Last time... **Testing, debugging, exceptions**

Exceptions

```python
try:
    ....
    ....
except:
    ....
finally:
    ....
```

Debugging
Lecture Overview

• How to develop a program
• Algorithmic Complexity
Lecture Overview

• How to develop a program
• Algorithmic Complexity
Program development methodology: Algorithm first, then Python

1. Define the problem
2. Decide upon an algorithm
3. Translate it into code

Try to do these steps in order
Program development methodology: Algorithm first, then Python

1. **Define the problem**
   A. Write an Natural Language description of the input and output for the whole program. (Do not give details about *how you will compute* the output.)
   B. Create test cases for the whole program
      - Input *and* expected output

2. Decide upon an algorithm

3. Translate it into code

Try to do these steps in order
Program development methodology: Algorithm first, then Python

1. Define the problem

2. **Decide upon an algorithm**
   A. Implement it in Algorithmic way (e.g. in English)
      • Write the recipe OR step-by-step instructions
   B. Test it using paper and pencil
      • Use small but not trivial test cases
      • Play computer, animating the algorithm
      • Be introspective
         • Notice what you really do
         • May be more or less than what you wrote down
         • Make the algorithm more precise

3. Translate it into code

Try to do these steps in order
Program development methodology: Algorithm first, then Python

1. Define the problem
2. Decide upon an algorithm

3. **Translate it into code**
   A. Implement it in Python
      - Decompose it into logical units (functions)
      - For each function:
        - Name it (important and difficult!)
        - Write its documentation string (its specification)
        - Write tests
        - Write its code
        - Test the function
   B. Test the whole program

Try to do these steps in order
Program development methodology: Algorithm first, then Python

1. Define the problem
2. Decide upon an algorithm
3. Translate it into code

Try to do these steps in order
- It’s OK (even common) to back up to a previous step when you notice a problem
- You are incrementally learning about the problem, the algorithm, and the code
- “Iterative development”
Waterfall Development Strategy

• Before the iterative model, we had the waterfall strategy.
• Each step handled once.
• The model had a limited capability and received too many criticisms.
• Better than nothing!!
• Do not dive in to code!!
• Please!!

* From wikipedia waterfall development model
Iterative Development Strategy

- Software development is a living process.
- Pure waterfall model wasn’t enough.
- Iterative development strategy suits best to our needs (for now).

*From Wikipedia Iterative development model*
Iterative Development Strategy -2-

* From wikipedia Iterative development model
The *Wishful Thinking* approach to implementing a function

• If you are not sure how to implement one part of your function, define a **helper function** that does that task
  • “I wish I knew how to do task X”
  • Give it a name and assume that it works
  • Go ahead and complete the implementation of your function, *using* the helper function (and assuming it works)
  • Later, implement the **helper function**
  • The helper function should have a **simpler/smaller task**
The *Wishful Thinking* approach to implementing a function

- Can you test the original function?
  - Yes, by using a *stub* for the *helper function*
  - Often a lookup table: works for only 5 inputs, crashes otherwise, or maybe just returns the same value every time
Why functions?

There are several reasons:

• Creating a new function gives you an opportunity to name a group of statements, which makes your program easier to read and debug.

• Functions can make a program smaller by eliminating repetitive code. Later, if you make a change, you only have to make it in one place.

• Dividing a long program into functions allows you to debug the parts one at a time and then assemble them into a working whole.

• Well-designed functions are often useful for many programs. Once you write and debug one, you can reuse it.
Lecture Overview

• How to develop a program
• Algorithmic Complexity
Lecture Overview

• How to develop a program
• Algorithmic Complexity

Slides based on material prepared by Ruth Anderson, Michael Ernst and Bill Howe in the course CSE 140
University of Washington
Measuring complexity

• Goals in designing programs
  1. It returns the correct answer on all legal inputs
  2. It performs the computation efficiently

• Typically (1) is most important, but sometimes (2) is also critical, e.g., programs for collision detection, avionic systems, drive assistance etc.

• Even when (1) is most important, it is valuable to understand and optimize (2)
Computational complexity

• How much time will it take a program to run?
• How much memory will it need to run?

• Need to balance minimizing computational complexity with conceptual complexity
  • Keep code simple and easy to understand, but where possible optimize performance
How do we measure complexity?

• Given a function, would like to answer: “How long will this take to run?”
• Could just run on some input and time it.
• Problem is that this depends on:
  1. Speed of computer
  2. Specifics of Programming Language implementation
  3. Value of input
• Avoid (1) and (2) by measuring time in terms of number of basic steps executed
Measuring basic steps

• Use a **random access machine (RAM)** as model of computation
  • Steps are executed sequentially
  • Step is an operation that takes constant time
    • Assignment
    • Comparison
    • Arithmetic operation
    • Accessing object in memory

• For point (3), measure time in terms of size of input
But complexity might depend on value of input?

```python
def linearSearch(L, x):
    for e in L:
        if e==x:
            return True
    return False
```

• If x happens to be near front of L, then returns True almost immediately
• If x not in L, then code will have to examine all elements of L
• Need a general way of measuring
Cases for measuring complexity

- **Best case:** minimum running time over all possible inputs of a given size
  - For linearSearch – constant, i.e. independent of size of inputs
- **Worst case:** maximum running time over all possible inputs of a given size
  - For linearSearch – linear in size of list
- **Average (or expected) case:** average running time over all possible inputs of a given size
- We will focus on worst case – a kind of **upper bound** on running time
Example

def fact(n):
    answer = 1
    while n > 1:
        answer *= n
        n -= 1
    return answer

• Number of steps
  • 1 (for assignment)
  • 5*n (1 for test, plus 2 for first assignment, plus 2 for second assignment in while; repeated n <mes through while)
  • 1 (for return)

• 5*n+2steps
• But as n gets large, 2 is irrelevant, so basically 5*n steps
Example

• What about the multiplicative constant (5 in this case)?
• We argue that in general, multiplicative constants are not relevant when comparing algorithms
Example

```python
def sqrtExhaust(x, eps):
    step = eps**2
    ans = 0.0
    while abs(ans**2 - x) >= eps and ans <= max(x, 1):
        ans += step
    return ans
```

• If we call this on 100 and 0.0001, will take one billion iterations of the loop
  • Have roughly 8 steps within each iteration
Example

def sqrtBi(x, eps):
    low = 0.0
    high = max(1, x)
    ans = (high + low)/2.0
    while abs(ans**2 - x) >= eps:
        if ans**2 < x:
            low = ans
        else:
            high = ans
        ans = (high + low)/2.0
    return ans

• If we call this on 100 and 0.0001, will take thirty iterations of the loop
  • Have roughly 10 steps within each iteration
• 1 billion or 8 billion versus 30 or 300 – it is size of problem that matters
Measuring complexity

• Given this difference in iterations through loop, multiplicative factor (number of steps within loop) probably irrelevant
• Thus, we will focus on measuring the complexity as a function of input size
  • Will focus on the largest factor in this expression
  • Will be mostly concerned with the worst case scenario
Asymptotic notation

• Need a formal way to talk about relationship between running time and size of inputs
• Mostly interested in what happens as size of inputs gets very large, i.e. approaches infinity
Example

def f(x):
    for i in range(1000):
        ans = i
    for i in range(x):
        ans += 1
    for i in range(x):
        for j in range(x):
            ans += 1

Complexity is $1000 + 2x + 2x^2$, if each line takes one step
Example

• $1000+2x+2x^2$

• If $x$ is small, constant term dominates
  • E.g., $x = 10$ then 1000 of 1220 steps are in first loop

• If $x$ is large, quadratic term dominates
  • E.g. $x = 1,000,000$, then first loop takes 0.000000005% of time, second loop takes 0.0001% of time (out of 2,000,002,001,000 steps)!
Example

• So really only need to consider the nested loops (quadratic component)

• Does it matter that this part takes $2x^2$ steps, as opposed to say $x^2$ steps?
  • For our example, if our computer executes 100 million steps per second, difference is 5.5 hours versus 2.25 hours
  • On the other hand if we can find a linear algorithm, this would run in a fraction of a second
  • So multiplicative factors probably not crucial, but order of growth is crucial
Rules of thumb for complexity

• Asymptotic complexity
  • Describe running time in terms of number of basic steps
  • If running time is sum of multiple terms, keep one with the largest growth rate
  • If remaining term is a product, drop any multiplicative constants

• Use “Big O” notation (aka Omicron)
  • Gives an upper bound on asymptotic growth of a function
Complexity classes

• $O(1)$ denotes constant running time
• $O(\log n)$ denotes logarithmic running time
• $O(n)$ denotes linear running time
• $O(n \log n)$ denotes log-linear running time
• $O(n^c)$ denotes polynomial running time ($c$ is a constant)
• $O(c^n)$ denotes exponential running time ($c$ is a constant being raised to a power based on size of input)
Constant complexity

• Complexity independent of inputs
• Very few interesting algorithms in this class, but can often have pieces that fit this class
• Can have loops or recursive calls, but number of iterations or calls independent of size of input
Logarithmic complexity

• Complexity grows as log of size of one of its inputs
• Example:
  • Bisection search
  • Binary search of a list
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        elif item < alist[midpoint]:
            last = midpoint-1
        else:
            first = midpoint+1

    return found
Logarithmic complexity

```python
def binarySearch(alist, item):
    first = 0
    last = len(alist)-1
    found = False

    while first<=last and not found:
        midpoint = (first + last)//2
        if alist[midpoint] == item:
            found = True
        elif item < alist[midpoint]:
            last = midpoint-1
        else:
            first = midpoint+1

    return found
```

- Only have to look at loop as no function calls
- Within while loop constant number of steps
- How many times through loop?
  - How many times can one divide indexes to find midpoint?
  - $O(\log(\text{len(alist)}))$
Linear complexity

- Searching a list in order to see if an element is present
- Add characters of a string, assumed to be composed of decimal digits

```python
def addDigits(s):
    val = 0
    for c in s:
        val += int(c)
    return val
```

- $O(len(s))$
Log-linear complexity

• Many practical algorithms are log-linear
• Very commonly used log-linear algorithm is **merge sort**
• Will return to this
Polynomial complexity

• Most common polynomial algorithms are quadratic, i.e., complexity grows with square of size of input
• Commonly occurs when we have nested loops or recursive function calls
Quadratic complexity

def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True
Quadratic complexity

def isSubset(L1, L2):
    for e1 in L1:
        matched = False
        for e2 in L2:
            if e1 == e2:
                matched = True
                break
        if not matched:
            return False
    return True

• Outer loop executed len(L1) times
• Each iteration will execute inner loop up to len(L2) times
• $O(len(L1) * len(L2))$
• Worst case when L1 and L2 same length, none of elements of L1 in L2
• $O(len(L1)^2)$
Quadratic complexity

Find intersection of two lists, return a list with each element appearing only once

```python
def intersect(L1, L2):
    tmp = []
    for e1 in L1:
        for e2 in L2:
            if e1 == e2:
                tmp.append(e1)
    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)
    return res
```
Quadratic complexity

```python
def intersect(L1, L2):
    tmp = []
    for el1 in L1:
        for el2 in L2:
            if el1 == el2:
                tmp.append(el1)

    res = []
    for e in tmp:
        if not(e in res):
            res.append(e)

    return res
```

- First nested loop takes \( \text{len}(L1) \times \text{len}(L2) \) steps
- Second loop takes at most \( \text{len}(L1) \) steps
- Latter term overwhelmed by former term
- \( \mathcal{O}(\text{len}(L1) \times \text{len}(L2)) \)
Exponential complexity

• Recursive functions where more than one recursive call for each size of problem
  • Towers of Hanoi
  • Fibonacci series
• Many important problems are inherently exponential
  • Unfortunate, as cost can be high
  • Will lead us to consider approximate solutions more quickly
def fib(N):
    if N == 1 or N == 0:
        return N
    else:
        return fib(N-1) + fib(N-2)
Exponential Complexity

```python
def fib(N):
    if N == 1 or N == 0:
        return N
    else:
        return fib(N-1) + fib(N-2)
```

• Assuming return statement is constant time
• Recall the recursive tree.
• Complexity of this function is $O(\sim 2^n)$
Factorial Complexity

• The travelling salesperson problem.

• A salesperson has to visit n towns. Each pair of towns is joined by a route of a given length. Find the shortest possible route that visits all the towns and returns to the starting point.
  
  • 1) Consider city 1 as the starting and ending point.
  • 2) Generate all (n-1)! Permutations of cities.
  • 3) Calculate cost of every permutation and keep track of minimum cost permutation.
  • 4) Return the permutation with minimum cost.

![Image of a triangle with cities and edges marked with distances]
Complexity classes

• $O(1)$ denotes constant running time.
• $O(\log n)$ denotes logarithmic running time
• $O(n)$ denotes linear running time
• $O(n \log n)$ denotes log-linear running time
• $O(n^c)$ denotes polynomial running time (c is a constant)
• $O(c^n)$ denotes exponential running time (c is a constant being raised to a power based on size of input)
• $O(n!)$ denotes factorial running time
Comparing complexities

• So does it really matter if our code is of a particular class of complexity?
• Depends on size of problem, but for large scale problems, complexity of worst case makes a difference
Comparing complexities - example

• There are alternative approaches with differing algorithm complexities for doing *sth.* on a list of *n* elements.
• Now you want to compare them. Assume that computer makes three billion calculations per second. Let's look for the running time of the algorithms.

<table>
<thead>
<tr>
<th>Complexity</th>
<th>n=10</th>
<th>n=1000</th>
<th>n=10^5</th>
<th>n=10^10</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(logn)</td>
<td>&lt; 1msec</td>
<td>&lt; 1msec</td>
<td>&lt; 1msec</td>
<td>&lt; 1msec</td>
</tr>
<tr>
<td>O(n)</td>
<td>&lt; 1msec</td>
<td>&lt; 1msec</td>
<td>&lt; 1msec</td>
<td>&lt; 1 min</td>
</tr>
<tr>
<td>O(nlogn)</td>
<td>&lt; 1msec</td>
<td>&lt; 1msec</td>
<td>&lt; 1 sec</td>
<td>&lt; 2 min</td>
</tr>
<tr>
<td>O(n^2)</td>
<td>&lt; 1msec</td>
<td>&lt; 1msec</td>
<td>&lt; 1 min</td>
<td>~1000 year</td>
</tr>
<tr>
<td>O(2^n)</td>
<td>&lt; 1 sec</td>
<td>&lt;1000 year</td>
<td>&lt;1000 year</td>
<td>&lt;1000 year</td>
</tr>
<tr>
<td>O(n!)</td>
<td>&lt; 1 sec</td>
<td>&lt;1000 year</td>
<td>&gt;1000 year</td>
<td>&gt;1000 year</td>
</tr>
</tbody>
</table>
Comparing the Complexities

Big-O Complexity Chart

- Operations
- Elements

- $O(n!)$
- $O(2^n)$
- $O(n^2)$
- $O(n \log n)$
- $O(n)$
- $O(\log n), O(1)$

Categories:
- Horrible
- Bad
- Fair
- Good
- Excellent
Constant versus Logarithmic
Observations

- A logarithmic algorithm is often almost as good as a constant time algorithm
- Logarithmic costs grow very slowly
Logarithmic versus Linear

![Graph comparing logarithmic and linear functions](image-url)
Observations

• Logarithmic clearly better for large scale problems than linear
• Does not imply linear is bad, however
Linear versus Log-linear

![Linear vs. Log-linear graph]

- Linear
- Log-linear
Observations

• While $\log(n)$ may grow slowly, when multiplied by a linear factor, growth is much more rapid than pure linear.
• $O(n \log n)$ algorithms are still very valuable.
Log-linear versus Quadratic

![Log-linear vs. Quadratic](image-url)
Observations

• Quadratic is often a problem, however.
• Some problems inherently quadratic but if possible always better to look for more efficient solutions
Quadratic versus Exponential

- Exponential algorithms very expensive
  - Right plot is on a log scale, since left plot almost invisible given how rapidly exponential grows
- Exponential generally not of use except for small problems
Warning

• Execution time and the algorithm complexity are different paradigms.
• Running time may differ even if two algorithms have the same algorithm complexity. (Even when their purpose are same)

```python
def factIT(n):
    answer = 1
    while n > 0:
        answer *= n
        n -= 1
    return answer

def factREC(n):
    if n == 0:
        return 1
    else:
        return n*factREC(n-1)
```

They have same complexity O(n). But their execution times are different.
Tips

- We know that, $O(2^n)$ algorithm complexity is bad. But, if we sure that $n$ won’t be up too high, it won’t be matter.
- When we calculate the big-O, we did not care about constant factors.
  - $5n + 37 \rightarrow O(n)$
- But, sometimes improving the constants does matter. (e.g. in game development; actually, everytime)
  - $5n+37 \rightarrow 5n+10$ (not worthy, but better than nothing)
  - $5n+37 \rightarrow 3n+12$ (better)