**MIT – 6.00.1x: Introduction to Computer Science and Programming**

**WEEK 5**

**Lecture 9: Efficiency and Orders of Growth**

Part 1: Measuring Complexity

* 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 the most important, but sometimes (2) is also critical, e.g. programs for collision detection need to be able to do computation quickly and make decisions quickly.
   * Even when (1) is most important, it is valuable to understand and optimize (2).

* Computational Complexity
  + How much time will a program take 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, we would like to answer, “How long will this take to run?”
  + We could just run the program on some input and time it. However, the problem is that this depends on:

1. Speed of computer
2. Specifics of Python specification
3. Value of input
   * Avoid (1) and (2) by measuring time in terms of number of basic steps or operations 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 amount of time, including:
* Assignment
* Comparison
* Arithmetic operation
* Accessing object in memory
  + - For point (3), measure time in terms of size of input.
* But Would Complexity Depend on the Actual Size of the Input?
  + Let’s define a function called linear search that searches through a list of numbers.

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 it returns True almost immediately.
  + If x is not in L, then code will have to examine all elements in L.
  + Need a general way of measuring this.
* 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 the list
  + **Average (or Expected) Case:** Average running time over all possible inputs of a given size.
  + However, we will focus on the worst case – a kind of **upper bound** on running time.
* Example
  + Let’s look at the case of the factorial function to see how what the computational complexity of the recursive algorithm is.

def factorial(n):

answer = 1

while n > 1:

answer \*= n

n -= 1

return answer

* + Number of steps
    - (for assignment)
    - (1 for test, plus 2 for first assignment, plus 2 for second assignment in while; repeated times through while loop)
    - (for return)
  + steps
  + But as gets very large, 2 is irrelevant, so basically steps.
  + What about the multiplicative constant ( in this case)?
  + We argue that in general, multiplicative constants are not relevant when comparing algorithms.
* Example
  + Let’s look at an example of trying to find the square root exhaustively through small steps.

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, it will take 1 billion iterations of the loop.
    - Have roughly 8 steps within each iteration.
* Example
  + Let’s look at an example of trying to find the square root through bisection search.

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, it will take approximately 30 iterations of the loop.
    - Have roughly 10 steps within each iteration.
  + 1 billion or 8 billion steps versus 30 or 300 steps – it is the size of the problem that matters.
* Measuring Complexity
  + Given this difference in iterations through the loop, the multiplicative factor (number of steps within the loop) is 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.

Part 2: Asymptotic Notation

* Asymptotic Notation
  + Need a formal way of talking about the relationship between the running time and the size of the inputs.
  + We are mostly interested in what happens as the size of the inputs gets very large, i.e. approaches infinity.
* Example
  + We will look at an example where one variable is being changed continuously through the 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

return ans

* + Here, ignoring all loop bindings inside for loops, the complexity is. However, considering all the loop bindings and variable assignments in for loops, the actual complexity of the function is.
    - steps for the first for loop, where is looping over integers and has steps per loop – setting to a number and then setting
    - steps are involved in the second for loop – variable assignment for and takes up steps in each iteration
    - steps are taken in the last loop. The inner for loop goes through iterations of the loop, and the last time it goes through the loop, it makes final check at the top, and then moves on to the next number in the outer for loop, which takes times to loop.
  + However, ignoring the complexities of for loop variable assignment, the program takes steps to run, for our purposes.
  + If is small, then the constant term dominates.
    - E.g. if then steps are in the first loop alone.
  + If is large, then the quadratic term dominates.
    - E.g. if then the first loop takes of run-time, the second loop takes of run-time (out of steps), and the third loop takes the remaining run-time!
  + So really, we only need to consider the nested loops (which compose the quadratic component of our expression).
  + Does it matter that this part takes steps, as opposed to steps?
    - For our example, if our computer executes million steps per second, the difference in run-time is hours versus hours.
    - On the other hand, if we can find a linear algorithm to perform the same computation, this would run in a fraction of a second.
    - So multiplicative factors are probably not as crucial, but orders of growth are extremely crucial to run-time.
* Rules of Thumb for Complexity
  + Asymptotic complexity
    - Describe running time in terms of number of basic steps.
    - If running time is the sum of multiple terms, keep the term with the largest order of growth.
    - If the 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.

Part 3: Complexity Classes

* Classes of Complexity
  + denotes constant running time
  + denotes logarithmic running time
  + denotes linear running time
  + denotes log-linear running time
  + denotes polynomial running time ( is a constant)
  + denotes exponential running time ( is a constant)
* Constant Complexity
  + Complexity is independent of inputs.
  + Very few interesting algorithms in this class, but code can often have pieces that fit this class.
  + Can have loops or recursive calls, but number of iterations or recursive calls are independent of the size of the input.
* Logarithmic Complexity
  + Complexity grows at the rate of the logarithm of the size of one of the inputs, thus it does still grow quite slowly – very efficient.
  + Examples:
    - Bisection search
    - Binary search of a list
* Logarithmic Complexity – Example
  + Here’s a piece of code to show how one might go about reasoning that an algorithm is in fact logarithmic.

def intToStr(i):

digits = '0123456789'

if i == 0:

return '0'

result = ''

while i > 0:

result += digits[i % 10]

i /= 10

return result

* + We only have to look at the while loop because the function has no other function calls. Also, it is the structure of highest complexity – this will determine the run-time of our program.
  + Within the while loop, there are a constant number of steps (i.e. 6 steps every time). But how many times do you go through the loop based on the value of the input?
    - How many times can one divide by? This function is of complexity.
* Linear Complexity
  + Searching a list or tuple or dictionary in order to see if an element is present.
  + Add the characters of a string, assumed to be composed of base 10 decimal digits.
* Linear Complexity – Example
  + Let’s use the example mentioned before to demonstrate how the algorithm to add decimal digits in a string is of linear complexity.

def addDigits(s):

total = 0

for digit in s:

total += int(digit)

return total

* + The complexity of this problem depends on the length of the string. This problem is of complexity.
* Linear Complexity – Another Example
  + Complexity can also depend on the number of recursive calls.

def factorial(n):

if n == 1:

return 1

return n \* factorial(n - 1)

* + Number of recursive calls?
    - , then, etc. until we get to
    - The complexity of each call is constant, while the number of calls varies linearly with the size of the input. Thus, the complexity of the algorithm is.
* Log-Linear Complexity
  + Many practical algorithms are of log-linear complexity. One very commonly used log-linear algorithm is called merge sort.
* Polynomial Complexity
  + Most common polynomial algorithms are quadratic, i.e. their complexity grows with the square of the size of the input.
  + Commonly occurs when we have nested loops or particular kinds of recursive function calls.
* Polynomial Complexity – Example
  + Here is a very common algorithm of quadratic complexity.

def isSubset(L1, L2):

for e1 in L1:

matched = False

for e2 in L2:

if e1 == e2:

matched = True

break

if matched == False:

return False

return True

* + Outer loop executed times. Each iteration of the outer loop will execute the inner loop up to times.
  + Thus, the complexity of the problem falls under.
  + However, the worst case scenario is when and are of the same length, and none of the elements of are in. Thus, the total complexity of this problem is in.
* Polynomial Complexity – Example
  + Here is another algorithm of quadratic complexity.

def intersect(L1, L2):

temp = []

for e1 in L1:

for e2 in L2:

if e1 == e2:

temp.append(e1)

result = []

for e in temp:

if e not in result:

result.append(e)

return result

* + First nested loop takes steps to execute completely.
  + Second loop takes at most steps. The latter loop is overwhelmed by the former loop in terms of complexity.
  + Thus, the overall complexity of the algorithm is quadratic, it falls in.
* Exponential Complexity
  + Recursive functions where there is more than one recursive call for each size of the problem.
    - Towers of Hanoi
  + Many important problems are inherently exponential.
    - Unfortunately, the cost of computing exponential algorithms is usually high.
    - This will lead us to consider approximate solutions to the same problem, as they can compute the solution more quickly.
* Exponential Complexity – Example
  + Let’s see an exponential algorithm to generate a list of subsets of a given set of elements.

def genSubsets(L):

result = []

if len(L) == 0:

return [[]]

smaller = genSubsets(L[:-1])

extra = L[-1:]

new = []

for small in smaller:

new.append(small + extra)

return smaller + new

* + Assuming append takes the same amount of time as other elementary operations, we can say the following.
  + The total time taken to execute the program includes the time to solve the smaller problem, plus the time needed to make a copy of all elements in the smaller problem.
  + But it is important to think about the size of the smaller problem. We know that for a set of size, there are cases or different possible subsets that can be created, assuming all the elements are unique.
  + So, to solve this problem, we need steps.
  + Thus, the math tells us that this is actually a problem in the complexity.

Part 4: Comparing Complexity Classes

* Comparing Complexities
  + So does it really matter if our code is of a particular class of complexity versus another class of complexity? How much of a difference in computation time could it make?
  + All of this depends on the size of the problem, but for large-scale problems, complexity of the worst case could make a significant difference.
* Observations – Logarithmic vs. Constant
  + A logarithmic algorithm is often almost as good as a constant time algorithm. This is because the logarithmic function grows very slowly, and thus the cost of computing a logarithmic algorithm grows very slowly.
* Observations – Linear vs. Logarithmic
  + Logarithmic algorithms are clearly better for large scale problems than linear algorithms, because they grow much more slowly.
  + However, this does not imply that linear algorithms are bad, because they can also be very useful.
* Observations – Log-Linear vs. Linear
  + While may grow slowly, it grows much more rapidly than a linear function when multiplied by a linear factor.
  + However, even though they increase in time rapidly, the algorithms in are still very valuable.
* Observations – Quadratic vs. Log-Linear
  + However, quadratic algorithms can often be a problem, because they grow quite rapidly in terms of time – they lose a lot of efficiency.
  + Some problems are inherently quadratic, but it is usually possible to look for more efficient solutions (e.g. you don’t always need nested loops of quadratic complexity in a function).
  + A solution of log-linear complexity is far more efficient and is much better to work with in terms of the worst-case scenario.
* Observations – Exponential vs. Quadratic
  + Exponential algorithms are very expensive. On a graph of quadratic versus exponential, as you zoom out, you can’t even see the graphs because the exponential graph is almost vertical, while the quadratic graph is almost horizontal in comparison.
  + Exponential algorithms are not generally used unless for small problems (e.g. Towers of Hanoi).

**Lecture 10: Memory and Search**

Part 1: Search Algorithms

* Algorithms and Data Structures
  + How do you find efficient algorithms?
    - Hard to invent new ones.
    - Easier to reduce a problem to known solutions.
* We want to understand the inherent complexity of the problem.
* We want to think about how to break the problem into smaller sub-problems.
* We want to relate sub-problems to other problems for which there already exist efficient algorithms.
* Search Algorithms
  + Search algorithm – a method for finding an item or group of items with specific properties within a collection of items.
  + The collection of items that I am looking in to find my item is called the *search space*.
  + We have seen examples of this – for example, finding the square root can be viewed as a search problem with many solutions.
    - Exhaustive enumeration
    - Bisection search
    - Newton-Raphson
* Linear Search and Indirection
  + Below is a simple search method to find an element within a list.

def search(L, e):

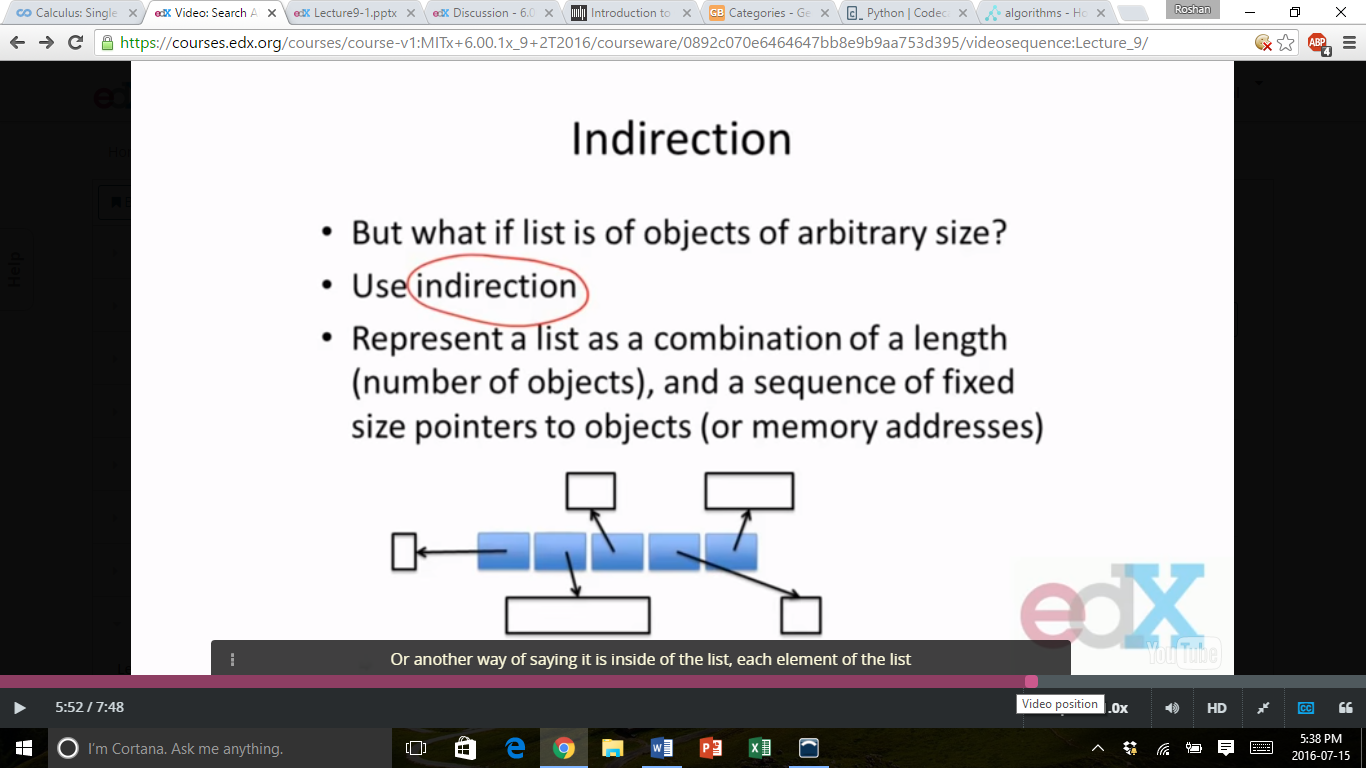
for i in range(len(L)):

if L[i] == e:

return True

return False

* + What is the complexity of this algorithm? Going by the worst case scenario, where the element is not in the list, the complexity of this algorithm is.
  + However, even *at best*, the search time is linear in the length of the list. But why is it *at best* linear? Should it not be constant in the *best* case scenario?
    - Prior to this, the assumption was always made that each test in the loop can be done in constant time.
    - But does Python actually retrieve theth element of a list in constant time?
* Indirection
  + Simple Case: I have a list of integers that I want to search through.
    - Each element is of the same size in the list (e.g. 4 units of memory, or 4 8-bit bytes).
    - Then the address, in memory, of theth element in my list is defined as start + 4 \* i, where start is the address of the start of the list.
    - So, yes, we can get to that point in memory only in constant time.
  + Harder Case: But what if the list is a list composed of objects of arbitrary size?
  + Use *indirection* 🡪 Let’s represent a list as a combination of a length (number of objects in the list), and a sequence of fixed size pointers to objects (or memory addresses).



* + If the length field is 4 units of memory (i.e. I need 4 units of memory to tell me how long something is), and each pointer occupies 4 units of memory, then the **address** of theth element in the list is stored at start + 4 + 4 \* i.
  + This address can be computed and found in constant time.
  + Thus, the whole search algorithm is linear. It’s called indirection because I first determine what location in the list I want to go to in constant time. Getting there, what I take out isn’t the value itself, but it’s a pointer to another location in memory. Then, I can go there in constant time and extract the value.
  + **Indirection:** Accessing an object by first accessing a pointer that contains a reference to the object that is being searched for.

Part 2: Binary Search

* Binary Search
  + Can we do better than for the search algorithm in terms of complexity?
  + If we don’t know anything about the values of the elements, then the answer is no, because it is an arbitrary list of arbitrary objects.
  + And, in the worst case, I will have to look at every single element at least once.
* What if the List is ordered?
  + Suppose the elements had already been sorted in ascending order, or have some kind of sequence to them that correlates to their properties.
  + Let’s look at the following code to see how a list of ascending order might be ordered.

def search(L, e):

for i in range(len(L)):

if L(i) == e:

return True

elif L(i) > e:

return False

return False

* + This improves the average complexity, but the worst case scenario is still that I would have to look through every element of the list and check for my element.
* Use Binary Search

1. Pick an index that divides the list in half.
2. Ask if.
3. If not ask if is larger or smaller than.
4. Depending on the answer, search the left or right half of for.
   * A new version of the divide-and-conquer algorithm.
     + Break a larger problem into smaller versions of the problem (smaller list), plus some simple operations.
     + Answer to the smaller problem is the answer to the original, larger problem.

* Binary Search
  + Here is an example of binary search being used to examine a list for an element.

def search(L, e):

def bSearch(L, e, low, high):

if high == low:

return L[low] == e

mid = low + int((high - low) / 2)

if L[mid] == e:

return True

elif L[mid] > e:

return bSearch(L, e, low, mid)

else:

return bSearch(L, e, mid + 1, high)

if len(L) == 0:

return False

else:

return bSearch(L, e, 0, len(L) - 1)

* Analysing Binary Search
  + Does the recursion halt?
    - Create a decrementing function so that the recursion does halt for sure.

1. Maps the values to which formal parameters are bound to non-negative integers, and in some sense, gives us the size of the problem.
2. When the value of the integer is less than or equal to 0, then the recursion terminates.
3. For each recursive call, the value of the function is strictly less than the value on entry to the instance of the function.
   * + Here, the small decrement in the function’s input is simply high – low.

* At least equal to 0 the first time the function is called (1).
* When exactly 0, no recursive call, returns a value (2).
* Otherwise, halt or recursively call with the value being halved (3).
  + - So, it does terminate.
  + What is the Complexity of the Algorithm?
    - How many recursive calls are made? (The amount of time taken for computation within each recursive call is constant.)
    - How many times can we divide high – low in half before it reaches a value of 0? The number of times that we can do this is captured by.
    - Thus, the complexity of the algorithm is.

Part 3: Selection Sort

* Sorting Algorithms
  + So what about the cost of a sorting algorithm? Let’s assume that the complexity of sorting a list can be denoted.
  + We don’t know what the actual complexity is, but we can just keep it as a general term.
  + Then, if we use a sort and search algorithm, we want to know if.
    - In other words, should we (1) sort the list and use binary search, or (2) use linear search.
  + Well, if we don’t know more about the list, then the answer is *no*. Because we can’t possibly create a sorting algorithm in *less* than linear time!
* Amortizing Costs
  + However, what if we want to search a list times? Then, the question we want to ask is if the cost of sorting and then binary searching an arbitrary number of times better than linear searching an arbitrary number of times, i.e..
    - This depends on, but one expects that if the sorting can be done efficiently, then it is better to sort the list first.
    - We are essentially amortizing, or spreading out the cost, of sorting over multiple searches (in some cases, in the order of millions), to make the sorting worthwhile.
* Selection Sort
  + Here is a piece of code showing an example of a selection sorting algorithm.

def selectionSort(L):

for i in range(len(L) - 1):

minIndex = i

minValue = L[i]

j = i + 1

while j < len(L):

if minValue > L[j]:

minIndex = j

minValue = L[j]

j += 1

temp = L[i]

L[i] = L[minIndex]

L[minIndex] = temp

return L

* Analysing Selection Sort
  + Loop Invariant
    - Given a prefix of the list L[0:i] and a suffix L[i + 1:len(L) – 1], then the prefix is sorted and no element in the prefix is larger than the smallest element in the suffix.

1. Base Case: Prefix is empty, suffix is the whole list – loop invariant holds true.
2. Induction Step: Move minimum element from the suffix to the end of the prefix. Since the invariant is true before the shift, the prefix is sorted after append – loop invariant holds true.
3. Exit: When I exit the loop, the prefix is the entire list, the suffix is empty, and so I have sorted the list.
   * Complexity
     + The complexity of the inner loop is, and the complexity of the outer loop is also. Thus, the overall complexity of the algorithm is, or quadratic complexity. This makes it relatively expensive.

Part 4: Merge Sort

* Merge Sort
  + This method uses a divide-and-conquer approach method to sort a list.
    1. If the list is of length 0 or 1, it has already been sorted.
    2. If the list has more than 1 element, you can split the list into 2 separate lists, and sort each one independently.
    3. Merge the results once both lists have been independently sorted.
* To merge, just look at the first element of each list, move the smaller one to the end of result.
* When one of the lists are empty, just copy the rest of the other list into the result.
* Complexity of Merge Sort
  + Comparison of the elements and copying each element are done in constant time.
  + The maximum number of comparisons made is.
  + The maximum number of copies that have to be made is.
  + Thus, we can say that merging two lists is linear in the length of the list.
* Merge
  + Here is an example of a merge algorithm that combines two lists that have already been sorted.

def merge(left, right, compare):

result = 0

i, j = 0, 0

while i < len(left) and j < len(right):

if left[i] <= right[j]:

result.append(left[i])

i += 1

else:

result.append(right[j])

j += 1

while i < len(left):

result.append(left[i])

i += 1

while j < len(right):

result.append(right[j])

j += 1

return result

* Complexity of Merge Sort
  + The complexity of the merge algorithm is in.
  + The complexity of the mergeSort algorithm is in.
  + Thus, the complexity of this algorithm is log-linear, or falls in, where.
  + Does come with a little bit of cost in memory, as it has to make a new copy of the list.

Part 4: Hashing

* Improving Efficiency
  + Combining binary search with merge sort is very efficient.
* If we search a list times, then the efficiency of the algorithm is.
  + Can We Do Better?
* Dictionaries use the concept of hashing to efficiently search information.
* The lookup can be done in time almost independent of the size of the dictionary.
* Hashing
  + Converts the key to access a value into an integer. It then uses this integer to index into a list (which can be done in constant time).
  + Conversion is done using a **hash function**. It maps large space of inputs to a smaller space of outputs. Thus, in principle, it is a many-to-one mapping.
  + When two inputs go the same output, they experience a **collision**. A really good hash function has uniform distribution – it minimizes the probability of a collision.
* Complexity of Hashing
  + If there are no collisions, then this is of complexity.
  + If everything is hashed to the same bucket, then the complexity is.
  + But, in general, you can trade of space in memory to make a large hash table, and with a good function get close to uniform distribution, and reduce complexity to close to.