CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

Winter, 2015

Table of Contents

Analysis of algorithms

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: We often want a formal proof of correctness of an algorithm we design. This might be accomplished through the use of **loop invariants** and mathematical induction.

Analysis of algorithms (cont.)

Efficiency: Given an algorithm A, we want to know how efficient it is. This includes several possible criteria:

- What is the **asymptotic complexity** of algorithm A?
- What is the **exact number** of specified computations done by A?
- How does the **average-case** complexity of A compare to the **worst-case** complexity?
- Is A the most efficient algorithm to solve the given problem? (For example, can we find a **lower bound** on the complexity of **any** algorithm to solve the given problem?)
- Are there problems that cannot be solved efficiently? This topic is addressed in the theory of **NP-completeness**.
- Are there problems that cannot be solved by **any** algorithm? Such problems are termed **undecidable**.

Design of algorithms

"Design" refers to **general strategies** for creating new algorithms. If we have good design strategies, then it will be easier to end up with correct and efficient algorithms. Also, we want to avoid using **ad hoc** algorithms that are hard to analyze and understand.

Here are some useful design strategies, many of which we will study:

divide-and conquer

greedy

dynamic programming

depth-first and breadth-first search

local search (not studied in this course)

linear programming (not studied in this course)

The "Maximum" problem

Problem

Maximum

Instance: *an array A of n integers,*

$$A = [A[1], \ldots, A[n]].$$

Find: *the maximum element in A.*

The **Maximum** problem has an obvious simple solution.

Algorithm: *FindMaximum*$(A = [A[1], \ldots, A[n]])$

$max \leftarrow A[1]$
for $i \leftarrow 2$ **to** n
do $\begin{cases} \textbf{if } A[i] > max \\ \quad \textbf{then } max \leftarrow A[i] \end{cases}$
return (max)

Correctness of FindMaximum

How can we formally prove that *FindMaximum* is correct?

Claim: At the end of iteration i ($i = 2, \ldots, n$), the current value of max is the maximum element in $[A[1], \ldots, A[i]]$.

The claim can be proven by induction. The base case, when $i = 2$, is obvious.

Now we make an induction assumption that the claim is true for $i = j$, where $2 \leq j \leq n - 1$, and we prove that the claim is true for $i = j + 1$ (fill in the details!).

When $j = n$ we are done and the correctness of *FindMaximum* is proven.

Analysis of FindMaximum

It is obvious that the complexity of *FindMaximum* is $\Theta(n)$.

More precisely, we can observe that the number of comparisons of array elements done by *FindMaximum* is **exactly** $n - 1$.

It turns out that *FindMaximum* is **optimal** with respect to the number of comparisons of array elements.

That is, any algorithm that correctly solves the **Maximum** problem for an array of n elements requires **at least** $n - 1$ comparisons of array elements.

How can we prove this assertion?

The "Max-Min" problem

Problem

Max-Min

Instance: *an array A of n integers, $A = [A[1], \ldots, A[n]]$.*

Find: *the maximum and the minimum element in A.*

The **Max-Min** problem also has an obvious simple solution.

Algorithm: *FindMaximumAndMinimum*$(A = [A[1], \ldots, A[n]])$

$max \leftarrow A[1]$
$min \leftarrow A[1]$
for $i \leftarrow 2$ **to** n
do $\begin{cases} \textbf{if } A[i] > max \\ \quad \textbf{then } max \leftarrow A[i] \\ \textbf{if } A[i] < min \\ \quad \textbf{then } min \leftarrow A[i] \end{cases}$
return (max, min)

Analysis of FindMaximumAndMinimum

Exercise: Give a formal proof by induction that *FindMaximumAndMinimum* is correct.

The complexity of *FindMaximumAndMinimum* is $\Theta(n)$

More precisely, *FindMaximumAndMinimum* requires $2n-2$ comparisons of array elements given an array of size n.

The **complexity** is optimal (why?), but there are are algorithms to solve the **Max-Min** problem which require fewer comparisons of array elements than *FindMaximumAndMinimum*.

Note: An algorithm requiring fewer comparisons of array elements **may or may not be faster** than *FindMaximumAndMinimum*.

A more significant improvement

With some ingenuity, we can actually reduce the number of comparisons of array elements by (roughly) 25%.

Suppose n is even and we consider the elements two at a time. Initially, we compare the first two elements and initialize maximum and minimum values. (**One comparison** is required here.)

Then, each time we compare a new pair of elements, we subsequently compare the larger of the two elements to the current maximum and the smaller of the two to the current minimum. (**Three comparisons** are done here to process two array elements.)

This yields an algorithm requiring a total of $3n/2 - 2$ comparisons.

An improved algorithm

Algorithm: *ImprovedFindMaximumAndMinimum*(A)

comment: assume n is even

if $A[1] > A[2]$ **then** $\begin{cases} max \leftarrow A[1] \\ min \leftarrow A[2] \end{cases}$

else $\begin{cases} max \leftarrow A[2] \\ min \leftarrow A[1] \end{cases}$

for $i \leftarrow 2$ **to** $n/2$

do $\begin{cases} \textbf{if } A[2i-1] > A[2i] \\ \quad \textbf{then } \begin{cases} \textbf{if } A[2i-1] > max \ \textbf{ then } max \leftarrow A[2i-1] \\ \textbf{if } A[2i] < min \ \textbf{ then } min \leftarrow A[2i] \end{cases} \\ \quad \textbf{else } \begin{cases} \textbf{if } A[2i] > max \ \textbf{ then } max \leftarrow A[2i] \\ \textbf{if } A[2i-1] < min \ \textbf{ then } min \leftarrow A[2i-1] \end{cases} \end{cases}$

return (max, min)

Optimality of the previous algorithm

It is possible to **prove** that any algorithm that solves the **Max-Min** problem requires at least $3n/2 - 2$ comparisons of array elements in the worst case.

Therefore the algorithm *ImprovedFindMaximumAndMinimum* is in fact **optimal** with respect to the number of comparisons of array elements required.

The "3SUM" problem

Problem

3SUM

Instance: *an array A of n distinct integers, $A = [A[1], \ldots, A[n]]$.*

Question: *do there exist three elements in A that sum to 0?*

The **3SUM** problem also has an obvious algorithm to solve it.

Algorithm: *Trivial3SUM*$(A = [A[1], \ldots, A[n]])$

$$
\begin{array}{l}
\textbf{for } i \leftarrow 1 \textbf{ to } n-2 \\
\quad \textbf{do } \begin{cases} \textbf{for } j \leftarrow i+1 \textbf{ to } n-1 \\ \quad \textbf{do } \begin{cases} \textbf{for } k \leftarrow j+1 \textbf{ to } n \\ \quad \textbf{do } \begin{cases} \textbf{if } A[i]+A[j]+A[k]=0 \\ \quad \textbf{then output } (i,j,k) \end{cases} \end{cases} \end{cases}
\end{array}
$$

The complexity of *Trivial3SUM* is $O(n^3)$.

A possible improvement

Instead of having three nested loops, suppose we have **two** nested loops (with indices i and j, say) and then we **search** for an $A[k]$ for which $A[i] + A[j] + A[k] = 0$.

If we try all possible k-values, then we basically have the previous algorithm.

What can we do to make the search more efficient?

What effect does this have on the complexity of the resulting algorithm?

An improved algorithm for the "3SUM" problem

Algorithm: *Improved3SUM*$(A = [A[1], \ldots, A[n]])$

sort A in increasing order
for $i \leftarrow 1$ **to** $n-2$
do $\begin{cases} \textbf{for } j \leftarrow i+1 \textbf{ to } n-1 \\ \textbf{do } \begin{cases} \text{perform a binary search for the value } A[k] = -A[i] - A[j] \\ \text{if the search is successful, } \textbf{output } (i,j,k) \end{cases} \end{cases}$

The complexity of *Improved3SUM* is $O(n \log n + n^2 \log n) = O(n^2 \log n)$.

A further improvement

In *Improved3SUM*, we **pre-sorted** the array A, which enabled us to do binary searches.

There is a better way to make use of the sorted are, however ...

Namely, for a given $A[i]$, we **simultaneously scan from both ends** of A looking for $A[j] + A[k] = -A[i]$.

We start with $j = i + 1$ and $k = n$.

At any stage of the algorithm, we either **increment** j or **decrement** k (or both, if $A[i] + A[j] + A[k] = 0$).

Does this remind you of a familiar algorithm you have seen in CS 240?

The resulting algorithm will have complexity $O(n \log n + n^2) = O(n^2)$.

A quadratic time algorithm for the "3SUM" problem

Algorithm: *Quadratic3SUM*$(A = [A[1], \dots, A[n]])$

sort A in increasing order
for $i \leftarrow 1$ **to** $n-2$
do
- $j \leftarrow i + 1$
- $k \leftarrow n$
- **while** $j < k$
  do
  - $S \leftarrow A[i] + A[j] + A[k]$
  - **if** $S < 0$ **then** $j \leftarrow j + 1$
  - **else if** $S > 0$ **then** $k \leftarrow k - 1$
  - **else**
    - $j \leftarrow j + 1$
    - $k \leftarrow k - 1$
    - **output** (i, j, k)

Problems

Problem: Given a problem instance I for a problem **P**, 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 **P** if, for every instance I of **P**, 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: $T_{\mathbf{M}}(I)$ denotes the running time (in seconds) of a program $\mathbf{M}$ on a problem instance I.

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

$$T_{\mathbf{M}}(n) = \max\{T_{\mathbf{M}}(I) : \mathsf{Size}(I) = n\}.$$

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

$$T_{\mathbf{M}}^{avg}(n) = \frac{1}{|\{I : \mathsf{Size}(I) = n\}|} \sum_{\{I:\mathsf{Size}(I)=n\}} T_{\mathbf{M}}(I).$$

Complexity

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

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

Running Time vs Complexity

Running time can only be determined by implementing a program and running it on a specific computer.

Running time is influenced by many factors, including the programming language, processor, operating system, etc.

Complexity (AKA **growth rate**) can be analyzed by high-level mathematical analysis. It is **independent** of the above-mentioned factors affecting running time.

Complexity is a less precise measure than running time since it is asymptotic and it incorporates unspecified constant factors and unspecified lower order terms.

However, if algorithm A has lower complexity than algorithm B, then a program implementing algorithm A will be faster than a program implementing algorithm B for sufficiently large inputs.

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 $n \geq n_0$.

Here the complexity of f is **not higher** than the complexity of g.

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 $n \geq n_0$.

Here the complexity of f is **not lower** than the complexity of g.

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 $n \geq n_0$.

Here f and g have the **same complexity**.

Order Notation (cont.)

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 $n \geq n_0$.

Here f has **lower complexity** than g.

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 $n \geq n_0$.

Here f has **higher complexity** than g.

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

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

$$f(n) \in \Theta(g(n)) \Leftrightarrow f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n))$$
$$f(n) \in o(g(n)) \Rightarrow f(n) \in O(g(n))$$
$$f(n) \in \omega(g(n)) \Rightarrow f(n) \in \Omega(g(n))$$

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:

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

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

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

Sequences

Arithmetic sequence:

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

Geometric sequence:

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

Arithmetic-geometric sequence:

$$\sum_{i=0}^{n-1}(a+di)r^i = \frac{a}{1-r} - \frac{(a+(n-1)d)r^n}{1-r} + \frac{dr(1-r^{n-1})}{(1-r)^2}$$

provided that $r \neq 1$.

Sequences (cont.)

Harmonic sequence:

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

More precisely, it is possible to prove that

$$\lim_{n\to\infty} (H_n - \ln n) = \gamma,$$

where $\gamma \approx 0.57721$ is **Euler's constant**.

Miscellaneous Formulae

$$\log_b xy = \log_b x + \log_b y$$

$$\log_b x/y = \log_b x - \log_b y$$

$$\log_b 1/x = -\log_b x$$

$$\log_b x^y = y \log_b x$$

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

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

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

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

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

Techniques for Algorithm Analysis

Two general strategies are as follows:

- Use Θ-bounds **throughout the analysis** and thereby 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

Algorithm: *LoopAnalysis1*$(n : integer)$

(1) $sum \leftarrow 0$

(2) **for** $i \leftarrow 1$ **to** n

do $\begin{cases} \textbf{for } j \leftarrow 1 \textbf{ to } i \\ \textbf{do } \begin{cases} sum \leftarrow sum + (i-j)^2 \\ sum \leftarrow sum/i \end{cases} \end{cases}$

(3) **return** (sum)

Θ-bound analysis

(1)	$\Theta(1)$
(2)	Complexity of inner **for** loop: $\Theta(i)$
	Complexity of outer **for** loop: $\sum_{i=1}^{n} \Theta(i) = \Theta(n^2)$
	Note: $\sum_{i=1}^{n} i = n(n+1)/2$
(3)	$\Theta(1)$
total	$\Theta(n^2)$

Example of Loop Analysis (cont.)

Proving separate O- and Ω-bounds

We focus on the two nested **for** loops (i.e., (2)).

The total number of iterations is $\sum_{i=1}^{n} i$, with $\Theta(1)$ time per iteration.

Upper bound:

$$\sum_{i=1}^{n} O(i) \leq \sum_{i=1}^{n} O(n) = O(n^2).$$

Lower bound:

$$\sum_{i=1}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(i) \geq \sum_{i=n/2}^{n} \Omega(n/2) = \Omega(n^2/4) = \Omega(n^2).$$

Since the upper and lower bounds **match**, the complexity is $\Theta(n^2)$.

Another Example of Loop Analysis

Algorithm: *LoopAnalysis2*$(A : array; n : integer)$

$$
\begin{array}{l}
max \leftarrow 0\\
\textbf{for } i \leftarrow 1 \textbf{ to } n\\
\quad \textbf{do } \left\{\begin{array}{l}
\textbf{for } j \leftarrow i \textbf{ to } n\\
\quad \textbf{do } \left\{\begin{array}{l}
sum \leftarrow 0\\
\textbf{for } k \leftarrow i \textbf{ to } j\\
\quad \textbf{do } \left\{\begin{array}{l}
sum \leftarrow sum + A[k]\\
\textbf{if } sum > max\\
\quad \textbf{then } max \leftarrow sum
\end{array}\right.
\end{array}\right.
\end{array}\right.\\
\textbf{return } (max)
\end{array}
$$

Yet Another Example of Loop Analysis

Algorithm: *LoopAnalysis3*$(n : integer)$

$sum \leftarrow 0$

for $i \leftarrow 1$ **to** n

$$\textbf{do} \begin{cases} j \leftarrow i \\ \textbf{while } j \geq 1 \\ \quad \textbf{do} \begin{cases} sum \leftarrow sum + i/j \\ j \leftarrow \left\lfloor \frac{j}{2} \right\rfloor \end{cases} \end{cases}$$

return (sum)