Sorting

CSC-114 Data Structure and Algorithms

Slides Credit: Ms. Saba Anwar, CIIT Lahore

Outline

- Sorting
  - Motivation
  - In-Place vs Non In-Place
  - Quadratic vs. Linearithmic
    - Insertion Sort
    - Selection Sort
    - Bubble Sort
    - Quick Sort
    - Merge Sort

Motivation

Sorting:

A process of arranging things in certain order

Motivation:

Few sample applications:

List of cities with largest populations

List of students in order of GPAs

Alphabetical list of words

Lexicographic order of dictionary

Files and folder arrangement in file explorer according to date, size, name etc.

0	**1**	**2**	**3**	**4**	**5**
8	20	15	-5	7	-55

$\downarrow$

0	**1**	**2**	**3**	**4**	**5**
-55	-5	7	8	15	20

In-Place vs. Not-In-Place Sorting

In-Place:

An in-place algorithm transforms an input into output with a constant amount of extra storage space

In the context of sorting, this means that the input array converted to sorted array without using a new array. Like

- Selection Sort
- Bubble Sort
- Insertion Sort
- Heap Sort

Not-In-Place:

Just opposite of in-place, requires amount of extra space proportional to size of input, in case of sorting it will be an additional array. Like

- Merge Sort

Stability

- If list contains many equal numbers, then there can be multiple correct sorted solutions
  - For example if we sort the list of three words {apple, chair, king} according to word length, then two possible solutions are:
    - {king, chair, apple}
    - {king, apple, chair}
- A sorting algorithm is stable if relative position of equal elements in output is equal to their relative position in input.
  - Stable sorted solution of above list is {king, apple, chair}

Time Complexity

- Quadratic→ O(n^2)
  - Selection Sort
  - Bubble Sort
  - Insertion Sort
- Linearithmic → O(nlogn)
  - Quick Sort
  - Merge Sort
  - Heap Sort
- Linear→ O(n)
  - Bucket Sort
  - Counting Sort
  - Radix Sort

Comparison vs Non Comparison Sorting

Sorting by comparing values as whole to each other. Complete data values are compared without any kind of assumptions about them.

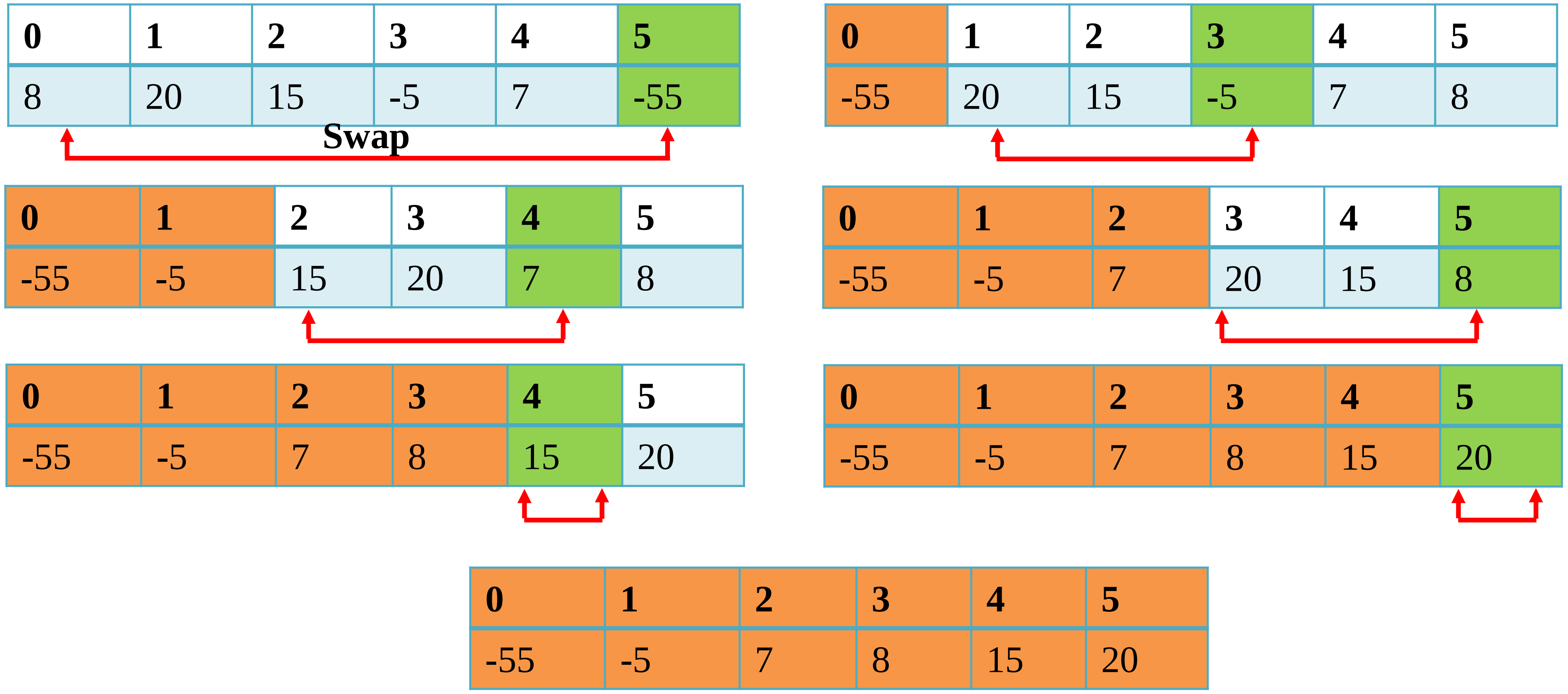
- Selection Sort
- Insertion Sort
- Bubble Sort
- Quick Sort
- Merge Sort
- Heap Sort

- Non-Comparison sorting sorts the numbers with partial comparison and may assume some constraints about data.
  - Bucket Sort
  - Counting Sort
  - Radix Sort

Selection Sort

- Iterative and in-place
- Main Idea
  - *Selects* the smallest element of the array and places it at index 0, then *selects* the second smallest and places it in index 1, then the third smallest in index 2, etc..
  - So at any time there are two partitions of list:
    - One that is sorted
    - One that is unsorted
- Advantages:
  - Very simple
  - Memory efficient: in-place means swapping elements within same array
- Disadvantages:
  - Slow: runs in quadratic O(n2) time

Selection Sort



0	1	2	3	4	5
8	20	15	-5	7	-55

0	1	2	3	4	5
-55	20	15	-5	7	8

0	1	2	3	4	5
-55	-5	15	20	7	8

0	1	2	3	4	5
-55	-5	7	20	15	8

0	1	2	3	4	5
-55	-5	7	8	15	20

0	1	2	3	4	5
-55	-5	7	8	15	20

0	1	2	3	4	5
-55	-5	7	8	15	20

Selection Sort

Algorithm: SELECTIONSORT(A[], N)
Input: an array of size N
Output: sorted array
Steps:

```
1.     Let pos_min=-1;
2.     for(count=0; count<N; count++)
3.         //find minimum
4.         pos_min=count //consider 1st element of remaining array minimum
5.         for(index=count; index < N; index++)
6.                 if (A[index]<A[pos_min])
7.                         pos_min=index
8.                 End if
9.         End loop
10.        //swap the min with value at count
11.        tmp=A[pos_min]
12.        A[pos_min]=A[count]
13.        A[count]=tmp
   1.  End loop
```

Insertion Sort

Iterative and in-place

Main Idea

- Arranges items one at a time by comparing each element with every element before it and inserting it into the correct position

Advantages:

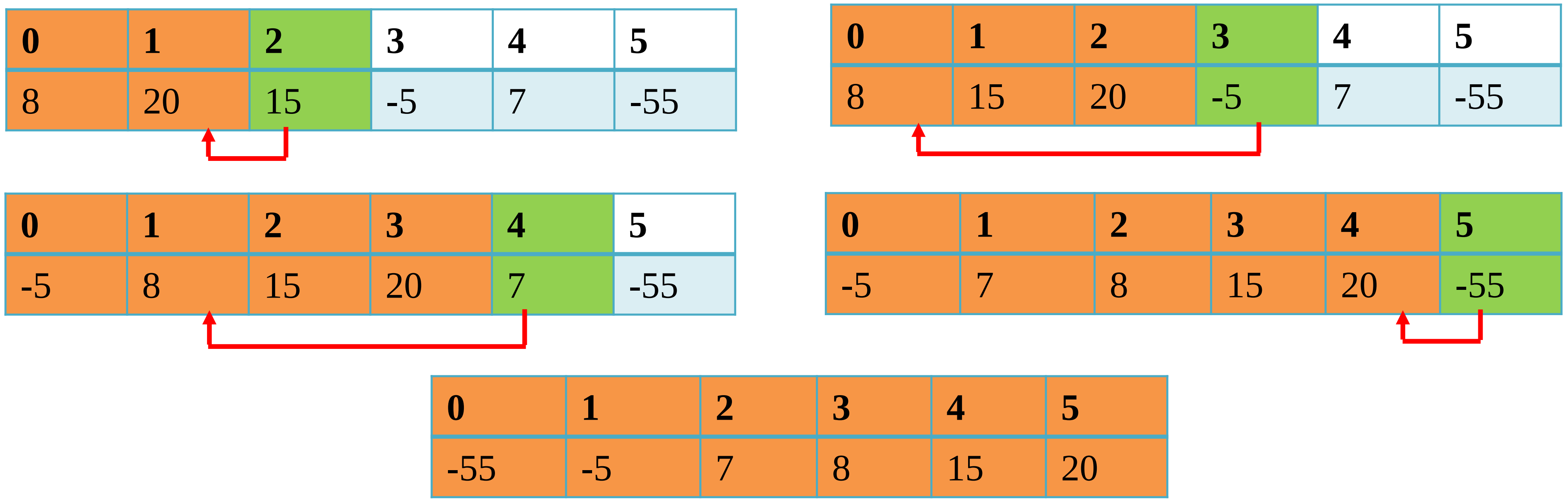
- Works particularly well if the list is already partially sorted
- Memory efficient: in-place

Disadvantages:

- Slow: runs in quadratic O(n2) time.

Insertion Sort

Find 1st number that is < its previous number, means it is incorrect position, find its correct location by comparing it numbers on left side. Shift numbers to right. Place number at founded location



0	**1**	**2**	**3**	**4**	**5**
8	20	15	-5	7	-55

0	**1**	**2**	**3**	**4**	**5**
8	15	20	-5	7	-55

0	**1**	**2**	**3**	**4**	**5**
-5	8	15	20	7	-55

0	**1**	**2**	**3**	**4**	**5**
-5	7	8	15	20	-55

0	**1**	**2**	**3**	**4**	**5**
-55	-5	7	8	15	20

Insertion Sort

Algorithm: INSERTIONSORT(A[], N)
Input: an array of size N
Output: sorted array
Steps:

```
1.  for(count=1; count<N; count++)
2.      v=A[count]
3.      j=count
4.      while(A[j-1]>v and j>0)          //find the correct position
5.              A[j] = A[j-1])
6.              j--
7.      End While
8.      A[j]=v                           //store v at correct position

9.  End loop
```

Bubble Sort

- Iterative and in-place
- Main Idea
  - Arranges items from first element to last, comparing each pair of elements and swapping their positions if needed.
- Advantages:
  - Very simple
  - Memory efficient: in-place
- Disadvantages:
  - Slow: runs in quadratic O(n2) time.

Bubble Sort

First iteration

0	1	2	3	4	5
8	20	15	-5	7	-55

0	1	2	3	4	5
8	20	15	-5	7	-55

0	1	2	3	4	5
8	15	20	-5	7	-55

0	1	2	3	4	5
8	15	-5	20	7	-55

0	1	2	3	4	5
8	15	-5	7	20	-55

0	1	2	3	4	5
8	15	-5	7	-55	20

One number is placed at right position. Now we need to repeat the same process on remaining list (Orange part is sorted)

Bubble Sort

2nd iteration

0	1	2	3	4	5
8	15	-5	7	-55	20

0	1	2	3	4	5
8	15	-5	7	-55	20

0	1	2	3	4	5
8	-5	15	7	-55	20

0	1	2	3	4	5
8	-5	7	15	-55	20

0	1	2	3	4	5
8	-5	7	-55	15	20

Another number is placed at right position. Now sorted list contain two numbers.

Bubble Sort

3rd iteration

0	1	2	3	4	5
8	-5	7	-55	15	20

0	1	2	3	4	5
-5	8	7	-55	15	20

0	1	2	3	4	5
-5	7	8	-55	15	20

0	1	2	3	4	5
-5	7	-55	8	-15	20

4th iteration

0	1	2	3	4	5
-5	7	-55	8	-15	20

0	1	2	3	4	5
-5	7	-55	8	15	20

0	1	2	3	4	5
-5	-55	7	8	15	20

Bubble Sort

5th iteration

0	1	2	3	4	5
-5	-55	7	8	15	20

0	1	2	3	4	5
-55	-5	7	8	15	20

6th iteration

0	1	2	3	4	5
-55	-5	7	8	15	20

Bubble Sort

Algorithm: BUBBLESORT(A[], N)
Input: an array of size N
Output: sorted array
Steps:

```
1.    for(count=N; count>=0; count--) //Outer loop to count iteration
2.      for(index=0; index < count; index++) //Inner loop for swaps
3.              if (A[index]<A[index+1])
4.                      temp = A[index]
5.                      A[index] = A[index+1]
6.                      A[index+1] = temp
7.              End if
8.      End loop
9.
10. End loop
```

Divide and Conquer

Divide-and-conquer is a general algorithm design paradigm:

- **Divide**: divide the input data S into disjoint subsets S_1, S_2, …, S_k
- **Recur**: solve the sub problems associated with S_1, S_2, …, S_k
- **Conquer**: combine the solutions for S_1, S_2, …, S_k into a solution for S

The base case for the recursion is generally sub problems of size 0 or 1

There are certain sorting algorithms which works in this way:

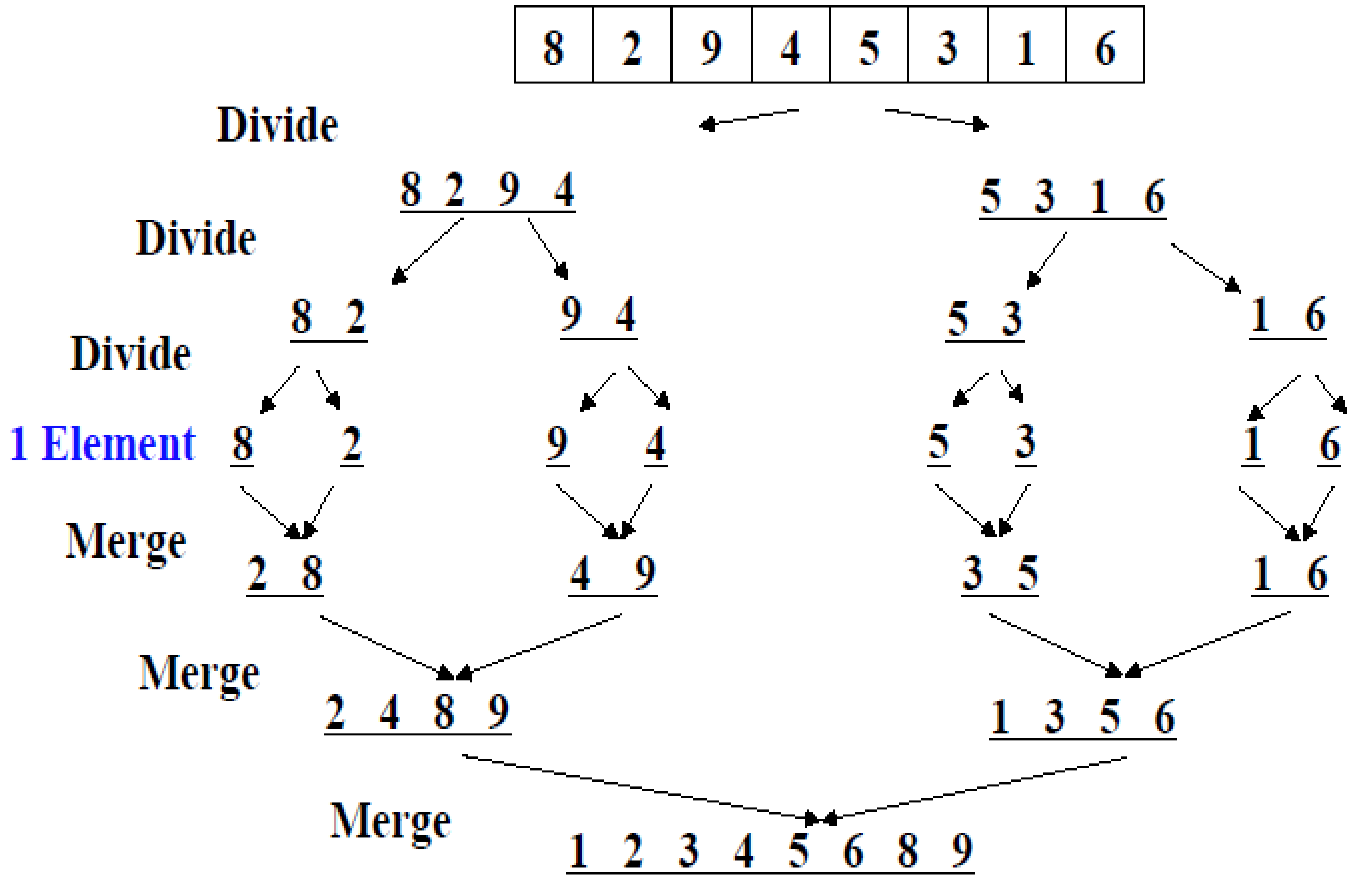
- Merge sort
- Quick sort

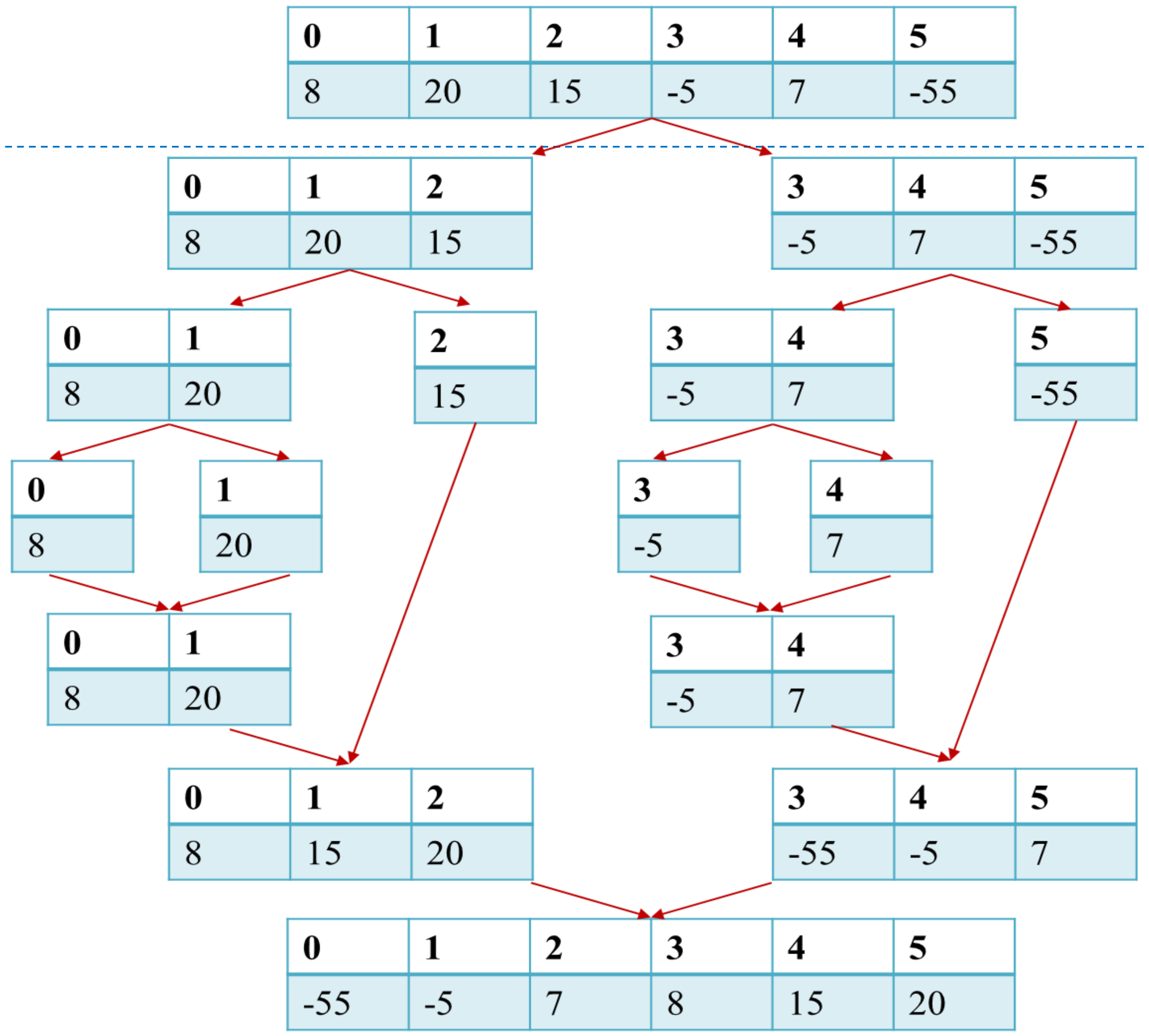
Merge Sort

- Merge sort is a sorting algorithm based on the divide-and-conquer paradigm
  - Like the quadratic sorts, merge sort is comparative
  - Merge sort is recursive and runs in O(n log n) time
- Main Idea:
  - **Divide**: partition list A of size N into two halves left and right of size N/2
  - **Recur**: sort the left and right half recursively
  - **Conquer**: merge the sorted left and right halves into a sorted sequence
- Advantages
  - Efficient, requires nlogn time
- Disadvantages
  - Not in-place, require O(N) extra space for merging
  - In-Place solutions do exist but more complex

Merge Sort

1. If a list has 1 element or 0 elements it is sorted
2. If a list has more than 2 elements, split into 2 separate lists
3. Perform this algorithm on each of those smaller lists
4. Take the 2 sorted lists and merge them together




0 1 2 3 4 5
8 20 15 -5 7 -55
0 1 2
8 20 15
3 4 5
-5 7 -55
0 1
8 20
2
15
3 4
-5 7
5
-55
0
8
1
20
3
-5
4
7
0 1
8 20
3 4
-5 7
0 1 2
8 15 20
3 4 5
-55 -5 7
0 1 2 3 4 5
-55 -5 7 8 15 20

Merge Sort

Algorithm: MERGESORT(A[], start, end)

Steps:

```
1. If (end-start+1)<2 //if size of array is 0 or 1
2.        return A
3. Else
4.  mid=(start+end)/2
5.  MERGESORT(A, start, mid)
6.  MERGESORT(A,mid+1, end)
7.  MERGE(A, start, mid,end)
8. End if
```

Merge Sort

MERGE(A[], start, mid, end)

```
1.  n1 = mid-start+1
2.  n2=end-mid
3.  left[n1], right[n2]
4.  i=start, j=mid+1
5.  //make copy of left and right half
6.  For i=0;  i<n1; i++
7.    left[i]=A[start+i]
8.  End For
9.  For j=0;j<n2;j++
10.   right[j]=arr1[j+mid+1]
11. End For
12. i=0, j=0, k=start
```

```
1.  While  i<n1 and j<n2
2.    if left[i]<=right[j]
3.        A[k]=left[i]
4.        i=i+1
5.    Else
6.       A[k]=right[j]
7.       j=j+1
8.    End if
9.    k=k+1
10. End While

11. While i<n1 //copy rest of left half
12.     A[k]=left[i]
13.     k++, i++
14. End While
15. While j<n2 //copy rest of right half
16.     A[k]=right[j]
17.     k++, j++
18. End While
```

Quick Sort

Quick sort is another sorting algorithm based on the divide-and-conquer paradigm.

- **Divide**: pick an element as pivot and partition list A into
  - elements less than pivot
  - elements greater than pivot
- **Recur**: quicksort both divisions
- **Conquer**: a sorted list in order of less than pivot, pivot and greater than pivot elements

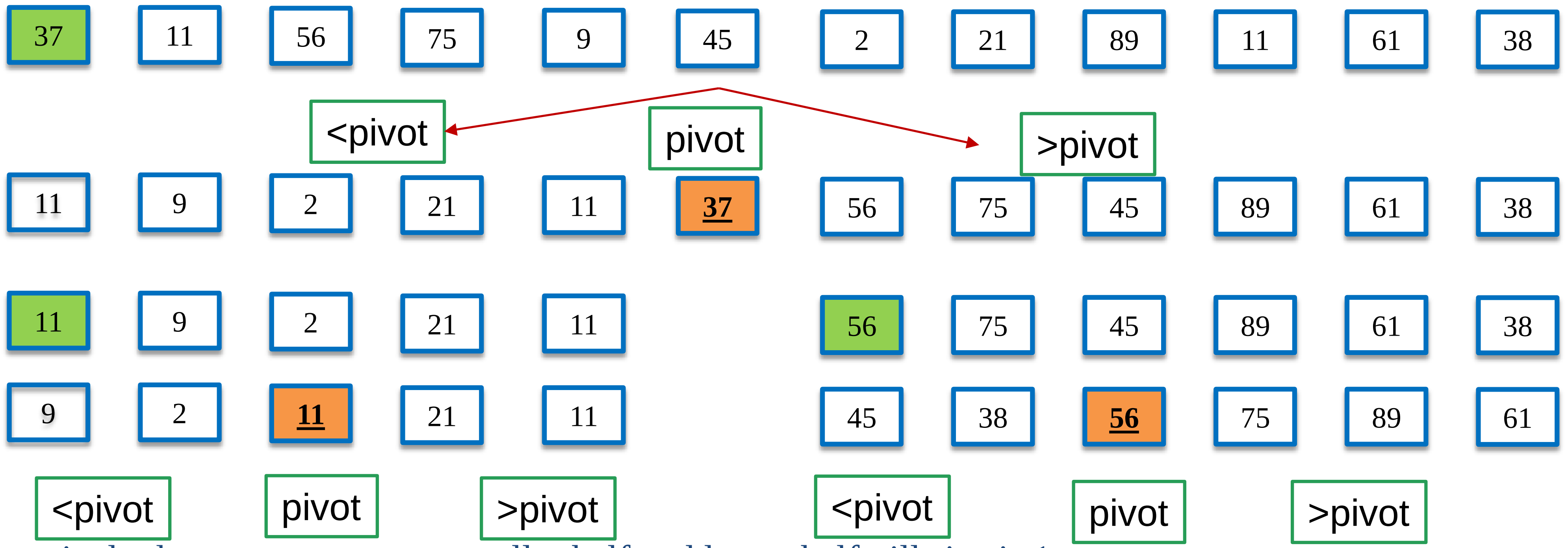
- Although merge sort is O(n log n), it is quite inconvenient for implementation with arrays, since we need extra space to merge.

In practice, the fastest sorting algorithm is Quicksort, which uses partitioning as its main idea.

Quick Sort

Let say 1st element is selected as pivot.

- Get the smaller numbers
- Get the larger numbers



Recursively do same process on smaller half and larger half, till size is 1.

Sorted list= smaller + Pivot + Larger

Quick Sort

37	11	56	75	9	45	2	21	89	11	61	38
11	9	2	21	11	**37**	56	75	45	89	61	38
11	9	2	21	11	**37**	56	75	45	89	61	38
9	2	**11**	21	11	**37**	45	38	**56**	75	89	61
9	2	**11**	21	11	**37**	45	38	**56**	75	89	61
2	**9**	**11**	11	**21**	**37**	38	**45**	**56**	61	**75**	61
2	**9**	**11**	11	**21**	**37**	38	**45**	**56**	61	**75**	61
2	**9**	**11**	**11**	**21**	**37**	**38**	**45**	**56**	**61**	**75**	**61**

Quick Sort

- How to choose pivot?
  - First element or Last element
  - Middle element
  - Randomly selected element
  - Mid of low, mid and high index
- How to do partition?
  - Elements that are less than pivot are shifted to left side of pivot
  - Elements that are greater than pivot are shifted to right side of pivot
- Different variations are present.
  - Two commonly used algo's are discussed here.

Quick Sort-1

Algorithm: QUICKSORT(A[], start, end)

Steps:

```
1.  If (start>=end)//base case, arraysize is 0 or 1
2.      return
3.  Else
4.    pivot_index=PARTITION(A, start, end)
5.    QUICKSORT(A, start, pivot_index-1)
6.    QUICKSORT(A,pivot_index+1, end)
7.  End if
```

This is known as lomuto partitioning scheme.

https://en.wikipedia.org/wiki/Quicksort#Lomuto_partition_scheme

PARTITION(A[], left, right)

```
1.  pivot=A[right]
2.  //shift numbers, most tricky part
3.   i=left;  j=left
4.  while (j<right) //start from left
5.      If A[j]<pivot //move smaller numbers to
    left of list
6.          swap(A[i],A[j])
7.           i=i+1
8.      End If
9.        j=j+1
10.   End While
11.    swap(A[i], pivot)
12.     return I
// i is the pivot location after moving smaller and
larger numbers
```

Quick Sort-1

```
Pivot=right, i=left, j=left
While(j<right)
    If(A[j] <= pivot)
        Swap (A[i], A[j])
        increment I
    increment j
Swap (A[i], pivot)
return i //→pivot_index
```

left, i, j — right

0	1	2	3	4	5
8	20	15	-5	7	-55

left, i — j (4) — right

0	1	2	3	4	5
8	20	15	-5	7	-55

0	1	2	3	4	5
-55	20	15	-5	7	8

Quick Sort-1

```
Pivot=A[right], i=left, j=left
While(j<right)
    If(A[j] <= pivot)
        Swap (A[i], A[j])
        increment I
    increment j
Swap (A[i], pivot)
return i //→pivot_index
```

	left, i, j				right
0	**1**	**2**	**3**	**4**	**5**
-55	20	15	-5	7	8

	left, i		j		right
0	**1**	**2**	**3**	**4**	**5**
-55	20	15	-5	7	8

	left	i		j	right
0	**1**	**2**	**3**	**4**	**5**
-55	-5	15	20	7	8

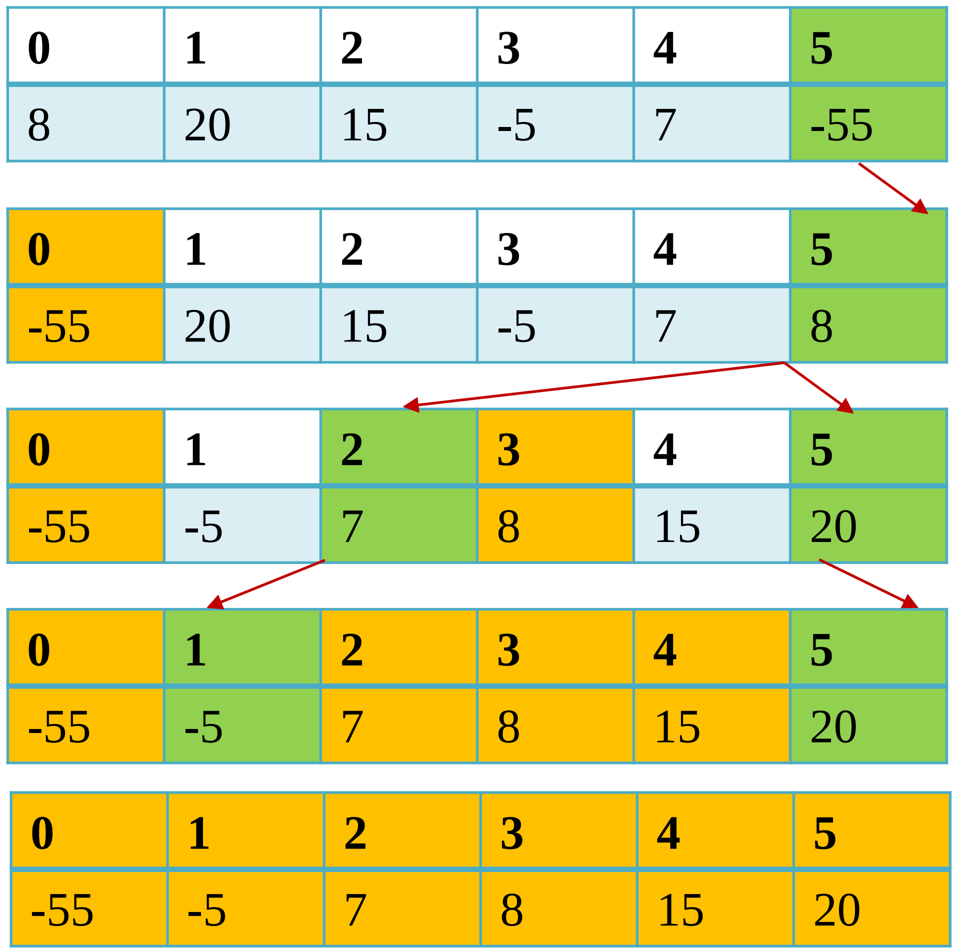
	left		i	j	right
0	**1**	**2**	**3**	**4**	**5**
-55	-5	7	20	15	8

0	**1**	**2**	**3**	**4**	**5**
-55	-5	7	8	15	20

Quick Sort-1

Complete Working

Array is shown after each call to Partition method



0	1	2	3	4	5
8	20	15	-5	7	-55

0	1	2	3	4	5
-55	20	15	-5	7	8

0	1	2	3	4	5
-55	-5	7	8	15	20

0	1	2	3	4	5
-55	-5	7	8	15	20

0	1	2	3	4	5
-55	-5	7	8	15	20

All steps for 1st call are shown in detail. For Subsequent calls only input and final output is shown

5	8	1	9	2	11	3	10	7

1st call to Partition method

left i j								right
5	8	1	9	2	11	3	10	7

5<pivot, swap with itself

left	i j							right
5	8	1	9	2	11	3	10	7

8>pivot, no swapping

left	i	j						right
5	8	1	9	2	11	3	10	7

1<pivot, swap with 8

left		i	j					right
5	1	8	9	2	11	3	10	7

p>pivot, no swap

left		i		j				right
5	1	8	9	2	11	3	10	7

2<pivot, swap with 8

left			i		j			right
5	1	2	9	8	11	3	10	7

11>pivot, no swap

left			i			j		right
5	1	2	9	8	11	3	10	7

3<pivot, swapped with 9

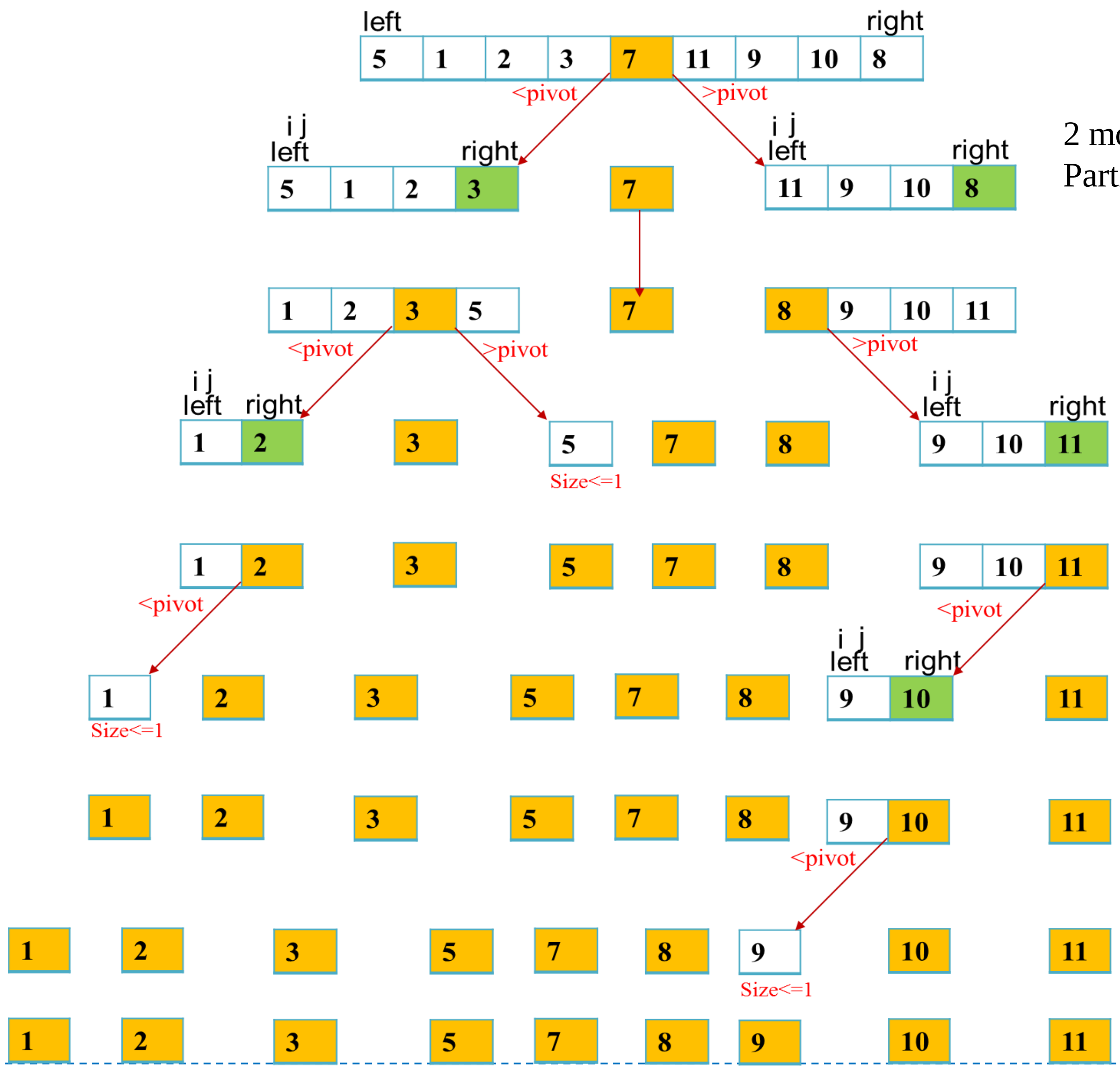
left				i			j	right
5	1	2	3	8	11	9	10	7

11>pivot, no swap

left				i				j right
5	1	2	3	8	11	9	10	7

j=right, stop. Swap pivot with 8 and return

5	1	2	3	7	11	9	10	8



2 more calls to Partition method

2 more calls to Partition method

1 more calls to Partition method

Quick Sort-2

Algorithm: QUICKSORT(A[], start, end)

Steps:

```
1.  If (start>=end)//base case, arraysize is 0 or 1
2.      return
3.  Else
4.    pivot_index=PARTITION(A, start, end)
5.    QUICKSORT(A, start, pivot_index)
6.    QUICKSORT(A,pivot_index+1, end)
7.  End if
```

A variation of Hoare's partitioning scheme.

https://en.wikipedia.org/wiki/Quicksort#Hoare_partition_scheme

PARTITION(A[], left, right)

```
1.   pivot=A[left]
2.   //shift numbers, most tricky part
3.    i=start;  j=end
4.   while (true)
5.     while (A[i]< pivot )
6.         i++
7.     while  (A[j] >pivot)
8.         j--
9.      If i<j      //swap
10.         temp=A[i]
11.         A[i]=A[j]
12.         A[j]=temp
            if(A[i]==A[j]) //otherwise loop will never end
              i=i+1
1.      Else
2.        return j
3.     End if
4.   End While
```

Quick Sort-2

```
Pivot=A[left], i=left, j=right
While(true)
    While(A[i]<pivot)
        i=i+1
    While(A[j]>pivot)
        j=j-1
    If(i<j)
        Swap (A[i], A[j])
    Else
        return j//→pivot_index
```

left, i					right, j
0	**1**	**2**	**3**	**4**	**5**
8	20	15	-5	7	-55

left, i					right, j
0	**1**	**2**	**3**	**4**	**5**
-55	20	15	-5	7	8

left	i			j	right
0	**1**	**2**	**3**	**4**	**5**
-55	8	15	-5	7	20

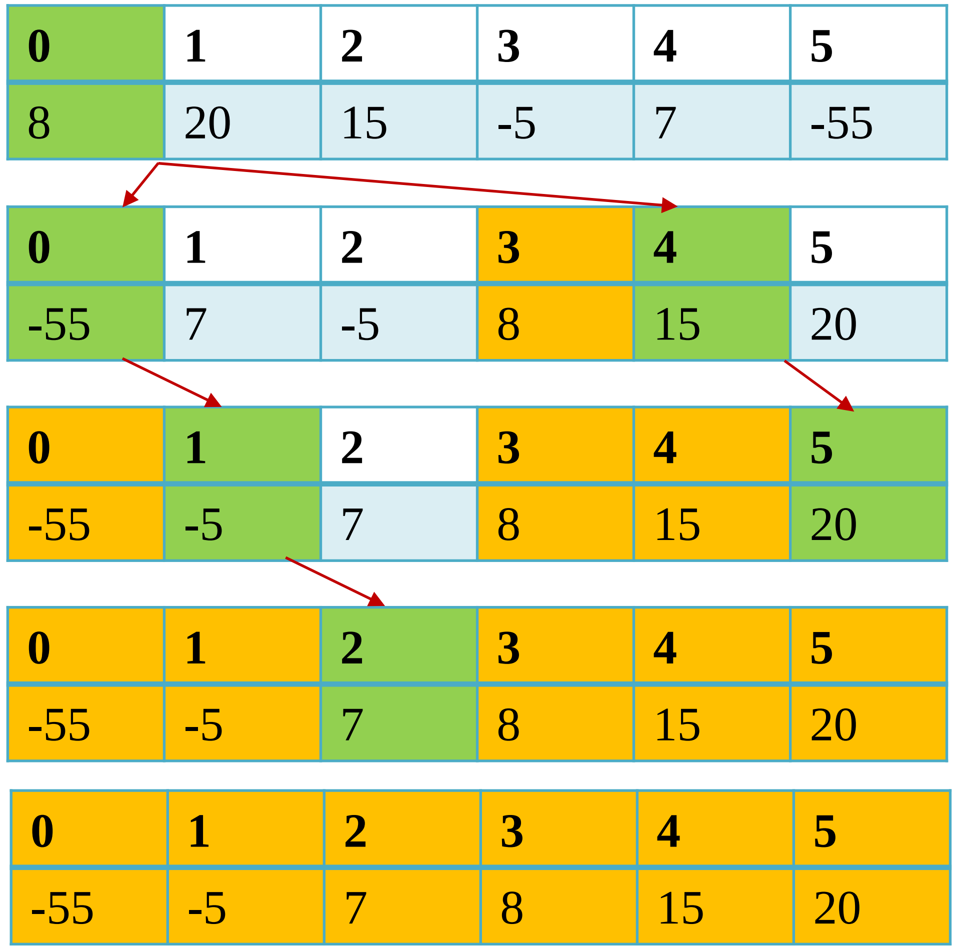
left		i		j	right
0	**1**	**2**	**3**	**4**	**5**
-55	7	15	-5	8	20

left		i	j		right
0	**1**	**2**	**3**	**4**	**5**
-55	7	8	-5	15	20

left		i	j		right
0	**1**	**2**	**3**	**4**	**5**
-55	7	-5	8	15	20

Quick Sort-2

Complete Working



0	1	2	3	4	5
8	20	15	-5	7	-55

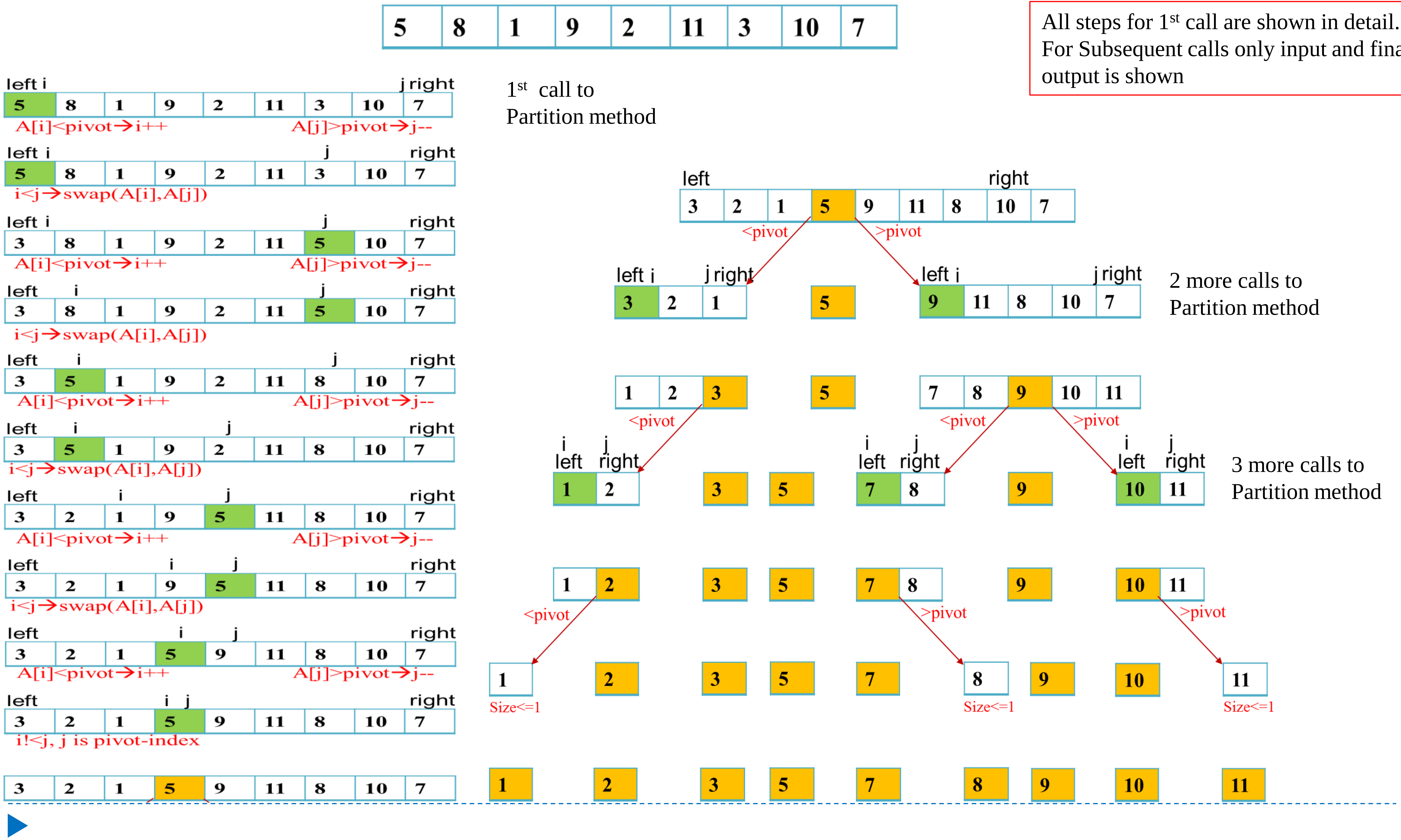
0	1	2	3	4	5
-55	7	-5	8	15	20

0	1	2	3	4	5
-55	-5	7	8	15	20

0	1	2	3	4	5
-55	-5	7	8	15	20

0	1	2	3	4	5
-55	-5	7	8	15	20

All steps for 1st call are shown in detail.
For Subsequent calls only input and final output is shown



Heap Sort

Heap sort is another recursive algorithm to sort the numbers using heap concept.

Main Idea:

- Build the heap from given list of numbers
- Repeatedly swap the root with last element of heap until list is sorted.

Advantages

- Time efficient-O(NlogN)
- In-Place

Disadvantages

- Uses an additional data structure-heap

Heap Sort

1st step: Building Heap

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
23	1	6	19	14	18	8	24	15

1. Start from middle location and go to start
2. Assume it as parent
3. Find index of maximum of parent, its left and right child
4. If parent is not maximum
5. swap parent with maximum
6. repeat step(2) on maximum index

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
23	1	6	19	14	18	8	24	15

Heap Sort

Middle location=4 to 0

- P=4, No left, right child

0	1	2	3	4	5	6	7	8
23	1	6	19	14	18	8	24	15

- P=3, L=7, R=8
  - Max= 7
  - Swap P and Max
  - Now P=Max=7, no left, right child

0	1	2	3	4	5	6	7	8
23	1	6	19	14	18	8	24	15

0	1	2	3	4	5	6	7	8
23	1	6	24	14	18	8	19	15

- P=2, L=5,R=6
  - Max=5
  - Swap P and Max
  - Now P=5, no left, right child

0	1	2	3	4	5	6	7	8
23	1	6	24	14	18	8	19	15

0	1	2	3	4	5	6	7	8
23	1	18	24	14	6	8	19	15

Heap Sort

P=1, L=3,R=4

- Max=3
- Swap P and Max
- Now P=3, L=7, R=8
  - Max=7
  - Swap P and Max
  - Now P= 7, No left, right

P=0, L=1,R=2

- Max=1
- Swap P and Max
- Now P= 1, L=3, R=4
- Max=1=P, No need of swapping,

List is finished

0	1	2	3	4	5	6	7	8
23	1	18	24	14	6	8	19	15

0	1	2	3	4	5	6	7	8
23	24	18	1	14	6	8	19	15

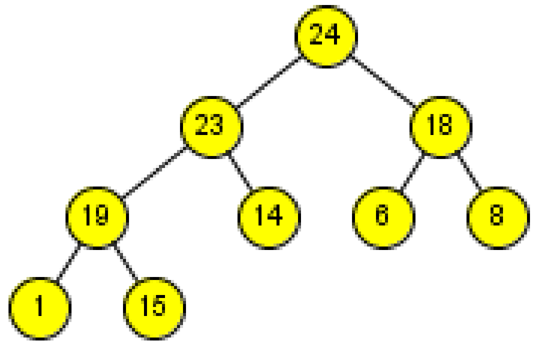
0	1	2	3	4	5	6	7	8
23	24	18	19	14	6	8	1	15

0	1	2	3	4	5	6	7	8
23	24	18	19	14	6	8	1	15

0	1	2	3	4	5	6	7	8
24	23	18	19	14	6	8	1	15

Heap Sort

Array is converted into a heap.



0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
23	1	6	19	14	18	8	24	15

$\downarrow$

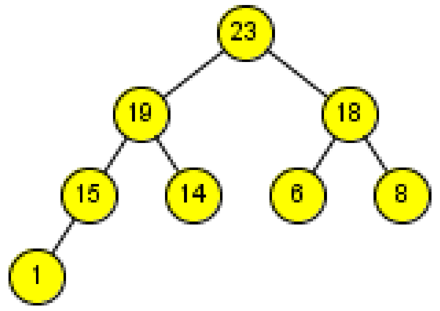
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
24	23	18	19	14	6	8	1	15

Now comes the 2nd part.

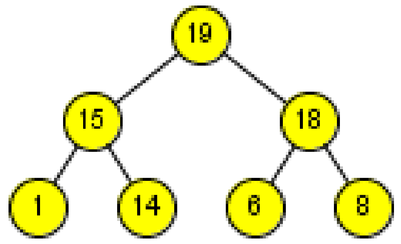
- Swap the root with last element of heap
  - 15 will go on top
  - Heap size will be reduced.
  - Heapify 15 to rebuild heap order.
- Repeatedly do the same process, until heap size becomes 1.

Heap Sort

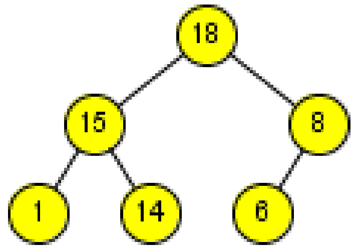
Swap root with last element of heap, rebuild heap



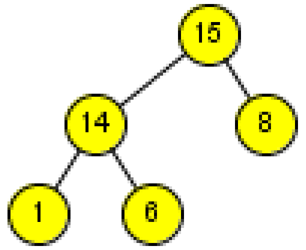
0	1	2	3	4	5	6	7	8
23	19	18	15	14	6	8	1	24



0	1	2	3	4	5	6	7	8
19	15	18	1	14	6	8	23	24

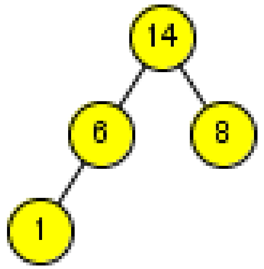


0	1	2	3	4	5	6	7	8
18	15	8	1	14	6	19	23	24

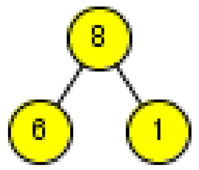


0	1	2	3	4	5	6	7	8
15	14	8	1	6	18	19	23	24

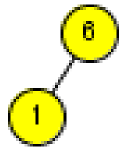
Heap Sort



0	1	2	3	4	5	6	7	8
14	6	8	1	15	18	19	23	24



0	1	2	3	4	5	6	7	8
8	6	1	14	15	18	19	23	24



0	1	2	3	4	5	6	7	8
6	1	8	14	15	18	19	23	24

0	1	2	3	4	5	6	7	8
1	6	8	14	15	18	19	23	24

0	1	2	3	4	5	6	7	8
1	6	8	14	15	18	19	23	24

Heap Sort

Algorithm: HEAP_SORT(A[], N)

```
1.  buildMaxHeap(A,N)
2.  For i= N-1; i>1; i--
3.      swap(A[0], A[i])
4.      maxHeapify(A, i, 0)
5.  End For
```

Algorithm: buildMaxHeap(A[], N)

```
1.  For i= N/2; i>=0; i--
2.      maxHeapify(A, N, i)
3.  End For
```

Algorithm: maxHeapify(A[], N, i)

```
1.  L=2*i+1
2.  R=2*i+2
3.  if  l<N and A[l]>A[i]
4.     largest=L
5.  else
6.     largest= i
7.  If r<N and A[r]> A[largest]
8.     largest=r
9.  If(largest != i)
10.       Swap(A[i], A[largest])
11.       maxHeapifyDown(A, largest)
12. End If
```

Comparison

	Worst Case	Best Case	In-Place	Stable
Selection	$O(n^2)$	$O(n^2)$	Yes	No
Insertion	$O(n^2)$	$O(n^2)$	Yes	Yes
Bubble	$O(n^2)$	$O(n^2)$	Yes	Yes
Quick	$O(n^2)$	O(n logn)	Yes	No
Merge	O(n logn)	O(n logn)	No	Yes
Heap	O(n logn)	O(n logn)	Yes	No

http://www.cprogramming.com/tutorial/computersciencetheory/sortcomp.html

Counting Sort

Counting Sort assumes that input data is in range of 0-k, Where k is a positive integer. So the highest value is k

Main Idea:

- Count frequency of each element j, $1 \leq j \leq k$.
  - Store results in an array C.
- Counting information from C can be used to determine the position of each element in the sorted array.
  - Modify the values of the C[j] so that now C[j] = the number of keys less than or equal to j.
  - that is C[j]=C[j]+C[j-1]+C[j-2]….
- Now there is a mystery. Use a new array B of same size as A.
  - If we pick the value A[j], let's call it m, now pick the value C[m], let's call it n.
    - B[n]=A[j] → element located at j is placed in its correct position of B.
    - Reduce the count of C[A[j]] by 1.

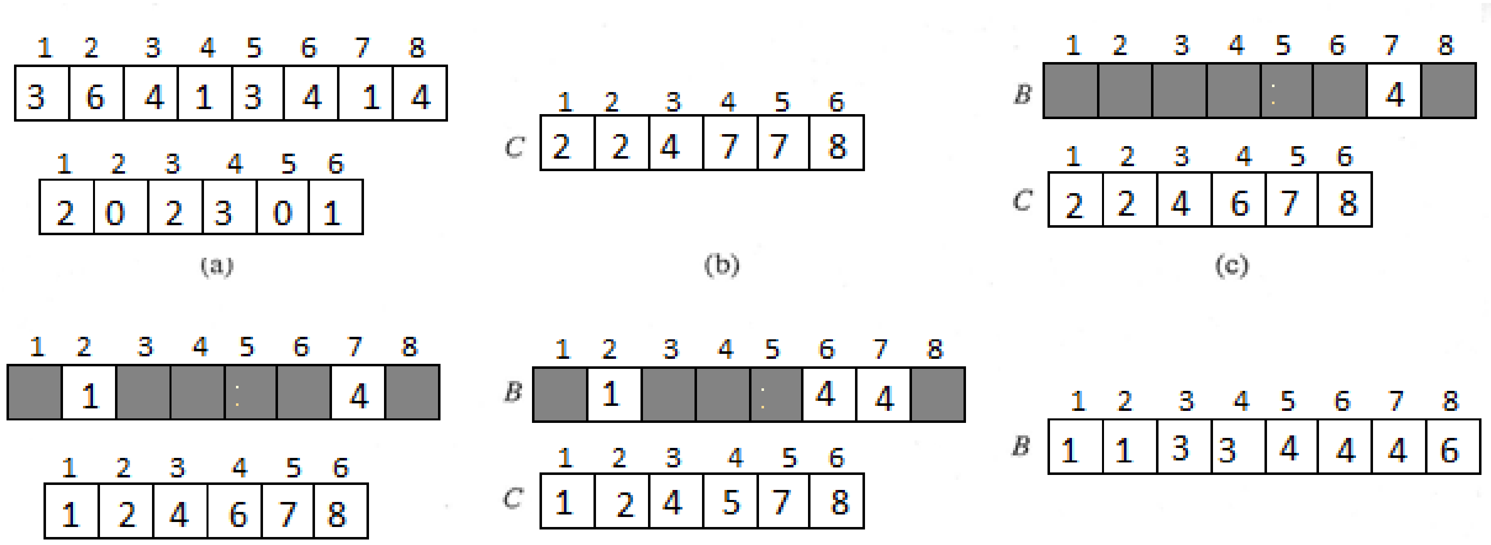
Advantages:

- Linear → O(n+k)
- Easy to code

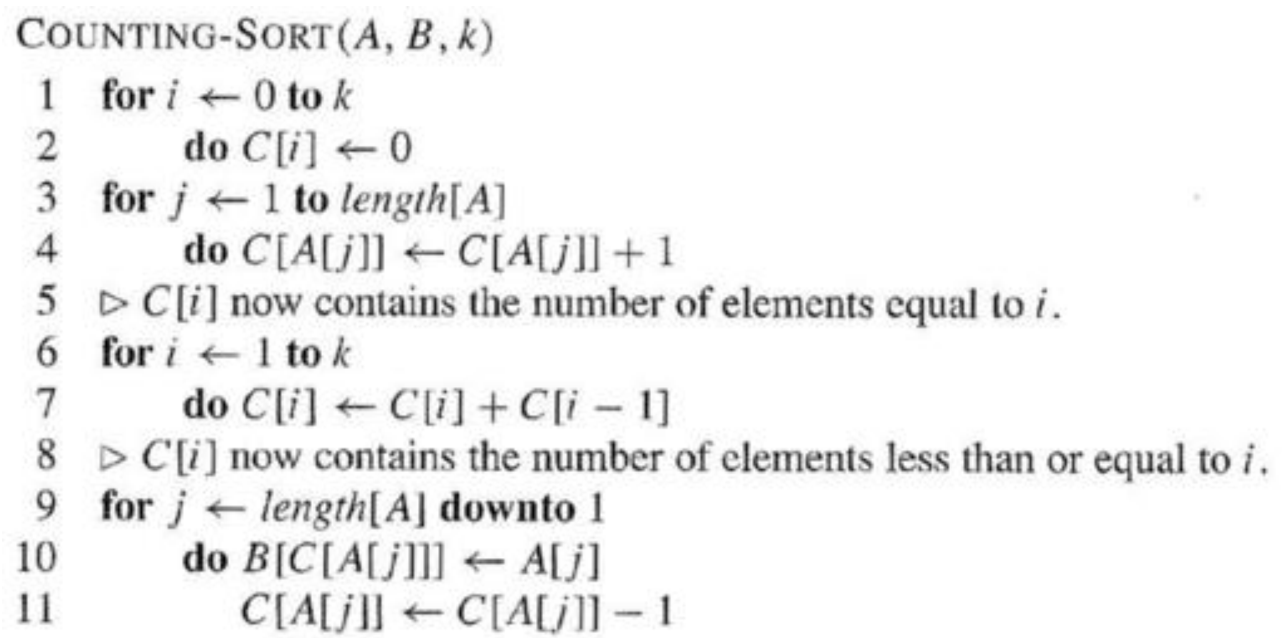
Disadvantages:

- Not in place

Counting Sort



Counting Sort



```
COUNTING-SORT(A, B, k)
for $i \leftarrow 0$ to $k$
        do $C[i] \leftarrow 0$
for $j \leftarrow 1$ to $length[A]$
        do $C[A[j]] \leftarrow C[A[j]] + 1$
$\triangleright$ $C[i]$ now contains the number of elements equal to $i$.
for $i \leftarrow 1$ to $k$
        do $C[i] \leftarrow C[i] + C[i-1]$
$\triangleright$ $C[i]$ now contains the number of elements less than or equal to $i$.
for $j \leftarrow length[A]$ downto 1
        do $B[C[A[j]]] \leftarrow A[j]$
           $C[A[j]] \leftarrow C[A[j]] - 1$
```

O(n+k)

Bucket Sort

Bucket sorts assumes that data is uniformly distributed within a range [0,1]

Main Idea:

- Creates n buckets of equal size to create partitions of range.
- Place all elements into their respective buckets.
- Sort individual buckets
- Concatenate the buckets

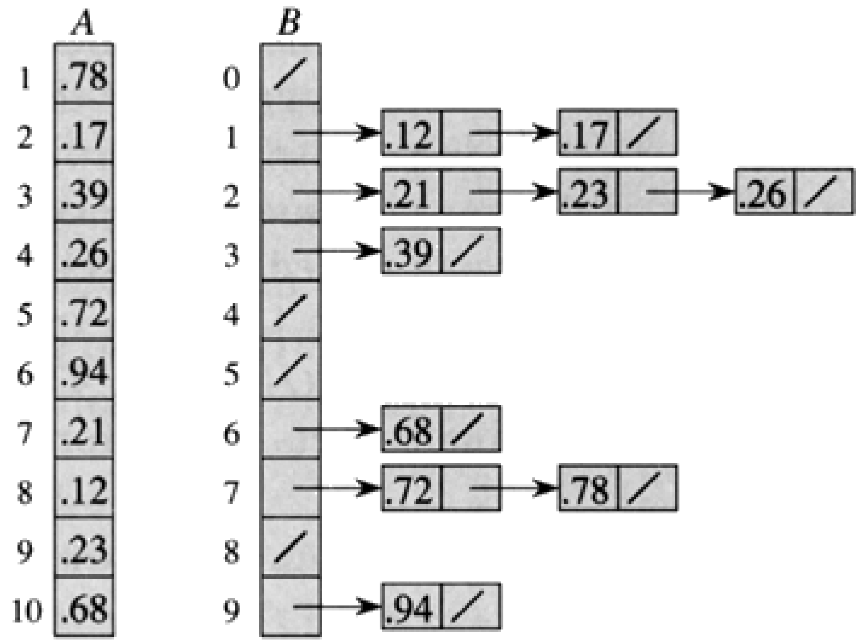
Advantages:

- Linear → O(n)
- Easy to code

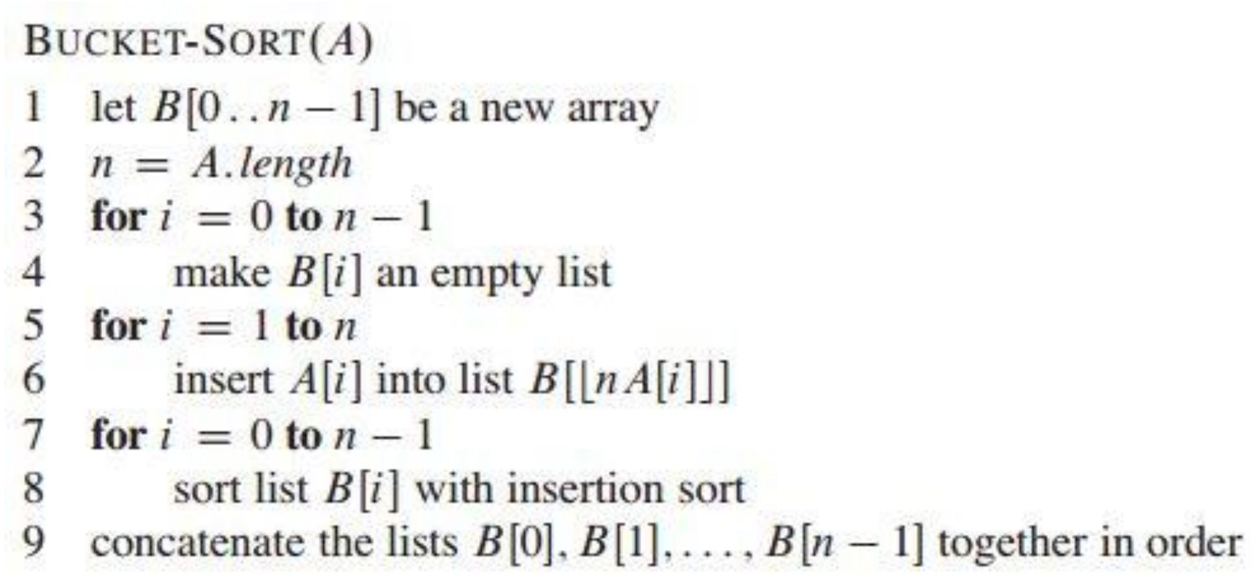
Disadvantages:

- Not in place

Bucket Sort


A
B
1 .78
2 .17
3 .39
4 .26
5 .72
6 .94
7 .21
8 .12
9 .23
10 .68
0
1 .12 .17
2 .21 .23 .26
3 .39
4
5
6 .68
7 .72 .78
8
9 .94

Bucket Sort



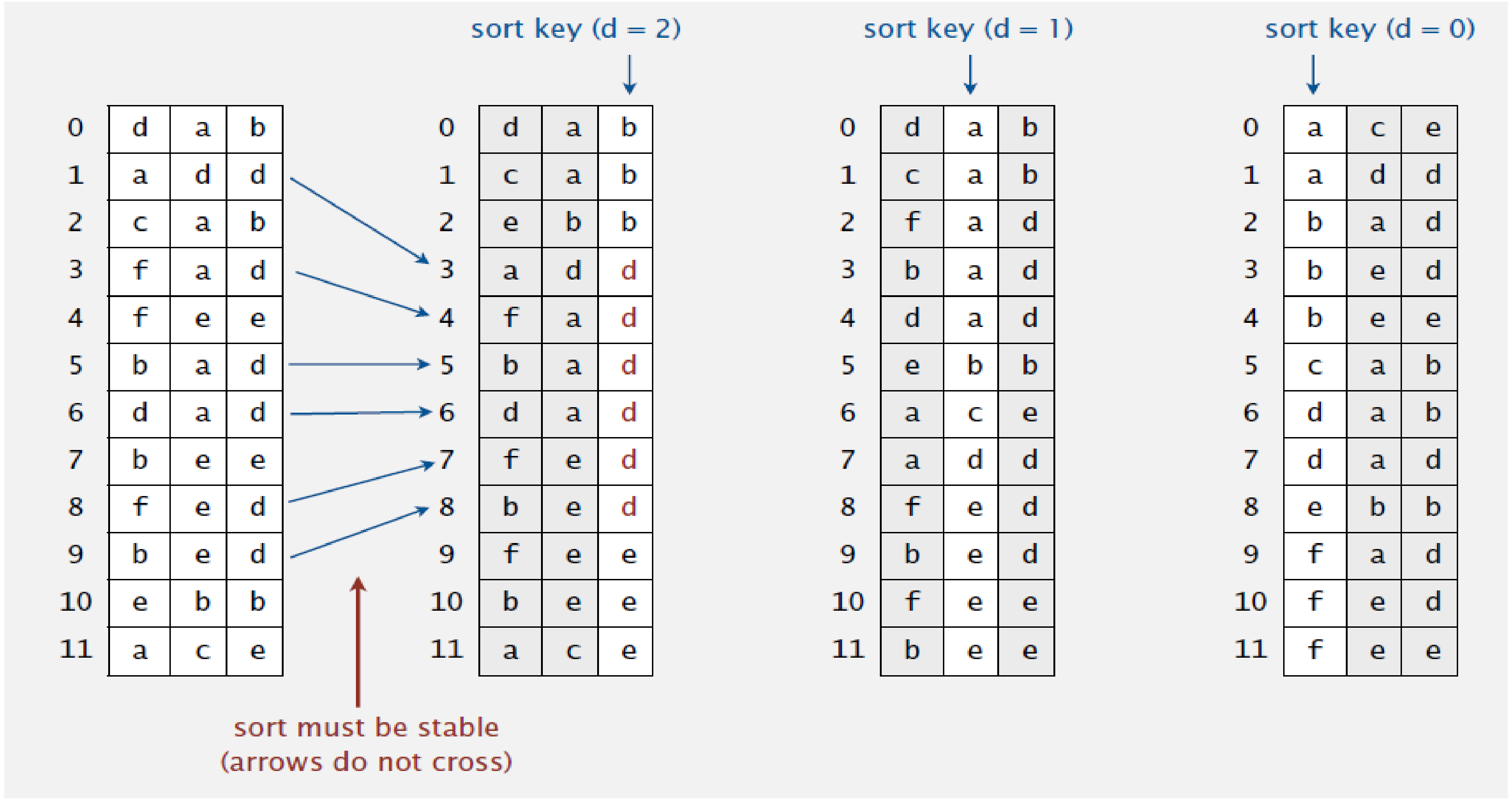
```
BUCKET-SORT($A$)
let $B[0..n-1]$ be a new array
$n = A.length$
for $i = 0$ to $n - 1$
    make $B[i]$ an empty list
for $i = 1$ to $n$
    insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
for $i = 0$ to $n - 1$
    sort list $B[i]$ with insertion sort
concatenate the lists $B[0], B[1], \ldots, B[n-1]$ together in order
```

Radix Sort

- Radix sort assumes that data elements are sequences of digits in a fixed range 0-R-1, and each data value is of fixed length k.
  - Like ASCII characters
  - Bank accounts
  - As characters also represented using integers, so this sort can also be applied to strings.
- It sorts the data in multiple passes on basis of significant digit of all data values.
  - Two classifications:
    - LSD-Least Significant Digit Radix Sort
      - Starts from least significant digit and moves to most significant digit
    - □ MSD-Most Significant Digit Radix Sort
      - Vice versa
  - It uses an intermediate sorting algorithm, which must be stable.
- Advantages:
  - Linear → O(n)
  - Easy to code
- Disadvantages:
  - Not in place

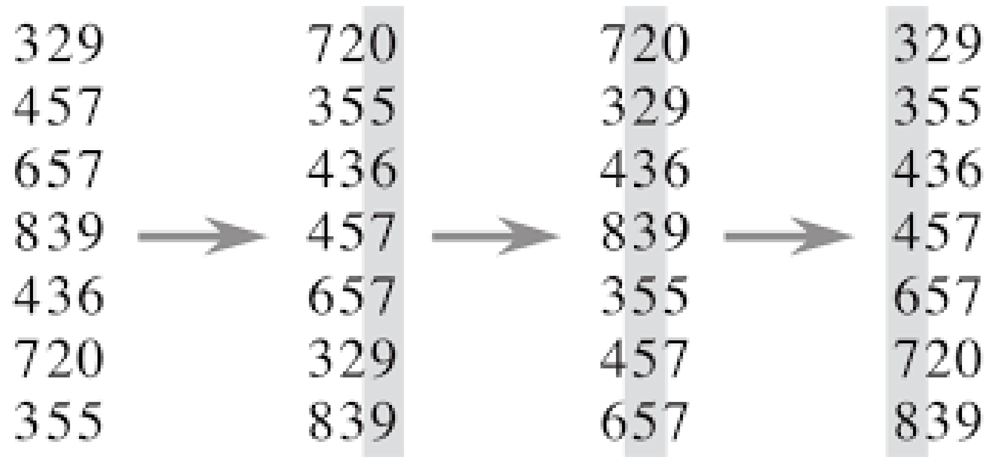
Radix Sort

Sort the data digit by digit.



0	d	a	b	0	d	a	b	0	d	a	b	0	a	c	e
1	a	d	d	1	c	a	b	1	c	a	b	1	a	d	d
2	c	a	b	2	e	b	b	2	f	a	d	2	b	a	d
3	f	a	d	3	a	d	d	3	b	a	d	3	b	e	d
4	f	e	e	4	f	a	d	4	d	a	d	4	b	e	e
5	b	a	d	5	b	a	d	5	e	b	b	5	c	a	b
6	d	a	d	6	d	a	d	6	a	c	e	6	d	a	b
7	b	e	e	7	f	e	d	7	a	d	d	7	d	a	d
8	f	e	d	8	b	e	d	8	f	e	d	8	e	b	b
9	b	e	d	9	f	e	e	9	b	e	d	9	f	a	d
10	e	b	b	10	b	e	e	10	f	e	e	10	f	e	d
11	a	c	e	11	a	c	e	11	b	e	e	11	f	e	e

Radix Sort


329
457
657
839
436
720
355
720
355
436
457
657
329
839
720
329
436
839
355
457
657
329
355
436
457
657
720
839

Radix Sort

```
Radix(A,k)
1.  For i=1 to k
2.      Use stable sort to sort A on digit I
3.      //counting sort will do the work
```