Sorting

Tan Cher Wah (isstcw@nus.edu.sg)

Sorting

- Sorting involves putting elements of a list in a certain order
- Given a list of numbers:

8	4	-6	28	1

- Sort in Ascending order:

-6	1	4	8	28

- Sort in Descending order:

28	8	4	1	-6

Sorting Algorithms

- Bubble Sort
- Selection Sort
- Insertion Sort
- Quick Sort

Bubble Sort

Overview: Bubble Sort

- A simple sorting algorithm that repeatedly steps through the list
- Compares adjacent elements and swaps them if they are in the wrong order
- Repeats this process until the list is sorted

Steps: Bubble Sort

1. Start with an unsorted list N items.

2. Repeat the following until end of list
   a. Compare each pair of adjacent elements in the list.
   b. If the left element is greater than the right element, swap them.
   c. Move to the next pair of elements and repeat the comparison and swap if necessary.

3. Perform Step 2 (N-1) times.

Bubble Sort

Bubble Sort, in ascending order, a list of integers: 8, 4, -6, 28, 1

8	4	-6	28	1
4	8	-6	28	1
4	-6	8	28	1
4	-6	8	28	1
4	-6	8	1	28

iteration 1

4	-6	8	1	28
-6	4	8	1	28
-6	4	8	1	28
-6	4	1	8	28
-6	4	1	8	28

iteration 2

-6	4	1	8	28
-6	4	1	8	28
-6	1	4	8	28
-6	1	4	8	28
-6	1	4	8	28

iteration 3

-6	1	4	8	28
-6	1	4	8	28
-6	1	4	8	28
-6	1	4	8	28
-6	1	4	8	28

iteration 4

Bubble Sort

An implementation of Bubble Sort in C# (ascending order)

```
void BubbleSort(int[] list)
{
    for (int i=0; i<list.Length-1; i++) {
        for (int j=0; j<list.Length-1; j++)
        {
            if (list[j] > list[j+1]) {
                // swap the two
                int tmp = list[j];
                list[j] = list[j+1];
                list[j+1] = tmp;
            }
        }
    }
}
```

All elements would have been sorted after n-1 iterations

Ends at 2nd last element because comparing $element_j$ and $element_{j+1}$

Optimisation: Bubble Sort

<u>Observation</u>

- If no swaps were made in a pass, the list is already sorted

<u>Optimisation Step</u>

- Keep track of whether any swaps were made in a pass using a flag

- If no swaps are made in a pass, the list is already sorted, and the algorithm can stop early

Bubble Sort

An optimised implementation of Bubble Sort in C# (ascending order)

```
public static void Sort(int[] list) {
    for (int i=0; i<list.Length-1; i++) {
        bool swapped = false;

        for (int j=0; j<list.Length-1; j++) {
            if (list[j] > list[j+1]) {
                // swap the two
                int tmp = list[j];
                list[j] = list[j + 1];
                list[j + 1] = tmp;

                swapped = true;
            }
        }

        if (!swapped) {
            break;
        }
    }
}
```

Track if there are any swaps during a pass

If there are no swaps in a pass, the list is sorted

Bubble Sort (Ascending)

Calling the Bubble Sort code from our Main function

```
int[] list = { 8, 4, -6, 28, 1 };

BubbleSort(list);

for (int i=0; i<list.Length; i++) {
    Console.Write(list[i] + " ");
}
```

Output

```
-6, 1, 4, 8, 28
```

Selecton Sort

Overview: Selection Sort

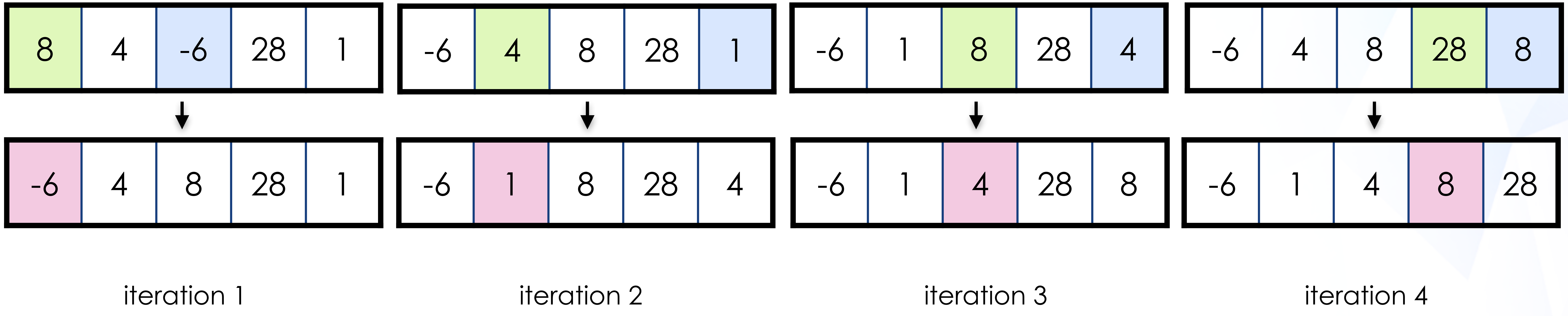
- A simple sorting algorithm that divides the input list into two parts: sorted and unsorted
- Repeatedly selects the minimum (or maximum) element from the unsorted part and moves it to the sorted part
- The list is sorted when there are no more elements in the unsorted part

Steps: Selection Sort

1. Start with an unsorted list.
2. Divide the list into two parts: the sorted part (initially empty) and the unsorted part (the entire list).
3. Find the minimum element in the unsorted part of the list.
4. Expand the sorted part by adding that minimum element to its end, and reduce the unsorted part by one element.
5. Repeat steps 3 to 4 until the entire list is sorted.

Selection Sort

Selection Sort, in ascending order, a list of integers: 8, 4, -6, 28, 1



Selection Sort

A C# Implementation of Selection Sort (in ascending order)

```
public void Sort(int[] arr) {
    for (int i=0; i < arr.Length-1; i++)
    {
        int min_idx = i;

        for (int j=i+1; j < arr.Length; j++)
        {
            if (arr[j] < arr[min_idx]) {
                // remember where min-value is
                min_idx = j;
            }
        }

        if (min_idx != i) {
            // swap the two
            int tmp = arr[i];
            arr[i] = arr[min_idx];
            arr[min_idx] = tmp;
        }
    }
}
```

After n-1 iterations, the $(n-1)^{th}$ element would be sorted; which follows that the n^{th} element is sorted as well

Examine every element with the min-value we have seen so far

Note down the location of a possible min-value for this iteration

Add this newly found min-value to our semi-sorted list (our semi-sorted list is in the front of the list)

Insertion Sort

Overview: Insertion Sort

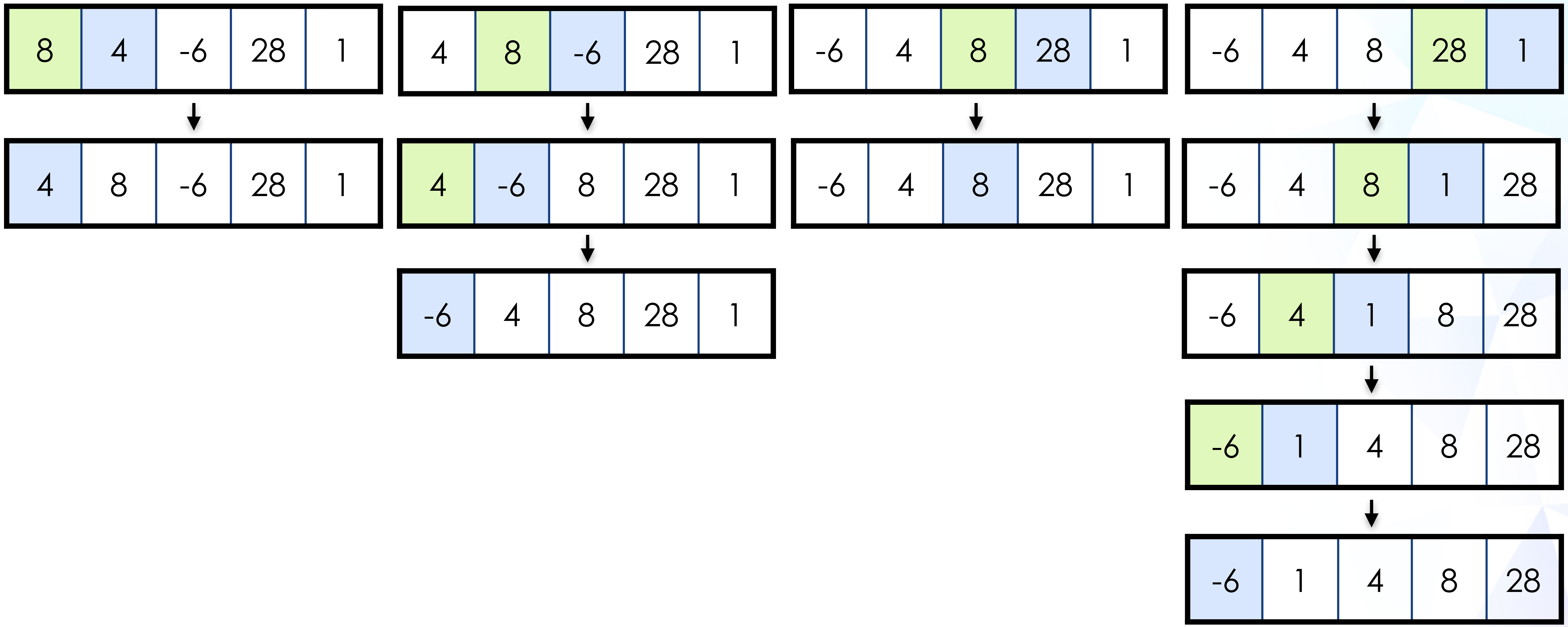
- Insertion Sort divides the input list into two parts: sorted and unsorted
- Each element, in the unsorted part, inserts itself in the correct sorting order within the sorted part
- The list is sorted when there are no more elements in the unsorted part

Steps: Insertion Sort

1. Start with an unsorted list.
2. Divide the list into two parts: the sorted part (initially with one element) and the unsorted part (the rest of the list).
3. Take the first element from the unsorted part.
4. Compare it to elements in the sorted part from right to left, and swap if that element is smaller than its immediate-left element.
5. Repeat step 4 until the immediate-left element of the sorted part is smaller than that element.
6. That element is now in the last position of the sorted part.
7. Repeat steps 3 to 5 until all elements are in the sorted part.

Insertion Sort

Insertion Sort a list of integers: 8, 4, -6, 28, 1



iteration 1 iteration 2 iteration 3 iteration 4

Insertion Sort

A C# Implementation of Insertion Sort (in ascending order)

```
void InsertionSort(int[] list)
{
    for (int i=1; i<list.Length; i++)
    {
        for(int pos=i; pos > 0; pos--) {
            if (list[pos] < list[pos - 1]) {
                // swap the two
                int tmp = list[pos - 1];
                list[pos - 1] = list[pos];
                list[pos] = tmp;
            }
            else {
                break;
            }
        }
    }
}
```

Must include the last element as it has not yet been examined and "inserted" into the correct order

Get the next element and "insert" it into the correct order

Keep checking against front elements; want to place the current element as front as possible

All front element will be smaller than this current element; no need to continue

QuickSort

Overview: QuickSort

- A widely used and efficient sorting algorithm that follows the divide-and-conquer approach
- It recursively divides the input list into smaller sublists, then sorts and combines them to produce the final sorted list

Steps: QuickSort

1. Choose a pivot element from the list (usually the first or last element).
2. Partition the list into two sublists:
   - Elements less than the pivot (left sublist)
   - Elements greater than or equal to the pivot (right sublist)
3. Recursively apply QuickSort to the left and right sublists.
4. Combine the sorted sublists and the pivot to produce the final sorted list.

QuickSort

Setup the Left, Right and Pivot markers for the sublist that we are working on

20	26	93	17	77	31	44	55	54
Left							Right	Pivot

Left and Right markers move towards each other

QuickSort

For the Left marker, keep moving right until a number larger than the Pivot value is found

20	26	93	17	77	31	44	55	54
		Left					Right	Pivot

93 > 54, so stop advancing Left marker

QuickSort

For the Right marker, keep moving left until a number smaller than or the same as the Pivot value is found

20	26	93	17	77	31	44	55	54
		Left				Right		Pivot

44 <= 54, so stop advancing Right marker

QuickSort

Swap the two values that are pointed by the Left and Right markers

20	26	44	17	77	31	93	55	54
		Left				Right		Pivot

44 and 93 are swapped in-place

QuickSort

Advance the Left and Right markers, with Left marker looking for a number greater than the Pivot value and Right marker a number smaller than or same as the Pivot value

20	26	44	17	77	31	93	55	54
				Left	Right			Pivot

Left and Right markers stopped at their respective positions

QuickSort

Swap the values that the Left and Right markers are pointing to

20	26	44	17	31	77	93	55	54
				Left	Right			Pivot

31 and 77 are swapped in-place

QuickSort

Advance the Left and Right markers as before, but do not swap when Left marker position >= Right marker position

20	26	44	17	31	77	93	55	54
				Right	Left			Pivot

31 and 77 are NOT swapped

QuickSort

Instead, swap the Pivot value with the value of the Left marker

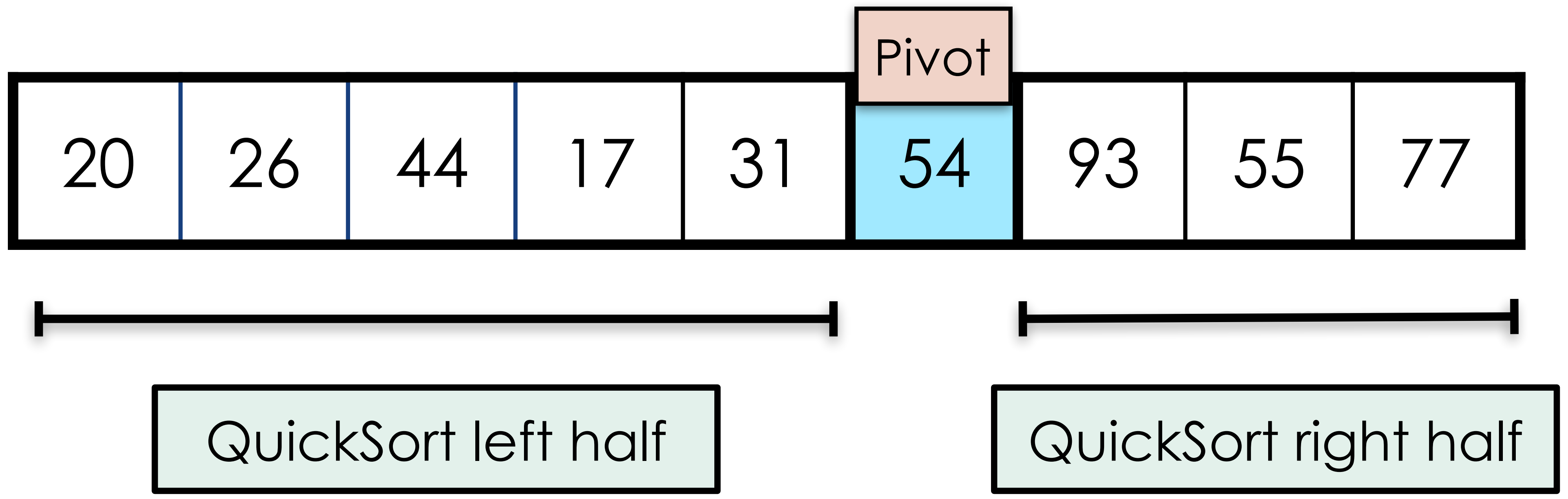
20	26	44	17	31	54	93	55	77
					Pivot			

Now, notice that:

- All values on the left <= 54
- All values on the right > 54

QuickSort

Now, repeat the process for the Left and Right sublists of the Pivot



Quick Sort

Calling the QuickSort code from our Main function

```
int[] list = { 0, 8, 4, -6, 2, 28, 1 };

QuickSort(list, 0, list.Length - 1);

for (int i = 0; i < list.Length; i++) {
   Console.Write(list[i] + " ");
}
```

Work on full length of list at the start

Output

```
-6, 0, 1, 2, 4, 8, 28
```

QuickSort

A C# Implementation of QuickSort (in ascending order)

```
static void QuickSort(int[] list, int left, int right)
{
    if (left >= right) {
        return;
    }

    int pivotIdx = Partition(list, left, right);

    QuickSort(list, left, pivotIdx - 1);
    QuickSort(list, pivotIdx + 1, right);
}
```

Partition a range of the given list

New pivot point after partitioning the given range

Recursively sort smaller ranges

QuickSort

```
int Partition(int[] list, int left, int right) {
    int low = left;
    int pivot = right;
    right = pivot - 1;

    while (true) {
        while (left < pivot && list[left] <= list[pivot]) {
            left++;
        }

        while (right > low  && list[right] > list[pivot]) {
            right--;
        }

        if (left < right) {
            swap(ref list[left], ref list[right]);
        }
        else {
            break;
        }
    }

    swap(ref list[left], ref list[pivot]);
    return left;
}
```

Work on a range of the original list

At this point,
- left-element > pivot-element
- right-element <= pivot-element

Our two pointers have crossed each other or at the same position; we have finished examining the range of elements

```
void swap(ref int a, ref int b)
{
    int tmp = a;
    a = b;
    b = tmp;
}
```

Implementation of the Partition function that partitions a range of elements with respect to a pivot.

When the Partition function exits, keys on the left of the pivot are smaller than or equal to the pivot-value, while keys on the right are larger than pivot-value.

The End