GIUSEPPE TURINI

CS-102: COMPUTING AND ALGORITHMS 2
LESSON 09

CS-203: COMPUTING AND ALGORITHMS 3
LESSON 01

ALGORITHM DESIGN AND ANALYSIS

Kettering UNIVERSITY

HIGHLIGHTS

Algorithmics

- **Algorithm Problem Solving:** The Problem, HW Capabilities, Exact / Approx
  - Design, Correctness, Analysis, Coding, Properties
- **Important Problems:** Sorting, Searching, String Processing
  - Graph Problems, Combinatorial, Geometric, and Numerical Problems
- **Important Data Structures:** Linear, Graphs, Trees, Sets, and Dictionaries

Efficiency Analysis

- **Analysis Framework:** Input Size, Running Time, and Orders of Growth
  - Worst / Best / Average-case Efficiencies
- **Asymptotic Notations:** Big O / Big Omega / Big Theta Notations
  - Properties, Orders of Growth and Limits, Basic Efficiency Classes
- Non-recursive / Recursive Analysis, Empirical Analysis, Algorithm Visualization
- **Data Structures Efficiency:** Linear, Graphs, Trees, Sets, and Dictionaries

STUDY GUIDE

STUDY MATERIAL

- **This slides.**

SELECTED EXERCISES

- **Set 1:** ex. 1.1.4-6, 1.1.8-9, 1.2.1-2, 1.2.4-5, 1.2.9, 1.3.1, 1.4.2-4, 1.4.9-10.
- **Set 2:** ex. 2.1.1-5, 2.1.8-10, 2.2.1-7, 2.2.9-12, 2.3.1-6, 2.3.9, 2.3.11-12, 2.4.1-5, 2.4.7-12, 2.5.4, 2.5.6-9, 2.6.1-4.

ADDITIONAL RESOURCES

- **"Introduction to the Design and Analysis of Algorithms (3rd Ed.)", chap. 1-2.**
- "Data Abstraction and Problem Solving with Java (3rd Ed.)", chap. 10.
- visualgo.net/en

ALGORITHMICS - INTRODUCTION

Algorithmics is the study of algorithms, and algorithm design techniques can be seen as problem-solving strategies even when no computer is involved.

*"... it has often been said that a person does not really understand something until after teaching it to someone else. Actually, a person does not **really** understand something until after teaching it to a **computer** ..."*
Donald E. Knuth. *Selected Papers on Computer Science*. 1996.

Why study algorithms?
Because: algorithms are the core of computer science, to provide a toolkit of known algorithms, and a framework to design/analyze algorithms.

The main issues related to algorithms are: how to perform the **algorithm design**, and the **efficiency analysis** of algorithms.

ALGORITHMICS - ALGORITHM DEFINITION

Algorithm: A sequence of non-ambiguous instructions to solve a problem.

Most algorithms are intended for computer implementation, but the notion of algorithm does not depend on such an assumption!

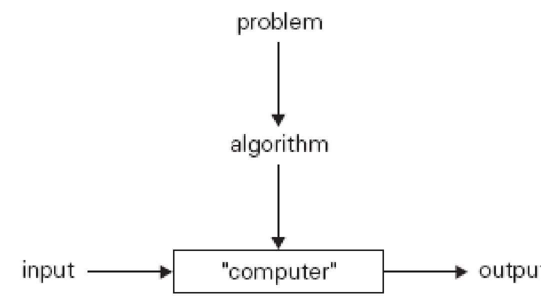

FIGURE 1.1 The notion of the algorithm.

In defining an algorithm, it is important to understand these important points:

- The **non-ambiguity** of each algorithm step cannot be compromised.
- The **range of inputs** for which an algorithm works has to be specified carefully.
- The same algorithm can be **represented in several different ways**.
- Algorithms for the **same problem can have different designs/performances**.

ALGORITHMICS - ALGORITHM EXAMPLE 1

METHODS FOR gcd(m,n): EUCLID'S ALGORITHM

Problem: Find gcd(m, n), the greatest common divisor of two non-negative, not both zero integers m and n.

Examples: gcd(60, 24) = 12, or gcd(60, 0) = 60, or gcd(0, 0) = ?

Euclid's algorithm is based on repeated application of equality

$$gcd(m, n) = \mathrm{gcd}(n, m\ mod\ n)$$

until the second number becomes 0, which makes the problem trivial.

Example:

$$\begin{aligned} gcd(60, 24) &= \mathrm{gcd}(24, 60\ mod\ 24) = \mathrm{gcd}(24, 12) = \\ &= \mathrm{gcd}(12, 24\ mod\ 12) = \mathrm{gcd}(12, 0) = 12 \end{aligned}$$

ALGORITHMICS - ALGORITHM EXAMPLE 2

METHODS FOR gcd(m,n): EUCLID'S ALGORITHM (IMPLEMENTATION 1)

Step 1: If n = 0, return m and stop; otherwise go to Step 2.
Step 2: Divide m by n and assign the value of the remainder to r.
Step 3: Assign the value of n to m and the value of r to n. Go to Step 1.

```
while( n != 0 ) {
   r = m mod n;
   m = n;
   n = r; }
return m;
```

ALGORITHMICS - ALGORITHM EXAMPLE 3

METHODS FOR gcd(m,n): EUCLID'S ALGORITHM (IMPLEMENTATION 2)

Step 1: Assign the value of min(m, n) to t.
Step 2: Divide m by t. If the remainder is 0, go to Step 3; otherwise, go to Step 4.
Step 3: Divide n by t. If the remainder is 0, return t and stop; otherwise, go to Step 4.
Step 4: Decrease t by 1 and go to Step 2.

```
t = min( m, n );
while( true ) {
   if( m mod t == 0 ) {
      if( n mod t == 0 ) {
         return t; } }
   t--; }
```

ALGORITHMICS - ALGORITHM EXAMPLE 4

METHODS FOR gcd(m,n): MIDDLE-SCHOOL PROCEDURE

Step 1: Find the prime factorization of m.
Step 2: Find the prime factorization of n.
Step 3: Find all the common prime factors.
Step 4: Compute product of all common prime factors, and return it as gcd(m, n).

Question: **Is this an algorithm?**

Answer: In this form, the middle-school procedure does not qualify as a legitimate algorithm because the prime factorization steps are not defined unambiguously (e.g. they require a list of prime numbers).

ALGORITHMICS - DESIGN-ANALYSIS PROCESS

Algorithms are **procedural solutions to problems**.
These solutions are **instructions to solve problems**.

This is the sequence of steps to design and analyze an algorithm:

1. **understand the problem;**
2. **check computing device capabilities;**
3. **choose exact vs approximate solving;**
4. **choose algorithm design technique;**
5. **design algorithm and data structures;**
6. **prove algorithm correctness;**
7. **analyze the algorithm;**
8. **code the algorithm.**

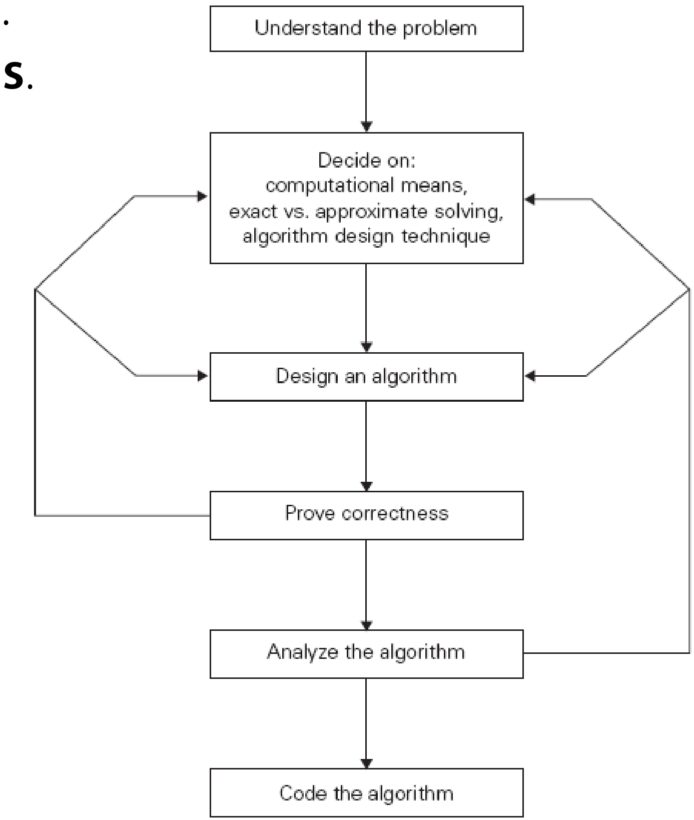

FIGURE 1.2 Algorithm design and analysis process.

ALGORITHMICS - UNDERSTAND THE PROBLEM

Understand the problem: is the 1st step in design-analysis of algorithms.

- Read the problem **description**.
- Solve a few **examples** by hand.
- Consider about **special cases**.

Problem instance: An input to an algorithm specifies an instance of the problem.

Legitimate inputs: The set of instances the algorithm can handle.

**A correct algorithm does not have to work all the time,
but it has to work correctly for all legitimate inputs.**

ALGORITHMICS - HW COMPUTING CAPABILITIES

If your target computing HW is a **random-access machine (RAM)**, executing instructions sequentially, 1-by-1, you need to design a **sequential algorithm**.

If your target computing HW is a **parallel computer**, able to execute multiple instructions at once, you need to design a **parallel algorithm**.

The study of techniques to design-analyze sequential algorithms is the current standard of algorithmics.

ALGORITHMICS - EXACT-APPROX SOLUTIONS

Exact algorithm: Is an algorithm that solves the problem exactly.

Approximation algorithm: Is an algorithm that solves the problem approximately.

Question: Why would one opt for an approximation algorithm?

Answer: Because:

- There are problems that cannot be solved exactly for most of their instances.
- Exact algorithms for a problem could be slower than approx algorithms.
- An approximation algorithm can be part of another algorithm that is exact.

ALGORITHMICS - ALGORITHM DESIGN TECHNIQUES

Algorithm design technique: It is a general approach to solve problems algorithmically applicable to a variety of problems from different areas of computing.
These techniques **provide guidance** for designing algorithms for new problems.
These techniques allow to **classify algorithms** using their underlying design idea.

The main algorithm design techniques are:

- brute-force and exhaustive search,
- divide-and-conquer, decrease-and-conquer, transform-and-conquer,
- space-and-time tradeoffs,
- greedy approach,
- dynamic programming,
- iterative improvement,
- backtracking,
- branch-and-bound, etc.

ALGORITHMICS - ALGORITHM DATA STRUCTURES

Choose data structures appropriate for the algorithm operations.

Example: Generate all prime integers not exceeding n ≥ 2 (Sieve of Eratosthenes)

```
for( p = 2; p <= n; p++ ) { A[p] = p; } // Init list A with integers from 2 to n.
for( p = 2; p <= floor( sqrt(n) ) ) {
   if( A[p] != 0 ) { // p has not been previously eliminated from list A.
      j = p * p; // Start checking from p^2.
      while( j <= n ) {
         A[j] = 0; // If less than n, mark current element as eliminated.
         j = j + p; } } } // Update current element to check next multiple value.
```

Question: It is better a linked list or an array to code the Sieve of Eratosthenes?

Answer: An array, because the access operation is faster on arrays than on linked lists.

ALGORITHMICS - ALGORITHM SPECIFICATIONS

Once you have designed an algorithm, you need to specify it in some fashion:

- **Natural language:** It is easy but its inherent ambiguity makes difficult to describe an algorithm in a succinct and and clear way.
- **Pseudocode:** It is a mix of a natural language and programming language constructs. It is usually more precise than natural language, and usually provides more succinct descriptions.
- **Flowchart:** It is a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the steps of the algorithm. It is convenient only for very simple algorithms.

ALGORITHMICS - PROVE CORRECTNESS

Once an algorithm has been designed, you have to prove its **correctness**: that the algorithm yields **correct results for all legitimate inputs in finite time**.

To prove that an algorithm is correct: You can use **mathematical induction** because the iterations of an algorithm are an natural (integer) sequence of steps.

To prove that an algorithm is incorrect: You need just 1 instance of its input for which the algorithm fails.

To prove that an approximation algorithms is correct: You need to demonstrate that the error produced does not exceed a predefined limit.

ALGORITHMICS - ALGORITHM ANALYSIS

We usually want our algorithms to possess several qualities:

- **Correctness:** Providing a correct result for every legitimate input in a finite time.
- **Time efficiency:** How fast the algorithm runs.
- **Space efficiency:** How much extra memory it uses.
- **Simplicity:** Easy to understand, to program, and to debug.
- **Generality of the problem:** Easier to design algorithms for general problems.
- **Generality of the input:** Handling a set of inputs that is natural for the problem.

"A designer knows he has arrived at perfection not when there is no longer anything to add, but when there is no longer anything to take away."

Antoine de Saint-Exupéry

ALGORITHMICS - ALGORITHM IMPLEMENTATION

Most algorithms will be ultimately implemented as computer programs.

Programming an algorithm presents both a peril and an opportunity:

- **The peril:** It lies in the possibility of making the transition from an algorithm to a program either incorrectly or very inefficiently.
- **The opportunity:** It is in allowing an empirical analysis of the algorithm.

ALGORITHMICS - ALGORITHM PROPERTIES

These are some common algorithm properties/types:

- **In-place:** These algorithms process the input using no auxiliary data structures (no extra memory excluding inputs and outputs).

- **Out-of-core:** These algorithms (aka external memory algorithms) are designed to process data that are too large to fit into the memory at once.

- **Stable:** These **sorting** algorithms preserve the order of records with equal keys (this property is defined only for sorting algorithms).

ALGORITHMICS - IMPORTANT PROBLEM TYPES

This is a brief list of the most important problem types:

- **Sorting:** Rearranging items in a list in non-decreasing order.
- **Searching:** Finding a value in a set.
- **String processing:** Searching a word in a text, etc.
- **Graph problems:** Finding shortest path between two nodes, etc.
- **Combinatorial problems:** Problems related to permutations, combinations, etc.
- **Geometric problems:** Solving geometrical problems (convex-hull, etc.).
- **Numerical problems:** Problems involving continuous mathematical objects.

These problems allow to show different design and analysis techniques.

ALGORITHMICS - DATA STRUCTURE TYPES

Algorithms work on data, so data organization is critical in design-analysis.

These are the main data structure types for computer algorithms:

- **Linear:** A data structure is linear if its data is arranged in a sequence (arrays etc.).
- **Graph:** A set of vertices/nodes connected in pairs by edges/links.
- **Tree:** A connected acyclic graph.
- **Set:** An unordered collection of distinct elements.
- **Dictionary:** A data structure implementing search, insertion, and removal by key.

ANALYSIS - INTRODUCTION

The **algorithm efficiency analysis** is an investigation in respect to 2 resources:

- **Time efficiency (complexity):** Investigation of the running time of an algorithm.
- **Space efficiency (complexity):** Investigation of the memory used.

This emphasis on efficiency is easy to explain.

- **First**, unlike simplicity and generality, efficiency can be measured.
- **Second**, the efficiency considerations are critical from a practical point of view.

ANALYSIS - INPUT SIZE

Most algorithms run longer on larger inputs.
Therefore, we investigate the
efficiency as a function of a parameter n indicating the input size.

Note: Some algorithms require multiple parameters to indicate their input size.

Note: Sometimes is the input "magnitude" that determines the input size. In such situations, it is better to use the bit-size of the input to represent the input size.

$$\text{"number of bits in the binary reprentation of n"} = b = \lfloor \log_2 n \rfloor + 1$$

ANALYSIS - RUNNING TIME UNITS 1

If we use **standard unit of time** (sec etc.) to measure the running time of an implementation of an algorithm, there are some **drawbacks**:

- Results will depend on the speed of a particular HW.
- Results will depend on the quality of the implementation (code, compiler etc.).
- It will be hard to measure the actual running time (time queries etc.).

**To measure of the efficiency of an algorithm,
we would like a metric that does not depend on these factors
(hardware and implementation).**

ANALYSIS - RUNNING TIME UNITS 2

A strategy is to count the number of times each algorithm operation is executed. However, this is both excessively difficult, and usually unnecessary.

The correct approach is:

1. To identify the most important operation of the algorithm (**basic operation**).
2. To compute the **number of times** the basic operation is executed.

As a rule, it is not difficult to identify the basic operation of an algorithm:
the basic operation is usually the most time-consuming operation in the innermost loop of the algorithm.

Example: For sorting algorithms usually the basic operation is the comparison.

ANALYSIS - RUNNING TIME UNITS 3

Example of input size and basic operation:

Problem	Input size measure	Basic operation
Search item in list	Number of items in list	Comparison
Multiply 2 matrices	Number of elements	Multiplication
Check prime number	Bit-size of number	Division
Graph problem	Number of vertices/edges	Visit/traverse vertex/edge

ANALYSIS - RUNNING TIME UNITS 4

So, the efficiency analysis of an algorithm is based on:
counting how many times the basic operation is executed on inputs of size n.

Example:

- $\mathbf{c_{op}}$ is the execution time of the algorithm basic operation on a particular HW.
- **C(n)** is the count of this basic operation for a particular implementation.
- We can estimate the running time **T(n)** of that implementation as:

$$T(n) \approx c_{op}\, C(n)$$

ANALYSIS - RUNNING TIME UNITS 5

$$T(n) \approx c_{op}\, C(n)$$

Of course this formula should be used with caution.

- The count **C(n)** does not include not-basic operations.
- The constant $\mathbf{c_{op}}$ is an approximation.

Still, unless the input size n is extremely large or very small,
the formula above gives a reasonable estimate of the algorithm running time.

ANALYSIS - RUNNING TIME UNITS 6

$$T(n) \approx c_{op}\, C(n)$$

This formula also makes it possible to answer questions as:

Question: How faster this algorithm runs on an HW that is 10 times faster?
Answer: 10 times.

Question: If C(n) = n(n-1)/2, how longer the algorithm runs if we double input size?
Answer: Approximately 4 times longer.

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2 \Longrightarrow \frac{T(2n)}{T(n)} \approx \frac{c_{op}\, C(2n)}{c_{op} C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4$$

ANALYSIS - RUNNING TIME UNITS 7

Question: If C(n) = n(n-1)/2, how longer the algorithm runs if we double input size?
Answer: Approximately 4 times longer.

$$C(n) = \frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \approx \frac{1}{2}n^2 \Longrightarrow \frac{T(2n)}{T(n)} \approx \frac{c_{op}\, C(2n)}{c_{op} C(n)} \approx \frac{\frac{1}{2}(2n)^2}{\frac{1}{2}n^2} = 4$$

Note: This answer is independent from the value of $\mathbf{c_{op}}$.

Note: This answer is independent from the multiplicative constant (**1/2**).

This is why the efficiency analysis ignores multiplicative constants and focuses on **the order of growth of C(n) within a constant multiple for large input sizes**.

ANALYSIS - ORDER OF GROWTH 1

Why focus on the order of growth of count C(n) for large input sizes?

- For small input sizes, the difference in running times is not significant to distinguish efficient/inefficient algorithms.
- For large input sizes, it is **the order of growth of the count C(n) that allows us to distinguish efficient/inefficient algorithms!**

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

ANALYSIS - ORDER OF GROWTH 2

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

This table is important for the efficiency analysis of algorithms:

- The function (count) in the table that is growing the slowest is the **logarithmic function $\log_2$ n**.
- The functions (counts) in the table that are growing really fast are the **exponential function 2^n**, and **factorial function n!**.

ANALYSIS - ORDER OF GROWTH 3

$$\log_a n = \log_a b \ \log_b n$$

Note: The formula above allows to change the logarithm base, leaving the count **C(n)** logarithmic (the difference is only a multiplicative constant). This is why:
we omit the logarithm base when we focus on the order of growth.

The **logarithmic function (log n)** grows so slow, that
algorithms with logarithmic counts run instantaneously on all input sizes.

The **exponential-growth functions (2^n and n!)**, while different, grow so fast that
algorithms with exponential counts take a lot of time even for small input sizes.

ANALYSIS - ORDER OF GROWTH 4

TABLE 2.1 Values (some approximate) of several functions important for analysis of algorithms

n	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10^1	$3.3 \cdot 10^1$	10^2	10^3	10^3	$3.6 \cdot 10^6$
10^2	6.6	10^2	$6.6 \cdot 10^2$	10^4	10^6	$1.3 \cdot 10^{30}$	$9.3 \cdot 10^{157}$
10^3	10	10^3	$1.0 \cdot 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

How these functions react to a ×2 increase in input size n?

- **Logarithmic function $\log_2 n$** increases by 1
- **Linear function n** increases twofold (×2).
- **Linearithmic function $n \log_2 n$** increases more than twofold (×2).
- **Quadratic function n^2** increases ×4.
- **Cubic function n^3** increases ×8.
- **Exponential function 2^n** gets squared.

ANALYSIS - EXAMPLE SEQUENTIAL SEARCH 1

The **sequential search** searches for a given item in a list of **n** items by checking successive items until either the given item is found or the list is exhausted.

```
int i = 0;
while( ( i < n ) && ( list[i] != k ) ) { i++; }
if( i < n ) { return i; }
else { return -1; }
```

The input size for this algorithm is the size of the list.
However, this algorithm could run differently for lists of the same size.

Example: Sequential search in a list with 10 items, the comparisons performed range from 1 (successful search at front) to 10 (failed search or successful search at back).

ANALYSIS - EXAMPLE SEQUENTIAL SEARCH 2

Example: Sequential search in a list with 10 items:

- In **worst case**, there are no matching items or the 1st matching item is the last in the list, so the sequential search makes the maximum number of comparisons:

$$\mathrm{C}_{\mathrm{worst}}(10) = 1 \text{ comparison for each list item} = 10$$

- In **best case**, the 1st matching item is the first in the list, so the sequential search makes the minimum number of comparisons:

$$\mathrm{C}_{\mathrm{best}}(10) = \text{"1 comparison with first list item"} = 1$$

- In **average case**, we have to compute the weighted average count considering all possible scenarios and their probability of happening. For example: successful searches with 80% probability, with equal probability on each item in the list:

$$\mathrm{C}_{\mathrm{avg}}(10) = [\ (1 \times 8\%) + (2 \times 8\%) + \cdots + (10 \times 8\%)\ ] + (10 \times 20\%) = 6.4$$

ANALYSIS - WORST-BEST-AVERAGE CASES

Some efficiencies depend not only on input size, but also on input details.

In these cases, we cannot describe the count C(n) as a function of the input size.
So, the analysis has to focus on specific types of inputs, in particular:

- The **best case** scenarios, when the algorithm runs the fastest, and $\mathbf{C_{best}\ (n)}$ counts the basic-operations performes in these scenarios with input size **n**.
- The **worst-case** scenarios, when the algorithms runs the slowest, and $\mathbf{C_{worst}(n)}$ counts the basic-operations performed in these scenarios with input size **n**.
- The **average-case**, when we consider the probability of all possible scenarios of input size n, classified using their count, and we average the results in: $\mathbf{C_{avg}\ (n)}$.

ANALYSIS - WORST-CASE EFFICIENCY

The **worst-case efficiency** of an algorithm is its efficiency for the worst-case input of size **n**, for which the algorithm runs the longest among all possible inputs of that size.

To determine the worst-case efficiency of an algorithm:

1. **Check the inputs** that yield the maximum count **C(n)** among all inputs of size **n**.

2. **Compute the count** in these worst-case scenarios as: $\mathbf{C_{worst}(n)}$.

The worst-case analysis guarantees that for any instance of size n, the running time will not exceed the worst-case count $C_{worst}(n)$.

ANALYSIS - BEST-CASE EFFICIENCY

The **best-case efficiency** of an algorithm is its efficiency for the best-case input of size **n**, for which the algorithm runs the fastest among all possible inputs of that size.

To determine the best-case efficiency of an algorithm:

1. **Check the inputs** that yield the minimum count **C(n)** among all inputs of size **n**.
2. **Compute the count** in these best-case scenarios as: $\mathbf{C_{best}(n)}$.

The best-case analysis is not important as the worst-case analysis, but it allows to discard an algorithm if its best-case efficiency is unsatisfactory.

ANALYSIS - AVERAGE-CASE EFFICIENCY 1

An algorithm **average-case efficiency** is its efficiency for the "typical " input of size **n**.

To determine the average-case efficiency of an algorithm:

1. **Classify all inputs** of size **n**, accordingly to the value of the count **C(n)**.

2. **Specify the probability** of each of these classes, accordingly to the problem.

3. **Compute the weighted arithmentic average** as: $\mathbf{C_{avg}(n)}$.

The average-case efficiency is the best approximation of the count C(n), whenever the count C(n) cannot be described by a function.

ANALYSIS - AVERAGE-CASE EFFICIENCY 2

Note: The average-case efficiency $\mathbf{C_{avg}(n) \neq (C_{worst}(n) + C_{best}(n)) / 2}$.

To analyze the average-case efficiency we need assumptions on all inputs of size **n**:

1. **Know/describe all inputs** of size **n**, otherwise we cannot classify them.
2. **Know/specify the probability** of each input class.

The analysis of the average-case efficiency is more difficult than the analyses of the worst-case and best-case efficiencies.

ANALYSIS - EXAMPLE SEQUENTIAL SEARCH 3

Problem: The **sequential search** searches for a given item in a list of **n** items by checking successive items until either the given item is found or the list is exhausted.

```
int i = 0;
while( ( i < n ) && ( list[i] != k ) ) { i++; }
if( i < n ) { return i; }
else { return -1; }
```

- **Input size:** The number of items in the list (**n**).
- **Basic operation:** The **comparison** between a list item and the input item.
- **Worst case:** $C_{worst}(n) = 1 \text{ comparison for each list item} = n$
- **Best case:** $C_{best}(n) = \text{"1 comparison with first list item"} = 1$

ANALYSIS - EXAMPLE SEQUENTIAL SEARCH 4

Problem: The **sequential search** searches for a given item in a list of **n** items by checking successive items until either the given item is found or the list is exhausted.

- **Average case:** Consider **p** the probability of a successful search.
  - **Successful searches:** The probability of the 1st match occurring at position **i** is **p/n** for every **i**, and the number of comparisons made is **i**.
  - **Unsuccessful searches:** The probability is **1-p** and the comparisons are **n**.

$$C_{avg}(n) = \left[1\frac{p}{n} + 2\frac{p}{n} + \cdots + n\frac{p}{n} \right] + n(1-p) = \frac{p(n+1)}{2} + n(1-p)$$

ANALYSIS - ASYMPTOTIC NOTATIONS

The efficiency analysis focuses on the order of growth of count **C(n)**.
To compare and rank such order of growth, we use 3 notations:

- **Big O notation, O.**
- **Big omega notation, Ω.**
- **Big tetha notation, Θ.**

Note: In the following discussion, **t(n)** and **g(n)** can be any non-negative functions defined on the set of natural numbers. In the context we are interested in, **t(n)** will be the running time of an algorithm (usually indicated by the basic operation count **C(n)**), and **g(n)** will be some simple function to compare **t(n)** with.

ANALYSIS - BIG O NOTATION 1

O(g(n)) is the set of all functions with a lower or same order of growth as g(n)

(to within a constant multiple, as n goes to infinity).

Example of assertions using Big O notation	
$456 \in O(n^2)$	Constant has lower order of growth than quadratic.
$100\ n + 5 \in O(n^2)$	Linear has lower order of growth than quadratic.
$n\ (n-1)/2 \in O(n^2)$	Quadratic has same order of growth than quadratic.
$n^3 \notin O(n^2)$	Cubic has higher order of growth than quadratic.
$0.001\ n^3 \notin O(n^2)$	Cubic has higher order of growth than quadratic.
$n^4 + n + 1 \notin O(n^2)$	4-deg-poly has higher order of growth than quadratic.

ANALYSIS - BIG O NOTATION 2

O(g(n)) is the set of all functions with a lower or same order of growth as g(n)
(to within a constant multiple, as n goes to infinity).

Function **t(n)** is in **O(g(n))**, if
t(n) is upper bounded by a
positive constant multiple of **g(n)** for all large **n**.

That is, if there are:

- A positive constant **c**.
- A non-negative integer $\mathbf{n_0}$.
- Such that:

$$t(n) \leq c\ g(n), \quad \text{for all } n \geq n_0.$$

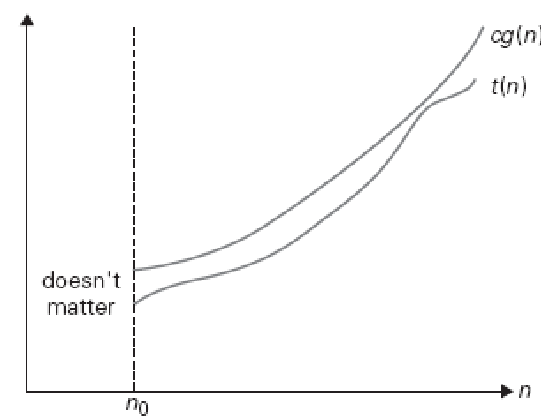

FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$.

ANALYSIS - BIG OMEGA NOTATION 1

$\Omega(g(n))$ is the set of all functions with a higher or same order of growth as $g(n)$
(to within a constant multiple, as n goes to infinity).

Example of assertions using Big Omega notation	
$n^4 \in \Omega(n^2)$	4-deg-poly has higher order of growth than quadratic.
$100\,n^3 + 5 \in \Omega(n^2)$	Cubic has higher order of growth than quadratic.
$n\,(n-1)/2 \in \Omega(n^2)$	Quadratic has same order of growth than quadratic.
$n \notin \Omega(n^2)$	Linear has lower order of growth than quadratic.
$999\,n + 999 \notin \Omega(n^2)$	Linear has lower order of growth than quadratic.
$987 \notin \Omega(n^2)$	Constant has lower order of growth than quadratic.

ANALYSIS - BIG OMEGA NOTATION 2

Ω(g(n)) is the set of all functions with a higher or same order of growth as g(n)
(to within a constant multiple, as n goes to infinity).

Function **t(n)** is in **Ω(g(n))**, if
t(n) is lower bounded by a
positive constant multiple of **g(n)** for all large **n**.

That is, if there are:

- A positive constant **c**.
- A non-negative integer $\mathbf{n_0}$.
- Such that:

$$t(n) \geq c\ g(n), \quad \text{for all } n \geq n_0.$$

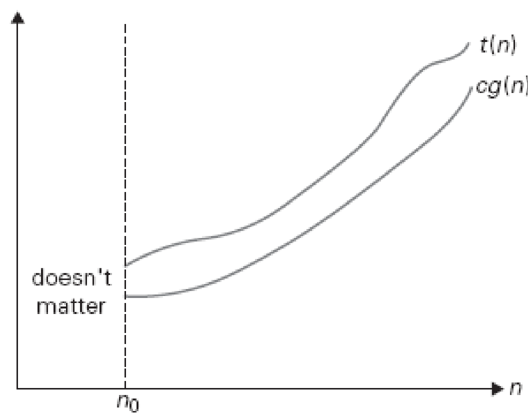

FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$.

ANALYSIS - BIG TETHA NOTATION 1

$\Theta(g(n))$ is the set of all functions with the same order of growth as $g(n)$
(to within a constant multiple, as n goes to infinity).

Example of assertions using Big Tetha notation	
$3n^2 + 2n + 1 \in \Theta(n^2)$	Quadratic has same order of growth than quadratic.
$n^2 + \sin n \in \Theta(n^2)$	Quadratic has same order of growth than quadratic.
$n^2 + \log n \in \Theta(n^2)$	Quadratic has same order of growth than quadratic.
$n \notin \Theta(n^2)$	Linear has lower order of growth than quadratic.
$n^3 \notin \Theta(n^2)$	Cubic has higher order of growth than quadratic.
$987 \notin \Theta(n^2)$	Constant has lower order of growth than quadratic.

ANALYSIS - BIG TETHA NOTATION 2

Θ(g(n)) is the set of all functions with the same order of growth as g(n)
(to within a constant multiple, as n goes to infinity).

Function **t(n)** is in **Θ(g(n))**, if
t(n) is both upper and lower bounded by two positive constant multiples of **g(n)** for all large **n**.

That is, if there are:

- A positive constants $\mathbf{c_1}$ and $\mathbf{c_2}$.
- A non-negative integer $\mathbf{n_0}$.
- Such that:

$$c_2\, g(n) \leq t(n) \leq c_1\, g(n), \quad \text{for all } n \geq n_0.$$

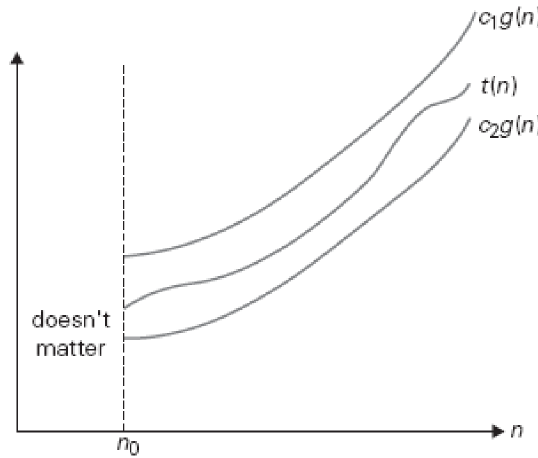

FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$.

ANALYSIS - LIMITS AND ORDERS OF GROWTH

To compare orders of growth of 2 functions,
compute the limit for n→∞ of the ratio of the 2 functions.

Three principal cases may arise (**c** is a positive constant):

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 \Rightarrow t(n) \text{ has lower order of growth than } g(n) \Rightarrow t(n) \in O(g(n)) \\ c \Rightarrow t(n) \text{ has same order of growth than } g(n) \Rightarrow t(n) \in \Theta(g(n)) \\ \infty \Rightarrow t(n) \text{ has higher order of growth than } g(n) \Rightarrow t(n) \in \Omega(g(n)) \end{cases}$$

Note: Sometimes, the limit of the ratio of the two functions may not exist, so in these cases you should use the definitions to compare the orders of growth.

ANALYSIS - ORDERS OF GROWTH PROPERTIES

Theorem:

$$\mathrm{t_1(n)} \in \mathrm{O(\,g_1(n)\,)} \quad \mathrm{AND} \quad \mathrm{t_2(n)} \in \mathrm{O(\,g_2(n)\,)}$$
$$\Downarrow$$
$$\mathrm{t_1(n) + t_2(n)} \in \mathrm{O(\,max\{\,g_1(n), g_2(n)\,\}\,)}$$

Note: Analogous assertions are true for **Ω** and **Θ** notations.

L'Hôpital's rule:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \lim_{n\to\infty} \frac{t'(n)}{g'(n)}$$

Stirling's formula:

$$n! \approx \sqrt{2\,\pi\,n}\left(\frac{n}{e}\right)^n, \quad \text{for large values of } n.$$

ANALYSIS - BASIC EFFICIENCY CLASSES

Basic asymptotic efficiency classes		
1	constant	Usually a short for best-case efficiencies.
log n	logarithmic	Reducing problem size by constant factor each iteration.
n	linear	Usually algorithms that scan a list of size n.
n log n	linearithmic	Many divide-and-conquer algorithms.
$\mathbf{n^2}$	quadratic	Algorithms with two embedded loops.
$\mathbf{n^3}$	cubic	Algorithms with three embedded loops.
$\mathbf{2^n}$	exponential	Algorithms that generate all subsets of an n-items set.
$\mathbf{3^n}$	exponential	a^n has different order of growth for different values of a.
n!	factorial	Algorithms generating all permutations of an n-items set.

ANALYSIS - NON-RECURSIVE ALGORITHMS 1

To analize the time-efficiency of non-recursive algorithms:

1. Decide on the parameters representing the **size of the input n**.

2. Identify the algorithm **basic operation** (usually located in the innermost loop).

3. Check if the count of the basic operation depends only on the input size. If it also depends on other properties, the **worst-/best-/average-cases** are necessary.

4. Set up a **sum** expressing the count of the basic operation.

5. Solve the **sum** into a **closed-form formula** or find its **order of growth**.

ANALYSIS - NON-RECURSIVE ALGORITHMS 2

Summation rules and formulas useful in analize non-recursive algorithms:

Rule 1: $\sum_{i=l}^{u} ca_i = c\sum_{i=l}^{u} a_i$

Rule 2: $\sum_{i=l}^{u}(a_i \mp b_i) = \sum_{i=l}^{u} a_i \mp \sum_{i=l}^{u} b_i$

Formula 1: $\sum_{i=l}^{u} 1 = u - l + 1,$ where $l \leq u$ are lower and upper limits.

Formula 2: $\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n\,(n+1)}{2} \approx \frac{1}{2}\, n^2 \in \Theta(n^2)$

ANALYSIS - EXAMPLE ELEMENT UNIQUENESS 1

Problem: The **element uniqueness** checks if all items in a list of **n** items are distinct.

```
boolean checkElementUniqueness( int[] list ) {
   for( int i = 0; i < list.length - 1; i++ ) {
      for( int j = i+1; j < list.length; j++ ) {
         if( list[i] == list[j] ) { return false; }
      }
   }
   return true;
}
```

- **Input size:** The number of items in the list (**n**).
- **Basic operation:** The **comparison** between 2 list items.
- **Best case:** $C_{best}(n) = \text{"}1^{st}\text{ comparison fails"} = 1$

ANALYSIS - EXAMPLE ELEMENT UNIQUENESS 2

- **Worst case:** The comparisons performed $C_{worst}(n)$ are the max (in lists of size **n**):
  - Lists with no equal items.
  - Lists in which the last 2 items are the only pair of equal items.

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [\,(n-1) - (i+1) + 1\,] = \sum_{i=0}^{n-2} (n-1-i) =$$

$$= (n-1) + (n-2) + \cdots + 1 = \frac{(n-1)\,n}{2} = \frac{1}{2}\,n^2 - \frac{1}{2}n \in \Theta(n^2)$$

ANALYSIS - RECURSIVE ALGORITHMS 1

To analize the time-efficiency of recursive algorithms:

1. Decide on the parameters representing the **size of the input n**.
2. Identify the algorithm **basic operation** (usually located in the innermost loop).
3. Check if the count of the basic operation depends only on the input size. If it also depends on other properties, the **worst-/best-/average-cases** are necessary.
4. Set up a **recursive definition** expressing the count of the basic operation.
5. Solve the **recursive definition** into a **closed-form formula** or find its **order of growth**.

ANALYSIS - RECURSIVE ALGORITHMS 2

Methods useful in analize recursive algorithms:

- Forward Substitutions
- **Backward Substitutions**
- **Smooth Function**
- **Smoothness Rule**
- Master Theorem

ANALYSIS - RECURSIVE ALGORITHMS 3

Methods useful in analize recursive algorithms: Backward Substitutions

1. Starting from the recursive count **C(n)**.
2. Apply iteratively the recurrence relation from the count definition.
3. Identify the pattern, parameterizing the equation.
4. Solve the pattern using last legit value for the parameter, and base case.

Methods useful in analize recursive algorithms: Smooth Function

Eventually Non-Decreasing Function: Let **C(n)** be a non-negative function defined on natural numbers. **C(n)** is eventually non-decreasing if there is a non-negative integer $\mathbf{n_0}$ so that **C(n)** is non-decreasing in the interval $[\, n_0 , \infty \,)$.

Smooth Function: Let **C(n)** be a non-negative function defined on natural numbers. **C(n)** is called **smooth** if it is **eventually non-decreasing** and:

$$C(\,2n\,) \in \Theta(\,C(n)\,)$$

Note: Fast-growing functions, such as $\mathbf{a^n}$ (with a>1) and **n!** are not smooth.

ANALYSIS - RECURSIVE ALGORITHMS 5

Methods useful in analize recursive algorithms: Smoothness Rule

Smoothness Rule: Let **C(n)** be an **eventually non-decreasing** function and let **f(n)** be a **smooth** function.

$$C(n) \in \Theta(\ f(n)\ ), \qquad \forall\ n = b^k, b \geq 2$$
$$\Downarrow$$
$$C(n) \in \Theta(\ f(n)\ ), \qquad \forall\ n$$

Note: Analogous results hold for **O** and **Ω** as well.

ANALYSIS - EXAMPLE FACTORIAL 1

Problem: Compute the **factorial F(n)=n!** (recursively) for a non-negative integer **n**.

```
int F( int n ) {
   if( n == 0 ) { return 1; }
   else { return ( F( n-1 ) * n ); }
}
```

$$F(n) = n! = \begin{cases} 1, & \text{if } n = 0. \\ F(n-1) \times n, & \text{if } n \geq 1. \end{cases}$$

- **Input size:** The magnitude of the input number (**n**).
- **Basic operation:** The **multiplication** between 2 integers.

ANALYSIS - EXAMPLE FACTORIAL 2

- **Count basic operation:** It depends only on input size, and we call it **M(n)**.

$$\text{algorithm factorial} = \text{F(n)} = \text{n!} = \begin{cases} 1, & \text{if n} = 0. \\ \text{F(n} - 1) \times \text{n}, & \text{if n} \geq 1. \end{cases}$$

$$\text{count multiplications} = \text{M(n)} = \begin{cases} 0, & \text{if n} = 0. \\ \text{M(n} - 1) + 1, & \text{if n} \geq 1. \end{cases}$$

Note: Once the recursive definition for the count is complete, we can use several techniques to solve it into a closed-form formula, or to determine its order of growth.

ANALYSIS - EXAMPLE FACTORIAL 3

$$\text{count multiplications} = M(n) = \begin{cases} 0, & \text{if } n = 0. \\ M(n-1) + 1, & \text{if } n \geq 1. \end{cases}$$

To solve **M(n)** we use the **method of backward substitutions:**

apply recurrence relation →

$$\begin{aligned} M(n) &= M(n-1) + 1 = \\ &= [\,M(n-2) + 1\,] + 1 = M(n-2) + 2 = \\ &= [\,M(n-3) + 1\,] + 2 = M(n-3) + 3 = \cdots \end{aligned}$$

$$\Downarrow$$

identify pattern → $M(n) = M(n-i) + i, \qquad i \in [\,0, n\,]$

$$\Downarrow$$

solve pattern for last i → $M(n) = M(n-i) + i = M(n-n) + n = M(0) + n = n$

ANALYSIS - EXAMPLE BINARY DIGITS 1

Problem: Find the **number of binary digits** of a positive integer **n**.

```
int F( int n ) {
   if( n == 1 ) { return 1; }
   else { return F( (int) Math.floor( n / 2 ) ) + 1; }
}
```

$$F(n) = \text{"number of binary digits of integer n"} = \begin{cases} 1, & \text{if } n = 1. \\ F(\lfloor n / 2 \rfloor) + 1, & \text{if } n > 1. \end{cases}$$

- **Input size:** The magnitude of the input number (**n**).
- **Basic operation:** The **addition** between 2 integers (e.g. increment by 1).

ANALYSIS - EXAMPLE BINARY DIGITS 2

- **Count basic operation:** It depends only on input size, and we call it **A(n)**.

$$\text{algorithm binary digits} = F(n) = \begin{cases} 1, & \text{if } n = 1. \\ F(\lfloor n/2 \rfloor) + 1, & \text{if } n > 1. \end{cases}$$

$$\text{count additions} = A(n) = \begin{cases} 0, & \text{if } n = 1. \\ A(\lfloor n/2 \rfloor) + 1, & \text{if } n > 1. \end{cases}$$

Note: The **floor** operator in recursive definition makes the method of **Backward Substitutions** useless on values of **$n \neq 2^k$**. So, we solve this recursive definition for **$n=2^k$ ($k=\log_2 n$)**, and then we use the **Smoothness Rule**.

ANALYSIS - EXAMPLE BINARY DIGITS 3

$$\text{count additions} = A(n) = \begin{cases} 0, & \text{if } n = 1. \\ A(\lfloor n / 2 \rfloor) + 1, & \text{if } n > 1. \end{cases}$$

To solve **A(n)** we use the **method of backward substitutions for $n=2^k$ ($k=\log_2 n$):**

apply recurrence relation →

$$\begin{aligned} A(n) &= A(2^k) = A(\lfloor 2^k / 2 \rfloor) + 1 = A(2^{k-1}) + 1 = \\ &= [A(\lfloor 2^{k-1} / 2 \rfloor) + 1] + 1 = A(2^{k-2}) + 2 = \\ &= [A(\lfloor 2^{k-2} / 2 \rfloor) + 1] + 2 = A(2^{k-3}) + 3 = \cdots \end{aligned}$$

$$\Downarrow$$

identify pattern → $A(2^k) = A(2^{k-i}) + i, \qquad i \in [0, k]$

$$\Downarrow$$

solve pattern for last i → $A(2^k) = A(2^{k-i}) + i = A(2^{k-k}) + k = A(1) + k = k$

ANALYSIS - EXAMPLE BINARY DIGITS 4

$$\text{count additions} = A(n) = \begin{cases} 0, & \text{if } n = 1. \\ A(\lfloor n / 2 \rfloor) + 1, & \text{if } n > 1. \end{cases}$$

The result of the **method of backward substitutions for $n=2^k$ ($k=\log_2 n$):**

$$A(n) = A\left(2^k\right) = k = \log_2 n \in \Theta(\log n), \qquad \forall\, n = 2^k$$

$$\Downarrow$$

$$\log n \text{ is smooth, so we can apply smoothness rule}$$

$$\Downarrow$$

$$A(n) = A\left(2^k\right) = k = \log_2 n \in \Theta(\log n), \qquad \forall\, n$$

ANALYSIS - EMPIRICAL ANALYSIS 1

To empirically analize the time-efficiency of algorithms:

1. Decide **metric** and its **units** to be measured (count, time).
2. Decide **input specs** (range, size) and create the **input sample** (set).
3. Run **algorithm on input sample**, and **record the data** (metric and units).
2. Process and **analyze the data**.

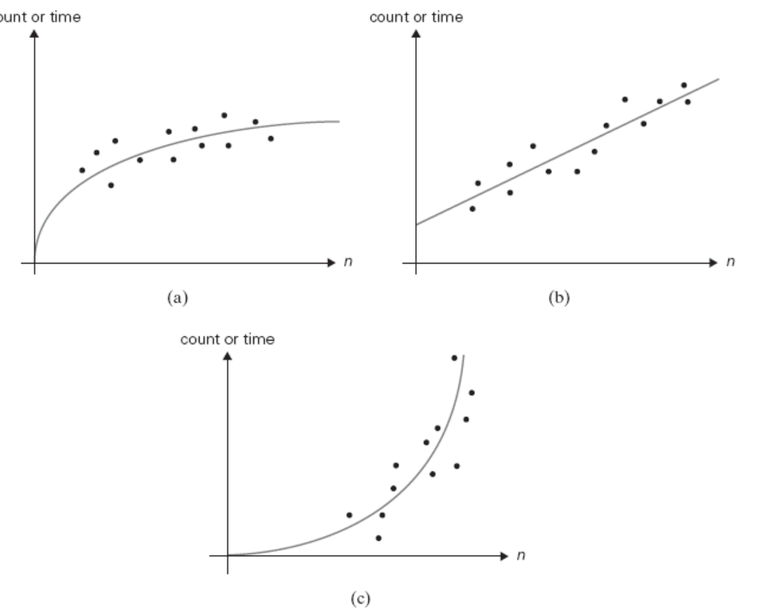

FIGURE 2.7 Typical scatter plots. (a) Logarithmic. (b) Linear. (c) One of the convex functions.

ANALYSIS - EMPIRICAL ANALYSIS 2

Considerations useful to empirically analize the time-efficiency of algorithms:

- The **experiment design** depends on the goal (comparison, efficiency check).
- Operating system **time queries** are not accurate.
- If running time is small, you can **measure several runs** at the same time.
- On a **time-sharing operating system**, time queries include other processes.
- Time different program parts can pinpoint a bottleneck (**profiling**).

ANALYSIS - ALGORITHM VISUALIZATION 1

There is another way to study algorithms, **algorithm visualization**:

- Visualize the **algorithm operations**.
- Display the **algorithm performance on different inputs**.
- Graph the **comparison of various algorithm performances** for same problem.

There are 2 main applications of algorithm visualization:

- **Research:** Help researchers uncover some unknown algorithm features.
- **Education:** Help students learning algorithms.

There are 2 principal variations of algorithm visualization:

- **Static:** Shows the progress of an algorithm through a series of still images.
- **Dynamic (Animation):** Shows a movie-like animation of algorithm operations.

ANALYSIS - ALGORITHM VISUALIZATION 2

Examples of algorithm visualization

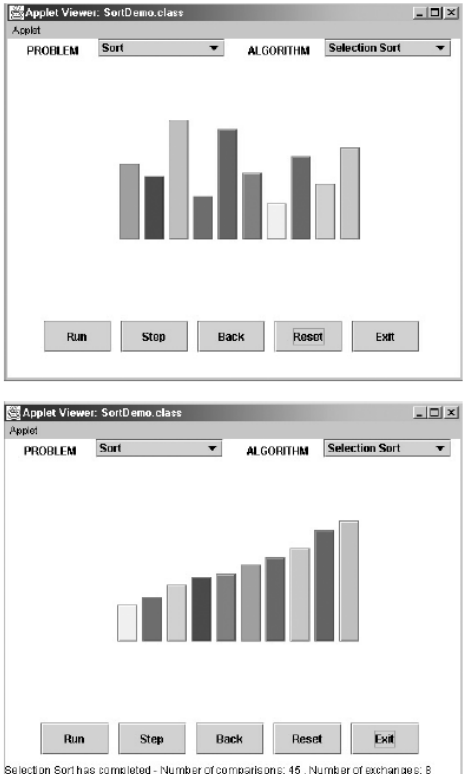

FIGURE 2.8 Initial and final screens of a typical visualization of a sorting algorithm using the bar representation.

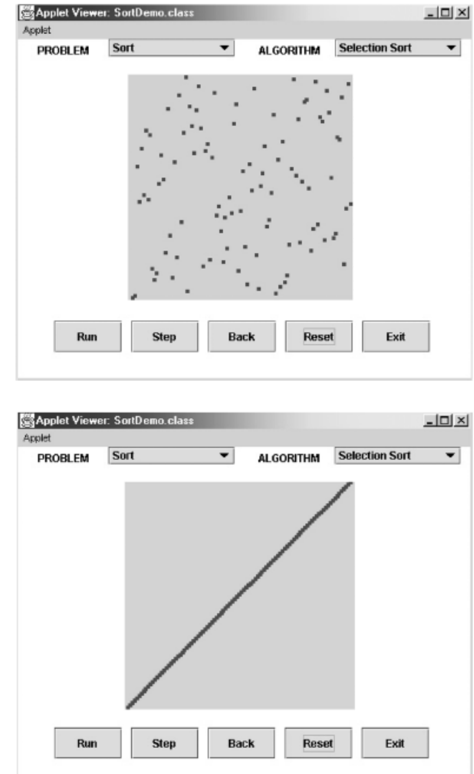

FIGURE 2.9 Initial and final screens of a typical visualization of a sorting algorithm using the scatterplot representation.

ANALYSIS - DATA STRUCTURES

- **Linear:** A data structure is linear if its data is arranged in a sequence.
  - **Array-Based:** Array-list, stacks, and queues.
  - **Reference-Based:** Linked lists, and all their variations (circular etc.).
- **Tree:** A connected acyclic graph.
  - **Reference-Based:** Hierarchy of nodes (standard, first-child-next-sibling).
  - **Array-Based:** Hierarchy implicit in array indexing (standard, heaps).
- **Graph:** A set of vertices/nodes connected in pairs by edges/links.
  - **Adjacency Matrix:** 2D array, storing vertex connectivity.
  - **Adjacency Lists:** 1D array of linked lists, storing vertex connectivity.
  - **Edge List:** 1D array of graph edges.
- **Set:** A data structure that can store unique items in no particular order.
  - **Bit Vector:** 1D bit-array associating each bit to an item in the set.
- **Dictionary:** A data structure implementing search, insertion, and removal by key.
  - **Hashing:** Hash table, array with custom key-based indexing.

ANALYSIS - LINEAR ARRAY-BASED 1

An **array** is a series of items of same type, **stored contiguously** in memory and accessible by using an **index**.

FIGURE 1.3 Array of n elements.

Each array item can be accessed in the same constant time regardless of where the item is indexed in the array.

Linear data structures using arrays:

- **Array-Based Lists:** Data packed in left part, leaving empty right part.
- **Stacks:** Top at array back, bottom at array front.
- **Queues:** Circular array to avoid data drifting.

ANALYSIS - LINEAR ARRAY-BASED 2

Efficiency of linear array-based data structures

array-list	**single access**	**$\Theta(1)$:** constant time.
array-list	**insert/remove**	**$\Omega(1)$, $O(n)$:** shifts are time-consuming.
array-list	**resize**	**$\Theta(n)$:** data transfer is time-consuming.

stack	**peek/pop/push**	**$\Theta(1)$:** constant time.
stack	**resize**	**$\Theta(n)$:** data transfer is time-consuming.

queue	**peek/dequeue/enqueue**	**$\Theta(1)$:** constant time.
queue	**resize**	**$\Theta(n)$:** data transfer is time-consuming.

ANALYSIS - LINEAR REFERENCE-BASED 1

A **linked list (LL)** is a series of nodes, each containing this information:

- **Data** of this LL item.
- **Links** to other LL nodes.

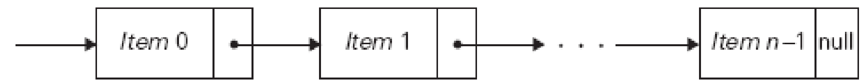

FIGURE 1.4 Singly linked list of n elements.

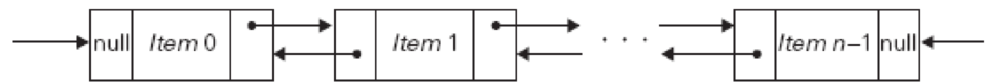

FIGURE 1.5 Doubly linked list of n elements.

Linked list items are accessed only by starting from an entry point (head, tail) so the access time depends on the length of the path entry-point-to-item.

Linear data structures using references:

- **Linked Lists (LLs):** Entry point at front (head), only forward traversal possible.
- **Circular/Doubly/Tail LLs:** Variations of LLs, diverse entry points and links.
- **Stacks:** Top at LL front, bottom at LL back.
- **Queues:** LL with tail, front at LL head, back at LL tail.

ANALYSIS - LINEAR REFERENCE-BASED 2

Efficiency of linear reference-based data structures		
linked list	**single access**	**$\Omega(1)$, $O(n)$:** access starts at entry point.
linked list	**insert/remove**	**$\Omega(1)$, $O(n)$:** limited by access time.
circular doubly LL	**single access**	**$\Omega(1)$, $O(n)$:** improved avg access time.
circular doubly LL	**insert/remove**	**$\Omega(1)$, $O(n)$:** improved avg access time.
stack	**peek/pop/push**	**$\Theta(1)$:** constant time.
queue	**peek/deq/enq**	**$\Theta(1)$:** constant time.

ANALYSIS - TREE REPRESENTATIONS 1

A **tree** is a set of nodes with
a **hierarchical structure**
(parent-child relations among nodes)

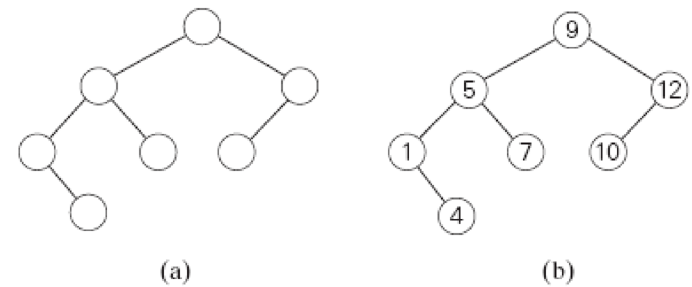

FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

The perfomance of a tree is mainly driven by its depth/height so it is convenient to control its shape to ensure the data is packed.

Tree representations:

- **Reference-Based:** Standard hierarchy of nodes linked using references.
- **Array-Based:** 1D array of linked lists, storing vertex connectivity.
- **First-Child-Next-Sibling:** Reference-based, non-standard.

ANALYSIS - TREE REPRESENTATIONS 2

Efficiency of tree representations		
reference-based BST	**search/ins/del**	**Ω(1), O(n):** depends on depth/shape.
reference-based BST	**traversal**	**Θ(n):** need to visit all nodes.
array-based BST	**search/ins/del**	**Ω(1), O(n):** depends on depth/shape.
array-based BST	**traversal**	**Θ(n):** need to visit all nodes.
array-based heap	**search**	**Θ(1):**
array-based heap	**insert/delete**	**Ω(1), O(log n):** always complete.

ANALYSIS - GRAPH REPRESENTATIONS 1

A **graph** is a set of vertices, connected by edges that could be weighted or directed/undirected.

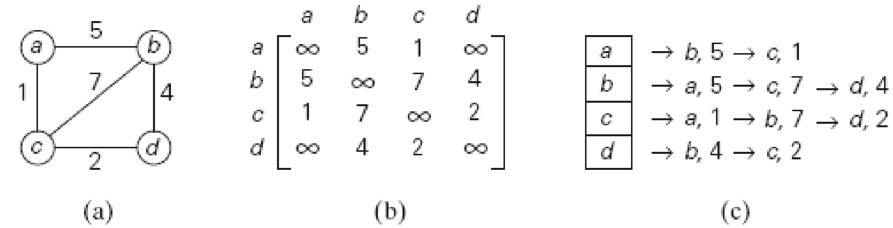

FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

Each array item can be accessed in the same constant time regardless of where the item is indexed in the array.

Graph representations:

- **Adjacency Matrix:** 2D array, storing vertex connectivity.
- **Adjacency Lists:** 1D array of linked lists, storing vertex connectivity.
- **Edge List:** 1D array of graph edges.

ANALYSIS - GRAPH REPRESENTATIONS 2

	Efficiency of graph representations	
adjacency matrix	**check edge a-b**	**$\Theta(1)$:** constant time.
adjacency matrix	**add new vertex**	**$\Theta(n^2)$:** resize 2D matrix.
adjacency matrix	**delete vertex**	**$\Theta(n)$:** clear 1 row and 1 column.

adjacency lists	**check edge a-b**	**$\Omega(1)$, $O(n)$:** depends on LL efficiency.
adjacency lists	**add new vertex**	**$\Theta(n)$:** resize 1D array.
adjacency lists	**delete vertex**	**$\Theta(1)$:** clear 1 array cell.

edge list	**any operation**	**$\Omega(1)$, $O(n)$:** same as array-list efficiency.