CS61A Lecture 7
Complexity and Orders of Growth
Jon Kotker and Tom Magrino
UC Berkeley EECS
June 27, 2012

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.

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.
COMPARISON OF ALGORITHMS
Which function is better?
Which function takes lesser time?

The iterative version of fib is quicker than the (naive) 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.

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.
  Other processes can be running at the same time.
• Algorithm needs to be implemented first!
  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.

World’s data will grow by 50X in next decade, IDC study predicts

Computeworl: In 2011 alone, 1.8 zettabytes (or 1.8 trillion gigabytes) of data will be created, the equivalent to every U.S. citizen writing 3 books per minute for 26,876 years. And over the next decade, the number of servers managing the world’s data stores will grow by ten times.
**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**

![CAUTION: THIS SIGN HAS SHARP EDGES
DO NOT TOUCH THE EDGES OF THIS SIGN](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 fact(n):
    k, prod = 1, 1
    while k <= n:
        prod *= k
        k += 1
    return prod
```

**Approximation:**
Arithmetic operations and assignments take 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
```

This loop runs n times.

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

ORDERS OF GROWTH

Time taken by \texttt{sum\_facts} is proportional to\n
\[
\sum_{k=1}^{n} (an + b) = \frac{1}{2} n \cdot (n + 1) + an + b
\]

Time taken by \texttt{sum\_facts} is proportional to\n
\[
\frac{1}{2} n^2 + (a + 1/2)n + b
\]

ORDERS OF GROWTH

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

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

For example, if the input size doubles, both functions \textit{quadruple}.
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 $f(n) \in O(g(n))$

Set of functions

BIG-O NOTATION: EXAMPLE

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.
**BIG-O NOTATION**

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**

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**

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>$O(n)$</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.
COMMOM RUNTINES

Generally, “efficient” code is code that has a polynomial asymptotic runtime. The lower the power on the polynomial, the better.

BIG-THETA AND BIG-OMEGA NOTATION

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 $f(n)$.

WHICH ALGORITHM IS BETTER?

```python
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
```

# The second one is better

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.