Sorting in Linear Time

Heejin Park

Hanyang University

Contents

- **Lower bounds for sorting**
- **Counting sort**
- **Radix sort**

Lower bounds for sorting

- **Comparison sorts**
  - Sorting algorithms using only comparisons to determine *the sorted order of the input elements*.
  - Use tests such as $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$.
  - Heapsort, Mergesort, Insertion sort, Selection sort, Quicksort
- **Lower bounds for (comparison) sorting** is $\theta(n)$
  - Any comparison sort must make $\Omega(n \lg n)$ comparisons in the worst case to sort n elements.

Lower bounds for sorting

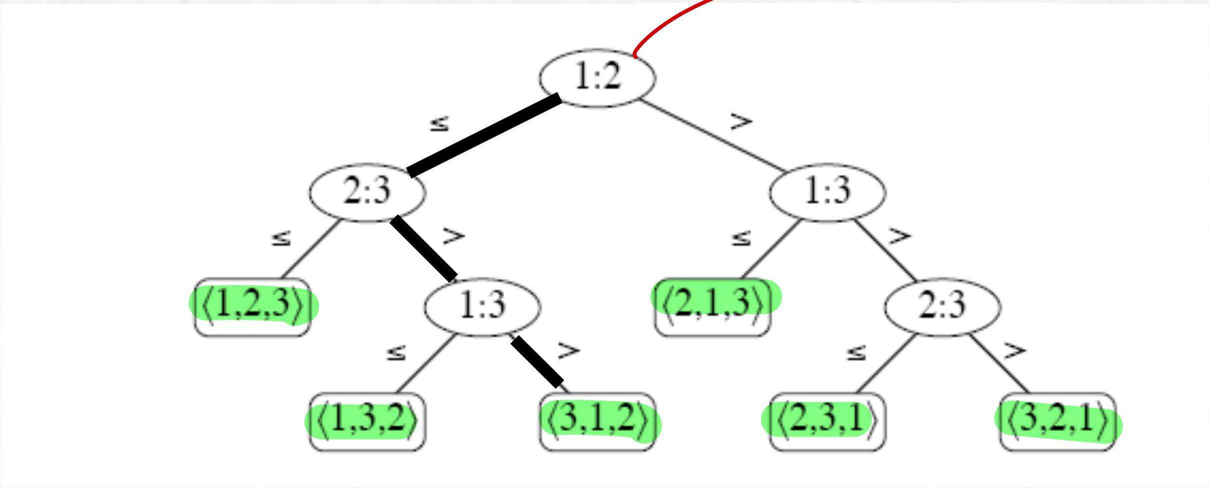
- **Comparison sort**
  - we assume without loss of generality that all of the input elements are distinct. 등호 필요없는
    - The comparisons $a_i \leq a_j$, $a_i \geq a_j$, $a_i > a_j$, and $a_i < a_j$ are all equivalent.
    - We assume that all comparisions have the form $a_i \leq a_j$

The decision-tree model

- Comparison sorts can be viewed in terms of ***decision trees***.
  - A full binary tree.
  - Each leaf is a permutation of input elements.
  - Each internal node $i{:}j$ indicates a comparison $a_i \le a_j$.

when 1, 2, 3 → 3! number of permutation

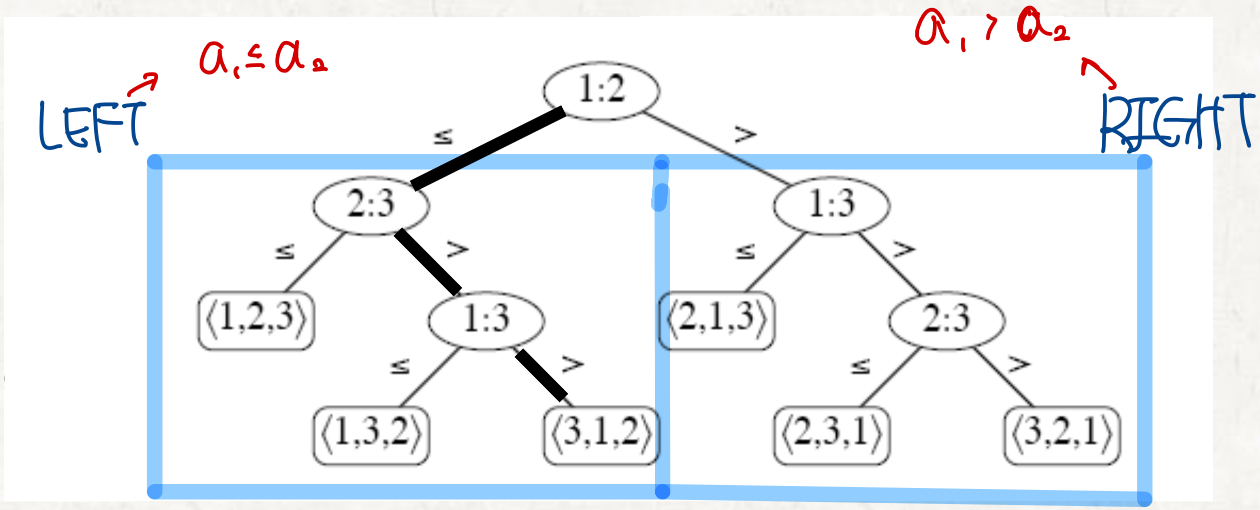
$a_1 \le a_2$



A decision tree for insertion sort

The decision-tree model

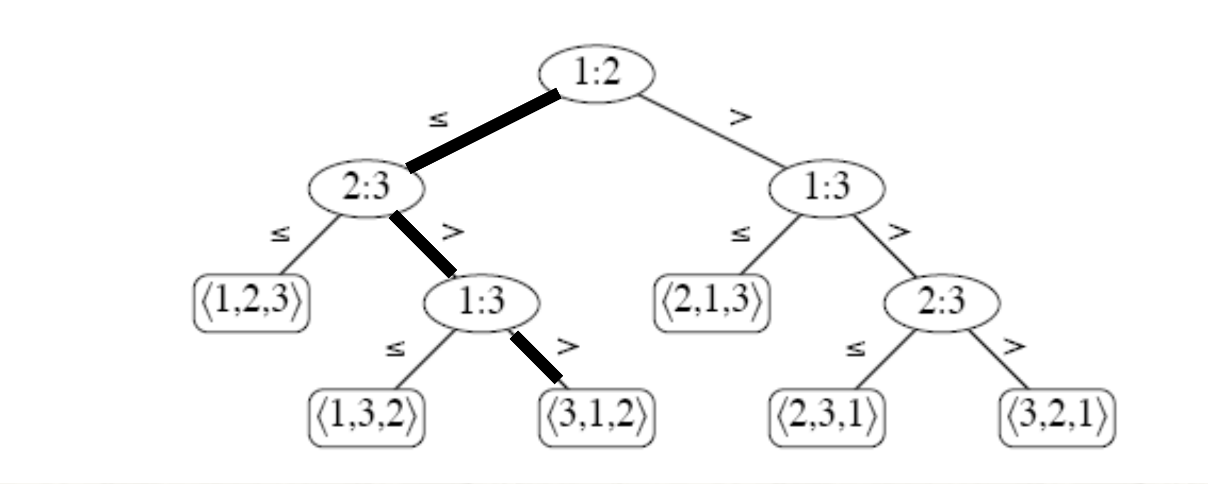
- The left subtree of the node *i*:*j* includes all permutations for $a_i \leq a_j$.
- The right subtree includes all permutations for $a_i > a_j$.



A decision tree for insertion sort

The decision-tree model

- The execution of the sorting algorithm corresponds to tracing a path from the root of the decision tree to a leaf.

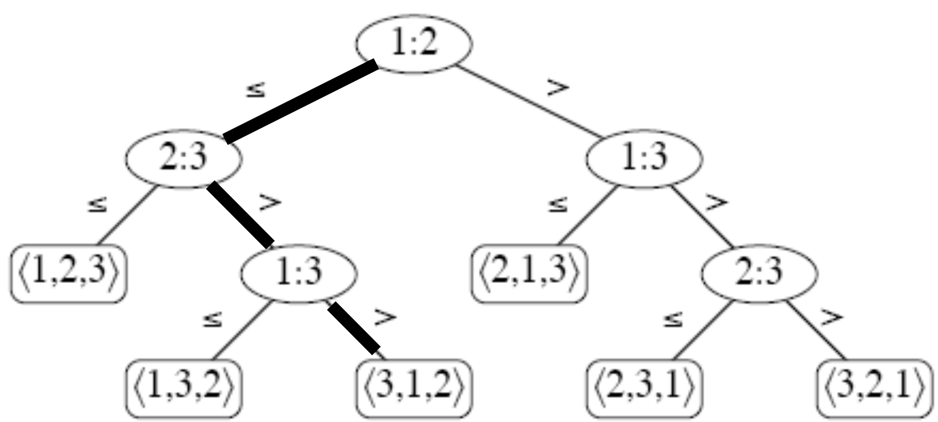


A decision tree for insertion sort

The decision-tree model

- the worst-case number of comparisons
  = the height of its decision tree. = comparison의 개수

height
0
1
2
3



A decision tree for insertion sort

The decision-tree model

- ***Theorem* 8.1**: Any comparison sort algorithm requires $\Omega(n \lg n)$ comparisons in the worst case.

Max number of nodes in level h → 2^h

decision tree

- ***Proof***:

→ element of set

If leaf node's level is h
→ max number of nodes in tree is $2^{h+1}-1$

  - Height: *h*, Number of element: *n*
  - The number of leaves: $n!$
    - Each permutations for *n* input elements should appear as leaves.
  - $n! \leq 2^h$
  - $\lg(n!) \leq h$
  - $\Omega(n \lg n)$ (by equation (3.18) : $\lg(n!) = \Theta(n\lg n)$).

↳ $\log n! < \log n^n = n\log n$

Self-study

- **Exercise 8.1-1**
  - The smallest depth of a leaf in a decision tree
- **Exercise 8.1-3**
  - Decision tree existence
- **Exercise 8.1-4**
  - Lower bound of a decision tree

Counting sort

- **Counting sort**

Running time : $\Theta(n)$

→ whole array once.

  - A sorting algorithm using *counting*.

0	1
3개	5개

A	0	1	1	0	1	1	0	1
B	0	0	0	1	1	1	1	1

  - Each input element x should be located in the ith place after sorting if the number of elements less than x is i-1.

Counting sort

k

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

k

Running time: $\Theta(n+k)$

	0	1	2	3	4	5
C	2	0	2	3	0	1
	$0 \to 1 \to 2$		$0 \to 1 \to 2$	$0 \to 1 \to 2$		$0 \to 1$

B	0	0	2	2	3	3	3	5

Counting sort

- **Counting sort**
  - Stable
    - Same values in the input array appear in the same order in the output array.

distinguishable (despite of same value)

A	2	5	3	0	2	3	0	3

→ 상대적인 순서가 그대로 유지되는 것!

B	0	0	2	2	3	3	3	5

Counting sort

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

k

	0	1	2	3	4	5
C	2	0	2	3	0	1
C'	2	2	4	7	7	8

prefix-sum of C

$$C'[i] = \sum_{k=0}^{i} C[k]$$

1. Running time: $\Theta(k^2)$

$1+2+3+ \cdots + k-1$

$= \frac{k^2-k}{2}$

2. Running time: $\Theta(k)$

$C'[i] = C'[i-1] + C[i]$

(simple version)

Counting sort

Backward

A

1	2	3	4	5	6	7	8
2	5	3	0	2	3	0	3

C'

0	1	2	3	4	5
2	2	4	7	7	8

-1

B

1	2	3	4	5	6	7	8
						3	

C'

0	1	2	3	4	5
2	2	4	6	7	8

-1

B

1	2	3	4	5	6	7	8
	0					3	

C'

0	1	2	3	4	5
1	2	4	6	7	8

B

1	2	3	4	5	6	7	8
	0				3	3	

C'

0	1	2	3	4	5
1	2	4	5	7	8

Counting sort

COUNTING-SORT(A, B, k)

input output

→ 0 ~ k : key values

$\Theta(k)$

for i = 0 to k

  $C[i] = 0$ → initializing

$\Theta(n)$

for j = 1 to $A.length$

  $C[A[j]] = C[A[j]] + 1$ → Counting

▷ $C[i]$ contains the number of elements equal to i.

$\Theta(k)$

for i = 1 to k 이미 'i-1'까지는 $C = C'$

  $C'[i] = C[i] + C'[i - 1]$ ∴ $C[i] = C[i] + C[i-1] = C'[i] - C[i-1] + C[i]$

▷ $C[i]$ contains the number of elements less than or equal to i.

$\Theta(n)$

for j = $A.length$ downto 1

  → index

  $B[\boxed{C[A[j]]}] = A[j]$

  $C[A[j]] = C[A[j]] - 1$

Counting sort

- The overall time is $\Theta(k+n)$ where k is the range of input integers.
- If $k = O(n)$, the running time is $\Theta(n)$.

Self-study

- **Exercise 8.2-1**
  - A counting-sort example
- **Exercise 8.2-3**
  - Counting-sort stability
- **Exercise 8.2-4**
  - A counting-sort application

Radix sort

- **Radix sort** (MSD → LSD)

MSD부터 정렬
MSD가 같다면 다음 숫자 기준
. .. LSD까지

326		326		326		326
453		453		435		435
608		435		453		453
835	→	608	→	608	→	608
751		690		690		690
435		751		704		704
704		704		751		751
690		835		835		835

소그룹화하는 것이 복잡하다

Radix sort

- **Radix sort** (MSD ← LSD)

순서그대로!

326	690	704	326
453	751	608	435
608	453	326	453
835	704	835	608
751	835	435	690
435	435	751	704
704	326	453	751
690	608	690	835

Radix sort

Radix sort가 아닌
counting sort라면?

앞장에서 $n=8$, $k=0 \sim 999$
=> linear time에 sort 불가능

RADIX-SORT(A, d)
1 for $i = 1$ to d
2 use a ***stable sort*** to sort array A on digit i

running time of counting sort of 1 digit

- RADIXSORT sorts in $\Theta(d(n + k))$ time when n d-digit numbers are given and each digit can take on up to k possible values.

10진수 → $0 \sim 9$
$k=10$!

- When d is constant and $k = O(n)$, radix sort runs in linear time.

Radix sort

- **Changing *d* and *k***

1326
4534
6018
8135

$d = ?$ 4
$k = ?$ 10

1326
4534
6018
8135

$d = ?$ 2 → 4/2
$k = ?$ 100 (0~99)

Radix sort

- ***Lemma* 8.4 (Self-study)**
  Given n b-bit numbers and any positive integer $r \le b$, RADIX-SORT correctly sorts these numbers in $\Theta((b/r)(n + 2^r))$ time.

$\Theta(\underbrace{d}_{b/r}(n+\underbrace{k}_{2^r}))$

$$\overbrace{\hphantom{1\quad 0\quad 1\quad \cdots\quad \cdots\quad \cdots\quad 1\quad 0\quad 1\quad 1}}^{b}$$

$$\left.\begin{array}{cccccccccc}
1 & 0 & 1 & \cdots & \cdots & \cdots & 1 & 0 & 1 & 1 \\
0 & 1 & & & & & 1 & 0 & 0 & 1 \\
\vdots & \vdots & & & & & & \vdots & \vdots & \vdots \\
0 & 1 & & & & & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & \cdots & \cdots & \cdots & 1 & 0 & 0 & 1
\end{array}\right\} n$$

$$\underbrace{\hphantom{1\quad 0\quad 0}}_{r}\quad \cdots \quad \underbrace{\hphantom{\cdots\quad \cdots\quad 1}}_{r}\quad \underbrace{\hphantom{0\quad 0\quad 1}}_{r}$$

Radix sort

- Computing optimal r minimizing $(b/r)(n + 2^r)$.

  1. $b < \lfloor \lg n \rfloor$

     for any value of r, $(n + 2^r) = \Theta(n)$ because $r \le b$. $< \lfloor \log n \rfloor$

     So choosing $r = b$ yields a running time : $(b/b)(n + 2^b) = \Theta(n)$, which is asymptotically optimal.

$r < \lfloor \log n \rfloor$

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

Radix sort

$\frac{b2^r}{r} \le \frac{b(n+2^r)}{r}$

- Computing optimal r minimizing $(b/r)(n + 2^r)$.

  2. $b \geq \lfloor \lg n \rfloor$

     $r = \lfloor \log n \rfloor$

     choosing $r = \lfloor \lg n \rfloor$ gives the best time to within a constant factor, $(b/\lg n)(n+2^{\lg n}) = (b/\lg n)(2n) = \Theta(bn/\lg n)$.

  - As we increase r above $\lfloor \lg n \rfloor$, the 2^r in the numerator increases faster than the r in the dominator.
  - As we decrease r below $\lfloor \lg n \rfloor$, then the b/r term increases and the $n + 2^r$ term remains at $\Theta(n)$.

1. $r > \log n \Rightarrow \frac{b}{r} \cdot 2^r \Rightarrow$ 항상 : $2^r > r$ $(r > \log n) \Rightarrow \frac{b}{r}(n+2^r) \uparrow$

2. $r < \log n \Rightarrow \frac{b}{r} \uparrow \rightarrow \frac{b}{r}(n+2^r) \uparrow$

Radix sort

- Compare radix sort with other sorting algorithms.
  - If $b = O(\lg n)$, we choose $r \approx \lg n$. → If $b = 4\log n$, $r = \log n$ & $d = 4$

Radix sort: $\Theta(n)$

Quicksort: $\Theta(n \lg n)$

$\Theta(b/r(n+2^r))$

$r = \lfloor \log n \rfloor$

$\Theta(bn/\log n)$

$\Theta(n)$

Radix sort

- The constant factors hidden in the Θ-notation differ.

  1. Radix sort may make fewer passes than quicksort over the *n* keys, each pass of radix sort may take significantly longer.
  2. Radix sort does not sort in place.

1. Radix sort의 $\Theta(dn)$의 'd'가 Quick sort의 $\Theta(n\log n)$보다 월등히 크다면 Quick sort가 더 빠를 수도 있다

Self-study

- **Exercise 8.3-1**
  - Radix sort example
- **Exercise 8.3-2**
  - Stability
- **Exercise 8.3-4**
  - Radix sort application