# Advanced Algorithms Analysis and Design

Lecture 10

## Today’s Lectures

Sorting algorithms

- Insertion Sort
- Insertion sort Analysis
- Radix Sort
- Merge Sort

## Insertion Sort

- Real life example:
  - An example of an insertion sort occurs in everyday life while playing cards. To sort the cards in your hand you extract a card, shift the remaining cards, and then insert the extracted card in the correct place. This process is repeated until all the cards are in the correct sequence.

## Insertion Sort algorithm

Insertion sort pseudocode (recall)

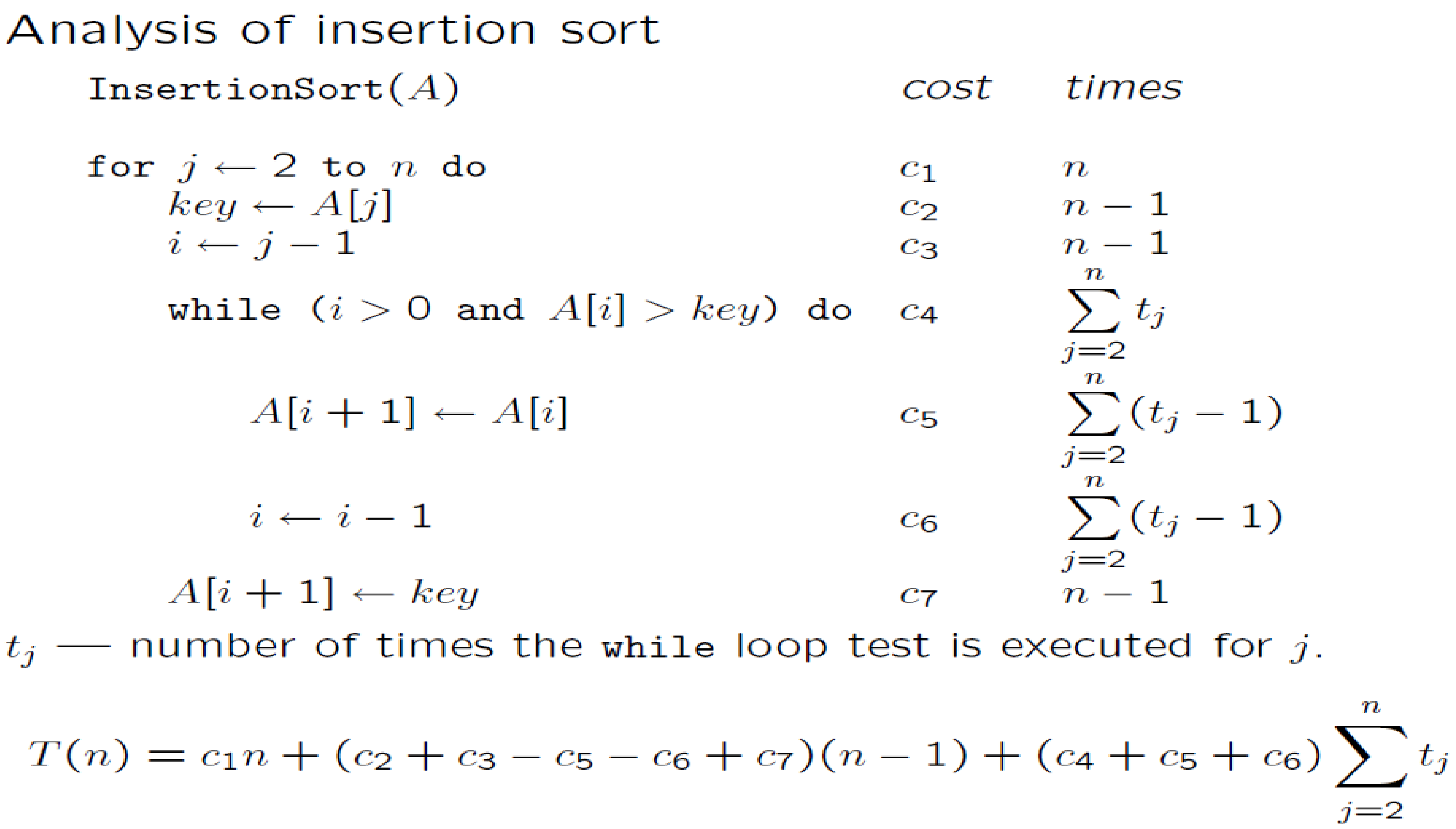
```
InsertionSort($A$) **sort $A[1..n]$ in place

for $j \leftarrow 2$ to $n$ do
    $key \leftarrow A[j]$ **insert $A[j]$ into sorted sublist $A[1..j-1]$
    $i \leftarrow j-1$
    while ($i > 0$ and $A[i] > key$) do
        $A[i+1] \leftarrow A[i]$
        $i \leftarrow i-1$
    $A[i+1] \leftarrow key$
```

5 7 0 3 4 2 6 1 (0)

5 7 $\underline{0}$ 3 4 2 6 1 (0)
**0** 5 7 $\underline{3}$ 4 2 6 1 (2)
0 **3** 5 7 $\underline{4}$ 2 6 1 (2)
0 3 **4** 5 7 $\underline{2}$ 6 1 (2)
0 **2** 3 4 5 7 $\underline{6}$ 1 (4)
0 2 3 4 5 **6** 7 $\underline{1}$ (1)
0 **1** 2 3 4 5 6 7 (6)

# Insertion Sort Anlysis



Analysis of insertion sort

| `InsertionSort(`$A$`)` | *cost* | *times* |
|---|---|---|
| `for` $j \leftarrow 2$ `to` $n$ `do` | $c_1$ | $n$ |
|     $key \leftarrow A[j]$ | $c_2$ | $n-1$ |
|     $i \leftarrow j-1$ | $c_3$ | $n-1$ |
|     `while` $(i > 0$ `and` $A[i] > key)$ `do` | $c_4$ | $\sum_{j=2}^{n} t_j$ |
|         $A[i+1] \leftarrow A[i]$ | $c_5$ | $\sum_{j=2}^{n} (t_j - 1)$ |
|         $i \leftarrow i-1$ | $c_6$ | $\sum_{j=2}^{n} (t_j - 1)$ |
|     $A[i+1] \leftarrow key$ | $c_7$ | $n-1$ |

$t_j$ — number of times the `while` loop test is executed for $j$.

$$T(n) = c_1 n + (c_2 + c_3 - c_5 - c_6 + c_7)(n-1) + (c_4 + c_5 + c_6) \sum_{j=2}^{n} t_j$$

# Insertion Sort Anlysis

- Best case (BC)
  - What is the best case?
    already sorted
  - One KC for each $j$, and so $\sum_{j=2}^{n} 1 = n - 1$

Best case (already sorted array): `tj = 1` for all j

→ While loop never enters, `O(n)` time

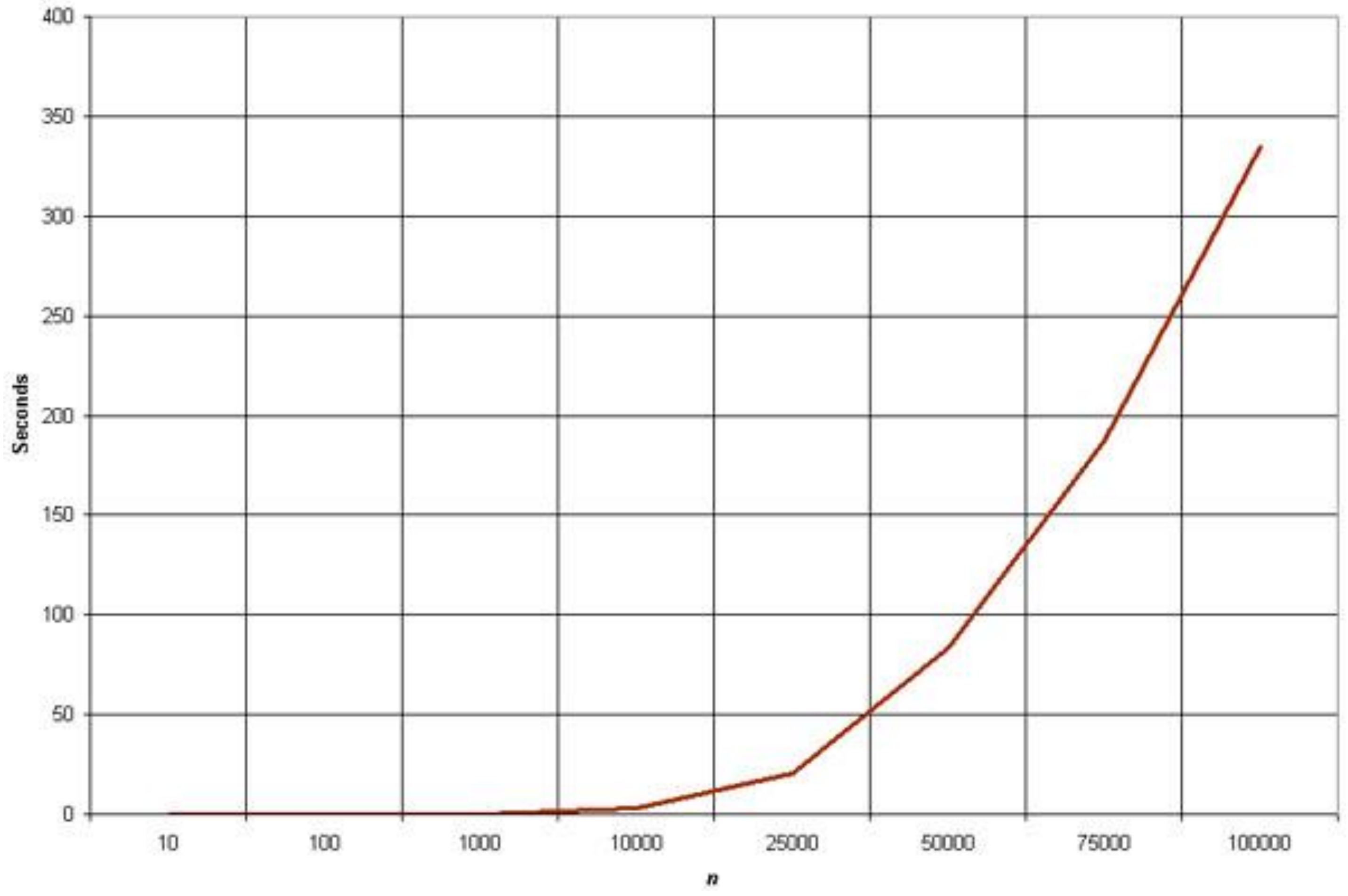
- Worst case (WC)
  - What is the worst case?
    reverse sorted
  - $j$ KC for fixed $j$, and so $\sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$
- Average case (AC)

Quadratic running time (i.e., O($n^2$))

# Insertion Sort

- Since we do not necessarily have to scan the entire sorted section of the array, the best, worst, and average cases for insertion sort all differ!
- ***Best case*****:** The best case input is an array that is already sorted. In this case insertion sort has a linear running **time (i.e., Θ($n$)).**
  - During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.
- ***Worst case:*** **T**he simplest worst case input is an array sorted in reverse order. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O($n^2$)).

# Empirical Analysis of Insertion Sort



The graph demonstrates the *n*^2 complexity of the insertion sort.

## Insertion Sort

- **Advantage:** The insertion sort is a good choice for sorting lists of a few thousand items or less.
- **Disadvantage:**The insertion sort shouldn't be used for sorting lists larger than a couple thousand items or repetitive sorting of lists larger than a couple hundred items

## Radix Sort

- An algorithm that is capable of sorting data in linear O(n) time is radix sort and the domain of the input is restricted – it must consist only of integers.

## Radix Sort

- It is used to sort numbers using 10 buckets for decimal numbers ranging 0 - 9.
- The given numbers are first sorted according to the unit digit and each number is placed in the concerned bucket, and then the buckets are combined in order before distributing according to the tens digit.
- This sorting process continues to the last digit (i.e. the most significant bit) in d-passes for d-digit numbers.
- For a 5- digit numbers, radix sort places numbers in the bucket in 5 passes and combines the buckets five times.
- Radix sort is sometimes used to sort records of information that are keyed by multiple fields (i.e. to sort dates by year, month and day).
- Number of comparisons (Cn) = d*s*n

Where d = Digits in a number (d=10 for decimal digit)
s = Number of digits in a number (s = 4 for 972, 8345 & 89 numbers)
n = Number of items (given numbers to be sorted)

## Radix Sort

Other names: *postal sort*, *bin sort*

Limit input to fixed-length numbers or words.

Represent symbols in some base *b*.

- Each input has exactly *d* “digits”.

Sort numbers *d* times, using 1 digit as key.

- Must sort from least-significant to most-significant digit.

## Radix Sort Algorithm

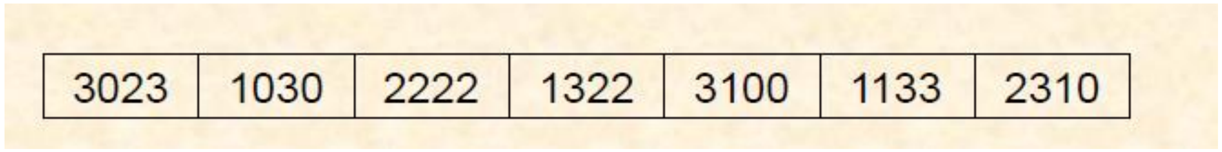
- Suppose keys are *k-digit* integers
- Radix sort uses an array of 10 queues, one for each digit 0 through 9
- Each object is placed into the queue whose index is the least significant digit (the 1's digit) of the object's key
- Objects are then dequeued from these 10 queues, in order 0 through 9, and put back in the original queue/list/array container; they're sorted by the last digit of the key

# Radix Sort Algorithm

- Process is repeated, this time using the 10’s digit instead of the 1’s digit; values are now sorted by last two digits of the key
- Keep repeating, using the 100’s digit, then the 1000’s digit, then the 10000’s digit, …
- Stop after using the most significant (10n-1’s ) digit
- Objects are now in order in original container

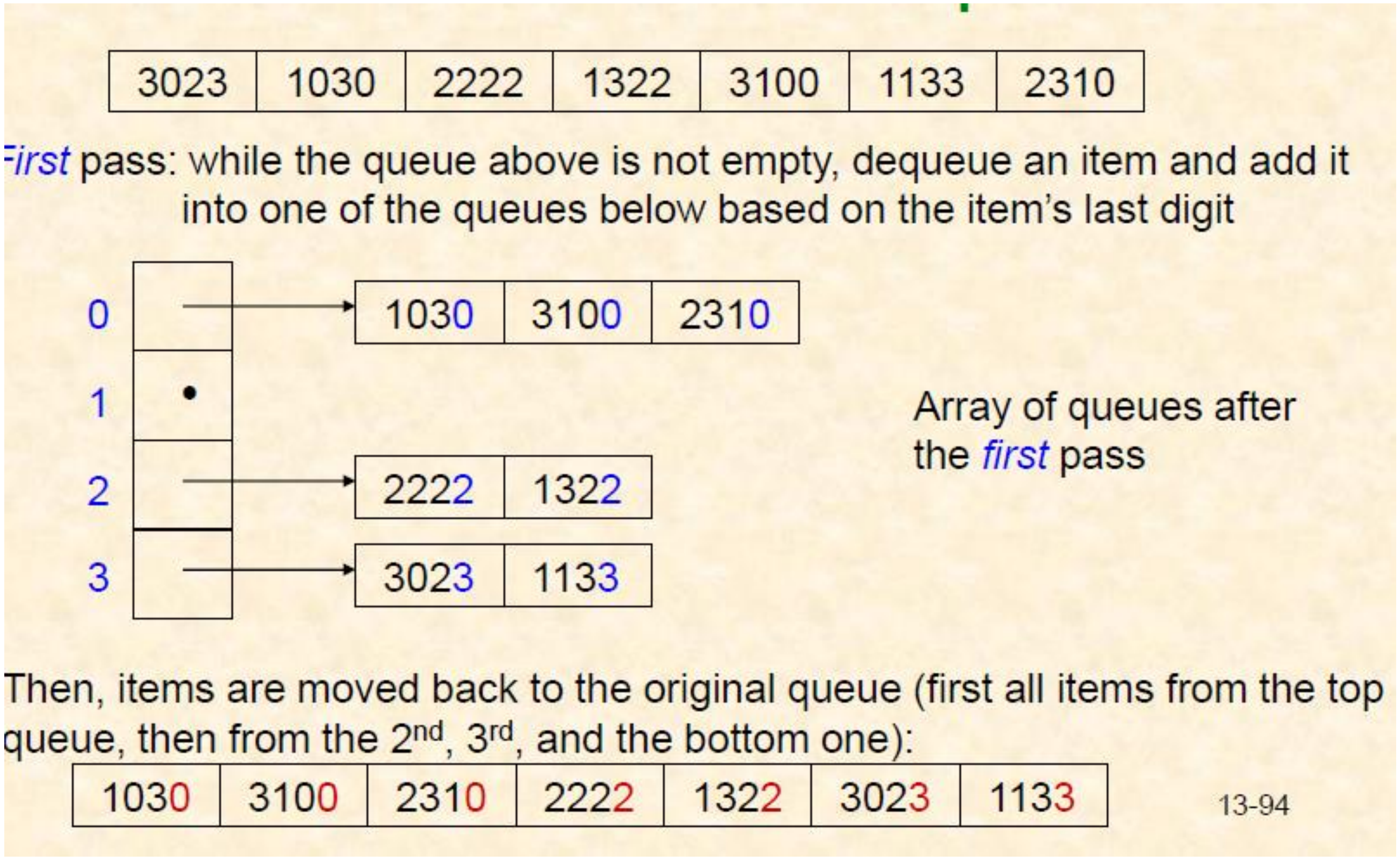
## Radix Sort Example

- Suppose keys are 4-digit numbers using only the digits 0, 1, 2 and 3, and that we wish to sort the following queue of objects whose keys are shown:



|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| 3023 | 1030 | 2222 | 1322 | 3100 | 1133 | 2310 |

# Radix Sort Example



| | | | | | | |
|---|---|---|---|---|---|---|
| 3023 | 1030 | 2222 | 1322 | 3100 | 1133 | 2310 |

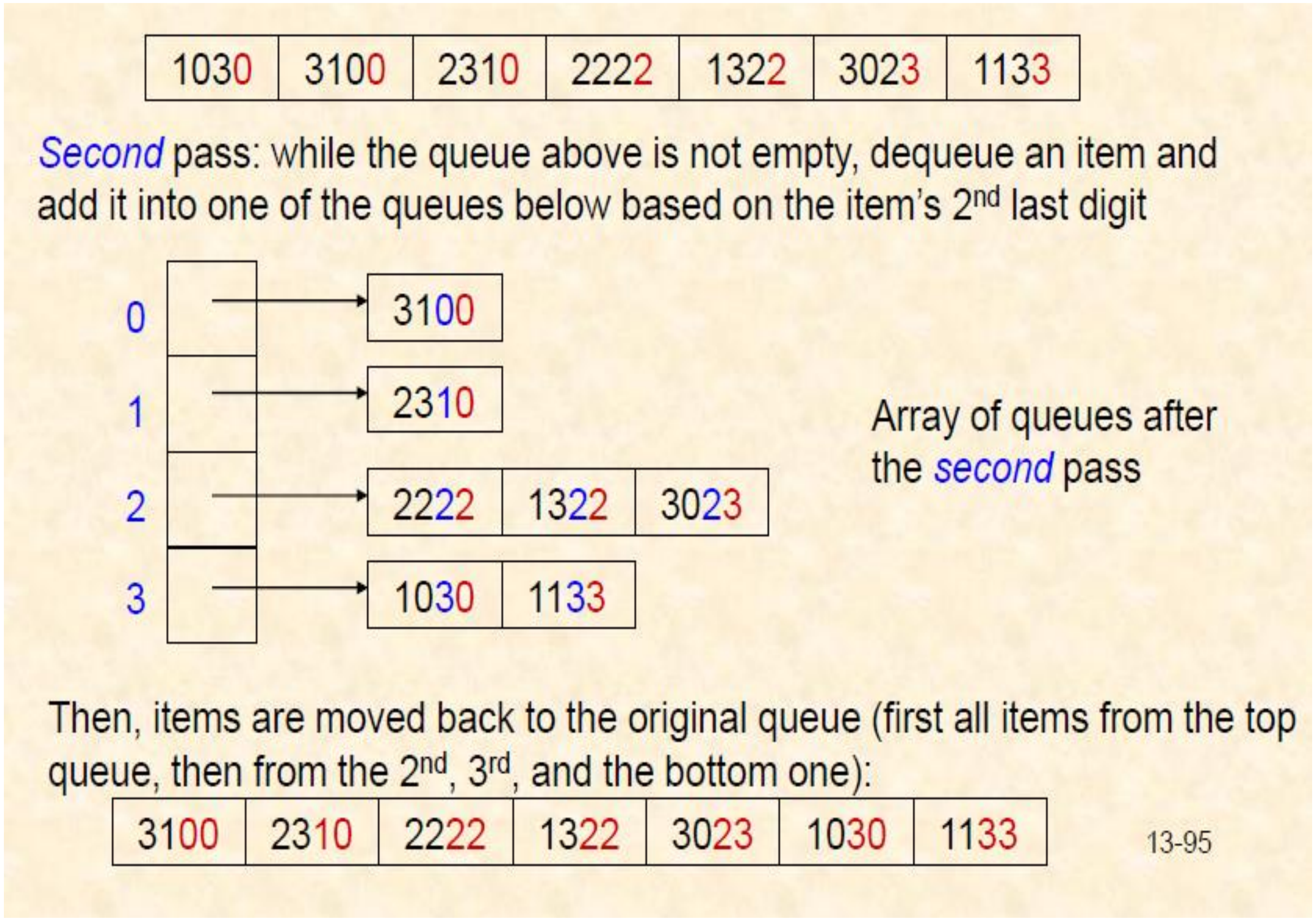
*irst* pass: while the queue above is not empty, dequeue an item and add it into one of the queues below based on the item's last digit

Array of queues after the *first* pass

Then, items are moved back to the original queue (first all items from the top queue, then from the 2nd, 3rd, and the bottom one):

| | | | | | | |
|---|---|---|---|---|---|---|
| 1030 | 3100 | 2310 | 2222 | 1322 | 3023 | 1133 |

# Radix Sort Example



| | | | | | | |
|---|---|---|---|---|---|---|
| 1030 | 3100 | 2310 | 2222 | 1322 | 3023 | 1133 |

*Second* pass: while the queue above is not empty, dequeue an item and add it into one of the queues below based on the item's 2nd last digit

| | | | |
|---|---|---|---|
| 0 | 3100 | | |
| 1 | 2310 | | |
| 2 | 2222 | 1322 | 3023 |
| 3 | 1030 | 1133 | |

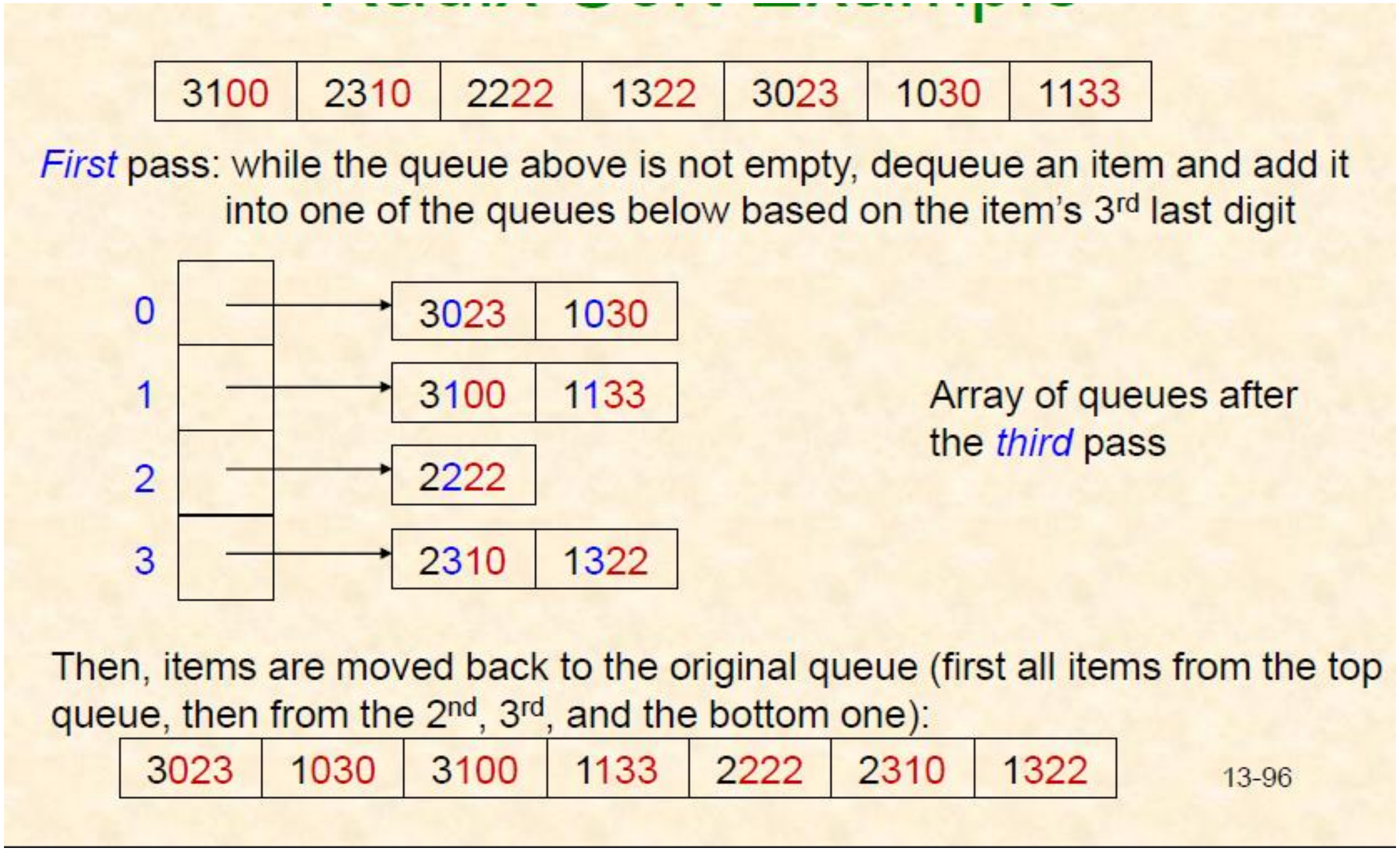
Array of queues after the *second* pass

Then, items are moved back to the original queue (first all items from the top queue, then from the 2nd, 3rd, and the bottom one):

| | | | | | | |
|---|---|---|---|---|---|---|
| 3100 | 2310 | 2222 | 1322 | 3023 | 1030 | 1133 |

13-95

# Radix Sort Example



| | | | | | | |
|---|---|---|---|---|---|---|
| 3100 | 2310 | 2222 | 1322 | 3023 | 1030 | 1133 |

*First* pass: while the queue above is not empty, dequeue an item and add it into one of the queues below based on the item's 3rd last digit

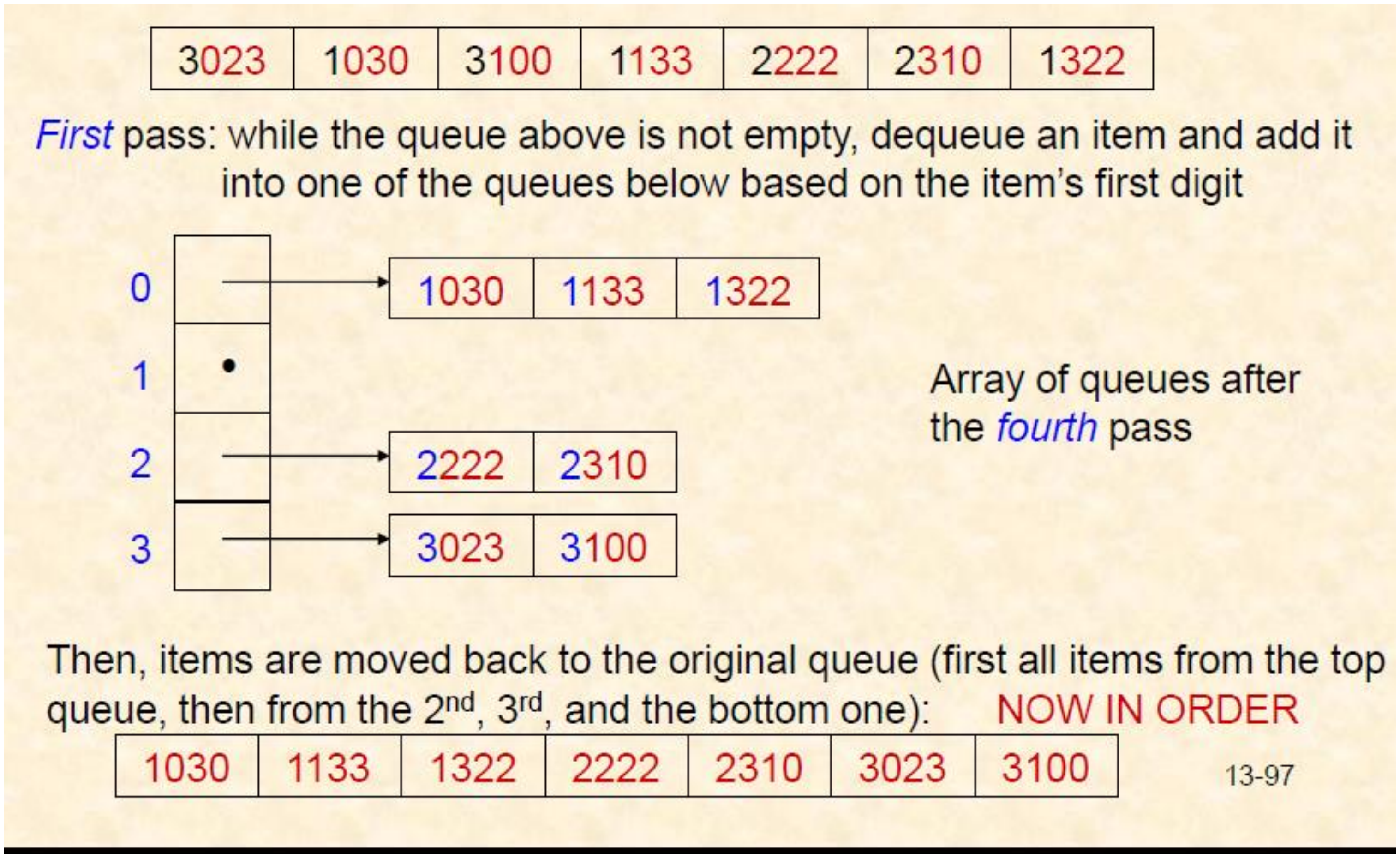
| | | |
|---|---|---|
| 0 | 3023 | 1030 |
| 1 | 3100 | 1133 |
| 2 | 2222 | |
| 3 | 2310 | 1322 |

Array of queues after the *third* pass

Then, items are moved back to the original queue (first all items from the top queue, then from the 2nd, 3rd, and the bottom one):

| | | | | | | |
|---|---|---|---|---|---|---|
| 3023 | 1030 | 3100 | 1133 | 2222 | 2310 | 1322 |

# Radix Sort Example



| | | | | | | |
|---|---|---|---|---|---|---|
| 3023 | 1030 | 3100 | 1133 | 2222 | 2310 | 1322 |

*First* pass: while the queue above is not empty, dequeue an item and add it into one of the queues below based on the item's first digit

Array of queues after the *fourth* pass

Then, items are moved back to the original queue (first all items from the top queue, then from the 2nd, 3rd, and the bottom one): NOW IN ORDER

| | | | | | | |
|---|---|---|---|---|---|---|
| 1030 | 1133 | 1322 | 2222 | 2310 | 3023 | 3100 |

13-97

## Analysis of radix sort

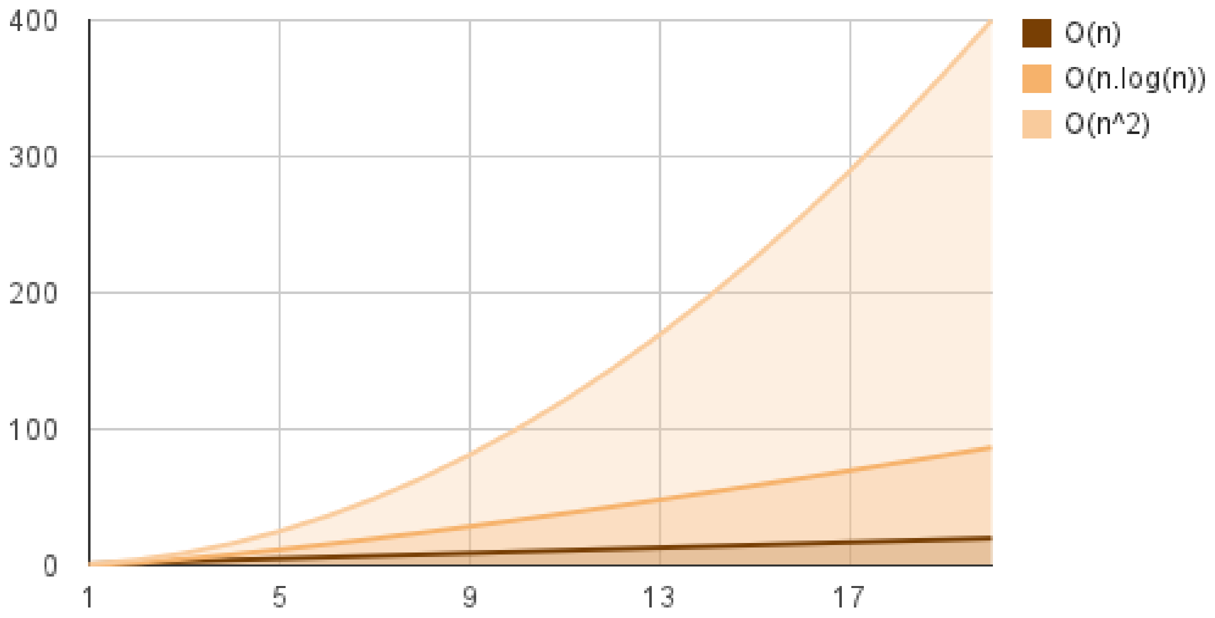
- *Consider n items to b sorted and each item has k digits. Then it requires k number of passes. And in each pass n operations are performed.*
- $\Theta(k \times n)$ time, where $k$ is taken to be a constant.
- ***Radix sort is capable of sorting data in linear time***

## Numerical

List of numbers:
48081, 90342, 90287, 90583, 53202, 65215, 78397, 48001, 972, 65315, 41983, 90283, 81664, 38107

| Unsorted list | bkt | 1st Pass | bkt | 2nd Pass | bkt | 3rd Pass | bkt | 4th Pass | bkt | 5th Pass | Sorted list |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 48081 | 0 | | 0 | 48001 | 0 | 48001 | | 90283 | 0 | 00972 | 00972 |
| 97342 | 1 | 48081 | | 53202 | | 48081 | 0 | 90287 | 1 | | 38107 |
| 90287 | | 48001 | | 38107 | 1 | 38107 | | 90583 | 2 | | 41983 |
| 90583 | | 97342 | 1 | 65215 | | 53202 | | 00972 | 3 | 38107 | 48001 |
| 53202 | 2 | 53202 | | 65315 | 2 | 65215 | 1 | 81664 | | 41983 | 48081 |
| 65215 | | 00972 | 2 | | | 90283 | | 41983 | 4 | 48001 | 53202 |
| 78397 | | 90583 | 3 | | | 90287 | 2 | | | 48081 | 65215 |
| 48001 | 3 | 41983 | 4 | 97342 | | 65315 | 3 | 53202 | 5 | 53202 | 65315 |
| 00972 | | 90283 | 5 | | 3 | 97342 | 4 | | 6 | 65215 | 78397 |
| 65315 | 4 | 81664 | 6 | 81664 | | 78397 | 5 | 65215 | | 65315 | 81664 |
| 41983 | 5 | 65215 | 7 | 00972 | 4 | | | 65315 | 7 | 78397 | 90283 |
| 90283 | | 65315 | | 48081 | 5 | 90583 | 6 | | 8 | 81664 | 90287 |
| 81664 | 6 | | | 90583 | 6 | 81664 | 7 | 97342 | | 90283 | 90583 |
| 38107 | | 90287 | 8 | 41983 | 7 | | | 48001 | 9 | 90287 | 97342 |
| | 7 | 78397 | | 90283 | 8 | | 8 | 48081 | | 90583 | |
| | | 38107 | | 90287 | 9 | 00972 | | 38107 | | 97342 | |
| | 8 | | 9 | 78397 | | 41983 | | 78397 | | | |
| | 9 | | | | | | 9 | | | | |

# Complexity Comparison



Linear function compared to n.log(n) and n^2

## Radix Sort

- Advantages:
  - It is fast with Linear complexity.
  - It is easy to understand and implement
- Disadvantage:
  - Works only with integers.

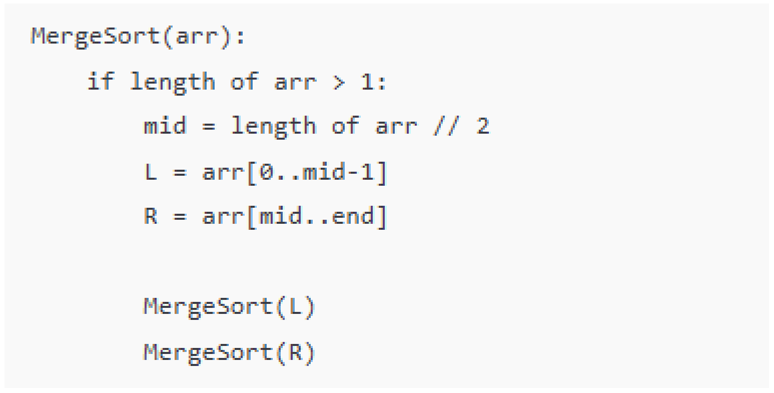
# Divide-and-Conquer

- Divide-and conquer is a general algorithm design paradigm:
  - Divide: divide the input data $S$ in two disjoint subsets $S_1$ and $S_2$
  - Recur: solve the subproblems associated with $S_1$ and $S_2$
  - Conquer: combine the solutions for $S_1$ and $S_2$ into a solution for $S$
- The base case for the recursion are subproblems of size 0 or 1

- Merge-sort is a sorting algorithm based on the divide-and-conquer paradigm
- Like heap-sort
  - It uses a comparator
  - It has $O(n \log n)$ running time
- Unlike heap-sort
  - It does not use an auxiliary priority queue
  - It accesses data in a sequential manner (suitable to sort data on a disk)

## Merge Sort

- Merge-sort on an input sequence ***Arr*** with ***n*** elements consists of three steps:
  - Divide: partition ***Arr*** into two sequences ***L*** and ***R*** of about $n/2$ elements each
  - Recur: recursively sort ***L*** and ***R***
  - Conquer: merge ***L*** and ***R*** into a unique sorted sequence

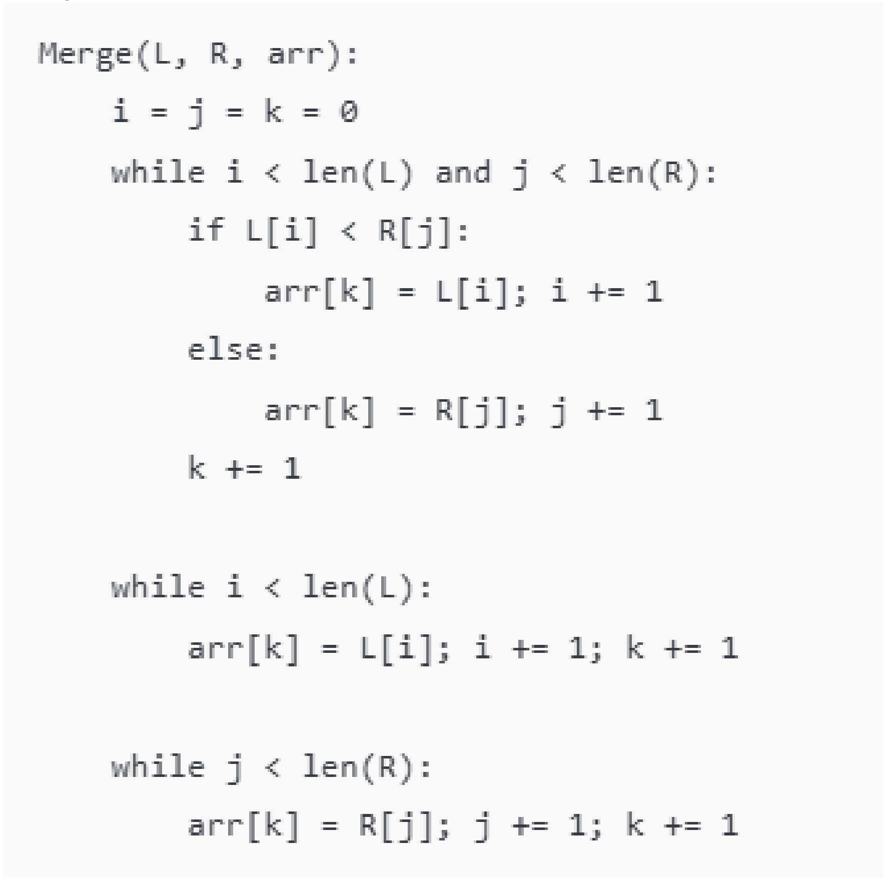


```
MergeSort(arr):
    if length of arr > 1:
        mid = length of arr // 2
        L = arr[0..mid-1]
        R = arr[mid..end]

        MergeSort(L)
        MergeSort(R)
```

# Merging Two Sorted Sequences

- The conquer step of merge-sort consists of merging two sorted sequences $\boldsymbol{L}$ and $\boldsymbol{R}$ into a sorted array.



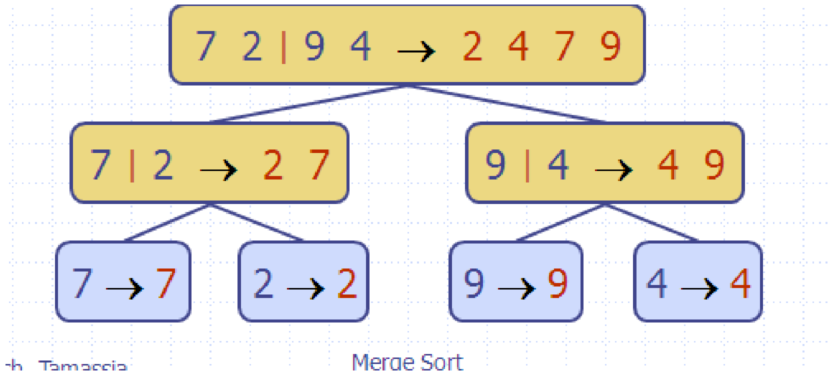
```
Merge(L, R, arr):
    i = j = k = 0
    while i < len(L) and j < len(R):
        if L[i] < R[j]:
            arr[k] = L[i]; i += 1
        else:
            arr[k] = R[j]; j += 1
        k += 1

    while i < len(L):
        arr[k] = L[i]; i += 1; k += 1

    while j < len(R):
        arr[k] = R[j]; j += 1; k += 1
```

# Merge-Sort Tree

- An execution of merge-sort is depicted by a binary tree
  - each node represents a recursive call of merge-sort and stores
    - unsorted sequence before the execution and its partition
    - sorted sequence at the end of the execution
  - the root is the initial call
  - the leaves are calls on subsequences of size 0 or 1



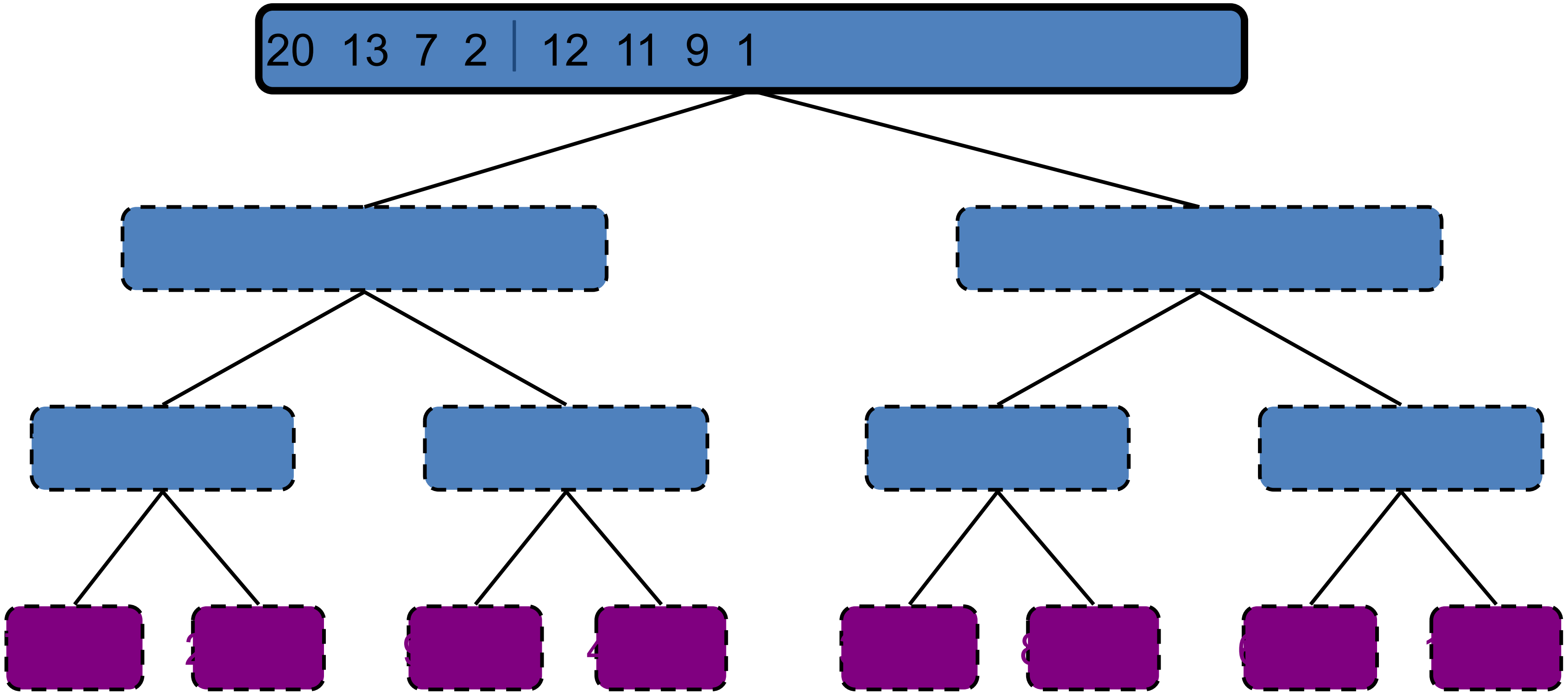
## Merge sort

```
MERGE-SORT $A[1 \,..\, n]$
  1. If $n = 1$, done.
  2. Recursively sort $A[\,1 \,..\, \lceil n/2 \rceil\,]$
     and $A[\,\lceil n/2 \rceil + 1 \,..\, n\,]$ .
  3. "Merge" the 2 sorted lists.
```

***Key subroutine:*** MERGE

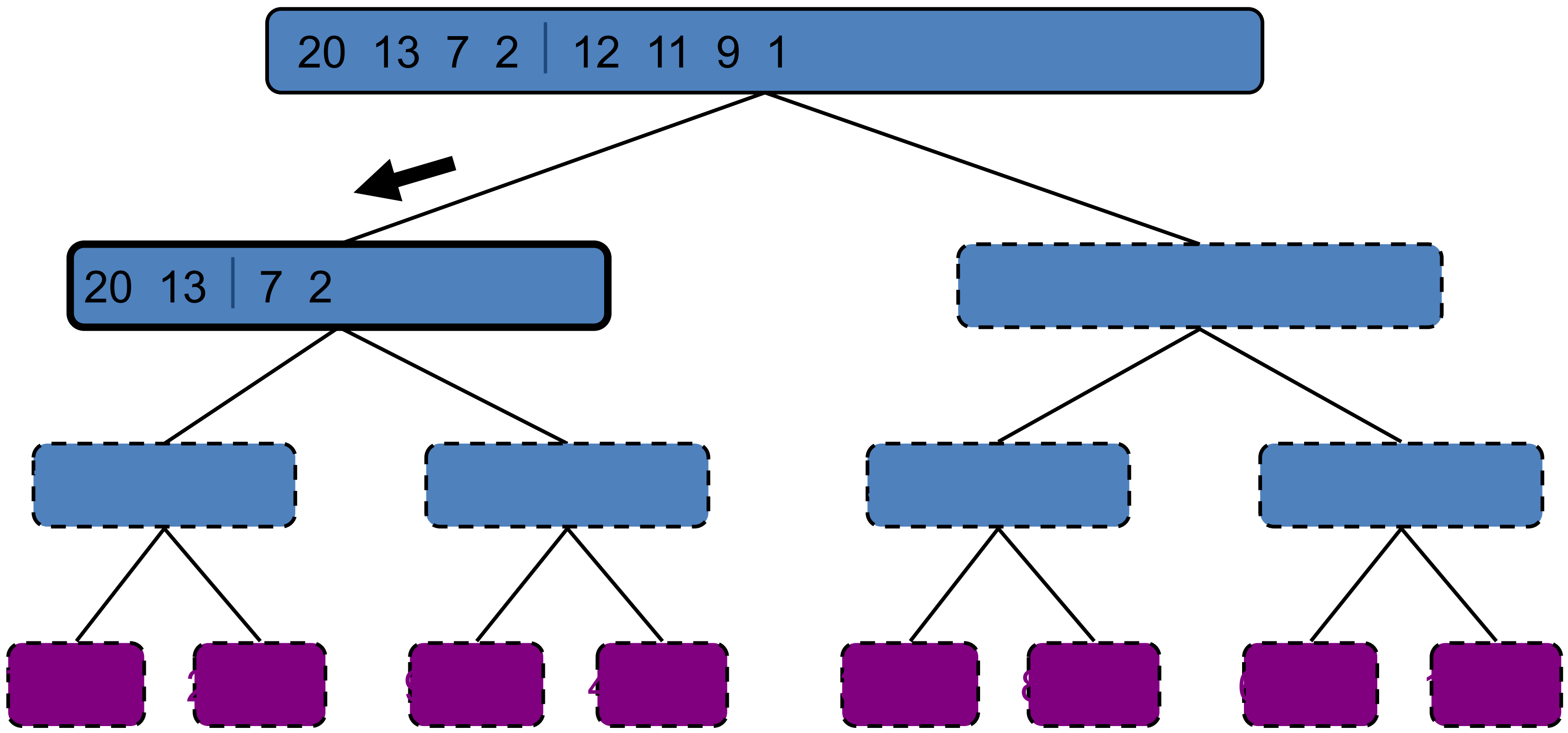
# Execution Example

- Example : 20 13 7 2 12 11 9 1



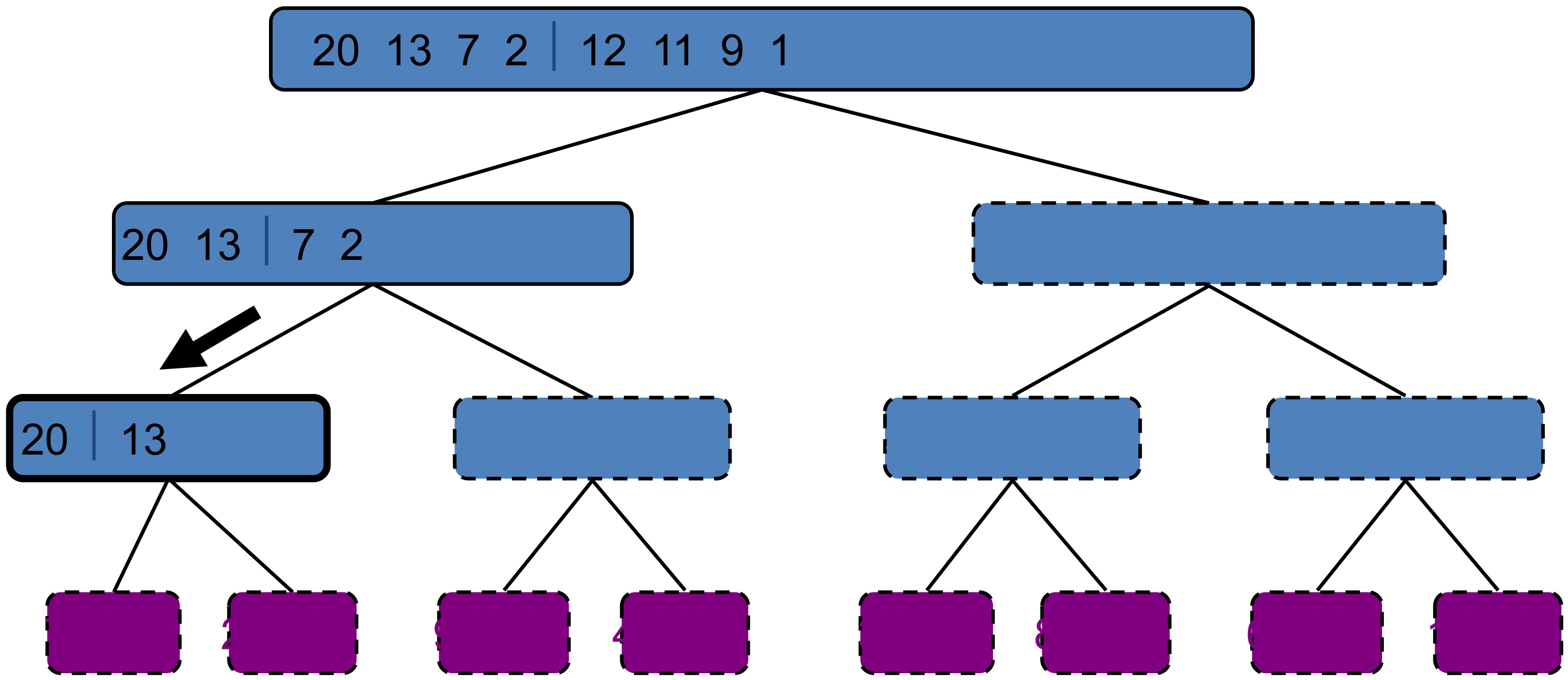
# Execution Example (cont.)

- Recursive call, partition



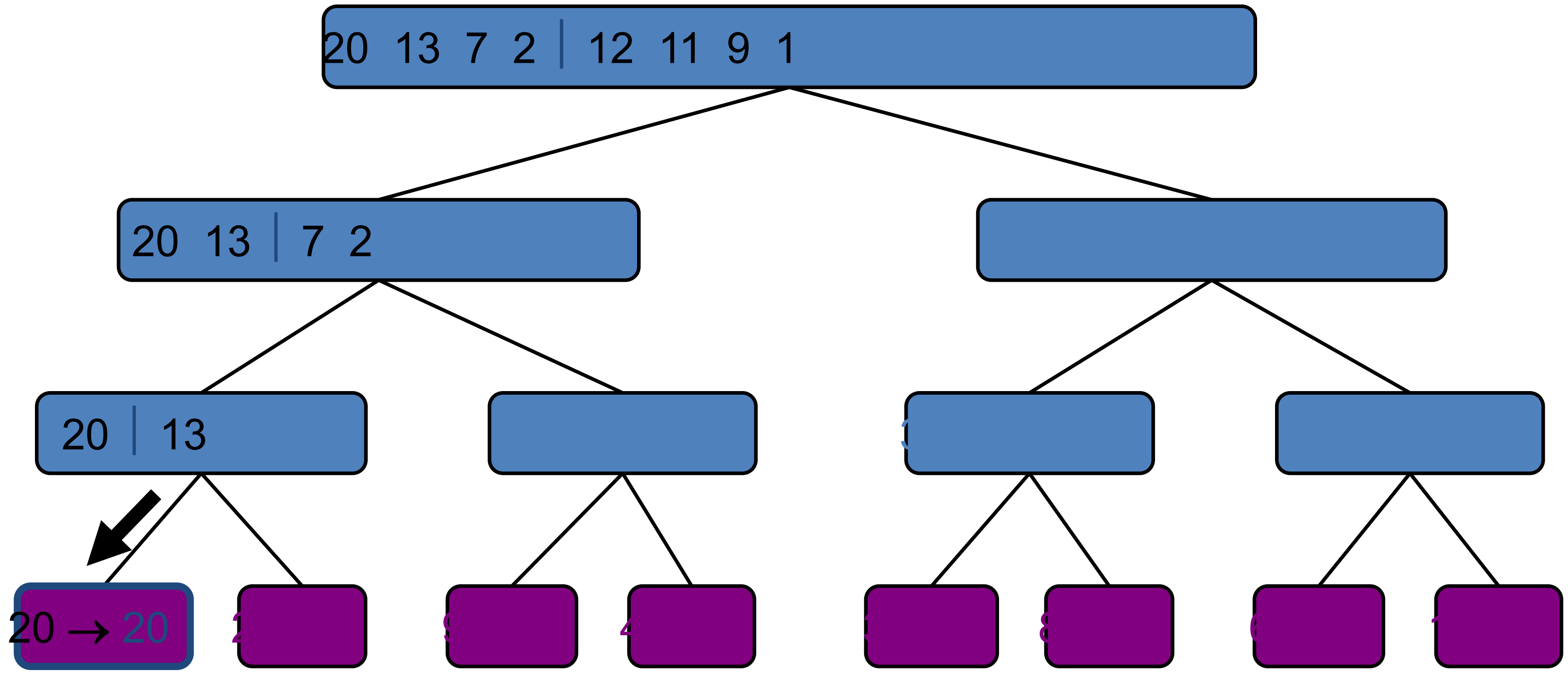
# Execution Example (cont.)

- Recursive call, partition



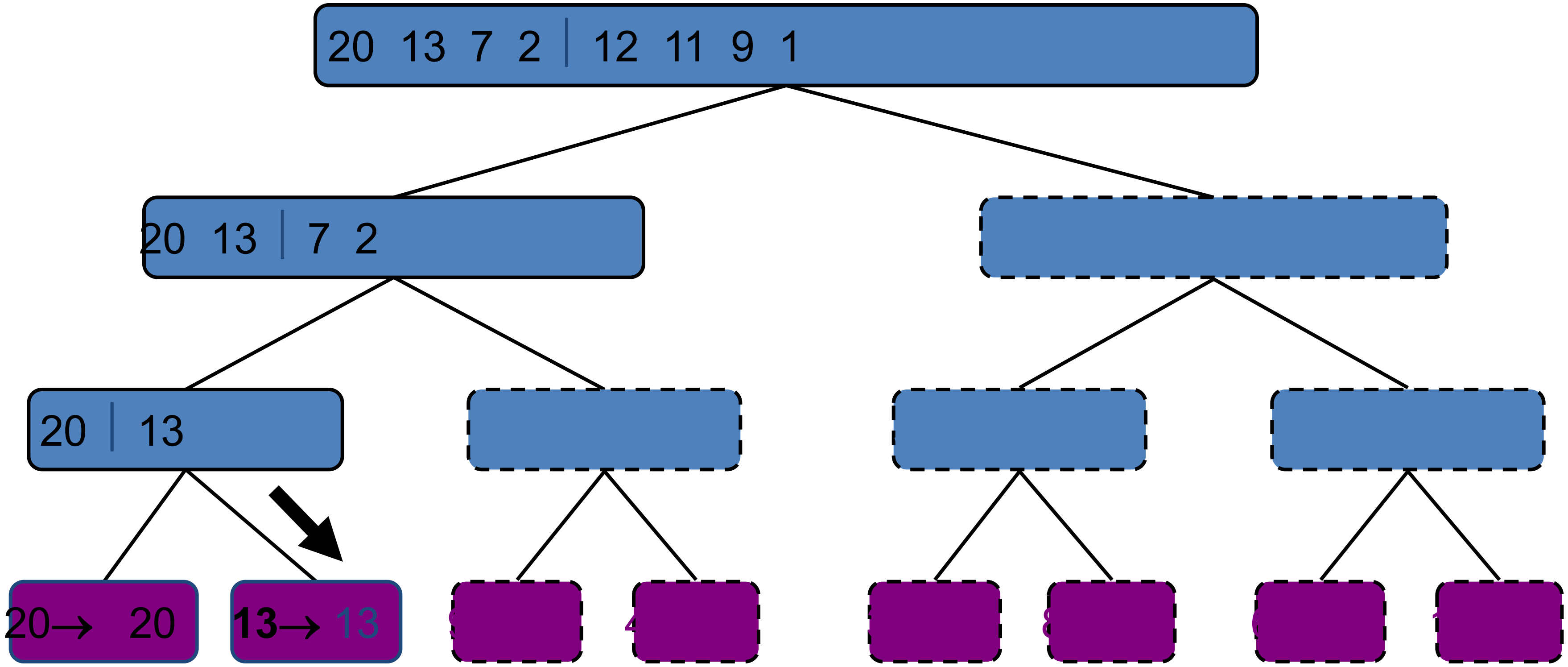
# Execution Example (cont.)

- Recursive call, base case



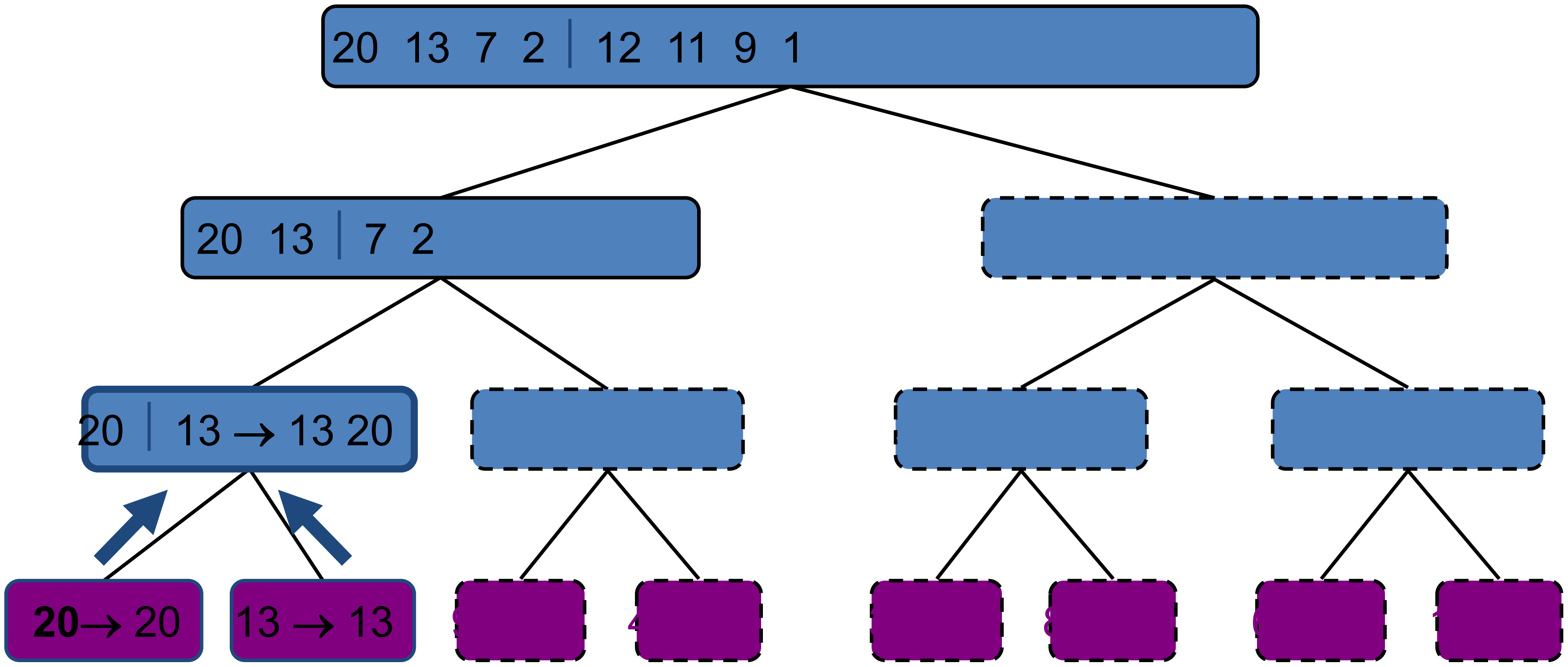
# Execution Example (cont.)

- Recursive call, base case



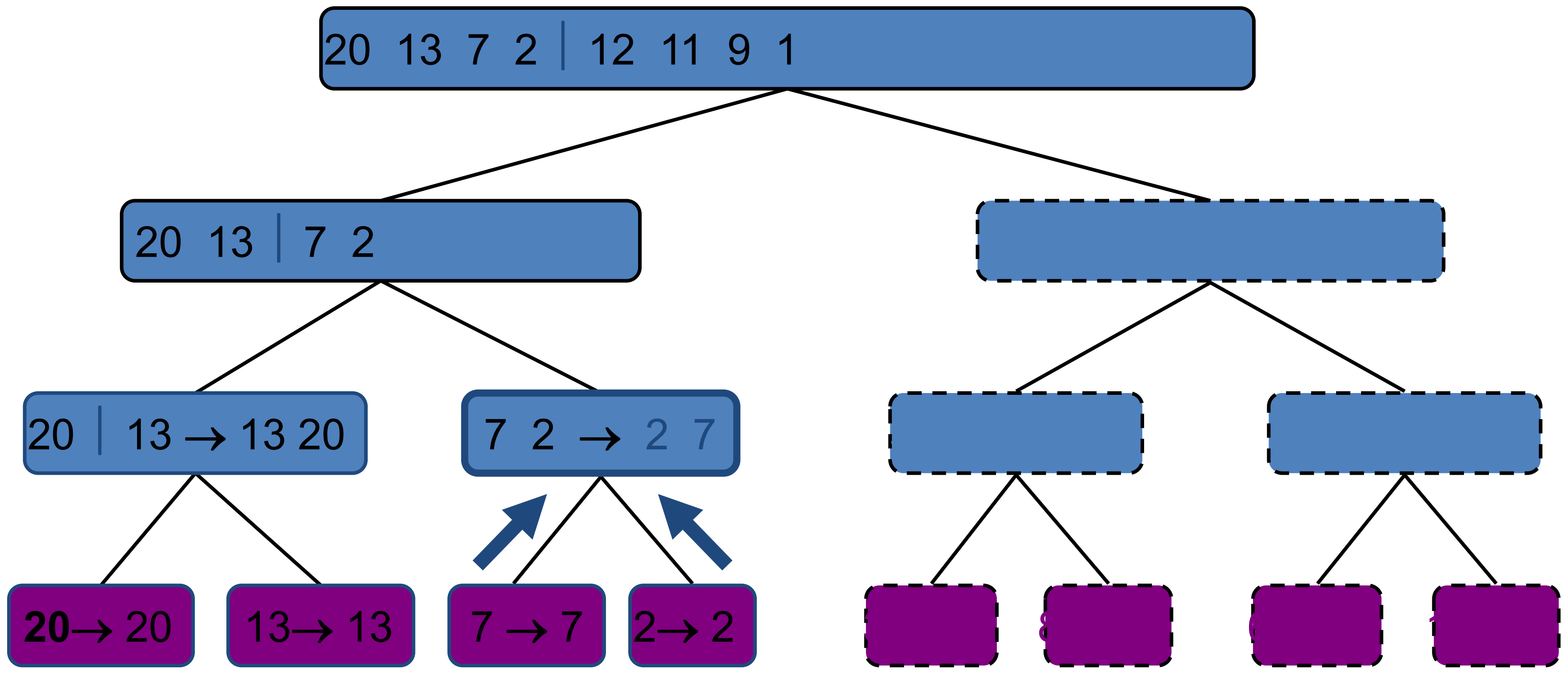
# Execution Example (cont.)

- Merge



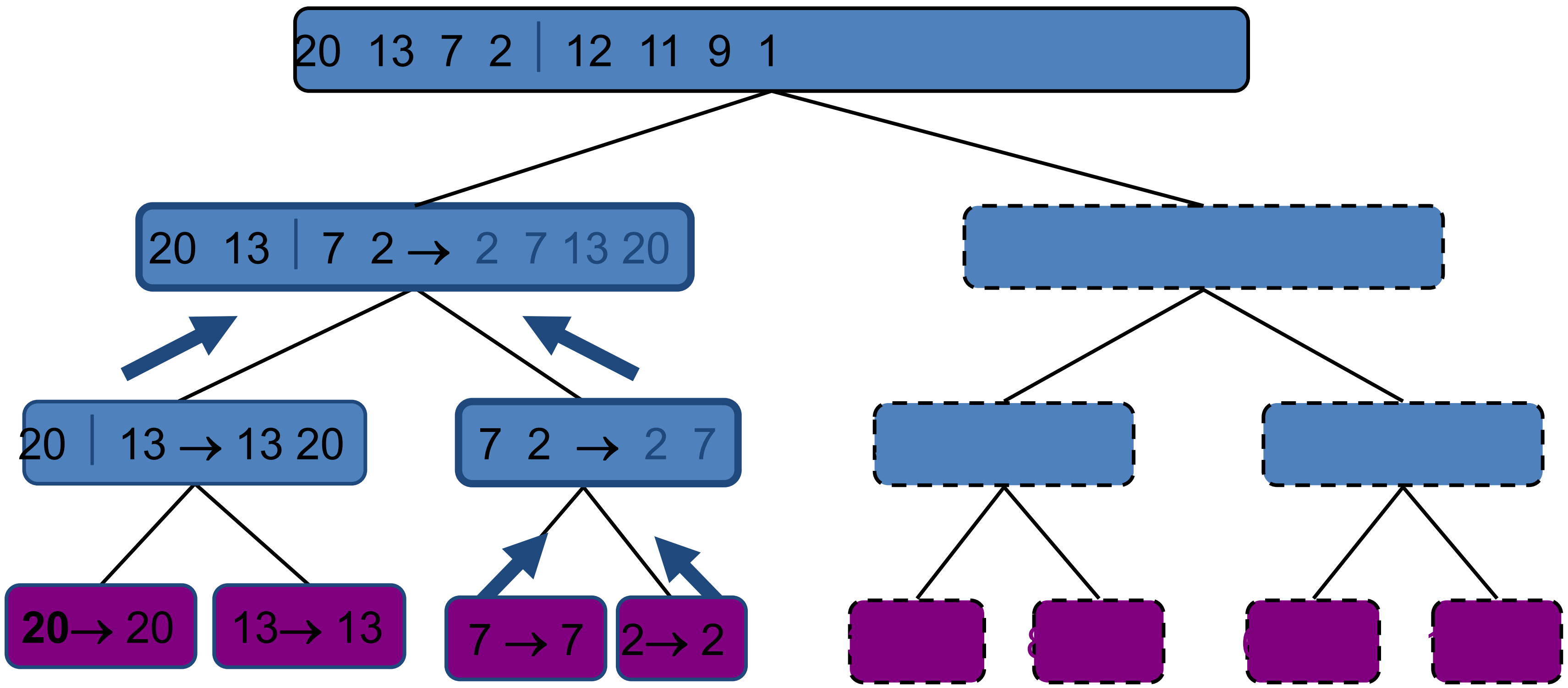
# Execution Example (cont.)

- Recursive call, …, base case, merge



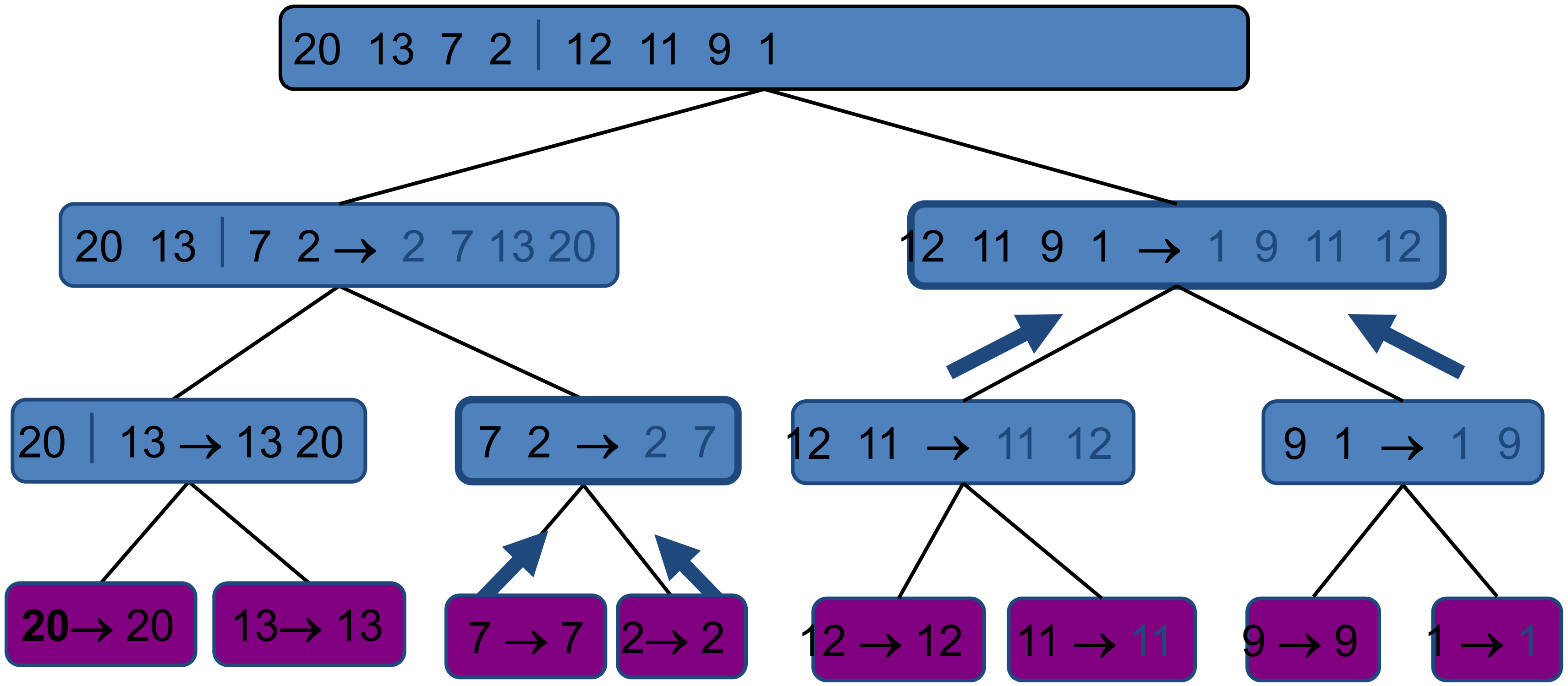
# Execution Example (cont.)

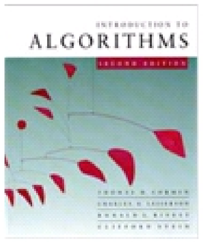
- Merge



# Execution Example (cont.)

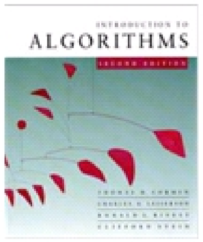
- Recursive call, …, merge, merge



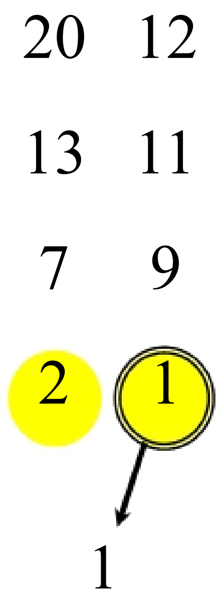


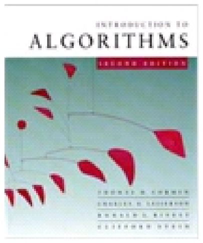
# Merging two sorted arrays

|  |  |
|---|---|
| 20 | 12 |
| 13 | 11 |
| 7 | 9 |
| 2 | 1 |

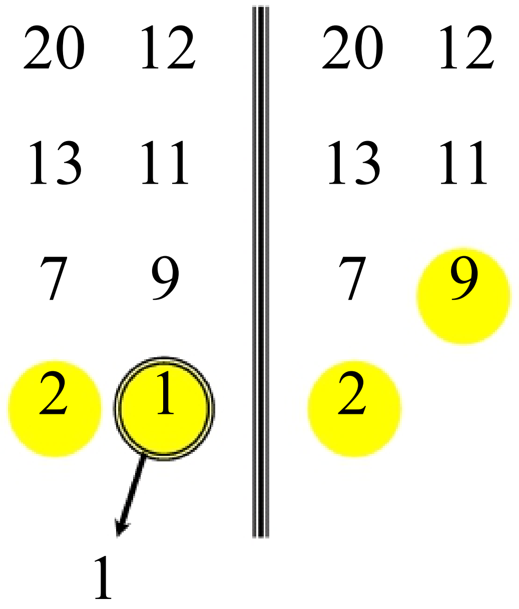


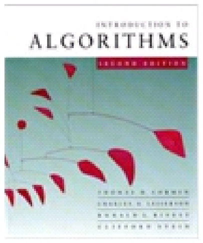
# Merging two sorted arrays



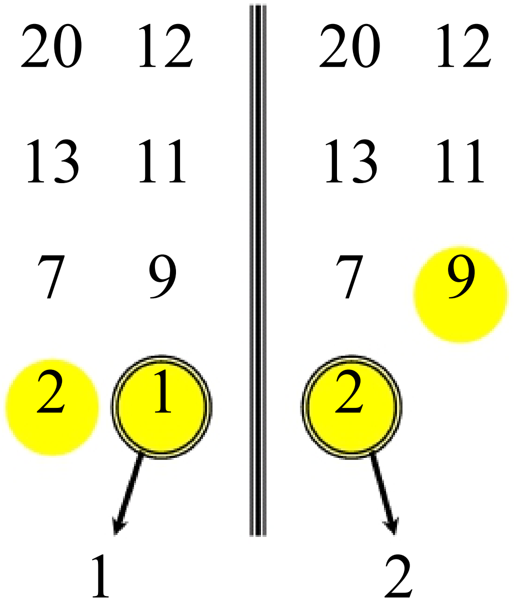

INTRODUCTION TO ALGORITHMS
SECOND EDITION

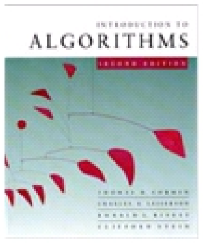
# Merging two sorted arrays



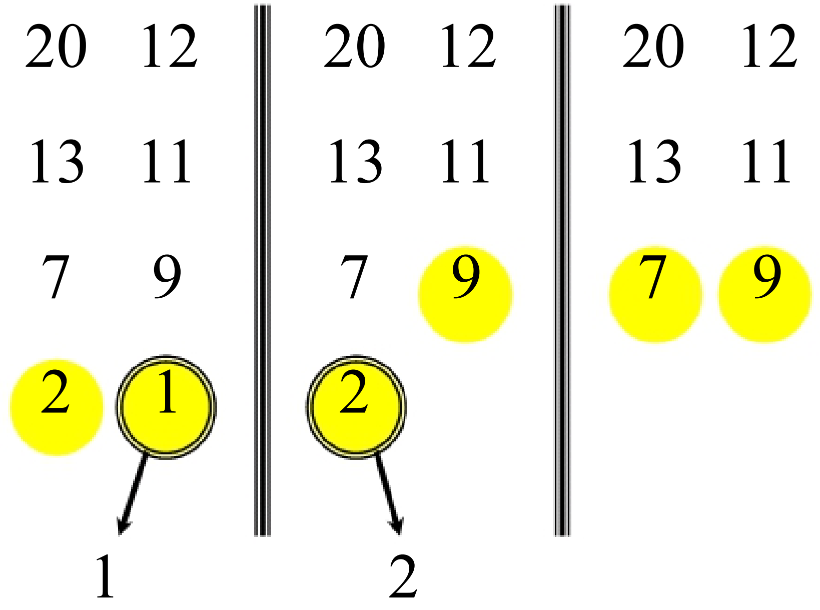


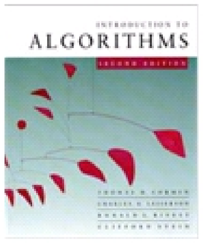
# Merging two sorted arrays



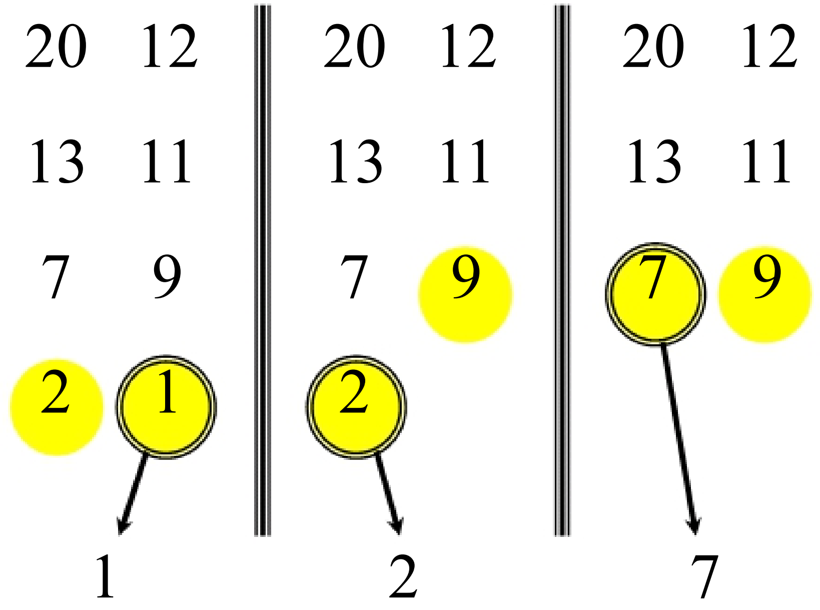


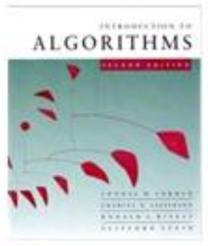
# Merging two sorted arrays



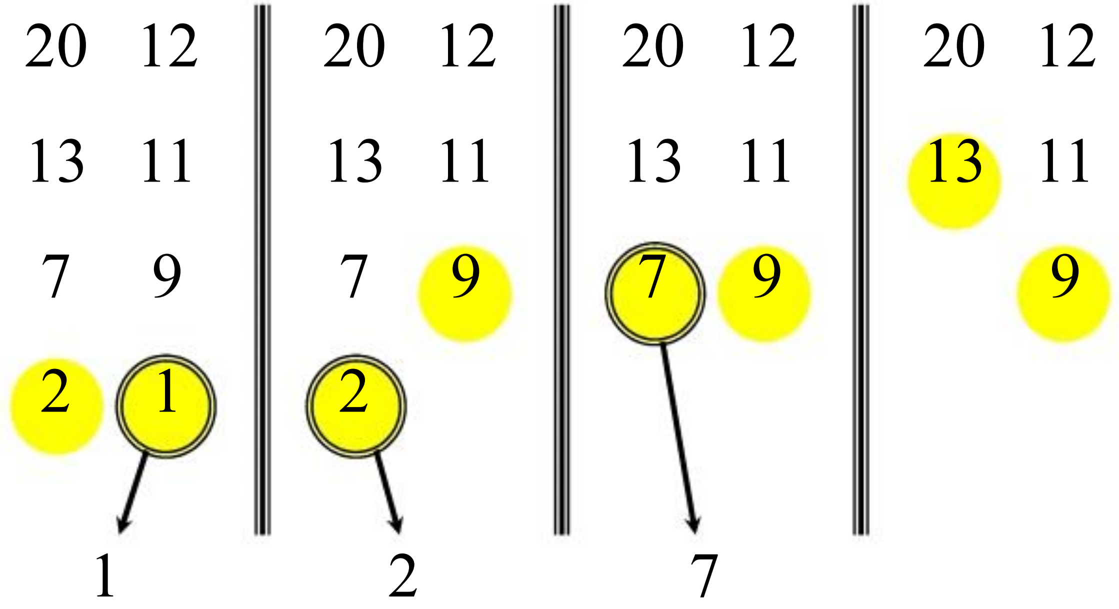


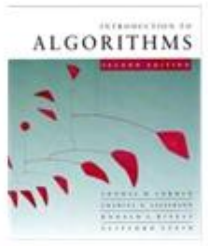
# Merging two sorted arrays



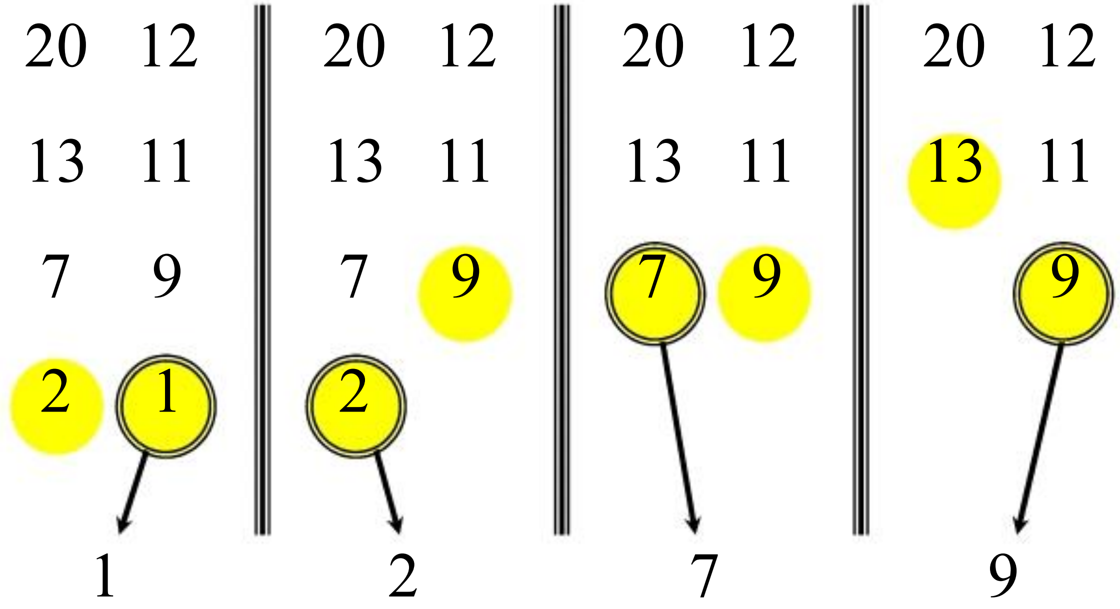


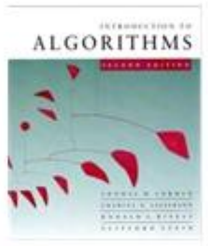
# Merging two sorted arrays



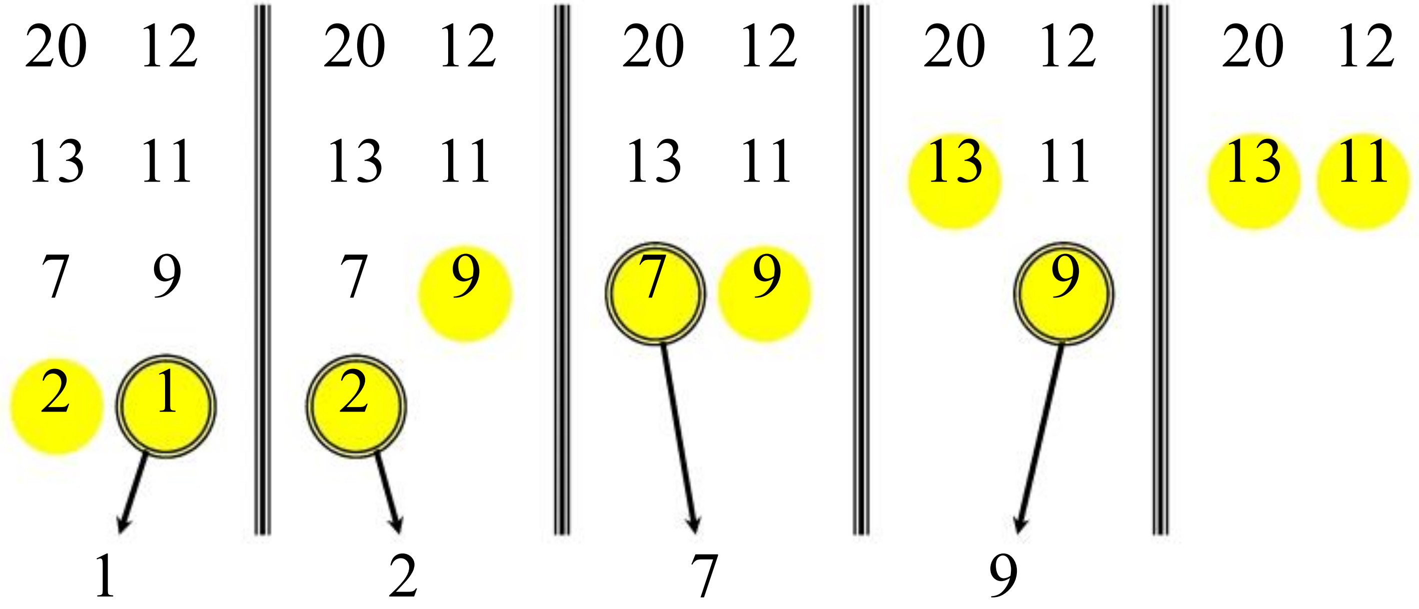


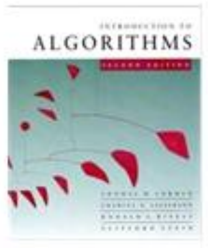
## Merging two sorted arrays



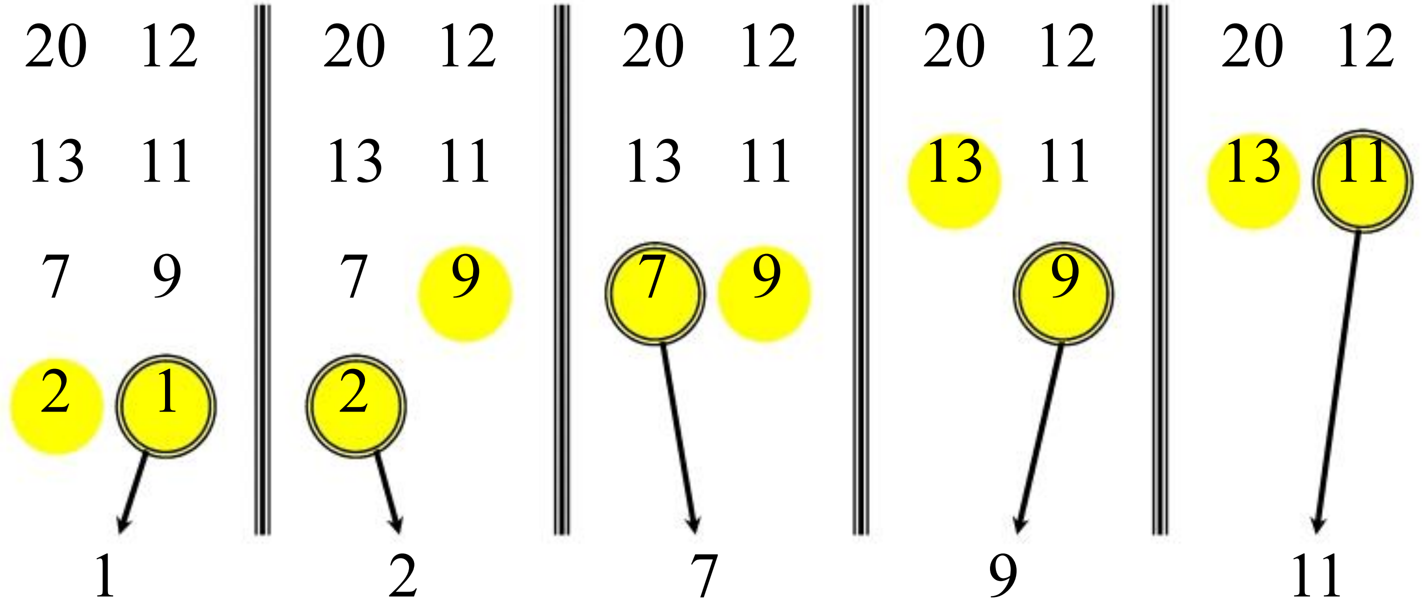


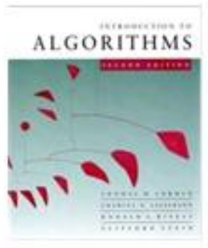
# Merging two sorted arrays



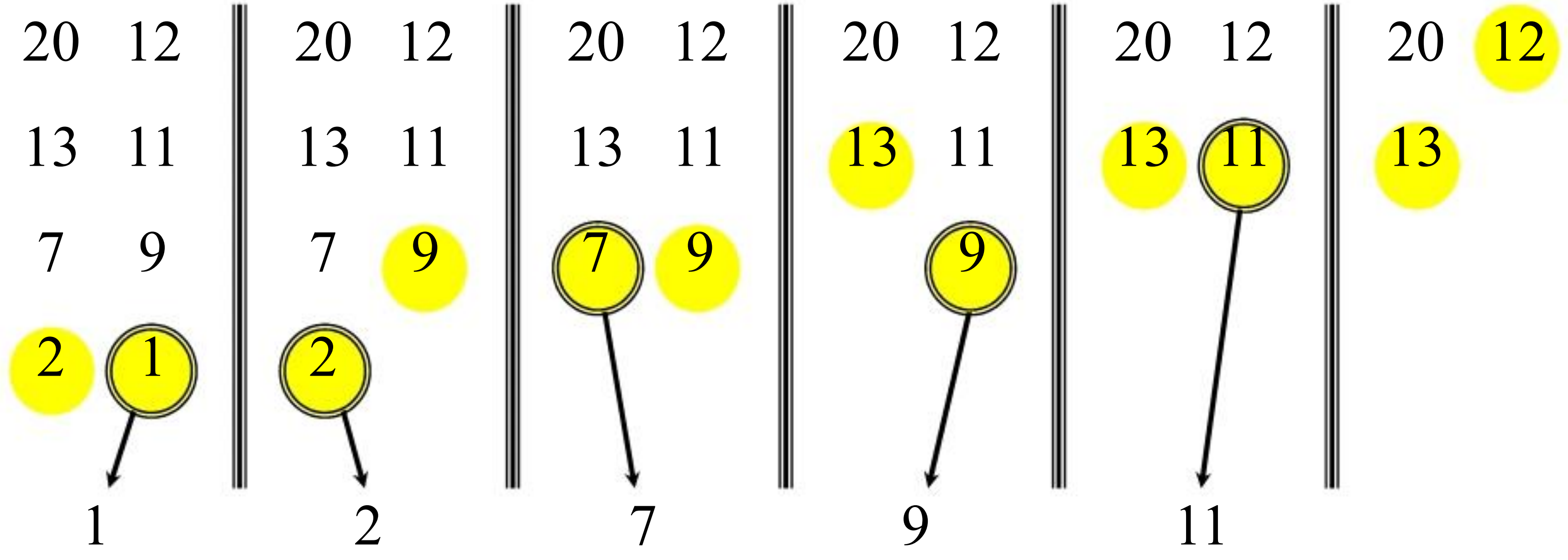


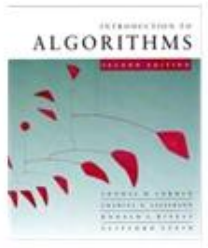
# Merging two sorted arrays



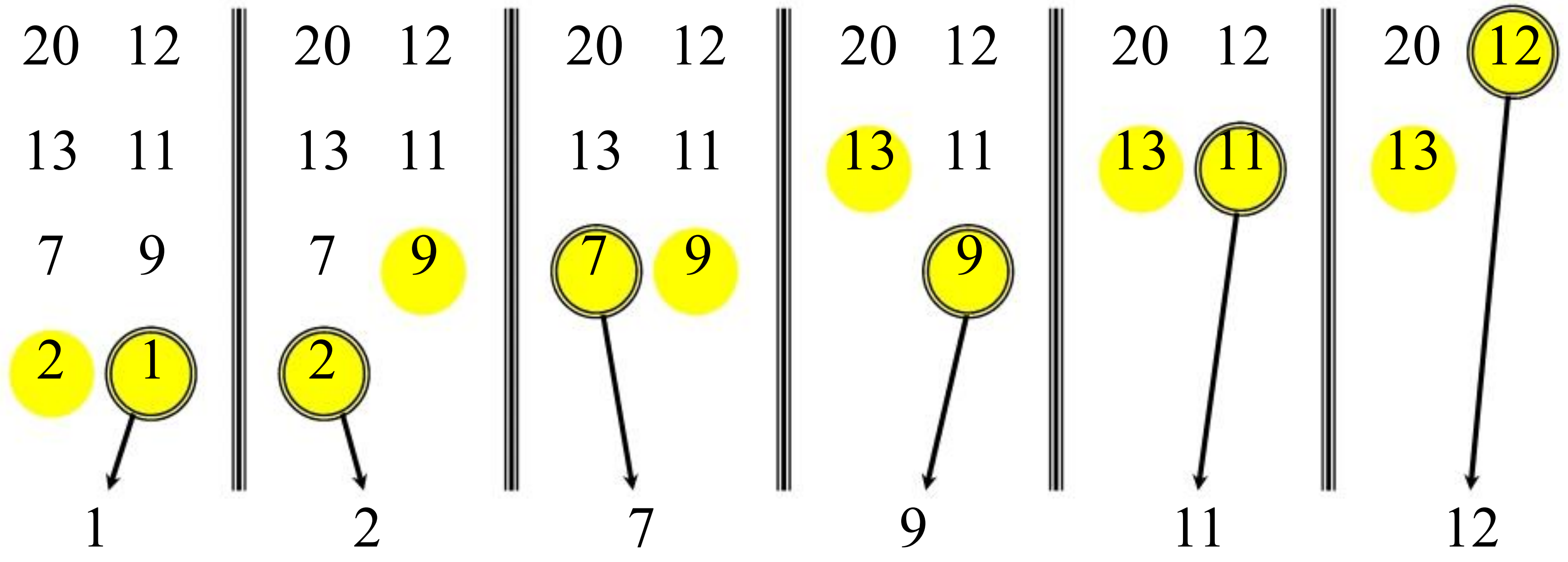


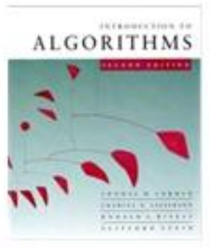
# Merging two sorted arrays



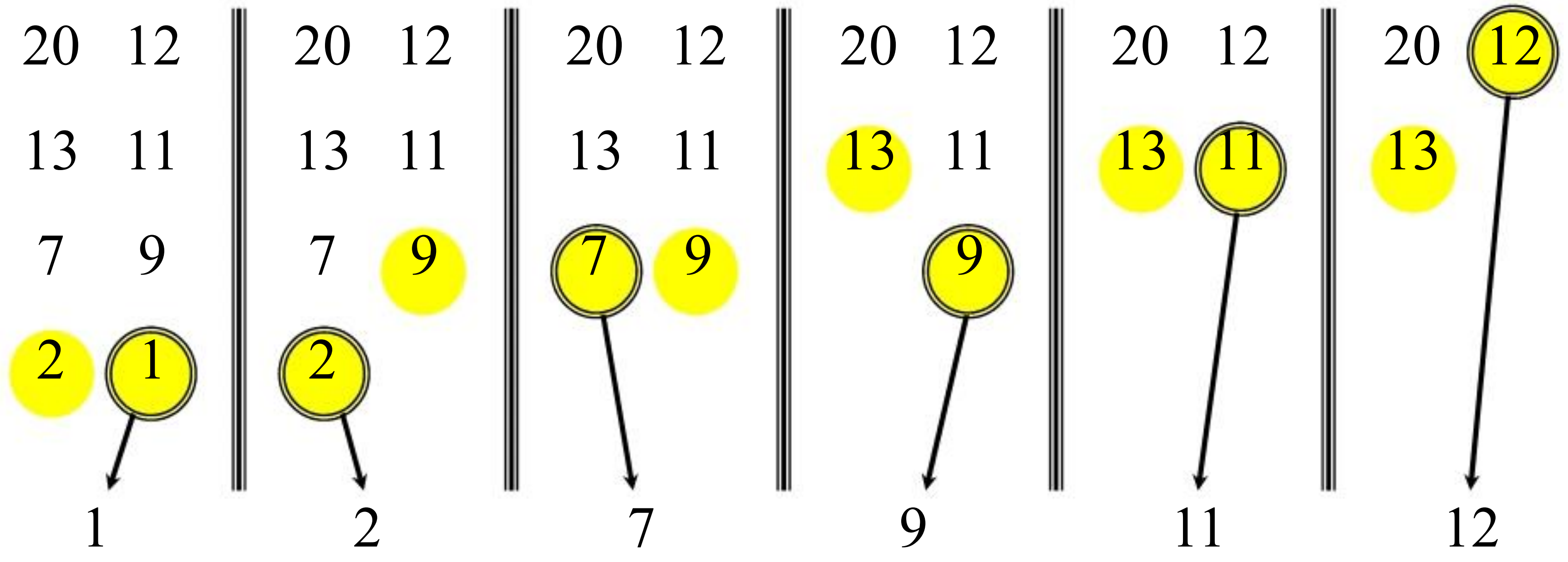


# Merging two sorted arrays





## Merging two sorted arrays



Time = $\Theta(n)$ to merge a total of $n$ elements (linear time).

# Analyzing merge sort

| | |
|---|---|
| $T(n)$ | MERGE-SORT $A[1\,.\,.\,n]$ |
| $\Theta(1)$ | 1. If $n = 1$, done. |
| $2T(n/2)$ | 2. Recursively sort $A[\,1\,.\,.\lceil n/2\rceil\,]$ and $A[n/2]+1\,.\,.\;n\;]$ |
| $\Theta(n)$ | ***3. "Merge"*** the 2 sorted lists |

The worst case and average case running time will be O(n log2 n).

## Analyzing merge sort (Un-rolling)

- Let T(n) the time used to sort n elements. As we can perform separation and merging in linear time, it takes cn time to perform these two steps, for some constant c. So,
- **$T(n) = 2T(n/2) + cn$**.
  In the same way:

  $T(n) = 4T(n/4) + 2cn$. Going in this way ...
- $T(n) = 2^m T(n/2^m) + mcn$, and
  $T(n) = 2^k T(n/2^k) + kcn$
- Here Assume $n=2^k$ and $k = \log_2 n$!
- So $T(n)=nT(1) + cn\log_2 n = O(n \log n)$.
- It takes $O(n \log n)$ time anyway.

# Analyzing merge sort (Using Master Theorem)

- Let T(n) the time used to sort n elements.
- **T(n) = 2T(n/2) + cn**.

The Master Theorem solves recurrences of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Where:

- $a \geq 1$: number of subproblems
- $b > 1$: factor by which the problem size is divided
- $f(n)$: cost of combining the results

Given:

- $a = 2$
- $b = 2$
- $f(n) = n$

Now compute:

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

If a=b :: Case 2

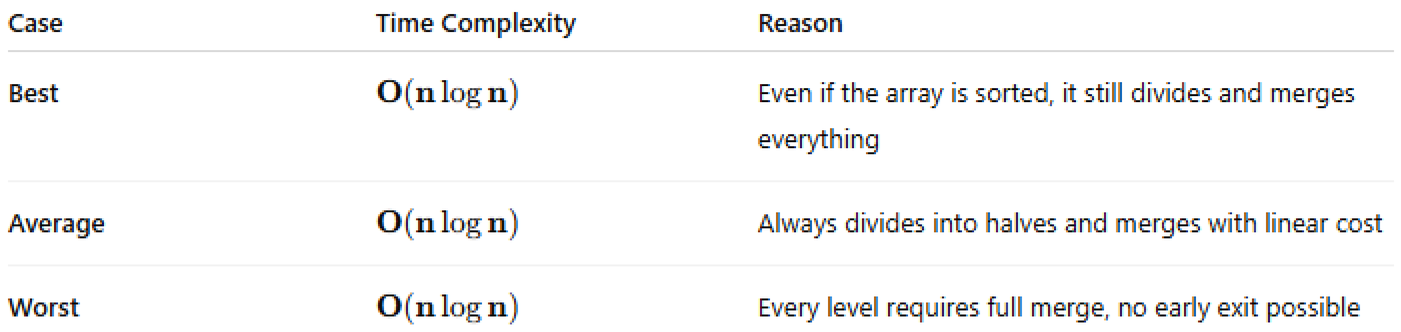
- $f(n) = \Theta(n^{\log_b a} \cdot \log^0 n) \Rightarrow p = 0$

So, by **Case 2**:

$$T(n) = \Theta\left(n \cdot \log^{0+1} n\right) = \boxed{\Theta(n \log n)}$$

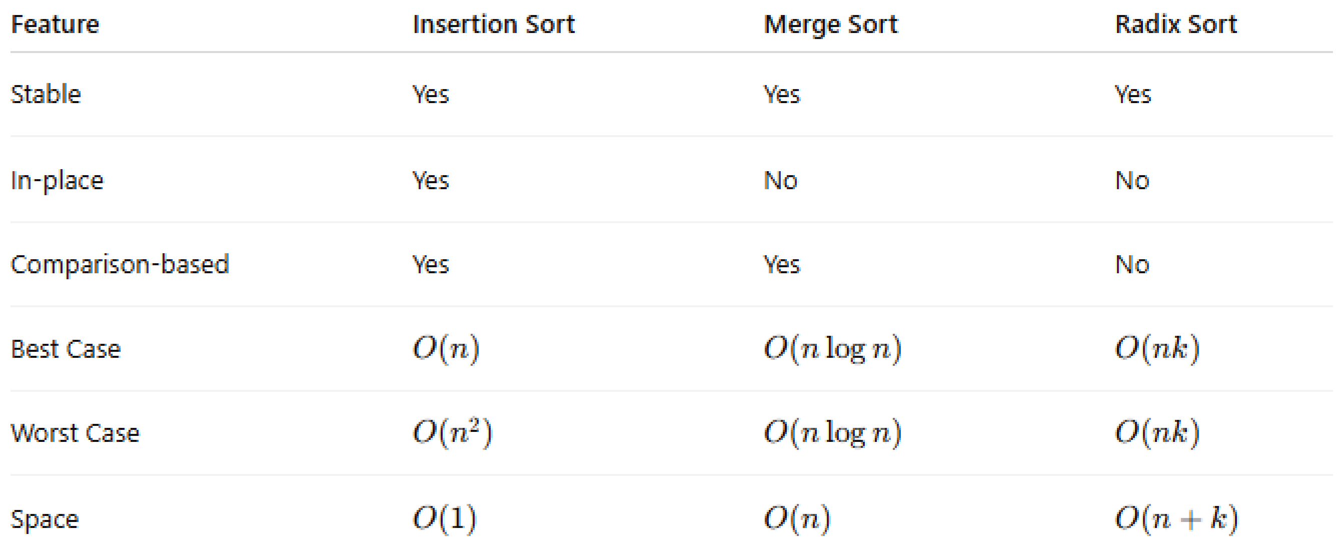
# Merge Sort

| Algorithm | Worst Case | Averge Case | Extra Memry |
|---|---|---|---|
| Merge Sort | nlogn-=O(nlogn) | nlogn=O(nlogn) | O(n) (Due to temporary arrays during merging) |



| Case | Time Complexity | Reason |
|---|---|---|
| Best | $O(n \log n)$ | Even if the array is sorted, it still divides and merges everything |
| Average | $O(n \log n)$ | Always divides into halves and merges with linear cost |
| Worst | $O(n \log n)$ | Every level requires full merge, no early exit possible |

# Comparison



| Feature | Insertion Sort | Merge Sort | Radix Sort |
|---|---|---|---|
| Stable | Yes | Yes | Yes |
| In-place | Yes | No | No |
| Comparison-based | Yes | Yes | No |
| Best Case | $O(n)$ | $O(n \log n)$ | $O(nk)$ |
| Worst Case | $O(n^2)$ | $O(n \log n)$ | $O(nk)$ |
| Space | $O(1)$ | $O(n)$ | $O(n + k)$ |

**Reading Assignment:**

- In Place  Comparison
- Stable sorting