Algorithm Analysis

Objectives

- How to estimate the time required for an algorithm
- How to use techniques that drastically reduce the running time of an algorithm
- How to use a mathematical framework that more rigorously describe the running time of an algorithm

What is Algorithm Analysis

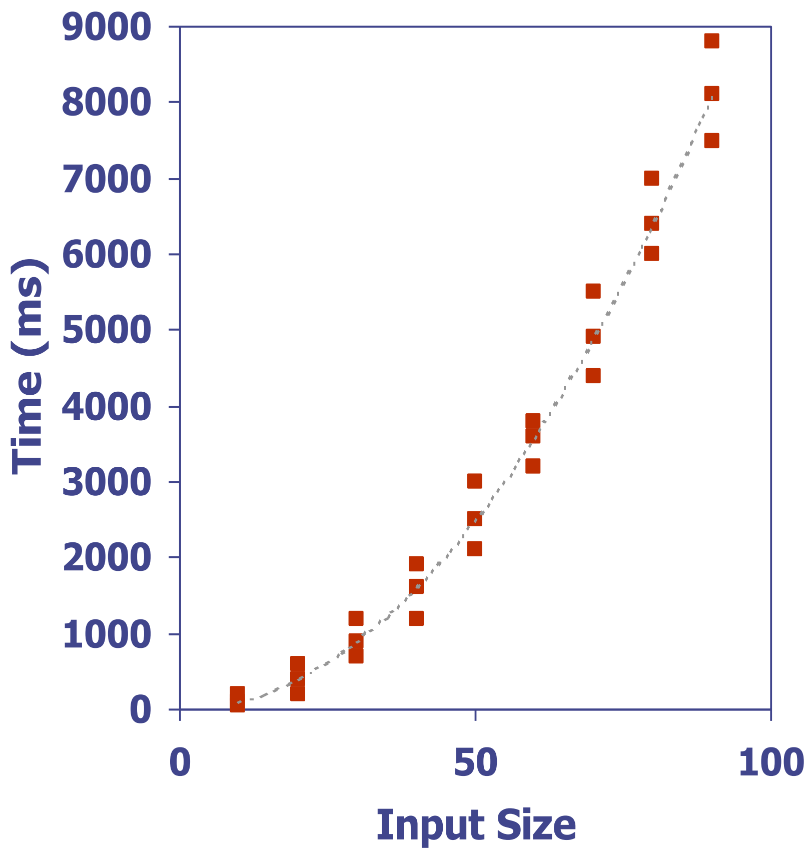
- The step to determine the amount of resources, such as time and space, that the algorithm will require to solve a problem.

Algorithm Efficiency and Big-Oh

- Getting a precise measure of the performance of an algorithm is difficult
- Big-Oh notation expresses the performance of an algorithm as a function of the number of items to be processed
- This permits algorithms to be compared for efficiency
- It does so independently of the underlying compiler

Experimental Studies

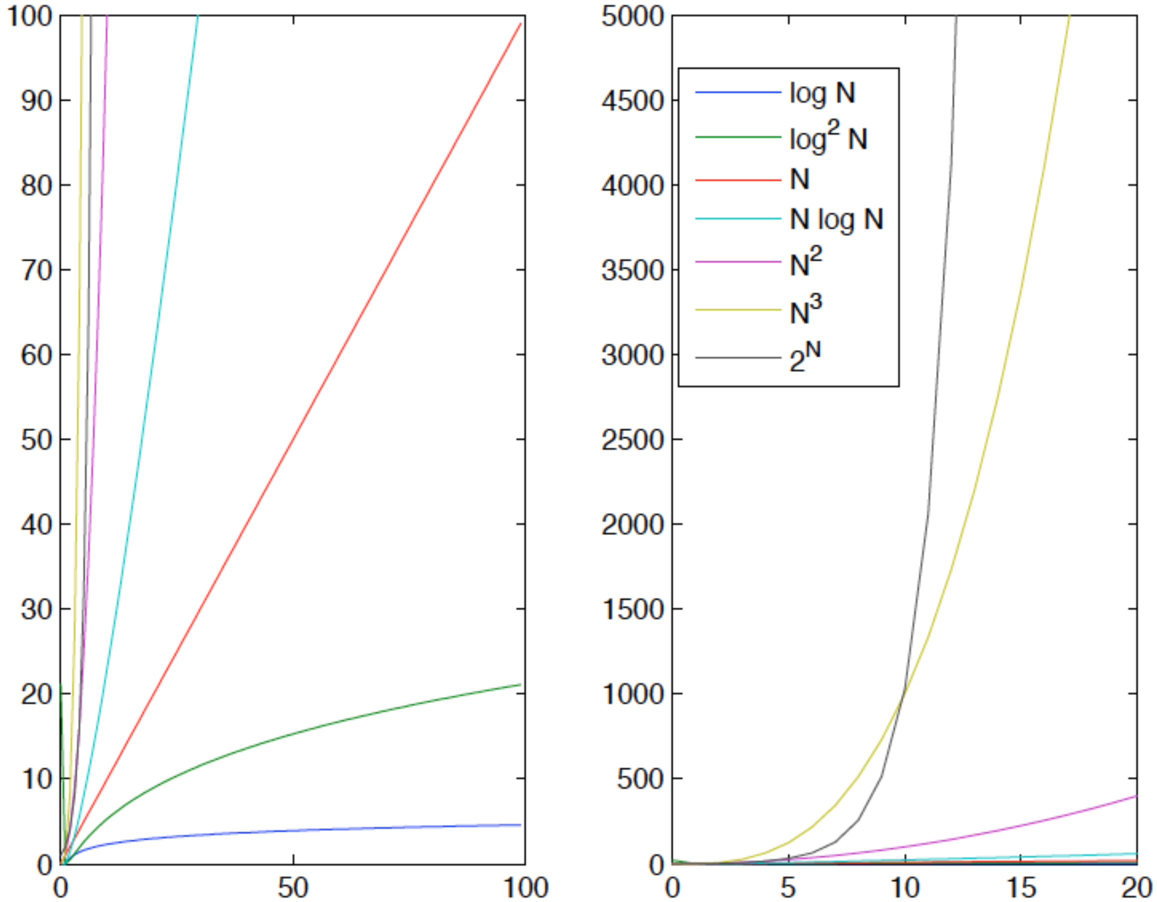
- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition
- Use a function, like the built-in clock() function, to get an accurate measure of the actual running time
- Plot the results



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

Common Running Time Functions



Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	*N* log *N*
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

Linear Growth Rate

- Processing time increases in proportion to the number of inputs n

```java
public static int search(int[] x, int target) {
  for(int i=0; i<x.length; i++) {
    if (x[i]==target)
      return i;
  }
  return -1; // target not found
}
```

  - Let n be x.length
  - Target not present: **for** loop will execute n times
  - Target present: **for** loop will execute (on average) (n + 1)/2 times
  - Therefore, the total execution time is directly proportional to n
  - This is described as a growth rate of order n or O(n)

n * m Growth Rate

- Processing time can be dependent on two different inputs n and m

```
public static boolean areDifferent(int[] x, int[] y) {
  for(int i=0; i<x.length; i++) {
    if (search(y, x[i]) != -1)
      return false;
  }
  return true;
}
```

  - The **for** loop will execute x.length times
  - But it will call **search**, which will execute y.length times
  - The total execution time is proportional to (x.length * y.length)
  - The growth rate has an order of n m or O(n m)

Quadratic Growth Rate

- Processing time proportional to square of number of inputs n

```
public static boolean areUnique(int[] x) {
  for(int i=0; i<x.length; i++) {
    for(int j=0; j<x.length; j++) {
      if (i != j && x[i] == x[j])
        return false;
    }
  }
  return true;
}
```

  - The **for** loop with i as index will execute x.length times
  - The **for** loop with j as index will execute x.length times
  - The total number of times the inner loop will execute is (x.length)^2
  - I The growth rate has an order of n^2 or O(n^2)

Running Time Example

Nested loop executes Simple Statement n^2 times

Loop executes 5 Simple Statements n times

25 Simple Statements are executed

Conclusion: the relationship between processing time and n (number of data items processed) is:
$\mathcal{T}(n) = n^2 + 5n + 25$

```
for (int i = 0; i < n; i++) {
 for (int j = 0; j < n; j++) {
         Simple Statement
  }
}
for (int i = 0; i < n; i++) {
   Simple Statement 1
   Simple Statement 2
   Simple Statement 3
   Simple Statement 4
   Simple Statement 5
}
Simple Statement 6
Simple Statement 7
...
Simple Statement 30
```

Why Not Compare Algorithms Using T ?

- Comparing algorithms based on running time T (n) is not convenient
- The formulas describing T can be complicated
- It is better to bound T by another function
- That way algorithms can be compared by comparing their bounds

- The growth rate of f (n) will be determined by the fastest growing term, which is the one with the largest exponent
  - In the example, T (n) = n2 + 5n + 25 has growth order $O(n^2)$
- In general, it is safe to ignore all constants and to drop the lower-order terms when determining the order of magnitude

- The Big-oh notation O(f (n)) for T means “roughly" that f (n) is a bound for T , for large enough values of n

Big-Oh Notation

- We adopt special notation to define upper bounds and lower bounds on functions
- The O() in the previous examples can be thought of as an abbreviation of “order of magnitude"
- A simple way to determine the big-O notation of an algorithm is to look at the loops and to
  - see whether the loops are nested
  - consider the number of times a loop is executed
- Assuming a loop body consists only of simple statements and executes at most n times,
  - a single loop is O (n)
  - a pair of nested loops is O (n^2)
  - a nested pair of loops inside another is O (n^3)
  - and so on . . .

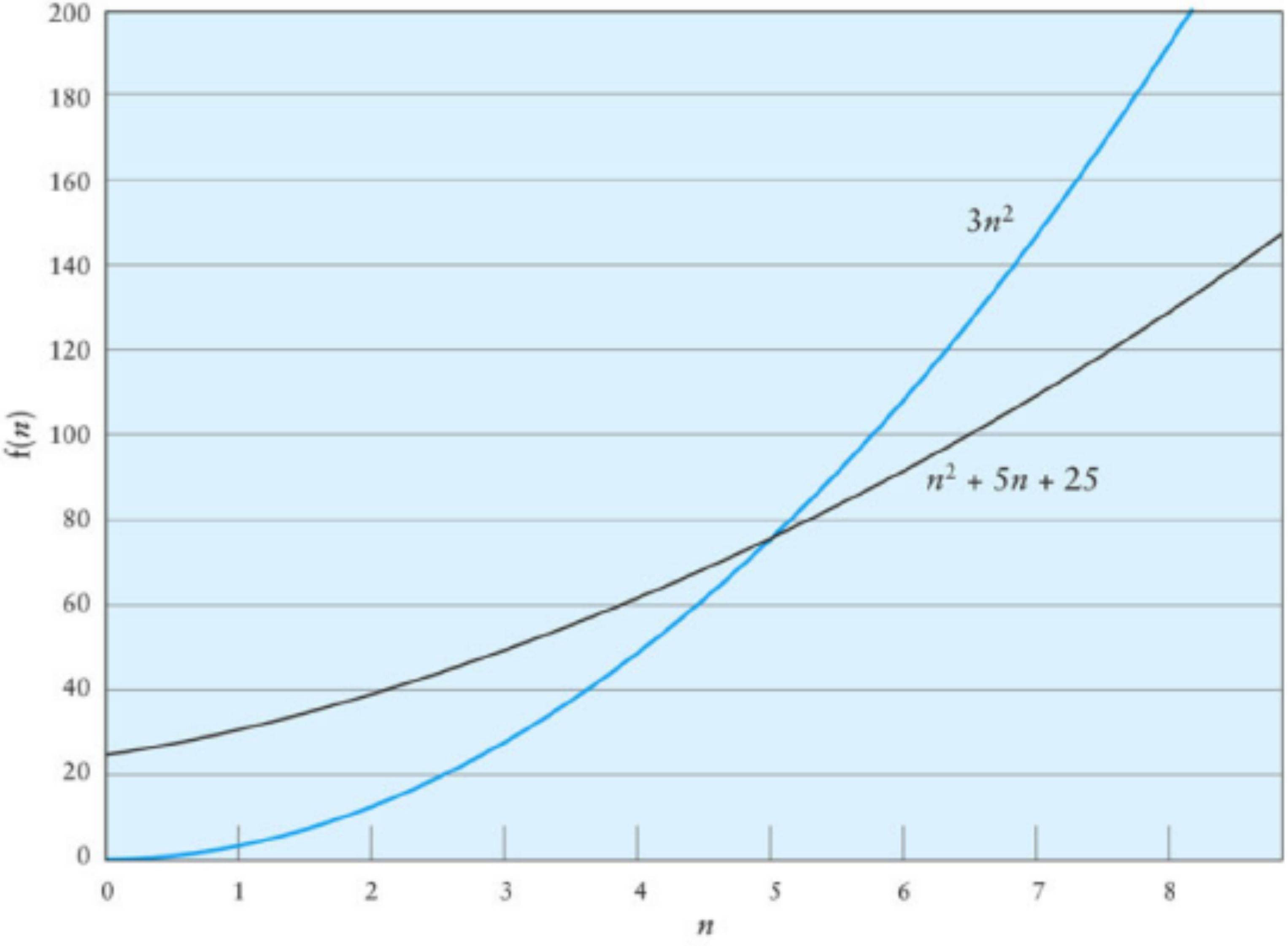
Big-Oh Notation

- You must also examine the number of times a loop is executed

```
for(int i=1; i < x.length; i *= 2) {
   // Do something with x[i]
}
```

  - The loop body will execute k times, with i having the following values: 1; 2; 4; 8; 16; ...; 2^k, until 2^k is greater or equal to x.length
  - Lets deduce the value of k
    - k = $\log_2$(x.length)
    - Thus we say the loop is $O(\log_2 n)$
  - Logarithmic functions grow slowly as the number of data items n increases

Big-Oh Example



Formal Definition

- Big-oh: For N greater than some constant N_0, There exists some constant c such that c*f(N) bounds T(N), i.e. T(N)<=c*f(N), when N>=N_0
- Alternately, O(f(N)) can be thought of as meaning

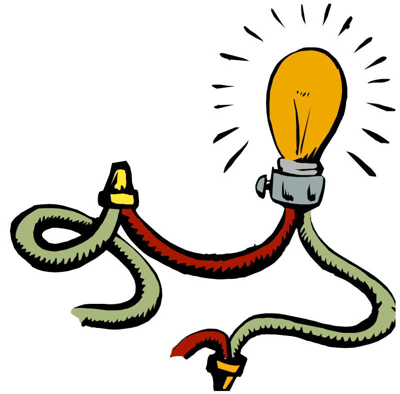
$$T(N) = O(f(N)) \leftarrow \lim_{N \to \infty} f(N) \geq \lim_{N \to \infty} T(N)$$

- Big-Oh notation is also referred to as asymptotic analysis, for this reason

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
  - $f(n) \geq c \bullet g(n)$ 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 $c' \bullet g(n) \leq f(n) \leq c'' \bullet g(n)$ for $n \geq n_0$
- **little-oh**
  - f(n) is o(g(n)) if, for any constant c > 0, there is an integer constant $n_0 \geq 0$ such that $f(n) \leq c \bullet g(n)$ for $n \geq n_0$
- **little-omega**
  - f(n) is ω(g(n)) if, for any constant c > 0, there is an integer constant $n_0 \geq 0$ such that $f(n) \geq c \bullet g(n)$ 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)

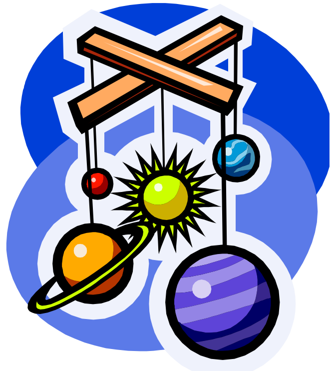
little-oh

- f(n) is o(g(n)) if f(n) is asymptotically **strictly less** than g(n)

little-omega

- f(n) is ω(g(n)) if is asymptotically **strictly greater** than 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 $f(n) \geq c \cdot g(n)$ 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 $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

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

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

  f(n) is $\omega(g(n))$ if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  need $5n_0^2 \geq c \cdot n_0 \rightarrow$ given c, the n_0 that satisfies this is $n_0 \geq c/5 \geq 0$

BACKUP SLIDES

Math Background: Exponents

- $X^A X^B = X^{A+B}$
- $X^A / X^B = X^{A-B}$
- $(X^A)^B = X^{AB}$
- $X^N + X^N = 2X^N \mathrel{!=} X^{2N}$
- $2^N + 2^N = 2^{N+1}$

Math Background: Logarithms

- $X^A = B$ iff $\log_X B = A$
- $\log_A B = \log_C B / \log_C A$; $A, B, C > 0, A \neq 1$
- $\log AB = \log A + \log B$; $A, B > 0$

Math Background: Series

$$\sum_{i=0}^{N} 2^i = 2^{N+1} - 1$$

$$\sum_{i=0}^{N} A^i = \frac{A^{N+1} - 1}{A - 1}$$

$$\sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2}$$

$$\sum_{i=1}^{N} i^2 = \frac{N(N+1)(2N+1)}{6} \approx \frac{N^3}{3}$$

Math Background: Proofs

- Proof by Induction:
  - Prove base case
  - Inductive hypothesis. Prove claim for current state assuming truth in previous state
- Proof by Contradiction: assume claim is false.
  - Show that assumption leads to contradiction