Algorithms and their Performance

Analysis of Algorithms

Joshua Knowles

School of Computer Science

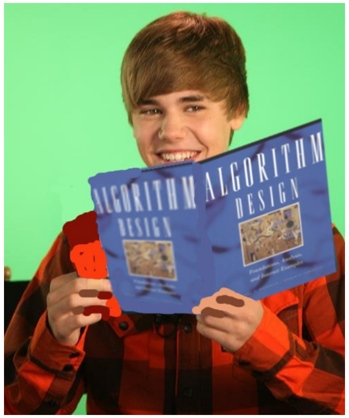
The University of Manchester

COMP26120 Semester 1, Week 7 LT 1.1, November 8th 2013

Reading Week Photos

Keep reading !

Reading Week Photos



ALGORITHM
DESIGN
OXYGEN SUPPLY
09:54
O_2
Pocketlint

This Lecture

Last time: Big-O and growth rates

$$\log n \quad \sqrt{n} \quad n \quad n^2 \quad 2^n \quad n!$$

This lecture:

- analysis of some algorithms

- more practice with Big-O

- experimentation: measuring running time

Analysis of Algorithms

Two methods:

1. Inspect the pseudocode
2. Time an implementation of the algorithm

Analysis of Algorithms

Two methods:

1. Inspect the pseudocode ***Analytical/theoretical approach***

2. Time an implementation of the algorithm ***Experimental/empirical approach***

Inspection — The Analytical Approach

is usually preferred because

- not affected by computer hardware
- not affected by software / programming language choice
- not affected by programming ‘tricks’
- yields a ***proof*** of the running time for inputs of ***any size*** n

Primitive Operations

When inspecting pseudocode, we count ***primitive operations***.

Eg, the number of

- memory accesses
- additions, multiplications, other arithmetic operations
- comparisons

We may count all these, or choose one or two. Should justify this choice.

These operations should all be $O(1)$, ie constant time: not dependent on input size.

Analysis Recipe

1. Obtain the pseudocode (and prove or assume it is ***correct***)
2. Look for the main operations and choose which to count
3. Identify what the (worst-case) input is
4. Count the operations, assuming the input size is n
5. Write down the equation $t(n) = \ldots$
6. **Simplify the equation using Big-Oh**

Often, steps 4–6 can be treated as one step

Example: Find Duplicates

```
Algorithm Find Duplicates:
1:  input: list $x$ of $n$ names (unsorted)
2:  for $i \leftarrow 1$ to $n$ do
3:      $mark[i] \leftarrow$ “”
4:  for $i \leftarrow 1$ to $n$ do
5:      $c \leftarrow x[i]$
6:      for $j \leftarrow i + 1$ to $n$ do
7:          if $x[j] = c$
8:              $mark[j] \leftarrow$ “ $*$ ”
9:  for $i \leftarrow 1$ to $n$ do
10:     print $x[i]$, $mark[i]$
```

List:
Sarah
John
Sarah
William
Adrian
Sarah
John
.
.
.

What is the time complexity?

Example: Find Duplicates

```
Algorithm Find Duplicates:
1:  input: list $x$ of $n$ names (unsorted)
2:  for $i \leftarrow 1$ to $n$ do
3:        $mark[i] \leftarrow$ “”
4:  for $i \leftarrow 1$ to $n$ do
5:        $c \leftarrow x[i]$
6:        for $j \leftarrow i+1$ to $n$ do
7:            if $x[j] = c$
8:                $mark[j] \leftarrow$ “ $*$ ”
9:  for $i \leftarrow 1$ to $n$ do
10:       print $x[i]$, $mark[i]$
```

List:
Sarah
John
Sarah*
William
Adrian
Sarah*
John*
.
.
.

What is the time complexity?

Example: Find Duplicates

Input size is number of names n

Let's count ***assignments*** and ***comparisons***

```
Algorithm Find Duplicates:
1: input: list $x$ of $n$ names (unsorted)
2: for $i \leftarrow 1$ to $n$ do
3:     $mark[i] \leftarrow$ “”
4: for $i \leftarrow 1$ to $n$ do
5:     for $j \leftarrow i + 1$ to $n$ do
6:         if $x[j] = x[i]$
7:             $mark[j] \leftarrow$ “ $*$ ”
8: for $i \leftarrow 1$ to $n$ do
9:     print $x[i]$, $mark[i]$
```

Worst case: whole list just one name duplicated: line 6 will always evaluate TRUE and line 7 will always execute.

Example: Find Duplicates

```
Algorithm Find Duplicates:
1: input: list $x$ of $n$ names (unsorted)
2: for $i \leftarrow 1$ to $n$ do                 $n + 1$ comparisons
3:     $mark[j] \leftarrow$ ""                     $n$ assignments
4: for $i \leftarrow 1$ to $n$ do                 $n + 1$ comparisons
5:     for $j \leftarrow i + 1$ to $n$ do          $n - 1 + n - 2 + \ldots + 2$ comp'ns
6:         if $x[j] = x[i]$                        $n - 1 + n - 2 + \ldots + 1$ comp'ns
7:             $mark[j] \leftarrow$ "$*$"          $n - 1 + n - 2 + \ldots + 1$ ass'nts
8: for $i \leftarrow 1$ to $n$ do                 $n + 1$ comparisons
9:     print $x[j]$, $mark[j]$
```

For lines 5–7, use the arithmetic progression:

$$\sum_{x=1}^{m} x = m(m+1)/2$$

and substitute in $n - 1$ for m. Then, the total number of operations

$$= 4n + 3 + 3.(n-1)(n)/2 + 1$$

$= (3/2)n^2 + (5/2)n + 4$ and this is $O(n^2)$

Knowing about big-O, could you shortcut this analysis?

Challenge Question

Did anyone solve the prize challenge question from last time?

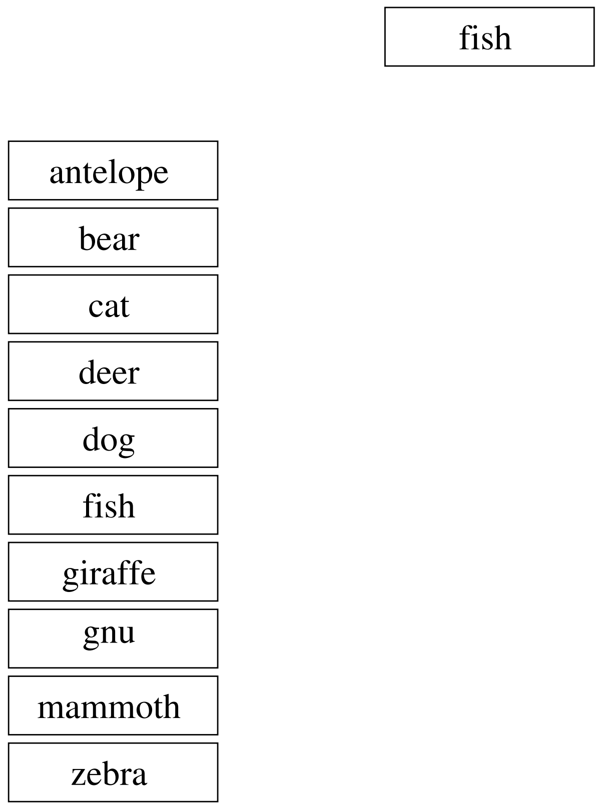
No Prize

I will have the Lamborghini sent back and the date with Johnny Depp / Scarlett Johansson cancelled

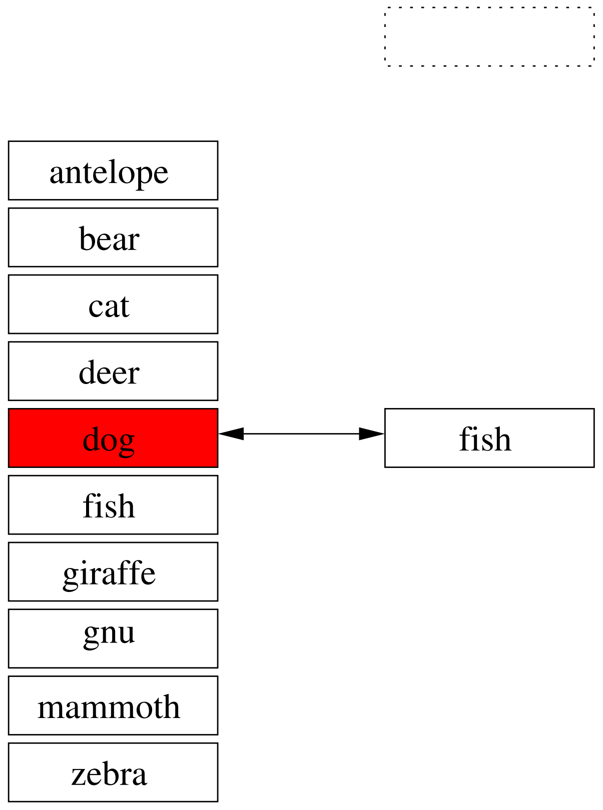
Prize

Well done - you have won a Mars bar

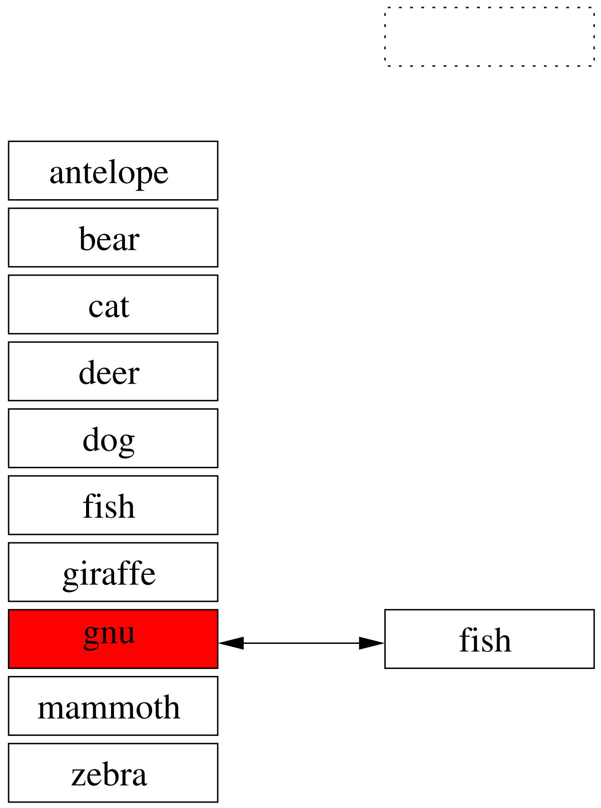
Example: Binary Search of a Dictionary



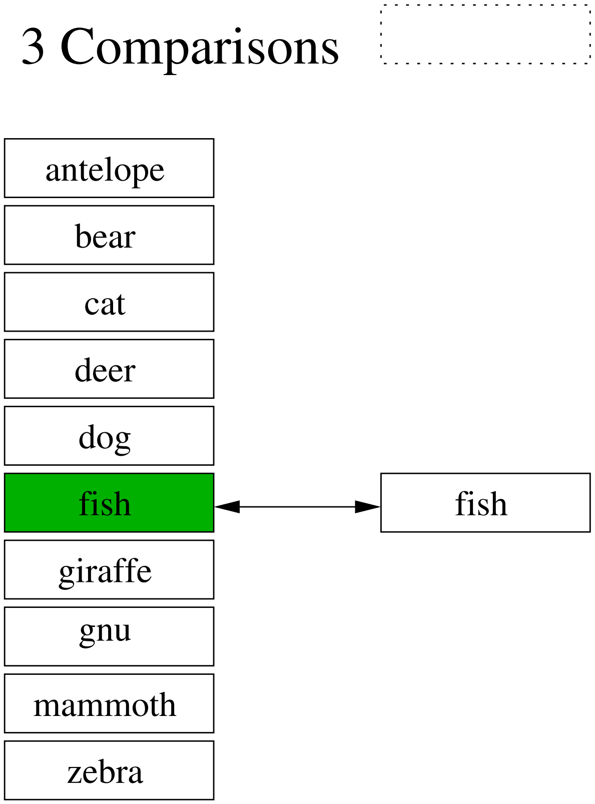
Example: Binary Search of a Dictionary



Example: Binary Search of a Dictionary



Example: Binary Search of a Dictionary



Binary Search of a Dictionary

```
**Algorithm** DictLookup:
1:  **Input:** a word $w$, a dictionary $D$ of $n$ words
2:  $beginp \leftarrow 0$, $ptr \leftarrow n/2$, $endp \leftarrow n$
3:  **while** ($beginp < endp$) **do**
4:      $val \leftarrow \text{strncmp}(w, D[ptr])$
5:      **if** $val = 0$
6:          print "word found !", exit
7:      **if** $val > 0$
8:          $beginp \leftarrow ptr + 1$
9:      **else**
10:         $endp \leftarrow ptr$
11:     $ptr \leftarrow (beginp + endp)/2$
```

What is the input size here? What is the time complexity?

Binary Search for a Dictionary

Assume that the word length $<<$ dictionary length. Hence, word length can be treated as a constant. So, n, the dictionary length, is the input size.

Each time the word is compared to an entry in the dictionary, (the number of times around the while loop) the number of remaining words it could be is at least halved.

In symbols: The number of remaining words it could be is $n/2^c$ where c is the number of comparisons. When this is less than or equal to 1, we have found the word.

We need to solve

$$n/2^c \leq 1,$$

for c. Rearranging, this is

$$2^c \geq n.$$

Take logs of both sides:

$$\log_2(2^c) \geq \log_2 n$$

$$c \geq \log_2 n.$$

For integer c, this is ensured if

$$c = \lfloor \log_2(n) + 1 \rfloor$$

comparisons (iterations around the while loop).

Since each while loop itself only has a constant number of operations, the overall asymptotic (worst case) time complexity is $O(\log_2 n)$.

Spellchecker

```
Algorithm SpellChecker:
1: input: a text file of $n$ words, a dictionary $D$ of $m$ words
2: read the text file and put words into an array, $word[]$
3: for $i \leftarrow 1$ to $n$ do
4:     $found? \leftarrow$ DictLookup$(word[i], D)$
5:     if $\neg found?$
6:         print $word[i]$ “misspelled? [newline]”
```

This algorithm uses the dictionary lookup from above.

For an input of n words and a dictionary of size m, it will run in worst case $O(n \log_2 m)$ time.

Example 4: A Recursive Algorithm

```
**Algorithm** HeapPermute:
**input:** a string $str$ of length $n$
heapPerm$(n)$
    **if**$(n = 0)$
        print $str$
    **else**
        **for** $i = 1$ to $n$ **do**
            heapPerm$(n - 1)$   [comment:  recursive call]
            **if** $n$ is odd
                swap $str[1]$ and $str[n]$
            **else**
                swap $str[i]$ and $str[n]$
```

What does this do?

What is the time complexity?

How many times are the swap operations done?

$$n+n*(n-1)+n*(n-1)*(n-2)+\ldots+n*(n-1)*(n-2)*\ldots*1$$

.

Explanation: swap() is called n times when heapPerm(n) is called. To that we must add the number of times swap() is called when we call heapPerm($n-1$). This is $n-1$ times for each call of heapPerm($n-1$) and there are n such calls. This is $n*(n-1)$. We then add the number of times we call swap() when heapPerm($n-2$) is called ... and so on.

For large n, it turns out that the number of calls to swap() is 2.7182818284 (or "e") times $n!$. Since each swap is a constant time operation, and e is a constant, overall the time complexity is $O(n!)$.

Now, can you guess/see what the algorithm does?

$\boxed{\textbf{\textit{NB}}\text{: Do not associate recursion with poor algorithmic efficiency.}}$

Experimental Analysis

Sometimes, inspecting code is difficult.

Or, we want to know how long a programme will run on a particular piece of hardware.

Or, we may only be interested in a small range of values of n.

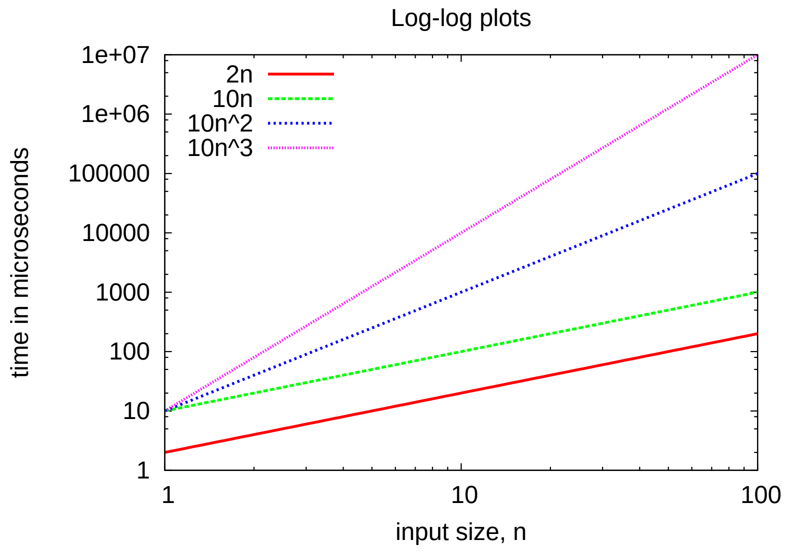
Then we may do an ***experimental analysis***

Experimental Recipe

1. Implement the algorithm
2. Run it for a range of input sizes
3. Make repeated measurements of running time, or primitive operations
4. Plot results of t against n and fit a line
5. Extrapolate to larger n if needed
6. Estimate big-O if needed

More details on pages 42–46 of the core textbook.

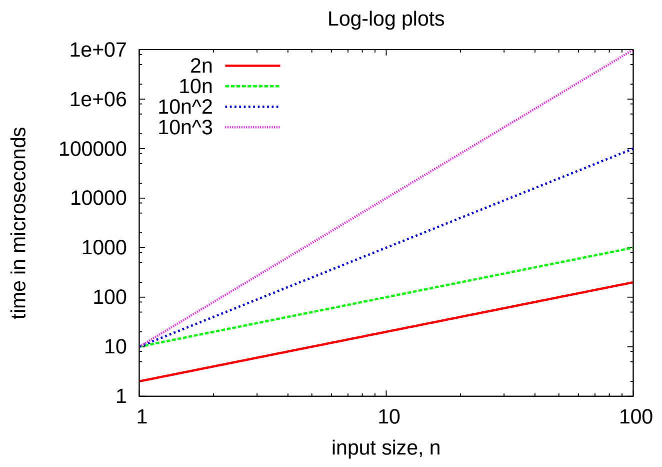
Log-log Plots



On a log-log plot, $c.n^k$ will appear as a straight line with gradient k and y-intercept c.

(see page 46 of the core text)

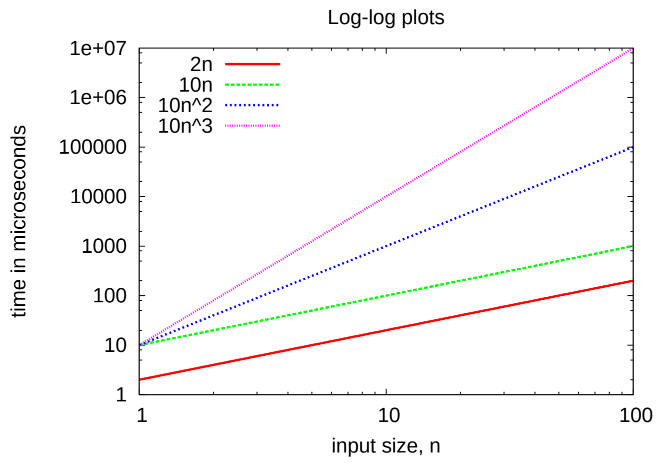
Log-log Plots



How ***NOT !*** to measure the gradient of the $10n^3$ line :
$(1e7 - 10)/(100 - 1) \approx 101,000$

Plainly, this is not the correct power

Log-log Plots

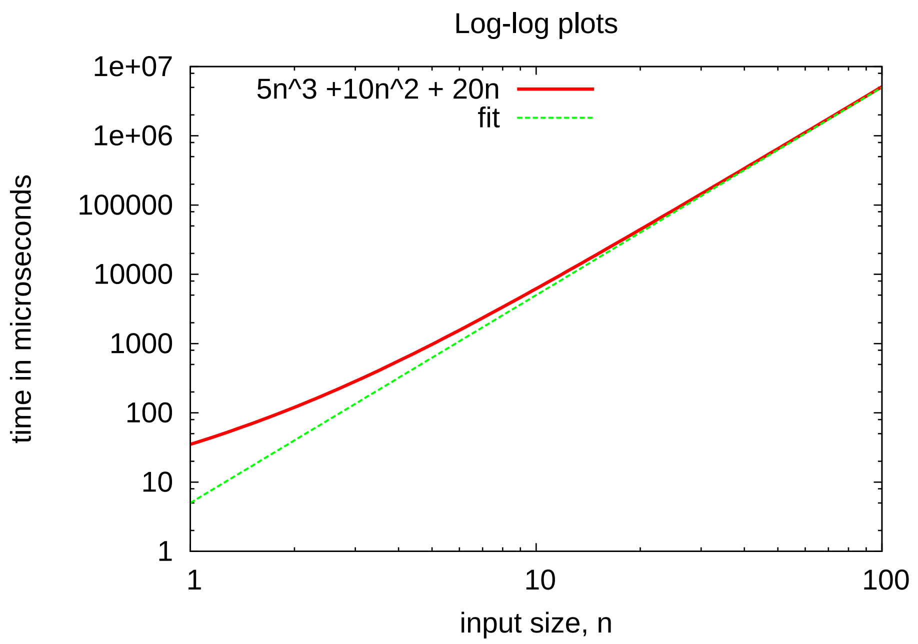


How do we measure the gradient then? Use the exponents:

$$(7-1)/(2-0)=3.$$

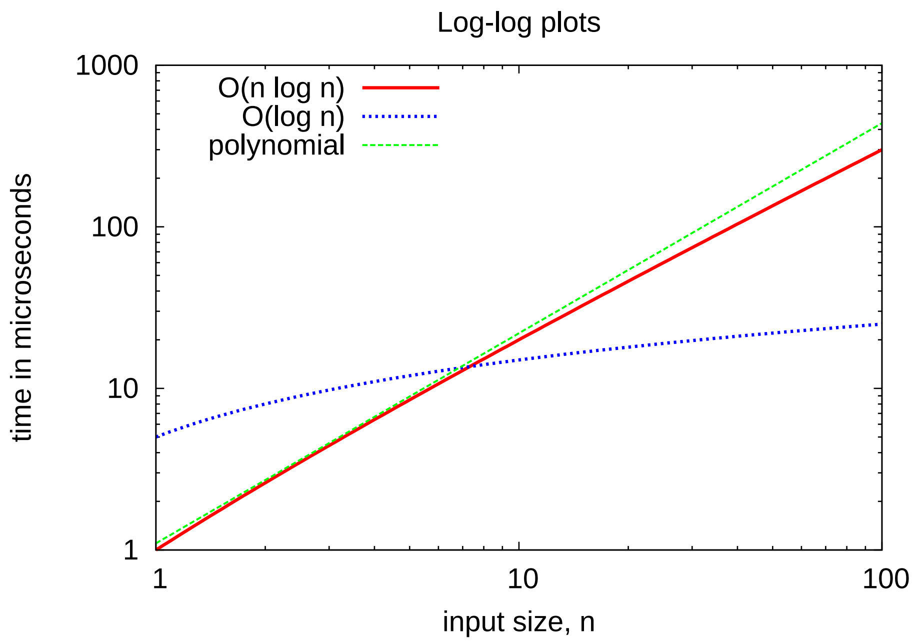
And to get the intercept, we do read the number (not the exponent). Here it is 10, as expected.

Log-log Plots



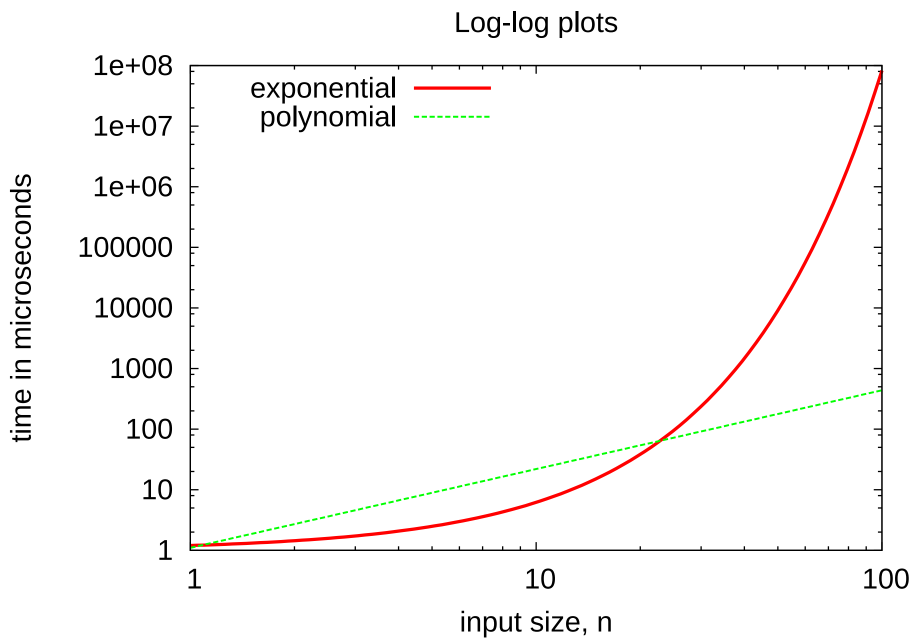
The low order terms of a polynomial — often the overhead — will result in a curve for the low n data points.

Log-log Plots



Curves of logs will curve “down”. They can be hard to detect.

Log-log Plots



Curves of exponentials will curve upwards.

Brooks...

...takes time out of Hacking case to brush up on amortized analysis

Summary

- Big-O simplifies complexity analysis
- Look at the loops when analysing iterative algorithms. Are they nested?
- Look at the recursive calls when analysing recursive algorithms
- Log-log plots useful for estimating the complexity from experimental data

Keep practising big-O. The effort will be paid back many times.

Complexity Lab: Advice on Experiments

Please look at

http://www.cs.manchester.ac.uk/ugt/2010/COMP26120/lab/ex6goodExperiments.html

- Repeat measurements and calculate mean and variance
- Think carefully about your range of N
- Too large, and you will not have time
- Too small, and you may not see the asymptotic complexity
- Estimate from the larger values of N to get the asymptotic performance
- Use Log-Log plots to estimate polynomials