UNIT 4C
Iteration: Scalability & Big O

Efficiency

- A computer program should be totally 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 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 < list.length do       n+1
    if list[index] == key then       n
        return index
    end
    index = index + 1                n
  end
  return nil                         1
end                        Total:    3n+3
```

Linear Search: Best Case

```
# let n = the length of list.
def search(list, key)
  index = 0                              1
  while index < list.length do           1
    if list[index] == key then           1
        return index                     1
    end
    index = index + 1
  end
  return nil
end                            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 array to the key.

Linear Search: Worst Case Simplified

```
# let n = the length of list.
def search(list, key)
  index = 0
  while index < list.length do
     if list[index] == key then        n
         return index
     end
     index = index + 1
  end
  return nil
end                               Total:    n
```

Linear Search: Best Case Simplified

```
# let n = the length of list.
def search(list, key)
  index = 0
  while index < list.length do
     if list[index] == key then        1
         return index
     end
     index = index + 1
  end
  return nil
end                               Total:    1
```

Order of Complexity

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

<u>Number of operations</u>	<u>Order of Complexity</u>	
n	O(n)	**Usually doesn't matter what the constants are... we are only concerned about the highest power of n.**
3n+3	O(n)	
2n+8	O(n)	

O(n) ("Linear")

O(n)

Number of
Operations

n

30

20

10

10   20   30   n
(amount of data)
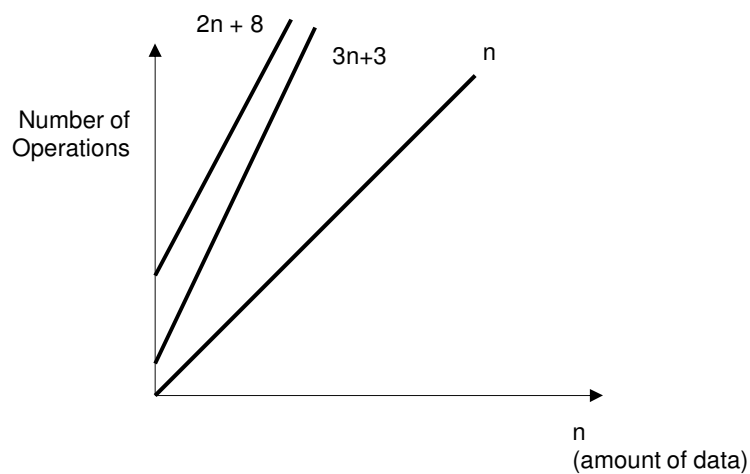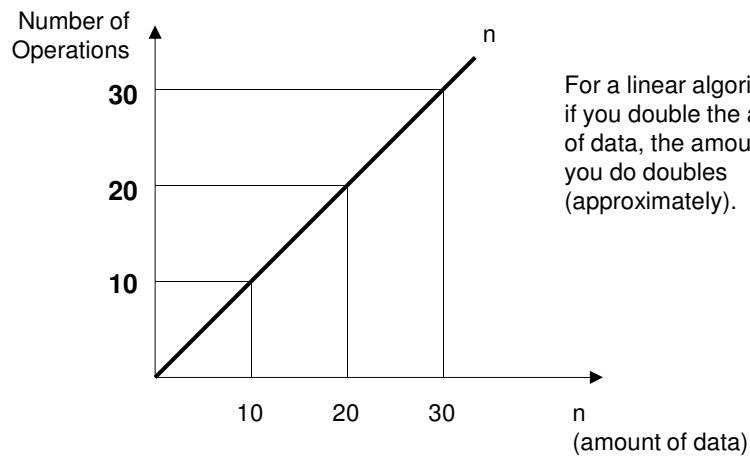
For a linear algorithm,
if you double the amount
of data, the amount of work
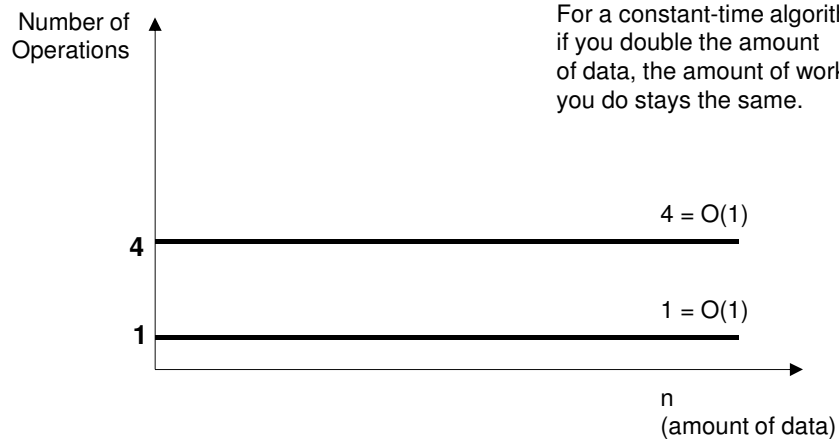you do doubles
(approximately).

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.

4 = O(1)

4

1 = O(1)

1

n
(amount of data)

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)
    a = list.clone                n
    i = 1
    while i != a.length do
        move_left(a, i)           n-1
        i = i + 1
    end
    return a
end
```

Insertion Sort: Worst Case

```
# let n = the length of list.
def move_left(a, i)
    x = a.slice!(i)
    j = i-1
    while j >= 0 && a[j] > x do        i+1
        j = j - 1
    end
    a.insert(j+1, x)
end
```

but how long do **slice!** and **insert** take?

move_left (alternate version)

```
# let n = the length of list.
def move_left(a, i)
    x = a[i]
    j = i-1
    while j >= 0 && a[j] > x do        i+1
        a[j+1] = a[j]
        j = j - 1
    end
    a[j+1] = x
end
```
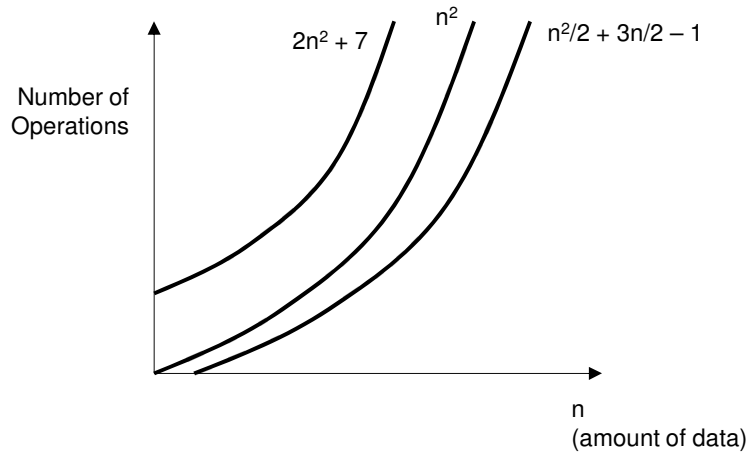
Insertion Sort: Worst Case

- So the total number of operations is
  n + (n-1 move_left's)
- But each move_left performs i+1 operations,
  where i varies from 1 to n-1:
- n-1 move_left's = 2 + 3 + 4 + ... + n
- Since 1 + 2 + ... + n = n(n+1)/2,
  n-1 move_left's = n(n+1)/2 − 1
- The total number of operations is:
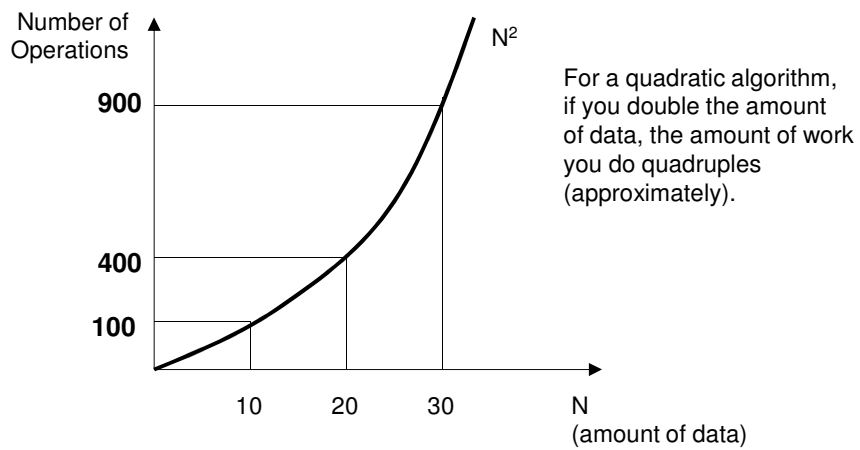 $n + n(n+1)/2 − 1 = n + n^2/2 + n/2 − 1 = n^2/2 + 3n/2 − 1$

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")

Number of Operations

$2n^2 + 7$ n^2 $n^2/2 + 3n/2 - 1$

n
(amount of data)

O(n^2)

Number of Operations

N^2

900

400

100

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

10   20   30   N
(amount of data)

Insertion Sort

- Worst Case: $O(n^2)$

- Best Case: ?

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