Analysis Tools
What does it mean for an algorithm to be efficient?
Characterizing Performance

Running time
Memory usage

Depends partly on hardware platform, implementation, operating system, etc.
Goals of Characterization

Predict performance on any input

Compare relative performance of algorithms/data structures

Do it without having to implement first
Experimental Analysis

Implement data structure and algorithm

Run on many inputs of different sizes and other characteristics
  • Record running time, memory usage, etc.

Perform statistical analysis
  • Plot data, find a best fitting curve
Limitations of Experimental Analysis

Requires implementations of each algorithm/data structure to be compared

Fair comparison must be on same hardware/software platform

Difficult to make good predictions
  • Test inputs may not fully characterize all possible inputs
  • Extrapolation of input sizes may not be accurate
    —Difficult to know what input range must be tested
Express algorithm as pseudo-code

Count maximum number of primitive operations

• As function of input size, $n$

Report analysis results in “Big-Oh” notation
Pseudo-code

Looks like generic high-level language
Designed for human readability
Express algorithm concisely

• But don’t skip important details
Pseudo-code Example

Algorithm: arrayMax(A,n)
Input: An array A storing n >=1 integers
Output: Maximum element value in A

1. currentMax ← A[0]
2. for i ← 1 to n-1 do
   1. if currentMax < A[i] then
      1. currentMax ← A[i]
3. return currentMax
Primitive Operations

Determine “running time” of pseudo-code algorithm (as a function of input size)

Assume each operation takes same time or some constant multiple

Just count operations

• assignment
• procedure call, return
• arithmetic operation, comparison
• indexing array, following reference
Counting Operations Example

currentMax ← A[0] 2 ops
for i ← 1 to n-1 do 2n-2 ops
    if currentMax < A[i] then 2(n-1) ops
        currentMax ← A[i] < 2n ops
return currentMax 1 op

Total operations: 6n-8n ops

Exact constants will not matter if we do asymptotic analysis
Asymptotic Analysis

Provides bounds on worst (or average) case behavior of algorithm

Emphasizes behavior “in the limit”, as $n$ grows to be very large

Constant factors are ignored

Is a way of “categorizing” algorithms
“Big-Oh” Notation

Given two functions, \( f(n) \) and \( g(n) \),

\( f(n) \) is \( O(g(n)) \) if there are constants \( c > 0 \) and \( n_0 \geq 1 \) such that \( f(n) \leq cg(n) \) for all \( n \geq n_0 \)

- “\( f(n) \) is order \( g(n) \)”

\( g(n) \) provides upper bound on \( f(n) \)

- in some sense, \( f(n) \leq g(n) \)
Analysis of maxArray

Let's say number of operations was exactly

\[ 8n = f(n) \]

Choose \( c=8 \) and \( n_0=1 \), and try \( g(n) = n \)

\( f(n) \) is \( O(n) \)
Proving Big-Oh by Example

1. Choose likely value for $c$
2. Find intersection of $f(n)$ and $cg(n)$
   - set equal and find roots
3. Choose largest intersection as $n_0$
4. Show that $cg(n) > f(n)$ for all $n > n_0$
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
Relatives of Big-Oh

big-Omega

- \( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that
  \[ f(n) \geq c \cdot g(n) \text{ for } n \geq n_0 \]

big-Theta

- \( f(n) \) is \( \Theta(g(n)) \) if there are constants \( c' > 0 \) and \( c'' > 0 \) and an integer constant \( n_0 \geq 1 \) such that
  \[ c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n) \text{ for } n \geq n_0 \]
Other useful notations

big-Oh \( O \) \( \leq \)
  - “upper bound”

little-oh \( o \) \( < \)

little-omega \( \omega \) \( > \)

big-Omega \( \Omega \) \( \geq \)
  - “lower bound”
  - \( f(n) \) is \( \Omega(g(n)) \) if \( g(n) \) is \( O(f(n)) \)

big-Theta \( \Theta \) \( = \)

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Greg Hager
Seven Important Functions

\[ 1 < \log n < n^{1/2} < n < n \log n < n^2 < n^2 \log n < n^3 < n^k < 2^n \]

- in increasing order
Seven Important Functions

Seven functions that often appear in algorithm analysis:

- Constant \( \approx 1 \)
- 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 of the function.
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

$$f(n) \leq cg(n) \quad \text{for } n \geq n_0$$

Example: $2n + 10$ is $O(n)$

- $2n + 10 \leq cn$
- $(c - 2) n \geq 10$
- $n \geq 10/(c - 2)$
- Pick $c = 3$ and $n_0 = 10$
More Big-Oh Examples

7n-2

7n-2 is O(n)

need c > 0 and n₀ ≥ 1 such that 7n-2 ≤ c•n for n ≥ n₀
this is true for c = 7 and n₀ = 1

3n³ + 20n² + 5

3n³ + 20n² + 5 is O(n³)

need c > 0 and n₀ ≥ 1 such that 3n³ + 20n² + 5 ≤ c•n³ for n ≥ n₀
this is true for c = 4 and n₀ = 21

3 log n + 5

3 log n + 5 is O(log n)

need c > 0 and n₀ ≥ 1 such that 3 log n + 5 ≤ c•log n for n ≥ n₀
this is true for c = 8 and n₀ = 2
Some exercises

1. Show any nth order polynomial is of order $a^n$
   interesting to plot what happens …

2. Show that $n \log n$ is $O(n^2)$, but not $\Omega(n^2)$ and thus not $\Theta(n^2)$

3. What is the complexity of recursive fibonacci?
   
   $\text{fib}(n) :=$
   
   if $(n \leq 1)$ return 1
   
   else return $\text{fib}(n-1) + \text{fib}(n-2)$
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 $O(3n)$”
Some General Rules for Analysis

1. Consecutive Statements: $T(s1) + T(s2); O(??)$
2. If/Else: Max($T(s1), T(s2)$) + $T(if); O(??)$
3. For loop: $T(body) \times N; O(??)$
4. Nested loops: Apply the above rule; $O(??)$
Math you need to Review

properties of logarithms:

\[ \log_b(xy) = \log_b x + \log_b y \]
\[ \log_b(x/y) = \log_b x - \log_b y \]
\[ \log_b a^x = x \log_b a \]
\[ \log_b a = \frac{\log_x a}{\log_x b} \]

properties of exponentials:

\[ a^{(b+c)} = a^b a^c \]
\[ a^{bc} = (a^b)^c \]
\[ a^b / a^c = a^{(b-c)} \]
\[ b = a^{\log_a b} \]
\[ b^c = a^{c \log_a b} \]

Summations
Logarithms and Exponents
Proof techniques
Basic probability

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Greg Hager
Some other useful formulas

\[ \sum_{i=1}^{n} i = \frac{n(n + 1)}{2} \]  

nested loop

\[ f(n) = \sum_{i=1}^{d} a_i n^i \]  
polynomial of order d

\[ \sum_{i=1}^{n} a^i = \frac{a^{n+1} - 1}{a - 1} \]  
geometric series
Another Example: Prefix Average

Goal: create an algorithm that, given an array of \( n \) numbers, computes the average of every prefix of the array.

e.g. for 1 2 3 4 5 yields

1 1.5 2 2.5 3
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 $X$  
  
  $A$ ← new array of $n$ integers

  for $i$ ← 0 to $n - 1$ do
    $s$ ← $X[0]$
    for $j$ ← 1 to $i$ do
      $s$ ← $s + X[j]$
    $A[i]$ ← $s / (i + 1)$

  return $A$
Arithmetic Progression

The running time of \textit{prefixAverages1} is $O(1 + 2 + \ldots + n)$

The sum of the first $n$ integers is $\frac{n(n + 1)}{2}$

- There is a simple visual proof of this fact

Thus, algorithm \textit{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  \textit{prefixAverages2}(X, n) \\
\textbf{Input} array \textit{X} of \(n\) integers \\
\textbf{Output} array \textit{A} of prefix averages of \textit{X} \\
\text{\textit{A} } \leftarrow \text{ new array of } n \text{ integers} \\
\text{\textit{s} } \leftarrow \text{ 0} \\
\text{\textbf{for} } i \leftarrow 0 \text{ \textbf{to} } n - 1 \text{ \textbf{do}} \\
\text{\textit{s} } \leftarrow \text{ \textit{s} } + \textit{X}[i] \\
\text{\textit{A}[i] } \leftarrow \text{ \textit{s} } / (i + 1) \\
\text{\textbf{return} } \textit{A} \\
1

Algorithm \textit{prefixAverages2} runs in \(O(n)\) time
Max Subsequence

Book talks about max subsequence sum … also worth reviewing to see how algorithm design can have a huge impact on performance
/**
 * Performs the standard binary search.
 * @return index where item is found, or -1 if not found
 */
public static <AnyType extends Comparable<? super AnyType>>
   int binarySearch( AnyType [ ] a, AnyType x )
{
    int low = 0, high = a.length - 1;

    while( low <= high )
    {
      int mid = ( low + high ) / 2;

      if( a[ mid ].compareTo( x ) < 0 )
        low = mid + 1;
      else if( a[ mid ].compareTo( x ) > 0 )
        high = mid - 1;
      else
        return mid; // Found
    }
    return NOT_FOUND; // NOT_FOUND is defined as -1
}
Euclids Algorithm for GCD

while (n != 0)
{
    long rem = m % n;
    m = n;
    n = rem;
}

We can prove this runs in $O(\log n)$ time based on this theorem

If $M > N$, then $M \mod N < M/2$
Recursive Power

public static long pow(long x, int n) {
    if (n == 0)
        return 1;
    if (n == 1)
        return x;
    if (isEven ( n ))
        return pow (x*x,n/2)
    else
        return pow (x*x,n/2)*x;
}
Summary

Basics of asymptotic analysis

Big O and relatives

Code analysis

Some surrounding mathematical rules that are useful for analysis
An algorithm is a step-by-step procedure for solving a problem in a finite amount of 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 (§3.2)

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 ← A[0]
for i ← 1 to n - 1 do
  if A[i] > currentMax then
    currentMax ← A[i]
return currentMax
Pseudocode Details

Control flow
- if ... then ... [else ...]
- while ... do ...
- repeat ... until ...
- for ... do ...
- Indentation replaces braces

Method declaration

\textbf{Algorithm method (arg [, arg...])}

Input ...
Output ...

Method call

\texttt{var.method (arg [, arg...])}

Return value

\texttt{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.
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 (§3.4)

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 $arrayMax(A, n)$

- $currentMax \leftarrow A[0]$  
- for $i \leftarrow 1$ to $n - 1$ do  
  - if $A[i] > currentMax$ then  
    - $currentMax \leftarrow A[i]$  
  - \{ increment counter $i$ \}  
- return $currentMax$

# operations

- 2
- $2n$
- $2(n - 1)$
- $2(n - 1)$
- $2(n - 1)$
- 1
- Total $8n - 2$
Algorithm \texttt{arrayMax} executes $8n - 2$ primitive operations in the worst case. Define:

- $a = \text{Time taken by the fastest primitive operation}$
- $b = \text{Time taken by the slowest primitive operation}$

Let $T(n)$ be worst-case time of \texttt{arrayMax}. Then

$$a (8n - 2) \leq T(n) \leq b(8n - 2)$$

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 alg $arrayMax$

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Greg Hager
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.

<table>
<thead>
<tr>
<th></th>
<th>( f(n) ) is ( O(g(n)) )</th>
<th>( g(n) ) is ( O(f(n)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( g(n) ) grows more</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>( f(n) ) grows more</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Same growth</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Johns Hopkins Department of Computer Science
Course 600.226: Data Structures, Professor: Greg Hager
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 determine that algorithm $arrayMax$ executes at most $8n - 2$ primitive operations.
- 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.
Math you need to Review

Summations

Logarithms and Exponents

properties of logarithms:
\[ \log_b(xy) = \log_b x + \log_b y \]
\[ \log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y \]
\[ \log_b x^a = a \log_b x \]
\[ \log_b a = \frac{\log_x a}{\log_x b} \]

properties of exponentials:
\[ a^{(b+c)} = a^b a^c \]
\[ a^{bc} = (a^b)^c \]
\[ a^b / a^c = a^{(b-c)} \]
\[ b = a^{\log_a b} \]
\[ c = a^{\log_b a} \]

Proof techniques

Basic probability
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 \( \Omega(g(n)) \) if \( f(n) \) is asymptotically greater than or equal to \( g(n) \)

big-Theta

- \( f(n) \) is \( \Theta(g(n)) \) 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 \(f(n) \geq c \cdot g(n)\) for \(n \geq n_0\)

  \text{let } c = 5 \text{ 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 \(n \geq n_0\)

  \text{let } c = 1 \text{ 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 \(f(n) \leq c \cdot g(n)\) for \(n \geq n_0\)

  \text{Let } c = 5 \text{ and } n_0 = 1