i206: Lecture 6:
Analysis of Algorithms

Tapan Parikh

Spring 2013

Some slides courtesy Marti Hearst, John Chuang and others

Analysis of Algorithms

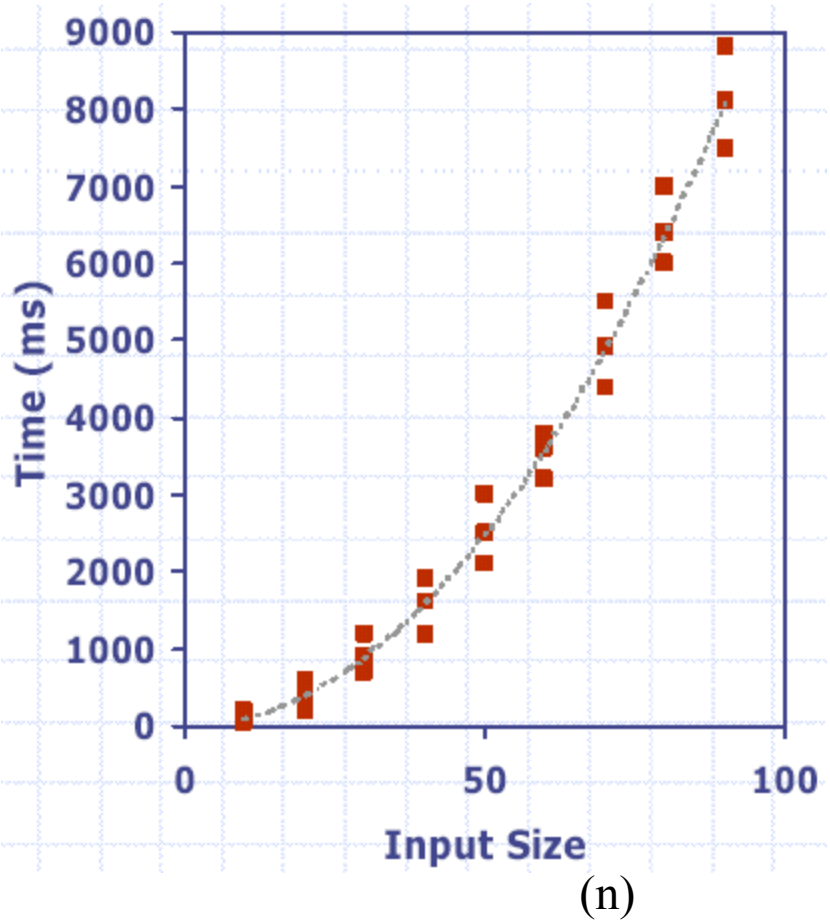
- Characterizing the running times of algorithms in terms of the size of the input
- Secondarily, characterizing the space usage of algorithms and associated data structures

Why Analysis of Algorithms?

- To find out
  - How long an algorithm takes to run
  - How to compare different algorithms
    - This is done at a very abstract level
    - This can be done **before** code is written
- Alternative: Performance analysis
  - Actually time each operation as the program is running
    - Specific to the machine and the implementation of the algorithm
  - Can only be done **after** code is written

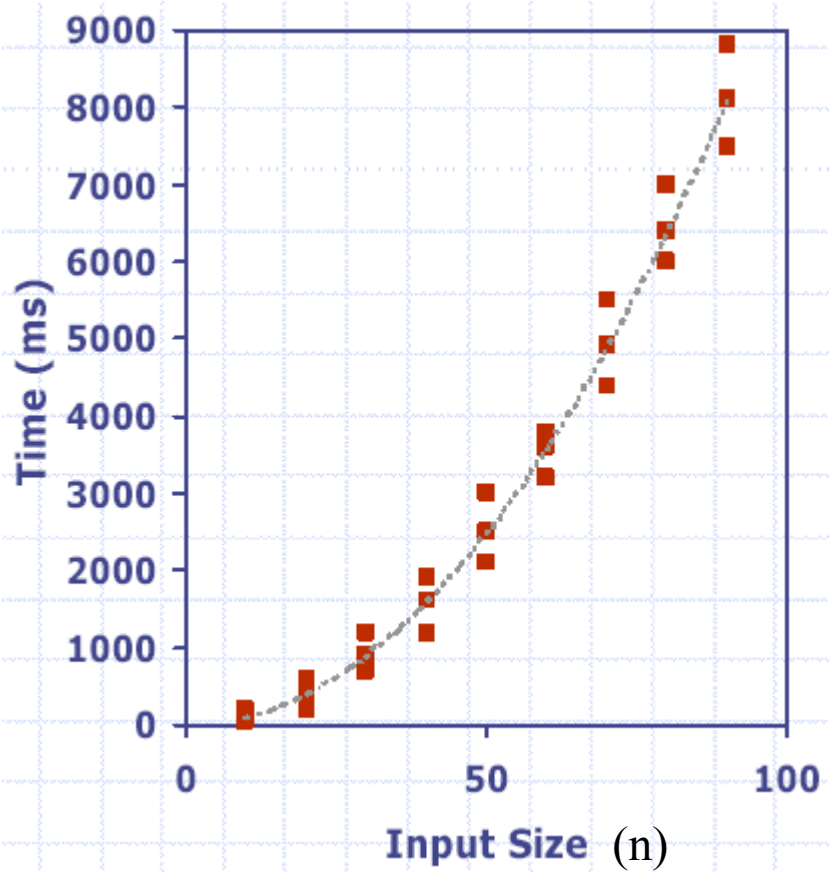
Running Time

- In general, running time increases with input size
- Running time also affected by:
  - Hardware environment (processor, clock rate, memory, disk, etc.)
  - Software environment (operating system, programming language, compiler, interpreter, etc.)



Quantifying Running Time

- Experimentally measure running time
  - Need to fully implement and execute
- Analysis of pseudo-code



The RAM Model

- Random Access Machine (not Memory)
- An idealized notion of how the computer works
  - Each ”primitive" operation (+, -, =, if) takes exactly 1 step.
  - Each memory access takes exactly 1 step
  - Loops and method calls are *not* simple operations, but depend upon the size of the data and the contents of the method.
- Measure the run time of an algorithm by counting the number of steps.

Adapted from http://www.cs.sunysb.edu/~algorith/lectures-good/node1.html

Primitive Operations

- Assign a value to a variable
- Call a method
- Arithmetic operation
- Comparing two numbers
- Indexing into an array
- Following an object reference
- Returning from a method

Counting Primitive Operations

```
Algorithm ArrayMax(A,n)
        Input: An array A storing N integers
        Output: The maximum element in A.

        currentMax ← A[0]
        for i← 1 to n-1 do
                if currentMax < A[i] then
                currentMax ←A[i]
        return currentMax
```

Counting Primitive Operations

Algorithm ArrayMax(A,n)
Input: An array A storing N integers
Output: The maximum element in A.

currentMax ← A[0] **2 steps + 1 to initialize i**
for i ← 1 **to** n-1 **do** **2 step each time (compare i to n, inc i)**
if currentMax < A[i] **then** **2 steps**
currentMax ← A[i] **2 steps**
return currentMax **1 step**

n-1 times

How often done??

Between 4(n-1) and 6(n-1) steps in the loop

Example: Counting Primitive Operations

Algorithm: arrayMax(A,n):

Input: An array A storing n >= 1 integers

Output: the maximum element in A

```
currentMax = A[0]
for (i=1; i<=n-1; i++)
   if currentMax < A[i]
   then currentMax = A[i]
Return currentMax
```

Two operations: indexing into array; assign value to variable

One operation: assign value to variable

Two operations repeated n times: subtract, compare

Four or six operations repeated (n-1) times:
- index and compare (for the if statement);
- index and assign (for the then statement if necessary)
- addition and assign (for the increment)

One operation: return value of variable

Total: 2 + 1 + 2n + 4(n-1) + 1 = 6n (best case)
Total: 2 + 1 + 2n + 6(n-1) + 1 = 8n – 2 (worst case)

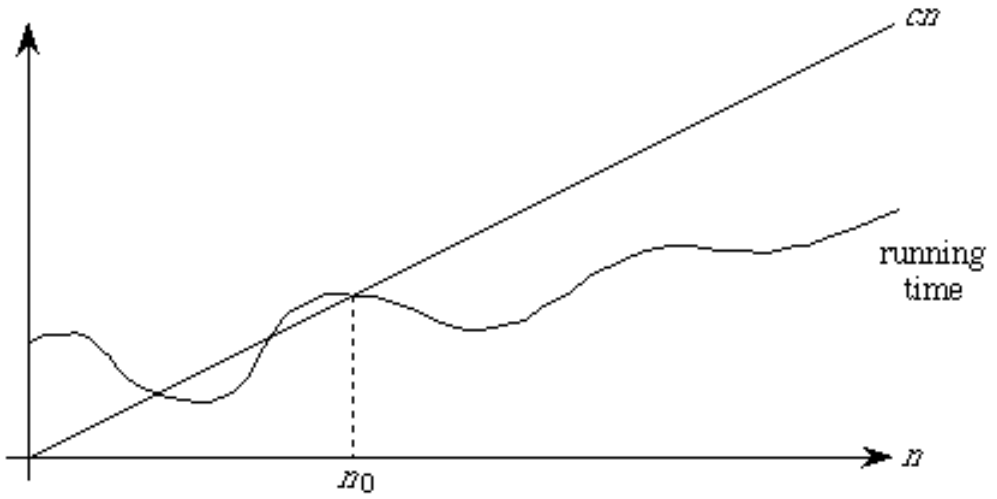
Algorithm Complexity

- Worst Case Complexity:
  - the function defined by the *maximum* number of steps taken on any instance of size *n*
- Best Case Complexity:
  - the function defined by the *minimum* number of steps taken on any instance of size *n*
- Average Case Complexity:
  - the function defined by the *average* number of steps taken on any instance of size *n*

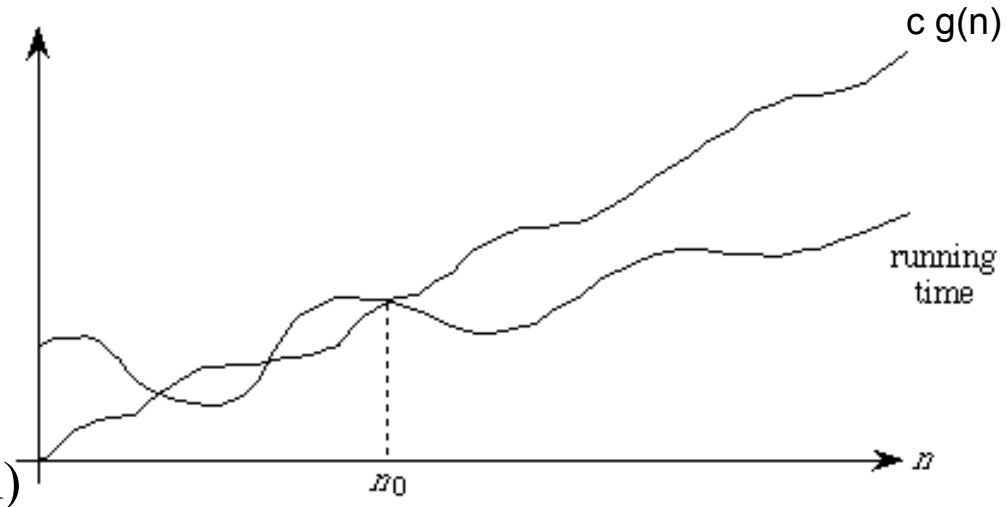
Adapted from http://www.cs.sunysb.edu/~algorith/lectures-good/node1.html

Definition of Big-Oh

A running time is $O(g(n))$ if there exist constants $n_0 > 0$ and $c > 0$ such that for all problem sizes $n > n_0$, the running time for a problem of size n is at most $c(g(n))$.



In other words, $c(g(n))$ is an **upper bound** on the running time for sufficiently large n.



Example: the function f(n)=8n–2 is O(n)
The running time of algorithm arrayMax is O(n)

http://www.cs.dartmouth.edu/~farid/teaching/cs15/cs5/lectures/0519/0519.html

More formally

- Let `f(n)` and `g(n)` be functions mapping nonnegative integers to real numbers.

- `f(n) is O(g(n))` if there exist positive constants `n0` and `c` such that for all `n>=n0, f(n) <= c*g(n)`

- Other ways to say this:

  `f(n) is` **`order`** `g(n)`

  `f(n) is` **`big-Oh of`** `g(n)`

  `f(n) is` **`Oh of`** `g(n)`

  `f(n) ∈ O(g(n))` (set notation)

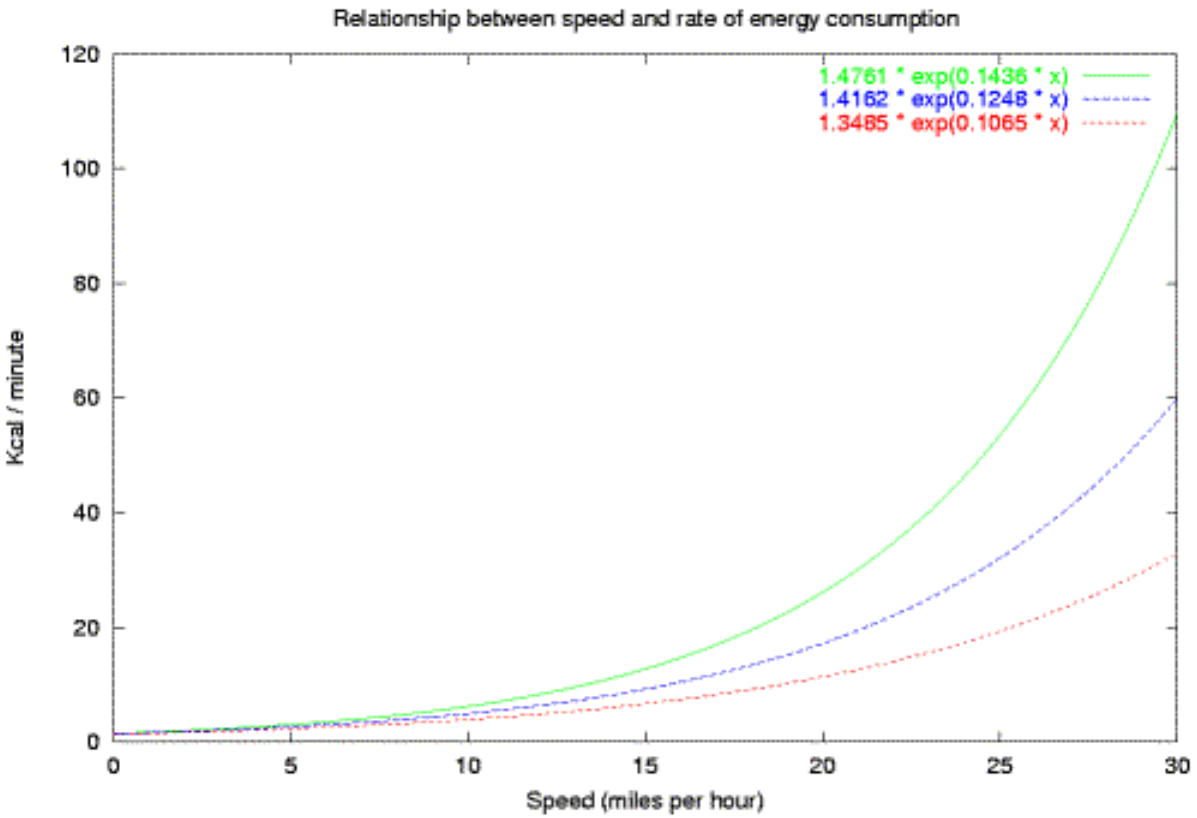
Digression: Mathematical Functions

- Function: a rule that
  - Coverts inputs to outputs in a well-defined way.
  - This conversion process is often called **mapping.**
  - Input space called the **domain**
  - Output space called the **range**

Example: How many calories does bicycling burn?

Miles/Hour vs. KiloCalories/Minute

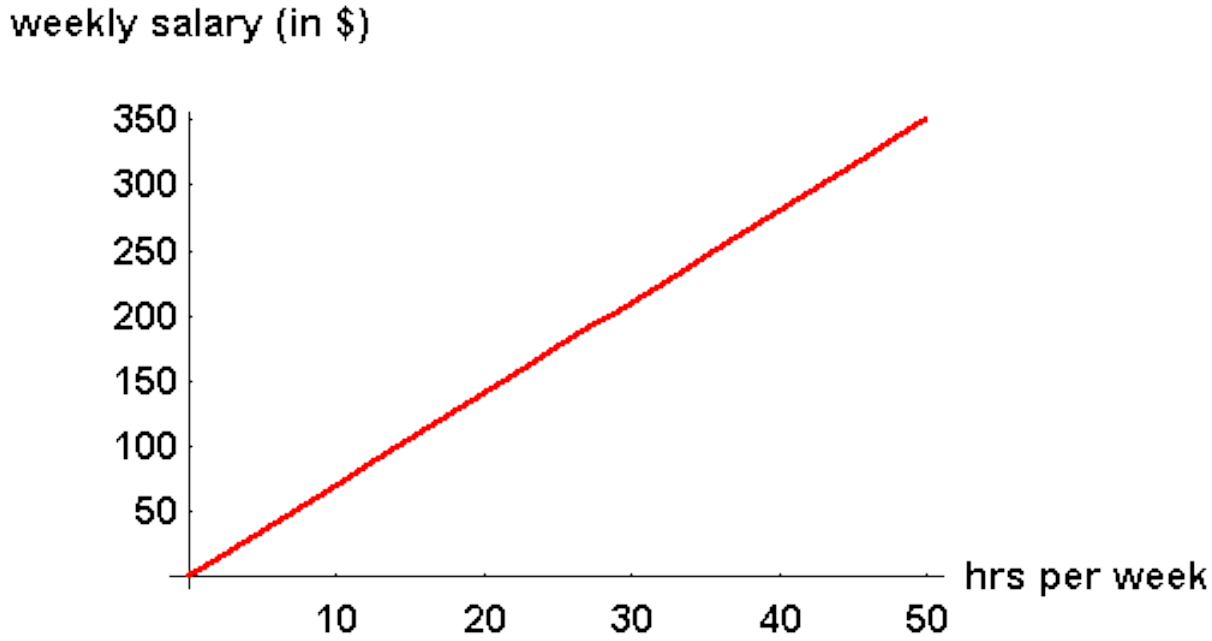
For a 150 lb rider.
Green: riding on a gravel road with a mountain bike
Blue: paved road riding a touring bicycle
Red: racing bicyclist.



From Whitt, F.R. & D. G. Wilson. 1982. *Bicycling Science (second edition)*.
http://www.frontier.iarc.uaf.edu/~cswingle/misc/exercise.phtml

Representing Functions

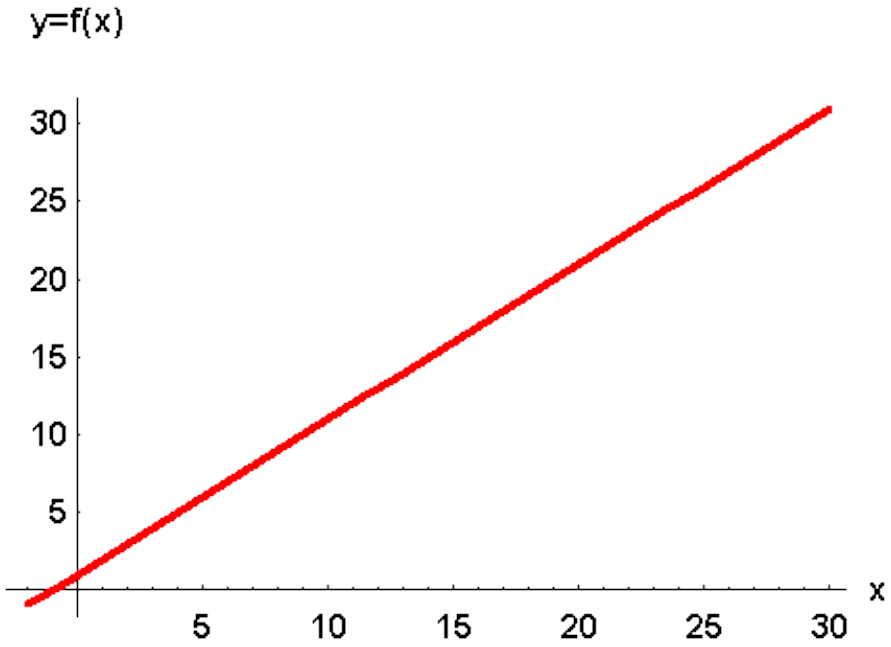
- Notation
  - Many different kinds
  - f(x) is read “f of x”
  - This means a function called f takes x as an argument
  - f(x) = y
  - This means the function f takes x, and the value of f(x) is determined by the expression y.



Here f(x) = 7x

A point on this graph can be called (x,y) or (x, f(x))

http://www.sosmath.com/algebra/logs/log1/log1.html



A straight line is defined by the function **y = ax + b**

a and **b** are constants

x is variable

Here y = 7x + 0

http://www.sosmath.com/algebra/logs/log1/log1.html

Exponents and Logarithms

- Exponents: shorthand for multiplication

$$4^5 = 4*4*4*4*4$$

- Logarithms: shorthand for exponents

  $2^3 = 8$ can be expressed as $\log_2 8 = 3$

  or we can say that the 'log base 2 of 8' = 3.

- How we use these?
  - Difficult computational problems grow exponentially
  - Logarithms are useful for “squashing” them

$$2^{16}$$

$$\log_2(2^{16}) = 16$$

Logarithm lets
you "grab the exponent"

$$5^{-2} = \frac{1}{5*5} = \frac{1}{25}$$

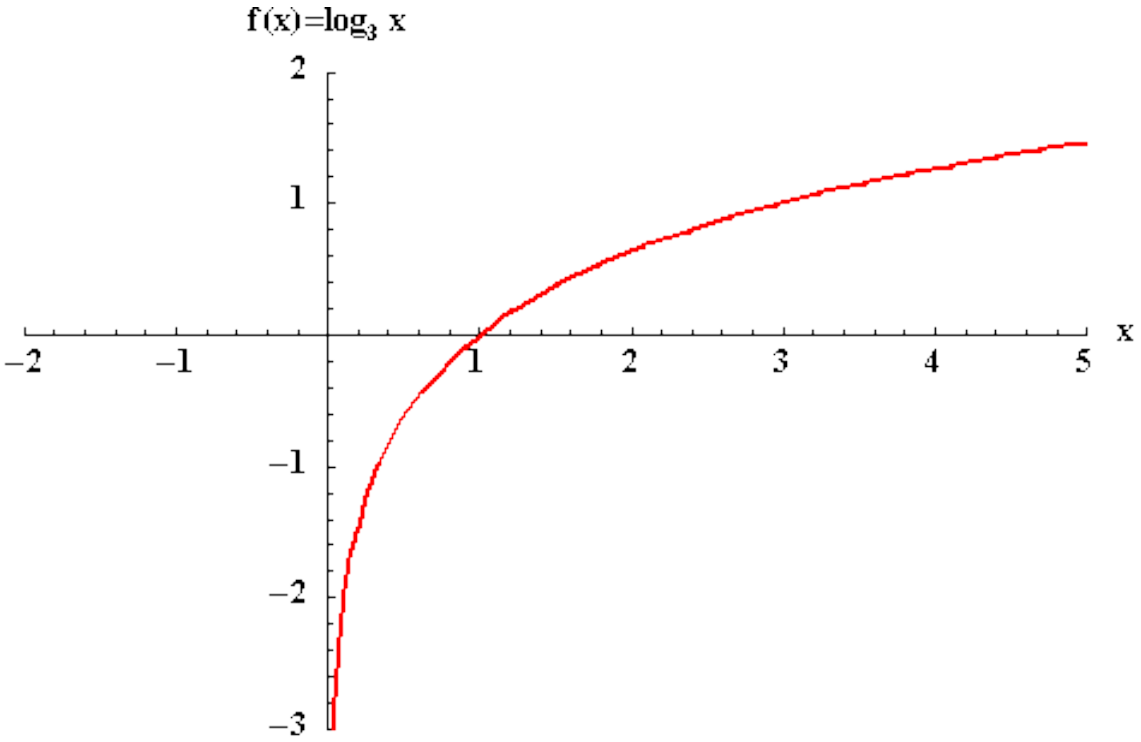
$$\log_5 \frac{1}{25} = -2$$

For $x > 0, a > 0, a \neq 1$

$$f(x) = \log_a(x) \text{ if and only if } a^{f(x)} = x$$

$$f(x) = \log_2(x) \text{ if and only if } 2^{f(x)} = x$$

http://www.sosmath.com/algebra/logs/log1/log1.html

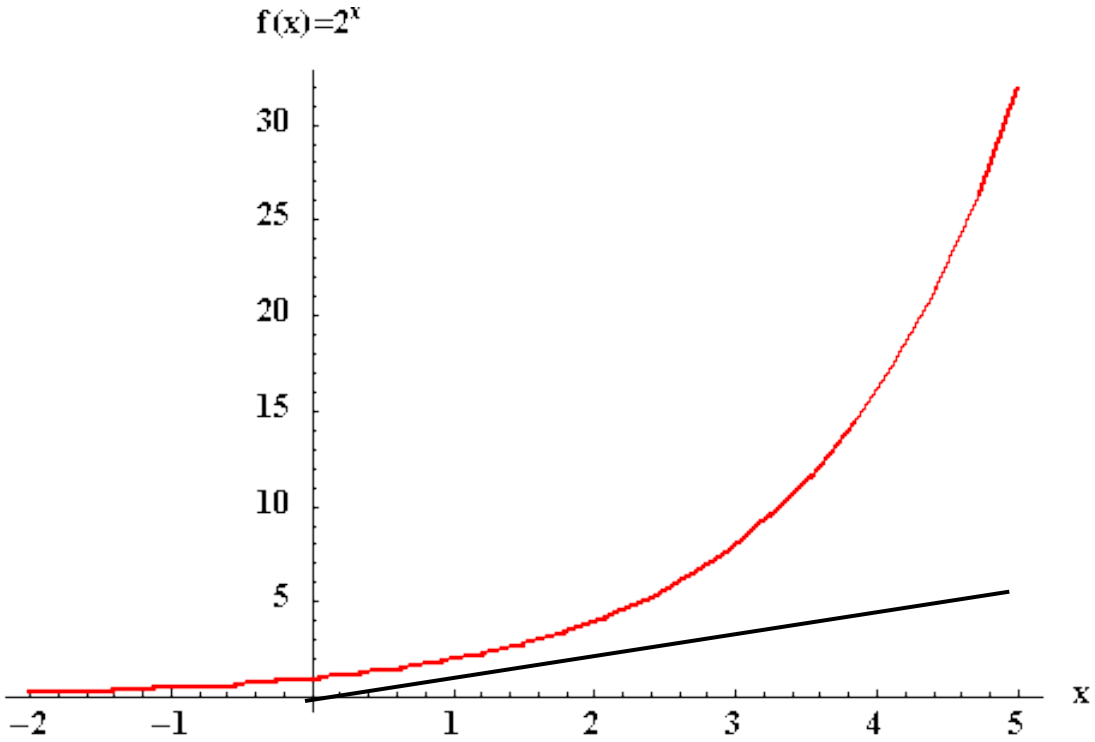


For $x > 0, a > 0, a \neq 1$

$$f(x) = \log_a(x) \text{ if and only if } a^{f(x)} = x$$

$$f(x) = \log_2(x) \text{ if and only if } 2^{f(x)} = x$$

http://www.sosmath.com/algebra/logs/log1/log1.html

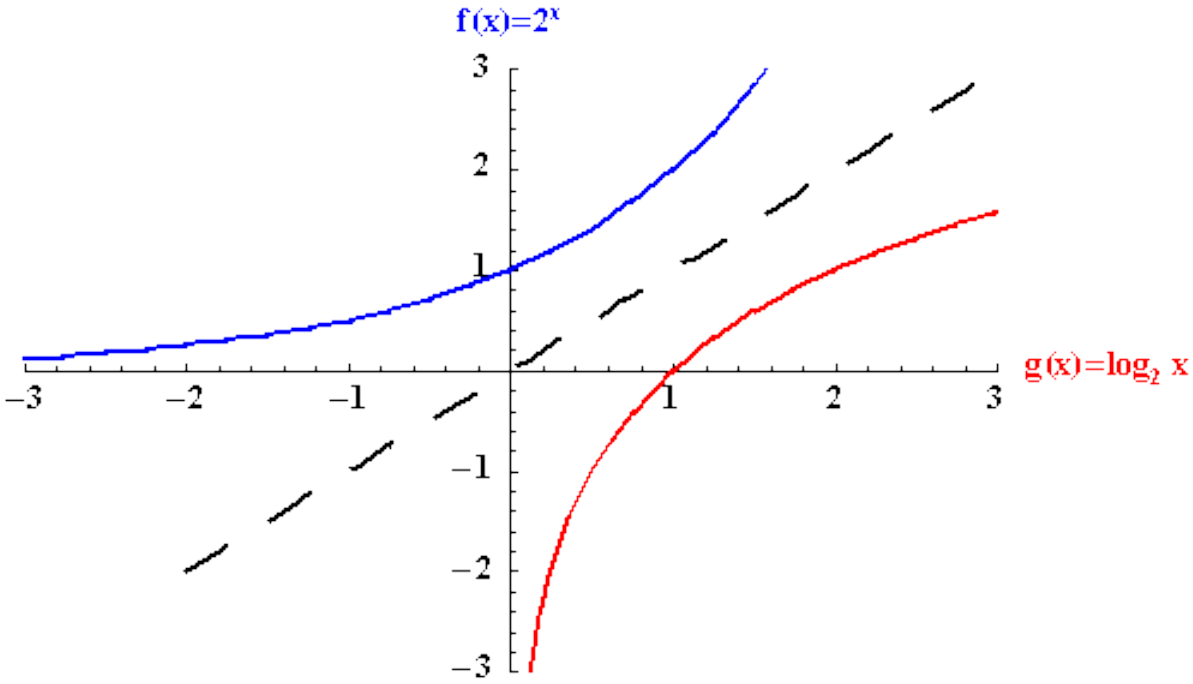


The exponential function f with base a is denoted by $f(x) = a^x$ and *x* is any real number.

Note how much more "quickly the graph grows" than the linear graph of f(x) = x

Example: If the base is 2 and x = 4, the function value f(4) will equal 16. A corresponding point on the graph would be (4, 16).

http://www.sosmath.com/algebra/logs/log1/log1.html



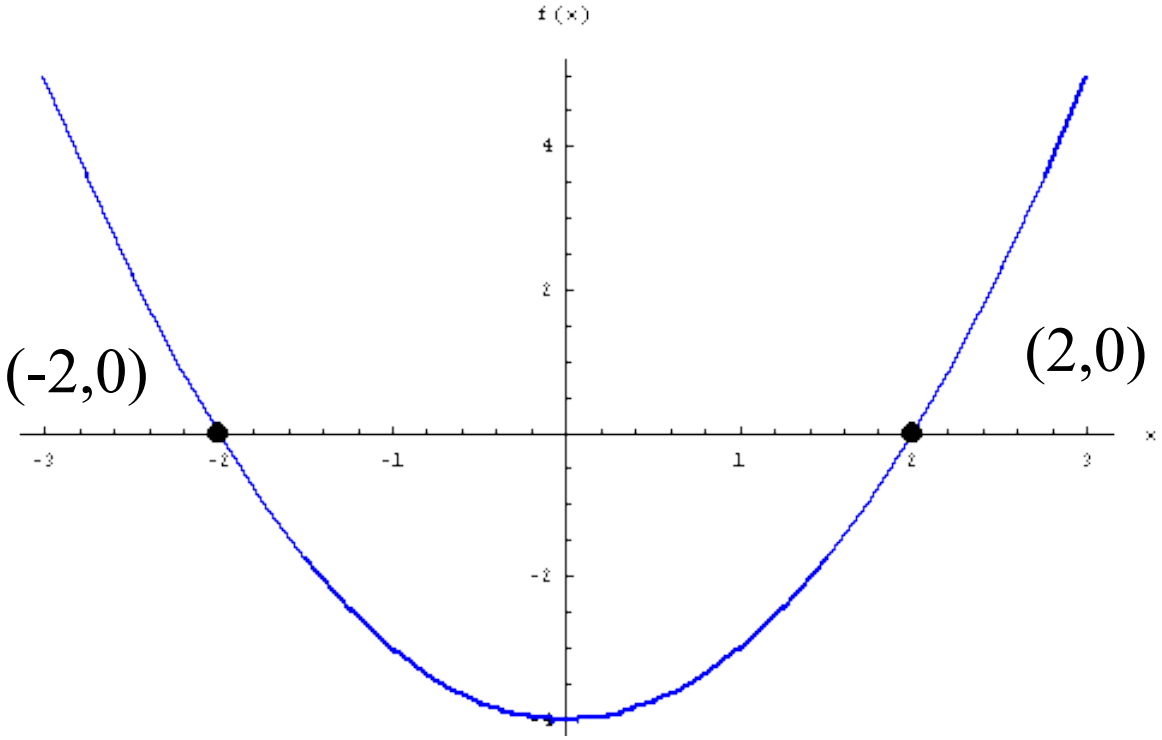
Logarithmic functions are the inverse of exponential functions.

If (4, 16) is a point on the graph of an exponential function, then (16, 4) would be the corresponding point on the graph of the inverse logarithmic function.

http://www.sosmath.com/algebra/logs/log1/log1.html

Other Functions

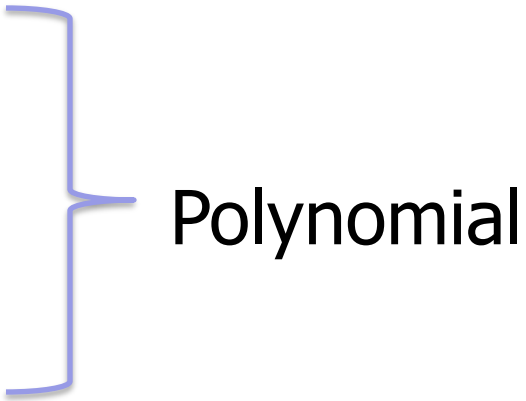
- Quadratic function: $f(x) = ax^2 + bx + c$
- This is a graph of: $f(x) = x^2 - 4$



http://www.sosmath.com/algebra/

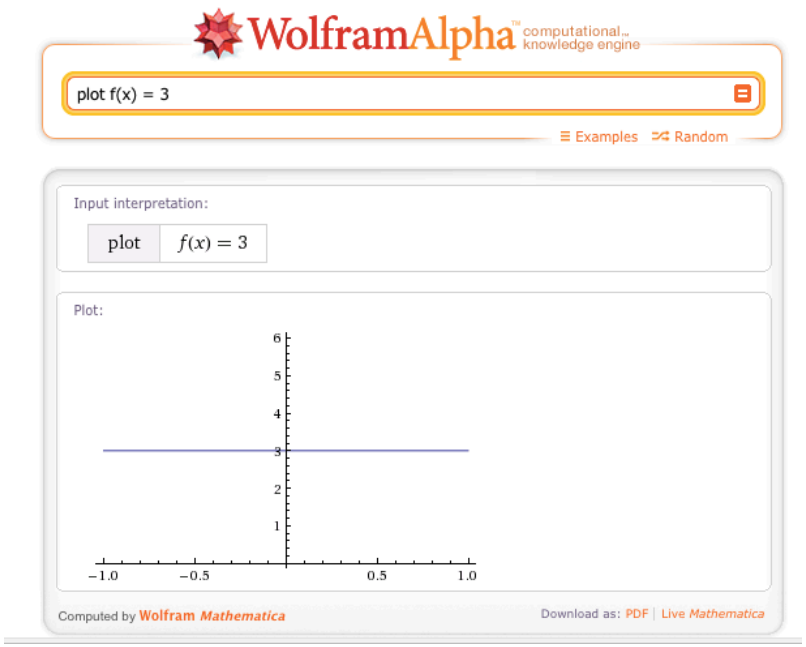
The Seven Common Functions

- Constant
- Linear
- Quadratic
- Cubic
- Exponential
- Logarithmic
- n-log-n



The Seven Common Functions

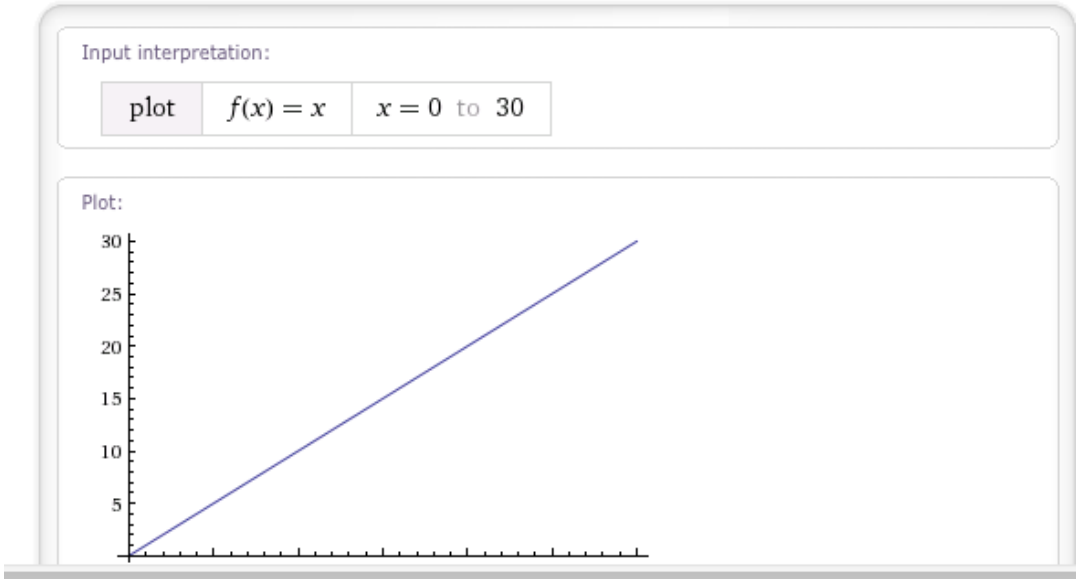
- Constant: f(n) = c
  - E.g., adding two numbers, assigning a value to some variable, comparing two numbers, and other basic operations
  - Useful free web tool: Wolfram's Alpha



The Seven Common Functions

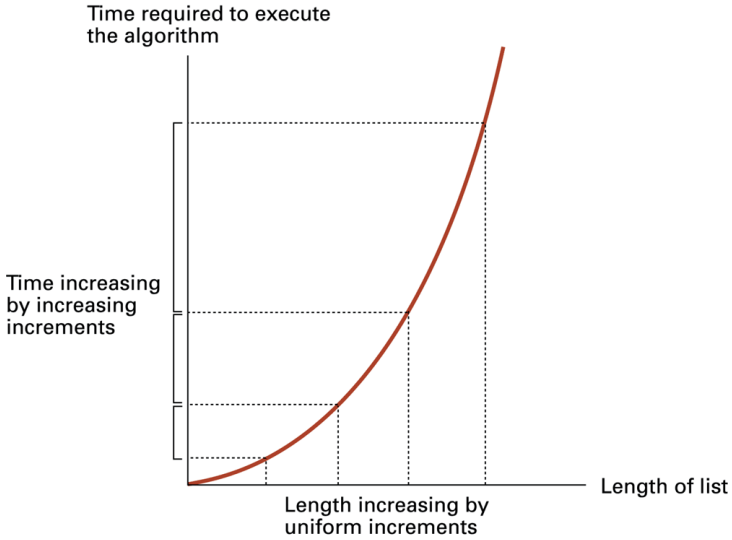
Linear: f(n) = n

- Do a single basic operation for each of *n* elements, e.g., reading *n* elements into computer memory, comparing a number *x* to each element of an array of size *n*



The Seven Common Functions

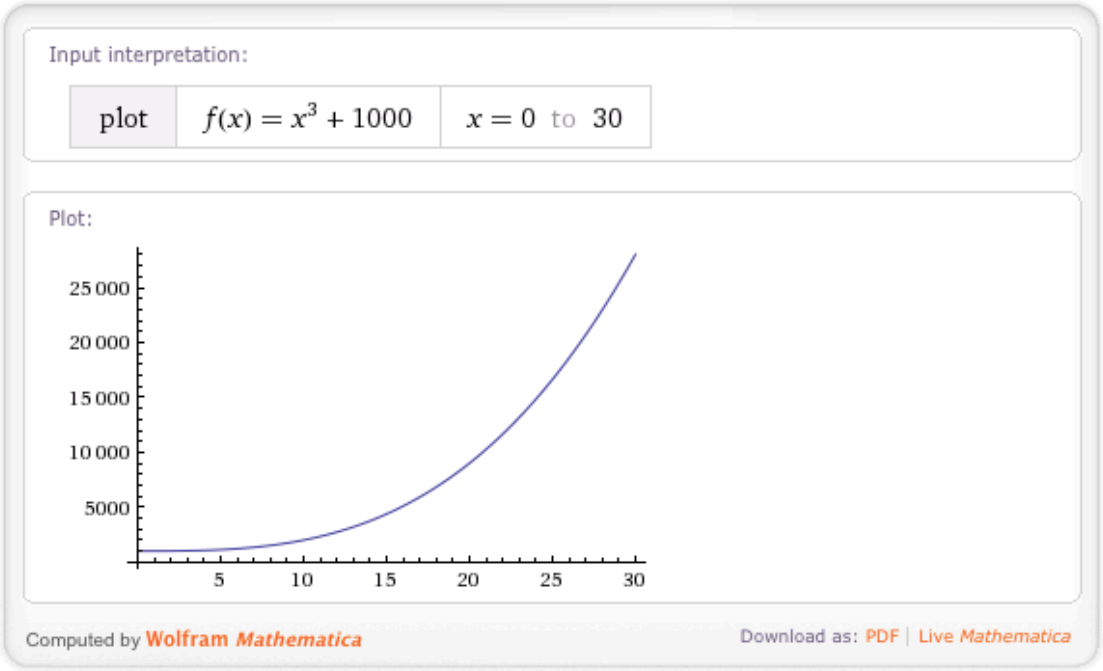
- Quadratic: $f(n) = n^2$
  - E.g., nested loops with inner and outer loops



Brookshear Figure 5.19

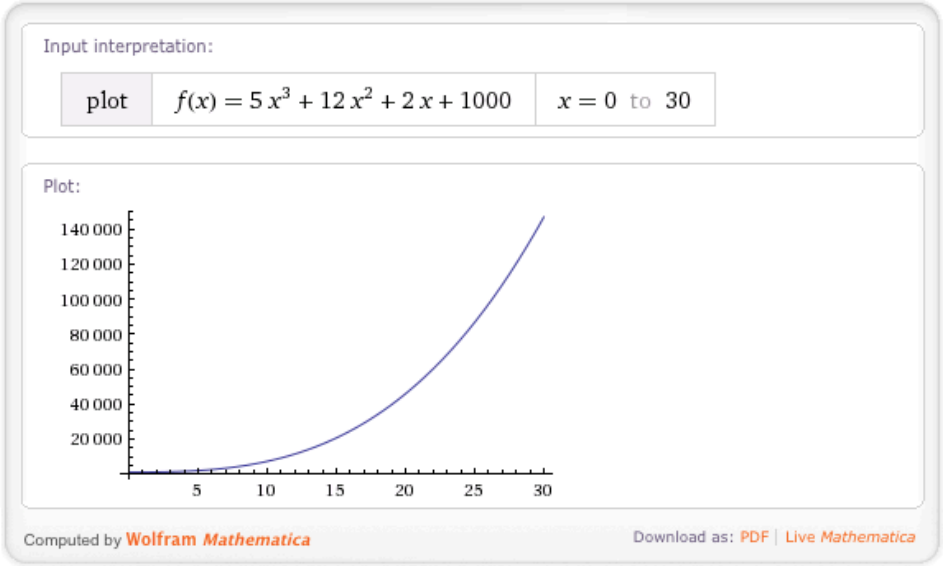
The Seven Common Functions

Cubic: $f(n) = n^3$



The Seven Common Functions

- More generally, all of the above are <u>polynomial</u> functions:
  - $f(n) = a_0 + a_1 n + a_2 n^2 + a_3 n^3 + \ldots + a_n n^d$
  - where the a_i's are constants, called the coefficients of the polynomial.



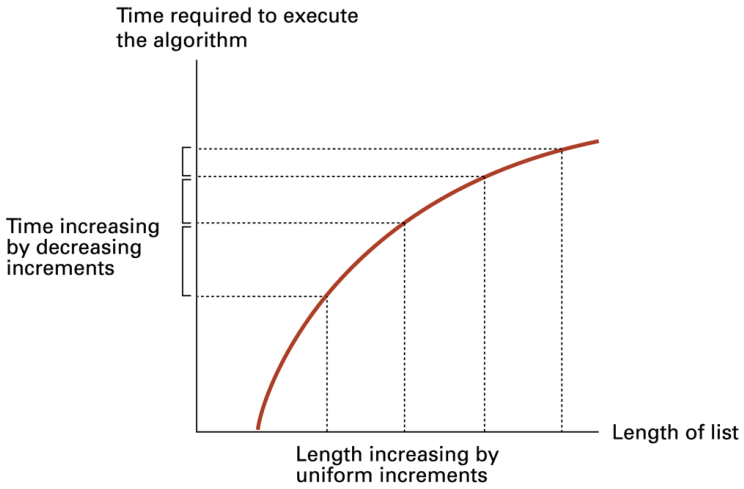
okshear Figure 5.19

The Seven Common Functions

Logarithmic:

$$f(n) = \log_b n$$

- Intuition: number of times we can divide *n* by *b* until we get a number less than or equal to 1
- The base is omitted for the case of *b*=2, which is the most common in computing:
  - log n = $\log_2 n$

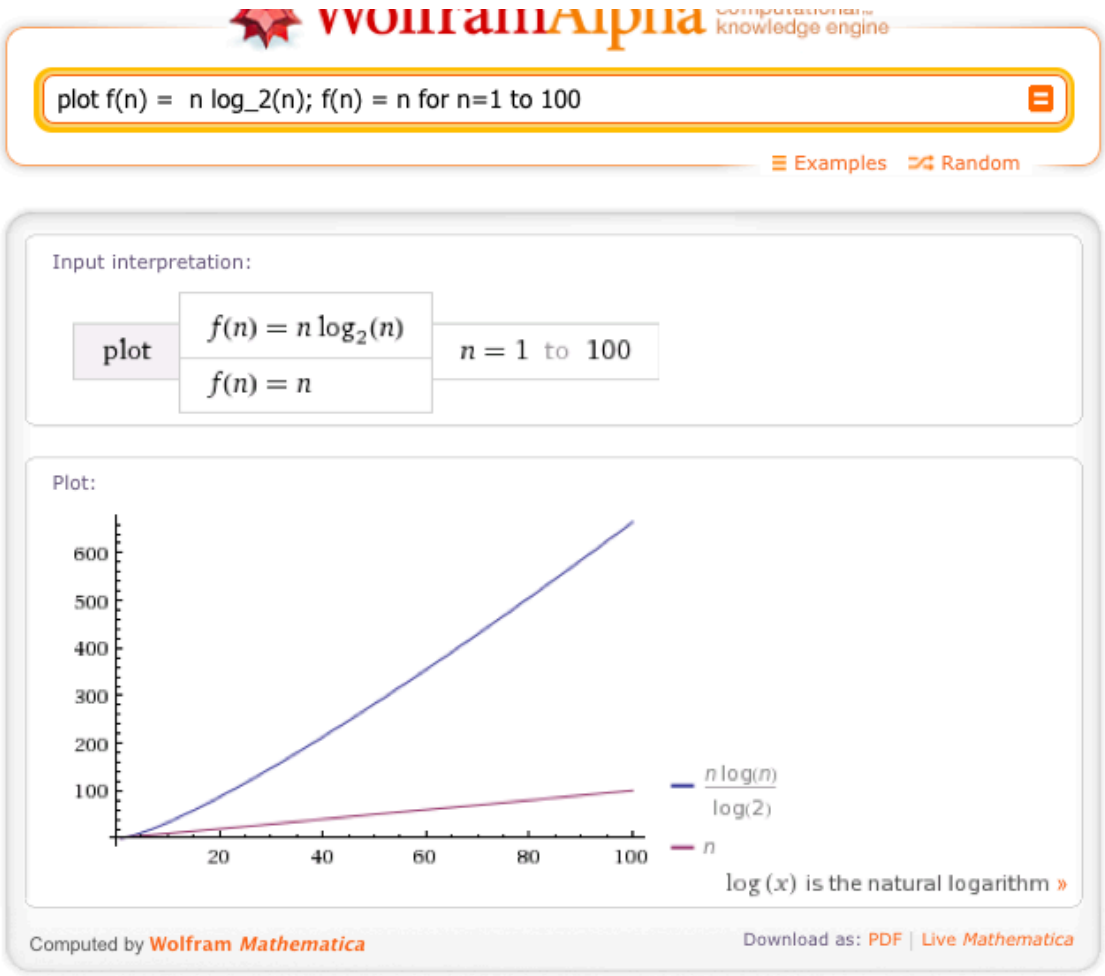


Brookshear Figure 5.20

The Seven Common Functions

n-log-n function: f(n) = n log n

- Product of linear and logarithmic

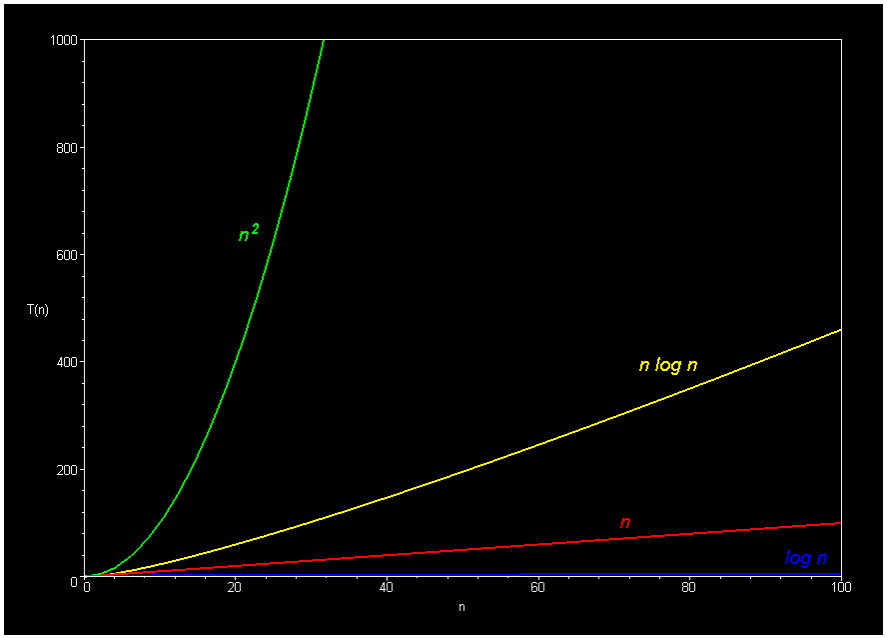


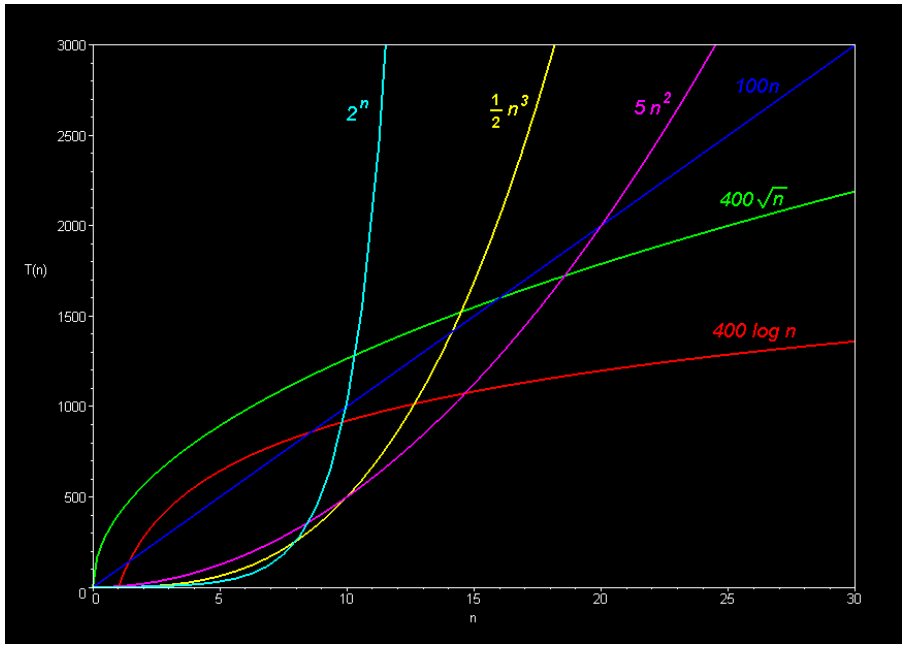
The Seven Common Functions

- Exponential: $f(n) = b^n$
  - Where *b* is a positive constant, called the base, and the argument *n* is the exponent
  - In algorithm analysis, the most common base is $b=2$
  - E.g., loop that starts with one operation and doubles the number of operations with each iteration
  - Similar to the geometric sum:
    - $1+2+4+8+\ldots+2^{n-1} = 2^n - 1$
  - BTW, $2^n - 1$ is the largest integer that can be represented in binary notation using *n* bits

Comparing Growth Rates

- $1 < \log n < n^{1/2} < n < n \log n < n^2 < n^3 < b^n$





http://www.cs.pomona.edu/~marshall/courses/2002/spring/cs50/BigO/

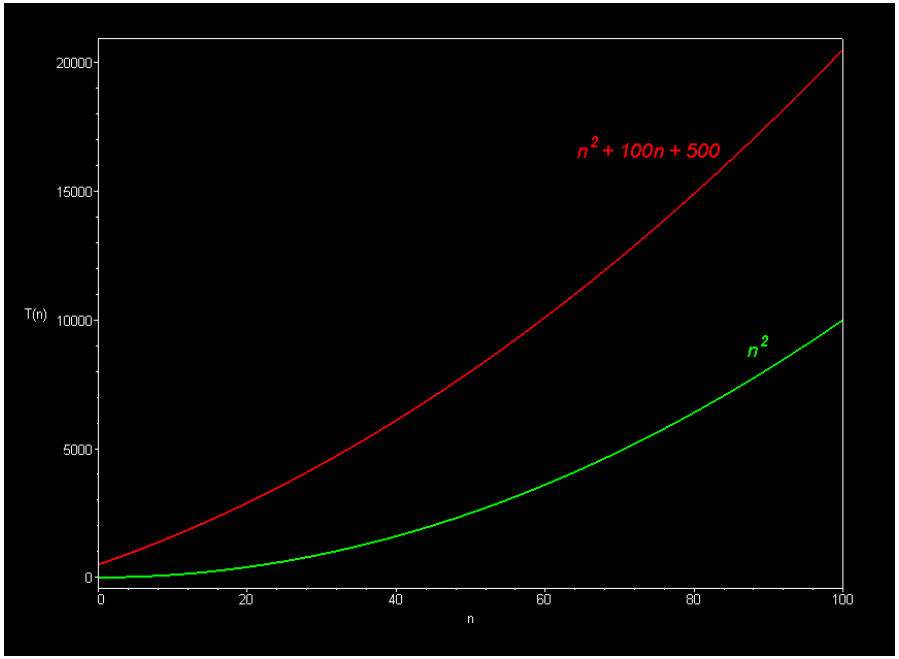
Function Pecking Order

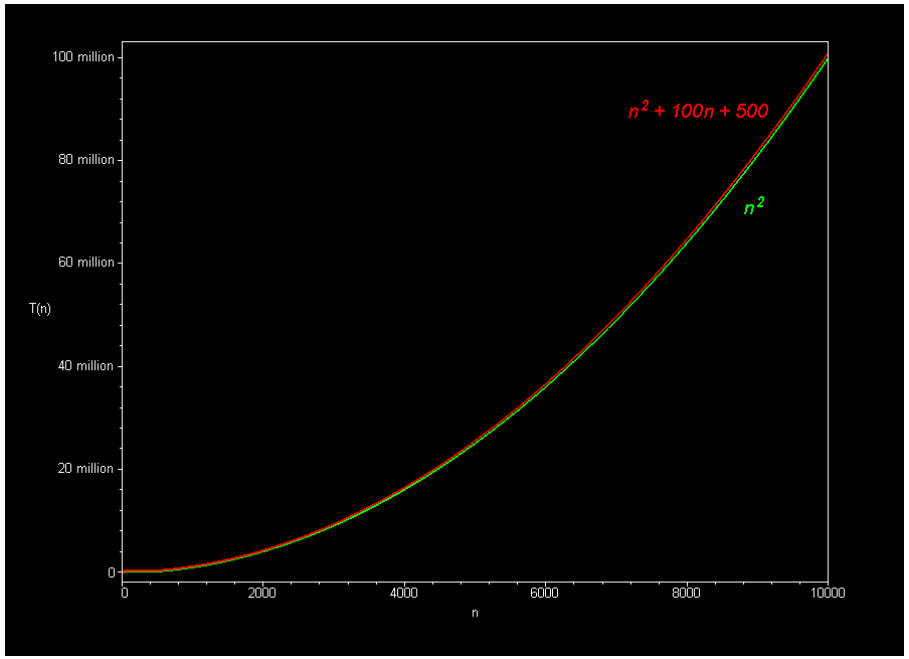
- In increasing order
  - logarithmic: $O(\log n)$
  - linear $O(n)$
  - quadratic $O(n^2)$
  - polynomial $O(n^k)$, k 1
  - exponential $O(a^n)$, n> 1

log(n)	n	n^2	n^5	2^n
1	2	4	32	4
2	4	16	1024	16
3	8	64	32768	256
4	16	256	1048576	65536
5	32	1024	33554432	4.29E+09
6	64	4096	1.07E+09	1.84E+19
7	128	16384	3.44E+10	3.4E+38
8	256	65536	1.1E+12	1.16E+77
9	512	262144	3.52E+13	1.3E+154
10	1024	1048576	1.13E+15	#NUM!

Polynomials

- Only the *dominant terms* of a polynomial matter in the long run. Lower-order terms fade to insignificance as the problem size increases.





http://www.cs.pomona.edu/~marshall/courses/2002/spring/cs50/BigO/

Comparing Running Times

- Comparing the asymptotic running time
  - an algorithm that runs in $\boldsymbol{O}(n)$ time is better than one that runs in $\boldsymbol{O}(n^2)$ time
  - similarly, $\boldsymbol{O}(\log n)$ is better than $\boldsymbol{O}(n)$
  - hierarchy of functions:
  - $\log n << n << n^2 << n^3 << 2^n$
- Caution!
  - Beware of very large constant factors. An algorithm running in time 1,000,000 n is still $\boldsymbol{O}(n)$ but might be less efficient on your data set than one running in time $2n^2$, which is $\boldsymbol{O}(n^2)$

Intuitions with Playing Cards

Summary: Analysis of Algorithms

- A method for determining the asymptotic running time of an algorithm
  - Here asymptotic means "as *n* gets very large"
- Useful for comparing algorithms
- Useful also for determining *tractability*
  - E.g., Exponential time algorithms are usually intractable.