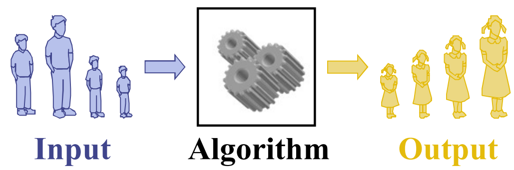
(Complexity) Analysis of Algorithms

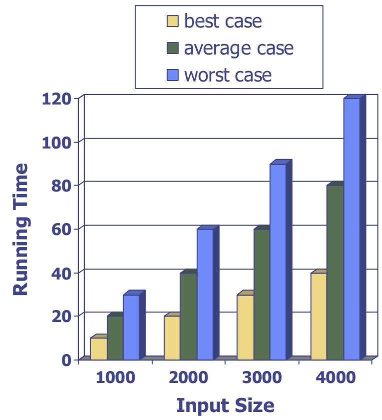


Complexity Analysis

- As true computer scientists, we need a way to compare the efficiency of algorithms
- Should we just use a stopwatch to time our programs?
  - No, different processors will cause the same program to run a different speeds.
- Instead, we will count the number of operations that must run.

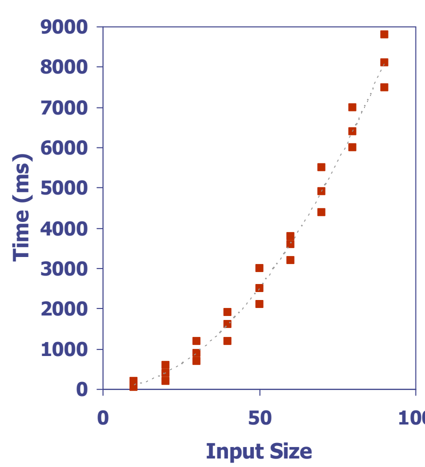
Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics



Experimental Studies

- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a method like System.currentTimeMillis() to get an accurate measure of the actual running time
- Plot the results



Limitations of Experiments

- It is necessary to implement the algorithm, which may be difficult

- Results may not be indicative of the running time on other inputs not included in the experiment.

- In order to compare two algorithms, the same hardware and software environments must be used

Theoretical Analysis

- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n.
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment

Pseudocode

- High-level description of an algorithm
- More structured than English prose
- Less detailed than a program
- Preferred notation for describing algorithms
- Hides program design issues

Example: find max element of an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

  $currentMax \leftarrow A[0]$
  for $i \leftarrow 1$ **to** $n - 1$ **do**
    if $A[i] > currentMax$ **then**
      $currentMax \leftarrow A[i]$
  return $currentMax$

Pseudocode Details

- Control flow
  - **if** … **then** … [**else** …]
  - **while** … **do** …
  - **repeat** … **until** …
  - **for** … **do** …
  - Indentation replaces braces
- Method declaration

  Algorithm *method* (*arg* [, *arg*…])
    Input …
    Output …
- Method call

  var.method (*arg* [, *arg*…])
- Return value

  return *expression*
- Expressions

  $\leftarrow$ Assignment (like $=$ in Java)

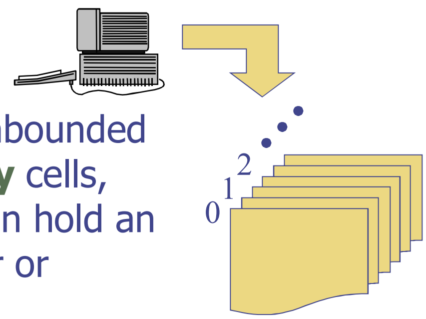
  $=$ Equality testing (like $==$ in Java)

  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

- A **CPU**

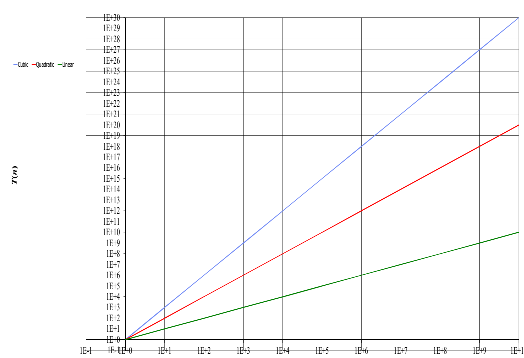
- An potentially unbounded bank of **memory** cells, each of which can hold an arbitrary number or character



- Memory cells are numbered and accessing any cell in memory takes unit time.

Seven Important Functions

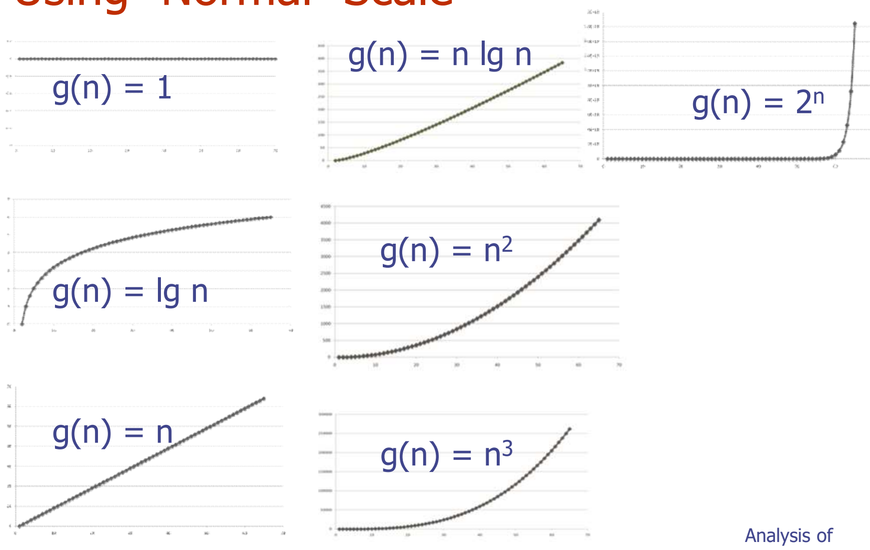
- Seven functions that often appear in algorithm analysis:
  - Constant ≈ 1
  - Logarithmic ≈ $\log n$
  - Linear ≈ n
  - N-Log-N ≈ $n \log n$
  - Quadratic ≈ n^2
  - Cubic ≈ n^3
  - Exponential ≈ 2^n

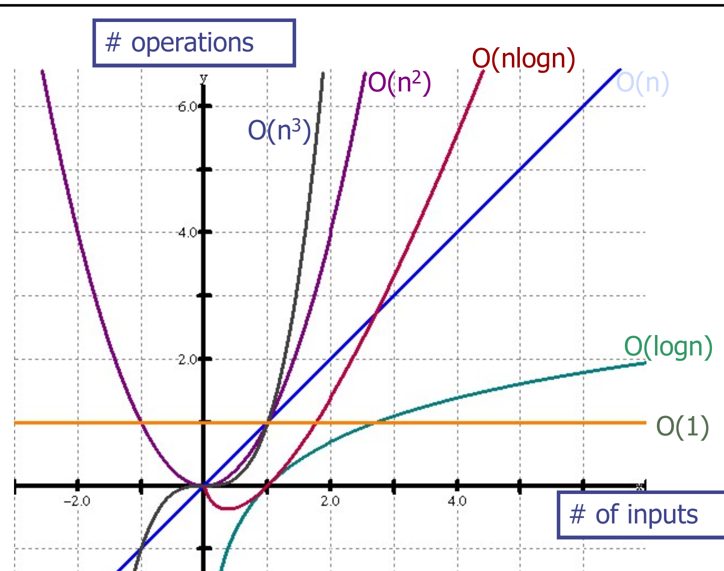


- In a log-log chart, the slope of the line corresponds to the growth rate

Functions Graphed Using "Normal" Scale

Slide by Matt Stallmann included with permission.





Algorithms are categorized by how their runtime increases in relation to the number of inputs.

Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model

- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Operations

n = number of elements

```
public static double avg(int[] a, int n){
      double sum=0;
      for(int j=0; j<n; j++)
            sum+=a[j];
      return (sum/n);
}
```

Loop runs **n** times
1 loop comparison
1 loop increment
1 operation
3n operations

Total Operations:

3n + 3

$O(n^2)$ Method

1 Loop assignme

3 operations for each of the n-1 iterations.

Total = $n(n-1)/2$ iterations

```
public static void bubble(int[]
  for (i=0; i<n-1; i++) {//how many are sorted
    for (j=0; j<n-1-i; j++)//u
      if (a[j+1] < a[j]) //com
        swap(a, j, j+1);
  }
}
```

$6[n(n-1)/2]$
$=3n(n-1)$
$=\mathbf{3n^2-3n}$
Operations from inner loop

Operation Count = $\mathbf{3n^2 - 2}$

Grouping Complexities

- So the run-time of our average function depends on the size of the array. Here are some possibilities:
  - Array size 1: runtime = 3(1) + 3 = 6
  - Array size 50: runtime = 3(50) + 3 = 153
  - Array size 1000: runtime = 3(1000) + 3 = 3003

Notice that increasing the size of the array by a constant multiple has approximately the same effect on the runtime.

Grouping Complexities

- In order to compare runtimes, it is convenient to have some grouping schema.
- Think of runtimes as a function of the number of inputs.
- How do mathematicians group functions?

Functions

- How might you “describe” these functions? What “group” do they belong to?
  - $f(x) = 7$
  - $f(x) = \log(x + 3)$
  - $f(x) = 3x + 5$
  - $f(x) = 4x^2 + 15x + 90$
  - $f(x) = 10x^3 – 30$
  - $f(x) = 2^{(3x + 3)}$

Functions

- How might you "describe" these functions? What "group" do they belong to?
  - $f(x) = 7$ **Constant**
  - $f(x) = \log(x + 3)$ **Logarithmic**
  - $f(x) = 3x + 5$ **Linear**
  - $f(x) = 4x^2 + 15x + 90$ **Quadratic**
  - $f(x) = 10x^3 - 30$ **Cubic**
  - $f(x) = 2^{(3x + 3)}$ **Exponential**

Big-O Notation

- Instead of using terms like linear, quadratic, and cubic, computer scientists use big-O notation to discuss runtimes.
- A function with linear runtime is said to be **of order n**.
- The shorthand looks like this: O(n)

Functions

- If the following are runtimes expressed as functions of the number of inputs (x), we would label them as follows:
  - $f(x) = 7$
  - $f(x) = \log(x + 3)$
  - $f(x) = 3x + 5$
  - $f(x) = 4x^2 + 15x + 90$
  - $f(x) = 10x^3 - 30$
  - $f(x) = 2^{(3x + 3)}$

Functions

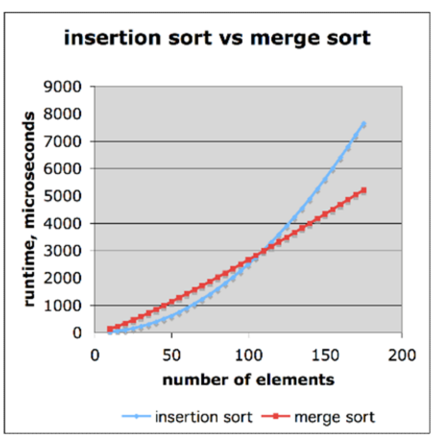
- If the following are runtimes expressed as functions of the number of inputs (x), we would label them as follows:
  - $f(x) = 7$ **O(1)**
  - $f(x) = \log(x + 3)$ **O(log n)**
  - $f(x) = 3x + 5$ **O(n)**
  - $f(x) = 4x^2 + 15x + 90$ **$O(n^2)$**
  - $f(x) = 10x^3 - 30$ **$O(n^3)$**
  - $f(x) = 2^{(3x + 3)}$ **$O(2^n)$**

The common Big-O values

- O(1) : constant
- O(log n): Logarithmic
- O(n) : Linear
- O(n logn) : n logn
- O(n^2): Quadratic
- O(n^3): Cubic
- O(2^n): exponential

Slide by Matt Stallmann included with permission.

Comparison of Two Algorithms



Insertion Sort is n^2 / 4

Merge sort is 2n log n

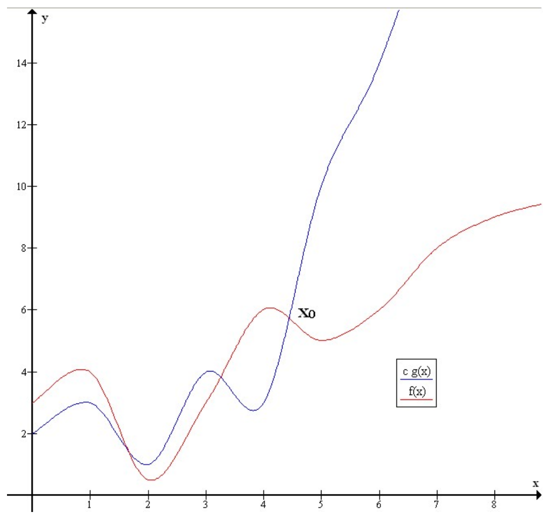
Sorting a million items:

Insertion Sort takes roughly 70 hours

Merge Sort takes roughly 40 seconds

This is a slow machine, but if 100 x as fast then it's 40 minutes versus less than 0.5 seconds

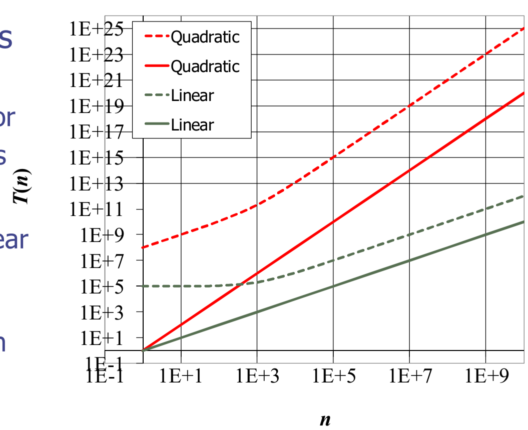
Which function is 'bigger?'



Source: Wikipedia

Constant Factors don't Matter!

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function

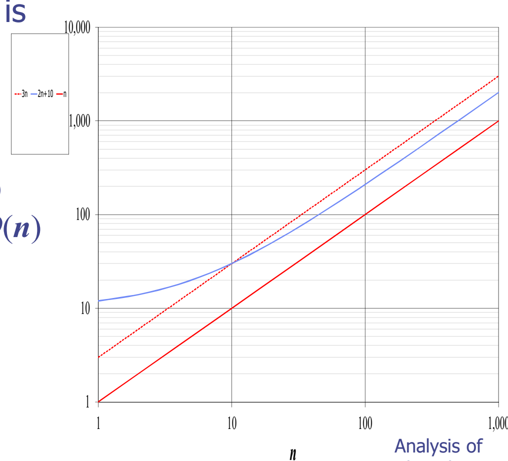


Big-Oh Notation

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

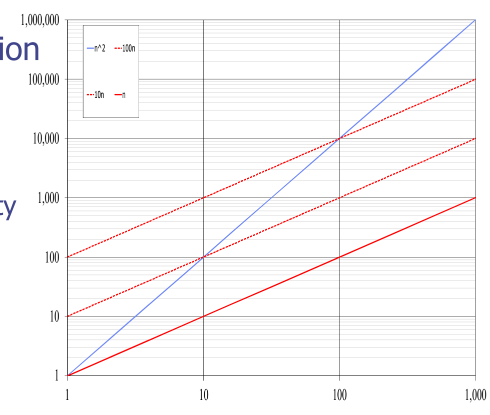
  $f(n) \leq cg(n)$ for $n \geq n_0$

- Example: $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2)\, n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$



Big-Oh Example

- Example: the function n^2 is not $O(n)$
  - $n^2 \leq cn$
  - $n \leq c$
  - The above inequality cannot be satisfied since c must be a constant



Pseudocode

Example: find max element of an array

Algorithm *arrayMax*(A, n)
  Input array A of n integers
  Output maximum element of A

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


Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important
- Assumed to take a constant amount of time in the RAM model
- Examples:
  - Evaluating an expression
  - Assigning a value to a variable
  - Indexing into an array
  - Calling a method
  - Returning from a method

Counting Primitive Operations

- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size

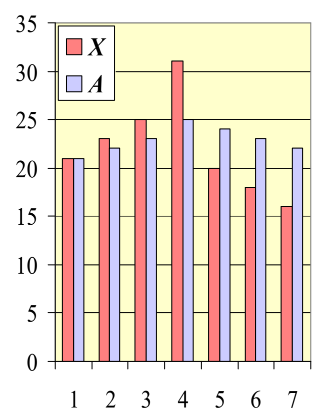
Algorithm $arrayMax(A, n)$	# operations
$currentMax \leftarrow A[0]$	2
for $i \leftarrow 1$ **to** $n - 1$ **do**	$2n$
if $A[i] > currentMax$ **then**	$2(n - 1)$
$currentMax \leftarrow A[i]$	$2(n - 1)$
{ increment counter i }	$2(n - 1)$
return $currentMax$	1
Total	$8n - 2$

Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages
- The i-th prefix average of an array X is average of the first $(i + 1)$ elements of X:

$$A[i] = (X[0] + X[1] + \ldots + X[i])/(i+1)$$

- Computing the array A of prefix averages of another array X has applications to financial analysis



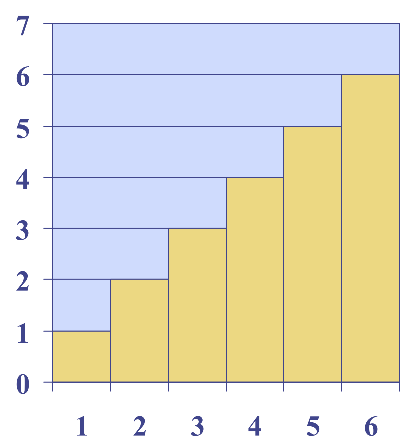
Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition

Algorithm *prefixAverages1*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	#operations
$A \leftarrow$ new array of n integers	n
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow X[0]$	n
for $j \leftarrow 1$ **to** i **do**	$1 + 2 + \ldots + (n - 1)$
$s \leftarrow s + X[j]$	$1 + 2 + \ldots + (n - 1)$
$A[i] \leftarrow s / (i + 1)$	n
return A	1

Arithmetic Progression

- The running time of *prefixAverages1* is $O(1 + 2 + \ldots + n)$
- The sum of the first n integers is $n(n + 1) / 2$
  - There is a simple visual proof of this fact
- Thus, algorithm *prefixAverages1* runs in $O(n^2)$ time



Prefix Averages (Linear)

- The following algorithm computes prefix averages in linear time by keeping a running sum

Algorithm *prefixAverages2*(X, n)	
Input array X of n integers	
Output array A of prefix averages of X	#operations
$A \leftarrow$ new array of n integers	n
$s \leftarrow 0$	1
for $i \leftarrow 0$ **to** $n - 1$ **do**	n
$s \leftarrow s + X[i]$	n
$A[i] \leftarrow s / (i + 1)$	n
return A	1

- Algorithm *prefixAverages2* runs in $O(n)$ time

General Guidelines

- The worst-case instructions determine worst-case behaviour, overall. (eg. A single n^2 statement means the whole algorithm is n^2.)
- Instant recognition:
  - Assignments/arithmetic is O(1),
  - Loops are O(n),
  - Two nested loops are O(n^2).
  - How about three nested loops?