Announcements

- Midterm next week Monday
- No class next Thursday
- Review this Thursday
More than one way to solve a problem

- There is always more than one way to solve a problem.
  - You can walk different paths from A to B.
- Some solutions are better than others.
- But how can we compare them?
Our programs (functions) implement algorithms

- **Algorithms** are descriptions of computations for solving a problem:
  - To find the min element in a list
    1. Initialize min to the first element
    2. Scan the list
    3. For each element, if it is smaller than min, update min

- Programs (functions for us) are executable interpretations of algorithms:

```
min = L[0]
for x in L:
    if x<min: min = x
```

- The same algorithm can be implemented in many different languages and on many different platforms.
How do we compare algorithms?

- For example, there is more than one way to search.
  - How do we compare algorithms to say that one is faster than another – independent of the hardware platform?

- Computer scientists use something called *Big-O notation*
  - It’s the *order of magnitude* of the running time of an algorithm with large input size

- Big-O notation ignores the differences between languages, even between compiled vs. interpreted, as well as hardware speed.
  - It focuses on the *number of steps* to be executed
  - It focuses on *scalability*
What question are we trying to answer?

- If I am given a larger problem to solve (larger input), how is the performance of the algorithm affected?

- If I change the input, how does the running time change?
How can we determine the complexity?

- Step 1: we must determine the “input,” or what data the algorithm operates over, and its size, measured fairly
  - Testing if $n$ is prime has input size proportional to $\log(n)$

- Step 2: determine how many operations are needed to be done for each piece of the input, measured fairly
  - Determining the smallest element in an unordered list of size $n$ is not constant-time $O(1)$ – it is $O(n)$

- Step 3: eliminate constants and smaller terms for big O notation
E.g., searching a phone book...

- In the first week we introduced the task of searching a phone book as an example algorithm. Let’s compare 3 algorithms:

  - **Algorithm 1:**
    - Start at the front, check each name one by one, until found or coming to the end of the book

  - **Algorithm 2:**
    - Use the index to jump to the correct letter, then search, as before, starting at that point sequentially

  - **Algorithm 3:**
    - Split in half, choose which half has the name, repeat recursively until name found or size is 0 and it is not found
Algorithm 1

# a phone_book is a list of entries
# an phone book entry is [name,number]
# we are searching for key_name

def algorithm1(phone_book, key_name):
    for k in range(len(phone_book)):
        if key_name == phone_book[k][0]:
            return phone_book[k][1]
    return

- Worst case: key_name not in book or close to the end
  - Computational work ~ len(phone_book)
  - We say algorithm1 is O(n) where n is len(phone_book)
  - Each time through loop is O(1) -- constant time
Algorithm 2

```python
# a phone_book is a list of entries
# an phone book entry is [name,number]
# we are searching for key_name

inx1 = first location of name starting key_name[0]
inx2 = last location of name with that letter
def algorithm2(phone_book, key_name, inx1, inx2):
    for k in range(inx1,inx2+1):
        if key_name == phone_book[k][0]:
            return phone_book[k][1]
    return

• Worst case: key_name not in book and lots of names with that first letter
  • Computational work ~ # of names with that letter which is ~ $n$
  • We say algorithm2 is $O(n)$ where $n$ is len(phone_book) in the worst case
  • Each time through loop is $O(1)$ -- constant time
  • There is a constant-factor speed-up immaterial to the scaling behavior
```
Algorithm 3

```python
def algorithm3(book, key, lo, hi):
    k = (hi+lo)/2
    if book[k][0] == key: return book[k][1]
    if book[k][0] < key:
        return algorithm3(book, key, k+1, hi)
    else:
        return algorithm3(book, key, lo, k-1)
    if hi < lo: return "not in book"
return
```

- Worst case: key is not in book
- Computational work $\sim \log_2(n)$, where $n$ is $\text{len(book)}$
- We say algorithm3 is $O(\log(n))$
- Each call is $O(1)$ -- constant time – except for the recursive calls
Nested loops are multiplicative

```python
def loops():
    count = 0
    for x in range(1, 5):
        for y in range(1, 3):
            count = count + 1
            print(x, y, "--Ran it", count, "times")

>>> loops()
1 1 --Ran it  1 times
1 2 --Ran it  2 times
2 1 --Ran it  3 times
2 2 --Ran it  4 times
3 1 --Ran it  5 times
3 2 --Ran it  6 times
4 1 --Ran it  7 times
4 2 --Ran it  8 times
```
Big-O notation ignores constants

- Consider if we executed a particular statement 3 times in the body of the loop
  - If we execute each loop 1 million times this constant becomes meaningless (ie: n = 1,000,000)

- For large n, $O(n^2) \equiv O(3n^2)$

```python
def loops(n):
    count = 0
    for x in range(1,n):
        for y in range(1,n):
            count = count + 1
            count = count + 1
            count = count + 1
```
Let's compare our phone book search algorithms

- **Algorithm 1:**
  - Start at the front, check each name one by one
  - $O(n)$

- **Algorithm 2:**
  - Use the index to jump to the correct letter
  - $O(n/26)$ ... $O(1/26 \times n)$ ... $O(n)$

- **Algorithm 3:**
  - Split in half, choose which half must have the name, repeat until found
  - $O(\log n)$
More on big O notation

- http://en.wikipedia.org/wiki/Big_Oh_notation
  - Additional background

- We mentioned that big O notation ignores constants
  - Let's look at this more formally:
  - Big O notation characterizes functions (algorithms in our case) by their growth rate
Identify the term that has the largest growth rate

<table>
<thead>
<tr>
<th>Num of steps</th>
<th>growth term</th>
<th>asympt. complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>$6n + 3$</td>
<td>$6n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>$2n^2 + 6n + 3$</td>
<td>$2n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>$2n^3 + 6n + 3$</td>
<td>$2n^3$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>$2n^{10} + 2^n + 3$</td>
<td>$2^n$</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td>$n! + 2n^{12} + 2^n + 3$</td>
<td>$n!$</td>
<td>$O(n!)$</td>
</tr>
</tbody>
</table>
Comparison of complexities: fastest to slowest

- \( O(1) \) – constant time
- \( O(\log n) \) – logarithmic time
- \( O(n) \) – linear time
- \( O(n \log n) \) – log linear time
- \( O(n^2) \) – quadratic time
- \( O(2^n) \) – exponential time
- \( O(n!) \) – factorial time
Do we know of any O(1) algorithms?

- These are “constant time” algorithms
- Simple functions that contain no loops are usually O(1)
- Examples:
  - project 1 computation;
  - “real feel” of temperature at certain humidity
Finding something in the phone book

- O(n) algorithm
  - Start from the beginning.
  - Check each page, until you find what you want.

- Not very efficient
  - Best case: One step
  - Worse case: n steps where n = number of pages
  - Average case: n/2 steps
What about algorithm 2?

- Recall that algorithm 2 for finding a name in the phone book used the index.
- Can we make this algorithm faster by having an index for each letter?
- Would such an algorithm have a lower running time than our binary search?
  - Would it have a lower complexity?
Clicker Question

What is the complexity of hiding the image in project 3, where the image is $n \times n$ pixels?

A. $O(1)$
B. $O(n)$
C. $O(n^2)$
D. $O(n^3)$
Not all algorithms are the same complexity

- There is a group of algorithms called *sorting algorithms* that place things (numbers, names) in a sequence.

- Some of the sorting algorithms have complexity around $O(n^2)$
  - If the list has 100 elements, it takes about 10,000 steps to sort them.

- However, others have complexity $O(n \log n)$
  - The same list of 100 elements would take only 460 steps.

- Think about the difference if you’re sorting your 1,000,000 customers…
We want to choose the algorithm with the best complexity.

- We want an algorithm which will be
  - Fast: a “lower” complexity means an algorithm will perform better on large input
  - Stable: gives accurate answers
  - Space efficient
  - Easy to implement and maintain
Homework

- Study for the midterm
Announcements

- Midterm on Monday
Lists, Strings, (Tuples, Dictionaries)

- All use bracket notation to access elements [ ]
- Lists, Tuples, and Strings use an index to access an element
  - We consider such structures ordered (as opposed to sets)
Creating Structures

- Lists – use the [ ] notation
  - List = [1, 2, 3, 4, 5, “foo”]

- Strings use the single or double quotes, or triple repeated quotes
  - String = “this is my string”
Immutable Structures

- Strings are considered immutable
  - What does this mean in practice?
  - We cannot assign new values to the indexed elements in strings

- Errors for strings:
Structures can contain other structures

- Lists can contain elements which themselves are Lists
- This is used in encodings:
  - Matrix encoding
  - Tree encoding
Specialized Structures built from structures containing structures

- Matrices
  - Represented as a list of lists
  - The internal lists are either *rows* or *columns*

- Trees
  - Represented as an arbitrary nesting of lists
  - The structure of the elements represents the *branching* of the tree
Matrices

- Review matrix multiplication
- Review how to populate a matrix
  - Go through the pre lab examples
- Review how to create a matrix
  - Python short hand
Traversing a Matrix

- Is B encoded column by column or row by row?
  - We do not know
  - …. But what if I told you this loop prints the matrix row by row?

\[
B = \begin{bmatrix}
1, 0, 0, \\
0.5, 3, 4, \\
-1, -3, 6, \\
0, 0, 0,
\end{bmatrix}
\]

```python
for j in range(3):
    for i in range(4):
        print (B[i][j])
```
How do we find out how the matrix is encoded?

- Step one: figure out the order in which the values are printed
  - 1, 0.5, -1, 0, 0, 3, -3, 0, 0, 4, 6, 0

- Step two: compare this to the matrix
  - \[ B = \begin{bmatrix} 1, & 0, & 0 \\ 0.5, & 3, & 4 \\ -1, & -3, & 6 \\ 0, & 0, & 0 \end{bmatrix} \]

- Step three: deduce the encoding by comparing the order to the matrix
  - The matrix is encoded column by column!
Applying the same intuition to matrices

- Lets traverse the matrix the other way!

```python
B = [[1, 0, 0], [0.5, 3, 4], [-1, -3, 6], [0, 0, 0]]
for j in range(3):
    for i in range(4):
        print(B[i][j])
for i in range(4):
    for j in range(3):
        print(B[i][j])
```
Trees

- Know how to select elements from a tree
- Know how to construct a tree using lists
- Distinguish internal nodes and leaves
Given the picture can you generate the python list?

```
Root
  Leaf2
  Leaf0 Leaf1
  Leaf3
  Leaf4
  Leaf5
```
Indices provide us a way to “traverse” the tree
Modules

- **urllib**
  - Allows us to get data directly from webpages

- **os**
  - Allows us to manage files and interact with the operating system
File I/O

- What is the difference between the various modes?
  - We saw in class “w” “r”

What is the difference between read, readline, and readlines?
Methods on Files

- object.method() syntax: this time files are our object
  - Example: file = open("myfile", "w")

- file.read() -- reads the file as one string

- file.readlines() -- reads the file as a list of strings

- file.write() -- allows you to write to a file

- file.close() -- closes a file
Strings and Parsing

- What are the most important operations?
  - find
  - rfind
  - split
  - strip
  - Slicing
String.find

- `string.find(sub)` – returns the lowest index where the substring `sub` is found or -1

- `string.find(sub, start)` – same as above, except using the slice `string[start:]`

- `string.find(sub, start, end)` – same as above, except using the slice `string[start:end]`
String.rfind

- `string.rfind(sub)` – returns the highest index where the substring `sub` is found or -1

- `string.rfind(sub, start)` – same as above, except using the slice `string[start:]`

- `string.rfind(sub, start, end)` – same as above, except using the slice `string[start:end]`
String

String.split(delimiter) breaks the string String into parts, separated by the delimiter

- print ("a b c d".split(" "))
  Would print: ['a', 'b', 'c', 'd']
Concrete Example

```python
foo = "there their they're"

elem = foo.split(" ")

for i in elem:
    print(i.split("e"))

['th', 'r', '']
['th', 'ir']
['th', "y'r", ']'
```

String.strip

- “hello helpful handy hammer”.strip(‘her’) results in “llo helpful handy hamm”
Manipulating Strings

- How might we reverse a string?

- We used the same technique for the problems
  - Build up a new string piece by piece
Example: Reversing Strings

def reverse(str):
    output = ""
    for i in range(0, len(str)):
        output = output + str[len(str)-i-1]
    print(output)
Useful things to know

- range function
- ord and chr functions
- int and type functions