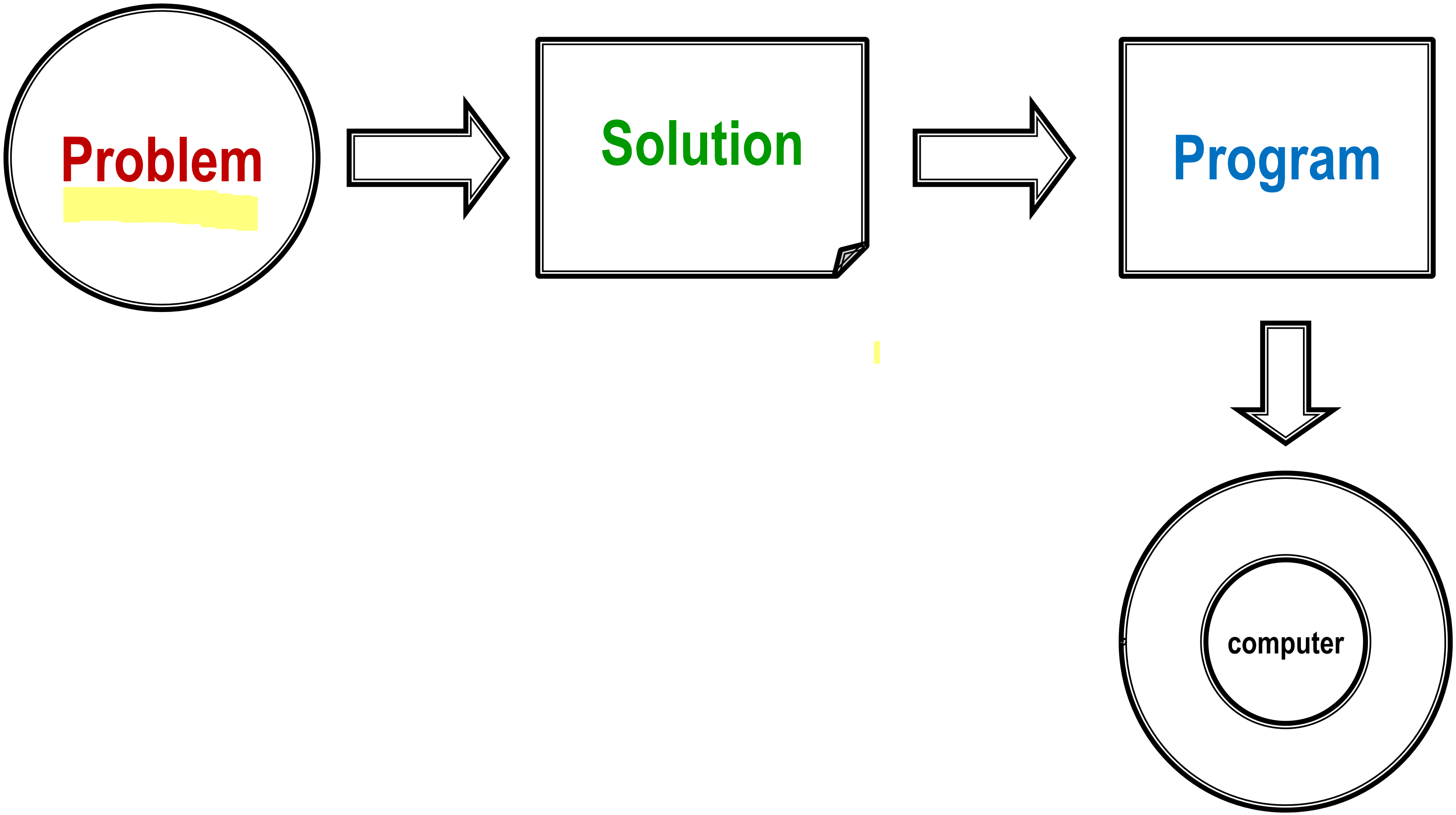
# Algorithms: Efficiency and Analysis

# Problem Solving

# The Problem Solving Process via Programs



# Computational Problem

- A problem is a question to which we seek an answer.
  - Example 1.1 sorting
  - Example 1.2 searching
- An instance of the problem.
  - Example 1.3
  - Example 1.4
- **Computational problems**

# ▶ QUIZ? Computing Problems & Instances?

- A computing problem ?
- An instance of the problem.

- The Selection Problem?
- An instance of the selection problem?

# Fundamental Computational Problems

- The sorting problem
- The searching problem
- The selection problem
- ...

# Solution

- Input (and store) **data.**
- **Process** (manipulate) data. (**algorithms**)
- Output **data**.

# Solution: Data Structures

- A simple data
  - Consists of atomic data items (values).
- A structured data
  - Consists of collections of data items (values), all related to each other in certain ways.
  - A particular way of storing and organizing data so that it can be used efficiently.
  - **Data Structures**

# Basic Data Structures

- Arrays
- Linked lists
- Stacks
- Queues
- Binary heaps
- Hash tables
- Binary trees
- Binary search trees

# Advanced Data Structures

- Skip lists
- Self-organizing lists
- Graphs
- Leftist heaps
- Skew heaps
- AVL trees
- Splay trees
- 2-3 trees
- 2-3-4 trees
- Red-Black trees
- B-trees

# Solution: Algorithms

- **Operations** that manipulate the data items in the data structures. (mini-algorithms!)
- **Algorithms** that use these operations.
  - A general step-by-step procedure for producing the solution to each instance of a problem.
  - Example 1.5 Sequential Search

- Efficient data structures are a key to designing efficient algorithms!

# ✓ Algorithm Design Approach/Paradigms

- Divide-and-Conquer
- Dynamic Programming
- Greedy Approach
- Backtracking
- Branch-and-Bound

# Solution = DS + Algorithms

- Efficient Solution =
  - Efficient Data Structures +
  - Efficient Operations +
  - Efficient Algorithms

- **QUIZ:** Does every problem have an algorithm?

# ✓ Computational Complexity for Problems

- The study of **all** possible algorithms that can solve a given problem.
- Determines a **lower bound** on the efficiency of all algorithms for a given problem.
- Lower Bound for fundamental problems:
  - The sorting problem
  - The searching problem
  - The selection problem

# ▶ QUIZ? Lower Bound?

- The sorting problem?
  - …
- The searching problem?
  - …
- The selection problem?
  - …

# ✓ Computationally Tractable & Intractable Problems

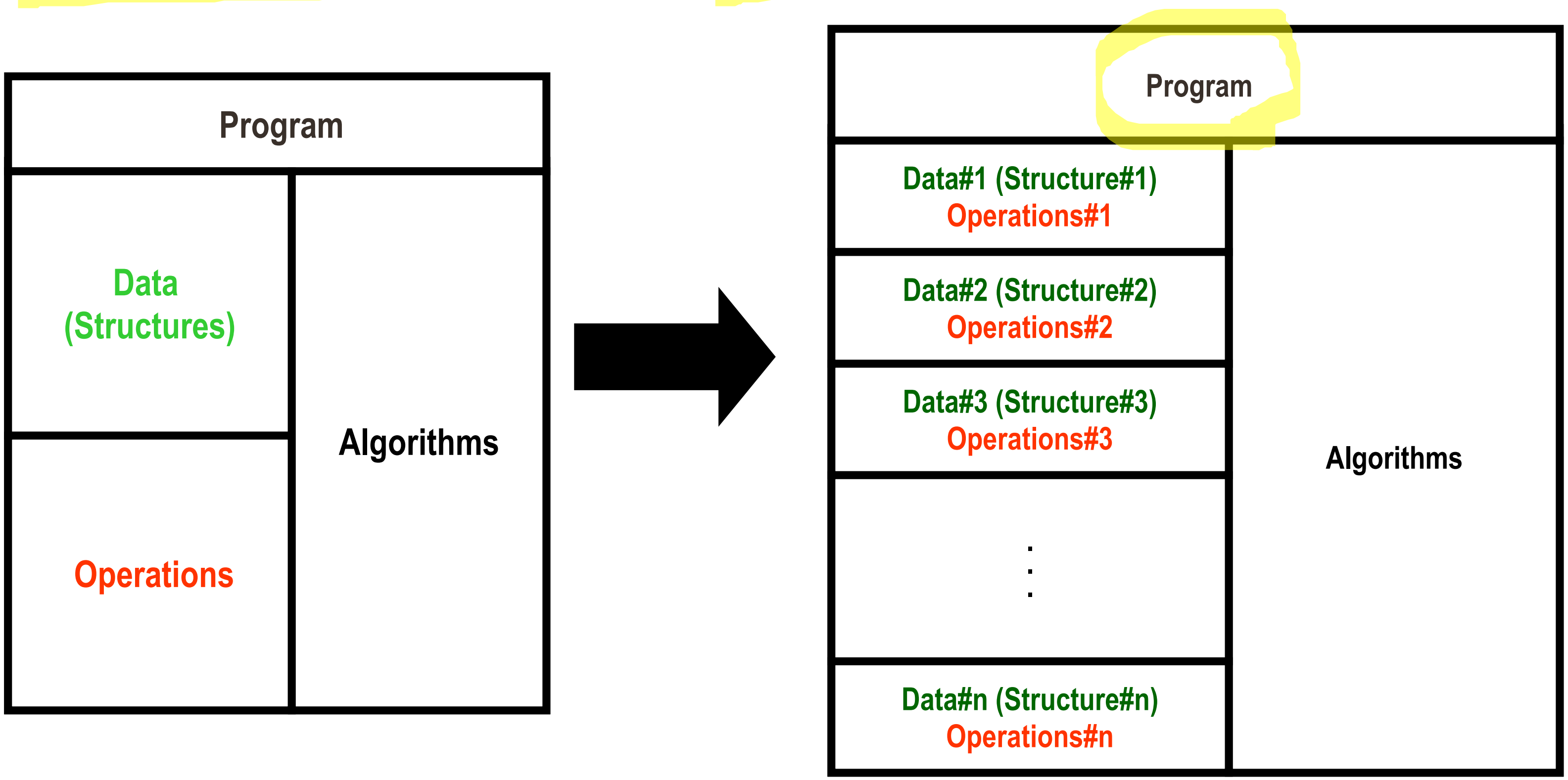
- Tractable Problems
  - A problem is called **tractable** if an efficient (**polynomial running time P**) algorithm is possible.
    - The sorting problem
    - The searching problem
    - The selection problem
    - ...
- Intractable (Hard) Problems
  - A problem is called **intractable** if an efficient algorithm is not possible.
    - **NP and NP-complete problems**
    - ...

# ▶ QUIZ? Tractable & Intractable Problem **Examples**

- Tractable Problems?
  - …
- Intractable (Hard) Problems?
  - …

# Program=Data Structures + Algorithms

- An implementation of a solution (design)!



# Algorithms: Efficiency

# Algorithms

- Correctness
- Efficiency

# Types of Algorithms?

- **Iterative** algorithms
  - Using **iteration**
- **Recursive** algorithms
  - Using **recursion**

# Example: Algorithms

- **Algorithm 1.1 Sequential Search - Iterative**
- Algorithm 1.2 Add Array Members
- **Algorithm 1.3 Exchange Sort**
- **Algorithm 1.4 Matrix Multiplication**
- **Algorithm 1.5 Binary Search – Iterative**

# Search - Sequential Search?

- Algorithm 1.1 Sequential Search
  - Linear time O(N)
- How to improve?
  - **Idea?**
  - **Sorting** + Search
  - Divide-and-Conquer
  - Top-down

# Search - Binary Search?

- **Algorithm 1.5 Binary Search – Iterative**
  - Divide-and-Conquer
  - Top-down
  - O(log N)

# Search - Sequential Search vs Binary Search

- Comparison of Binary Search and Sequential Search:
  - Table 1.1 Comparison

# ▶QUIZ?

- Write an iterative **Sequential Search** algorithm?
  - Algorithm 1.1
- Write an iterative **Binary Search** algorithm?
  - Algorithm 1.5

# The Fibonacci Sequence

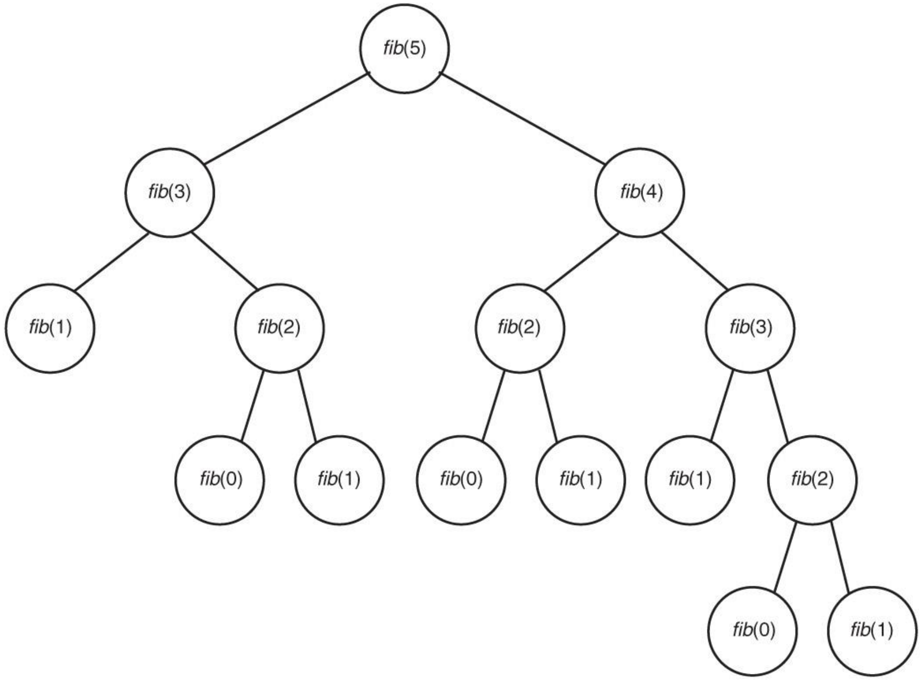
- Fib(n) = Fib(n – 1) + Fib(n – 2) for n >= 2;
- Fib(0) =0
- Fib(1) = 1

# The Fibonacci Sequence – Recursive?

- Algorithm 1.6 nth Fibonacci Term (Recursive)
  - **Divide-and-Conquer**
  - Top-down
  - Exponential time $O(2^N)$

# The Fibonacci Sequence - Recursive

- fib(5)?
  - Fig 1.2



# The Fibonacci Sequence - Recursive

- fib(n)?
  - Table
- T(n) = The number of terms in the recursion tree
  - Theorem 1.1
- This algorithm is extremely inefficient.
- How to improve?
- **Idea?**
  - **Dynamic Programming**
  - **Bottom-up**

# The Fibonacci Sequence – Iterative?

- Algorithm 1.7 nth Fibonacci Term (Iterative)
  - **Dynamic Programming**
  - Bottom-up
  - Linear time O(N)

# The Fibonacci Sequence Algorithms

- Comparison of Algorithm 1.6 nth Fibonacci Term (Recursive) and Algorithm 1.7 nth Fibonacci Term (Iterative):
  - Table 1.2 Comparison
  - Algorithm 1.6 nth Fibonacci Term (Recursive)
    - Divide-and-Conquer
    - Top-down
  - Algorithm 1.7 nth Fibonacci Term (Iterative)
    - Dynamic Programming
    - Bottom-up

# ✓Algorithm Efficiency

- Measure the efficiency of an algorithm in terms of
  - **Time** (required for an algorithm)
  - **Space** (required for a data structure)
- **Complexity Theory!**
  - Time complexity
  - Space complexity

# ✓ Analysis of Algorithms

- Algorithm analysis measures the **efficiency** of an algorithm as a function of the input size.
- We want a measure that is independent of
  - the computer,
  - the programming language,
  - the programmer, and
  - all the complex details of the algorithm.

# ✓ Time Complexity Analysis

- In general, the running time of the algorithm increases with the size of the input.
- The total running time is proportional to how many times some basic operation is done.
- Therefore, we analyze an algorithm's efficiency by determining
  - **the number of some basic operation as a function of the size of the input.**

# Basic Operation

- A time complexity analysis determines how many times the basic operation is done for each value of the input size.
- There is no hard and fast rule for choosing the basic operation.
  - It is largely a matter of judgment and experience.

# ✓ Time Complexity Cases

- Every Case Time
- Worst Case Time
- Average Case Time
- Best Case Time

# Every-Case Time Complexity?

- **T(n)** = the number of times the algorithm does the basic operation for an instance of size n.
  - Called the **every-case time complexity** of the algorithm.
- Analysis of Algorithm 1.2 (Add Array Members)
- Analysis of Algorithm 1.3 (Exchange Sort)
- Analysis of Algorithm 1.4 (Matrix Multiplication)

# Worst-Case Time Complexity?

- **W(n)** = the **maximum** number of times the algorithm will ever do its basic operation for an input size of n.
  - Called the **worst case time complexity** of the algorithm.
- Analysis of Algorithm 1.1 (Sequential Search)

# Average (Expected)-Case Time Complexity?

- **A(n)** = the **average (expected)** value of the number of times the algorithm does the basic operation for an input size of n.
  - Called an **average-case time complexity** analysis.
- Analysis of Algorithm 1.1 (Sequential Search)

# Best-Case Time Complexity?

- **B(n)** = the **minimum** number of times the algorithm will ever do its basic operation for an input size of n .
  - Called the **best-case time complexity** of the algorithm.
- Analysis of Algorithm 1.1 (Sequential Search)

# ✓ Asymptotic Behavior?

- The behavior of an algorithm for **very large problem sizes**.
  - How quickly the algorithm's time/space requirement grows as a function of the problem size?
- Measure the efficiency of an algorithm as a **growth rate function** of the algorithm.
  - **An estimating technique!**
  - **But, proved to be useful!**

# ▶QUIZ?

- Why not the exact time behavior of an algorithm?

# Asymptotic Order of Growth

- The asymptotic running time of the algorithm A for the problem size n: $\text{GrowthRateTime}_A(n)$

The (Asymptotic) Efficiency of an Algorithm =
A Growth Rate of the Function of the Problem Size

How it grows?

# ▶QUIZ?

- Types of algorithms?
- The time complexity (running time) of an algorithm?
- Time complexity cases?
- Asymptotic behavior?
- Why not the exact behavior of an algorithm?

# ✓ Notations for Asymptotic Behavior?

- For asymptotic upper bound
  - **Big-O**
- For asymptotic lower bound
  - **Big-Ω**
- **Big-Θ**

# Time Complexity Bounds

- Upper Bound
- Lower Bound

# Asymptotic Upper Bound

- An asymptotic bound as function of the size of the input, on the worst (slowest, most amount of space used) an algorithm will take to solve a problem.
  - No input will cause the algorithm to use more resources than the bound.

# ✓ Big-O Notation for (Asymptotic) Upper Bound?

- Let f(n) be a function which is non-negative for all integers $n \geq 0$.

$$f(n) = \mathbf{O}\,(g(n))$$

$$f(n) \in \mathbf{O}\,(g(n))$$

"f(n) is **big-oh** g(n)"

if

there exist a constant $c > 0$ and a constant $n_0$

such that

$f(n) \leq c * g(n)$ for all integers $n \geq n_0$.

# Conventions for Big-O Expressions

- Drop all but the most significant terms.
  - $O(n^2 + n \log n + n + 1) \rightarrow O(n^2)$
  - $O(n \log n + n + 1) \rightarrow O(n \log n)$
  - $O(n + 1) \rightarrow O(n)$
- Drop constant (usually small!) coefficients.
  - $O(2 n^2) \rightarrow O(n^2)$
  - $O(1024) \rightarrow O(1)$

**What dominates?**

# Example: Big-O

- $\log n = O(n)$
- $n = O(n)$
- $100\,n + 10 \log n = O(n)$
- Example 1.7
- Example 1.8
- Example 1.9
- Example 1.10
- Example 1.11

# Algorithm Growth Rates

- A constant growth rate O(1)
- A logarithmic logarithmic growth rate (log (log N))
- A logarithmic growth rate (log N)
- A logarithmic squared growth rate ($\log^2 N$)
- A linear growth rate O(N)
- A linear-logarithmic (?) growth rate O(N log N)

# Algorithm Growth Rates

- A quadratic growth rate $O(N^2)$
- A cubic growth rate $O(N^3)$
- A polynomial growth rate $O(N^k)$ for a constant k.
- An exponential growth rate $O(2^N)$
- A factorial growth rate $O(N!)$

# Common Complexity Classes

- $O(1)$
- $O(\log N)$
- $O(N)$
- $O(N \log N)$
- $O(N^2)$
- $O(2^N)$

# O(1) - Constant Growth Rates

- The amount of space or time is independent of the amount of data.
- If the amount of data doubles, the amount of space or time will stay the same!
- Example:
  - An item can be added to the beginning of a linked list in constant time independent of the number of items in the list.

# O(log N) - Logarithmic Growth Rates

- If the amount of data doubles, the amount of space or time will increase by 1!
- Example:
  - The worst-case time for binary search is logarithmic in the size of the array.

# O(N) - Linear Growth Rates

- If the amount of data doubles, the amount of space or time will also double!
- Example:
  - The time needed to print all of the values stored in an array is linear in the size of the array.

# O($N^2$) - Quadratic Growth Rates

- If the amount of data doubles, the amount of space or time will quadruple!
- Example:
  - The amount of space needed to store a two-dimensional square array is quadratic in the number of rows.

# $O(2^N)$ - Exponential Growth Rates

- If the amount of data increase by 1, the amount of space or time will double!
- Example:
  - The number of moves required to solve the Towers of hanoi puzzle is exponential in the number of disks used.

# ▶ QUIZ? Common Complexity Classes **Examples**

- $O(1)$
- $O(\log N)$
- $O(N)$
- $O(N \log N)$
- $O(N^2)$
- $O(2^N)$

# Comparison of Growth Rates

$O(N!)$

$O(2^N)$

$O(N^3)$

$O(N^2)$

$O(N \log N)$

$O(N)$

$O(\log^2 N)$

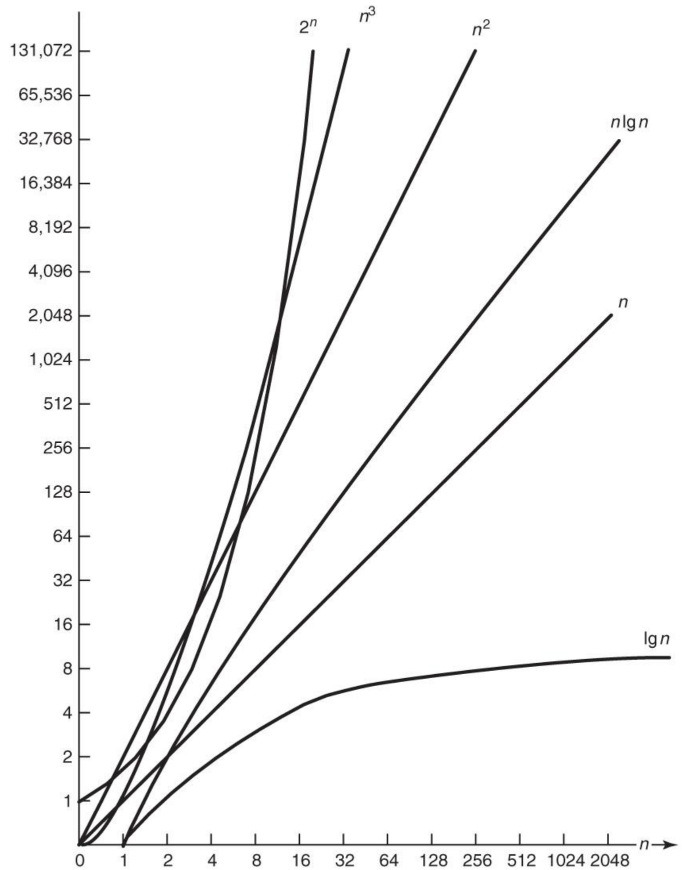
$O(\log N)$

$O(\log (\log N))$

$O(1)$

# Comparison of Growth Rates

- Figure 1.3



# Comparison of Growth Rates

- Table 1.3
- Table 1.4

# Asymptotic Lower Bound

- An asymptotic bound as function of the size of the input, on the best (fastest, least amount of space used) an algorithm will take to solve a problem.
  - No algorithm can use fewer resources than the bound.

# ✓Big-Ω Notation for (Asymptotic) Lower Bounds?

- Let f(n) be a function which is non-negative for all integers $n \geq 0$.

$f(n) = \Omega(g(n))$

$f(n) \in \Omega(g(n))$

"f(n) is **(big-)omega** g(n)"

if

there exist a constant $c > 0$ and a constant $n_0$

such that

$c * g(n) \leq f(n)$ for all integers $n \geq n_0$.

# Example: Big-$\Omega$

- $n = \Omega(n)$
- $n^2 = \Omega(n)$
- $2^n = \Omega(n)$
- Example 1.12
- Example 1.13
- Example 1.14
- Example 1.15

# ✓Big-Θ Notation

- Let f(n) be a function which is non-negative for all integers $n \geq 0$.

$$f(n) = \Theta(g(n))$$

$$f(n) \in \Theta(g(n))$$

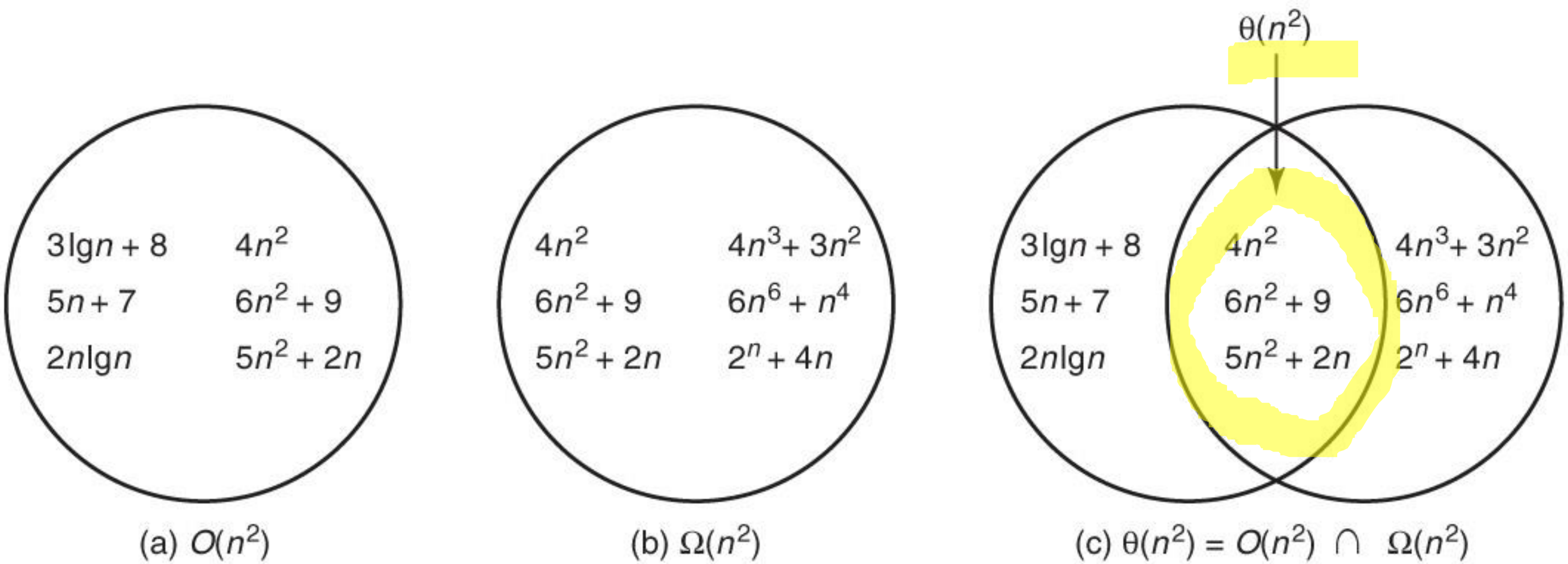
"f(n) is **(big-) theta** g(n)"

if and only if

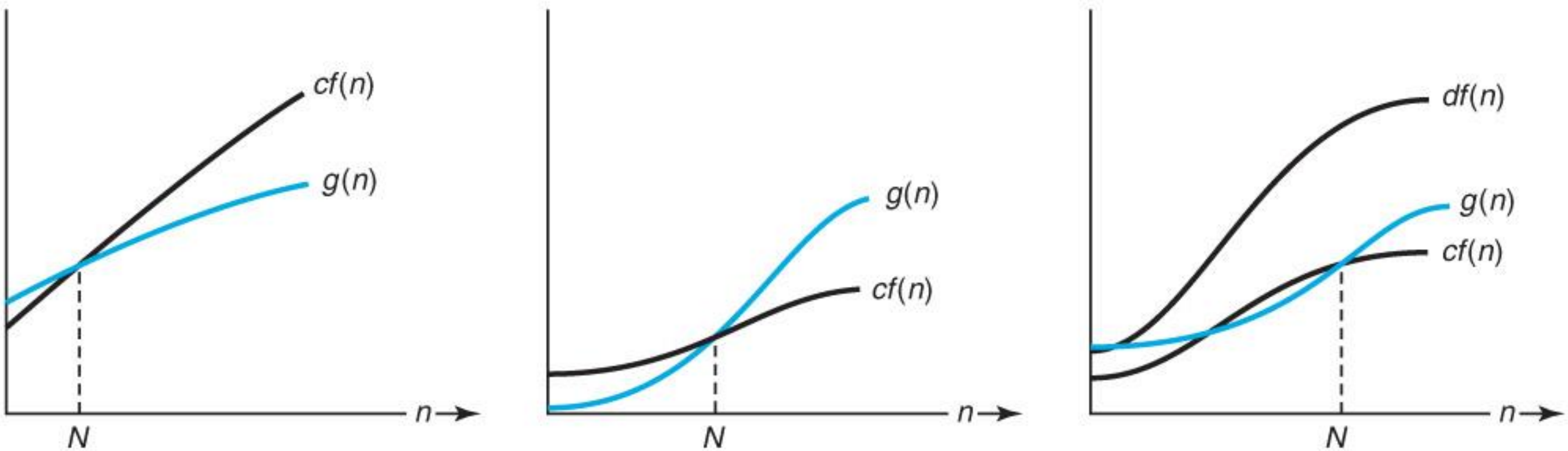
$$f(n) \text{ is } O(g(n)) \text{ and } f(n) \text{ is } \Omega(g(n))$$

# Example: Big-Θ

- Example 1.16
- Figure 1.6



# O, Ω and Θ




(a) $g(n) \in O(f(n))$ (b) $g(n) \in \Omega(f(n))$ (c) $g(n) \in \Theta(f(n))$

# Asymptotic Behaviors of Polynomial Functions

- If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ then $f(n) = O(n^m)$
- If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ then $f(n) = \Omega(n^m)$
- If $f(n) = a_m n^m + \ldots + a_1 n + a_0$ then $f(n) = \Theta(n^m)$

# Tight Bound vs Loose Bound

$$\boxed{F} \quad = \quad O(1) \quad O(\log N) \quad O(N) \quad O(N \log N) \quad O(N^2) \quad O(2^N)$$

Tight bound $\longleftrightarrow$ Loose bound

$$\Omega(1) \ \Omega(\log N) \ \Omega(N) \ \Omega(N \log N) \ \Omega(N^2) \ \Omega(2^N) \quad = \quad \boxed{F}$$

Loose bound $\longleftrightarrow$ Tight bound

# ▶QUIZ?

- Notations for Asymptotic Behavior?
- Big-O Notation for (Asymptotic) Upper Bound?
- Big-$\Omega$ Notation for (Asymptotic) Lower Bounds?

# ✓ A Fast Computer or a Fast Algorithm?

The Order (Growth Rate) of an Algorithm is more important than the Speed of a Computer.

# Example: Time Complexity

- If an $O(n^2)$ algorithm takes 3.1 msec to run on an array of 200 elements, how long would you expect it to take to run on a similar array of 40,000 elements?

  - $c \bullet 200^2 = 3.1$ msec $\Rightarrow$ $c = 3.1 / 200^2$
  - $c \bullet 40000^2 = (3.1 / 200^2)\ 40000^2 = 124000$ msec

  - **124000 msec = 124 seconds**

# ▶QUIZ?

- If an O(n log n) algorithm takes 3.1 msec to run on an array of 200 elements, how long would you expect it to take to run on a similar array of 40,000 elements?
  - **1240 msec   1.24 seconds**

  - $c \bullet (200\ \log 200) = 3.1$ msec $\Rightarrow$ $c = 3.1 / (200\ \log 200)$
  - $c \bullet (40000 \log 40000) = (3.1 / 200\ \log 200)\ (40000 \log 40000)$ $= 1240$ msec

# ▶QUIZ?

- If an O(n) algorithm takes 3.1 msec to run on an array of 200 elements, how long would you expect it to take to run on a similar array of 40,000 elements?
  - **620 msec   .620 seconds**

  - $c \bullet 200 = 3.1$ msec $\Rightarrow$ $c = 3.1 / 200$
  - $c \bullet 40000 = (3.1 / 200)\ 40000 = 620$ msec

# ▶QUIZ?

- If an $O(\log n)$ algorithm takes 3.1 msec to run on an array of 200 elements, how long would you expect it to take to run on a similar array of 40,000 elements?
  - **6.2 msec .0062 seconds**

  - $c \bullet \log 200 = 3.1$ msec $\Rightarrow c = 3.1 / \log 200$
  - $c \bullet \log 40000 = (3.1 / \log 200) \log 40000 = 6.2$ msec

# Example: Time Complexity

- Suppose you have a computer that requires 1 minute to solve problem instances of size n =1,000. Suppose you buy a new computer that runs 1,000 times faster than the old one. What instance sizes can be run in 1 minute, assuming the following time complexities $T(n) = n$ for our algorithm?

- $c \bullet 1000 = 1 \text{ min} = c/1000 \bullet n$
- $\mathbf{n = 10^6}$

# ▶QUIZ?

- Suppose you have a computer that requires 1 minute to solve problem instances of size n =1,000. Suppose you buy a new computer that runs 1,000 times faster than the old one. What instance sizes can be run in 1 minute, assuming the following time complexities $T(n) = n^2$ for our algorithm?

- **$10^{4.5}$**
- $c \bullet 1000^2 = 1 \text{ min} = c/1000 \bullet n^2$

# ✓ Algorithms: Analysis

# How to Calculate the Running Time Complexity

- Depends on the type of an algorithm!

1. Iterative algorithms
   - Summation
2. Recursive algorithms
   - Recurrence equation/relation

# 1. How to Calculate the Running Time Complexity – Iterative Algorithms?

- Iterative algorithms
  - **Summation**

$$\Sigma$$

# Running Time Calculation - Sequence

- Single assignment statement
  - O(1)
- Simple expression
  - O(1)
- Statement1; Statement2; …;Statementn
  - The maximum of O(Statement1), O(Statement2), …, and O(Statementn).

# Running Time Calculation-Conditional

- IF Condition THEN Statement1 ELSE Statement2;
  - O(Condition) + The maximum of O(Statement1) and O(Statement2).
  - The maximum of O(Condition), O(Statement1) and O(Statement2).

# Running Time Calculation - Iteration

- FOR (i=1; i<=N; i++) Statement
  - O(N×Statement) where N= The number of loop iterations.
- FOR (S1 ; S2 ; S3) Statement
  - O(S1 + S2×(N+1) + S3×N + Statement×N)
  - The maximum of O(S1), O(S2× (N+1)), O(S3 × N) and O(Statement × N)

# Running Time Calculation- Iteration

- WHILE (condition) Statement
  - O(N×Statement) where N= The number of loop iterations.

# Example: Running Time Complexity – Iterative Algorithms

```
sum = 0;
for (i=1; i<=n; i++)
    sum += n;
```

**Total = O(n)**

# Example: Running Time Complexity – Iterative Algorithms

```
sum1 = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        sum1 ++;
```

$$\sum_{i=1,n} \sum_{j=1,n} 1 = n^2$$

**Total = $O(n^2)$**

# Example: Running Time Complexity – Iterative Algorithms

```
sum2 = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++)
        sum2 ++;
```

$\sum i{=}1{,}n \;\; \sum j{=}1{,}i \; 1 = n(n+1)/2$

**Total = $O(n^2)$**

# Example: Running Time Complexity – Iterative Algorithms

```
sum1 = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=n; j++)
        sum1 ++;
sum2 = 0;
for (i=1; i<=n; i++)
    for (j=1; j<=i; j++)
        sum2 ++;
```

**Total = $O(n^2)$**

# Example: Running Time Complexity – Iterative Algorithms

```
sum = 0;
for (j=1; j<=n; j++)
     for (i=1; i<=j; i++)
         sum ++;
for (k=0; k<=n; k++)
A[k] = k;
```

$\boxed{\text{Total} = O(n^2)}$

# ▶QUIZ?

Assume $n = 2^k$

```
sum1 = 0;
for (i=1; i<=n; i*=2)
      for (j=1; j<=n; j++)
            sum1 ++;
```

$\sum_{i=0,\log n} \sum_{j=1,n} 1 = n(\log n+1)$

$i = 1, 2, 4, 8, \ldots, n$

$i = 2^0\ 2^1\ 2^2\ 2^3 \ldots 2^{\log n}$

$\sum_{i=1,2,4,8,\ldots,n} \left(\sum_{j=1,n} 1\right) = ?$

**Total = O(n log n)**

# QUIZ?

```
sum2 = 0;
for (i=1; i<=n; i*=2)
    for (j=1; j<=i; j++)
        sum2 ++;
```

Assume $n = 2^k$

$i = 1, 2, 4, 8, \dots, n$

$$\sum_{i=1,2,4,8,\dots,n} \left(\sum_{j=1,i} 1\right)$$
$$= 1 + 2 + 4 + 8 + \dots + n$$
$$= 2^0 + 2^1 + 2^2 + 2^3 + \dots + 2^{\log n}$$
$$= \sum_{i=0,\log n} 2^i$$
$$= 2n - 1$$

$$\sum_{i=0,n} 2^i = 2^{(n+1)} - 1$$

Example A.3

$$2^{\log_2 n} = n$$

Example A.8

**Total = O(n)**

# ▶QUIZ?

```
sum1 = 0;
for (i=1; i<=n; i*=2)
    for (j=1; j<=n; j++)
        sum1 ++;
sum2 = 0;
for (i=1; i<=n; i*=2)
    for (j=1; j<=i; j++)
        sum2 ++;
```

**Total = O(n log n)**

# ▶QUIZ?

```
int fun(int n)
{
  int count = 0;
  for (int i = n; i > 0; i /= 2)
     for (int j = 0; j < i; j++)
        count += 1;
  return count;
}
```

**Total = O(n)**

# ▶QUIZ?

- **Algorithm 1.5**
  - Binary Search – Iterative
- **O(log N)**

# 2. How to Calculate the Running Time Complexity – Recursive Algorithms?

- Recursive algorithms

- **Recurrence equation/relation**

# Using Recurrence Relations for Recursive Algorithms

- The running time of an recursive algorithm can often be described by a recurrence relation or equation.
  - A mathematical formula that generates the terms in a sequence from previous terms.

# Example: Running Time Complexity – Recursive Algorithms

```
1   unsigned int Factorial (unsigned int n)
2   {
3      if (n == 0)
4         return 1;
5      else
6         return n * Factorial (n-1);
7   }
```

```
1
2
3 O(1)
4 O(1)
5
6  Factorial (n-1)
7    }
```

$$
\begin{aligned}
T(n) &= O(1) && \text{if } n=0 \\
T(n) &= T(n-1) + O(1) && \text{if } n>0
\end{aligned}
$$

# Solving By Substitution

$$T(n) = O(1) \quad \text{if } n=0$$
$$T(n) = T(n-1) + O(1) \quad \text{if } n>0$$

$$
\begin{aligned}
\mathbf{T(n)} &= T(n-1) + O(1) \\
&= T(n-2) + O(1) + O(1) \\
&= T(n-3) + O(1) + O(1) + O(1) \\
&= T(n-4) + O(1) + O(1) + O(1) + O(1) \\
&\;. \\
&\;. \\
&= T(0) + O(1) + O(1) + \ldots\ldots + O(1) \\
&= O(1) + n \times O(1) \\
&= \mathbf{O(n)}
\end{aligned}
$$

$\Sigma$

# Running Time Complexity – Recursive Algorithms

```
1    unsigned int Factorial (unsigned int n)
2    {
3        if (n == 0)
4            return 1;
5        else
6            return n * Factorial (n-1);
7    }
```

```
1
2
3 O(1)
4 O(1)
5
6  Factorial (n-1)
7    }
```

$$T(n) = O(1) \qquad \text{if } n=0$$
$$T(n) = T(n-1) + O(1) \qquad \text{if } n>0$$

$$\text{Total} = O(n)$$

# Solving Recurrences by Substitution

- $T(n) = T(n-1) + 1$ if $n>0$
- $T(0) = 1$
  - $T(n) = T(n-1) + 1$
  - $= T(n-2) + 1 + 1$
  - $= T(n-3) + 1 + 1 + 1$
  - $= T(n-4) + 1 + 1 + 1 + 1$
  - .
  - .
  - $= T(n-n) + 1 + 1 + \ldots\ldots + 1$
  - $= 1 + n \times 1$
  - $= 1 + n$
  - $= O(n)$
  - $O(n)$

# Solving Recurrences by Substitution

- $T(n) = T(n-1) + n$ if $n>0$
- $T(0) = 1$
  - $T(n) = T(n-1) + n$
  - $= T(n-2) + (n-1) + n$
  - $= T(n-3) + (n-2) + (n-1) + n$
  - $= T(n-4) + (n-3) + (n-2) + (n-1) + n$
  - .
  - .
  - $= T(n-n) + 1 + 2 + \ldots\ldots. + (n-2) + (n-1) + n$
  - $= 1 + 1 + 2 + \ldots\ldots. + (n-2) + (n-1) + n$
  - $= 1 + n(n+1)/2$
  - $= O(n^2)$
- $O(n^2)$
- Example B.21

# Solving Recurrences by Substitution

- $T(n) = T(n/2) + 1$ if $n>0$
- $T(1) = 1$
  - $T(n) = T(n/2^1) + 1$
  - $= T(n/2^2) + 1 + 1$
  - $= T(n/2^3) + 1 + 1 + 1$
  - $= T(n/2^4) + 1 + 1 + 1 + 1$
  - .
  - .
  - $= T(n/2^{\log n}) + 1 + 1 + \ldots\ldots + 1$
  - $= 1 + \log n \times 1$
  - $= 1 + \log n$
  - $= O(\log n)$
  - $O(\log n)$
  - Example B.1

# QUIZ?

- **$T(n) = 3\,T(n-1)$**
- $T(n) = n\,T(n-1)$
- **$T(n) = 2\,T(n/2) + n$**

# QUIZ:

- **$T(n) = 3\,T(n-1)$**
  - $O(3^n)$
- $T(n) = n\,T(n-1)$
  - $O(n!)$
- **$T(n) = 2\,T(n/2) + n$**
  - $O(n \log n)$

# ▶ QUIZ?

```
int recursive (int n) {
if(n == 1)
return (1);
else
return (recursive (n-1) + recursive (n-1));
}
```

$$T(n) = 2\,T(n-1) + 1$$

$$O(2^n)$$

# ✓ A General Method for Some Recurrence Relations

- T(n) = aT(n/b) + c*n^k
- T(1) = d
  - T(n) = O(n^k) if a < b^k
  - T(n) = O(n^k log n) if a = b^k
  - T(n) = O(n^log _b(a)) if a > b^k
- Theorem B.5 A Master Theorem
  - Example B.26
  - Example B.27

# ▶QUIZ?

- **$T(n) = T(n/2) + 1$**
- $T(n) = 4\,T(n/2) + n$
- $T(n) = T\,(n/2) + n^2$
- **$T(n) = 2\,T(n/2) + n$**
- **$T(n) = 2\,T(n/2) + 1$**

# ▶QUIZ:

- **$T(n) = T(n/2) + 1$**
  - $O(\log n)$
- $T(n) = 4\,T(n/2) + n$
  - $O(n^2)$
- $T(n) = T(n/2) + n^2$
  - $O(n^2)$
- **$T(n) = 2\,T(n/2) + n$**
  - $O(n \log n)$
- **$T(n) = 2\,T(n/2) + 1$**
  - $O(n)$

# ✓ A General Method for Some Recurrence Relations

- Theorem B.6 **A Master Theorem**
  - Example B.28

# ▶QUIZ?

- Exercise 25 (Appendix B)

# Common Recurrence Relations

- $T(n) = T(n-1) + \Theta(1)$
  - $T(n) = O(n)$
- $T(n) = T(n-1) + \Theta(n)$
  - $T(n) = O(n^2)$

# Common Recurrence Relations

- $T(n) = T(n/2) + \Theta(1)$
  - $T(n) = O(\log n)$

Example B.1 & Example B.18

# Common Recurrence Relations

- $T(n) = 2T(n/2) + \Theta(1)$
  - $T(n) = O(n)$
- $T(n) = 2\,T(n/2) + \Theta(n)$
  - $T(n) = O(n \log n)$

Example B.19

# ▶ QUIZ?

- $T(n) = O(1)$ if $n=1$
- $T(n) = T(n/2) + O(n)$ if $n>1$

# ▶ QUIZ?

- $T(n) = O(1)$ if $n=1$
- $T(n) = T(n/3) + O(1)$ if $n>1$

# ▶QUIZ?

- $T(n) = O(1)$ if $n=1$
- $T(n) = 8T(n/2)$ if $n>1$

$$x^{\log_a y} = y^{\log_a x}$$ Example A.8

# ▶QUIZ?

- $T(n) = O(1)$ if $n=1$
- $T(n) = 8T(n/2) + n^2$ if $n>1$

# ▶QUIZ?

- $T(n) = O(1)$ if $n=1$
- $T(n) = 7T(n/2)$ if $n>1$

# ▶QUIZ?

- $T(n) = O(1)$ if $n=1$
- $T(n) = 7T(n/2) + n^2$ if $n>1$

# ▶QUIZ?

- $T(n) = O(1)$ if $n=1$
- $T(n) = 2T(n-1) + O(1)$ if $n>1$

# QUIZ?

- $T(n) = 0$ if $n=1$
- $T(n) = T(n-1) + 2/n$ if $n>1$

Example B.22
Example A.8

# ✓ Amortized Behavior

- So far, we considered the worst/average cost for a single operation
- How about the cost for a series/sequence of N operations?
  - N times the worst-case cost of any one operation.
  - Or better cost?
- Observation?
  - Any one particular operation may be slow, but the average time over a sufficiently large number of operations is fast.

# Amortized Analysis

- Cost for a series/sequence of N operations?
- The **amortized cost** of **N** operations is the total cost of the operations divided by **N**.

- **Amortized analysis!**
  - **Self-Adjusting Data Structures!**

# Space Requirement (Complexity) of Algorithms

- Time requirements for an algorithm that manipulates a data structure.
- Space requirements for the data structure itself.

# ✓ Time and Space Tradeoff in Algorithms?

One can often achieve a reduction in time requirements if one is willing to sacrifice space requirements or vice versa.

# Textbook Readings

- Chapter 1:
  - 1.1
  - 1.2
  - 1.3 (1.3.1 & 1.3.2 only)
  - 1.4 (1.4.1 & 1.4.2 only)
  - 1.5