CSE3081 Design and Analysis of Algorithms

Dept. of Computer Engineering,
Sogang University

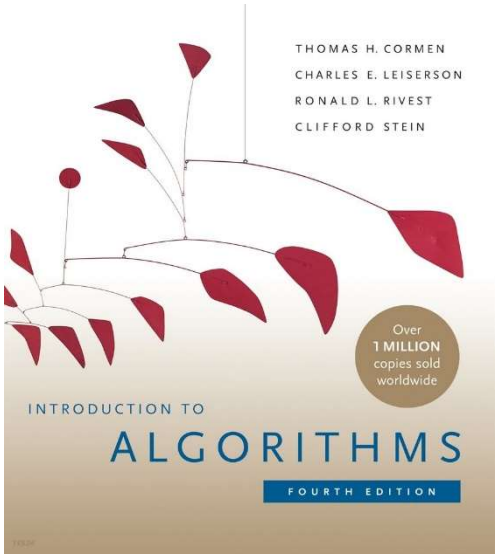
Course Intro

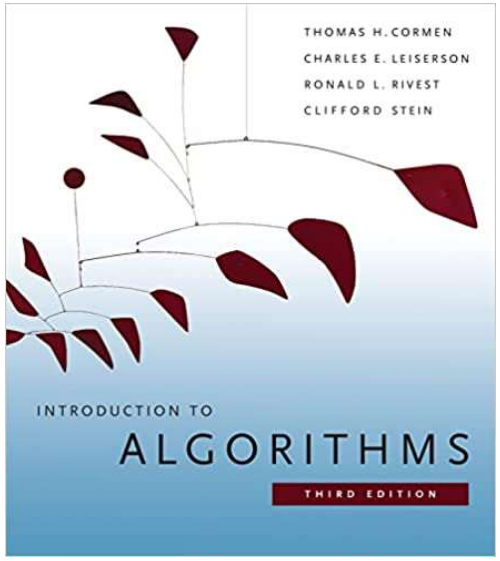
- Course objective
  - Learn how to design/analyze/implement algorithms in order to efficiently solve problems using computers
  - Specifically,
    - Learn various well-known algorithms that are used to solve computer science problems
    - Learn to implement algorithms using programming language such as C/C++
    - Learn to evaluate the design and its implementation

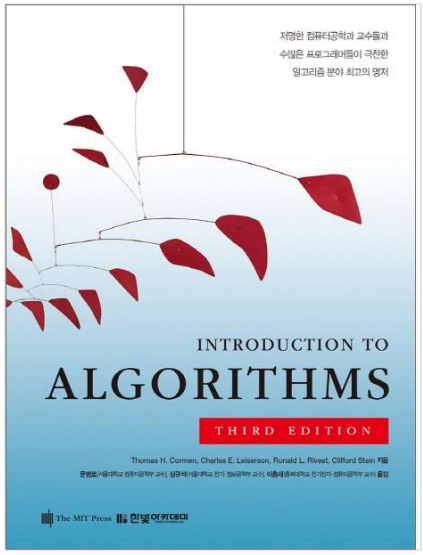
- Prerequisites
  - CSE3080 Data Structures
  - You need to have an intermediate level of C/C++ programming.

Textbook

- Introduction to Algorithms, Fourth Edition
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein







- Lecture slides will cover all of the necessary materials.
- You may use the book as a supplementary material.

Topics covered

- Definition of algorithm
- Characterizing performance of algorithms
- Divide-and-Conquer
- Randomized Algorithms
- Sorting Algorithms
- Dynamic Programming
- Greedy Algorithms
- Advanced Data Structures
- Graph Algorithms
- Linear Programming
- Machine-Learning Algorithms
- etc.

Evaluation

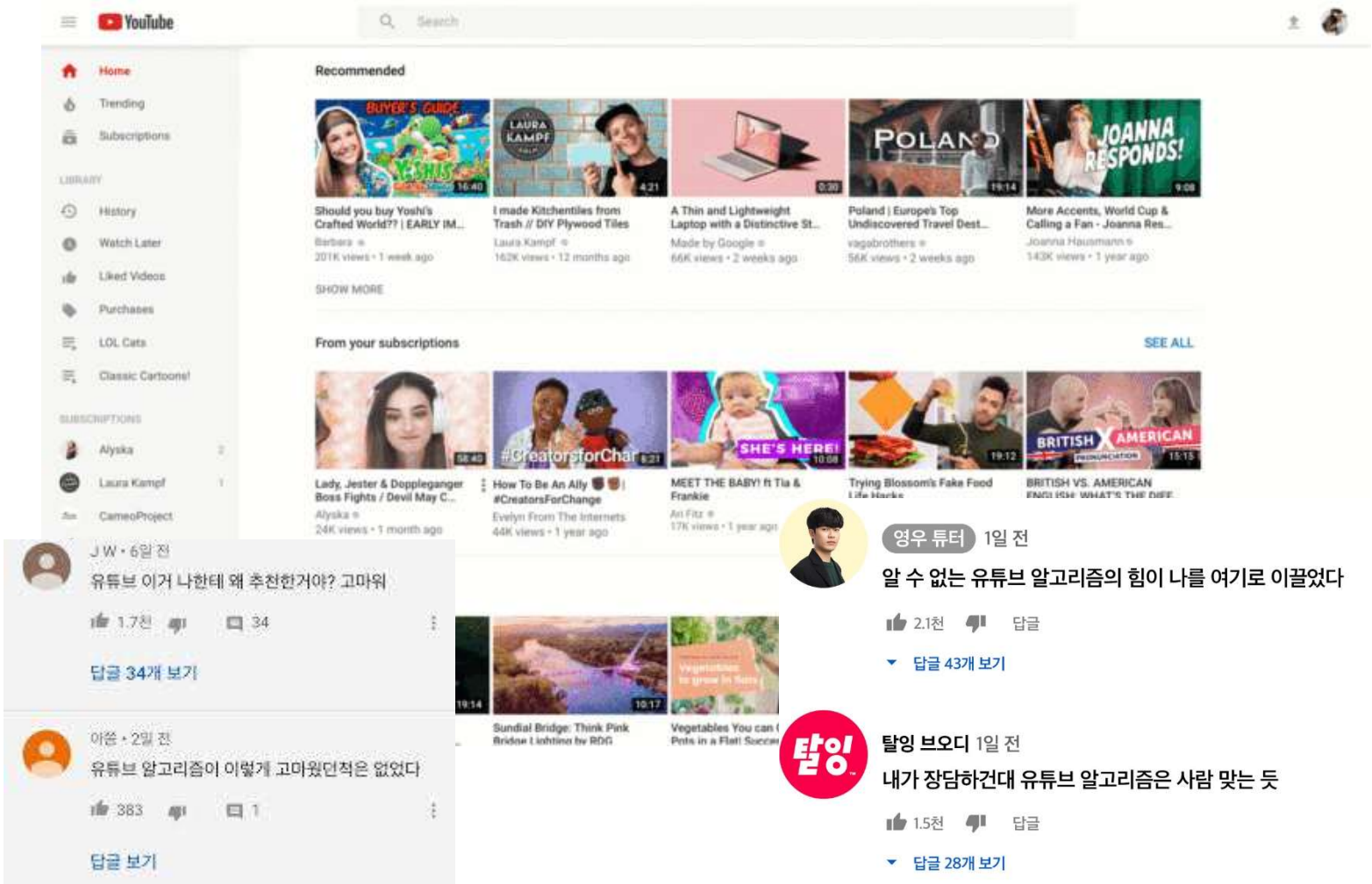
- Mid-term exam (30%)
- Final exam (30%)
- 3-4 Programming assignments (30%)
- Others (10%)
  - participation, quizzes if there is any

- Notes on programming assignments
  - Copying is NOT allowed
    - Code and documents are checked using copy-checking software.
    - If duplicates are found, ALL of them will receive zero score.
    - What's the meaning of taking the course if we copy others' work to get scores?

Chapter 1.
The Role of Algorithms in Computing

1.1 Algorithms

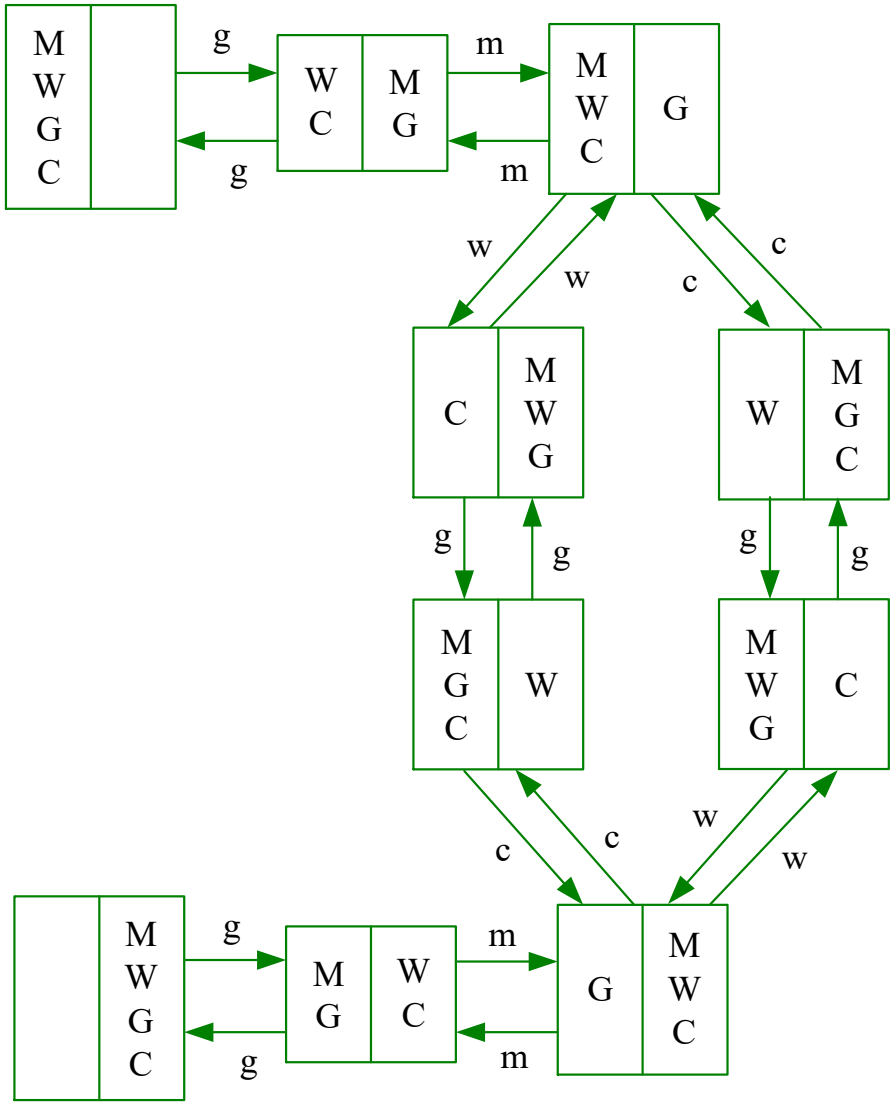
알고리즘?


YouTube
Search
Home
Trending
Subscriptions
LIBRARY
History
Watch Later
Liked Videos
Purchases
LOL Cats
Classic Cartoons!
SUBSCRIPTIONS
Alyska
Laura Kampf
CameoProject
Recommended
Should you buy Yoshi's Crafted World?? | EARLY IM...
I made Kitchentiles from Trash // DIY Plywood Tiles
A Thin and Lightweight Laptop with a Distinctive St...
Poland | Europe's Top Undiscovered Travel Dest...
More Accents, World Cup & Calling a Fan - Joanna Res...
SHOW MORE
From your subscriptions
SEE ALL
Lady, Jester & Doppleganger Boss Fights / Devil May C...
How To Be An Ally | #CreatorsForChange
MEET THE BABY! ft Tia & Frankie
Trying Blossom's Fake Food
BRITISH VS. AMERICAN
Sundial Bridge: Think Pink
J W • 6일 전
유튜브 이거 나한테 왜 추천한거야? 고마워
1.7천
34
답글 34개 보기
이쓴 • 2일 전
유튜브 알고리즘이 이렇게 고마웠던적은 없었다
383
1
답글 보기
영우 튜터 1일 전
알 수 없는 유튜브 알고리즘의 힘이 나를 여기로 이끌었다
2.1천
답글
답글 43개 보기
탈잉 브오디 1일 전
내가 장담하건대 유튜브 알고리즘은 사람 맞는 듯
1.5천
답글
답글 28개 보기

Example Problem: Crossing the river

- A fisherman wants to cross a river with a wolf, a goat, and a cabbage. The fisherman can only take one more on the boat, when crossing the river. If the goat and the cabbage are left on one side, the goat will eat the cabbage. If the wolf and the goat are left on one side, the wolf will eat the goat. How can the fisherman take all three safely across the river, with the minimum number of crossings?

Analyzing the problem



Finding out the solution

- We can represent "states" as vertices
  - States: a snapshot of two sides of the river

- If we can go from one state to another state by a single step (fisherman crossing the river with one other object), we connect the two vertices with an edge

- The problem can be represented as a graph!

- Now what algorithm should we apply to solve this problem?

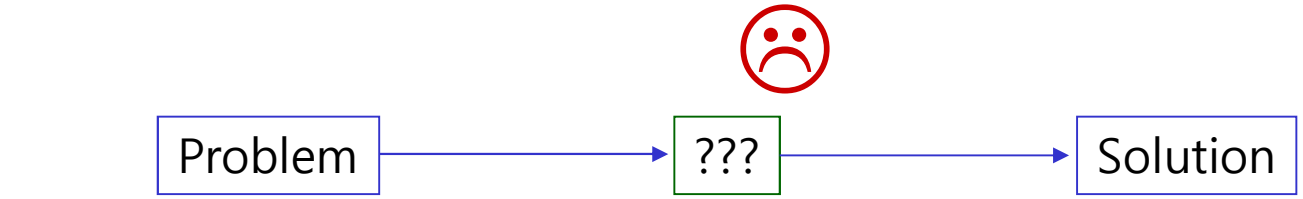
Algorithm: Informal Definitions

- A well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output in a finite amount of time.

- A sequence of computational steps that transform the input into the output.

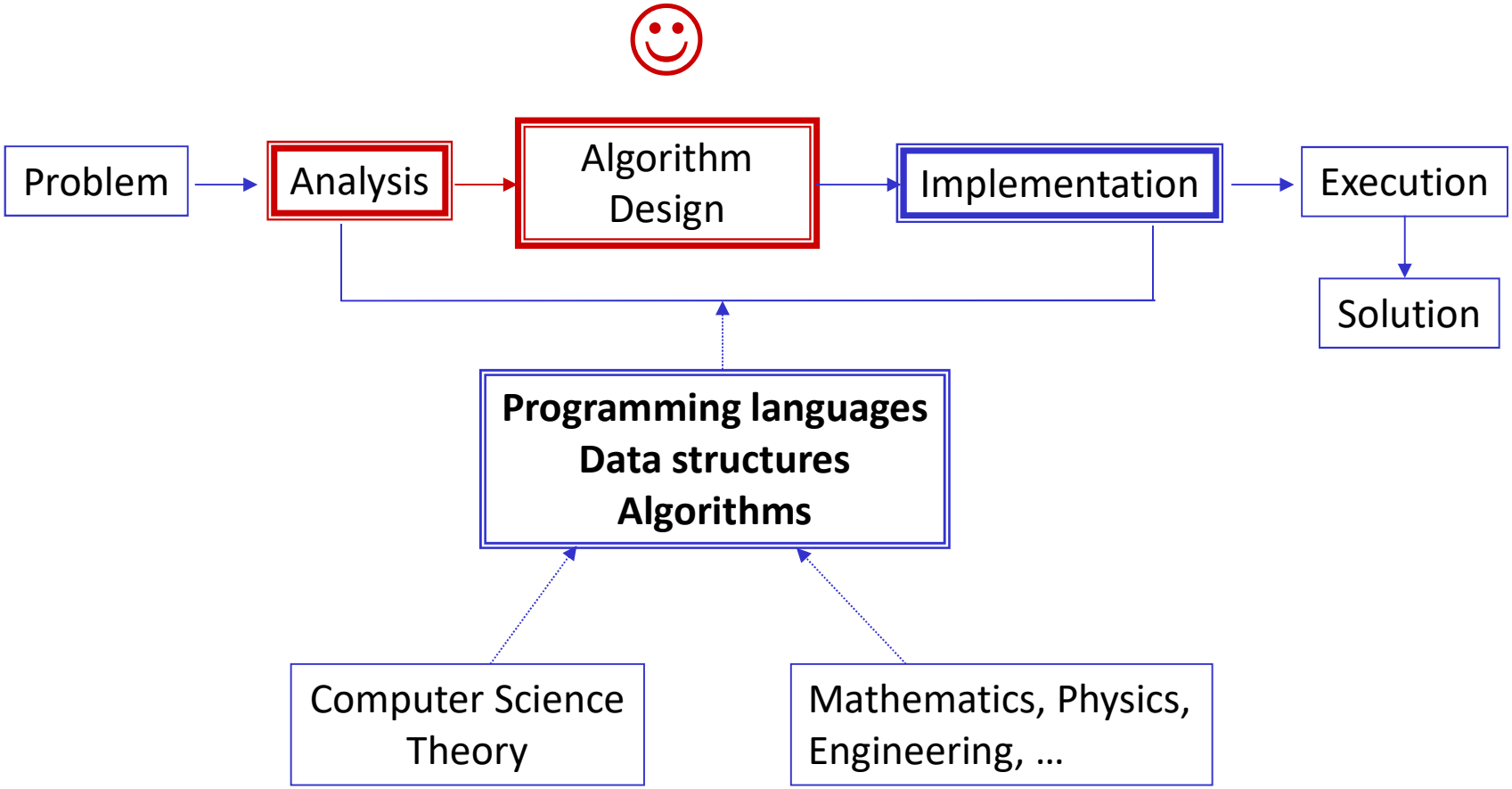
- A tool for solving a well-defined computational problem.

Problem solving in computer science



Unmanned vehicles
Document writing
Home banking
Internet newspapers
Document translations
Circuit design
Gene analysis
Online games
Video editing
Speech recognition
Building design
Weather forecast
Stock market prediction
Etc.

Problem solving pipeline



Algorithm: Example

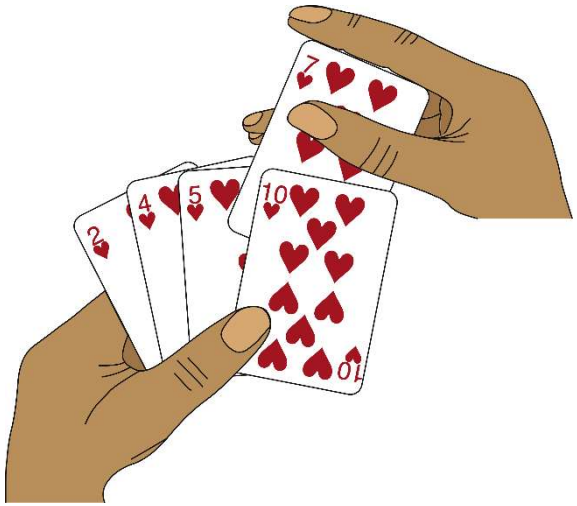
- We need to sort a sequence of numbers into monotonically increasing order.

- We can define the sorting problem as follows.

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

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

- For example,
  - If the input is <31, 41, 59, 26, 41, 48>, then
  - the output shall be <26, 31, 41, 41, 58, 59>.



- In this case, the input <31, 41, 59, 26, 41, 48> is called an instance of the problem.
  - There can be many instances of the problem

Algorithm: Example

- Sorting is one of the fundamental operations in computer science
- A large number of sorting algorithms exist
  - insertion sort
  - selection sort
  - bubble sort
  - merge sort
  - heap sort
  - quick sort
  - etc.

Comparison sorts

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space.[5][6]
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm).[7]
In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging.[8]
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have d inversions.
Block sort	n	$n \log n$	$n \log n$	1	Yes	Insertion & Merging	Combine a block-based $O(n)$ in-place merge algorithm[9] with a bottom-up merge sort.
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes n-1 comparisons when the data is already sorted or reverse sorted.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space, when using linked lists, or when made as a variant of Insertion Sort instead of swapping the two items.[10]
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes n-1 comparisons when the data is already sorted or reverse sorted.
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Exchange sort	n^2	n^2	n^2	1	No	Exchanging	Tiny code size.
Tree sort	$n \log n$	$n \log n$	$n \log n$ (balanced)	n	Yes	Insertion	When using a self-balancing binary search tree.
Cycle sort	n^2	n^2	n^2	1	No	Selection	In-place with theoretically optimal number of writes.
Library sort	$n \log n$	$n \log n$	n^2	n	No	Insertion	Similar to a gapped insertion sort. It requires randomly permuting the input to warrant with-high-probability time bounds, which makes it not stable.
Patience sorting	n	$n \log n$	$n \log n$	n	No	Insertion & Selection	Finds all the longest increasing subsequences in $O(n \log n)$.
Smoothsort	n	$n \log n$	$n \log n$	1	No	Selection	An adaptive variant of heapsort based upon the Leonardo sequence rather than a traditional binary heap.
Strand sort	n	n^2	n^2	n	Yes	Selection	
Tournament sort	$n \log n$	$n \log n$	$n \log n$	n[11]	No	Selection	Variation of Heapsort.
Cocktail shaker sort	n	n^2	n^2	1	Yes	Exchanging	A variant of Bubblesort which deals well with small values at end of list
Comb sort	$n \log n$	n^2	n^2	1	No	Exchanging	Faster than bubble sort on average.
Gnome sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.
Odd-even sort	n	n^2	n^2	1	Yes	Exchanging	Can be run on parallel processors easily.

https://en.wikipedia.org/wiki/Sorting_algorithm

- Which algorithm is the best for a given application? It depends on
  - number of items to be sorted
  - the extent to which the items are already somewhat sorted,
  - possible restrictions on the item values
  - the architecture of the computer
  - the kind of storage devices to be used, etc.

Algorithm: Correctness

- An algorithm for a computational problem is correct if,
  - for every problem instance provided as input, it halts
    - finishes its computing in finite time
  - outputs the correct solution to the problem instance.

- A correct algorithm is said to solve the given computational problem.

- An incorrect algorithm might not halt on some input instances, or might halt with incorrect answer.

- Incorrect algorithms may still be useful, if you can control their error rate.
  - The output may be wrong, but at least similar to the correct answer.
  - It is much faster than a correct algorithm.

Algorithm: Problems

- Examples of computational problems that can be solved by algorithms
  - The Human Genome Project
    - Identifying all the roughly 30,000 genes in human DNA
    - Determining the sequences of the roughly 3 billion chemical base pairs
    - Storing this information in databases
    - Developing tools for data analysis
      - Determining similarity between DNA sequences
  - Internet
    - finding good routes on which the data travels
    - using a search engine to quickly find pages on which particular information resides
  - Electronic Commerce
    - public cryptography and digital signatures

Algorithm: Problems

- Examples of computational problems that can be solved by algorithms

  - Other applications
    - allocating scarce resources in the most beneficial way
    - determining where to spend money for advertising commercial products
    - assigning crews to flights in the least expensive way possible
    - determining the shortest route from one intersection to another
    - given a library of mechanical parts, list them in order so that each part appears before any part that uses it
    - determine whether an image represents a cancerous tumor or a benign one
    - compressing a large file into a small one

Algorithm: Solutions

- There are many existing algorithms for various computational problems
  - sorting algorithms
  - shortest path algorithms
  - minimum spanning tree algorithms
  - etc.

- You can map a problem into one of the known problems and use existing algorithms to solve it.
  - Finding the best route from home to work (navigation) → convert the roads and intersections as edges and vertices in a graph and use a shortest-path algorithm.

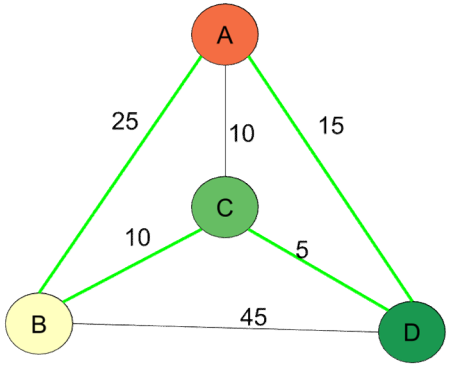
- There are also well established "techniques" that can be used to design new algorithms
  - divide-and-conquer
  - dynamic programming
  - etc.

Related Concepts: Data Structures

- A data structure is a way to store and organize data in order to facilitate access and modifications.
- Using appropriate data structure is an important part of algorithm design.
- No single data structure works well for all purposes, and so you should know the strength and limitations of several of them.
- Examples: arrays, linked lists, stacks, queues, binary trees, B-trees, B+-trees, red-black trees, heaps, disjoint sets, graphs, etc.

Related Concepts: Hard Problems

- There are known problems for which we do not have a fast algorithm.
  - They are called NP-complete problems.
  - There is no proof that a fast algorithm does not exist for these problems, but we still do not have it.
  - If your problem can be converted into a known NP-complete problem, your problem is also a hard problem.
  - If you find a solution to a particular NP-complete problem, you are solving a large set of NP-complete problem!
- Example: traveling salesman problem
  - Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?"



Related Concepts: Alternative Computing Models

- Parallel computing
  - Multi-core processors and GPUs
  - Algorithms can be designed with parallelism in mind for better performance.

- Online algorithms
  - In some real-world problems, the input is not available when the algorithm begins.
  - Instead, the input arrives over time, and the algorithm must decide how to proceed without knowing what data will arrive in the future.
  - In a data center, jobs are constantly arriving and departing, and a scheduling algorithm must decide when and where to run a job, without knowing what jobs will be arriving in the future.
  - Traffic must be routed in the internet based on the current state, without knowing about where traffic will arrive in the future.

1.2 Algorithms as a technology

Algorithm is a Technology

- We consider these as technologies
  - advanced computer architectures and fabrications technologies
  - easy-to-use, intuitive, graphical user interfaces (GUIs)
  - object-oriented systems
  - integrated web technologies
  - fast networking, both wired and wireless
  - machine learning
  - mobile devices

- Algorithm is also a technology
  - Fast hardware rely on hardware design algorithms
  - Networking relies on routing algorithms
  - To run a program written in high-level programming language, it must be processed by a compiler, interpreter, or assembler, all of which make extensive use of algorithms.
  - Machine learning itself is a collection of algorithms

Efficiency

- Different algorithms devised to solve the same problem often differ dramatically in their efficiency.

- Example: Sorting algorithms
- Insertion sort takes time roughly equal to $c_1 n^2$ to sort n items.
  - c_1 is a constant.
  - it takes time roughly proportional to n^2.

- Merge sort takes roughly equal to $c_2 n \log_2 n$ to sort n items.
  - c_2 is a constant.
  - it takes time roughly proportional to $n \log_2 n$.

- As n grows large, merge sort performs much faster than insertion sort
  - When $n = 1000$, $\log_2 n = 10$.
  - When $n = 1000000$, $\log_2 n = 20$.

Chapter 2.
Getting Started

2.1 Insertion Sort

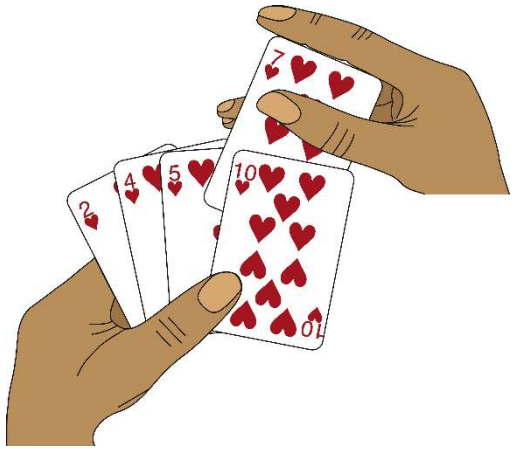
The First Problem: Insertion Sort

- Problem definition

Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$.

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

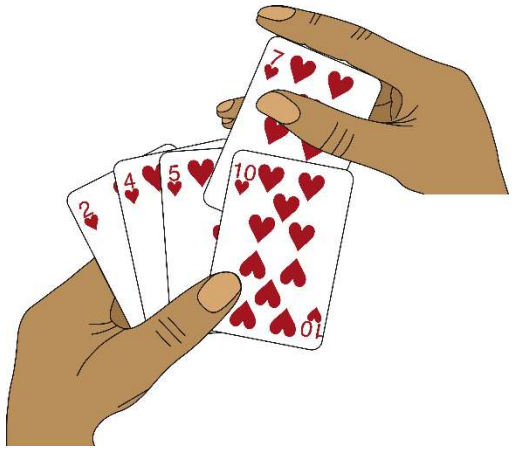
- keys: the numbers to be sorted
  - student ID
- satellite data: data associated with key
  - age, gender, birthday, GPA, etc.
- record: keys + satellite data



Pseudocode

- Algorithms can be described as procedures written in a pseudocode.
  - Similar to programming language such as C, C++, Java, or Python.
  - English phrases or sentences are used to describe procedures.

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

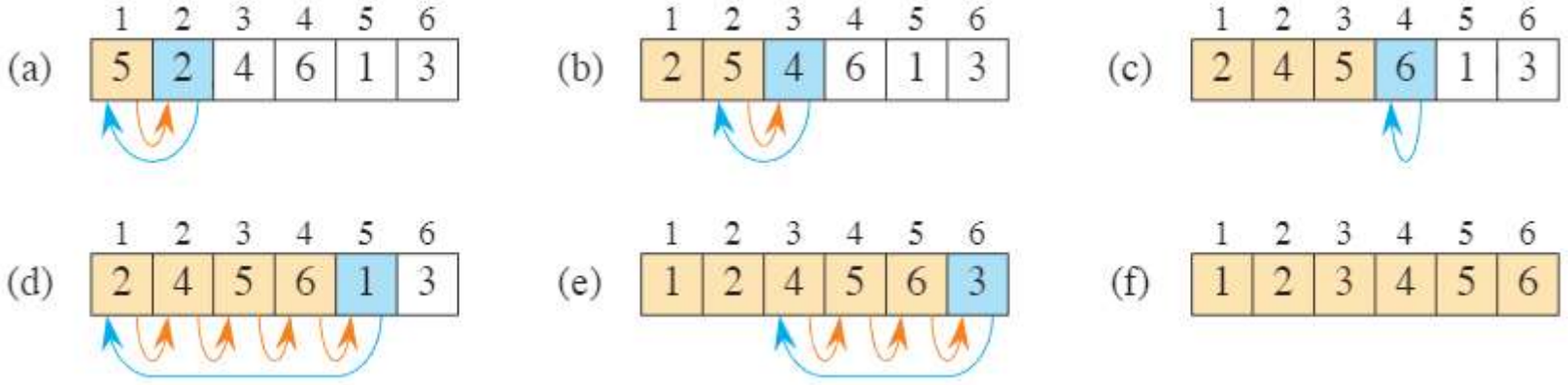


Insertion Sort Algorithm

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- A[1:i-1] is a subarray that contains A[1] through A[i-1].

- Example input: <5, 2, 4, 6, 1, 3>



Insertion Sort Algorithm

- At the start of each iteration of the for loop, the subarray A[1:i-1] consists of the elements originally in A[1:i-1], but in sorted order.
  - A[1:i-1] is said to have loop invariant property.

- We can prove the correctness of the algorithm using the loop invariant property.
  - Initialization: It is true prior to the first iteration of the loop.
    - When i=2, we are considering only the first element.
    - Since it is sorted, it is true.
  - Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration
    - Assume that when i=k, A[1:k-1] contains elements originally in A[1:k-1] and is sorted.
    - Then, after the loop, A[1:k] contains elements originally in A[1:k] and is sorted.
  - Termination: The loop terminates, and when it terminates, the invariant gives us a useful property that helps show that the algorithm is correct
    - When the loop is done, A[1:n] contains elements originally in A[1:n] and is sorted.
    - Problem solved.

Pseudocode Conventions (1)

- Indentation indicates block structure
  - for, while, repeat-until, if-else

```
INSERTION-SORT(A, n)
1  for i = 2 to n
2      key = A[i]
3      // Insert A[i] into the sorted subarray A[1 : i − 1].
4      j = i − 1
5      while j > 0 and A[j] > key
6          A[j + 1] = A[j]
7          j = j − 1
8      A[j + 1] = key
```

- The loop counter retains its value after the loop is exited
  - for i = 2 to n
  - after the loop, i = n+1

Pseudocode Conventions (2)

- The symbol "//" indicates that the remainder of the line is a comment.
- Variables (such as *i*, *j*, and *key*) are local to the given procedure. Global variables are not used without explicit indication.
- A[i] indicates the ith element of the array A.
  - Array index may start from 0 or 1.
- A[i:j] is a subarray of A consisting of the elements A[i], A[i+1], ..., A[j].
- *x.f*: x is an object name, f is an attribute name.

Pseudocode Conventions (3)

- A variable representing an array or object is treated as a pointer (or a reference)
  - If x is an object, setting y = x causes y.f to equal x.f.
  - y points to the same object as x.

- Attribute notation can cascade.
  - x.f.g

- Parameters are passed to a procedure by value.
  - The called procedure receives its own copy of the parameters, and if it assigns a value to a parameter, the change is *not* seen by the calling procedure.

- A **return** statement immediately transfers control back to the point of call in the calling procedure.
  - multiple values can be returned

Pseudocode Conventions (4)

- The Boolean operators "and" and "or" are short circuiting.
  - Expression "x and y" is evaluated by first evaluating x. If x evaluates to FALSE, then the entire expression must be FALSE, and therefore y is not evaluated.
  - If x evaluates to TRUE, then y is evaluated.

- The keyword **error** indicates that an error occurred because conditions were wrong for the procedure to have been called.

2.2 Analyzing Algorithms

Analyzing Algorithms

- Analyzing an algorithm means predicting the resources that the algorithm requires.
  - resources: **computation time**, memory, communication bandwidth, energy consumption

- Assumption: the program runs on a generic one-processor, random-access machine (RAM).
  - Instructions execute one after another, with no concurrent operations
  - Each instruction takes the same amount of time as any other instruction
  - Each data access (reading and writing) takes the same amount of time as any other data access
  - The model contains instructions commonly found in real computers
    - arithmetic: add, subtract, multiply, divide, remainder, floor, ceiling
    - movement: load, store, copy
    - control: conditional and unconditional branch, subroutine call and return
  - Data types: integer, floating point, character
  - Data is represented by bits

Analyzing Algorithms

- We want to predict how long it will take to run a program.
- The actual amount of time depends on various factors such as hardware, operating systems, programming language, etc.

- We can analyze the algorithm itself without considering its implementation.
  - Examine how many times each line of pseudocode is executed.
  - How long each line of pseudocode takes to run.

- We can describe the running time of a program as a function of the size of its input.
  - input size: number of items to sort
    - If we have more items to sort, it is intuitive that the sorting algorithm will take longer time.
  - running time: number of instructions and data accesses executed.

Analyzing INSERTION-SORT

- Let n be the size of input.
- We count the number of times a line in the pseudocode is executed.

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

- Add all these and we get the running time.

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

The Best Case and The Worst Case

- Even for inputs of a given size, the algorithm's running time depends on *which* input of that size is given.

- The best case: the input array is already sorted
  - the while loop of lines 5-7 always exists upon the first test in line 5.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n-1) + c_8(n-1) \\ &= (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

  - the running time is a linear function of n.
- The worst case: the input array is in reverse order
  - the while loop exists only when j reaches 0.

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_4(n-1) + c_5\left(\frac{n(n+1)}{2} - 1\right) \\ &\quad + c_6\left(\frac{n(n-1)}{2}\right) + c_7\left(\frac{n(n-1)}{2}\right) + c_8(n-1) \\ &= \left(\frac{c_5}{2} + \frac{c_6}{2} + \frac{c_7}{2}\right)n^2 + \left(c_1 + c_2 + c_4 + \frac{c_5}{2} - \frac{c_6}{2} - \frac{c_7}{2} + c_8\right)n \\ &\quad - (c_2 + c_4 + c_5 + c_8) \ . \end{aligned}$$

  - the running time is a quadratic function of n.

Worst-Case and Average-Case Analysis

- When analyzing running time, we often focus on the worst-case running time
  - The worst-case running time of an algorithm gives an upper bound on the running time for any input.
  - For some algorithms, the worst case occurs fairly often.
    - searching for an item that is not in a database
  - The "average case" is often roughly as bad as the average case.
    - In insertion sort, suppose we have a random input array.
    - On average, half the elements in A[1:i-1] are less than A[i], and half the elements are greater. Therefore, A[i] is compared with half of the subarray A[1:i-1], and so t_i is about $i/2$.
    - The resulting average-case running time is a quadratic function of the input size.

- Average-case running time
  - In some cases, we are interested in the average-case running time.
  - We consider all input cases, and assume their probability of appearance is equal.

Order of Growth

- The worst-case running time of insertion sort can be expressed as:
  - $T(n) = an^2 + bn + c$
  - for constants a, b, and c

- We can further simplify the running time by focusing on the order of growth (or rate of growth)
  - when n increases, how fast does the running time grow?

- We only consider the leading term of a formula
  - If $T(n) = an^2 + bn + c$, we are only interested in n^2

- To highlight the order of growth, we use the Greek letter Θ.
  - The worst-case running time of insertion sort is $\Theta(n^2)$.
    - It means the running time is roughly proportional to n^2 when n is large.
  - The base-case running time of insertion sort is $\Theta(n)$.

2.3 Designing Algorithms

Algorithm Design Techniques

- Various algorithm design techniques exist.

- Insertion sort uses the incremental method.
  - For each element A[i], insert it into its proper place in the subarray A[1:i], having already sorted the subarray A[1:i-1]

- Many useful algorithms are recursive in structure
  - To solve a given problem, they recurse (call themselves) one or more times to handle closely related subproblems.
  - These algorithms typically follow the divide-and-conquer method.

Divide-and-Conquer

- If the problem is small enough, you call it the "base case".
  - In this case, you just solve it directly without recursing.

- Otherwise, we call it the "recursive case", and we perform the three characteristic steps

- Divide the problem into one or more subproblems that are smaller instances of the same problem.

- Conquer the subproblems by solving them recursively.

- Combine the subproblem solutions to form a solution to the original problem.

Sorting by Divide-and-Conquer: Merge Sort

- Staring with the input array A[1:n], merge sort sorts a subarray A[p:r], recursing down to smaller and smaller subarrays.

> **Divide** the subarray $A[p:r]$ to be sorted into two adjacent subarrays, each of half the size. To do so, compute the midpoint q of $A[p:r]$ (taking the average of p and r), and divide $A[p:r]$ into subarrays $A[p:q]$ and $A[q+1:r]$.
>
> **Conquer** by sorting each of the two subarrays $A[p:q]$ and $A[q+1:r]$ recursively using merge sort.
>
> **Combine** by merging the two sorted subarrays $A[p:q]$ and $A[q+1:r]$ back into $A[p:r]$, producing the sorted answer.

- When the subarray contains a single element, it is the "base case"; The single element subarray is always sorted.

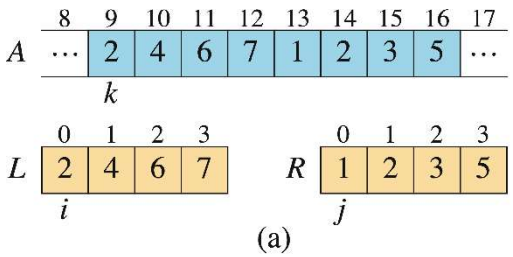
Sorting by Divide-and-Conquer: Merge Sort

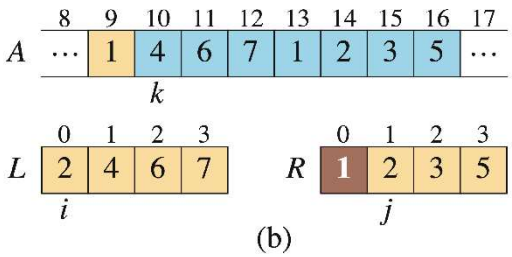
- The key operation of merge sort is the "combine" step.

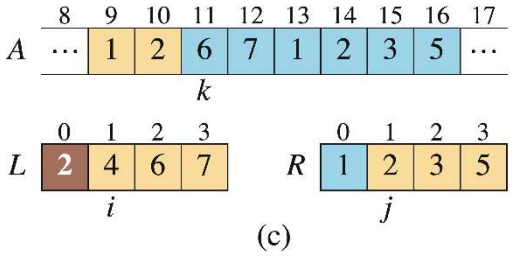
> **Combine** by merging the two sorted subarrays $A[p:q]$ and $A[q+1:r]$ back into $A[p:r]$, producing the sorted answer.

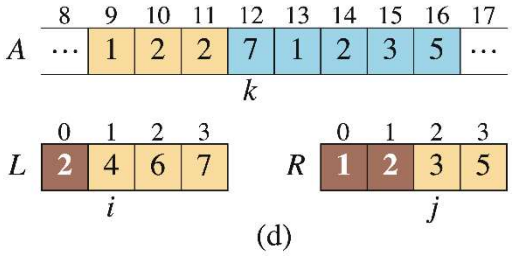
- The merge procedure merges two adjacent subarrays A[p:q] and A[q+1:r] that are already sorted, and outputs a single sorted array A[p:r].
  - $p \leq q < r$

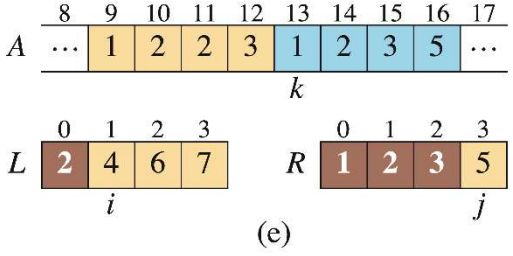
```
MERGE(A, p, q, r)
1  n_L = q − p + 1          // length of A[p : q]
2  n_R = r − q              // length of A[q + 1 : r]
3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
4  for i = 0 to n_L − 1     // copy A[p : q] into L[0 : n_L − 1]
5      L[i] = A[p + i]
6  for j = 0 to n_R − 1     // copy A[q + 1 : r] into R[0 : n_R − 1]
7      R[j] = A[q + j + 1]
8  i = 0                    // i indexes the smallest remaining element in L
9  j = 0                    // j indexes the smallest remaining element in R
10 k = p                    // k indexes the location in A to fill
11 // As long as each of the arrays L and R contains an unmerged element,
   //     copy the smallest unmerged element back into A[p : r].
12 while i < n_L and j < n_R
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
18     k = k + 1
19 // Having gone through one of L and R entirely, copy the
   //     remainder of the other to the end of A[p : r].
20 while i < n_L
21     A[k] = L[i]
22     i = i + 1
23     k = k + 1
24 while j < n_R
25     A[k] = R[j]
26     j = j + 1
27     k = k + 1
```

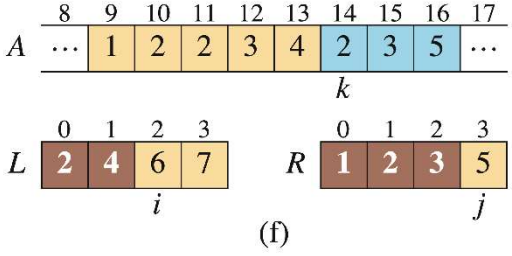

8 9 10 11 12 13 14 15 16 17
A ⋯ 2 4 6 7 1 2 3 5 ⋯
k
0 1 2 3
L 2 4 6 7
i
0 1 2 3
R 1 2 3 5
j
(a)

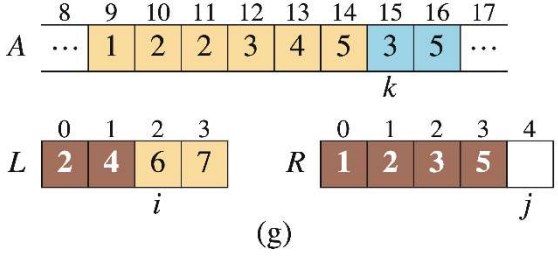

8 9 10 11 12 13 14 15 16 17
A ⋯ 1 4 6 7 1 2 3 5 ⋯
k
0 1 2 3
L 2 4 6 7
i
0 1 2 3
R 1 2 3 5
j
(b)

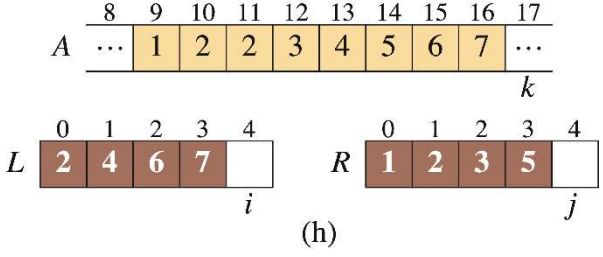

8 9 10 11 12 13 14 15 16 17
A ⋯ 1 2 6 7 1 2 3 5 ⋯
k
0 1 2 3
L 2 4 6 7
i
0 1 2 3
R 1 2 3 5
j
(c)


8 9 10 11 12 13 14 15 16 17
A ⋯ 1 2 2 7 1 2 3 5 ⋯
k
0 1 2 3
L 2 4 6 7
i
0 1 2 3
R 1 2 3 5
j
(d)


8 9 10 11 12 13 14 15 16 17
A ⋯ 1 2 2 3 1 2 3 5 ⋯
k
0 1 2 3
L 2 4 6 7
i
0 1 2 3
R 1 2 3 5
j
(e)


8 9 10 11 12 13 14 15 16 17
A ⋯ 1 2 2 3 4 2 3 5 ⋯
k
0 1 2 3
L 2 4 6 7
i
0 1 2 3
R 1 2 3 5
j
(f)


8 9 10 11 12 13 14 15 16 17
A ⋯ 1 2 2 3 4 5 3 5 ⋯
k
0 1 2 3
L 2 4 6 7
i
0 1 2 3 4
R 1 2 3 5
j
(g)


8 9 10 11 12 13 14 15 16 17
A ⋯ 1 2 2 3 4 5 6 7 ⋯
k
0 1 2 3 4
L 2 4 6 7
i
0 1 2 3 4
R 1 2 3 5
j
(h)

Analysis of Merge Operation

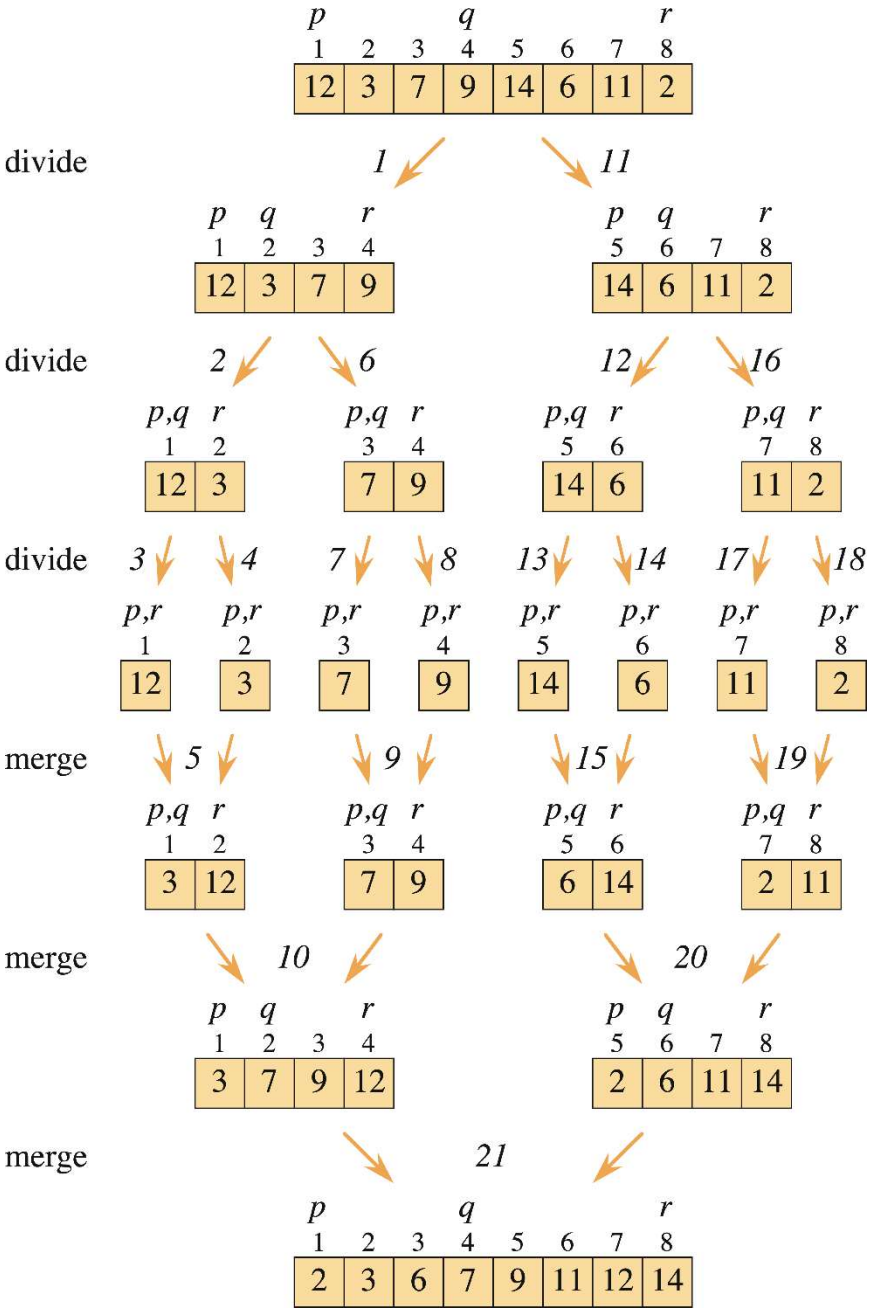
- Function MERGE(A, p, q, r) runs in $\Theta(n)$ when $n = r - p + 1$.

- Lines 1-3 and 8-10 takes constant time.
- For loops of lines 4-7 take $\Theta(n_L + n_R) = \Theta(n)$ time.
- Observing the three while loops of lines 12-18, 20-23, and 24-27,
  - Each iteration of these loops copies exactly one value from L or R back into A, and every value is copied back into A exactly once.
  - Therefore, these three loops together make a total of n iterations.
  - The total time spent in these three loops is $\Theta(n)$.

Merge Sort

- The algorithm using MERGE operation as a subprocedure.

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```


p q r
1 2 3 4 5 6 7 8
12 3 7 9 14 6 11 2
divide
1
11
p q r
1 2 3 4
12 3 7 9
p q r
5 6 7 8
14 6 11 2
divide
2
6
12
16
p,q r
1 2
12 3
p,q r
3 4
7 9
p,q r
5 6
14 6
p,q r
7 8
11 2
divide
3
4
7
8
13
14
17
18
p,r
1
12
p,r
2
3
p,r
3
7
p,r
4
9
p,r
5
14
p,r
6
6
p,r
7
11
p,r
8
2
merge
5
9
15
19
p,q r
1 2
3 12
p,q r
3 4
7 9
p,q r
5 6
6 14
p,q r
7 8
2 11
merge
10
20
p q r
1 2 3 4
3 7 9 12
p q r
5 6 7 8
2 6 11 14
merge
21
p q r
1 2 3 4 5 6 7 8
2 3 6 7 9 11 12 14

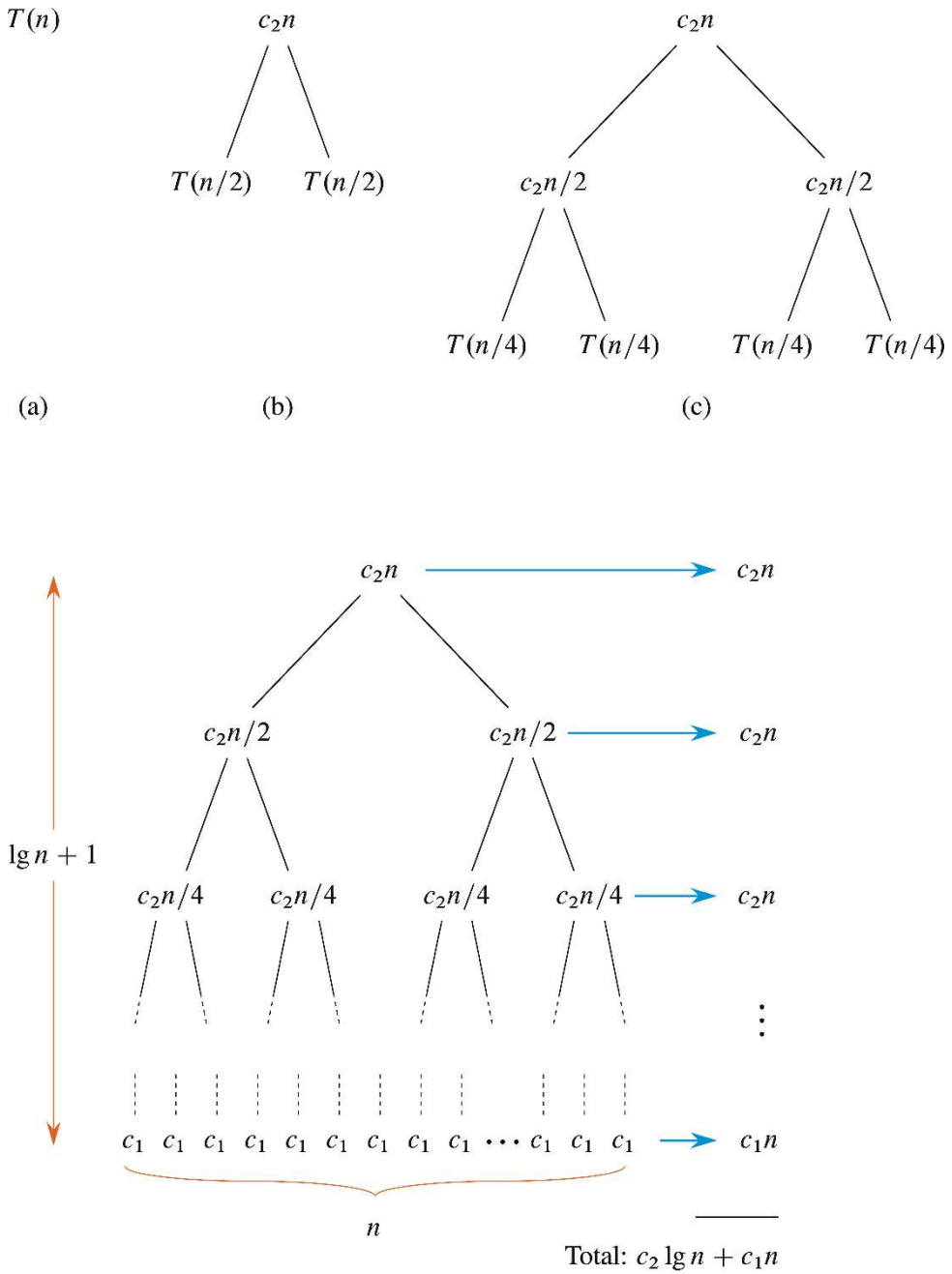
Analysis of Merge Sort

- For recursive algorithms, we can describe its running time by a recurrence equation.

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

- When we divide a problem of n, size of the subproblem becomes $n/2$.
- Thus, we can express the running time as,
  - $T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$

- Solving this recurrence equation, we get
  - $T(n) = \Theta(n \log n)$


T(n)
c2n
T(n/2)
T(n/2)
c2n
c2n/2
c2n/2
T(n/4)
T(n/4)
T(n/4)
T(n/4)
(a)
(b)
(c)
lg n + 1
c2n
c2n/2
c2n/2
c2n/4
c2n/4
c2n/4
c2n/4
c1 c1 c1 c1 c1 c1 c1 c1 c1 ··· c1 c1 c1
c2n
c2n
c2n
c1n
n
Total: c2 lg n + c1n
(d)

End of Class

Questions?

Instructor office: AS-1013

Email: jso1@sogang.ac.kr