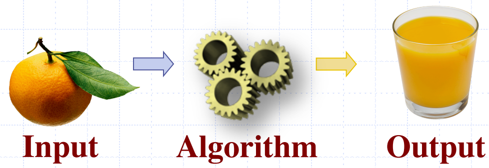
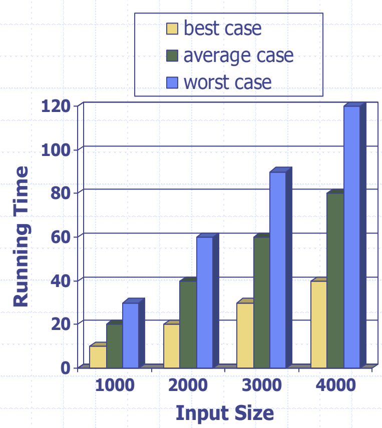
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Analysis of Algorithms



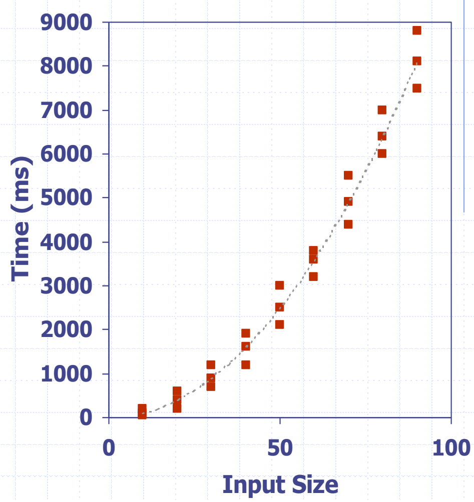
Running Time

- 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, noting the time needed:
- Plot the results

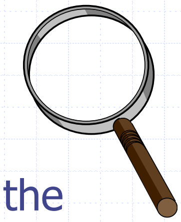



```
long startTime = System.currentTimeMillis( );      // record the starting time
/∗ (run the algorithm) ∗/
long endTime = System.currentTimeMillis( );        // record the ending time
long elapsed = endTime − startTime;                // compute the elapsed time
```


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

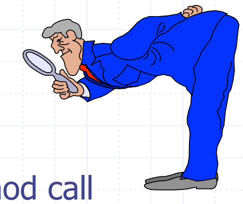


- Uses a high-level description of the algorithm instead of an implementation
- Characterizes running time as a function of the input size, n
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/ software environment

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

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

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

  return *expression*
- Expressions:

  $\leftarrow$ Assignment

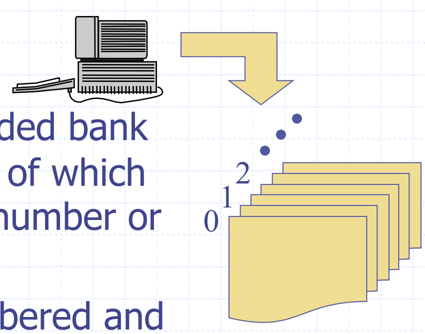
  $=$ Equality testing

  n^2 Superscripts and other mathematical formatting allowed

The Random Access Machine (RAM) Model

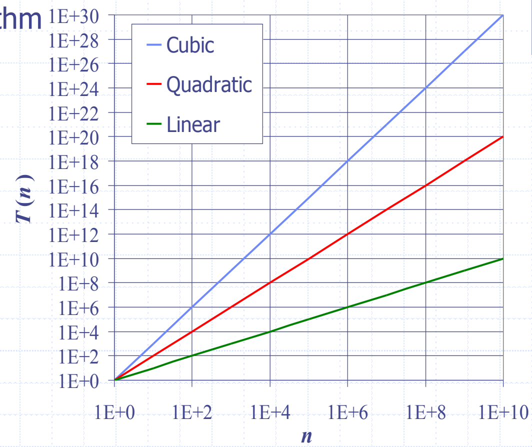
A RAM consists of

- 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



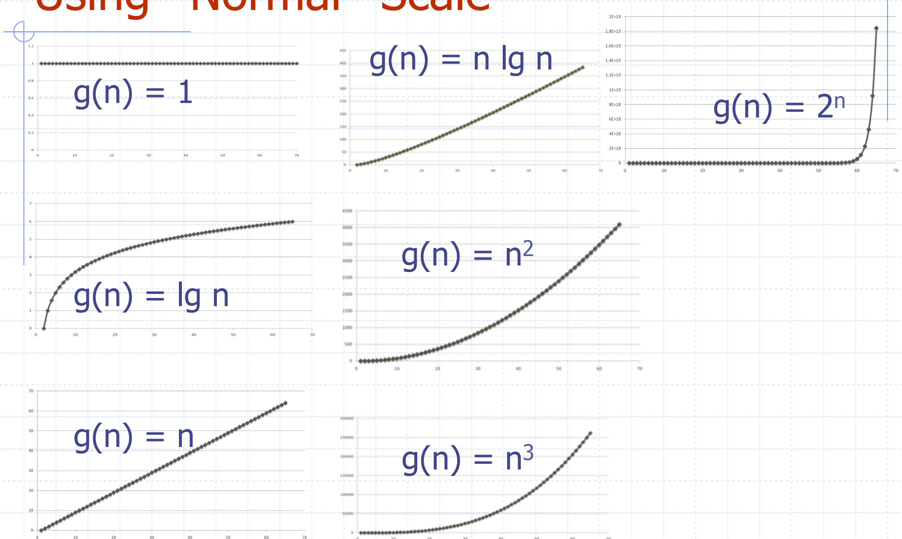
Seven Important Functions

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



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

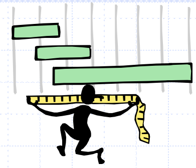
Counting Primitive Operations

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

```
1 /** Returns the maximum value of a nonempty array of numbers. */
2 public static double arrayMax(double[ ] data) {
3   int n = data.length;
4   double currentMax = data[0];            // assume first entry is biggest (for now)
5   for (int j=1; j < n; j++)               // consider all other entries
6     if (data[j] > currentMax)             // if data[j] is biggest thus far...
7       currentMax = data[j];               // record it as the current max
8   return currentMax;
9 }
```

- Step 3: 2 ops, 4: 2 ops, 5: 2n ops, 6: 2n ops, 7: 0 to n ops, 8: 1 op

Estimating Running Time



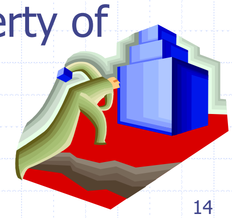
- Algorithm arrayMax executes $5n + 5$ primitive operations in the worst case, $4n + 5$ in the best case. Define:
  - a = Time taken by the fastest primitive operation
  - b = Time taken by the slowest primitive operation
- Let $T(n)$ be worst-case time of arrayMax. Then

$$a\,(4n + 5) \leq T(n) \leq b(5n + 5)$$

- Hence, the running time $T(n)$ is bounded by two linear functions

Growth Rate of Running Time

- Changing the hardware/ software environment
  - Affects $T(n)$ by a constant factor, but
  - Does not alter the growth rate of $T(n)$
- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm arrayMax



Slide by Matt Stallmann included with permission.

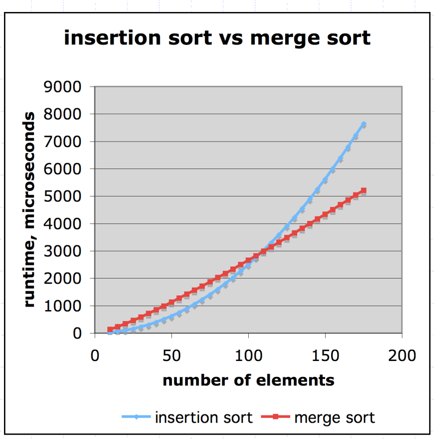
Why Growth Rate Matters

if runtime is...	time for n + 1	time for 2 n	time for 4 n
c lg n	c lg (n + 1)	c (lg n + 1)	c(lg n + 2)
c n	c (n + 1)	2c n	4c n
c n lg n	~ c n lg n + c n	2c n lg n + 2cn	4c n lg n + 4cn
c n^2	~ c n^2 + 2c n	**4c n^2**	16c n^2
c n^3	~ c n^3 + 3c n^2	8c n^3	64c n^3
c 2^n	c 2^{n+1}	c 2^{2n}	c 2^{4n}

runtime quadruples when problem size doubles

Slide by Matt Stallmann included with permission.

Comparison of Two Algorithms



insertion sort is
n^2 / 4

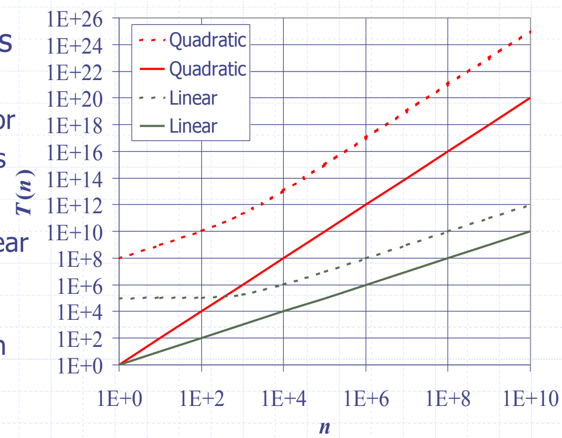
merge sort is
2 n lg n

sort a million items?
insertion sort takes roughly 70 hours
while
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

Constant Factors

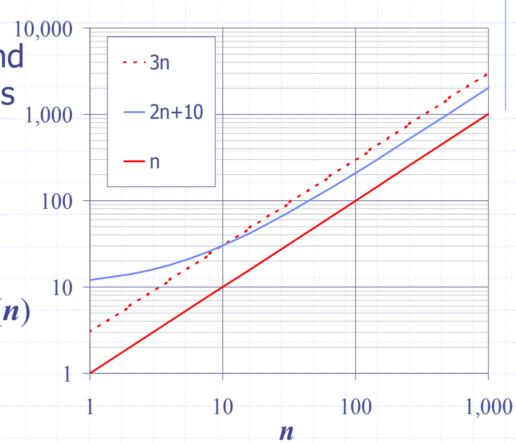
- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function



Big-Oh Notation

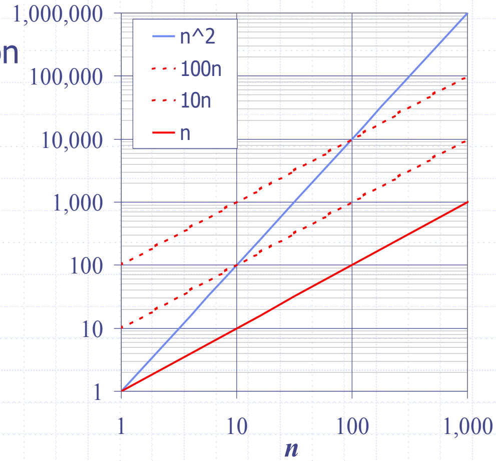
- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there are positive constants c and n_0 such that

  $f(n) \leq cg(n)$ 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 $O(n)$
  - $n^2 \le cn$
  - $n \le c$
  - The above inequality cannot be satisfied since c must be a constant



More Big-Oh Examples

- 7n - 2

  7n-2 is O(n)
  need c > 0 and $n_0 \ge 1$ such that $7n - 2 \le cn$ for $n \ge n_0$
  this is true for c = 7 and $n_0 = 1$

- $3n^3 + 20n^2 + 5$

  $3n^3 + 20n^2 + 5$ is $O(n^3)$
  need c > 0 and $n_0 \ge 1$ such that $3n^3 + 20n^2 + 5 \le cn^3$ for $n \ge n_0$
  this is true for c = 4 and $n_0 = 21$

- 3 log n + 5

  3 log n + 5 is O(log n)
  need c > 0 and $n_0 \ge 1$ such that $3 \log n + 5 \le c \log n$ for $n \ge n_0$
  this is true for c = 8 and $n_0 = 2$

Big-Oh and Growth Rate

- The big-Oh notation gives an upper bound on the growth rate of a function
- The statement "$f(n)$ is $O(g(n))$" means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$
- We can use the big-Oh notation to rank functions according to their growth rate

	$f(n)$ is $O(g(n))$	$g(n)$ is $O(f(n))$
$g(n)$ grows more	Yes	No
$f(n)$ grows more	No	Yes
Same growth	Yes	Yes

Big-Oh Rules

- If is $f(n)$ a polynomial of degree d, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- Use the simplest expression of the class
  - Say "$3n + 5$ is $O(n)$" instead of "$3n + 5$ is $O(3n)$"

Asymptotic Algorithm Analysis

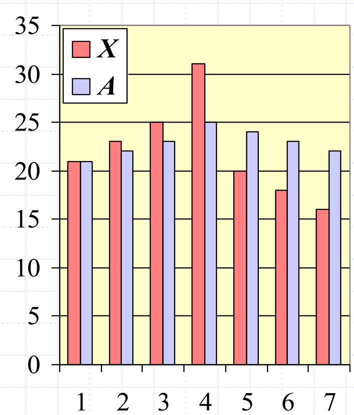
- The asymptotic analysis of an algorithm determines the running time in big-Oh notation
- To perform the asymptotic analysis
  - We find the worst-case number of primitive operations executed as a function of the input size
  - We express this function with big-Oh notation
- Example:
  - We say that algorithm arrayMax "runs in $O(n)$ time"
- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations

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



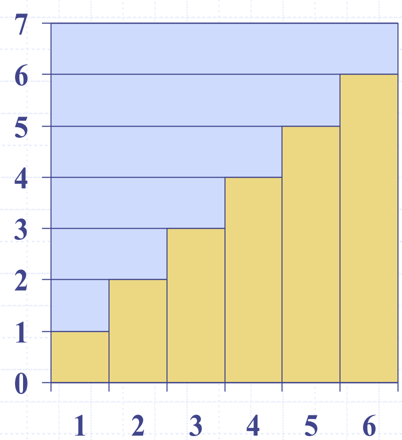
Prefix Averages (Quadratic)

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

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage1(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];                    // filled with zeros by default
  for (int j=0; j < n; j++) {
    double total = 0;                             // begin computing x[0] + ... + x[j]
    for (int i=0; i <= j; i++)
      total += x[i];
    a[j] = total / (j+1);                         // record the average
  }
  return a;
}
```


Arithmetic Progression

- The running time of prefixAverage1 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 prefixAverage1 runs in $O(n^2)$ time



Let us analyze the prefixAverage1 algorithm.

- The initialization of n = x.length at line 3 and the eventual return of a reference to array a at line 11 both execute in $O(1)$ time.
- Creating and initializing the new array, a, at line 4 can be done with in $O(n)$ time, using a constant number of primitive operations per element.
- There are two nested **for** loops, which are controlled, respectively, by counters j and i. The body of the outer loop, controlled by counter j, is executed n times, for $j = 0, \dots, n-1$. Therefore, statements total = 0 and a[j] = total / (j+1) are executed n times each. This implies that these two statements, plus the management of counter j in the loop, contribute a number of primitive operations proportional to n, that is, $O(n)$ time.
- The body of the inner loop, which is controlled by counter i, is executed $j+1$ times, depending on the current value of the outer loop counter j. Thus, statement total += x[i], in the inner loop, is executed $1+2+3+\cdots+n$ times. By recalling Proposition 4.3, we know that $1+2+3+\cdots+n = n(n+1)/2$, which implies that the statement in the inner loop contributes $O(n^2)$ time. A similar argument can be done for the primitive operations associated with maintaining counter i, which also take $O(n^2)$ time.

The running time of implementation prefixAverage1 is given by the sum of these terms. The first term is $O(1)$, the second and third terms are $O(n)$, and the fourth term is $O(n^2)$. By a simple application of Proposition 4.8, the running time of prefixAverage1 is $O(n^2)$.

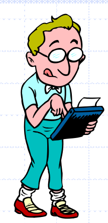
Prefix Averages 2 (Linear)

The following algorithm uses a running summation to improve the efficiency

```
/** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
public static double[ ] prefixAverage2(double[ ] x) {
  int n = x.length;
  double[ ] a = new double[n];              // filled with zeros by default
  double total = 0;                         // compute prefix sum as x[0] + x[1] + ...
  for (int j=0; j < n; j++) {
    total += x[j];                          // update prefix sum to include x[j]
    a[j] = total / (j+1);                   // compute average based on current sum
  }
  return a;
}
```

Algorithm prefixAverage2 runs in $O(n)$ time!

Math you need to Review



- Summations
- Powers
- Logarithms
- Proof techniques
- Basic probability

- Properties of powers:
  - $a^{(b+c)} = a^b a^c$
  - $a^{bc} = (a^b)^c$
  - $a^b / a^c = a^{(b-c)}$
  - $b = a^{\log_a b}$
  - $b^c = a^{c*\log_a b}$
- Properties of logarithms:
  - $\log_b(xy) = \log_b x + \log_b y$
  - $\log_b (x/y) = \log_b x - \log_b y$
  - $\log_b xa = a\log_b x$
  - $\log_b a = \log_x a/\log_x b$

The analysis of the running time of algorithm prefixAverage2 follows:

- Initializing variables n and total uses $O(1)$ time.
- Initializing the array a uses $O(n)$ time.
- There is a single **for** loop, which is controlled by counter j. The maintenance of that loop contributes a total of $O(n)$ time.
- The body of the loop is executed n times, for $j = 0, \ldots, n-1$. Thus, statements total += x[j] and a[j] = total / (j+1) are executed n times each. Since each of these statements uses $O(1)$ time per iteration, their overall contribution is $O(n)$ time.
- The eventual return of a reference to array A uses $O(1)$ time.

The running time of algorithm prefixAverage2 is given by the sum of the five terms. The first and last are $O(1)$ and the remaining three are $O(n)$. By a simple application of Proposition 4.8, the running time of prefixAverage2 is $O(n)$, which is much better than the quadratic time of algorithm prefixAverage1.

Relatives of Big-Oh

big-Omega

- f(n) is Ω(g(n)) if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

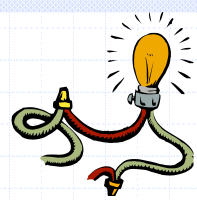
$$f(n) \geq c\, g(n) \text{ for } n \geq n_0$$

big-Theta

- f(n) is Θ(g(n)) if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that

$$c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0$$

Intuition for Asymptotic Notation



big-Oh

- f(n) is O(g(n)) if f(n) is asymptotically less than or equal to g(n)

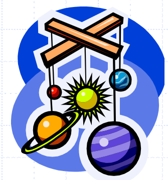
big-Omega

- f(n) is Ω(g(n)) if f(n) is asymptotically greater than or equal to g(n)

big-Theta

- f(n) is Θ(g(n)) if f(n) is asymptotically equal to g(n)

Example Uses of the Relatives of Big-Oh



- **$5n^2$ is $\Omega(n^2)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 5$ and $n_0 = 1$

- **$5n^2$ is $\Omega(n)$**

  $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \geq c\, g(n)$ for $n \geq n_0$

  let $c = 1$ and $n_0 = 1$

- **$5n^2$ is $\Theta(n^2)$**

  $f(n)$ is $\Theta(g(n))$ if it is $\Omega(n^2)$ and $O(n^2)$. We have already seen the former, for the latter recall that $f(n)$ is $O(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\mathrm{f}(n) \leq c\, g(n)$ for $n \geq n_0$

  Let $c = 5$ and $n_0 = 1$

