Chapter 2
Getting Started

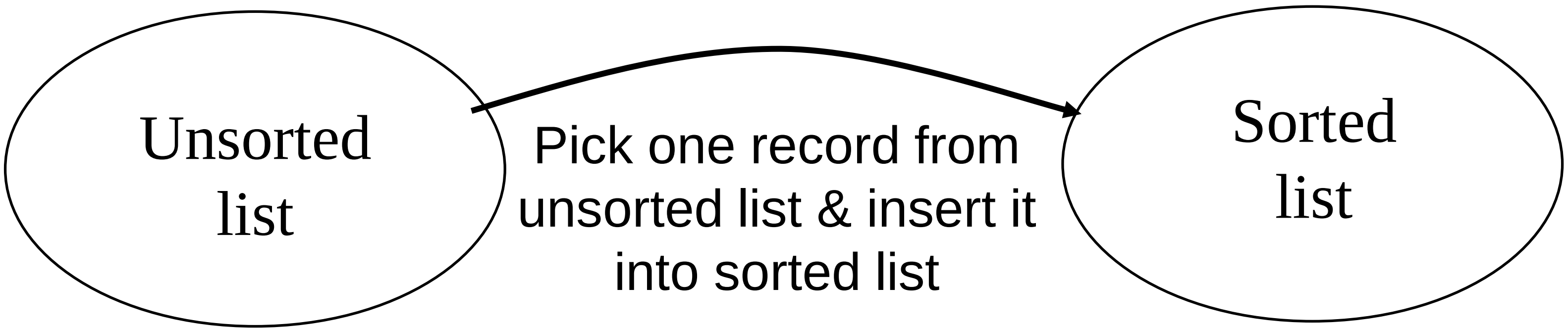
Algorithm Analysis

School of CSEE

Getting Started

- This chapter will give you an idea of the framework that will be used throughout the book.
- We will begin with the example of ‘Insertion Sort’ then take a look at the ‘Mergesort’ briefly.

An Example : Insertion Sort



Example)

27	4	21	38	1	12

4	27	21	38	1	12

4	21	27	38	1	12

4	21	27	38	1	12

1	4	21	27	38	12

1	4	12	21	27	38

An Example : Insertion Sort

1	2	3	4	5	6	
4	21	27	38	1	12	Key = 1, j = 5
4	21	27	38	38	12	i = 4
4	21	27	27	38	12	i = 3
4	21	21	27	38	12	i = 2
4	4	21	27	38	12	i = 1 → i = 0
1	4	21	27	38	12	

Insertion sort

Insertion-Sort (A);

for $j \leftarrow 2$ to $length(A)$

do $key \leftarrow A[j]$

► **Insert $A[j]$ into the sorted sequence $A[1..j-1]$.**

$i \leftarrow j - 1;$

while $i > 0$ and $A[i] > key$

do $A[i+1] \leftarrow A[i];$

$i \leftarrow i-1;$

$A[i+1] \leftarrow key;$

Algorithm

- What we are going to learn?
  - - Designing the algorithm 알고리즘 만들고
  - - Analyzing the algorithm 옳은지 (빠른지, 좋은지) 증명.
    시간, 메모리

Design paradigms

- Insertion-sort uses *incremental* approach : having sorted the subarray A[1..j-1], we insert the single element A[j] into its proper place, yielding the sorted subarray A[1..j].
- cf) Divide-and-conquer approach : A problem is divided into a number of like problems of smaller size to yield small results that can be combined to produce a solution to the original problem.

  : 3 steps
  - Divide
  - Conquer
  - Combine

Other design paradigms

- Greedy
- Dynamic Programming
- Branch and Bound
- Backtracking
- Brute force ?

→ optimaltzation
모든 경우를 다 따져봄 (약간 무식)

Analysis

- Correctness

증명까진 아니지만 논리적으로 설명할 수 있어야 함

: Proving the correctness of the algorithm

- Efficiency

: Obtaining the time complexity of the algorithm

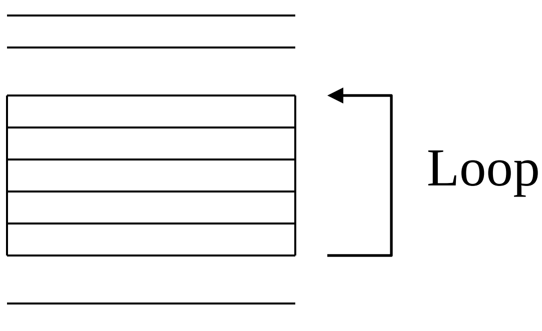
Correctness

모든 input에 대해서 맞는 output을 내놓고 stop 해야됨.

- **An algorithm is said to be *correct* if, for every input instance, it halts with the correct output.**
- **We say that a correct algorithm *solves* the given computational problem.**

Loop invariants

- Loop invariants → Loop이 돌아가면서 변하지 않는 성질.
  - Program structure



- Definition: (Loop invariant)
  - Loop invariants are conditions and relationships that are satisfied by the variables and data structures at the end of each iteration of the loop.

조건 관계

Loop invariants

- Often use loop invariants to help us understand why an algorithm is correct.
- Must show three things about a loop invariants (similar to mathematical induction)　:
  - **[illegible] : It is true prior to the first iteration of the loop.**

    (a base case of the induction)
  - **[illegible] : If it is true before an iteration of the loop, it remains true before the next iteration.**

    (inductive step)
  - **[illegible] : When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.**

Correctness of insertion sort

- Loop invariant : At the start of each iteration of the for loop, the subarray $A[1..j-1]$ consists of the elements originally in $A[1..j-1]$ but in sorted order.
- Initialization : when j=2, $A[1..j-1]$ consists of the single element $A[1]$. Trivially sorted.
- Maintenance : Informally, the body of outer '*for*' loop works by moving $A[j-1]$, $A[j-2]$, $A[j-3]$, and so on, by one position to the right until the proper position for $A[j]$ is found.
- Termination : The outer '*for*' loop ends when $j = n+1$. Thus, $A[1..n]$ consists of the elements originally in $A[1..n]$ but in sorted order.

Efficiency

- Predicting the resources – time, storage - that the algorithm requires.

저장 공간과 시간.

as a function of the input size *n*

- Space requirement --- not a big deal
- Time requirement --- in terms of the number of basic operations

Efficiency

- We need a model of the implementation technology.
- Random-access machine (RAM) model - a generic one-processor model of computation
  - **Instructions are executed one after another, with no concurrent operations.** → Instruction이 하나하나 sequence하게 실행됨.
  - **It contains instructions commonly found in the real computers** → Arithmetic, data movement, Control. etc) magic - 우리 이런건 생각하지 않기로 해요
  - **Each instruction takes a constant time**
    - **Arithmetic (add, subtract, multiply, divide, remainder, floor, ceiling, shift left/right for mult/div by 2^k)**
    - **Data movement (load, store, copy)**
    - **Control (conditional and unconditional branch, subroutine call and return)**
  - **Data types : integers, floating point**
  - **No memory hierarchy, i.e., no cache or virtual memory**

* 굉장히 단순한 모델에서 알고리즘이 돌아간다고 가정함. *

Efficiency

- Need to specify running time for a particular input size

특정한 input size에서 시간이 얼마나 걸리는지.
(=execution time)

- Input size : depends on the problem

$O(n\log n)$, $O(n^2)$

  – sorting *n* numbers : number of items in the input
  – multiplying two integers : total number of bits needed to represent the input in ordinary binary notation.
  – Graph algorithms : number of vertices and edges

(Number of edges, Number of vertex 가 중요함)

Efficiency

- Running time of an algorithm on a particular input
  - The number of primitive operations or "steps" executed.

    add, compare, move
  - Steps are defined to be machine-independent
  - Each line of pseudocode requires a constant amount of time.

    ↳ Constant time으로 생각하기.
  - Each line may take different amount of time.

Time complexity Analysis

- **Worst-case:** (usually) 최악의 경우에 이 시간이 걸림.
  - $T(n)$ = maximum time of algorithm on any input of size n.
- **Average-case:** (sometimes) Input이 한정된 것이 아니여서 구하기 어려움.
  - $T(n)$ = expected time of algorithm over all inputs of size n.
  - Need assumption of statistical distribution of inputs.
- **Best-case:** (bogus) 실제로는 알고리즘에서 의미가 X. 빠른 case 하나 구해놓고 빠르다고 우기는 느낌.
  - Cheat with a slow algorithm that works fast on *some* input.

는 보임

Time complexity Analysis

- Usually, interested in the worst-case running time because
  - It gives an upper bound 어떤 Input이 들어와도 이거보다 오래 걸리지 않다.
  - For some algorithms, the worst case occurs often. → worst case가 자주 일어날 수도 있음.
  - Average case is often as bad as the worst case. → average case에서 worst case가 자주 일어날 수도 있음.

- Average-case or ***expected*** running time – use ***probabilistic analysis*** 각각의 case가 일어날 확률이 어떻게 되는가?
  - Need assumption about the distribution of the input.
  - ***Randomized*** algorithm : permute the input

Analysis of insertion-sort

		시간	몇 번 반복되는가.
	Inserion-Sort (*A*);	Cost	times
1	for ***j*** ← ***2*** to ***length(A)***	c_1	n
2	do ***key*** ← ***A*[j]**	c_2	$n-1$
3	► Insert A[j] into the sorted sequence A[1..j-1].	0	$n-1$
4	*i* ← *j* - 1;	c_4	$n-1$
5	**while** *i* > 0 and *A*[*i*] > *key*	c_5	t_j
6	**do** *A*[*i*+1] ← *A*[*i*];	c_6	$t_j - 1$
7	*i* ← *i*-1;	c_7	$t_j - 1$
8	*A*[*i*+1] ← *key*;	c_8	$n-1$

For $j=2,\ldots,n$, let t_j be the number of times that the while loop is executed for that value j.

Analysis of insertion-sort

$\Sigma(\text{Cost} \cdot \text{Time})$

- $T(n) = c_1 n + c_2(n\text{-}1) + c_4(n\text{-}1) + c_5 \Sigma\, t_j + c_6 \Sigma(t_j\text{-}1) + c_7 \Sigma(t_j\text{-}1) + c_8(n\text{-}1)$

- What can $T(n)$ be?
  - Best case -- inner loop body never executed
    - 각각의 while이 한 번만 돌아가는 경우
    - $t_j = 1 \Rightarrow T(n)$ is a linear function. $T(n) = \Theta(n)$. $O(n)$
  - Worst case -- inner loop body executed for all previous elements ⇒ while에서 array 끝까지 비교할 때,
    - $t_j = i \Rightarrow T(n)$ is a quadratic function. $T(n) = \Theta(n^2)$.
  - Average case $\frac{1}{2}n(n+1) = \underline{\frac{1}{2}n^2} + \frac{1}{2}n$
    - ???

Merge Sort

Pseudo code # recursion

```
MergeSort(A, left, right) {
    if (left < right) {
        mid = floor((left + right) / 2);
        MergeSort(A, left, mid);
        MergeSort(A, mid+1, right);
        Merge(A, left, mid, right);   #Combine
    }
}
```

Merge( ) takes two sorted subarrays of A and merges them into a single sorted subarray of A (how long should this take?)

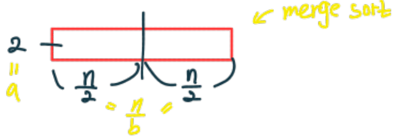
Merge sort

- Use divide-and-conquer paradigm
- Divide : divide the n-element sequence to be sorted into two subsequences of $n/2$ elements each.
- Conquer : sort the two subsequences recursively using merge sort.
- Combine : merge the two sorted subsequences to produce the sorted answer.

Analyzing divide-and-conquer algorithms

* 점화식 *

- Use a ***recurrence equation*** (or a ***recurrence***) to describe the running time of a divide-and-conquer algorithm.
- $T(n)$: running time on a problem of size n.
- If the problem size is small enough ($n \le c$) for some constant c, the straightforward solution takes constant time, $\Theta(1)$.
- a : number of subproblems
- n/b : input size of the subproblem
- $D(n)$: time to divide
- $C(n)$: time to combine



$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c \\ a\,T(n/b) + D(n) + C(n) & \text{otherwise.} \end{cases}$$

→ base case

→ general case

몇개인가 divide combine.

Analysis of merge sort

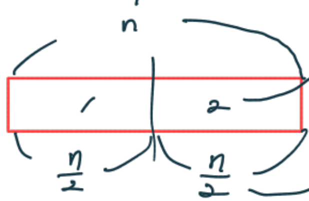
1. ***Divide:*** Trivial
2. ***Conquer:*** Recursively sort 2 subarrays.
3. ***Combine:*** Linear-time merge.

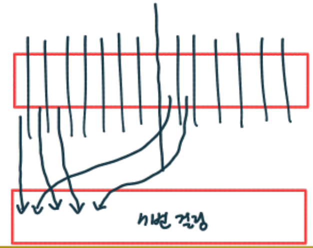
$$T(n) = \underbrace{2\,T(n/2)}_{\text{merge sort}} + \Theta(n)$$

\# subproblems

subproblem size

Work dividing and combining





divide : constant time $\Theta(1)$

combine : $\Theta(n)$

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

Analysis of merge sort

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

By the master theorem (in Ch 4), we can show that $T(n) = \Theta(n \lg n)$.

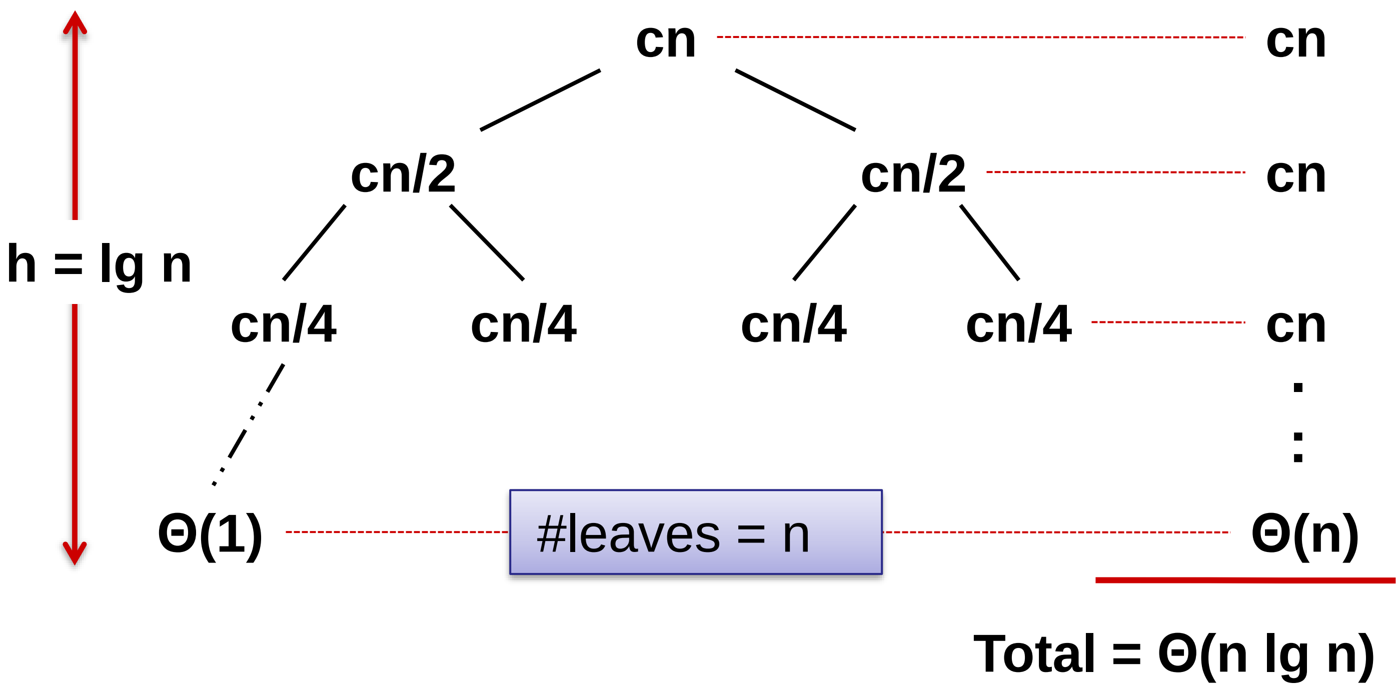
- Compared to insertion sort ($\Theta(n^2)$ worst-case time), merge sort is faster.

  Insertion sort Θ(n²)과 비교하면, 최대한 time complexity 에서 Θ(nlogn)은 빠른 편임.

- On small inputs, insertion sort may be faster. But, for large enough inputs, merge sort will always be faster, because its running time grows more slowly than insertion sort's.

Recurrence for merge sort

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



recursion tree ·

Keyword

- Designing Algorithm – design paradigms
- Analysis of Algorithm

  - Correctness : proof

  - Efficiency : time requirement

    # Worst case

    # Average case

P

A