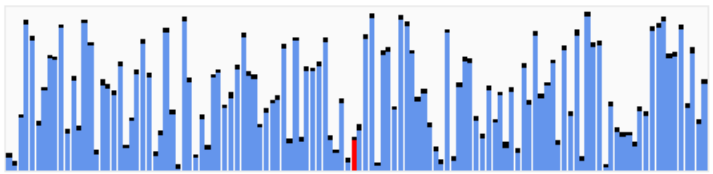
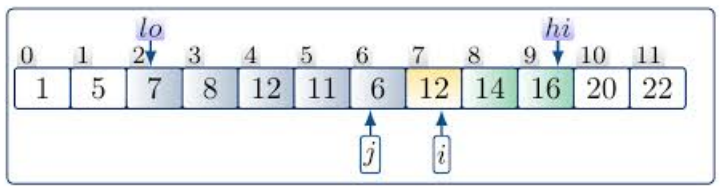

THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND

Algorithms and Data Structures

Lecture 9 Analysis of sorting algorithms

Jiamou Liu
The University of Auckland



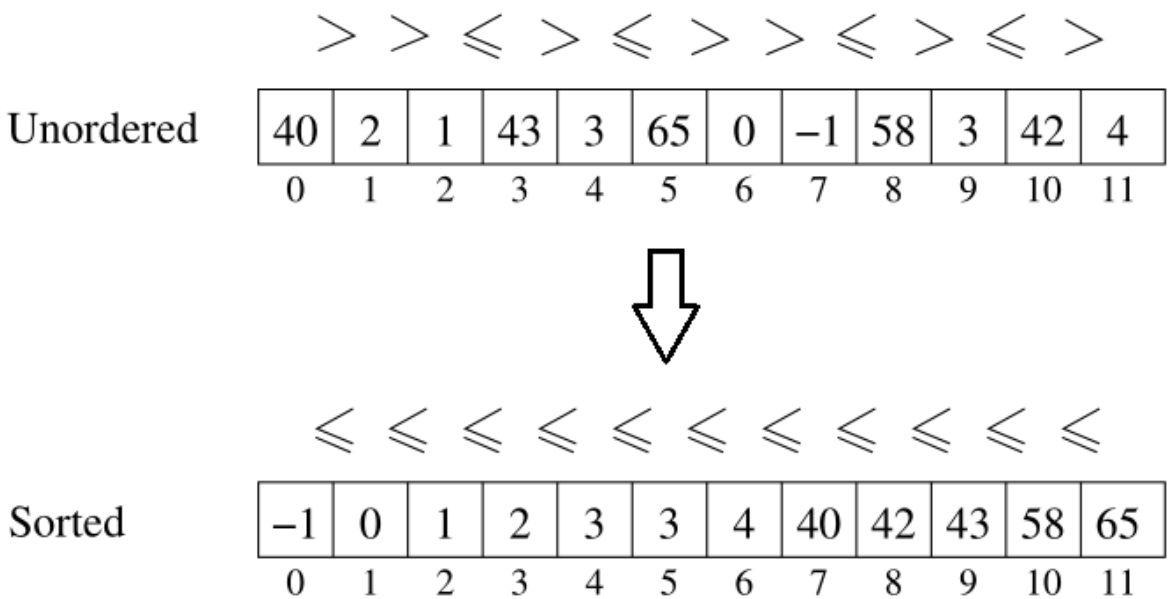


The Sorting Problem

Definition

The sorting problem seeks to arrange a list (i.e., array or linked list) of keys (e.g., integers, strings) so that every adjacent pair of keys is in the designated order.

- **[INPUT]** An (unsorted) list of keys $a = [0..n-1]$
- **[OUTPUT]** A sorted list containing the same set of elements $\{a[0], a[1], \ldots, a[n-1]\}$



Algorithm 1: Selection Sort

Algorithm idea

The selection sort algorithm performs the following operations:

1. Split the input list into sorted and unsorted sublists.
2. Sorted sublist is initially empty, and the unsorted sublist is the whole list.
3. Find a maximal element of the unsorted part by the sequential scan.
4. Move the maximal element to the head of the sorted part.
5. If the unsorted sublist is empty then terminate else go to (3)

Example. Sort the array $[43, 83, 0, 71, 91, 99, 15]$.

43	83	0	71	91	99	15
43	83	0	71	91	99	15
43	83	0	71	91	15	99
43	83	0	71	91	15	99
43	83	0	71	15	91	99
43	83	0	71	15	91	99
43	15	0	71	83	91	99
43	15	0	71	83	91	99
43	15	0	71	83	91	99
43	15	0	71	83	91	99
0	15	43	71	83	91	99
0	15	43	71	83	91	99
0	15	43	71	83	91	99
0	15	43	71	83	91	99

- Red part is sorted.
- Black part is unsorted.

Selection Sort Algorithm

```
1: function SelectionSort(a[0..n − 1])
2:     for ℓ ← n − 1 to 0 do
3:         k ← findmax(a[0..ℓ])        ▷ Find k where a[k] is maximum in a
4:         if k ≠ ℓ then
5:             swap(a, ℓ, k)           ▷ Swap a[ℓ] with a[k]
6:     return a
```

`findmax` algorithm

```
1: function findmax(a[0..ℓ])
2:     max ← 0
3:     for i ← 1 to ℓ do
4:         if (a[max] < a[i]) then max ← i        ▷ Comparison
5:     return max
```

- `findmax` iterates over the entire unsorted part (setting $i = 0..\ell$),
- Each step of `findmax` compares $a[max]$ with $a[i]$.

Comparison-based Sorting Algorithm

Definition

A sorting algorithm is comparison-based if it only uses the order relation to compare keys, i.e., we determine the order of elements by answering questions of the type "is $x < y$?".

E.g.

- Selection sort is a comparison-based sorting algorithm: comparisons are done in `findmax`.
- Non-comparison-based sorting algorithms include bucket sort, radix sort, etc.[1]

Important operations in a comparison-based sort:

- comparison: $\Theta(1)$ running time.
- swap: 3 variable updates. $\Theta(1)$ running time.

[1] Out of the scope of this course.

Analysis of Selection Sort

Question. What is the running time of the selection sort algorithm?

```
1: function SelectionSort(a[0..n − 1])
2:     for ℓ ← n − 1 to 0 do
3:         k ← findmax(a[0..ℓ])      ▷ Find k where a[k] is maximum in a
4:         if k ≠ ℓ then
5:             swap(a, ℓ, k)                    ▷ Swap a[ℓ] with a[k]
6:     return a
```

- Finding the maximum of $a[0..\ell]$ takes ℓ comparisons.
- Number of comparisons: $(n-1) + (n-2) + \cdots + 1$ which is $\Theta(n^2)$.
- Number of swaps: at most $n - 1$
- Worst-case running time: $\Theta(n^2)$

Algorithm 2: Merge Sort

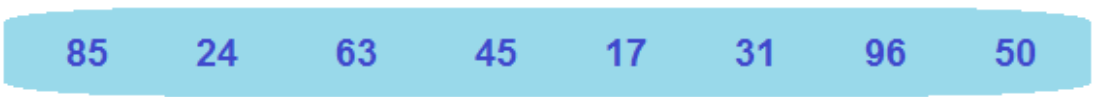
Mergesort is a comparison-based algorithm that solves the sorting problem using the idea of divide-and-conquer:

1. To sort an input list, split it into two sublists;
2. recursively sorts each sublist;
3. then combine the sorted sublists to sort the original list.

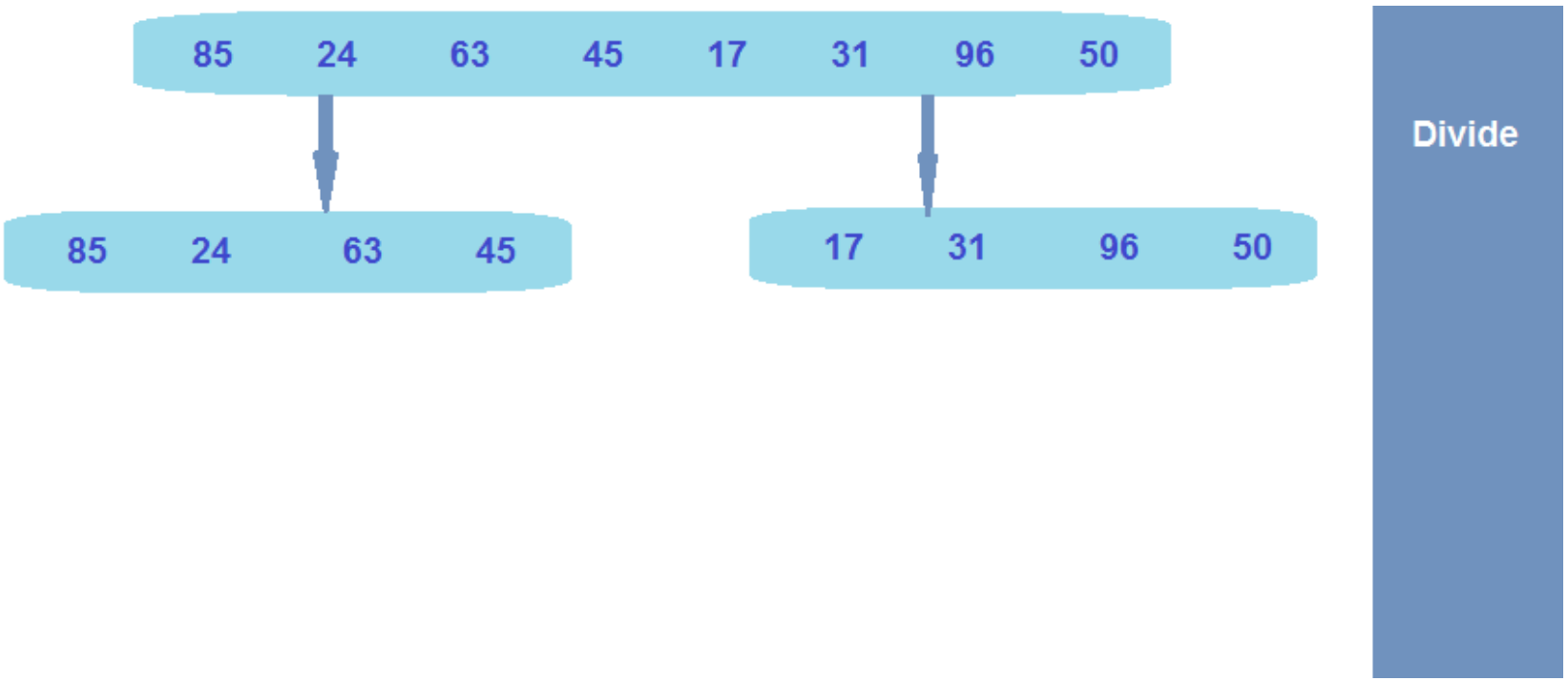
Mergsort algorithm

```
function MergeSort(list input[0..n − 1])
    if n > 1 then
        s ← ⌊(n − 1)/2⌋                          ▷ median index of list
        ℓ ← mergesort(input[0..s])                ▷ sort left half
        r ← mergesort(input[s + 1..n − 1])        ▷ sort right half
        input ← merge(ℓ, r)                       ▷ merge both halves
    return input
```

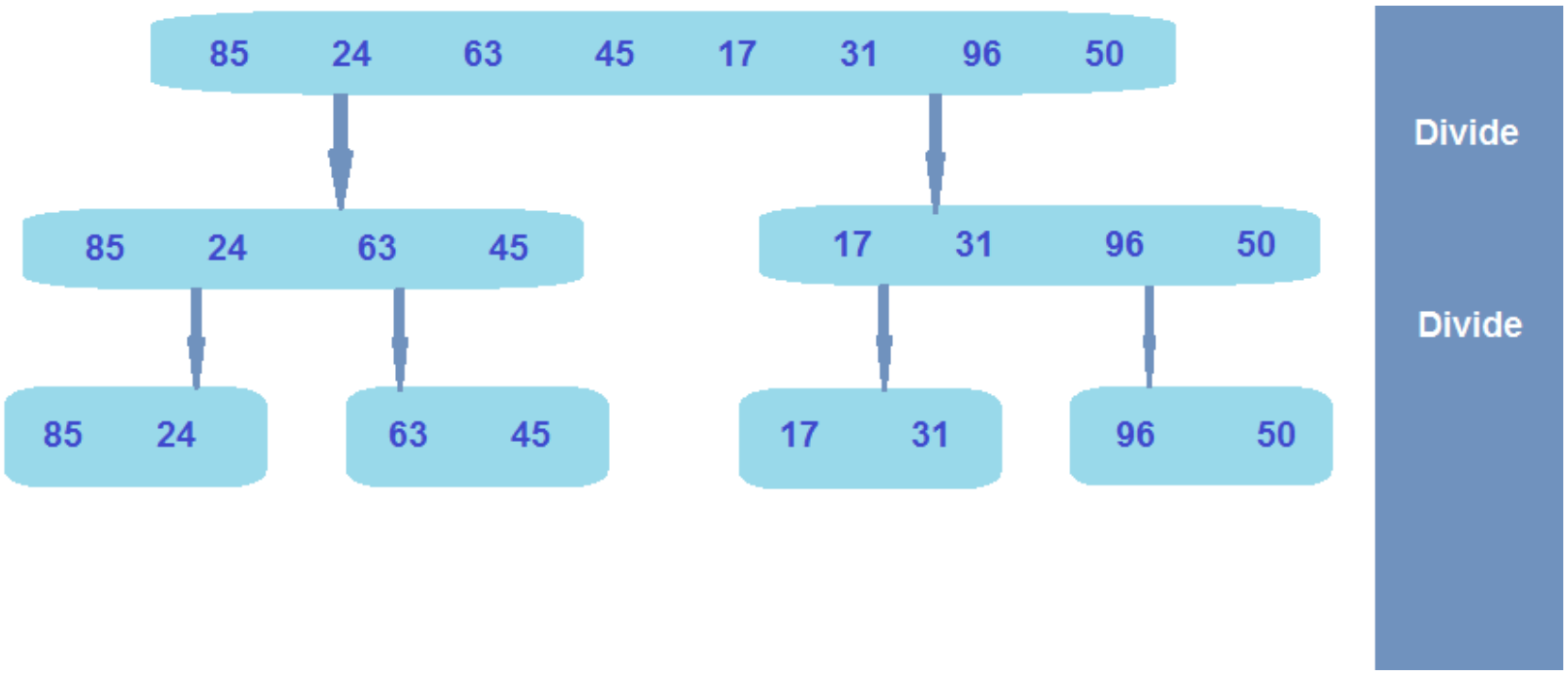
Merge Sort: An Example



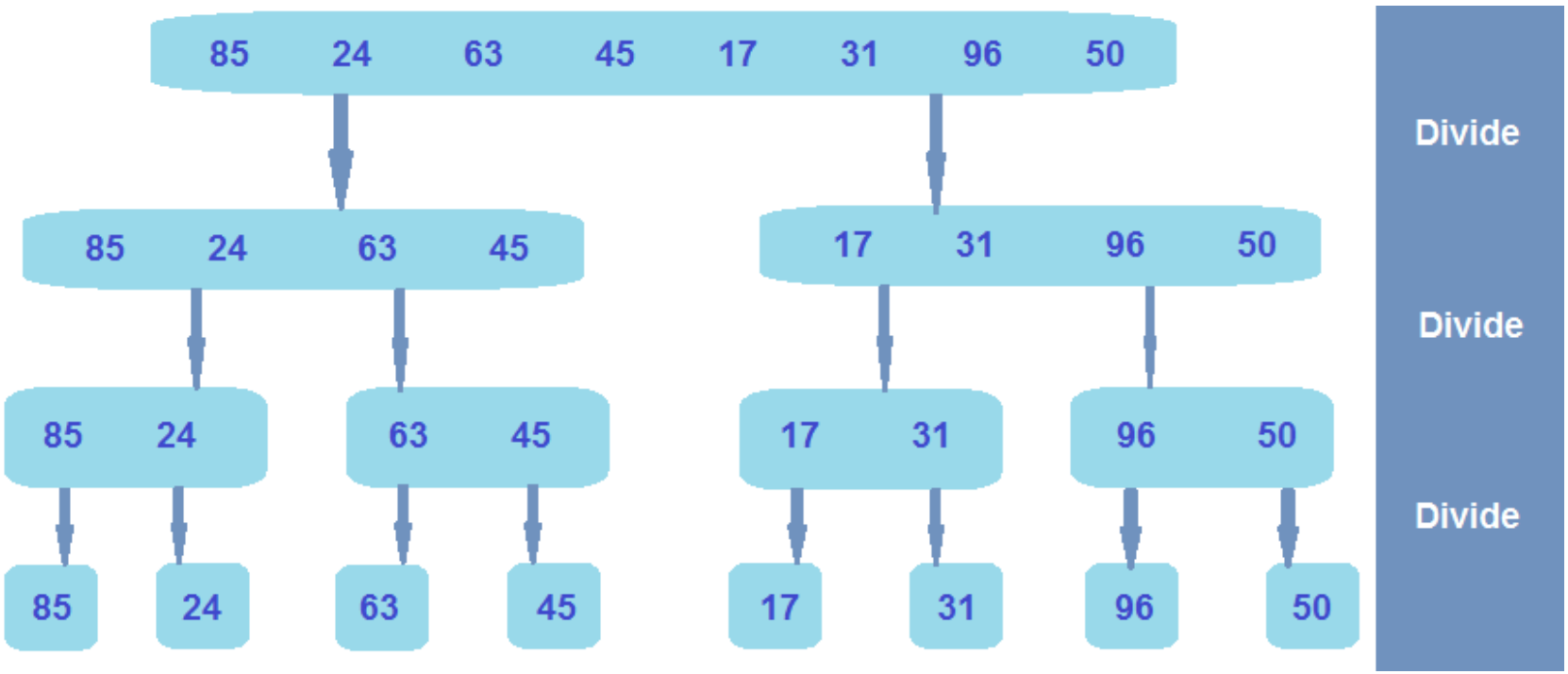
Merge Sort: An Example



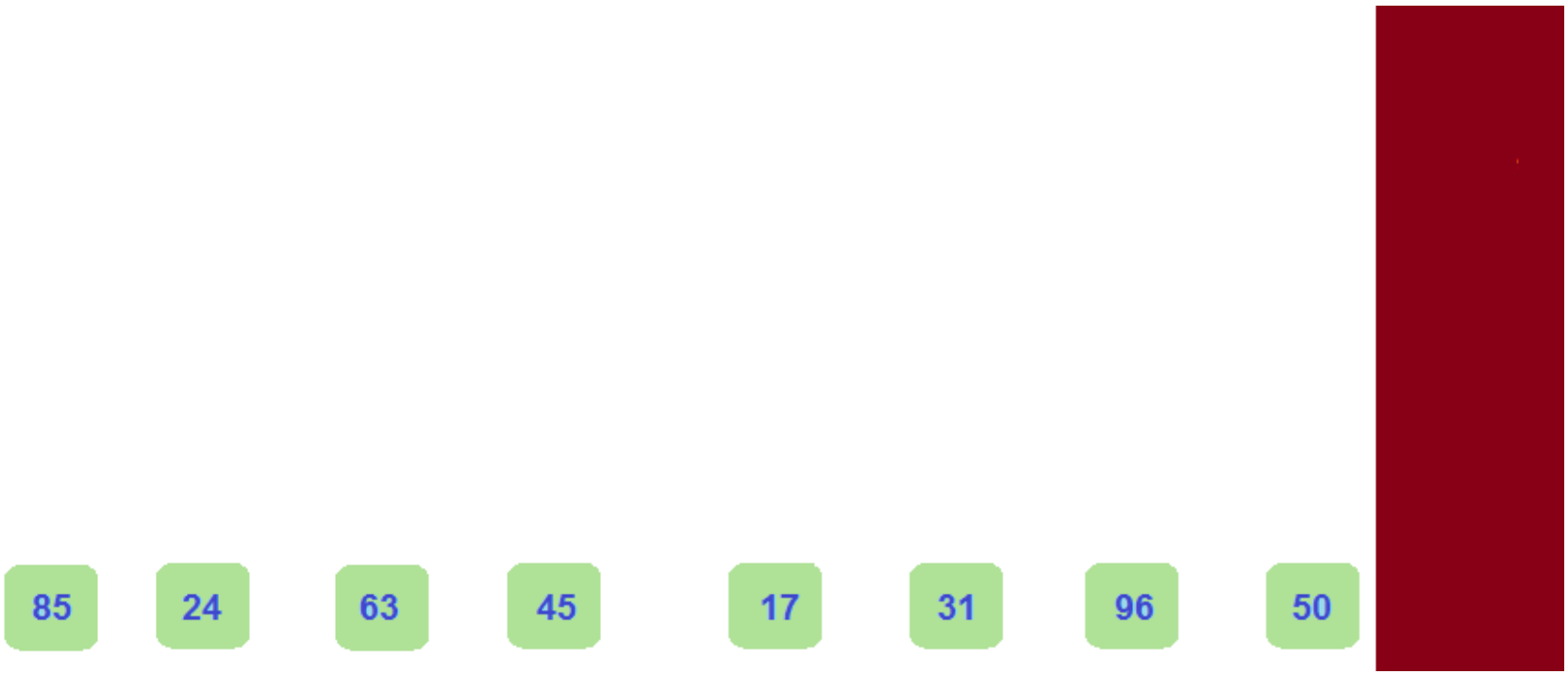
Merge Sort: An Example



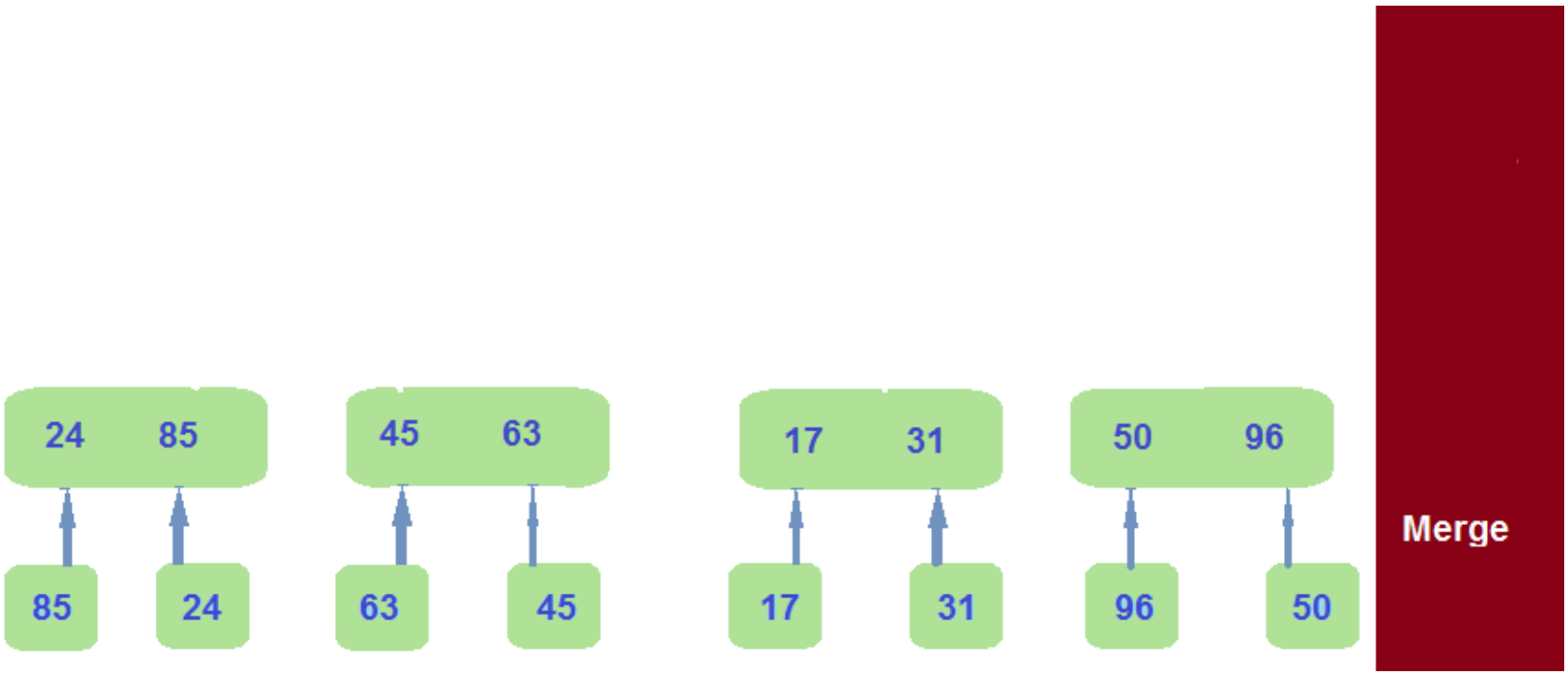
Merge Sort: An Example



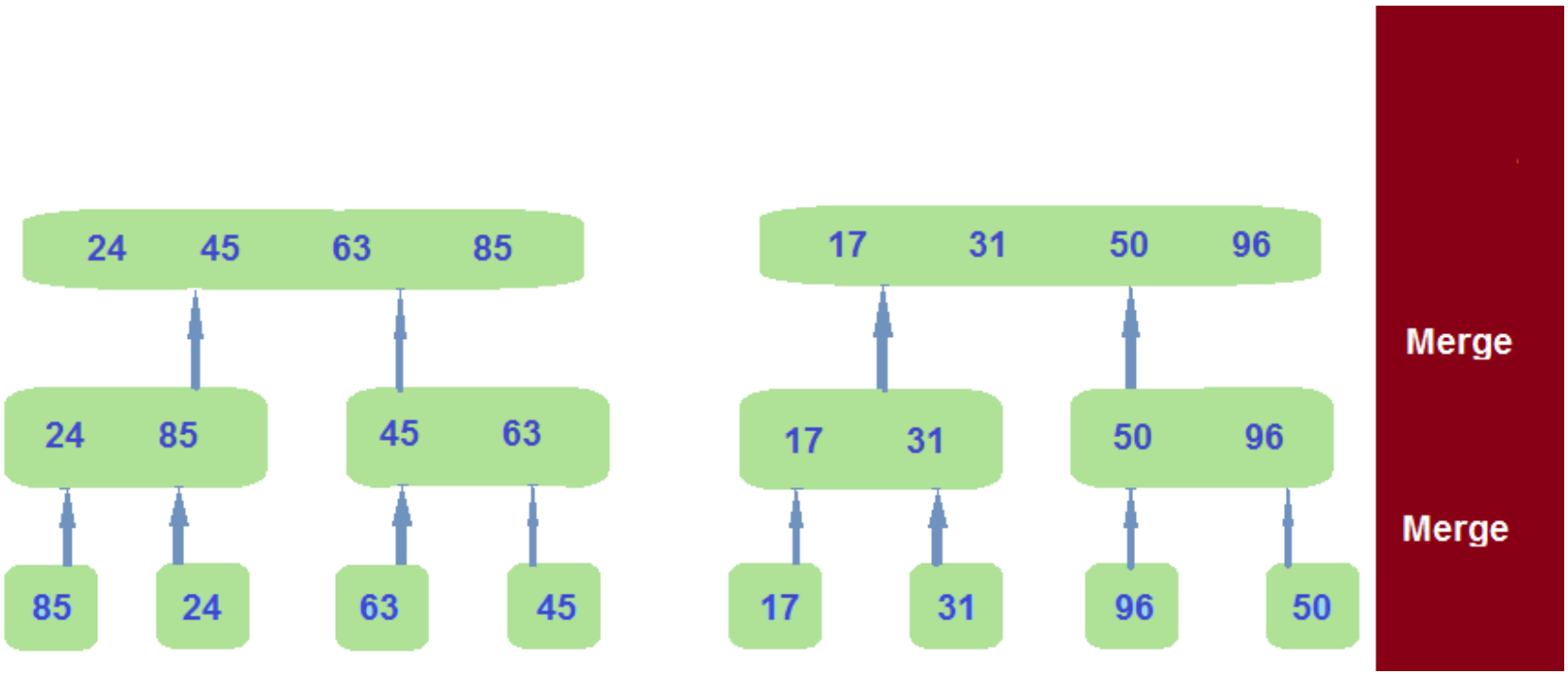
Merge Sort: An Example



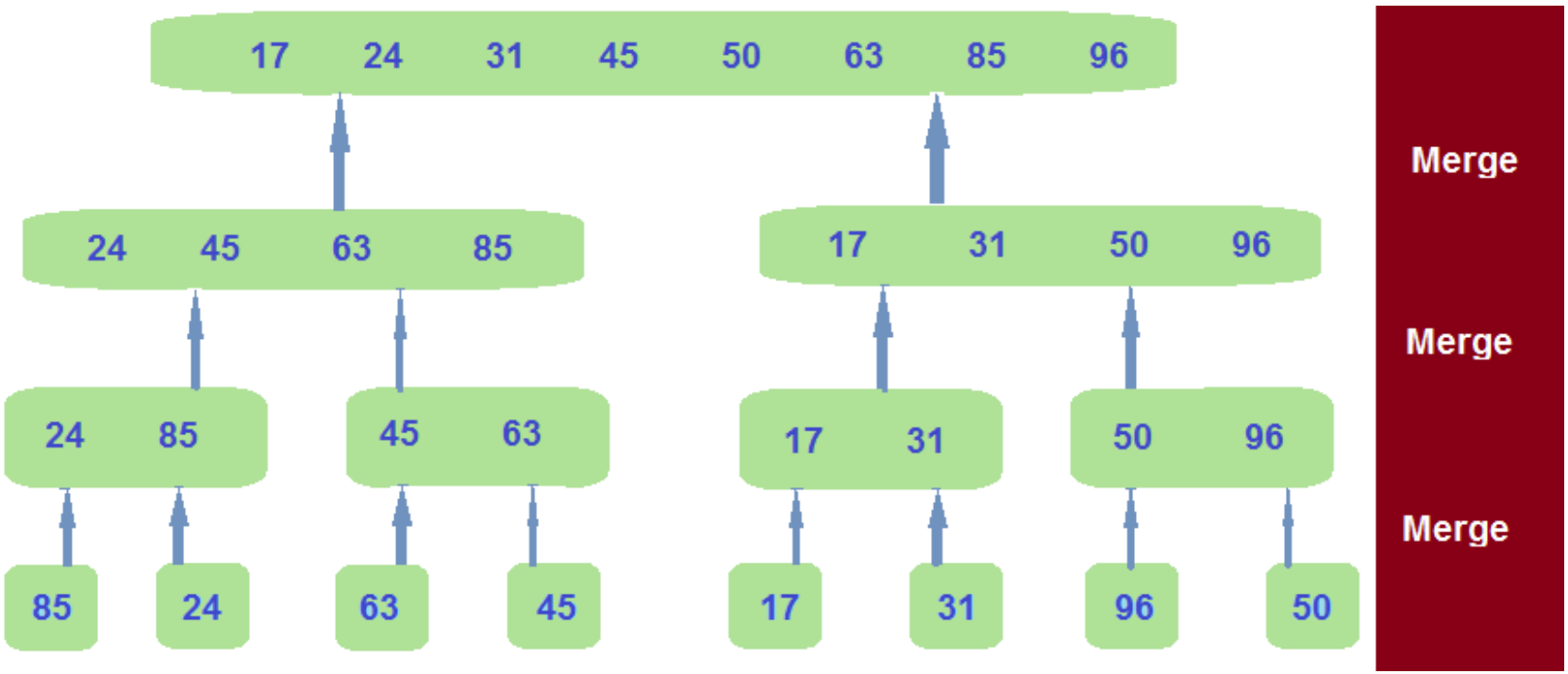
Merge Sort: An Example



Merge Sort: An Example



Merge Sort: An Example



The Merge Operation

Question. How to perform the merge operation efficiently?

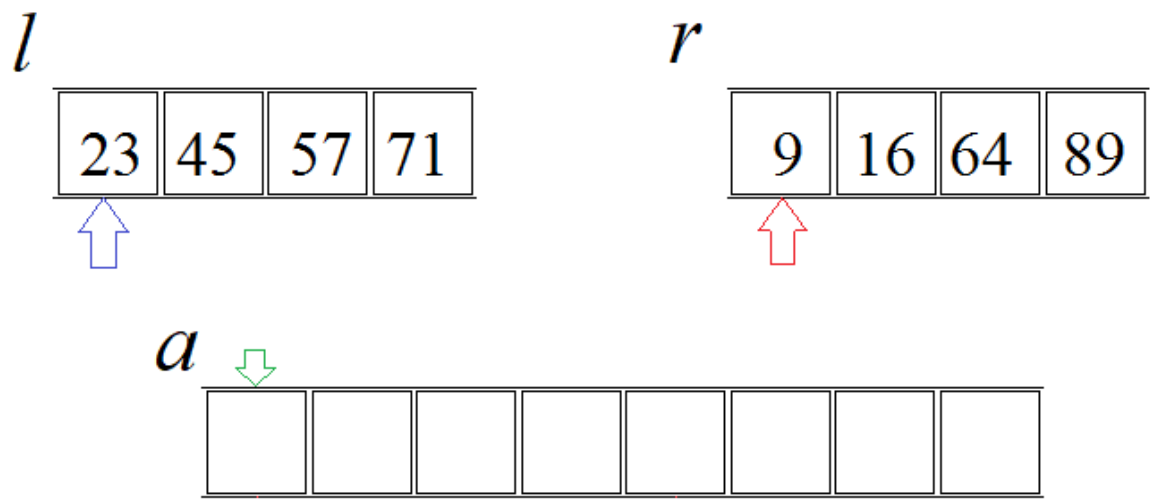
Idea:

- Create a pointer for each sublist.
- Create a merged list and a third pointer for this list.
- Start pointers at the beginning of each list.
- Compare the elements being pointed to and choose the lesser one to start the sorted list. Increment that pointer.
- Iterate until one pointer reaches the end of its list.
- Copy remaining elements of the other list to the end of the final sorted list.

The Merge Procedures

Suppose

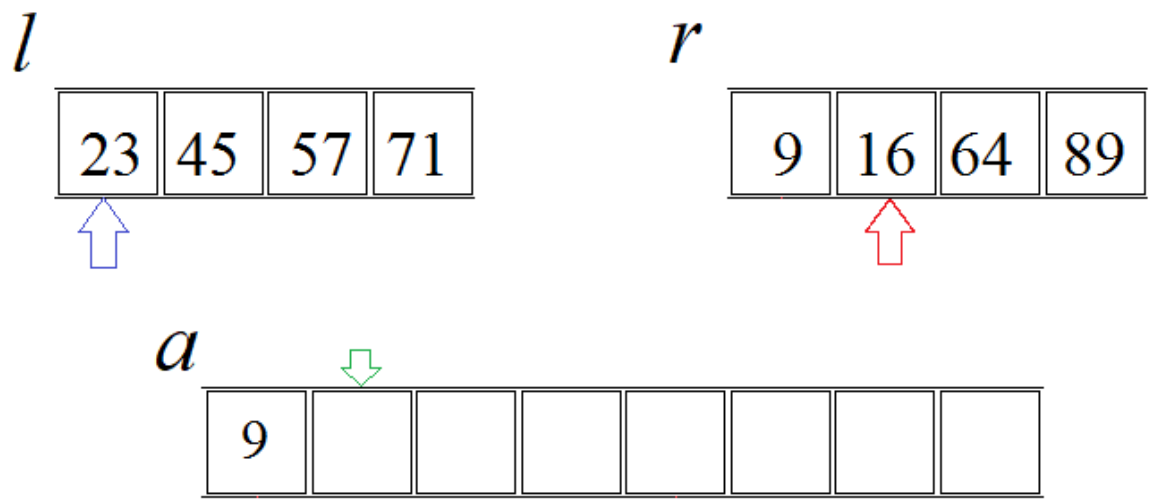
- ℓ[0..s] are sorted
- r[0..t] are sorted



The Merge Procedures

Suppose

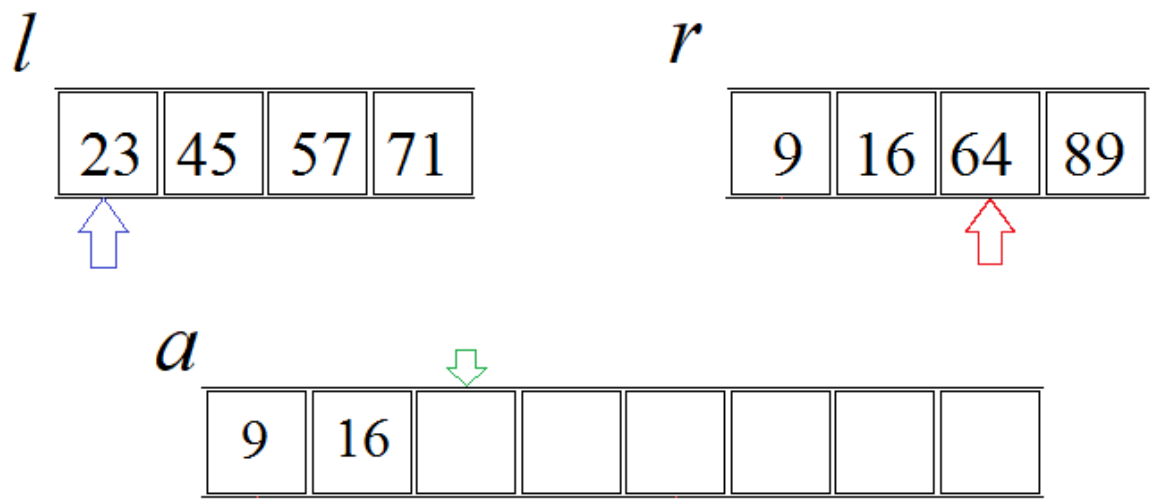
- ℓ[0..s] are sorted
- r[0..t] are sorted



The Merge Procedures

Suppose

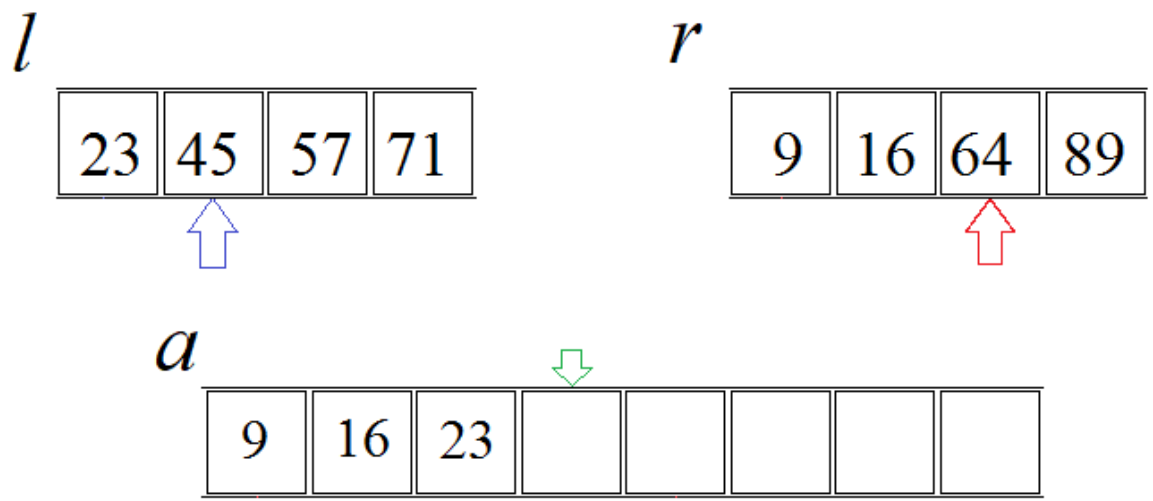
- ℓ[0..s] are sorted
- r[0..t] are sorted



The Merge Procedures

Suppose

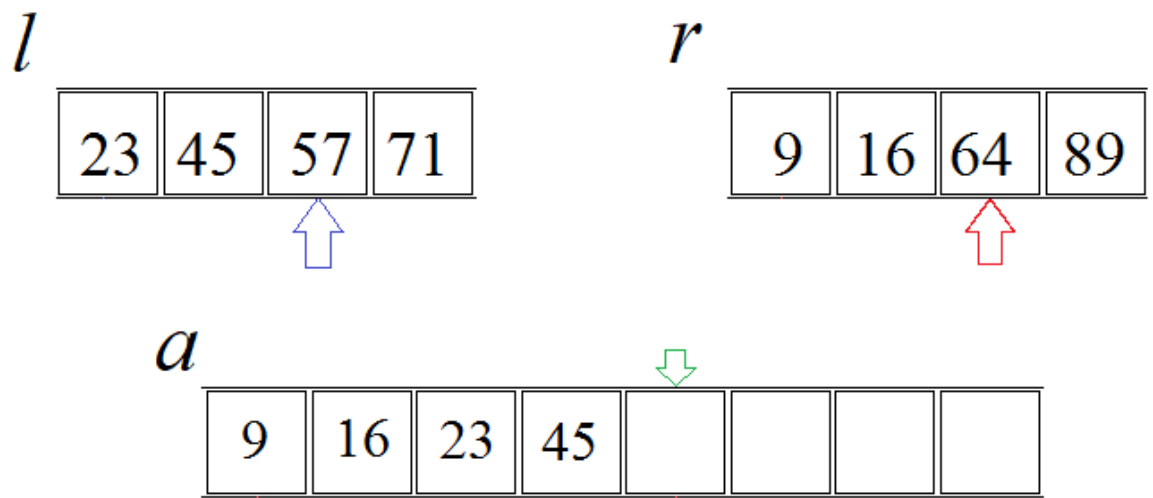
- ℓ[0..s] are sorted
- r[0..t] are sorted



The Merge Procedures

Suppose

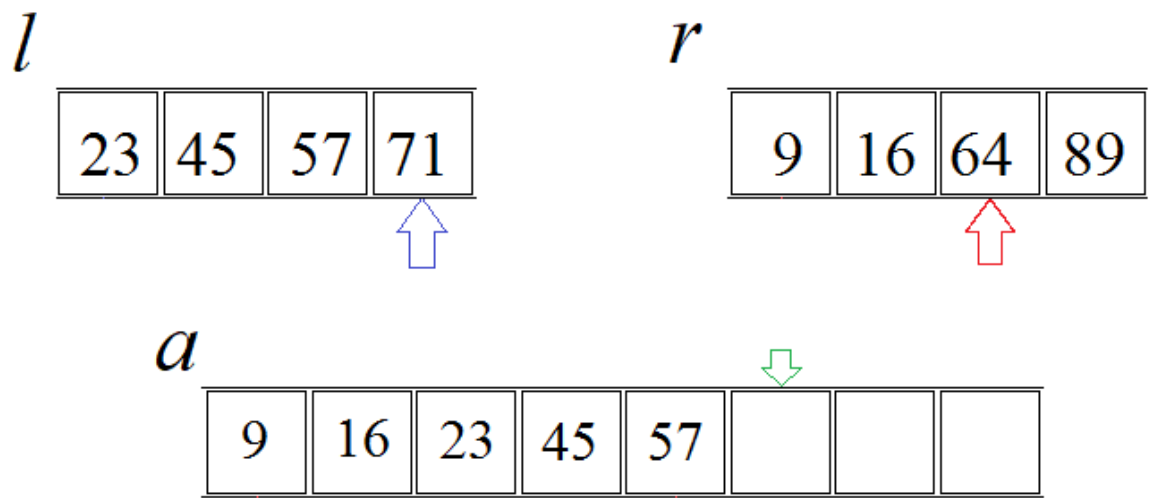
- ℓ[0..s] are sorted
- r[0..t] are sorted



The Merge Procedures

Suppose

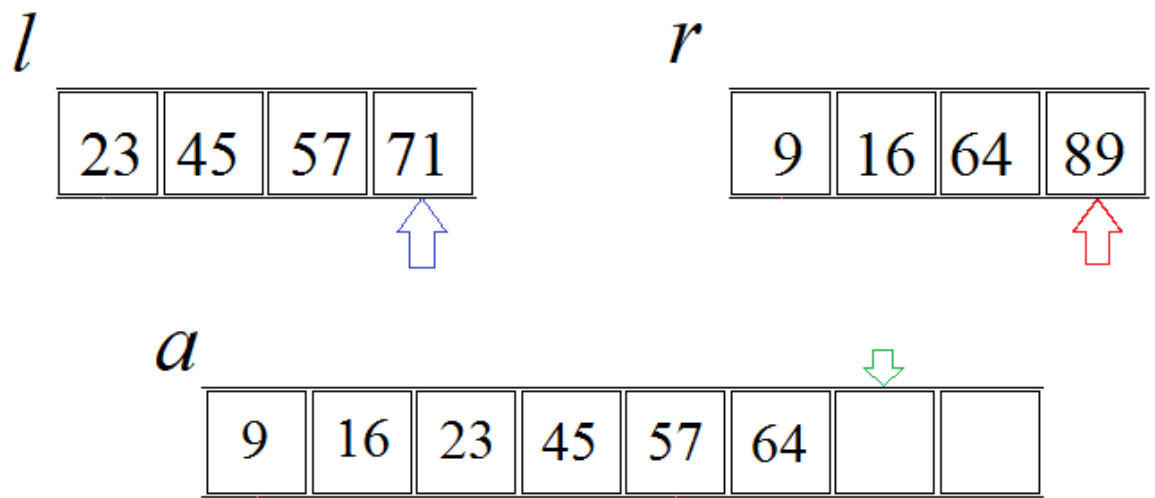
- ℓ[0..s] are sorted
- r[0..t] are sorted



The Merge Procedures

Suppose

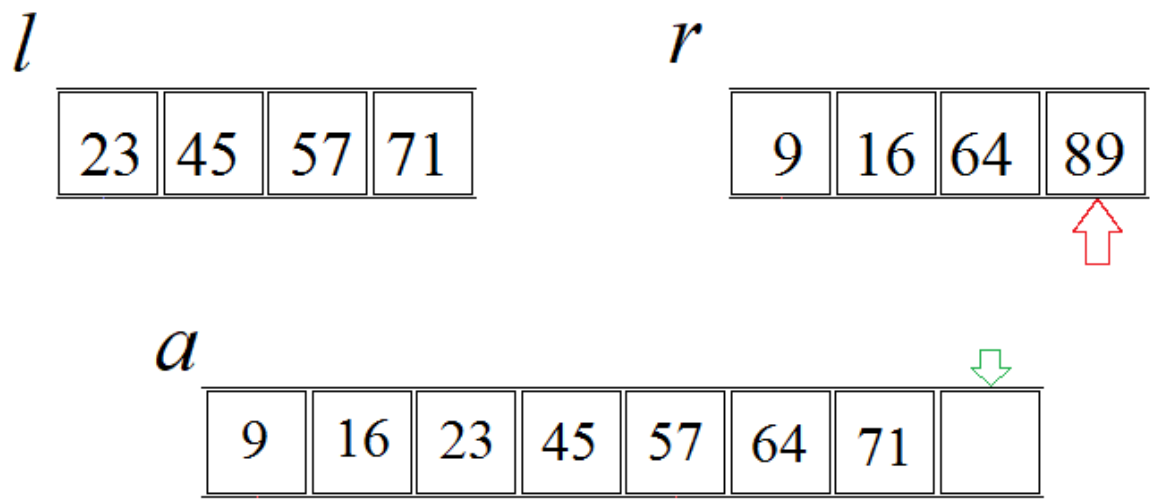
- ℓ[0..s] are sorted
- r[0..t] are sorted



The Merge Procedures

Suppose

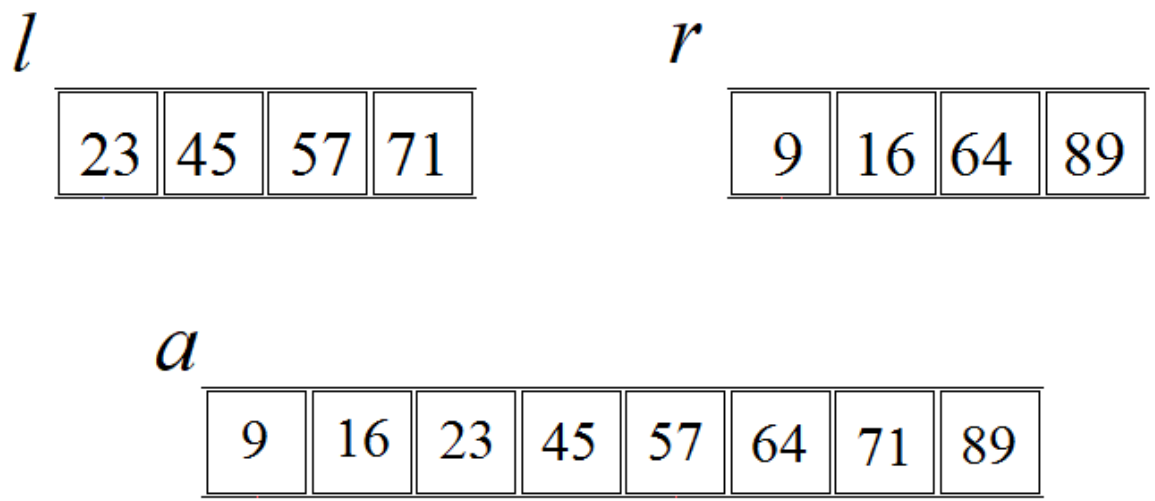
- ℓ[0..s] are sorted
- r[0..t] are sorted



The Merge Procedures

Suppose

- ℓ[0..s] are sorted
- r[0..t] are sorted



The merge operation

function MERGE(list $\ell[0..s]$, $r[0..t]$)
    $p_1 \leftarrow 0$
    $p_2 \leftarrow 0$
    $p_3 \leftarrow 0$
    Initialise a new list $a[0..s+t+1]$
    while $p_1 \leq s$ and $p_2 \leq t$ **do**
        if $r[p_2] < \ell[p_1]$ **then**
            $a[p_3] \leftarrow r[p_2]; p_2 \leftarrow p_2 + 1; p_3 \leftarrow p_3 + 1$
        else
            $a[p_3] \leftarrow \ell[p_1]; p_1 \leftarrow p_1 + 1; p_3 \leftarrow p_3 + 1$
    while $p_1 \leq s$ **do**
        $a[p_3] \leftarrow \ell[p_1]; p_1 \leftarrow p_1 + 1; p_3 \leftarrow p_3 + 1$
    while $p_2 \leq t$ **do**
        $a[p_3] \leftarrow r[p_2]; p_2 \leftarrow p_2 + 1; p_3 \leftarrow p_3 + 1$
    return a

Running Time Analysis

Mergesort recurrence: Let $T(n)$ denote the running time of merge sort over a list of n items. Then

$T(n) = 2T(n/2) + n$ when $n > 1$ and n is a power of 2, and $T(1) = 0$

Apply the master theorem, mergesort runs in time $\Theta(n \log n)$.

Running Time Analysis

Mergesort recurrence: Let $T(n)$ denote the running time of merge sort over a list of n items. Then

$T(n) = 2T(n/2) + n$ when $n > 1$ and n is a power of 2, and $T(1) = 0$

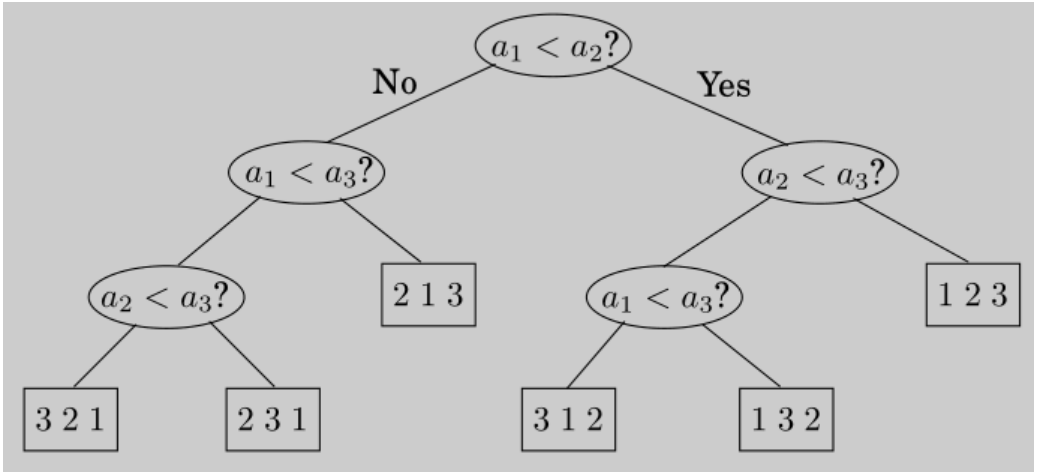
Apply the master theorem, mergesort runs in time $\Theta(n \log n)$.

Question.

- Is merge sort the most efficient algorithm for sorting?
- What is the shortest time must any sorting algorithms run in?

Lower Bound of Comparison-based Sorting

The comparison tree contains all possible outcomes of comparisons



- The depth of the tree is the number of comparisons on the longest path from the root to a leaf.
- For $n > 0$, let $h(n)$ be the height of the comparison tree for n numbers.

Fact.

- Any comparison-based sorting algorithm uses at least $h(n)$ comparisons in the worst case.
- Otherwise there are permutations that are indistinguishable.

Observation 1

- The comparison tree for n numbers is a binary tree
- A binary tree with k leaves has at least height $\log k$

Therefore $h(n) \geq \log k$ where k is the number of leaves.

Observation 2

- Every leaf in the comparison tree is a permutation of n numbers
- There are $n!$ number of permutations with n numbers

Therefore $k = n!$

In conclusion,

$$h(n) \geq \log n!$$

$$\begin{aligned} n! &= 1 \times 2 \times 3 \times \ldots \times n-1 \times n \\ &> \lfloor n/2 \rfloor \times \ldots \times n-1 \times n \\ &> (n/2)^{n/2} \end{aligned}$$

We thus have $\log n! > \log(n/2)^{n/2} = n/2(\log n - 1)$ and

$$h(n) \text{ is } \Omega(n \log n)$$

$$\begin{aligned} n! &= 1 \times 2 \times 3 \times \ldots \times n-1 \times n \\ &> \lfloor n/2 \rfloor \times \ldots \times n-1 \times n \\ &> (n/2)^{n/2} \end{aligned}$$

We thus have $\log n! > \log(n/2)^{n/2} = n/2(\log n - 1)$ and

$$h(n) \text{ is } \Omega(n \log n)$$

Conclusion:

- Any comparison-based sorting algorithm must use $\Omega(n \log n)$ number of comparisons in the worst case.
- The best time complexity for any comparison-based sorting algorithm is $\Theta(n \log n)$.
- Merge sort is an optimal comparison-based sorting algorithm.

Summary

In this lecture, we look at algorithms for the sorting problem. We introduced two comparison-based sorting algorithms.

1. **Algorithm 1: SelectionSort.** Running time $O(n^2)$
2. **Algorithm 2: MergeSort.**
   - Based on the divide and conquer technique
   - Split input arrays into halves
   - Perform merge operation that involves comparisons
   - Running time $O(n \log n)$
   - $O(n \log n)$ matches the running time lower bound for comparison-based sorting algorithms.


THE UNIVERSITY OF
AUCKLAND
Te Whare Wānanga o Tāmaki Makaurau
NEW ZEALAND