Analysis of Algorithms
Outline and Reading

- Running time (§1.1)
- Pseudo-code (§1.1)
- Counting primitive operations (§1.1)
- Asymptotic notation (§1.2)
- Asymptotic analysis (§1.2)
- Case study (§1.3.1, §1.4)
Running Time

- The running time of an algorithm varies with the input and typically grows with the input size.
- Average case difficult to determine.
- We focus on the worst case running time:
  - Easier to analyze.
  - Crucial to applications such as games, finance and robotics.

![Graph showing running time vs input size for best case, average case, and worst case.](image)
Experimental Studies

- 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 time vs input size]
Limitations of Experiments

- 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

- Uses a high-level description of the algorithm instead of an implementation
- Takes into account all possible inputs
- Allows us to evaluate the speed of an algorithm independent of the hardware/software environment
Pseudocode

- 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 …

Method call
var.method (arg [, arg…])

Return value
return expression

Expressions
Assignment (like = in Java)
Equality testing (like == in Java)
Superscripts and other mathematical formatting allowed
Primitive Operations

- Basic computations performed by an algorithm
- Identifiable in pseudocode
- Largely independent from the programming language
- Exact definition not important (we will see why later)

Examples:
- Evaluating an expression
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method
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)**

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

**# operations**

- 2
- 2 + `n`
- 2(`n – 1`)
- 2(`n – 1`)
- 2(`n – 1`)
- 1

**Total** 7`n – 1`
Estimating Running Time

Algorithm \textit{arrayMax} executes $7n - 1$ primitive operations in the worst case.

Define

- $a$ Time taken by the fastest primitive operation.
- $b$ Time taken by the slowest primitive operation.

Let $T(n)$ be the actual worst-case running time of \textit{arrayMax}. We have

$$a(7n - 1) \leq T(n) \leq b(7n - 1)$$

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 $T(n)$

- The linear growth rate of the running time $T(n)$ is an intrinsic property of algorithm `arrayMax`
Growth Rates

- Growth rates of functions:
  - Linear $\approx n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$

- In a log-log chart, the slope of the line corresponds to the growth rate of the function
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.
Big-Oh Notation

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 $n_0 = 10$
Big-Oh Notation (cont.)

- 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
The big-Oh notation gives an upper bound on the growth rate of a function. The statement “$f(n)$ is $O(g(n))$” means that the growth rate of $f(n)$ is no more than the growth rate of $g(n)$. We can use the big-Oh notation to rank functions according to their growth rate:

<table>
<thead>
<tr>
<th></th>
<th>$f(n)$ is $O(g(n))$</th>
<th>$g(n)$ is $O(f(n))$</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>
Classes of Functions

Let \( \{g(n)\} \) denote the class (set) of functions that are \( O(g(n)) \)

We have
\[
\{n\} \subset \{n^2\} \subset \{n^3\} \subset \{n^4\} \subset \{n^5\} \subset \ldots
\]

where the containment is strict
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 $O(3n)$”
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.
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 $X$.
  \[ A[i] = X[0] + X[1] + \ldots + X[i] \]
- Computing the array $A$ of prefix averages of another array $X$ has applications to financial analysis.

![Bar chart showing the arrays $X$ and $A$]
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`

<table>
<thead>
<tr>
<th>#operations</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n</code></td>
</tr>
</tbody>
</table>

1. `A ← ` new array of `n` integers
2. `n`
3. `for i ← 0 to n - 1 do`
4. `1 + 2 + ... + i`
5. `A[i] ← ` `s / (i + 1)`
6. `n`

return `A`
Arithmetic Progression

The running time of `prefixAverages1` is $O(1 + 2 + \ldots + n)$

The sum of the first $n$ integers is $\frac{n(n + 1)}{2}$

- There is a simple visual proof of this fact

Thus, algorithm `prefixAverages1` runs in $O(n^2)$ time
Prefix Averages (Linear)

The following algorithm computes prefix averages in linear time by keeping a running sum.

Algorithm \textit{prefixAverages2}(X, n)

<table>
<thead>
<tr>
<th>Input</th>
<th>array \textit{X} of \textit{n} integers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>array \textit{A} of prefix averages of \textit{X}</td>
</tr>
<tr>
<td>\textit{A}</td>
<td>new array of \textit{n} integers</td>
</tr>
<tr>
<td>\textit{s}</td>
<td>0</td>
</tr>
</tbody>
</table>

\begin{minipage}{0.7\textwidth}
\begin{algorithm}
\SetAlgoLined
\textbf{for} \textit{i} $\leftarrow$ 0 \textbf{to} \textit{n} $-$ 1 \textbf{do}
\State \textit{s} $\leftarrow$ \textit{s} + \textit{X}[\textit{i}]
\State \textit{A}[\textit{i}] $\leftarrow$ \textit{s} / (\textit{i} + 1)
\EndFor
\end{algorithm}
\end{minipage}

\textbf{Algorithm} \textit{prefixAverages2} runs in \textit{O(n)} time.