Outline
- Big-oh notation
- Big-theta notation
- Big-omega notation
- asymptotic algorithm analysis

Analysis of Algorithms

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**

```java
Algorithm arrayMax(A, n)
{
    Input array A of n integers
    Output maximum element of A
    currentMax = A[0];
    for (i = 1; i < n; i++)
        if (A[i] > currentMax)
            currentMax = A[i];
    return currentMax;
}
```

Java-Like Pseudocode Details

- Control flow
  - if ... [else ...]
  - while ...
  - do ... while ...
  - for ...
- Method declaration

```java
Algorithm method (arg [ , arg ...])
Input ...
Output ...
```

- Method call
  - var.method (arg [ , arg ...])
- Return value
  - return expression
- Expressions
  - Assignment
  - Equality testing
  - 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

- 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 `arrayMax(A, n)`

- `currentMax = A[0]`: 2 operations
- `for i = 1; i < n; i++`:
  - `if (A[i] > currentMax)`:
    - `currentMax = A[i]`: 2 operations
- `// increment counter i`:
- `return currentMax;`

Total: \(8n - 2\) operations

Estimating Running Time

Algorithm `arrayMax` executes \(8n - 2\) 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:

- \(a(8n - 2) \leq T(n) \leq b(8n - 2)\)

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 `arrayMax`.

Constant Factors

- The growth rate is not affected by constant factors or lower-order terms.
- Examples:
  - \(10^n + 10^2\) is a linear function
  - \(10^2n + 10^n\) is a quadratic function

Big-Oh Notation

- Given functions \(f(n)\) and \(g(n)\), we say that \(f(n) = \text{o}(g(n))\) 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 \(O(n)\)

- Examples:
  - \(2n + 10 \leq cn\) if \(c = 3\) and \(n_0 = 10\)
  - \(c = 2\) is \(O(n)\)
  - \(c = 10\) is \(O(n)\)
  - \(c = 100\) is \(O(n)\)
  - \(c = 1\) is \(O(n)\)
  - \(c = 1\) and \(n_0 = 10\)

Big-Oh Example

- Example: the function \(n^2\) is not \(O(n)\)
  - \(n^2 \leq cn\) for \(n \leq c\)
  - The above inequality cannot be satisfied since \(c\) must be a constant.
To perform the asymptotic analysis,

\[ 3n^2 + 20n + 5 \]

is \( O(n^2) \)

need \( c > 0 \) and \( n_0 \geq 1 \) such that \( 3n^2 + 20n + 5 \leq cn^2 \) for \( n \geq n_0 \)

this is true for \( c = 4 \) and \( n_0 = 21 \)

\[ 3 \log n + 5 \]

is \( O(\log n) \)

need \( c > 0 \) and \( n_0 \geq 1 \) such that \( 3 \log n + 5 \leq c\log n \) for \( n \geq n_0 \)

this is true for \( c = 8 \) and \( n_0 = 1 \)

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:

- Say "2n is \( O(n) \)" instead of "2n is \( O(n^2) \)"
- Use the smallest possible class of functions
- Use the simplest expression of the class
- Say "3n + 5 is \( O(n) \)" instead of "3n + 5 is \( O(3n^2) \"

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.

The following algorithm computes prefix averages in quadratic time by applying the definition:

```plaintext
Algorithm prefixAverages(X, n)

1. Input array \( X \) of \( n \) integers
2. Output array \( A \) of prefix averages of \( X \)
3. \( A = \) new array of \( n \) integers;
   \( A[0] = \) \( X[0] \);
4. for \( i = 1 \) to \( n-1 \) do
   5.   for \( j = 1 \) to \( i \) do
      6.     \( s = s + X[j] \);
    7.   \( A[i] = s / i + 1 \);
8. return \( A \);
```

We can use the big-Oh notation to rank functions according to their growth rate.

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

We determine that the algorithm \( \text{arrayMax} \) executes at most \( 5n + 2 \) primitive operations.

We say that the 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.

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 \( 5n + 2 \) primitive operations.
- We say that algorithm \( \text{arrayMax} \) "runs in \( O(n) \) time".

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 use the simplest expression of the class:

- Say "2n is \( O(n) \)" instead of "2n is \( O(n^2) \)"
- Use the smallest possible class of functions
- Use the simplest expression of the class
- Say "3n + 5 is \( O(n) \)" instead of "3n + 5 is \( O(3n^2) \"
The running time of \texttt{pref\texttt{2}xA\texttt{v}\texttt{e}\texttt{r}\texttt{s}\texttt{1}} 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 \texttt{prefixA\texttt{v}e\texttt{r}\texttt{e}\texttt{s}\texttt{1}} runs in $O(n^2)$ time.

Algorithm \texttt{prefixA\texttt{v}e\texttt{r}\texttt{e}\texttt{s}\texttt{2}} runs in $O(n)$ time.

\begin{itemize}
\item \textbf{Big-Theta} \quad \text{f}(n) is $\Theta(g(n))$ if $f(n)$ is asymptotically \textbf{equal} to $g(n)$
\end{itemize}

\begin{itemize}
\item \textbf{Big-Omega} \quad \text{f}(n) is $\Omega(g(n))$ if $f(n)$ is asymptotically \textbf{greater than or equal} to $g(n)$
\end{itemize}

\begin{itemize}
\item \textbf{Big-Oh} \quad \text{f}(n) is $O(g(n))$ if $f(n)$ is asymptotically \textbf{less than or equal} to $g(n)$
\end{itemize}

\textbf{Math you need to Review}

\begin{itemize}
\item \textbf{Summations}
\item \textbf{Logarithms and Exponents}
\item \textbf{Proof techniques}
\item \textbf{Basic probability}
\end{itemize}

\textbf{Prefix Averages (Linear)}

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

\begin{itemize}
\item Algorithm \texttt{prefixA\texttt{v}e\texttt{r}\texttt{e}\texttt{s}\texttt{2}} runs in $O(n)$ time
\end{itemize}

\textbf{Relatives of Big-Oh}

\begin{itemize}
\item \textbf{big-Omega} \quad \text{f}(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $\text{f}(n) \geq cg(n)$ for $n \geq n_0$
\item \textbf{big-Theta} \quad \text{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 $c'g(n) \leq f(n) \leq c''g(n)$ for $n \geq n_0$
\end{itemize}

Example Uses of the Relatives of Big-Oh

\begin{itemize}
\item $5n$ is $\Omega(n^2)$
\item \text{f}(n) is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that \text{f}(n) $\geq $ $cg(n)$ for $n \geq n_0$
\item \text{f}(n) is $\Theta(1)$
\item $f(n) = \Theta(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that \text{f}(n) $\geq $ $cg(n)$ for $n \geq n_0$
\item $5n^2$ is $\Omega(n^2)$
\item $f(n)$ is $O(1)$ if it is $O(g(n))$ for some $g(n)$. 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 $f(n) \leq cg(n)$ for $n \geq n_0$
\item \text{f}(n) is $\Theta(1)$
\item $f(n) = \Theta(g(n))$ if $g(n)$ is $O(1)$ and $O(1)$. 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 $f(n) \leq cg(n)$ for $n \geq n_0$
\end{itemize}
References

1. Chapter 4, Data Structures and Algorithms by Goodrich and Tamassia.