Lecture 2:
Analysis of Algorithms
Asymptotic notation

Instructor: Olga Vekslar
Outline

- Comparing algorithms
- Pseudocode
- Theoretical Analysis of Running time
  - primitive Operations
  - counting primitive operations
- Asymptotic analysis of running time
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 algorithm needs to run to completion

- **Space complexity**
  - The amount of memory algorithm needs to run

- Occasionally will look at space complexity, but mostly interested in time complexity in this course

- Thus 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

- Algorithm running time typically grows with the input size

- **Analyze running time as a function of input size**
  - $T(n)$, where $n$ is integer expressing the input size
  - Example: $n$ is the size of the input array
How to Calculate Running Time

- Even on inputs of the same size, running time can be 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 running 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

- Implementing algorithm
- Run the program with inputs of varying size and composition
- Use System.currentTimeMillis() to measure actual running time
- Plot results
Limitations of Experiments

- Experimental evaluation of running time is useful but
  - necessary to implement algorithm, which may be difficult
  - results may not be indicative of the running time on other inputs not included in the experiment
  - 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 implementation
- Characterizes running time as a function of the input size, $n$
- Takes into account all possible inputs
- Allows to evaluate the speed of algorithm independent of hardware/software
We 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, and hides program design issues.

Example: find max array element

Algorithm \text{arrayMax}(A, n)
\begin{itemize}
  \item \textbf{Input:} array $A$ of $n$ integers
  \item \textbf{Output:} maximum element of $A$
\end{itemize}

\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*}
Pseudocode Details

- **Control flow**
  - `if ... then ... [else ...]`
  - `while ... do ...`
  - `repeat ... until ...`
  - `for ... do ...`
  - Indentation replaces braces

- **Method declaration**

  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`

  Algorithm `method (arg, arg...)`
  
  **Input ...**  
  **Output ...**
Pseudocode Details

- Method call
  ```
  var.method (arg [, arg ...])
  ```
- Return value
  ```
  return expression
  ```
- Expressions
  - Assignment (like = in Java)
  - Equality testing (like == in Java)
  - $n^2$ superscripts and other mathematical formatting allowed

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 \( \text{currentMax} \)
Primitive Operations

- For theoretical analysis, count **primitive** or **basic** operations
  - are simple computations performed by algorithm

- Basic operations are:
  - Identifiable in pseudocode
  - Largely independent from the programming language
  - Exact definition not important (will see why later)
  - **Assumed to take a constant amount of time**
    - i.e. independent of the input size
Primitive Operations

- Examples of primitive operations:
  - evaluating an expression \( x^2 + e^y \)
  - assigning a value to a variable \( \text{cnt} \leftarrow \text{cnt} + 1 \)
  - indexing into an array \( A[5] \)
  - calling a method \( \text{mySort}(A,n) \)
  - returning from a method \( \text{return}(\text{cnt}) \)
Counting Primitive Operations

- Determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

**Algorithm arrayMax(A, n)**

```
Algorithm arrayMax(A, n) # operations
    currentMax ← A[0] 2
    for i ← 1 to n − 1 do
        if A[i] > currentMax then
            currentMax ← A[i] 2(n − 1)
        { increment counter i } 2(n − 1)
    return currentMax 1

Total 7n − 1
```
Estimating Running Time

- Algorithm \texttt{arrayMax} executes \(7n - 1\) primitive operations in the worst case.

- Let
  
  \(a\) = time taken by the fastest primitive operation

  \(b\) = time taken by the slowest primitive operation

- Let \(T(n)\) be worst-case time of \texttt{arrayMax}, then

  \[a(7n - 1) \leq T(n) \leq b(7n - 1)\]

- bounded by two linear functions
Growth Rate of Running Time

\[ a(7n - 1) \leq T(n) \leq b(7n - 1) \]

- \( T(n) \) has a **linear growth** rate
  - grows proportionally with \( n \), i.e. running time is \( n \) times a constant factor
- Changing hardware/software environment affects \( T(n) \) by a constant factor, but does not change growth rate
- Thus linear growth rate of \( T(n) \) is an intrinsic property of algorithm **arrayMax**
- Want to focus on the **growth rate** of an algorithm, i.e. “the big picture”
Growth Rates Examples

These 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 $\approx 2^n$
## Comparison of Growth Rates

<table>
<thead>
<tr>
<th>n</th>
<th>log(n)</th>
<th>n</th>
<th>nlog(n)</th>
<th>n^2</th>
<th>n^3</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4096</td>
<td>65536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1024</td>
<td>32768</td>
<td>4.3x10^9</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4096</td>
<td>262144</td>
<td>1.8x10^19</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16384</td>
<td>2097152</td>
<td>3.4x10^38</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>256</td>
<td>2048</td>
<td>65536</td>
<td>16777218</td>
<td>1.2x10^77</td>
</tr>
<tr>
<td>Running Time in ms (10^{-3} of sec)</td>
<td>Maximum Problem Size (n)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>--------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1000 ms \ (1 \text{ second})</td>
<td>60000 ms \ (1 \text{ minute})</td>
<td>36*10^5 ms \ (1 \text{ hour})</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>1000</td>
<td>60,000</td>
<td>3,600,000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n^2</td>
<td>32</td>
<td>245</td>
<td>1,897</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2^n</td>
<td>10</td>
<td>16</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Constant Factors

- The growth rate is not affected by:
  - constant factors or
  - lower-order (slowly growing) terms

- Examples
  - $10^2n + 10^5$ is a linear function
  - $10^5n^2 + 10^8n$ is a quadratic function

- How do we “ignore” constant factors and focus on the essential part (growth rate) of the running time?
Asymptotic Analysis Motivation

- big-Oh notation is used widely to characterize running times and space bounds
- big-Oh notation allows us to
  - ignore constant factors and
  - ignore lower order terms
  - focus on the main components that affect growth rate
Asymptotic Analysis Motivation

- Want to show $T(n)$ has some growth rate
  - Example: $T(n) = 2n + 10$
- Rename $T(n) = f(n)$ for now
  - consistency with the most commonly used notation
- Growth rate is also some function, call it $g(n)$
  - $g(n) = n$, $g(n) = n^2$, etc.
- Want to show $f(n)$ has growth rate $g(n)$
  - Example: $f(n) = 2n + 10$ has growth rate $g(n) = n$
- **Main step:** show $f(n)$ grows slower or the same as $g(n)$
Main step: show $f(n)$ grows slower or the same as $g(n)$

Need to show $f(n) \leq g(n)$
Asymptotic Analysis Motivation

- Need to show \( f(n) \leq g(n) \)

- initial range of \( n \) can be ignored, interested in what happens eventually, for large \( n \)
  - thus name “asymptotic” analysis

- formally: for \( n \geq n_0 \), for some constant integer \( n_0 \geq 1 \)
Constants do not affect growth rate, want to ignore them.

Instead of \( f(n) \leq g(n) \) show \( f(n) \leq c \cdot g(n) \) for a positive constant \( c \)

and, as before, for \( n \geq n_0 \), for some constant integer \( n_0 \geq 1 \)
Big-Oh Notation Definition

- Given functions $f(n)$ and $g(n)$, we say that $f(n)$ is $O(g(n))$ if there is a real constant $c > 0$ and an integer constant $n_0 \geq 1$ such that

$$f(n) \leq cg(n) \text{ for } n \geq n_0$$

- Meaning: $f(n)$ does not grow faster than $g(n)$ asymptotically
Example Proof

- Show that $2n + 10$ is $O(n)$
  - $2n + 10 \leq cn$
  - $(c - 2) n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and $n_0 = 10$

- For $c = 3$, we took the exact intersection point for $n_0$
- But $n_0 = 11, 12, \ldots$ would work just as well
- There are infinitely many choices for $c$ and $n_0$
“Negative” Example

Example: \( n^2 \) is not \( O(n) \)

- Suppose \( n^2 \leq cn \) for some \( c > 0 \) and all \( n \geq n_0 \)
- \( n \leq c \) for some \( c > 0 \) and all \( n \geq n_0 \)
- take \( m = \lceil c \rceil + n_0 \)
- \( m > c \) and \( m > n_0 \)
- Contradiction!
More Big-Oh Examples

- $7n + 2$ is $O(n)$

Proof:

- $7n + 2 \leq 7n + 2n$ for all $n \geq 1$

- $7n + 2 \leq 9n$ for all $n \geq 1$

Take $c = 9$ and $n_0 = 1$, then

- $7n + 2 \leq cn$ for all $n \geq n_0$
More Big-Oh Examples

- $3n^3 + 20n^2 + 5$ is $O(n^3)$

**Proof:**

$3n^3 + 20n^2 + 5 \leq 3n^3 + 20n^3 + 5n^3$ for $n \geq 1$

$3n^3 + 20n^2 + 5 \leq 28n^3$ for $n \geq 1$

Take $c = 28$, $n_0 = 1$ then

$3n^3 + 20n^2 + 5 \leq cn^3$ for $n \geq n_0$
More Big-Oh Examples

- $3 \log n + 5$ is $O(\log n)$

Proof:

$3 \log n + 5 \leq 3 \log n + 5 \log n = 8 \log n$ for $n \geq 2$

$3 \log n + 5 \leq 8 \log n$ for $n \geq 2$

Take $c = 8$, $n_0 = 2$, then

$3 \log n + 5 \leq c \log n$ for $n \geq n_0$
Big-Oh Etiquette

- Use the smallest possible class of functions
  - $2n$ is $O(n^2)$
    - true but is nota as accurate and informative
    - therefore considered to be a “poor taste”
  - $2n$ is $O(n)$
    - precise and accurate
    - preferred statement

- Use the simplest expression of the class
  - $3n + 5$ is $O(3n)$
    - true but more complicated than needed
  - $3n + 5$ is $O(n)$
    - preferred statement
Big-Oh is an Upper Bound

- Statement “\( f(n) \) is \( O(g(n)) \)” means that the growth rate of \( f(n) \) is no more than the growth rate of \( g(n) \)
- Thus big-Oh notation gives an upper bound on the growth rate of a function

<table>
<thead>
<tr>
<th>( g(n) ) grows more</th>
<th>( f(n) ) is ( O(g(n)) )</th>
<th>( g(n) ) is ( O(f(n)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

- Are there theoretical concepts that say
  - \( f(n) \) grows at the same rate as \( g(n) \)?
  - \( f(n) \) and \( g(n) \) have the same growth rate?
big-Omega

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

- means $f(n)$ is asymptotically greater than or equal to $g(n)$

- $f(n)$ is $\Omega(g(n))$ if and only if $g(n)$ is $O(f(n))$
Big-Theta, a Relative of Big-Oh

- **big-Theta**
  - \( f(n) \) is \( \Theta(g(n)) \) if there are real constants \( c' > 0 \) and \( c'' > 0 \) and an integer constant \( n_0 \geq 1 \) such that
    \[
    c'g(n) \leq f(n) \leq c''g(n) \text{ for } n \geq n_0
    \]
  - means \( f(n) \) is asymptotically equal to \( g(n) \)
  - \( f(n) \) is \( \Theta(g(n)) \) if and only if \( g(n) \) is \( O(f(n)) \) and if \( f(n) \) is \( O(g(n)) \)
Big-Oh Polynomial Rule

\[ f(n) = a_0 + a_1n + a_2n^2 + \ldots + a_d n^d \]

- If \( f(n) \) is a polynomial of degree \( d \), then \( f(n) \) is \( O(n^d) \)
  1. Drop lower-order terms
  2. Drop constant factors
Useful Big-Oh Rules

1. If \( d(n) \) is \( O(f(n)) \) and \( e(n) \) is \( O(g(n)) \) then
   \[
   d(n) + e(n) \text{ is } O(f(n) + g(n))
   \]
   \[
   d(n)e(n) \text{ is } O(f(n)g(n))
   \]

2. If \( d(n) \) is \( O(f(n)) \) and \( f(n) \) is \( O(g(n)) \) then
   \[
   d(n) \text{ is } O(g(n))
   \]

3. If \( p(n) \) is a polynomial in \( n \) then
   \[
   \log p(n) \text{ is } O(\log(n))
   \]
Asymptotic Algorithm Analysis

- Asymptotic algorithm analysis determines running time in big-Oh notation (or big-Theta, or big-Omega)

- To perform the asymptotic analysis
  - find the worst-case number of primitive operations executed as a function of input size
  - express this function with big-Oh notation

- Example:
  - Algorithm arrayMax executes at most $7n - 1$ primitive operations
  - Algorithm arrayMax runs in $O(n)$ time

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

- The $i$-th prefix average of an array $X$ is average of first $(i + 1)$ elements of $X$:
  \[ A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{i+1} \]

- Computing array $A$ of prefix averages of another array $X$ has applications to financial analysis.
Algorithm \textit{prefixAverages1}(X, n)

\textbf{Input} \ array \( X \) \ of \( n \) \ integers

\textbf{Output} \ array \( A \) \ of \ prefix \ averages \ of \( X \)

\begin{itemize}
  \item \( A \leftarrow \) \ new \ array \ of \( n \) \ integers
  \item \textbf{for} \( i \leftarrow 0 \) \ \textbf{to} \( n - 1 \) \ \textbf{do}
    \begin{itemize}
      \item \( s \leftarrow X[0] \)
      \item \textbf{for} \( j \leftarrow 1 \) \ \textbf{to} \( i \) \ \textbf{do}
        \begin{itemize}
          \item \( s \leftarrow s + X[j] \)
        \end{itemize}
      \item \( A[i] \leftarrow s / (i + 1) \)
    \end{itemize}
  \end{itemize}
\end{itemize}

Return \( A \)
Arithmetic Progression

- Running time of \texttt{prefixAverages1} is \(O(1 + 2 + \ldots + n)\)

- Adding up

\[
\begin{align*}
S &= 1 + 2 + \ldots + n \\
S &= n + (n-1) + \ldots + 1 \\
2S &= (n+1) + (n+1) + \ldots + (n+1)
\end{align*}
\]

2S = (n+1)n

\[
S = (n+1)n / 2
\]

- The sum of the first \(n\) integers is \(n(n + 1)/2\)

- \texttt{prefixAverages1} runs in \(O(n^2)\) time
Algorithm `prefixAverages2(X, n)`

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

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

1. $A \leftarrow$ new array of $n$ integers
2. $s \leftarrow 0$
3. for $i \leftarrow 0$ to $n - 1$ do
   1. $s \leftarrow s + X[i]$
   2. $A[i] \leftarrow s / (i + 1)$
4. return $A$

- $4n + 2$ is $O(n)$
- Algorithm `prefixAverages2` runs in $O(n)$ time
Math you need to Review

- Summations
- Logarithms and Exponents
  - **properties of logarithms:**
    \[
    \log_b(xy) = \log_b x + \log_b y \\
    \log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y \\
    \log_b x^a = a \log_b x \\
    \log_b a = \frac{\log_x a}{\log_x b}
    \]
  - **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

- We focus on the asymptotic growth using big-Oh notation, but practitioners do care about constant factors occasionally.

- Suppose
  - 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.