CS 341: Algorithms
Module 2: Analyzing Algorithms

Armin Jamshidpey, Eugene Zima

Based on lecture notes by many previous CS 341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2019

Algorithms

Define "Algorithm" ...
In this course, we study the design and analysis of algorithms. "Analysis" refers to mathematical techniques for establishing both the correctness and efficiency of algorithms. Correctness: The level of detail required in a proof of correctness depends greatly on the type and/or difficulty of the algorithm.

Algorithms

What makes an algorithm "good"?

- It is correct (terminates with right answer)
- It is efficient

What does this mean?
Need a model of computation to be precise
Model should not be so specific (eg Pentium II) that conclusions do not transfer
Every programming language comes with a model of computation (often implied, with some details deliberately vague)

Models of computation

- Very simple ones
  E.g. finite-state machine with added linear tape (Turing machine)
  Mathematically clean, hard to write algorithms (and also hard to get ...)
- More complicated ones
  E.g. multiprocessor Sun workstation with caches and virtual memory
  More realistic, harder to analyze

Models of computation

Specifying a model (ideal)
Design single-processor machine with random-access memory and specific instruction set
Write all algorithms in assembler
Knuth (The Art of Computer Programming, 3 volumes) did this

Models of computation

Specifying a model (practice)
Write algorithms in high-level pseudocode that doesn't assume too much
Word-RAM model:
1. Each memory cell is a word of some bits that can hold an integer value.
2. Random access of memory: accessing a word at the location specified by the value of another word takes constant time.
3. Basic operations (arithmetic, shifting, logical, comparison) on words take constant time.
4...
Is word large enough to hold any integer? (very unlikely ... keep large integers in array of words ... watch the cost of operations)
Large enough to hold an address of a data structure? (very likely ... if the data structure fits into RAM)

Problems

Problem: Given a problem instance, carry out a particular computational task.

Problem Instance: Input for the specified problem.

Problem Solution: Output (correct answer) for the specified problem.

Size of a problem instance: Size(I) is a positive integer which is a measure of the size of the instance I.

Algorithms and Programs

Algorithm: An algorithm is a step-by-step process (e.g., described in pseudocode) for carrying out a series of computations, given some appropriate input.

Algorithm solving a problem: An Algorithm A *solves* a problem Π if, for every instance I of Π, A finds a valid solution for the instance I in finite time.

Program: A program is an implementation of an algorithm using a specified computer language.

Running Time

Running Time of a Program: $T_M(I)$ denotes the running time (in seconds) of a program M on a problem instance I.

Worst-case Running Time as a Function of Input Size: $T_M(n)$ denotes the *maximum* running time of program M on instances of size n:

$$T_M(n) = \max\{ T_M(I) : \text{Size}(I) = n\}.$$

Average-case Running Time as a Function of Input Size: $T_M^{avg}(n)$ denotes the *average* running time of program M over all instances of size n:

$$T_M^{avg}(n) = \frac{1}{|\{I : \text{Size}(I) = n\}|} \sum_{\{I:\text{Size}(I)=n\}} T_M(I).$$

Complexity

Worst-case complexity of an algorithm: Let $f : \mathbb{Z}^+ \to \mathbb{R}$. An algorithm A has *worst-case complexity* $f(n)$ if there exists a program M implementing the algorithm A such that $T_M(n) \in \Theta(f(n))$.

Average-case complexity of an algorithm: Let $f : \mathbb{Z}^+ \to \mathbb{R}$. An algorithm A has *average-case complexity* $f(n)$ if there exists a program M implementing the algorithm A such that $T_M^{avg}(n) \in \Theta(f(n))$.

Order Notation

O-notation: $f(n) \in O(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$.

Ω-notation: $f(n) \in \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0$.

Θ-notation: $f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$.

o-notation: $f(n) \in o(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) \leq c\, g(n)$ for all $n \geq n_0$.

ω-notation: $f(n) \in \omega(g(n))$ if for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq c\, g(n) \leq f(n)$ for all $n \geq n_0$.

Techniques for Order Notation

Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Suppose that

$$L = \lim_{n \to \infty} \frac{f(n)}{g(n)}.$$

Then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty. \end{cases}$$

Relationships between Order Notations

$f(n) \in \Theta(g(n)) \Leftrightarrow g(n) \in \Theta(f(n))$

$f(n) \in O(g(n)) \Leftrightarrow g(n) \in \Omega(f(n))$

$f(n) \in o(g(n)) \Leftrightarrow g(n) \in \omega(f(n))$

$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$

$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$

$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$

Algebra of Order Notations

"Maximum" rules: Suppose that $f(n) > 0$ and $g(n) > 0$ for all $n \geq n_0$. Then:

$$O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$$
$$\Theta(f(n) + g(n)) = \Theta(\max\{f(n), g(n)\})$$
$$\Omega(f(n) + g(n)) = \Omega(\max\{f(n), g(n)\})$$

"Summation" rules: Assume I is a finite set. Then:

$$O\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} O\left(f(i)\right)$$
$$\Theta\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Theta(f(i))$$
$$\Omega\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Omega(f(i))$$

Summation Formulae

Arithmetic sequence:

$$\sum_{i=0}^{n-1}(a+di) = na + \frac{dn(n-1)}{2} \in \Theta(n^2).$$

Geometric sequence:

$$\sum_{i=0}^{n-1} a\, r^i = \begin{cases} a\frac{r^n-1}{r-1} \in \Theta(r^n) & \text{if } r > 1 \\ na \in \Theta(n) & \text{if } r = 1 \\ a\frac{1-r^n}{1-r} \in \Theta(1) & \text{if } 0 < r < 1. \end{cases}$$

Harmonic sequence:

$$\sum_{i=1}^{n} \frac{1}{i} \in \Theta(\log n)$$

Miscellaneous Formulae

$$\log_b a = \frac{1}{\log_a b}$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$a^{\log_b c} = c^{\log_b a}$$

$$n! \in \Theta\left(n^{n+1/2} e^{-n}\right)$$

$$\log n! \in \Theta(n \log n)$$

Techniques for Algorithm Analysis

Two general strategies are as follows.

- Use Θ-bounds throughout the analysis and obtain a Θ-bound for the complexity of the algorithm.

- Prove a O-bound and a matching Ω-bound separately to get a Θ-bound.
  Sometimes this technique is easier because arguments for O-bounds may use simpler upper bounds (and arguments for Ω-bounds may use simpler lower bounds) than arguments for Θ-bounds do.

Techniques for Loop Analysis

Identify "elementary operations" that require constant time (denoted $\Theta(1)$ time).

The complexity of a loop is expressed as the sum of the complexities of each iteration of the loop.

Analyze independent loops separately, and then add the results (use "maximum rules" and simplify whenever possible).

If loops are nested, start with the innermost loop and proceed outwards. In general, this kind of analysis requires evaluation of nested summations.

Example of Loop Analysis

```
sum := 0;
for i := 1 to n do
   for j := 1 to i do
      sum := sum + (i-j)^2;
   sum := sum / i;
print(sum)
```

Loop Analysis for Bentley's Problem, Solution 1

```
max := 0;
for i := 1 to n do
  for j := i to n do
    sum := 0;
    for k := i to j do
      sum := sum + A[k];
    if sum > max then max := sum;
```

Another Example of Loop Analysis

```
sum := 0;
for i := 1 to n do
   j := i;
   while j >= 1 do
      sum:= sum + i/j;
      j := floor(j/2);
print(sum)
```