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

Running Time (§3.1)

Experimental Studies (§ 3.1.1)

Write a program implementing the algorithm
Run the program with inputs of varying size and composition
Use a function, like the built-in clock() function, 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 (§3.1.2)

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/Function call
  - var.method (arg, arg ...)
- Return value
  - return expression
- Expressions
  - Assignment (like in C++)
  - Equality testing (like in C++)
  - 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.

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 (§3.4.1)

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

```
Algorithm arrayMax(A, n)
# operations
currentMax ← A[0] 2
for i ← 1 to n − 1 do
  if A[i] > currentMax then
    currentMax ← A[i]
    { increment counter i }
return currentMax
```
Total 7n − 1

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 worst-case time of arrayMax. Then
  - a (7n − 1) ≤ T(n) ≤ 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.

**Examples**

- \( 10^2n + 10^5 \) is a linear function
- \( 10^5n^2 + 10^8n \) is a quadratic function

**Big-Oh Notation (§3.5)**

- Given functions \( f(n) \) and \( g(n) \), we say that \( f(n) = O(g(n)) \) 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 \( O(n) \).

**Big-Oh Example**

- Example: the function \( n^2 \) is not \( O(n) \).

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

- \( 3\log n + \log\log n \) is \( O(n) \).

**More Big-Oh Examples**

- \( 7n^2 \) is \( O(n^2) \).
  - This is true for \( c = 7 \) and \( n_0 = 1 \).

- \( 3n^3 + 20n^2 + 5 \) is \( O(n^3) \).
  - This is true for \( c = 4 \) and \( n_0 = 21 \).

- \( 3\log n + \log\log n \) is \( O(n) \).
  - 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 "\( f(n) = O(g(n)) \)" 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.
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 \texttt{arrayMax} executes at most $7n - 1$ primitive operations
  - We say that algorithm \texttt{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 $X$:
  $$A[i] = \frac{X[0] + X[1] + \ldots + X[i]}{i+1}$$
- 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 \texttt{prefixAverages1(X, n)}

\begin{itemize}
  \item Input array $X$ of $n$ integers
  \item Output array $A$ of prefix averages of $X$
  \item #operations
  \end{itemize}

\begin{verbatim}
A ← new array of $n$ integers
for i ← 0 to $n$ − 1 do
  s ← $X[0]$ + $X[1]$ + \ldots + $X[i]$
  for j ← 1 to $i$ do
    s ← s + $X[j]$
  A[i] ← s / $(i+1)$
return A
\end{verbatim}

- Algorithm \texttt{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 \texttt{prefixAverages2(X, n)}

\begin{itemize}
  \item Input array $X$ of $n$ integers
  \item Output array $A$ of prefix averages of $X$
  \item #operations
  \end{itemize}

\begin{verbatim}
A ← new array of $n$ integers
s ← 0
for i ← 0 to $n$ − 1 do
  s ← s + $X[i]$
  A[i] ← s / $(i+1)$
return A
\end{verbatim}

- Algorithm \texttt{prefixAverages2} runs in $O(n)$ time

Arithmetic Progression

- The running time of \texttt{prefixAverages1} is $O(1 + 2 + \ldots + n)$
- The sum of the first $n$ integers is $n(n + 1)/2$
- Thus, algorithm \texttt{prefixAverages1} runs in $O(n^2)$ time
Math you need to Review

- Summations (Sec. 1.3.1)
- Logarithms and Exponents (Sec. 1.3.2)

- properties of logarithms:
  \[ \log_b(xy) = \log_b x + \log_b y \]
  \[ \log_b \left( \frac{x}{y} \right) = \log_b x - \log_b y \]
  \[ \log_b a = \frac{\log_c x}{\log_c b} \]

- properties of exponentials:
  \[ a^{m+n} = a^m a^n \]
  \[ a^{-n} = \frac{1}{a^n} \]
  \[ \frac{a^m}{a^n} = a^{m-n} \]
  \[ b = a^{\log_b n} \]

- Proof techniques (Sec. 1.3.3)
- Basic probability (Sec. 1.3.4)

Intuition for Asymptotic Notation

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

**big-Omega**
- \( f(n) = \Omega(g(n)) \) if \( f(n) \) is asymptotically greater than or equal to \( g(n) \)

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

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

**little-omega**
- \( f(n) = \omega(g(n)) \) if \( f(n) \) is asymptotically strictly greater than \( g(n) \)

Example Uses of the Relatives of Big-Oh

\[ 5n^2 = \Omega(n) \]
\[ f(n) = O(g(n)) \] if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \leq cg(n) \) for \( n \geq n_0 \)
\[ \text{let } c = 5 \text{ and } n_0 = 1 \]

\[ 5n^2 = \Theta(n) \]
\[ f(n) = O(g(n)) \] if there is a constant \( c > 0 \) and an integer constant \( n_0 \geq 1 \) such that \( f(n) \leq cg(n) \) for \( n \geq n_0 \)
\[ \text{let } c = 1 \text{ and } n_0 = 1 \]

\[ 5n^2 = o(n) \]
\[ f(n) = O(g(n)) \] if, for any constant \( c > 0 \), there is an integer constant \( n_0 \geq 0 \) such that \( f(n) < cg(n) \) for \( n \geq n_0 \)
\[ \text{need } 5n^2 < cn_0 \rightarrow c \text{ given, the } n_0 \text{ that satisfies this is } n_0 = 5 \]