COMP S265F Unit 3: Divide and Conquer: Linear Time Selection

Dr. Keith Lee

School of Science and Technology

The Open University of Hong Kong

Overview

- **Divide and Conquer**
- **Example 1: Finding maximum of n numbers**
  - Divide & Conquer approach
  - Proof of correctness: by M.I.
  - Time complexity analysis: Simplifying n to powers of 2
- **Example 2: MergeSort**
  - Algorithm
  - Sample run: Divide & Conquer, Merging two sorted lists
  - Time complexity analysis
- **Example 3: Linear time selection**
  - Basic Operation: Divide a set N by a number **a**
  - $O(n^2)$-time algorithm
  - $O(n)$-time algorithm using a clever way to pick the number **a**

Divide and Conquer

- A basic algorithm design technique

- **Divide**
  - Given some problem, divide it into a number of similar, but simpler problems.
  - Solve each of these problems recursively.

- **Conquer (or Combine)**
  - Combine the solutions of each of these subproblems into a solution of the original problem.

Example 1: Finding the maximum of n numbers

A divide & conquer approach:

- If $n = 1$, then return the number immediately.
- Otherwise, do the following:
  - divide the numbers into two equal groups,
  - for each of the 2 groups, find recursively the maximum of that group.
  - compare the two numbers found and return the larger one.

Example: Finding the maximum of n numbers

- If $n = 1$, then return the number immediately.
- Otherwise, do the following:
  - divide the numbers into two equal groups,
  - for each of the 2 groups, find recursively the maximum of that group.
  - compare the two numbers found and return the larger one.

Pseudo-code:

```
max([a_1, a_2, …, a_n]):
    if n = 1:
        return a_1
    else:
        m1 = max([a_1, a_2, …, a_{n/2}])
        m2 = max([a_{n/2+1}, a_{n/2+2}, …, a_n])
        compare m1, m2 and return the larger one
```

Proof of Correctness

- Base Case: When n=1, `max` is obviously correct.
- Induction Hypothesis: Suppose `max` correctly finds the maximum of any n-1 numbers.
- Consider any input of n numbers $a_1,\ldots,a_n$.

Proof of Correctness (cont')

- Base Case: When n=1, `max` is obviously correct.
- Induction Hypothesis: Suppose `max` correctly finds the maximum of any n-1 numbers.
- Consider any input of n numbers $a_1,\ldots,a_n$.
- Note that `max` divides the n numbers into two groups $S1=\{a_1, \ldots, a_{n/2}\}$ and $S2=\{a_{n/2+1}, \ldots, a_n\}$ and call recursively
  - m1= `max`($[a_1, \ldots, a_{n/2}]$), and
  - m2 = `max` ($[a_{n/2+1}, \ldots, a_n]$),

and returns the maximum m of m1 and m2.

Proof of Correctness (cont')

- Induction Hypothesis: Suppose `max` correctly finds the maximum of any n-1 numbers.
- Note that `max` divides the n numbers into two groups S1={a_1, …, $a_{n/2}$} and S2={$a_{n/2+1}$, …, a_n} and call recursively
  - m1= `max`([a_1, …, $a_{n/2}$]), and
  - m2 = `max` ([$a_{n/2+1}$, …, a_n]),

  and returns the maximum m of m1 and m2.
- Obviously, |S1| and |S2| is smaller than or equal to n-1.
- By the Induction Hypothesis, m1 is the maximum of S1 and m2 is the maximum of S2.
- As m is the maximum of S1 $\cup$ S2, `max` correctly finds the maximum of the n numbers, which completes the proof.

Time complexity

- Let T(n) be the total number of comparisons `max` made (in the worst case) to find the maximum of n numbers.
- How large is T(n)?

```
max([a1, a2, …, an]):
    if n = 1:
        return a1
    else:
        m1 = max([a1, a2, …, an/2])
        m2 = max([an/2+1, an/2+2, …, an])
        compare m1, m2 and return the larger one
```

T(n)

O(1)

$T\left(\frac{n}{2}\right)$

$T\left(\frac{n}{2}\right)$

O(1)

- Therefore, $T(n) = 2\,T\left(\frac{n}{2}\right) + O(1)$.

Time complexity: Simplifying n

- In this example, we assume n is a power of 2.
- Making this assumption simplifies the computation because we don't need to consider the annoying case when $\frac{n}{2}$ is not an integer.
- Even under this assumption, the analysis gives us sufficient information about T(n).
- To handle the case when n is general, we need to use the ceiling and floor function in our computation.

What is T(n)?

- Let $T(n) = 2\,T\left(\frac{n}{2}\right) + c$, for some constant c.
- Let $\mathbf{n = 2^k}$, so $\mathbf{k = \log_2 n}$ is an integer.
- $T(n) = 2\,T\left(\frac{n}{2}\right) + c$

$= 2\left(2\,T\left(\frac{n}{4}\right) + c\right) + c = 4\,T\left(\frac{n}{4}\right) + c \times (1 + 2)$

$= 4\left(2\,T\left(\frac{n}{8}\right) + c\right) + c \times (1 + 2) = 8\,T\left(\frac{n}{8}\right) + c \times (1 + 2 + 4)$

$= \ldots$

$= n\,T\left(\frac{n}{n}\right) + c \times \left(1 + 2 + 4 + \ldots + \frac{n}{2}\right)$

$= n\,T(1) + c \times (1 + 2^1 + 2^2 + \ldots + \mathbf{2^{k-1}})$ [$T(1)$, c are constants.]

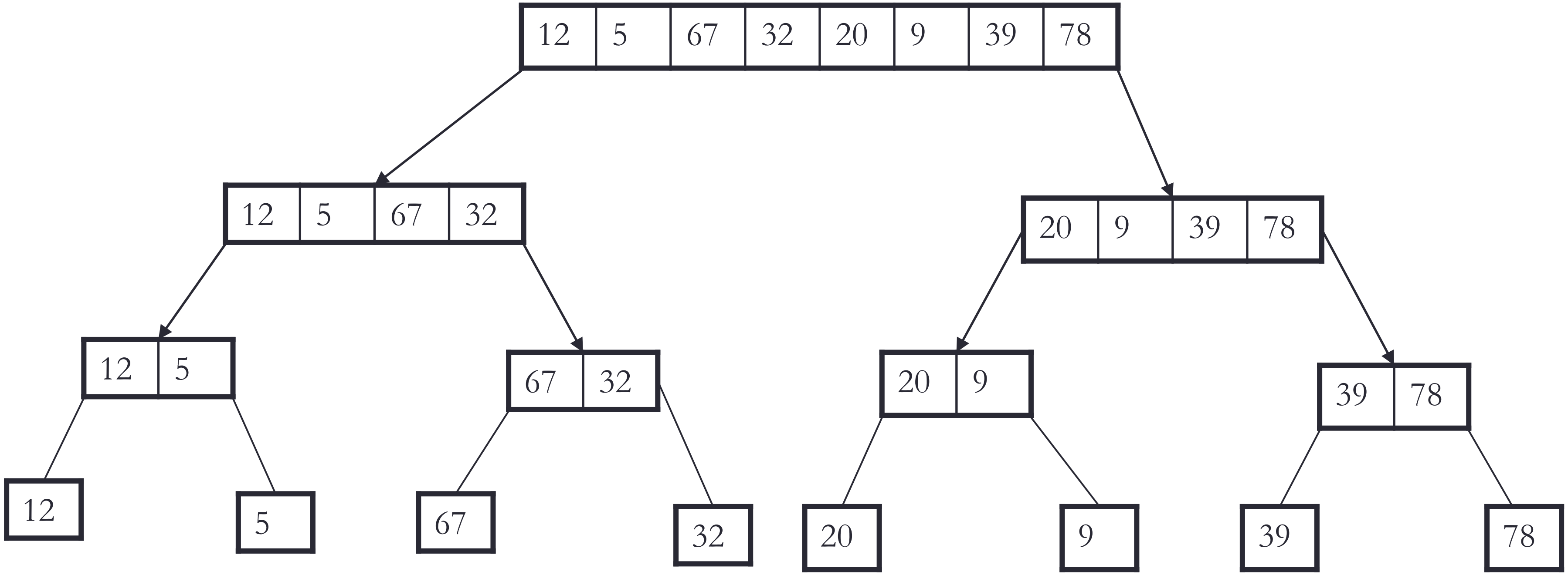
$= O\left(n + \frac{2 \times \mathbf{2^{k-1}} - 1}{2 - 1}\right) = O(n + n - 1) = O(n).$

Example 2: MergeSort

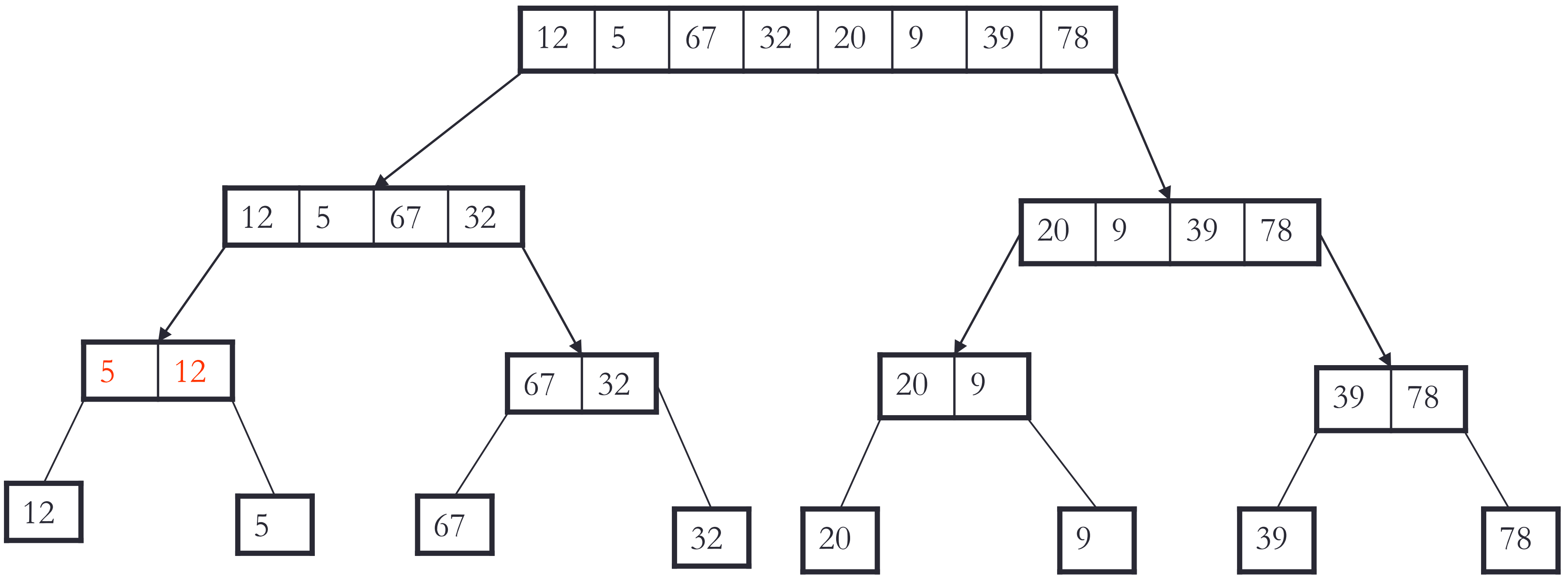
- We can apply an almost identical algorithmic framework to solve another problem.
- For example, we can follow the same framework to sort n numbers.

```
sort([a_1, a_2, …, a_n]):
    if n = 1:
        return [a_1]
    else:
        S1 = sort([a_1, a_2, …, a_{n/2}])
        S2 = sort([a_{n/2+1}, a_{n/2+2}, …, a_n])
        merge S1, S2 into a single sorted list
```

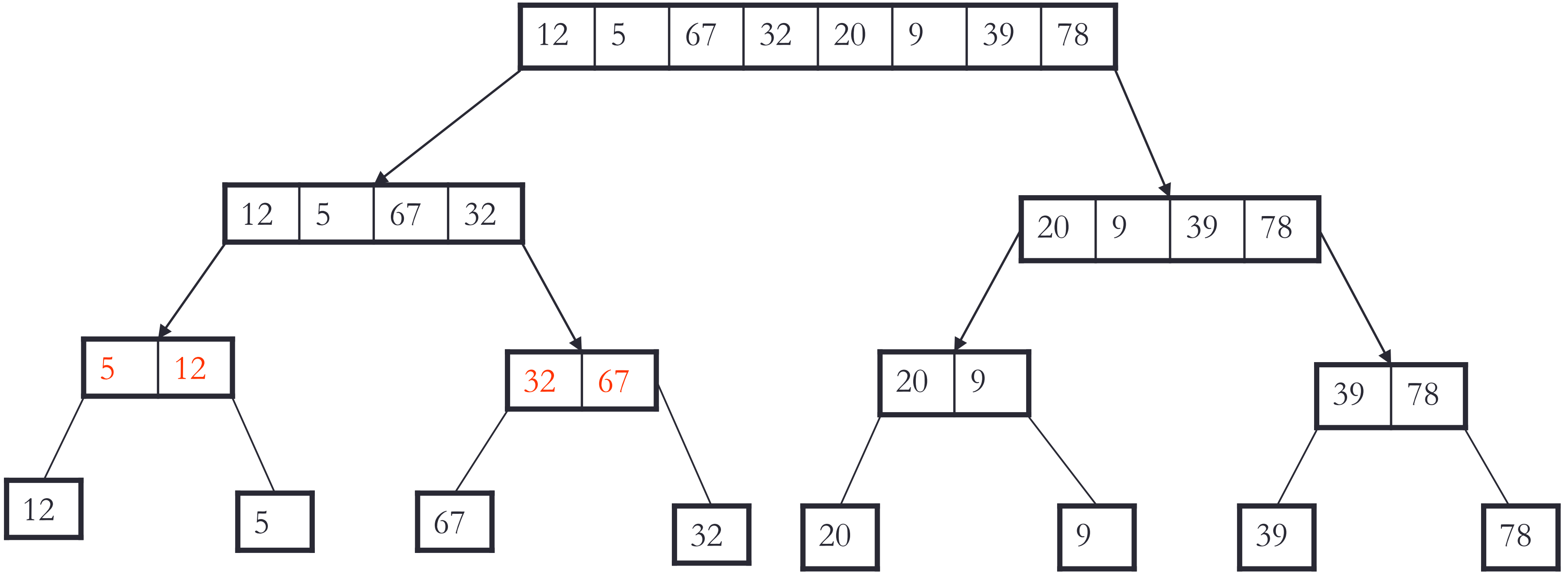
MergeSort: A sample run - Divide



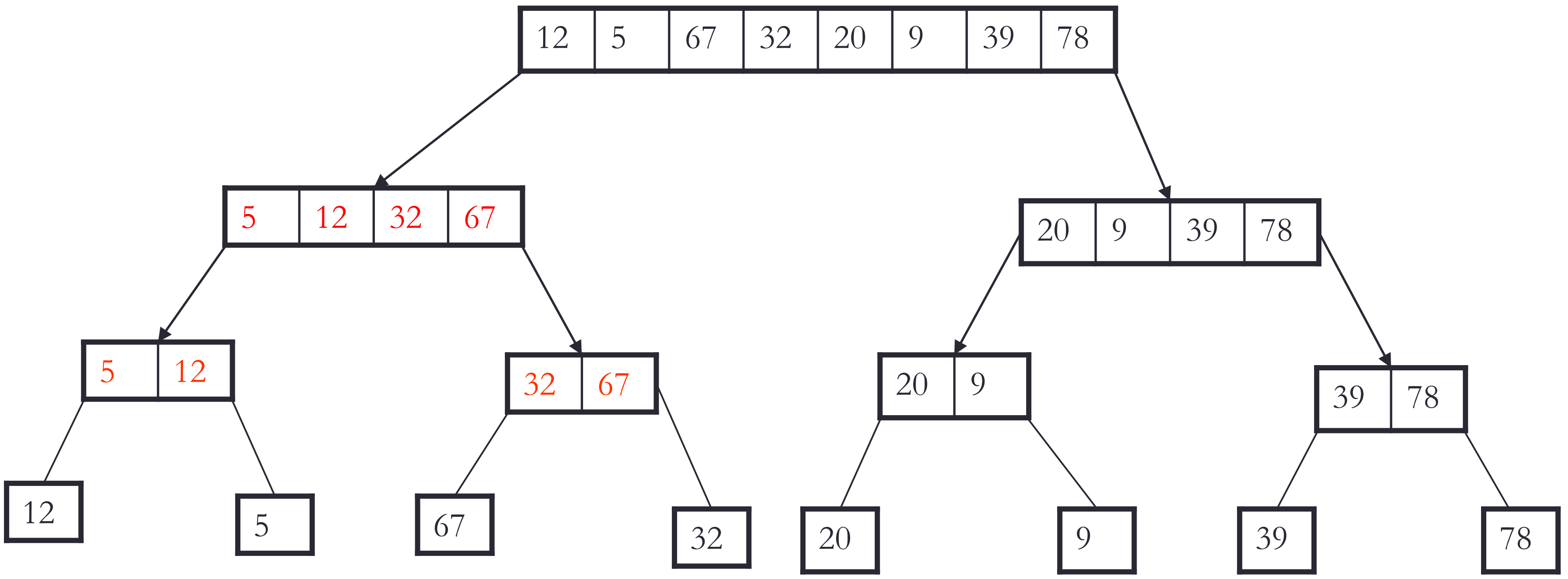
MergeSort: Conquer (Step 1)



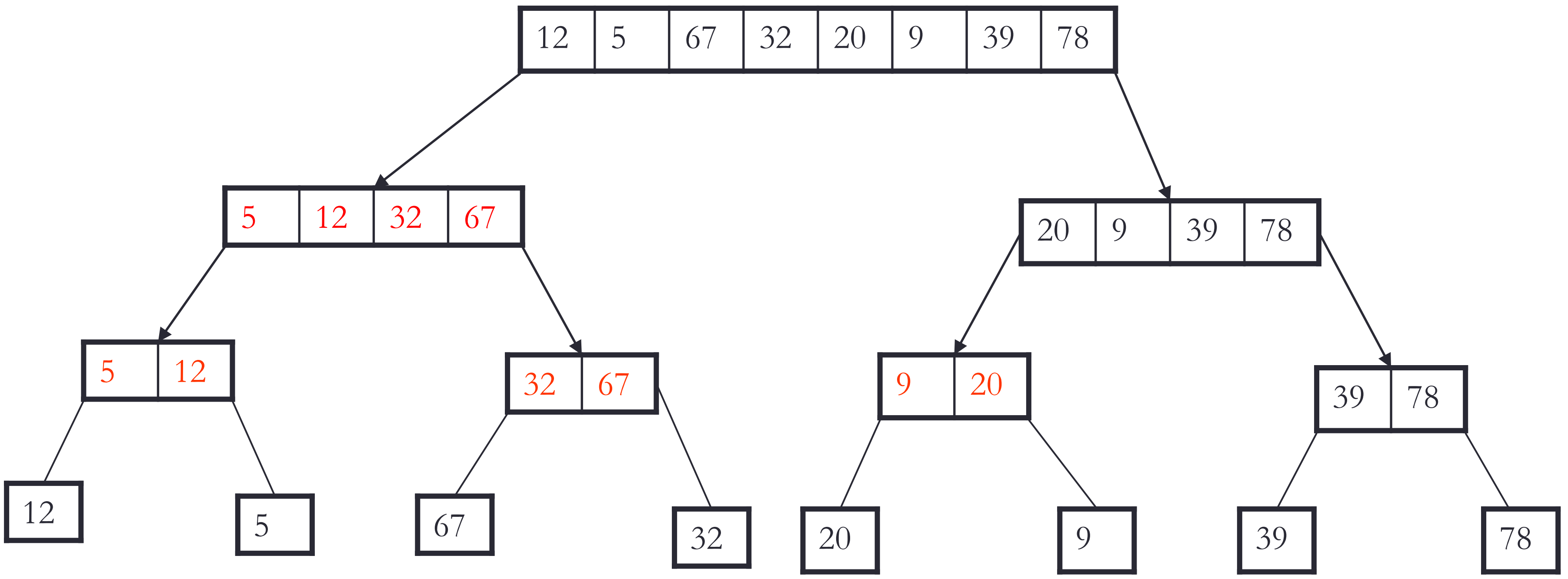
MergeSort: Conquer (Step 2)



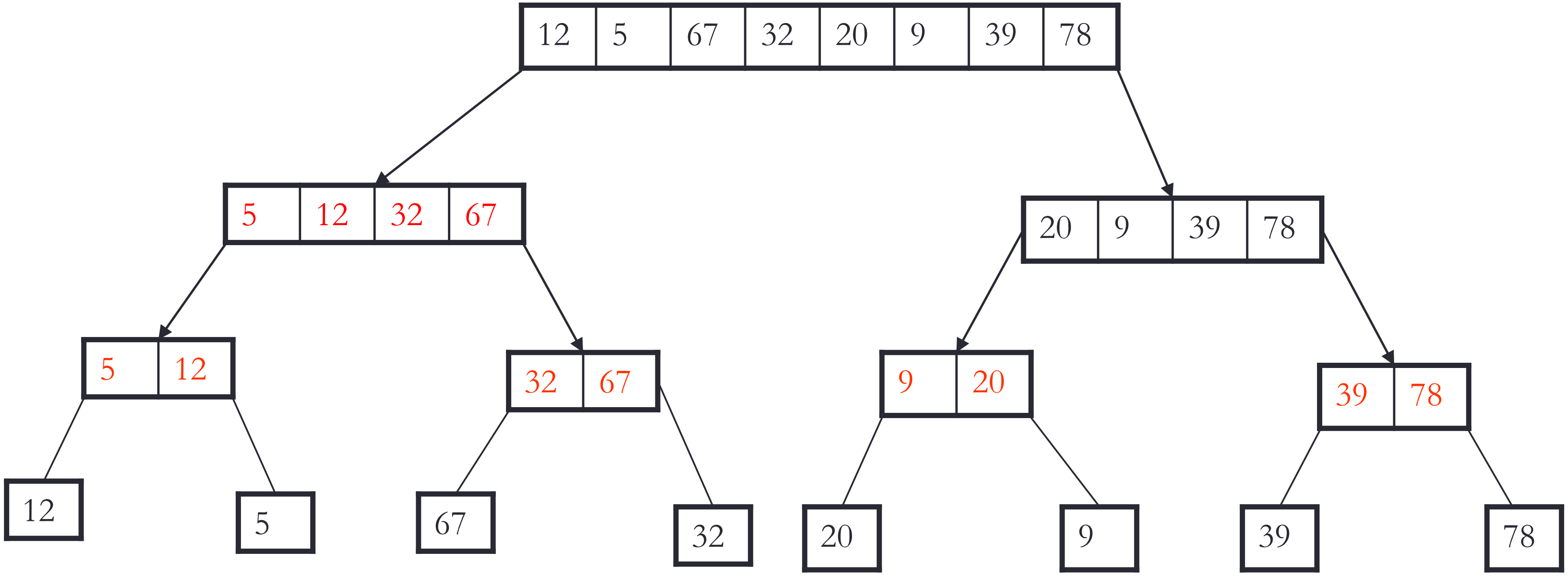
MergeSort: Conquer (Step 3)



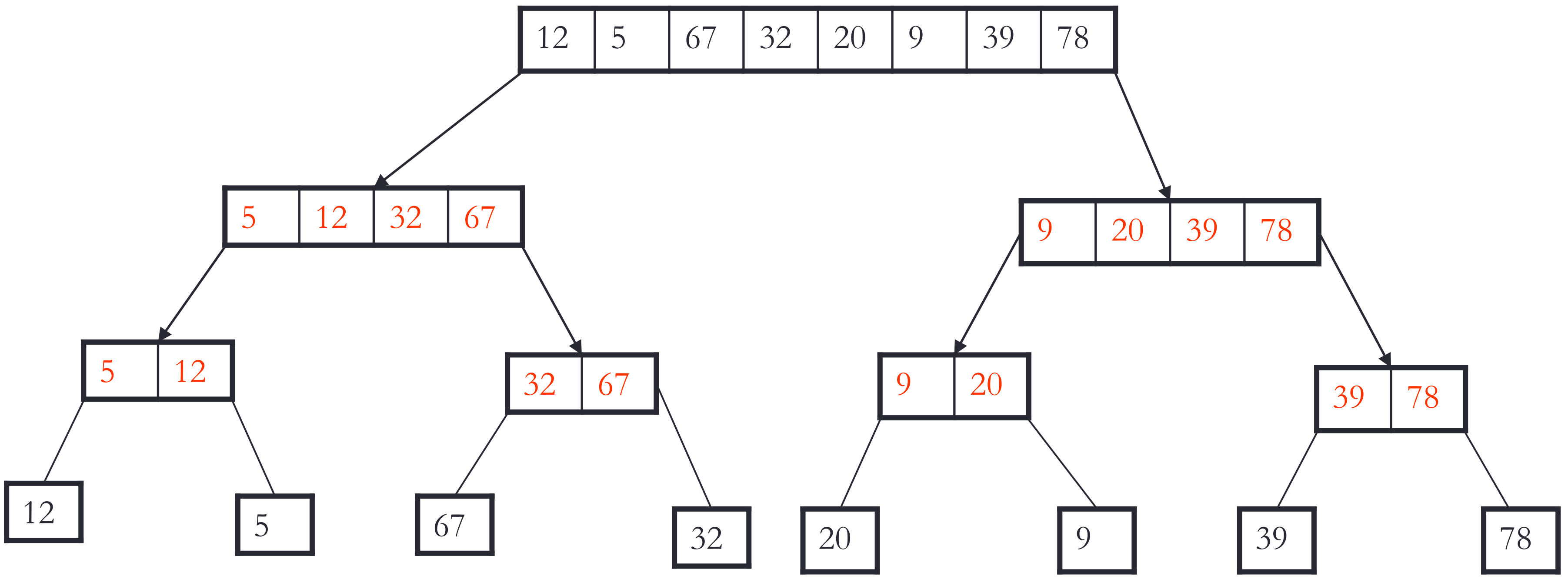
MergeSort: Conquer (Step 4)



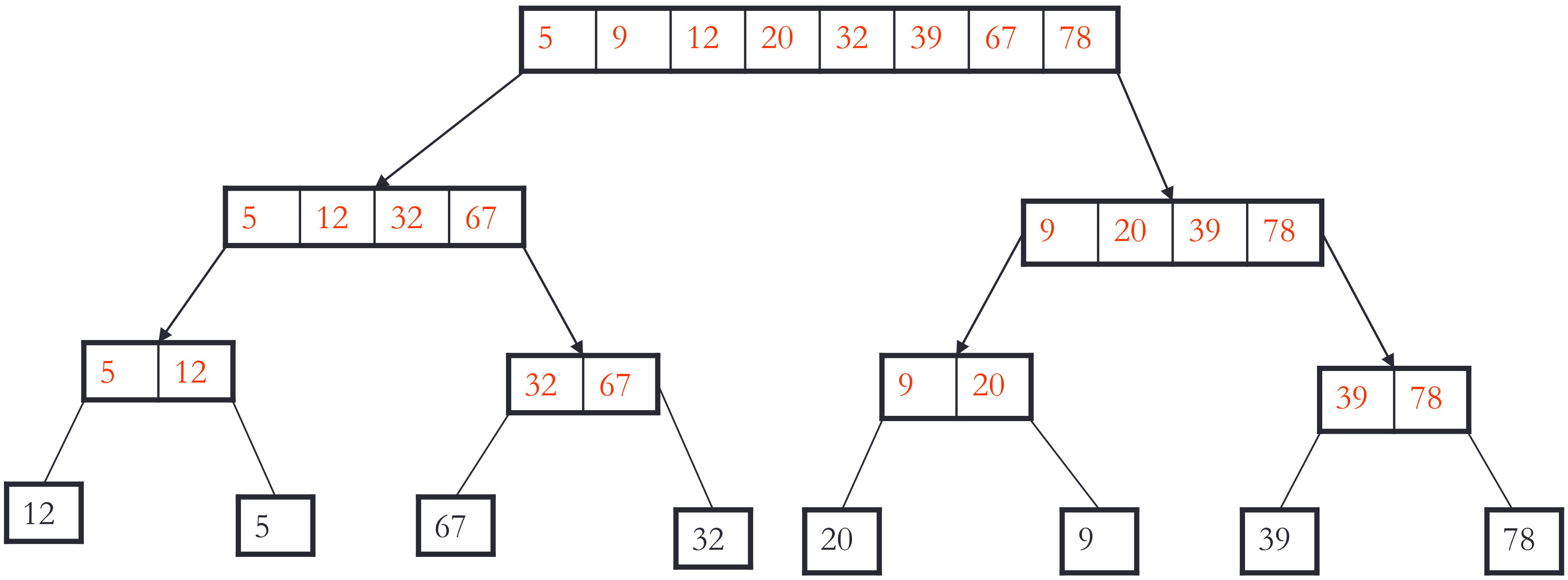
MergeSort: Conquer (Step 5)



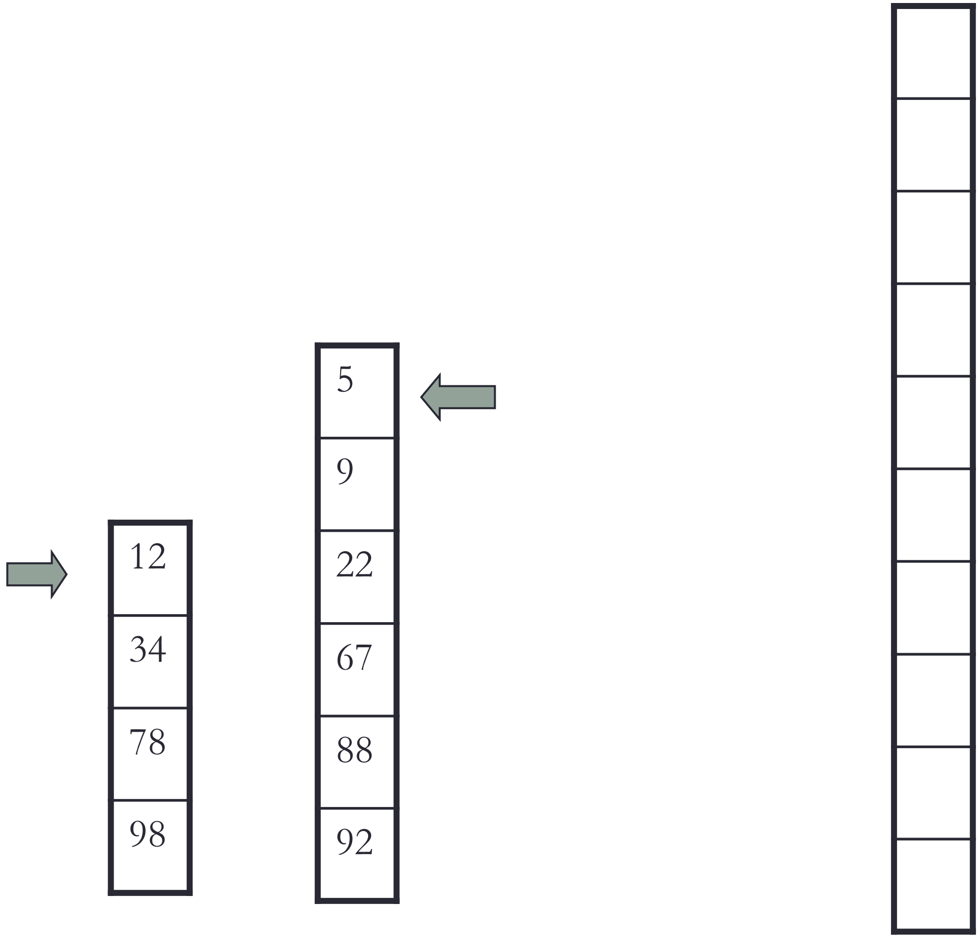
MergeSort: Conquer (Step 6)

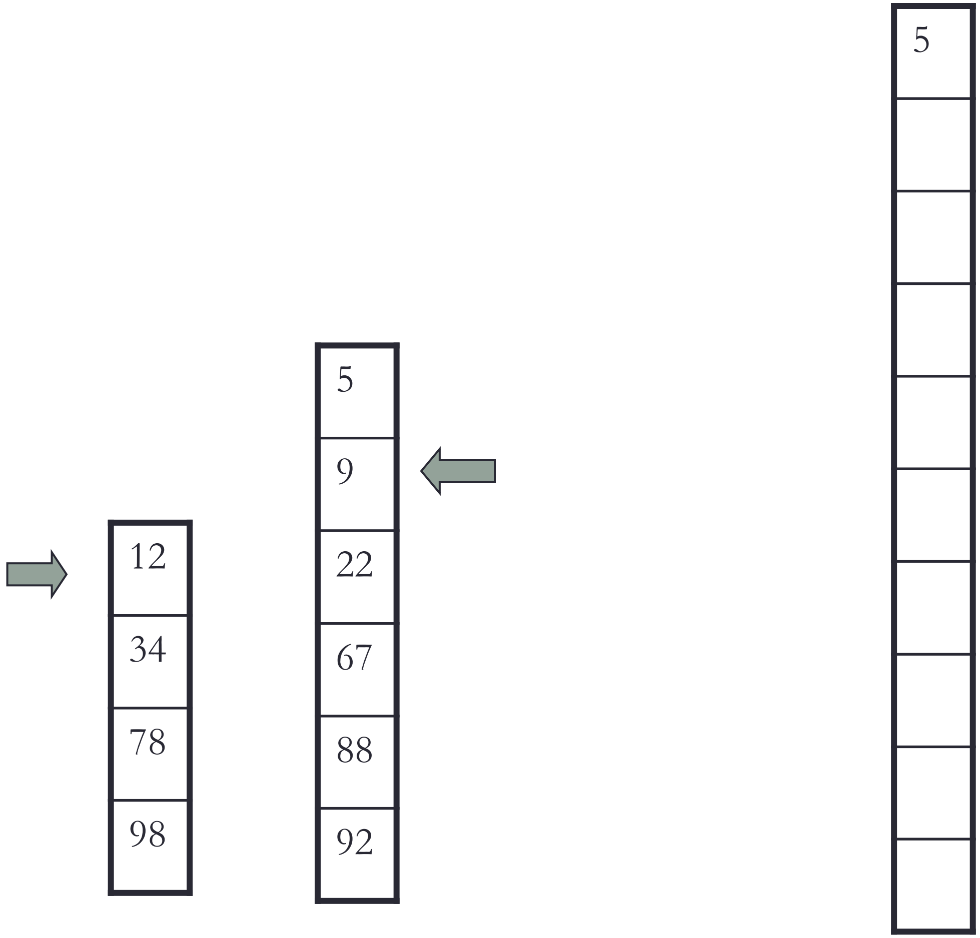


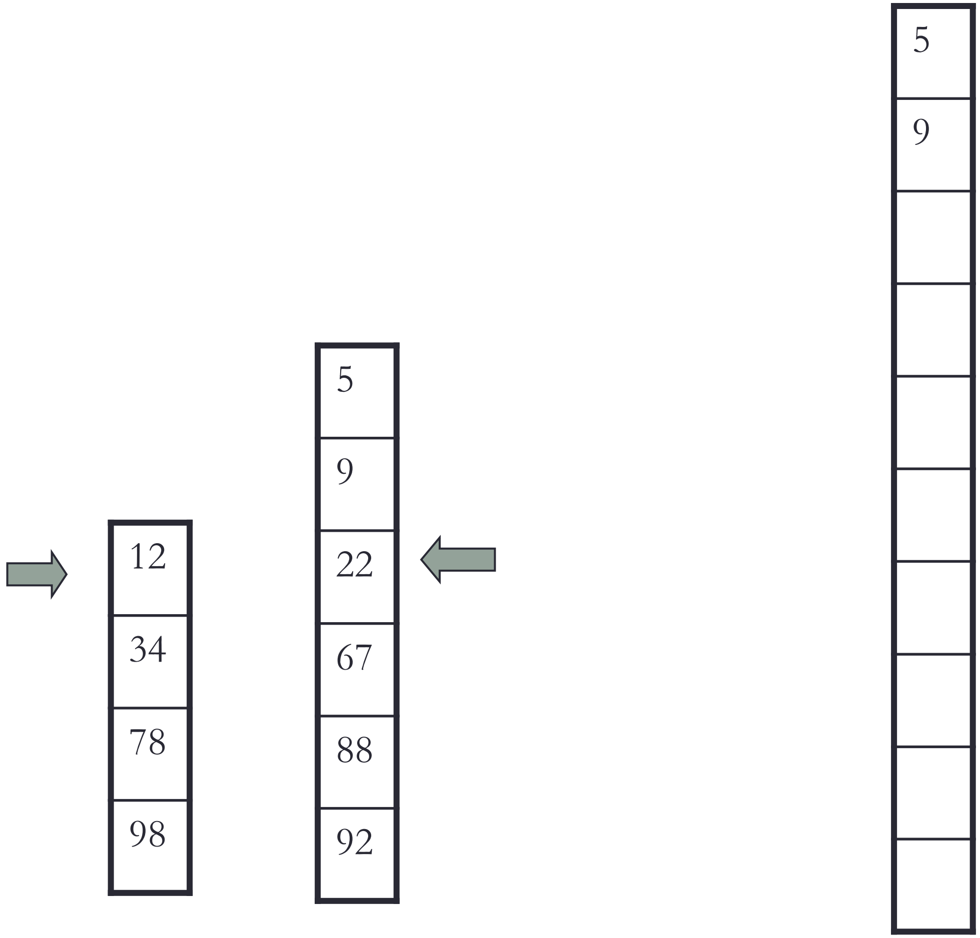
MergeSort: Conquer (Step 7)

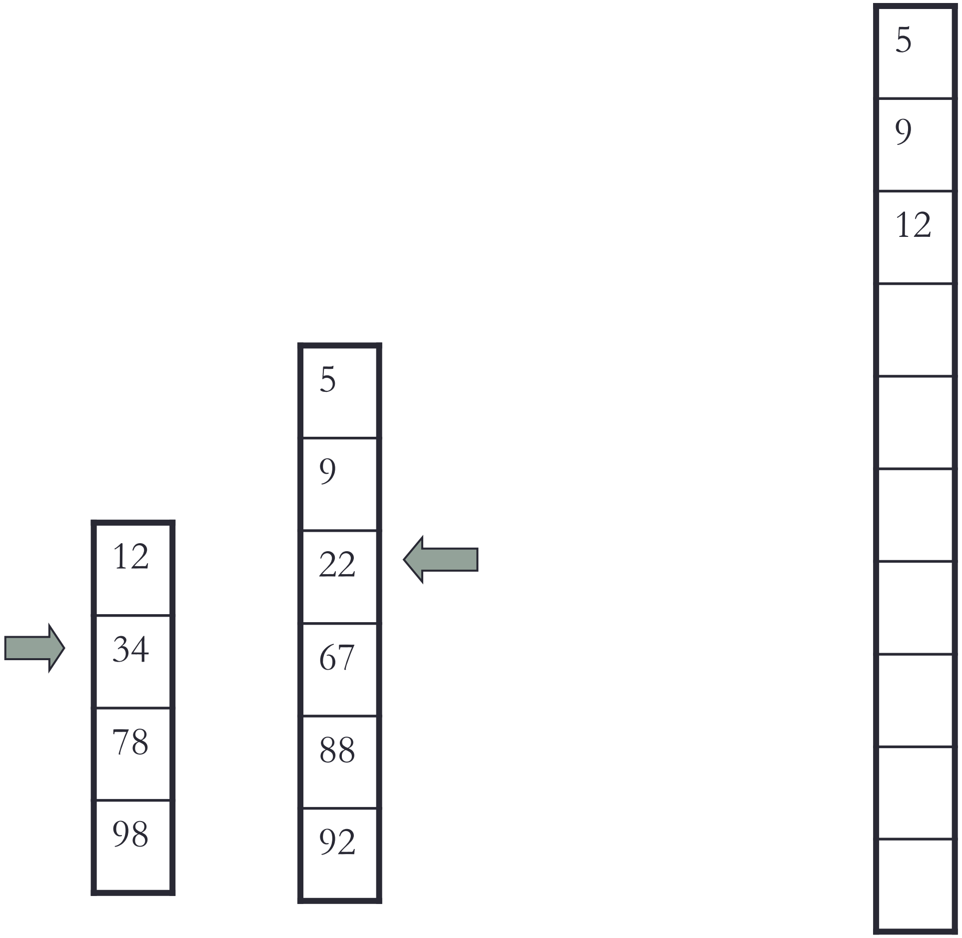


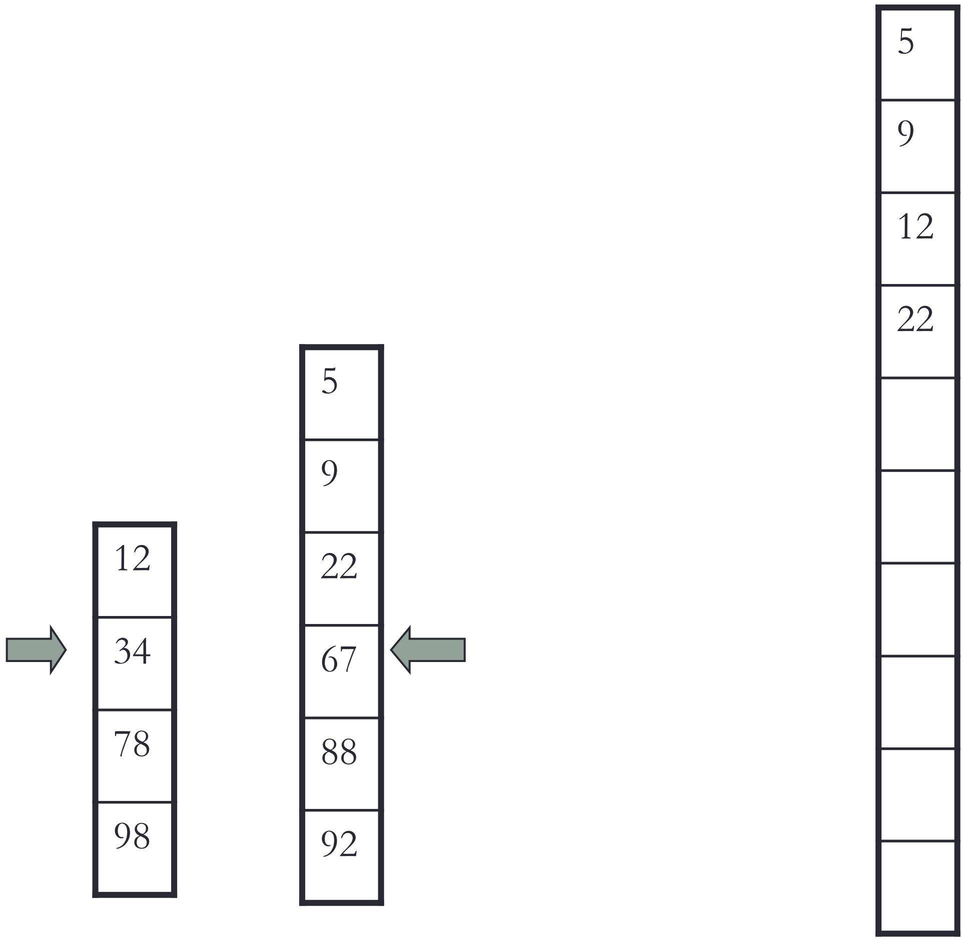
How to merge two sorted list?

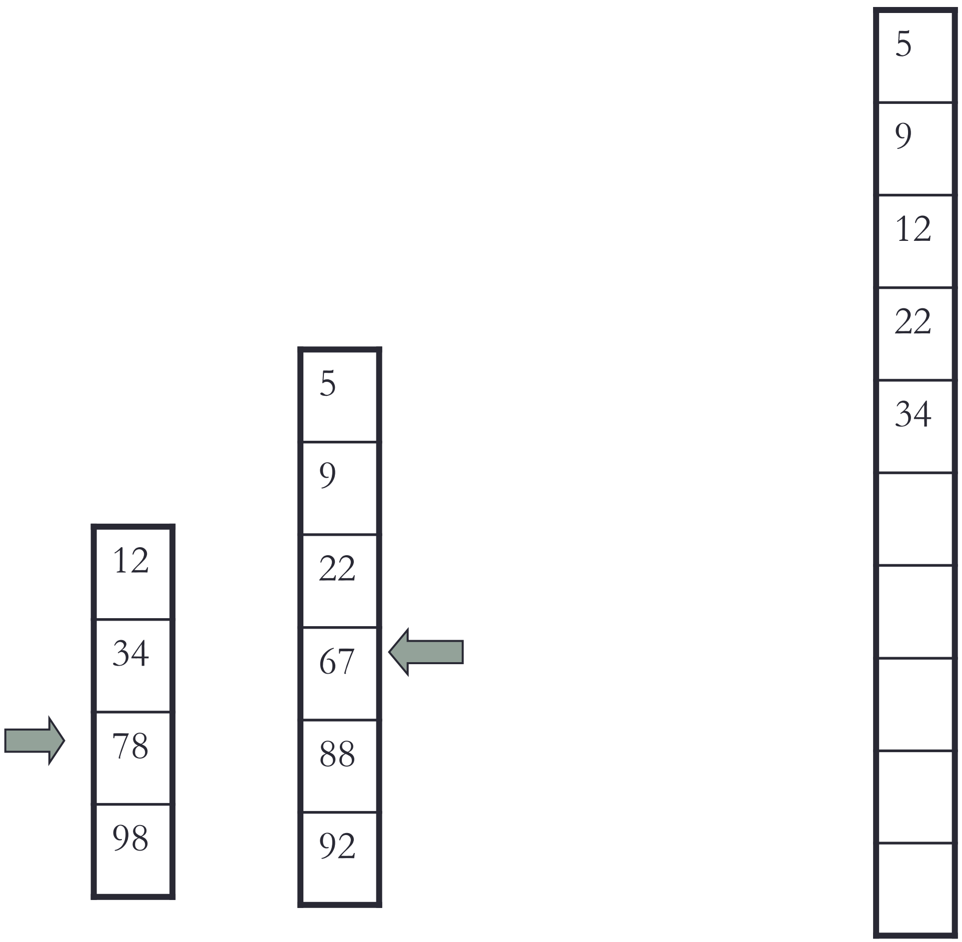


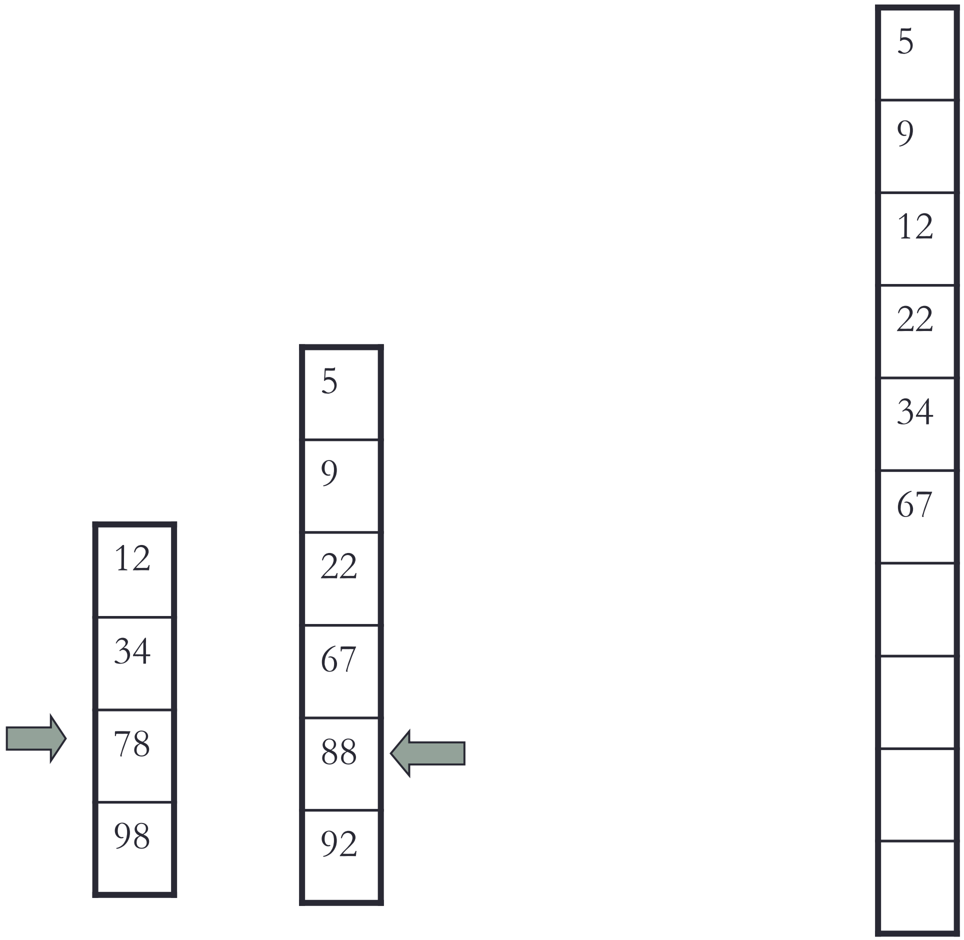

12
34
78
98
5
9
22
67
88
92
5

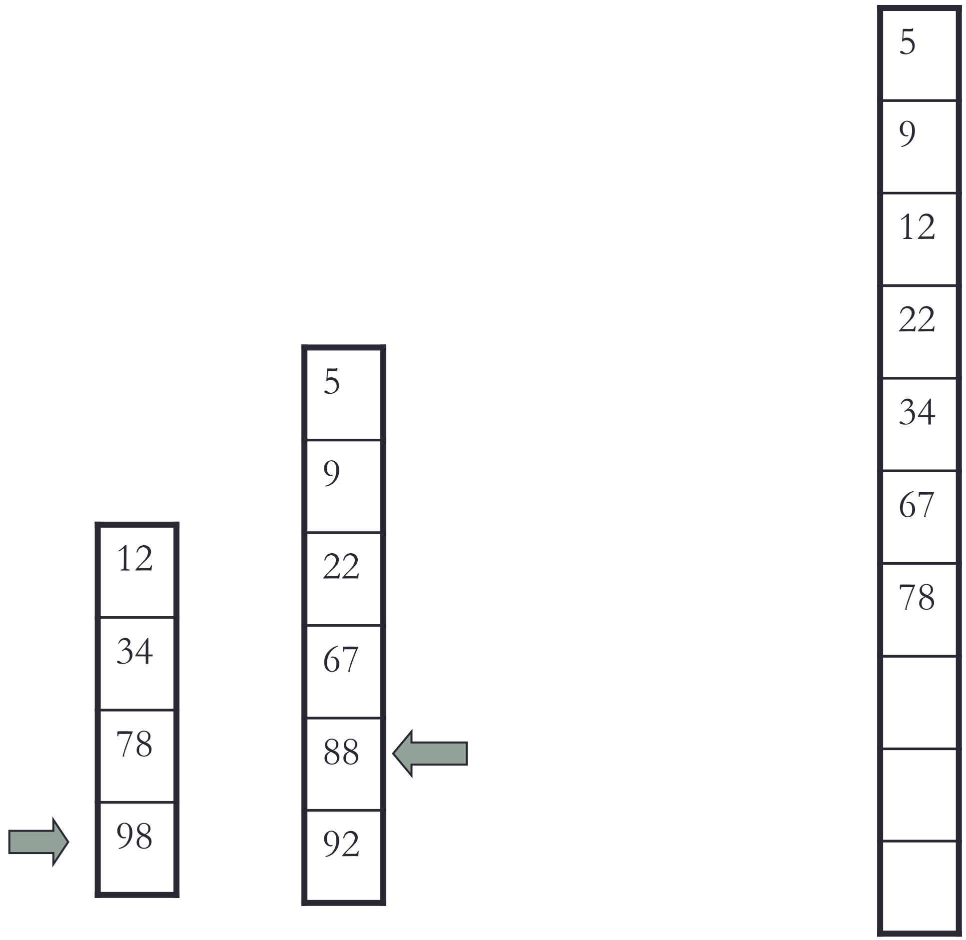

12
34
78
98
5
9
22
67
88
92
5
9

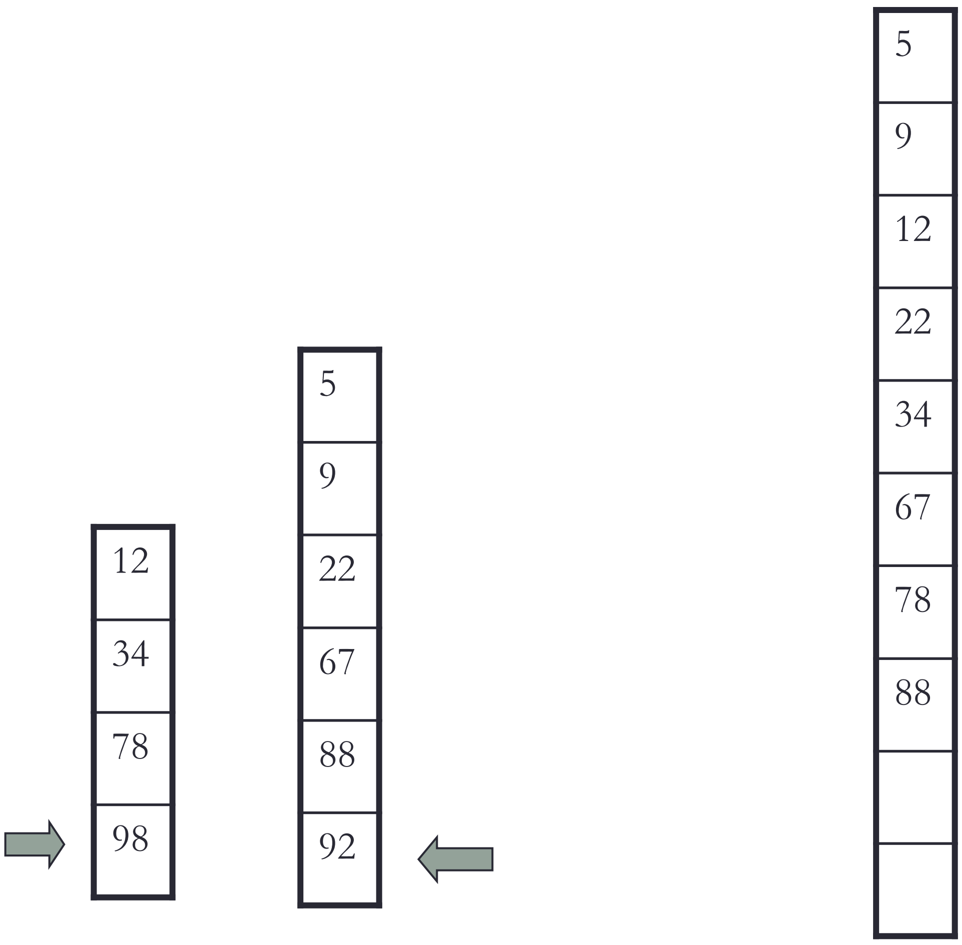

12
34
78
98
5
9
22
67
88
92
5
9
12

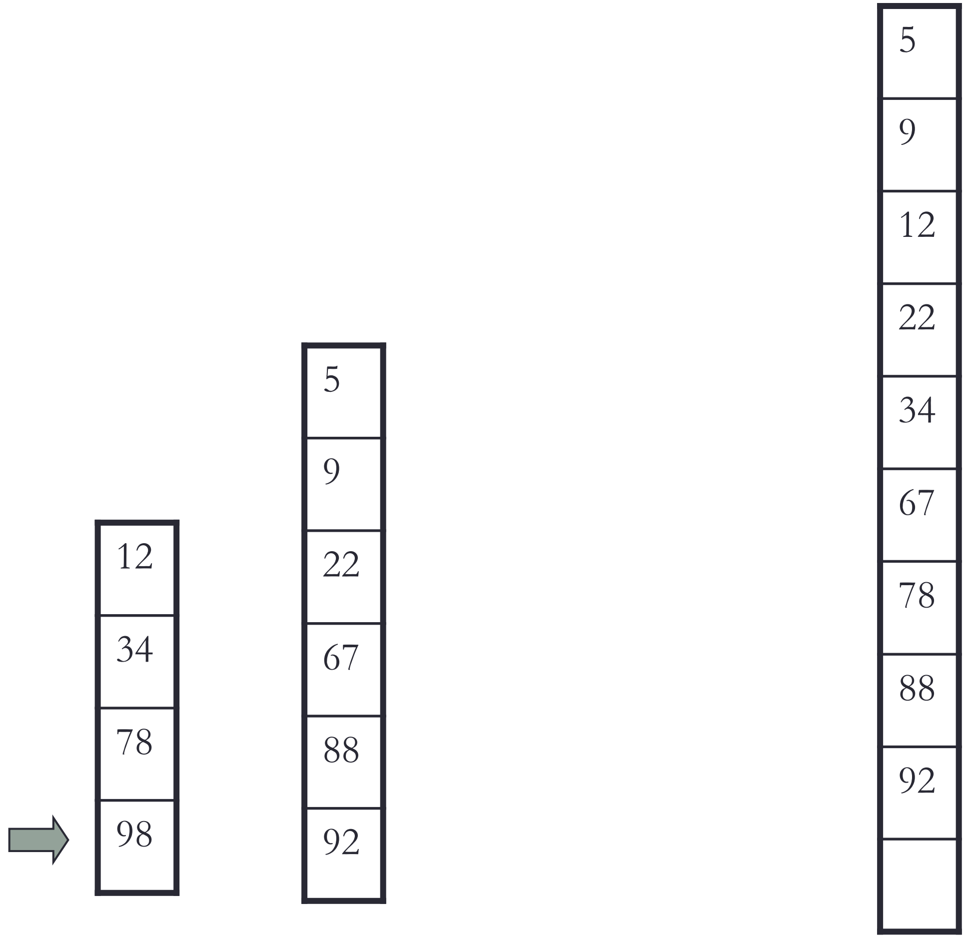

12
34
78
98
5
9
22
67
88
92
5
9
12
22

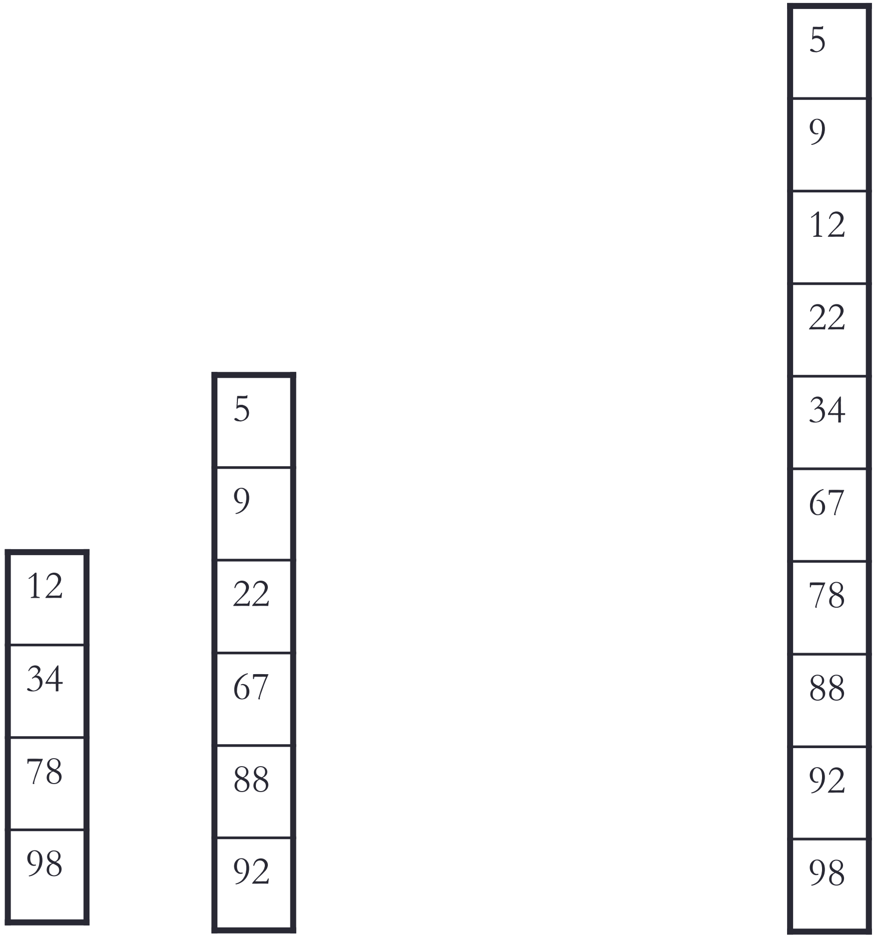

12
34
78
98
5
9
22
67
88
92
5
9
12
22
34


12
34
78
98
5
9
22
67
88
92
5
9
12
22
34
67


12
34
78
98
5
9
22
67
88
92
5
9
12
22
34
67
78


12
34
78
98
5
9
22
67
88
92
5
9
12
22
34
67
78
88


12
34
78
98
5
9
22
67
88
92
5
9
12
22
34
67
78
88
92



- Thus, merging just requires going through the two lists once.

MergeSort: Time complexity

- Let T(n) be the total number of steps `sort` made (in the worst case) to sort n numbers.
- How large is T(n)?

```
sort([a1, a2, …, an]):                                        ← T(n)
    if n = 1:                                                 } O(1)
        return [a1]
    else:
        S1 = sort([a1, a2, …, an/2])                          ← T(n/2)
        S2 = sort([an/2+1, an/2+2, …, an])                    ← T(n/2)
        merge S1, S2 into a single sorted list                } O(n)
```

- Therefore, $T(n) = 2\,T\left(\frac{n}{2}\right) + \mathbf{O(n)}$.

MergeSort: What is T(n)?

- Let $T(n) = 2\,T\left(\frac{n}{2}\right) + cn$, for some constant c.
- Let $n = 2^k$, so $k = \log_2 n$ is an integer.
- $T(n) = 2\,T\left(\frac{n}{2}\right) + cn$

$= 2\left(2\,T\left(\frac{n}{4}\right) + c \cdot \frac{\mathbf{n}}{\mathbf{2}}\right) + cn = 4\,T\left(\frac{n}{4}\right) + 2 \cdot cn$

$= 4\left(2\,T\left(\frac{n}{8}\right) + c \cdot \frac{\mathbf{n}}{\mathbf{4}}\right) + 2 \cdot cn = 8\,T\left(\frac{n}{8}\right) + 3 \cdot cn$

$= \ldots$

$= n\,T\left(\frac{n}{n}\right) + k \cdot cn$

$= n\,T(1) + \log_2 n \cdot cn$ [$T(1)$, c are constants.]

$= O(n + n \log n) = O(n \log n)$.

Example 3: Linear time selection

The problem

- Input: n distinct numbers, and an integer k (where $1 \le k \le n$).
- Output: The kth largest of these n numbers.

Example

- Input: 34, 8, 19, 73, 44, and an integer k=3
- Output: 34

- People used to believe that at least $\Omega(n \log n)$ comparisons are necessary to solve the problem.
- Blum, Floyd, Pratt, Rivest and Tarjan showed that the problem can be solved using only $O(n)$ comparisons.

What is the k-th largest of n numbers?

- For ease of discussion, we assume all numbers are distinct.
- Given a set N of n numbers.
- If x is the k-th largest number in N, then
  - there are exactly k numbers in N that are ≥ x;
  - there are exactly n-k numbers in N that are < x.

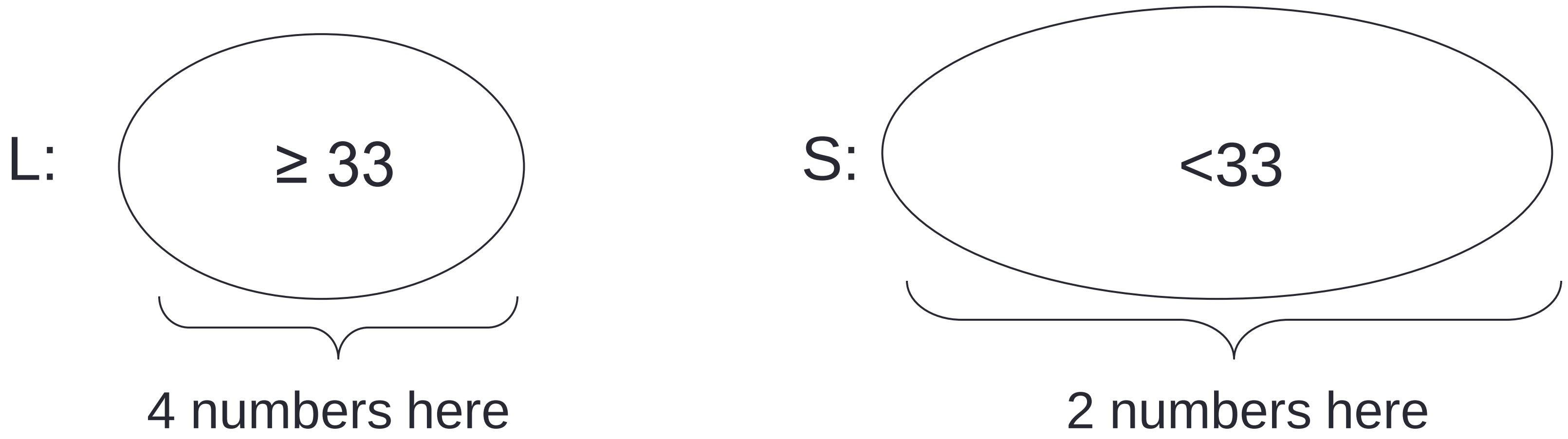
Example:

- N = {12, 33, 76, 29, 54, 62}.
- n = 6, k = 4.
- The number 33 is the 4-th largest in N ⇒
  - There are 4 numbers, namely 33, 76, 54, 62, that are ≥ 33, and
  - There are 6-4 = 2 numbers, namely 12, 29, that are < 33.

Basic operation: Divide N by the number a

> Divide N into the two sets
> L = {x | x ≥ a} and S = {x | x < a}.

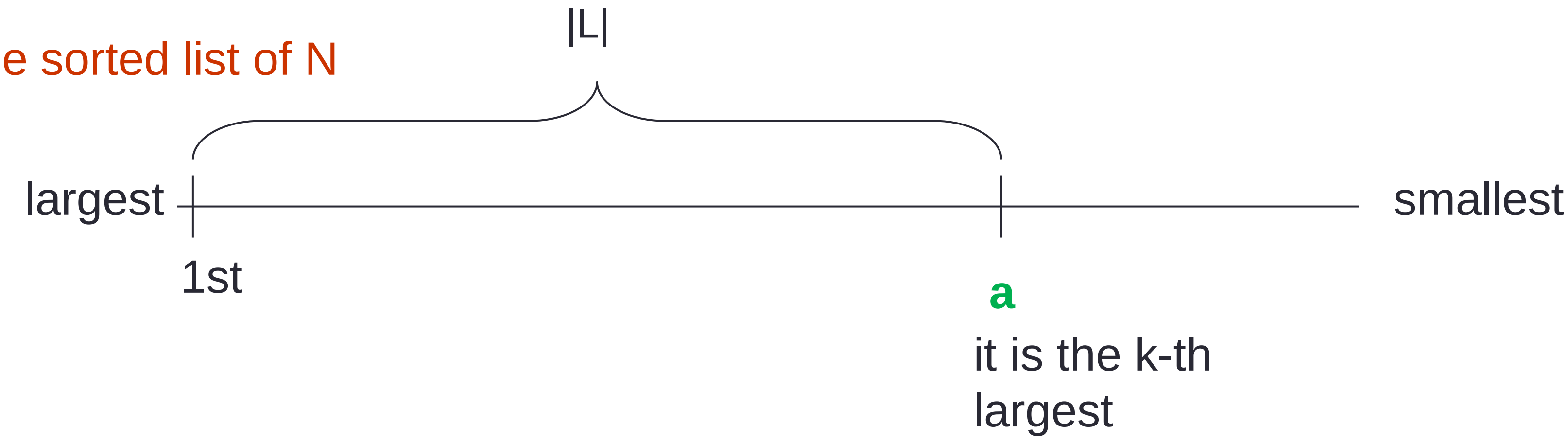
- **Example:** N = {12, 33, 76, 29, 54, 62}. Let a = 33. Then,



Basic Operation: Case 1

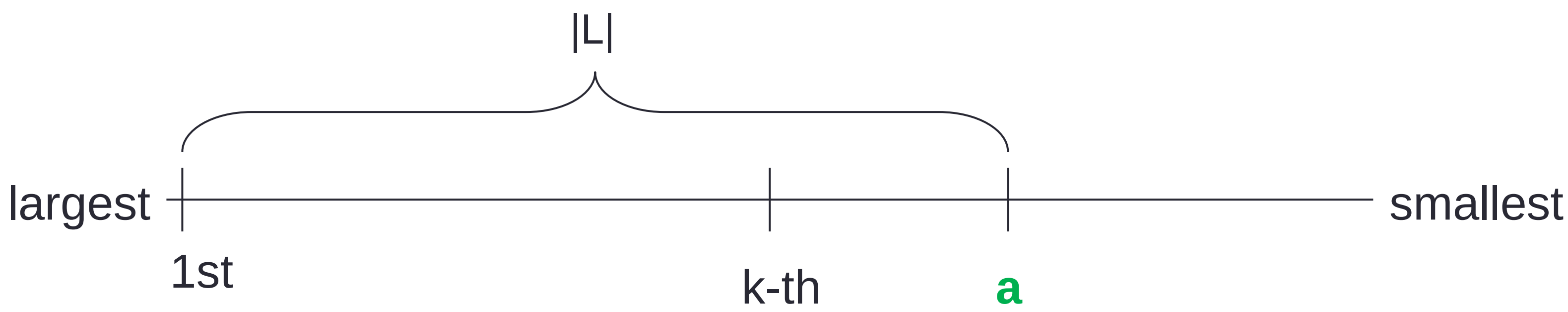
- Note that for an arbitrary number **a**, if we divide N by a to get L and S, we have three possible cases:
  - **Case 1:** |L| = k: **a** is the k-th largest; it is the solution.

The sorted list of N



Basic Operation: Case 2

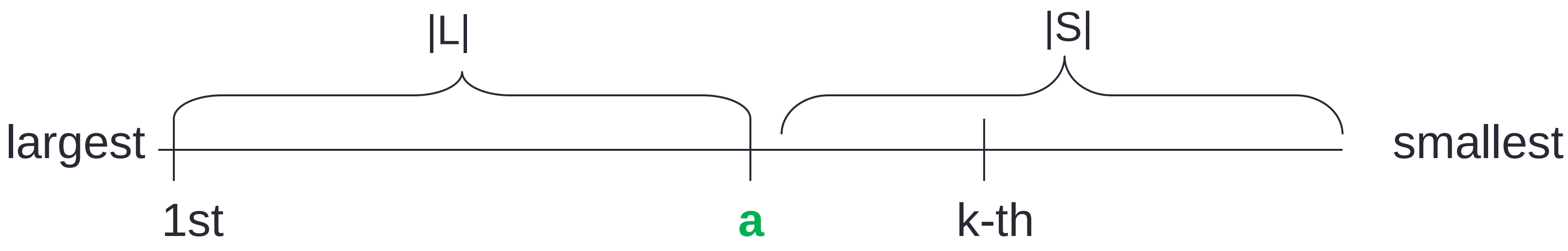
- **Case 1:** |L| = k: **a** is the k-th largest; it is the solution.
- **Case 2:** |L| > k:



- **Key observation:** the solution is the k-th largest in L-{**a**}.
- Thus, we recursively find the k-th largest in L-{**a**}.

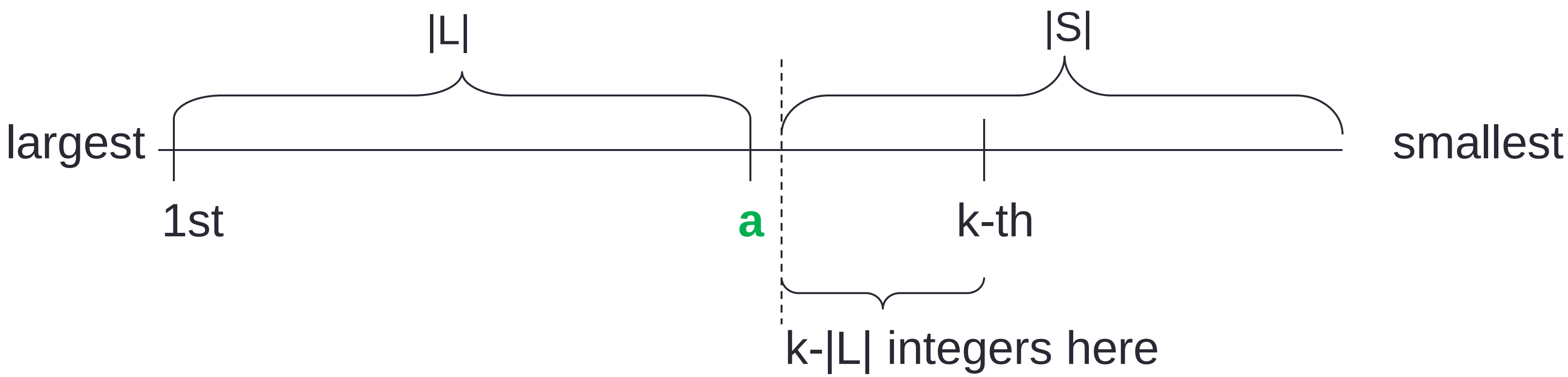
Basic Operation: Case 3

- **Case 1:** |L| = k: **a** is the k-th largest; it is the solution.
- **Case 2:** |L| > k: The solution is k-th largest in L.
- **Case 3: |L| < k:**



Basic Operation: Case 3

- **Case 1:** |L| = k: **a** is the k-th largest; it is the solution.
- **Case 2:** |L| > k: The solution is k-th largest in L.
- **Case 3: |L| < k:**



- **Key observation:** the solution is the k-|L| largest in S.
- Thus, we find recursively the k-|L| largest in S.

The algorithm

```
largest(N, k):
    Pick some number a in N arbitrarily.
    Divide N into
      L = {x |x ≥ a} and S = {x | x < a}.
    if |L| = k: return a
    if |L| > k: return largest(L-{a}, k)
    if |L| < k: return largest(S, k-|L|)
```

The algorithm: Time complexity

```
largest(N, k):                                          T(n)
    Pick some number a in N arbitrarily.                O(1)
    Divide N into
      L = {x |x ≥ a} and S = {x | x < a}.               O(n)
    if |L| = k: return a                                O(1)
    if |L| > k: return largest(L-{a}, k)                T(|L-{a}|)
                                                                     or
    if |L| < k: return largest(S, k-|L|)                T(|S|)
```

Time Complexity (worst case)

$T(n) = \max\{T(|L-\{a\}|), T(|S|)\} + O(n)$

$\leq T(n-1) + O(n)$ (because $|L-\{a\}|$ and $|S| \leq n-1$)

$\leq T(n-2) + O(n-1) + O(n)$

$\leq \ldots$

$\leq O(1 + 2 + \ldots + n) = O(n^2)$.

The algorithm: Why $O(n^2)$ time?

In the worst case,

- We may pick the smallest number **a** and waste $O(n)$ time for finding L and S.
- But we can only reduce the problem size by 1 (i.e., $T(n) \rightarrow T(n-1)$).

- Can we pick some **a** that guarantees |L| and |S| being at least some fraction of n?

A clever way to pick a

- Divide (arbitrary) the **n** numbers into **n/5** groups each with 5 numbers (except possibly the last group).
- For each group
  - sort the 5 numbers in descending order and then determine their median, i.e., the 3rd largest.
- The number **a** is just the median of “these n/5 medians”.

We can prove that such an **a** is always

- smaller than or equal to at least 3n/10 numbers, and
- larger than at least 3n/10 numbers.

A clever way to pick a: Example

n = 30, 3n/10 = 9

98	189	100	90	132	72
83	42	79	89	120	70
50	19	30	70	80	55
12	15	4	61	22	54
9	8	1	18	14	52

132	90	72		98	100	189
120	89	70		83	79	42
80	70	55	a	50	30	19
22	61	54		12	4	15
14	18	52		9	1	8

A clever way to pick a: Example

$\frac{n}{5} \times \frac{1}{2}$ $\frac{n}{5} \times \frac{1}{2}$

$3 \times \frac{n}{5} \times \frac{1}{2} = \frac{3n}{10}$

132		90		72			98		100		189
v		v		v							
120		89		70			83		79		42
v		v		v							
80	>	70	>	55	a	>	50	>	30	>	19
							v		v		v
22		61		54			12		4		15
							v		v		v
14		18		52			9		1		8

$3 \times \frac{n}{5} \times \frac{1}{2} = \frac{3n}{10}$

Linear Time Selection

The algorithm:

- Group the **n** numbers into groups of 5 numbers, and for each group, sort the 5 numbers in ascending order and find the median;
- Find (recursively) the median, **a**, of these **n/5** medians;
- Using this **a** to partition the **n** numbers into two groups L and S such that all numbers in L are greater than or equal to **a**, and all numbers in S are smaller than **a**.
- **If |L| ≥ k**, find the k-th largest in L;
- **Otherwise**, find the (k-|L|)-st largest in S.

7n/5

T(n/5)

n

T(|L|) or
T(|S|)

Time complexity:

$$T(n) = T(|L|) + T\left(\frac{n}{5}\right) + \frac{12n}{5} \quad \text{or} \quad T(|S|) + T\left(\frac{n}{5}\right) + \frac{12n}{5}$$

Linear Time Selection: Time Complexity

Note that

- |L| ≤ 7n/10; otherwise, there are fewer than 3n/10 numbers smaller than a;
- |S| ≤ 7n/10; otherwise, there are fewer than 3n/10 numbers larger than or equal to a.

Thus, we have

- $T(n) \leq T\left(\frac{7n}{10}\right) + T\left(\frac{n}{5}\right) + \frac{12n}{5}$.

By mathematical induction, we can prove that

- $T(n) \leq 24n$.