Data Structures and Algorithms

Lecture 2: Analysis of Algorithms, Asymptotic notation

Lilia Georgieva
Outline

- Pseudocode
- Theoretical Analysis of Running time
  - Primitive Operations
  - Counting primitive operations
- Asymptotic analysis of running time
Pseudocode

- In this course, we will mostly use pseudocode to describe an algorithm.
- Pseudocode is a 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

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 ...

Algorithm *arrayMax*(A, n)

Input: array A of n integers

Output: maximum element of A

\[ \text{currentMax} \leftarrow A[0] \]

for \( i \leftarrow 1 \) to \( n - 1 \) do

  if \( A[i] > \text{currentMax} \) then

    \[ \text{currentMax} \leftarrow A[i] \]

return \text{currentMax}
Pseudocode Details

- Method call
  
  \[ \text{var.method (arg [, arg ...])} \]

- Return value
  
  \[ \text{return expression} \]

- Expressions
  
  - Assignment
    (like = in Java)
  
  - Equality testing
    (like == in Java)

  \( n^2 \) superscripts and other mathematical formatting allowed

Algorithm \( \text{arrayMax}(A, n) \)

**Input:** array \( A \) of \( n \) integers

**Output:** maximum element of \( A \)

\[
\begin{align*}
\text{currentMax} & \leftarrow A[0] \\
\text{for } i & \leftarrow 1 \text{ to } n - 1 \text{ do} \\
\quad & \text{if } A[i] > \text{currentMax} \text{ then} \\
\quad & \quad \text{currentMax} \leftarrow A[i] \\
\text{return } \text{currentMax}
\end{align*}
\]
Comparing Algorithms

- Given 2 or more algorithms to solve the same problem, how do we select the best one?

- Some criteria for selecting an algorithm
  1) Is it easy to implement, understand, modify?
  2) How long does it take to run it to completion?
  3) How much of computer memory does it use?

- Software engineering is primarily concerned with the first criteria

- In this course we are interested in the second and third criteria
Comparing Algorithms

- Time complexity
  - The amount of time that an algorithm needs to run to completion
- Space complexity
  - The amount of memory an algorithm needs to run
- We will occasionally look at space complexity, but we are mostly interested in time complexity in this course
- Thus in this course the better algorithm is the one which runs faster (has smaller time complexity)
How to Calculate Running time

- Most algorithms transform input objects into output objects

\[
\begin{array}{cccc}
5 & 3 & 1 & 2 \\
\end{array} \xrightarrow{\text{sorting algorithm}} \begin{array}{cccc}
1 & 2 & 3 & 5 \\
\end{array}
\]

- The running time of an algorithm typically grows with the input size
  - idea: analyze running time as a function of input size
How to Calculate Running Time

- Even on inputs of the same size, running time can be very different
  - Example: algorithm that finds the first prime number in an array by scanning it left to right
- Idea: analyze running time in the
  - best case
  - worst case
  - average case
How to Calculate Running Time

- Best case running time is usually useless
- Average case time is very useful but often difficult to determine
- We focus on the **worst case** running time
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics
Experimental Evaluation of Running Time

- 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

![Graph showing input size vs. time](image)
Limitations of Experiments

- Experimental evaluation of running time is very useful but
  - 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 of Running Time

- Uses a pseudo-code 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.
For theoretical analysis, we assume RAM model for our “theoretical” computer

Our RAM model consists of:

- 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.
For theoretical analysis, we will count **primitive** or **basic** operations, which are simple computations performed by an algorithm.

Basic operations are:

- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)
- Assumed to take a constant amount of time in the RAM model
Primitive Operations

Examples of primitive operations:
- Evaluating an expression: \( x^2 + e^y \)
- Assigning a value to a variable: \( \text{cnt} \leftarrow \text{cnt} + 1 \)
- Calling a method: \( \text{mySort(A,n)} \)
- Returning from a method: \( \text{return(cnt)} \)
Counting Primitive Operations

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

**Algorithm arrayMax(A, n)**

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

Total 

```
2 + n  
2(n − 1)  
2(n − 1)  
2(n − 1)  
1
```

```
Total 7n − 1
```

Analysis of Algorithms 17
Estimating Running Time

- Algorithm \textit{arrayMax} executes $7n - 1$ 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
  \[ a (7n - 1) \leq T(n) \leq b(7n - 1) \]
- 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 $T(n)$

- Thus we focus on the big-picture which is the **growth rate** of an algorithm

- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm *arrayMax*
  - Algorithm *arrayMax* grows proportionally with $n$, with its true running time being $n$ times a constant factor that depends on the specific computer.
Constant Factors

- 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

- How do we get rid of the constant factors to focus on the essential part of the running time?
Big-Oh Notation Motivation

- The big-Oh notation is used widely to characterize running times and space bounds.
- The big-Oh notation allows us to ignore constant factors and lower order terms and focus on the main components of a function which affect its growth.
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 $n_0 = 10$
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 \( n_0 \geq 1 \) s.t. \( 3n^3 + 20n^2 + 5 \leq c \cdot n^3 \) 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 \( n_0 \geq 1 \) s.t. \( 3 \log n + 5 \leq c \cdot \log n \) 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 “$f(n) \in O(g(n))$” 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>Condition</th>
<th>$f(n) \in O(g(n))$</th>
<th>$g(n) \in O(f(n))$</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>

Analysis of Algorithms 25
Big-Oh Rules

- If is $f(n)$ a polynomial of degree $d$, then $f(n)$ is $O(n^d)$, i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say “$2n$ is $O(n)$” instead of “$2n$ is $O(n^2)$”
- Use the simplest expression of the class
  - Say “$3n + 5$ is $O(n)$” instead of “$3n + 5$ is $O(3n)$”
Big-Oh Rules

- If \( f(n) \) is a polynomial of degree \( d \), then \( f(n) \) is \( O(n^d) \), i.e.,
  1. Drop lower-order terms
  2. Drop constant factors
- Use the smallest possible class of functions
  - Say “\( 2n \) is \( O(n) \)” instead of “\( 2n \) is \( O(n^2) \)”
- Use the simplest expression of the class
  - Say “\( 3n + 5 \) is \( O(n) \)” instead of “\( 3n + 5 \) 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 primitive operations executed as a function of the input size.
  - We express this function with big-Oh notation.

- Example:
  - We determine that algorithm `arrayMax` executes at most \( 7n - 1 \) primitive operations.
  - We say that algorithm `arrayMax` “runs in \( O(n) \) time.”

- Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations.
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
Computing Prefix Averages

- We further illustrate asymptotic analysis with two algorithms for prefix averages.
- The $i$-th prefix average of an array $X$ is average of the first $(i+1)$ elements of $X$:
  \[ A[i] = \frac{(X[0] + X[1] + \ldots + X[i])}{(i+1)} \]
- Computing the array $A$ of prefix averages of another array $X$ has applications to financial analysis.
Prefix Averages (Quadratic)

- The following algorithm computes prefix averages in quadratic time by applying the definition.

**Algorithm `prefixAverages1(X, n)`**

**Input** array $X$ of $n$ integers

**Output** array $A$ of prefix averages of $X$

#operations

$A \leftarrow \text{new array of } n \text{ integers} \quad n$

$\text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do} \quad n$

$s \leftarrow X[0] \quad n$

$\text{for } j \leftarrow 1 \text{ to } i \text{ do} \quad 1 + 2 + \ldots + (n - 1)$

$s \leftarrow s + X[j] \quad 1 + 2 + \ldots + (n - 1)$

$A[i] \leftarrow s / (i + 1) \quad n$

return $A \quad 1$
The running time of \texttt{prefixAverages1} is $O(1 + 2 + \ldots + n)$.

The sum of the first $n$ integers is $n(n + 1)/2$.

- There is a simple visual proof of this fact.

Thus, algorithm \texttt{prefixAverages1} runs in $O(n^2)$ time.
The following algorithm computes prefix averages in linear time by keeping a running sum:

**Algorithm** \( \text{prefixAverages2}(X, n) \)

**Input** array \( X \) of \( n \) integers

**Output** array \( A \) of prefix averages of \( X \)

\[
\begin{align*}
A & \leftarrow \text{new array of } n \text{ integers} \\
s & \leftarrow 0 \\
\text{for } i & \leftarrow 0 \text{ to } n - 1 \text{ do} \\
& \quad s \leftarrow s + X[i] \\
& \quad A[i] \leftarrow s / (i + 1) \\
\text{return } A
\end{align*}
\]

Algorithm \( \text{prefixAverages2} \) runs in \( O(n) \) time.
More Examples

Algorithm $\text{SumTripleArray}(X, n)$

Input triple array $X[][][][]$ of $n$ by $n$ by $n$ integers

Output sum of elements of $X$  #operations

\begin{align*}
    s &\leftarrow 0 & 1 \\
    &\text{for } i \leftarrow 0 \text{ to } n - 1 \text{ do} & n \\
    &\quad \text{for } j \leftarrow 0 \text{ to } n - 1 \text{ do} & n + n + \ldots + n = n^2 \\
    &\quad \text{for } k \leftarrow 0 \text{ to } n - 1 \text{ do} & n^2 + n^2 + \ldots + n^2 = n^3 \\
    &\quad \quad s &\leftarrow s + X[i][j][k] & n^2 + n^2 + \ldots + n^2 = n^3 \\
    &\text{return } s & 1
\end{align*}

- Algorithm $\text{SumTripleArray}$ runs in $O(n^3)$ time
Useful Big-Oh Rules

- If $f(n)$ is a polynomial of degree $d$, then $f(n)$ is $O(n^d)$

  $$f(n) = a_0 + a_1 n + a_2 n^2 + \ldots + a_d n^d$$

- If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$ then
  - $d(n) + e(n)$ is $O(f(n) + g(n))$
  - $d(n)e(n)$ is $O(f(n)g(n))$

- If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$ then $d(n)$ is $O(g(n))$

- If $p(n)$ is a polynomial in $n$ then $\log p(n)$ is $O(\log(n))$
Relatives of Big-Oh

- **big-Omega**
  - $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

- **big-Theta**
  - $f(n)$ is $\Theta(g(n))$ if there are constants $c' > 0$ and $c'' > 0$ and an integer constant $n_0 \geq 1$ such that $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$ for $n \geq n_0$
Intuition for Asymptotic Notation

**Big-Oh**
- f(n) is $O(g(n))$ if f(n) is asymptotically less than or equal to g(n)

**big-Omega**
- f(n) is $\Omega(g(n))$ if f(n) is asymptotically greater than or equal to g(n)
- Note that f(n) is $\Omega(g(n))$ if and only if g(n) is $O(f(n))$

**big-Theta**
- f(n) is $\Theta(g(n))$ if f(n) is asymptotically equal to g(n)
- Note that f(n) is $\Theta(g(n))$ if and only if if g(n) is $O(f(n))$ and if f(n) is $O(g(n))$
Example Uses of the Relatives of Big-Oh

- \(5n^2\) is \(\Omega(n^2)\)
  
  \(f(n)\) is \(\Omega(g(n))\) if there is a constant \(c > 0\) and an integer constant \(n_0 \geq 1\) such that \(f(n) \geq c \cdot g(n)\) for \(n \geq n_0\)
  
  Let \(c = 5\) and \(n_0 = 1\)

- \(5n^2\) is \(\Omega(n)\)
  
  \(f(n)\) is \(\Omega(g(n))\) if there is a constant \(c > 0\) and an integer constant \(n_0 \geq 1\) such that \(f(n) \geq c \cdot g(n)\) for \(n \geq n_0\)
  
  Let \(c = 1\) and \(n_0 = 1\)

- \(5n^2\) is \(\Theta(n^2)\)
  
  \(f(n)\) is \(\Theta(g(n))\) if it is \(\Omega(n^2)\) and \(O(n^2)\). We have already seen the former, for the latter recall that \(f(n)\) is \(O(g(n))\) if there is a constant \(c > 0\) and an integer constant \(n_0 \geq 1\) such that \(f(n) \leq c \cdot g(n)\) for \(n \geq n_0\)
  
  Let \(c = 5\) and \(n_0 = 1\)
Math you need to Review

- Summations
- Logarithms and Exponents
  - properties of logarithms:
    \[ \log_b(xy) = \log_b x + \log_b y \]
    \[ \log_b (x/y) = \log_b x - \log_b y \]
    \[ \log_b x^a = a \log_b x \]
    \[ \log_b a = \log_x a/\log_x b \]
  - properties of exponentials:
    \[ a^{(b+c)} = a^b a^c \]
    \[ a^{bc} = (a^b)^c \]
    \[ a^b /a^c = a^{(b-c)} \]
    \[ b = a^{\log_a b} \]
    \[ b^c = a^{c \log_a b} \]
Final Notes

- Even though in this course we focus on the asymptotic growth using big-Oh notation, practitioners do care about constant factors occasionally.
- Suppose we have 2 algorithms:
  - Algorithm A has running time $30000n$
  - Algorithm B has running time $3n^2$
- Asymptotically, algorithm A is better than algorithm B.
- However, if the problem size you deal with is always less than 10000, then the quadratic one is faster.