IDMA 2025: WEEK 1

SEARCHING, SORTING & ASYMPTOTICS

SRIKANTH SRINIVASAN

Algorithms & Complexity Section

DIKU

sysr@di.ku.dk

<u>Searching & Sorting</u>

→ Data indexed by numbers / names.
- ↳ Items in a warehouse indexed by I.D.
- ↳ Books in a library indexed by title.

→ Want to:
- ↳ <u>Search</u> & find item in <u>sorted</u> data.
- ↳ <u>Sort</u> data for efficient search.

Random Access Machine (RAM) model

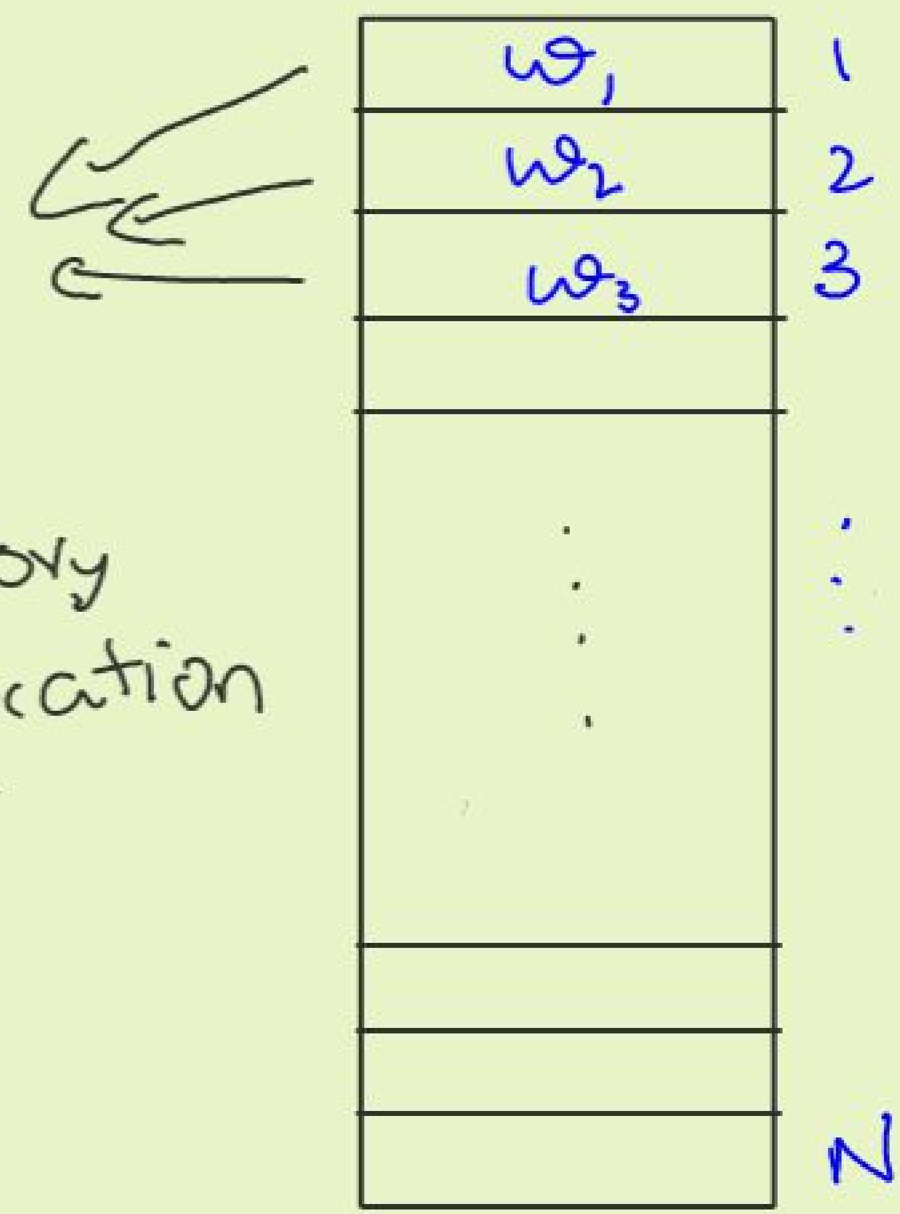
→ Memory partitioned into (eg. 01001100) "words"

→ Unit operations:

  → read contents of memory location

  → write —— " ——

  → perform operation



Searching

Input: An array A of sorted numbers

$$A[i] \leq A[2] \leq \cdots$$

& a number x

Output: An i such that $A[i] = x$.
(if it exists)

Eg. $A =$

1									10
1	4	5	7	9	10	11	14	16	20

Simple algorithm

Go through array elements one-by-one until x found.

```
i := -1
j := 1
while (i < 0 and j ≤ n)
        if (A[j] == x)
                i := j

        j = j+1
return i
```

$x = 9$:

↓	↓	↓	↓	↓					
1	4	5	7	9	10	11	14	16	20

Running time:

\# steps executed

Depends on:

→ Array length n

→ Input numbers

Simple algorithm

Go through array elements one-by-one until x found.

```
i := -1
j := 1
while (i < 0 and j ≤ n)
        if (A[j] == x)
                i := j
        j := j+1
return i
```

$x = 9$:

1	4	5	7	9	10	11	14	16	20

Worst-case Running time:

max # steps on array of length n

$= 2 + 9n + 1$

$= 3 + 9n = \Theta(n)$

3, 9 - "constants"

Θ notation (Informal)

Running time is $\Theta(f(n))$ means that the worst-case running time is $f(n)$ "up to constant factors".

Why ignore constants?

→ Usually are not too big.

→ For large input sizes n, $f(n)$ determines the running time.

Linear Search

Go through array elements one-by-one until x or $y > x$ found.

```
i := -1
j := 1
while (i<0 & j≤n & A[j]≤x)
        if (A[j] == x)
                i := j
        j := j+1
return i
```

$x = 8$:

$\downarrow\ \downarrow\ \downarrow\ \downarrow\ \downarrow$

1	4	5	7	9	10	11	14	16	20

Worst-case Running Time:

$$10n + 4 = \Theta(n)$$

Binary Search

1									10
1	4	5	7	9	10	11	14	16	20

→ Start at mid point: index $\left\lfloor \frac{1+10}{2} \right\rfloor = 5$

→ If middle element is x, we found x & we can stop.

→ If middle element less than x, then x cannot be in left half.

→ If — " — greater than x, then
———— " —— right half.

→ Repeat on the correct half!

Binary Search

1	2	3	4	5	6	7	8	9	10
1	4	5	7	9	10	11	14	16	20

Example:

$x = 10$

$mid_1 = \lfloor \frac{1+10}{2} \rfloor = 5$, $A[5] = 9$

$mid_2 = \lfloor \frac{6+10}{2} \rfloor = 8$, $A[8] = 14$

$mid_3 = \lfloor \frac{6+7}{2} \rfloor = 6$, $A[6] = 10$

✓

$x = 6$

$mid_1 = 5$, $A[5] = 9$

$mid_2 = 2$, $A[2] = 4$

$mid_3 = 3$, $A[3] = 5$

$mid_4 = 4$, $A[4] = 7$

✗

Pseudo code for Binary Search

```
Binary-Search(A, x, lo, hi)          → Start with lo = 1
                                                  hi = n
    if (lo > hi)
        return -1                       Correctness?

                                        Running time?
    else
      mid = ⌊(lo + hi)/2⌋
      if (A[mid] == x)
          return mid                  → "Recursion"
      else if (A[mid] > x)
          return Binary-Search(A, x, lo, mid-1)
      else return Binary-Search(A, x, mid+1, hi)
```

Correctness via Invariants

Invariant: Statement that is

→ True before execution (Initialization)

→ Maintained during execution (Maintenance)

→ Shows that output is correct at the end of execution. (Termination)

Pseudo code for Binary Search

```
Binary-Search(A, x, lo, hi)
  if (lo > hi)
      return -1
  else
    mid = ⌊(lo + hi)/2⌋
    if (A[mid] == x)
        return mid
    else if (A[mid] > x)
        return Binary-Search(A, x, lo, mid-1)
    else return Binary-Search(A, x, mid+1, hi)
```

Invariant:

"If x is in A, then it lies between lo & hi"

→ Initially, $lo = 1$, $hi = n$

→ Maintenance: Sorted-ness.

→ Termination: Must end when $lo = hi = mid$

Pseudocode for Binary Search

```
Binary-Search(A, x, lo, hi)
    if (lo > hi)
        return -1
    else
      mid = ⌊(lo + hi)/2⌋
      if (A[mid] == x)
          return mid
      else if (A[mid] > x)
          return Binary-Search(A, x, lo, mid-1)
      else return Binary-Search(A, x, mid+1, hi)
```

Running time $\Theta(\log n)$

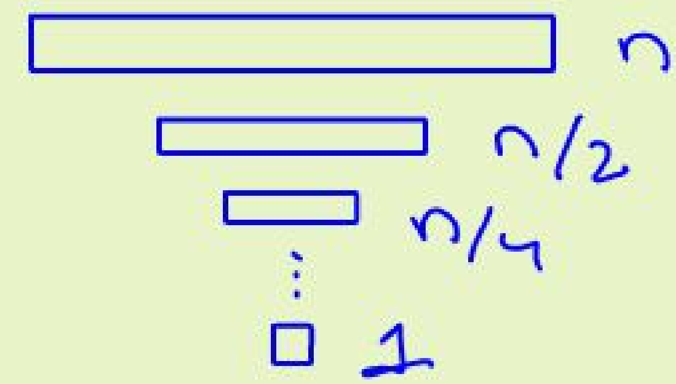
$\rightarrow R$ calls

$n \geq 2^R \Rightarrow R \leq \log_2 n$

Pseudocode for Binary Search

```
Binary-Search(A, x, lo, hi)
   if (lo > hi)
        return -1
   else
      mid = ⌊(lo + hi)/2⌋
      if (A[mid] == x)
            return mid
      else if (A[mid] > x)
            return Binary-Search(A, x, lo, mid-1)
      else return Binary-Search(A, x, mid+1, hi)
```

Running time $\Theta(\log n)$



→ Running time =

$\Theta(R) = \Theta(\log n)$

Linear vs. Binary Search

n	$\log_2 n$
1.000	≈ 10
1.000.000	≈ 20
1.000.000.000	≈ 30

Sorting

Input: Array A of numbers.

Output: The numbers in sorted order.

Two algorithms:

- → Insertion Sort $\Theta(n^2)$
- → Merge Sort $\Theta(n \log n)$

Insertion Sort

Idea: Sort array from left to right

16	11	9	1	4	14	5	7	20	10

11	16	9	1	4	14	5	7	20	10

9	11	16	1	4	14	5	7	20	10

1	9	11	16	4	14	5	7	20	10

⋮

1	4	5	7	9	10	11	14	16	20

Pseudocode

```
Insertion-Sort(A)
 for(i := 2 to n)
    j := i
    while(j>1 & A[j-1]>A[j])
        tmp := A[j-1]
        A[j-1] := A[j]
        A[j] := tmp
        j := j-1
```

Correctness invariant

At beginning of for loop

A[1,..,i-1] is sorted.

→ Initialization

→ Maintenance

→ Termination

Pseudocode

```
Insertion-Sort(A)
  for(i := 2 to n)
    j := i
    while(j>1 & A[j-1]>A[j])
      tmp := A[j-1]
      A[j-1] := A[j]
      A[j] := tmp
      j := j-1
```

Running time

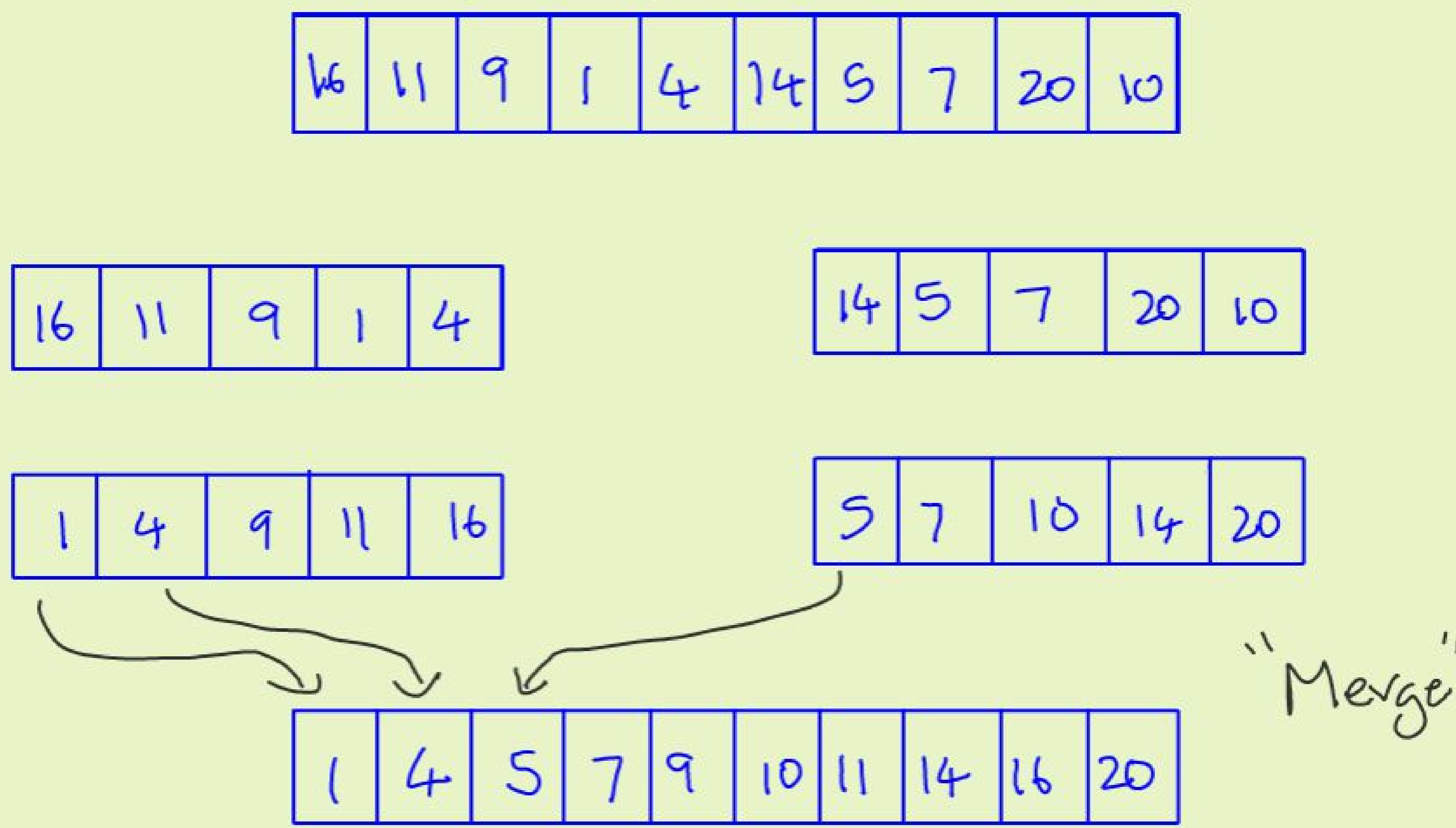
$$\le \underbrace{c}_{i=2} + \underbrace{2c}_{i=3} + 3c + \cdots + \underbrace{(n-1)c}_{i=n}$$

$$= c \cdot (1+2+\cdots+(n-1))$$

$$= \Theta(n^2)$$

Merge Sort

Divide-and-conquer paradigm



Merge Algorithm

```
Merge (L, R, M)
    n = length(L)
    m = length(R)
    Check that length(M) = n+m
    L[n+1], R[m+1] := ∞
    i, j := 1
    for (k = 1 to n+m)
        if (L[i] ≤ R[j])
            M[k] := L[i]
            i := i+1
        else M[k] := R[j]
            j := j+1
```

Correctness:

Suitable invariant (see CLRS)

Running time:

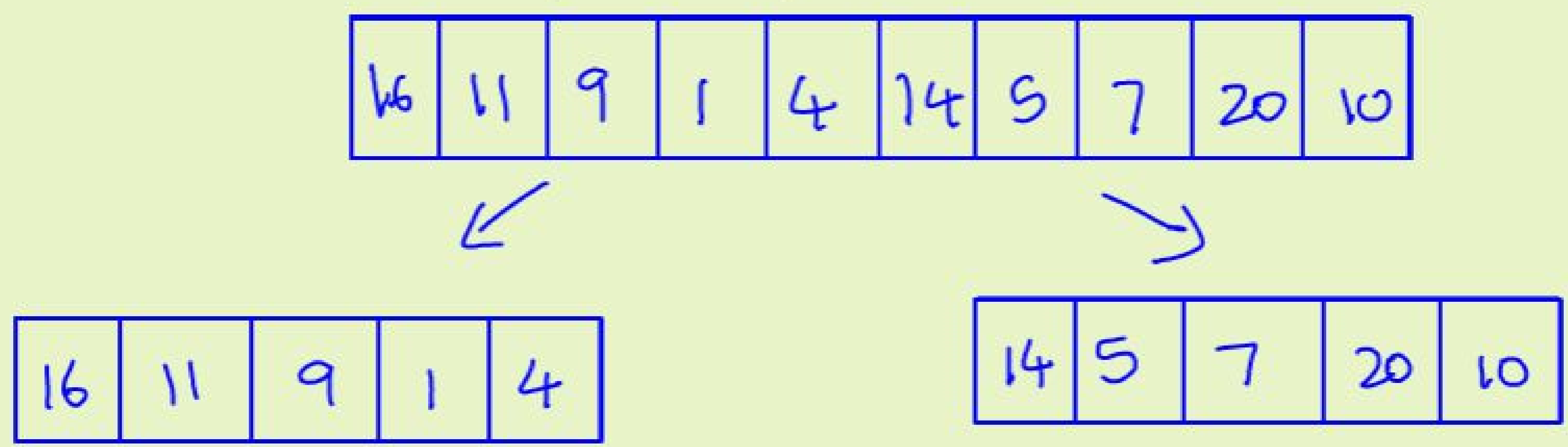
$\Theta(n+m)$

Merge Sort

```
MergeSort(A)
  if (length(A) > 1)
      mid = ⌊(1+length(A))/2⌋
      L := new array of length mid
      R := new array of length length(A)-mid
      for (i = 1 to mid)
          L[i] := A[i]
      for (i = 1 to length(A)-mid)
          R[i] := A[mid+i]
      Mergesort(L)
      Mergesort(R)
      Merge(L,R,A)
```

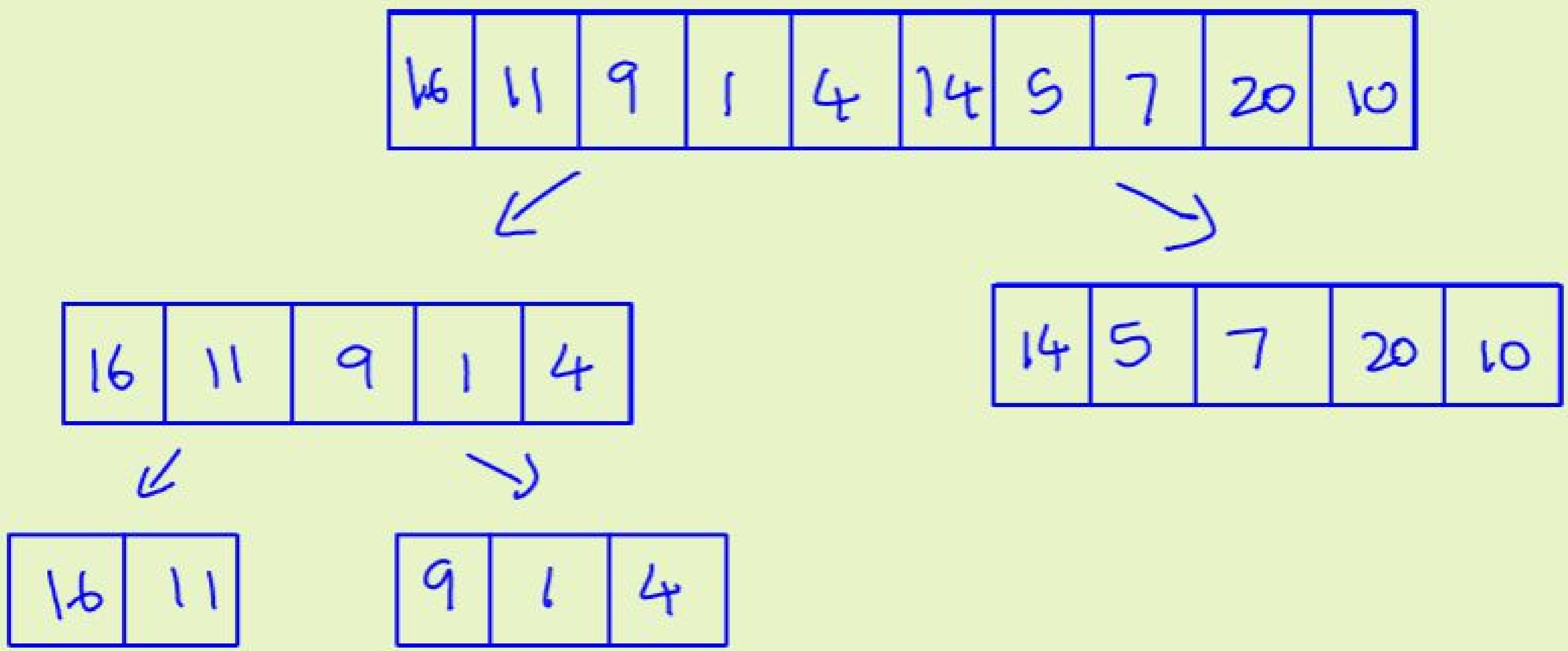
Merge Sort

Divide-and-conquer paradigm



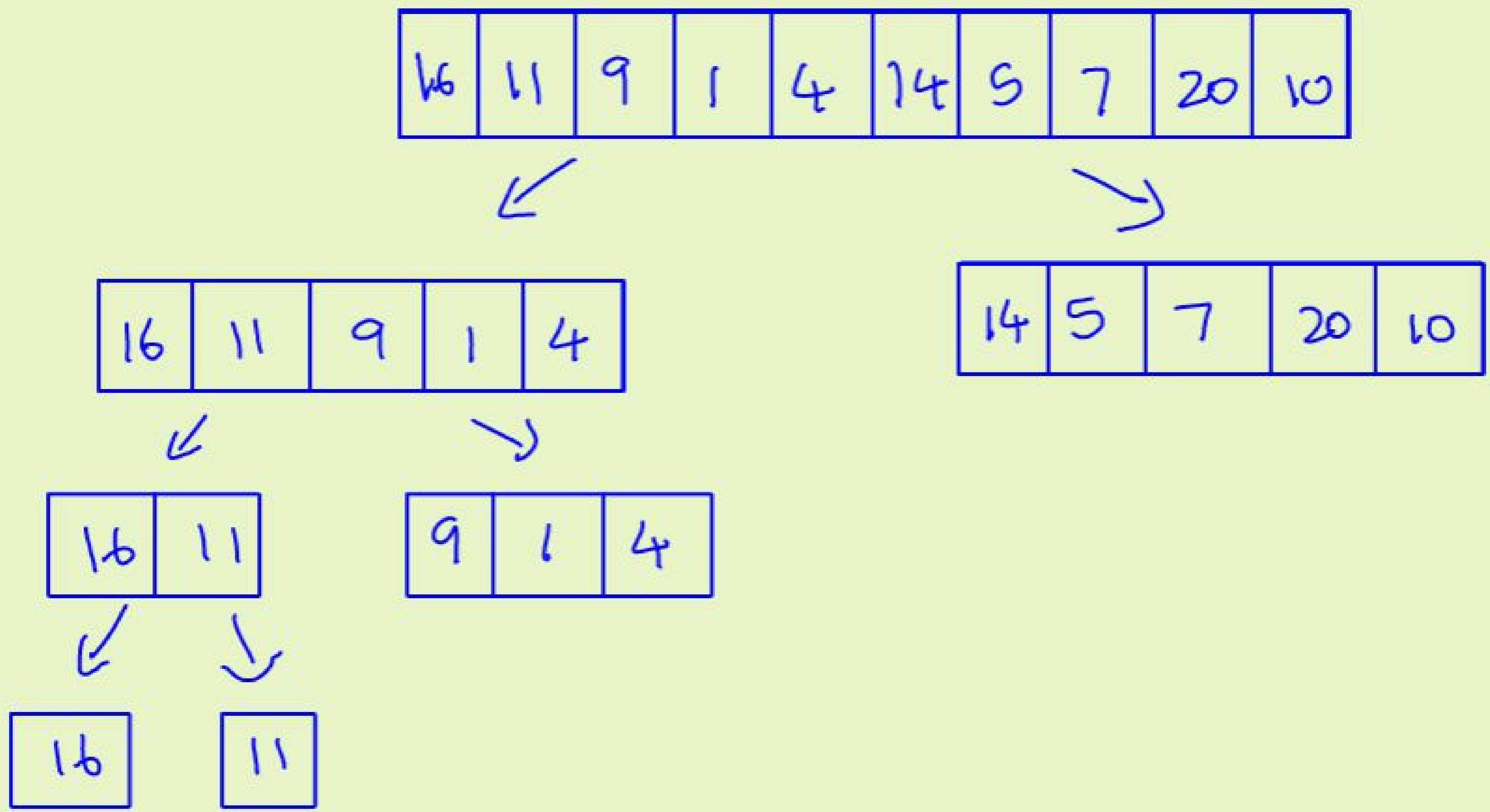
Merge Sort

Divide-and-conquer paradigm



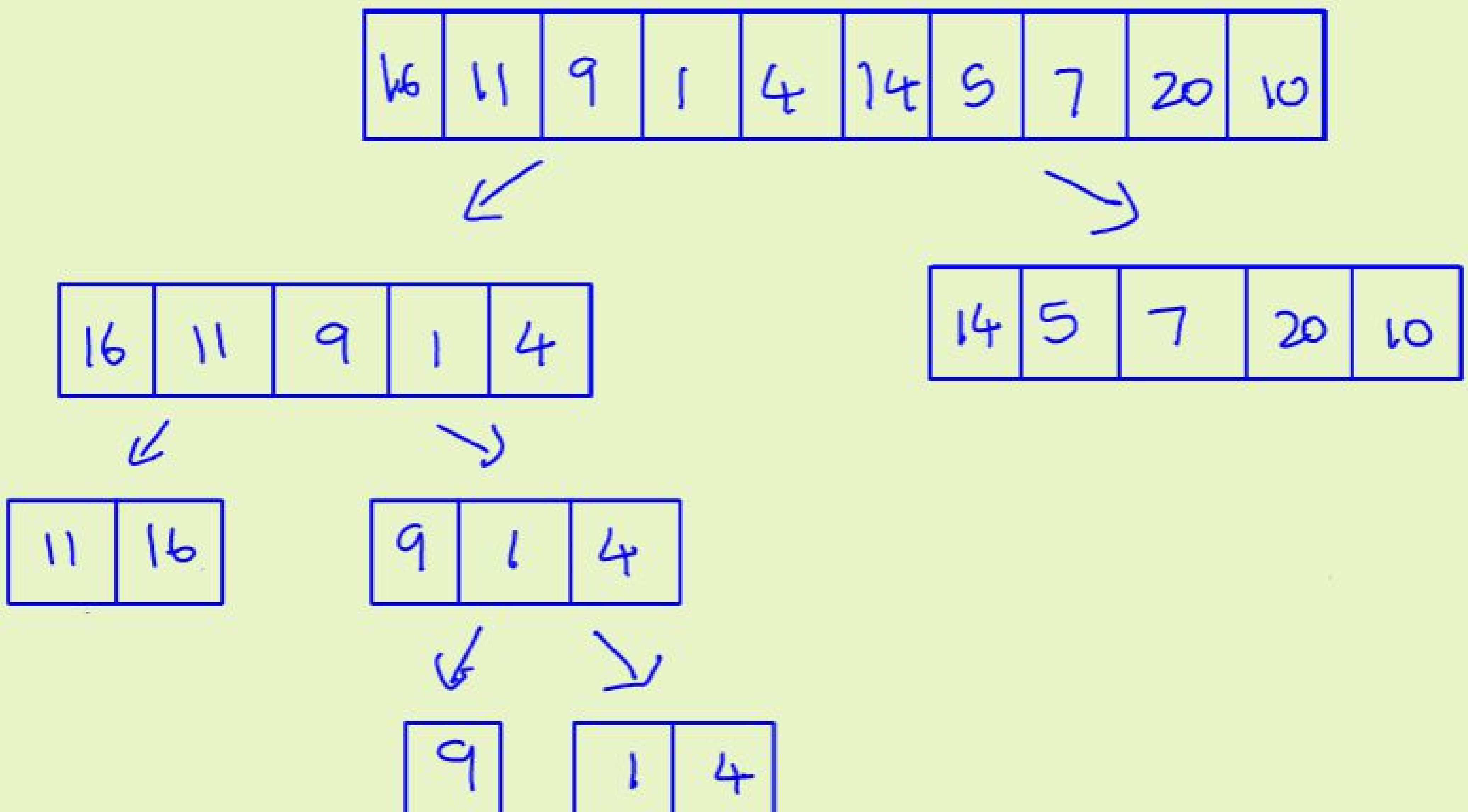
Merge Sort

Divide-and-conquer paradigm



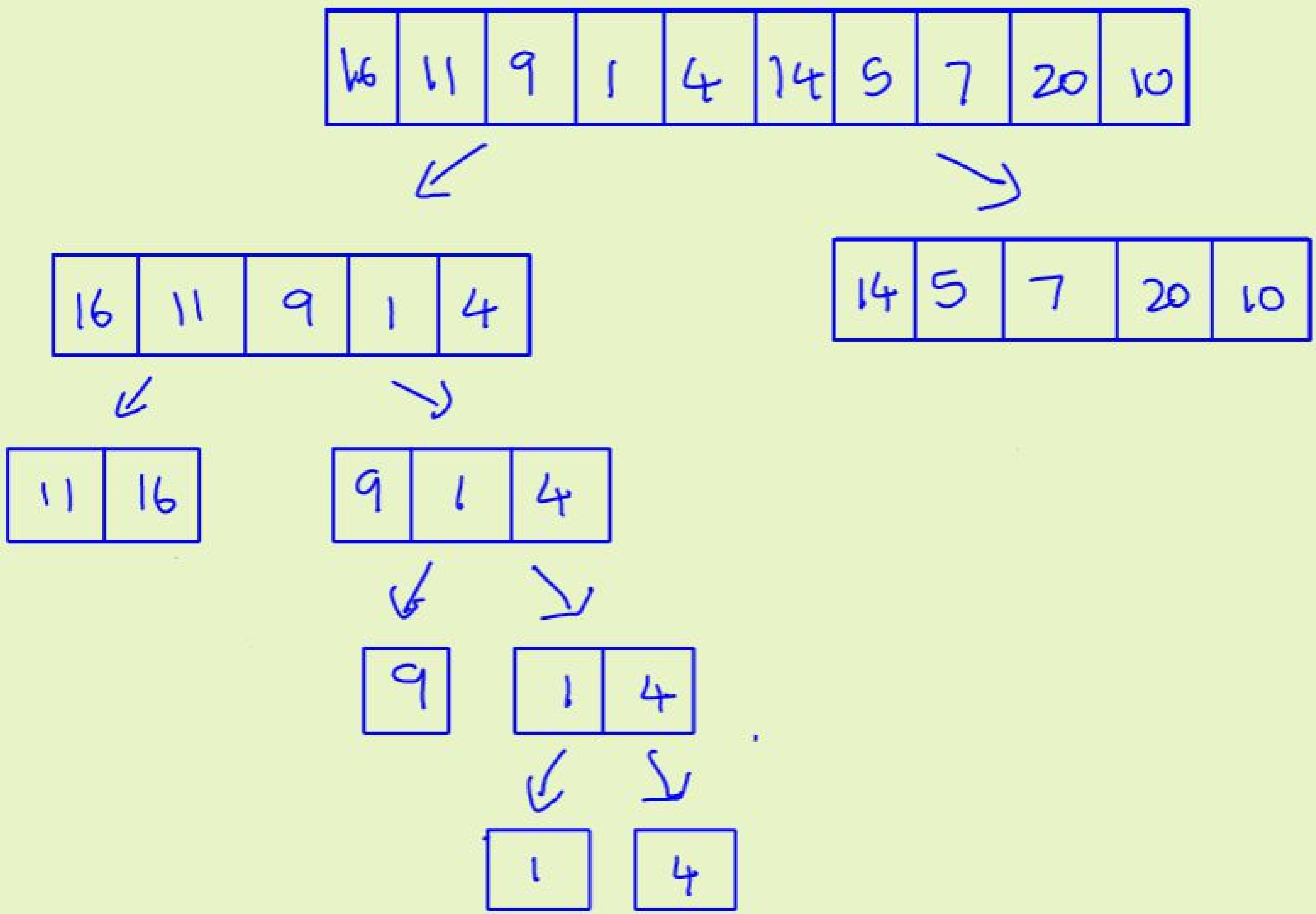
Merge Sort

Divide-and-conquer paradigm



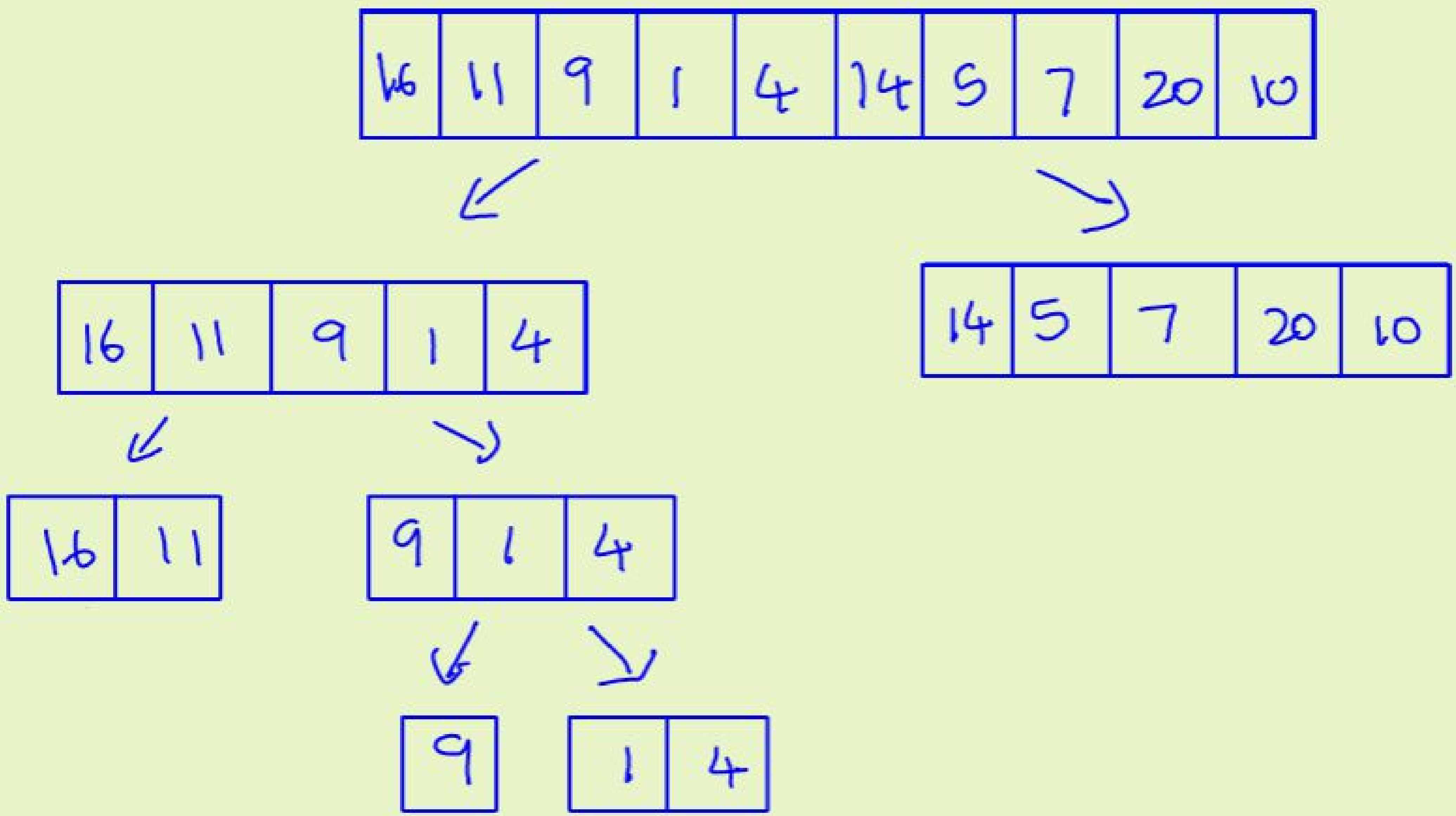
Merge Sort

Divide-and-conquer paradigm



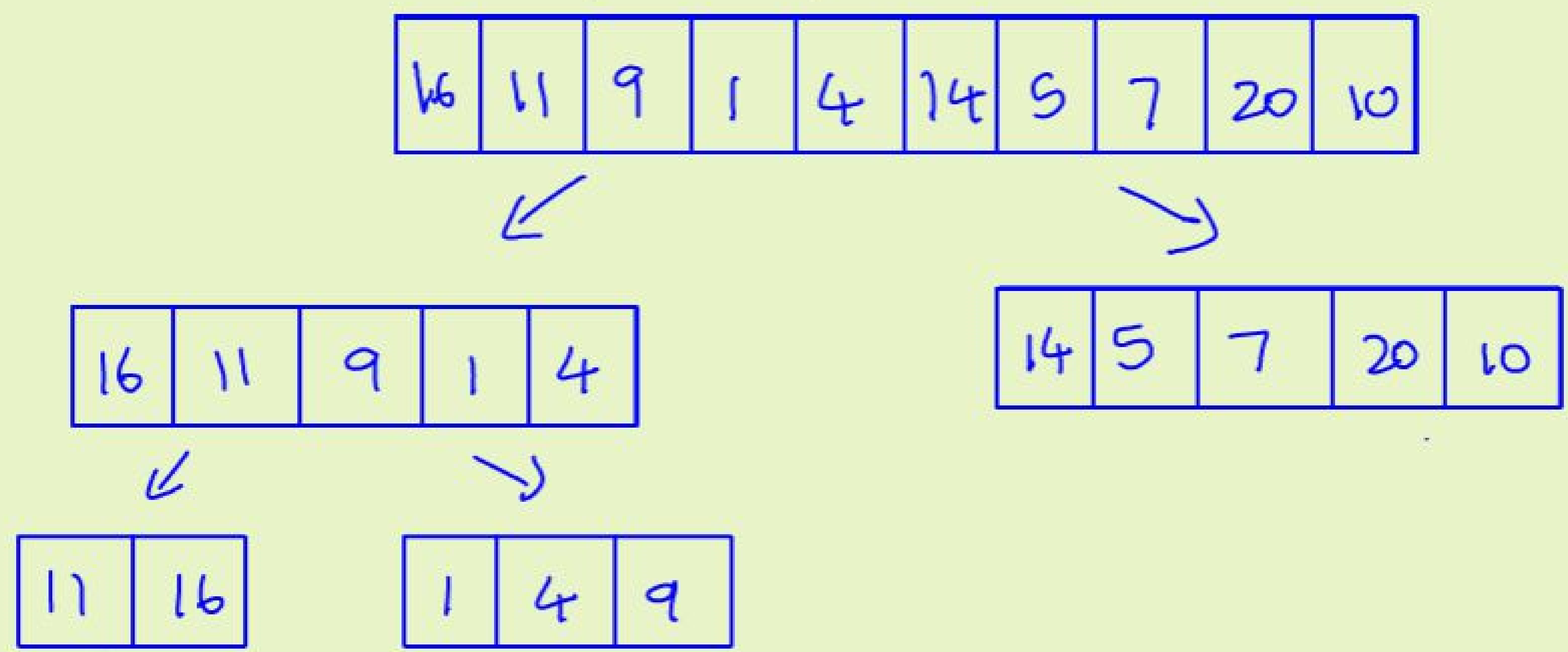
Merge Sort

Divide-and-conquer paradigm



Merge Sort

Divide-and-conquer paradigm



Merge Sort

Divide-and-conquer paradigm

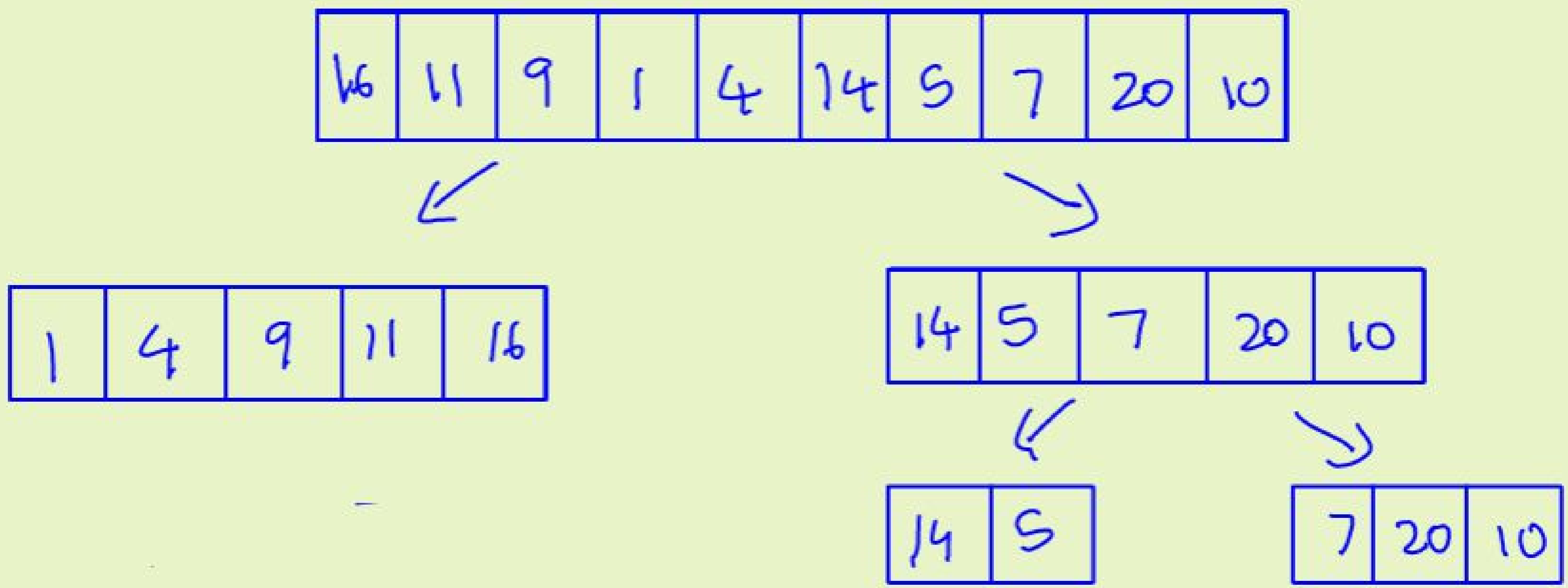
16	11	9	1	4	14	5	7	20	10

1	4	9	11	16

14	5	7	20	10

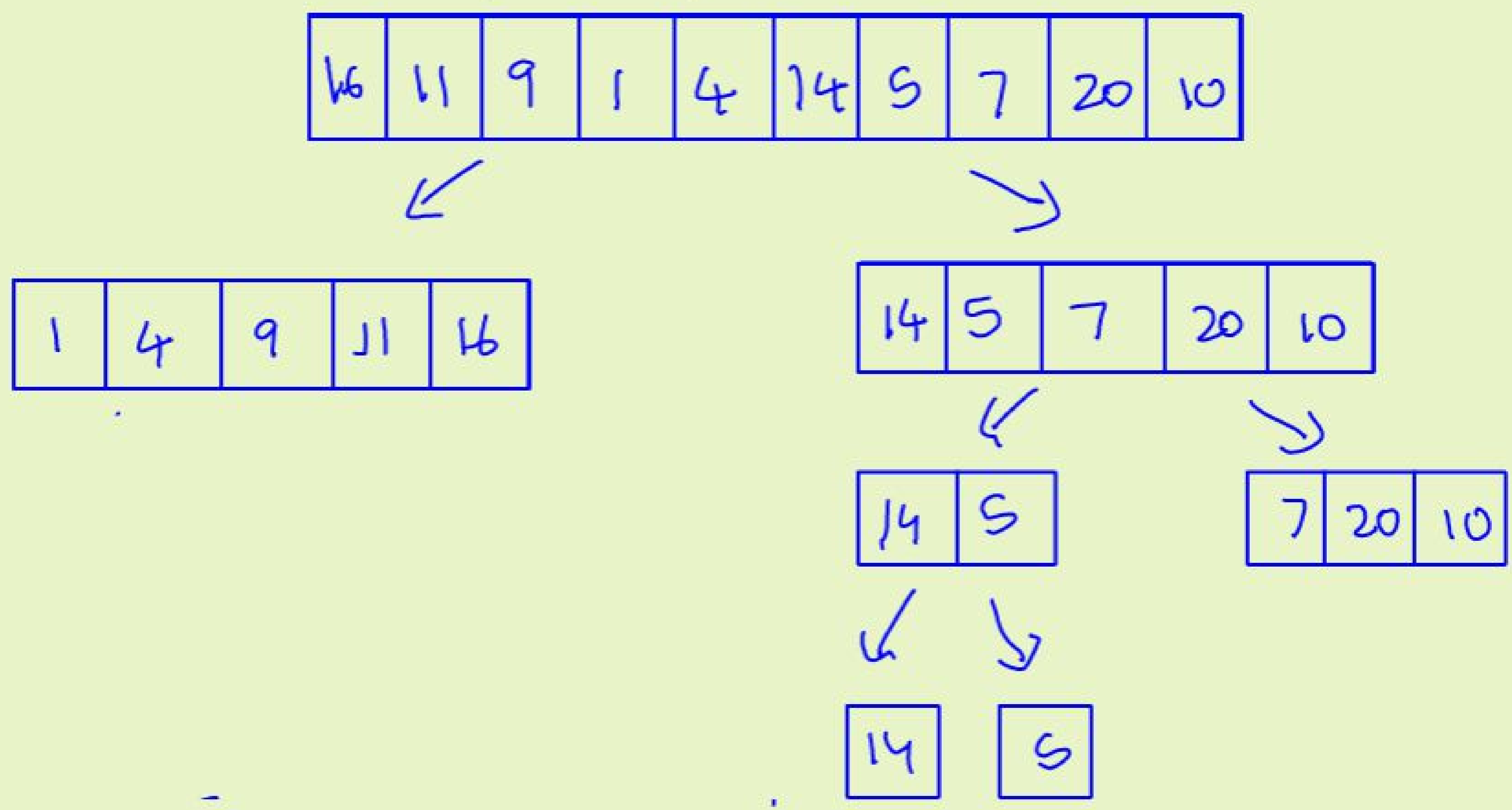
Merge Sort

Divide-and-conquer paradigm



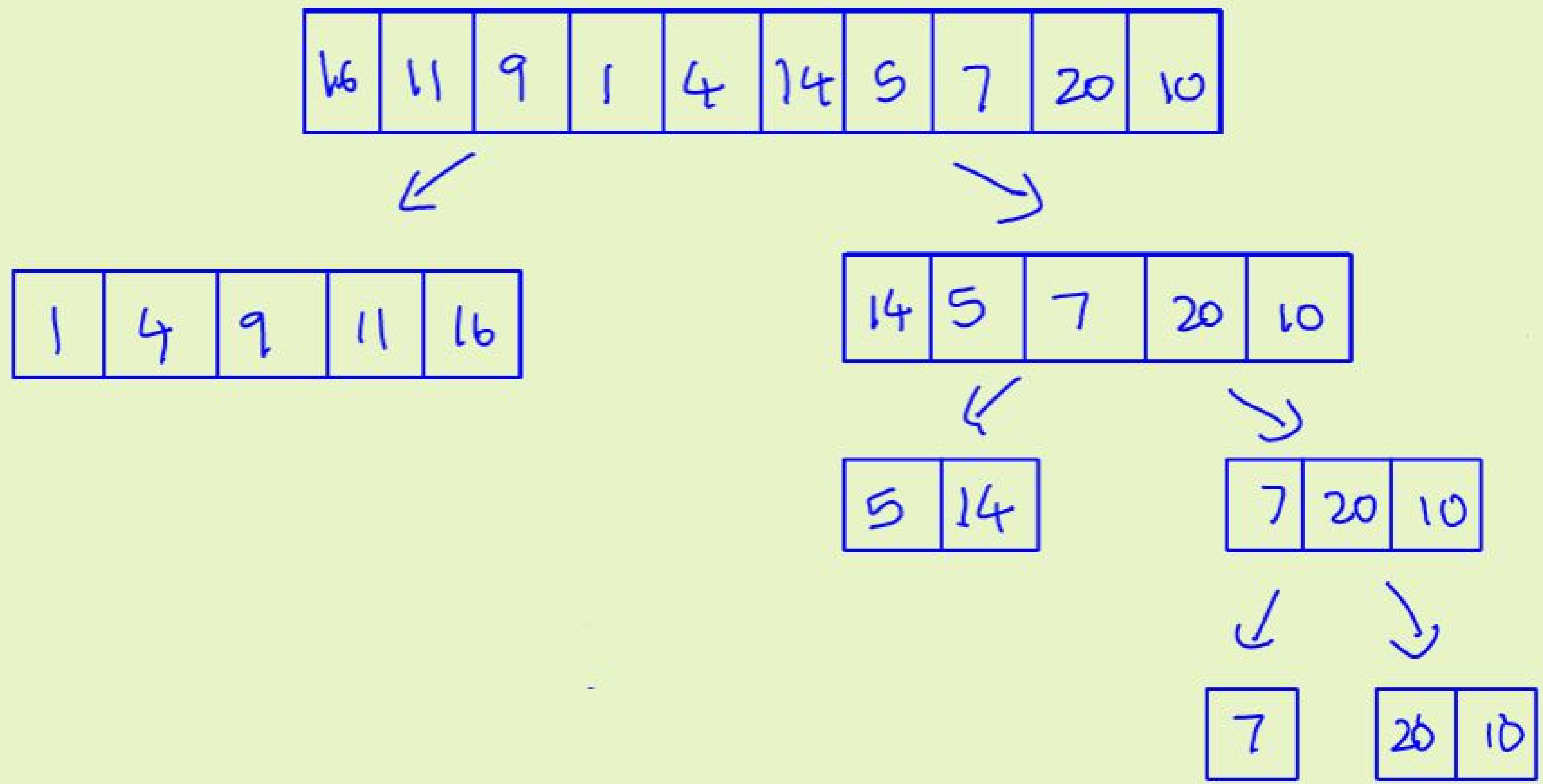
Merge Sort

Divide-and-conquer paradigm



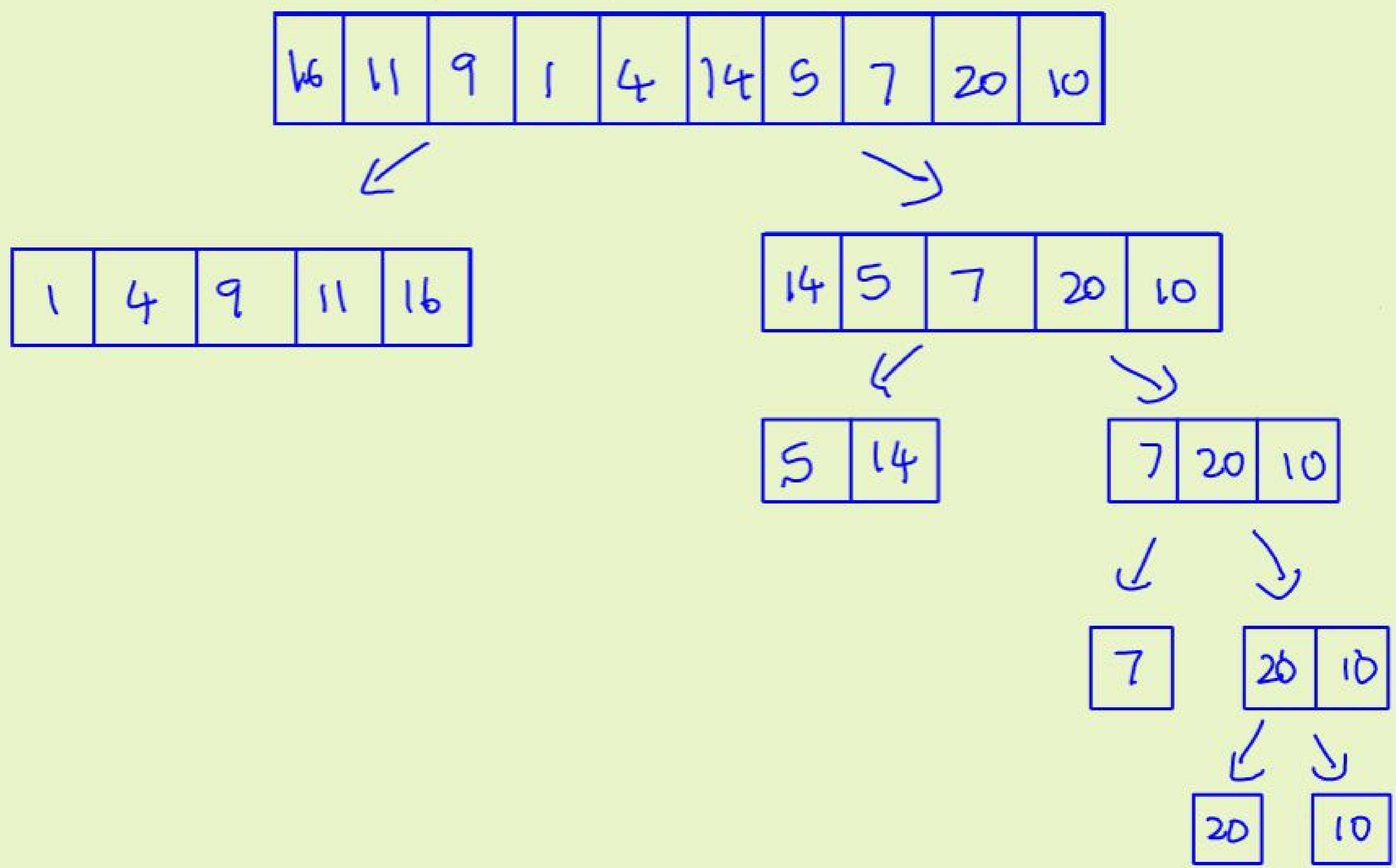
Merge Sort

Divide-and-conquer paradigm



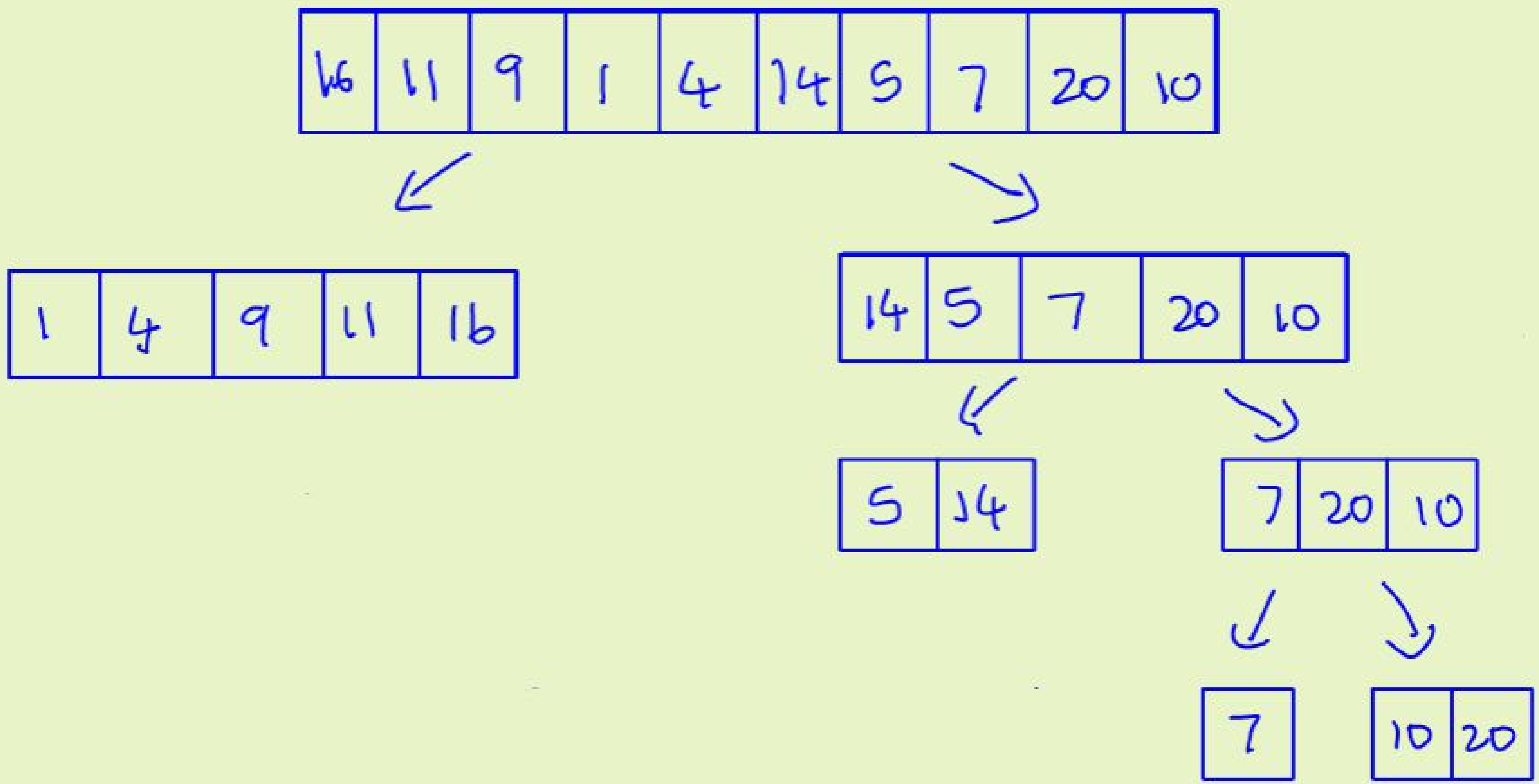
Merge Sort

Divide-and-conquer paradigm



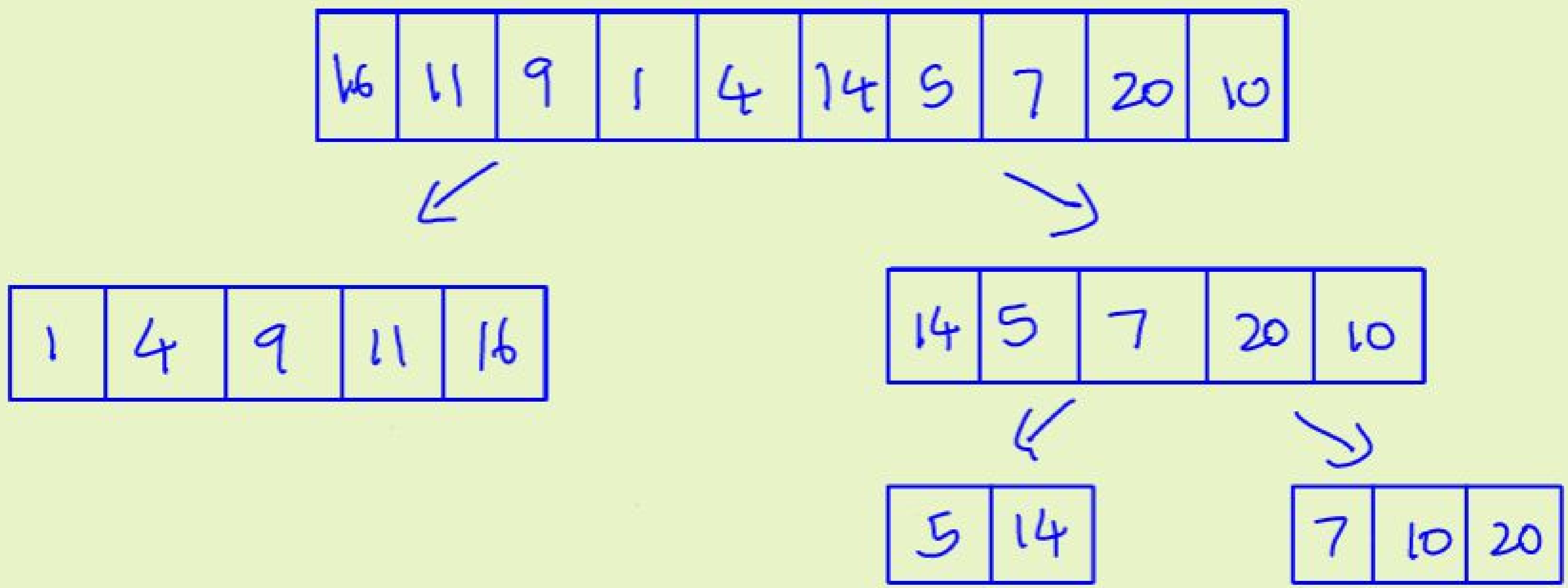
Merge Sort

Divide-and-conquer paradigm



Merge Sort

Divide-and-conquer paradigm



Merge Sort

Divide-and-conquer paradigm

16	11	9	1	4	14	5	7	20	10

1	4	9	11	16

5	7	10	14	20

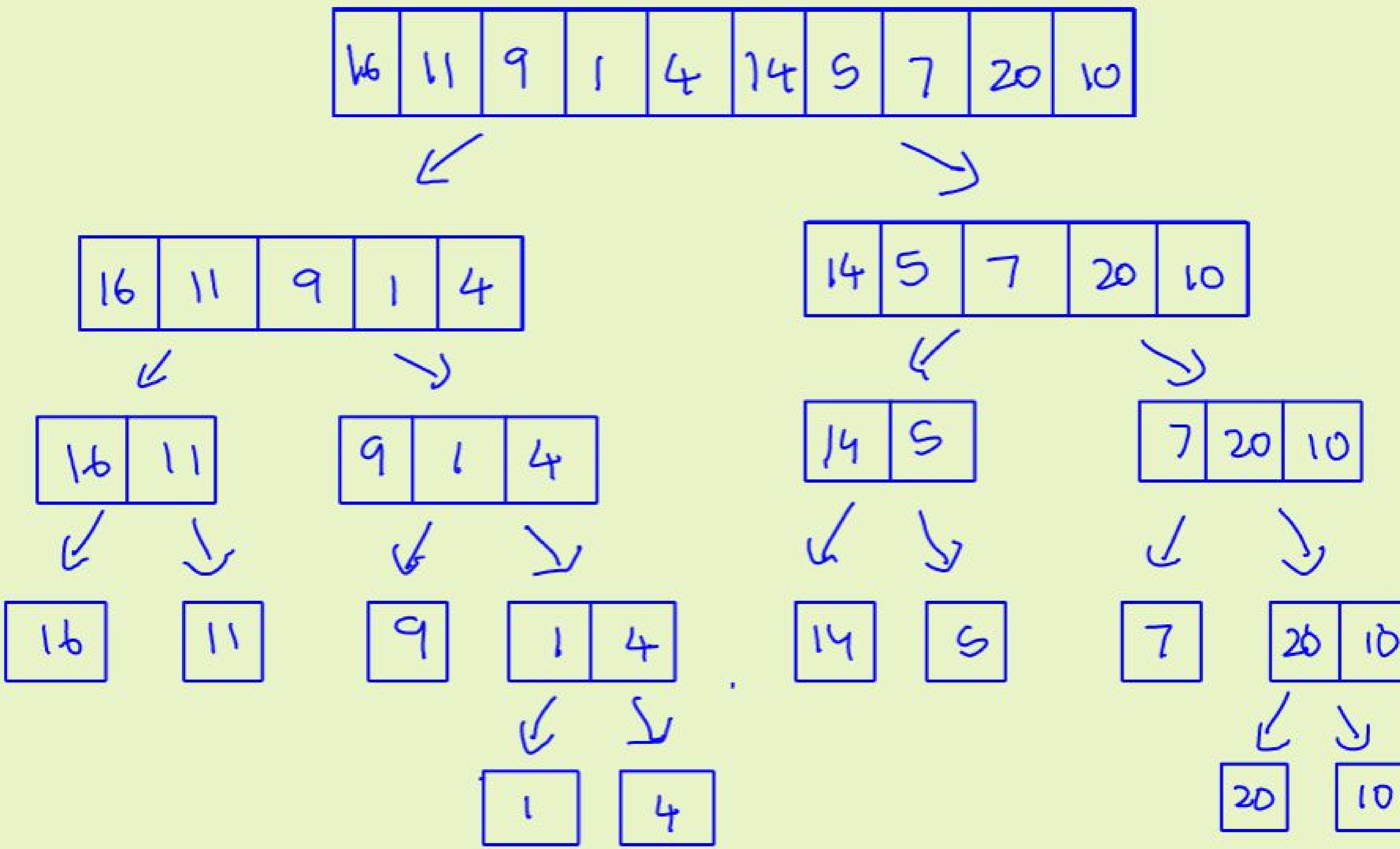
Merge Sort

Divide-and-conquer paradigm

1	4	5	7	9	10	11	14	16	20

Merge Sort

Recursion Tree



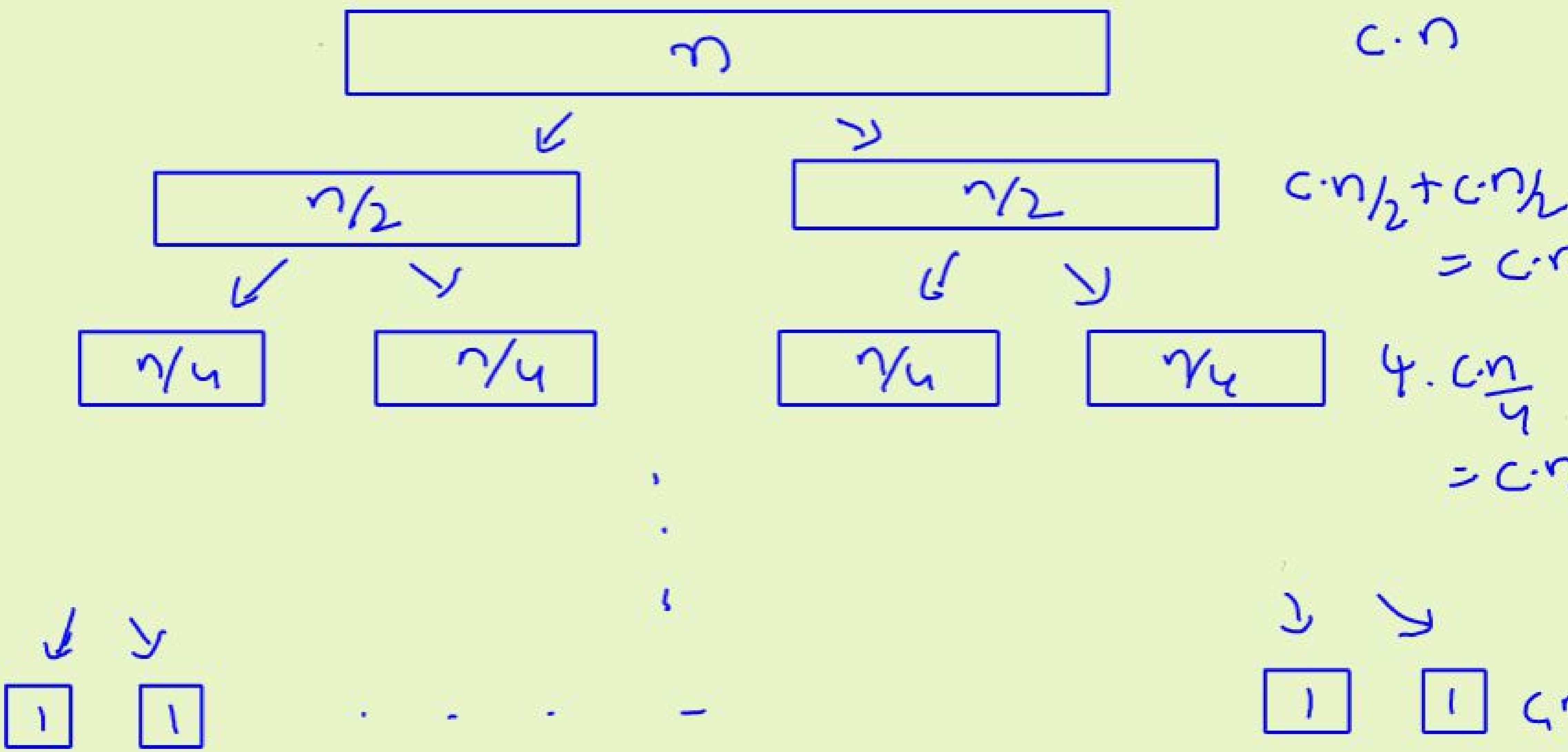
Merge Sort

```
MergeSort(A)
  if (length(A) > 1)
      mid = ⌊(1+length(A))/2⌋
      L := new array of length mid
      R := new array of length length(A)-mid
      for (i=1 to mid)
          L[i] := A[i]
      for (i=1 to length(A)-mid)
          R[i] := A[mid+i]
      Mergesort(L)
      Mergesort(R)
      Merge(L,R,A)
```

Correctness:

Induction

Running Time of Merge Sort



Running time $\leq c \cdot n \cdot \log n$

Asymptotics O, Ω, Θ

$\rightarrow$ $T(n) = O(f(n))$ (also $T(n) \in O(f(n))$)

"$T(n)$ grows at most as fast as $f(n)$ ignoring constant factors."

Attempt 1 at definition: There is a constant K s.t $T(n) \leq K \cdot f(n)$ for all n

Is $n^2 = O((n-1)^2)$? No.

We are interested in what happens for large n.

Asymptotics O, Ω, Θ

→ $T(n) = O(f(n))$ (also $T(n) \in O(f(n))$)

"$T(n)$ grows at most as fast as $f(n)$ ignoring constant factors."

Definition: There are constants n_0, K s.t

$$T(n) \leq K \cdot f(n) \text{ for all } n \geq n_0$$

Is $n^2 = O((n-1)^2)$?

Is $n^2 = O(n)$?

YES: $n_0 = 2$, $K = 4$

Asymptotics O, Ω, Θ

→ $T(n) = \Omega(f(n))$ (also $T(n) \in \Omega(f(n))$)

"$T(n)$ grows at least as fast as $f(n)$ ignoring constant factors."

Definition : There are constants n_0, K s.t

$T(n) \geq K \cdot f(n)$ for all $n \geq n_0$

Is $n^2 - 3n = \Omega(n)$? Is $n^2 - 3n = \Omega(n^2)$?

YES: $n_0 = 5$, $K = \frac{1}{2}$

Asymptotics O, Ω, Θ

→ $T(n) = \Theta(f(n))$ (also $T(n) \in \Theta(f(n))$)

"$T(n)$ grows exactly as fast as $f(n)$ ignoring constant factors."

Definition: $T(n) = O(f(n))$ &
$T(n) = \Omega(f(n))$

Is $n^2 - 3n = \Theta(n)$?

Limit definitions (works often, but not always)

$T(n), f(n) \geq 0$

$T(n) = O(f(n))$: $\lim_{n \to \infty} \frac{T(n)}{f(n)}$ exists & not ∞

$T(n) = \Omega(f(n))$: $\lim_{n \to \infty} \frac{f(n)}{T(n)}$ —— " ——

Issue: limit may not exist

Informal use

→ "Running time is $O(n^2)$" instead of "Running time is $\Theta(n^2)$"