### What’s an Algorithm?
- Computer Science is about problem-solving using computers.
- Software is a solution to some problems.
- Algorithm is a recipe/design inside a software.
- Informally, an algorithm is a method for solving a well-specified computational problem.
- Algorithms become more and more important in digital age.

### Experimental Studies
- Write a program implementing the algorithm
- Run the program with inputs of varying size and composition, noting the time needed:
- 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.
- Easy map to primitive operations of CPU.

Algorithm `arrymax(A, n)`:

```plaintext
Input: An array \( A \) storing \( n \geq 1 \) integers.
Output: The maximum element in \( A \).
for \( i = 1 \) to \( n \) do
  if currentMax < \( A[i] \) then
    currentMax = \( A[i] \);
return currentMax.
```

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:
- Arithmetic operations
- Assigning a value to a variable
- Indexing into an array
- Calling a method
- Returning from a method

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.
Functions Graphed Using “Normal” Scale

- \( g(n) = 1 \)
- \( g(n) = n \log n \)
- \( g(n) = 2^n \)
- \( g(n) = n \)
- \( g(n) = n^2 \)
- \( g(n) = n^3 \)

Running Time
- The running time of an algorithm typically grows with the input size.
- Average case time is often difficult to determine.
- We focus primarily on the worst case running time.
  - Easier to analyze
  - Crucial to applications such as games, finance and robotics

Counting Primitive Operations
- Example: By inspecting the pseudocode, we can determine the minimum and maximum number of primitive operations executed by an algorithm, as a function of the input size.

```
Algorithm arrayMax(A, n):
    Input: An array A storing n ≥ 1 integers.
    Output: The maximum element in A.
    currentMax ← A[0]
    for i = 1 to n − 1 do
        if currentMax < A[i] then
            currentMax ← A[i]
    return currentMax
```

Minimum: \( 2 + 3(n-1) + 2(n-1) + 1 = 5n \)
Maximum: \( 2 + 3(n-1) + 4(n-1) + 1 = 7n - 2 \)

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

Estimating Running Time
- Algorithm arrayMax executes \( 7n - 2 \) primitive operations in the worst case, \( 5n \) in the best 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
  \[ a(5n) \leq T(n) \leq b(7n - 2) \]
- Hence, the running time \( T(n) \) is bounded by two linear 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>( cn + 1 )</td>
<td>( 2cn )</td>
<td>( 4cn )</td>
</tr>
<tr>
<td>( cn\lg n )</td>
<td>( - cn\lg n + cn )</td>
<td>( 2cn\lg n + 2cn )</td>
<td>( 4cn\lg n + 4cn )</td>
</tr>
<tr>
<td>( cn^2 )</td>
<td>( - cn^2 + 2cn )</td>
<td>( 4cn^2 )</td>
<td>( 16cn^2 )</td>
</tr>
<tr>
<td>( cn^3 )</td>
<td>( - cn^3 + 3cn^2 )</td>
<td>( 8cn^3 )</td>
<td>( 64cn^3 )</td>
</tr>
<tr>
<td>( c2^n )</td>
<td>( c2^{n+1} )</td>
<td>( c2^{2n} )</td>
<td>( c2^{3n} )</td>
</tr>
</tbody>
</table>
Analyzing Recursive Algorithms

- Use a function, $T(n)$, to derive a **recurrence relation** that characterizes the running time of the algorithm in terms of smaller values of $n$.

```plaintext
Algorithm recursiveMax(A, n):
    Input: An array A storing $n \geq 1$ integers.
    Output: The maximum element in A.
    if $n = 1$ then
        return $A[0]$
    else
        return $\max \{ \text{recursiveMax}(A, n-1), A[n-1] \}$

$T(n) = \begin{cases} 
    3 & \text{if } n = 1 \\
    T(n-1) + 7 & \text{otherwise},
\end{cases}$
```

Constant Factors

- The growth rate is minimally affected by:
  - constant factors or
  - lower-order terms

Examples

- $10^n + 10^2$ is a linear function
- $10^n + 10^2n$ 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 $n \geq n_0$.
- We also say $g(n)$ is an asymptotic upper bound for $f(n)$.

Example: $2n + 10$ is $O(n)$

$$2n + 10 \leq cn$$

- $c = 2$ and $n \geq 10$
- $a = 10$ and $e = 2$

Big-Oh

- $f(n)$ is $\Omega(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \geq cg(n)$ for $n \geq n_0$.

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 $f(n) \geq cg(n)$ 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 $c'g(n) \leq f(n) \leq c''g(n)$ for $n \geq n_0$.
- Theorem: $\Theta$ is an equivalence relation.

Intuition for Asymptotic Notation

- **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

- $S_n$ 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 $f(n) \geq cg(n)$ for $n \geq n_0$.
  - Let $c = 5$ and $n_0 = 1$
- $S_n$ is $\Theta(n)$
  - $f(n)$ is $\Theta(g(n))$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq cg(n)$ for $n \geq n_0$.
  - Let $c = 1$ and $n_0 = 1$
- $S_n$ is $O(\log n)$
  - $f(n)$ is $O(\log n)$ if it is $O(n^c)$ and $O(n^{c'})$. We have already seen the former, for the latter recall that $f(n)$ is $\log n$ if there is a constant $c > 0$ and an integer constant $n_0 \geq 1$ such that $f(n) \leq c \log n$ for $n \geq n_0$.
  - Let $c = 5$ and $n_0 = 1$
Big-Oh, Big-Theta, Big Omega 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 \( O(n^2) \)”
  - Say “\( 3n + 5 \) is \( O(n) \)” instead of “\( 3n + 5 \) is \( O(n^2) \)”

Little omega

\( f(n) \) grows faster than \( g(n) \) (or \( g(n) \) grows slower than \( f(n) \)) if

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = 0, \]

Notation: \( f(n) = o(g(n)) \)
pronounced "little oh"

\[ \Theta(n^3): \quad n^3 \]
\[ 5n^3 + 4n \]
\[ 105n^3 + 4n^2 + 6n \]

\[ \Theta(n^2): \quad n^2 \]
\[ 5n^2 + 4n + 6 \]
\[ n^2 + 5 \]

\[ \Theta(\log n): \quad \log n \]
\[ \log n^2 \]
\[ \log (n + n^3) \]

A Case Study in Algorithm Analysis

- Given an array of \( n \) integers, find the subarray, \( A_{j \ldots k} \) that maximizes the sum

\[ \sum_{j \leq i \leq k} a_i = a_j + a_{j+1} + \cdots + a_k = \sum a_i \]

- In addition to being an interview question for testing the thinking skills of job candidates, this maximum subarray problem also has applications in pattern analysis in digitized images.
A First (Slow) Solution

Compute the maximum of every possible subarray summation $A[j,k]$ of the array $A$ separately.

- The outer loop, for index $j$, will iterate $n$ times, its middle-inner loop, for index $k$, will iterate $j \sim n$ times, and the inner-most loop, for index $i$, will iterate $j \sim k$ times.
- Thus, the running time of the MaxSubSlow algorithm is $O(n^3)$.

An Improved Algorithm

- A more efficient way to calculate these summations is to consider prefix sums $S_t = a_1 + a_2 + \ldots + a_t = \sum_{i=1}^{t} a_i$.
- If we are given all such prefix sums (and assuming $S_0 = 0$), we can compute any summation $s_{j,k} = S_k - S_{j-1}$ in constant time as

Example:

<table>
<thead>
<tr>
<th>$i$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>-2</td>
<td>-4</td>
<td>3</td>
<td>-1</td>
<td>5</td>
<td>6</td>
<td>-7</td>
<td>-2</td>
<td>4</td>
<td>-3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>$S$</td>
<td>0</td>
<td>-2</td>
<td>-6</td>
<td>-3</td>
<td>4</td>
<td>1</td>
<td>7</td>
<td>0</td>
<td>-2</td>
<td>-1</td>
<td>-1</td>
<td></td>
</tr>
<tr>
<td>Max</td>
<td>$S_9 - S_2 = 7 - (-6) = 13$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

An Improved Algorithm, cont.

- Compute all the prefix sums $\rightarrow O(n)$
- Then compute all the subarray sums $\rightarrow O(n^2)$

Algorithm MaxSubSlow($A$):
- Input: An $n$-element array $A$ of numbers, indexed from 1 to $n$.  
- Output: The maximum subarray sum of array $A$.
- $S_t = 0$ // the initial prefix sum
- for $i = 1$ to $n$ do
  - $S_k = a_i$ // the maximum found so far
  - $m = 0$ // the maximum
- for $k = 1$ to $n$ do
  - $m = a_k$ + $M_{k-1}$ // the maximum found so far
  - if $m > 0$ then
    - $M_k = m$
  - else
    - $M_k = 0$
- return $M_{n}$

A Linear-Time Algorithm

- Instead of computing prefix sum $S_t = s_{1,t}$, let us compute a maximum suffix sum $M_t$, which is the maximum of any subarray (including the empty one) ending at $t$.
  - $M_t = \max\{0, \max_{j=1,\ldots,t} (s_{j,t})\}$
  - If $M_t > 0$, then it is the summation value for a maximum subarray that ends at $t$, and if $M_t = 0$, then we can safely ignore any subarray that ends at $t$.
  - If we know all the $M_t$ values, for $t = 1, 2, ..., n$, then the solution to the maximum subarray problem would simply be the maximum of all these values.

An Improved Algorithm, cont.

- Compute all the prefix sums $\rightarrow O(n)$
- Then compute all the subarray sums $\rightarrow O(n^2)$

Algorithm MaxSubFastest($A$):
- Input: An $n$-element array $A$ of numbers, indexed from 1 to $n$.  
- Output: The maximum subarray sum of array $A$.
- $M_0 = 0$ // the initial prefix maximum
- for $t = 1$ to $n$ do
  - $M_t = \max\{0, M_{t-1} + A[t]\}$
- $m = 0$ // the maximum found so far
- for $t = 1$ to $n$ do
  - $m = \max(m, M_t)$
- return $m$
Math you need to Review

- **Summations**
  - Properties of powers:
    \[ a^{b+c} = a^b a^c \]
    \[ (ab)^c = a^c b^c \]
    \[ \frac{a^b}{a^c} = a^{b-c} \]
    \[ b^c = a^{\log_a b} \]

- **Powers**
  - Properties of logarithms:
    \[ \log_b(xy) = \log_b x + \log_b y \]
    \[ \log_b \left( \frac{x}{y} \right) = \log_b x - \log_b y \]
    \[ \log_b x^a = a \log_b x \]
    \[ \frac{\log_a b}{\log_b a} = \frac{\log_a x}{\log_a b} \]

- **Logarithms**
  - Summations
    \[ \sum_{i=1}^{n} 1/i = O(\ln n) \]
    using Integral of 1/x.
  - Powers
    \[ \sum_{i=1}^{n} i = O(n \log n) \]
    using Stirling’s approximation
    \[ n! \approx \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \]

- **Proof techniques**

- **Basic probability**

---

Logarithms

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

- Summations
  \[ \sum_{i=1}^{n} i = O(n \log n) \]
  using integral of 1/x.
  \[ \sum_{i=1}^{n} i^2 = O(n^3) \]

The Factorial Function

- Definition:
  \[ n! = 1 \cdot 2 \cdot 3 \cdots (n-1) \cdot n \]

- Stirling’s approximation:
  \[ n! \approx \sqrt{2\pi n} \left( \frac{n}{e} \right)^n \]
  or \[ \log(n!) = O(n \log n) \]

- How to approve it?

Bounds of Factorial Function

- Let \[ \log n! = \sum_{x=1}^{n} \log x, \]
  then \[ \int \log x \, dx \leq \sum_{x=1}^{n} \log x \leq \int (\log x + 1) \, dx \]
  which gives \[ n \log \left( \frac{n}{e} \right) + 1 \leq n! \leq (n+1) \log \left( \frac{n+1}{e} \right) + 1. \]

- So \[ e \left( \frac{n}{e} \right)^n \leq n! \leq e \left( \frac{n+1}{e} \right)^{n+1}. \]
  \[ n! \approx \sqrt{2\pi n} \left( \frac{n}{e} \right)^n. \]
Average Case Analysis

- In worst case analysis of time complexity we select the maximum cost among all possible inputs of size n.
- In average case analysis, the running time is taken to be the average time over all inputs of size n.
  - Unfortunately, there are infinite inputs.
  - It is necessary to know the probabilities of all input occurrences.
  - The analysis is in many cases complex and lengthy.

What is the average case of executing “currentMax \(\leftarrow A[i]\)”?

Algorithm `arrayMax(A, n)`:

Input: an array A of size n ≥ 1 integers.
Output: The maximum element in A.

```
1. currentMax \(\leftarrow A[0]\)
2. for i \(\leftarrow 1\) to n - 1 do
   if currentMax \(\leftarrow A[i]\) then
5. return currentMax
```

Number of Assignments: the worst case is n. If numbers are randomly distributed, then the average case is \(1+1/2+1/3+1/4+\ldots+1/n = O(\log n)\).

This is because A[1] has only 1/i probability to be the max of A[1], A[2], ..., A[i], under the assumption that all numbers are randomly distributed.

Dynamic Array Description

- Let `add(e)` be the operation that adds element e at the end of the array.
- When the array is full, we replace the array with a larger one.
- But how large should the new array be?
  - Incremental strategy: increase the size by a constant c.
  - Doubling strategy: double the size.

Comparison of the Strategies

- We compare the incremental strategy and the doubling strategy by analyzing the total time \(T(n)\) needed to perform a series of \(n\) add operations.
- We assume that we start with an empty list represented by a growable array of size 1.
- We call amortized time of an add operation the average time taken by an add operation over the series of operations, i.e., \(T(n)/n\).

Incremental Strategy Analysis

- Over \(n\) add operations, we replace the array \(k = n/c\) times, where \(c\) is a constant.
- The total time \(T(n)\) of a series of \(n\) add operations is proportional to
  \[ n + c + 2c + 3c + 4c + \ldots + kc = n + c(1 + 2 + 3 + \ldots + k) = n + c(k(k + 1)/2)\]
- Since \(c\) is a constant, \(T(n)\) is \(O(n + k^2)\), i.e., \(O(n^2)\).
- Thus, the amortized time of an add operation is \(O(n)\).
Doubling Strategy Analysis:
The Aggregate Method

- We replace the array $k = \log_2 n$ times.
- The total time $T(n)$ of a series of $n$ push operations is proportional to $n + 1 + 2 + 4 + 8 + \ldots + 2^k = n + \sum_{i=1}^{k} 2^i = n + 2^k - 1$.
- $T(n)$ is $O(n)$.
- The amortized time of an add operation is $O(1)$.

Doubling Strategy Analysis:
The Accounting Method

We view the computer as a coin-operated appliance that requires one cyber-dollar for a constant amount of computing time.

For this example, we shall pay each add operation 3 cyber-dollars.

- Set a saving account with $s_0 = 0$ initially.
- The $i$th operation has a budget cost of $a_i = 3$, which is the amortized cost of each operation.
- The account value after the $i$th add operation is $s_i = s_{i-1} + a_i - c_i$.

Summary

- Worst-case complexity: given an upper bound at the worst case.
- Average complexity: Assume a probability distribution of all inputs, give the complexity under this distribution.
- Amortized complexity: Compute the worst case of the sum of a sequence of operations, and then divide it by the number of operations.