UNIT 4C
Iteration: Scalability & Big O

Announcements

- If you feel that the course is slipping away please contact the instructors immediately
- The written exam is on Wed. February 20. We will offer
  - A sample exam
  - Review sessions
- No programming assignment is due exam's week but there will be a problem set

After 2 Weeks of Programming

some happy moments ...

After 2 weeks of Programming

some angry moments

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

Linear Search

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

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 < 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
  while index < list.length do
     if list[index] == key then
          return index
     end
     index = index + 1
  end
  return nil
end
```

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 < 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
```


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

Linear Search: Best Case Simplified

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

Linear Search: Worst Case Simplified

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

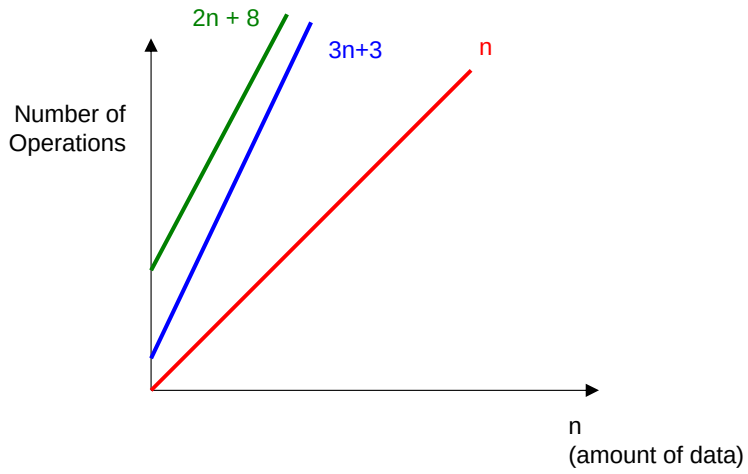

Order of Complexity

- 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:

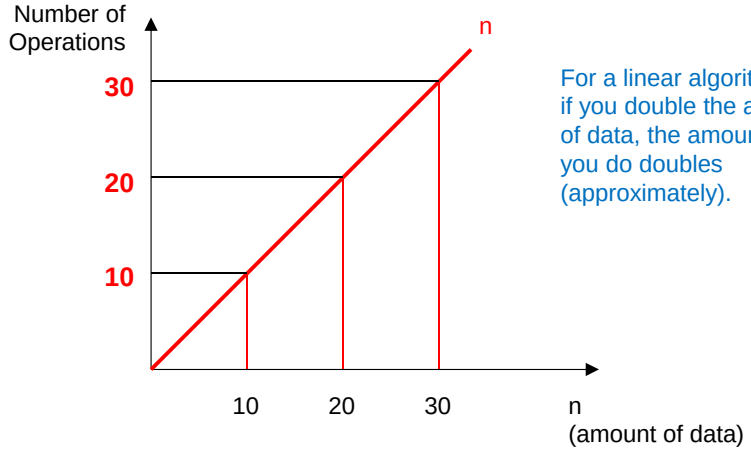
Number of operations	Order of Complexity
n	O(n)
3n+3	O(n)
2n+8	O(n)

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

O(n) (“Linear”)

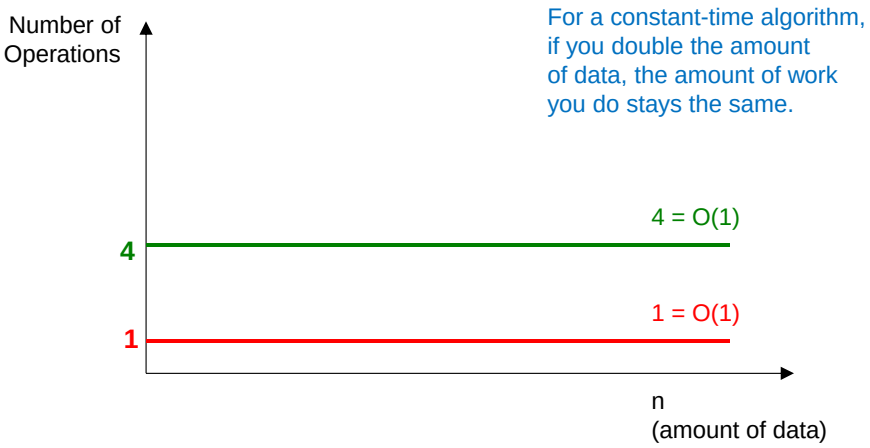


O(n)



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

O(1) ("Constant-Time")



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)

Recall Insertion Sort

```
def isort (list)
  result = [ ]
  for val in list do
    # some code here to find the
    # place to insert val
    result.insert(place, val)
  end
  return result
end
```

Constructing the result array by inserting each element in its right place

Insertion Sort (Destructive)

- Instead of constructing a new sorted list from scratch, we will “modify” the list we have to sort

Insertion Sort (destructive)

```
# let n = the length of list.
# first copy the original argument array
# "list" so that it is not modified
def isort(list)
     a = list.clone
     i = 1
     while i != a.length do
          move_left(a, i)
          i = i + 1
     end
     return a
end
```


Insertion Sort (move_left)

```
# let n = the length of list.
def move_left(a, i)
     x = a[i] # x is val to be put in its place
     j = i-1
     while j >= 0 && a[j] > x do
          a[j+1] = a[j]
          j = j - 1
     end
     a[j+1] = x
end
```

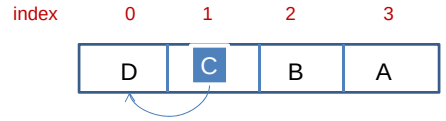
Insertion Sort: Worst Case

```
# let n = the length of list.
# first copy the argument array “list” so
# that it is not modified
def isort(list)
     a = list.clone
     i = 1
     while i != a.length do    n-1 iter.
          move_left(a, i)
          i = i + 1
     end
     return a
end
```


Move_left: Worst case

```
# let n = the length of list.
def move_left(a, i)
     x = a[i] # x is val to be put in its place
     j = i-1
     while j >= 0 && a[j] > x do   i iter.
          a[j+1] = a[j]
          j = j – 1
     end
     a[j+1] = x
end
```

Example: Tracing isort

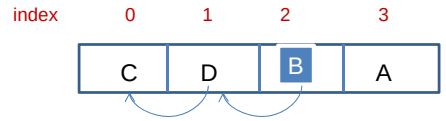


move_left(a,1): 1 iteration

1. C is less then D
   So, D moves to index 1 to make room C
Finally, C is written to index 0

Suppose we want to sort in ascending order

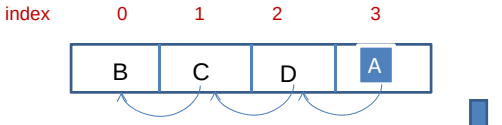
Example: Tracing isort



move_left(a,2): 2 iterations

1. B is less then D
   So, D moves to index 2 to make room for B
2. B is also less then C
   So, C moves to index 1 to make room for B
Finally, B is written to index 0

Example: Tracing isort

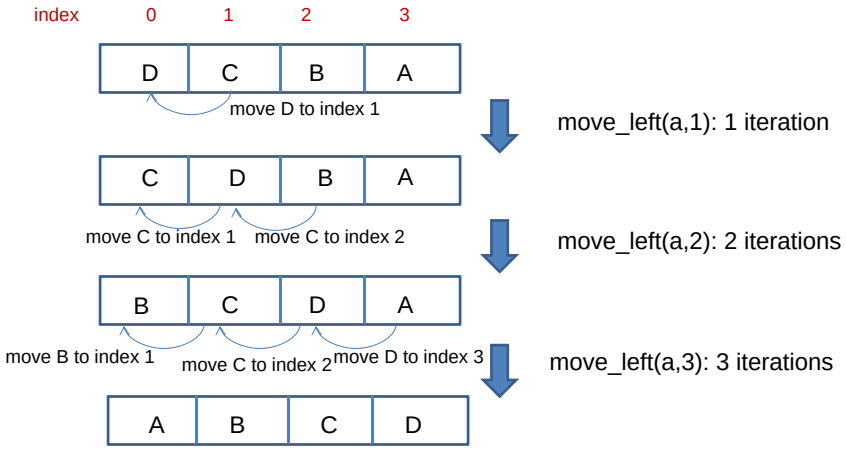


move_left(a,3): 3 iterations

1. A is less then D
   So, D moves to index 3 to make room for A
2. A is also less then C
   So, C moves to index 2 to make room for A
3. A is also less then B
   So, B moves to index 1 to make room for A

Finally, A is written to index 0

Example: Tracing isort



Insertion Sort: Worst Case (generalized)

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

Adding 1 through n

	1	+	2	+	3	+	...	+ (n-1)	SUM
+	(n-1)	+	(n-2)	+	(n-3)	+	...	+ 1	SUM
	n	+	n	+	n	+	...	+ n	

- $2 \times SUM = n \times (n-1)$, $SUM = n \times (n-1) / 2$
- The total number of operations is:
  $n + n \times (n-1)/2$, $n + n^2/2 + n/2 = n^2/2 + 3n/2$

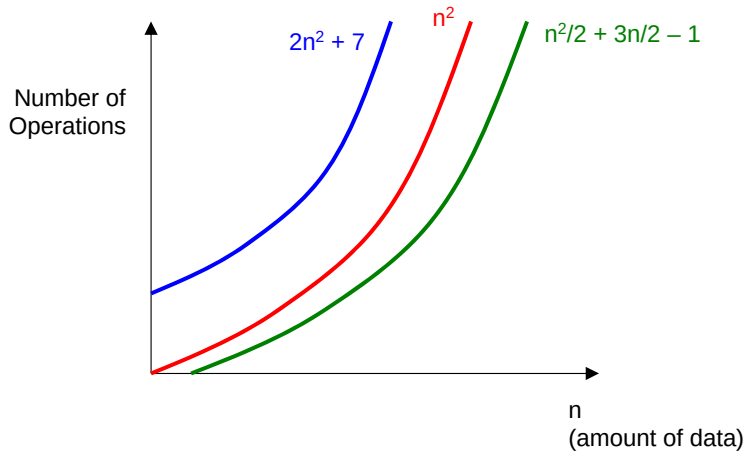
Observe that the highest ordered term is n^2

Order of Complexity

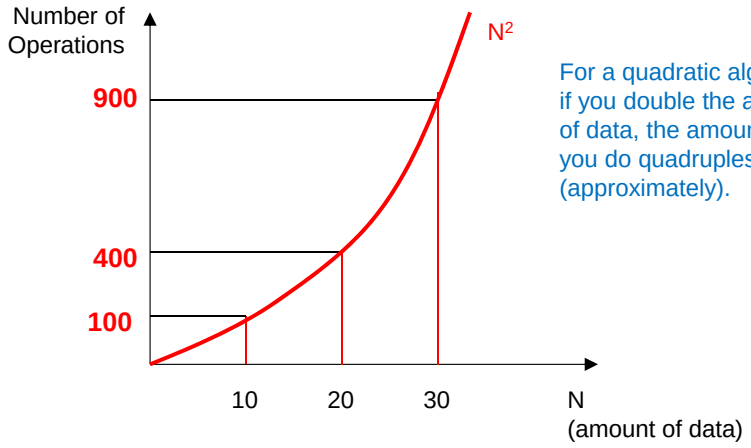
Number of operations	Order of Complexity
n^2	$O(n^2)$
$n^2/2 + 3n/2$	$O(n^2)$
$2n^2 + 7$	$O(n^2)$

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.

Next Week

- A new technique called recursion
- More sorting and searching using recursion
- Do the online module on recursion as a preparation for the next lecture (see problem set 4)

move_left (alternate version)

```
# 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 iter.
          j = j - 1
     end
     a.insert(j+1, x)
end
```

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