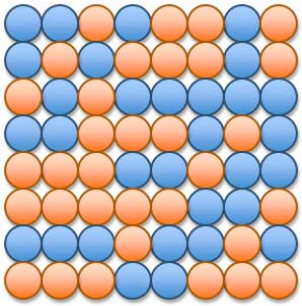


UNIT 4C
Iteration: Scalability & Big O

Course Announcements

- No collaboration on assignments, including in the help sessions. If you need help, ask a CA.
- Questions about grading?
  - Ask your CA to explain the grade.
  - If issue can't be resolved, talk to Dave or Dilsun.
  - You have one week from the time you get your grades on an assignment to contest the grading.
- First written exam is Monday, October 1.
  - We will have a review session on Sunday.

Loop Invariants Again

- What is a loop invariant?
  - A statement about the computation that remains true every time around the loop.
- Why are they useful?
  - They form the basis of an *inductive proof* that the algorithm produces the desired result.

Example: Sum Of Array Elements

```
def sum(list)
  result = 0
  index = 0
  while index < list.length do
    result = result + list[index]
    index = index + 1
  end
  return result
end
```

```
>> sum([3, 7, 2, 4, 6])
```

What Is the Invariant?

- Must be true at start of body: a precondition.
- Might not be true in the middle of the body.
- Must be true at end of body: a postcondition.

index <= list.length and
result == sum of list[0..index-1]

Example: Sum Of Array Elements

```
def sum(list)
  result = 0
  index = 0
  while index < list.length do
    result = result + list[index]
    index = index + 1
  end
  return result
end

>> sum([3, 7, 2, 4, 6])
```

result is 0
index is 0

result is 3
index is 1

Example: Sum Of Array Elements

```
def sum(list)
  result = 0
  index = 0
  while index < list.length do
    result = result + list[index]
    index = index + 1
  end
  return result
end

>> sum([3, 7, 2, 4, 6])
```

result is 3
index is 1

result is 10
index is 2

The Loop Invariant Is Always True

>> sum([3, 7, 2, 4, 6])

result	index
0	0
3	1
10	2
12	3
16	4
22	5

index <= list.length and
result == sum of
list[0..index-1]

After the Loop

- When do we exit the WHILE loop?
  - When index == list.length
- What does the invariant tell us at this point?
  - result == sum of list[0..index-1]
  - So result == sum of list[0..list.length-1]
  - But that's everything in the list!
- Therefore, the result returned by sum must be the sum of every element in the list.
- We've proved the function is correct.

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
  - function calls
  - return statements
  - ...

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
```

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
```


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: 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
```


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
```

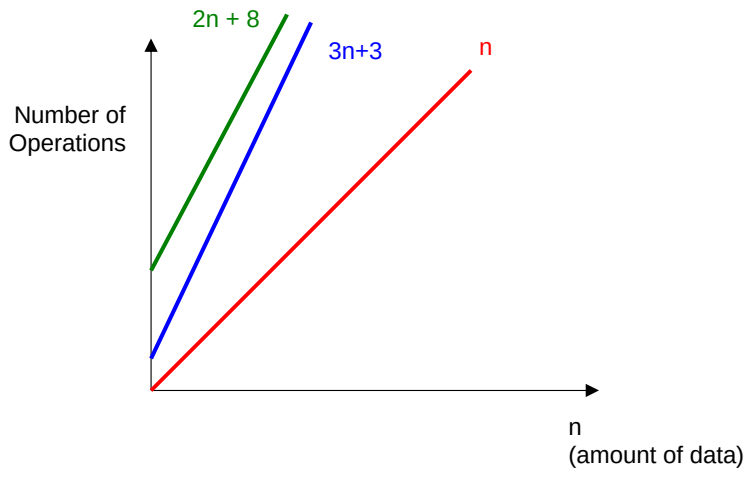
Order of Complexity

- 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:

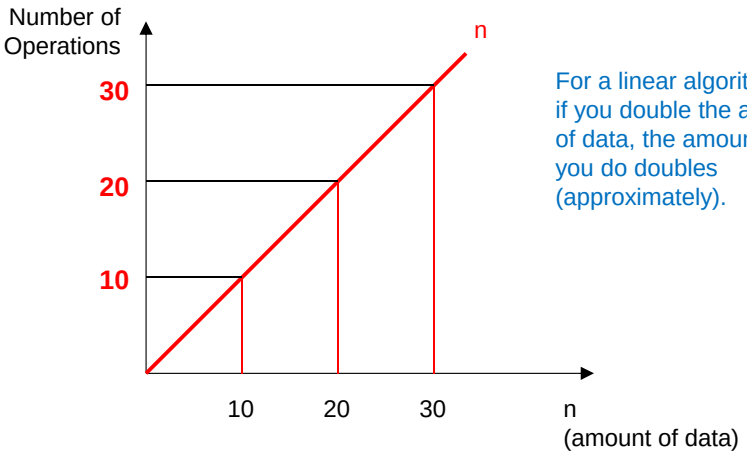
Number of operations	Order of Complexity
n	O(n)
3n+3	O(n)
2n+8	O(n)

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

O(n) ("Linear")

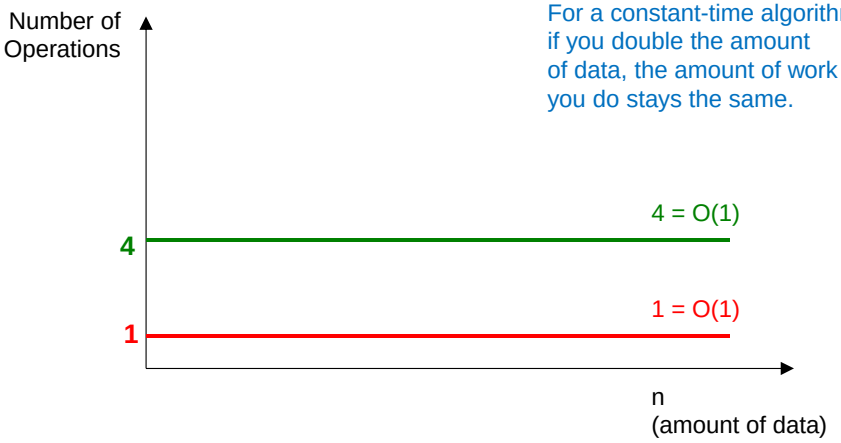


O(n)



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

O(1) ("Constant-Time")



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: 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

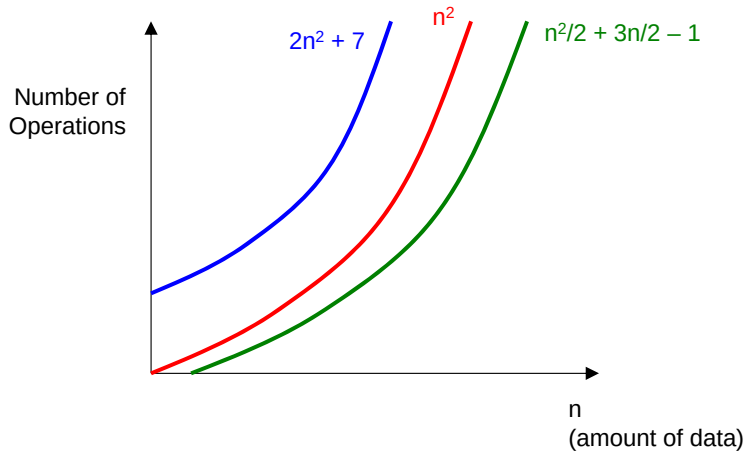
- So the total number of operations is
  (n for list.clone) + (n-1 move_left's)
- But each move_left performs i+1 operations,
  where i varies from 1 to n-1:
- n-1 move_left's = 2 + 3 + 4 + ... + n operations
- Since 1 + 2 + ... + n = n(n+1)/2,
  n-1 move_left's = n(n+1)/2 – 1
- The total number of operations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

Order of Complexity

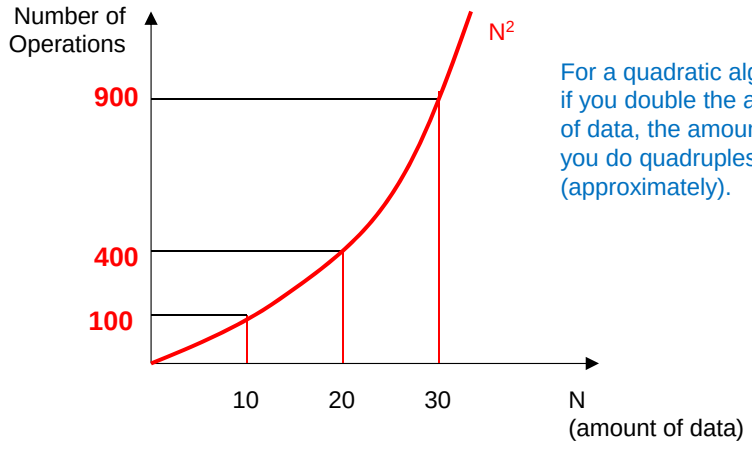
Number of operations	Order of Complexity
n^2	$O(n^2)$
$n^2/2 + 3n/2 - 1$	$O(n^2)$
$2n^2 + 7$	$O(n^2)$

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

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



$O(n^2)$



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: ?
- Average Case: ?

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