CS 177
Big-O Notation and Algorithms
Week 11
Announcements

- Project 4 due Wed., Nov 7
- Project 5 coming soon; also a team project
  - This will include the brief essay question for the team course of Science
- New course CS 290 00, Spring 2013
  - Contemporary Issues in a Digital World
    - Instructor: Robb Cutler.
    - Syllabus at http://cs4edu.cs.purdue.edu/cidw
- No class Thursday
ANY QUESTIONS?
There is always more than one way to solve a problem.

- Two different ways to walk from A to B.
  - Walk the hypotenuse
  - Walk the sides from A to C to B

- However, some solutions are better than others.

- But how do we compare them?
  - In this case, it’s obvious to walk the hypotenuse.
We Implement Algorithms

- *Algorithms* are descriptions of computations for solving a problem:

Problem:
- Find the index of a specific item (for example, the number 3) in a list of integers
  1. Scan the list
  2. For each element, check if it is the specific item

Programs are executable interpretations of algorithms:
```python
count = 0
for x in L:
    if x == 3: print count
    count += 1
```
How we can compare two algorithms to determine which one is better (for large inputs)?

We use the *order of magnitude* of the running time of an algorithm with large input size.

We denote this order of magnitude by the so-called Big-O notation.

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*.
More on Big-O notation

- Big-O notation ignores constants because they don’t affect the growth much
- The growth term is the term that will cause the most rapid growth
  - Example:
    - $f(n) = 2n + 20$
    - $2n$ is the growth term
    - But complexity is $O(n)$ because constants are ignored
Comparison of complexities:

- $O(1)$ – 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
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>1. $2n^3 + 1$</td>
<td>$2n^3$</td>
<td>$O(n^3)$</td>
</tr>
<tr>
<td>2. $10n + 3$</td>
<td>$10n$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>3. $4n^2 + 7n$</td>
<td>$4n^2$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>4. $10n^{10} + 2^n + 3$</td>
<td>$2^n$</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td>5. $23n + 4$</td>
<td>$23n$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

NOTE: 2 and 5 have the same complexity and $2^n$ has a larger growth rate than $n^{10}$ (just do the math....)
Exercise 1:
find the smallest element in a list of integers

- Three cases to consider:
  1. The list is **NOT** sorted:
     - [1, 5, 89, 3 ,4,121,15]
  2. The list is **sorted in ascending order**:
     - [1, 3, 4, 5, 15, 89, 121]
  3. The list is **sorted in descending order**:
     - [121, 89, 15, 5, 4, 3, 2, 1]
Exercise 1: unsorted list

```python
#Find the smallest element in a list
def algorithm1(listx):
    min = listx[0]
    for k in listx:
        if k < min: min = k
    print("Smallest is", min)
```

- The only thing we know is that the list is unsorted.
  - hence we need to go through the whole list

- What is the complexity?
  - \( O(n) \) where \( n \) is the length of the list
Exercise 1:
list sorted in ascending order

```python
#Find the smallest element in a list
def algorithm2(listx):
    min = listx[0]
    print("Smallest is", min)
```

- In this case we took advantage of the fact that the list is sorted in ascending order. *But this algorithm would work in this case only!*

- What is the complexity of this algorithm?
  - O(1)
Exercise 1:
list sorted in descending order

```python
#Find the smallest element in a list
def algorithm1(listx):
    min = listx[-1]
    print("Smallest is", min)
```

- What is the complexity?
- \( O(1) \) where \( n \) is the length of the list
- In this case we took advantage of the fact that the list is sorted in descending order. \textit{But this algorithm would work in this case only!}
Summing up…

- If our list contains integers, then there are only three cases to consider:
  A. The list is not sorted
  B. The list is sorted in ascending order
  C. The list is sorted in descending order

- What is the best case?
- What is the worst case?
We should design our algorithm so that it can work in all the cases we saw before. Hence:

```python
#Find the smallest element in a list
def algorithm1(listx):
    min = listx[0]
    for k in listx:
        if k < min: min = k
    print("Smallest is", min)
```

The complexity of this algorithm is its running time in the worst case(s): $O(n)$
Exercise 2:
Search for a specific element is in a list

- Identify the best and worst cases:

  - Best case?
    - O(1)
  - Worst case?
    - O(n)
Exercise 2
Search for a specific element is in a list

#See if a specific element is in a list
def algorithm3(listx, element):
    for k in range(len(listx)):
        if listx[k] == element:
            print(element, “has been found”)
            return
    print(element, “was not found”)

• What is the complexity for the worst case?
  \[ O(n) \text{ where } n \text{ is } \text{len(listx)} \]

• What is the complexity for the best case?
  \[ O(1) \]
ANY QUESTIONS?