Session 15

Searching and Sorting

Searching Algorithms

Searching Algorithm is an algorithm made up of a series of instructions that retrieves information stored within some data structure, or calculated in the search space of a problem domain.

Linear Search

Linear Search is a method for finding a target value within a list. It sequentially checks each element of the list for the target value until a match is found or until all the elements have been searched.

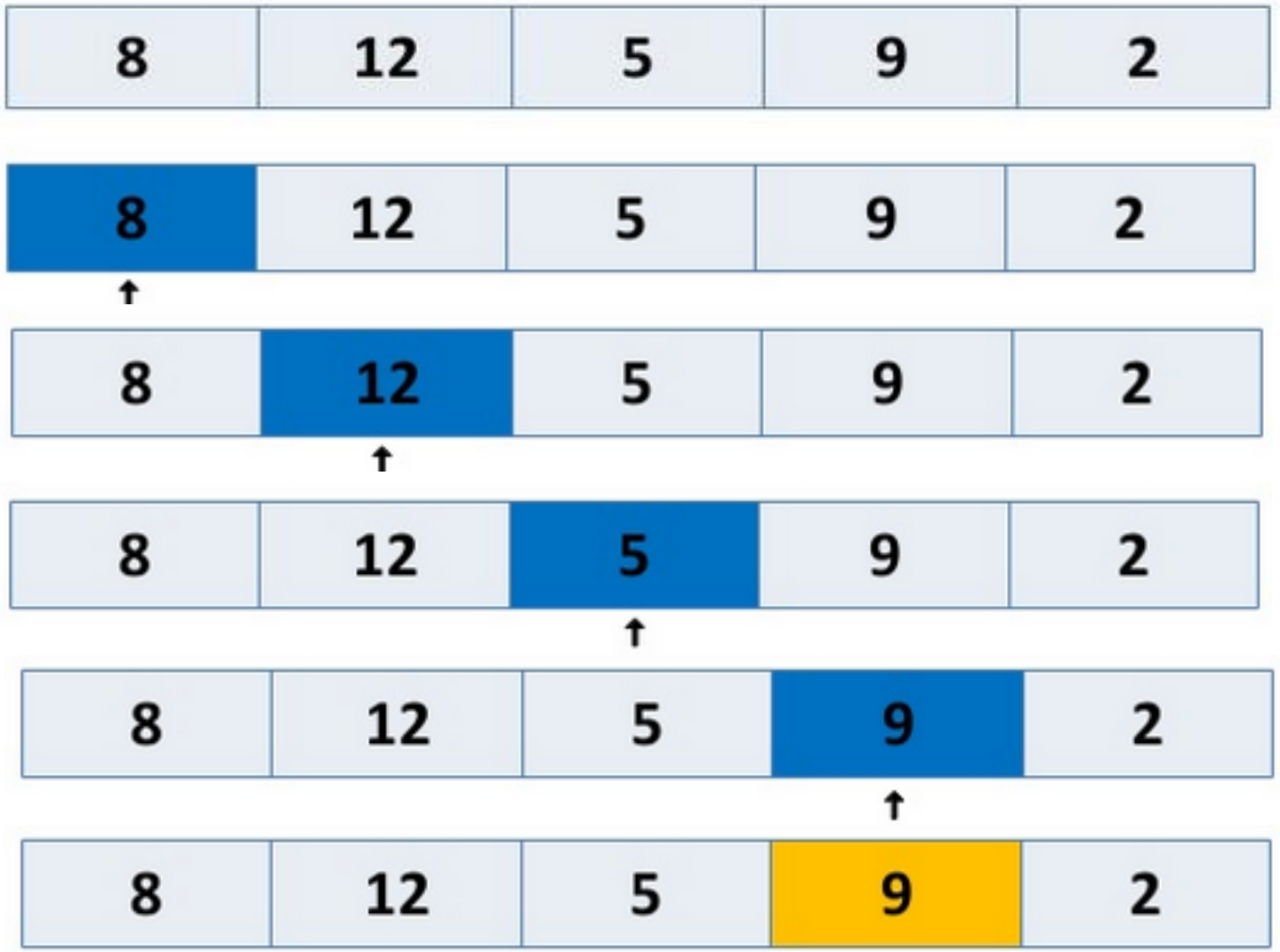
Linear search Algorithm

Step1: Start from the leftmost element of array and one by one compare x with each element of array.

Step2: If x matches with an element, return the index.

Step3: If x doesn't match with any of elements, return -1.

- Assume the following Array:
- Search for 9



Found at index = 3

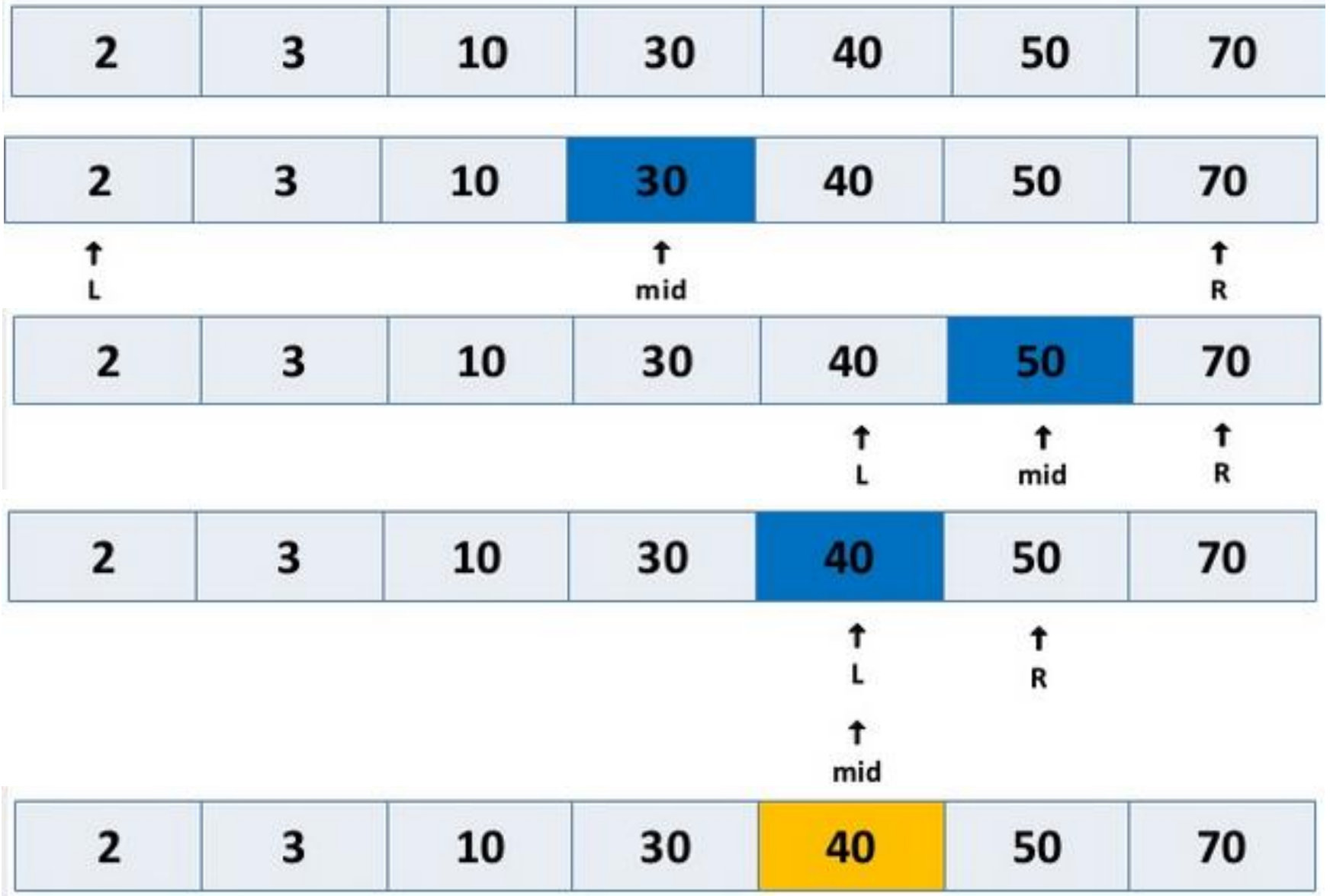
Binary Search

Binary Search is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that is used to solve problems. Binary search use sorted array by repeatedly dividing the search interval in half.

Binary Search Algorithm

- Step1: Compare x with the middle element.
- Step2: If x matches with middle element, we return the mid index.
- Step3: Else If x is greater than the mid element, search on right half.
- Step4: Else If x is smaller than the mid element. search on left half.

- Assume the following Array:
- Search for 40



found at index = 4

Jump Search

Jump Search is a searching algorithm for sorted arrays. The basic idea is to check fewer elements (than linear search) by jumping ahead by fixed steps or skipping some elements in place of searching all elements.

Jump Search Algorithm

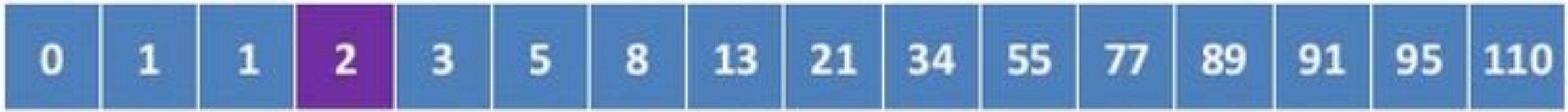
- **Step1:** Calculate Jump size
- Step2: Jump from index i to index i+jump
- Step3: If x = = arr[i+jump] return x
- Else jump back a step
- Step4: Perform linear search

- Assume the following sorted array:
- Search for 77
- Calculate:

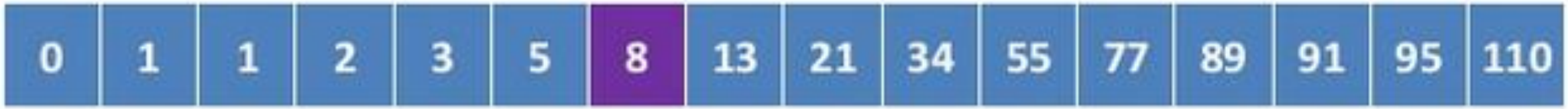
0	1	1	2	3	5	8	13	21	34	55	77	89	91	95	110

- Size of array n =16
- Jump size = sqrt(n)= 4

- Jump from index 0 to index 3
- Compare index value with search number 2<77

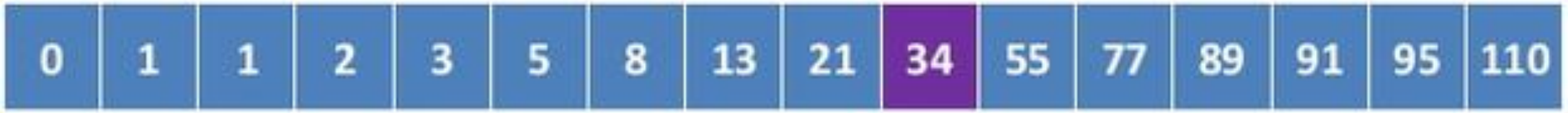


- Jump from index 3 to index 6
- Compare index value with search number 8<77



Jump from index 6 to index 9

Compare index value with search number 34<77



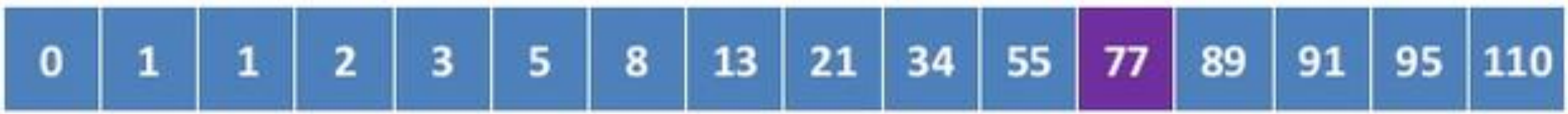
Jump from index 9 to index 12

Compare index value with search number 89>77

jump back a step

Perform linear search

Compare found at index 11



Interpolation Search

The **Interpolation Search** is an improvement over Binary Search for instances. On the other hand interpolation search may go to different locations according the value of key being searched.

Interpolation Search Algorithm

- **Step1:** In a loop, calculate the value of "pos" using the position formula.
- **Step2:** If it is a match, return the index of the item, and exit.
- **Step3:** If the item is less than arr[pos], calculate the position of the left sub-array. Otherwise calculate the same in the right sub-array.
- **Step4:** Repeat until a match is found or the sub-array reduces to zero.

Interpolation Search

// The idea of formula is to return higher value of pos
// when element to be searched is closer to arr[hi]. And
// smaller value when closer to arr[lo]

pos = lo + [(x-arr[lo])*(hi-lo) / (arr[hi]-arr[Lo])]

arr[] ==> Array where elements need to be searched
x ==> Element to be searched
lo ==> Starting index in arr[]
hi ==> Ending index in arr[]

- Assume the following sorted array:
- Search for x= 18

10	12	13	16	18	19	20	21	22	23	24	33	35	42	47

- Calculate **pos = lo + [(x-arr[lo])*(hi-lo) / (arr[hi]-arr[Lo])]**
- **Lo=0, hi=14, arr[lo]=10, arr[hi]= 47**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	12	13	16	18	19	20	21	22	23	24	33	35	42	47

- Calculate **pos = 3**
- **Compare with x=18**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	12	13	**16**	18	19	20	21	22	23	24	33	35	42	47

- Calculate **pos = lo + [(x-arr[lo])*(hi-lo) / (arr[hi]-arr[Lo])]**
- **Lo=4, hi=14, arr[lo]=18, arr[hi]= 47**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	12	13	16	18	19	20	21	22	23	24	33	35	42	47

- Calculate **pos = 4**
- **Compare with x=18 , found at index 4**

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	12	13	16	18	19	20	21	22	23	24	33	35	42	47

Sorting Algorithms

Outline

- **Overview of Sorting**
  - Measuring Performance
- **Sorting Algorithms**
  - Bubble Sort
  - Insertion Sort
  - Merge Sort
  - Selection Sort
  - Quick Sort
- **Visualizing Sorting Performance**
  - Sample Source Code and Demo

Sorting Overview

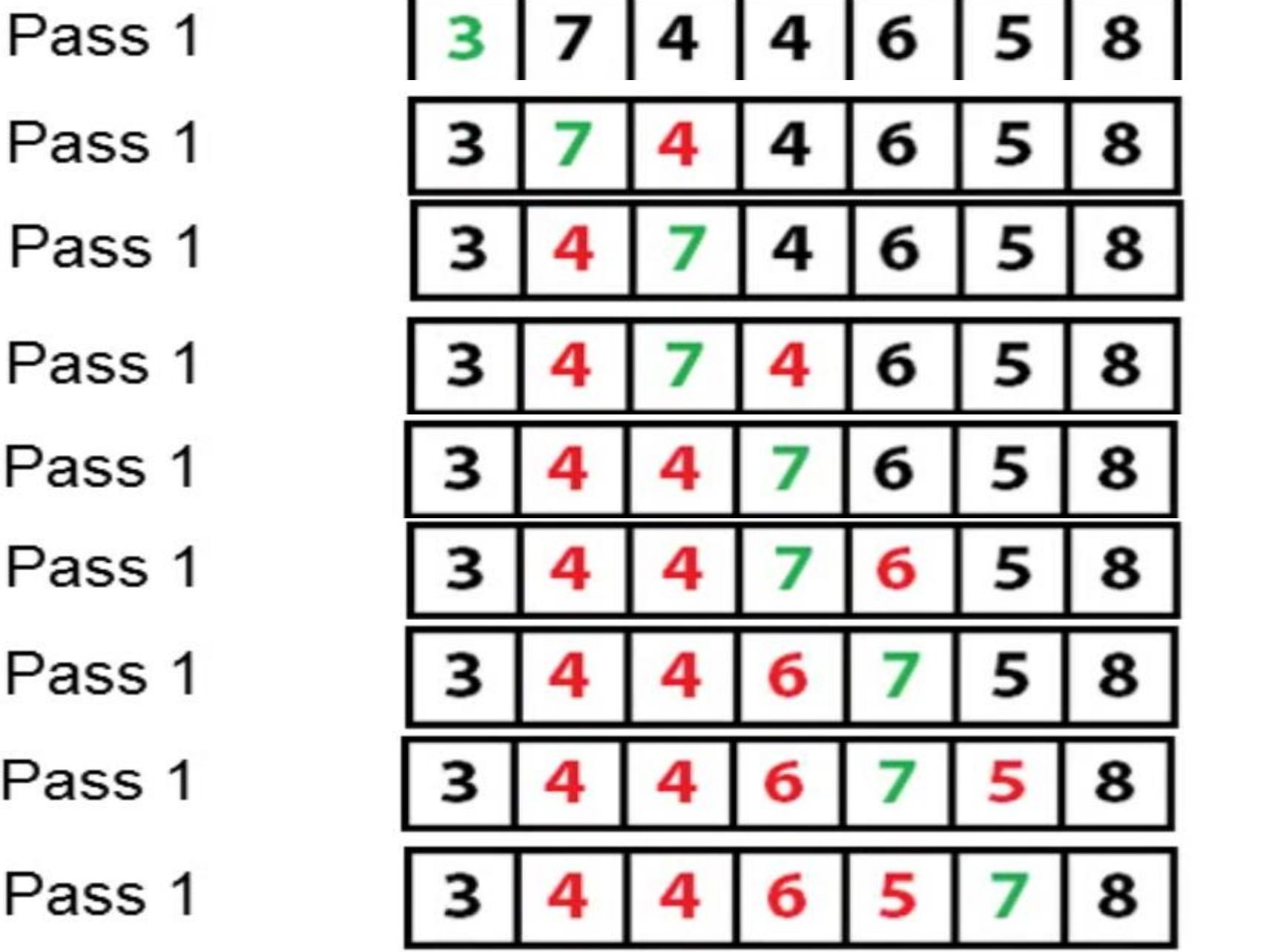
- **Arranging data in a collection based on a comparison algorithm**
  - E.g., Any object with a notion of greater-than/less-than/equality
- **Two general families of sorting algorithms**
  - Linear Sorting
  - Divide and Conquer
- **Linear algorithms treat the problem of sorting as a single large operation.**
- **Divide and Conquer algorithms partition the data to be sorted into smaller sets that can be independently sorted.**

Measuring Performance

- **Comparisons**
  - When two values of the input array are compared for relative equality
    - Equal to, Greater then, Less then
- **Swaps**
  - When two values stored in the input array are swapped
  - E.g., [1, 0] => [0, 1]
- **Performance Considerations**
  - Comparisons and Swaps both have a cost
  - Reducing either or both can improve performance
  - The cost of both operations depends on many factors.

Bubble Sort

- **Simplest sorting algorithm**
- **On Each Pass**
  - Compare each array item to it's right neighbor
  - If the right neighbor is smaller then Swap right and left
  - Repeat for the remaining array items
- **Perform subsequent passes until no swaps are performed**


Pass 1
3 7 4 4 6 5 8
Pass 1
3 7 4 4 6 5 8
Pass 1
3 4 7 4 6 5 8
Pass 1
3 4 7 4 6 5 8
Pass 1
3 4 4 7 6 5 8
Pass 1
3 4 4 7 6 5 8
Pass 1
3 4 4 6 7 5 8
Pass 1
3 4 4 6 7 5 8
Pass 1
3 4 4 6 5 7 8

Pass 1	3	4	4	6	5	7	8
Pass 2	3	4	4	6	5	7	8
Pass 2	3	4	4	6	5	7	8
Pass 2	3	4	4	6	5	7	8
Pass 2	3	4	4	6	5	7	8
Pass 2	3	4	4	6	5	7	8
Pass 2	3	4	4	5	6	7	8

Pass 2	3	4	4	5	6	7	8

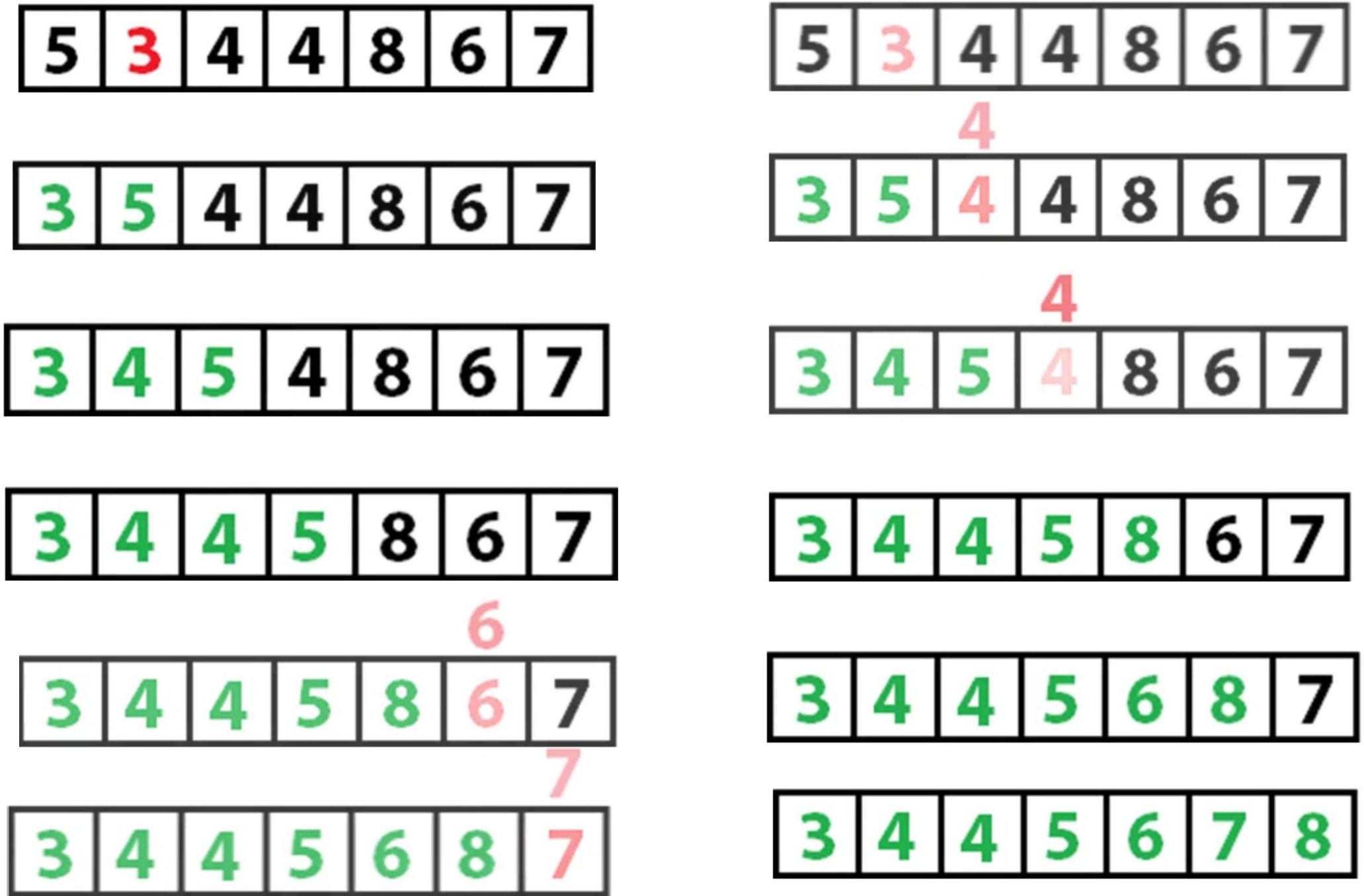
Pass 3	3	4	4	5	6	7	8
Pass 3	3	4	4	5	6	7	8
Pass 3	3	4	4	5	6	7	8
Pass 3	3	4	4	5	6	7	8
Pass 3	3	4	4	5	6	7	8
Pass 3	3	4	4	5	6	7	8

Bubble Sort Performance

- **Worst case performance: $O(n^2)$**
  - Not appropriate for large unsorted data sets.
- **Average case performance: $O(n^2)$**
  - Not appropriate for large unsorted data sets.
- **Best case performance: O(n)**
  - Very good best case performance and can efficiently sort small and nearly sorted data sets
- **Space required: O(n)**
  - Bubble sort operates directly on the input array meaning it is a candidate algorithm when minimizing space is paramount.

Insertion Sort

- Sorts each item in the array as they are encountered
- As the current item works from left to right
  - Everything left of the item is known to be sorted
  - Everything to the right is unsorted
- The current item is "inserted" into place within the sorted section

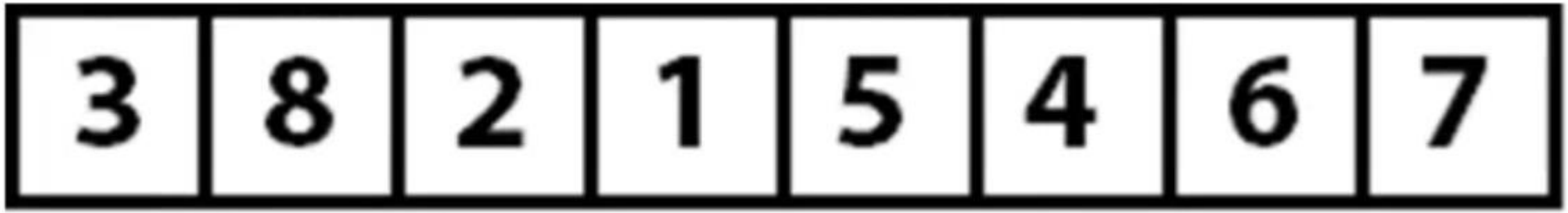

5 3 4 4 8 6 7
3 5 4 4 8 6 7
3 4 5 4 8 6 7
3 4 4 5 8 6 7
3 4 4 5 8 6 7
3 4 4 5 6 8 7
3 4 4 5 6 8 7
3
5 3 4 4 8 6 7
4
3 5 4 4 8 6 7
4
3 4 5 4 8 6 7
3 4 4 5 8 6 7
3 4 4 5 6 8 7
3 4 4 5 6 7 8

Insertion Sort Performance

- **Worst case performance: $O(n^2)$**
  - Not appropriate for large unsorted data sets.
- **Average case performance: $O(n^2)$**
  - Not appropriate for large unsorted data sets.
- **Best case performance: O(n)**
  - Very good best case performance and can efficiently sort small and nearly sorted data sets
- **Space required: O(n)**
  - Insertion sort operates directly on the input array meaning it is a candidate algorithm when minimizing space is paramount.

Selection Sort

- **Sorts the data by finding the smallest item and swapping it into the array in the first unsorted location.**
- **Algorithm:**
  - Enumerate the array from the first unsorted item to the end
  - Identify the smallest item
  - Swap the smallest item with the first unsorted item



3	8	2	1	5	4	6	7

3	8	2	1	5	4	6	7

1	8	2	3	5	4	6	7

1	8	2	3	5	4	6	7

1	2	8	3	5	4	6	7

1	2	8	3	5	4	6	7

1	2	3	8	5	4	6	7

1	2	3	8	5	4	6	7

1	2	3	4	5	8	6	7

1	2	3	4	5	8	6	7

1	2	3	4	5	8	6	7

1	2	3	4	5	6	8	7

1	2	3	4	5	6	8	7

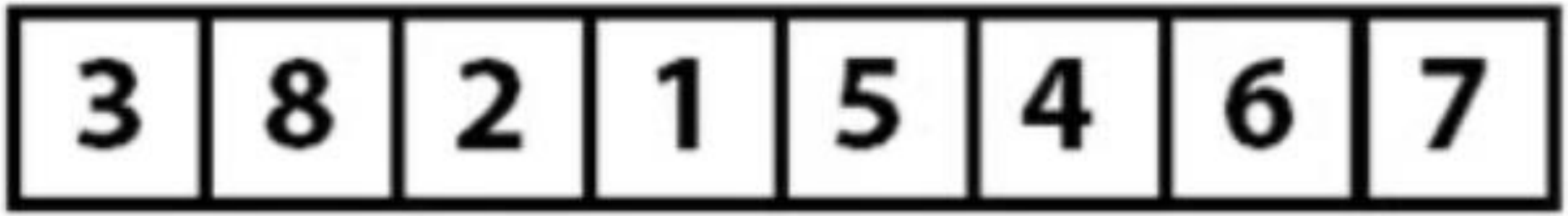
1	2	3	4	5	6	7	8

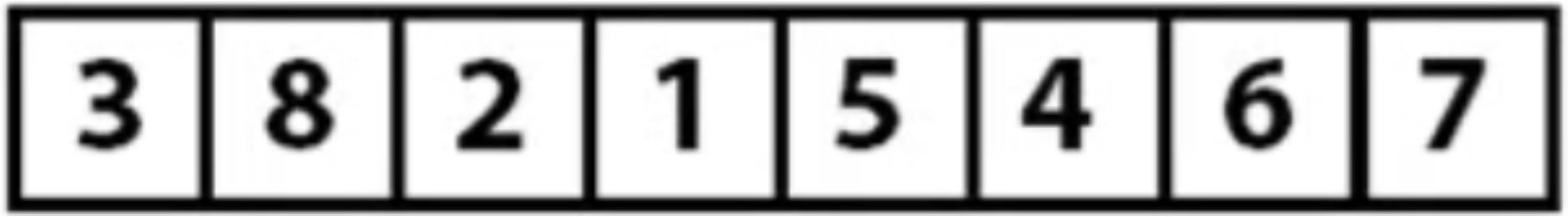
Selection Sort Performance

- **Worst case performance: $O(n^2)$**
  - Not appropriate for large unsorted data sets.
- **Average case performance: $O(n^2)$**
  - Not appropriate for large unsorted data sets.
  - Typically performs better than bubble but worse than insertion sort
- **Best case performance: $O(n^2)$**
  - Not appropriate for large unsorted data sets.
- **Space required: O(n)**
  - Selection sort operates directly on the input array meaning it is a candidate algorithm when minimizing space is paramount.

Merge Sort

- The array is recursively split in half
- When the array is in groups of 1, it is reconstructed in sort order
- Each reconstructed array is merged with the other half

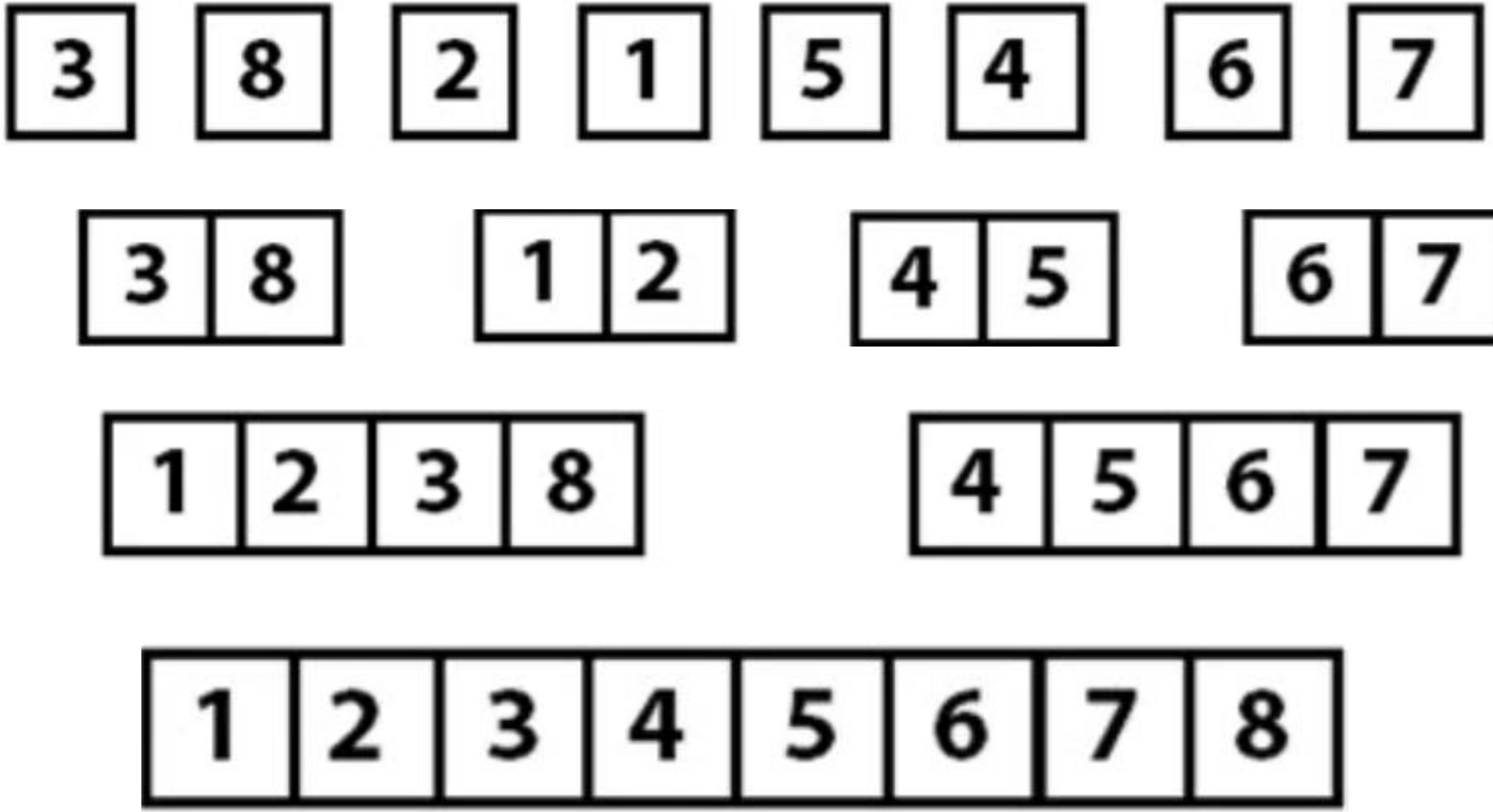



3 8 2 1 5 4 6 7


3 8 2 1
5 4 6 7


3 8
2 1
5 4
6 7


3 8 2 1 5 4 6 7

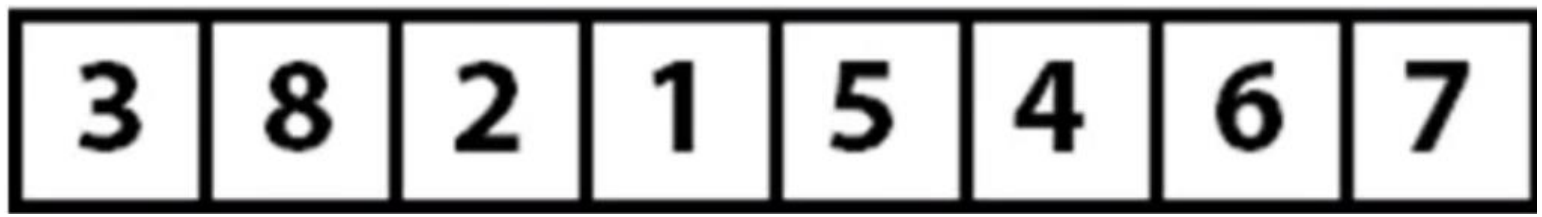

3 8 2 1 5 4 6 7
3 8 1 2 4 5 6 7
1 2 3 8 4 5 6 7
1 2 3 4 5 6 7 8

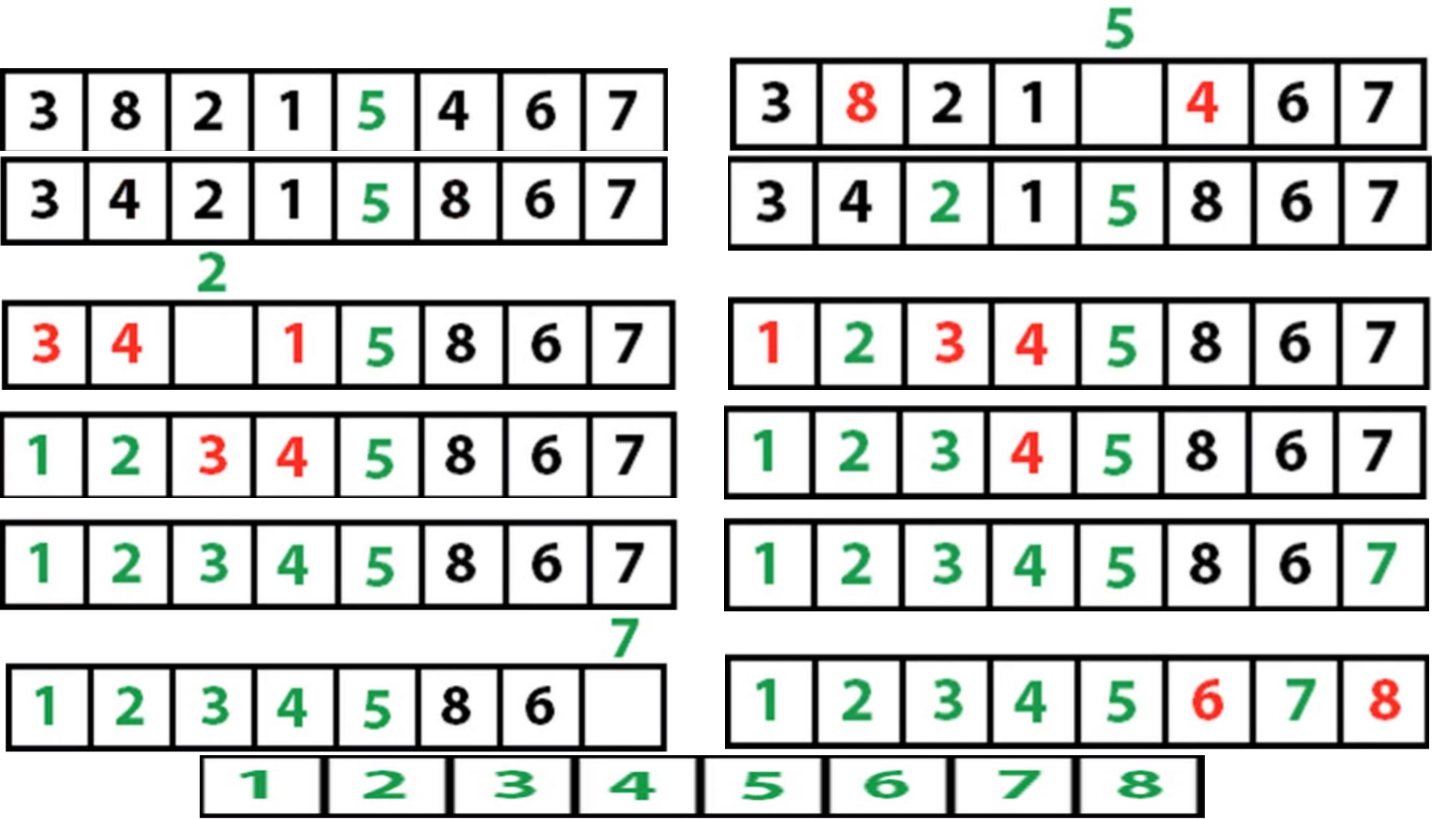
Quick Sort Performance

- **Worst case performance:$O(n^2)$**
  - Not appropriate for large pathologically sorted (inverse sorted) data sets.
- **Average case performance:O(n log n)**
  - Appropriate for large data sets
- **Best case performance:O(n log n)**
  - Very good best case performance and can efficiently sort small and nearly sorted data sets
- **Space required: O(n)**
  - As a recursive algorithm the array space as well as the stack space must be considered. There exist optimizations to reduce space usage further.

Quick Sort

- Divide and Conquer algorithm
- Pick a pivot value and partition the array
- Put all values before the pivot to the left and above to the right
  - The pivot point is now sorted – everything right is larger, everything left is smaller.
- Perform pivot and partition algorithm on the left and right partitions
- Repeat until sorted



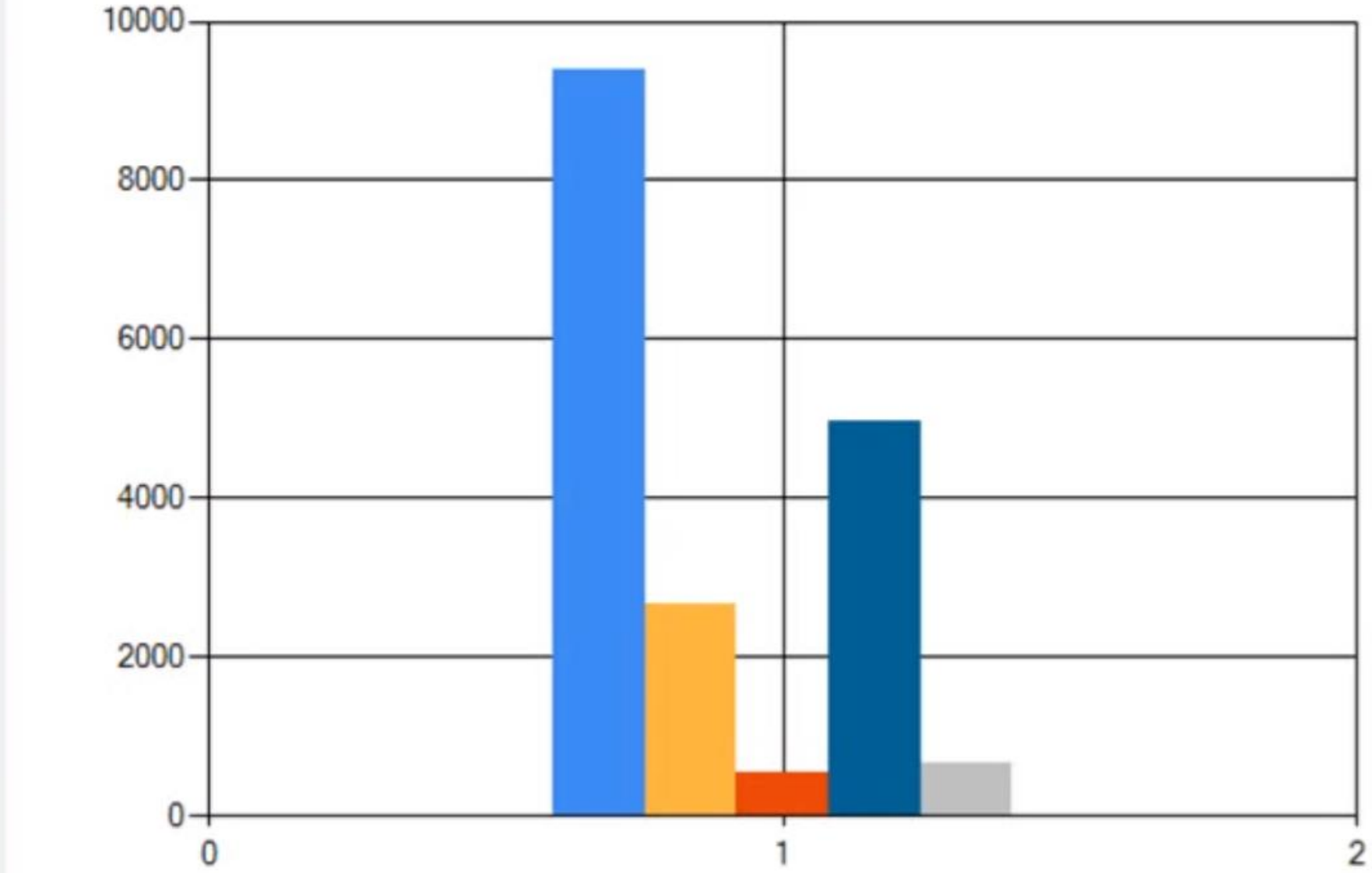

3 8 2 1 5 4 6 7
3 4 2 1 5 8 6 7
2
3 4 1 5 8 6 7
1 2 3 4 5 8 6 7
1 2 3 4 5 8 6 7
7
1 2 3 4 5 8 6
1 2 3 4 5 6 7 8
5
3 8 2 1 4 6 7
3 4 2 1 5 8 6 7
1 2 3 4 5 8 6 7
1 2 3 4 5 8 6 7
1 2 3 4 5 6 7 8
1 2 3 4 5 6 7 8

Quick Sort Performance

- **Worst case performance: $O(n^2)$**
  - Not appropriate for large pathologically sorted (inverse sorted) data sets.
- **Average case performance: O(n log n)**
  - Appropriate for large data sets
- **Best case performance: O(n log n)**
  - Very good best case performance and can efficiently sort small and nearly sorted data sets
- **Space required: O(n)**
  - As a recursive algorithm the array space as well as the stack space must be considered. There exist optimizations to reduce space usage further.

Visualizing Sorting Performance

- **Graphically comparing the cost of sorting various data sets**
- **Variable Item Counts**
  - Ability to set the item count to any range from 0 to billions.
- **Variable data sorting**
  - Pre-Sorted
  - Reversed Sorted
  - Random Data
- **Measurement type**
  - Comparisons
  - Swaps (or merge sort assignments)



Item Count

100

Data Order

Random

Operation

Comparisons

Execute

Summary

- **Overview of Sorting**
  - Measuring Performance
- **Sorting Algorithms**
  - Bubble Sort
  - Insertion Sort
  - Selection Sort
  - Merge Sort
  - Quick Sort
- **Visualized Sorting Performance**
  - Sample Source Code and Demo

References

- **Wikipedia.org Sorting Articles**
  - http://en.wikipedia.org/wiki/Bubble_sort
  - http://en.wikipedia.org/wiki/Insertion_sort
  - http://en.wikipedia.org/wiki/Merge_sort
  - http://en.wikipedia.org/wiki/Selection_sort
  - http://en.wikipedia.org/wiki/Quicksort