Data Processing and Analysis in Python
Lecture 11
Complexity Analysis

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

DR. ADAM LEE

Measuring the Efficiency of Algorithms

- When choosing algorithms, we often have to settle for a **space** versus **time** tradeoff
  - An algorithm can be designed to gain faster run time at the cost of using extra space (memory), or the other way around
- Memory is now quite inexpensive for desktop and laptop computers, but not yet for miniature devices
- One way to measure the time cost of an algorithm is to use computer's clock to obtain actual run time
  - **Benchmarking** or **profiling**
- Can use **time**() in time module
  - Returns number of seconds that have elapsed between current time on the computer's clock and January 1, 1970

Example – Timing1

```
import time
problemSize = 10000000
print("%12s16s" % ("Problem Size", "Seconds"))
for count in range(5):
    start = time.time()

    # Start of the algorithm
    work = 1
    for x in range(problemSize):
        work += 1
        work -= 1
    # End of the algorithm

    elapsed = time.time() - start
    print("%12d%16.3f" % (problemSize, elapsed))
    problemSize *= 2
```

Problem Size	Seconds
10000000	3.8
20000000	7.591
40000000	15.352
80000000	30.697
160000000	61.631

Figure 11-1 The output of the tester program

Example – Timing2

```
import time
problemSize = 1000
print("%12s10s" % ("Problem Size", "Seconds"))
for count in range(5):
    start = time.time()

    # Start of the algorithm
    work = 1
    for j in range(problemSize):
        for k in range(problemSize):
            work += 1
            work -= 1
    # End of the algorithm

    elapsed = time.time() - start
    print("%12d%10.3f" % (problemSize, elapsed))
    problemSize *= 2
```

Problem Size	Seconds
1000	0.387
2000	1.581
4000	6.463
8000	25.702
16000	102.666

Figure 11-2 The output of the second tester program with a nested loop and initial problem size of 1000

Counting Instructions

- Problems in Timing examples:
  - Running times of an algorithm differ from machine to machine
  - Running time varies with OS and programming language, too
  - Impractical to determine the running time for large data sets
- Another technique is to count the instructions executed with different problem sizes
  - We count the instructions in the high-level code in which the algorithm is written, not instructions in the executable machine language program
- Distinguish between:
  - Instructions that execute the same number of times regardless of problem size
  - Instructions whose execution count varies with problem size

Example – Counting

```
problemSize = 1000
print("%12s%15s" % ("Problem Size", "Iterations"))
for count in range(5):
    number = 0

    # Start of the algorithm
    work = 1
    for j in range(problemSize):
   for k in range(problemSize):
     number += 1
     work += 1
     work -= 1
    # End of the algorithm

    print("%12d%15d" % (problemSize, number))
    problemSize *= 2
```

Problem Size	Iterations
1000	1000000
2000	4000000
4000	16000000
8000	64000000
16000	256000000

Figure 11-3 The output of a tester program that counts iterations

Example – Countfib

```
def fib(n, counter = None):
  if counter: counter.increment()
    if n < 3:
        return 1
    else:
        return fib(n - 1, counter) + fib(n - 2, counter)

problemSize = 2
print("%12s%15s" % ("Problem Size", "Calls"))
for count in range(5):
    counter = Counter()

    # Start of the algorithm
    fib(problemSize, counter)
    # End of the algorithm

    print("%12d%15s" % (problemSize, counter))
    problemSize *= 2
```

Problem Size	Calls
2	1
4	5
8	41
16	1973
32	4356617

Figure 11-4 The output of a tester program that runs the Fibonacci function

Orders of Complexity

Problem Size n	Exponential (2^n)	Quadratic (n^2)	Linear (n)	Logarithmic ($\log_2 n$)
100	Off the charts	10,000	100	7
1,000	Off the charts	1,000,000	1000	10
1,000,000	Really off the charts	1,000,000,000,000	1,000,000	20

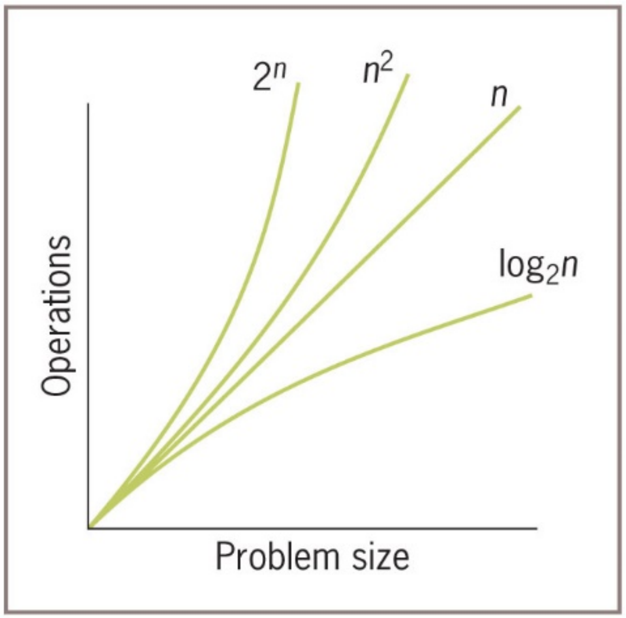

Figure 11-6 A graph of some sample orders of complexity

UNIVERSITY OF MARYLAND
ROBERT H. SMITH SCHOOL OF BUSINESS

Big-O Notation

- The amount of work in an algorithm typically is the sum of several terms in a polynomial
  - We focus on one term as **dominant**
- As *n* becomes large, the dominant term becomes so large that the amount of work represented by the other terms can be ignored
  - Asymptotic analysis
- **Big-O notation**: used to express the efficiency or computational complexity of an algorithm
  - $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, etc.

Measuring the Memory Used by an Algorithm

- A complete analysis of the resources used by an algorithm includes the amount of memory required
- We focus on rates of potential growth
  - Some algorithms require the same amount of memory to solve any problem
  - Other algorithms require more memory as the problem size gets larger

Best-Case, Worst-Case, and Average-Case Performance

- Analysis of a linear search considers three cases:
  - In the worst case, the target item is at the end of the list or not in the list at all
    O(n)
  - In the best case, the algorithm finds the target at the first position, after making one iteration
    O(1)
  - Average case: add number of iterations required to find target at each possible position; divide sum by n
    O(n)

Search and Sort Algorithms

- Search Algorithms
  - Sequential Search or Linear Search
  - Binary Search
  - ...
- Sort Algorithms
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
  - Quicksort
  - Merge Sort
  - ...

Recursive Fibonacci is an Exponential Algorithm: $O(2^n)$

Problem Size	Calls
2	1
4	5
8	41
16	1973
32	4356617

Figure 11-4 The output of a tester program that runs the Fibonacci function

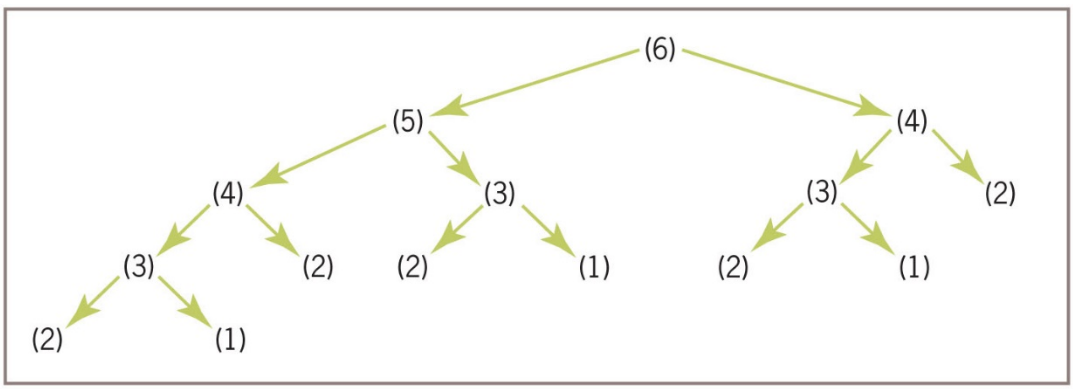

Figure 11-11 A call tree for `fib(6)`

- Can we convert Fibonacci to a Linear Algorithm?