Dynamic Arrays / Performance

![Image of Slinky toy and boy holding it]

- **Logical size**: The size of the data stored in the array.
- **Capacity**: The total size of the array, which can be greater than the logical size.

Given an initial shape, "Slinky" easily and deliberately collapses and rears up and down a flight of stairs. It is simply a spring, but it does what that made E. F. Jones, Philadelphia engineer, think of converting it into a toy.

"Slinky" goes places when, with a flash of the wrist, it leaves one into the laughing room. Much of its fame depends on the name it's been given.

Produced in very small quantity in Philadelphia, 100,000 of the springs sold so exactly that the moment it is tilted, it keeps its shape and remains the same in a number of cities.

---

*Fundamentals of Computer Science*
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 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?

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

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

John von Neumann (1945)
Algorithmic Successes

- **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

```python
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.
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
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

<table>
<thead>
<tr>
<th>N</th>
<th>ancient laptop</th>
<th>modern desktop</th>
<th>My desktop</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0.33</td>
<td>0.01</td>
<td>0.004</td>
</tr>
<tr>
<td>200</td>
<td>2.04</td>
<td>0.04</td>
<td>0.008</td>
</tr>
<tr>
<td>400</td>
<td>11.23</td>
<td>0.16</td>
<td>0.021</td>
</tr>
<tr>
<td>800</td>
<td>94.96</td>
<td>0.63</td>
<td>0.061</td>
</tr>
<tr>
<td>1600</td>
<td>734.03</td>
<td>4.33</td>
<td>0.38</td>
</tr>
<tr>
<td>3200</td>
<td>5815.30</td>
<td>33.69</td>
<td>2.917</td>
</tr>
<tr>
<td>6400</td>
<td>47311.43</td>
<td>263.82</td>
<td>23.23</td>
</tr>
</tbody>
</table>
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 $T = aN^{b}$

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>0.16</td>
<td>-</td>
</tr>
<tr>
<td>800</td>
<td>0.63</td>
<td>3.94</td>
</tr>
<tr>
<td>1600</td>
<td>4.33</td>
<td>6.87</td>
</tr>
<tr>
<td>3200</td>
<td>33.69</td>
<td>7.78</td>
</tr>
<tr>
<td>6400</td>
<td>263.82</td>
<td>7.83</td>
</tr>
</tbody>
</table>

Keith’s Desktop data

<table>
<thead>
<tr>
<th>Constant from ratio</th>
<th>Hypothesis</th>
<th>Order of growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$T = aN$</td>
<td>linear, $O(N)$</td>
</tr>
<tr>
<td>4</td>
<td>$T = aN^2$</td>
<td>quadratic, $O(N^2)$</td>
</tr>
<tr>
<td>8</td>
<td>$T = aN^3$</td>
<td>cubic, $O(N^3)$</td>
</tr>
<tr>
<td>16</td>
<td>$T = aN^4$</td>
<td>$O(N^4)$</td>
</tr>
</tbody>
</table>
## Estimating Constant, Making Predictions

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>0.16</td>
<td>-</td>
</tr>
<tr>
<td>800</td>
<td>0.63</td>
<td>3.94</td>
</tr>
<tr>
<td>1600</td>
<td>4.33</td>
<td>6.87</td>
</tr>
<tr>
<td>3200</td>
<td>33.69</td>
<td>7.78</td>
</tr>
<tr>
<td>6400</td>
<td>263.82</td>
<td>7.83</td>
</tr>
</tbody>
</table>

Keith’s Desktop data

\[ T = a N^3 \]

\[ 263.82 = a (6400)^3 \]

\[ a = 1.01 \times 10^{-09} \]

**Prediction:**

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

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

= 280 hours

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>11.23</td>
<td>-</td>
</tr>
<tr>
<td>800</td>
<td>94.96</td>
<td>8.45</td>
</tr>
<tr>
<td>1600</td>
<td>734.03</td>
<td>7.72</td>
</tr>
<tr>
<td>3200</td>
<td>5815.30</td>
<td>7.92</td>
</tr>
<tr>
<td>6400</td>
<td>47311.43</td>
<td>8.14</td>
</tr>
</tbody>
</table>

Keith’s Laptop data

\[ T = a N^3 \]

\[ 47311.43 = a (6400)^3 \]

\[ a = 1.80 \times 10^{-07} \]

**Prediction:**

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

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

= 50133 hours
• **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?

$$1.01 \times 10^{-9} (100000)^2 \log(100000) = 168 \text{ secs}$$

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

$$1.80 \times 10^{-7} (100000)^2 \log(100000) = 29897 \text{ secs}$$

This assumes the same constant. Really should do the doubling experiment again with the new algorithm.
## Order of Growth

<table>
<thead>
<tr>
<th>Doubling hypothesis ratio</th>
<th>Hypothesis</th>
<th>Order of growth</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$T = a$</td>
<td>constant, $O(1)$</td>
</tr>
<tr>
<td>1</td>
<td>$T = a \log N$</td>
<td>logarithmic, $O(\log N)$</td>
</tr>
<tr>
<td>2</td>
<td>$T = a N$</td>
<td>linear, $O(N)$</td>
</tr>
<tr>
<td>2</td>
<td>$T = a N \log N$</td>
<td>linearithmic, $O(N \log N)$</td>
</tr>
<tr>
<td>4</td>
<td>$T = a N^2$</td>
<td>quadratic, $O(N^2)$</td>
</tr>
<tr>
<td>8</td>
<td>$T = a N^3$</td>
<td>cubic, $O(N^3)$</td>
</tr>
<tr>
<td>$2^N$</td>
<td>$T = a 2^N$</td>
<td>exponential, $O(2^N)$</td>
</tr>
</tbody>
</table>
### Order of Growth: Consequences

<table>
<thead>
<tr>
<th>order of growth</th>
<th>predicted running time if problem size is increased by a factor of 100</th>
</tr>
</thead>
<tbody>
<tr>
<td>linear</td>
<td>a few minutes</td>
</tr>
<tr>
<td>linearithmic</td>
<td>a few minutes</td>
</tr>
<tr>
<td>quadratic</td>
<td>several hours</td>
</tr>
<tr>
<td>cubic</td>
<td>a few weeks</td>
</tr>
<tr>
<td>exponential</td>
<td>forever</td>
</tr>
</tbody>
</table>

*Effect of increasing problem size for a program that runs for a few seconds*
A small number of functions describe the running time of many fundamental algorithms!

- \( \log N \) for
  ```
  while N > 1:
    N = N / 2
    ...
  ```

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

- \( N^2 \) for
  ```
  for i in range(0, N):
    for j in range(0, N):
      for k in range(0, N):
        ...
  ```

- \( N^3 \) for
  ```
  for i in range(0, N):
    for j in range(0, N):
      for k in range(0, N):
        ...
  ```

- \( 2^N \) for
  ```
  def f(N):
    if N == 0:
      return
    f(N - 1)
    f(N - 1)
    ...
  ```

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

  \[
  \text{for } i \in \text{range}(0, N): \\
  \quad \text{for } j \in \text{range}(0, N): \\
  \quad \quad \text{for } k \in \text{range}(0, N): \\
  \quad \quad \quad \text{count } += 1
  \]

  \[
  \text{for } i \in \text{range}(0, N): \\
  \quad \text{for } j \in \text{range}(0, N): \\
  \quad \quad \text{for } k \in \text{range}(0, 10000): \\
  \quad \quad \quad \text{count } += 1
  \]

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>6.85</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>53.48</td>
<td>7.8</td>
</tr>
<tr>
<td>20000</td>
<td>425.97</td>
<td>8.0</td>
</tr>
</tbody>
</table>

\[425.97 = a (20000^3)\]
\[a = 1.06 \times 10^{-6}\]

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>13.40</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>53.20</td>
<td>3.97</td>
</tr>
<tr>
<td>20000</td>
<td>212.49</td>
<td>3.99</td>
</tr>
</tbody>
</table>

\[212.49 = a (20000^2)\]
\[a = 5.31 \times 10^{-7}\]
for i in range(0, N):
    for j in range(0, N):
        for k in range(0, N):
            count += 1

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

N³

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>6.85</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>53.48</td>
<td>7.8</td>
</tr>
<tr>
<td>20000</td>
<td>425.97</td>
<td>8.0</td>
</tr>
</tbody>
</table>

425.97 = a \times (20000³)

a = 1.06 \times 10^{-6}

N²

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>13.40</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>53.20</td>
<td>3.97</td>
</tr>
<tr>
<td>20000</td>
<td>212.49</td>
<td>3.99</td>
</tr>
</tbody>
</table>

212.49 = a \times (20000²)

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

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

N³

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>1.59</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>11.08</td>
<td>6.97</td>
</tr>
<tr>
<td>20000</td>
<td>86.36</td>
<td>7.79</td>
</tr>
</tbody>
</table>

86.36 = a \times (20000³)

a = 2.16 \times 10^{-7}

N²

<table>
<thead>
<tr>
<th>N</th>
<th>T(N)</th>
<th>ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>5000</td>
<td>0.11</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>0.37</td>
<td>3.36</td>
</tr>
<tr>
<td>20000</td>
<td>1.47</td>
<td>3.97</td>
</tr>
</tbody>
</table>

1.47 = a \times (20000²)

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 $T(N) = a \cdot N^b$
Summary

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