Quick Sort Example

- Example 7.3
  - n = 10
  - input list (26, 5, 37, 1, 61, 11, 59, 15, 48, 19)

R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}	*left*	*right*
[26	5	37	1	61	11	59	15	48	19]	1	10
[11	5	19	1	15]	26	[59	61	48	37]	1	5
[1	5]	11	[19	15]	26	[59	61	48	37	1	2
1	5	11	[19	15]	26	[59	61	48	37]	4	5
1	5	11	15	19	26	[59	61	48	37]	7	10
1	5	11	15	19	26	[48	37]	59	[61]	7	8
1	5	11	15	19	26	37	48	59	[61]	10	10
1	5	11	15	19	26	37	48	59	61		

Figure 7.1: Quick Sort example

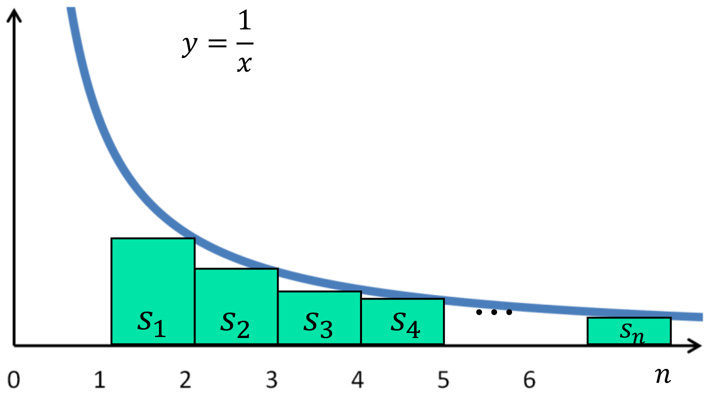
Quick Sort Analysis

- Time complexity: $T(n) = T(i) + T(n - i + 1) + n$
  - Performance depends on the selection of pivot.
  - Worst case : divide $n - 1$ and 1 element
    - $T(n) = T(n-1) + n$
      $= T(n-2) + (n-1) + n$
      $= T(1) + \sum_{i=2}^{n} i$
      $= O(n^2)$
  - Best case : divide $\frac{n}{2}$ and $\frac{n}{2}$ elements
    - $T(n) \leq cn + 2T(n/2)$, for some constant c
      $\leq cn + 2(cn/2 + 2T(n/4))$
      $\leq 2cn + 4T(n/4)$
      :
      $\leq cn\log_2 n + nT(1) = O(n\log n)$
- Unstable sorting
- Good(best) sorting method
  - The average computing time is O(nlogn)

Quick Sort Average Time

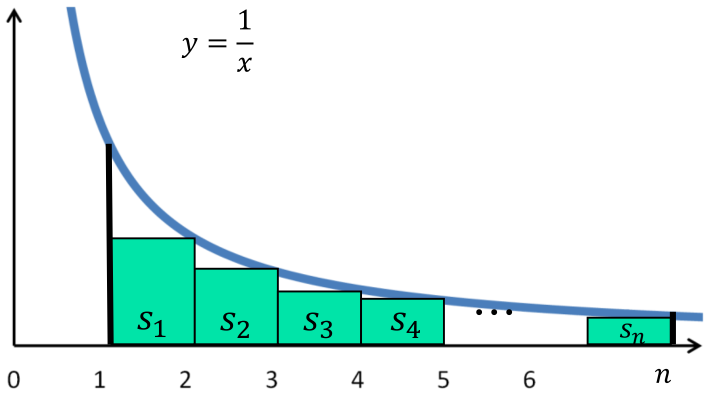
- Lemma 7.1: Let $T_{avg}(n)$ be the expected time for function QuickSort to sort a list with records. Then there exists a constant k such that $T_{avg}(n) \leq kn\log_e n$ for $n \geq 2$.

Analysis of Quicksort



- $S = s_1 + s_2 + s_3 + \cdots s_n = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

Analysis of Quicksort



- $S = s_1 + s_2 + s_3 + \cdots s_n = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$

$$< \int_1^n \frac{1}{x} dx = [\ln x]_1^n = \ln n$$

부분적분

- $\int x \log x \, dx = \frac{x^2 \log x}{2} - \frac{x^2}{4}$

Quick Sort Average Time

- Proof ($T_{avg}(n) \le kn\log_e n$ for $n \ge 2$)
  - We have

$$T_{avg}(n) \le cn + \frac{1}{n}\sum_{j=1}^{n}(T_{avg}(j-1) + T_{avg}(n-j)) = cn + \frac{2}{n}\sum_{j=0}^{n-1}T_{avg}(j) \quad (7.1)$$

  - We assume $T_{avg}(0) \le b$ and $T_{avg}(1) \le b$
  - Induction base: For n=2, $T_{avg}(2) \le 2c + 2b \le 2k\log_e 2.$
  - Induction hypothesis: Assume $T_{avg}(n) \le kn\log_e n$ for $1 \le n < m$
  - Induction step: From Eq. (7.1) and the induction hypothesis we have

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2}{m}\sum_{j=2}^{m-1}T_{avg}(j) \le cm + \frac{4b}{m} + \frac{2k}{m}\sum_{j=2}^{m-1} j\log_e j \quad (7.2)$$

  - Since $j\log_e j$ is an increasing function of j, Eq. (7.2) yields

$$T_{avg}(m) \le cm + \frac{4b}{m} + \frac{2k}{m}\sum_{j=2}^{m-1} j\log_e j \le cm + \frac{4b}{m} + \frac{2k}{m}\int_2^m x\log_e x\,dx = cm + \frac{4b}{m} + \frac{2k}{m}\left[\frac{m^2\log_e m}{2} - \frac{m^2}{4}\right]$$

$$= cm + \frac{4b}{m} + km\log_e m - \frac{km}{2} \le km\log_e m, \text{ for } m \ge 2$$

How fast can we sort ?

- Worst : $O(n^2)$
- Best : $O(n\log_2 n)$
- Decision tree : describing sorting process
  - vertex - key comparison
  - branch - result

Outline

- All the sorting algorithms introduced thus far are comparison sorts since *the sorted order they determine is based only on comparisons between the input elements*.
- We prove that any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort n elements.
- Thus, merge sort and heapsort are asymptotically optimal, and no comparison sort exists that is faster by more than a constant factor.
- We examine other sorting algorithms, such as counting sort and radix sort that run in linear time.
- Those algorithms use operations other than comparisons to determine the sorted order.
- Consequently, the $\Omega(n \log n)$ lower bound does not apply to them.

Lower Bounds for Sorting

- Comparison sorting
  - Use only comparisons between elements to gain order information about an input sequence $<a_1, a_2, \ldots, a_n>$.
  - Given a two elements a_i and a_j, we perform only one of the tests $a_i < a_j$, $a_i \le a_j$, $a_i \ge a_j$, or $a_i > a_j$ to determine their relative order.
- Our assumption
  - All input elements are distinct (i.e., we do not check $a_i = a_j$).
  - The comparisons $a_i < a_j$, $a_i \le a_j$, $a_i \ge a_j$, or $a_i > a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j.
  - Thus, all comparisons have the form $a_i \le a_j$.

Decision Tree Model

- A Decision tree is a full binary tree representing the comparisons between elements performed by a particular sorting algorithm operating on an input of a given size.
- In a decision tree, we annotate each internal node by i:j for some i and j in the range $1 \leq i, j \leq n$, where n is the number of elements in the input sequence - each internal node indicates a comparison $a_i \leq a_j$.
- We also annotate each leaf by a permutation $\pi(1), \pi(2), \ldots, \pi(n)$.
- The execution of the sorting algorithm corresponds to tracing a simple path from the root of the decision tree down to a leaf.
- When we come to a leaf, the sorting algorithm has established the ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq \ldots \leq a_{\pi}(n)$.

Decision Tree Model

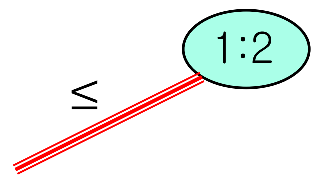
- We consider only decision trees in which each permutation appears as a reachable leaf.
  - Because any correct sorting algorithm must be able to produce each permutation of its input, each of the n! permutations on n elements must appear as one of the leaves of the decision tree for a comparison sort to be correct.
  - Furthermore, each of these leaves must be reachable from the root by a downward path corresponding to an actual execution of the comparison sort.

Insertion Sort

index	1	2	3
value	6	8	5

sorted

$a_1 = 6,\ a_2=8,\ a_3=5$

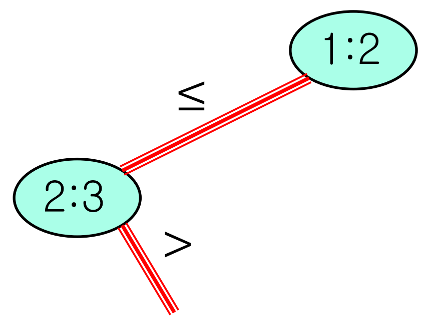


Insertion Sort

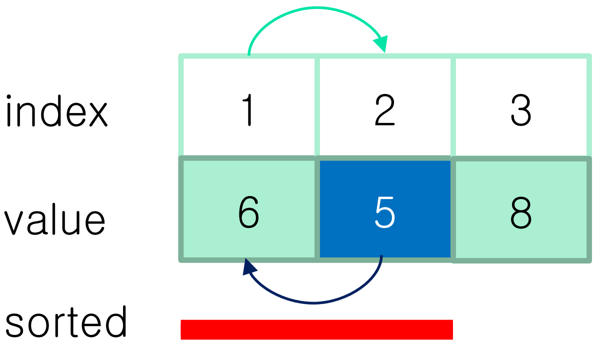
index	1	2	3
value	6	8	5

sorted

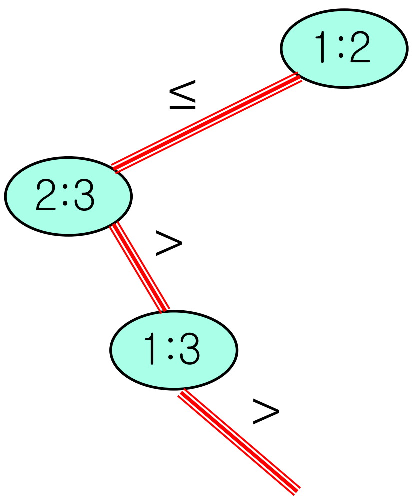
$a_1 = 6,\ a_2 = 8,\ a_3 = 5$



Insertion Sort



$a_1 = 6,\ a_2 = 8,\ a_3 = 5$

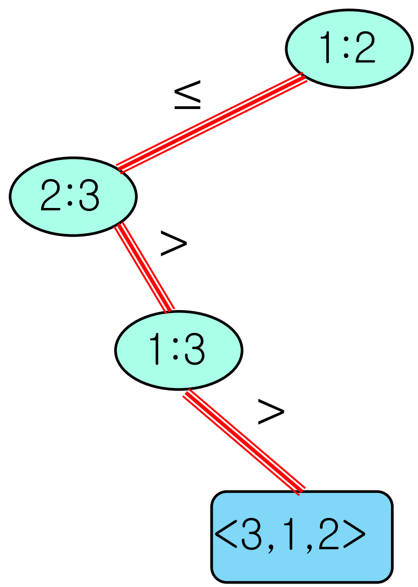


Insertion Sort

index	1	2	3
value	5	6	8

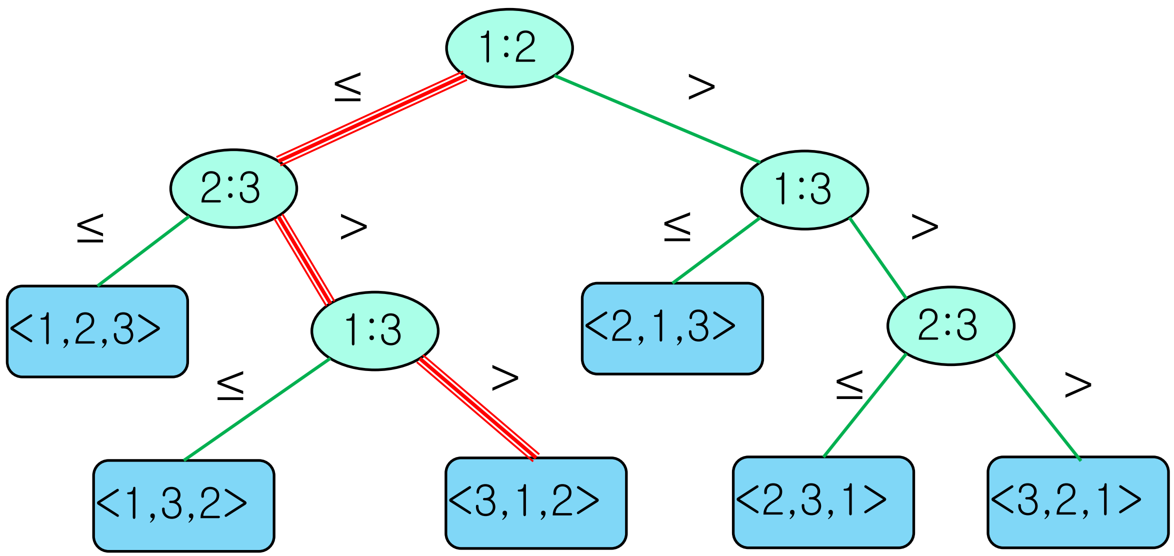
sorted

$a_1 = 6,\ a_2 = 8,\ a_3 = 5$



The Decision tree for Insertion Sort

- The decision tree corresponding to the insertion sort algorithm operating on an input sequence of three elements.

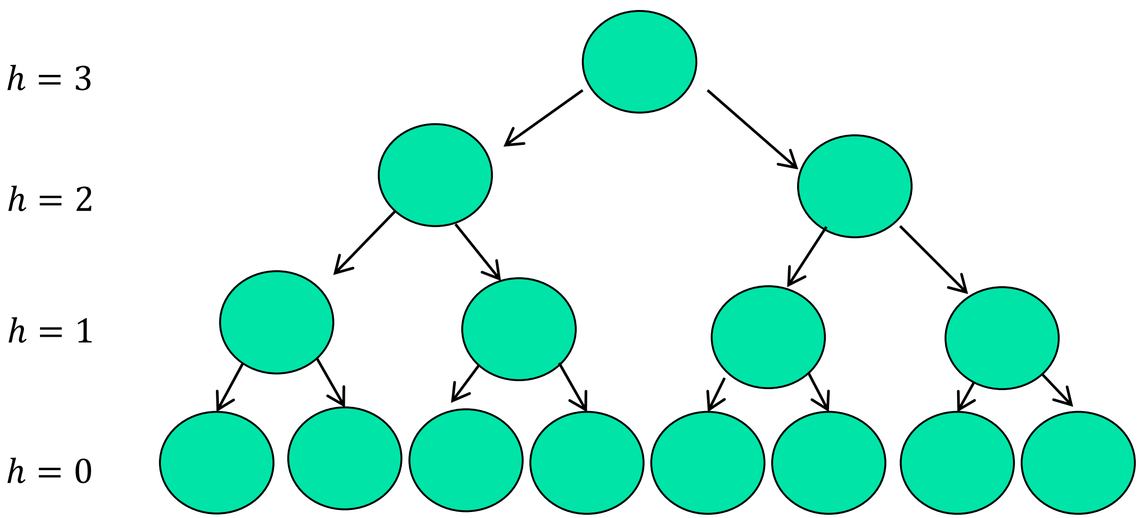


A Lower Bound for the Worst Case

- The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs.
- Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.
- A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm.

A Binary Tree of Height h

- A binary tree of height h has no more than 2^h leaf nodes



A Lower Bound for the Worst Case

- Theorem 8.1
  - Any comparison sort algorithm requires $\Omega(n \log n)$ comparisons in the worst-case.
- Proof
  - It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements.
  - Because each of the $n!$ Permutations of the input appears as some leaf, $n! \leq l$.
  - Since a binary tree of height h has no more than 2^h, we have

$$n! \leq l \leq 2^h.$$

  - Thus,

$$\begin{aligned} h &\geq \log(n!) \\ &= \log(n(n-1)(n-2)\dots 1) \end{aligned}$$

A Lower Bound for the Worst Case

- Theorem 8.1
  - Any comparison wort algorithm requires $\Omega(n \log n)$ comparisons in the worst-case.
- Proof
  - It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements.
  - Because each of the $n!$ Permutations of the input appears as some leaf, $n! \leq l$.
  - Since a binary tree of height h has no more than 2^h leaf nodes, we have

$$n! \leq l \leq 2^h.$$

  - Thus,

$$\begin{aligned} h &\geq \log(n!) \\ &= \log(n(n-1)(n-2)\ldots 1) \\ &= \log n + \log(n-1) + \cdots + \log 1 \end{aligned}$$

A Lower Bound for the Worst Case

- Theorem 8.1
  - Any comparison wort algorithm requires $\Omega(n \log n)$ comparisons in the worst-case.
- Proof
  - It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements.
  - Because each of the $n!$ Permutations of the input appears as some leaf, $n! \leq l$.
  - Since a binary tree of height h has no more than 2^h leaf nodes, we have

$$n! \leq l \leq 2^h.$$

  - Thus,

$$\begin{aligned} h &\geq \log(n!) \\ &= \log(n(n-1)(n-2)\dots 1) \\ &= \log n + \log(n-1) + \cdots + \log 1 \\ &\geq \log n + \log(n-1) + \cdots + \log\left(\frac{n}{2}\right) \end{aligned}$$

A Lower Bound for the Worst Case

- Theorem 8.1
  - Any comparison wort algorithm requires $\Omega(n \log n)$ comparisons in the worst-case.
- Proof
  - It suffices to determine the height of a decision tree in which each permutation appears as a reachable leaf.
  - Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements.
  - Because each of the $n!$ Permutations of the input appears as some leaf, $n! \leq l$.
  - Since a binary tree of height h has no more than 2^h leaf nodes, we have

$$n! \leq l \ \leq 2^h.$$

  - Thus,

$$\begin{aligned} h &\geq \log(n!) \\ &= \log(n(n-1)(n-2) \ldots 1) \\ &= \log n + \log(n-1) + \cdots + \log 1 \\ &\geq \log n + \log(n-1) + \cdots + \log\left(\frac{n}{2}\right) \\ &\geq \frac{n}{2} \log\left(\frac{n}{2}\right) = \Omega(n \log n) \end{aligned}$$

A Lower Bound for the Worst Case

- Corollary 8.2
  - Heapsort and merge sort are asymptotically optimal comparison sorts.
- Proof
  - The O(n lg n) upper bounds on the running times for heapsort and merge sort match the $\Omega(n \log n)$ worst-case lower bound from Theorem 8.1.