EdX 6.00x Notes

# Lecture 9:

* Measuring complexity
  + Goals in designing programs
    - 1. It returns the correct answer on all legal issues
    - 2. It performs the computation efficiently
  + Typically (1) is important, but sometimes (2) is also critical, e.g., programs for collision detection
  + 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 computation 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 ask how long does 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 Python 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
* 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
* Multiplicative constants
  + We argue in general, multiplicative constants are not relevant when comparing algorithms
  + It is the size of the problem that matters
* Measuring complexity
  + Given this different 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 scemario
* Asymptotic notation
  + Need a formal way to talk about relationship between the running time and size of input
  + Mostly interested in what happens as inputs gets very large, i.e. approaches infinity
* Example:
  + 1000 + 2x + 2x2
  + 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
  + So really only need to consider the quadratic component
  + Does it matter that this part takes 2x2 steps, as opposed to say x2 steps?
    - 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(nc) denotes polynomial running time (c is a constant)
  + O(cn) 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 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
* 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
  + Complexity can depend on number of recursive calls
* Log-linear complexity
  + Many practical algorithms are log-linear
  + Very commonly used log-linear algorithm is merge sort
* Polynomial complexity
  + Most common polynomial algorithms are quadratic, i.e. complexity grows with square size of input
  + Commonly occurs when we have nested loops or recursive function calls
* Exponential complexity
  + Recursive functions where more than one recursive call for each size of problem
    - Towers of Hanoi
  + Many important problems are inherently exponential
    - Unfortunate, as cost can be high
    - Will lead us to consider approximate solutions more quickly
* 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
* Observations
  + A logarithmic algorithm is often almost as good as a constant time algorithm
  + Logarithmic costs grow very slowly
  + Logarithmic clearly better for large scale problems than linear
  + While log(n) may grow slowly, when multiplied by a linear factor, growth is much more rapid than pure linear
  + Quadratic is often a problem, however some problems inherently quadratic but if possible always better to look for more efficient solutions
  + Exponential algorithms are very expensive and generally not of use except for small problems