Tutorial 10
Merge Sort and Quicksort

CSCI2100A/ESTR2102 Data Structures (2025 Spring)

Outline

- Merge Sort
  - Prerequisite: “Merging Two Sorted Arrays”
  - Algorithm
  - Implementation
- Quicksort
  - Algorithm
  - Implementation
  - Analysis
- Exercise

Merge Sort

Prerequisite: Merging Two Sorted Arrays

- *Input*: Two sorted sub-arrays of sizes m and n respectively
- *Output*: A merged and sorted array of size $(m + n)$

1	14	19	25

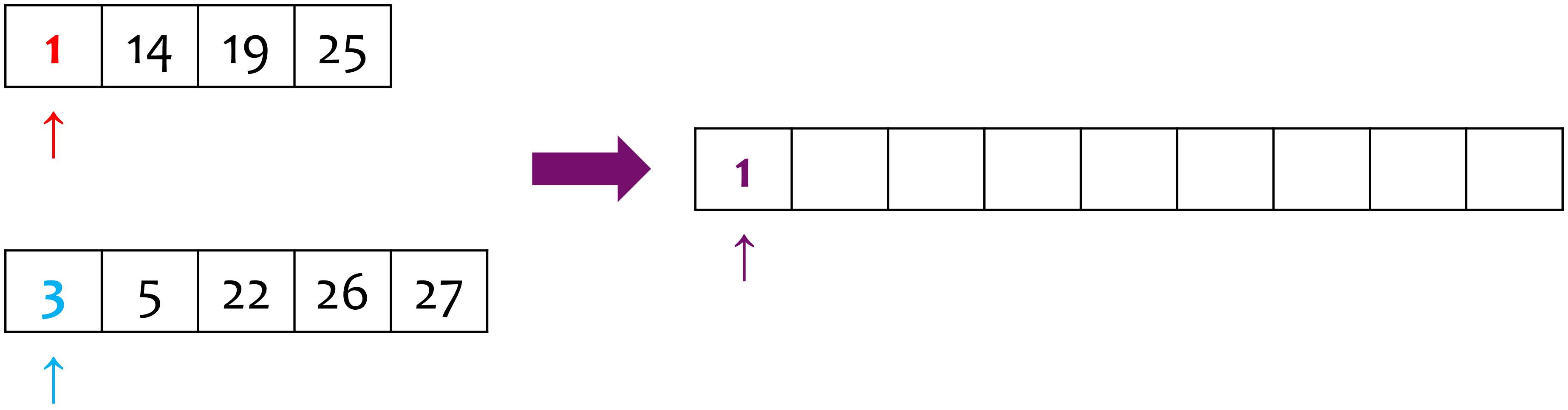
3	5	22	26	27

1	3	5	14	19	22	25	26	27

- Put a “pointer” to the first element of each sub-array
- While both “pointers” are not out-of-bounds do:
  - Compare an element from each sub-array each time and put the **smaller** element into the end of the output array
    - The current element must be the **smallest** element **in the sub-array** and the **largest** element **in the output array**
  - Update the “pointers”
- Put the **remaining** elements to the output array. Keep the order.

Example: Merging Two Sorted Arrays

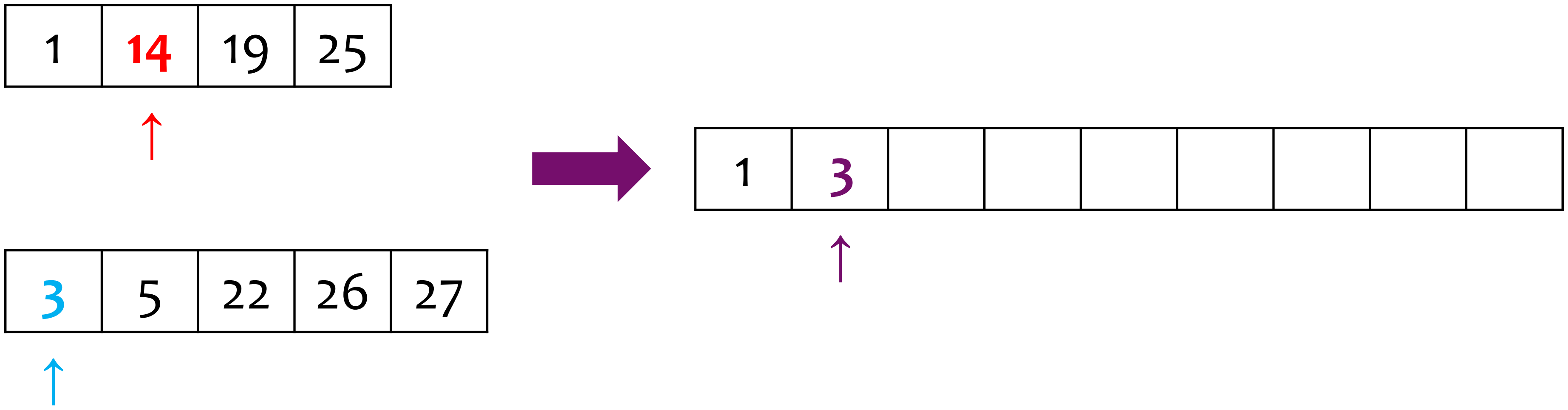
- Put a pointer to the first element of each sub-array
- While both pointers are not out-of-bounds do:
  - Compare an element from each sub-array each time and put the smaller element into the end of the output array
  - Update the pointers



Since 1 < 3 , we put 1 into the output array.

Example: Merging Two Sorted Arrays

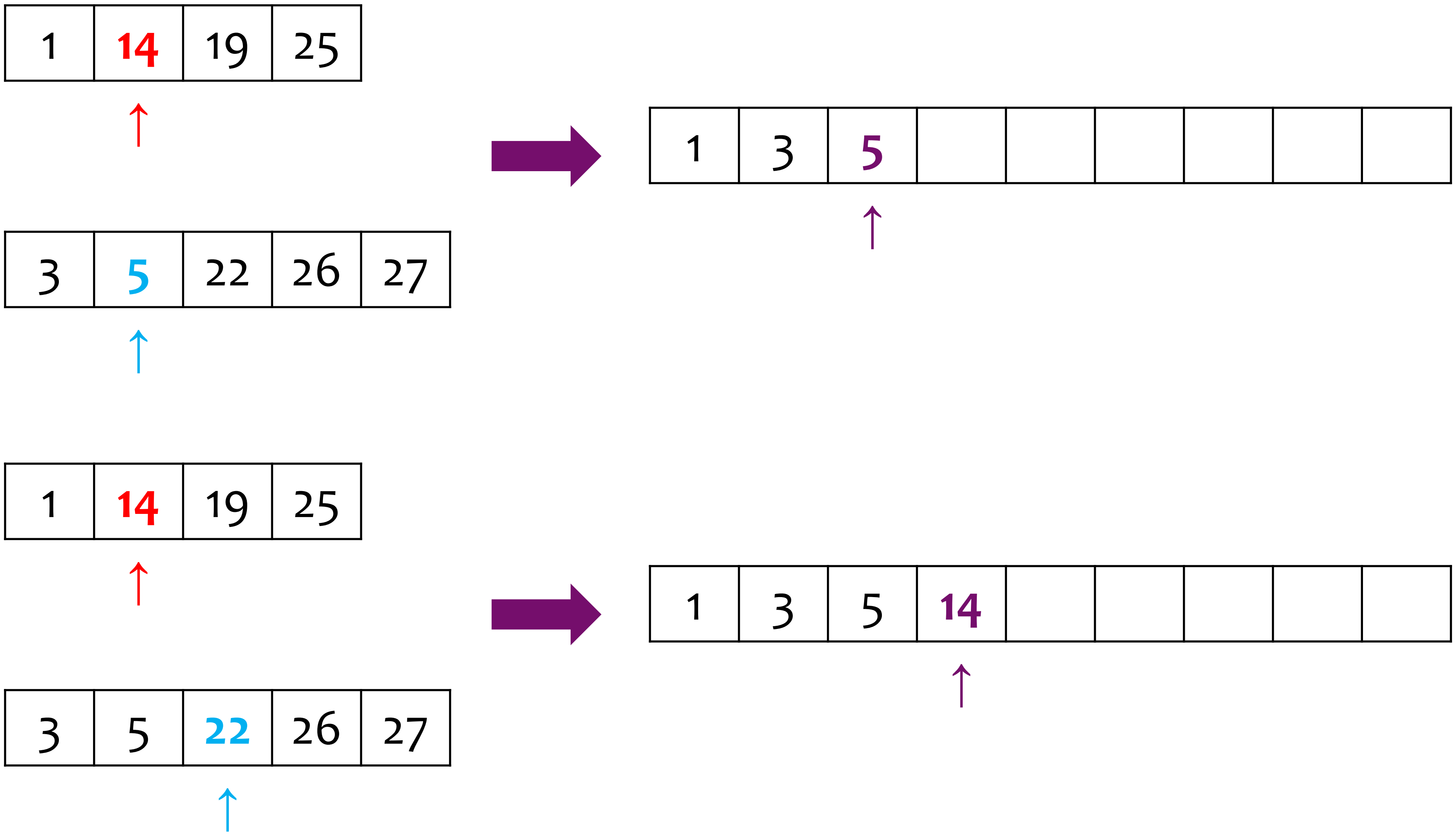
- While both pointers are not out-of-bounds do:
  - Compare an element from each sub-array each time and put the smaller element into the end of the output array
  - Update the pointers



Since 3 < 14 , we put 3 into the output array.

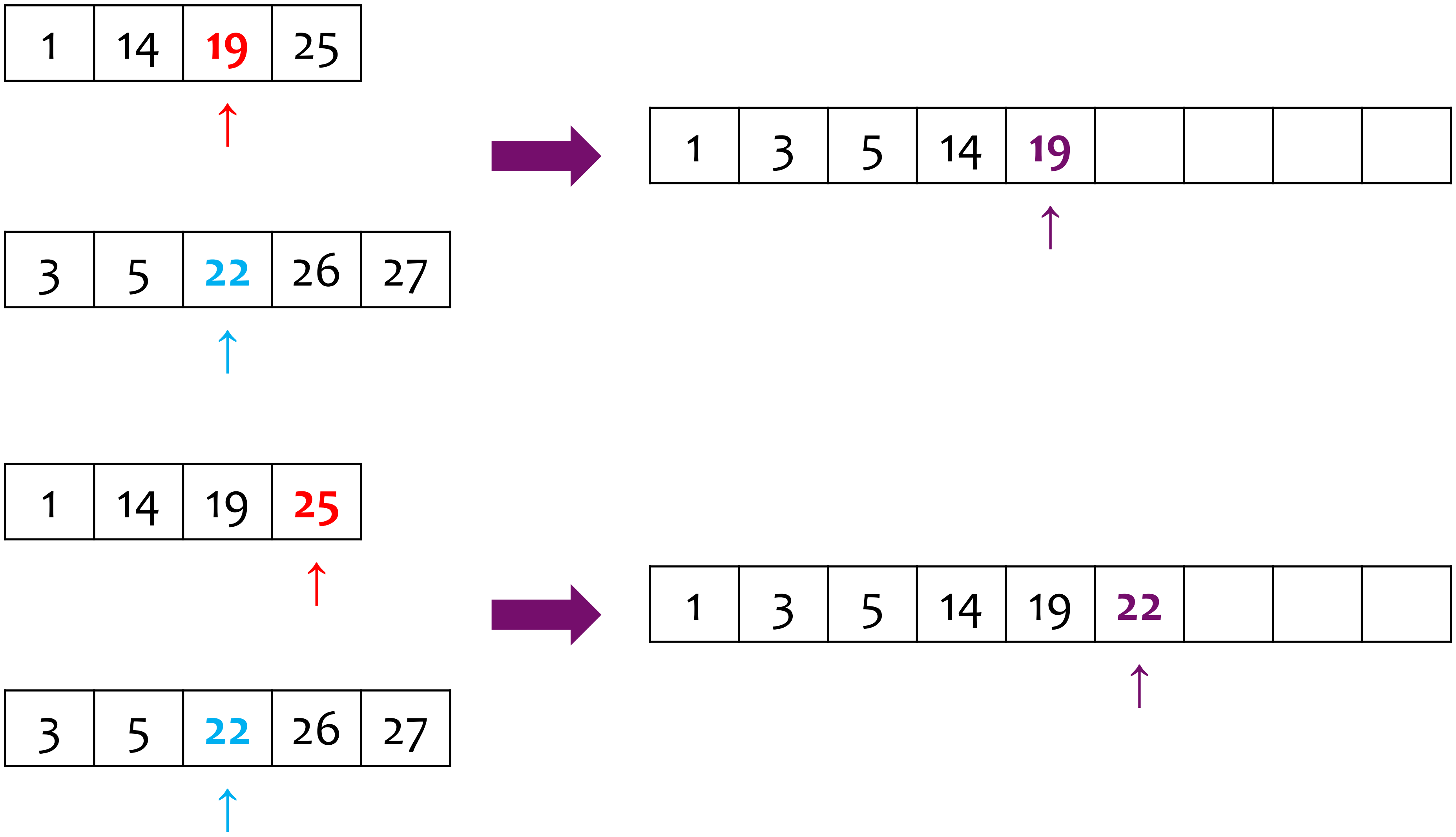
Example: Merging Two Sorted Arrays

- Repeat until one of the pointers is out-of-bounds



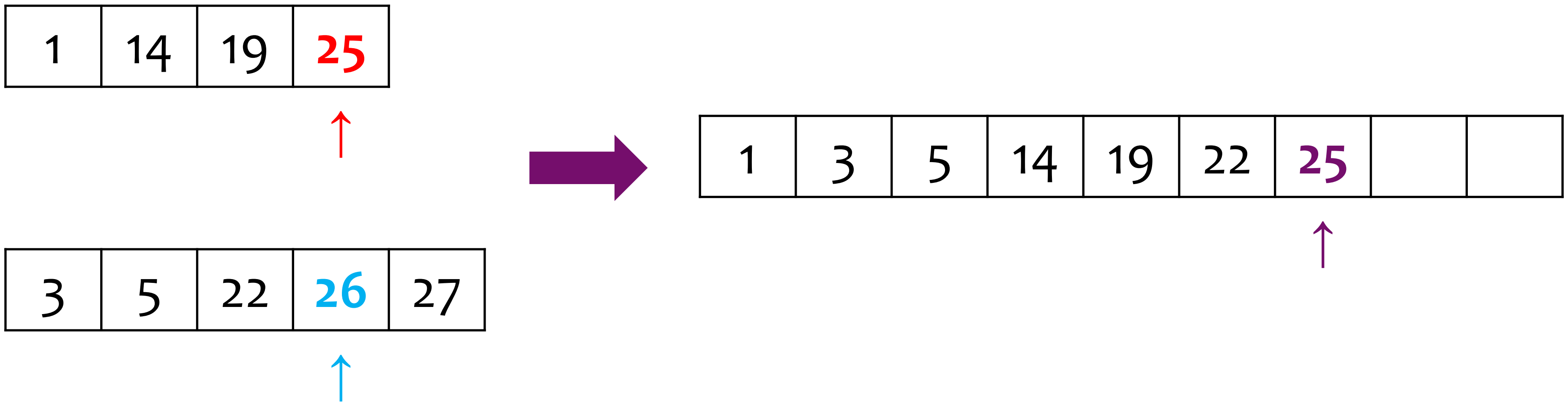
Example: Merging Two Sorted Arrays

- Repeat until one of the pointers is out-of-bounds

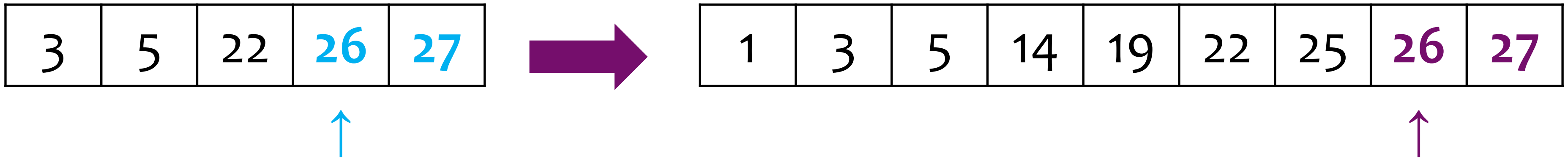


Example: Merging Two Sorted Arrays

- Repeat until one of the pointers is out-of-bounds



- Put the remaining elements to the output array. Keep the order.



Implementation of “Merging Two Sorted Arrays”

```
void merge_two_arrays(int n1, int n2, int *arr1, int *arr2, int *merged) {
    int i = 0; // Pointer to the first element of arr1
    int j = 0; // Pointer to the first element of arr2
    int k = 0; // Pointer to the first element of the merged array

    // Both pointers are not out-of-bounds
    while (i < n1 && j < n2) {
        // Put smaller element into output and update pointers
        if (arr1[i] <= arr2[j]) {
            merged[k++] = arr1[i++];
        } else {
            merged[k++] = arr2[j++];
        }
    }

    ... // Remaining elements
}
```

Implementation of “Merging Two Sorted Arrays”

```c
void merge_two_arrays(int n1, int n2, int *arr1, int *arr2, int *merged) {
    ...

    // Remaining elements
    while (i < n1) {
        merged[k++] = arr1[i++];
    }

    while (j < n2) {
        merged[k++] = arr2[j++];
    }
}
```

- *Remarks*:
  - Time complexity: $O(m + n)$
  - We implement the “pointers” as variables which store the array indices

Merge Sort Algorithm: Divide-and-conquer

- *Input*: An array of size n
- *Output*: A sorted array

19	25	14	1	26	22	5	27	3

→

1	3	5	14	19	22	25	26	27

- When $n > 1$ do:
  - Divide: Divide the array into two sub-arrays (**disjoint subsets**) of smaller size
  - Conquer: Solve the sub-problem (sorting) for each sub-array
  - Merge: Merge the sorted sub-arrays to form the required sorted array (**"Merging two sorted arrays"**)
- When $n = 0$ or 1, the arrays must be sorted

Recursion!

Example: Merge Sort

19	25	14	1

- Divide: Divide the array into two sub-arrays of smaller size
  - In general, we divide it into **two halves** (or ± 1)

19	25

14	1

- Conquer: Solve the sub-problem for each sub-array
  1. Consider [19, 25]
     - Divide: Divide the array into two sub-arrays of smaller size

19

25

     - Conquer: Solve the sub-problem for each sub-array **(Done ∵ Base case)**
     - Merge: Merge the sorted sub-arrays to form the required sorted array

19	25

Example: Merge Sort

19	25	14	1

- Divide: Divide the array into two sub-arrays of smaller size
  - In general, we divide it into **two halves** (or ± 1)

19	25

14	1

- Conquer: Solve the sub-problem for each sub-array

2. Consider [14, 1]
   - Divide: Divide the array into two sub-arrays of smaller size

14

1

   - Conquer: Solve the sub-problem for each sub-array **(Done $\because$ Base case)**
   - Merge: Merge the sorted sub-arrays to form the required sorted array

1	14

Example: Merge Sort

19	25	14	1

- Divide: Divide the array into two sub-arrays of smaller size
  - In general, we divide it into **two halves** (or ± 1)

19	25

14	1

- Conquer: Solve the sub-problem for each sub-array

19	25

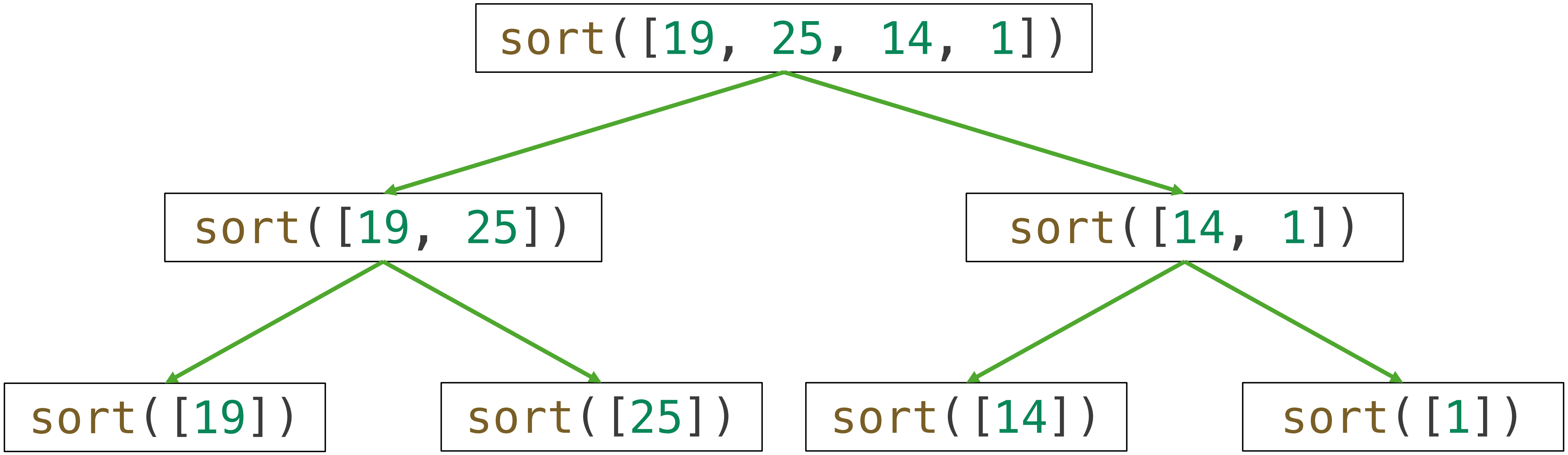
1	14

- Merge: Merge the sorted sub-arrays to form the required sorted array

1	14	19	25

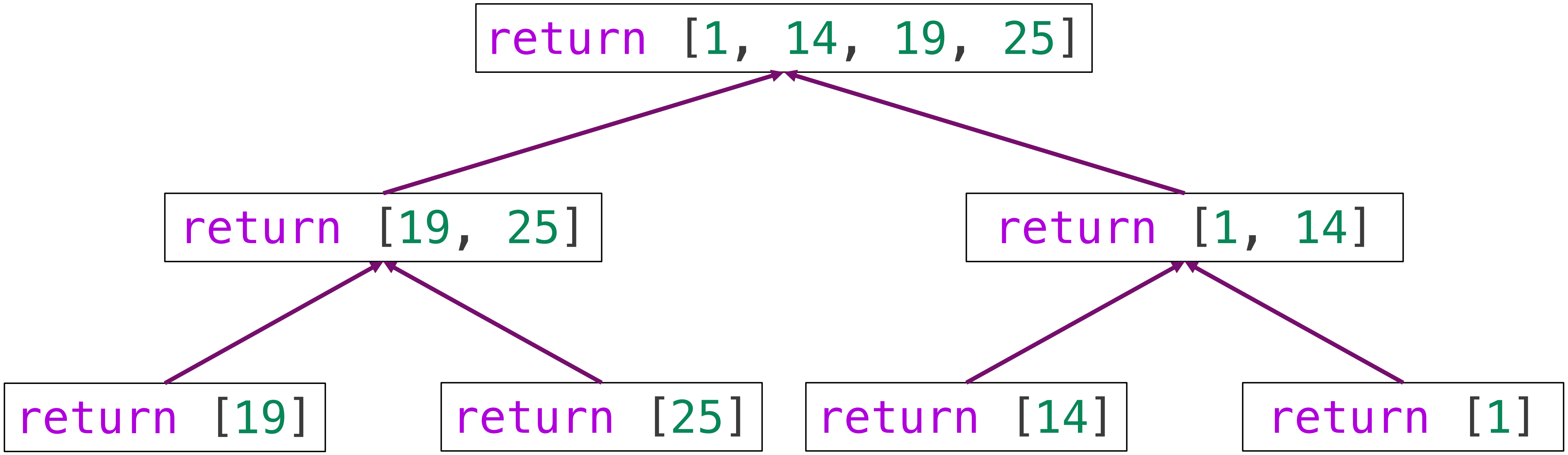
Merge Sort Tree

- We can depict the merge sort by a binary tree
  - Each node represents a recursive call
  - Each recursive call **makes two recursive calls** (“left child” and “right child”)
  - The root is the input array
  - The leaves are the base cases (termination condition)

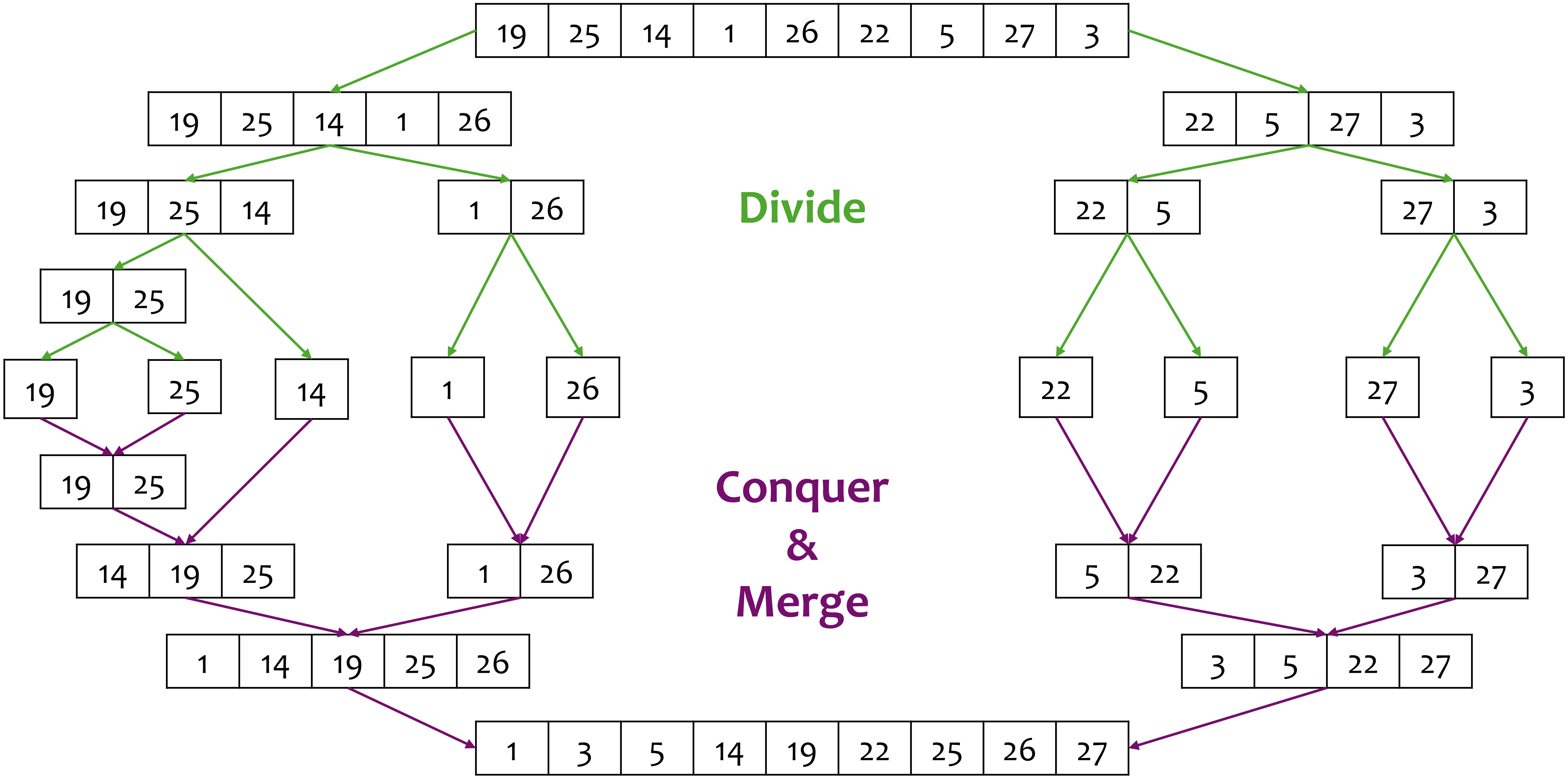


Merge Sort Tree

- We can depict the merge sort by a binary tree
  - ***Before:*** Each node stores the **unsorted** sub-array
  - ***After:*** Each node stores the **sorted** sub-array



Example: Merge Sort

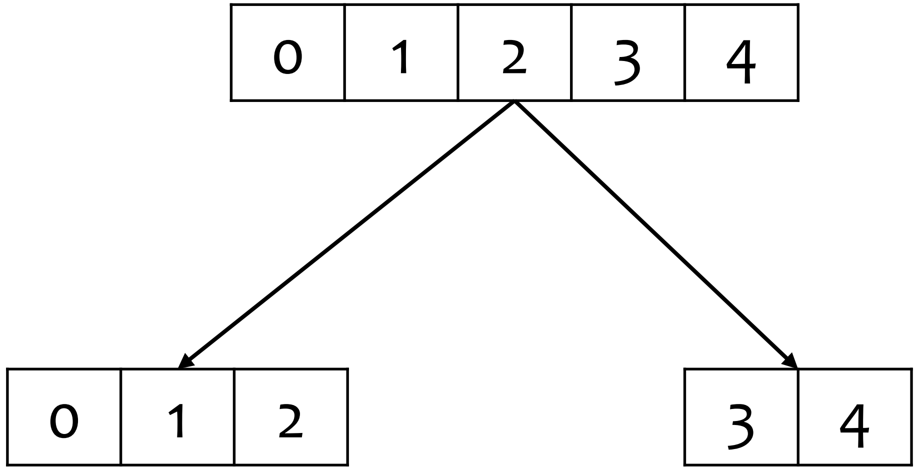


Implementation of Merge Sort (Version 1)

```
void merge_sort(int n, int *arr) {
    int i; // Loop variable
    int n1, n2; // Number of elements in the sub-arrays
    int *arr1, *arr2; // Sub-arrays

    if (n > 1) {
        // Divide: Divide the array into two sub-arrays of smaller size
        ... // The next page

        // Conquer: Solve the sub-problem for each sub-array
        merge_sort(n1, arr1);
        merge_sort(n2, arr2);

        // Merge: Merge the sorted sub-arrays to form the required sorted array
        merge_two_arrays(n1, n2, arr1, arr2, arr);
        free(arr1);
        free(arr2);
    }
}
```

Implementation of Merge Sort (Version 1)

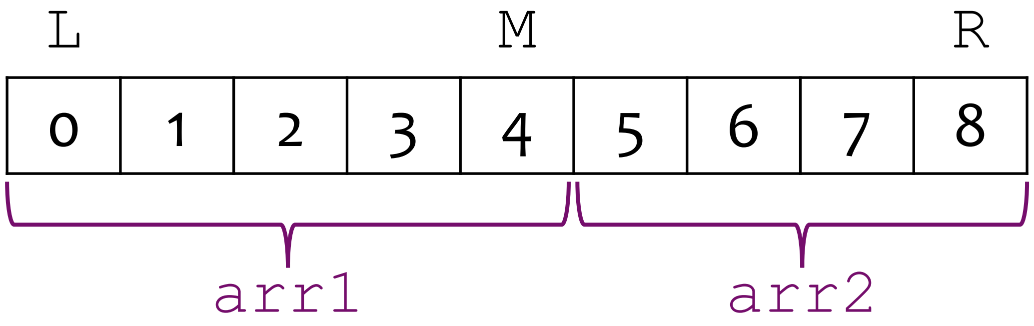
```
...

// Divide: Divide the array into two sub-arrays of smaller size
n1 = n / 2;
n2 = n - n1;
arr1 = (int *)malloc(n1 * sizeof(int));
arr2 = (int *)malloc(n2 * sizeof(int));

for (i = 0; i < n1; i++) {
    arr1[i] = arr[i];
}
for (i = 0; i < n2; i++) {
    arr2[i] = arr[n1+i];
}

...
```



Implementation of Merge Sort (Version 2)

- What if we use only one array to represent the entire input array (`arr`) and one buffer array (`tmp`) in total?
  - Store the **indices** to the first elements of each sub-array

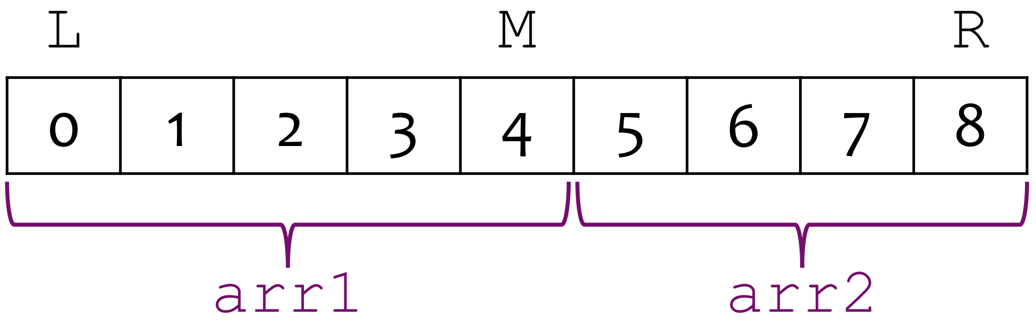
```
void msort(int *arr, int *tmp, int left, int right) {
    int mid;
    if (left < right) {
        mid = (left + right) / 2;
        msort(arr, tmp, left, mid);
        msort(arr, tmp, mid+1, right);
        merge_two_arrays(arr, tmp, left, mid, right);
    }
}
```



Implementation of Merge Sort (Version 2)

- Slightly modify the function `merge_two_arrays()`

```
void merge_two_arrays(int *arr, int *tmp, int left, int mid, int right) {
    int i = 0; // Pointer to the first element of the left sub-array
    int j = mid + 1; // Pointer to the first element of the right sub-array
    int k = 0; // Pointer to the merged array

    // Both pointers are not out-of-bounds
    while (i <= mid && j <= right) { ...
    }

    ... // Remaining elements (with similar changes)

    // Copy back
    for (k = 0; k <= right; k++) {
        arr[k] = tmp[k];
    }
}
```



Implementation of Merge Sort (Version 2)

- Wrap the function `msort()`
  - Create and destroy the buffer array

```c
void merge_sort(int n, int *arr) {
    int *tmp = (int *)malloc(n * sizeof(int)); // Buffer
    msort(arr, tmp, 0, n-1);
    free(tmp);
}
```

L								R
0	1	2	3	4	5	6	7	8

Quicksort

Quicksort Algorithm: Divide-and-conquer

- *Input*: An array of size n
- *Output*: A sorted array

19	25	14	1	26	22	5	27	3

$\Rightarrow$

1	3	5	14	19	22	25	26	27

- When $n > 1$ do:
  - Divide: Choose a **pivot** (denote its key by p). Re-arrange the elements such that

Elements with key $\leq p$	Pivot p	Elements with key $> p$

  - Conquer: Solve the sub-problem (sorting) for each half

- When $n = 0$ or 1, the arrays must be sorted

Recursion!

Example: Quicksort

19	25	14	1	26	22	5	27	3

- Divide: Choose a pivot and re-arrange the elements
  - For simplicity, we choose the **first element** as the pivot

1. Put a "pointer" to the first element and a "pointer" to the last element.

19	25	14	1	26	22	5	27	3
↑ L								↑ R

2. Repeat the following steps while "L" is on the left of "R"
   ① Move "L" to the right while `arr[L] <= p` and "L" is on the left of "R"

19	25	14	1	26	22	5	27	3
(p)	↑ L							↑ R

25 > 19
∴ Stop

Example: Quicksort

19	25	14	1	26	22	5	27	3

- Divide: Choose a pivot and re-arrange the elements
  2. Repeat the following steps while “L” is on the left of “R”

     ② Move “R” to the left while `arr[R] > p`

19	25	14	1	26	22	5	27	3
(p)	↑ L							↑ R

     ③ If “L” is on the left of “R”, swap `arr[L]` and `arr[R]`

19	3	14	1	26	22	5	27	25
(p)	↑ L							↑ R

Example: Quicksort

19	25	14	1	26	22	5	27	3

- Divide: Choose a pivot and re-arrange the elements
  2. Repeat the following steps while “L” is on the left of “R”
     ① Move “L” to the right while `arr[L] <= p` and “L” is on the left of “R”
     ② Move “R” to the left while `arr[R] > p`

26 > 19
∴ Stop

19	3	14	1	26	22	5	27	25
(p)				↑ L		↑ R		

5 <= 19
∴ Stop

③ If “L” is on the left of “R”, swap `arr[L]` and `arr[R]`

19	3	14	1	5	22	26	27	25
(p)				↑ L		↑ R		

Example: Quicksort

19	25	14	1	26	22	5	27	3

- Divide: Choose a pivot and re-arrange the elements
  2. Repeat the following steps while “L” is on the left of “R”
     ① Move “L” to the right while `arr[L] <= p` and “L” is on the left of “R”
     ② Move “R” to the left while `arr[R] > p`

5 <= 19
∴ Stop

19	3	14	1	5	22	26	27	25
(p)				↑ R	↑ L			

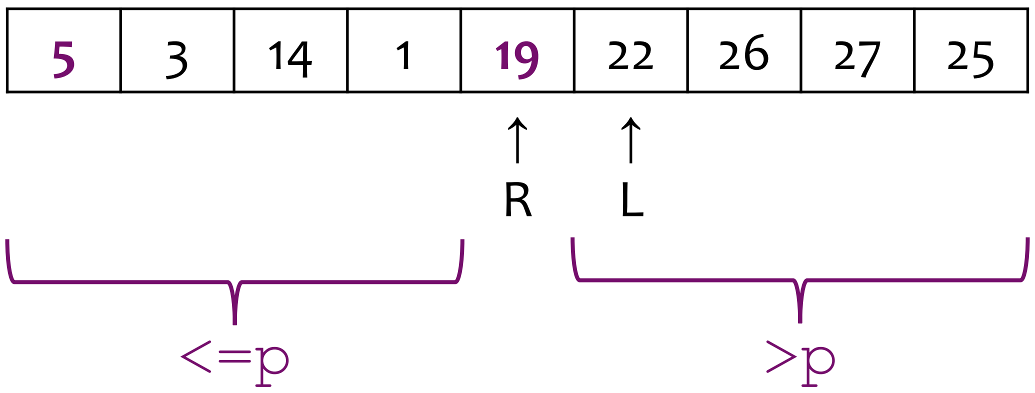
22 > 19
∴ Stop

     ③ If “L” is on the left of “R”, swap `arr[L]` and `arr[R]`
        - No need to swap!

Example: Quicksort

19	25	14	1	26	22	5	27	3

- Divide: Choose a pivot and re-arrange the elements
  3. Swap the pivot and `arr[R]`
     - “R” is the new position of the pivot



- Conquer: Solve the sub-problem for each half

1	3	5	14	19	22	25	26	27

Example: Quicksort

(Input)	19	25	14	1	26	22	5	27	3
D	5	3	14	1	**19**	22	26	27	25
C1-D	1	3	**5**	14	19	22	26	27	25
C1-C1-D	**1**	3	5	14	19	22	26	27	25
C1-C1-C2	1	3	5	14	19	22	26	27	25
C1-C2	1	3	5	14	19	22	26	27	25
C1-End	1	3	5	14	19	22	26	27	25

D: Divide (Partition)
C: Conquer
p: Pivot after partition
: Left sub-array
: Right sub-array
: Sorted

Example: Quicksort

C1-End	1	3	5	14	19	22	26	27	25
C2-D	1	3	5	14	19	**22**	26	27	25
C2-C2-D	1	3	5	14	19	22	25	**26**	27
C2-C2-C1	1	3	5	14	19	22	25	26	27
C2-C2-C2	1	3	5	14	19	22	25	26	27
C2-C2	1	3	5	14	19	22	25	26	27
C2-End	1	3	5	14	19	22	25	26	27
(Output)	1	3	5	14	19	22	25	26	27

D: Divide (Partition)
C: Conquer
p: Pivot after partition
: Left sub-array
: Right sub-array
: Sorted

Example: Quicksort

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]	Left	Right
[19	25	14	1	26	22	5	27	3]	1	9
[5	3	14	1]	**19**	[22	26	27	25]	1	4
[1	3]	**5**	[14]	19	[22	26	27	25]	1	2
1	3	5	[14]	19	[22	26	27	25]	4	4
1	3	5	14	19	[22	26	27	25]	6	9
1	3	5	14	19	**22**	[26	27	25]	7	9
1	3	5	14	19	22	[25]	**26**	[27]	7	7
1	3	5	14	19	22	25	26	[27]	9	9
1	3	5	14	19	22	25	26	27		

Implementation of Quicksort

```
void q_sort(ElementType *arr, int left, int right) {
    int pivot; // Position of the pivot after partition

    if (left < right) { // Not the base case
        // Divide: Partition
        pivot = partition(arr, left, right);

        // Conquer: Solve sub-problems
        q_sort(arr, left, pivot-1);
        q_sort(arr, pivot+1, right);
    }
}
```

- *Remarks*: There is a built-in quicksort function `qsort()` in `<stdlib.h>`.

Implementation of Quicksort

```
int partition(ElementType *arr, int left, int right) {
    int pivot = arr[left]; // Key value of pivot (First element as pivot)
    int l = left; // Left pointer
    int r = right; // Right pointer

    while (l < r) {
        while (arr[l] <= pivot && l < r) l++; // Move to the right
        while (arr[r] > pivot) r--; // Move to the left
        if (l < r) {
            swap(arr, l, r);
        }
    }

    arr[left] = arr[r];
    arr[r] = pivot;
    return r; // New position of pivot
}
```

```
void swap(ElementType *arr, int a, int b) {
    ElementType tmp = arr[a];
    arr[a] = arr[b];
    arr[b] = tmp;
}
```

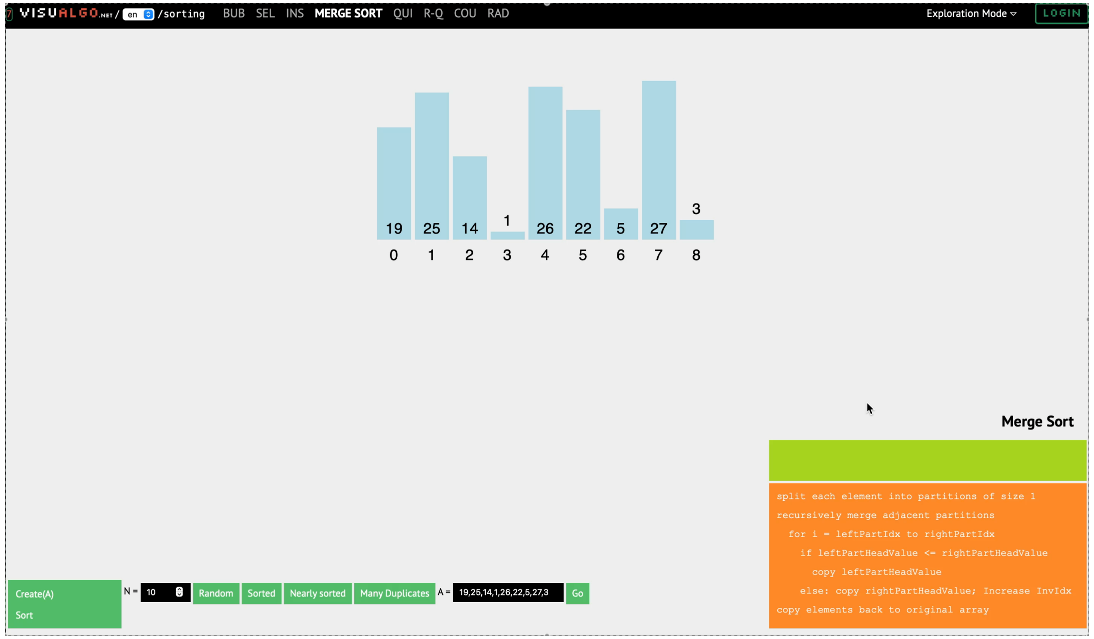
Analysis

	Merge Sort	**Quicksort**
Stable	Yes	No
Best-case Time Complexity	$O(n \log n)$	$O(n \log n)$
Average-case Time Complexity	$O(n \log n)$	$O(n \log n)$
Worst-case Time Complexity	$O(n \log n)$	$O(n^2)$
Space Complexity (Auxiliary)	$O(n)$	$O(\log n)$

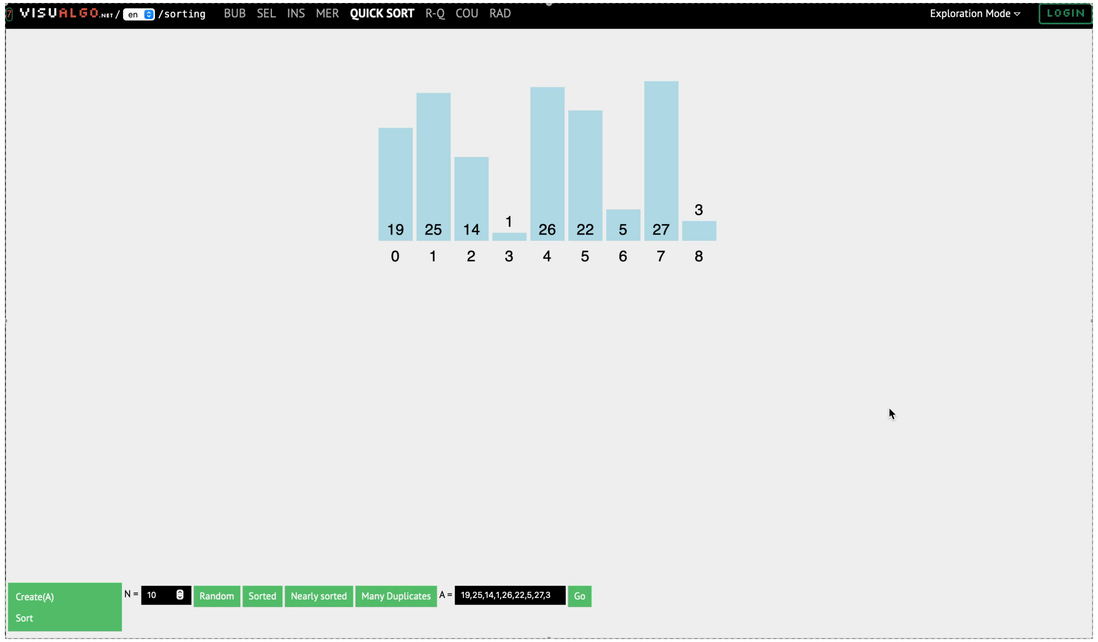
- *Remarks*:
  - Stability: After sorting, objects with the same key appear in the same order as in the original unsorted array
  - Auxiliary space: Additional space used (Do not count the space for storing the input array)

Visualization

Merge Sort



Quicksort



- Try it: https://visualgo.net/en/sorting

Exercise

Exercise 1

- Given a sequence [20, 25, 3, 28, 17, 30, 21, 0].

a) Sort the sequence in ascending order using merge sort.
   i. Plot a figure to show the result of each pass.
   ii. How many comparisons have you performed?

b) Sort the sequence in ascending order using quicksort. Suppose we always choose the first element as the pivot.
   i. Illustrate the result after each pass using a table like in the lecture notes.
   ii. Suggest an arrangement of the elements in the sequence such that the number of comparisons performed is maximized.

Exercise 2

- Given a list [D, A, T, S, R, U, C, E]. The list is sorted in alphabetical order using (i) bubble sort or (ii) quicksort with the first element as the pivot. The following shows some of the intermediate steps in the correct order. Identify the sorting algorithm used.

  - A, D, C, T, S, R, U, E
  - A, C, D, E, R, T, S, U

Exercise 3

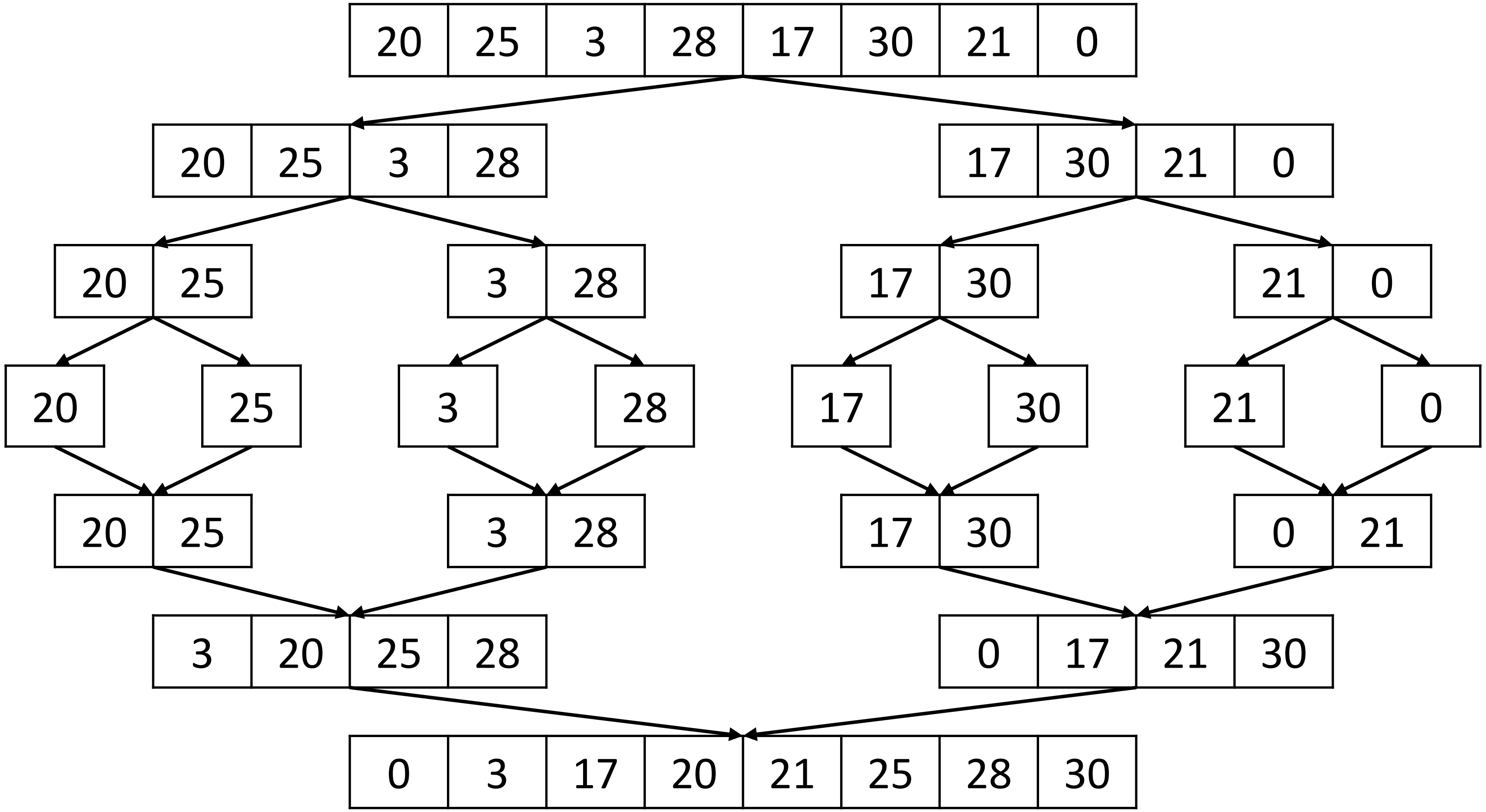
- Given a list [20, 25, 3, 26, 14, 30].

a) List all the inversion pairs in the list. Sort them in ascending order by the first element and then by the second element.

b) Sort the list in ascending order using insertion sort. Illustrate the result after each pass.

Answers to Exercise 1a (For Reference Only)

i.



ii. (1+1+1+1)+(3+3)+7 = 17 comparisons

Answers to Exercise 1b (For Reference Only)

i.

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	Left	Right
[20	25	3	28	17	30	21	0]	1	8
[17	0	3]	**20**	[28	30	21	25]	1	3
[3	0]	**17**	20	[28	30	21	25]	1	2
0	3	17	20	[28	30	21	25]	5	8
0	3	17	20	[21	25]	**28**	[30]	5	6
0	3	17	20	21	25	28	[30]	8	8
0	3	17	20	21	25	28	30		

ii. [0, 3, 17, 20, 21, 25, 28, 30] or [30, 28, 25, 21, 20, 17, 3, 0]

Answers to Exercise 2 & 3 (For Reference Only)

Exercise 2

- Bubble sort

Exercise 3

a) (20, 3), (20, 14), (25, 3), (25, 14), (26, 14)

b) 20, 25, 3, 26, 14, 30

20, 25, 3, 26, 14, 30

3, 20, 25, 26, 14, 30 [2 swaps: 25 <-> 3, 20 <-> 3]

3, 20, 25, 26, 14, 30

3, 14, 20, 25, 26, 30 [3 swaps: 26 <-> 14, 25 <-> 14, 20 <-> 14]

3, 14, 20, 25, 26, 30

- *Remarks*: Number of inversion pairs = Number of swaps in insertion sort