Bot With Boyish Personality Wins Biggest Turing Test

Eugene Goostman, a chatbot with the personality of a 13-year-old boy, won the biggest Turing test ever staged, on 23 June.

*Turing test*: Measure of machine intelligence proposed by Alan Turing. A human talks via a text interface to either a bot or a human: the human has to determine which (s)he is talking to.

Turing suggested that if a machine could fool the human 30% of the time, it passed the test. Eugene passed 29% of the time.

Eugene was programmed to have a “consistent and specific” personality.

TODAY

• Time complexity of functions.
• Recursion review.
**Problem Solving: Algorithms**

An *algorithm* is a step-by-step description of how to perform a certain task.

For example, how do we bake a cake?

*Step 1:* Buy cake mix, eggs, water, and oil.

*Step 2:* Add the cake mix to a mixing bowl.

... and so on.
**Problem Solving: Algorithms**

The functions we write in Python *implement* algorithms for computational problems.

For a lot of problems, there are *many different* algorithms to find a solution.

How do we know which algorithm is *better*?
COMPARISON OF ALGORITHMS

How do we know which algorithm (and the function that implements it) is better?

- Amount of time taken.
- Size of the code.
- Amount of non-code space used.
- Precision of the solution.
- Ease of implementation.

... among other metrics.

We will focus on this.
COMPARISON OF ALGORITHMS
Which function is better?
Which function takes lesser time?

def fib(n):
    if n == 0:
        return 0
    prev, curr, k = 0, 1, 1
    while k < n:
        prev, curr = curr, curr + prev
        k = k + 1
    return curr

def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)

FIGHT!!
COMPARISON OF ALGORITHMS

The *iterative* version of *fib* is quicker than the (naïve) *recursive* version of *fib*.

Difference is only visible for *larger* inputs.

*Idea:*

Measure the *runtime* of a function for *large* inputs. Computers are already quick for small inputs.

In module 2, we will see that we can make the two equally efficient.
RUNTIME ANALYSIS

How do we measure the *runtime* of a function? *Simplest way*: Measure with a stopwatch.

Is this the *best* way?
RUNTIME ANALYSIS

Measuring raw runtime depends on many factors:

• Different computers can have different runtimes.
• Same computer can have different runtimes on the same input.
  \[\text{Other processes can be running at the same time.}\]
• Algorithm needs to be implemented first!
  \[\text{Can be tricky to get right.}\]
• Function can take prohibitively long time to run.
RUNTIME ANALYSIS

Problem:
How do we *abstract* the computer away?
Can we compare runtimes *without* implementing the algorithms?
RUNTIME ANALYSIS: BIG DATA

Humans are producing a *lot* of data *really quickly.*

http://www.computerworld.com/s/article/9217988/World_s_data_will_grow_by_50X_in_next_decade_IDC_study_predicts
Runtime Analysis

Big Idea:
Determine how the worst-case runtime of an algorithm scales as we scale the input.

The less the runtime scales as the input scales, the better the algorithm.
It can handle more data quicker.
ANNOUNCEMENTS

• Waitlist is cleared. If you’re still on the waitlist by the end of this week, please let us know!
• Next week, we will move to **105 Stanley** for the rest of the summer.
• Midterm 1 is on **July 9**.
  – We will have a review session closer to the date.
• If you need accommodations for the midterm, please notify DSP by the end of this week.
• HW1 grade should be available on glookup.
BEWARE: APPROXIMATIONS AHEAD

http://www.gamesdash.com/limg/1/283/beware-of-the-sign.jpg
ORDERS OF GROWTH

```python
def add_one(n):
    return n + 1

def mul_64(n):
    return n * 64

def square(n):
    return n * n
```

Time taken by these functions is roughly *independent* of the input size.

These functions run in *constant time*.
ORDERS OF GROWTH

```python
def add_one(n):
    return n + 1

def mul_64(n):
    return n * 64

def square(n):
    return n * n
```

Approximation: Arithmetic operations and assignments take constant time.
**ORDERS OF GROWTH**

def fact(n):
    k, prod = 1, 1
    while k <= n:
        prod = prod * k
        k = k + 1
    return prod

Constant-time operations

This loop runs $n$ times.

Constant-time operations

Total time for all operations is proportional to $n$. 
def fact(n):
    k, prod = 1, 1
    while k <= n:
        prod = prod * k
        k = k + 1
    return prod

ORDERS OF GROWTH

Time taken by this function scales roughly \textit{linearly} as the input size scales.

This function runs in \textit{linear time}.
def sum_facts(n):
    '''Adds factorials of integers from 1 to n.'''
    sum, k = 0, 1
    while k <= n:
        sum += fact(k)
        k = k + 1
    return sum

This loop runs \( n \) times.

For the \( k \)th loop, \texttt{fact} runs in time proportional to \( k \).
Time taken by `sum_facts` is proportional to

\[
\begin{align*}
1 + 2 + \ldots + n + an + b &= \frac{1}{2} n(n + 1) + an + b \\
&= \frac{1}{2} n^2 + (a + 1/2)n + b
\end{align*}
\]
ORDERS OF GROWTH

The constants $a$ and $b$ do not actually matter.

For really large values of $n$, $n^2$ suppresses $n$.

<table>
<thead>
<tr>
<th>$n$</th>
<th>1</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>10000</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n^2$</td>
<td>1</td>
<td>100</td>
<td>1000</td>
<td>10000</td>
<td>100000</td>
</tr>
</tbody>
</table>

For really large values of $n$,

$$\frac{1}{2}n^2 + (a + 1/2)n + b \approx \frac{1}{2}n^2.$$
ORDERS OF GROWTH

One more approximation:
We only care about how the runtime scales as the input size scales, so the constant factor is irrelevant.

\[ \frac{1}{2} n^2 \] scales similarly to \( n^2 \).

For example, if the input size doubles, both functions quadruple.
 ORDERS OF GROWTH

def sum_facts(n):
    """Adds factorials of integers from 1 to n."""
    sum, k = 0, 1
    while k <= n:
        sum += fact(k)
        k = k + 1
    return sum

Time taken by this function scales roughly quadratically as the input size scales.

This function runs in quadratic time.
ORDERS OF GROWTH

A few important observations:

1. We only care about really large input values, since computers can deal with small values really quickly.
2. We can ignore any constant factors in front of polynomial terms, since we want to know how the runtime scales.
3. We care about the worst-case runtime. If the function can be linear on some inputs and quadratic on other inputs, it runs in quadratic time overall. This can happen if your code has an if statement, for example.

How do we communicate the worst-case asymptotic runtime to other computer scientists?
**BIG-O NOTATION**

Let $f(n)$ be the runtime of a function. It depends on the input size $n$.

We can then say

$$f(n) \in O(g(n))$$

“Belongs to”

*Set of functions*
BIG-O NOTATION

\[ f(n) \in O(g(n)) \]

if there are two integers \( c, N \) such that

\[ f(n) < c \cdot g(n) \]

for all \( n > N \).

Intuitively, \( g(n) \) is an “upper bound” on \( f(n) \).

“For large input values”

“Ignore the constant factors”
BIG-O NOTATION: EXAMPLE

Once \( n^2 \) surpasses \( n \) after \( N \), \( n \) can never catch up to \( n^2 \), and \( n^2 \) grows much faster.
**Big-O Notation: Example**

\[ n \in O(n^2) \]

if there are two integers \( c, N \) such that

for all \( n \geq 1 \),

\[ f(n) < c \cdot g(n). \]
**Big-O Notation**

In this class, we are not going to worry about finding the values of $c$ and $N$.

We would like you to get a basic intuition for how the function behaves for *large inputs*.

CS61B, CS70 and CS170 will cover this topic in much more detail.
Remember:
Constant factors do not matter.
Larger powered polynomial terms suppress smaller powered polynomial terms.
We care about the worst-case runtime.
**Big-O Notation**

Constant factors do not matter.

<table>
<thead>
<tr>
<th>Size of input ($N$)</th>
<th>$t_1(n) = 3n^3$</th>
<th>$t_2(n) = 19,500,000n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.10 microseconds</td>
<td>200 milliseconds</td>
</tr>
<tr>
<td>100</td>
<td>3.0 milliseconds</td>
<td>2.0 seconds</td>
</tr>
<tr>
<td>1000</td>
<td>3.0 seconds</td>
<td>20 seconds</td>
</tr>
<tr>
<td>10000</td>
<td>49 minutes</td>
<td>3.2 minutes</td>
</tr>
<tr>
<td>100000</td>
<td>35 days (est.)</td>
<td>32 minutes</td>
</tr>
<tr>
<td>1000000</td>
<td>95 years (est.)</td>
<td>5.4 hours</td>
</tr>
</tbody>
</table>

Jon Bentley ran two different programs to solve the same problem. The cubic algorithm was run on a Cray supercomputer, while the linear algorithm was run on a Radio Shack microcomputer. The microcomputer beat out the super computer for large $n$.

From *Programming Pearls* (Addison-Wesley, 1986)
BIG-O NOTATION

Which of these are correct?

• \( \frac{1}{2} n^2 \in O(n^2) \)
• \( n^2 \in O(n) \)
• \( 15000n + 3 \in O(n) \)
• \( 5n^2 + 6n + 3 \in O(n^2) \)
BIG-O NOTATION

Which of these are correct?

- \( \frac{1}{2}n^2 \in O(n^2) \) **Correct**
- \( n^2 \in O(n) \) **Incorrect**
- \( 15000n + 3 \in O(n) \) **Correct**
- \( 5n^2 + 6n + 3 \in O(n^2) \) **Correct**
How does this relate to **asymptotic** runtime?

If a function runs in **constant time**, its runtime is in $O(1)$.  
(“Its runtime is bounded above by a constant multiple of 1.”)

If a function runs in **linear time**, its runtime is in $O(n)$.  
(“Its runtime is bounded above by a constant multiple of $n$.”) 

If a function runs in **quadratic time**, its runtime is in $O(n^2)$.  
(“Its runtime is bounded above by a constant multiple of $n^2$.”)
## Common Runtimes

<table>
<thead>
<tr>
<th>Class of Functions</th>
<th>Common Name</th>
<th>Commonly found in</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td>Constant</td>
<td>Searching and arithmetic</td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td>Logarithmic</td>
<td>Searching</td>
</tr>
<tr>
<td>$O(\sqrt{n})$</td>
<td>Root-$n$</td>
<td>Primality checks</td>
</tr>
<tr>
<td>$O(n)$</td>
<td>Linear</td>
<td>Searching, sorting</td>
</tr>
<tr>
<td>$O(n \log n)$</td>
<td>Linearithmic/loglinear</td>
<td>Sorting</td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td>Quadratic</td>
<td>Sorting</td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td>Cubic</td>
<td>Matrix multiplication</td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td>Exponential</td>
<td>Enumeration</td>
</tr>
</tbody>
</table>

There are many problems for which the worst-case runtime is exponential. There has yet been no proof that these problems have polynomial solutions, and there has been no proof that a polynomial solution does not exist. One example is the problem of finding the shortest tour through a set of cities.
Generally, “efficient” code is code that has a polynomial asymptotic runtime. The lower the power on the polynomial, the better.
We defined earlier $f(n) \in O(g(n))$

$g(n)$ is an **upper bound** on $f(n)$.

If $f(n) = O(g(n))$, then $g(n) \in \Omega(f(n))$.

$f(n)$ is a **lower bound** on $g(n)$.

If $f(n) = O(g(n))$ and $g(n) = O(f(n))$, then

$f(n) = \Theta(g(n))$.

$g(n)$ is a **tight bound** on $f(n)$.
def sum1(n):
    ''' Adds all numbers from 1 to n. '''
    sum, k = 0, 1
    while k <= n:
        sum += k
        k += 1
    return sum

def sum2(n):
    ''' Adds all numbers from 1 to n. '''
    return (n * (n+1))/2
WHICH ALGORITHM IS BETTER?

def sum1(n):
    ''' Adds all numbers from 1 to n. '''
    sum, k = 0, 1
    while k <= n:
        sum += k
        k += 1
    return sum

#The second one is better
def sum2(n):
    ''' Adds all numbers from 1 to n. '''
    return (n * (n+1))/2
CONCLUSION

• One measure of efficiency of an algorithm and the function that implements it is to measure its runtime.
• In asymptotic runtime analysis, we determine how the runtime of a program scales as the size of the input scales.
• Big-O, Big-Omega and Big-Theta notations are used to establish relationships between functions for large input sizes.
• Preview: The other computational player: data.