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

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.

We focus on the worst case running time.

Easier to analyze
- Crucial to applications such as games, finance and robotics.

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.

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.

Pseudocode (§1.1)

Example: find max element of an array

```plaintext
Algorithm maxElement(A, n)
Input array A of n integers
Output maximum element of A

1. currentMax ← A[0]
2. for i ← 1 to n - 1 do
   3. if A[i] > currentMax then
      4. currentMax ← A[i]
5. return currentMax
```

Example: find max element of an array

```plaintext
Algorithm maxElement(A, n)
Input array A of n integers
Output maximum element of A

1. currentMax ← A[0]
2. for i ← 1 to n - 1 do
   3. if A[i] > currentMax then
      4. currentMax ← A[i]
5. return currentMax
```
**Pseudocode Details**

- Control flow
  - if
  - else
  - while
  - repeat
  - until
  - for
- Method declaration
  - Algorithm `method arg1 arg2 ...`
  - Input
  - Output

**The Random Access Machine (RAM) Model**

- A CPU
- An 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.

**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

**Counting Primitive Operations (§1.1)**

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

**Estimating Running Time**

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

- Growth rates of functions:
  - Linear: \( n \)
  - Quadratic: \( n^2 \)
  - Cubic: \( n^3 \)

- 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.

Big-Oh Notation (§1.2)

- 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 \( n \geq n_0 \).

Examples

\[ 2n + 10 \text{ is } O(n) \]
\[ n^2 + 10 \text{ is } O(n^2) \]

Big-Oh Example

- Example: the function \( n^3 \) is not \( O(n) \).

More Big-Oh Examples

- \( 7n^2 \): \( O(n^2) \)
  - need \( c > 0 \) and \( n_0 > 1 \) such that \( 7n^2 \leq cn^2 \) for \( n \geq n_0 \)
  - this is true for \( c = 7 \) and \( n_0 = 1 \)

- \( 3n^3 + 20n^2 + 5 \) is \( O(n^3) \)
  - need \( c > 0 \) and \( n_0 > 1 \) such that \( 3n^3 + 20n^2 + 5 \leq cn^3 \) for \( n \geq n_0 \)
  - this is true for \( c = 4 \) and \( n_0 = 21 \)

- \( 3 \log n + \log \log n \) is \( O(\log n) \)
  - need \( c > 0 \) and \( n_0 > 3 \) such that \( 3 \log n + \log \log n \leq cn \) for \( n \geq n_0 \)
  - this is true for \( c = 4 \) 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 \( f(n) \text{ is } O(g(n)) \) 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>( f(n) \text{ is } O(g(n)) )</th>
<th>( g(n) \text{ is } O(f(n)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) ) grows more</td>
<td>( g(n) ) grows more</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Same growth</td>
<td></td>
</tr>
</tbody>
</table>

Yes

Big-Oh Rules

- If is \( f(n) \) a polynomial of degree \( d \), then \( f(n) \) is \( O(n^d) \), i.e.,
  - Drop lower-order terms
  - 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 \( X \):
- 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

1. \( A[0] \leftarrow \text{new array of } n \text{ integers} \)
2. \( s \leftarrow X[0] \)
3. \( A[0] \leftarrow s / (1 + 1) \)
4. \( i \leftarrow 0 \) to \( n - 1 \) do
5. \( s \leftarrow X[i] \)
6. \( A[i] \leftarrow s / (i + 1) \)
7. return \( A \)
```

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 \( X \) #operations

1. \( A \leftarrow \text{new array of } n \text{ integers} \)
2. \( s \leftarrow 0 \)
3. \( i \leftarrow 0 \) to \( n - 1 \) do
4. \( s \leftarrow s + X[i] \)
5. \( A[i] \leftarrow s / (i + 1) \)
6. return \( A \)
```

Arithmetic Progression

- The running time of \( \text{prefixAverages1} \) is \( \Theta(n + 1) \)
- The sum of the first \( n \) integers is \( \frac{n(n + 1)}{2} \)
  - There is a simple visual proof of this fact
- Thus, \( \text{prefixAverages1} \) runs in \( \Theta(n) \) time
Analysis of Algorithms 25

Math you need to Review
- Summations (Sec. 1.3.1)
- Logarithms and Exponents (Sec. 1.3.2)
  - properties of logarithms:
    \[ \log_b(xy) = \log_b x + \log_b y \]
    \[ \log_b \left(\frac{x}{y}\right) = \log_b x - \log_b y \]
    \[ \log_b x^a = a \log_b x \]
  - properties of exponentials:
    \[ a^{b+c} = a^b a^c \]
    \[ a^{bc} = (a^b)^c \]
    \[ a^{b/c} = \frac{a^b}{a^c} \]
- Proof techniques (Sec. 1.3.3)
- Basic probability (Sec. 1.3.4)

Analysis of Algorithms 26

Relatives of Big-Oh

- big-Omega
  - \( f(n) \) is \( \Omega(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \) 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 \( n_0 \) such that \( c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n) \) for \( n \geq n_0 \)
- little-oh
  - \( f(n) \) is \( o(g(n)) \) if, for any constant \( c > 0 \), there is an integer constant \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for \( n \geq n_0 \)
- little-omega
  - \( f(n) \) is \( \omega(g(n)) \) if, for any constant \( c > 0 \), there is an integer constant \( n_0 \) such that \( f(n) \geq c \cdot g(n) \) for \( n \geq n_0 \)

Analysis of Algorithms 27

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 \( g(n) \)
- little-oh
  - \( f(n) \) is \( o(g(n)) \) if \( f(n) \) is asymptotically strictly less than \( g(n) \)
- little-omega
  - \( f(n) \) is \( \omega(g(n)) \) if \( f(n) \) is asymptotically strictly greater than \( g(n) \)

Analysis of Algorithms 28

Example Uses of the Relatives of Big-Oh

- \( 5n^2 \text{is } O(n^2) \)
  - \( f(n) \) is \( O(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for \( n \geq n_0 \)
  - let \( c = 5 \) and \( n_0 = 1 \)
- \( 5n \text{is } O(n) \)
  - \( f(n) \) is \( O(g(n)) \) if there is a constant \( c > 0 \) and an integer constant \( n_0 \) such that \( f(n) \leq c \cdot g(n) \) for \( n \geq n_0 \)
  - let \( c = 1 \) and \( n_0 = 1 \)
- \( 5n^2 \text{is } O(n^3) \)
  - \( f(n) \) is \( O(g(n)) \) if there is a constant \( c > 0 \), there is an integer constant \( n_0 \), 0 such that \( f(n) \leq c \cdot g(n) \) for \( n \geq n_0 \)
  - need \( 5n^2 \leq c \cdot n^3 \) given \( c \), the \( n_0 \) that satisfies this is \( n_0 = 5 \cdot 5 = 25 \)