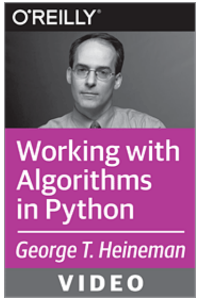
O'REILLY®

Analyzing Algorithms
A Brief Introduction



O'REILLY

Algorithm Formalities

- Definition of an *algorithm*
  - An algorithm describes the **computational steps** to compute an exact answer for a single **problem instance** on a **sequential deterministic computer**
- How to compare two different algorithms that solve the same problem?

Algorithm Formalities

- Focus on behavior inherent in the algorithm
  - Abstract away from a specific implementation…
  - And programming language used…
  - And computing platform on which code executes (i.e., RAM, CPU clock speed)
- In long run, these differences do not matter

Asymptotic Analysis

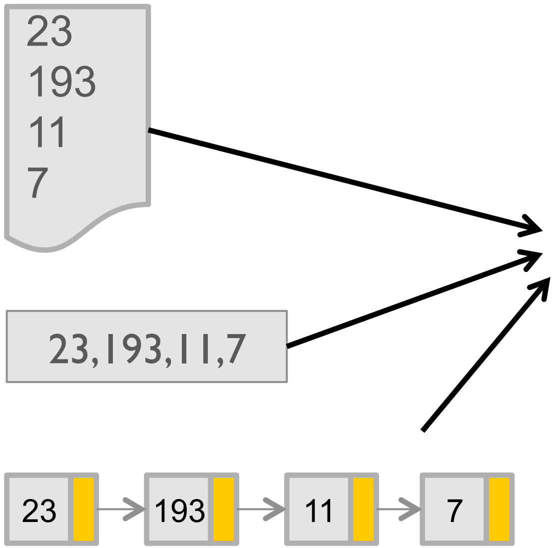
- Characterize **time complexity**
  - Time for algorithm to complete
  - Calculate time as function *t(n)* relating the number of steps to problem instance size, *n*
- Characterize **space complexity**
  - Amount of computer storage required
  - Determine required space *s(n)* in similar fashion

Algorithm Cost Model

- From algorithm description one can determine total number of steps based on *n*
  - Sequential, deterministic computing platform
- Assume a **constant cost** to every operation
  - Enables using *t(n)* to estimate execution time

Small Algorithm Example
SUM n NUMBERS

This algorithm uses **n** addition operations regardless of how data is stored



SUM *n* NUMBERS

```
for d in numbers:
    sum = sum + d
```

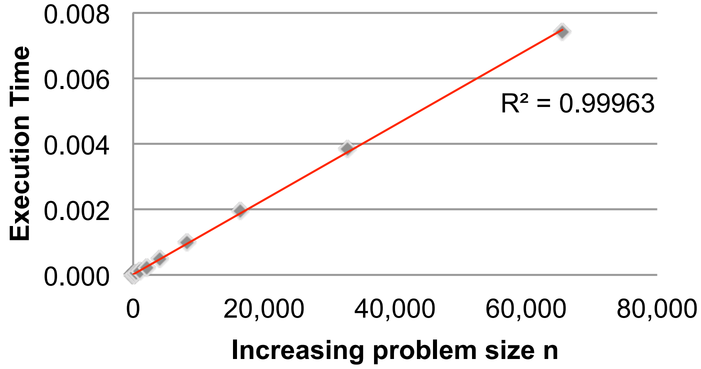
Time complexity: $t(n)$ is directly proportional to n

Asymptotic Growth

- Evaluate *t(n)* as problem size *n* doubles
  - The true measure for understanding performance
  - Goal to determine **order of growth** or O(*f(n)*)
  - A formal treatment of O(*f(n)*) can be found in textbooks

Asymptotic Growth

- Execution times of SUM reveal correlation between *n* and *t(n)*
  – SUM exhibits linear behavior
  – SUM is O(n)



Asymptotic Growth

- Execution times of SUM reveal correlation between n and $t(n)$
  - For sufficiently large values of n, $t(n)$ is at most $c*n$ where c is some fixed constant
  - In theory, don't need to determine c
  - Here what matters is $f(n) = n$

Performance Computation

Add 2^n random digits for $n = 1$ to 16

```
scores = {}
trial = 1
while trial <= 16:
    numbers = [random.randint(1,9) for i in range(2**trial)]
    now = time()
    sum = 0
    for d in numbers:
        sum = sum + d
    done = time()

    scores[trial] = (done-now)
    trial += 1
```

Review code to see expectation of *O(n)* behavior

Consider Alternate Problem

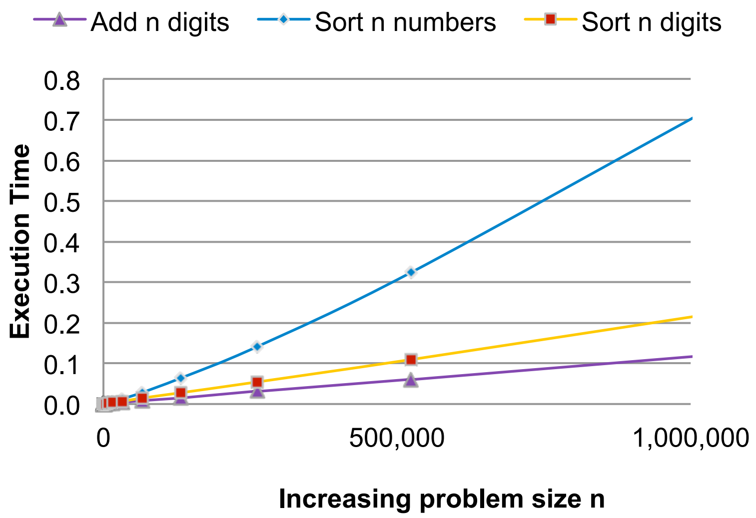
Sort 2^n random digits for n = 1 to 16

```
scores = {}
trial = 1
while trial <= 16:
    numbers = [random.randint(1,9) for i in range(2**trial)]
    now = time.clock()
    numbers.sort()
    done = time.clock()

    scores[trial] = (done-now)
    trial += 1
```

Unclear what performance expectation you should have, since code uses built-in Python *sort()* function

Compare Three Execution Times



SORT N RANDOM NUMBERS ($1 – 2^n$)

SORT N RANDOM DIGITS (1 – 9)

SUM N RANDOM DIGITS (1 – 9)

Behavior of SORT N RANDOM NUMBERS looks different. Its performance seems to accelerate, but at what rate?

Asymptotic Growth Defined By Family

		Linear		Quadratic			Exponential
n	**log(n)**	**n**	**n log(n)**	**n^2**	**n^3**	**n^4**	**2^n**
2	1	2	2	4	8	16	4
4	2	4	8	16	64	256	16
8	3	8	24	64	512	4096	256
16	4	16	64	256	4096	65536	65536
32	5	32	160	1024	32768	1048576	4.29E+09
64	6	64	384	4096	262144	16777216	1.84E+19
128	7	128	896	16384	2097152	2.68E+08	3.4E+38
256	8	256	2048	65536	16777216	4.29E+09	1.16E+77
512	9	512	4608	262144	1.34E+08	6.87E+10	1.3E+154
1024	10	1024	10240	1048576	1.07E+09	1.1E+12	∞
2048	11	2048	22528	4194304	8.59E+09	1.76E+13	∞
4096	12	4096	49152	16777216	6.87E+10	2.81E+14	∞
8192	13	8192	106496	67108864	5.5E+11	4.5E+15	∞
16384	14	16384	229376	2.68E+08	4.4E+12	7.21E+16	∞
32768	15	32768	491520	1.07E+09	3.52E+13	1.15E+18	∞

Performance Families

O(*1*)

O(*log n*)

O(*n*)

O(*n log(n)*)

O(n^2)

O(2^n)

Asymptotic Growth

- Classify algorithm by "closest" performance family
  - Informally, select *f(n)* that is best match
  - For example, SUM could be described as $O(2^n)$ since its behavior doesn't exceed this growth
  - However, you should always choose family that comes closest

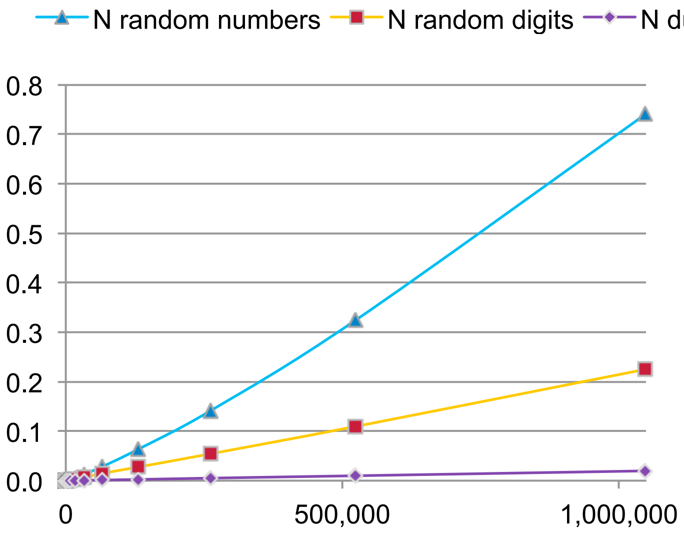
Problem Instances

- Best Case
  - Those problem instances that require <u>the least work</u>
- Worst Case
  - Those problem instances that require <u>the most work</u>

Problem Instances

- Average Case
  - Vast majority of remaining instances describe how the algorithm will perform “on average”
- Example: Sorting *n* numbers
  - Best Case: numbers are already sorted
  - Worst Case: numbers are in reverse order
  - Average Case: numbers are “randomly” distributed

Performance



SORT *n* RANDOM NUMBERS classified as O(*n log*(*n*))

- Best-case of O(*n*) when numbers already sorted
- Sorting *n* random digits is still classified O(*n log* (*n*))
  - It has a smaller constant *c* than performance of sorting *n* random numbers

Analysis Using Decomposition

Does X have a duplicate

```
def hasDuplicates(X):
 for i in range(len(X)-1):
   for j in range(i+1,len(X)):
     if X[i] == X[j]:
       return True
 return False
```

Nested **for** loop defines O(n^2) structure. **if** statement executes

$$\frac{n*(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$

times, resulting in O(n^2) because constant is ½. Ignore remaining as it is overpowered by the n^2 term

Analysis Using Composition

Does X have duplicate or None

```
def hasDuplicateOrNone(X):
 for i in range(len(X)):
  if X[i] == None:
   return True

 for i in range(len(X)-1):
  for j in range(i+1,len(X)):
   if X[i] == X[j]:
    return True
 return False
```

Two sequential code blocks may exhibit different O() behavior

- O() behavior of their composition is the more powerful function
- First **for** loop exhibits O(n) behavior
- Second block exhibits O(n^2)

Performance is classified as O(n^2) not O(n^2+n)

Summary
O(*f*(*n*)) is shorthand

- O(1) reflects fixed performance independent of problem size *n*
- O(*log n*) is extremely efficient
  - As problem size doubles, you only need a **fixed number of extra steps**

Summary
O($f(n)$) is shorthand

- O(n) is best polynomial efficiency
  - Performance **directly correlates** to size of problem instance
- O(n^2) and higher reflect a sobering reality
  - As problem size doubles, algorithm must work **four times as hard**
- O(2^n) is exponential time and reflects brute force

Summary
O(*f*(*n*)) is shorthand

- O(*n log* (*n*)) is the "sweet spot" for algorithms
  - Many naïve algorithms are O(n^2)
  - Using the right data structure and elegant design, these can be converted into more efficient O(*n log*(*n*))