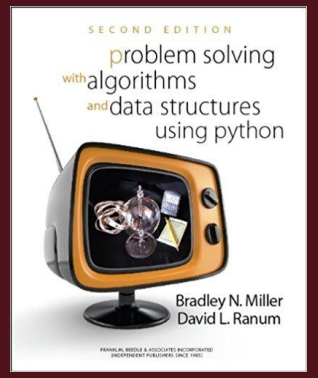
Problem Solving with Algorithms and Data Structure Using Python



Unit 3: Analysis

LECTURE 1: ANALYSIS

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- To understand why algorithm analysis is important.
- To be able to use “Big-O” to describe execution time.
- To understand the “Big-O” execution time of common operations on Python lists and dictionaries.
- To understand how the implementation of Python data impacts algorithm analysis.
- To understand how to benchmark simple Python programs.

What Is Algorithm Analysis?

LECTURE 1

How to compare programs?

- It is very common for beginning computer science students to compare their programs with one another. You may also have noticed that it is common for computer programs to look very similar, especially the simple ones.
- An interesting question often arises. When two programs solve the same problem but look different, is one program better than the other?

Demo Program: sumOfN.py

```
def sumOfN(n):
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i

    return theSum


print(sumOfN(10))
```

Demo Program: foo.py

```
def foo(tom):
    fred = 0
    for bill in range(1,tom+1):
       barney = bill
       fred = fred + barney

    return fred

print(foo(10))
```

Different Program and Same Algorithm

- The question we raised earlier asked whether one function is better than another. The answer depends on your criteria.
- The function **sumOfN** is certainly better than the function foo if you are concerned with **readability**.
- In fact, you have probably seen many examples of this in your introductory programming course since one of the goals there is to help you write programs that are easy to read and easy to understand.
- In this course, however, we are also interested in characterizing the algorithm itself.

Empirical Comparison

- Algorithm analysis is concerned with comparing algorithms based upon the amount of **computing resources** that each algorithm uses
- From this perspective, the two functions above seem very similar. They both use essentially the same algorithm to solve the summation problem.
- At this point, it is important to think more about what we really mean by computing resources. There are two different ways to look at this.
  - One way is to consider the amount of space or **memory** an algorithm requires to solve the problem. The amount of space required by a problem solution is typically dictated by the problem instance itself. Every so often, however, there are algorithms that have very specific space requirements, and in those cases we will be very careful to explain the variations.

Empirical Comparison

- As an alternative to space requirements, we can analyze and compare algorithms based on the amount of **time** they require to execute.
  - One way we can measure the execution time for the function **sumOfN** is to do a benchmark analysis. This means that we will track the actual time required for the program to compute its result. **(Measurement of Time)**
  - In Python, we can **benchmark** a function by noting the starting time and ending time with respect to the system we are using. In the time module there is a function called time that will return the current system clock time in seconds since some arbitrary starting point. By calling this function twice, at the beginning and at the end, and then computing the difference, we can get an exact number of seconds (fractions in most cases) for execution.

Empirical Comparison

ActiveCode 2 shows the sumOfN2 function with the timing calls embedded before and after the summation. The function returns a tuple consisting of the result and the amount of time (in seconds) required for the calculation.

ActiveCode 3, which shows a different means of solving the summation problem. This function, sumOfN3, takes advantage of a closed equation $\sum_{i=1}^{n} i = \frac{(n)(n+1)}{2}$ to compute the sum of the first n integers without iterating.

```
import time
def sumOfN2(n):
    start = time.time()
    theSum = 0
    for i in range(1,n+1):
        theSum = theSum + i
    end = time.time()
    return theSum,end-start

def sumOfN3(n):
    start = time.time()
    theSum = (n*(n+1))/2
    end = time.time()
    return theSum, end-start

n = 10000000
for i in range(5):
    print("Sum is %d required %10.7f seconds "%sumOfN2(n))
print("------")

for i in range(5):
    print("Sum id %d required %10.7f seconds " % sumOfN3(n))
```

Empirical Model

Demo Program: sumOfN2.py

```
Sum is 50000005000000 required  0.9230399 seconds
Sum is 50000005000000 required  0.8981094 seconds
Sum is 50000005000000 required  0.9574416 seconds
Sum is 50000005000000 required  0.9105866 seconds
Sum is 50000005000000 required  0.8984089 seconds
------
Sum id 50000005000000 required  0.0000000 seconds
Sum id 50000005000000 required  0.0000000 seconds
Sum id 50000005000000 required  0.0000000 seconds
Sum id 50000005000000 required  0.0000000 seconds
Sum id 50000005000000 required  0.0000000 seconds
```

Summary

- But what does this benchmark really tell us?
- Intuitively, we can see that the iterative solutions seem to be doing more work since some program steps are being repeated.
- Also, the time required for the iterative solution seems to increase as we increase the value of n. However, there is a problem.
- If we ran the same function on a different computer or used a different programming language, we would likely get different results. It could take even longer to perform **sumOfN3** if the computer were older.

Summary

- We need a better way to characterize these algorithms with respect to execution time. The **benchmark** technique computes the actual time to execute. It does not really provide us with a useful measurement, because it is dependent on a particular machine, program, time of day, compiler, and programming language.
- Instead, we would like to have a characterization that is independent of the program or computer being used. This measure would then be useful for judging the algorithm alone and could be used to compare algorithms across implementations.

Other Empirical Comparison Idea?

- Monte Carlo Simulation
- Overloading of +, -, *, /, % or compare operators.

Analytical Models

LECTURE 2

Analysis of Algorithms

- A complete analysis of the running time of an algorithm involves the following steps:
  1. Implement the algorithm completely.
  2. Determine the time required for each basic operation.
  3. Identify unknown quantities that can be used to describe the frequency of execution of the basic operations.
  4. Develop a realistic model for the input to the program.

Analysis of Algorithms

5. Analyze the unknown quantities, assuming the modelled input.
6. Calculate the total running time by multiplying the time by the frequency for each operation, then adding all the products.

Classical algorithm analysis on early computers could result in exact predictions of running times. Modern systems and algorithms are much more complex, but modern analyses are informed by the idea that exact analysis of this sort could be performed **in principle**.

Time Complexity

A good basic unit of computation for comparing the summation algorithms shown earlier might be to count the number of assignment statements performed to compute the sum. In the function **sumOfN**, the number of assignment statements is 1 (**theSum=0**) plus the value of n (the number of times we perform **theSum=theSum+i**).

We can denote this by a function, call it **T**, where **T(n)=1+n**. The parameter **n** is often referred to as the "size of the problem," and we can read this as "**T(n)** is the time it takes to solve a problem of size **n**, namely **1+n** steps."

Big-O Notation

- Computer scientists prefer to take this analysis technique one step further. It turns out that the exact number of operations is not as important as determining the most dominant part of the **T(n)** function.
- In other words, as the problem gets larger, some portion of the **T(n)** function tends to overpower the rest. This dominant term is what, in the end, is used for comparison.

Big-O Notation

- The order of magnitude function describes the part of **T(n)** that increases the fastest as the value of n increases. Order of magnitude is often called Big-O notation (for “order”) and written as **O(f(n))**.
- It provides a useful approximation to the actual number of steps in the computation. The function **f(n)** provides a simple representation of the dominant part of the original **T(n)**.

Big-O Notation (Asymptotic Behavior)

- In the above example, T(n)=1+n. As n gets large, the constant 1 will become less and less significant to the final result.
- If we are looking for an approximation for T(n), then we can drop the 1 and simply say that the running time is O(n).
- It is important to note that the 1 is certainly significant for T(n). However, as n gets large, our approximation will be just as accurate without it.

Big-O Notation (Asymptotic Behavior)

- As another example, suppose that for some algorithm, the exact number of steps is $T(n)=5n^2+27n+1005$. When n is small, say 1 or 2, the constant 1005 seems to be the dominant part of the function.
- However, as n gets larger, the n^2 term becomes the most important. In fact, when n is really large, the other two terms become insignificant in the role that they play in determining the final result.
- Again, to approximate T(n) as n gets large, we can ignore the other terms and focus on $5n^2$. In addition, the coefficient 5 becomes insignificant as n gets large. We would say then that the function T(n) has an order of magnitude $f(n)=n^2$, or simply that it is $O(n^2)$.

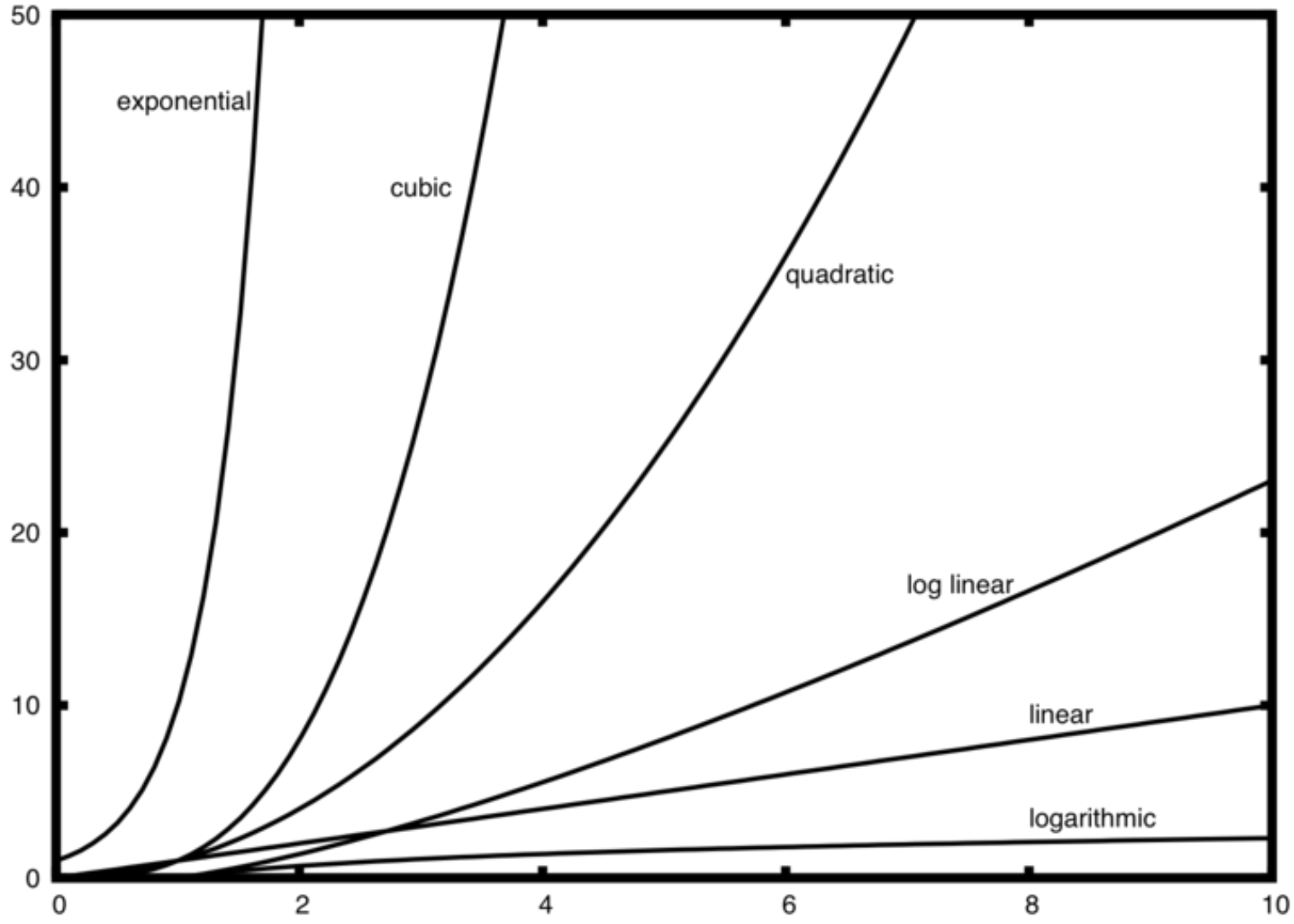
Best Case, Average Case, Worst Case

- The performance of an algorithm depends on the exact values of the data rather than simply the size of the problem.
- For these kinds of algorithms we need to characterize their performance in terms of **best case**, **worst case**, or **average case** performance.
- The **worst case** performance refers to a particular data set where the algorithm performs especially poorly. Whereas a different data set for the exact same algorithm might have extraordinarily good performance. However, in most cases the algorithm performs somewhere in between these two extremes (average case). It is important for a computer scientist to understand these distinctions so they are not misled by one particular case.

Best Case, Average Case, Worst Case

- A number of very common order of magnitude functions will come up over and over as you study algorithms.
- These are shown in Table 1. In order to decide which of these functions is the dominant part of any T(n) function, we must see how they compare with one another as n gets large.

f(n)	Name
1	Constant
$\log n$	Logarithmic
n	Linear
$n \log n$	Log Linear
n^2	Quadratic
n^3	Cubic
2^n	Exponential


50
40
30
20
10
0
0
2
4
6
8
10
exponential
cubic
quadratic
log linear
linear
logarithmic

Case Study

LECTURE 3

Listing 2

```
a=5
b=6
c=10
for i in range(n):
    for j in range(n):
        x = i * i
        y = j * j
        z = i * j
for k in range(n):
    w = a*k + 45
    v = b*b
d = 33
```

Case Study

- The number of assignment operations is the sum of four terms.
- The first term is the constant 3, representing the three assignment statements at the start of the fragment.
- The second term is $3n^2$, since there are three statements that are performed n^2 times due to the nested iteration.
- The third term is 2n, two statements iterated *n* times. Finally, the fourth term is the constant 1, representing the final assignment statement.

Case Study

- This gives us $T(n)=3+3n^2+2n+1=3n^2+2n+4$ $T(n)=3+3n^2+2n+1=3n^2+2n+4$. By looking at the exponents, we can easily see that the n^2 term will be dominant and therefore this fragment of code is $O(n^2)$.

- Note that all of the other terms as well as the coefficient on the dominant term can be ignored as *n* grows larger.

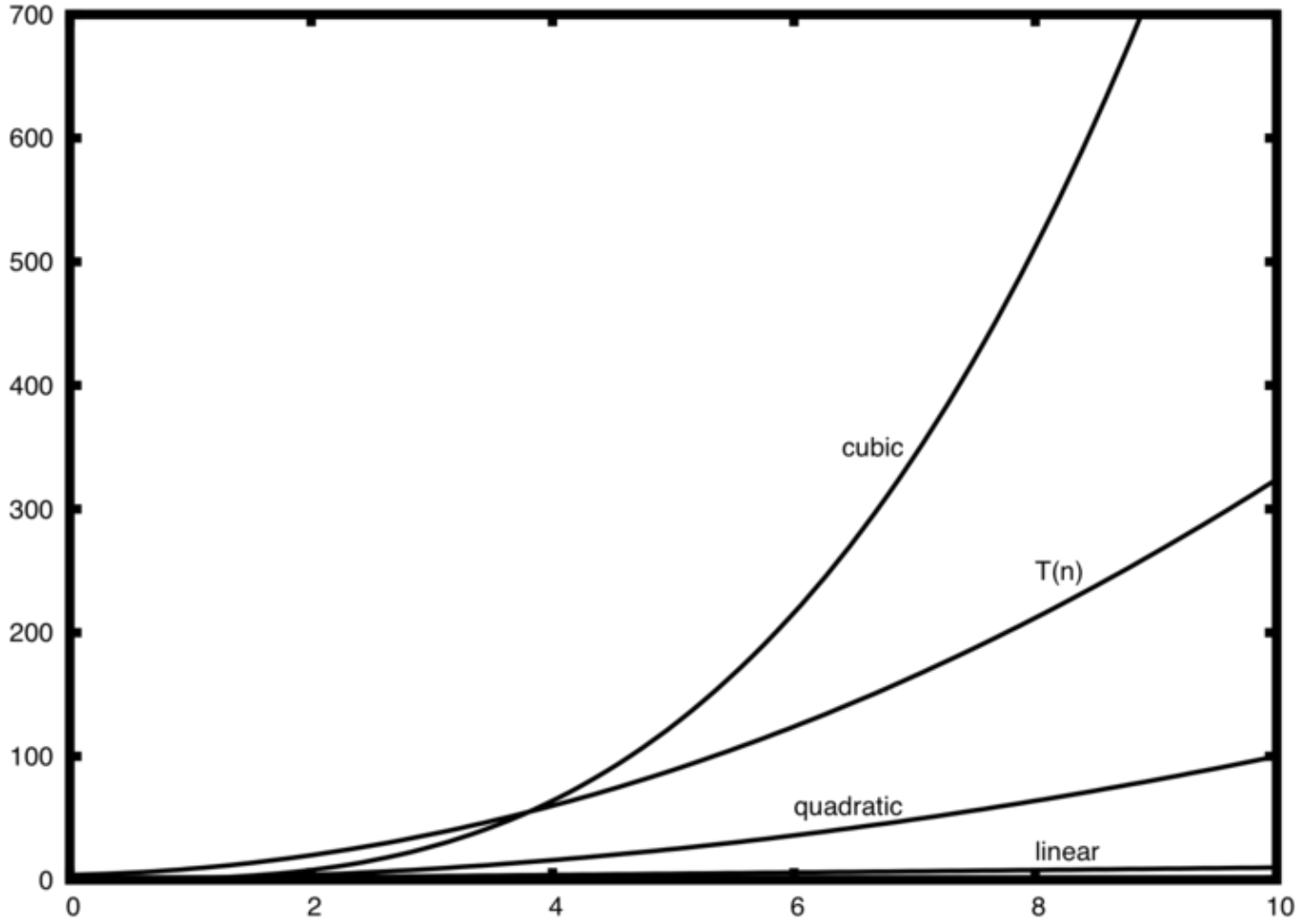


Figure shows a few of the common Big-O functions as they compare with the T(n) function discussed above. Note that T(n) is initially larger than the cubic function. However, as n grows, the cubic function quickly overtakes T(n). It is easy to see that T(n) then follows the quadratic function as n continues to grow.

An Anagram Detection Example

LECTURE 4

Anagram

- A good example problem for showing algorithms with different orders of magnitude is the classic anagram detection problem for strings.
- One string is an anagram of another if the second is simply a rearrangement of the first. For example, 'heart' and 'earth' are anagrams.
- The strings 'python' and 'typhon' are anagrams as well. For the sake of simplicity, we will assume that the two strings in question are of equal length and that they are made up of symbols from the set of 26 lowercase alphabetic characters.
- Our goal is to write a **boolean** function that will take two strings and return whether they are anagrams.


LISTEN
SILENT

Solution 1: Checking Off

Demo Program: anagram1.py

- Our first solution to the anagram problem will check to see that each character in the first string actually occurs in the second. If it is possible to "checkoff" each character, then the two strings must be anagrams.
- Checking off a character will be accomplished by replacing it with the special Python value None. However, since strings in Python are immutable, the first step in the process will be to convert the second string to a list.
- Each character from the first string can be checked against the characters in the list and if found, checked off by replacement. **anagram1.py** shows this function.

```
def anagramSolution1(s1,s2):
    alist = list(s2)
    pos1 = 0
    stillOK = True

    while pos1 < len(s1) and stillOK:
        pos2 = 0
        found = False
        while pos2 < len(alist) and not found:
            if s1[pos1] == alist[pos2]:
                found = True
            else:
                pos2 = pos2 + 1
        if found:
            alist[pos2] = None
        else:
            stillOK = False

        pos1 = pos1 + 1
    return stillOK

print(anagramSolution1('abcd','dcba'))
```

Solution 1: Checking Off

Demo Program: anagram1.py

- To analyze this algorithm, we need to note that each of the n characters in **s1** will cause an iteration through up to n characters in the list from **s2**. Each of the n positions in the list will be visited once to match a character from **s1**. The number of visits then becomes the sum of the integers from **1** to **n**. We stated earlier that this can be written as

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

$$= \frac{1}{2}n^2 + \frac{1}{2}n$$

- As n gets large, the n^2 term will dominate the n term and the $\frac{1}{2}$ can be ignored. Therefore, this solution is $O(n^2)$.

Solution 2: Sort and Compare

Demo Program: anagram2.py

- Another solution to the anagram problem will make use of the fact that even though s1 and s2 are different, they are anagrams only if they consist of exactly the same characters.
- So, if we begin by sorting each string alphabetically, from a to z, we will end up with the same string if the original two strings are anagrams. **anagram2.py** shows this solution.
- Again, in Python we can use the built-in sort method on lists by simply converting each string to a list at the start.

```
def anagramSolution2(s1,s2):
    alist1 = list(s1)
    alist2 = list(s2)
    alist1.sort()
    alist2.sort()
    pos = 0
    matches = True

    while pos < len(s1) and matches:
        if alist1[pos]==alist2[pos]:
            pos = pos + 1
        else:
            matches = False
    return matches

print(anagramSolution2('abcde','edcba'))
```

Solution 2: Sort and Compare

Demo Program: anagram2.py

- At first glance you may be tempted to think that this algorithm is O(n), since there is one simple iteration to compare the n characters after the sorting process.
- However, the two calls to the Python sort method are not without their own cost.
- As we will see in a later chapter, sorting is typically either $O(n^2)$ or O(nlogn), so the sorting operations dominate the iteration. In the end, this algorithm will have the same order of magnitude as that of the sorting process.

Solution 3: Brute Force

Demo Program: anagram3.py

- A brute force technique for solving a problem typically tries to exhaust all possibilities. For the anagram detection problem, we can simply generate a list of all possible strings using the characters from s1 and then see if s2 occurs.
- However, there is a difficulty with this approach. When generating all possible strings from s1, there are n possible first characters, n−1 possible characters for the second position, n−2 for the third, and so on.

Solution 3: Brute Force

Demo Program: anagram3.py

- The total number of candidate strings is n∗(n−1)∗(n−2)∗...∗3∗2∗1, which is n!. Although some of the strings may be duplicates, the program cannot know this ahead of time and so it will still generate n! different strings.
- It turns out that n! grows even faster than 2n as n gets large. In fact, if s1 were 20 characters long, there would be 20!=2,432,902,008,176,640,000 possible candidate strings. If we processed one possibility every second, it would still take us 77,146,816,596 years to go through the entire list. This is probably not going to be a good solution.

Solution 4: Count and Compare

Demo Program: anagram4.py

- Our final solution to the anagram problem takes advantage of the fact that any two anagrams will have the same number of a's, the same number of b's, the same number of c's, and so on. In order to decide whether two strings are anagrams, we will first count the number of times each character occurs.
- Since there are 26 possible characters, we can use a list of 26 counters, one for each possible character. Each time we see a particular character, we will increment the counter at that position. In the end, if the two lists of counters are identical, the strings must be anagrams. **anagram4.py** shows this solution.

```
def anagramSolution4(s1,s2):
    c1 = [0]*26
    c2 = [0]*26
    for i in range(len(s1)):
        pos = ord(s1[i])-ord('a')
        c1[pos] = c1[pos] + 1
    for i in range(len(s2)):
        pos = ord(s2[i])-ord('a')
        c2[pos] = c2[pos] + 1
    j = 0
    stillOK = True
    while j<26 and stillOK:
        if c1[j]==c2[j]:
            j = j + 1
        else:
            stillOK = False
    return stillOK

print(anagramSolution4('apple','pleap'))
```

Solution 4: Count and Compare

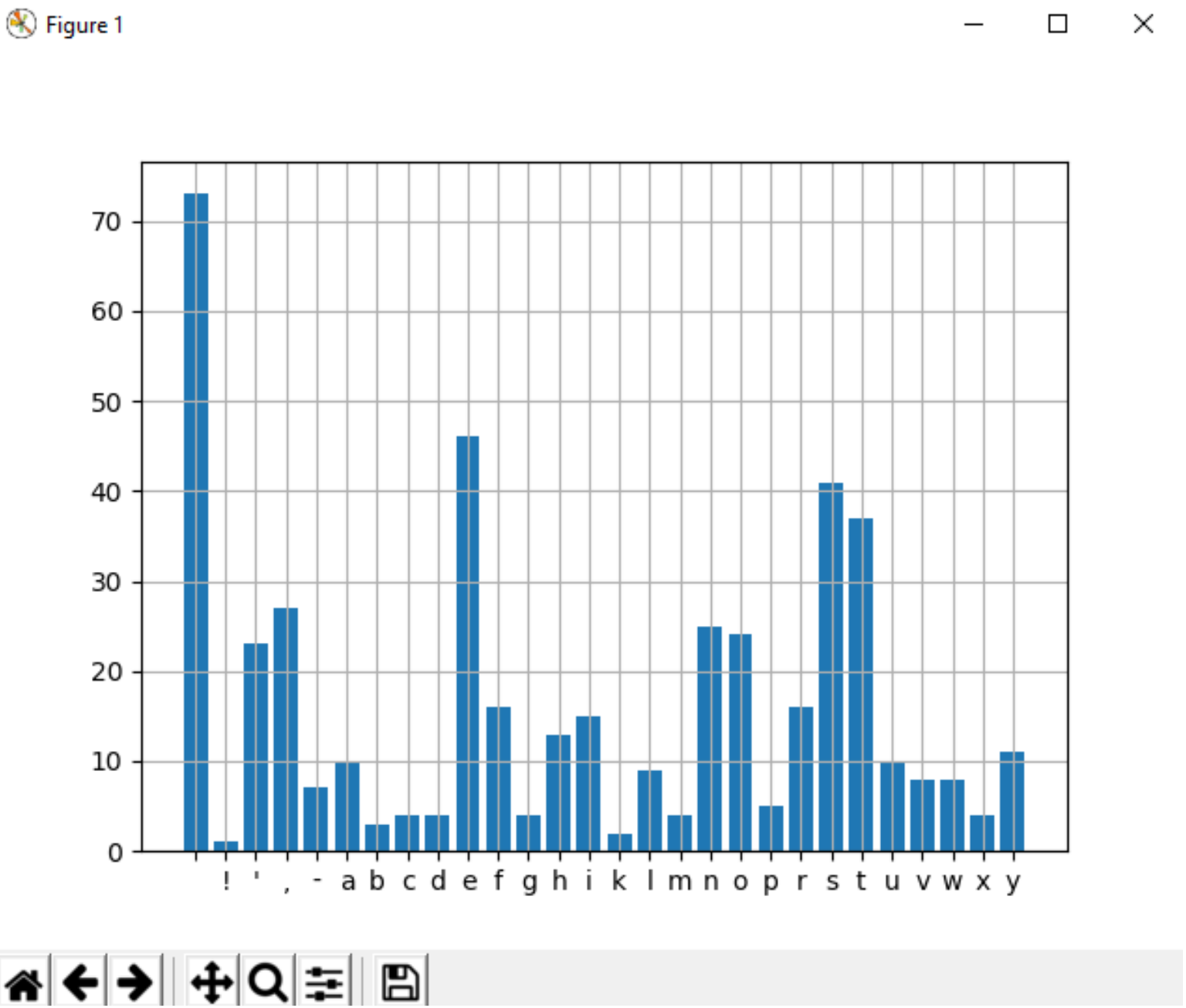
Demo Program: anagram4.py

- Again, the solution has a number of iterations. However, unlike the first solution, none of them are nested. The first two iterations used to count the characters are both based on n. The third iteration, comparing the two lists of counts, always takes 26 steps since there are 26 possible characters in the strings. Adding it all up gives us T(n)=2n+26 steps. That is O(n). We have found a linear order of magnitude algorithm for solving this problem.
- Before leaving this example, we need to say something about space requirements.
- Although the last solution was able to run in linear time, it could only do so by using **additional storage** to keep the two lists of character counts. In other words, this algorithm sacrificed space in order to gain time.

Solution 4: Count and Compare

Demo Program: anagram4.py

- This is a common occurrence. On many occasions you will need to make decisions between time and space trade-offs. In this case, the amount of extra space is not significant. However, if the underlying alphabet had millions of characters, there would be more concern. As a computer scientist, when given a choice of algorithms, it will be up to you to determine the best use of computing resources given a particular problem.



Histogram of Strings

Demo Program: histogram.py

Go PyCharm!!!

```
from pylab import *
from collections import Counter
from pprint import *

SENTENCE = """Only the fool would take trouble to verify that his sentence was composed of ten
a's, three b's, four c's,four d's, forty-six e's, sixteen f's, four g's, thirteen h's, fifteen
i's, two k's, nine l's, four m's, twenty-five n's, twenty-four o's, five p's, sixteen r's, forty-
one s's, thirty-seven t's, ten u's, eight v's, eight w's, four x's, eleven y's, twenty-seven
commas, twenty-three apostrophes, seven hyphens and, last but not least, a single !"""

# generate histogram
letters_hist = Counter(SENTENCE.lower().replace('\n', ''))
pprint(letters_hist)
letters_hist = dict(sorted(letters_hist.items()))
counts = letters_hist.values()
letters = letters_hist.keys()

# graph data
figure()
bar_x_locations = arange(len(counts))
bar(bar_x_locations, counts, align = 'center')
xticks(bar_x_locations, letters)
grid()
show()
```

Development of Algorithms

LECTURE 5

Classification of Design Techniques

Design of Algorithms

- Recursive
- Brute-Force
- Divide-and-Conquer
- Depth First
- Breadth First
- Backtracking
- Greedy -- local optimal
- Branch and Bound
- Dynamic Programming

Algorithm Expression

Python is my Algorithmic Expression

- There are many different ways to express an algorithm, including **natural language**, **pseudocode**, **flowcharts**, and **programming languages**.
- Natural language expressions of algorithms tend to be verbose and ambiguous, and are rarely used for complex or technical algorithms.
- Pseudocode and flowcharts are structured ways to express algorithms that avoid many of the ambiguities common in natural language statements, while remaining independent of a particular implementation language.
- Programming languages are primarily intended for expressing algorithms in a form that can be executed by a computer, but are often used as a way to define or document algorithms.

Program efficiency: time vs. space

- It is interesting to know how much of a particular resource (such as time or storage) is required for a given algorithm.
- Methods have been developed for the analysis of algorithms to obtain such quantitative answers, such as the big **O** notation.
- For example, the time needed for traversing an array of n slots is proportional to n, and we say the time is in the order of **O(n)**.
- However, accessing the **i**th element in an array takes only constant time, which is independent of the size of the array, thus is in the order of **O(1)**.

Running Time Notation Definitions

- **Common functions used in analysis:**
  - Constant function f(n) = C -- Constant algorithm does not depend on the input size.
  - Logarithm function f(n) = log n -- Logarithm function gets slightly slower as n grows.
  - Linear function f(n) = n -- Whenever n doubles, so does the running time.
  - N-Log-N function f(n) = n log n -- It grows a little faster than the linear function.
  - Quadratic function f(n) = n^2 -- Whenever n doubles, the running time increases fourfold.
  - Cubic Function and Other Polynomials
  - Exponential Function f(n) = b^n
  - Factorial Function f(n) = n!

The Development Process

- Specification of the task
- Design of a solution
- Implementation (coding) of the solution
- Analysis of the solution
- Testing and debugging
- Maintenance

Analysis and Testing

Analysis

Time analysis vs. space analysis.

Worst-case, average-case and best-case analyses.

Testing

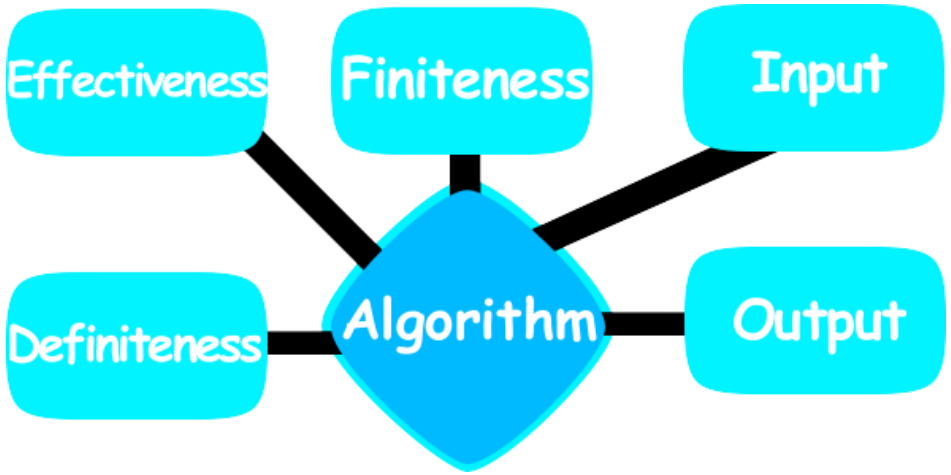
To serve as good test data, your test inputs need two properties:

1. You must know what output a correct program should produce for each test input.
2. The test inputs should include those inputs that are most likely to cause errors.

How to find test data that are most likely to cause errors? Try to **test boundary values**, which are particularly apt to cause errors, and extreme values. A boundary value of a problem is an input that is one step away from a different kind of behavior.

Fully Exercising Code

- make sure that each line of your code is executed at least once by some of your test data. If there is part of your code that is sometimes skipped altogether, make sure there is at least one test input that actually does skip this part of your code.

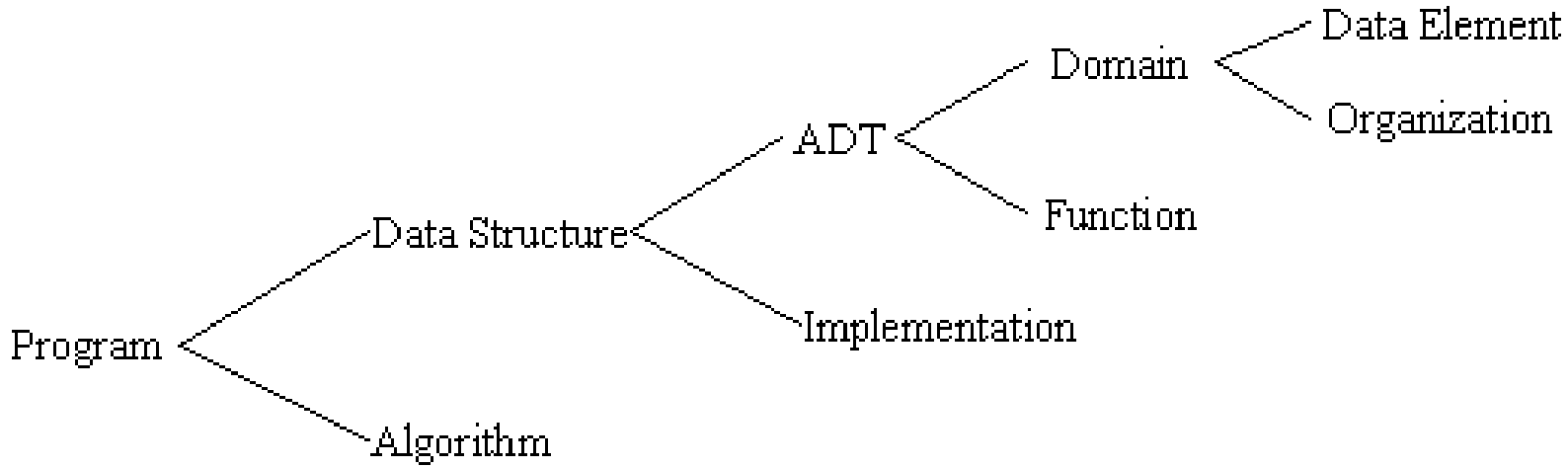


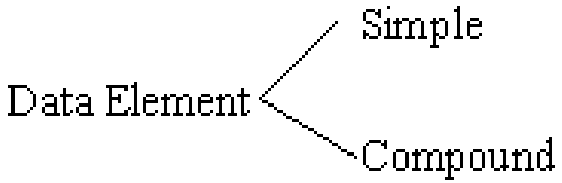
Performance of Data Structure

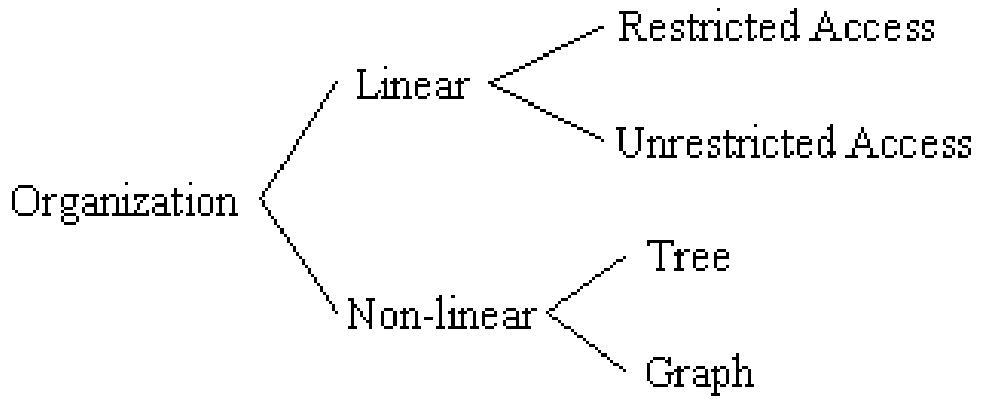
LECTURE 6

Performance of Python Data Structures

- Now that you have a general idea of Big-O notation and the differences between the different functions, our goal in this section is to tell you about the Big-O performance for the operations on Python lists and dictionaries.
- We will then show you some timing experiments that illustrate the costs and benefits of using certain operations on each data structure. It is important for you to understand the efficiency of these Python data structures because they are the building blocks we will use as we implement other data structures in the remainder of the book.
- In this section we are not going to explain why the performance is what it is. In later chapters you will see some possible implementations of both lists and dictionaries and how the performance depends on the implementation.


Program
Data Structure
ADT
Domain
Data Element
Organization
Function
Implementation
Algorithm


Data Element
Simple
Compound


Organization
Linear
Restricted Access
Unrestricted Access
Non-linear
Tree
Graph


Function
Access Function
Modification Function

What are data structures?

There are many definitions available:

- A data structure is **an aggregation of data** components that together constitute a meaningful whole.
- A data structure is **a way of arranging data** in a computer's memory or other disk storage.
- A data structure is **a collection of data**, organized so that items can be stored and retrieved by some fixed techniques.

There are several common data structures: arrays, linked lists, queues, stacks, binary trees, hash tables, graphs, etc. These data structures can be classified as either *linear* or *nonlinear* data structures, based on how the data is conceptually organized or aggregated.

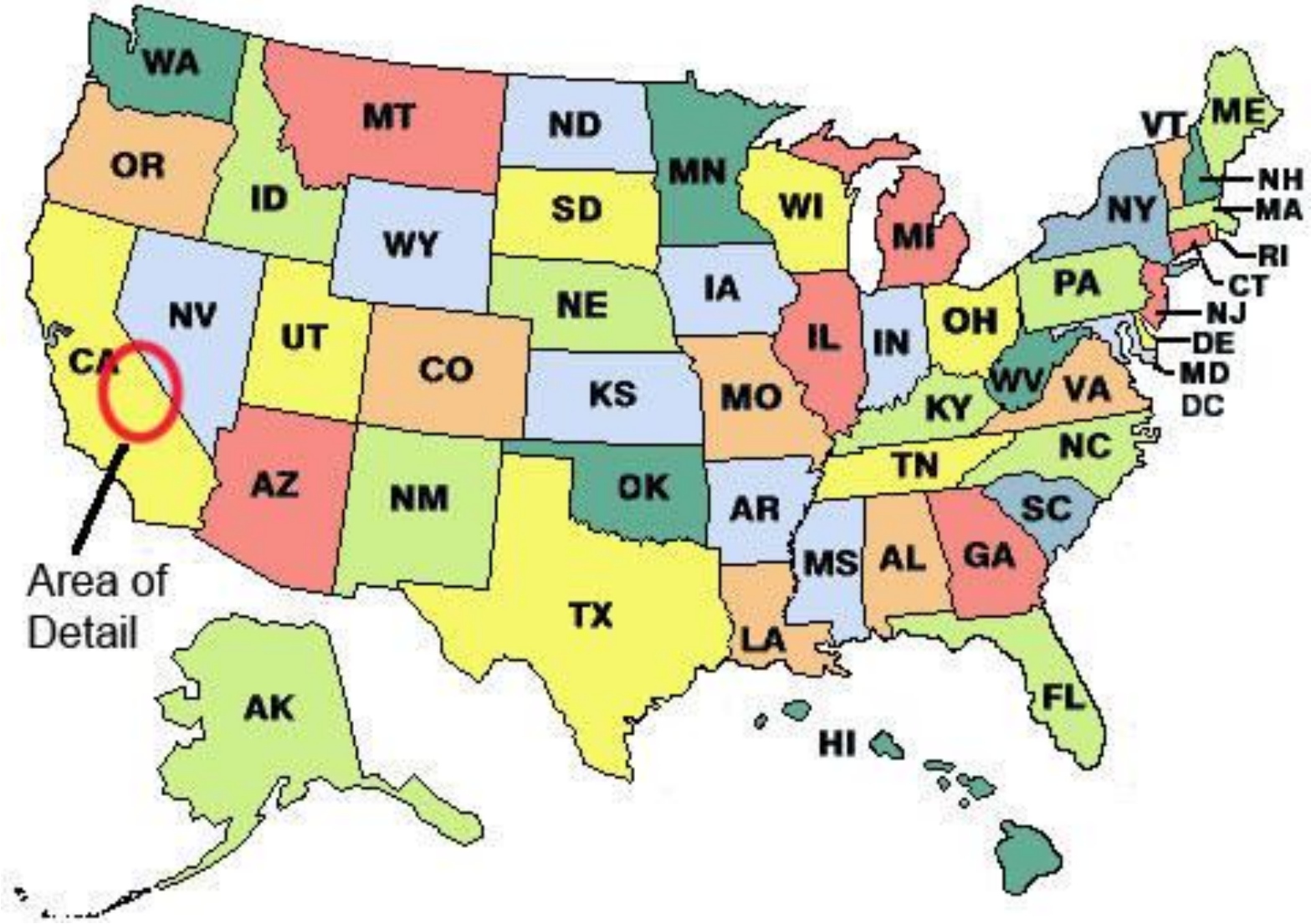
Data Structures

Linear structures:

The **array**, **list**, **queue**, and **stack** belong to this category. Each of them is a collection that stores its entries in a linear sequence, and in which entries may be added or removed at will. They differ in the restrictions they place on how these entries may be added, removed, or accessed. The common restrictions include FIFO and LIFO.

Non-linear structures:

Trees and **graphs** are classical non-linear structures. Data entries are not arranged in a sequence, but with different rules.


WA
OR
CA
ID
NV
MT
WY
UT
AZ
CO
NM
ND
SD
NE
KS
OK
TX
MN
IA
MO
AR
LA
WI
IL
MS
MI
IN
AL
OH
KY
TN
GA
FL
SC
NC
VA
WV
PA
NY
VT
ME
NH
MA
RI
CT
NJ
DE
MD
DC
AK
HI
Area of
Detail

What are abstract data types (ADT)?

- Remember the goal of software development?
- **Robustness**, **adaptability**, and **reusability**.
- Out of this effort to write better code arose a new metaphor for using and building data structures: abstract data type, which highlights the notion of abstractness.

What are abstract data types (ADT)?

- When we say "data type", we often refer to the primitive data types built into a language, such as **integer**, **real**, **character**, and **boolean**.
- An integer, is most likely implemented or represented in four bytes in the computer. However, when we use integers, we do not worry at all about its internal representation, or how these operations are implemented by the compiler in machine code.
- Additionally, we know that, even when we run our program on a different machine, the behavior of an integer does not change, even though its internal representation may change. What we know is that we can use primitive data types via their operational interface -- '+', '-', '*' and '/' for integers. The primitive data types were abstract entries.

What are abstract data types (ADT)?

- A stack or a queue is an example of an ADT.
- Both stacks and queues can be implemented using an array. It is also possible to implement stacks and queues using linked lists.
- This demonstrates the "abstract" nature of stacks and queues: how they can be considered separately from their implementation.

What are abstract data types (ADT)?

- Applying the idea of abstraction to data structures, we have ADT for data structures. On the one hand, an ADT makes a clean separation between interface and implementation, the user only sees the interface and therefore does not need to tamper with the implementation.
- On the other hand, if the implementation of an ADT changes, the code that uses the ADT does not break, since the interface remains the same.
- Thus, the abstraction makes the code more robust and easier to maintain. Moreover, once an ADT is built, it may be used multiple times in various contexts. For example, the list ADT may be used directly in application code, or may be used to build another ADT, such as a stack.

How do I choose the right data structures?

Interface and Efficiency

- When writing a program, one of the first steps is determining or choosing the data structures. What are the "right" data structures for the program?
- The **interface** of operations supported by a data structure is one factor to consider when choosing between several available data structures.
- Another important factor is the **efficiency** of the data structure: how much space does the data structure occupy, and what are the running times of the operations in its interface?

Lists

LECTURE 7

Lists

- The designers of Python had many choices to make when they implemented the list data structure. Each of these choices could have an impact on how fast list operations perform.
- To help them make the right choices they looked at the ways that people would most commonly use the list data structure and they optimized their implementation of a list so that the most common operations were very fast.
- Of course they also tried to make the less common operations fast, but when a tradeoff had to be made the performance of a less common operation was often sacrificed in favor of the more common operation.

Lists

- Two common operations are indexing and assigning to an index position. Both of these operations take the same amount of time no matter how large the list becomes. When an operation like this is independent of the size of the list they are O(1).
- Another very common programming task is to grow a list. There are two ways to create a longer list. You can use the append method or the concatenation operator. The append method is O(1). However, the concatenation operator is O(k) where k is the size of the list that is being concatenated. This is important for you to know because it can help you make your own programs more efficient by choosing the right tool for the job.

```
from timeit import Timer
def test1():
    l = []
    for i in range(1000):
        l = l + [i]
def test2():
    l = []
    for i in range(1000):
        l.append(i)
def test3():
    l = [i for i in range(1000)]
def test4():
    l = list(range(1000))

t1 = Timer("test1()", "from __main__ import test1")
print("concat ",t1.timeit(number=1000), "milliseconds")
t2 = Timer("test2()", "from __main__ import test2")
print("append ",t2.timeit(number=1000), "milliseconds")
t3 = Timer("test3()", "from __main__ import test3")
print("comprehension ",t3.timeit(number=1000), "milliseconds")
t4 = Timer("test4()", "from __main__ import test4")
print("list range ",t4.timeit(number=1000), "milliseconds")
```

Results:

concat 1.0619512102782382 milliseconds
append 0.0651755541930886 milliseconds
comprehension 0.027182621775242888 milliseconds
list range 0.012522388772757242 milliseconds

The timeit module does this because it wants to run the timing tests in an environment that is uncluttered by any stray variables you may have created, that may interfere with your function's performance in some unforeseen way.

Operation	Big-O Efficiency
index []	O(1)
index assignment	O(1)
append	O(1)
pop()	O(1)
pop(i)	O(n)
insert(i,item)	O(n)
del operator	O(n)
iteration	O(n)
contains (in)	O(n)
get slice [x:y]	O(k)
del slice	O(n)
set slice	O(n+k)
reverse	O(n)
concatenate	O(k)
sort	O(n log n)
multiply	O(nk)

timeit module

https://docs.python.org/3/library/timeit.html

- As a way of demonstrating this difference in performance let's do another experiment using the **timeit** module.
- Our goal is to be able to verify the performance of the pop operation on a list of a known size when the program pops from the end of the list, and again when the program pops from the beginning of the list.
- We will also want to measure this time for lists of different sizes.
- What we would expect to see is that the time required to pop from the end of the list will stay constant even as the list grows in size, while the time to pop from the beginning of the list will continue to increase as the list grows.

Demo Program: Listing4.py

```
import timeit
popzero = timeit.Timer("x.pop(0)", "from __main__ import x")
popend = timeit.Timer("x.pop()", "from __main__ import x")

x = list(range(2000000))
t1=popzero.timeit(number=1000)
print(t1)
x = list(range(2000000))
t2=popend.timeit(number=1000)
print(t2)
```

```
1.1411430901358997
6.33991753300478e-05
```

More Testing on pop(0), and pop()

- While our first test does show that pop(0) is indeed slower than pop(), it does not validate the claim that pop(0) is O(n) while pop() is O(1).
- To validate that claim we need to look at the performance of both calls over a range of list sizes. Listing 5 implements this test.

Demo Program: Listing5.py

```
from timeit import Timer
popzero = Timer("x.pop(0)",
                "from __main__ import x")
popend = Timer("x.pop()",
               "from __main__ import x")
print("          pop(0)              pop()")
for i in range(1000000,100000001,1000000):
    x = list(range(i))
    pt = popend.timeit(number=1000)
    x = list(range(i))
    pz = popzero.timeit(number=1000)
    print("%15.5f, %15.5f" %(pz,pt))
```

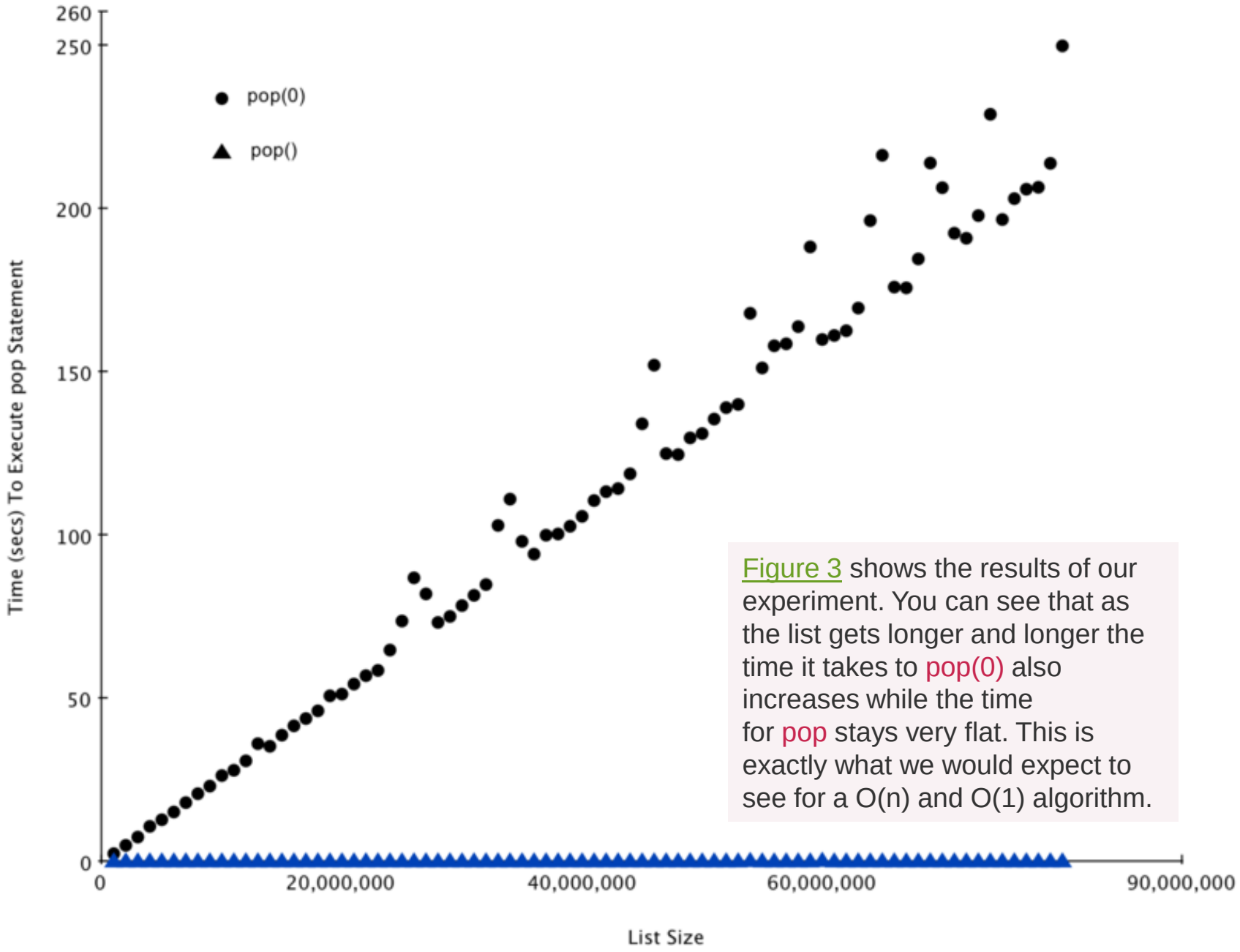


Figure 3 shows the results of our experiment. You can see that as the list gets longer and longer the time it takes to pop(0) also increases while the time for pop stays very flat. This is exactly what we would expect to see for a O(n) and O(1) algorithm.

Dictionary

LECTURE 8

Dictionary

- The second major Python data structure is the dictionary. As you probably recall, dictionaries differ from lists in that you can access items in a dictionary by a key rather than a position. Later in this book you will see that there are many ways to implement a dictionary. The thing that is most important to notice right now is that the get item and set item operations on a dictionary are O(1).
- Another important dictionary operation is the contains operation. Checking to see whether a key is in the dictionary or not is also O(1). The efficiency of all dictionary operations is summarized in Table 3.

Dictionary

- One important side note on dictionary performance is that the efficiencies we provide in the table are for average performance. In some rare cases the contains, get item, and set item operations can degenerate into O(n) performance but we will get into that in a later chapter when we talk about the different ways that a dictionary could be implemented.

operation	Big-O Efficiency
copy	O(n)
get item	O(1)
set item	O(1)
delete item	O(1)
contains (in)	O(1)
iteration	O(n)

Efficiency of Dictionary

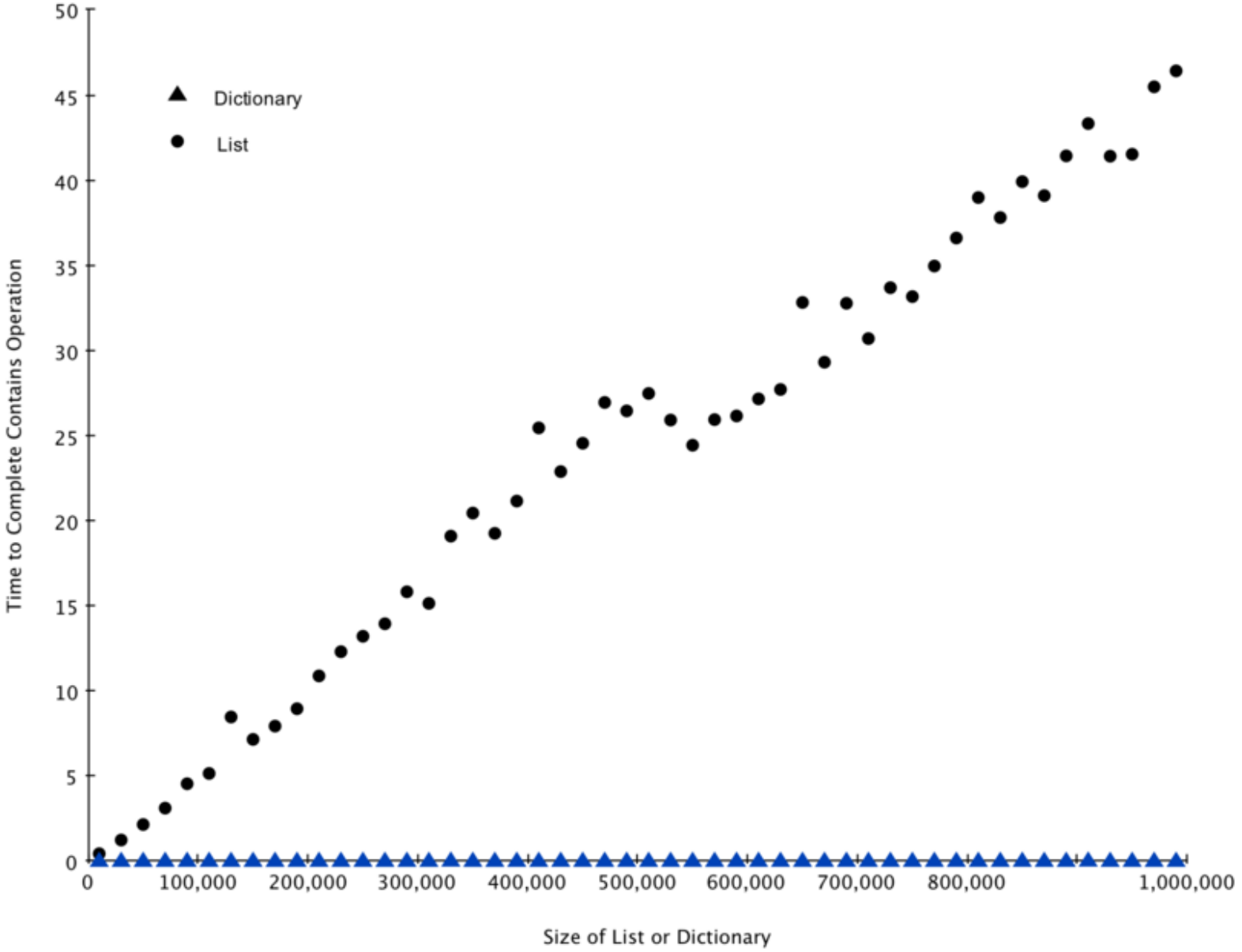
- For our last performance experiment we will compare the performance of the contains operation between lists and dictionaries. In the process we will confirm that the contains operator for lists is O(n) and the contains operator for dictionaries is O(1). The experiment we will use to compare the two is simple. We'll make a list with a range of numbers in it. Then we will pick numbers at random and check to see if the numbers are in the list. If our performance tables are correct the bigger the list the longer it should take to determine if any one number is contained in the list.
- We will repeat the same experiment for a dictionary that contains numbers as the keys. In this experiment we should see that determining whether or not a number is in the dictionary is not only much faster, but the time it takes to check should remain constant even as the dictionary grows larger.

Demo Program: Listing6.py

```
import timeit
import random

for i in range(10000,1000001,20000):
    t = timeit.Timer("random.randrange(%d) in x"%i,
                     "from __main__ import random,x")
    x = list(range(i))
    lst_time = t.timeit(number=1000)
    x = {j:None for j in range(i)}
    d_time = t.timeit(number=1000)
    print("%d,%10.3f,%10.3f" % (i, lst_time, d_time))
```

10000,	0.047,	0.001
30000,	0.140,	0.001
50000,	0.235,	0.001
70000,	0.336,	0.001
90000,	0.417,	0.001
110000,	0.529,	0.001
130000,	0.594,	0.001
150000,	0.725,	0.001
170000,	0.807,	0.001
190000,	0.911,	0.001
210000,	1.009,	0.001
230000,	1.060,	0.001
250000,	1.159,	0.001
270000,	1.333,	0.001
290000,	1.357,	0.001
310000,	1.616,	0.001
330000,	1.564,	0.001
350000,	1.716,	0.001
370000,	1.879,	0.001
390000,	1.879,	0.002
410000,	1.984,	0.001
430000,	2.073,	0.001
450000,	2.117,	0.001
470000,	2.199,	0.001
490000,	2.489,	0.001
510000,	2.441,	0.001
530000,	2.436,	0.001
550000,	2.737,	0.001
570000,	2.735,	0.001
590000,	2.998,	0.001
610000,	3.011,	0.001
630000,	3.214,	0.001
650000,	3.241,	0.002
670000,	3.383,	0.001
690000,	3.552,	0.001
710000,	3.389,	0.001
730000,	3.575,	0.001
750000,	3.669,	0.001
770000,	3.810,	0.001
790000,	3.753,	0.001
810000,	4.122,	0.001
830000,	3.981,	0.001
850000,	4.027,	0.001
870000,	4.286,	0.001
890000,	4.380,	0.001
910000,	4.457,	0.001
930000,	4.731,	0.001
950000,	4.732,	0.001
970000,	4.829,	0.001
990000,	5.071,	0.001


Dictionary
List
Time to Complete Contains Operation
Size of List or Dictionary
0
5
10
15
20
25
30
35
40
45
50
0
100,000
200,000
300,000
400,000
500,000
600,000
700,000
800,000
1,000,000

Summary

LECTURE 9

Summary

- Algorithm analysis is an implementation-independent way of measuring an algorithm.
- Big-O notation allows algorithms to be classified by their dominant process with respect to the size of the problem.