Mathematical Foundations

The Constant Function:

\[ f(n) = c \]

For some fixed constant \( c \), such as \( c = 5 \), \( c = 100 \) or \( c = 1000,000 \).

- Regardless of the value of \( n \), \( f(n) \) will always be equal to the constant value \( c \).
- The most fundamental constant function is:
  \[ g(n) = 1 \]
- Any constant function \( f(n) = c \), can be written as a constant \( c \) times \( g(n) \). That is:
  \[ f(n) = cg(n) \]
Mathematical Foundations

The Logarithm Function:

\[ f(n) = \log_b n \text{ for some constant } b > 1 \]

\[ x = \log_b n \text{ if and only if } b^x = n \]

- The value \( b \) is known as the base of the logarithm. The most common base for the logarithm function in computer sciences is 2. Therefore, it is typical not to write the base when it is 2:
  \[ \log n = \log_2 n \]

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

The linear Function:

\[ f(n) = n \]

The N-Log-N Function:

\[ f(n) = n\log n \]

The Quadratic Function:

\[ f(n) = n^2 \]

The Cubic Function:

\[ f(n) = n^3 \]

- All the function listed so far are part of a larger class of functions called the polynomial.

\[ f(n) = a_0 + a_1n + a_2n^2 + a_3n^3 + \ldots + a_dn^d \]

Where \( a_0, a_1, a_2, \ldots, a_d \) are constants, called coefficients of the polynomial, and \( a_d \neq 0 \). The integer \( d \), which indicates the highest power in the polynomial, is called the degree of the polynomial.
The Exponential Function:

\[ f(n) = b^n \]

where \( b \) is a positive constant, called the base, and the argument \( n \) is the exponent.

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

Comparing Growth Rate in Tabular form:

<table>
<thead>
<tr>
<th>Function</th>
<th>10</th>
<th>100</th>
<th>1,000</th>
<th>10,000</th>
<th>100,000</th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \log_2 n )</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>13</td>
<td>16</td>
<td>19</td>
</tr>
<tr>
<td>( n )</td>
<td>10</td>
<td>10^2</td>
<td>10^3</td>
<td>10^4</td>
<td>10^5</td>
<td>10^6</td>
</tr>
<tr>
<td>( n \times \log_2 n )</td>
<td>30</td>
<td>664</td>
<td>9,965</td>
<td>10^5</td>
<td>10^6</td>
<td>10^7</td>
</tr>
<tr>
<td>( n^2 )</td>
<td>10^2</td>
<td>10^4</td>
<td>10^6</td>
<td>10^8</td>
<td>10^10</td>
<td>10^12</td>
</tr>
<tr>
<td>( n^3 )</td>
<td>10^3</td>
<td>10^6</td>
<td>10^9</td>
<td>10^12</td>
<td>10^15</td>
<td>10^18</td>
</tr>
<tr>
<td>( 2^n )</td>
<td>10^3</td>
<td>10^{30}</td>
<td>10^{301}</td>
<td>10^{3,010}</td>
<td>10^{30,103}</td>
<td>10^{301,030}</td>
</tr>
</tbody>
</table>
Mathematical Foundations

Comparing Growth Rate in graphical form:

Analysis of Algorithms

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
Analysis of Algorithms

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.
Analysis of Algorithms

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 \( \text{arrayMax}(A, n) \)

Input array \( A \) of \( n \) integers

Output maximum element of \( A \)

\[
\text{currentMax} \leftarrow A[0]
\]

for \( i \leftarrow 1 \) to \( n - 1 \) do

if \( A[i] > \text{currentMax} \) then

\[
\text{currentMax} \leftarrow A[i]
\]

return \( \text{currentMax} \)
Analysis of Algorithms

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)
  - `n^2` 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).**

- **Assumed to take a constant execution time.**

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

- Primitive operation corresponds to a low-level instruction with a constant execution time.
- Instead of determining the specific execution time of each primitive operation, simply count how many primitive operations are executed.
- This operation count will correlate to an actual running time in a specific computer.
- By inspecting the pseudocode, we can determine the maximum number of primitive operations executed by an algorithm, as a function of the input size.

Counting Primitive Operations

<table>
<thead>
<tr>
<th>Statements</th>
<th>S/E</th>
<th>Freq.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Algorithm Sum1(a[],n)</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2 {</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 S = 0.0;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4 for i=1 to n do</td>
<td>1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td>5 s = s+a[i];</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>6 return s;</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>7 }</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

\[2n + 3\]
### Counting Primitive Operations

<table>
<thead>
<tr>
<th>Statements</th>
<th>S/E</th>
<th>Freq.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Algorithm Sum2(a[], n, m)</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2 {</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 for i=1 to n do;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 for j=1 to m do</td>
<td>1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td>5 s = s+a[i][j];</td>
<td></td>
<td>n(m+1)</td>
<td>n(m+1)</td>
</tr>
<tr>
<td>6 return s;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 }</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2nm+2n+2

### Estimating Running Time

- Algorithm **Sum1** executes 2n + 3 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 **Sum1**. Then
  \[
  a(2n + 3) \leq T(n) \leq b(2n + 3)
  \]
- 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 $Sum1$.

Why Growth Rate Matters

<table>
<thead>
<tr>
<th>if runtime is...</th>
<th>time for $n + 1$</th>
<th>time for $2n$</th>
<th>time for $4n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c \log n$</td>
<td>$c \log (n + 1)$</td>
<td>$c (\log n + 1)$</td>
<td>$c(\log n + 2)$</td>
</tr>
<tr>
<td>$cn$</td>
<td>$c (n + 1)$</td>
<td>$2cn$</td>
<td>$4cn$</td>
</tr>
<tr>
<td>$cn \log n$</td>
<td>$\sim c n \log n + c n$</td>
<td>$2cn \log n + 2cn$</td>
<td>$4cn \log n + 4cn$</td>
</tr>
<tr>
<td>$c n^2$</td>
<td>$\sim c n^2 + 2cn$</td>
<td>$4cn^2$</td>
<td>$16cn^2$</td>
</tr>
<tr>
<td>$cn^3$</td>
<td>$\sim c n^3 + 3cn^2$</td>
<td>$8cn^3$</td>
<td>$64cn^3$</td>
</tr>
<tr>
<td>$c 2^n$</td>
<td>$c 2^{n+1}$</td>
<td>$c 2^{2n}$</td>
<td>$c 2^{4n}$</td>
</tr>
</tbody>
</table>

runtime quadruples when problem size doubles
Comparison of Two Algorithms (an example)

insertion sort is \( \frac{n^2}{4} \)
merge sort is \( 2n \log n \)

sort a million items using a basic PC?
insertion sort takes roughly 70 hours
while
merge sort takes roughly 40 seconds

Constant Factors

- The growth rate is not affected by
  - constant factors or
  - lower-order terms
- Examples
  - \( 10^5 n + 10^5 \) is a linear function
  - \( 10^5 n^2 + 10^6 n \) 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 $O(n)$
  - $2n + 10 \leq cn$
  - $10 \leq (c - 2) n$
  - $10/(c - 2) \leq n$
  - 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 \( 7n-2 \leq cn \) 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 \( 3n^3 + 20n^2 + 5 \leq cn^3 \) 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 \( 3 \log n + 5 \leq c \log n \) 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.
**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 \( \text{Sum1} \) executes at most \( 2n + 3 \) primitive operations
  - We say that algorithm \( \text{Sum1} \) “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 (an example)

- 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.

Computing Prefix Averages (an example)

- The following algorithm computes prefix averages in quadratic time.

<table>
<thead>
<tr>
<th>Statements</th>
<th>S/E</th>
<th>Freq.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algorithm prefixAverages ( f(X, n) )</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>{</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>( A \leftarrow ) new array of ( n ) integers</td>
<td>1</td>
<td>( n )</td>
</tr>
<tr>
<td>4</td>
<td>for ( i \leftarrow 0 ) to ( n - 1 ) do</td>
<td>1</td>
<td>( n+1 )</td>
</tr>
<tr>
<td>5</td>
<td>( s \leftarrow X[0] )</td>
<td>1</td>
<td>( n )</td>
</tr>
<tr>
<td>6</td>
<td>for ( i \leftarrow 1 ) to ( i / )do</td>
<td>1</td>
<td>( 1+2+3+\ldots+(n-1) )</td>
</tr>
<tr>
<td>7</td>
<td>( s \leftarrow s + X[i] )</td>
<td>1</td>
<td>( 1+2+3+\ldots+(n-1) )</td>
</tr>
<tr>
<td>8</td>
<td>( A[i] \leftarrow s / (i+1) )</td>
<td>1</td>
<td>( n )</td>
</tr>
<tr>
<td>9</td>
<td>return ( A )</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>}</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Thus, Algorithm \( \text{prefixAverages} \) is \( O(n^2) \).
Computing Prefix Averages (an example)

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

<table>
<thead>
<tr>
<th>Statements</th>
<th>S/E</th>
<th>Freq.</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>n+1</td>
<td>n+1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td></td>
<td>0</td>
</tr>
</tbody>
</table>

Thus, Algorithm \textit{prefixAverages2} is $O(n)$. \hspace{1cm} 4n+3