Iteration:
Sorting, Scalability, Big O Notation
Announcements

- Yesterday?
  - Lab 4
  - PA 3
  - OLI

- Tonight
  - PS 4
  - Lab 5
  - Extension on PA 4, now due tomorrow

- Tomorrow
  - PS 5
  - PA 4
Yesterday

- Sieve of Eratosthenes
- Character Comparisons (Unicode)
- Linear Search
- Sorting
Today

- Insertion Sort
- Scalability
- Big O Notation
Sorting
In-place Insertion Sort

- **Idea:** during sorting, a *prefix* of the list is *already sorted.* (This prefix might contain one, two, or more elements.)

- Each element that we process is inserted into the correct place in the sorted prefix of the list.

- Result: sorted part of the list gets bigger until the whole thing is sorted.
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort

sorted part
In-place Insertion Sort Algorithm

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

1. Set $i = 1$.

2. While $i$ is not equal to $n$, do the following:
   a. Insert $a[i]$ into its correct position in $a[0]$ to $a[i]$ (inclusive).
   b. Add 1 to $i$.

3. Return the list $a$ (which is now sorted).
Example

\[ a = [53, 26, 76, 30, 14, 91, 68, 42] \]

\[ i = 1 \]

Insert \( a[1] \) into its correct position in \( a[0..1] \) and then add 1 to \( i \):

53 moves to the right,
26 is inserted into the list at position 0

\[ a = [26, 53, 76, 30, 14, 91, 68, 42] \]

\[ i = 2 \]
def isort(items):
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items

insert a[i] into a[0..i] in its correct sorted position
But now we have to write the `move_left` function!
Moving left using search

To move the element x at index i “left” to its correct position, remove it, start at position i-1, and search from right to left until we find the first element that is less than or equal to x.

Then insert x back into the list to the right of that element.

(The Python insert operation does not overwrite. Think of it as “squeezing into the list”.)
move_left via linear search

sorted part
move_left via linear search
move_left via linear search

sorted part
In-place Insertion Sort

sorted part
76:

\[ a = [26, \underline{53}, \underline{76}, 30, 14, 91, 68, 42] \]

Searching from right to left starting with 53, the first element less than 76 is 53. Insert 76 to the right of 53 (where it was before).

14:

\[ a = [26, 30, \underline{53}, \underline{76}, 14, 91, 68, 42] \]

Searching from right to left starting with 76, all elements left of 14 are greater than 14. Insert 14 into position 0.

68:

\[ a = [14, 26, 30, 53, 76, 91, \underline{68}, 42] \]

Searching from right to left starting with 91, the first element less than 68 is 53. Insert 68 to the right of 53.
The move_left algorithm

Given a list $a$ of length $n$, $n > 0$ and a value at index $i$ to be moved left in the list.

1. Remove $a[i]$ from the list and store in $x$.
2. Set $j = i-1$.
3. While $j \geq 0$ and $a[j] > x$, subtract 1 from $j$.
4. (At this point, what do we know? Either $j$ is ..., or $a[j]$ is ...) Insert $x$ into position $a[j+1]$. 
Our algorithm says to “remove” and “insert” elements of a list.

But how do we do that?

Fortunately there are built-in Python operations for that.
Removing a list element: pop

```python
>>> a = ["Wednesday", "Monday", "Tuesday"]
>>> day = a.pop(1)
>>> a
['Wednesday', 'Tuesday']
>>> day
'Monday'
>>> day = a.pop(0)
>>> day
'Wednesday'
>>> a
['Tuesday']
```
Inserting an element: insert

```python
>> a = [10, 20, 30]
=> [10, 20, 30]
>> a.insert(0, "foo")
=> ["foo", 10, 20, 30]
>> a.insert(2, "bar")
=> ["foo", 10, "bar", 20, 30]
>> a.insert(5, "baz")
=> ["foo", 10, "bar", 20, 30, "baz"]
```
def move_left(items, i):
    x = items.pop(i)
    j = i - 1
    while j >= 0 and items[j] > x:
        j = j - 1
    items.insert(j + 1, x)
def move_left(items, i):
    # Insert the element at items[i] into its place
    x = items.pop(i)
    j = i - 1
    while j > 0 and items[j] > x:
        j = j - 1
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
Why should we believe our code works?

- We can test it:

```python
>>> data = [13, 78, 18, 25, 100, 89, 12]
>>> isort(data)
[13, 12, 18, 25, 78, 89, 100]

>>> 
```

- Hmmmm. What went wrong?
Using assert to debug

- What do we know has to be true for move_left to do the right thing?

- We have a loop that decreases j and checks for an element at index j smaller than or equal to x. **When should it stop looping?**
  - When the value of j is -1,
  - or when the item at index j is <= x
  - j == -1 or items[j] <= x
def move_left(items, i):
    # Insert the element at items[i] into its place
    x = items.pop(i)
    j = i - 1
    while j > 0 and items[j] > x:
        j = j - 1
    assert(j == -1 or items[j] <= x)
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
Run the same test again

```python
>>> data = [13, 78, 18, 25, 100, 89, 12]
>>> isort(data)
[13, 12, 18, 25, 78, 89, 100]
Traceback (most recent call last):
  File "<stdin>"", line 1, in <module>
  File "isort.py", line 16, in isort
      move_left(items, i)
  File "isort.py", line 7, in move_left
    assert(j == -1 or items[j] <= x)
AssertionError
```

This tells us we did something wrong with the loop!
Where’s the bug?

```python
def move_left(items, i):
    # Insert the element at items[i] into its place
    x = items.pop(i)
    j = i - 1
    while j > 0 and items[j] > x:
        j = j - 1
    assert(j == -1 or items[j] <= x)
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
```

**FALSE!**
Why??????
The fix

def move_left(items, i):
    # Insert the element at items[i] into its place
    x = items.pop(i)
    j = i - 1
    while j >= 0 and items[j] > x:
        j = j - 1
    assert(j == -1 or items[j] <= x)
    items.insert(j + 1, x)

def isort(items):
    # In-place insertion sort
    i = 1
    while i < len(items):
        move_left(items, i)
        i = i + 1
    return items
Run the same test again

```python
>>> data = [13, 78, 18, 25, 100, 89, 12]
>>> isort(data)
[12, 13, 18, 25, 78, 89, 100]
```

Hurray!

Do we know for sure that the program will always do the right thing now?
Thinking like a computer scientist

Code Analysis
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

- We think of an operation as any computation that is independent of the size of our input.
Linear Search

# let n = the length of list.

def search(list, key):
    index = 0
    while index < len(list):
        if list[index] == key:
            return index
        index = index + 1
    return None

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
    while index < len(list):
        if list[index] == key:
            return index
        index = index + 1
    return None

Total: 4
# let n = the length of list.

def search(list, key):
    index = 0
    while index < len(list):
        if list[index] == key:
            return index
        index = index + 1
    return None

Worst case: the key is not an element in the list
# let n = the length of list.

def search(list, key):
    index = 0
    while index < len(list):
        if list[index] == key:
            return index
        index = index + 1
    return None

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
- As long as operations are faster than some constant (1 ns? 1 μs? 1 year?), it does not matter
# let n = the length of list.
def search(list, key):
    index = 0
    while index < len(list):
        if list[index] == key:
            return index
        index = index + 1
    return None
# let n = the length of list.

def search(list, key):
    index = 0
    while index < len(list):
        n iterations
        if list[index] == key:
            return index
        return None
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:

<table>
<thead>
<tr>
<th>Number of operations</th>
<th>Order of Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$3n+3$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$2n+8$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Usually doesn't matter what the constants are... we are only concerned about the highest power of $n$. 
Why don’t constants matter?

\[(n=1) \quad 45n^3 + 20n^2 + 19 = 84\]

\[(n=2) \quad 45n^3 + 20n^2 + 19 = 459\]

\[(n=3) \quad 45n^3 + 20n^2 + 19 = 1414\]
O(n) ("Linear")

\[ \text{Number of Operations} \]

\[ 2n + 8 \quad 3n + 3 \quad n \]

\[ n \] (amount of data)
For a linear algorithm, if you double the amount of data, the amount of work you do doubles (approximately).
For a constant-time algorithm, if you double the amount of data, the amount of work you do stays the same.

4 = O(1)

1 = O(1)
Linear Search

- Best Case: $O(1)$
- Worst Case: $O(n)$
- Average Case: ?
  - Depends on the distribution of queries
  - But can’t be worse than $O(n)$
# let n = the length of list.

def isort(list):
    i = 1
    while i != len(list):
        move_left(list, i)
        i = i + 1

    return list
# let n = the length of list.
def move_left(a, i):
    x = a.pop(i)
    j = i - 1
    while j >= 0 and a[j] > x:  # i iterations
        j = j - 1
    a.insert(j + 1, x)

but how long do pop and insert take?
Doubling the size of the list doubles the cost (time) of insert or pop. These functions take linear time.
# let n = the length of list.

def move_left(a, i):
    x = a.pop(i)  # n iterations
    j = i - 1
    while j >= 0 and a[j] > x:  # i iterations
        j = j - 1
    a.insert(j + 1, x)  # n iterations
Insertion Sort: what is the cost of move_left?

# let n = the length of list.

```python
def move_left(a, i):
    x = a.pop(i)                      # n iterations
    j = i - 1
    while j >= 0 and a[j] > x:        # i iterations
        j = j - 1
    a.insert(j + 1, x)               # n iterations
```

Total cost (at most): $n + i + n$

But what is $i$? To find out, look at isort, which calls move_left, supplying a value for $i$
Insertion Sort: what is the cost of the whole thing?

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

Total cost: cost of move_left as i goes from 1 to n-1

Cost of all the move_lefts:
    n + 1 + n
    + n + 2 + n
    + n + 3 + n
    ...
    + n + n-1 + n
```

Total cost: cost of move_left as i goes from 1 to n-1

Cost of all the move_lefts:
    n + 1 + n
    + n + 2 + n
    + n + 3 + n
    ...
    + n + n-1 + n
Figuring out the sum

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

\[(n-1) \times 2n + 1 + 2 + 3 + \ldots + n-1\]
Adding 1 through n-1

\[(6 \times 7) / 2\] blue circles
Adding 1 through n-1

- We saw $1 + 2 + ... + 6 = (6 * 7) / 2$

- Generalizing, $1 + 2 + ... + n-1 = (n-1)(n) / 2$

- So our whole cost is:
  - $(n-1)*2n + 1 + 2 + 3 ... + n-1$
  - $= (n-1)*2n + (n-1)(n) / 2$
  - $= 2n^2 - 2n + (n^2 - n) / 2$
  - $= (5n^2 - 5n) / 2 = (5/2)n^2 - (5/2)n$

- Observe that the highest-order term is $n^2$
## Order of Complexity

<table>
<thead>
<tr>
<th>Number of operations</th>
<th>Order of Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$(5/2)n^2 - (1/2)n$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$2n^2 + 7$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

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

$f(n)$ is $O(g(n))$ means $f(n) < g(n) \cdot k$ for some positive $k$. 
“Big O” notation expresses an upper bound: 
\[ f(n) \text{ is } O(g(n)) \text{ means } f(n) < g(n) \cdot k \] 
(whenever \( n \) is large enough)

So if \( f(x) \) is \( O(n^2) \), then \( f(x) \) is \( O(n^3) \) too!

But we always use the smallest possible function, and the simplest possible.

We say \( 3n^2 + 4n + 1 \) is \( O(n^2) \), not \( O(n^3) \)

We say \( 3n^2 + 4n + 1 \) is \( O(n^2) \), not \( O(3n^2 + 4n) \)

...even though all of the above are true
$O(n^2)$ ("Quadratic")

Number of Operations vs. $n$ (amount of data):
- $2n^2 + 7$
- $n^2$
- $n^2/2 + 3n/2 - 1$
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.
Big O

- $O(1)$: constant
- $O(\log n)$: logarithmic
- $O(n)$: linear
- $O(n \log n)$: log linear
- $O(n^2)$: quadratic
- $O(n^3)$: cubic
- $O(2^n)$: exponential