Analysis of Algorithms I

Outline

• running time
• pseudocode
• primitive operations
• asymptotic notation

More case studies: linear search, insertion-sort, matrix multiplication
Motivation

- Very similar problem can have very different solutions. How to compare between different algorithms in order to select the best one for a particular application?
  - Correctness
  - Simplicity
  - Security
  - Maintainability
  - ...
  - Response time
  - Space usage

- In this course, we focus on the running time, with space usage being also of interest in some occasions.
Characters of Running Time

- Most algorithms transform input objects into output objects.
- Running time typically grows with input size, \( n \).
- Fixed \( n \), running time may still vary on different inputs: best case, average case, worst case.
  - Ex. linear search
    - insertion-sort
- We focus on the worst case running time.
  - A guarantee to the user
  - Crucial to applications such as games, finance and robotics

![Graph showing running time with input size](image)
Linear Search

Algorithm $\text{LinearSearch}(v, S[0, \ldots, n-1])$

Input: array $S$ of $n$ elements, value $v$

Output: position $i$ if $S[i] = v$; $-1$ if $v \not\in S$

\[
i \leftarrow 0
\]
while $(i < n)$ and $(S[i] \neq v)$ do
\[
i = i + 1
\]
if $i < n$
\[
\text{return } i
\]
else
\[
\text{return } -1
\]

Ex.

[3, 6, 1, 5, 8]

Running time

(fixed $n$)

- Best case?
- Average case?
- Worst case?
Experimental Evaluation of Running Time

- 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 of Running Time

- 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 $A$

$currentMax \leftarrow A[0]$

for $i \leftarrow 1$ to $n - 1$ do
  if $A[i] > currentMax$ then
    $currentMax \leftarrow A[i]$

return $currentMax$
Pseudocode Details

- **Control flow**
  - if … then … [else …]
  - while … do …
  - repeat … until …
  - for … do …
  - Indentation replaces braces

- **Method call**
  \[ \text{var.method (arg [, arg...])} \]

- **Return value**
  \[ \text{return expression} \]

- **Expressions**
  \[ \leftarrow \text{Assignment} \]
  \[ = \text{Equality testing} \]
  \[ n^2 \text{ Superscripts and other mathematical formatting allowed} \]
The Random Access Machine (RAM) Model

- RAM model is independent of hardware architectures.
- A CPU
  + - * / < > =?
- A 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, e.g., + - * /, <, >, ==
- 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** $arrayMax(A, n)$

- $currentMax \leftarrow A[0]$  
- for $i \leftarrow 1$ to $n - 1$ do 
  - if $A[i] > currentMax$ then 
    - $currentMax \leftarrow A[i]$ 
  
- return $currentMax$

<table>
<thead>
<tr>
<th># operations</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>$5n - 2$</td>
</tr>
</tbody>
</table>
Algorithm \textit{arrayMax} executes $5n - 2$ 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 \textit{arrayMax}. Then

$$a(5n - 2) \leq T(n) \leq b(5n - 2)$$

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 $T(n)$ of `arrayMax` is an intrinsic property of the algorithm
Seven Important Functions

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

- In a log-log chart, the slope of the line corresponds to the growth rate.
### Growth Rates of some Important Functions

<table>
<thead>
<tr>
<th>n</th>
<th>lg n</th>
<th>n</th>
<th>n lg n</th>
<th>n²</th>
<th>n³</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4,096</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1,024</td>
<td>32,768</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
<td>262,144</td>
<td>1.84 \times 10^{19}</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
<td>2,097,152</td>
<td>3.40 \times 10^{38}</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>256</td>
<td>2,048</td>
<td>65,536</td>
<td>16,777,216</td>
<td>1.15 \times 10^{77}</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
<td>134,217,728</td>
<td>1.34 \times 10^{154}</td>
</tr>
</tbody>
</table>
Growth Rates of some Important Functions
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>$c \lg n$</td>
<td>$c \lg(n+1)$</td>
<td>$c(\lg n+1)$</td>
<td>$c(\lg n+2)$</td>
</tr>
<tr>
<td>$cn$</td>
<td>$(n+1)$</td>
<td>$2cn$</td>
<td>$4cn$</td>
</tr>
<tr>
<td>$cn \lg n$</td>
<td>$\sim cn \lg n + cn$</td>
<td>$2cn \lg n + 2cn$</td>
<td>$4cn \lg n + 4cn$</td>
</tr>
<tr>
<td>$cn^2$</td>
<td>$\sim cn^2 + 2cn$</td>
<td>$4cn^2$</td>
<td>$16cn^2$</td>
</tr>
<tr>
<td>$cn^3$</td>
<td>$\sim cn^3 + 3cn^2$</td>
<td>$8cn^3$</td>
<td>$64cn^3$</td>
</tr>
<tr>
<td>$c2^n$</td>
<td>$2^{n+1}$</td>
<td>$2^{2n}$</td>
<td>$2^{4n}$</td>
</tr>
</tbody>
</table>

The table shows how the runtime changes when the problem size doubles. For example, when the runtime is $c \lg n$, the time for $n+1$ is $c \lg(n+1)$, for $2n$ it is $c(\lg n + 1)$, and for $4n$ it is $c(\lg n + 2)$. As the problem size increases, the runtime grows quadratically or exponentially, depending on the function of $n$.
Growth Rates of some Important Functions

<table>
<thead>
<tr>
<th>n</th>
<th>lg n</th>
<th>n</th>
<th>n lg n</th>
<th>n^2</th>
<th>n^3</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>3</td>
<td>8</td>
<td>24</td>
<td>64</td>
<td>512</td>
<td>256</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>16</td>
<td>64</td>
<td>256</td>
<td>4,096</td>
<td>65,536</td>
</tr>
<tr>
<td>32</td>
<td>5</td>
<td>32</td>
<td>160</td>
<td>1,024</td>
<td>32,768</td>
<td>4,294,967,296</td>
</tr>
<tr>
<td>64</td>
<td>6</td>
<td>64</td>
<td>384</td>
<td>4,096</td>
<td>262,144</td>
<td>1.84 x 10^{19}</td>
</tr>
<tr>
<td>128</td>
<td>7</td>
<td>128</td>
<td>896</td>
<td>16,384</td>
<td>2,097,152</td>
<td>3.40 x 10^{38}</td>
</tr>
<tr>
<td>256</td>
<td>8</td>
<td>256</td>
<td>2,048</td>
<td>65,536</td>
<td>16,777,216</td>
<td>1.15 x 10^{77}</td>
</tr>
<tr>
<td>512</td>
<td>9</td>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
<td>134,217,728</td>
<td>1.34 x 10^{154}</td>
</tr>
</tbody>
</table>
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
The “Big Idea”: 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 \( n \geq n_0 \).
Big-Oh Notation

- Example:

  $2n + 10$ is $O(n)$
  
  - $2n + 10 \leq cn$
  - $(c - 2) n \geq 10$
  - $n \geq 10/(c - 2)$
  - Pick $c = 3$ and
    
    $n_0 = 10$
  
  - How about
    
    $(c = 4, n_0 = 5)$?
    
    $(c = 12, n_0 = 1)$?
Big-Oh Example

Example:

- \( n^2 \) is not \( O(n) \)
  - \( n^2 \leq cn \) ?
  - \( n \leq c \) ?
  - The above inequality cannot be satisfied since \( c \) must be a constant

A contradiction!
More Big-Oh Examples

- **5n-2**
  - 5n-2 is $O(n)$
  - need $c > 0$ and $n_0 \geq 1$ such that $5n-2 \leq c \cdot n$ for $n \geq n_0$
  - this is true for $c = 5$ 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 $n_0 = 21$. (how about $c = 28$, $n_0 = 1$?)

- **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>$f(n)$ grows more</td>
<td>Yes</td>
<td>No</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 \( O(3n) \)”

“best” Big-Oh characterization
More Big-Oh Rules

- If $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$ then
  - $d(n) + e(n)$ is $O(f(n) + g(n))$
  - $d(n) e(n)$ is $O(f(n) g(n))$

- If $d(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$ then $d(n)$ is $O(g(n))$

- If $p(n)$ is a polynomial in $n$ then $\log p(n)$ is $O(\log(n))$
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 $5n - 2$ 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 the 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 `prefixAverages1(X, n)`

- **Input** array `X` of `n` integers
- **Output** array `A` of prefix averages of `X`

1. `A ← new array of `n` integers
2. for `i ← 0` to `n − 1`
   1. `s ← X[0]`
   2. for `j ← 1` to `i`
      1. `s ← s + X[j]`
      2. `A[i] ← s / (i + 1)`
3. return `A`
The running time of \textit{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 \textit{prefixAverages1} runs in \( O(n^2) \) time.
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
- $A \leftarrow$ new array of $n$ integers $n$
- $s \leftarrow 0$ $1$
- for $i \leftarrow 0$ to $n - 1$ do $n$
  - $s \leftarrow s + X[i]$ $n$
  - $A[i] \leftarrow s / (i + 1)$ $n$
- return $A$ $1$

Algorithm `prefixAverages2` runs in $O(n)$ time.
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 $n_0 = 1$
Cautions about Big-Oh

- The constant factors do matter when they are large.
  
  **Ex.**
  
  \[ T_A(n) = 30000n \]
  
  \[ T_B(n) = 3n^2 \]

- Asymptotically, \( T_A \) is linear and \( T_B \) is quadratic.

- In practice, if \( n \) is no more than 10000, algorithm B should be used.
Comparison of two Sorting Algorithms

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

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

This is a slow machine, but if 100x faster then it’s 40 minutes versus less than 0.5 seconds.

sort 50 items?

by Matt Stallmann
Math You need to Review

- **Summations**

- **Logarithms**
  - Properties of logarithms:
    - $\log_b(xy) = \log_bx + \log_by$
    - $\log_b(x/y) = \log_bx - \log_by$
    - $\log_bx^a = a\log_bx$
    - $\log_ba = \log_xa/\log_xb$

- **Exponents**
  - Properties of exponentials:
    - $a^{(b+c)} = a^ba^c$
    - $a^{bc} = (a^b)^c$
    - $a^b/a^c = a^{(b-c)}$
    - $b = a^{\log_ba}$
    - $b^c = a^{c\log_ab}$
Exercises:
Analysis of linear search, insertion-sort and classic matrix multiplication
**Insertion-sort**

Algorithm *InsertionSort*(A[1, ..., n])

**Input:** array A of n elements


for j ← 2 to n do

    key ← A[j]

    i ← j - 1

    while (i > 0) and (A[i] > key) do

        A[i+1] ← A[i]

        i ← i - 1

    A[i+1] ← key

---

Ex.

[8, 2, 4, 9, 3, 6]

Running time (fixed n)

- Best case?
- Average case?
- Worst case?