Design and Analysis of Algorithms

Chapter 4 – Sorting Algorithms

Instructor: Suleiman Gargaare

Sorting Algorithms

- In computer science, ‘sorting’ usually refers to **bringing a set of items into some well-defined order**. Sorting is important because having the items in order makes it much easier to find a given item, such as the cheapest item.
- A Sorting Algorithm is an algorithm that puts elements of a list in a certain order.
- It is used to rearrange a given array or list data structure elements according to a particular pattern (For e.g. Ascending or Descending order).
- 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.

Common Sorting Strategies

One way of organizing the various sorting algorithms is by classifying the underlying idea, or ‘strategy’. Some of the key strategies are:

- **Exchange Strategy**: If two items are found to be out of order, exchange them. Repeat till all items are in order.
- **Selection Strategy**: Find the smallest item, put it in the First position, find the smallest of the remaining items, put it in the second position . . .
- **Divide & Conquer Strategy**: Recursively split the problem into smaller sub-problems till you just have single items that are trivial to sort. Then put the sorted ‘parts’ back together in a way that preserves the sorting.

Properties of Sorting Algorithms

- **In-place Sort** – The input and output occupy same memory storage space, without using any additional memory.
- **Stable Sort** – It preserves the original order of ties (elements of the same value)
- **Comparison** – Sort or arrange items by comparing with each other.
- **Non-Comparison** – Sort or arrange items without comparing with each other, by making certain assumptions about the data that are going to sort.

Types of Sorting Algorithms

- Incremental comparison sorting
  - Selection sort
  - Bubble sort
  - Insertion sort
- Recursive comparison sorting
  - Merge sort
  - Quick sort
- Non-comparison sorting
  - Count sort
  - Radix sort

Selection Sort

- Idea:
  - Find the smallest element in the array
  - Exchange it with the element in the first position
  - Find the second smallest element and exchange it with the element in the second position
  - Continue until the array is sorted

Bubble Sort

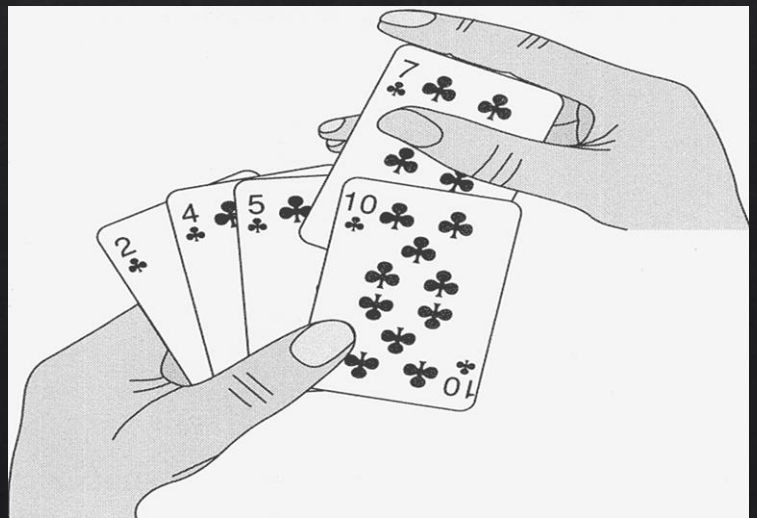
Given a list of N elements, repeat the following steps N-1 times:

- For each pair of adjacent numbers, if number on the left is greater than the number on the right, swap them.
- "Bubble" the largest value to the end using pair-wise comparisons and swapping.

16 12	16 12	22 14	22 14	22 17	22 17
6	12	14	14	17	22
6	12	12	14	17	22
6	8	12	14	17	22

Insertion Sort

- Idea: like sorting a hand of playing cards
  - Start with empty left hand and cards face down on the table.
  - Remove one card at a time from the table, and insert it into the correct position in the left hand
    - compare it with each card already in the hand, from right to left
  - The cards held in the left hand are sorted
    - these cards were originally the top cards of the pile on the table



Insertion Sort Algorithm

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	2.78	7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
		7.42	0.56	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
		0.56	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	2.78	7.42	1.12	1.17	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56		1.12	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	1.12	2.78	7.42	1.17	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	1.12		1.17	7.42	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	1.12	1.17	2.78	7.42	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	1.12	1.17		7.42	0.32	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	1.12	1.17		0.32	7.42	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	1.12	1.17	0.32	2.78	7.42	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	1.12	0.32	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.56	0.32	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	7.42	6.21	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	6.21	7.42	4.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	6.21	4.42	7.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	4.42	6.21	7.42	3.14	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	4.42	6.21	3.14	7.42	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	4.42	3.14	6.21	7.42	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

Insertion Sort

```
InsertionSort(A, n)
1.  for i = 1 to n-1
2.      key = A[i]
3.      j = i - 1
4.      while (j >= 0) and (A[j] > key)
5.          A[j+1] = A[j]
6.          j = j - 1
7.      A[j+1] = key
```

Array index	0	1	2	3	4	5	6	7	8	9
	0.32	0.56	1.12	1.17	2.78	3.14	4.42	6.21	7.42	7.71

DONE!

Types of Sorting Algorithms

- Incremental comparison sorting
  - Selection sort
  - Bubble sort
  - Insertion sort
- Recursive comparison sorting
  - Merge sort
  - Quick sort
- Non-comparison sorting
  - Count sort
  - Radix sort

Recursive Comparison Sorting

There are many ways to design algorithms:

Insertion/Bubble/Selection sort uses an incremental approach

- Having sorted to array A[i..j]
- We insert the single element A[j+1] into its proper place, to get a sorted array A[i..j+1]

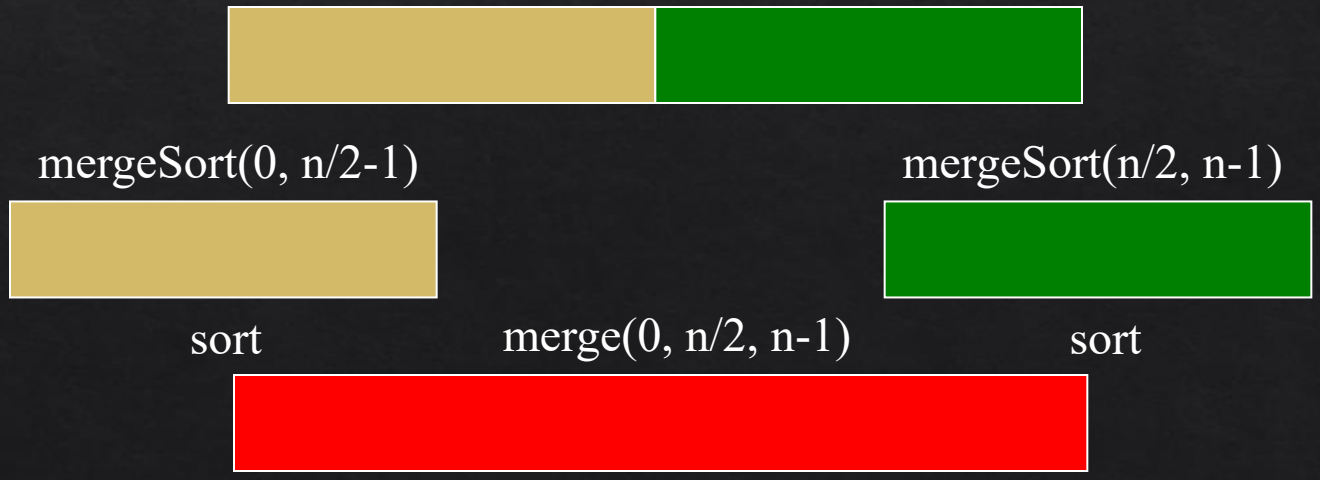
An alternative design approach is "Divide and Conquer"

- **Divide** the problems into a number of sub-problems
- **Conquer** the sub-problems by solving them recursively. If sub-problem sizes are small enough, just solve them in a straight forward manner.
- **Combine** the solutions to the sub-problems into the solution for the original problem.

Merge Sort

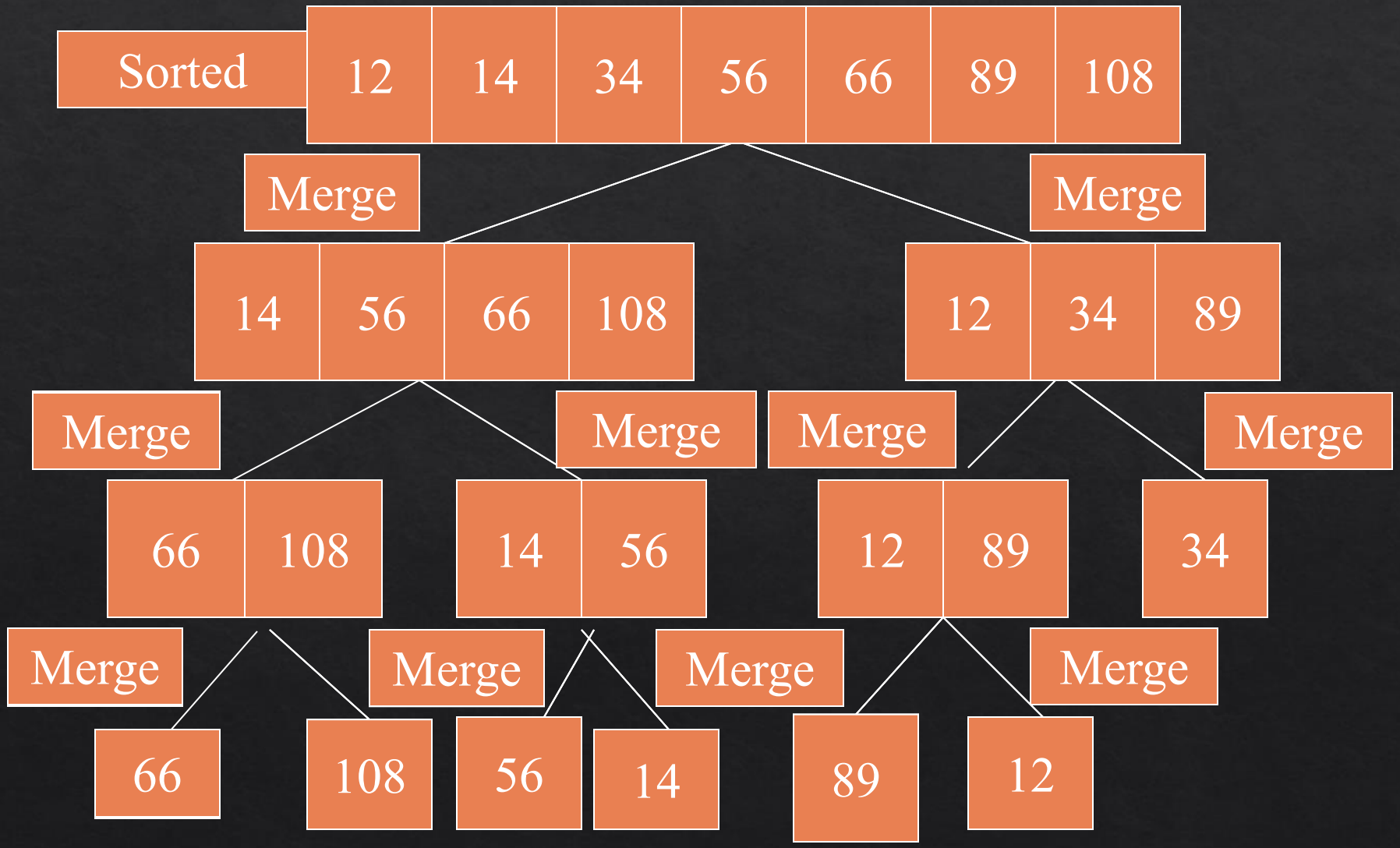
- Idea: Order a list of values by recursively dividing the list in half until each sub-list has one element, then recombining

- More specifically:
  - Divide the n element sequence to be sorted into two subsequences of n/2 elements each.
  - Conquer: Sort the two subsequences to produce the sorted answer.
  - Combine: Merge the two sorted sub sequences to produce the sorted answer.



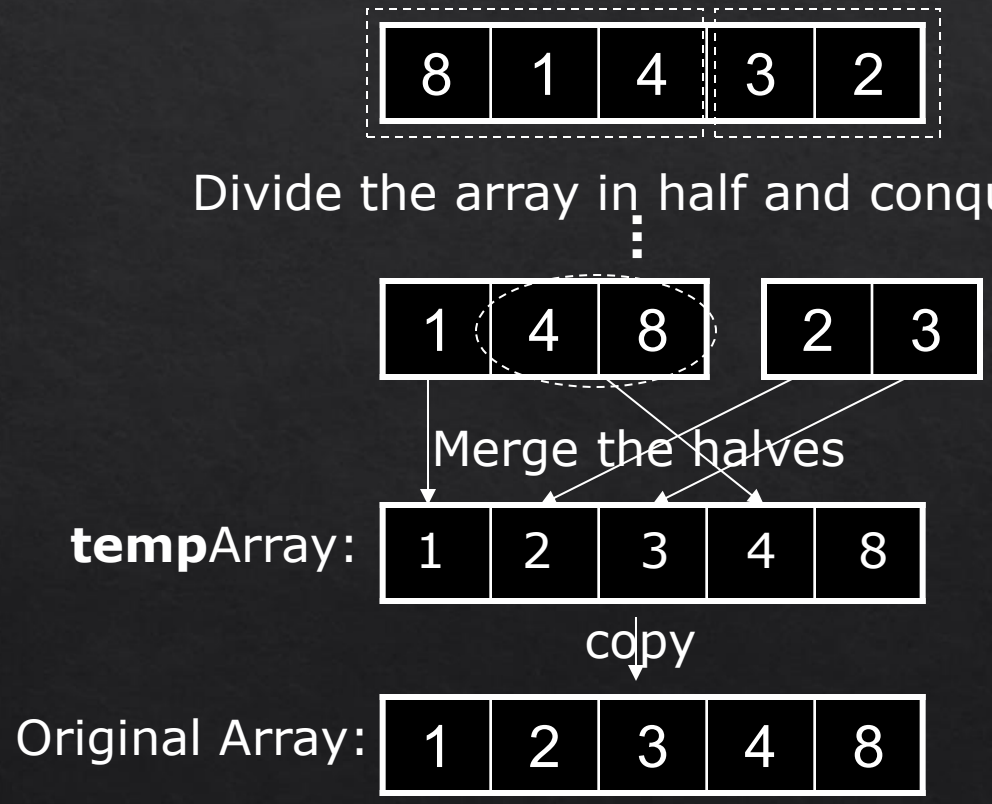
Merge Sort

When the sequences to be sorted has length 1.



Merge Function

Given two sorted arrays, *merge* operation produces a sorted array with all the elements of the two arrays



Quick Sort

- **Quick Sort**: orders a list of values by partitioning the list around one element called a ***pivot***, then sorting each partition. It is based on divide and conquer approach.

- Key Points: Given an array **S** to be sorted
  - Pick any element **v** in array **S** as the **pivot** (i.e. **partition element**)
  - Partition the remaining elements in **S** into two groups
    - `S1` = {all elements in `S-{v}` that are smaller than **v**}
    - `S2` = {all elements in `S-{v}` that are larger than **v**}
  - apply the quick sort algorithm (**recursively**) to **both partitions**

- Trick lies in handling the partitioning
  - i.e. picking a good pivot is essential to performance

Quick Sort Illustrated

How Fast Can We Sort?

- Selection Sort, Bubble Sort, Insertion Sort ➔ $O(n^2)$
- Merge Sort, Quick Sort ➔ $O(n \log n)$

- What is common to all these algorithms?
  - Make **comparisons** between input elements

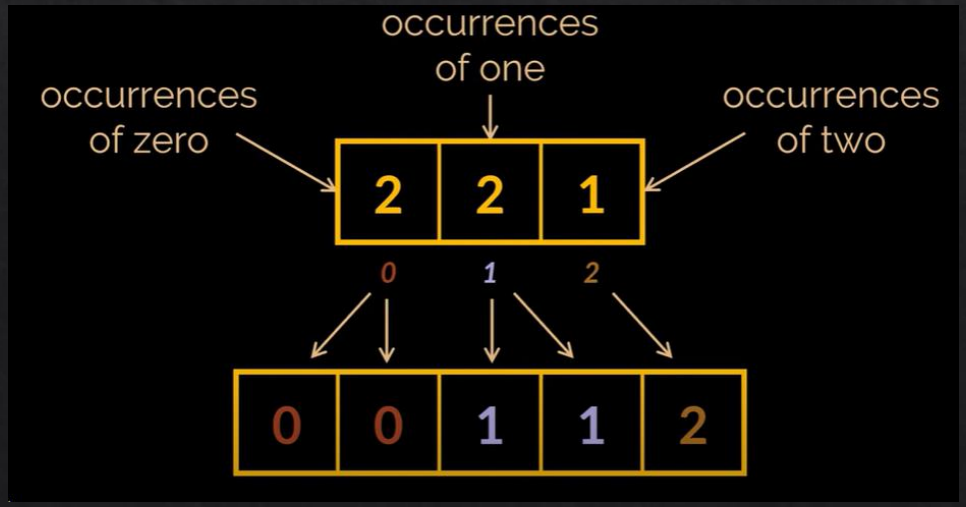
- Lower bound for comparison based sorting
  - **Theorem:** To sort n elements, comparison sorts **must** make $\Omega(n \log n)$ comparisons in the worst case.

Types of Sorting Algorithms

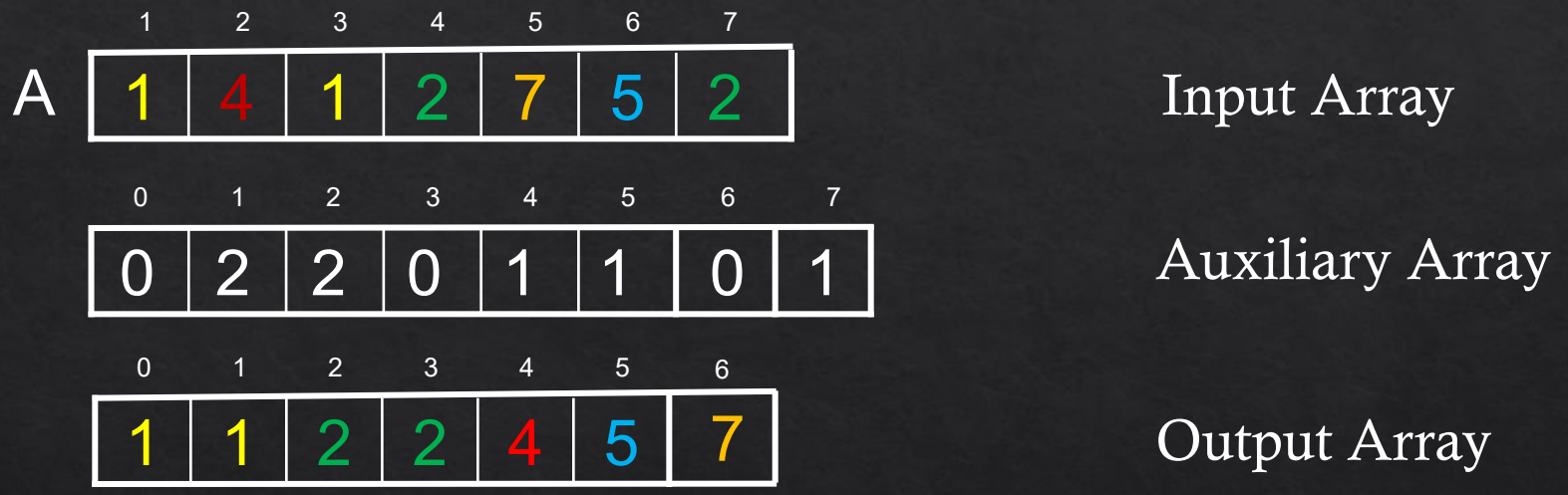
- Incremental comparison sorting
  - Selection sort
  - Bubble sort
  - Insertion sort
- Recursive comparison sorting
  - Merge sort
  - Quick sort
- Non-comparison sorting
  - Count sort
  - Radix sort

Counting Sort

- **Counting Sort is a sorting technique based on keys between a specific range.**
- It works counting the number of objects having distinct key values,
  - then doing some arithmetic to calculate position of each object in the output sequence.



Counting Sort



Radix Sort

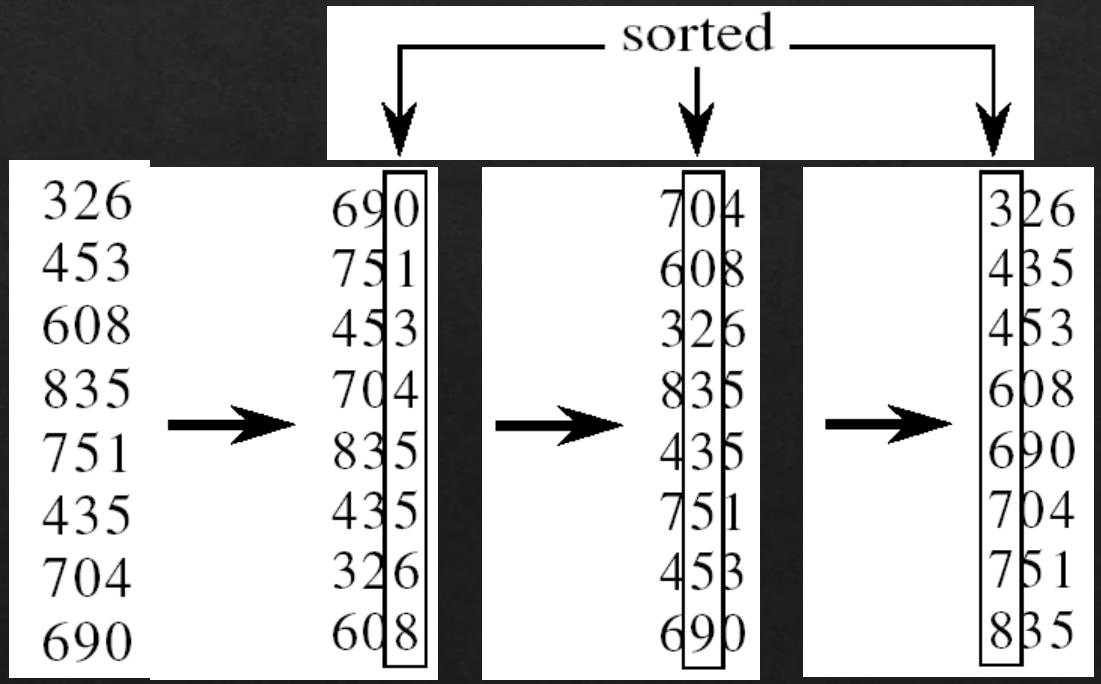
- **Radix sort is a non-comparative sorting algorithm.**
- It avoids comparison by creating and distributing elements into buckets according to their radix. For elements with more than one significant digit, this bucketing process is repeated for each digit, until all digits have been considered.
- For this reason, radix sort has also been called **bucket sort** & **digital sort**.

  - Complexity: O(dn)

326
453
608
835
751
435
704
690

Radix Sort

```
RadixSort(A, d)
    for i=1 to d
        StableSort(A) on digit i
```



THE END

THANK YOU!