An **algorithm** is a step-by-step procedure for solving a problem in a finite amount of time.

### Running Time

- Most algorithms transform input objects into output objects.
- The running time of an algorithm typically grows with the input size.
- Average case time is 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 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

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
- 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

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
  return expression

- Return value

- Expressions
  ← Assignment (like = in Java)
  = Equality testing (like == in Java)
  \( n^2 \) Superscripts and other mathematical formatting allowed

---

The Random Access Machine (RAM) Model

- A CPU

- An 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.
Seven Important Functions (§4.1)

- Seven functions that often appear in algorithm analysis:
  - Constant = 1
  - Logarithmic = \( \log n \)
  - Linear = \( n \)
  - \( N-\log N = n \log n \)
  - Quadratic = \( n^2 \)
  - Cubic = \( n^3 \)
  - Exponential = \( 2^n \)

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

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)
- Assumed to take a constant amount of time in the RAM model

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 \texttt{arrayMax}(A, n)
\begin{align*}
\text{currentMax} & \leftarrow A[0] & \text{# operations} & \quad 2 \\
\text{for } i & \leftarrow 1 \text{ to } n - 1 \text{ do} & & \quad 2n \\
\quad \text{if } A[i] > \text{currentMax} \text{ then} & & \quad 2(n - 1) \\
\quad \text{currentMax} & \leftarrow A[i] & & \quad 2(n - 1) \\
\{ \text{ increment counter } i \} & & \quad 2(n - 1) \\
\text{return currentMax} & & 1 \\
\end{align*}

Total \quad 8n - 2

Estimating Running Time

Algorithm \texttt{arrayMax} executes \(8n - 2\) primitive operations in the worst case. Define:
\begin{align*}
a & = \text{Time taken by the fastest primitive operation} \\
b & = \text{Time taken by the slowest primitive operation} \\
\end{align*}

Let \(T(n)\) be worst-case time of \texttt{arrayMax}. Then
\[ a \cdot (8n - 2) \leq T(n) \leq b \cdot (8n - 2) \]

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 \textit{arrayMax}

Constant Factors

- The growth rate is not affected by
  - constant factors or lower-order terms
- Examples
  - $10^n + 10^5$ is a linear function
  - $10^n + 10^2n$ is a quadratic function
Big-Oh Notation (§4.2.5)

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 Example

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 $n_0 \geq 1$ such that $3n^3 + 20n^2 + 5 \leq c \cdot n^3$ 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 $n_0 \geq 1$ such that $3 \log n + 5 \leq c \cdot \log n$ 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 “$f(n)$ 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>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>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Same growth</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
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 \( \text{arrayMax} \) executes at most \( 8n - 2 \) primitive operations
  - We say that algorithm \( \text{arrayMax} \) “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 $X$: $A[i] = (X[0] + X[1] + \ldots + X[i]) / (i+1)$.
- 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

$A \leftarrow$ new array of $n$ integers  

$n$

for $i \leftarrow 0$ to $n - 1$ do  

$s \leftarrow X[0]$  

$n$

for $j \leftarrow 0$ to $i$ do  

$s \leftarrow s + X[j]$  

$1 + 2 + \ldots + (n - 1)$

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

$1 + 2 + \ldots + (n - 1)$

return $A$  

1
Arithmetic Progression

- The running time of `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 `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 `prefixAverages2(X, n)`
  
  **Input** array $X$ of $n$ integers
  
  **Output** array $A$ of prefix averages of $X$  

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

  Algorithm `prefixAverages2` runs in $O(n)$ time
Math you need to Review

- Summations
- Logarithms and Exponents
- Proof techniques
- Basic probability

**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:**
- \( a^{b+c} = a^b a^c \)
- \( a^{bc} = (a^b)^c \)
- \( a^b / a^c = a^{b-c} \)
- \( b = a^{\log_a b} \)
- \( b^c = a^{c \log_a b} \)

---

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 \( g(n) \)

**Big-Theta**
- \( f(n) \) is \( \Theta(g(n)) \) if \( f(n) \) is asymptotically equal to \( g(n) \)

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 \( n_0 = 1 \)