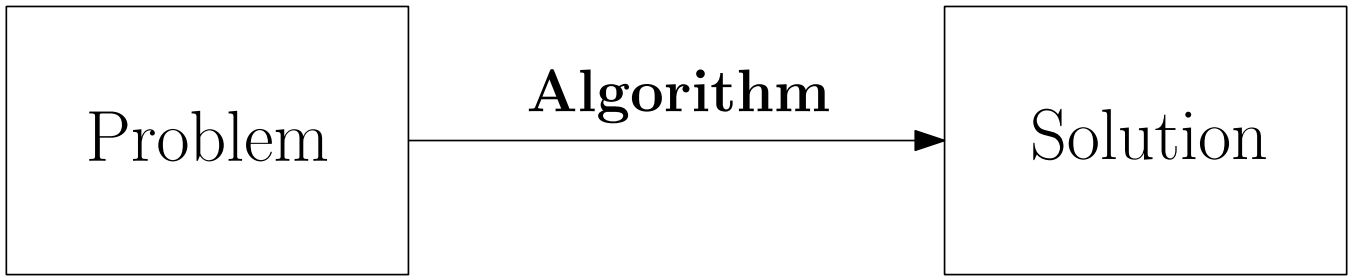
Topic 1: Foundations

Agenda

- What is an algorithm?
- Computational Algorithms: Real-World Applications
- Algorithmic Thinking
- Pseudocodes & Flowcharts
- Algorithm Analysis: Time/Space, Asymptotic Notation
- Algorithm Design: Recursion, Divide & Conquer, Randomization

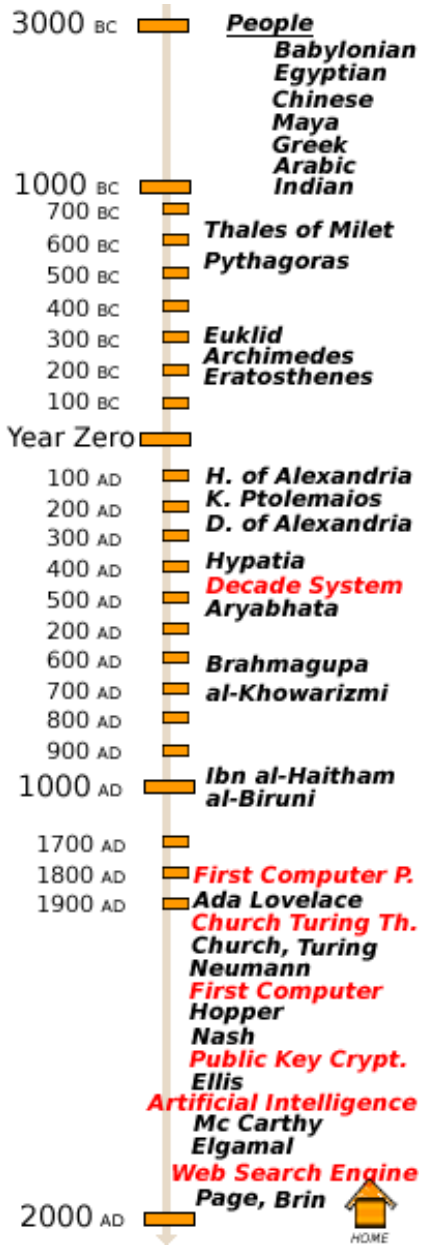
What is an algorithm?

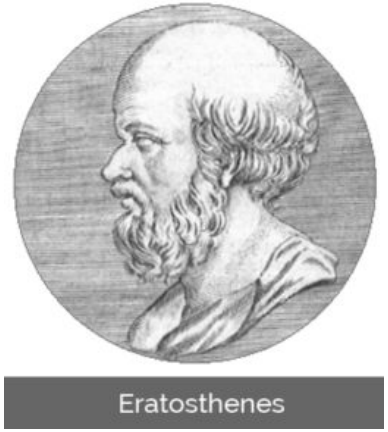


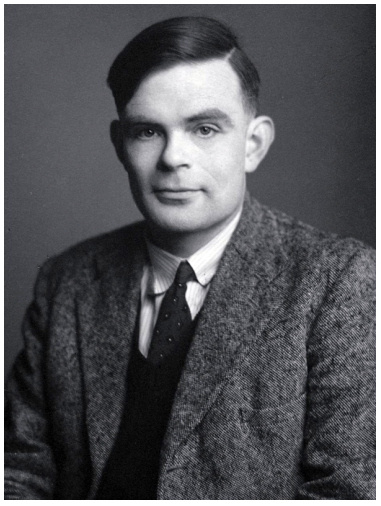
Algorithm *is a recipe (well-defined procedure) for solving a problem that takes a value (or a set of values) as input, and produces some value (or a set of values) as output.*

Program *is a specific implementation instance of the algorithm(s) in a specific programming language.*

History of Algorithms

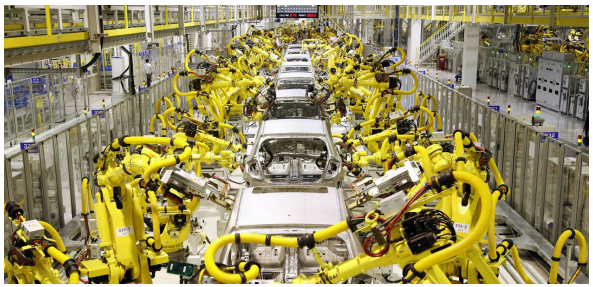



Eratosthenes



Source: http://cs-exhibitions.uni-klu.ac.at/index.php?id=340

Real-World Applications



Course Philosophy

In this course, we will inculcate **algorithmic thinking**, which is a fundamental skill in the domain of computing.

Algorithmic thinking comprises of two paradigms:

- Algorithm Design
- Algorithm Analysis

Before we delve into these two paradigms, we will first learn about writing algorithms.

- Pseudocodes
- Flowcharts

How do we write algorithms?

Algorithms are not written in programming languages!

Following are different syntaxes used in practice to write an algorithm for fizz-buzz game:

Fortran style pseudo code

```
program fizzbuzz
  Do i = 1 to 100
    set print_number to true
    If i is divisible by 3
      print "Fizz"
      set print_number to false
    If i is divisible by 5
      print "Buzz"
      set print_number to false
    If print_number, print i
    print a newline
  end do
```

Pascal style pseudo code

```
procedure fizzbuzz
  For i := 1 to 100 do
    set print_number to true;
    If i is divisible by 3 then
      print "Fizz";
      set print_number to false;
    If i is divisible by 5 then
      print "Buzz";
      set print_number to false;
    If print_number, print i;
    print a newline;
  end
```

C style pseudo code:

```
void function fizzbuzz {
  for (i = 1; i <= 100; i++) {
    set print_number to true;
    If i is divisible by 3 {
      print "Fizz";
      set print_number to false; }
    If i is divisible by 5 {
      print "Buzz";
      set print_number to false; }
    If print_number, print i;
    print a newline;
  }
}
```

Structured Basic style pseudo code

```
Sub fizzbuzz()
  For i = 1 to 100
    print_number = True
    If i is divisible by 3 Then
      Print "Fizz"
      print_number = False
    End If
    If i is divisible by 5 Then
      Print "Buzz"
      print_number = False
    End If
    If print_number = True Then print i
    Print a newline
  Next i
End Sub
```

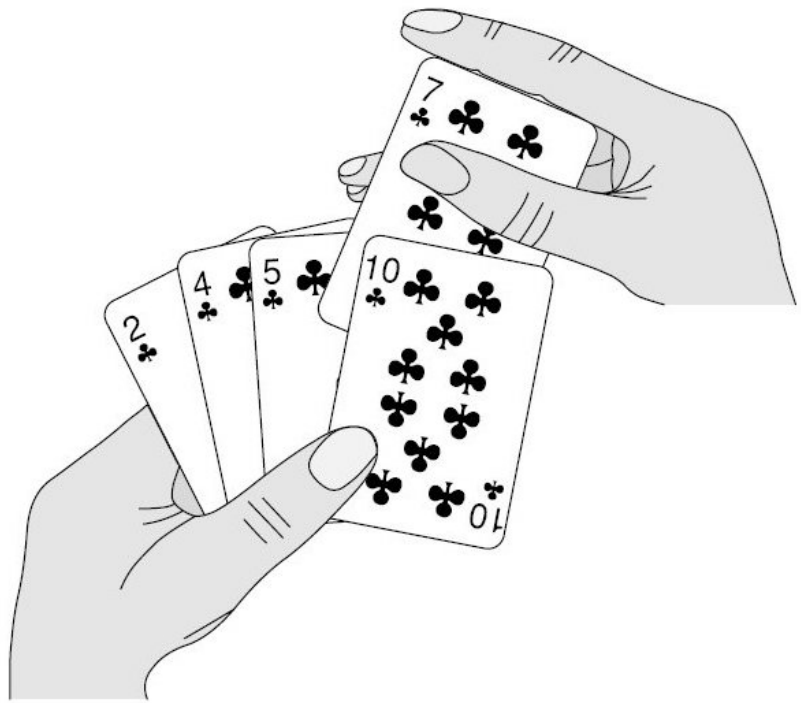
Source: https://en.wikipedia.org/wiki/Pseudocode

In this course, we will use mathematical pseudocodes.

Example: Insertion Sort

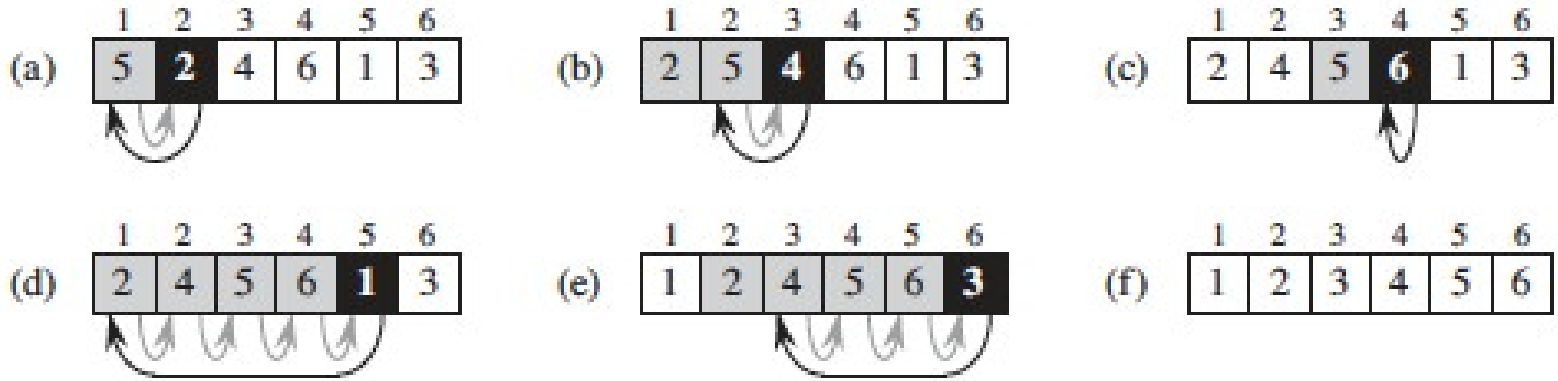
Input: A array of n numbers, $A = \{a_1, \cdots, a_n\}$

Output: A permutation (reordering) $A' = \{a'_1, \cdots, a'_n\}$ of the input sequence such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.



Source: Figure 2.1 in the textbook.

Pseudocode for Insertion Sort

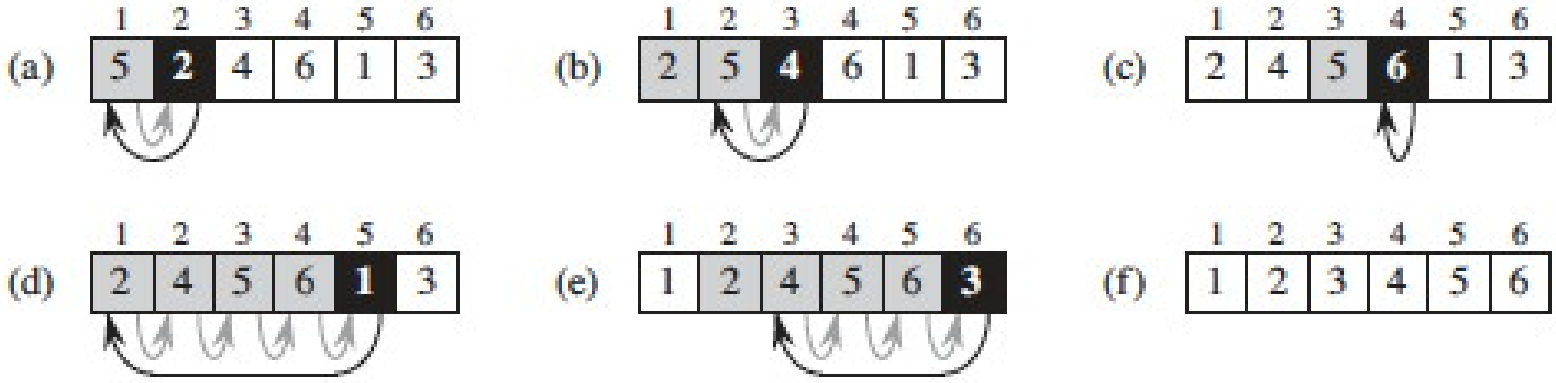


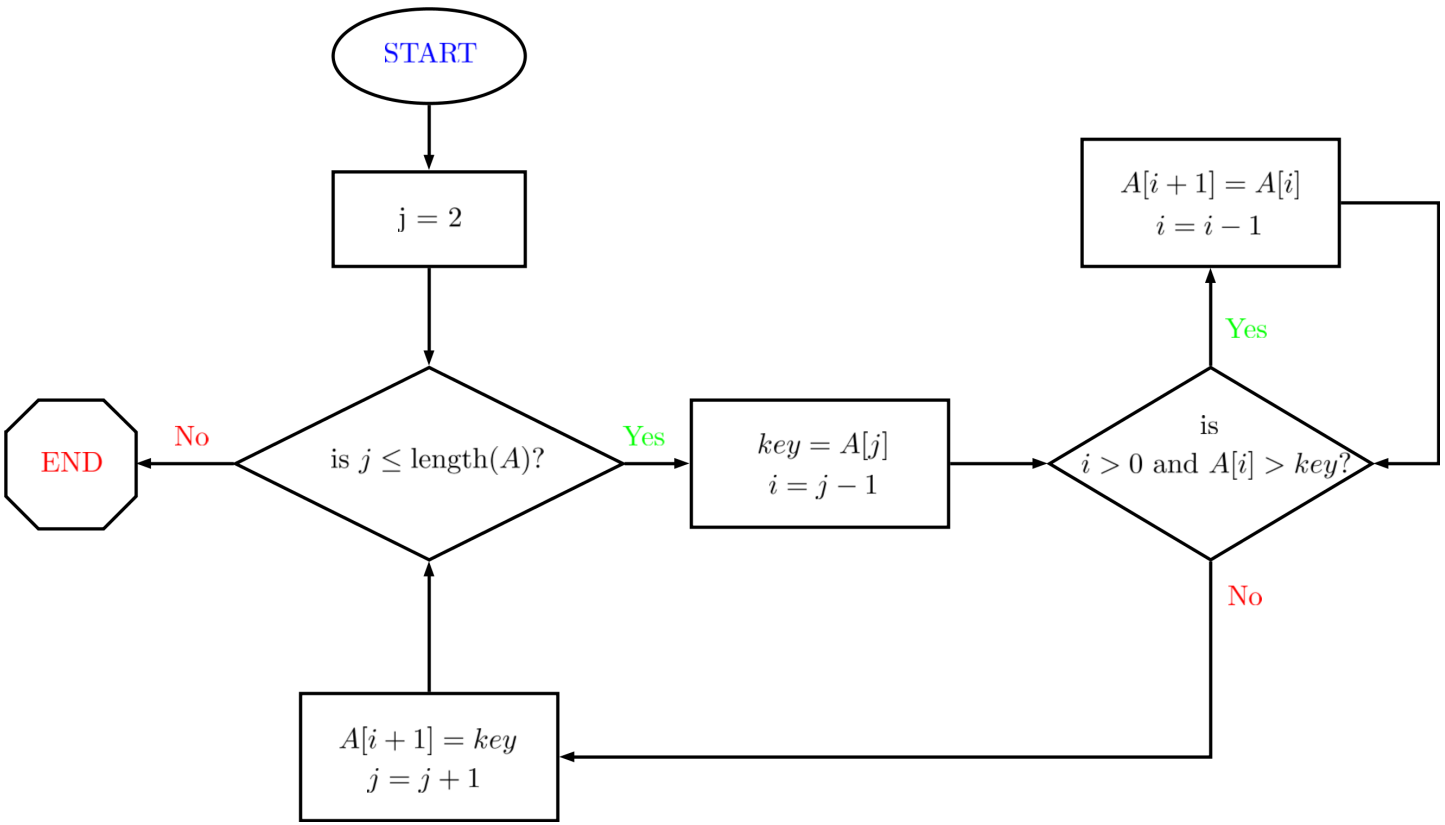
Source: Figure 2.2 in the textbook.

INSERTION-SORT(A)

1 **for** $j = 2$ **to** $A.length$
2 $key = A[j]$
3 // insert $A[j]$ into the sorted sequence $A[1 : j-1]$.
4 $i = j - 1$
5 **while** $i > 0$ and $A[i] > key$
6 $A[i+1] = A[i]$
7 $i = i - 1$
8 $A[i+1] = key$

Flowchart for Insertion Sort





Algorithm Analysis

But, a problem may have several algorithms.
(e.g. insertion sort, merge sort...)

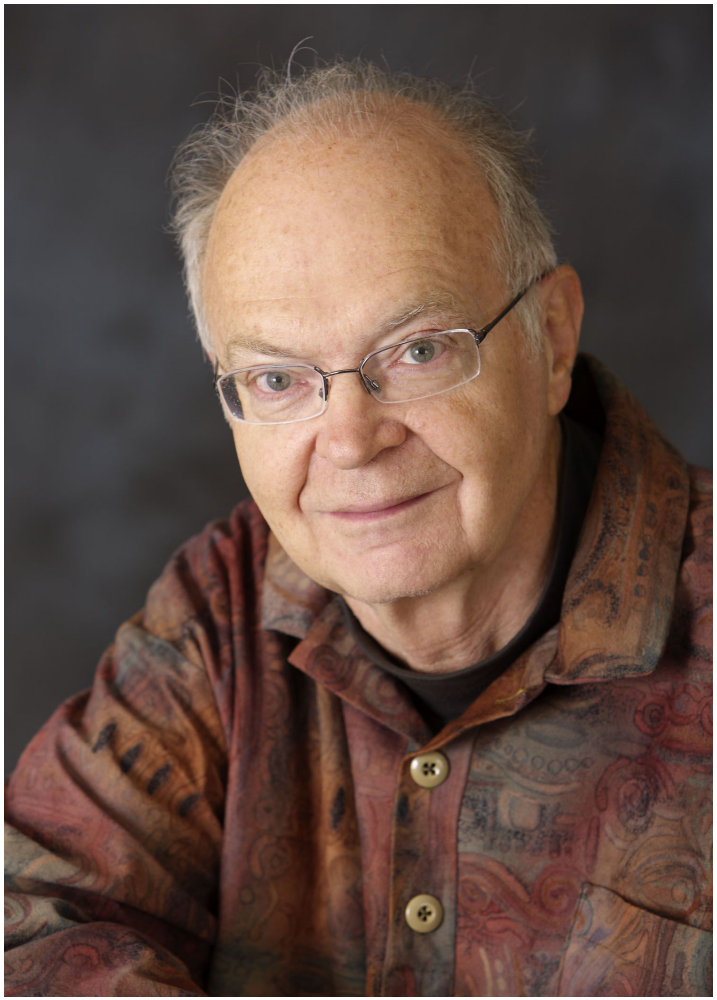
Which algorithm suits best for a given setting?

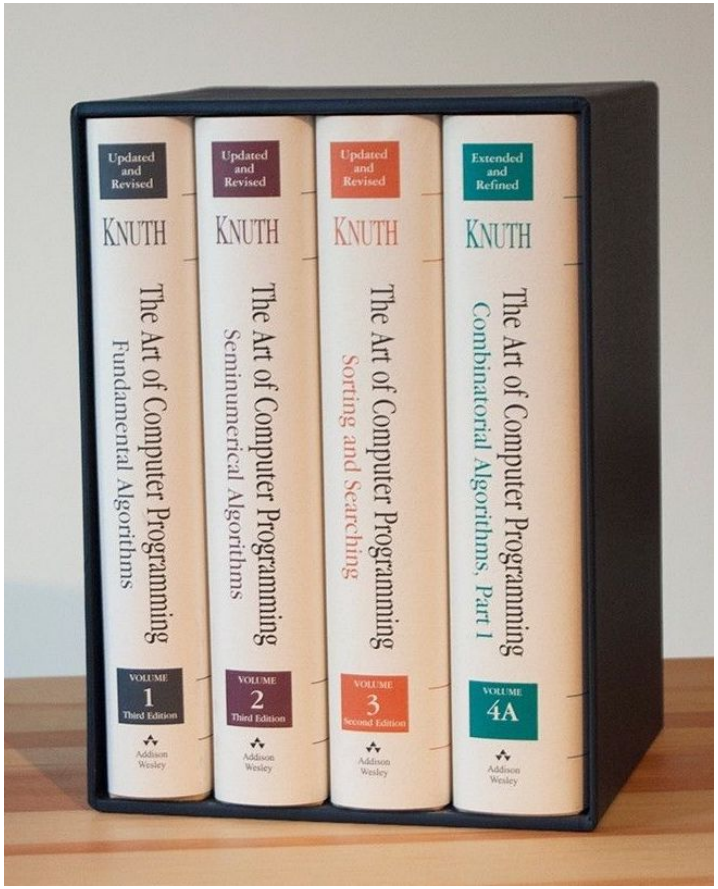
- Correctness
- Memory (Space)
- Run-time Complexity
- Asymptotic Growth

What if computers are infinitely fast and computer memory was free? Do we still need to do this course?

- Well! Even though we have a **correct** method, we still want the algorithm to **terminate**.

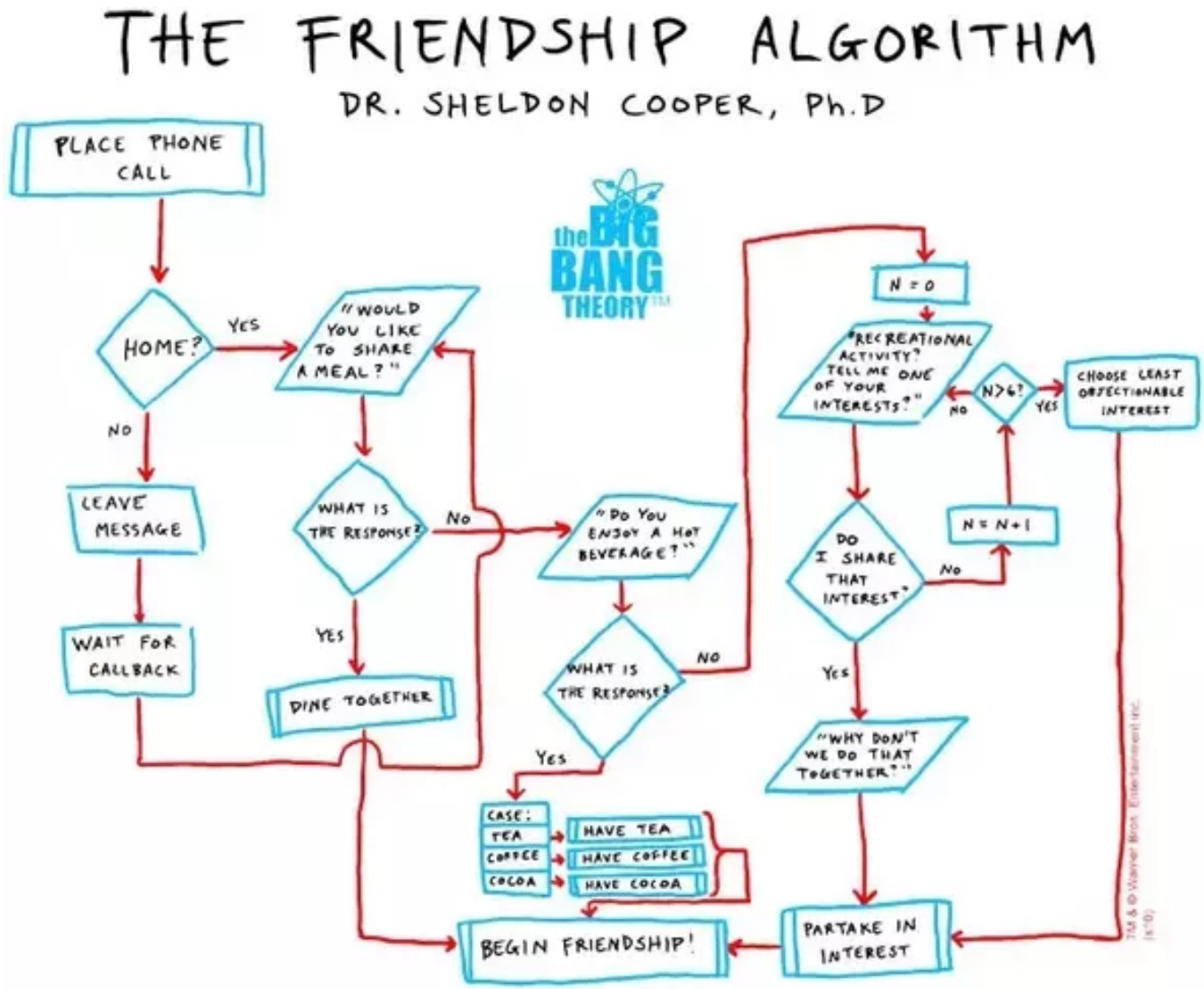
The Birth of Algorithm Analysis





Note: Although studied later in historical context, let us first look into correctness of algorithms.

Why analyze correctness?



Correctness: Some History

Robert W. Floyd

ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an
formal definitions of the meanings of programs in ap
programming languages, in such a way that a rigorous sta
for proofs about computer programs, including proc
equivalence, and termination. The basis of our approac
an interpretation of a program: that is, an associatio
with each connection in the flow of control through a p
proposition is asserted to hold whenever that connection i
an interpretation from being chosen arbitrarily, a condi
each command of the program. This condition guarant
a command is reached by way of a connection whose ass
is then true, it will be left (if at all) by a connectio
proposition will be true at that time. Then by induction
commands executed, one sees that if a program is enter
whose associated proposition is then true, it will be le
connection whose associated proposition will be true at
means, we may prove certain properties of programs, par
of the form: "If the initial values of the program va
relation R_1, the final values on completion will satisfy
Proofs of termination are dealt with by showing that eacl
decreases some entity which cannot decrease indefini

These modes of proof of correctness and terminatio
they are based on ideas of Perlis and Gorn, and may
earliest appearance in an unpublished paper by Gorn.
of formal standards for proofs about programs in lang
assignments, transfer of control, etc., and the proposal

An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming, proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

1. Introduction

Computer programming is an exact science in that all the properties of a program and all the consequences of executing it in any given environment can, in principle, be found out from the text of the program itself by means of purely deductive reasoning. Deductive reasoning involves the application of valid rules of inference to sets of valid axioms. It is therefore desirable and interesting to elucidate the axioms and rules of inference which underlie our reasoning about computer programs. The exact choice

of axioms it is possible to deduce such simple theorems as:

$$x = x + y \times 0$$

$$y \leqslant r \supset r + y \times q = (r - y) + y \times (1 + q)$$

The proof of the second of these is:

A5 $(r - y) + y \times (1 + q)$

$= (r - y) + (y \times 1 + y \times q)$

A9 $= (r - y) + (y + y \times q)$

A3 $= ((r - y) + y) + y \times q$

A6 $= r + y \times q$ provided $y \leqslant r$

The axioms A1 to A9 are, of course, true of the traditional infinite set of integers in mathematics. However, they are also true of the finite sets of "integers" which are manipulated by computers provided that they are confined to *nonnegative* numbers. Their truth is independent of the size of the set; furthermore, it is largely independent of the choice of technique applied in the event of "overflow"; for example:

(1) Strict interpretation: the result of an overflowing operation does not exist; when overflow occurs, the offending program never completes its operation. Note that in this case, the equalities of A1 to A9 are strict, in the sense that both sides exist or fail to exist together.

(2) Firm boundary: the result of an overflowing operation is taken as the maximum value represented.

(3) Modulo arithmetic: the result of an overflowing operation is computed modulo the size of the set of integers represented.

These three techniques are illustrated in Table II by addition and multiplication tables for a trivially small model in which 0, 1, 2, and 3 are the only integers represented.

It is interesting to note that the different systems satisfying axioms A1 to A9 may be rigorously distinguished from each other by choosing a particular one of a set of mutually exclusive supplementary axioms. For example, infinite

Correctness

- **Functional Correctness:** For each input, produce expected output
- **Partial Correctness:** If an output is returned, it is correct.
- **Total Correctness:** Partial correctness + Termination.

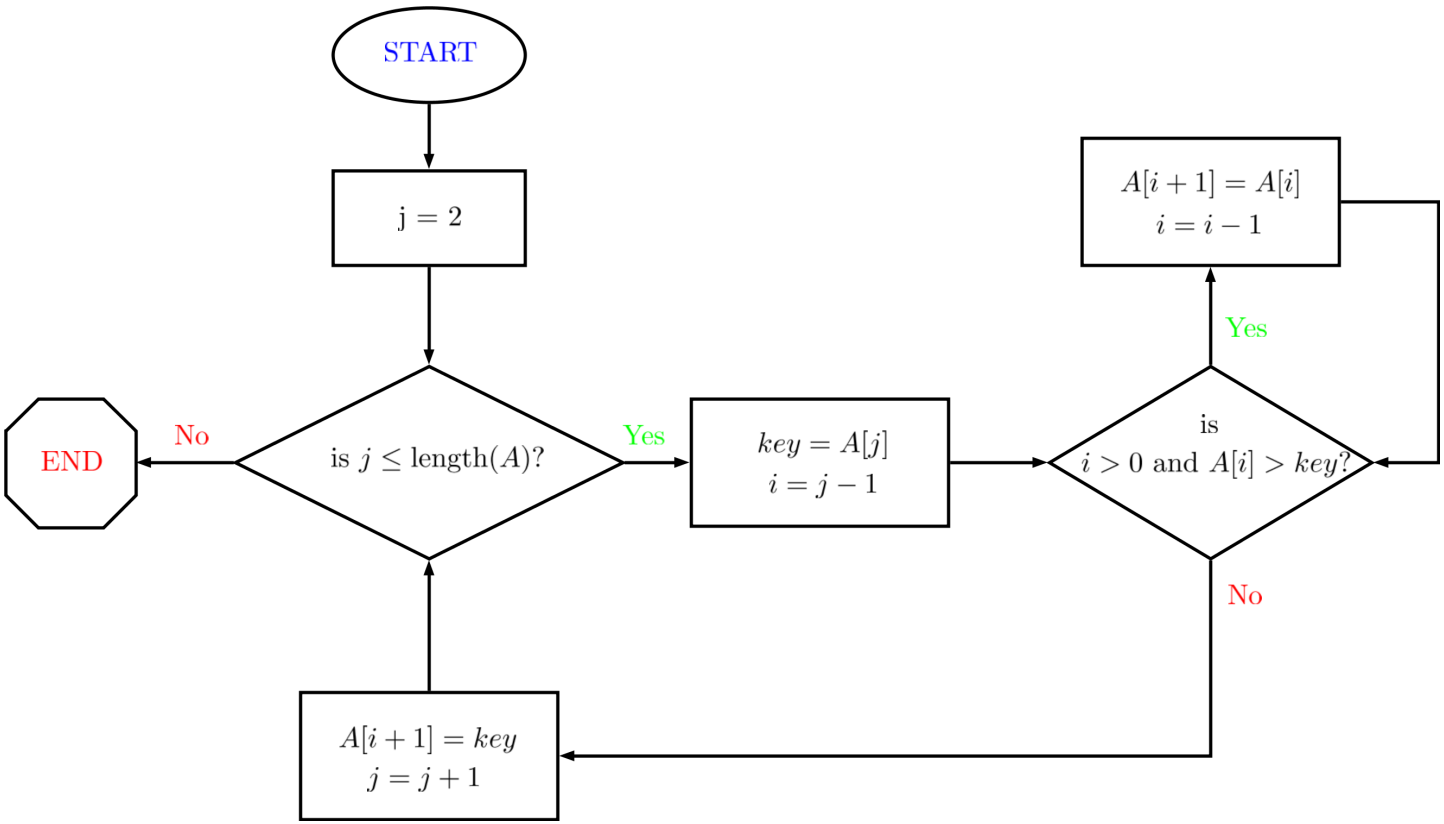
To verify partial correctness, we rely on **loop invariants:**

- A property of a loop that does not change before/after each iteration.
- Logical assertion
- Proof Techniques: Principle of Mathematical Induction.

Example 1: Insertion Sort

Let $A[1 : j-1] = \{A[1], \cdots, A[j-1]\}$.

is $A[1 : j-1]$ sorted in an ascending order?



Insertion Sort: Proof of Correctness

Initialization: $A[1:j-1]|_{j=2} = A[1]$, which has a single-entry. In other words, it is sorted trivially in an ascending manner.

Maintenance: Let $A[1:j-1]$ be sorted in an ascending order. In the iteration indexed as j, we have $key = a_j$. Then, we have two cases:

1. If $A[j-1] \leq a_j$, the iteration is complete, and the loop invariant for the next iteration $A[1:j]$ is therefore sorted.
2. If $A[i] \leq a_j \leq A[i+1] \leq A[i+2] \leq \cdots \leq A[j-1]$ for some $i \in \{0, \cdots, j-2\}$, a single iteration of the *for* loop results in the following assignments: $A[j] = A[j-1]$, $A[j-1] = A[j-2]$, $\cdots$, $A[i+2] = A[i+1]$, and $A[i+1] = a_j$. Consequently,

$$A'[1:j] = \{A[1], \cdots, A[i], a_j, A[i+1], \cdots, A[j-1]\}$$

   is also a sorted array in an ascending fashion.

Termination: By the principle of induction, $A[1:j-1]$ is sorted in an ascending manner before/after the j^{th} iteration, for any $j = 2, \cdots, A.length$. In other words, A remains sorted at termination.

Insertion Sort: Run-Time Analysis

- Let $n = A.length$.
- Let the time taken to execute the i^{th} line once be c_i, for all $i = 1, \cdots, n$.
- Let t_j denote the number of iterations taken by the *while* loop within the j^{th} iteration.

	INSERTION-SORT(A)	$cost$	$times$
1	**for** $j = 2$ **to** $A.length$	c_1	n
2	$key = A[j]$	c_2	$n-1$
3	// insert $A[j]$ into $A[1..j-1]$.	0	$n-1$
4	$i = j - 1$	c_4	$n-1$
5	**while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6	$A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7	$i = i - 1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8	$A[i+1] = key$	c_8	$n-1$

Insertion Sort: Run-Time Analysis

INSERTION-SORT(A)	$cost$	$times$
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // insert $A[j]$ into $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} t_j + c_6 \sum_{j=2}^{n}(t_j - 1) + c_7 \sum_{j=2}^{n}(t_j - 1) + c_8(n-1)$$

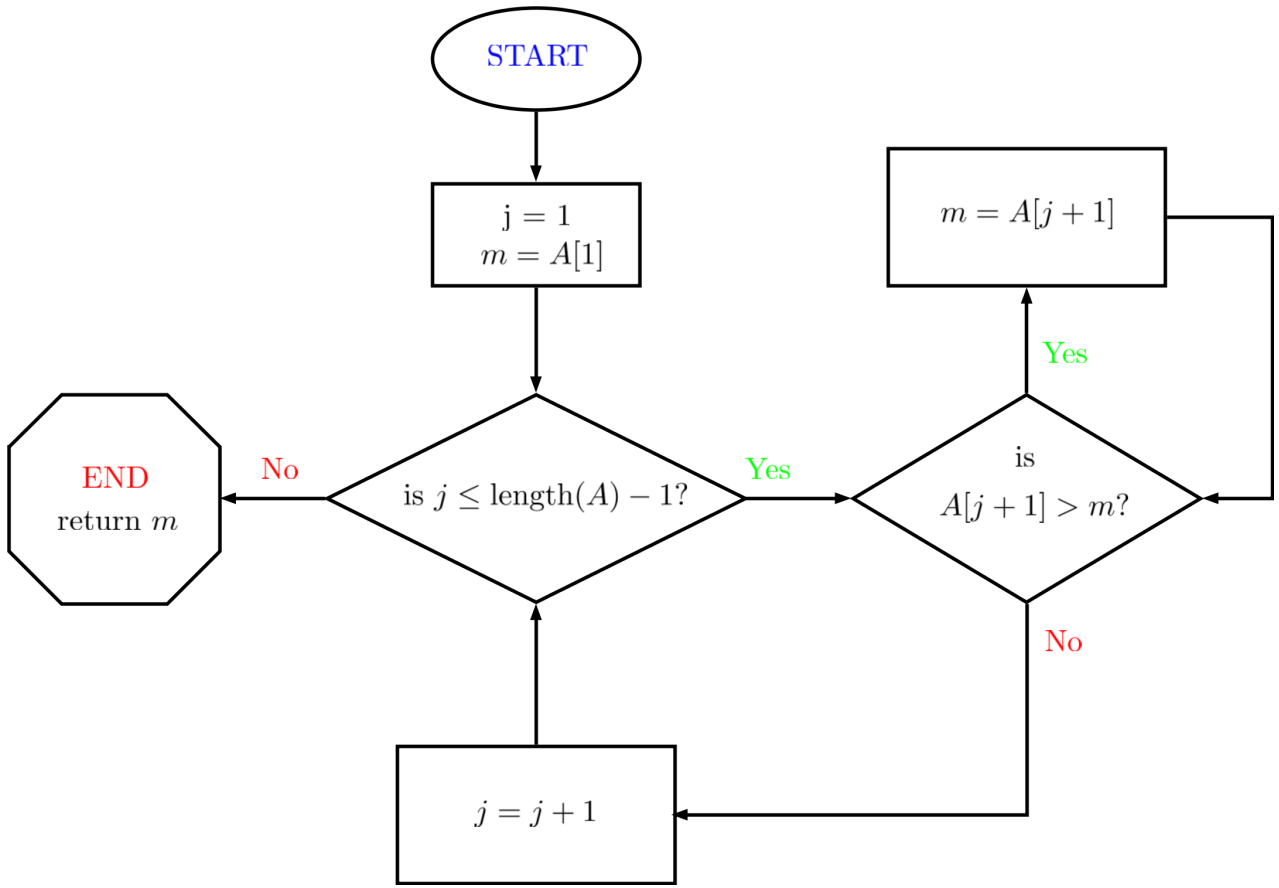
- Best case scenario? (Linear)
- Worst case scenario? (Quadratic)

Example 2: Find maximum element

Input: A array of n numbers, $A = \{a_1, \cdots, a_n\}$

Output: The maximum element in A, i.e. $m = \max(A)$.

Analysis: Pseudocode? Correctness? Run-time performance?



Order of Growth

Our focus: Worst-case running time.

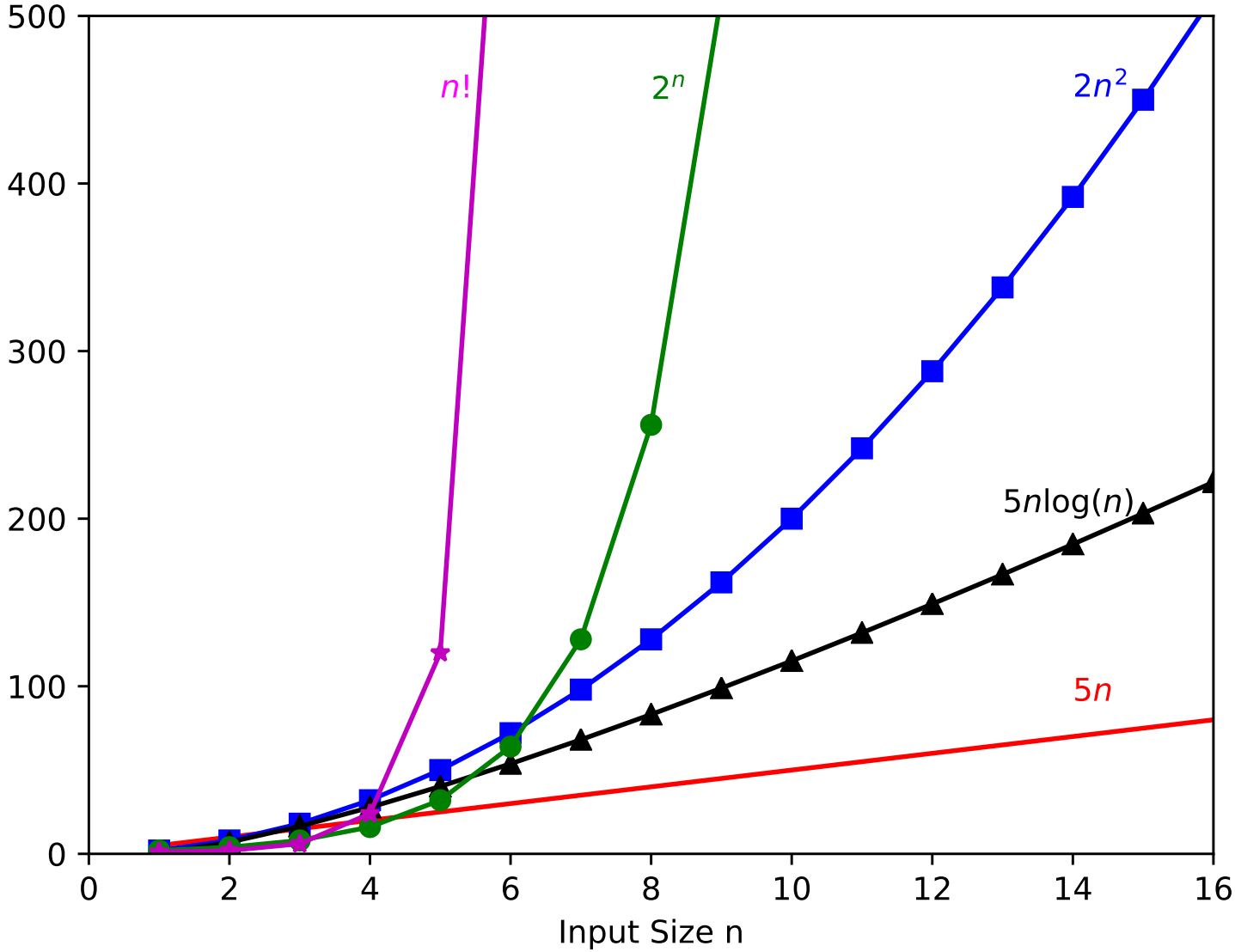
- Upper bound to the real instance.
- Average performance is typically comparable to worst-case performance. Why?

What if, the input size n is very large?

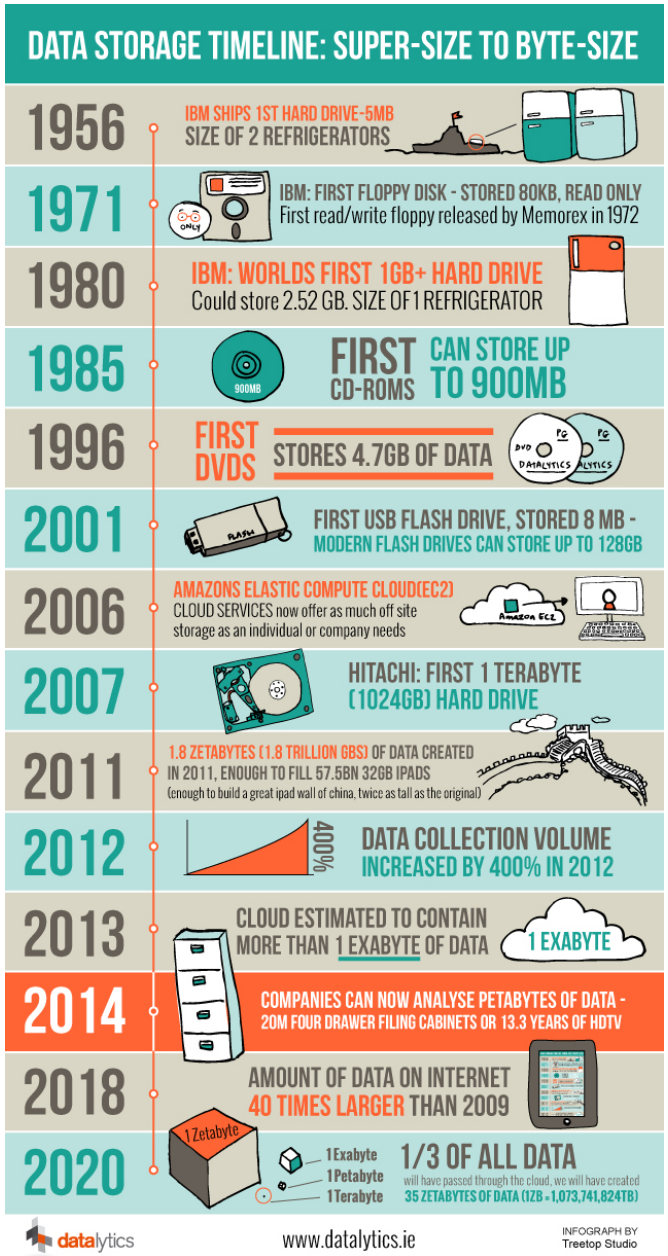
For example, the worst-case performance of insertion sort is of the form $an^2 + bn + c$.

- n^2 term grows rapidly, when compared to lower-order terms.
- We label this behavior as $\Theta(n^2)$.

Behavior of Some Important Functions



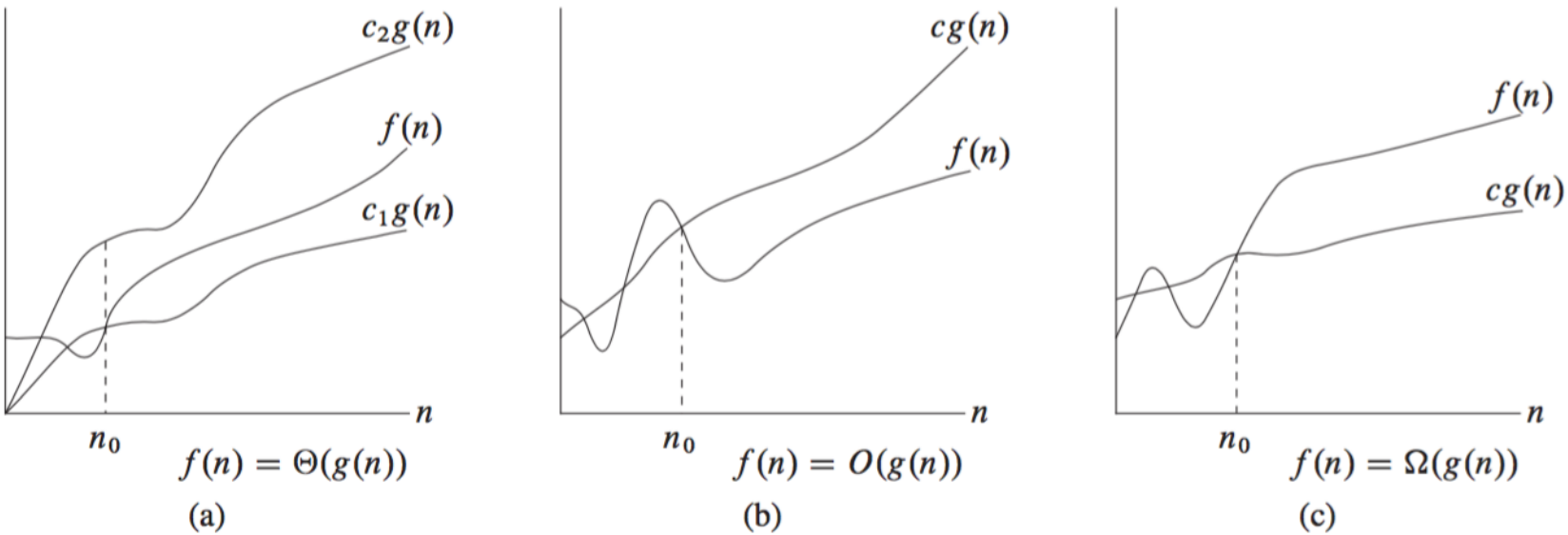
Data Size: Growth over Time



Growth of Functions

Three fundamental definitions:

- **Θ-notation:** Asymptotically tight bound
- **O-notation:** Asymptotic upper bound
- **Ω-notation:** Asymptotic lower bound



Source: Figure 3.1 in the textbook.

Formal Definitions

$$\Theta(g(n)) = \{f(n) \mid \text{there exists positive real constants } c_1, c_2 \text{ and a positive integer } n_0 \text{ such that, for all } n \geq n_0,\ 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\}.$$

$$O(g(n)) = \{f(n) \mid \text{there exists positive real constant } c \text{ and a positive integer } n_0 \text{ such that, for all } n \geq n_0,\ 0 \leq f(n) \leq c g(n)\}.$$

$$\Omega(g(n)) = \{f(n) \mid \text{there exists positive real constant } c \text{ and a positive integer } n_0 \text{ such that, for all } n \geq n_0,\ 0 \leq c g(n) \leq f(n)\}.$$

Theorem

For any two functions $f(n)$ and $g(n)$, we have $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Examples

Consider the function $f(n) = \frac{1}{2}n^2 - 3n$.

Exercise

Show that $f(n) = \Theta(n^2)$.

Exercise

Show that $f(n) \neq \Theta(n^3)$.

Some tricks of the trade:

- $an^2 + bn + c = an^2 + \Theta(n)$
- $O(1)$ is used to denote constants.

Asymptotically Weak Bounds

$$o(g(n)) = \{f(n) \mid \text{there exists positive real constant } c \text{ and a positive integer } n_0 \text{ such that, for all } n \geq n_0,\ 0 \leq f(n) < cg(n)\}.$$

$$\omega(g(n)) = \{f(n) \mid \text{there exists positive real constant } c \text{ and a positive integer } n_0 \text{ such that, for all } n \geq n_0,\ 0 \leq cg(n) < f(n)\}.$$

In other words, we have

- **Asymptotically smaller bound:** If $f(n) = o(g(n))$, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

- **Asymptotically larger bound:** If $f(n) = \omega(g(n))$, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

Comparing Functions

Transitivity:

- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.
- $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n))$, then $f(n) = \Omega(h(n))$.
- $f(n) = o(g(n))$ and $g(n) = o(h(n))$, then $f(n) = o(h(n))$.
- $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n))$, then $f(n) = \omega(h(n))$.

Reflexivity:

- $f(n) = \Theta(f(n))$.
- $f(n) = O(f(n))$.
- $f(n) = \Omega(f(n))$.

Symmetry:

- $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.

Comparing Functions (cont...)

Transpose Symmetry:

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
- $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

Trichotomy: Exactly one holds true: $a < b$, $a = b$, or $a > b$.

- Holds true for real numbers.
- Does not hold true for functions, in general.
- Example: The functions n and $n^{1+\sin n}$ cannot be compared.

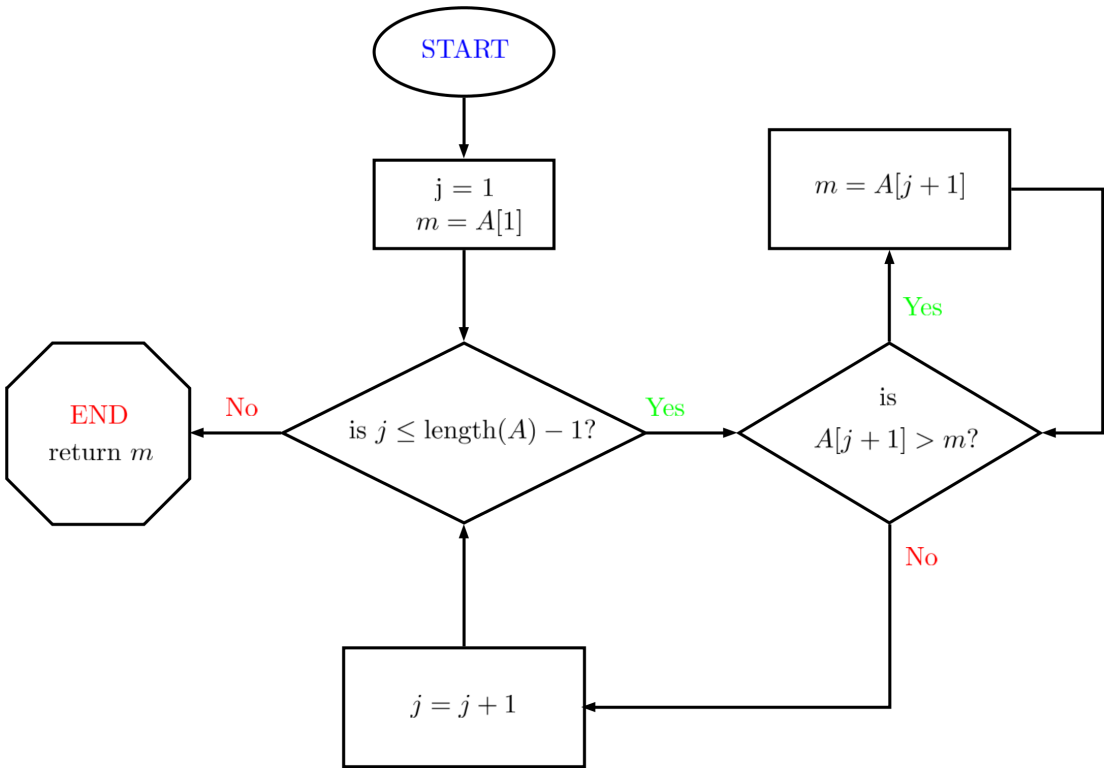
Example 2: Find maximum element

Input: A array of n numbers, $A = \{a_1, \cdots, a_n\}$

Output: The maximum element in A, i.e. $m = \max(A)$.

Loop-Invariant: $m = \max(A[1:j])$

Run-time Complexity: $O(n)$



Recursion

A function calling itself directly or indirectly.

Examples:

- **Factorial:** $n! = n \cdot (n-1)!$ for all $n = 1, 2, \cdots$
- **Fibonacci Numbers:** $F(n) = F(n-1) + F(n-2)$ for all $n = 3, 4, \cdots$ for a fixed $F(1)$ and $F(2)$.

Example 1: Factorial

```
FACTORIAL(n)
1  if n == 0
2        return 1
3  return n · FACTORIAL(n − 1)
```

Running Time: Assuming each line takes unit time to execute, we have

$$T(n) = \begin{cases} 2, & \text{if } n = 0 \\ T(n-1) + 2, & \text{if } n > 0. \end{cases}$$

$$= 2(n+1) = \Theta(n)$$

Example 2: Fibonacci

```
FIBONACCI(n)
1  if n == 0
2        return 0
3  if n == 1
4        return 1
5  return FIBONACCI(n − 1) + FIBONACCI(n − 2)
```

Running Time: Assuming each line takes unit time to execute, we have

$$T(n) = \begin{cases} 2, & \text{if } n = 0 \\ 3, & \text{if } n = 1 \\ T(n-1) + T(n-2) + 3, & \text{if } n > 1. \end{cases}$$

Example 2: Fibonacci

$$
\begin{aligned}
T(n) &= \begin{cases} 2, & \text{if } n = 0 \\ 3, & \text{if } n = 1 \\ T(n-1) + T(n-2) + 3, & \text{if } n > 1. \end{cases} \\
&= O(2^n)
\end{aligned}
$$

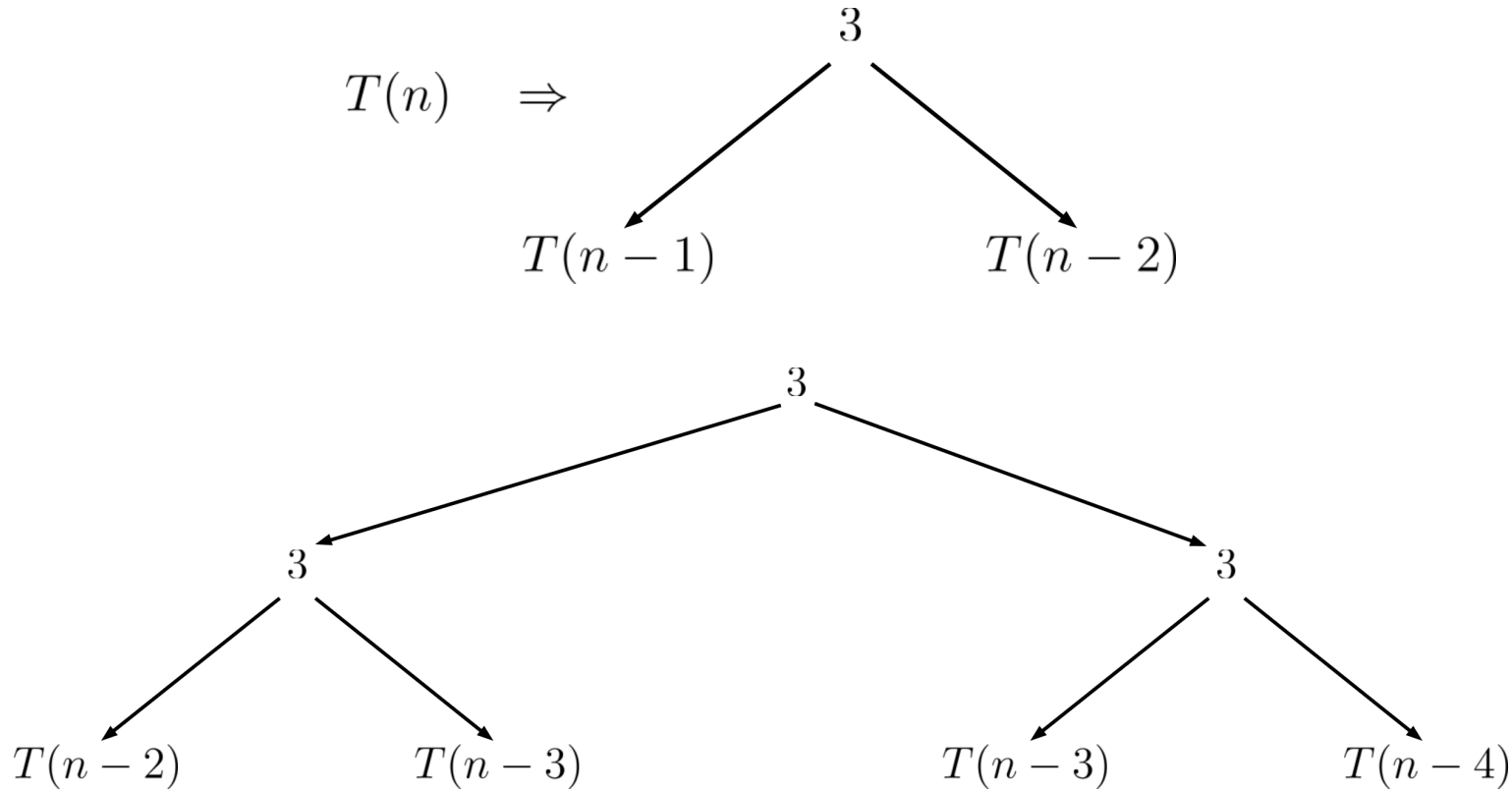
This technique is called *substitution method*.

Proof technique: Induction Principle.

Recursion Trees

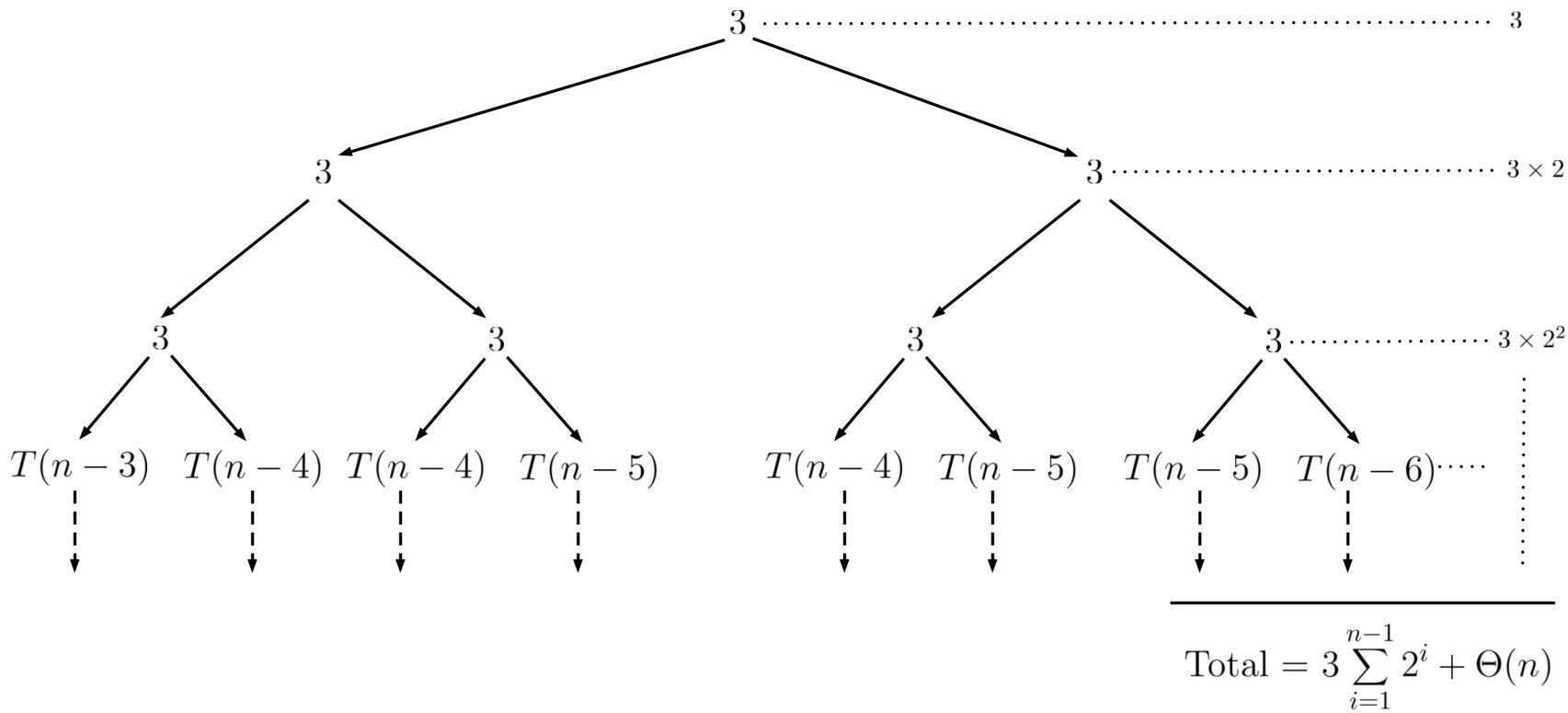
The drawback of substitution method is that it requires us to do some guesswork.

Fibonacci: $T(n) = T(n-1) + T(n-2) + 3.$



Recursion Trees

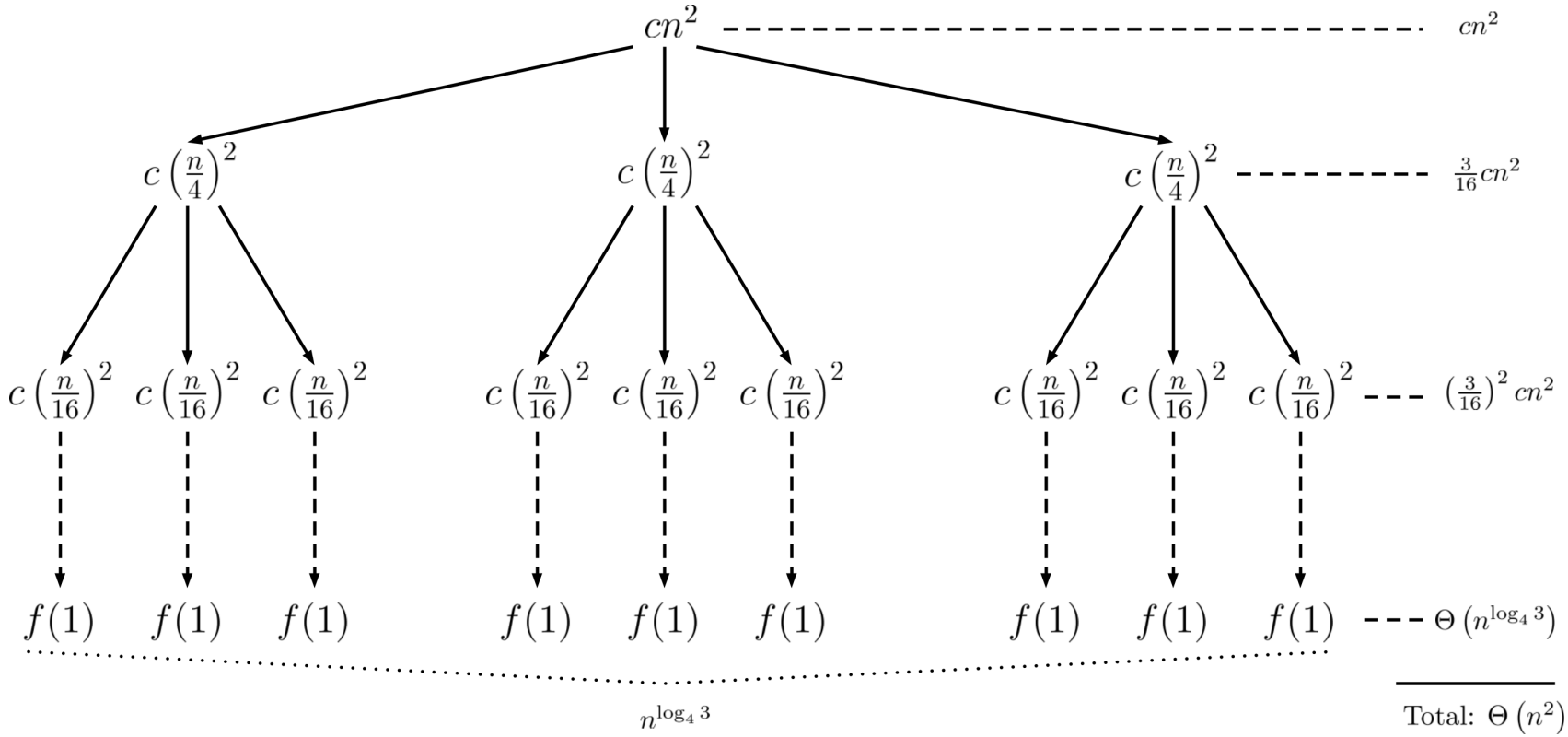
Fibonacci: $T(n) = T(n-1) + T(n-2) + 3.$



$$T(n) \quad = \quad \Theta(2^n) + \Theta(n) \quad = \quad \Theta(2^n)$$

Recursion Trees

Example 3: $f(n) = 3f(n/4) + cn^2$ for some constant $c > 0$.



$$
\begin{aligned}
f(n) &= \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \\
&< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta(n^{\log_4 3}) \;=\; \Theta(n^2)
\end{aligned}
$$

Master method

Theorem

Let $a \geq 1$ and $b > 1$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence

$$T(n) = aT(n/b) + f(n),$$

where we interpret n/b to mean either $\lceil n/b \rceil$ or $\lfloor n/b \rfloor$. Then, $T(n)$ has the following asymptotic bounds:

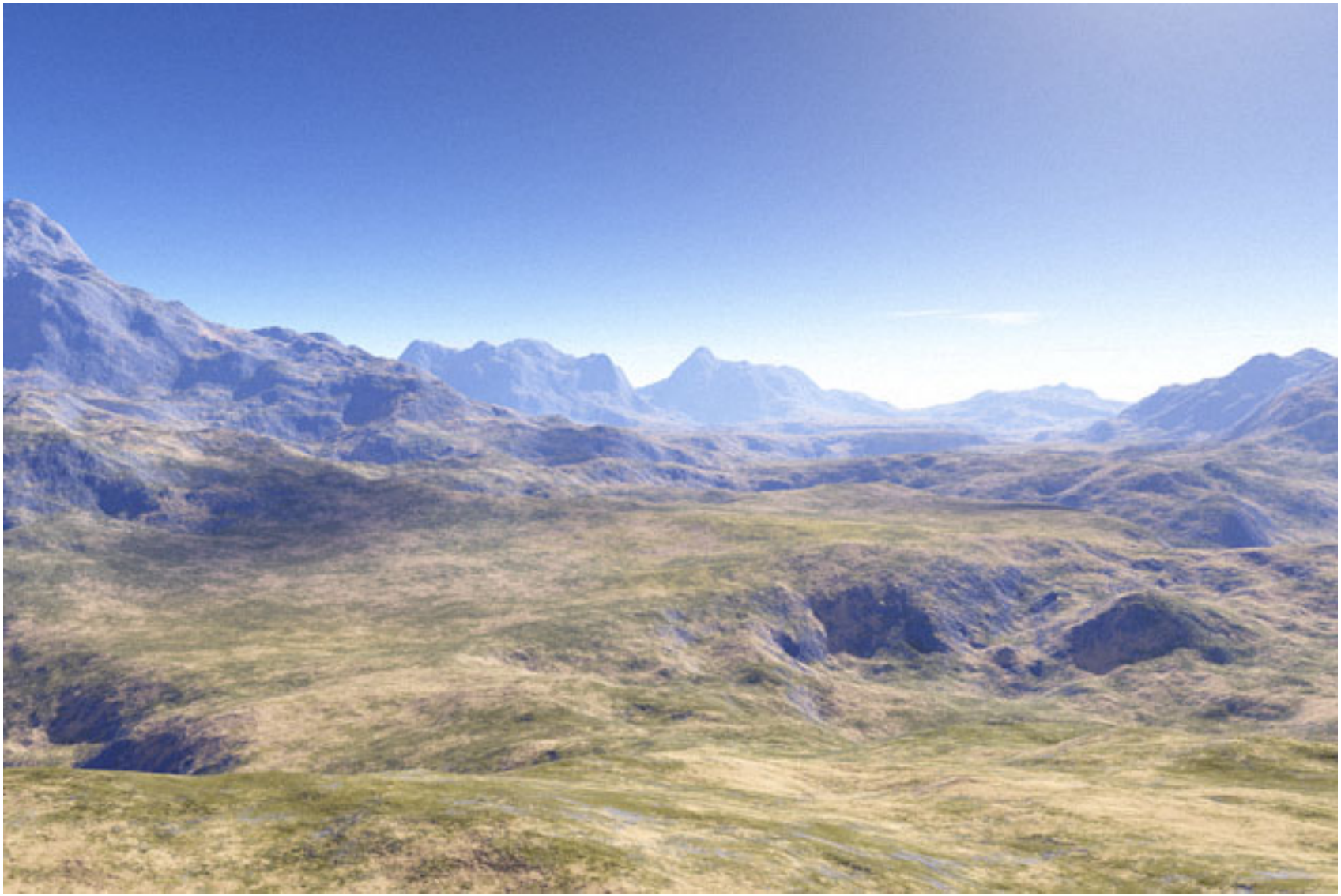
1. If $f(n) = O(n^{\log_b a - \epsilon})$ for some constant $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$.
2. If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
3. If $f(n) = O(n^{\log_b a + \epsilon})$ for some constant $\epsilon > 0$, and if $af(n/b) \leq cf(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$.

Practical Significance of Recursion

Example: Repeat simple shapes to obtain shapes that resemble real-world objects.

Source: https://www.cs.uic.edu/ jbell/CourseNotes/ComputerGraphics/images/fractalTrees.jpg

3-D Fractals for Computer Graphics



Source: https://upload.wikimedia.org/wikipedia/commons/6/6e/FractalLandscape.jpg

Design of Algorithms

Insertion sort uses an **incremental** design approach.

INSERTION-SORT(A)

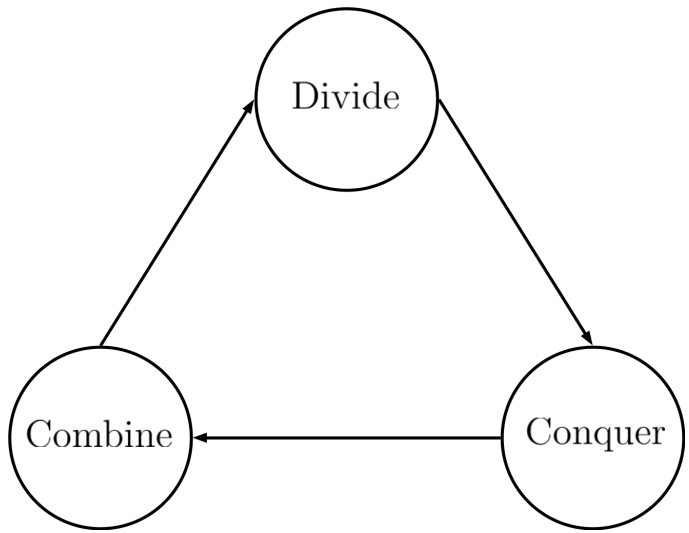
```
1  for   j = 2 to A.length
2        key = A[j]
3        //  insert A[j] into A[1..j − 1].
4        i = j − 1
5        while   i > 0 and A[i] > key
6              A[i + 1] = A[i]
7              i = i − 1
8        A[i + 1] = key
```

- Run-time Complexity: $\Theta(n^2)$.
- Can we do any better?

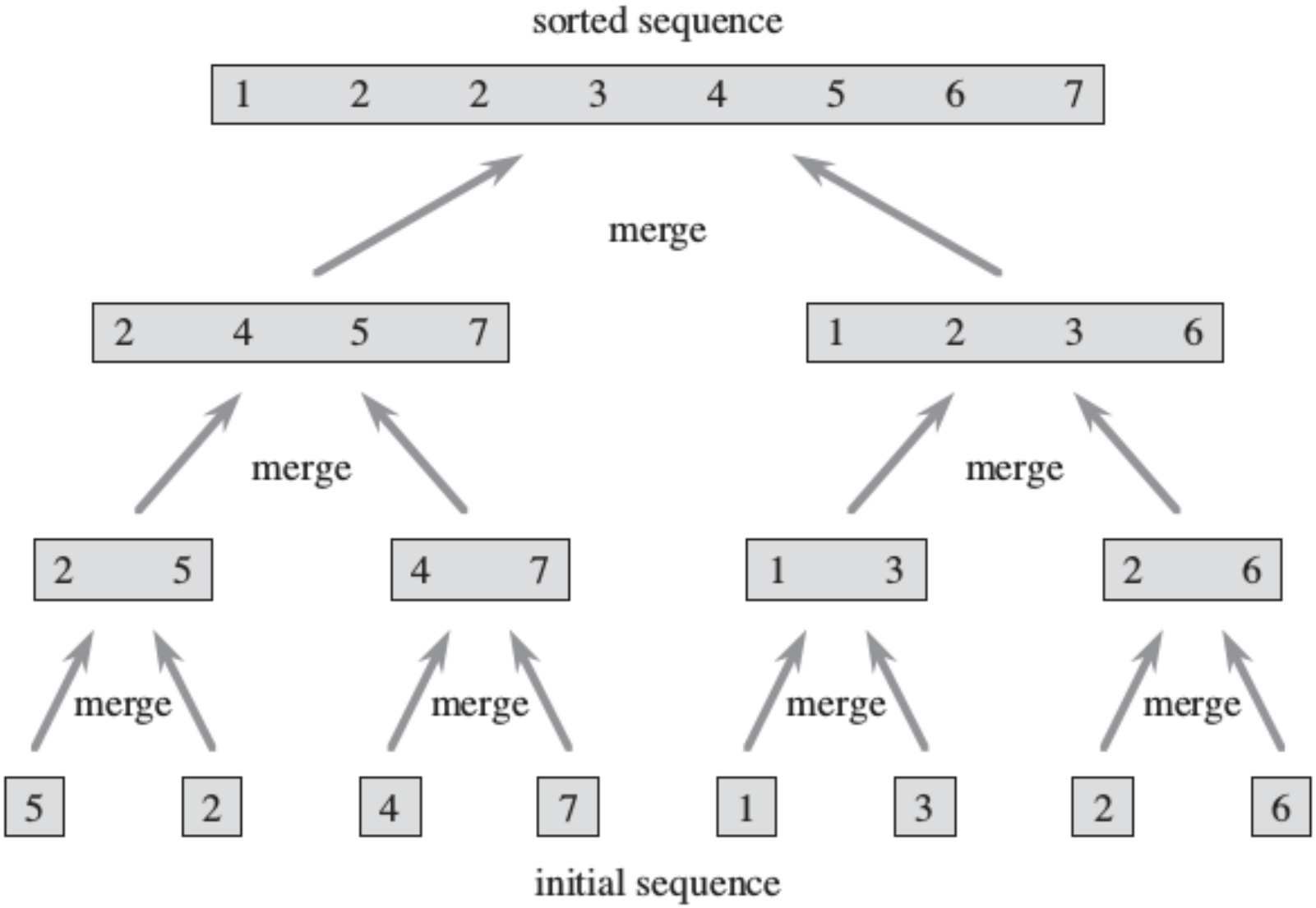
Divide and Conquer

A design approach inspired from recursion.

- **Divide** the problem into several subproblems, each being smaller instances of the original problem.
- **Conquer** the subproblems by solving them recursively.
- **Combine** the solutions to subproblems into the solution of the original problem.



Merge Sort



Source: Figure 2.4 in *CLRS*

Merge Sort

$\textsc{Merge-Sort}(A, p, r)$

```
1  if p < r
2        q = ⌊(p + r)/2⌋                  // Divide step
3        Merge-Sort(A, p, q)              // Conquer step
4        Merge-Sort(A, q + 1, r)          // Conquer step
5        Merge(A, p, q, r)                // Combine step
```

To sort the entire array A, call $\textsc{Merge-Sort}(A, 1, A.length)$.

For any indices p, q and r such that $p \leq q < r$, if both $A[p : q]$ and $A[q + 1 : r]$ are already sorted, how do we merge $A[p : q]$ and $A[q + 1 : r]$ into a single sorted array $A[p : r]$?

Merge Sort: Combine step

```
MERGE(A, p, q, r)
 1  n_1 = q − p + 1                    // Length of A[p : q]
 2  n_2 = r − q                        // Length of A[q + 1 : r]
 3  // Let L[1 : n_1 + 1] and R[1 : n_2 + 1] be two new arrays.
 4  for   i = 1 to n_1
 5        L[i] = A[p + i − 1]
 6  for   j = 1 to n_2
 7        R[j] = A[q + j]
 8  L[n_1 + 1] = ∞
 9  R[n_2 + 1] = ∞
10  i = 1
11  j = 1
12  for   k = p to r
13        if L[i] ≤ R[j]
14              A[k] = L[i]
15              i = i + 1
16        else
17              A[k] = R[j]
18              j = j + 1
```

Improvement in Run-time Complexity

- **Divide:** $\Theta(1)$
- **Conquer:** $2T(n/2)$
- **Combine:** $\Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1, \\ 2T(n/2) + \Theta(n) & \text{if } n > 1. \end{cases}$$

Using Master's theorem, we have $T(n) = \Theta(n \log n)$.

Matrix Multiplication

Given two $n \times n$ matrices A and B, find $C = AB$.

Traditional approach: $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, for all $i, j = 1, \cdots, n$.

Run-time Complexity: $\Theta(n^3)$.

Can we do any better?

Note: If both A and B are carved into four $n/2 \times n/2$ blocks, i.e., if $A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix}$ and $B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix}$, we have

$$C = \begin{bmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{bmatrix}.$$

Run-time Complexity

Does this help? **No!**

Run-time Complexity: $T(n) = 8T(n/2) + \Theta(n^2) = \Theta(n^3)$.

Can we do any better? **Yes!**

Idea: Instead of eight $n/2$-size products, reduce this to seven $n/2$-size products.

Strassen's Algorithm

Define

$$C = \begin{bmatrix} P_5 + P_4 - P_2 + P_6 & P_1 + P_2 \\ P_3 + P_4 & P_5 + P_1 - P_3 - P_7 \end{bmatrix}$$

where $P_1 = A_{11}S_1$, $P_2 = S_2B_{22}$, $P_3 = S_3B_{11}$, $P_4 = A_{22}S_4$, $P_5 = S_5S_6$, $P_6 = S_7S_8$ and $P_7 = S_9S_{10}$ and

$$\begin{array}{lcllcl} S_1 & = & B_{12} - B_{22} & S_6 & = & B_{11} + B_{22} \\ S_2 & = & A_{11} + A_{12} & S_7 & = & A_{12} - A_{22} \\ S_3 & = & A_{21} + A_{22} & S_8 & = & B_{21} + B_{22} \\ S_4 & = & B_{21} - B_{11} & S_9 & = & A_{11} - A_{21} \\ S_5 & = & A_{11} + A_{22} & S_{10} & = & B_{11} + B_{12} \end{array}$$

Run-time Complexity:

$$\begin{aligned} T(n) &= 7T(n/2) + \Theta(n^2) \\ &= \Theta(n^{\log 7}). \end{aligned}$$

Greatest Common Divisor

- $gcd(a, b)$ is the largest integer that divides both a and b.
- Example: $gcd(12, 20) = 4$
- If $gcd(a, b) = 1$, then a and b are relatively prime to each other.

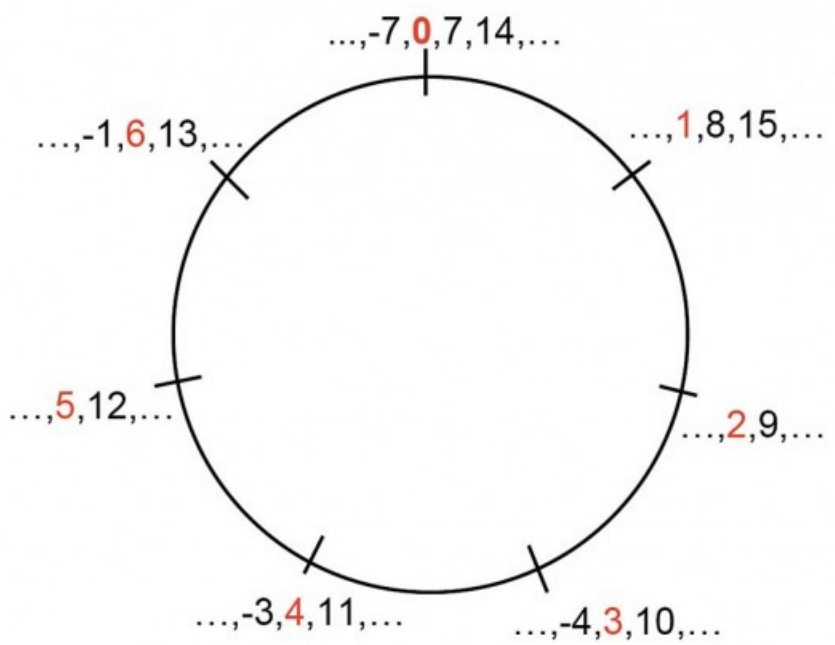
How to compute $gcd(a, b)$?

Prime Factorization:

- Find the prime factors of a and b, and save the common factors.
- Example: $gcd(18, 45) = gcd(2 \cdot 3^2, 3^2 \cdot 5) = 3^2 = 9$.
- **Hard problem:** No polynomial-time algorithm.
- Example: Twin Primes

Euclid's Rule

Euclid's rule: If x and y are positive integers such that $x \geq y$, then $gcd(x, y) = gcd(x \mod y, y)$.
Modular arithmetic: A cyclic representation of numbers within a finite range $\{0, 1, \cdots, n-1\}$, as in a clock.



- If $x = qn + r$ with $0 \leq r < n$, then $x \mod n = r$.
- $x \equiv y \pmod n \iff n$ divides $(x - y)$.

Euclid's Algorithm

Consider two positive numbers a and b such that $a \geq b$.

EUCLID-GCD(b, a)

```
1  if b = 0
2        return a
3  return EUCLID-GCD(b, a mod b)
```

Theorem

If $a \geq b$, then $a \mod b < a/2$.

In other words, if a and b are n-bit integers, it takes $2n$ recursive calls to reach the base case. As a result, we have $O(n^3)$, since conversion from integer system to modular arithmetic system requires division, which is $O(n^2)$.

Randomization

Randomization can manifest in two different ways:

- Random Input (Probabilistic Analysis)
- Random Algorithm (Designs with Randomness)

Let us first build some preliminary background in probability.

Introduction to Probability

- **Random Experiment:** A procedure whose outcome cannot be predicted with certainty; e.g. Toss a fair coin, and report heads or tails.
- **Sample Space:** Set of all possible outcomes; e.g. $\Omega = \{H, T\}$.
- **Probability:** Measure of chance; e.g. $P(H) = P(T) = \frac{1}{2}$.

Types of Experiments:

- **Discrete:** Finite (or, countably infinite) list of possibiities
- Continuous: Uncountably infinite list of possibilities

Probability Rules

Notation:

- Sample space: $\Omega = \{\omega_1, \omega_2, \cdots, \omega_n\}$
- Outcome: $\omega \in \Omega$
- Probability of an event A: $P(A)$

Properties:

- $0 \leq P(A) \leq 1$, $P(\Omega) = 1$, $P(A^c) = 1 - P(A)$.
- Disjoint events: If A and B are disjoint events, $P(A \cup B) = P(A) + P(B)$.
- Inclusion-Exclusion: For any two events A and B, $P(A \cup B) = P(A) + P(B) - P(A \cap B)$.
- Conditional Probability: $P(A|B) = \dfrac{P(A \cap B)}{P(B)}$
- $P(A|B) = P(A)$ if A and B are independent events.

Random Variable

A mapping of an event to some real number.

$$X : \Omega \to \mathbb{R}$$

Example 1: In a coin-toss experiment, let

$$X = \begin{cases} 1, & \text{if heads} \\ -1, & \text{if tails.} \end{cases}$$

Example 2: Indicator function of an event A is defined as

$$1_A(x) = \begin{cases} 1, & \text{if } x \in A \\ 0, & \text{otherwise.} \end{cases}$$

"$X = a$" denotes the event $\{\omega \in \Omega | X(\omega) = a\}$.

Expectation

Average value of a random variable.

If $X \in \{x_1, \cdots, x_n\}$ and $p_i = P(X = x_i)$, then

$$\mathbb{E}(X) = \sum_{i=1}^{n} p_i x_i$$

Example 1: $\mathbb{E}(X) = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot (-1) = 0.$

Example 2: $\mathbb{E}\left[1_A(X)\right] = \sum_{i \in A} p_i \cdot 1 + \sum_{i \notin A} p_i \cdot 0 = \sum_{i \in A} p_i.$

Algorithm Analysis: Random Inputs

Consider $\textsc{Insertion-Sort}(A)$.

Given an array $A = [a_1, \cdots, a_N]$, let (i, j) denote a swap in A if $a_i > a_j$ for any $i < j$.

Example: $A = [3,\ 2,\ 1,\ 4,\ 5]$.

All possible swaps: $(1, 2)$, $(1, 3)$, $(2, 3)$.

Minimum number of swaps: 0

Maximum number of swaps: $\binom{n}{2}$

The goal of sorting is to remove all swaps.

Counting swaps in Insertion-Sort

Let

$$S_A(i,j) = \begin{cases} 1, & \text{if } (i,j) \text{ is a swap in } A. \\ 0, & \text{otherwise.} \end{cases}$$

The total number of swaps in the array A are given by

$$S_A = \sum_{i<j} S_A(i,j).$$

Therefore, the average number of swaps is

$$\mathbb{E}(S_A) = \sum_{i<j} \mathbb{E}(S_A(i,j)).$$

If $P(S_A(i,j)) = \frac{1}{2}$, then

$$\mathbb{E}(S_A(i,j)) = P(S_A(i,j)) \cdot 1 + [1 - P(S_A(i,j))] \cdot 0 = \frac{1}{2}.$$

Insertion Sort: Average Run-Time Analysis

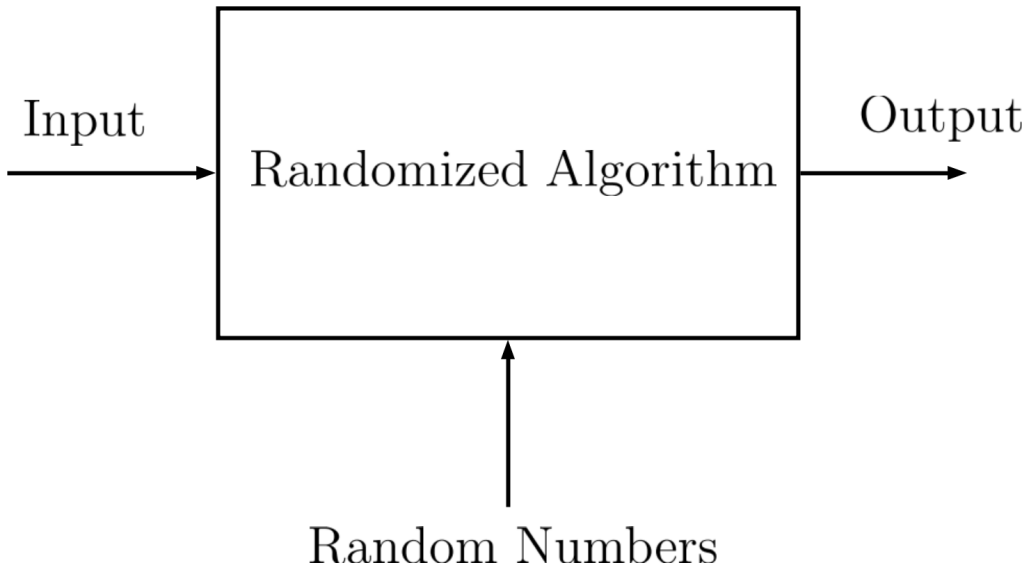
INSERTION-SORT(A)	$cost$	$times$
1 **for** $j = 2$ **to** $A.length$	c_1	n
2 $key = A[j]$	c_2	$n-1$
3 // insert $A[j]$ into $A[1..j-1]$.	0	$n-1$
4 $i = j-1$	c_4	$n-1$
5 **while** $i > 0$ and $A[i] > key$	c_5	$\sum_{j=2}^{n} t_j$
6 $A[i+1] = A[i]$	c_6	$\sum_{j=2}^{n}(t_j - 1)$
7 $i = i-1$	c_7	$\sum_{j=2}^{n}(t_j - 1)$
8 $A[i+1] = key$	c_8	$n-1$

$$\mathbb{E}[T(n)] = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{j=2}^{n} \mathbb{E}(t_j) + c_6 \sum_{j=2}^{n} (\mathbb{E}(t_j) - 1)$$
$$+ c_7 \sum_{j=2}^{n} (\mathbb{E}(t_j) - 1) + c_8(n-1)$$

where $\mathbb{E}(t_j) = \sum_{i<j} \mathbb{E}[S_A(i,j)] = \frac{j}{2}$.

Randomized Algorithms

Randomized algorithms uses a source of random numbers to make random choices during execution.



Applications:

- Security: Random key generators.
- Testing: Monte Carlo simulations (Output is not correct with some probability, as opposed to Las Vegas algorithms.)

Randomly Permuted Arrays

Produce a random permutation of the array.

Let us consider the following example:

PERMUTE-BY-SORTING(A)

1 $n = A.length$
2 // let $P[1:n]$ be a new array.
3 **for** $i = 1$ **to** n
4 $P[i] = \text{RANDOM}(1, n^3)$ // n^3 chosen for uniqueness.
5 sort A using P as sort keys.

Example: Let $A = [1,\ 2,\ 3,\ 4]$ and let the random priorities be $P = [36,\ 3,\ 62,\ 19]$. Then the output is $A' = [2,\ 4,\ 1,\ 3]$.

Uniform Random Permutation

Produce every permutation of the numbers 1 *through* n*, equally likely.*

Let E_i denote an event where $A[i]$ receives the i^{th} smallest priority. Then,

$$\begin{aligned} P(E_1 \cap \cdots \cap E_n) &= P(E_1) \cdot P(E_2|E_1) \cdots P(E_n|E_1 \cap \cdots \cap E_{n-1}) \\ &= \frac{1}{n} \cdot \frac{1}{n-1} \cdots 1 \\ &= \frac{1}{n!} \end{aligned}$$

Works the same way for any permutation $\sigma = \{\sigma(1), \cdots, \sigma(n)\}$ of the set $\{1, 2, \cdots, n\}$. Just define E_i as an event where $A[i]$ receives the $\sigma(i)^{th}$ priority.

Summary

- Algorithms: Pseudocodes and Flowcharts
- Analysis: Correctness and Run-time
- Asymptotic Notation
- Recursion: Recursion Trees, Master Theorem
- Divide and Conquer
- Randomization