Teams

**Task:** answer questions to get team points
- Score kept during semester
- Periodic immediate rewards

**Divide room into 4 teams**
- rearrange now if you like,
- sit same place in later classes,
- double agents allowed from my point of view, and
- citizen-exchange allowed from my point of view.
Teams

Choose Team Captain

Choose Team Names

- (stage left)
- (stage right)
- (stage far) S.S. H
- (stage near) The Backrows
You have 43 fish in a pond.
17 fish drown.
[TEAMS] How many fish are still alive?
43

(was told to me by then 7-year old son)
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 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

- 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 \( A \)

\[
\text{currentMax} \leftarrow A[0] \\
\text{for } i \leftarrow 1 \text{ to } n - 1 \text{ do} \\
\quad \text{if } A[i] > \text{currentMax} \text{ then} \\
\quad \quad \text{currentMax} \leftarrow A[i] \\
\text{return currentMax}
\]
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++)

\(n^2\) 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

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)`

1. `currentMax ← A[0]`
2. `for i ← 1 to n − 1 do`
   - `if A[i] > currentMax then`
     - `currentMax ← A[i]`
   - `{ increment counter i }`
3. `return currentMax`

# operations

- `2`
- `2 + n`
- `2(n − 1)`
- `2(n − 1)`
- `2(n − 1)`
- `1`

Total: `7n − 1`
Estimating Running Time

Algorithm \textit{arrayMax} executes $7n - 1$ primitive operations in the worst case.

Define:

- $a = \text{Time taken by the fastest primitive operation}$
- $b = \text{Time taken by the slowest primitive operation}$

Let $T(n)$ be worst-case time of \textit{arrayMax}. Then

$$a (7n - 1) \leq T(n) \leq b(7n - 1)$$

Hence, the running time $T(n)$ is bounded by two linear functions
Power Law

Log-log plot. Plot running time $T(N)$ vs. input size $N$ using log-log scale.
Power Law

Log-log plot. Plot running time $T(N)$ vs. input size $N$ using log-log scale.

Regression. Fit straight line through data points: $a N^b$.

Hypothesis. The running time is about $1.006 \times 10^{-10} \times N^{2.999}$ seconds.

Let $T(N) = a N^b$, where $a = 2^c$

Thus $\log(T(N)) = b \log N + c$

After regression:

$b = 2.999$

$c = -33.2103$
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 $\approx n$
  - Quadratic $\approx n^2$
  - Cubic $\approx n^3$

- In a log-log chart, the slope of the line corresponds to the growth rate of the function.
## Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>problem size solvable in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1970s</td>
</tr>
<tr>
<td></td>
<td>1980s</td>
</tr>
<tr>
<td></td>
<td>1990s</td>
</tr>
<tr>
<td></td>
<td>2000s</td>
</tr>
<tr>
<td>1</td>
<td>any</td>
</tr>
<tr>
<td>log N</td>
<td>any</td>
</tr>
<tr>
<td>N</td>
<td>millions</td>
</tr>
<tr>
<td>N log N</td>
<td>hundreds of thousands</td>
</tr>
<tr>
<td>N^2</td>
<td>hundreds</td>
</tr>
<tr>
<td>N^3</td>
<td>hundred</td>
</tr>
<tr>
<td>2^N</td>
<td>any</td>
</tr>
</tbody>
</table>

- **1**: Any problem size is solvable in minutes.
- **log N**: Any problem size is solvable in minutes.
- **N**: Problem sizes of tens of millions, hundreds of millions, billions can be handled.
- **N log N**: Problem sizes of hundreds of thousands, millions can be handled.
- **N^2**: Problem sizes of hundreds, thousands can be handled.
- **N^3**: Problem sizes of hundred, hundreds, thousand can be handled.
## Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>problem size solvable in minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1970s</td>
</tr>
<tr>
<td>1</td>
<td>any</td>
</tr>
<tr>
<td>log N</td>
<td>any</td>
</tr>
<tr>
<td>N</td>
<td>millions</td>
</tr>
<tr>
<td>N log N</td>
<td>hundreds of thousands</td>
</tr>
<tr>
<td>N^2</td>
<td>hundreds</td>
</tr>
<tr>
<td>N^3</td>
<td>hundred</td>
</tr>
<tr>
<td>2^N</td>
<td>20</td>
</tr>
</tbody>
</table>
## Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>name</th>
<th>description</th>
<th>effect on a program that runs for a few seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>time for 100x more data</td>
</tr>
<tr>
<td>1</td>
<td>constant</td>
<td>independent of input size</td>
<td>–</td>
</tr>
<tr>
<td>log N</td>
<td>logarithmic</td>
<td>nearly independent of input size</td>
<td>–</td>
</tr>
<tr>
<td>N</td>
<td>linear</td>
<td>optimal for N inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>N log N</td>
<td>linearithmic</td>
<td>nearly optimal for N inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>N^2</td>
<td>quadratic</td>
<td>not practical for large problems</td>
<td>several hours</td>
</tr>
<tr>
<td>N^3</td>
<td>cubic</td>
<td>not practical for medium problems</td>
<td>several weeks</td>
</tr>
<tr>
<td>2^N</td>
<td>exponential</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>name</th>
<th>description</th>
<th>effect on a program that runs for a few seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>constant</td>
<td>independent of input size</td>
<td>–</td>
</tr>
<tr>
<td>log N</td>
<td>logarithmic</td>
<td>nearly independent of input size</td>
<td>–</td>
</tr>
<tr>
<td>N</td>
<td>linear</td>
<td>optimal for N inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>N log N</td>
<td>linearithmic</td>
<td>nearly optimal for N inputs</td>
<td>a few minutes</td>
</tr>
<tr>
<td>N^2</td>
<td>quadratic</td>
<td>not practical for large problems</td>
<td>several hours</td>
</tr>
<tr>
<td>N^3</td>
<td>cubic</td>
<td>not practical for medium problems</td>
<td>several weeks</td>
</tr>
<tr>
<td>2^N</td>
<td>exponential</td>
<td>useful only for tiny problems</td>
<td>forever</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>time for 100x more data</th>
<th>size for 100x faster computer</th>
</tr>
</thead>
<tbody>
<tr>
<td>a few minutes</td>
<td>100x</td>
</tr>
<tr>
<td>several hours</td>
<td>10x</td>
</tr>
<tr>
<td>several weeks</td>
<td>4–5x</td>
</tr>
<tr>
<td>forever</td>
<td>1x</td>
</tr>
</tbody>
</table>
## Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>problem size solvable in minutes</th>
<th>time to process millions of inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>log N</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>N</td>
<td>millions</td>
<td>tens of millions</td>
</tr>
<tr>
<td>N log N</td>
<td>hundreds of thousands</td>
<td>millions</td>
</tr>
<tr>
<td>N^2</td>
<td>hundreds</td>
<td>thousand</td>
</tr>
<tr>
<td>N^3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>problem size solvable in minutes</th>
<th>time to process millions of inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>log N</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>N</td>
<td>millions</td>
<td>tens of millions</td>
</tr>
<tr>
<td>N log N</td>
<td>hundreds of thousands</td>
<td>millions</td>
</tr>
<tr>
<td>N²</td>
<td>hundreds</td>
<td>thousand</td>
</tr>
<tr>
<td>N³</td>
<td>hundred</td>
<td>hundreds</td>
</tr>
</tbody>
</table>
## Practical implications of order-of-growth

<table>
<thead>
<tr>
<th>growth rate</th>
<th>problem size solvable in minutes</th>
<th>time to process millions of inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>log N</td>
<td>any</td>
<td>any</td>
</tr>
<tr>
<td>N</td>
<td>millions</td>
<td>tens of millions</td>
</tr>
<tr>
<td>N log N</td>
<td>hundreds of thousands</td>
<td>millions</td>
</tr>
<tr>
<td>N^2</td>
<td>hundreds</td>
<td>thousand</td>
</tr>
<tr>
<td>N^3</td>
<td>hundred</td>
<td>hundreds</td>
</tr>
</tbody>
</table>
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

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 $O(n)$.

For what $c$ and $n$ is $cg(n) \geq f(n)$?

- $2n + 10 \leq cn$
- $(c - 2) n \geq 10$
- $n \geq 10/(c - 2)$
- 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

e.g. Is there a c and n₀ such that c•n ≥ 7n-2?

What is the big-Oh?

7n-2 is \( O(n) \)

[TEAMS] What are the values for c and n₀ ?
Need c > 0 and n₀ ≥ 1 such that 7n-2 ≤ c•n for n ≥ n₀
This is true for c = 7 and n₀ = 1
More Big-Oh Examples

- \(2n^2 + 16\)
  \(2n^2 + 16\) is \(O(n^2)\)
  need \(c > 0\) and \(n_0 \geq 1\) such that \(2n^2 + 16 \leq c \cdot n^2\) for \(n \geq n_0\)
  this is true for \(c = 3\) and \(n_0 = 4\)

- \(3n^3 + 2n^2 + 9\)
  \(3n^3 + 2n^2 + 9\) is \(O(n^3)\)
  need \(c > 0\) and \(n_0 \geq 1\) such that \(3n^3 + 2n^2 + 9 \leq c \cdot n^3\) for \(n \geq n_0\)
  this is true for \(c = 4\) and \(n_0 = 3\)
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></th>
<th>$f(n)$ is $O(g(n))$</th>
<th>$g(n)$ is $O(f(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$g(n)$ grows more</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>$f(n)$ grows more</td>
<td>No</td>
<td>Yes</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 \( O(3n) \)”
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 $X$:
  \[ A[i] = \frac{(X[0] + X[1] + \ldots + X[i])}{(i+1)} \]
- Computing the array $A$ of prefix averages of another array $X$ has applications to financial analysis.

![Graph showing prefix averages](image)
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 $X$

$n \leftarrow$ new array of $n$ integers

for $i \leftarrow 0$ to $n - 1$ do

$s \leftarrow X[0]$

for $j \leftarrow 1$ to $i$ do

$s \leftarrow s + X[j]$

$A[i] \leftarrow s / (i + 1)$

return $A$
Arithmetic Progression

- 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
Prefix Averages

- Asymptotic analysis tells us the complexity of the original algorithm.

[TEAMS] Given such a tool, how can we make the algorithm more efficient?
Prefix Averages (Linear)

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

Algorithm \(\text{prefixAverages2}(X, n)\)

**Input** array \(X\) of \(n\) integers

**Output** array \(A\) of prefix averages of \(X\)

\[
\begin{align*}
A & \leftarrow \text{new array of } n \text{ integers} \\
& \quad \text{#operations} \\
s & \leftarrow 0 \\
& \quad n \\
\text{for } i & \leftarrow 0 \text{ to } n - 1 \text{ do} \\
& \quad s \leftarrow s + X[i] \\
& \quad A[i] \leftarrow s / (i + 1) \\
& \text{return } A \\
\end{align*}
\]

Algorithm \(\text{prefixAverages2}\) runs in \(O(n)\) time.
Math you need to Review

- Summations
- Logarithms and Exponents

- **properties of logarithms:**
  \[ \log_b(xy) = \log_b x + \log_b y \]
  \[ \log_b (x/y) = \log_b x - \log_b y \]
  \[ \log_b x^a = a\log_b x \]
  \[ \log_b a = \log_x a / \log_x b \]

- **properties of exponentials:**
  \[ a^{(b+c)} = a^b a^c \]
  \[ a^{bc} = (a^b)^c \]
  \[ a^b / a^c = a^{(b-c)} \]
  \[ b = a^{\log_a b} \]
  \[ b^c = a^{c \cdot \log_a b} \]
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 $n \geq n_0$

- **little-o**
  - $f(n)$ is $o(g(n))$ if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ 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 $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n)$ 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 $g(n)$

**little-o**
- $f(n)$ 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 $g(n)$
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 $n \geq n_0$

  [TEAMS] What are values for $c$ and $n_0$?
  
  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 $\omega(n)$**
  
  $f(n)$ is $\omega(g(n))$ if, for any constant $c > 0$, there is an integer constant $n_0 \geq 0$ such that $f(n) \geq c \cdot g(n)$ for $n \geq n_0$

  Need $5n_0^2 \geq c \cdot n_0 \rightarrow$ given $c$, the $n_0$ that satisfies this is $n_0 \geq c/5 \geq 0$
Euclid’s Algorithm

An algorithm for computing the greatest common divisor (GCD) of two numbers $M \geq N$:

Algorithm GCD($M$, $N$)

while ($N \neq 0$)

rem $\leftarrow M \mod N$

$M \leftarrow N$

$N \leftarrow$ rem

endwhile

return $M$

[TEAMS] What is the Big-Oh?
Euclid’s Algorithm

An algorithm for computing the greatest common divisor (GCD) of two numbers $M \geq N$:

Algorithm GCD(M, N)

while $(N \neq 0)$

rem ← $M \mod N$

$M ← N$

$N ← \text{rem}$

endwhile

return $M$

The algorithm GCD runs is $O(\log n)$
Euclid’s Algorithm

Why?
- If \( M \geq N \), then \((M \mod N) < M/2\)
- Thus, each iteration at least halves the value of \( M \)
Exponentiation

A recursive algorithm to compute $X$ to the power $N$:

```
Algorithm pow(X, N)
    if (N == 0) return 1
    if (N == 1) return X
    if (N is even)
        return pow(X*X, N/2)
    else
        return pow(X*X, N/2) * X
```

[TEAMS] What is the Big-Oh?
Exponentiation

A recursive algorithm to compute $X$ to the power $N$:

```python
Algorithm pow(X, N)
    if (N == 0) return 1
    if (N == 1) return X
    if (N is even)
        return pow(X*X, N/2)
    else
        return pow(X*X, N/2) * X
```

The algorithm $pow$ is $O(\log n)$