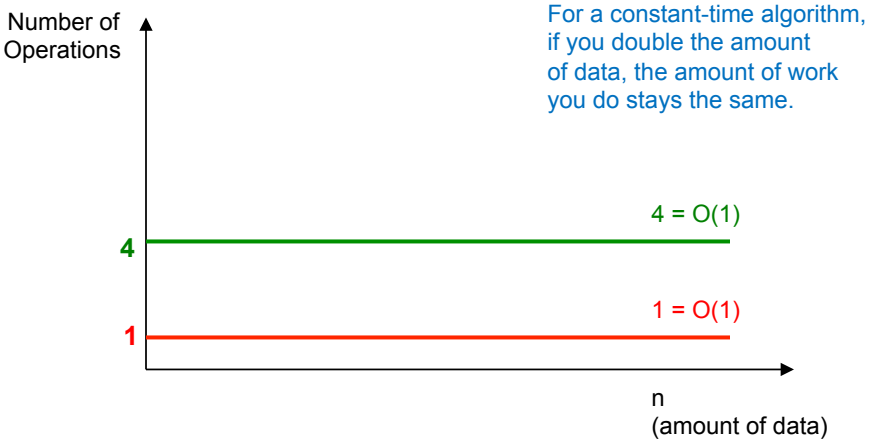


O(n)



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

O(1) ("Constant-Time")



Linear Search

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

Insertion Sort: Worst Case

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

There are `n-1 move_left` operations. But how many operations does each `move_left` take?

Insertion Sort: Worst Case

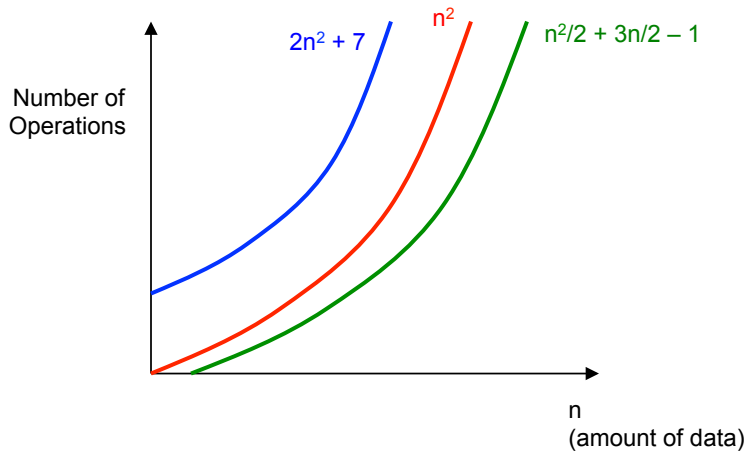
- When i = 1, move_left shifts at most 1 element.
- When i = 2, move_left shifts at most 2 elements.
- ...
- When i = n-1, move_left shifts at most n-1 elements.
- The maximum number of elements shifted, S, approximates the total amount of work done in the worst case.
- $S = 1 + 2 + ... + (n-1) = n(n-1)/2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

Order of Complexity

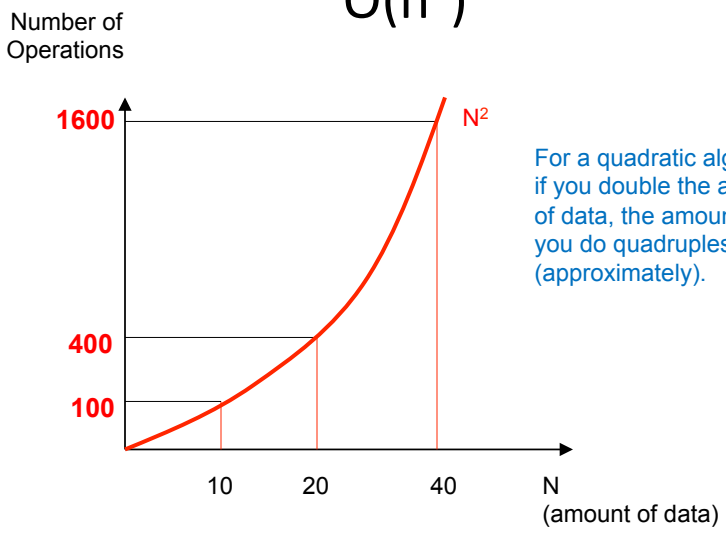
Number of operations	Order of Complexity
n^2	$O(n^2)$
$n^2/2 + 3n/2 - 1$	$O(n^2)$
$2n^2 + 7$	$O(n^2)$

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

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



$O(n^2)$



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

Insertion Sort

- Worst Case: $O(n^2)$
- Best Case: $O(n)$ Why?

We'll compare these algorithms with others soon to see how scalable they really are based on their order of complexities.