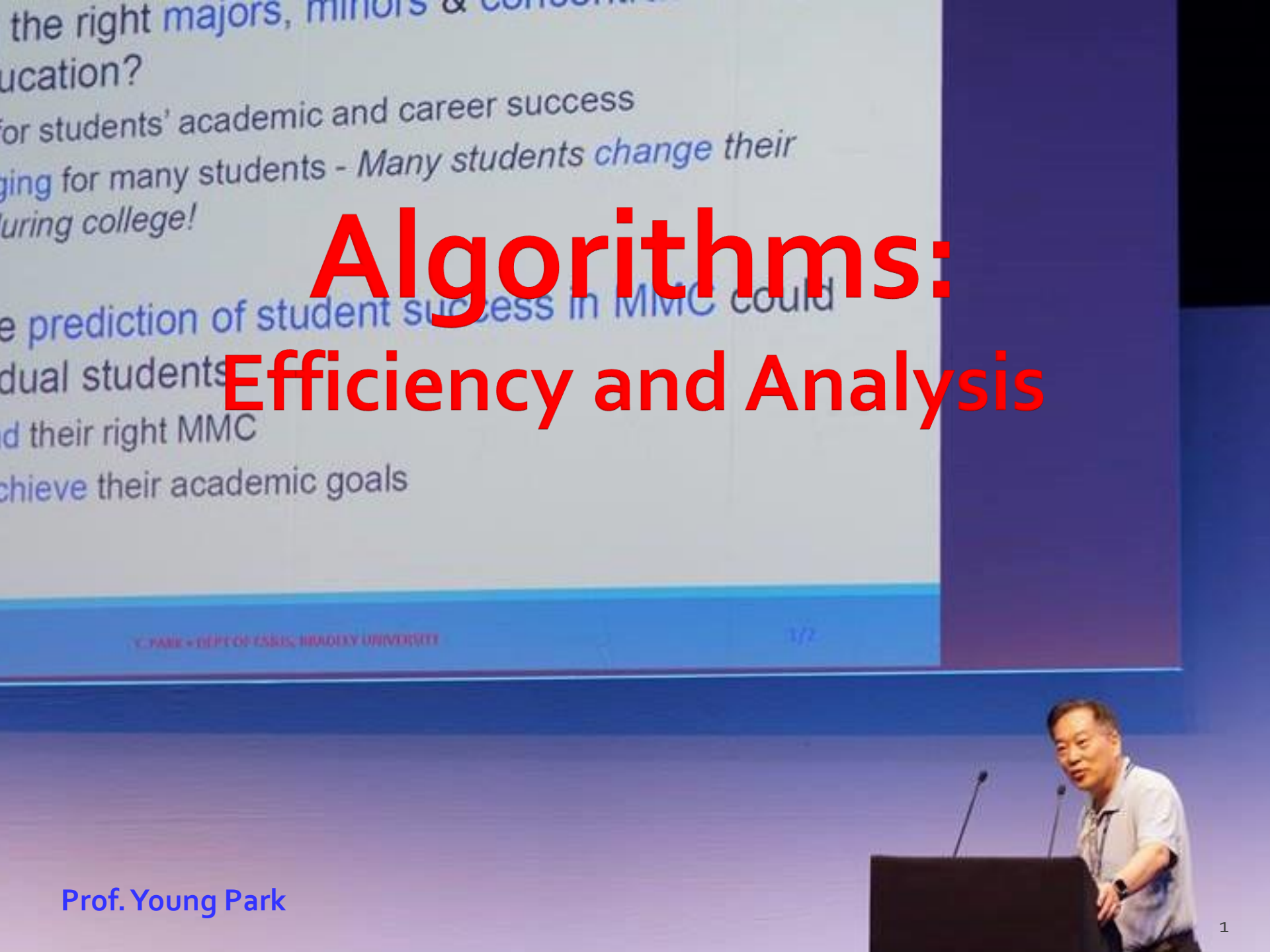
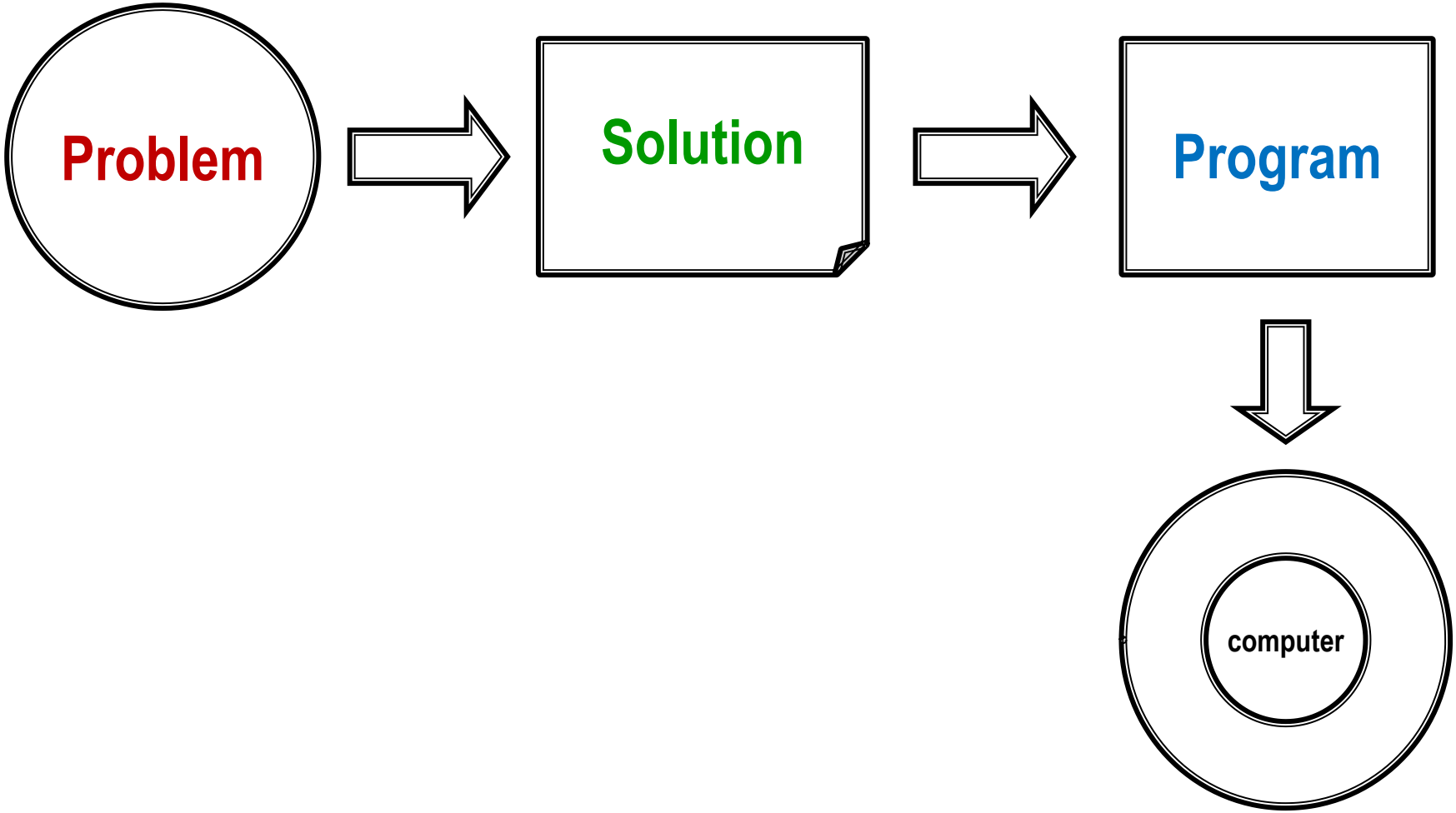
Algorithms:
Efficiency and Analysis



Prof. Young Park

✓ 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**

Fundamental Computational Problems

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

Solution

- Input (and store) **data.**
- Process 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

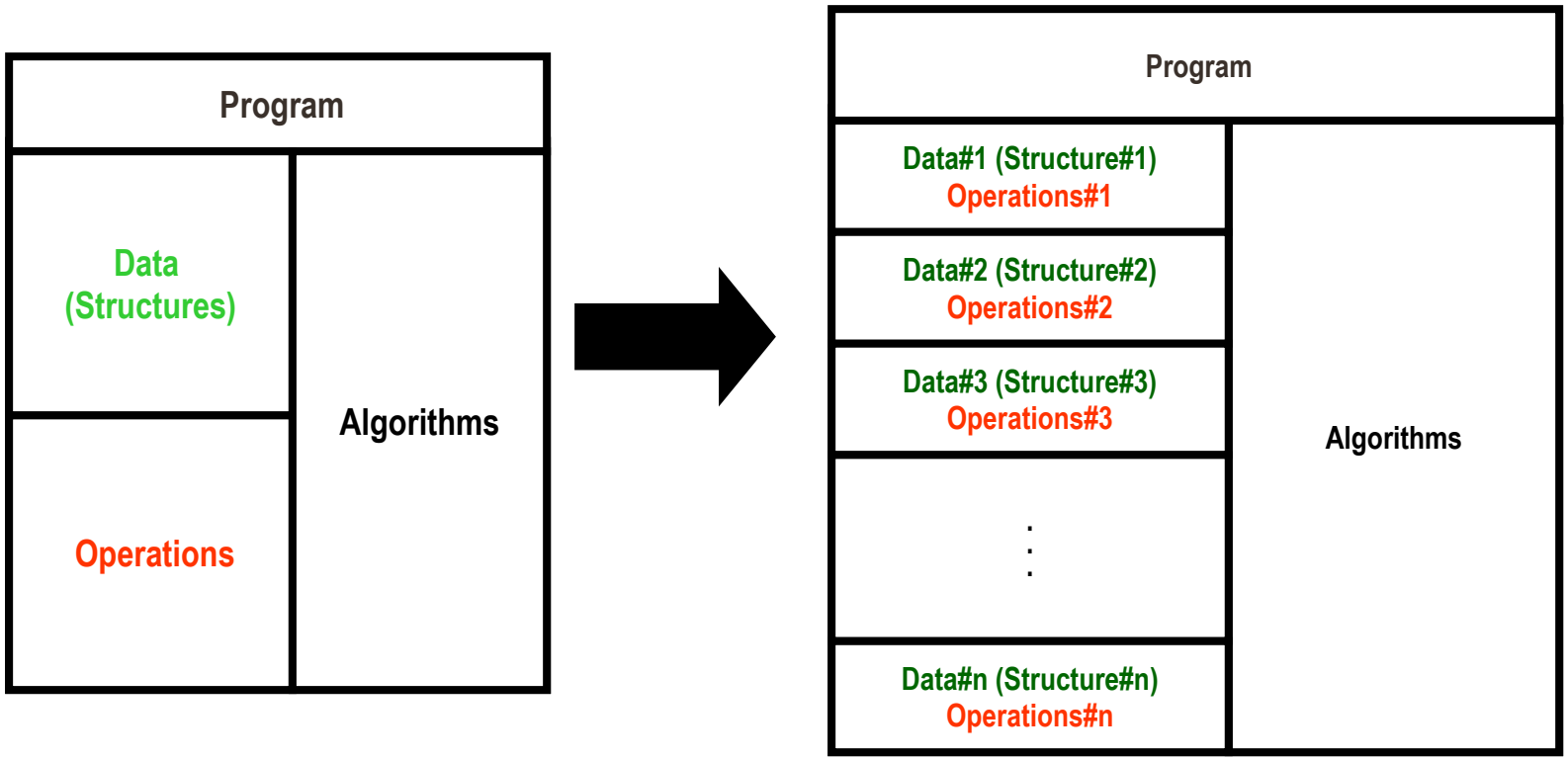
- Divide-and-Conquer
- Dynamic Programming
- Greedy Approach
- Backtracking
- Branch-and-Bound
- Randomization

Solution = DS + Algorithms

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

Program=Data Structures + Algorithms

- An implementation of a solution (design)!



✓ Algorithms

Types of Algorithms

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

Algorithm Efficiency

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

  - **Time complexity**
  - **Space complexity**

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**

- **Algorithm 1.6** N-th **Fibonacci** Term-**Recursive**
- **Algorithm 1.7** N-th **Fibonacci** Term -**Iterative**

Search - Sequential Search

- Algorithm 1.1 Sequential Search on an **unsorted (ordered) array**
  - Linear time **O(N)**
- **Can we improve?**
- **Idea?**
  - **Sorted (Ordered)** for Better Search!
  - **Better Data Structures!**

Search - Binary Search

- **Algorithm 1.5 Binary Search** – Iterative **– on a sorted (ordered) array**
  - Divide-and-Conquer
  - **O(log N)**

▶QUIZ?

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

Computing The Fibonacci Sequence

- (0,) 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, …

- $fib_0 = 0$
- $fib_1 = 1$
- $fib_n = fib_{n-1} + fib_{n-2}$ for n >= 2;

The Fibonacci Sequence – Recursive

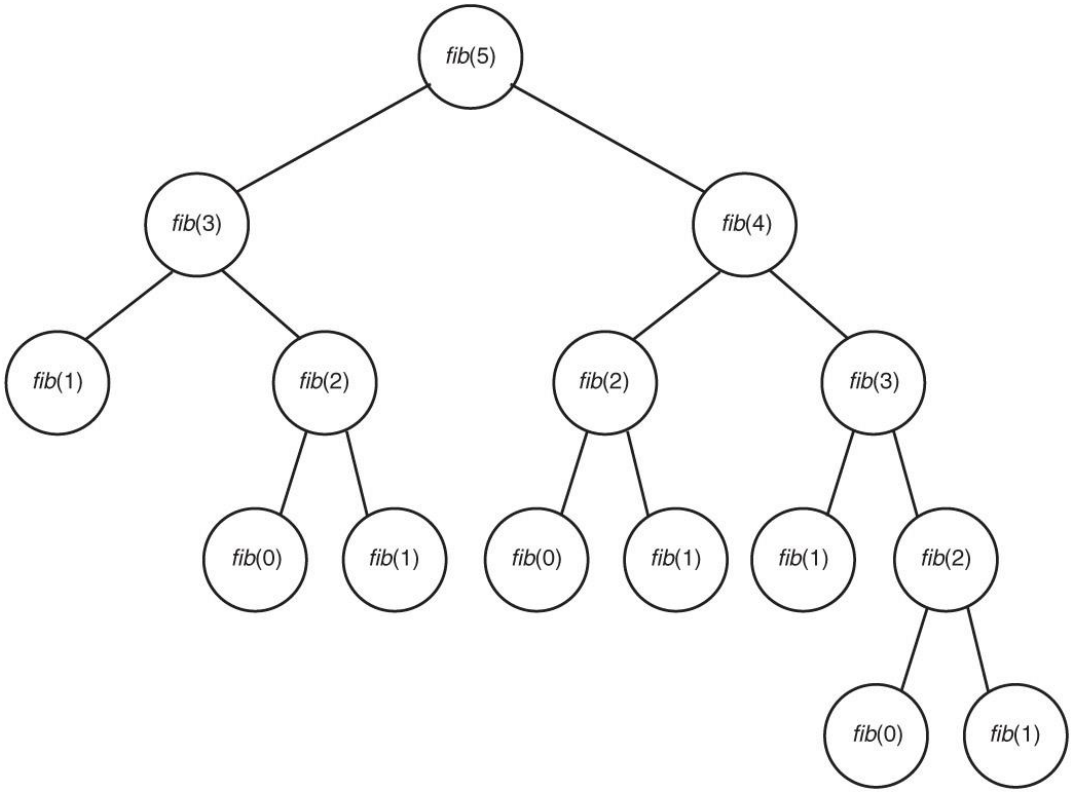
- Algorithm 1.6 N-th Fibonacci Term (Recursive)
  - **Divide-and-Conquer**
  - Top-down
  - Exponential time **$O(2^N)$**

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

- Example: fib(5)?
- 5

The Fibonacci Sequence - Recursive

- fib(5)?



The Fibonacci Sequence - Recursive

- This **D&C-based recursive** algorithm is extremely inefficient – exponential time $O(2^N)$.

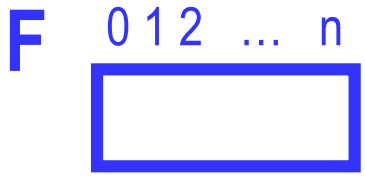
- **Can we improve?**
- **Idea?**
  - **Dynamic Programming**

  - **Better Algorithm Paradigm!**

The Fibonacci Sequence – Iterative

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

- **The Array F**
  - F[i] = F[i – 1] + F[i – 2] for n >= 2;
  - F[0]=0
  - F[1] = 1



- Example: fib(5)?
  - F[5]?
  - 5

Computational Complexity of 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

Intractability of 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 there is no efficient algorithm.
    - **NP and NP-complete problems**
    - NP-Hard problems
    - …

✓ Algorithm Time Complexity

Time 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 time efficiency by determining
  - **the number of some basic operation as a function of the size of the input.**

Basic Operations of Algorithms

- 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.

- **Examples:** Basic operations?
  - Algorithm 1.1 Sequential Search - Iterative
  - Algorithm 1.2 Add Array Members
  - Algorithm 1.3 Exchange Sort
  - Algorithm 1.4 Matrix Multiplication

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 **asymptotic 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!**

Asymptotic Order of Growth

- The asymptotic running time of the algorithm A for the problem size n: **GrowthRate**$_{\text{Time}}(n)$

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?

Asymptotic Time Complexity
Bounds

- Upper Bound
- Lower Bound

Notations for Asymptotic Time Complexity

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

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 $n \geq o$.

$f(n) = \mathbf{O}(g(n))$
$f(n) \in \mathbf{O}(g(n))$
"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 coefficients.
  - $O(2\ n^2) \rightarrow O(n^2)$
  - $O(1024) \rightarrow O(1)$

What dominates?

Example: Big-O

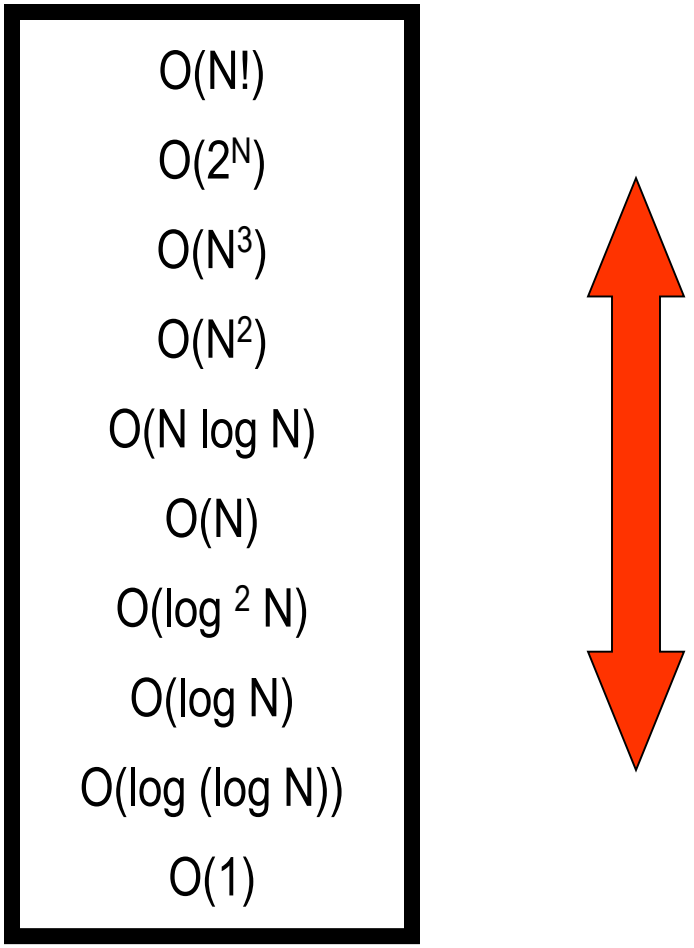
- $\log n = O(n)$
- $n = O(n)$
- $100\,n + 10 \log n = O(n)$

- Example 1.7
- Example 1.8
- Example 1.11

Algorithm Growth Rates

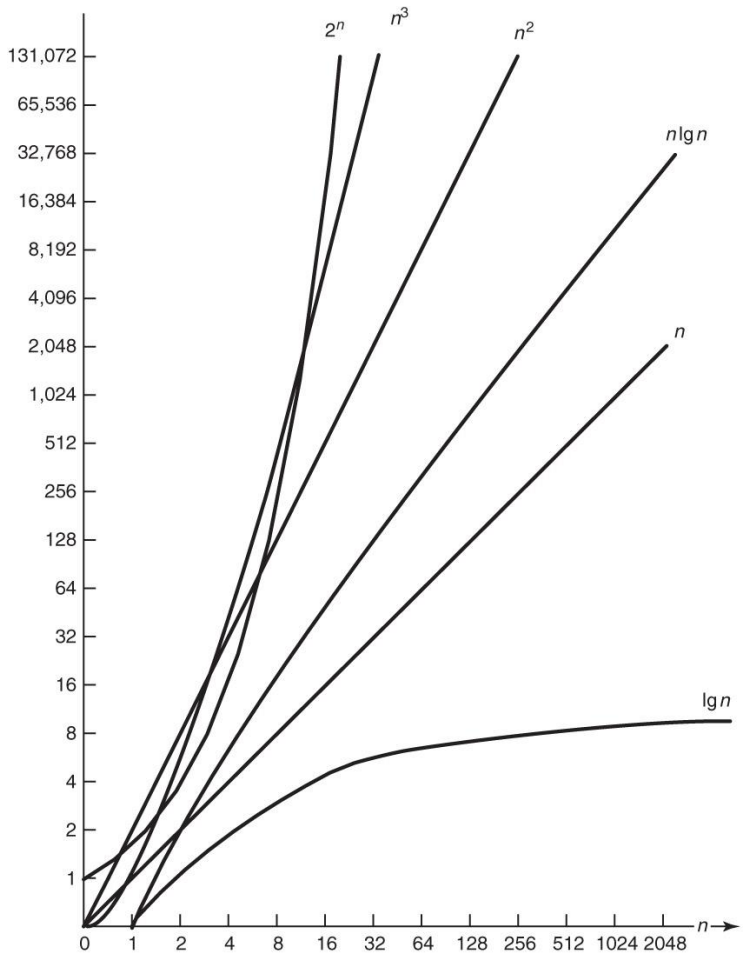
- A constant growth rate **O(1)**
- A logarithmic growth rate **O(log N)**
- A logarithmic squared growth rate **O(log ^{2} N)**
- A linear growth rate **O(N)**
- A linear-logarithmic (?) growth rate **O(N log N)**
- A quadratic growth rate **O(N^2)**
- A cubic growth rate **O(N^3)**
- A polynomial growth rate **O(N^k)** for a constant k.
- An exponential growth rate **O(2^N)**
- A factorial growth rate **O(N!)**

Comparison of Growth Rates



Comparison of Growth Rates

- Figure 1.3



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 of n items 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 of n items.

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 of n items.

$O(N^2)$ - 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 (N by N)array is quadratic in the number N 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 N of disks used.

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 $n \geq o$.

$f(n) = \Omega(g(n))$
$f(n) \in \Omega(g(n))$
"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-Ω

- $n = \Omega(n)$
- $n^2 = \Omega(n)$
- $2^n = \Omega(n)$

- 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 ≥ o.

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

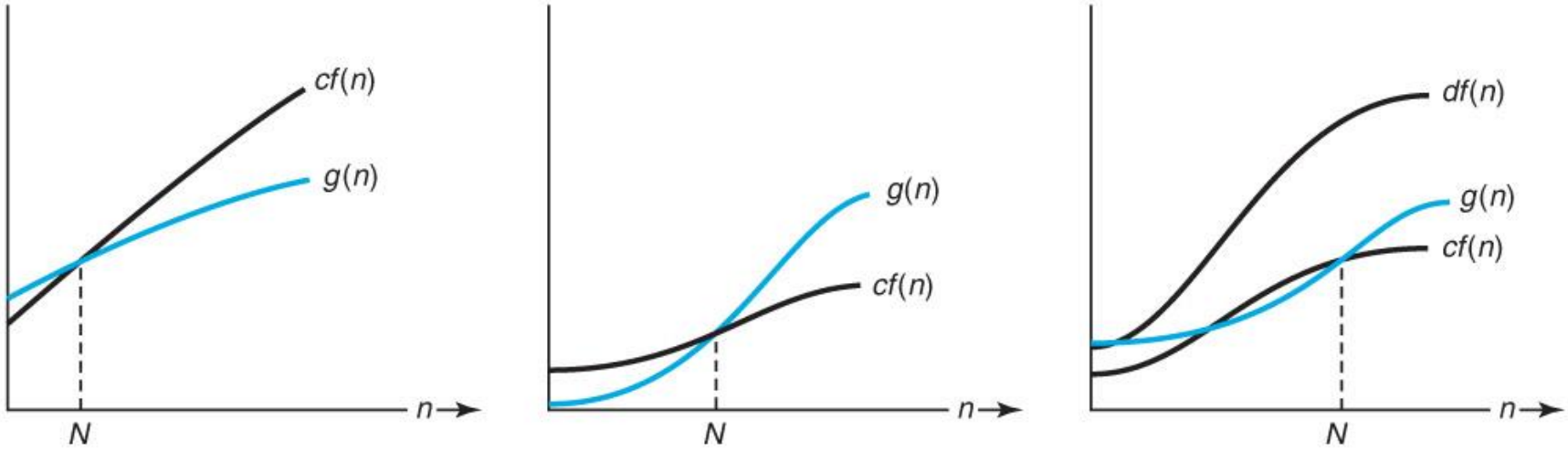
$f(n) \in \Theta(g(n))$

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

if and only if

$f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$

O, Ω and Θ



Asymptotic Behaviors of **Polynomial Functions**

- 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 $f(n) = \Theta(n^m)$

▶QUIZ?

- Notations for Asymptotic Behavior?
- Big-O Notation for (Asymptotic) Upper Bound?
- Big-Ω 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.

✓ How to Calculate the Running Time Complexity of Algorithm?

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**

$$\sum$$

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

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

$\sum_{i=1,n} 1 = 1+1+1+\ldots+1 = n$

Total = O(n)

Example: Running Time Complexity – Iterative Algorithms

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

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

Total = $O(n^2)$

Example: Running Time Complexity – Iterative Algorithms

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

$\sum_{i=1,n} \sum_{j=1,i} 1$
$= 1 + 2 + 3 + \ldots + n$
$= n(n+1)/2$

Total = $O(n^2)$

Example: Running Time Complexity – Iterative Algorithms

```
1   sum1 = 0;
2   for (i=1; i<=n; i++)
3       for (j=1; j<=n; j++)
4           sum1 ++;
5   sum2 = 0;
6   for (i=1; i<=n; i++)
7       for (j=1; j<=i; j++)
8           sum2 ++;
```

Total = $O(n^2)$

Example: Running Time Complexity – Iterative Algorithms

```
1   sum = 0;
2   for (j=1; j<=n; j++)
3       for (i=1; i<=j; i++)
4           sum ++;
5   for (k=0; k<=n; k++)
6   A[k] = k;
```

Total = $O(n^2)$

▶QUIZ?

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

Assume $n = 2^k$

$\sum i=1,2,4,8,\ldots,n \ (\sum j=1,n \ 1)$
$= \sum i=0,\log n \ (\sum j=1,n \ 1)$
$= n (\log n+1)$

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

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

Total = O(n log n)

▶QUIZ?

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

Assume $n = 2^k$

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

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

$= 1 + 2 + 4 + 8 + \ldots + n$

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

$= \sum_{i=0,\log n} 2^i$

$= 2n - 1$

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

Example A.3

$2^{\log_2 n} = n$

Example A.8

Total = O(n)

▶QUIZ?

```
1   sum1 = 0;
2   for (i=1; i<=n; i*=2)
3       for (j=1; j<=n; j++)
4           sum1 ++;
5   sum2 = 0;
6   for (i=1; i<=n; i*=2)
7       for (j=1; j<=i; j++)
8           sum2 ++;
```

Total = O(n log n)

▶QUIZ?

- **Algorithm 1.5**
  - Binary Search – Iterative
- Calculate Big-O?
  - **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    }
```

Recurrence Equation

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

Solving By Substitution

Recurrence Equation

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

Substitution

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

$$\sum$$

$$\sum_{i=1,n} 1 = n$$

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    }
```

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

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

QUIZ? By Substitution

- **T(n) = 2 T(n-1) T(0) = 1**
  - $O(2^n)$
- **T(n) = n T(n-1) T(0) = 1**
  - O(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

Solving Recurrences by Master Theorem

- **T(n) = T(n/2) + 1** if n>1
- **T(1) = 1**
  - a=1 b=2, k=0 => a = b^k case!
  - **O(n^o log n)**
  - **O(log n)**
- Example B.1

▶QUIZ? Using Master Theorem

- **T(n) = 2 T(n/2) + n T(1) = 1**
  - O(n log n)
- **T(n) = 2 T(n/2) + 1 T(1) = 1**
  - O(n)

QUIZ? By Substitution

- **T(n) = 2 T(n/2) + n T(1) = 1**
  - O(n log n)
- **T(n) = 2 T(n/2) + 1 T(1) = 1**
  - O(n)

T(n) = 2 T(n/2^1) + n
= 2 [2 T(n/2^2) + n/2] + n
= 2^2 T(n/2^2) + n + n
= 2^2 [2 T(n/2^3) + n/2^2] + n + n
= 2^3 T(n/2^3) + n + n + n
.
.
= $2^{\log n}$ T(n/$2^{\log n}$) + n + … + n + n + n
= n T(1) + (log n) n
= n + n log n
= **O(n log n)**

T(n) = 2 T(n/2^1) + 1
= 2 [2 T(n/2^2) + 1] + 1
= 2^2 T(n/2^2) + 2 + 1
= 2^2 [2 T(n/2^3) + 1] + 2 + 1
= 2^3 T(n/2^3) + 2^2 + 2 + 1
.
.
= $2^{\log n}$ T(n/$2^{\log n}$) + $2^{\log n-1}$ + … + 2^2 + 2 + 1
= $2^{\log n}$ + $2^{\log n-1}$ + … + 2^2 + 2 + 1
= $2^{\log n+1}$-1 = 2 n- 1
= **O(n)**

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

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

▶QUIZ?

```
1  int recursive (int n) {
2  if(n == 1)
3  return (1);
4  else
5  return (recursive (n-1) + recursive (n-1));
6  }
```

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

$$\mathbf{O(2^n)}$$

▶QUIZ?

- **Algorithm 2.1**
  - Binary Search – Recursive
- Calculate Big-O?

$$T(n) = 1 \quad \text{if } n=1$$
$$T(n) = T(n/2) + 1 \quad \text{if } n>1$$

  - **O(log N)**

✓ From Asymptotic Time to Exact Time

Example: Asymptotic Time to Exact Time

- 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 • (200 log200) =3.1 msec $\Rightarrow$ c = 3.1 / (200 log200)
  - c • (40000 log40000) = (3.1 /200 log200) (40000 log40000) = 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: Asymptotic Time to Exact Time

- 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?

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

▶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?

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

✓ Amortized Time Complexity

Amortized Time Complexity

- 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 Time Analysis

- The **amortized cost** of an operation is the total cost of the N operations divided by **N**.
- **Amortized analysis!**
  - **Self-Adjusting Data Structures!**

▶QUIZ?

- Amortized Behavior?
- Asymptotic behavior?

Space Requirement (Complexity) of Algorithms

- 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 (tradeoff).

Homework Assignment

Homework Assignment?

- Chapter 1: Exercise #4, #25, #30 and #34
- Appendix B: #24

✓ 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
- Appendix A & B
  - A.1, A.5
  - B.3

the right majors, minors &

ucation?

or students' academic and career success

ging for many students - *Many students change their during college!*

e prediction of student success in MMC could dual students

d their right MMC

chieve their academic goals

END

Prof. Young Park