Unit 02: Pseudocode and Counting

Anthony Estey

CSC 225: Algorithms and Data Structures I

University of Victoria

*Thanks to Dr. Rich Little at the University of Victoria for providing some of the contents of these slides

Unit 02 Overview

- Supplemental Reading:
  - Algorithm Design and Analysis. *Michael Goodrich and Roberto Tamassia*
    - Pages 1 - 9
- Learning Objectives: (You should be able to…)
  - understand why we will use pseudocode to support or analysis of algorithms and data structures
  - understand the syntax of pseudocode, and how it maps to operations in a programming language like Java, C, or C++
  - understand the methodology we will use in this course to analyze algorithms and data structures, based on the size of the input data, n
  - determine the number of operations required to execute an algorithm through an analysis of pseudocode

Algorithms and Data structures

- An **algorithm** is a step-by-step procedure for performing some task in a finite amount of time
- A **data structure** is a systematic way of organizing and accessing data

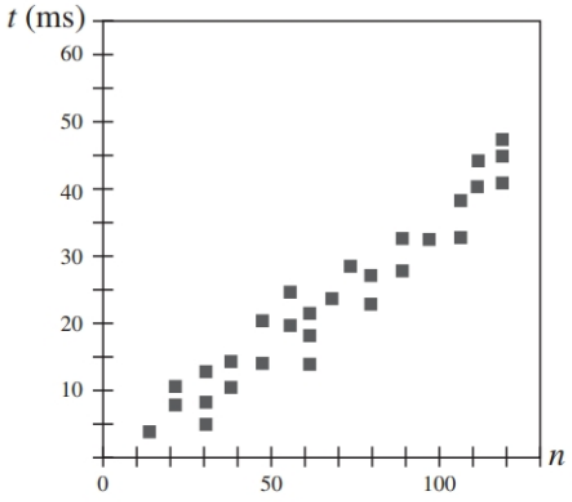
- Analysis:
  - In the world of IT, ***scalability*** refers to the ability of a system to gracefully accommodate growing sizes of inputs or amounts of workload
  - The main focus of our analysis will be to character ***running time***,
  - but it is also important to consider ***space usage*** as well

Analysis

- What is the "right" way to analyze an algorithm?
  - Idea: We could just execute it and see how long it takes to complete
  - Problem: It is difficult to setup a controlled environment to accurately compare two different algorithms
- What is the best input data to test with?
  - Probably best to perform several experiments on many different test inputs and with test inputs of various sizes
- What is the best way to interpret / visualize the results?
  - Plot on a graph:
    - x-coordinate represents the input size, n
    - y-coordinate represents the running time, t

Analysis

- Assume our approach is plot multiple input sets on a graph that illustrates the relationship between input size and execution time
  - The hardware environment will affect the results (processor, memory, etc)
  - and the software environment will too (OS, programming language, compiler)
  - So we can't get you to compare results!
- If we can control for these variables, we may see something like:



- In a perfect world, our analysis can:
  - Take into account all possible inputs
  - Evaluate relative efficiency of any two algorithms in a way that is not dependent on the hardware and software environment

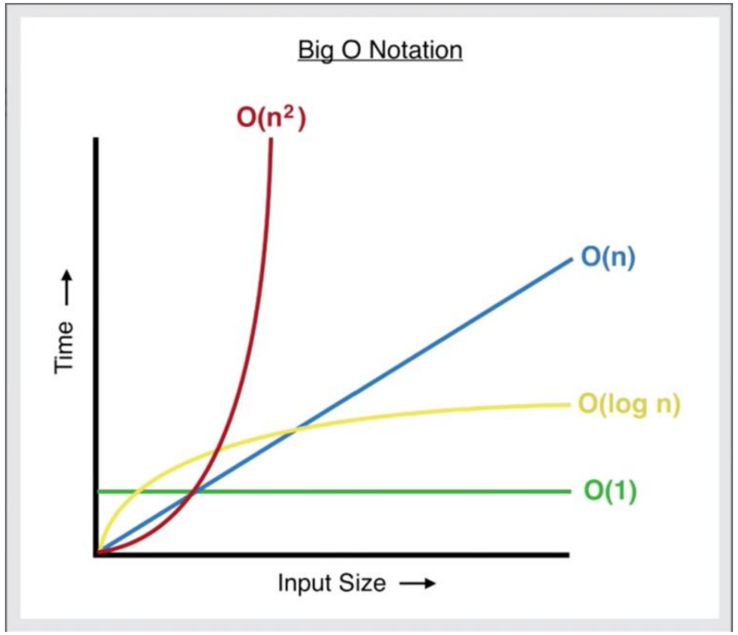
Analysis

- How can we achieve these goals?
  - We want an environment where analysis can be performed that illustrates the efficiency of an algorithm that accounts for all input sizes, without having to implement and execute multiple controlled experiments

- The rest of this unit will focus on an approach that aims to do this

Methodology

- Proposed methodology:
  - Associate each algorithm a function $f(n)$ that characterizes the running time of the algorithm in terms of the input size n
- Methodology requirements:
  - A language for describing the algorithms we want to analyze
    - Pseudocode
  - A metric for measuring algorithm runtime
    - Counting primitive operations
  - An approach for characterizing the runtimes
    - Worse-case analysis (big-Oh)
- Result:



Pseudocode

- Programmers are often asked to describe algorithms in a way that is intended to be read by humans (as opposed to executed by a machine)
  - Compared to code, these descriptions should be easier to read and understand

- These descriptions facilitate high-level analysis of a data structure or algorithm – we call such descriptions ***pseudocode***

Example

Java code:

```
public int arrayMax(int[] A, int n) {
    int currentMax = A[0];
    for (int k=1; k < n; k++) {
        if (currentMax < A[k]) {
            currentMax = A[k];
        }
    }
    return currentMax;
}
```

Pseudocode:

Algorithm arrayMax(A, n)

    Input: An array A storing $n \geq 1$ integers

    Output : The maximum element in A

    $currentMax \leftarrow A[0]$

    for $\mathrm{k} \leftarrow 1$ to $\mathrm{n} - 1$ **do**

        if $currentMax < A[k]$ **then**

            $currentMax \leftarrow A[k]$

        end

    end

    return $currentMax$

Expressions

- Standard mathematical symbols are used to express numeric and Boolean expressions
- The left arrow sign $\leftarrow$ is used as the assignment operator
  - equivalent to the = operator used for assignment in Java, C, C++, etc
- The equal sign = is used to test equality
  - equivalent to using == in Java, C, C++, etc

Method Declarations

- General form:

  Algorithm name(param1, param2, …):

  $\boldsymbol{Input}$: <Description>

  $\boldsymbol{Output}$: <Description>

- Specific example:

  Algorithm arrayMax(A, n):

  $\boldsymbol{Input}$: An array A storing $n \geq 1$ integers

  $\boldsymbol{Output}$: The maximum element in A

Method calls and return statements

- Method call:
  - [object.]method(args)
  - Example: $arrayMax(A, 13)$

- Return statements:
  - **return** value

Array indexing and field selection

- $A[i]$ represents the ith cell in the array A
  - An array of size n is indexed from $A[0]$ to $A[n-1]$

- Dot (.) notation is used to access fields within an structure/object
  - $r.key$ represents the key field in the structure named r

Decision structures

- General form:

  if condition **then**

  true-actions

  [else

  false-actions]

  end

- Specific example:

  if $currentMax < A[k]$ **then**

  $currentMax \leftarrow A[k]$

  end

Repetition structures: while loops

- General form:

while condition **do**

  actions

end

- Specific example:

while $k < n$ **do**

  $count \leftarrow 2 * count$

  $k \leftarrow k + 1$

end

Repetition structures: repeat loops

- General form:

  repeat

  actions

  until condition

- Specific example:

  repeat

  $count \leftarrow 2 * count$

  $k \leftarrow k + 1$

  until $k \geq n$

Repetition structures: for loops

- General form:

  for step-definition **do**

  actions

  end

- Specific example:

  for $k \leftarrow 1$ **to** $n-1$

  $count \leftarrow 2 * count$

  end

A metric for measuring algorithm runtime

- Two approaches
  - Counting primitive operations and computing upper and lower bounds
    - Covered in CSC 115/116, CSC 225, CSC 326, CSC 425, CSC 426, etc.
  - Instrument the code to measure computer clock cycles
    - Covered in in SENG 265, (and sometimes in projects of higher level Systems courses)

What is the running time?

Algorithm arrayMax(A, n)

$\boldsymbol{Input}$: An array A storing $n \geq 1$ integers

$\boldsymbol{Output}$: The maximum element in A

$currentMax \leftarrow A[0]$

for $\mathrm{k} \leftarrow 1$ to $\mathrm{n} - 1$ **do**

  if $currentMax < A[k]$ **then**

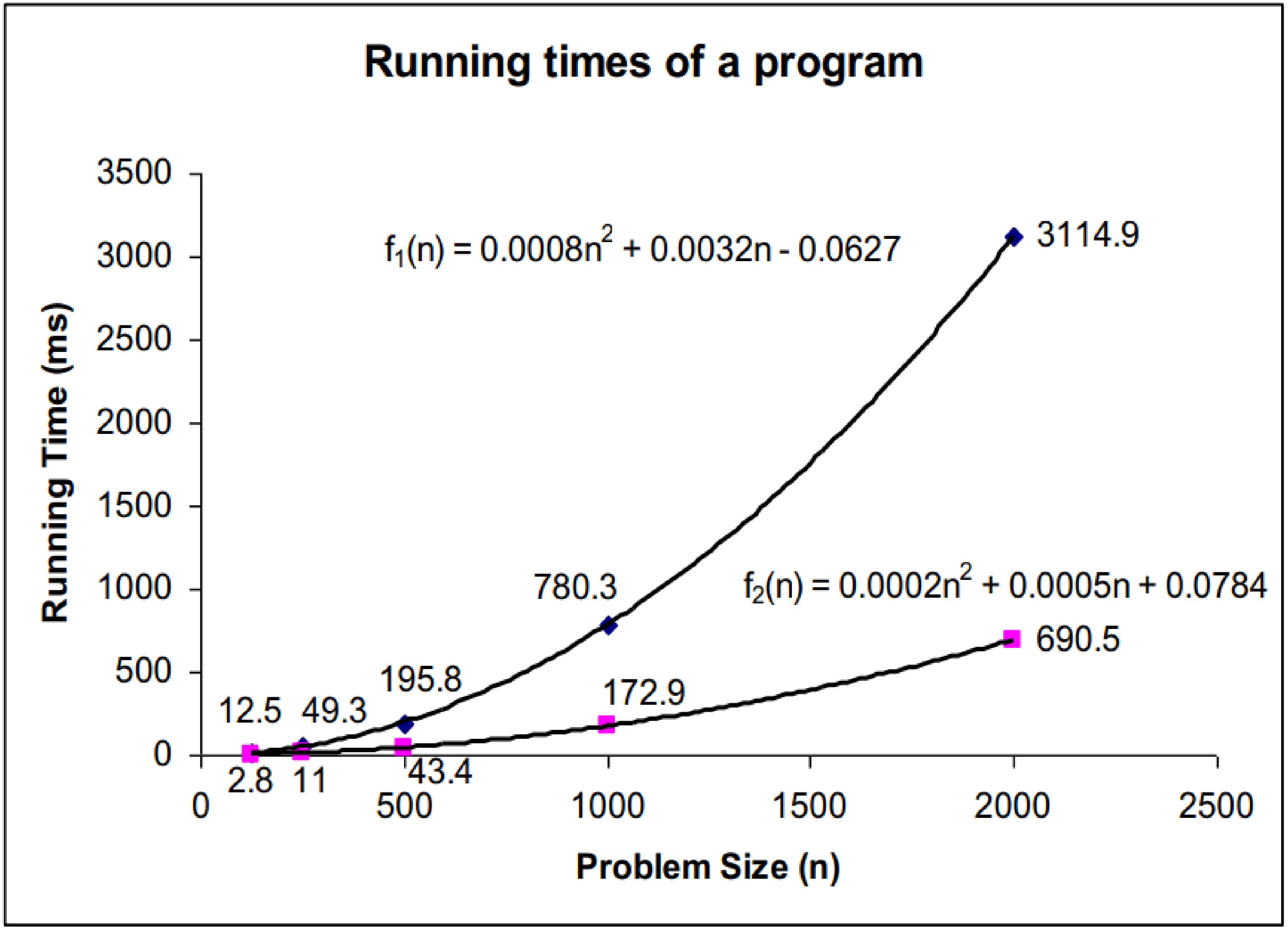
    $currentMax \leftarrow A[k]$

  end

end

return $currentMax$

Instrumentation to measure clock cycles (SENG 265)



Instrumentation to measure clock cycles (SENG 265)

- Limitations of this approach:
  - We must first implement the algorithm (in Java, C, C++, etc)
  - We need to determine which inputs to test, and how many
  - We must run all experiments in the same, controlled environment

- Earlier in this unit, our claim was that the methodology we will use in the course will:
  - take all possible inputs into account
  - compare the efficiency of two different algorithms independent from the underlying hardware and software configurations
  - be able to perform the analysis before needed to implement a solution

Using our methodology

Algorithm arrayMax(A, n)

$\boldsymbol{Input}$: An array A storing $n \geq 1$ integers

$\boldsymbol{Output}$: The maximum element in A

$currentMax \leftarrow A[0]$

for $\mathrm{k} \leftarrow 1$ to $\mathrm{n} - 1$ **do**

  if $currentMax < A[k]$ **then**

    $currentMax \leftarrow A[k]$

  end

end

return $currentMax$

Could count line numbers

Why are some arrows colored differently?

What are some other issues with counting line numbers?

Primitive Operations

- Our approach will be to count primitive operations:
  - Assignment statements (A)
  - Comparisons (C)
  - Boolean expressions (B)
  - Array indexing (I)
  - Selector/Object references (R)
  - Arithmetic operations:
    - Add/Subtract (S)
    - Multiplication/Division (M)
  - Trigonometric operations (sin, cos, tan) (T)
  - Method/function calls (M)

A for-loop declaration has multiple primitive operations (assignment, comparison, addition, possibly array indexing…)

Runtime analysis

- Categories of algorithm running times:
  - Best-case analysis: $T_b(n)$
  - Average-case analysis: $T_a(n)$
  - Worst-case analysis: $T(n)$
- Which is the best choice?

Algorithm arrayMax(A, n)

$\boldsymbol{Input}$: An array A storing $n \geq 1$ integers

$\boldsymbol{Output}$: The maximum element in A

$currentMax \leftarrow A[0]$
for $\mathrm{k} \leftarrow 1$ to $\mathrm{n} - 1$ **do**
  if $currentMax < A[k]$ **then**
    $currentMax \leftarrow A[k]$
  end
end
return $currentMax$

Runtime analysis

- Average-case
  - calculate the expected running times based on a given input data distribution
  - requires math and probability theory (we are getting there)
  - focused on more in CSC 226 (but mentioned in CSC 225 as well)
- Best-case
  - The minimum number of primitive operations (depending on *n*), given the most advantageous and best input configuration of size *n*
- Worst-case:
  - What is the maximum number of primitive operations (depending on *n*) executed by the algorithm, taken over all inputs of size *n*?
  - Worst-case analysis is most common and may aid in the design of the algorithm

Runtime analysis (worst-case)

Algorithm arrayMax(A, n)

$\boldsymbol{Input}$: An array A storing $n \geq 1$ integers

$\boldsymbol{Output}$: The maximum element in A

$currentMax \leftarrow A[0]$ 1I + 1A

for $\mathrm{k} \leftarrow 1$ to $\mathrm{n} - 1$ **do** 1A

if $currentMax < A[k]$ **then**

$currentMax \leftarrow A[k]$

end

end

return $currentMax$

Runtime analysis (worst-case)

How many times is our condition $\mathrm{k} \leq \mathrm{n}-1$ executed?

The n-1 times the condition is true, when k is $1, 2, 3, \ldots, \mathrm{n}$-1.

Once when the loop is terminated, when $\mathrm{k} = \mathrm{n}$.

Algorithm arrayMax(A, n)

Input: An array A storing $n \geq 1$ integers

Output : The maximum element in A

$currentMax \leftarrow A[0]$ — 1I + 1A

for $\mathrm{k} \leftarrow 1$ to $\mathrm{n}-1$ **do** — 1A + (n)*(1C)

    if $currentMax < A[k]$ **then**

        $currentMax \leftarrow A[k]$

    end

end

return $currentMax$

Runtime analysis (worst-case)

After each iteration of the loop, we will increment k by 1 (sum operation (S) followed by an assignment operation (A)).

This will happen at the end of all n-1 iterations of the loop.

Algorithm arrayMax(A, n)

$\boldsymbol{Input}$: An array A storing $n \geq 1$ integers

$\boldsymbol{Output}$: The maximum element in A

$currentMax \leftarrow A[0]$ 1I + 1A

for $\mathrm{k} \leftarrow 1$ to $\mathrm{n} - 1$ **do**   1A + (n)*(1C) + (n-1)*(1S+1A)

    if $currentMax < A[k]$ **then**

        $currentMax \leftarrow A[k]$

    end

end

return $currentMax$

Runtime analysis (worst-case)

Similarly, for all n-1 iterations of the loop the if-statement is executed (an array index (I) and a comparison (C)).

And in the worst-case, we update currentMax each time.

Algorithm arrayMax(A, n)

$\boldsymbol{Input}$: An array A storing $n \geq 1$ integers

$\boldsymbol{Output}$: The maximum element in A

$currentMax \leftarrow A[0]$	1I + 1A
for $\mathrm{k} \leftarrow 1$ to $\mathrm{n}-1$ **do**	1A + (n)*(1C) + (n-1)*(1S+1A)
if $currentMax < A[k]$ **then**	(n-1)*(1I + 1C)
$currentMax \leftarrow A[k]$	(n-1)*(1I + 1A)
end	
end	
return $currentMax$	

Runtime analysis (worst-case)

Algorithm arrayMax(A, n)

Input: An array A storing $n \geq 1$ integers

Output : The maximum element in A

$currentMax \leftarrow A[0]$	1I + 1A
for $\mathrm{k} \leftarrow 1$ to $\mathrm{n} - 1$ **do**	1A + (n)*(1C) + (n-1)*(1S+1A)
if $currentMax < A[k]$ **then**	(n-1)*(1I + 1C)
$currentMax \leftarrow A[k]$	(n-1)*(1I + 1A)
end	
end	
return $currentMax$	1A

$$\boxed{\begin{aligned} T(n) &= 4 + (n-1)(6) + n \\ T(n) &= 7n - 2 \end{aligned}}$$