CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

January 3, 2019
1 Course Information
2 Introduction
3 Divide-and-Conquer Algorithms
4 Greedy Algorithms
5 Dynamic Programming Algorithms
6 Graph Algorithms
7 Intractability and Undecidability
Table of Contents

1 Course Information
Course mechanics

My Sections:

- LEC 003, T Th 4:00–5:20, PHY 235
- LEC 004, T Th 1:00–2:20, MC 2034

My Scheduled Office Hours:

- Tuesday, 11:00–12:00, DC 3522
Course mechanics

- **Come to class!** Not all the material will be on the slides or in the text.
- You will need an account in the student.cs environment
- The course website can be found at https://www.student.cs.uwaterloo.ca/~cs341/
  - Syllabus, calendar, policies, etc. can be found there.
Learn and Piazza

- Slides and assignments will be available on the course website.
- Grades and assignment solutions will be available on Learn.
- Discussion related to the course will take place on Piazza (piazza.com).
  - General course questions, announcements
  - Assignment-related questions
  - You will be getting an invitation via email to join Piazza in the first week of classes.
- Keep up with the information posted on the course website, Learn and Piazza.
Course Information

**Courtesy**

- Please silence cell phones and other mobile devices before coming to class.
- Questions are encouraged, but *please refrain from talking* in class – it is distracting to your classmates who are trying to listen to the lectures and to your professor who is trying to think, talk and write at the same time.
- Carefully consider whether *using your laptop, ipad, smartphone, etc., in class* will help you learn the material and follow the lectures.
- Do not play games, tweet, watch youtube videos, update your facebook page or use a mobile device in any other way that will distract your classmates.
Course syllabus

- You are expected to be familiar with the contents of the course syllabus
- Available on the course home page
- If you haven’t read it, read it as soon as possible
Plagiarism and academic offenses

- We take academic offences very seriously
- There is a good discussion of plagiarism online:
  - https://uwaterloo.ca/academic-integrity/sites/ca.
    academic-integrity/files/uploads/files/revised_
    undergraduate_tip_sheet_2013_final.pdf
- Read this and understand it
  - Ignorance is no excuse!
  - Questions should be brought to instructor
- Plagiarism applies to both text and code
- You are free (even encouraged) to exchange ideas, but no sharing code or text
Plagiarism (2)

- Common mistakes
  - Excess collaboration with other students
    - Share ideas, but no design or code!
  - Using solutions from other sources (like for previous offerings of this course, maybe written by yourself)
- More information linked to from course syllabus
Grading scheme for CS 341

- Midterm (25%)
  - Tuesday, Feb. 26, 2019, 7:00–8:50 PM

- Assignments (30%)
  - There will be five assignments.
  - Work alone
  - See syllabus for reappraisal policies, academic integrity policy, and other details

- Final (45%)

- For medical conditions, you need to submit a Verification of Illness form.
Assignments

- All sections will have the same assignments, midterm and final exam.
- Assignments will be due at 6:00 PM on the due date.
- No late submissions will be accepted.
- You need to notify your instructor well before the due date of a severe, long-lasting or ongoing problem that prevents you from doing an assignment.
Assignment due dates

- Assignment 1: due Friday Jan. 25
- Assignment 2: due Friday Feb. 8
- Assignment 3: due Friday March 1
- Assignment 4: due Friday March 22
- Assignment 5: due Friday April 5
Textbook


- You are expected to know
  - all the material presented in class
  - relevant textbook sections, as listed on course website
Table of Contents

2 Introduction
- Algorithm Design and Analysis
- The 3SUM Problem
- Reductions
- Definitions and Terminology
- Order Notation
- Formulae
- Loop Analysis Techniques
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: The level of detail required in a proof of correctness depends greatly on the type and/or difficulty of the algorithm.
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 examples of useful design strategies, many of which we will study:

- reductions
- divide-and-conquer
- greedy
- dynamic programming
- depth-first and breadth-first search
- local search
- exhaustive search (backtracking, branch-and-bound)
The “3SUM” Problem

Problem 2.1

3SUM


Question: do there exist three distinct elements in $A$ whose sum equals 0?

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

Algorithm: $\text{Trivial3SUM}(A = [A[1], \ldots, A[n]])$

for $i \leftarrow 1$ to $n - 2$

   for $j \leftarrow i + 1$ to $n - 1$

      for $k \leftarrow j + 1$ to $n$


              then output $(i, j, k)$

The complexity of $\text{Trivial3SUM}$ is $O(n^3)$. 
An 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 sequentially 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* \( (A = [A[1], \ldots, A[n]]) \)

\[ \text{sort}(A) \]

\[ \text{for } i \leftarrow 1 \text{ to } n - 2 \]

\[ \text{for } j \leftarrow i + 1 \text{ to } n - 1 \]

\[ \text{do } \left\{ \right. \]

\[ \text{do } \left\{ \right. \]


\[ \text{in the subarray } A[j + 1], \ldots, A[n] \]

\[ \text{if the search is successful} \]

\[ \text{then output } (i, j, k) \]

\[ \text{do } \left. \right\} \]

\[ \text{do } \left. \right\} \]

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

The \( n \log n \) term is the *sort*. The \( n^2 \log n \) term accounts for \( n^2 \) binary searches.
A Further Improvement

The sort is an example of pre-processing.

It modifies the input to permit a more efficient algorithm to be used (binary search as opposed to linear search).

Note that a pre-processing step is only done once.

However, there is a better way to make use of the sorted array.


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 $A[i] + A[j] + A[k] = 0$).

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\((A = [A[1], \ldots, A[n]])\)

\[\text{sort}(A)\]

\[\text{for } i \leftarrow 1 \text{ to } n - 2 \]

\[j \leftarrow i + 1\]

\[k \leftarrow n\]

\[\text{while } j < k \]

\[\text{do }\]


\[\text{if } S < 0 \text{ then } j \leftarrow j + 1\]

\[\text{else if } S > 0 \text{ then } k \leftarrow k - 1\]

\[\text{else}\]

\[\text{output } (i, j, k)\]

\[j \leftarrow j + 1\]

\[k \leftarrow k - 1\]
Example

Consider the sorted array \(-11 \ -10 \ -7 \ -3 \ 2 \ 4 \ 8 \ 10\).

The algorithm will not find any solutions when \(i = 1\):

\[
\begin{array}{cccccccc}
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
\end{array}
\]

\(S = -11\)

\(S = -8\)

\(S = -4\)

\(S = 1\)

\(S = -1\)

\(S = 1\)

When \(i = 2\), we have

\[
\begin{array}{cccccccc}
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
-11 & -10 & -7 & -3 & 2 & 4 & 8 & 10 \\
\end{array}
\]

\(S = -7\)

\(S = -3\)

\(S = 2\)

\(S = 0\)
Reductions

Suppose $\Pi_1$ and $\Pi_2$ are problems and we can use a hypothetical algorithm solving $\Pi_2$ as a subroutine to solve $\Pi_1$.

Then we have a **Turing reduction** (or more simply, a **reduction**) from $\Pi_1$ to $\Pi_2$; this is denoted $\Pi_1 \leq \Pi_2$.

The hypothetical algorithm solving $\Pi_2$ is called an **oracle**.

The reduction must treat the oracle as a **black box**.

There may be more than one call made to the oracle in the reduction.

Suppose

- $\Pi_1 \leq \Pi_2$ and
- we also have an algorithm $A_2$ that solves $\Pi_2$.

If we plug $A_2$ into the reduction, then we obtain an algorithm that solves $\Pi_1$.

Reductions potentially allow **re-using** code, which may be advantageous.
A Simple Reduction

Consider the algebraic identity

\[ xy = \frac{(x + y)^2 - (x - y)^2}{2}. \]

This identity allows us to show that \textbf{Multiplication} \leq \textbf{Squaring}.

Algorithm: \textit{MultiplicationToSquaring}(x, y)

\begin{itemize}
  \item \textbf{external} \textit{ComputeSquare}
  \item \textit{s} \leftarrow \textit{ComputeSquare}(x + y)
  \item \textit{t} \leftarrow \textit{ComputeSquare}(x - y)
  \item \textbf{return} \((s - t)/2\)
\end{itemize}

Note that the “division by 2” just consists of deleting the low-order bit, which is guaranteed to be a 0.
The “Target 3SUM” Problem

Problem 2.2

Target3SUM


Question: do there exist three distinct elements in $A$ whose sum equals $T$?

It is straightforward to modify any algorithm solving the 3SUM problem so it solves the Target3SUM problem.

Another approach is to find a reduction Target3SUM $\leq$ 3SUM. This would allow us to re-use code as opposed to modifying code.
Target3SUM ≤ 3SUM


This suggests the following approach, which works if $T$ is divisible by three.

**Algorithm:** Target3SUMto3SUM($A = [A[1], \ldots, A[n]], T$)
- comment: assume $T$ is divisible by 3
- external 3SUM-solver

  for $i \leftarrow 1$ to $n$
  do $B[i] \leftarrow A[i] - T/3$
  return (3SUM-solver($B$))

**Modification:** the transformation $B[i] \leftarrow 3A[i] - T$ works for any integer $T$. 
Complexity analysis

Suppose we replace the oracle 3SUM-solver by a “real” algorithm. What is the complexity of the resulting algorithm?

If we plug in Trivial3SUM, the complexity is $O(n + n^3) = O(n^3)$.

If we plug in Improved3SUM, the complexity is $O(n + n^2 \log n) = O(n^2 \log n)$.

If we plug in Quadratic3SUM, the complexity is $O(n + n^2) = O(n^2)$.

In each case, it turns out that the “$n$” term is subsumed by the second term.
Many-one Reductions

The reduction on the previous slide had a very special structure:

- We transformed an instance of the first problem to an instance of the second problem.
- We called the oracle once, on the transformed instance.

Reductions of the this form, in the context of decision problems, are called **many-one reductions** (also known as **polynomial transformations** or **Karp reductions**).

We will many examples of these in the section on **intractability**.
The “3array3SUM” Problem

**Problem 2.3**

**3array3SUM**

**Instance:** three arrays of $n$ distinct integers, $A$, $B$ and $C$.

**Question:** do there exist array elements, one from each of $A$, $B$ and $C$, whose sum equals 0?

**Algorithm:** $3array3SUMto3SUM(A, B, C)$

```plaintext
external 3SUM-solver

for $i ← 1$ to $n$
    
    do
        
        $D[i] ← 10A[i] + 1$
        $E[i] ← 10B[i] + 2$
        $F[i] ← 10C[i] - 3$
    
    let $A'$ denote the concatenation of $D$, $E$ and $F$

    if $3SUM-solver(A') = (i, j, k)$
        then return $(i, i - n, j - 2n)$
```

D.R. Stinson (SCS) CS 341 January 3, 2019 31 / 294
3array3SUM \leq 3SUM \ (cont.)

To show that 3array3SUMto3SUM is a reduction, we show that \((i, j, k)\) is a solution to the instance \(A'\) of 3SUM if and only if \((i, i - n, j - 2n)\) is a solution to the instance \(A, B, C\) of 3array3SUM.

Assume first that \((i', j', k')\) is a solution to 3array3SUM. Then

\[ A[i'] + B[j'] + C[k'] = 0.\]

Hence,

\[ D[i'] + E[j'] + F[k'] = 10A[i'] + 1 + 10B[j'] + 2 + 10C[k'] - 3 = 0\]

and thus

\[ A'[i'] + A'[j' + n] + A'[k' + 2n] = 0.\]

Conversely, suppose that \(A'[i] + A'[j] + A'[k] = 0.\) We claim that this sum consists of one element from each of \(A, B\) and \(C\). This can be proven by considering the sum modulo 3 and observing that the only way to get a sum that is divisible by three is \(1 + 2 - 3 \mod 3.\) The result follows by translating back to the original arrays \(A, B\) and \(C.\)
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**: $\text{Size}(I)$ 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: $T_M(I)$ 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: $T_M^{\text{avg}}(n)$ denotes the average running time of program $M$ over all instances of size $n$:

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

**Worst-case complexity of an algorithm:** Let $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$. 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 $f : \mathbb{Z}^+ \rightarrow \mathbb{R}$. An algorithm $A$ has **average-case complexity** $f(n)$ if there exists a program $M$ implementing the algorithm $A$ such that $T_M^{avg}(n) \in \Theta(f(n))$. 
# 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 $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

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

$\Omega$-notation:

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

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

$\Theta$-notation:

$f(n) \in \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 > 0$ such that $0 \leq c_1g(n) \leq f(n) \leq c_2g(n)$ for all $n \geq n_0$.

Here $f$ and $g$ have the same complexity.
Order Notation (cont.)

\textit{o-notation:}

\[ f(n) \in o(g(n)) \text{ if for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0. \]

Here \( f \) has \textit{lower complexity} than \( g \).

\textit{ω-notation:}

\[ f(n) \in \omega(g(n)) \text{ if for all constants } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0. \]

Here \( f \) has \textit{higher complexity} than \( g \).
Exercises

1. Let \( f(n) = n^2 - 7n - 30 \). Prove from first principles that \( f(n) \in O(n^2) \).

2. Let \( f(n) = n^2 - 7n - 30 \). Prove from first principles that \( f(n) \in \Omega(n^2) \).

3. Suppose \( f(n) = n^2 + n \). Prove from first principles that \( f(n) \notin O(n) \).
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

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

1. Compare the growth rate of the functions $(\ln n)^2$ and $n^{1/2}$.

2. Use the limit method to compare the growth rate of the functions $n^2$ and $n^2 - 7n - 30$. 
Additional Exercises

1. Compare the growth rate of the functions $(3 + (-1)^n)n$ and $n$.

2. Compare the growth rates of the functions $f(n) = n |\sin \pi n/2| + 1$ and $g(n) = \sqrt{n}$. 
Relationships between Order Notations

\[ f(n) \in \Theta(g(n)) \iff g(n) \in \Theta(f(n)) \]
\[ f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n)) \]
\[ f(n) \in o(g(n)) \iff g(n) \in \omega(f(n)) \]

\[ f(n) \in \Theta(g(n)) \iff f(n) \in O(g(n)) \text{ and } f(n) \in \Omega(g(n)) \]
\[ f(n) \in o(g(n)) \implies f(n) \in O(g(n)) \]
\[ f(n) \in \omega(g(n)) \implies f(n) \in \Omega(g(n)) \]
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: Supose $I$ is a finite set. Then

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

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

\[
\Omega\left(\sum_{i \in I} f(i)\right) = \sum_{i \in I} \Omega(f(i))
\]
### Some Common Growth Rates (in increasing order)

<table>
<thead>
<tr>
<th>Category</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Polynomial</strong></td>
<td>$\Theta(1)$, $\Theta(\log n)$, $\Theta(\sqrt{n})$, $\Theta(n)$, $\Theta(n^2)$, $\Theta(n^c)$, $\Theta(n^{\sqrt{n} \log_2 n})$ (graph isomorphism), $\Theta(e^{c(\log n)^{1/3}(\log \log n)^{2/3}})$ (number field sieve)</td>
</tr>
<tr>
<td><strong>Exponential</strong></td>
<td>$\Theta(1.1^n)$, $\Theta(2^n)$, $\Theta(e^n)$, $\Theta(n!)$, $\Theta(n^n)$</td>
</tr>
</tbody>
</table>
Sequences

**Arithmetic sequence:**

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

**Geometric sequence:**

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

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 \( r \neq 1 \).

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.
Logarithm Formulae

1. $\log_b xy = \log_b x + \log_b y$
2. $\log_b \frac{x}{y} = \log_b x - \log_b y$
3. $\log_b \frac{1}{x} = -\log_b x$
4. $\log_b x^y = y \log_b x$
5. $\log_b a = \frac{1}{\log_a b}$
6. $\log_b a = \frac{\log_c a}{\log_c b}$
7. $a^{\log_b c} = c^{\log_b a}$
**Miscellaneous Formulae**

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

\[ \log n! \in \Theta(n \log n) \]

Another useful formula is

\[ \sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}, \]

which implies that

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

A sum of powers of integers when \( c \geq 1: \)

\[ \sum_{i=1}^{n} i^c \in \Theta(n^{c+1}). \]
Two General Strategies for Loop Analysis

Sometimes a $O$-bound is sufficient. However, we often want a precise $\Theta$-bound. Two general strategies are as follows:

- Use $\Theta$-bounds **throughout the analysis** and thereby obtain a $\Theta$-bound for the complexity of the algorithm.
- Prove a $O$-bound and a **matching** $\Omega$-bound **separately** to get a $\Theta$-bound. Sometimes this technique is easier because arguments for $O$-bounds may use simpler upper bounds (and arguments for $\Omega$-bounds may use simpler lower bounds) than arguments for $\Theta$-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.
Elementary Operations in the Unit Cost Model

For now, we will work in the **unit cost model**, where we assume that arithmetic operations such as $+$, $-$, $\times$ and integer division take time $\Theta(1)$. This is a reasonable assumption for integers of **bounded size** (e.g., integers that fit into one work of memory).

If we want to consider the complexity of arithmetic operation on integers of arbitrary size, we need to consider **bit complexity**, where we express the complexity as a function of the length of the integers (as measured in bits). We will see some examples later, such as **multiprecision multiplication**.
Example of Loop Analysis

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

1. \(\text{sum} \leftarrow 0\)
2. \(\text{for } i \leftarrow 1 \text{ to } n\)
   
   \[
   \left\{ \begin{array}{l}
   \text{for } j \leftarrow 1 \text{ to } i \\
   \quad \text{do } \left\{ \begin{array}{l}
   \text{sum} \leftarrow \text{sum} + (i - j)^2 \\
   \quad \text{sum} \leftarrow \lceil \text{sum}/i \rceil
   \end{array} \right\
   \end{array} \right\
   \]
3. \(\text{return } (\text{sum})\)

**Θ-bound analysis**

1. \(Θ(1)\)
2. Complexity of inner for loop: \(Θ(i)\)
   
   Complexity of outer for loop: \(\sum_{i=1}^{n} Θ(i) = Θ(n^2)\)
3. \(Θ(1)\)

\[
\text{total } = Θ(1) + Θ(n^2) + Θ(1) = Θ(n^2)
\]
Example of Loop Analysis (cont.)

Proving separate $O$- and $\Omega$-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: \textit{LoopAnalysis2}(A : array; n : integer)

\begin{align*}
& \text{max} \leftarrow 0 \\
& \text{for } i \leftarrow 1 \text{ to } n \\
& \quad \text{for } j \leftarrow i \text{ to } n \\
& \quad \quad \text{do } \\
& \quad \quad \quad \text{sum} \leftarrow 0 \\
& \quad \quad \quad \text{for } k \leftarrow i \text{ to } j \\
& \quad \quad \quad \quad \text{do } \\
& \quad \quad \quad \quad \quad \text{sum} \leftarrow \text{sum} + A[k] \\
& \quad \quad \quad \quad \quad \text{if } \text{sum} > \text{max} \\
& \quad \quad \quad \quad \quad \quad \text{then } \text{max} \leftarrow \text{sum} \\
& \text{return } (\text{max})
\end{align*}
Another Example of Loop Analysis (cont.)

**Θ-bound analysis** The innermost loop (for \(k\)) has complexity \(Θ(j - i + 1)\). The next loop (for \(j\)) has complexity

\[
\sum_{j=i}^{n} Θ(j - i + 1) = Θ\left(\sum_{j=i}^{n} (j - i + 1)\right)
\]

\[
= Θ(1 + 2 + \cdots + (n - i + 1))
\]

\[
= Θ((n - i + 1)(n - i + 2)).
\]

The outer loop (for \(i\)) has complexity

\[
\sum_{i=1}^{n} Θ((n - i + 1)(n - i + 2)) = Θ\left(\sum_{i=1}^{n} (n - i + 1)(n - i + 2)\right)
\]

\[
= Θ(1 \times 2 + 2 \times 3 + \cdots + n(n + 1))
\]

\[
= Θ\left(n^3/3 + n^2 + 2n/3\right) \text{ from Maple}
\]

\[
= Θ(n^3).
\]
Another Example of Loop Analysis (cont.)

Proving an $\Omega$-bound

Consider two loop structures:

\[
\begin{align*}
&L_1 & & L_2 \\
i = 1, \ldots, n/3 & & i = 1, \ldots, n \\
j = 1 + 2n/3, \ldots, n & & j = i + 1, \ldots, n \\
k = 1 + n/3, \ldots, 1 + 2n/3 & & k = i \ldots, j
\end{align*}
\]

It is easy to see that $L_1 \subset L_2$. $L_2$ is loop structure of the given algorithm. There are $(n/3)^3 = n^3/27$ iterations in $L_1$ and $n^3$ iterations in $L_3$. Therefore the number of iterations in $L_2$ is $\Omega(n^3)$. 

Yet Another Example of Loop Analysis

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

\[
\begin{array}{l}
\text{sum} \leftarrow 0 \\
\text{for } i \leftarrow 1 \text{ to } n \\
\quad \begin{cases} 
j \leftarrow i \\
\quad \text{while } j \geq 1 \\
\quad \quad \begin{cases} 
\text{do } \{ 
sum \leftarrow sum + i/j \\
\quad j \leftarrow \lceil j/2 \rceil 
\}
\end{cases} \\
\end{cases} \\
\end{cases} \\
\text{return } (sum)
\end{array}
\]