Algorithms and Data Structures (CSci 115)

California State University Fresno

College of Science and Mathematics

Department of Computer Science

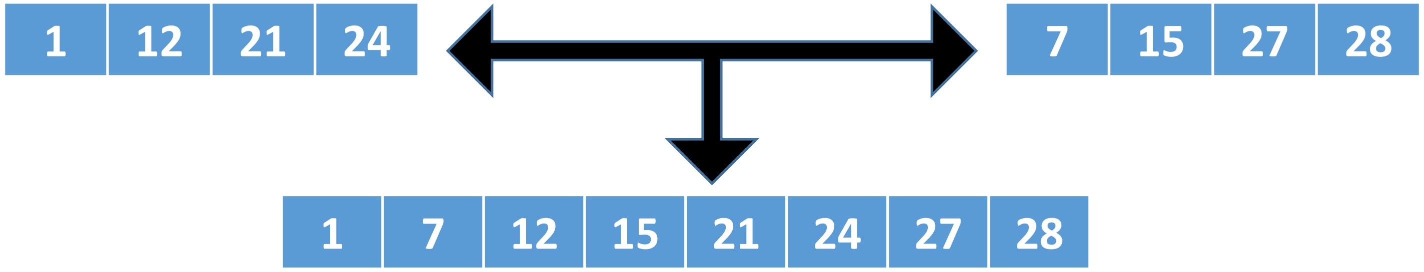
H. Cecotti

Learning outcomes

- Sorting algorithms
  - **Mergesort**
    - Merge 2 sorted arrays
    - Split then Merge
  - **Quicksort**
    - Order 2 arrays based on a pivot
    - Order then split

Algorithm - Merging

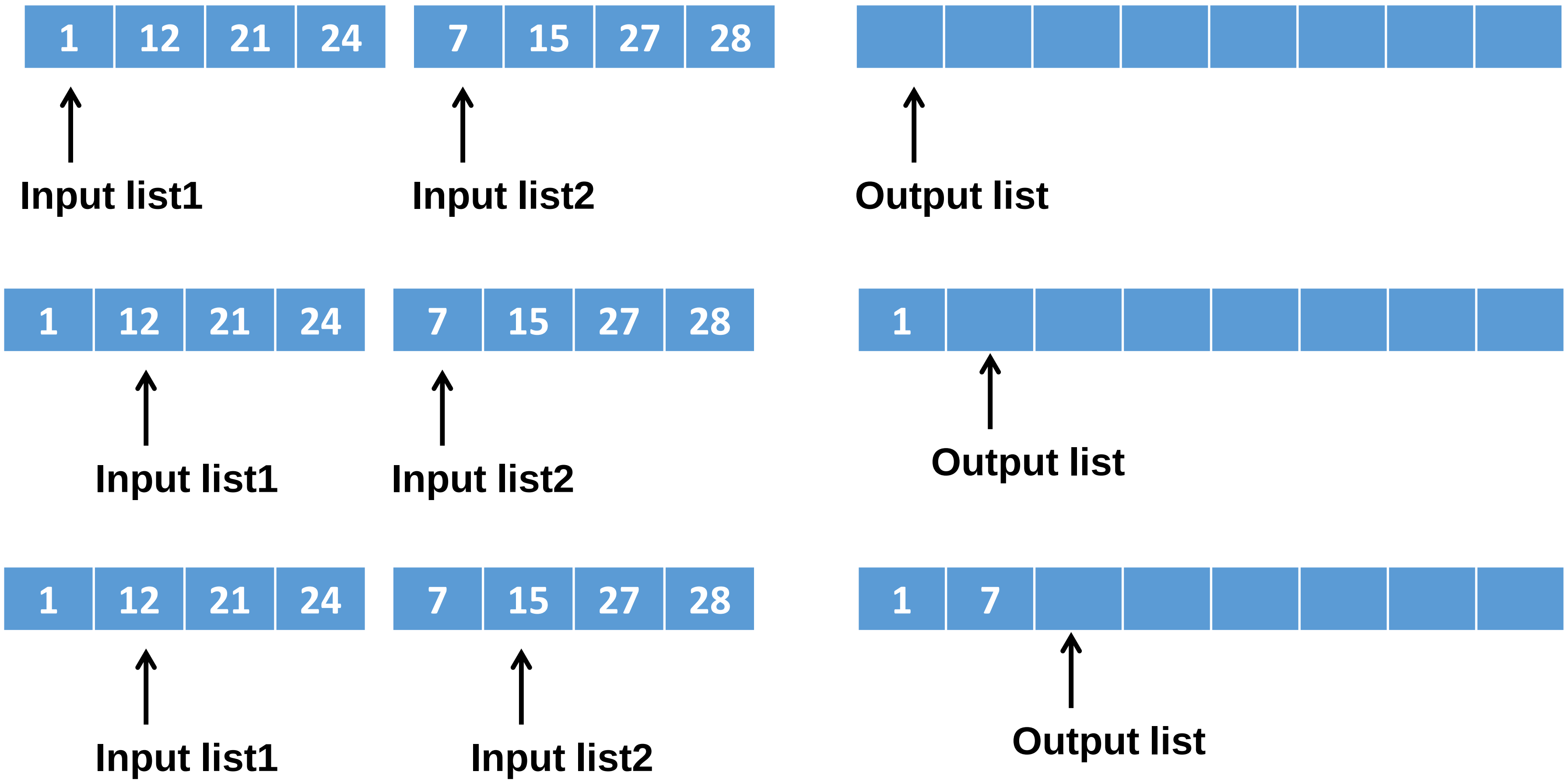
- **Merging** is a process whereby we combine two (or more) **sorted arrays** into a single sorted array
  - Start with a number of sorted (INPUT) arrays and create an OUTPUT array that is also sorted
- The Output array contains the values from the various Input arrays

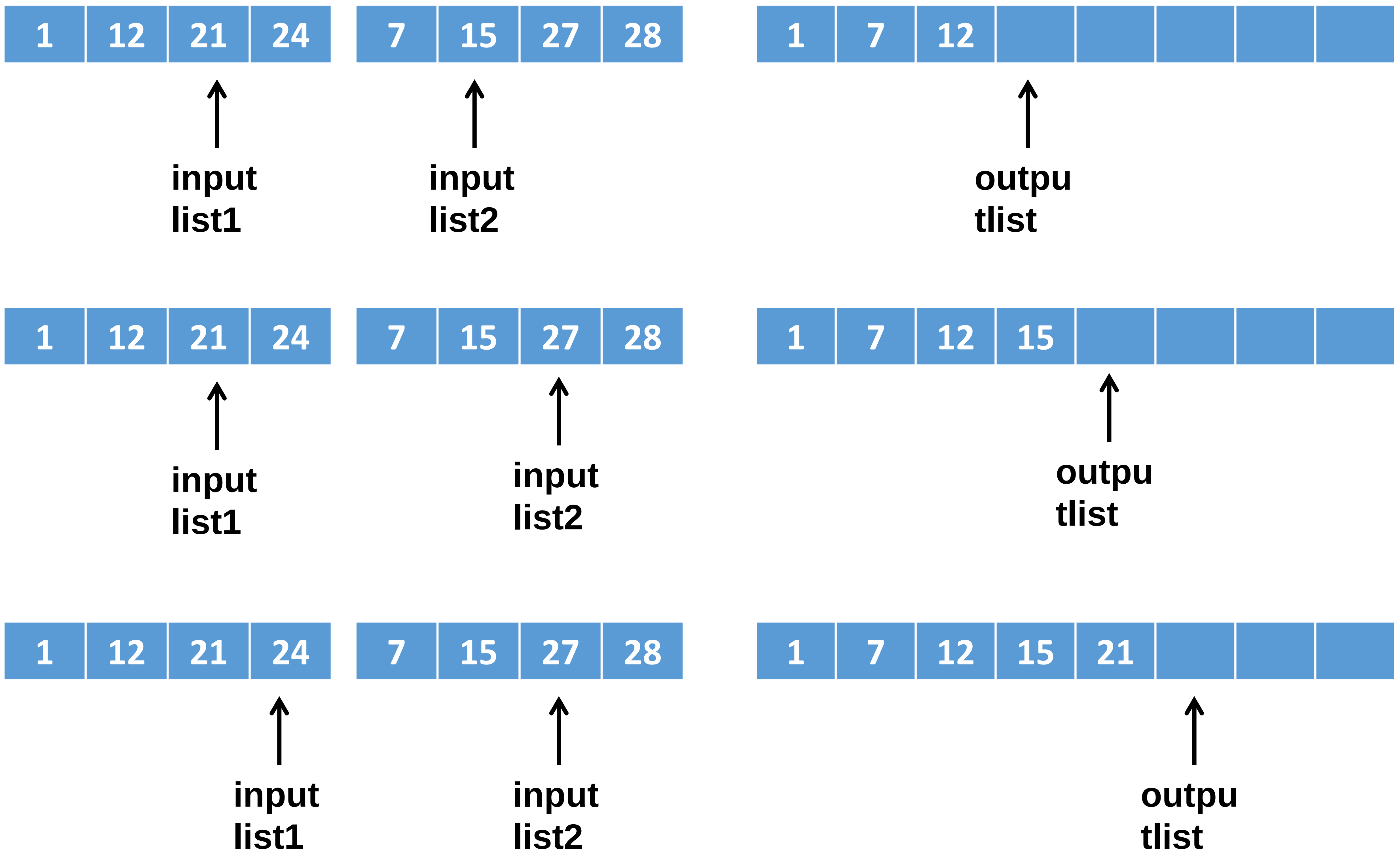


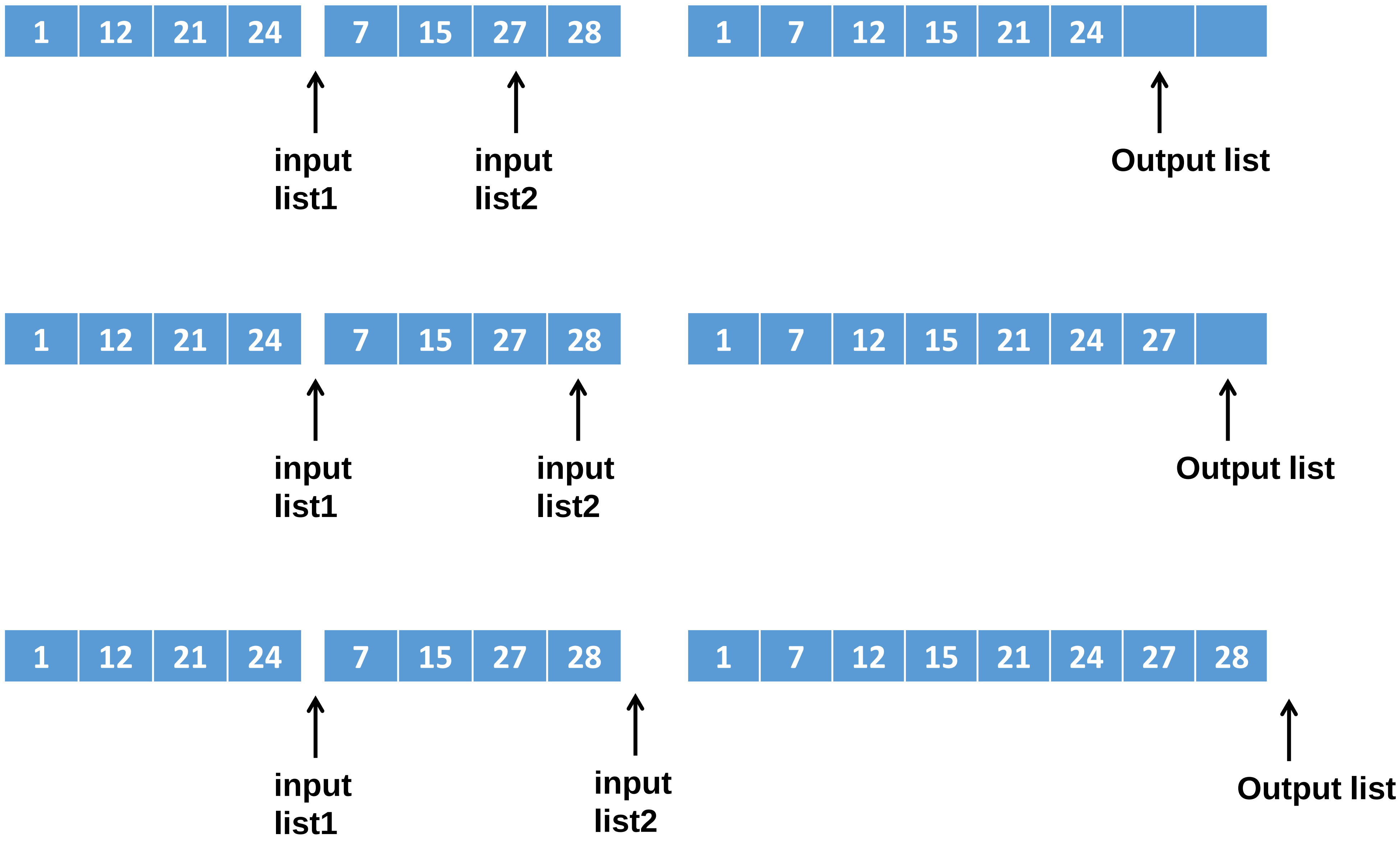
Merging Two Sorted Arrays

- Set **markers** at the 1st items of the 2 sorted (input) arrays AND the output array (the final SORTED array)
- COMPARE the 2 items at the markers of the input arrays to see which of them should be placed into the output array
- Place the identified item within the output array AND THEN ADVANCE the marker in the array from which the item was taken
- Advance the pointer in the output array
- Repeat the comparison until one of the input arrays is exhausted
- COPY all the remaining items from the array which is not exhausted into the output array to complete the operation

Merging - Example

- 


1 12 21 24
7 15 27 28
1 7 12
input list1
input list2
outpu tlist
1 12 21 24
7 15 27 28
1 7 12 15
input list1
input list2
outpu tlist
1 12 21 24
7 15 27 28
1 7 12 15 21
input list1
input list2
outpu tlist


1 12 21 24
7 15 27 28
1 7 12 15 21 24
input list1
input list2
Output list
1 12 21 24
7 15 27 28
1 7 12 15 21 24 27
input list1
input list2
Output list
1 12 21 24
7 15 27 28
1 7 12 15 21 24 27 28
input list1
input list2
Output list

Algorithm - Merging

```
WHILE ( (leftIndex <= leftEnd) AND (rightIndex <= rightEnd))
    IF (left [leftIndex] < right [rightIndex])
        answer [answerIndex] = left [leftIndex]
        Add 1 to answerIndex
        Add 1 to leftIndex
    ELSE
        answer [answerIndex] = right [rightIndex]
        Add 1 to answerIndex
        Add 1 to rightIndex
WHILE (leftIndex <= leftEnd)
        answer [answerIndex] = left [leftIndex]
        Add 1 to answerIndex
        Add 1 to leftIndex
WHILE (rightIndex <= rightEnd)
        answer [answerIndex] = right [rightIndex]
        Add 1 to answerIndex
        Add 1 to rightIndex
```

```
int leftIndex = 0;
int leftEnd = left.length – 1;
int rightIndex = 0;
int rightEnd = right.length – 1;
int answerIndex = 0;
```

left	1	3	5	7

right	2	4	5

answer	1	2	3	4	5	5	7

Algorithm - Merging

myArray	45	2	4	6	3	5	1
		left Index			right Index	right End	

merge (int [] anyArray, int leftIndex, int rightIndex, int rightEnd)

merge (myArray, 1, 4, 5);

myArray	45	2	3	4	5	6	1

Algorithm - Merging

anyArray	45	2	4	6	3	5	1

merge (myArray, 1, 4, 5);

leftIndex = 1; rightIndex = 4; rightEnd = 5;

support							

```
WHILE ( (leftIndex <= leftEnd) AND (rightIndex <= rightEnd))
      IF (anyArray [leftIndex] < anyArray [rightIndex])
            support  [tempPos] = anyArray [leftIndex]
            Add 1 to tempPos
            Add 1 to leftIndex
      ELSE
            support [tempPos] = anyArray [rightIndex]
            Add 1 to tempPos
            Add 1 to rightIndex
```

```
int leftEnd = rightIndex – 1;

int [] support = new int
      [anyArray.length]
int tempPos = leftIndex;
```

support		2	3	4	5		

Algorithm - Merging

WHILE (leftIndex <= leftEnd)
    support [tempPos] = anyArray [leftIndex]
    Add 1 to tempPos
    Add 1 to leftIndex

WHILE (rightIndex <= rightEnd)
    support [tempPos] = anyArray [rightIndex]
    Add 1 to tempPos
    Add 1 to rightIndex

Copy relevant part of support array back
FOR (int count = 0; count < numElements; count++, rightEnd--)
    anyArray [rightEnd] = support [rightEnd]

anyArray	45	2	4	6	3	5	1

merge (myArray, 1, 4, 5);

leftIndex = 1; rightIndex = 4; rightEnd = 5;

support		2	3	4	5	6	

anyArray	45	2	3	4	5	6	1

int numElements =
    rightEnd – leftIndex + 1;

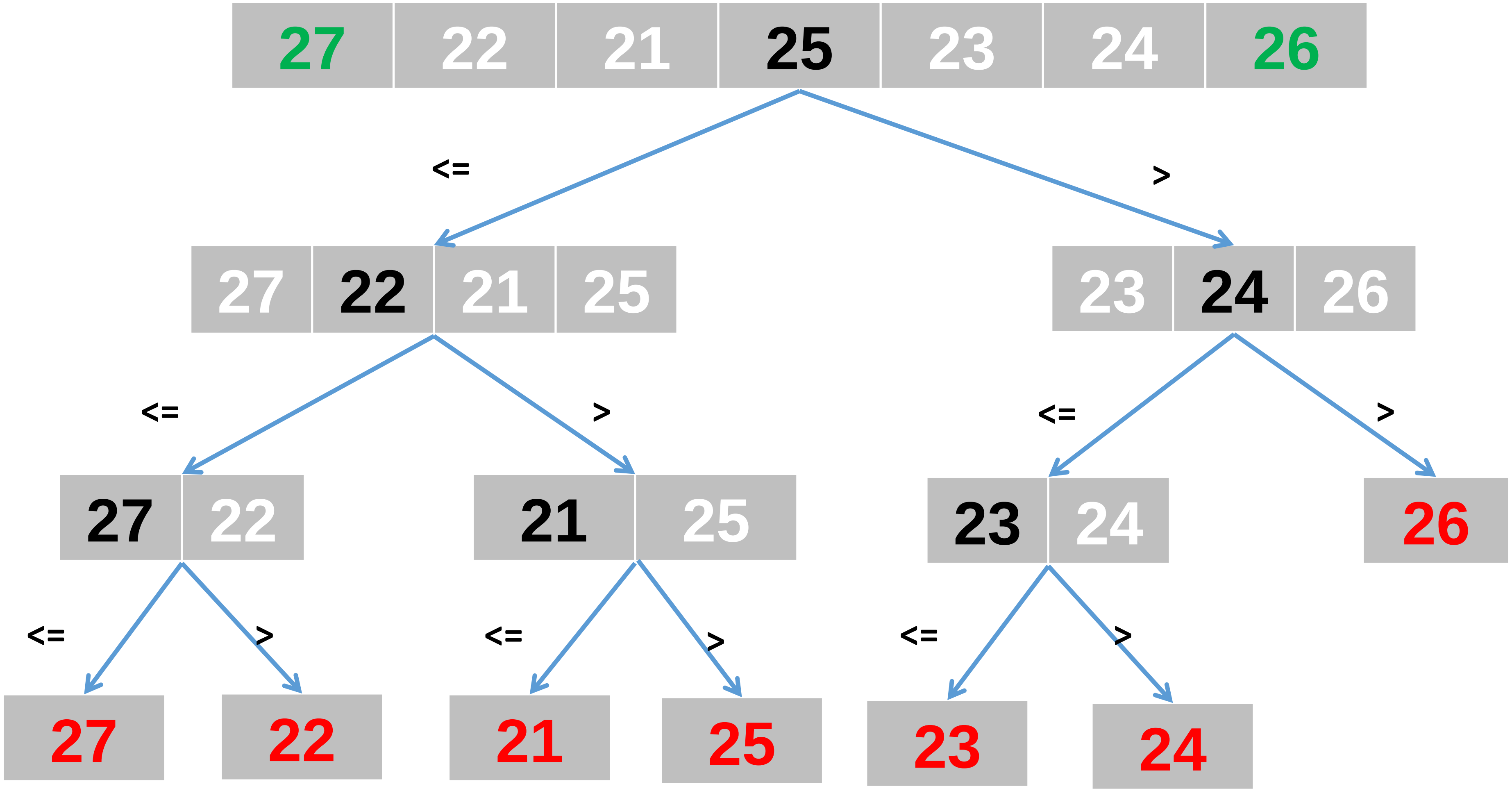
MergeSort

- Assume we are asked to sort an array into ascending order
    - ➢The **MergeSort** of an array is based on the following observations:
- We can split the array into 2 “halves”
- We can sort each half into ascending order
- It is then easy to **merge** the two sorted halves into a single sorted array
    - We simply keep comparing the bottom element in each half and move the smallest into a new array
- Identical principles apply for the descending order problem

MergeSort

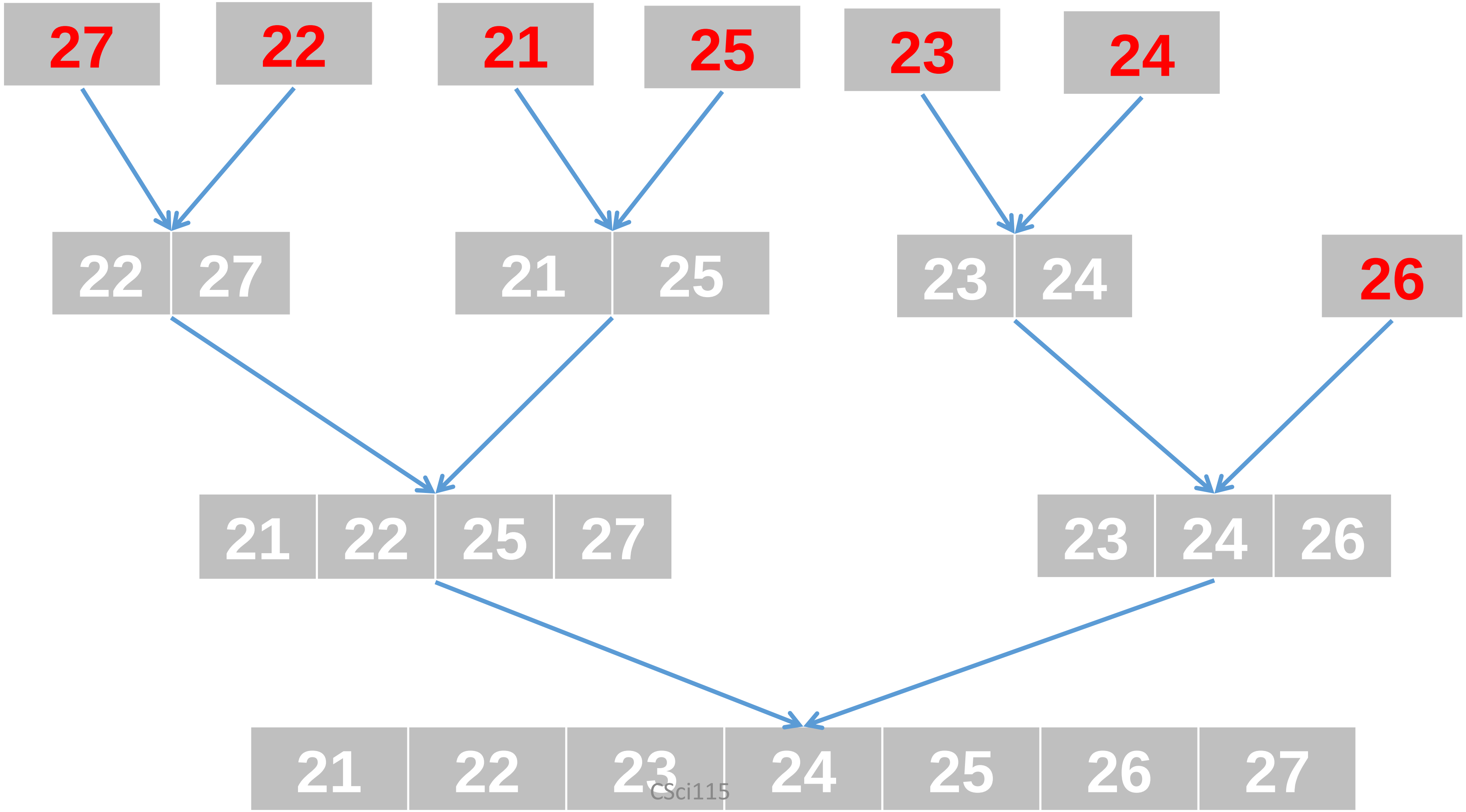
- The MergeSort algorithm involves 3 steps
  1. If the number of items to sort in an array is either 0 or 1 – the array is (by definition) in order - **FINISHED**
  2. We **recursively** sort the first and second halves separately
  3. Merge the two sorted halves into a sorted group
- The MergeSort algorithm is an **O (N log N)** algorithm

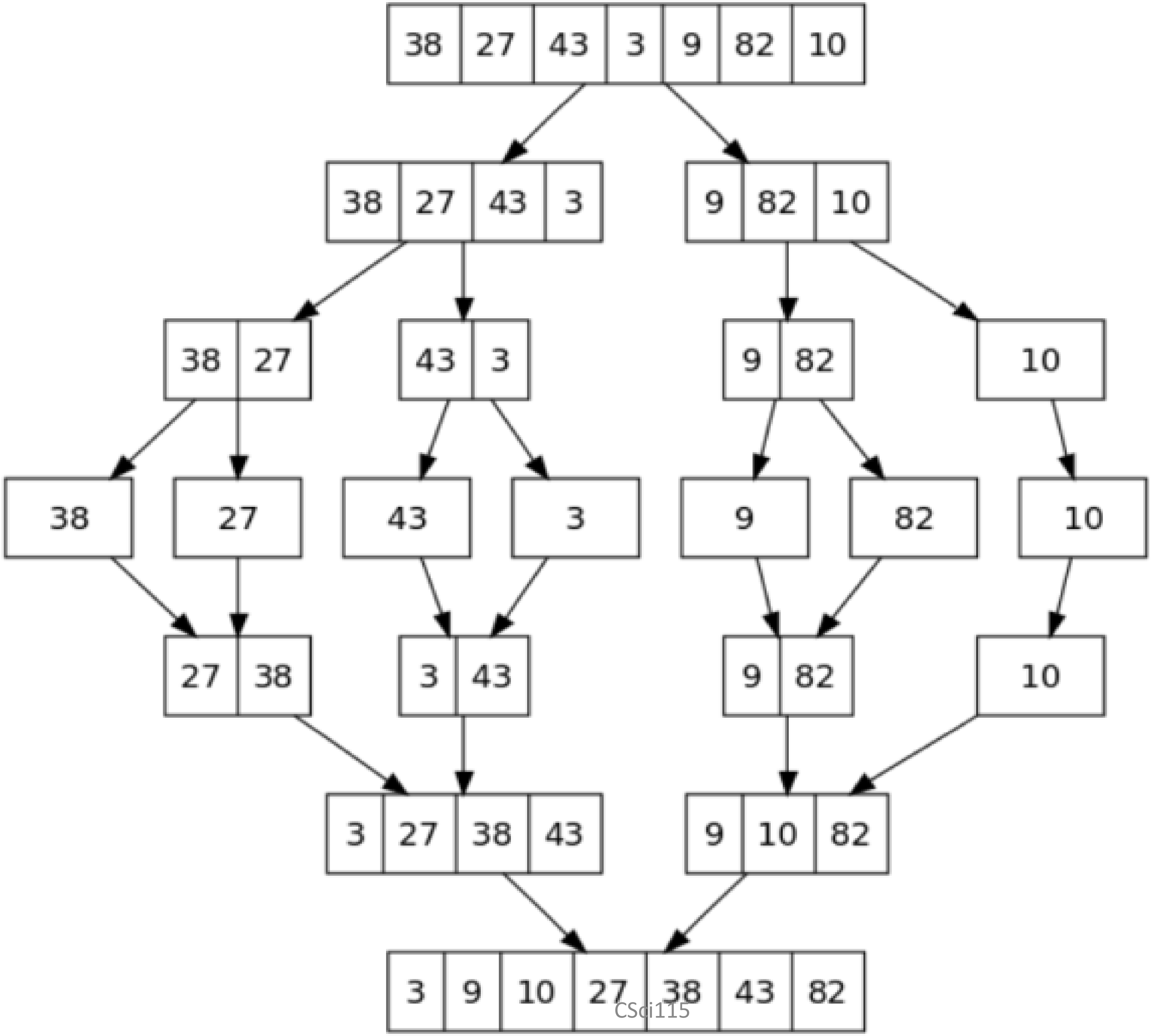
MergeSort

- 

27 22 21 25 23 24 26
<=
>
27 22 21 25
23 24 26
<=
>
<=
>
27 22
21 25
23 24
26
<=
>
<=
>
<=
>
27
22
21
25
23
24

MergeSort (ordering as we merge)

- 


38 27 43 3 9 82 10
38 27 43 3
9 82 10
38 27
43 3
9 82
10
38
27
43
3
9
82
10
27 38
3 43
9 82
10
3 27 38 43
9 10 82
3 9 10 27 38 43 82

MergeSort

- Pseudo-code

```
MergeSort(array A, int p, int r) {
    if (p < r) {                // we have at least 2 items
        q = (p + r)/2
        MergeSort(A, p, q)      // sort A[p..q]
        MergeSort(A, q+1, r)    // sort A[q+1..r]
        Merge(A, p, q, r)       // merge everything together
    }
}
```

- Analysis
  - Master theorem (case 2, c=1) → O(n log n)

$$T(n) = \begin{cases} 1 & \text{if } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + n & \text{otherwise.} \end{cases}$$

QuickSort

- Fastest known sorting algorithm
  - Average running time is: **O (N log N)**
  - Based on a “Divide and Conquer” approach
- Steps
  - We select a “pivot” value:
    - /!\ it is **NOT** the same as the value in the middle of the array!!
  - We use this “pivot” value to generate 2 sub-lists
    - List 1: Values less than the pivot
    - List 2: Values greater than or equal to the pivot
  - We sort the 2 sub-arrays and note that we can have the whole array sorted by arranging the components thus:

    sorted sub-list 1, sorted sub-list 2

QuickSort

- Sorting of each of the ‘sub-arrays’ is once again performed in a recursive manner using the QuickSort technique
- The process continues until each sub-array contains only a single element

Pivot Selection

- ANY VALUE can be selected as the pivot.
- BETTER – to make a ‘educated’ choice

OPTION 1. POPULAR CHOICE

- Use the FIRST element in the list
  - Simple
  - Acceptable if the input is random
  - However, if data is (say) pre-sorted or in reverse order this gives us a poor partitioning
  - NOT NORMALLY RECOMMENDED

Pivot Selection

OPTION 2. SAFE CHOICE

- Use the MIDDLE element
  - i.e. the element at position (low + high) / 2
  - This is a 'passive' choice
    - we do not try to choose a 'good' pivot - we are merely trying to avoid picking a bad pivot
- **OPTION 3. MEDIAN-OF-THREE partitioning**
  - Attempts to pick a "better than average" pivot
  - Pick the MEDIAN of the *first*, *middle* and *last* elements
  - Example:

    With input: **8**, 11, 14, 19, **16**, 13, 15, 12, 17, **10**

    These three elements are: **8**, **10** and **16**

    The value **10** is chosen as the pivot

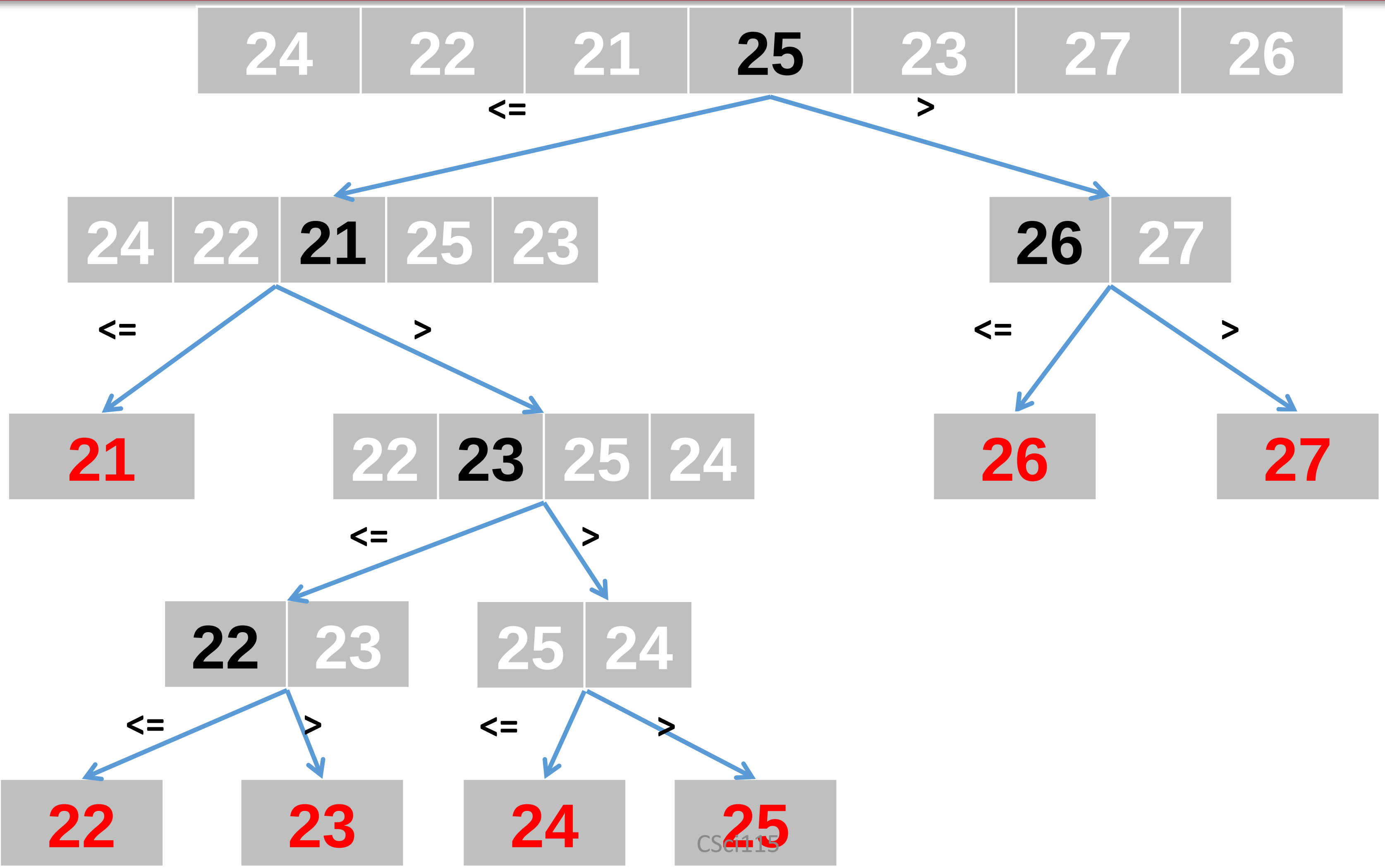
Partition function

- Rearrange the array A[p..r] into 2 (possibly empty) subarrays.
  - From index p to index r
- 2 arrays as outputs: A[p..q-1] and A[q+1.. r] such that each element of A[p..q-1] is less than or equal to A[q] (the pivot), which is, in turn, less than or equal to each element of A[q+1..r].
- The index q is evaluated as part of this partitioning procedure.

- Partition function
  - Input: Array A, int p, int r
  - Output: Array A, position of q,

QuickSort: Using middle element as pivot

▪



QuickSort: Using middle element as pivot

	24	22	21	**25**	23	27	26	

Initial pivotValue is 25

SWAP into last location	24	22	21	26	23	27	**25**	
Position pointers	24	22	21	26	23	27	**25**	
Advance to get a value > pivotValue	24	22	21	26	23	27	**25**	Retreat to get a value < pivotValue
SWAP	24	22	21	**25**	23	27	26	
ADVANCE	24	22	21	**25**	23	27	26	RETREAT

QuickSort: Using middle element as pivot

24	22	21	**25**	23	27	26

Advance to get a value > pivotValue

24	22	21	**25**	23	27	26

Retreat to get a value < pivotValue

ARROWS HAVE CROSSED SO FINISHED THIS PHASE

Quicksort

```
QUICKSORT(int[] A, int p, int r)
{
    If (p < r)
    {
        q=PARTITION(A,p,r)   // return q and update A
        QUICKSORT(A,p,q-1) // apply on the first part of A (all the elements<A[q])
        QUICKSORT(A,q+1,r) // apply on the second part of A (all the elements>A[q])
    }
}
```

- where
  - A: the array
  - p: index of the first element
  - r: index of the last element,
  - q: **index of the pivot** returned by the partition function. (index, not the value !!!)

QuickSort: Performance

- Average performance of QuickSort for both swaps and comparisons
  - **Complexity:** **O(N log N)**
- Major problem with QuickSort:
  - Under certain conditions the partition phase fails to divide the list at all
- Therefore
  - the QuickSort algorithm is far from quick in the <u>worst case</u> performance, degenerating to an $\mathbf{O(N^2)}$ algorithm

Summary of Sorting

- Selection, Insertion and Bubble Sorts are O (N^2) algorithms
  - Not so good
- MergeSort and QuickSort are O (N log N) for random lists
  - Quite Good
- This does **not** mean that MergeSort or QuickSort are **always** best for all your sorting problems
  - if we introduce immediate checking to see whether the list provided is already ordered the Insertion sort is one of the best algorithms
  - For data that is actually initially sorted, the QuickSort algorithm is one of the worst algorithms

Finally…

- If the elements of the array contain large data records, movement of this data is very expensive
- If this is the case, the Selection Sort (which involves few movements) is probably the <u>best</u> since is it linear **O(N)** for assignments.
    - Question: What is your view on this for the Bubble Sort?
        - Memory efficient (just swap values)
- If the data is stored as a linked list (see later) rather than an array, the MergeSort is one of the most efficient algorithms for sorting linked lists

Questions ?

- Midterm question
  - Determine the initial conditions of an array so …
    - It corresponds to the worst case of a given algorithm
    - It corresponds to the best case of an algorithm
- Reading
  - See code in the CSci115 book