Algorithm, Data Structure, Program

Algorithm

- Well-defined, a *finite* step-by-step computational procedure that takes some values, or set of values, as input and produces some value, or set of values, as output.

• Procedure
  - a sequence of computational steps that transform the input into the output. *Finiteness* of the steps of a procedure is not guaranteed.

• Program
  - algorithm represented in a program language

• Data structure
  - data structure suited for input, output, and temporal data during computing.
  - Passive objects

Program = Algorithm + Data Structure
What is a Computable Problem

Computable (Solvable) problem...

There is an algorithm with finite program steps to solve the problem on a computer

Many different algorithms for a computable problem are available!

Analysis of algorithms is essential to design good algorithms

How can we evaluate algorithms?

Measure running time of a program

• 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

Theoretical Analysis (Cont’d)

Algorithm analysis needs an abstract machine as a measure for evaluation of algorithms

Abstract Machines:
- Very simple computer models for measurement of program execution steps and memory requirement
- Turing Machine
- Random Access Machine (RAM)
  - Program correctness
  - Computation complexity evaluation in terms of speed and memory
Abstract Machine Example: Turing Machine

Proposed by Alan M. Turing in 1936 (Pre-Electric-Computer Era)

- Processor
  - Finite State Machine
- Tape
  - Memory with unlimited cells
  - Blank or symbol
- Head
  - Head for tape reads and writes

Turing Machine can perform any computation that any general-purpose computer can, but has a quite different architecture from conventional computers.

Pseudo-code

- 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 ← A[0]
for i ← 1 to n - 1 do
  if A[i] > currentMax then
    currentMax ← A[i]
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 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

Von-Neumann Computer Model:
Base Model for Modern Computer Systems
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
- 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
Analysis of Algorithm Complexity

- **Execution Steps**
  - Number of primitive steps/operations to solve a given problem.

- **Memory Requirement**
  - Total amount of memory required to hold inputs, outputs and temporal results

- Example: Search a desired datum among N data
  - Case 1
    - A data set of N elements is randomly stored.
  - Case 2
    - A data set of N elements is stored in an ascending/descending order.

---

Search Algorithms

- **Case 1 Random stored data**
  - Sequential search
    - First element
    - ... (represented by a dotted line)
  - Binary search
    - Comparison process (if smaller, if larger)

- **Case 2 Sorted data**
  - Binary search
    - Comparison process (if smaller, if larger)
## Summary of Search Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Insert new data</th>
<th>Delete data</th>
<th>Initial data arrangement for N times</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequential search</td>
<td>Best 1, Worst N,</td>
<td></td>
<td>1</td>
<td>N</td>
</tr>
<tr>
<td></td>
<td>Average N/2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>searches</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary search</td>
<td>log N</td>
<td>log N</td>
<td>log N searches, N/2 moves</td>
<td>N log N</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### N vs. Log N: As a Function of Input Data

- **Very slow growth**
- Algorithms with log N steps are very good ones!
Metrics for Complexity Analysis

- Many programs are extremely sensitive to their input data
  - Algorithms are evaluated as a function of the number of input data.

- Two Metrics for Algorithm Analysis
  - **Space Complexity**
    - Total amount of memory to store inputs, outputs, and temporal data
  - **Time Complexity**
    - Number of (program) steps to solve the problem

So, What's a Bad Algorithm... yes, it is computable, but it needs
- the enormous amount of memory
- the enormous number of execution steps

Complexity in Worst Case and Average Case

- **Worst Case Complexity**
  - The worst-case execution complexity of an algorithm gives an upper bound on the execution steps for any input.
  - Evaluate the algorithm using the worst possible input configuration.
    - Guarantee that the algorithm will never take any longer.
    - For some algorithms, the worst case rarely occurs. In such a case, the worst case complexity does not reflect the actual program behavior.

- **Average Complexity (Expected Complexity)**
  - Estimate the complexity of an algorithm with considering input occurrence probability. Evaluate the algorithm using typical input data.
    - Reflects the actual behavior of an algorithm, but hard to constitute average inputs for a particular problem.
    - Assume that all inputs of a given size are equally likely
Estimating Running Time

- Assume that an algorithm executes $7n-1$ 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 the algorithm. Then
  - $a(7n-1) \leq T(n) \leq b(7n-1)$
- 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 the algorithm.
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.

Constant Factors

- The growth rate is not affected by:
  - constant factors or
  - lower-order terms

- Examples
  - $10^5n + 10^5$ is a linear function
  - $10^5n^2 + 10^5n$ is a quadratic function
Big O-notation: Quantitative and Approximate Analysis of Algorithm Complexity

- **Big O-notation** gives an upper bound for a function to within a constant factor.
  - If there exist positive constants \( n_0 \) and \( c \) such that to the right of \( n_0 \), the value of \( f(n) \) always lies on or below \( cg(n) \), i.e.,

  \[
  f(n) \leq cg(n) \quad n: \text{number of inputs}
  \]

  The complexity of a given algorithm:
  \( O(g(n)) \)

Analysis of Algorithm Complexity using Big O-Notation

Example: An algorithm with \( 2n^2 \) execution steps is roughly evaluated with a complexity of \( O(n^2) \) steps

- **O-notation** gives approximate evaluation of algorithms… ignore details!!

- Just consider the **leading/dominant term** of mathematical expression of algorithm complexity

  \( n^2, 2.5n^2, 100n^2, 1000+5n+2n^2... \) \( \rightarrow O(n^2) \) !!
Big-O Example

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

- \(2n + 10 \leq cn\)
- \((c - 2) n \geq 10\)
- \(n \geq \frac{10}{c - 2}\)

- Pick \(c = 3\) and \(n_0 = 10\)

Example:

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-O Examples

- $7n^2$ is $O(n)$
  need $c > 0$ and $n_0 \geq 1$ such that $7n^2 \leq c \cdot n$ for $n \geq n_0$
  this is true for $c = 7$ and $n_0 = 1$

- $3n^3 + 20n^2 + 5$ 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 $n_0 = 21$

- $3 \log n + \log \log n$ is $O(\log n)$
  need $c > 0$ and $n_0 \geq 1$ such that $3 \log n + \log \log n \leq c \cdot \log n$ for $n \geq n_0$
  this is true for $c = 4$ 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>$f(n)$ grows more</th>
<th>$g(n)$ is $O(f(n))$</th>
<th>$f(n)$ is $O(g(n))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Same growth</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Big-Oh Rules

• If \( f(n) \) a polynomial of degree \( d \), then \( f(n) = O(n^d) \), 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 the algorithm executes at most \( 7n - 1 \) primitive operations
  - We say that the algorithm “runs in \( O(n) \) time”
• Since constant factors and lower-order terms are eventually dropped anyhow, we can disregard them when counting primitive operations
Good Algorithm or Bad Algorithm?: Quantitative Definition

Good Algorithms...
- Time/Space Complexity is $O(n)$, $O(\log n)$, $O(n\log n)$, $O(n^2)$...
- Polynomial Growth of Computation Steps and/or memory requirement as the number of inputs increases

Bad Algorithms...
- Time/Space Complexity is $O(k^n)$ where $k$ is a constant.
- Exponential Growth of Computation Steps and/or Memory Requirement as the number of inputs increases

If algorithms with exponential growth of complexity are only solutions to a problem, the problem is called a hard problem or intractable problem.

Polynomial-time solvable problems are called easy problems or tractable problems.

Growth of Functions

<table>
<thead>
<tr>
<th>n: number of inputs</th>
<th>Unit: seconds if not specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>$O(1)$</td>
<td></td>
</tr>
<tr>
<td>$O(\log n)$</td>
<td></td>
</tr>
<tr>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>$O(n\log n)$</td>
<td></td>
</tr>
<tr>
<td>$O(n^2)$</td>
<td></td>
</tr>
<tr>
<td>$O(n^3)$</td>
<td></td>
</tr>
<tr>
<td>$O(2^n)$</td>
<td></td>
</tr>
</tbody>
</table>

n: number of inputs
Unit: seconds if not specified

Using more powerful computers is a good solution to hard problems?
Computer Power Goes Up Steadily, and...

The Number of Transistors Per Chip Doubles Every 18 Months!

Source: Intel

Yes, We Have Powerful Computers at Low Cost, but...

<table>
<thead>
<tr>
<th>Year</th>
<th>Computer</th>
<th>Performance Ratio</th>
<th>Memory Ratio</th>
<th>Price</th>
<th>Price/Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1965</td>
<td>IBM System 360/50</td>
<td>0.15MIPS(1)</td>
<td>64KB(1)</td>
<td>360M yen</td>
<td>1</td>
</tr>
<tr>
<td>1977</td>
<td>DEC VAX11/780</td>
<td>1MIPS(6.7)</td>
<td>1MB(16)</td>
<td>72M yen</td>
<td>33 times</td>
</tr>
<tr>
<td>2001</td>
<td>Pentium 4 PC</td>
<td>3792MIPS(25,280)</td>
<td>256MB(4,096)</td>
<td>250K yen</td>
<td>More than 36M times!</td>
</tr>
</tbody>
</table>

Performance (ratio) 0.15MIPS(1) 1MIPS(6.7) 3792MIPS(25,280)
Memory (ratio) 64KB(1) 1MB(16) 256MB(4,096)
Price 360M yen 72M yen 250K yen
Price/performance 1 33 times More than 36M times!

Source: Various
It's a mere drop in the bucket for hard problems...

Example

• For 4 algorithms $A_1, A_2, A_3, A_4$ whose complexities are $O(n)$, $O(n^2)$, $O(n^{100})$, $O(2^n)$, respectively, evaluate increases in input size when an 100 times faster computer is available.

Assumptions
• A new computer M’ is 100 times faster than an old computer M.
• $m_x$ is a data size/sec on M with algorithm $x$.
• $m'_x$ is a data size/sec on M’ with algorithm $x$.

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>Time Complexity</th>
<th>Increase in Input Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_1$</td>
<td>$O(n)$</td>
<td>$m_1$ 100$m_1$</td>
</tr>
<tr>
<td>$A_2$</td>
<td>$O(n^2)$</td>
<td>$m_2$ 10$m_2$</td>
</tr>
<tr>
<td>$A_3$</td>
<td>$O(n^{100})$</td>
<td>$m_3$ 1.047$m_3$</td>
</tr>
<tr>
<td>$A_4$</td>
<td>$O(2^n)$</td>
<td>$m'_4$ $m'_4 + 6.644$</td>
</tr>
</tbody>
</table>

Algorithm Analysis Example: MSP Problem

Minimum Spanning Tree Problem
• For a given graph $G(V, E)$, find an acyclic subset $T \subseteq E$ that connects all of the vertices and whose total weight is minimized, where $V$ is the set of vertices and $E$ is the set of edges.

Weighed Graph → Minimum Spanning Tree
Algorithm Theory

MST Algorithm (Kruskal’s Algorithm)

Step 1
• Sort edges in an ascending order

Step 2
• Prepare $v$ sets, each of which has each vertex as an entry, where $v$ is the number of vertices.

Step 3
• Pick up an edge with the smallest weight.
• If two vertices of the edge belong to the different sets, the edge is the edge of the spanning tree, and merge the two sets to which these two vertices belong.
• Otherwise, discard the edge, and check the next smallest edge.
• This step is repeated until all the edges have gone.

Algorithm Behavior

1st iteration
2nd iteration
3rd iteration
4th iteration
5th iteration

Initial Sets 1st iteration 2nd iteration

merge

merge

merge

merge

Final MST

weighted graph
Complexity Analysis: MST Problem

\[ m = |E|, \ n = |V| \text{ where } n < m \leq n(n-1)/2 \]

- Computational complexity for sorting
  - Sort \( m \) elements \( m \log m \)
- Computational complexity for set operations
  - One set operation for \( n \) elements \( \log n \)
  - (Find vertices connected by a given edge and merge two sets)
    \( m \) iterations of the set operation \( m \log n \)

- Total Computation Complexity
  \( O(m \log m) + O(m \log n) \Rightarrow O(m \log m) \)
- Space Complexity
  - Memory for graph \( O(m+n) \Rightarrow O(m) \),
  - Working memory for set operations \( O(n) \),
  - Total \( O(m) + O(n) \Rightarrow O(m) \)

Complexity Analysis: Optimization Problem

- For \( n+1 \) natural number \( (a_1, a_2, a_3, \ldots, a_n, b) \), if there is a 0-1 vector \( x=(x_1, x_2, \ldots, x_n) \) that satisfies \( \Sigma a_i x_i = b \), output \textit{yes}, otherwise \textit{no}.
  - Example: \( x=(3, 7, 5, 8, 2) \), \( b=11 \)

- Algorithm
  - Represent \( x \) in \( n+1 \)-bit number, and check throughout all the 0-1 combinations, e.g., from \( x=(000\ldots0) \) to \( (111\ldots1) \). If there is a combination that satisfies the condition, output \textit{yes}, otherwise \textit{no}. 
Complexity Analysis: Optimization Problem

- Number of iterations to check throughout 000...0 to 111...1
  - $2^{n+1} \Rightarrow O(2^n)$
- Operation within one iteration: calculations for weighted sum and condition evaluation
  - $O(n)$

- Total computation complexity
  - $O(2^n) \times O(n) \Rightarrow O(n \cdot 2^n)$
  - Hard Problem!

- Space complexity
  - $O(n)$ for storing each of vectors $a$ and $x$
  - Total Space Complexity $O(n)$

Possible Solutions to Hard Problems

- Translate the hard problems to easy problems for which algorithms generating quasi-optimal solutions/output with polynomial complexity are available.
- Develop heuristic approaches that generate quasi-optimal solutions/output
  - Neural Network
  - Genetic algorithm
- Develop parallel algorithms on parallel computers (brute-force!)
Training an artificial neural network with pairs of inputs and their correct outputs, instead of programming.

**Problem class where there is no good deterministic algorithm!!**
Trend in Supercomputer Performance

Single-CPU Performance, or Massive Parallelism? (As of Nov. 02)
Hard Problems are also Useful...
Example: Secure Networks with Cryptography

Sender
Plain text → key
Encrypted text

Receiver
Plain text ← key
Encrypted text

Encrypted Data Transmission

Decryption without an encryption key leads to exponential computation complexity.

Intrusion by someone else is very hard!

Summary

- **Complexity analysis of algorithms gives a clue to judge whether the algorithms are efficient or not.**
  - Time Complexity
  - Space Complexity

- **Analysis of an algorithm is a cycle process of**
  - Analyzing
  - Estimating
  - Refining the analysis until an answer to the desired level of detail is reached

- **Design efficient algorithms with polynomial-time/space complexity of** $O(n)$, $O(\log n)$, $O(n \log n)$, or $O(n^2)$!