Sorting

IT5003: Data Structures and Algorithms
(**AY2019/20 Semester 1**)

Lecture Outline

- Iterative sorting algorithms (comparison based)
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Recursive sorting algorithms (comparison based)
  - Merge Sort
  - Quick Sort
- Radix sort (non-comparison based)
- Properties of Sorting
  - In-place sort, stable sort
  - Comparison of sorting algorithms
- Note: we only consider sorting data in **ascending order**

Why Study Sorting?

- When an input is sorted, many problems become easy (e.g. searching, min, max, *k*-th smallest)

- Sorting has a variety of interesting algorithmic solutions that embody many ideas
  - Comparison vs non-comparison based
  - Iterative
  - Recursive
  - Divide-and-Conquer
  - Best / Worst / Average-case bounds
  - Randomized algorithms

Applications of Sorting

- Uniqueness testing
- Deleting duplicates
- Prioritizing events
- Frequency counting
- Reconstructing the original order
- Set intersection/union
- Finding a target pair x, y such that $x+y = z$
- Efficient searching

Congrats! You have been selected…….

SELECTION SORT

Selection Sort : **Idea**

- **Observations:**
  - It is simple to find the ***largest item*** in a list of unsorted numbers
  - We know the ***correct position*** of the largest item once it is found
  - Once the largest item is in the correct position, it can be "***ignored***"
- Try applying the idea:

A	0	1	2	3	4	5	6	7	8	9	10
	8	3	-5	4	2	-10	13	2	-7	9	0

Selection Sort: **Attempt One**

- Use a temporary array to store the result?

A	29	10	14	37	13
Result					**37**

- Difficulty and drawbacks:
  - Need to "remove" the max number from the original array every round
  - Need to keep track of the current position in the result[] array
  - **Wasteful**: Need an additional size N array for result

Selection Sort: **Attempt Two**

29	10	14	37	13
29	10	14	13	37
13	10	14	29	37
13	10	14	29	37
10	13	14	29	37

37 is the largest, swap it with the last element, i.e. **13**

x Unsorted items

x **Largest item for current iteration**

x **Sorted items**

Sorted!

Selection Sort: **Implementation**

```
def selectionSort(array):
    n = len(array)
    for i in range(n-1,0, -1):
        maxIdx = i
        for j in range(0, i):
            if array[j] > array[maxIdx]:
                maxIdx = j
        swapElement(array, maxIdx, i)
```

Step 1: Search for maximum element

Step 2: Swap maximum element with the last item

```
#useful helper function
def swapElement(array, x, y):
    temp = array[x]
    array[x] = array[y]
    array[y] = temp
```

Selection Sort : **Analysis**

```
def selectionSort(array):
    n = len(array)
    for i in range(n-1,0, -1):
        maxIdx = i

        for j in range(0, i):
            if array[j] > array[maxIdx]:
                maxIdx = j


        swapElement(array, maxIdx, i)
```

Number of times executed
N-1
(N-1)+(N-2)+…+1 = N(N-1)/2
N-1

Total
c_1(N-1) + c_2*N*(N-1)/2 = O(N^2)

Just _________ (insert your answer)

INSERTION SORT

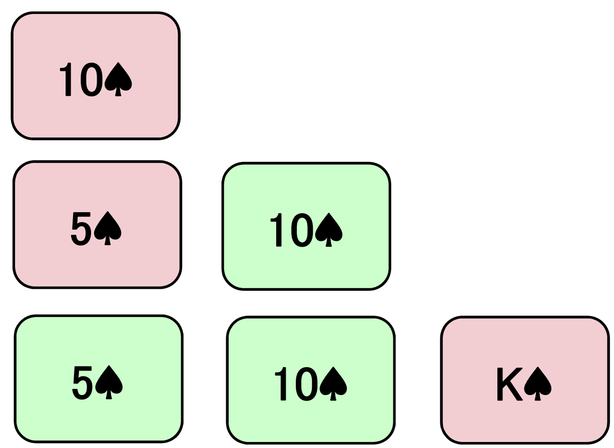
Insertion Sort: **Idea**

- Similar to how most people arrange a hand of poker cards
  1. Start with one card in your hand
  2. Pick the next card and **insert it into its proper sorted order**
  3. Repeat previous step for all cards

1st card: **10♠**

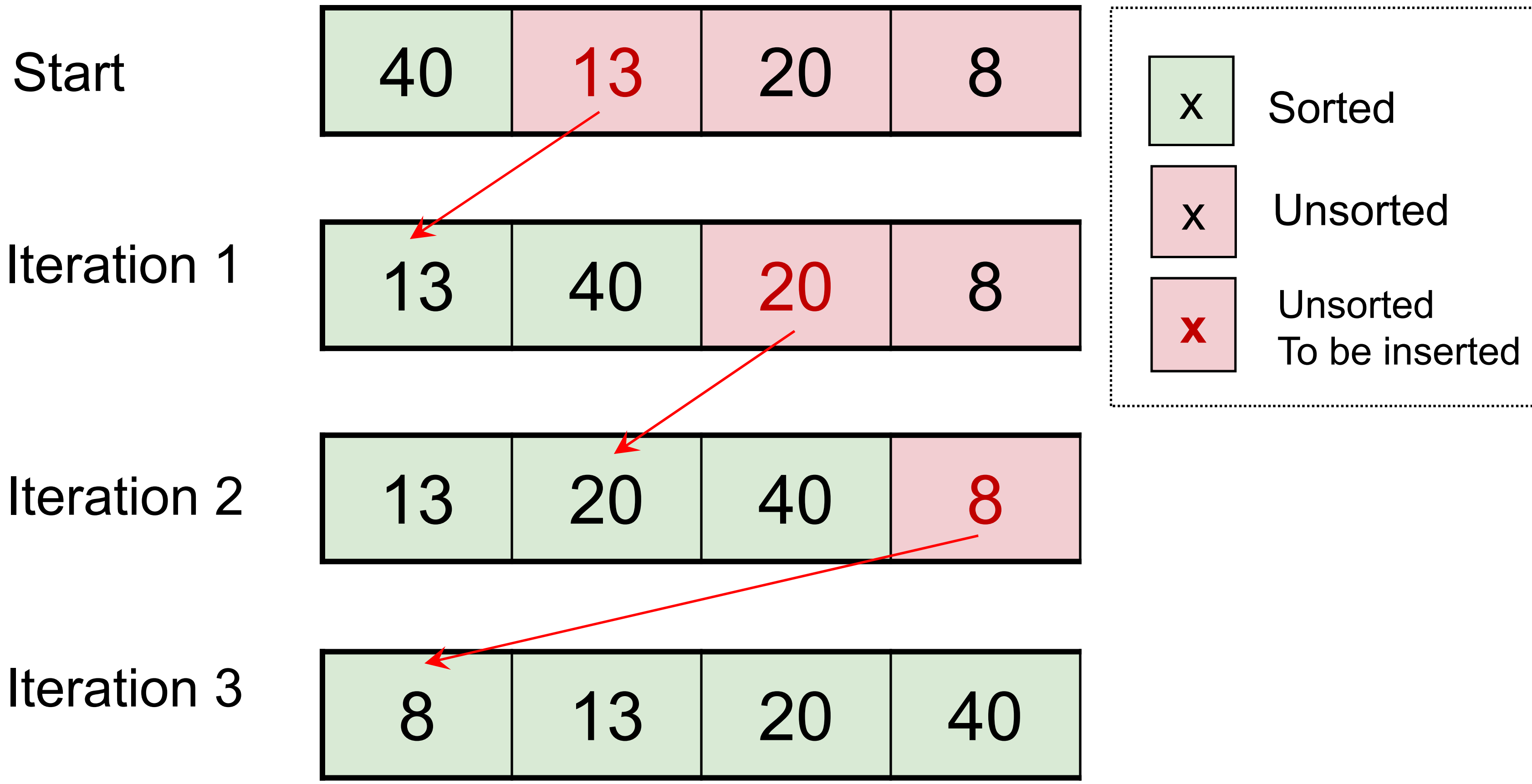
2nd card: **5♠**

3rd card: **K♠**



… … … …

Insertion Sort: **Illustration**



Insertion Sort : **Implementation**

```
def insertionSort(array):
    n = len(array)

    for i in range (1, n):
        next = array[i]
        j = i-1

        while j >= 0 and array[j] > next:
            swapElement(array, j+1, j)
            j = j-1

        array[j+1] = next;
```

next is the item to be inserted

Shift sorted items to make place for **next**

Insert **next** to the correct location

next

10

10	29	14	37	13

Insertion Sort : **Analysis**

```python
def insertionSort(array):
    n = len(array)

    for i in range (1, n):
        next = array[i]
        j = i-1

        while j >= 0 and array[j] > next:
            swapElement(array, j+1, j)
            j = j-1

        array[j+1] = next;
```

Number of times executed
N-1
Best Case: N-1 **Worst Case:** (N-1)+(N-2)+…+1 = N(N-1)/2
N-1

Best Case
O(N)

Worst Case
$O(N^2)$

- The inner loop is **input sensitive**:
  - **Best case**: Input is already sorted
  - **Worst case**: Input is in reverse order

Bubble to the top!

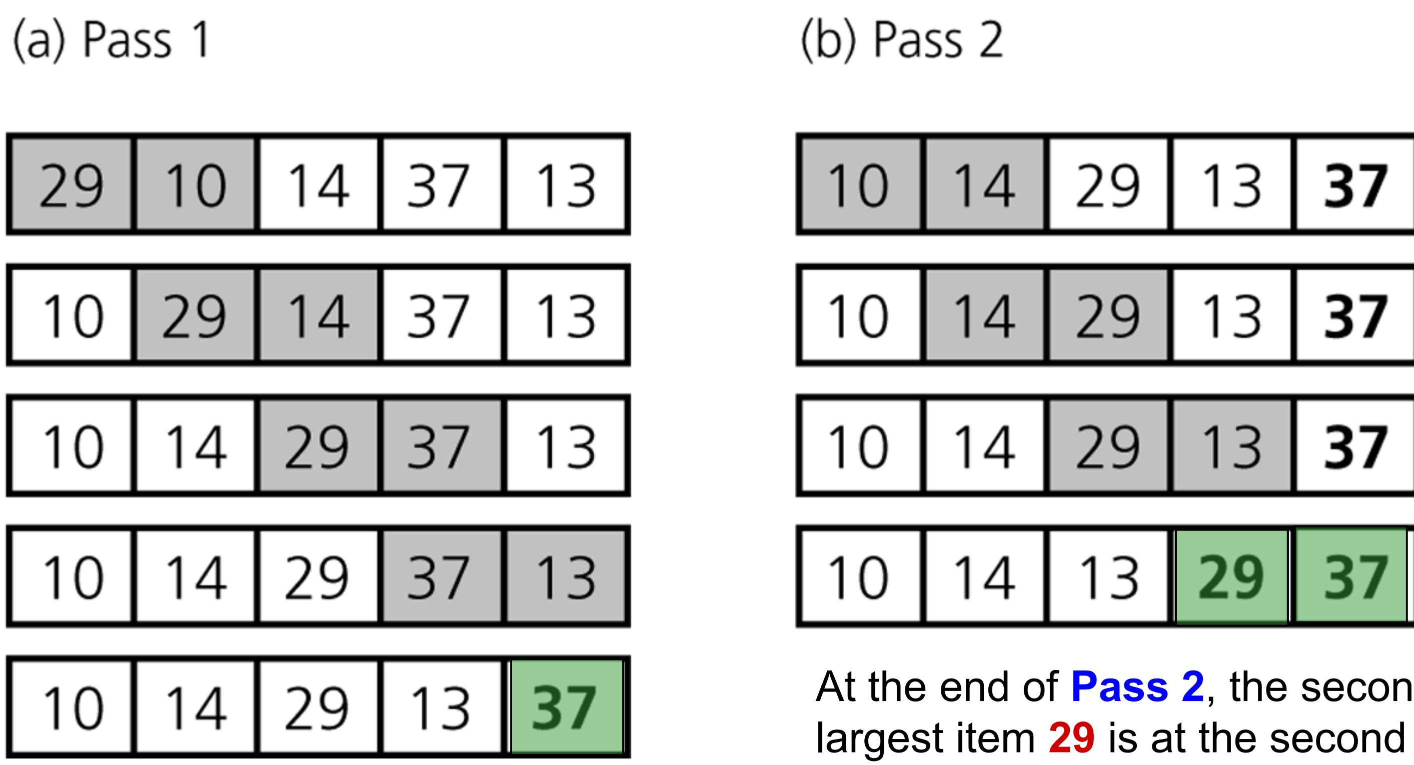
BUBBLE SORT

Bubble Sort: Idea

- Given an array of *n* items
  1. Compare pair of adjacent items
  2. Swap if the items are out of order
  3. Repeat until the end of array
     - The largest item will be at the last position
  4. Reduce *n* by 1 and go to Step 1

- Analogy
  - Large item is like "bubble" that floats to the end of the array

Bubble Sort: Illustration



At the end of **Pass 1**, the largest item **37** is at the last position.

At the end of **Pass 2**, the second largest item **29** is at the second last position.

Bubble Sort: Implementation

```
def bubbleSort(array):
    n = len(array)

    for i in range(n-1, 0, -1):
        for j in range (1, i+1):
            if array[j-1] > array[j]:
                swapElement(array, j, j-1)
```

Step 1: Compare adjacent pairs of numbers

Step 2: Swap if the items are out of order

Try it out!

29	10	14	37	13

Bubble Sort : **Analysis**

```
def bubbleSort(array):
    n = len(array)

    for i in range(n-1, 0, -1):
        for j in range (1, i+1):
            if array[j-1] > array[j]:
                swapElement(array, j, j-1)
```

Number of times executed
N
(N-1)+(N-2)+…+0 = N(N-1)/2

Total
c *N*(N-1)/2 = $O(N^2)$

Bubble Sort: Early Termination

- Bubble Sort is inefficient with a $O(n^2)$ time complexity
- However, it has an interesting property
  - Given the following array, how many times will the inner loop swap a pair of item?

3	6	11	25	39

- Idea
  - If we go through the inner loop with no swapping
    ➔ the array is sorted
    ➔ can stop early!

Bubble Sort v2.0: Implementation

```
def bubbleSortEarly(array):
    n = len(array)
    for i in range(n-1, 0, -1):
        isSorted = True

        for j in range (1, i+1):
            if array[j-1] > array[j]:
                swapElement(array, j, j-1)
                isSorted = False

        if isSorted:
            return
```

Assume the array is sorted before the inner loop

Any swapping will invalidate the assumption

If the flag remains **true** after the inner loop ➔ sorted!

Bubble Sort Ver 2.0 : **Analysis**

- **Worst-case**
  - Input is in descending order
  - Running-time remains the same: $O(n^2)$

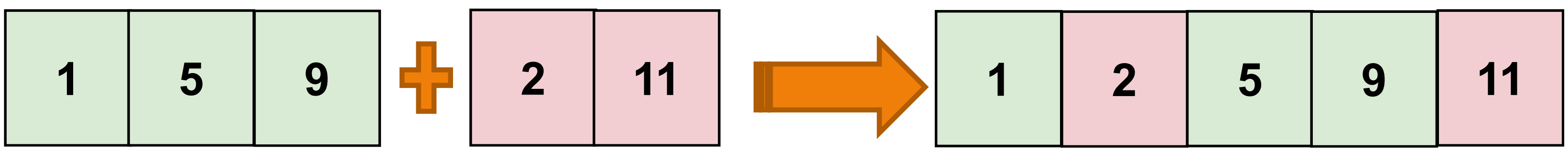
- **Best-case**
  - Input is already in ascending order
  - Algorithm returns after a single outer-loop
  - Running time: $O(n)$

left, right, left, left, right === merged!

MERGE SORT

Merge Sort : **Idea**

- Suppose we only know **how to merge** two **sorted** sets of elements into one



- Question:
  - Where do we get the two sorted sets?
- Idea (use merge to sort ***n*** items):
  - **Merge** each pair of elements into sets of ***2***
  - **Merge** each pair of sets of 2 into sets of ***4***
  - Repeat previous step for sets of 4 …
  - **Final step**: **Merges** 2 sets of ***n/2*** elements to obtain a sorted set

Divide and Conquer Method

- A powerful problem solving technique

- Divide-and-conquer method solves problem in the following steps:
  - **Divide Step:**
    - divide the large problem into smaller problems
    - **Recursively** solve the smaller problems
  - **Conquer Step:**
    - combine the results of the smaller problems to produce the result of the larger problem

Merge Sort : **Divide and Conquer**

- **Merge Sort** is a divide-and-conquer sorting algorithm

- **Divide Step:**
  - Divide the array into two (equal) halves
  - Recursively sort the two halves

- **Conquer Step:**
  - Merge the two halves to form a sorted array

Merge Sort : Illustartion

7	2	6	3	8	4	5

Divide into two halves

7	2	6	3

8	4	5

Recursively sort the halves

2	3	6	7

4	5	8

Merge them

2	3	4	5	6	7	8

- Question:
  - How should we sort the halves in the 2^{nd} step?

Merge Sort : Implementation

```
def mergeSort( array, low, high ):

   if low < high:
      mid = (low+high) // 2

      mergeSort(array, low, mid)
      mergeSort(array, mid+1, high)

      merge(array, low, mid, high)
```

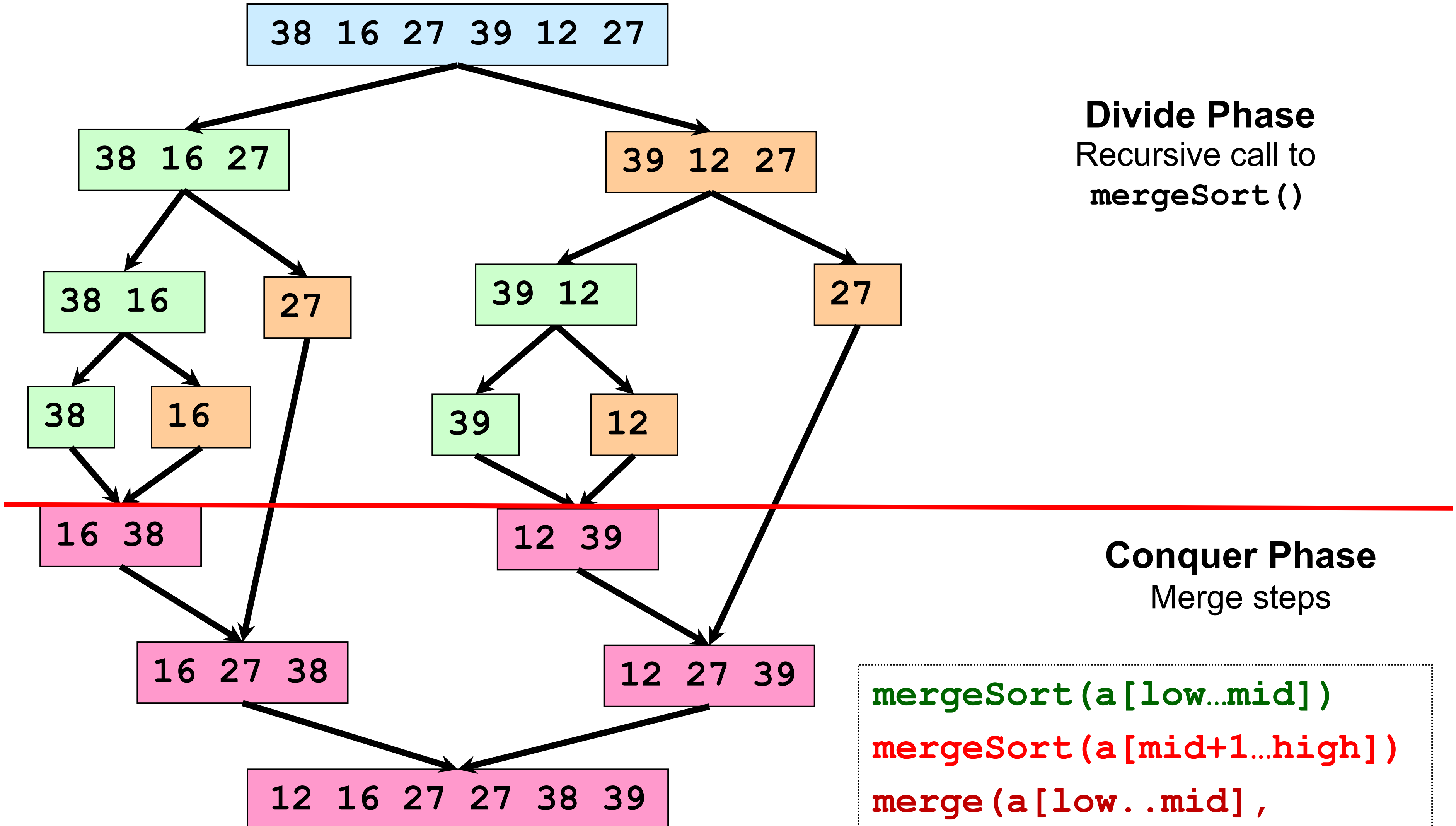
Merge sort on **a[low...high]**

Divide a[ ] into two halves and **recursively** sort them

Conquer: merge the two sorted halves

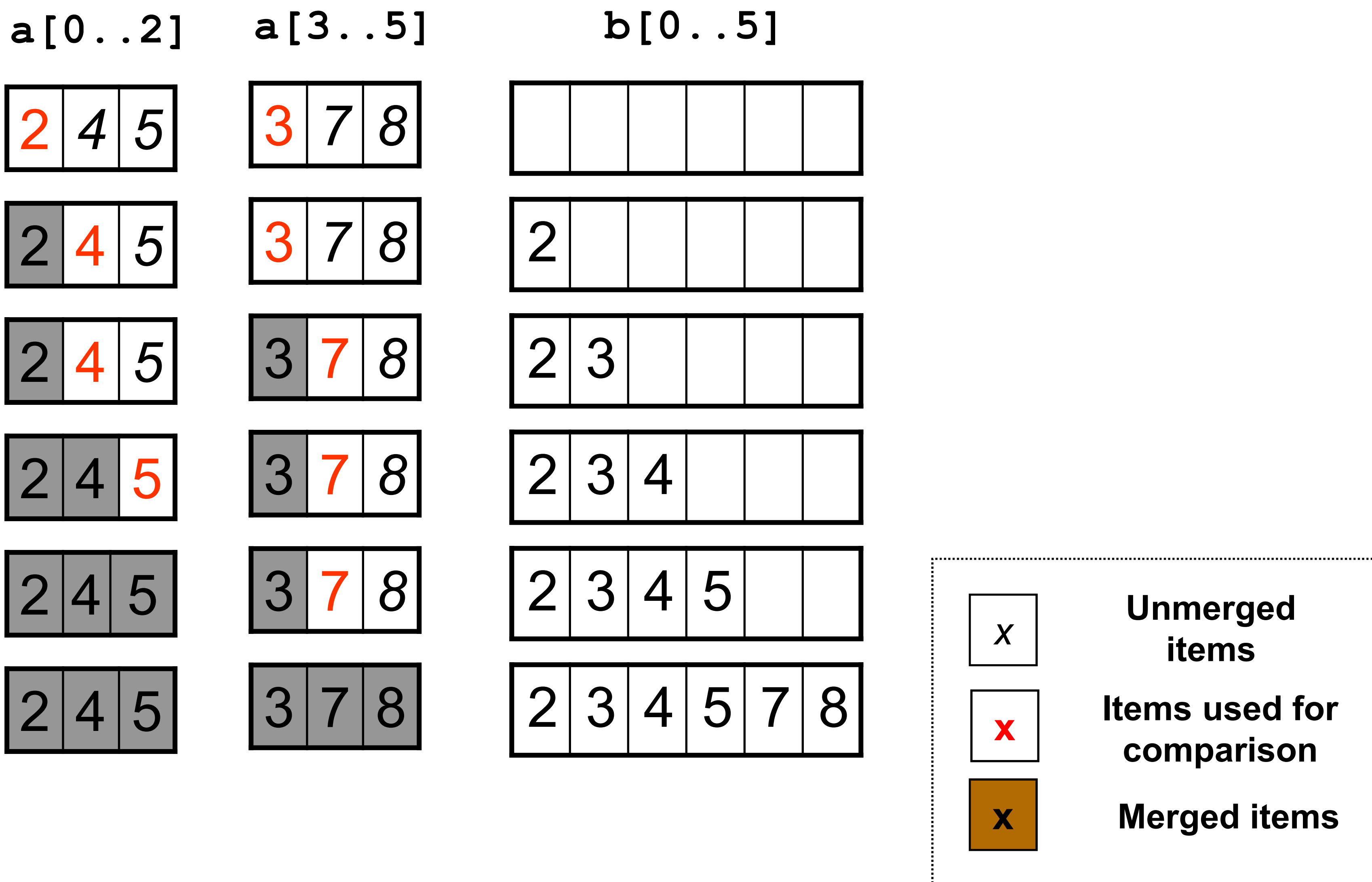
Function to merge **a[low…mid]** and **a[mid+1…high]** into **a[low…high]**

Merge Sort: An example




```
mergeSort(a[low…mid])
mergeSort(a[mid+1…high])
merge(a[low..mid],
      a[mid+1..high])
```

Merge Sort : Merging Two Sorted Halves



Merge Sort : **Merge()** code

```
def merge( array, low, mid, high ):
    n = high-low+1
    result = []
    left = low
    right = mid+1


    while left <= mid and right <= high:
        if array[left] <= array[right]:
            result.append(array[left])
            left = left + 1
        else:
            result.append(array[right])
            right = right + 1


    #continue on next slide
```

result is a temporary array to store result

Normal Merging
Where both halves have unmerged items

Merge Sort : **`Merge()`** code (cont)

```
while left <= mid:
    result.append(array[left])
    left = left + 1

while right <= high:
    result.append(array[right])
    right = right + 1

for k in range(0, n):
    array[low+k] = result[k];
```

Remaining items are copied into **`result[]`**

Merged result are copied back into **`array[]`**

- Questions to ponder:
  - Why don't we use array slicing in mergeSort?
  - Why do we need a separate result[] array during merge?

Merge Sort: `MergeSortHelper()` code

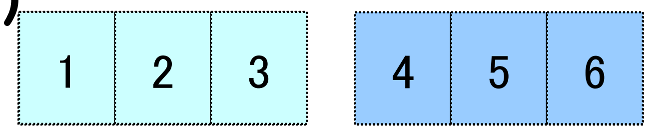
```python
def mergeSortHelper(array):
      mergeSort(array, 0, len(array)-1)
```

- To make mergeSort() consistent with other sorting algorithm (which take in only an **array** as parameter):
  - We need a helper function as above
  - Only to kick start the first recursive mergeSort() function with the right parameter
- A common "trick" to keep function interface similar

Merge Sort : **Analysis**

- In `mergeSort()`
  - the bulk of work is done in the merge step
- For `merge(a, low, mid, high)`

  Total items = `k` = `(high-low+1)`



  - Number of comparisons ≤ `k-1`
  - Number of moves from original array to temporary array = `k`
  - Number of moves from temporary array back to original array = `k`
- In total, no. of operations ≤ `3k-1` = `O(k)`
- The important question is:
  - **How many times `merge()` is called?**

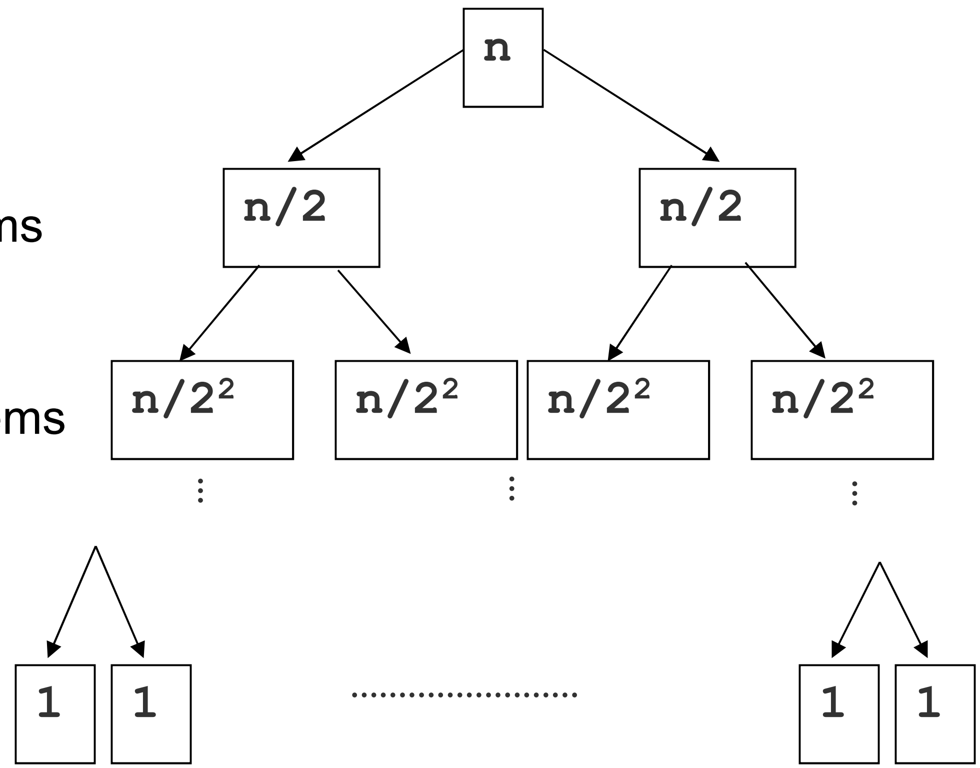
Merge Sort : Analysis (2)

Level 0:
mergeSort `n` items

Level 1:
mergeSort `n/2` items

Level 2:
mergeSort $n/2^2$ items

Level (log n):
mergeSort **1** item



Level 0:
1 call to mergeSort

Level 1:
2 calls to mergeSort

Level 2:
$\mathbf{2^2}$ calls to mergeSort

Level (log n):
$\mathbf{2^{\lg n}(= n)}$ calls to mergeSort

$$n/(2^k) = 1 \rightarrow n = 2^k \rightarrow k = \lg n$$

Merge Sort : Analysis (3)

Level	#Calls of *merge()*	Size of each *merge()*	Total Cost per level
0	---	---	---
1	1	2 x n/2 items	O(n)
2	2	2 x $n/2^2$ items	O(n)
. k .	2^{k-1}	2 x $n/2^k$ items	O(n)
lg n	$2^{\lg n-1}$	2 x $n/2^{\lg n}$ items	O(n)

- Grand total:
  - Number of levels x **O(n)** ➔ **O(n lg n)**

Merge Sort : **Pros and Cons**

- **Pros:**
  - **Optimal comparison based sorting**
  - The performance is *guaranteed*
    - Unaffected by original ordering of the input
  - Suitable for extremely large number of inputs
    - Can operate on the input portion by portion

- **Cons:**
  - Not easy to implement
  - Requires additional storage during merging operation
    - **O(n)** extra memory storage needed

Quickly do a quick sort!

QUICK SORT

Quick Sort : Idea

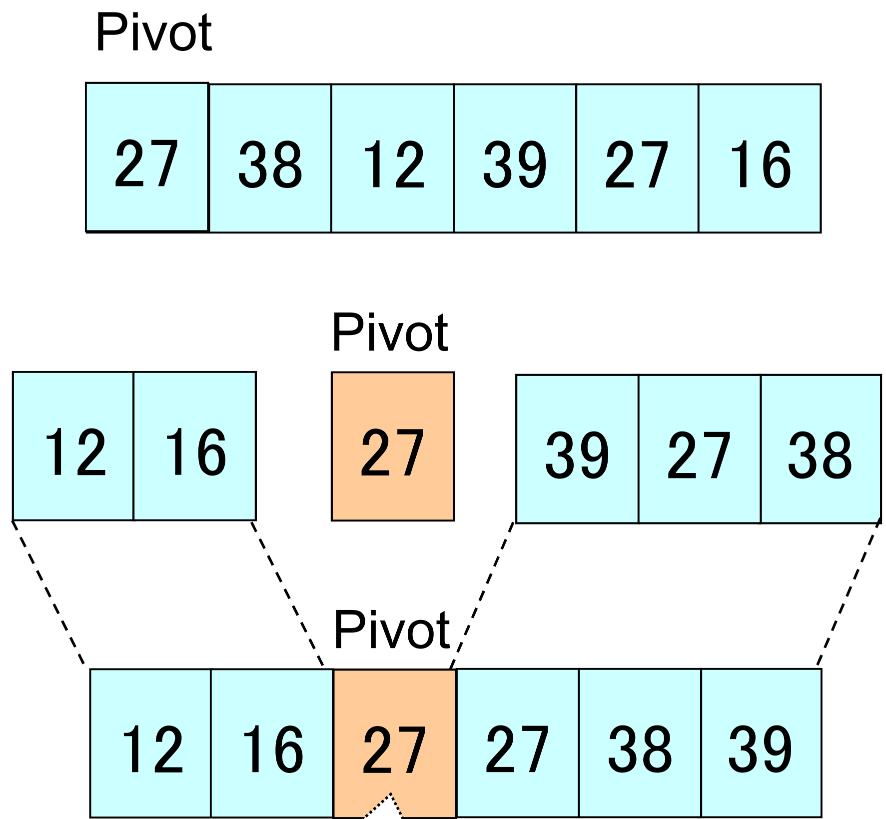
- **Quick Sort** is a divide-and-conquer algorithm
- **Divide Step:**
  - Choose an item ***p*** (known as **pivot**) and partition the items of `a[i..j]` into two parts:
    - Items that are smaller than ***p***
    - Items that are greater than or equal to ***p***
  - Recursively sort the two parts
- **Conquer Step:** Do nothing!
- Comparison:
  - **Merge Sort** spends most of the time in conquer step but very little time in divide step

Quick Sort : Divide Step Example

Choose first element as pivot

Partition **`a[]`** about the pivot **`27`**

Recursively sort the two parts



Notice anything special about the position of pivot in the final sorted items?

Quick Sort : Code

```
def quickSort ( array, low, high ):
    if low < high :
        pivotIdx = partition( array, low, high )

        quickSort( array, low, pivotIdx-1)
        quickSort( array, pivotIdx + 1, high )
```

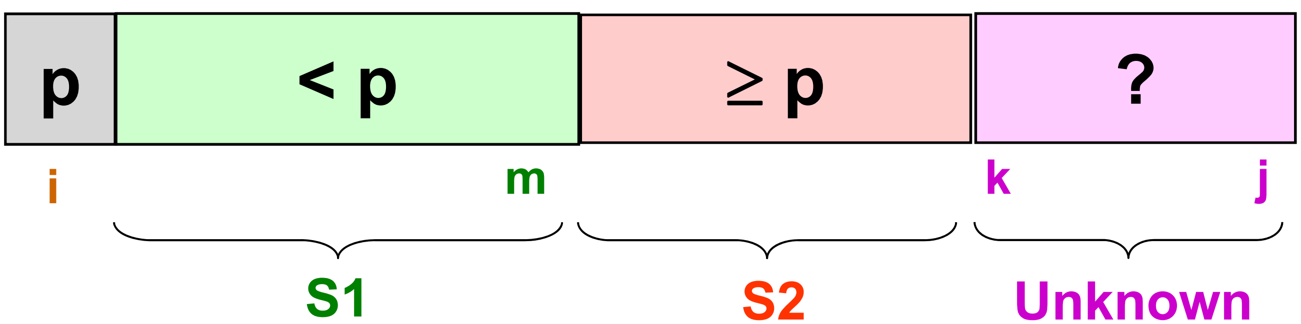
Partition **a[low...high]** and return the index of the pivot item

Recursively sort the two portions

- **partition()** splits **a[low...high]** into two portions
  - **a[low ... pivot – 1]** and **a[pivot + 1 ... high]**
- Pivot item does not participate in any further sorting

Quick Sort : Partition Algorithm

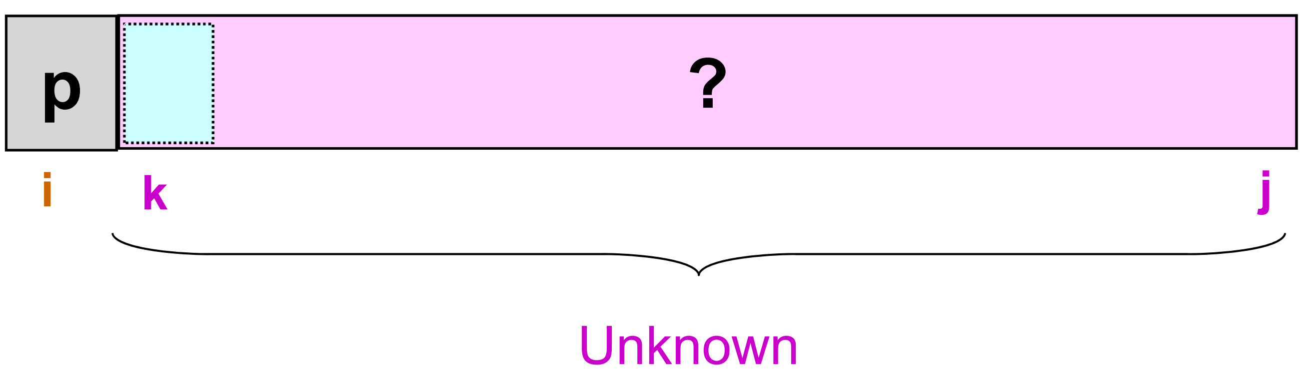
- To partition `a[i..j]`, we choose `a[i]` as the **pivot** `p`
- The remaining items (`a[i+1..j]`) are divided into three regions:
  - S1 = `a[i+1...m]` : items **<** **p**
  - S2 = `a[m+1...k-1]` : items ≥ **p**
  - Unknown = `a[k...j]` : items to be assigned to S1 or S2



Quick Sort : Partition Algorithm (2)

- Initially, regions **S1** and **S2** are empty
  - All items excluding `p` are in the unknown region
- For each item `a[k]` in the unknown region
  - Compare `a[k]` with `p`:
    - If `a[k]` **>=** `p`, put it into **S2**
    - Otherwise, put `a[k]` into **S1**

S1 = `a[i+1...m]` : items **<** **p**
S2 = `a[m+1...k-1]` : items ≥ **p**
Unknown = `a[k...j]`

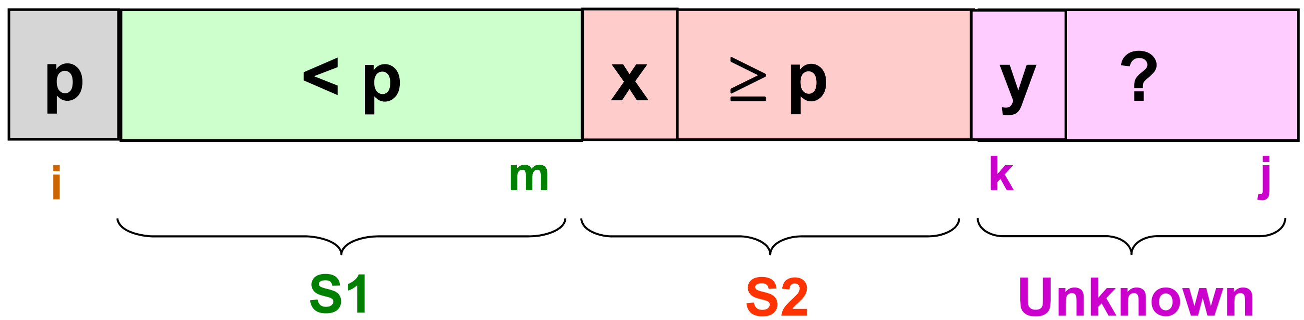


Quick Sort : Partition Algorithm (3)

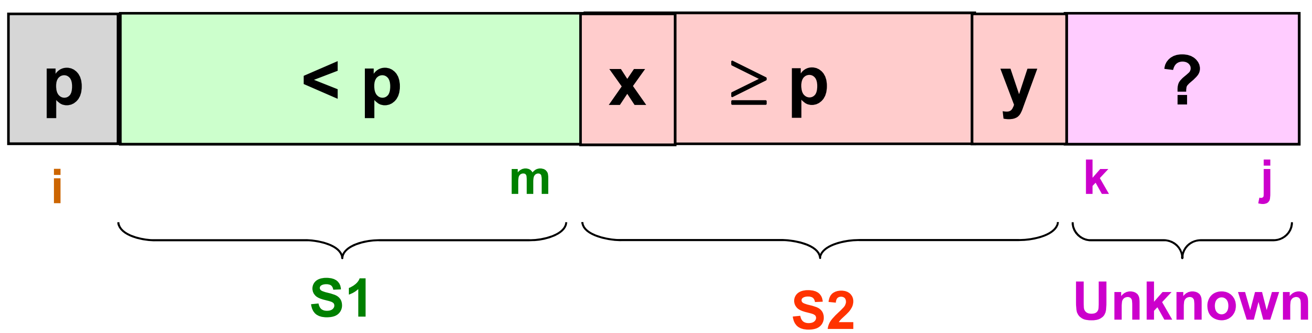
- Case 1: item `y` at `a[k]` >= `p`

S1 = `a[i+1...m]` : items < p
S2 = `a[m+1...k-1]` : items ≥ p
Unknown = `a[k...j]`

`if y ≥ p`



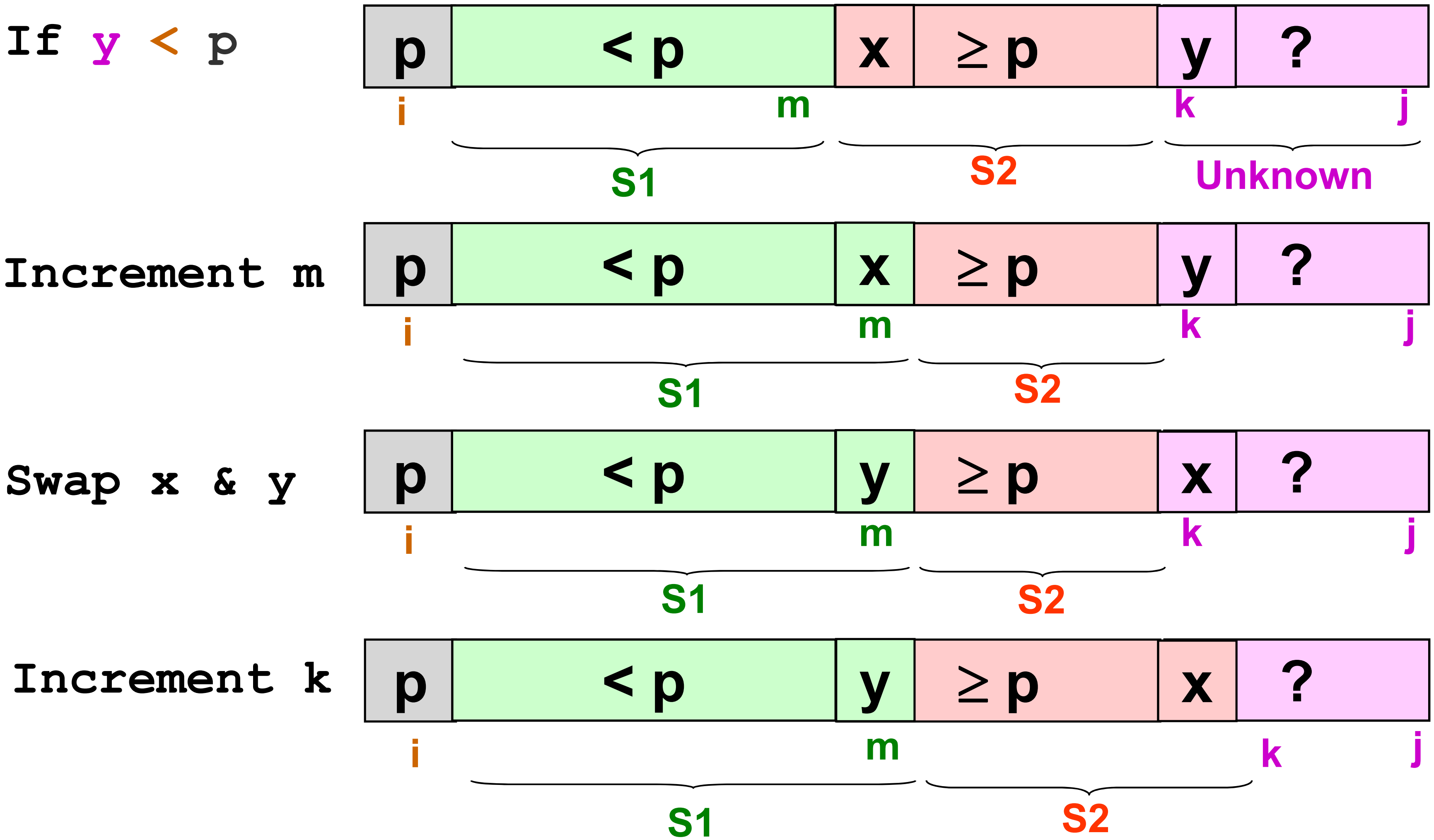
`Increment k`



Quick Sort : Partition

S1 = `a[i+1...m]` : items < p
S2 = `a[m+1...k-1]` : items ≥ p
Unknown = `a[k...j]`

- Case 2: item **y** at `a[k]` < `p`



Quick Sort : Partition Algorithm Code

```
def partition( array, i, j ):
    pivot = array[i];
    middle = i;


    for k in range (i+1,j+1):
        if array[k] < pivot:
            middle = middle + 1
            swapElement(array, k, middle)
       #else


    swapElement(array, i, middle);


    return middle
```

S1 and **S2** empty initially

Go through each element in unknown region

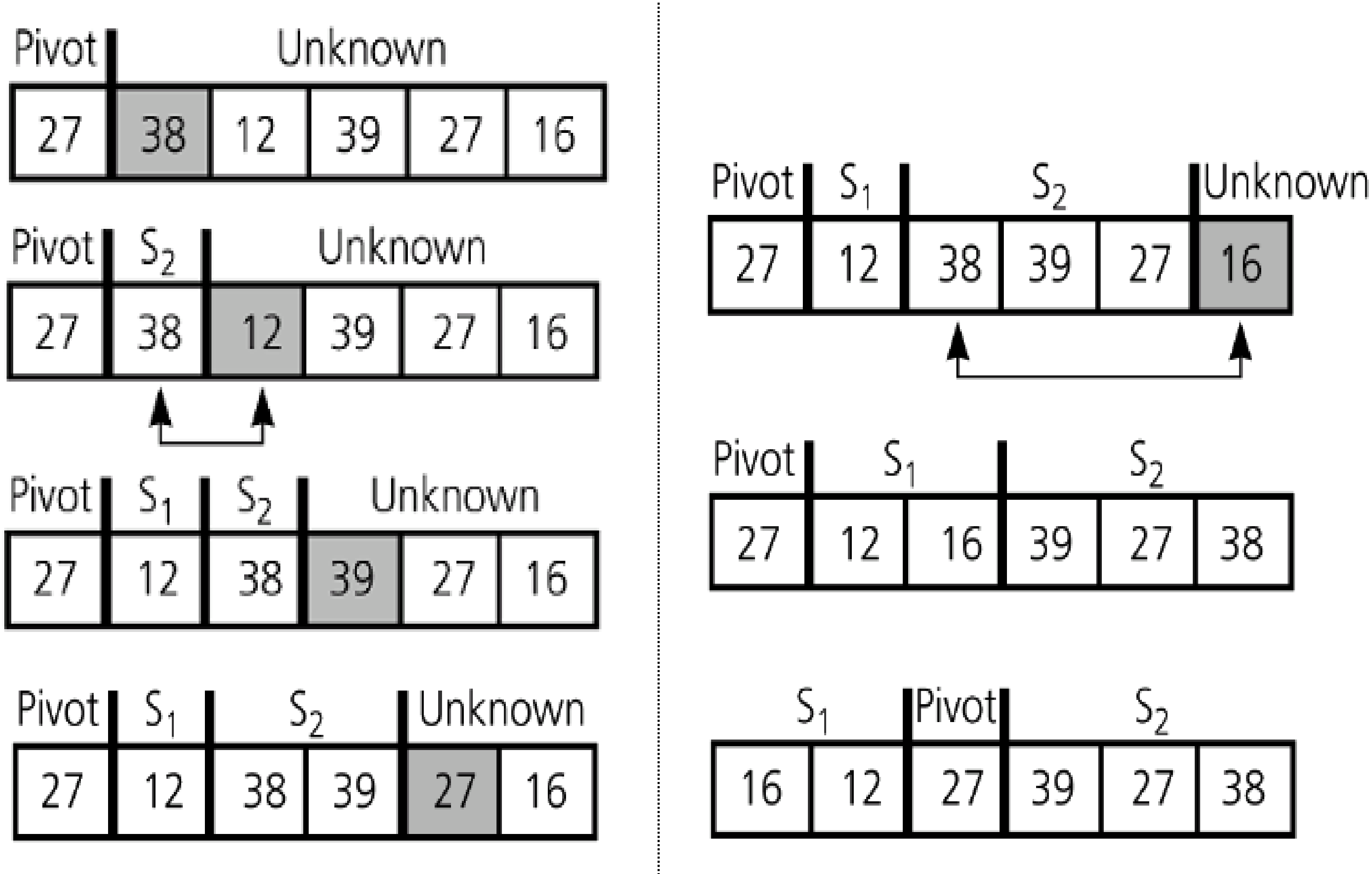
Case 2

Case 1: Do nothing!

Swap pivot with **a[m]**

return the index of pivot element

Quick Sort : Partition Example



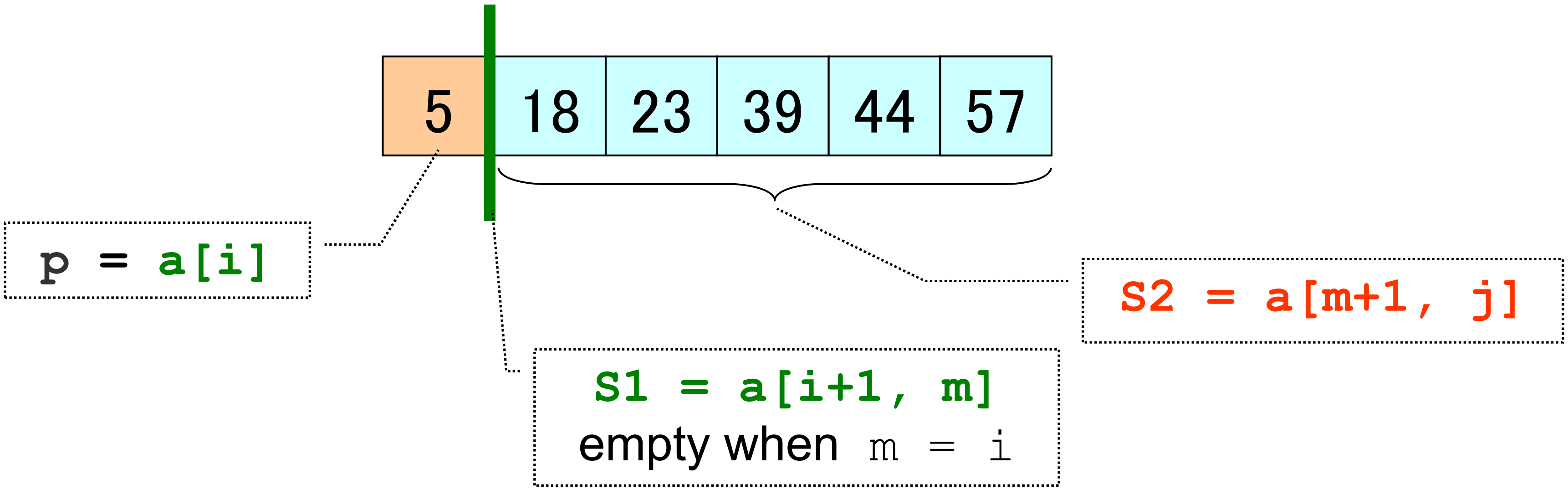
Quick Sort : ***`Partition()`*** Analysis

- There is only a single for-loop
  - Number of iterations = number of items, **`N`**, in the unknown region
    - **`N = high - low`**
  - Complexity is **`O(N)`**

- Similar to **Merge Sort**, the complexity is then dependent on the number of times ***`partition()`*** is called

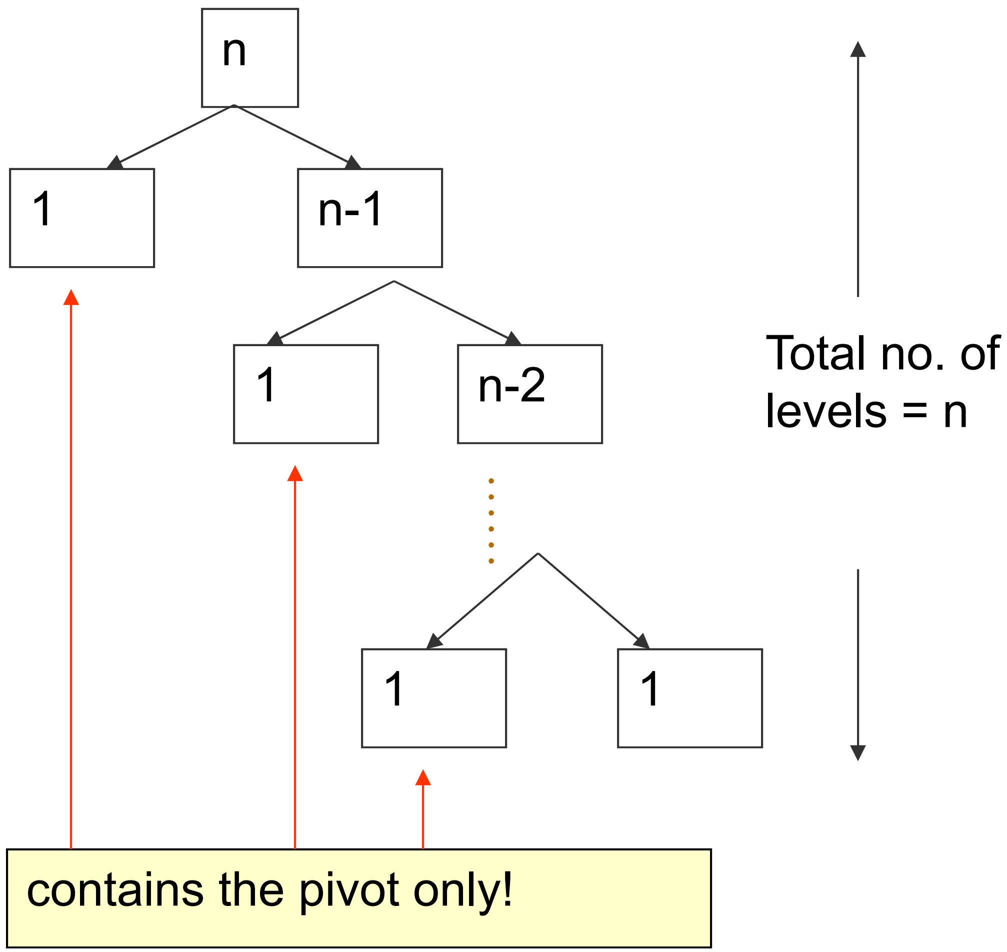
Quick Sort : Worst Case Analysis

- When the array is already in ascending order:



- What is the pivot index returned by `partition()`?
  - What is the effect of `swap(a, i, m)` ?
- **S1** is empty, while **S2** contains every item except the pivot…

Quick Sort : Worst Case Analysis (2)



As partition takes `O(n)` time, the algorithm in the worst case takes time `n+(n-1)+...+1 =` $O(n^2)$

- What is the main cause of this "disaster" ?

Quick Sort : Best/Average Case Analysis

- Best case occurs when partition always splits the array into two equal halves.
  - Depth of recursion is `lg n`
    - Recall the number of levels in merge sort?
  - Time complexity is `O(n lg n)`
- In practice, worst case is rare, and on the average we get some good splits and some bad ones.
  - Average time is `O(n lg n)`

Lower Bound: Comparison Based Sorting

- It is established that:
  - All **Comparison Based** sorting algorithms have a **complexity lower bound** of **`n lg n`**

- Any comparison based sorting algorithm with a **worst case complexity** **`O(n lg n)`**

  ➔ It is **optimal**

Sort without comparing number?!?!

RADIX SORT

Radix Sort : Idea

- Treats each data to be sorted as a *character string*
  - i.e. not using the numerical value directly

- It is not using comparison, i.e., **no comparison between the data is needed**

- In each iteration:
  - Organize the data into "bins" according to the ***next*** character in each data
  - The groups are then “concatenated” for next iteration

Radix Sort : Example

Start 0123, 2154, 0222, 0004, 0283, 1560, 1061, 2150

(1560,2150) (1061) (0222) (0123,0283) (2154,0004)

1560, 2150, 1061, 0222, 0123, 0283, 2154, 0004

(0004) (0222,0123) (2150,2154) (1560,1061) (0283)

0004, 0222, 0123, 2150, 2154, 1560, 1061, 0283

(0004,1061) (0123,2150,2154) (0222,0283) (1560)

0004, 1061, 0123, 2150, 2154, 0222, 0283, 1560

(0004,0123,0222,0283) (1061,1560) (2150,2154)

Sorted 0004, 0123, 0222, 0283, 1061, 1560, 2150, 2154

Radix Sort : Main Driver

```
def radixSort( array):
    numDigit = int(math.log10(max(array))) + 1

   for power in [10**i for i in range(numDigit)]:

       digitBin = [[] for d in range(10)]

       distribute(array, digitBin, power)

       collect(digitBin, array)
```

power = 1, 10, 100, 1000, ….., up to $10^{numDigit}$

- **distribute()**: Organize all items in **array** into "bins" based on specific digit, as indicated by the power
- **collect()**: Place items from the "bins" back into **array**, i.e. “concatenate” the groups

Radix Sort : Distribute

```
def distribute(array, digitBin, power):
    for item in array:
        digit = (item // power ) % 10

        digitBin[digit].append( item )
```

The item is placed in the bin as determined by the specific digit

- Use 1234 as example:
  - power = 1 (**1234** // 1) % 10 ➔ 1234 % 10 ➔ **4**
  - power = 10 (**1234** // 10) % 10 ➔ 123 % 10 ➔ **3**
  - power = 100 (**1234** // 100) % 10 ➔ 12 % 10 ➔ **2**
  - power = 1000 (**1234** // 1000) % 10 ➔ 1 % 10 ➔ **1**

Radix Sort : Collect

```
def collect(digitBin, array):
    startIdx = 0

    for eachBin in digitBin:
        array[startIdx:] = eachBin
        startIdx += len(eachBin)
```

- Basic Idea:
  - Start with digitBin[0]
    - Replace all items in array[0…..]
  - digitBin[1] replace all items in array[len(digitBin[0]……]
  - Repeat for each bin

Radix Sort : Analysis

- For each iteration
  - We go through each item once to place them into group
  - Then go through them again to concatenate the groups
  - Complexity is `O(n)`

- Number of iterations is `d`, the maximum number of digits (or maximum number of characters)
  - Complexity is thus `O(d n)`

The Good, The Bad and The Ugly: Sorting Algorithm Version

PROPERTIES OF SORTING

Property: In-Place Sorting

- A sort algorithm is said to be an **in-place** sort
  - if it has a **space complexity of O(1)** excluding the original list of numbers
  - i.e. it can only use **constant amount of extra space** during the sorting process

- Question:
  - Merge Sort is not in-place, why?
  - Is Radix Sort in-place?

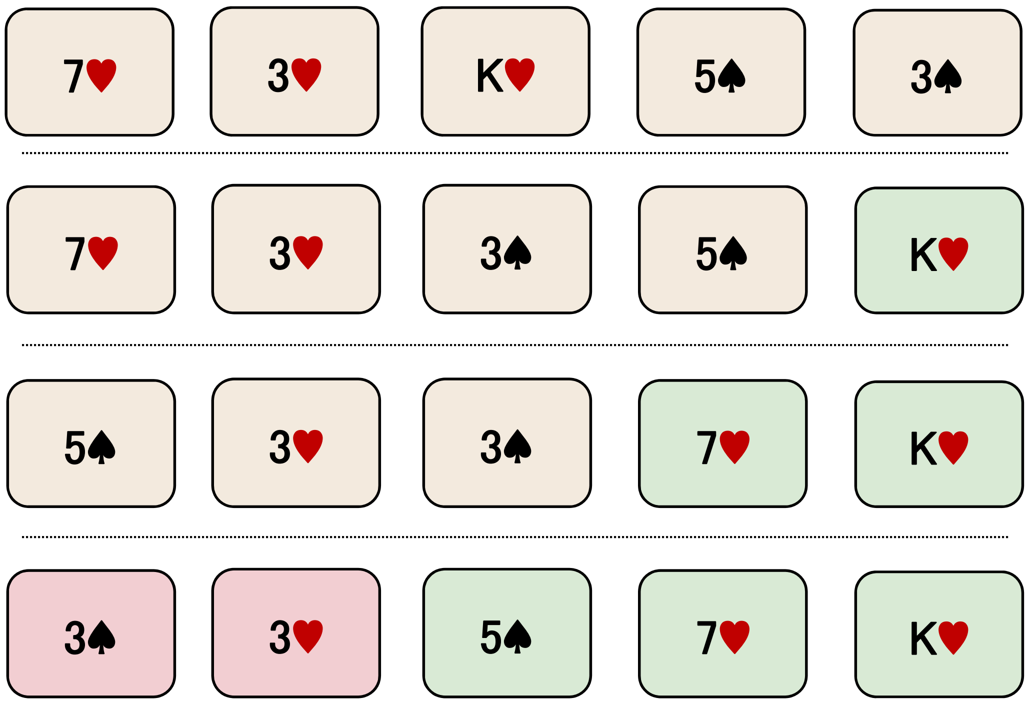
Property: Stable Sorting

- A sorting algorithm is **stable** if the **relative order** of **elements with the same key value** is preserved by the algorithm

- Example application of **stable sort**
  1. Assume that **names** have been sorted in alphabetical order
  2. Now, if this list is sorted again by **tutorial group number**, a stable sort algorithm would ensure that all students in the same tutorial groups still appear in alphabetical order of their names

Counter-Example: **Non-Stable Sort**

■ Selection Sort



Originally sorted by suit
♥ < ♠

Stable sorting should result in
3♥ < 3♠

Counter-Example: **Non-Stable Sort**

- **Quick Sort:**
  - During **partition** phase (1285 is the pivot)

1285	*5a*	150	4746	602	*5b*	8356

1285	*5a*	150	602	*5b*	4746	8356

5b	*5a*	150	602	1285	4746	8356

  - Observe that the last swapping operation (to put pivot in the right location) can break the ordering of similar items

Sorting Algorithms : **Summary**

	Worst Case	Best Case	In-place?	Stable?
Selection Sort	$O(n^2)$	$O(n^2)$	Yes	No
Insertion Sort	$O(n^2)$	$O(n)$	Yes	Yes
Bubble Sort	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble Sort 2	$O(n^2)$	$O(n)$	Yes	Yes
Merge Sort	$O(n \lg n)$	$O(n \lg n)$	No	Yes
Quick Sort	$O(n^2)$	$O(n \lg n)$	Yes	No
Radix sort	$O(dn)$	$O(dn)$	No	yes

- Note: as implemented in this lecture

Summary

- Comparison-Based Sorting Algorithms
  - Iterative Sorting
    - Selection Sort
    - Bubble Sort
    - Insertion Sort
  - Recursive Sorting
    - Merge Sort
    - Quick Sort
- Non-Comparison-Based Sorting Algorithms
  - Radix Sort
- Properties of Sorting Algorithms
  - In-Place
  - Stable