Lecture 9: End of Python Fundamentals, Efficiency, Big O

MET CS 521 Information Structures in Python

Professor Alan Burstein

Reminders

- HW 4 assigned (due next week)
- Quiz 3 on Gradescope (due next week)

Plagiarism

- Multiple students have been caught with identical code
- I know my assignments have been on Chegg
    - If you use Chegg code, someone else will to and I can see that
- Working together and having identical code also counts as plagiarism
    - Each assignment should be done independently and ideas/concepts can be discussed
    - Sharing answers is just as bad as receiving it

Previously

1. Classes and Objects
2. Abstract Classes
3. Interfaces
4. Data Encapsulation
5. Inheritance
6. Polymorphism

Lecture 9 Agenda

1. Fundamentals of Python Conclusion
2. Efficiency of Programmers
3. Big O Notation

Fundamentals of Python

- ☑ Variable, Types, and Expressions
- ☑ Control Flow (if, for, while, try)
- ☑ Classes, Methods, and Functions
- ☑ Core Data Structures (Tuples, Lists, Dictionaries)
- ☑ Adanced Data Structures (Sets, Nodes, and LinkedLists)
- ☑ Core Algorithms and Strategies (Binary Search, Sorting, Recursion)
- ☑ Object-Oriented Design

Professional Applications of Python

- [x] Collaboration (git)
- [x] Style (commenting and naming
- [ ] Evaluating efficiency
- [ ] Using Python to present information (Matplotlib)
- [ ] Using Python for Data Science (Pandas)
- [ ] Security
- [ ] Machine Learning (SKlearn)

Who Cares About Efficiency?

- Computers are fast
    - Fastest Computer IBM Summit ~100PFLOPS
        - 10^17 floating point operations per second
- Data is bigger
    - Google indexes 30 **trillion pages** and handles 63,000 request per second
        - Naive search: 2*10^19, or would need 200 super computers
    - There are 52! Ways to order a deck of cards
        - 8*10^67

We Care About Efficiency

- If we make inefficient code, even very efficient computers cannot do the task.
- Recall our recursive Fibonacci Solution

```
def fib(n):
    if n == 1:
        return 1
    elif n == 2:
        return 1
    else:
        return fib(n-1) + fib(n-2)
```

- If n is 100, this will take 2^{100} operations, which would take the super computer 40 million years
- In the modern world, solutions must scale!

Algorithm vs Implementation

- We want to separate the idea of the theoretical algorithm and how we implement it
    - Using List vs Tuple or For vs While have minor differences, but more-or-less the same
- However, the fibonacci recursive algorithm is very different than the for loop algorithm

Time and Space

- There are two major concerns for a program efficiency:
    1. Time - how long it takes to run
    2. Space/Memory - how much memory does the machine need to do it
- This is often a trade-off where faster code may rely on more memory (such as memory caching(saving) to retrieve information faster)
- Most modern applications don’t need to worry about memory often compared to speed so we will focus on time efficiency
    - But there can be important times when this is a factor (low level systems, distributed systems)

How to Evaluate Efficiency

- Time
- Count Operations
- Others

Goals

- Distinguishes algorithms
- Implementation not important
- Consistent between environmental
- Predictable
- Can be expressed as a concise expression

Timing

- [x] Distinguishes algorithms
- [ ] Implementation not important (different results)
- [ ] Consistent between environment (computer speed)
- [ ] Predictable (difficult to calculate relationship and requires experiment)
- [ ] Can be expressed as a concise expression (…)

Counting Operations

- [x] Distinguishes algorithms
- [ ] Implementation not important (different results)
- [x] Consistent between environment (computer speed)
- [ ] Predictable (difficult to calculate relationship and requires experiment)
- [x] Can be expressed as a concise expression (…)

How Can We Improve It?

- We like counting operations, but the difference between minor differences in operations
    - “+” vs “acccessing” vs len()
- We want to relate time to compute input size
- We care about scalability, so we will focus on large input size
    - Defining input size
        - length of list, value of integer
        - Think about the “important” input(s), what will vary at scale

What About When Input Matters…

Example: Finding Element in List

- Element is first (will only take one operation)
    - Best case
- Element is somewhere in the middle (will take ~length/2)
    - Average case
- Element is last (will take length)
    - Worst case

Which Should We Analyze…

- All 3 have their uses
- However, we will focus on **worst case**
- Worse cast is when your app/server crashes
- Hope for the best, plan for the worst
  - Caveat being that if worst case is highly unlikely we may care more about average case

Big O

- Big O is the way we evaluate worst case run time of programs
- Tight upper bound on run time
- Asymptotic growth as N gets large
- We only really care about the **dominating term**
- O(N) where N is the input size (evaluate as N gets large)
  - Length of list
  - Magnitude of int
  - Up to you what N is

What Does This Mean

When N is on the order of millions, only the biggest term matters and constant factors are irrelevant

- $O(N^2 + 1) = O(N^{2)}$
- $O(N^2 + 100*N) = O(N^{2)}$
- $O(10 *N^2 + 1000*N + 1000000) = O(N^{2)}$

Example 1: Factorial

```
def fact(n):
    result = 1
    for i in range(1, n+1):
        result *= i
    return result
```

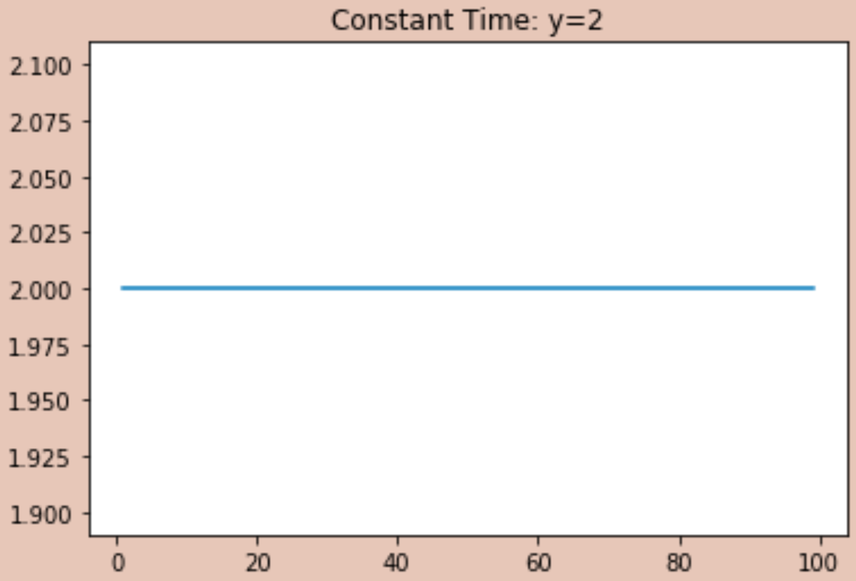
O(N)

Most Common Categories of Growth

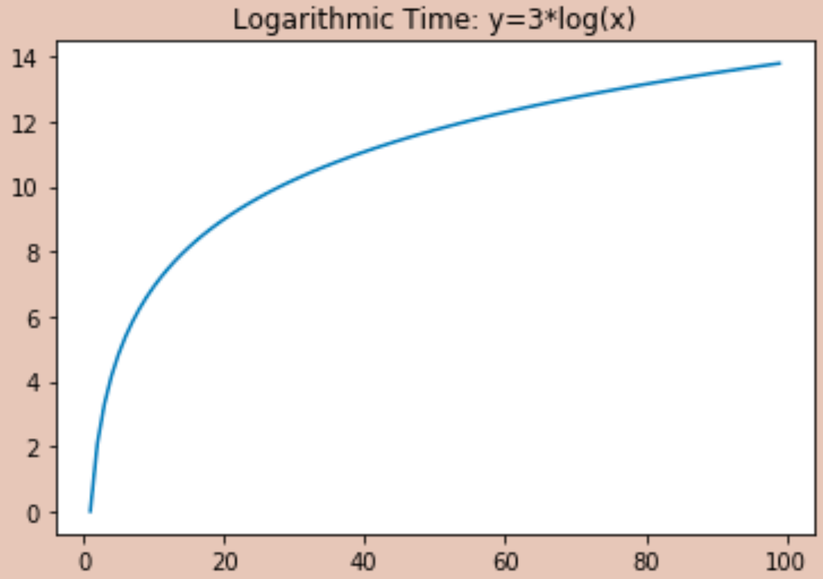
- Constant
- Linear
- Quadratic
- Logarithmic
- Exponential

Most Efficient

Ex: Looking up in a dictionary

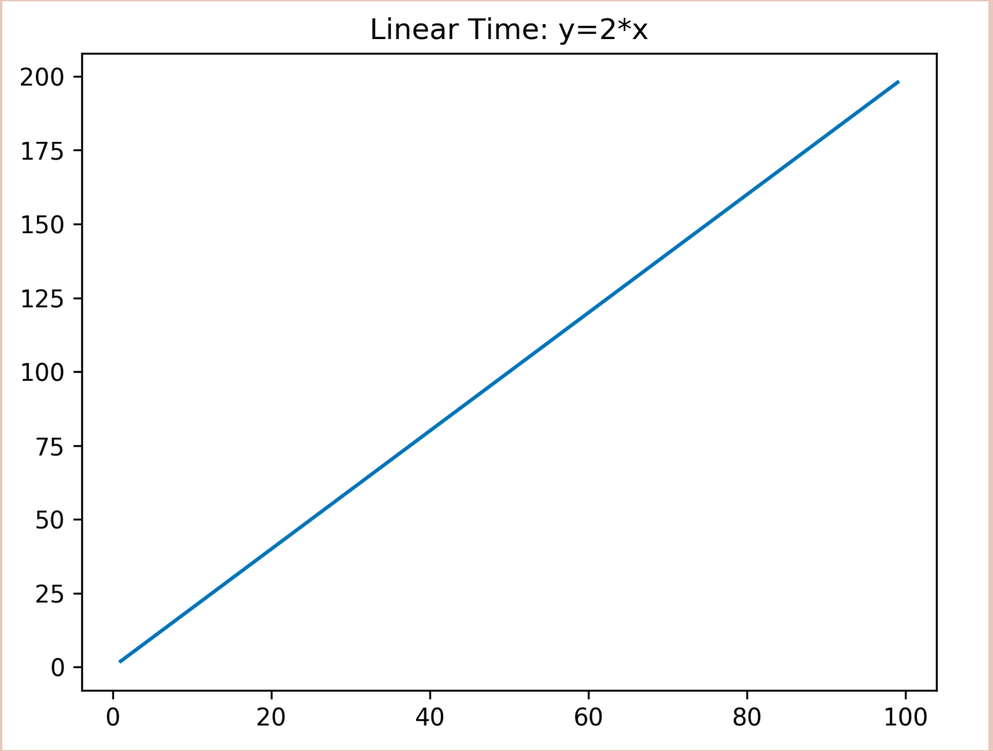


Ex: Binary Search



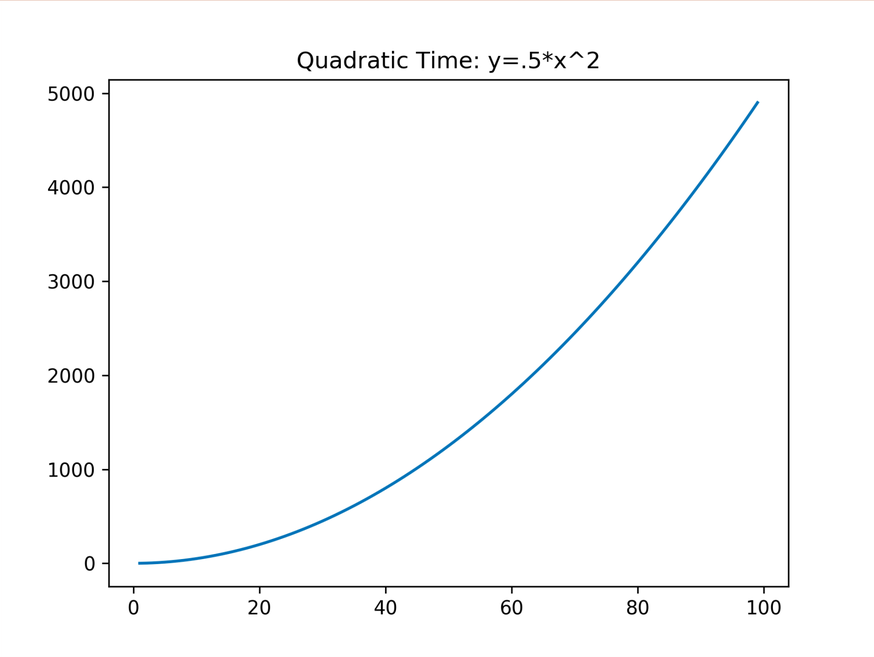
Normal

Ex: Lookup in a list, looping through list

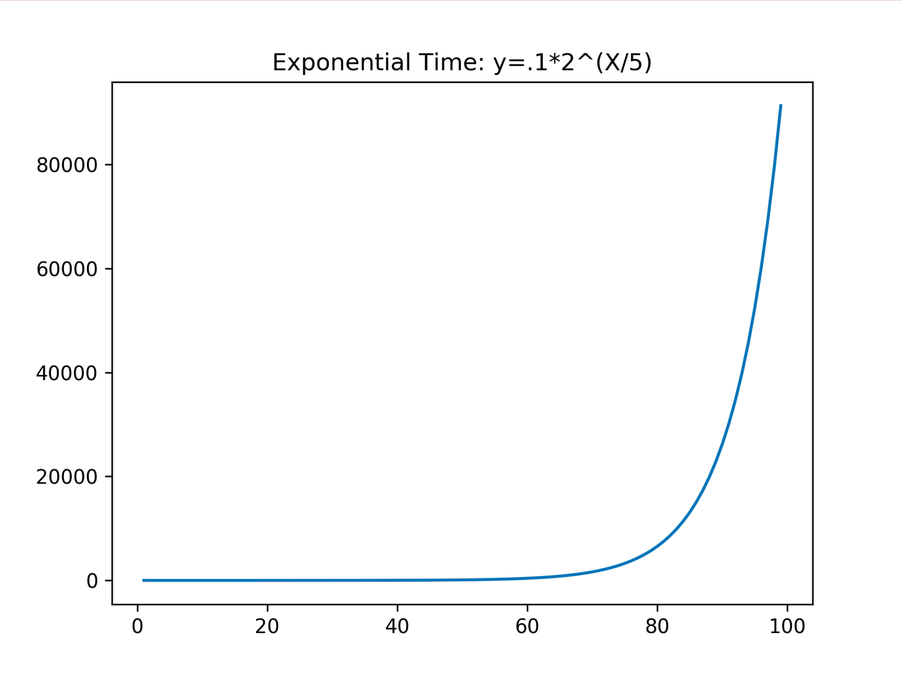


Inefficient

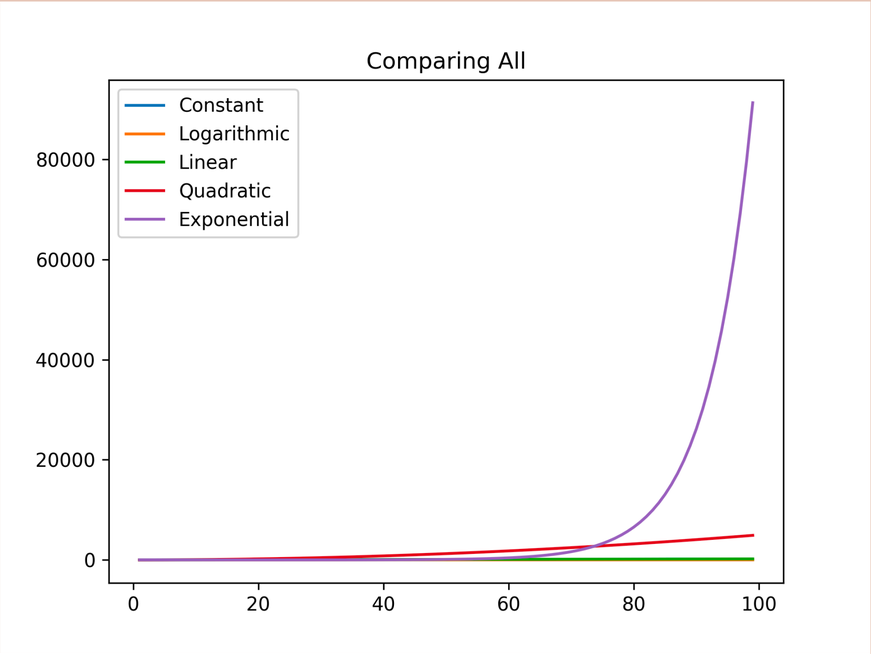
Ex: Insertion Sort

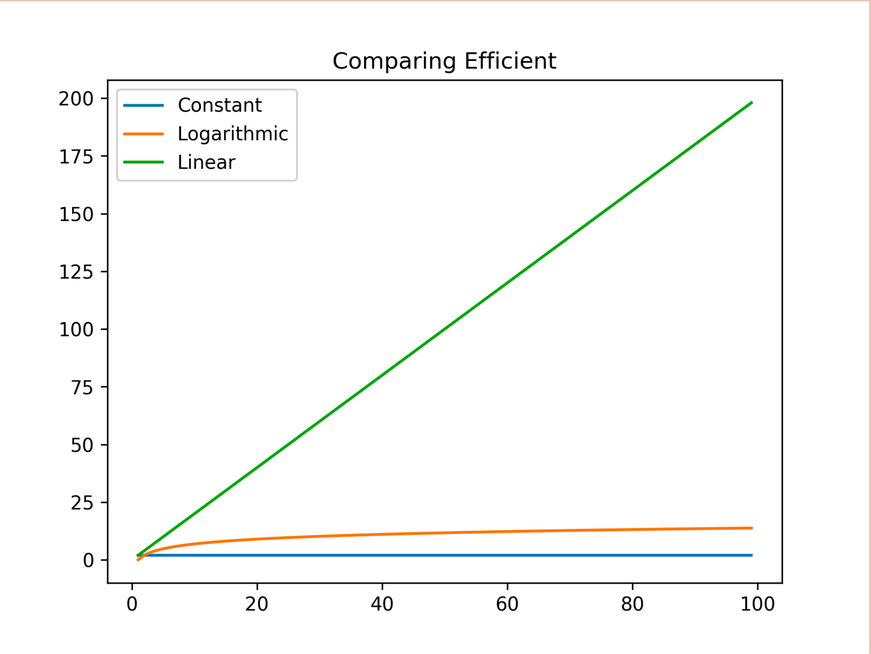


Ex: Recursive Fibonacci, Brute forcing passwords



Comparing them

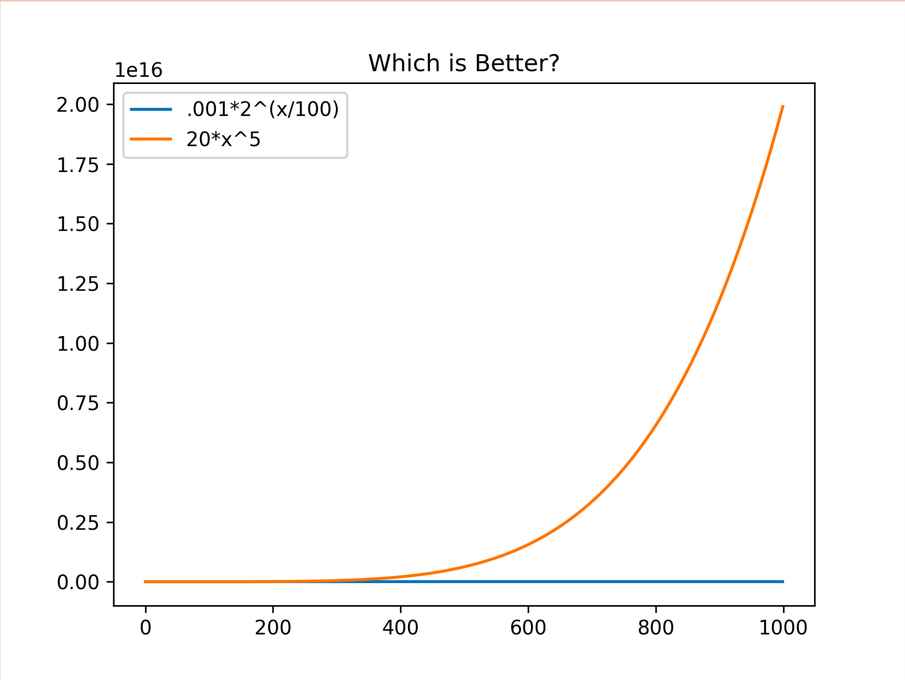


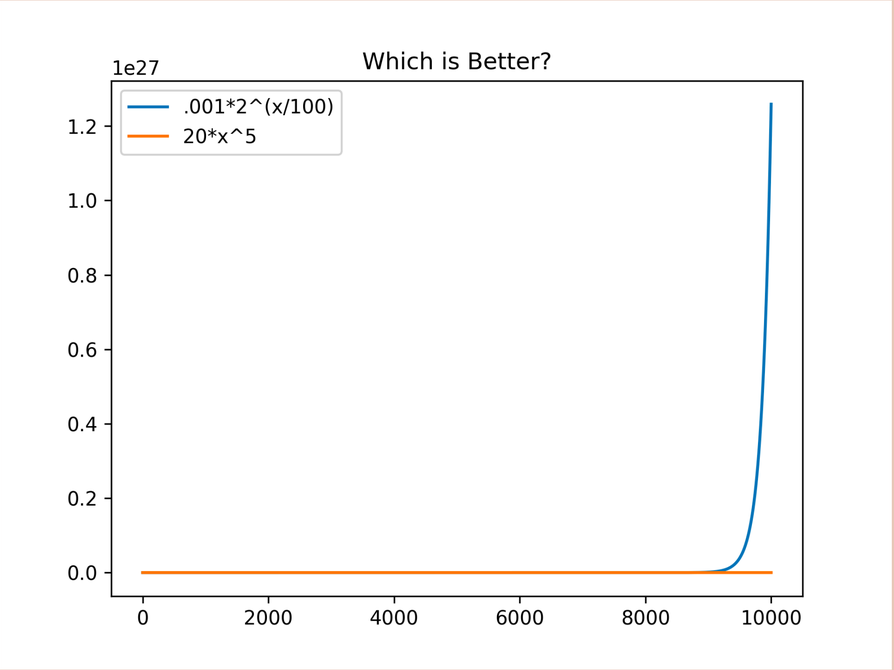


Comparing them: which is better

$$y = \frac{2^{\frac{x}{100}}}{1000}$$

$$y = 20x^5$$

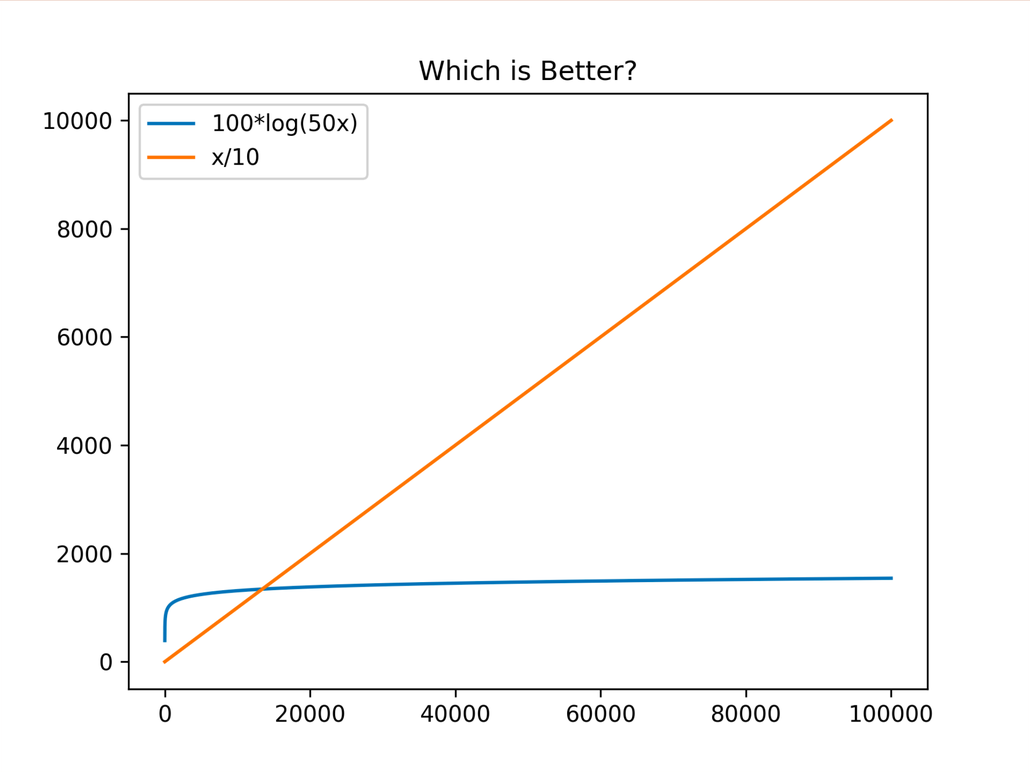




Comparing them: which is better

$$y = \frac{x}{10}$$

$$y = 100ln(50x)$$

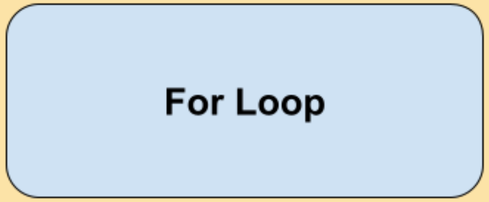


10 MIN BREAK

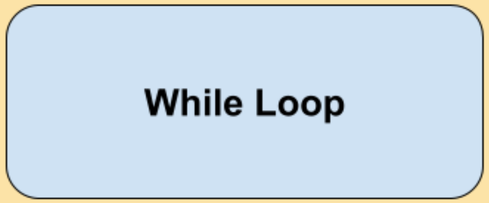
Analyzing Programs by Combining Classes

- Most programs won’t just be one of the above classes. You will frequently combine them
- Example… if you do a binary search for every element in the list, you will have Log-linear time O(NLog(N))
  - Understanding how to analyze an entire program is essential

Sequential Code



O(N)

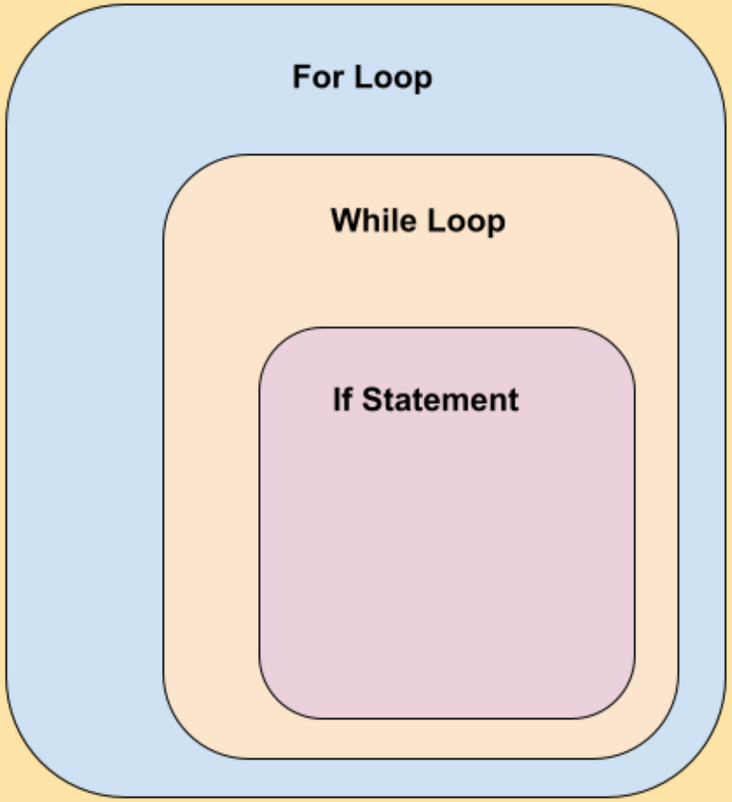


O(N)

O(1)

$$O(N + N + 1) = O(N)$$

Nested Code



O(N)

O(N)

O(1)

$$O(N * N * 1) = O(N^2)$$

Example 2: Insertion Sort

```
def insertion_sort(arr):
     for i in range(len(arr)):
         current_element = arr[i]

        # Find correct place in list
         j = 0
         while j<i and arr[j]<=current_element:
            j += 1

        # Move all larger elements over 1
         for x in range(i, j-1, -1):
             arr[x] = arr[x-1]

         # Insert current element
         arr[j] = current_element

 return arr
```

$O(N * (2N)) = O(N^2)$

Github Activity 1 (15 minutes)

Simplify the following Big O expressions **by finding the dominant term**

- $n^2 + 2n + 2$
- $2n^2 + 1000000n + 310$
- $log(n) + n + 4$
- $0.000001*n*log(n) + 300n$
- $2n^{20} + 3^n$

Big O at a Glance

- **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)

Analyzing 2 More Previous Algorithms

- Binary search
    - Each time, we halve the input space
- Merge sort
    - We divide input in half multiple times
    - We combine all lists

Example 3: Binary Search

```
def binarySearch (arr, left, right, x):
    # Check base case
    if right >= left:
        mid = (left + right - 1) // 2

        # If element is present at the middle
        if arr[mid] == x:
            return mid

        # If element is smaller than mid,
        elif arr[mid] > x:
            return binarySearch(arr, left, mid-1, x)

        # If element is bigger
        else:
            return binarySearch(arr, mid + left, right, x)

    else:
    # Element is not present in the array
    return -1
```

Each iteration is O(1)

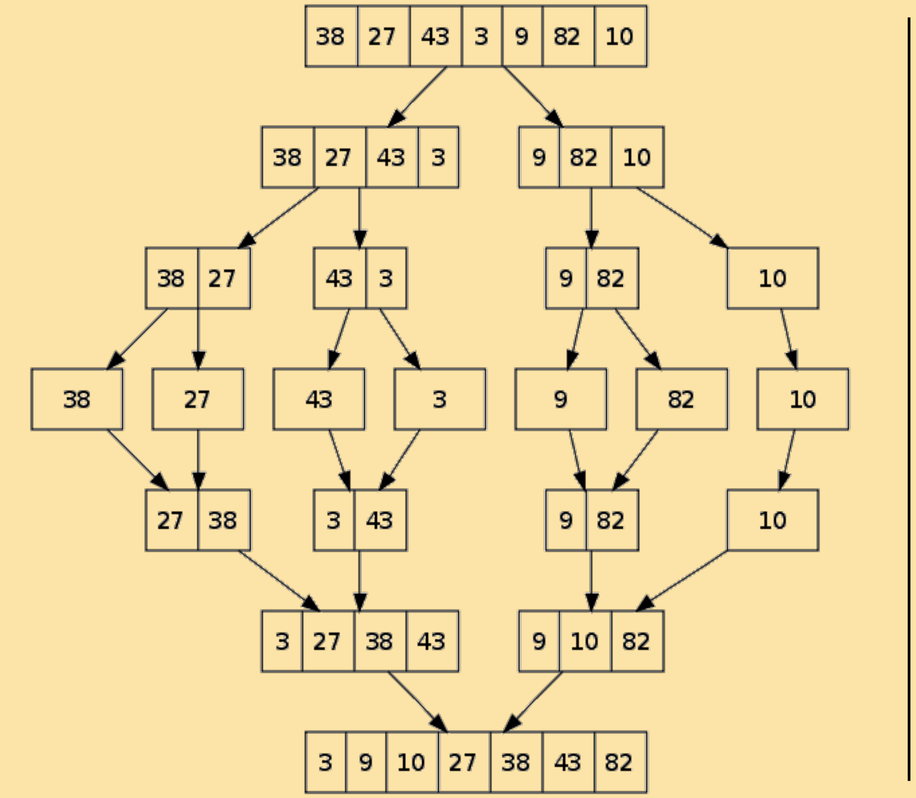
Each iteration will half right-left

log(len(arr)) iterations

O(log(N))

Example 4: Merge Sort

N = len(arr)



2 * log(N)

Github Activity 2 (10 min)

What's the runtime of the following code?

```
def random_function(N):
    total = 0

    for i in range(N):
        j = 0
        while j<i:
            j+=1
            total += 1
        for x in range(i, 1, -1):
            total += 1

    print("hi")

    return total
```

Quick Analysis of Some of Our Data Structures

- Searching/Inserting
    1. Do we have to look at each element in the data structure?
        - LinkedLists, Lists, tuples
        - O(N)
    2. Do we apply a function instead?
        - Dictionaries, sets (also based on hashing)
        - O(1)

Why You Should Care

- Evaluate code
- Use the right tools
    - Big O of different data structures
- Optimize for bottlenecks
- Technical interviews

Downloading a data set

- Hopefully you already were able to figure this one out
- First step is to find your data in a raw format (ie .csv file online)
  - If you are using an API, then you are required to have a script which calls the API and builds a data file (I recommend .csv)
- How can we get raw data from a GitHub repo
- requests module
  - requests.get(url, allow_redirects=True)
  - r.content.decode("utf-8")[:100]
  - open(file, ‘wb').write(content)

Conclusions

- Big O - https://web.mit.edu/16.070/www/lecture/big_o.pdf
- Extra Practice Problems on Blackboard

Admin

- Lab 3 due tomorrow
- Quiz 3 due tomorrow
- HW 4 due next Wednesday
  - Hardest assignment of the class
  - If at least 17 students submits by Sunday night
    - At least 1 point from autograder
    - I will extend the deadline by 1 week