Introduction to Computer Science & Engineering

Lecture 6: Problem Solving Algorithms

Jeonghun Park

Problem Solving

- The act of finding a solution of a perplexing, distressing, vexing, or unsettled question

jeonghun.park@knu.ac.kr

Problem Solving

- How do we solve problems?
  - Understand the problem
  - Devise a plan
  - Carry out the plan
  - Look back

jeonghun.park@knu.ac.kr

Problem Solving by Computers

- Maybe different from what we do with our hands

jeonghun.park@knu.ac.kr

Strategies

- A solution can be found in questions
  - What do I know about the problem?
  - What is the information that I have to process in order the find the solution?
  - What does the solution look like?
  - What sort of special cases exist?
  - How will I recognize that I have found the solution?

jeonghun.park@knu.ac.kr

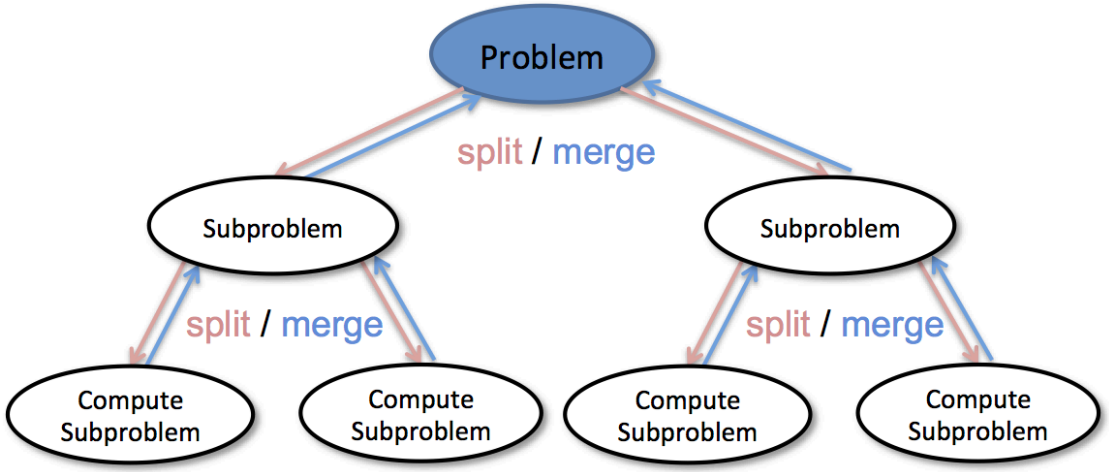
There Is No “Totally New” Problems

- Never reinvent the wheel
  - Similar problems come up again and again in different guises
  - A good programmer recognizes a task or subtask that has been solved before and plugs in the solution
  - Can you think of two similar problems?

jeonghun.park@knu.ac.kr

Divide and Conquer

- Break up a large problem into smaller units and solve each smaller problem
  - Applies the concept of abstraction
  - The divide-and-conquer approach can be applied over and over again until each subtask is manageable



jeonghun.park@knu.ac.kr

Computer Problem Solving

- Analysis and Specification Phase
  - Analyze
  - Specification
- Algorithm Development Phase
  - Develop algorithm
  - Test algorithm
- Implementation Phase
  - Code algorithm
  - Test algorithm
- Maintenance Phase
  - Use
  - Maintain

jeonghun.park@knu.ac.kr

Computer Problem Solving

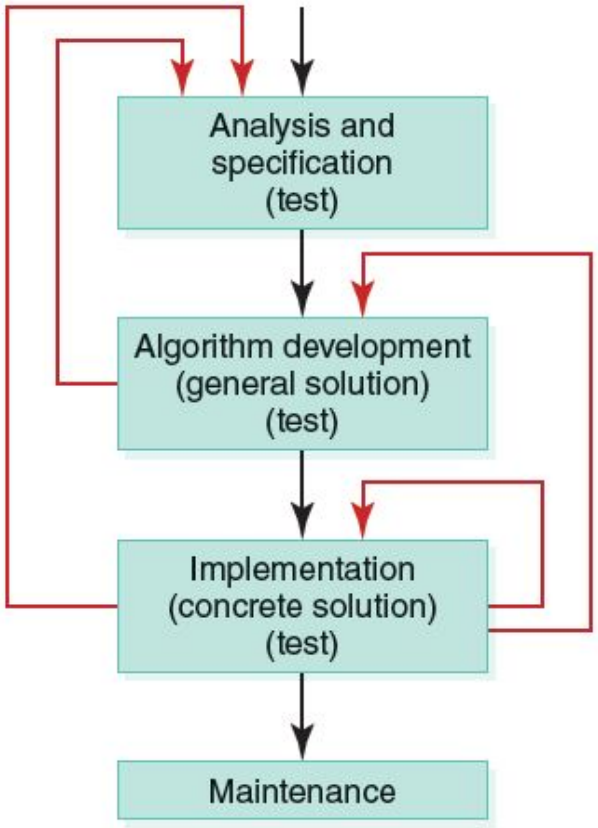


FIGURE 7.3 The interactions among the four problem-solving phases

jeonghun.park@knu.ac.kr

Algorithms

- A set of unambiguous instructions for solving a problem or subproblem in a **finite** amount of time using a **finite** amount of data
- Abstract Step
  - An algorithmic step containing unspecified details
- Concrete Step
  - An algorithm step in which all details are specified

jeonghun.park@knu.ac.kr

Algorithm Development

- Two methodologies used to develop computer solutions to a problem
- Top-down design
  - focuses on the **tasks** to be done
- Object-oriented design
  - focuses on the **data** involved in the solution

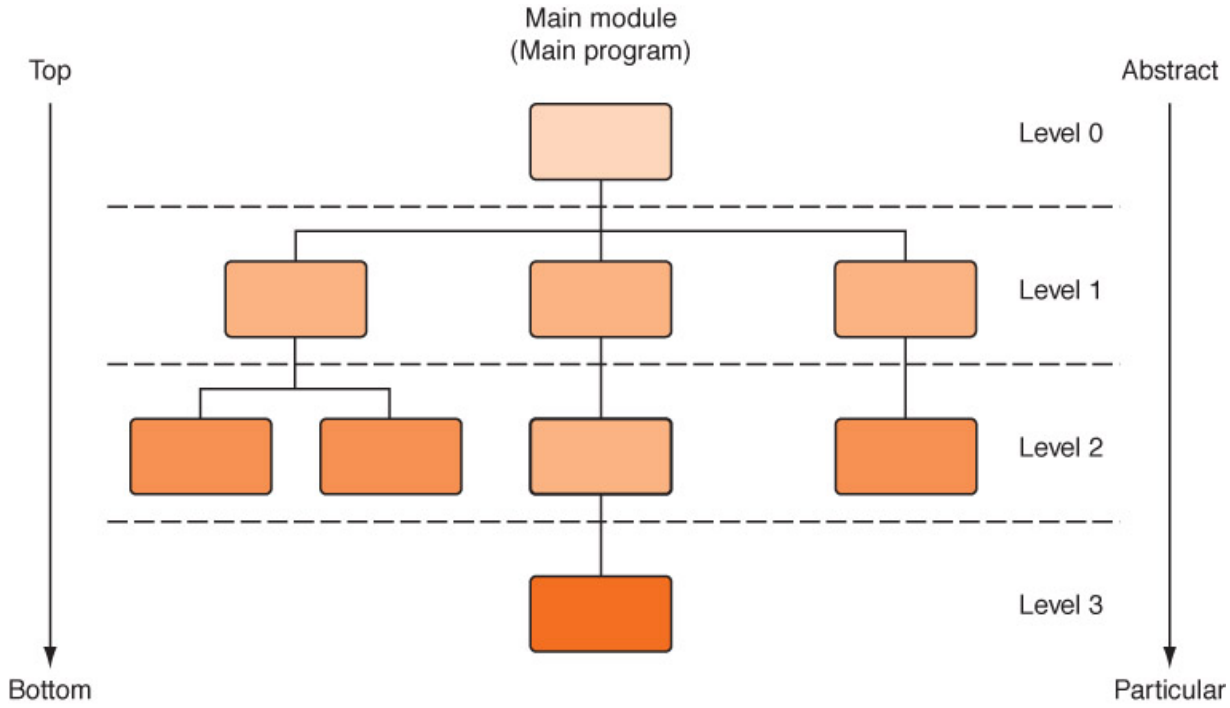
jeonghun.park@knu.ac.kr

Methodology

- Analyze the Problem
  - Understand the problem!!
  - Develop a plan of attack
- List the Main Tasks (becomes Main Module)
  - Restate problem as a list of tasks (modules)
  - Give each task a name
- Write the Remaining Modules
  - Restate each abstract module as a list of tasks
  - Give each task a name
- Re-sequence and Revise as Necessary
  - Process ends when all steps (modules) are concrete

jeonghun.park@knu.ac.kr

Top-Down Design



- Process continues for as many levels as it takes to make every step concrete
- Name of (sub)problem at one level becomes a module at next lower level

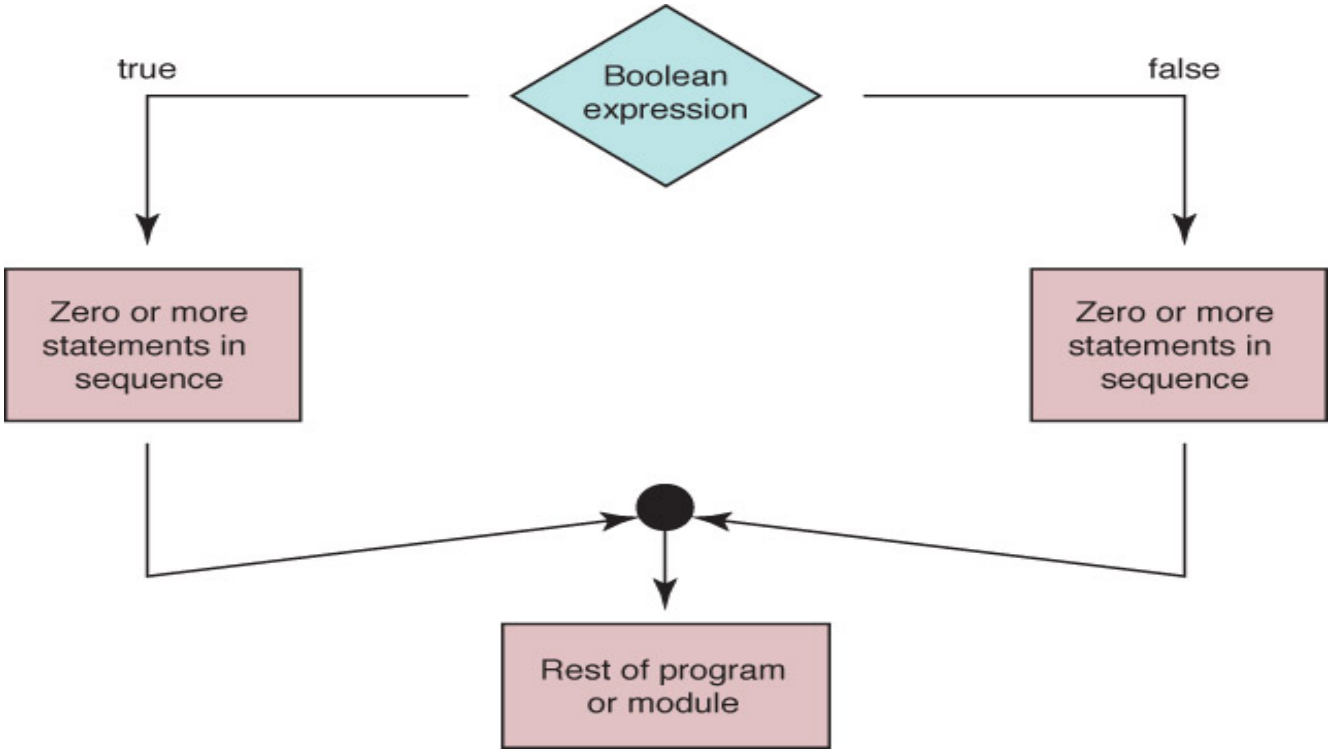
jeonghun.park@knu.ac.kr

Control Structures

- An instruction that determines the order in which other instructions in a program are executed
- It is similar to building blocks that constructs an algorithm

jeonghun.park@knu.ac.kr

Selection Statement



jeonghun.park@knu.ac.kr

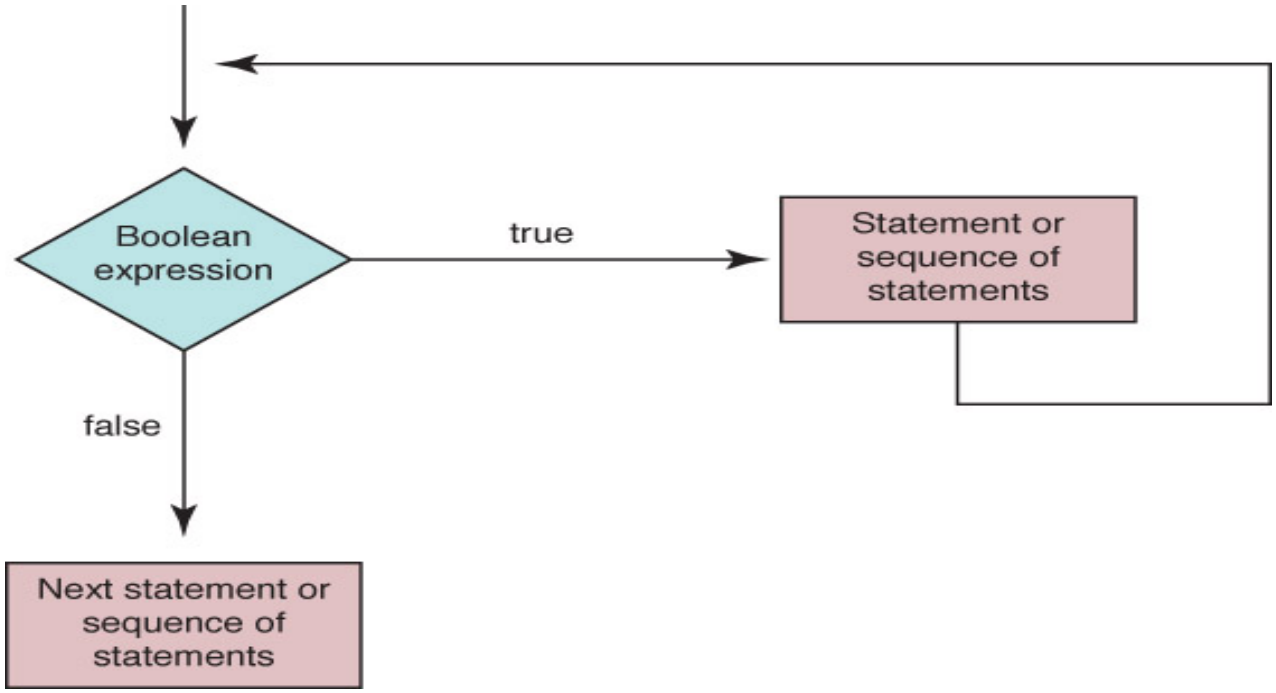
Examples

- Weather determine problem:

```
IF (temperature > 90)
    Write “Texas weather: wear shorts”
ELSE IF (temperature > 70)
    Write “Ideal weather: short sleeves are fine”
ELSE IF (temperature > 50)
    Write “A little chilly: wear a light jacket”
ELSE IF (temperature > 32)
    Write “Philadelphia weather: wear a heavy coat”
ELSE
    Write “Stay inside”
```


jeonghun.park@knu.ac.kr

Looping Statement



jeonghun.park@knu.ac.kr

Examples

- A count-controlled loop

```
Set sum to 0
Set count to 1
While (count <= limit)
  Read number
  Set sum to sum + number
  Increment count
Write "Sum is " + sum
```


jeonghun.park@knu.ac.kr

Examples

- An event-controlled loop

```
Set sum to 0
Set allPositive to true
WHILE (allPositive)
  Read number
  IF (number > 0)
  Set sum to sum + number
  ELSE
  Set allPositive to false
Write "Sum is " + sum
```


jeonghun.park@knu.ac.kr

Examples

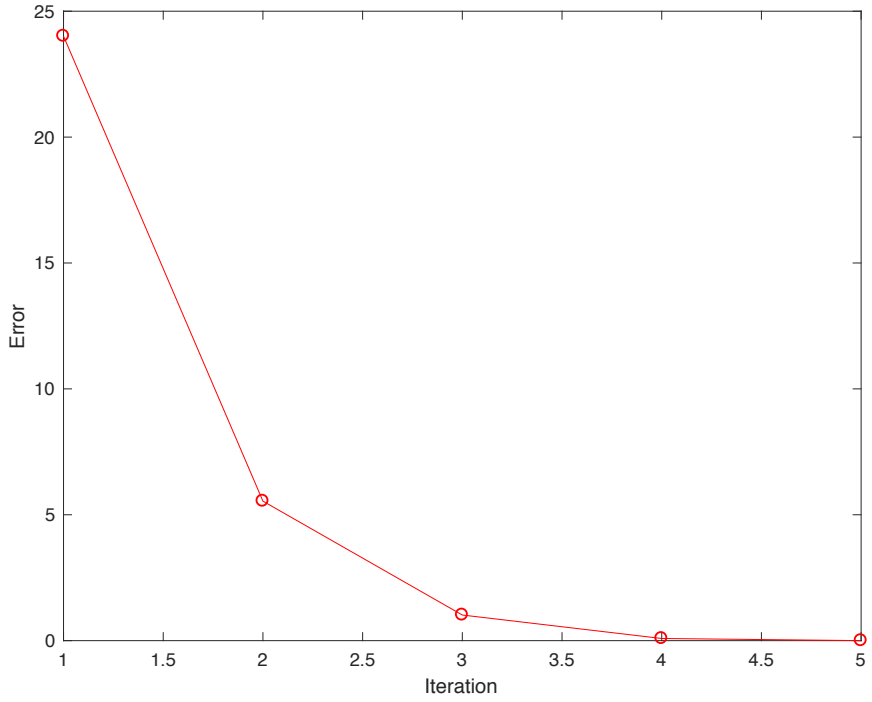
- Find the square root problem
  - Calculate $\sqrt{z}$ given z

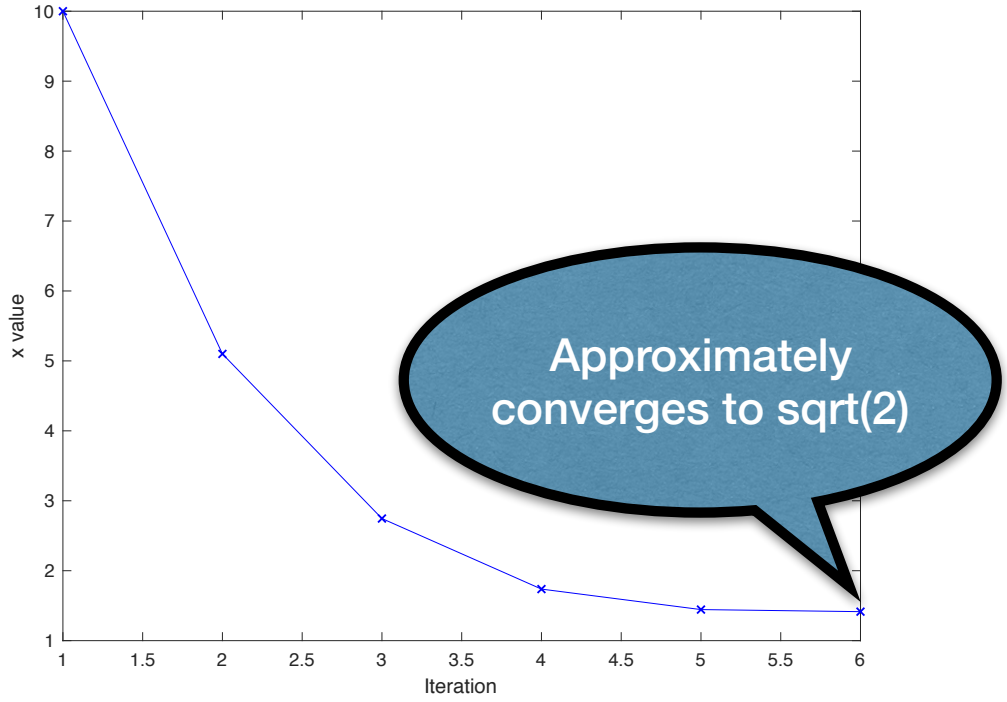
Read z
Set $\tilde{x}_{\text{init}}$
Set $\epsilon = 1$
Set $\tilde{x}_{\text{old}} \leftarrow \tilde{x}_{\text{init}}$
WHILE (ϵ > 0.001)
  Set $\tilde{x}_{\text{new}} \leftarrow (\tilde{x}_{\text{old}} + (z/\tilde{x}_{\text{old}}))/2$
  Set $\epsilon = \left|z - \tilde{x}^2_{\text{new}}\right|$
Write out $\tilde{x}_{\text{new}}$

jeonghun.park@knu.ac.kr

Plots

- In this example, we use
  - $\tilde{x}_{\text{init}} = 10,\ z = 2$





jeonghun.park@knu.ac.kr

Data Types

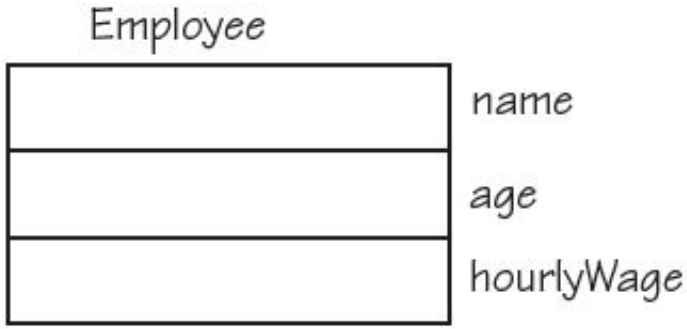


FIGURE 7.6 Record Employee

- Structure

Employee employee
Set employee.name to "Frank Jones"
Set employee.age to 32
Set employee.hourlyWage to 27.50

jeonghun.park@knu.ac.kr

Arrays

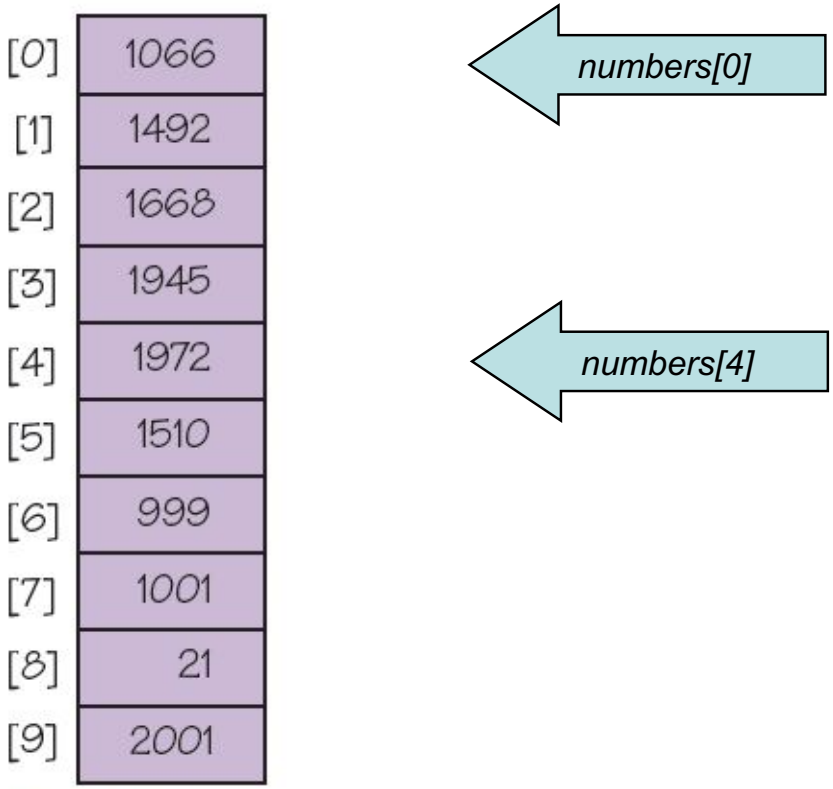


FIGURE 7.5 An array of ten numbers

jeonghun.park@knu.ac.kr

Arrays

- As data is being read into an array, a counter is updated so that we always know how many data items were stored
- If the array is called *list*, we are working with

list[0] to list[length-1] or
list[0]..list[length-1]

jeonghun.park@knu.ac.kr

Unsorted Array

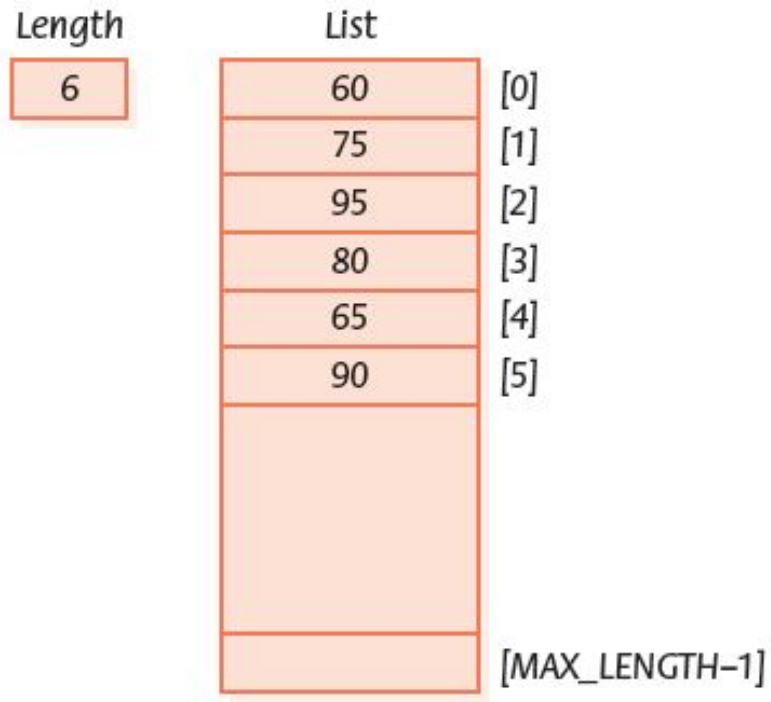


FIGURE 7.7 An unsorted array

jeonghun.park@knu.ac.kr

Algorithms with Arrays

- Array size check algorithm

```
integer data[20]
Write “How many values?”
Read length
Set index to 0
WHILE (index < length)
  Read data[index]
  Set index to index + 1
```


jeonghun.park@knu.ac.kr

Algorithms with Arrays

- Sequential search algorithm

Set Position to 0
Set found to FALSE
WHILE (position < length AND NOT found)
IF (numbers [position] equals searchitem)
Set Found to TRUE
ELSE
Set position to position + 1

jeonghun.park@knu.ac.kr

Boolean Operations

- Boolean Operators
  - A Boolean variable is a location in memory that can contain either *true* or *false*
  - Boolean operator AND returns *TRUE* if both operands are true and *FALSE* otherwise
  - Boolean operator OR returns *TRUE* if either operand is true and *FALSE* otherwise
  - Boolean operator NOT returns *TRUE* if its operand is false and *FALSE* if its operand is true

jeonghun.park@knu.ac.kr

Sorted Array

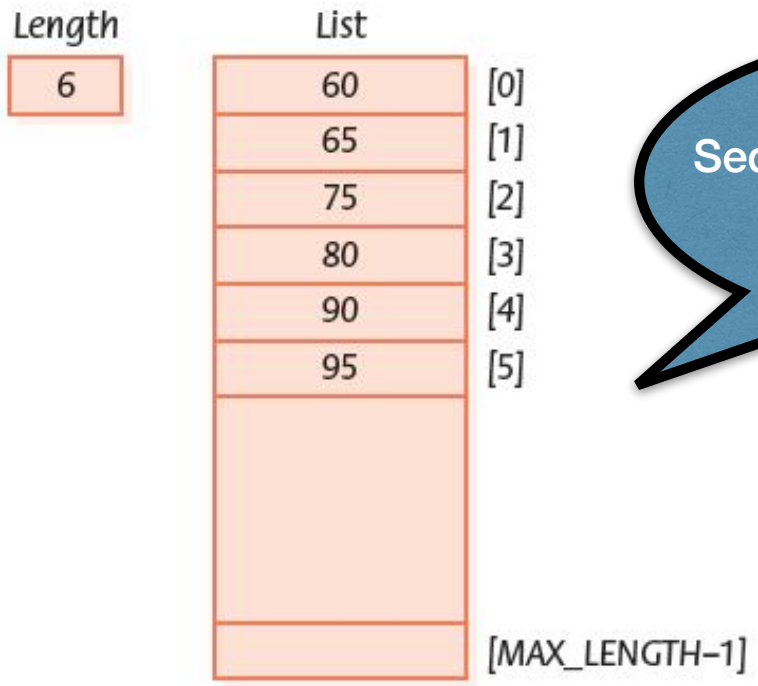


FIGURE 7.8 A sorted array

jeonghun.park@knu.ac.kr

Binary Search Algorithm

- We already know the solution:
  - Binary search algorithm

```
Set first to 0
Set last to length-1
Set found to FALSE
WHILE (first <= last AND NOT found)
    Set middle to (first + last)/ 2
     IF (item equals data[middle]))
         Set found to TRUE
     ELSE
         IF (item < data[middle])
             Set last to middle – 1
  ELSE
  Set first to middle + 1
RETURN found
```


jeonghun.park@knu.ac.kr

Binary Search Details

Length	Items	
11	ant	[0]
	cat	[1]
	chicken	[2]
	cow	[3]
	deer	[4]
	dog	[5]
	fish	[6]
	goat	[7]
	horse	[8]
	rat	[9]
	snake	[10]
	. . .	

FIGURE 7.9 Binary search example

Searching for *cat*

First	Last	Middle	Comparison	
0	10	5	*cat* < *dog*	
0	4	2	*cat* < *chicken*	
0	1	0	*cat* > *ant*	
1	1	1	*cat* = *cat*	**Return: true**

Searching for *fish*

First	Last	Middle	Comparison	
0	10	5	*fish* > *dog*	
6	10	8	*fish* < *horse*	
6	7	6	*fish* = *fish*	**Return: true**

Searching for *zebra*

First	Last	Middle	Comparison	
0	10	5	*zebra* > *dog*	
6	10	8	*zebra* > *horse*	
9	10	9	*zebra* > *rat*	
10	10	10	*zebra* > *snake*	
11	10		*first* > *last*	**Return: false**

FIGURE 7.10 Trace of the binary search

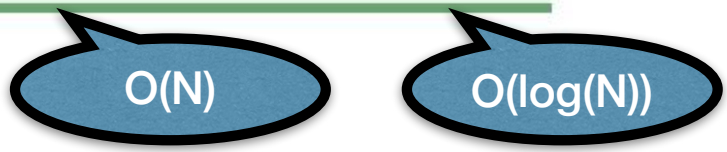
jeonghun.park@knu.ac.kr

Complexity

TABLE 7.1 Average Number of Comparisons

Onion; © matka_Wariatka/ShutterStock, Inc.

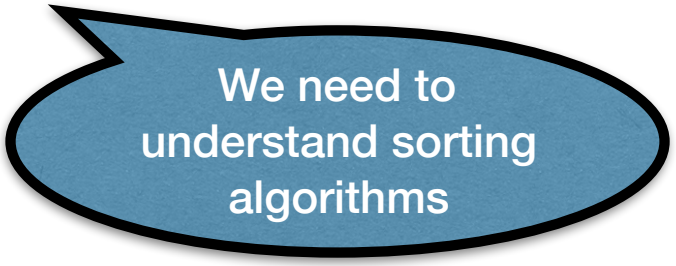
Length	Sequential Search	Binary Search
10	5.5	2.9
100	50.5	5.8
1000	500.5	9.0
10000	5000.5	12.0



jeonghun.park@knu.ac.kr

Sorting

- So in conclusion, sorting is powerful in search algorithm
- One may throw this question:
  - If we have unsorted array, is it better to
  - Directly apply sequential search vs.
  - First sort the array then use binary search



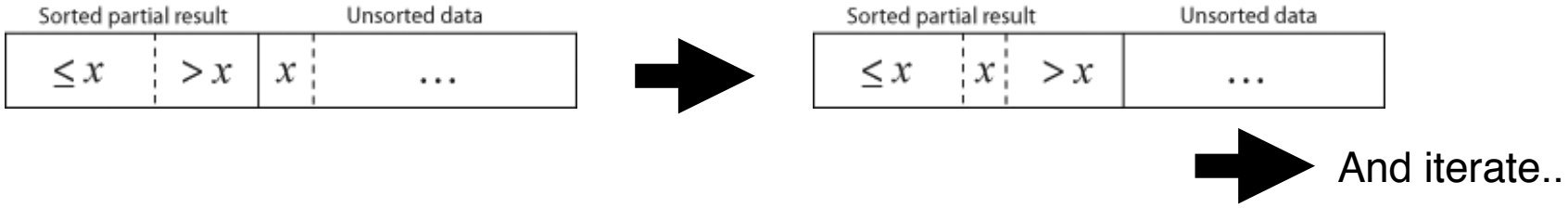
jeonghun.park@knu.ac.kr

Sorting Algorithm

- Sorting
  - Arranging items in a collection so that there is an ordering on one (or more) of the fields in the items
- Sort Key
  - The field (or fields) on which the ordering is based
- Sorting algorithms
  - Algorithms that order the items in the collection based on the sort key

jeonghun.park@knu.ac.kr

Insertion Sort



Try it with
6 5 3 1 8 7 3 2

https://en.wikipedia.org/wiki/File:Insertion-sort-example-300px.gif

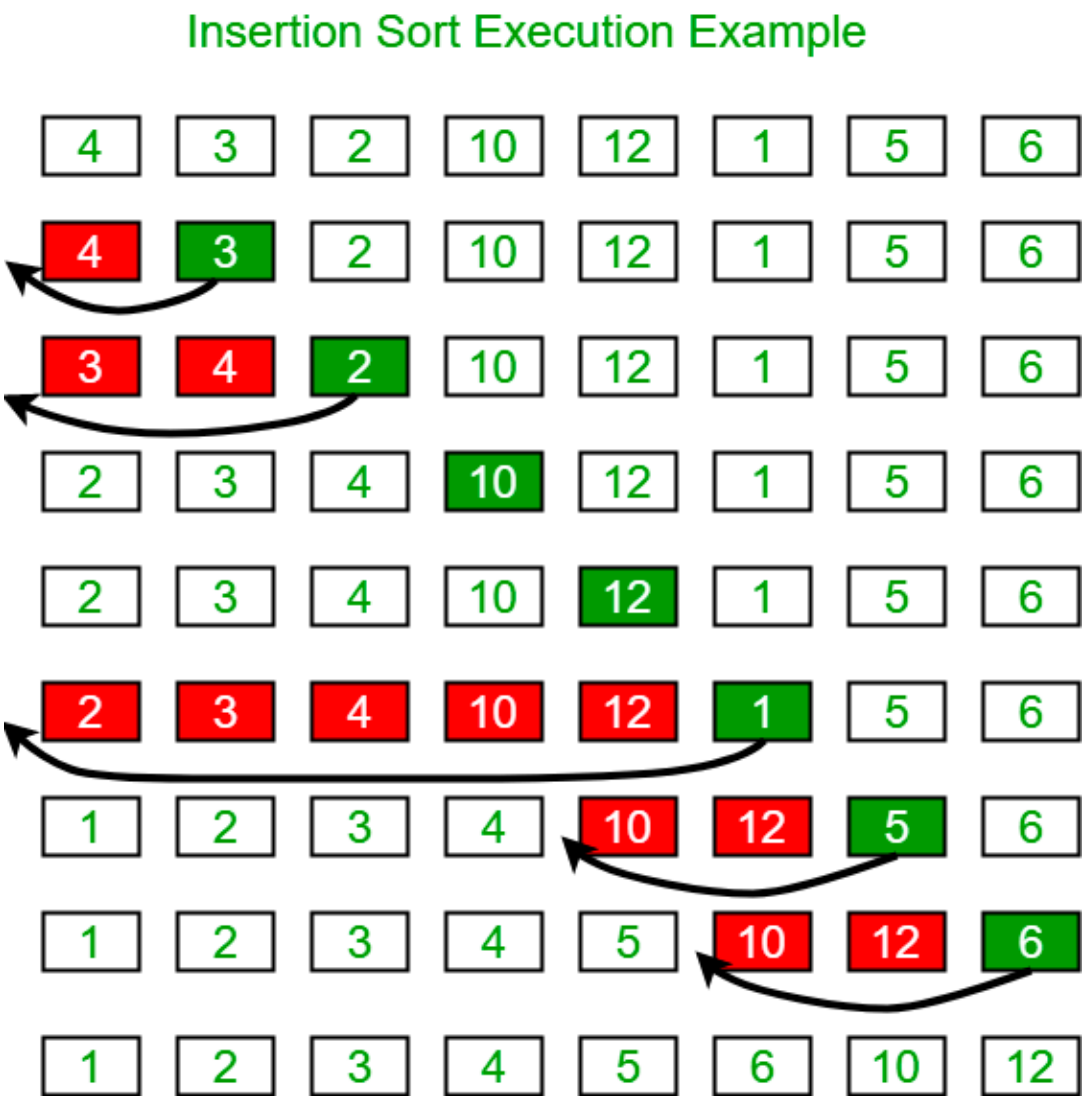
jeonghun.park@knu.ac.kr

Insertion Sort Implementation

```
i ← 1
while i < length(A)
    x ← A[i]
    j ← i - 1
    while j >= 0 and A[j] > x
        A[j+1] ← A[j]
        j ← j - 1
    end while
    A[j+1] ← x[4]
    i ← i + 1
end while
```


jeonghun.park@knu.ac.kr

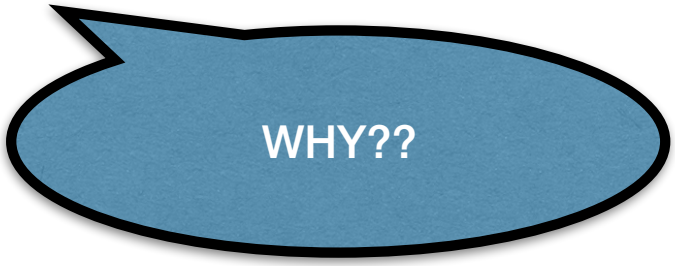
Insertion Sort



jeonghun.park@knu.ac.kr

Complexity

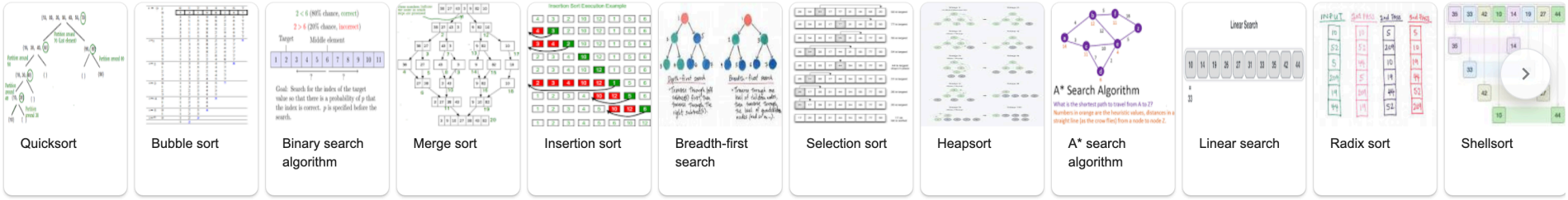
- Best
  - O(N)
- Average
  - $O(N^2)$
- Worst
  - $O(N^2)$
- Memory
  - O(1)

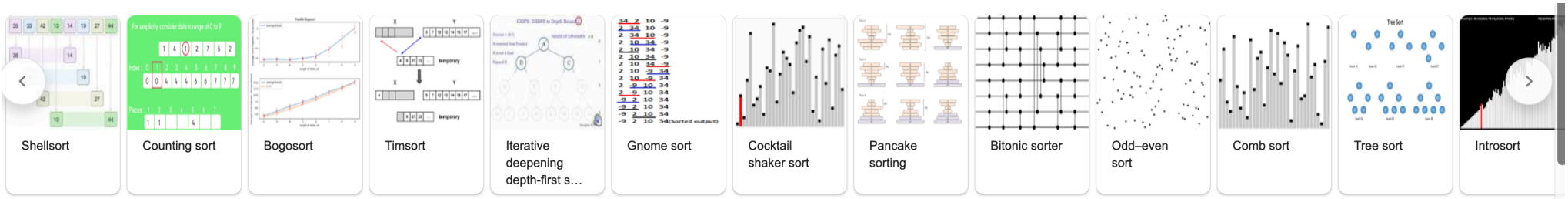


jeonghun.park@knu.ac.kr

Other Sorting Algorithms

sorting algorithms





...

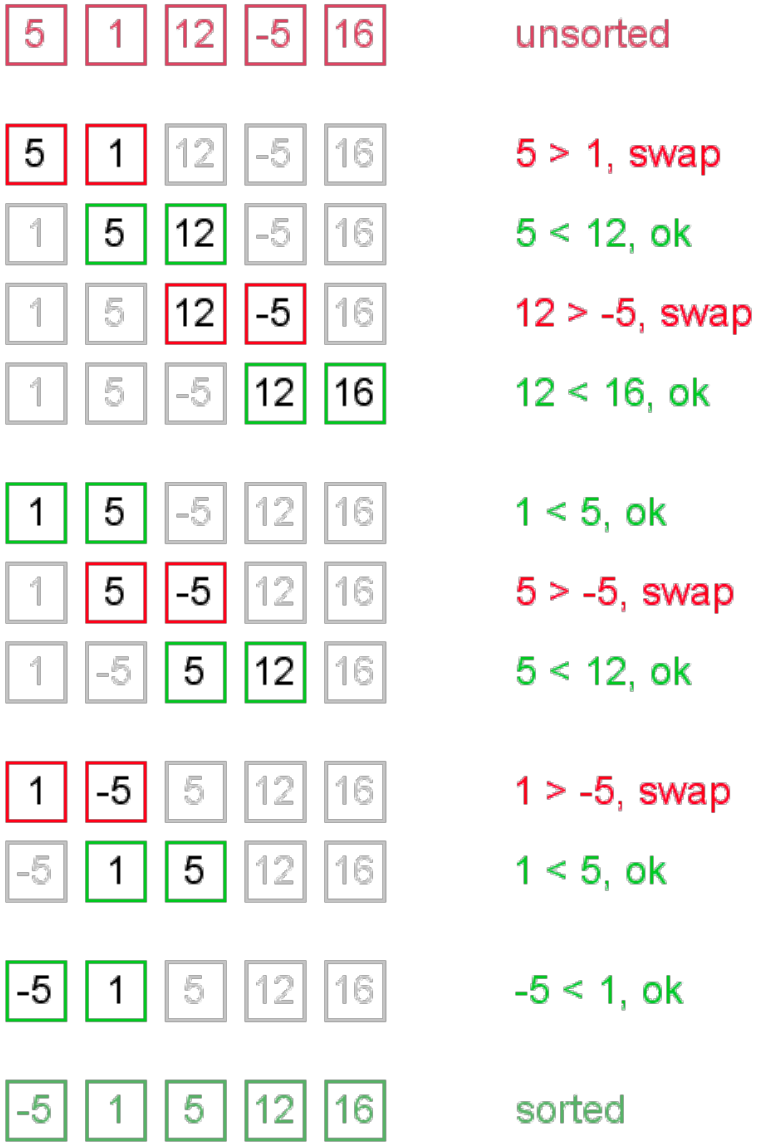
jeonghun.park@knu.ac.kr

Bubble Sort

- A basic approach is same
  - Find the next unsorted item
  - Put it to the proper place
- But it's comparison window size is always 2

jeonghun.park@knu.ac.kr

Bubble Sort



jeonghun.park@knu.ac.kr

Bubble Sort Implementation

```
n ⟵ length(A)
i ⟵ 1
sort_flag ⟵ 1
while (sort_flag == 1)
  sort_flag ⟵ 0
  while (i <= n-1)
    if A[i] > A[i+1]
      x ⟵ A[i+1]
      y ⟵ A[i]
      A[i] ⟵ x
      A[i+1] ⟵ y
      sort_flag ⟵ 1
    else
    end
    i ⟵ i+1
end while
```


jeonghun.park@knu.ac.kr

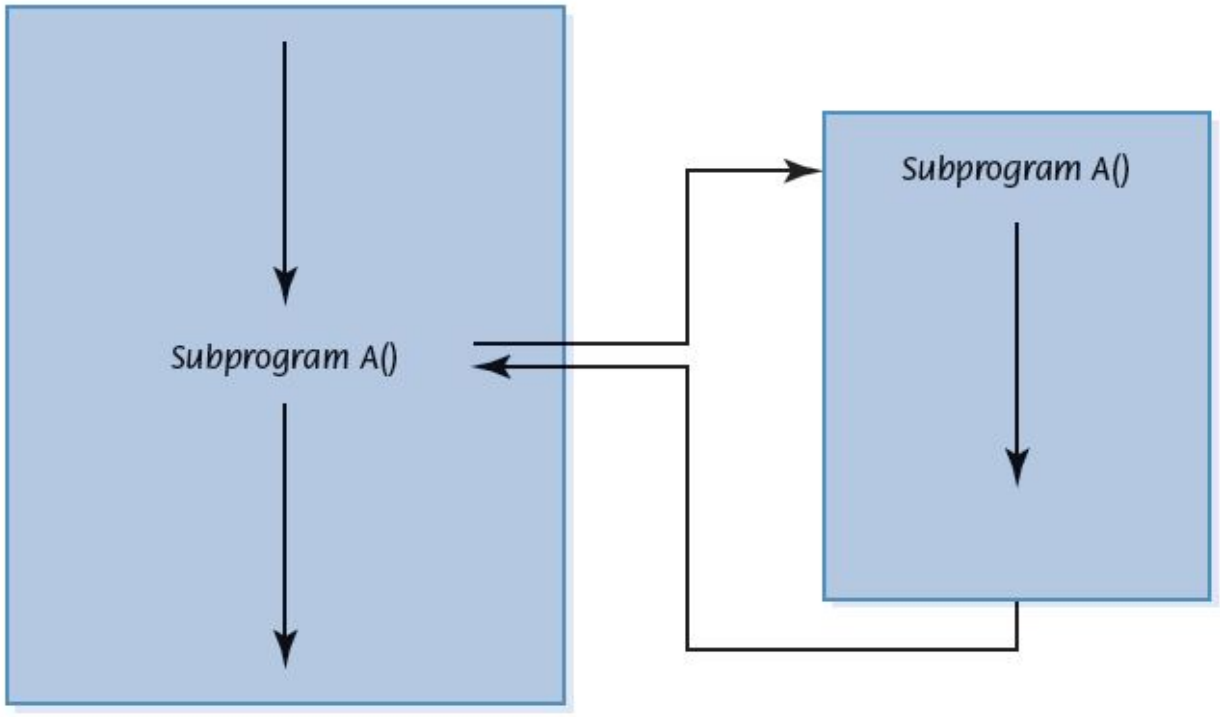
Is it better?

- What is the worst complexity of bubble sort?

jeonghun.park@knu.ac.kr

Subprogram Statement

(a) Subprogram A does its task and calling unit continues with next statement



jeonghun.park@knu.ac.kr

Recursion

- The ability of subprogram to call itself
- For example,
  - We want to calculate N! = N * (N – 1)!
  - Base case: Factorial(0) = 1 (0! = 1)
  - General case: Factorial(N) = (N-1)*Factorial(N-1)

jeonghun.park@knu.ac.kr

Recursion

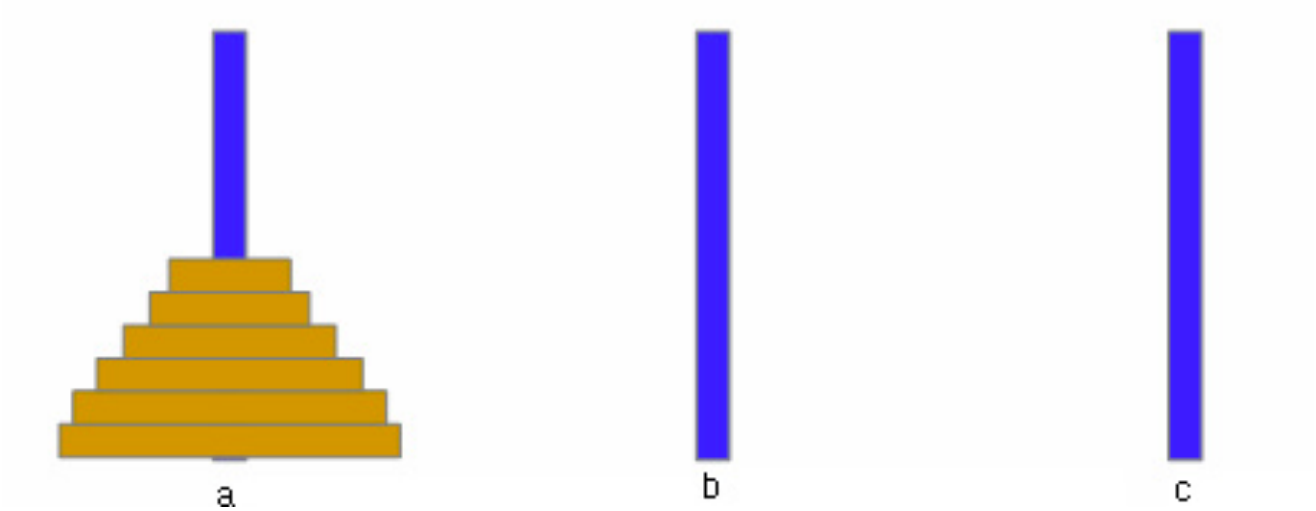
```
Write "Enter n"
Read n
Set result to Factorial(n)
Write result + " is the factorial of " + n

Factorial(n)
IF (n equals 0)
      RETURN 1
ELSE
      RETURN n * Factorial(n-1)
```


jeonghun.park@knu.ac.kr

Recursion

- Tower of Hanoi Problem



Move the disks in “a” to “c” with the same order of “a”

jeonghun.park@knu.ac.kr

Recursion

- Pseudocode

```
FUNCTION MoveTower(disk, source, dest, spare):
IF disk == 0, THEN:
    move disk from source to dest
ELSE:
    MoveTower(disk - 1, source, spare, dest)   // Step 1 above
    move disk from source to dest              // Step 2 above
    MoveTower(disk - 1, spare, dest, source)   // Step 3 above
END IF
```


jeonghun.park@knu.ac.kr

Quicksort

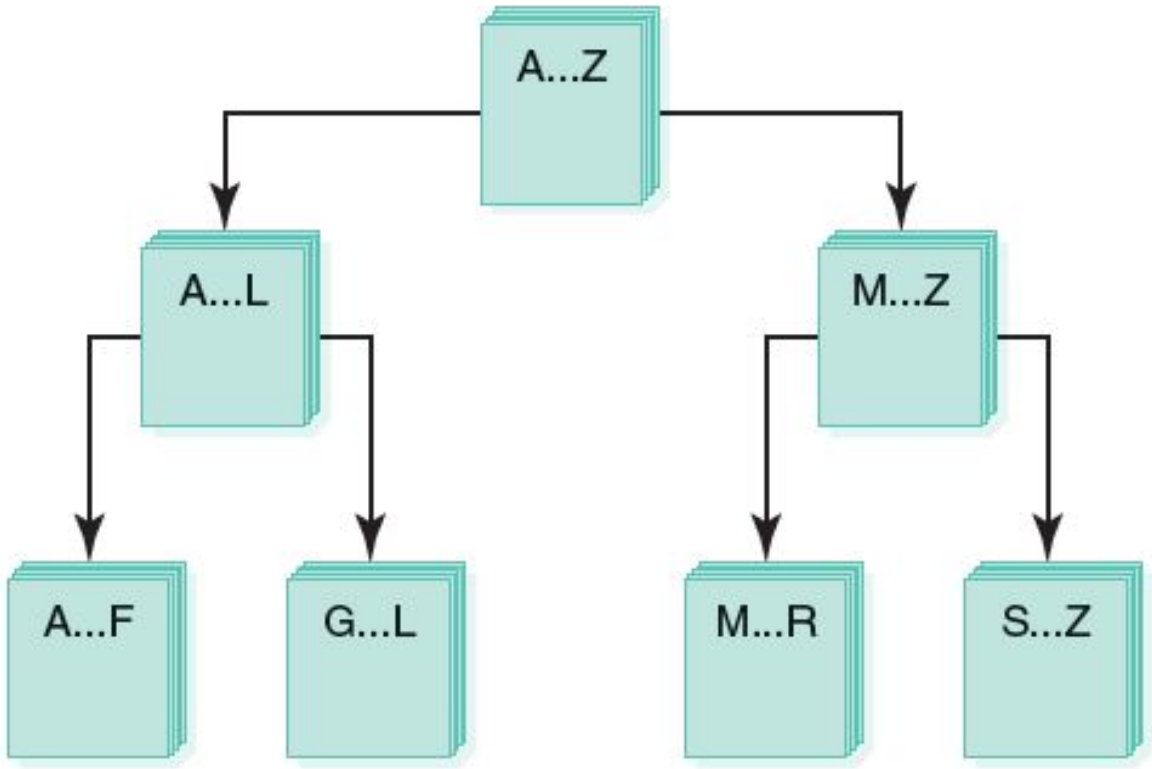


FIGURE 7.15 Ordering a list using the Quicksort algorithm

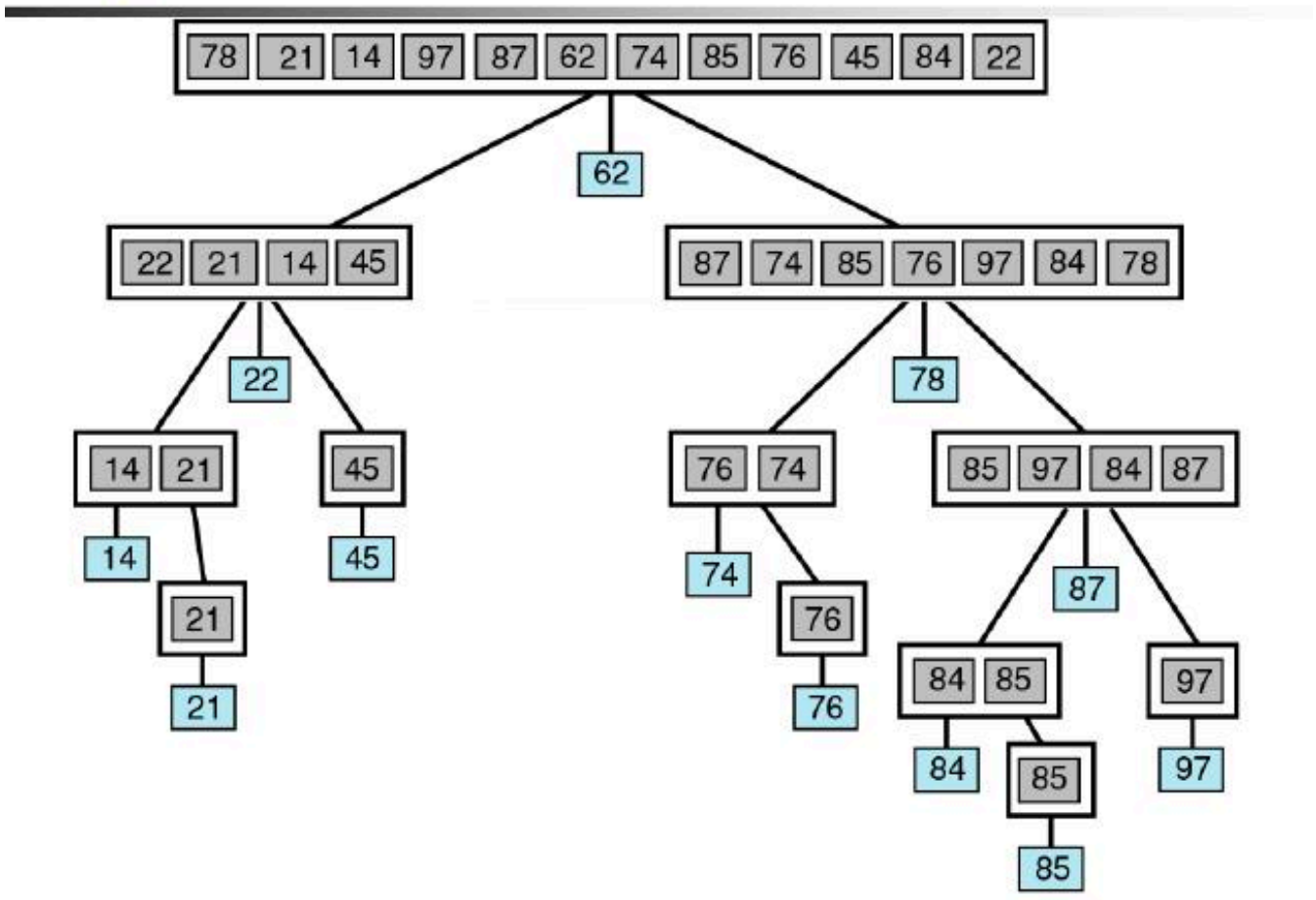
jeonghun.park@knu.ac.kr

Quicksort

- Pick an element, called a *pivot*, from the array.
- *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
- Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

jeonghun.park@knu.ac.kr

Quicksort



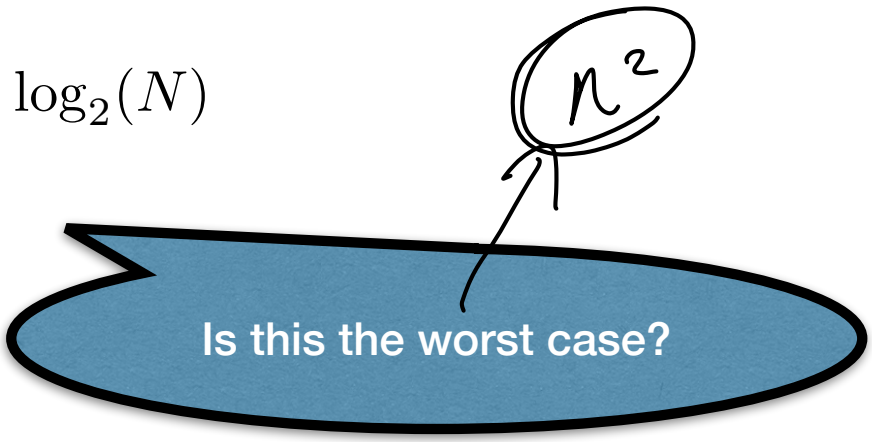
jeonghun.park@knu.ac.kr

Quicksort Complexity

- Assume that we have N size array, and T(N) is the complexity.
- Then we have:
  - $T(N) = 2T(N/2) + N$
    $= 2(2T(N/4) + N/2) + N$
    $= 4T(N/4) + 2N$

    $\vdots$

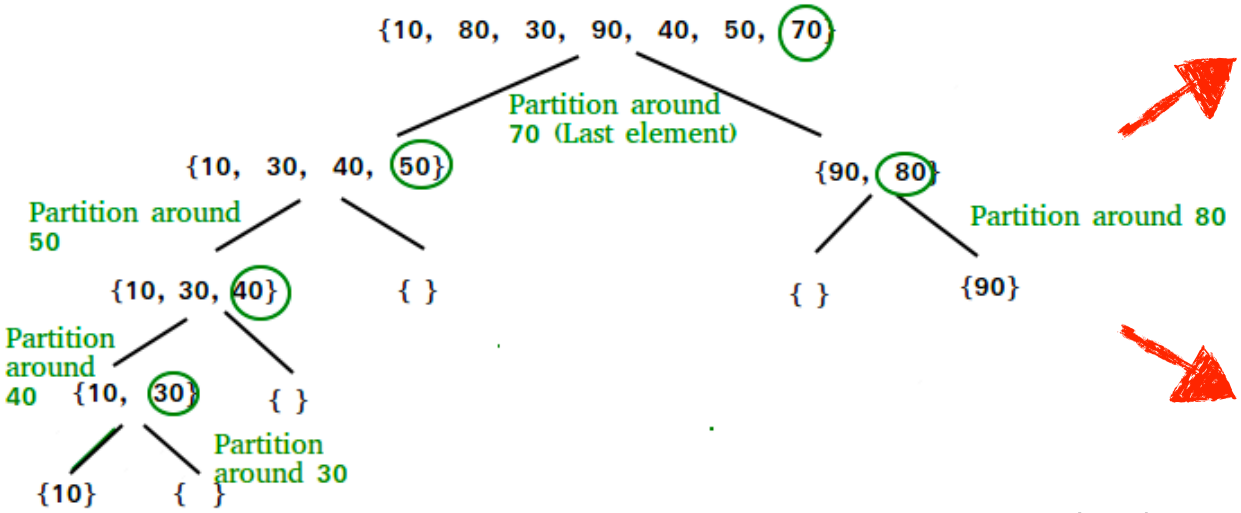
    $= 2^x T(N/2^x) + xN,\ x = \log_2(N)$
    $= NT(1) + N\log_2(N)$
    $\approx O(N\log_2(N))$

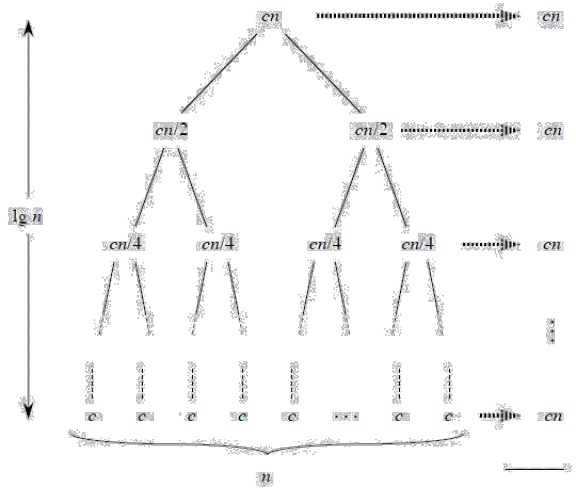


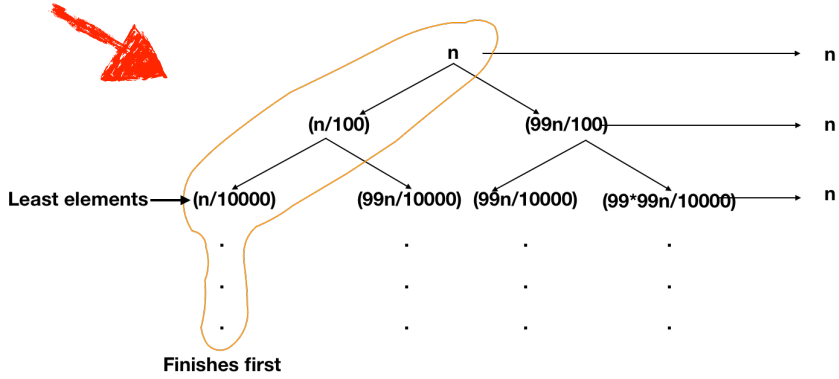
jeonghun.park@knu.ac.kr

Quicksort Implementation

- We can do it by using recursion

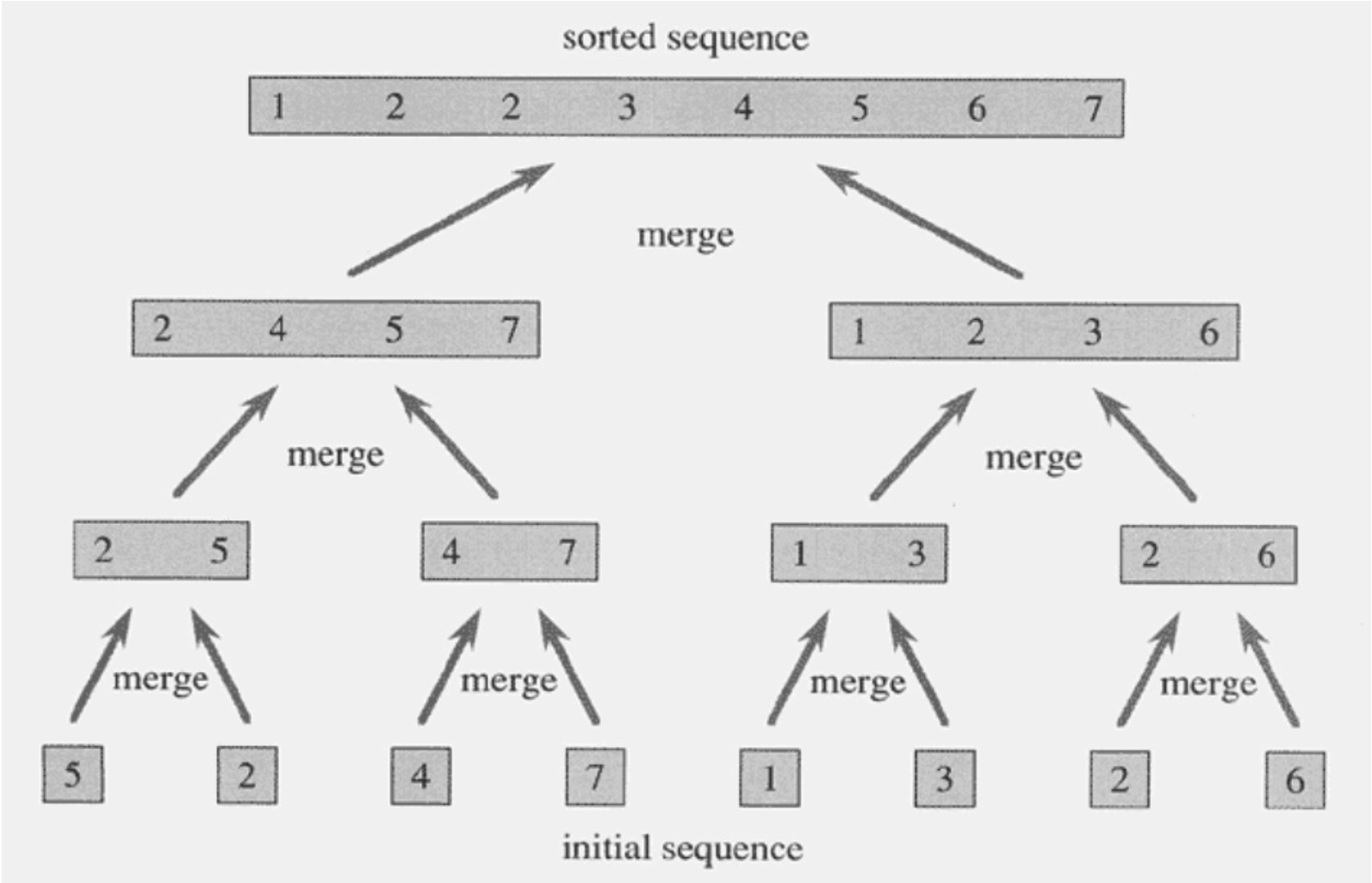






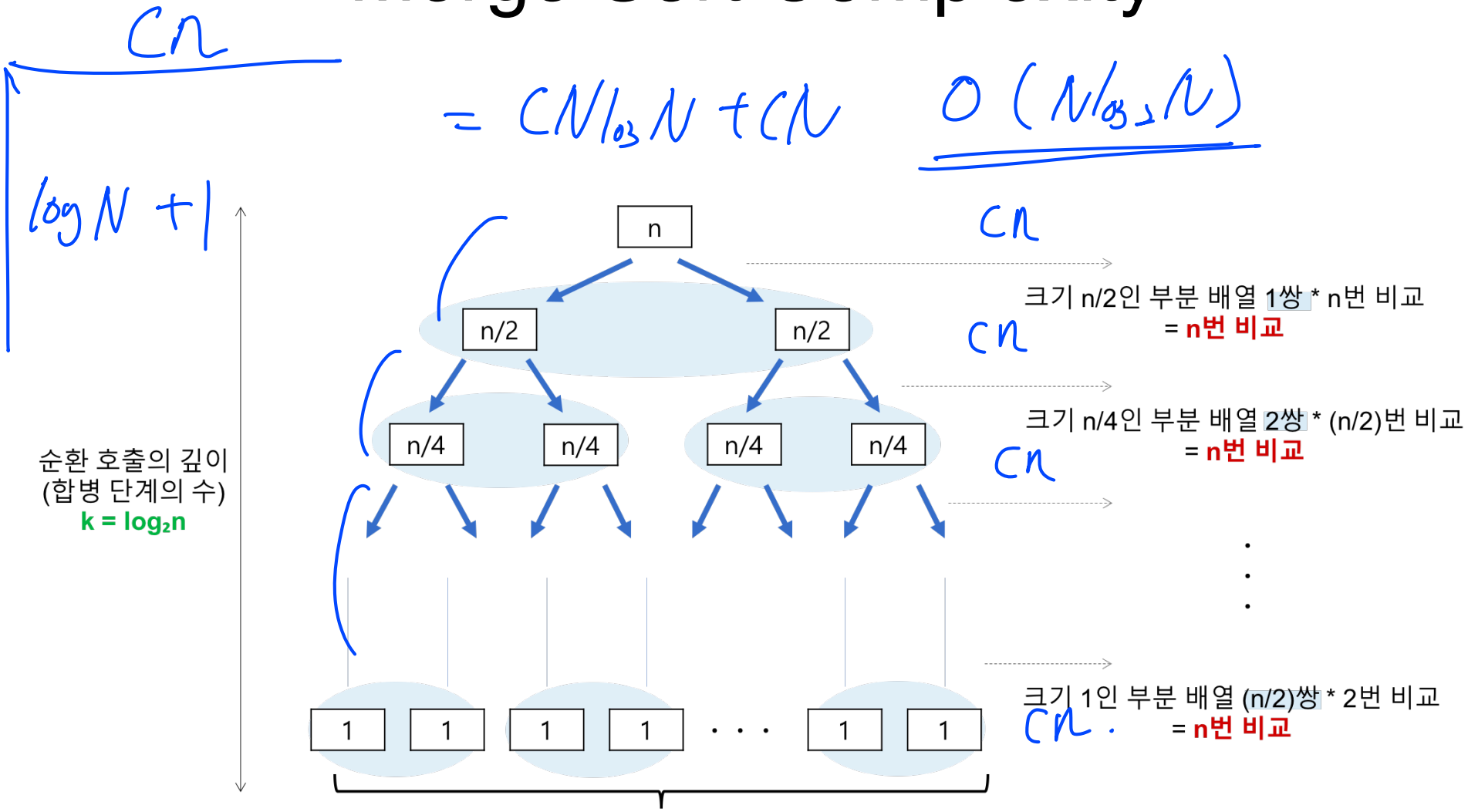
jeonghun.park@knu.ac.kr

Merge Sort



jeonghun.park@knu.ac.kr

Merge Sort Complexity



$$= O(N \log_2(N))$$

Can we prove it analytically?

jeonghun.park@knu.ac.kr