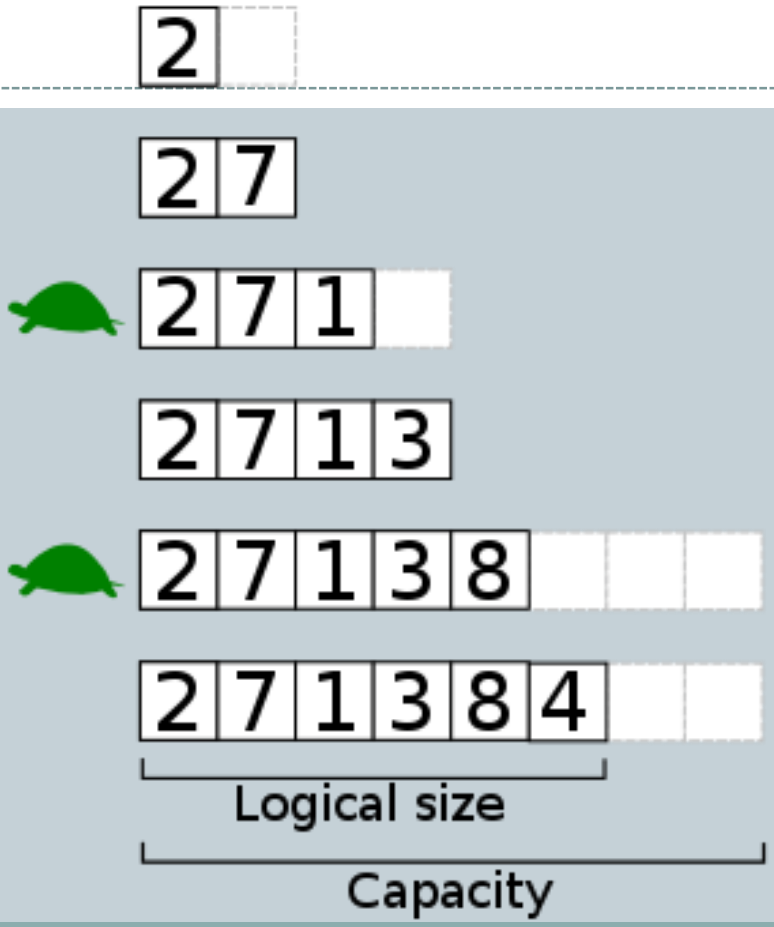
Dynamic Arrays / Performance



Outline

- Python Lists
  - Dynamically sized
  - Operations
  - Why is that a Problem?
- Performance

Operations on Lists

- Inserting at a Position
  - append – adds one item to end
  - insert
    - motorcycles.insert(0, 'ducati')
- Removing an Element
  - del motorcycles[0]
  - pop
    - motorcycles.pop()
    - motorcycles.pop(0)
  - Remove by value
    - motorcycles.remove('ducati')
- Sort
  - cars.sort()
  - sortedCars = sorted(cars)
- Searching
  - L.index(x)
  - L.count(x)

Under the Hood with Lists

- What if we need to add another element?
- What if we want to remove an element?
- What if we want to find out if a particular value is in the list?

Let's Write One...

```
class Duck:

    def __init__(self, name):
        self.name = name
        print("Quack\n")

    def toString(self):
        return self.name
```

Let's say we have a Duck class, and we want to create a list of Ducks named Daffy, Donald and Dead. The definition of the Duck class is shown above. We then want to remove the first Duck from the list and then remove the Dead one. At each point, we want to print out the list of Ducks.

The Challenge of Programming (One of Many?)

Q: Will my program be able to solve a large practical problem?

compile debug solve problems in practice

Key insight. [Knuth 1970s]
Use the scientific method to understand performance.

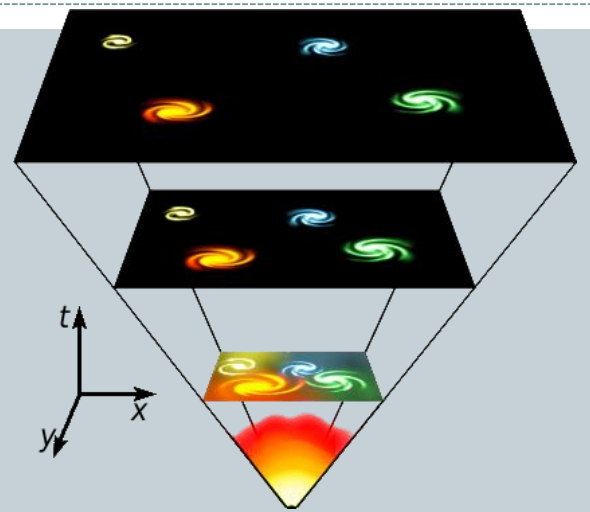
Scientific Method

- Scientific method
  - Observe some feature of the natural world
  - Hypothesize a model that is consistent with the observations
  - Predict events using the hypothesis
  - Verify the predictions by making further observations
  - Validate by repeating until hypothesis and observations agree
- Principles
  - Experiments must be reproducible
  - Hypotheses must be falsifiable

Hypothesis: *All swans are white*




Why Performance Analysis

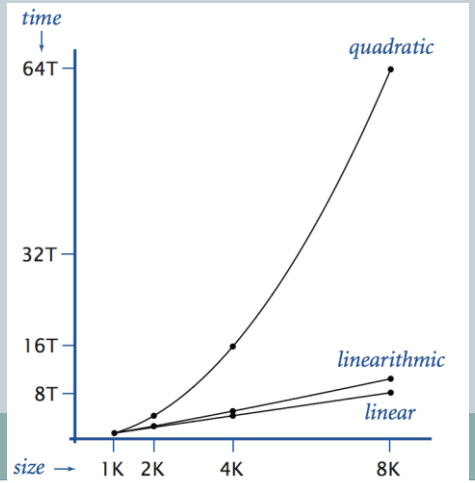


- Predicting performance
  - *When* will my program finish?
  - *Will* my program finish?
- Compare algorithms
  - Should I change to a more complicated algorithm?
  - Will it be worth the trouble?
- Basis for inventing new ways to solve problems
  - Enables new technology
  - Enables new research

Algorithmic Successes

John von Neumann
(1945)

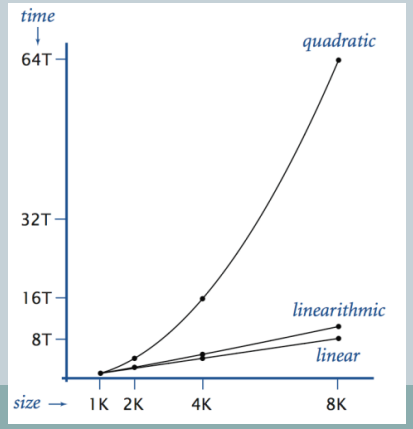
- Sorting
  - Rearrange array of N item in ascending order
  - Applications: databases, scheduling, statistics, genomics, …
  - Brute force: N^2 steps
  - Mergesort: $N \log N$ steps, enables new technology



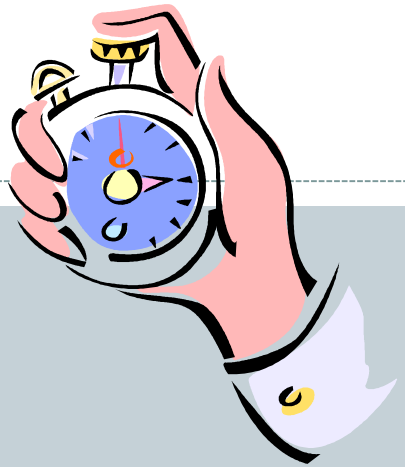
Algorithmic Successes

Freidrich Gauss (1805)

- Discrete Fourier transform
  - Break down waveform of N samples into periodic components
  - Applications: DVD, JPEG, MRI, astrophysics, ...
  - Brute force: N^2 steps
  - FFT algorithm: $N \log N$ steps, enables new technology



Performance Metrics



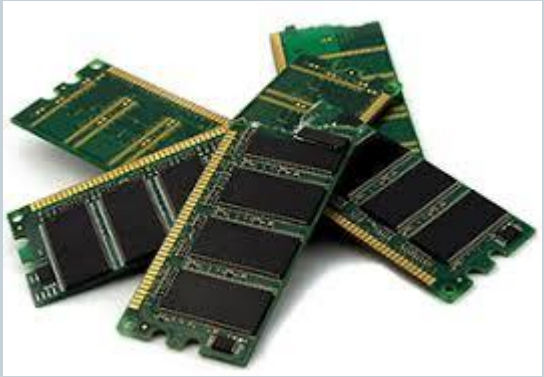
- What do we care about?
  - Time, how long do I have to wait?
    - Measure with a stop watch (real or virtual)
    - Run in a performance profiler
      - Often part of an IDE (e.g. Microsoft Visual Studio)
      - Sometimes standalone (e.g. gprof)
      - Helps you determine bottleneck in your code

```
import time

t1 = time.time()
# Put the code you want to time here
t2 = time.time()
print(t2-t1)
```

Measuring how long some code takes.

Performance Metrics

- What do we care about?
  - Space, do I have the resources to solve it?
    - Usually we care about physical memory
      - 8 GB = 8.6 billion places to store a byte (byte = 256 possibilities)
      - Python float, 64-bits = 8 bytes
      - 8 GB / 8 bytes = over 1 million floats!
    - Can swap to disk for some extra space
      - But much much slower



A “Simple" Problem

- Sum-Three problem
  - Given N integers, find all triples that sum to 0

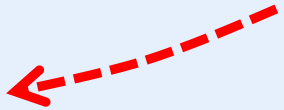
```
% type 8ints.txt
8
30 -30 -20 -10 40 0 10 5

% python SumThree.py 8ints.txt
30 -30 0
30 -20 -10
-30 -10 40
-10 0 10
```

Brute force algorithm:
Try all possible triples and see if they sum to 0.

Sum Three: Brute-Force

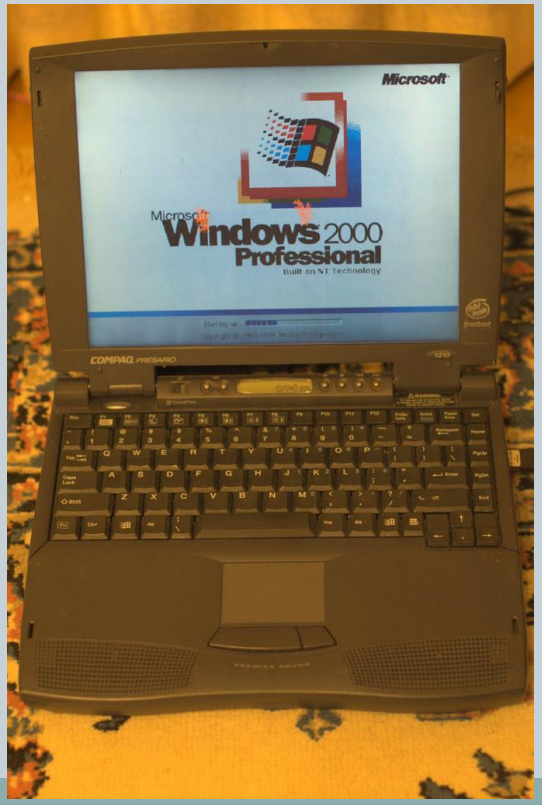
```
import sys

with open(sys.argv[1], 'r') as f:
    lines = f.read().split()
    count = int(lines[0])
    nums = []

    for i in range(1, count+1):
        nums.append(int(lines[i]))

    for i in range(0, len(nums)):
        for j in range(i + 1, len(nums)):
            for k in range(j + 1, len(nums)):
                if nums[i] + nums[j] + nums[k] == 0:
                    print(str(nums[i]) + " " + str(nums[j]) + " " + str(nums[k]))
```

All possible triples i < j < k from the set of integers.



Empirical Analysis: Sum Three

- Run program for various input sizes, 2 machines:
  - Keith's first laptop: Pentium 1, 150Mhz, 80MB RAM
  - Keith's desktop: Phenom II, 3.2Ghz (3.6Ghz turbo), 32GB RAM



VS.

Empirical Analysis: Sum Three

- Run program for various input sizes, 2 machines:
  - Keith's first laptop: Pentium 1, 150Mhz, 80MB RAM
  - Keith's desktop: Phenom II, 3.2Ghz (3.6Ghz turbo), 32GB RAM
  - My desktop: Intel i7, 3.6 GHz, 16GB RAM

N	ancient laptop	modern desktop	My desktop
100	0.33	0.01	0.004
200	2.04	0.04	0.008
400	11.23	0.16	0.021
800	94.96	0.63	0.061
1600	734.03	4.33	0.38
3200	5815.30	33.69	2.917
6400	47311.43	263.82	23.23

Doubling Hypothesis

- Cheap and cheerful analysis
  - Time program for input size N
  - Time program for input size 2N
  - Time program for input size 4N
  - …
  - Ratio T(2N) / T(N) approaches a constant
  - Constant tells you the exponent in $T = aN^b$

N	T(N)	ratio
400	0.16	-
800	0.63	3.94
1600	4.33	6.87
3200	33.69	7.78
6400	263.82	7.83

Keith's Desktop data

Constant from ratio	Hypothesis	Order of growth
2	$T = a\ N$	linear, O(N)
4	$T = a\ N^2$	quadratic, $O(N^2)$
8	$T = a\ N^3$	cubic, $O(N^3)$
16	$T = a\ N^4$	$O(N^4)$

Estimating Constant, Making Predictions

N	T(N)	ratio
400	0.16	-
800	0.63	3.94
1600	4.33	6.87
3200	33.69	7.78
6400	263.82	7.83

Keith's Desktop data

$T = a N^3$

$263.82 = a (6400)^3$

$a = 1.01 \times 10^{-09}$

N	T(N)	ratio
400	11.23	-
800	94.96	8.45
1600	734.03	7.72
3200	5815.30	7.92
6400	47311.43	8.14

Keith's Laptop data

$T = a N^3$

$47311.43 = a (6400)^3$

$a = 1.80 \times 10^{-07}$

Prediction:

How long for desktop to solve a 100,000 integer problem?

$1.01 \times 10^{-09} (100000)^3 = 1006393$ secs

$= 280$ hours

Prediction:

How long for laptop to solve a 100,000 integer problem?

$1.80 \times 10^{-07} (100000)^3 = 1.80 \times 10^{08}$ secs

$= 50133$ hours

Bottom Line

- My sum three algorithm sucks
  - Does not scale to large problems → an algorithm problem
  - 15 years of computer progress didn't help much
  - My algorithm: $O(N^3)$
  - A slightly more complicated algorithm: $O(N^2 \log N)$

Using the better algorithm, how long does it take the modern desktop to solve a 100,000 integer problem?

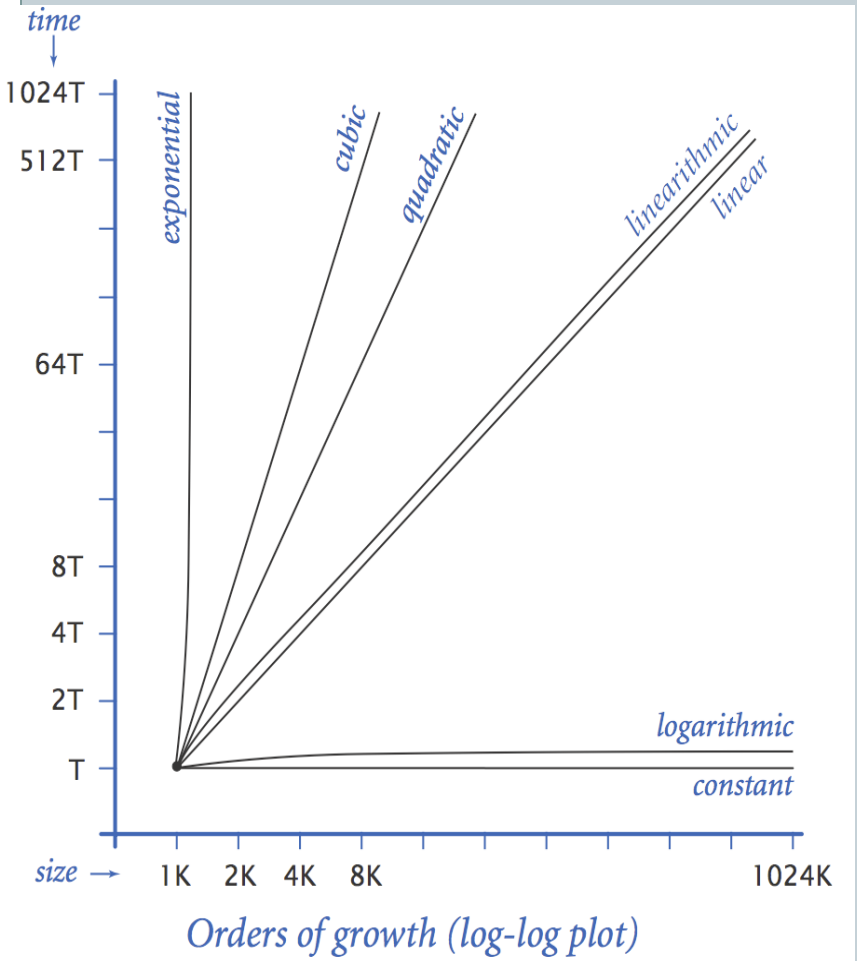
$1.01 \times 10^{-09} (100000)^2 \log(100000) = 168$ secs

Using the better algorithm, how long does it take the ancient laptop to solve a 100,000 integer problem?

$1.80 \times 10^{-07} (100000)^2 \log(100000) = 29897$ secs

This assumes the same constant. Really should do the doubling experiment again with the new algorithm.

Order of Growth



Orders of growth (log-log plot)

Doubling hypothesis ratio	Hypothesis	Order of growth
1	$T = a$	constant, $O(1)$
1	$T = a \log N$	logarithmic, $O(\log N)$
2	$T = a N$	linear, $O(N)$
2	$T = a N \log N$	linearithmic, $O(N \log N)$
4	$T = a N^2$	quadratic, $O(N^2)$
8	$T = a N^3$	cubic, $O(N^3)$
2^N	$T = a 2^N$	exponential, $O(2^N)$

Order of Growth: Consequences

order of growth	*predicted running time if problem size is increased by a factor of 100*
linear	a few minutes
linearithmic	a few minutes
quadratic	several hours
cubic	a few weeks
exponential	forever

Effect of increasing problem size for a program that runs for a few seconds

Order of Growth

A small number of functions describe the running time of many fundamental algorithms!

```
while N > 1:
    N = N / 2
    ...
```

log N

```
for i in range(0, N):
    ...
```

N

```
for i in range(0, N):
    for j in range(0, N):
        ...
```

N^2

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            ...
```

N^3

```
def g(N):
    if N == 0:
        return
    g(N / 2)
    g(N / 2)
    for i in range(0, N):
        ...
```

N log N

```
def f(N):
    if N == 0:
        return
    f(N - 1)
    f(N - 1)
    ...
```

2^N

Growth of Nested Loops

- Nested loops
  - A good clue to order of growth
  - But each loop must execute "on the order of" N times
  - If loop not a linear function of N, loop doesn't cause order to grow

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            count += 1
```

N^3

N	T(N)	ratio
5000	6.85	-
10000	53.48	7.8
20000	425.97	8.0

$425.97 = a\ (20000^3)$

$a = 1.06 \times 10^{-6}$

```
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, 10000):
            count += 1
```

N^2

N	T(N)	ratio
5000	13.40	-
10000	53.20	3.97
20000	212.49	3.99

$212.49 = a\ (20000^2)$

$a = 5.31 \times 10^{-7}$

```
for i in range(0, N):
   for j in range(0, N):
      for k in range(0, N):
         count += 1
```

N^3

N	T(N)	ratio
5000	6.85	-
10000	53.48	7.8
20000	425.97	8.0

$425.97 = a\ (20000^3)$

$a = 1.06 \times 10^{-6}$

```
for i in range(0, N):
   for j in range(0, N):
      for k in range(0, 10000):
         count += 1
```

N^2

N	T(N)	ratio
5000	13.40	-
10000	53.20	3.97
20000	212.49	3.99

$212.49 = a\ (20000^2)$

$a = 5.31 \times 10^{-7}$

```
for i in range(0, N):
   for j in range(0, N):
      for k in range(0, N / 5):
         count += 1
```

N^3

N	T(N)	ratio
5000	1.59	-
10000	11.08	6.97
20000	86.36	7.79

$86.36 = a\ (20000^3)$

$a = 2.16 \times 10^{-7}$

```
for i in range(0, N):
   for j in range(0, N):
      for k in range(0, 10):
         count += 1
```

N^2

N	T(N)	ratio
5000	0.11	-
10000	0.37	3.36
20000	1.47	3.97

$1.47 = a\ (20000^2)$

$a = 3.68 \times 10^{-9}$

Recap on Performance

- Introduction to Analysis of Algorithms
  - Today: simple empirical estimation
  - Next year: an entire semester course
- The algorithm matters
  - Faster computer only buys you out of trouble temporarily
  - Better algorithms enable new technology!
- The data structure matters
- Doubling hypothesis
  - Measure time ratio as you double the input size
  - If the ratio = 2^b, runtime of algorithm $T(N) = a N^b$

Summary

- Python Lists
  - Dynamically sized
  - Operations
  - Why is that a Problem?
- Performance