Analysis of Algorithms (Chapter 4)

COMP53
Oct 1, 2007
An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

Algorithms transform input objects into output objects.
Analysis of Algorithms

Analysis of algorithms is the process of determining the resources used by an algorithm in terms of time and space.

Time is typically more interesting than space.
Running Time

• The *running time* of an algorithm changes with the size of the input.

• We try to characterize the relationship between the input size and the algorithm running time by a characteristic function.

• The input size is usually referred to as *n*. 
Running Time Example

Figure 1

x: input size, y: running time
Best, Worst and Average Case

• For a particular problem size \( n \), we can find:
  • Best case: the input that can be solved the fastest
  • Worst case: the input that will take the longest
  • Average case: average time for all inputs of the same size.
Best, Worst and Average Case

- Average case time is often difficult to determine.
- Best case is often trivial and misleading.
- We’ll focus on worst case running time analysis.
  - Easier to analyze
  - Sufficient for common applications
Methods of Analysis

• *Experimental Studies*: Run experiments on implementations of algorithms and record actual time or operation counts.

• *Theoretical Analysis*: Determine time or operation counts from mathematical analysis of the algorithm
  – doesn’t require an implementation (or even a computer)
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 $A$

```plaintext
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
  Algorithm method (arg [, arg...])
  Input ...
  Output ...

• Method call
  var.method (arg [, arg...])

• Return value
  return expression

• Expressions
  ← 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

- A 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 $\approx 1$
  – 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
Orders of Growth

- Cubic
- Quadratic
- Linear
Basic Operations

• Rather than worrying about actual time, theoretical analysis estimates time by counting some basic operation.

• Actual time is not important, since it varies based on hardware and software in use.

• If the basic operation count gives an accurate estimate of the algorithm running time, then we can use it to compare different algorithms for the same problem.
Basic Operations

• Are identifiable in the pseudocode
• Are largely independent of any programming language
• Are assumed to take a constant amount of time in the RAM model

• Examples:
  – Comparing two values
  – Multiplying two values
  – Assigning a value to a variable
  – Indexing into an array
  – Calling a subroutine
Counting Operations

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

```
Algorithm arrayMax(A, n)

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

# operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>currentMax ← A[0]</td>
<td>2</td>
</tr>
<tr>
<td>for i ← 1 to n − 1 do</td>
<td>2n</td>
</tr>
<tr>
<td>if A[i] &gt; currentMax then</td>
<td>2(n − 1)</td>
</tr>
<tr>
<td>currentMax ← A[i]</td>
<td>2(n − 1)</td>
</tr>
<tr>
<td>{ increment counter i }</td>
<td>2(n − 1)</td>
</tr>
<tr>
<td>return currentMax</td>
<td>1</td>
</tr>
</tbody>
</table>

Total: \(8n - 2\)
Estimating Running Time

• Algorithm \textit{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 \textit{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 algorithm `arrayMax`
Math We’ll Need

• Summations
• Logarithms and Exponents

properties of logarithms:

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

properties of exponentials:

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

• Proof techniques
• Basic probability

See Appendix A for Useful Mathematical Facts
Logarithms and Exponents

- Logarithms approximate the number of digits in a number, given a particular base.
- Logarithms are the inverse of exponents.
- \( \log_{10}(100,000) = 5, \quad 10^5 = 100,000 \)
- \( \log_2(256) = 8, \quad 2^8 = 256 \)
- \( \log_8(4096) = 4, \quad 8^4 = 4096 \)
Exponential Growth

Figure 1

- Linear scale
- Logarithmic scale

$x: n, y: 2^n$
Examples: linear and quadratic

- 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
Examples: linear and quadratic

blue: $10^2n + 10^5$  
green: $10^5n^2 + 10^8n$
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
More Big-Oh Examples

- $7n-2$
  
  $7n-2$ is $O(n)$
  
  need $c > 0$ and $n_0 \geq 1$ such that $7n-2 \leq c \cdot n$ for $n \geq n_0$
  
  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 $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ for $n \geq n_0$
  
  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 $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ for $n \geq n_0$
  
  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

<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>
Big-Oh Rules

• If is \( f(n) \) a polynomial of degree \( d \), then \( f(n) \) is \( O(n^d) \)
  1. Drop lower-order terms
  2. Drop constant factors

• Use the smallest possible class of functions
  – Say “2\( n \) is \( O(n) \)” instead of “2\( n \) is \( O(n^2) \)”

• Use the simplest expression of the class
  – Say “3\( n + 5 \) is \( O(n) \)” instead of “3\( n + 5 \) 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 basic operations as a function of the input size
  – We express this function with big-Oh notation

• Example:
  – We determine that algorithm \texttt{arrayMax} executes at most $8n - 2$ primitive operations
  – We say that algorithm \texttt{arrayMax} “runs in $O(n)$ time”
Example: Sequential Search

• Algorithm SequentialSearch(A, x):
  Input: An array A and a target x
  Output: The position of x in A
  for i ← 0 to n-1 do
    if A[i] = x
      return i
  return -1

• Worst case: n comparisons
• Sequential search is O(n)
Example: Insertion Sort

**Algorithm** InsertionSort(A):

*Input:* An array $A$ of $n$ comparable elements

*Output:* The array $A$ with elements rearranged in non-decreasing order

```
for $i \leftarrow 1$ to $n-1$ do
    \{Insert $A[i]$ at its proper location in $A[0], A[1], \ldots, A[i-1]$\}
    $cur \leftarrow A[i]$
    $j \leftarrow i - 1$
    while $j \geq 0$ and $a[j] > cur$ do
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j - 1$
    $A[j+1] \leftarrow cur \{cur$ is now in the right place$\}$
```

**Code Fragment 3.6:** Intermediate-level description of the insertion-sort algorithm.
Example: Insertion Sort

• Outer loop: \( i \leftarrow 1 \) to \( n-1 \)
• Inner loop (worst case): \( j \leftarrow i-1 \) down to \( 0 \)
• Worst case comparisons:

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

• insertion sort is \( O(n^2) \)