fit@hcmus

PROGRAMMING TECHNIQUES

Week 9b: Sorting – Part 2

Faster Sorting Algorithms

Nguyễn Hải Minh - 04/2024

Today content

- Faster Sorting Algorithms
  - Merge Sort
  - Quick Sort
- Properties of Sorting
- Sorting with different criteria

FASTER SORTING ALGORITHMS

- Merge Sort
- Quick Sort

Merge Sort - Idea

- Support we only know how to merge two sorted sets of items into one
  - Merge {1, 5, 7} with {2, 9} → {1, 2, 5, 7, 9}
- But where do we get the two sorted sets in the first place?
- Idea:
  - Merge each pair of items into sets of 2
  - Merge each pair of sets of 2 into sets of 4
  - Repeat previous step for sets of 4
  - Final step: merge two sets of n/2 items to obtain a fully sorted set

Divide and Conquer Method

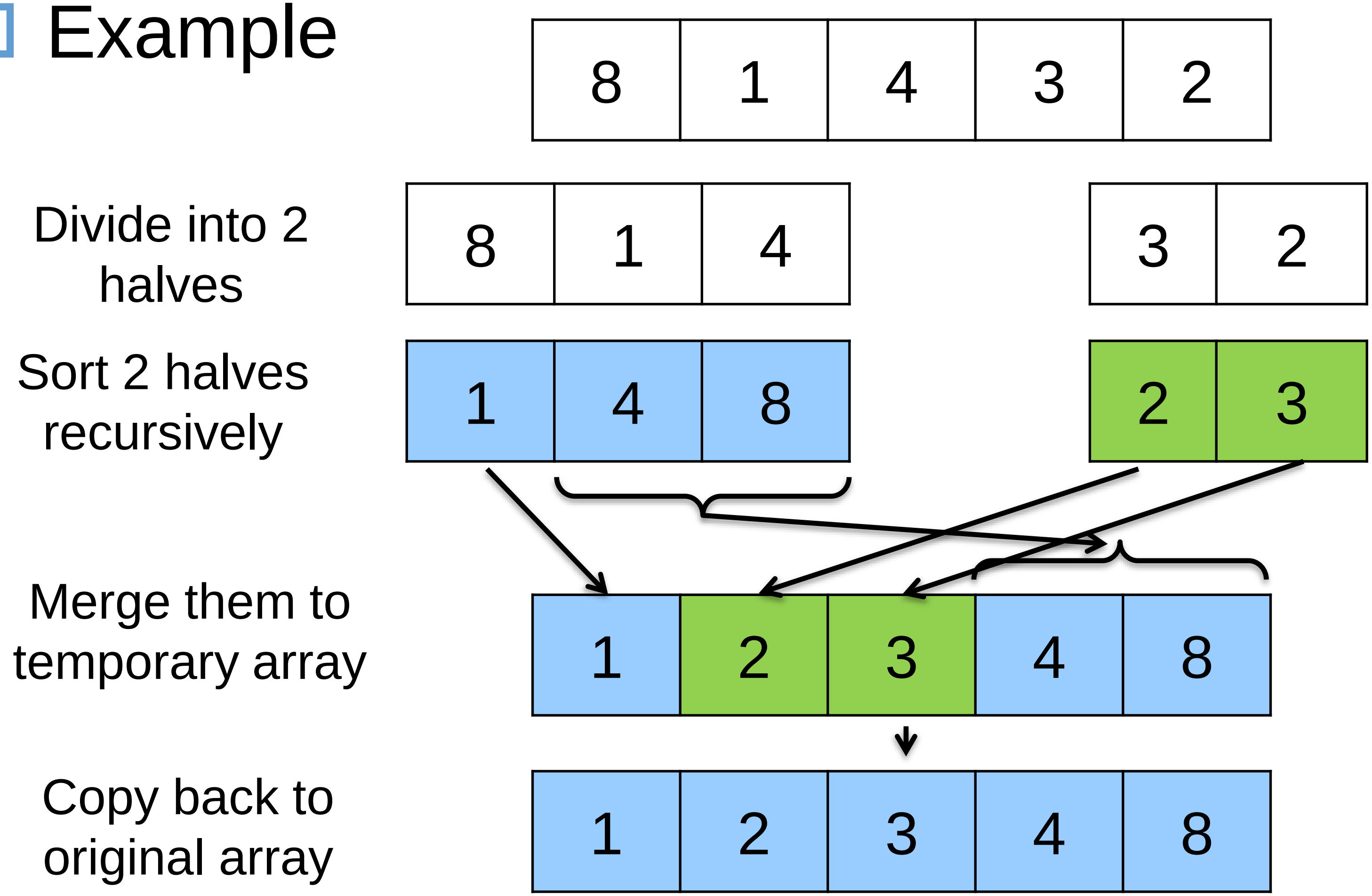
- A powerful problem-solving technique
- Divide-and-conquer method solves problem in the following steps:
  - Divide step:
    - Divide the large problem into smaller problems
    - Recursively solve the smaller problems
  - Conquer step:
    - Combine the results of smaller problems to produce the result of the larger problem

Divide and Conquer – Merge Sort

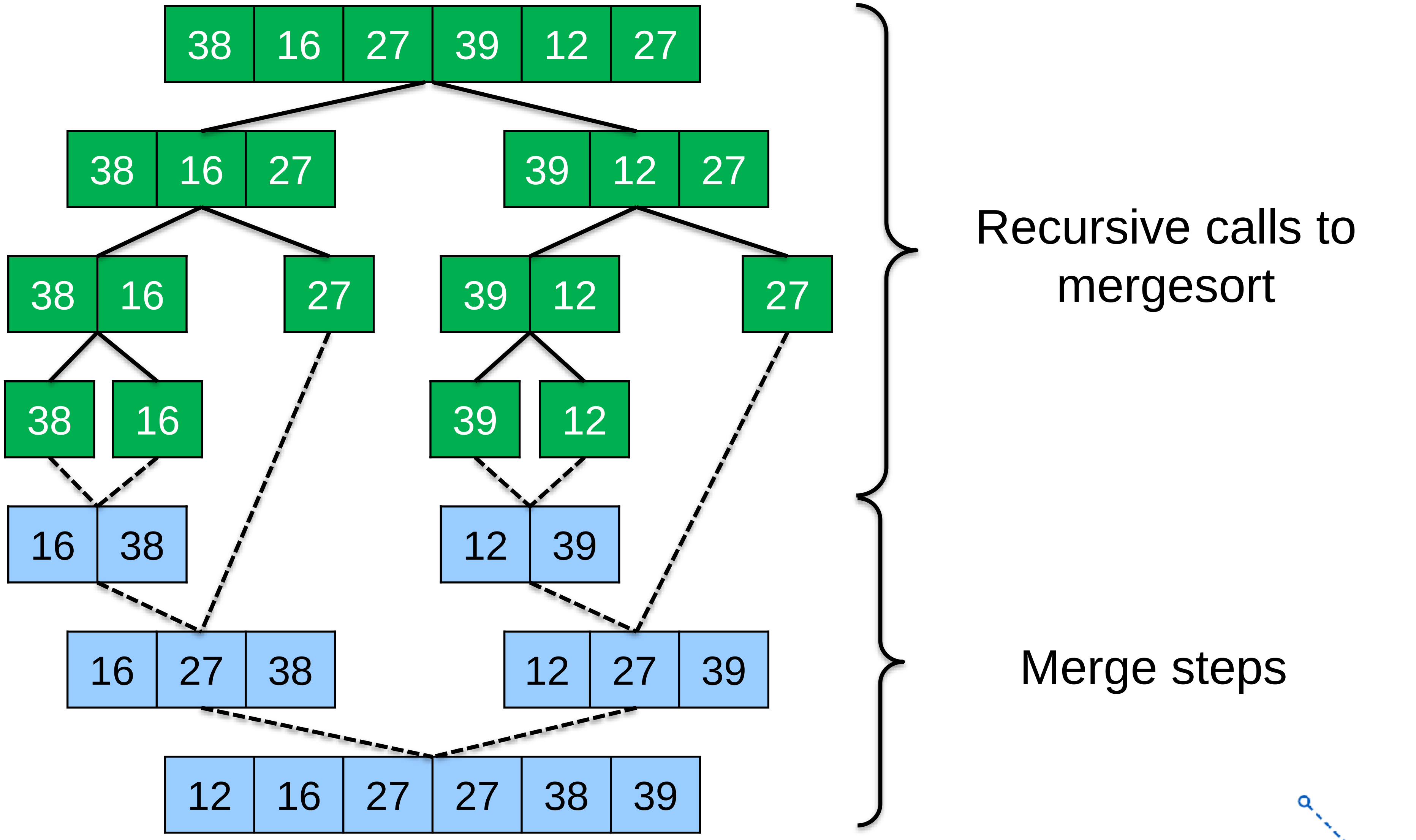
- Merge sort is a divide and conquer sorting algorithm
- Divide step:
  - Divide the array into two (equal) halves
  - Recursively sort the two halves
- Conquer step:
  - Merge the two halves to form a sorted array

Merge Sort - Illustration

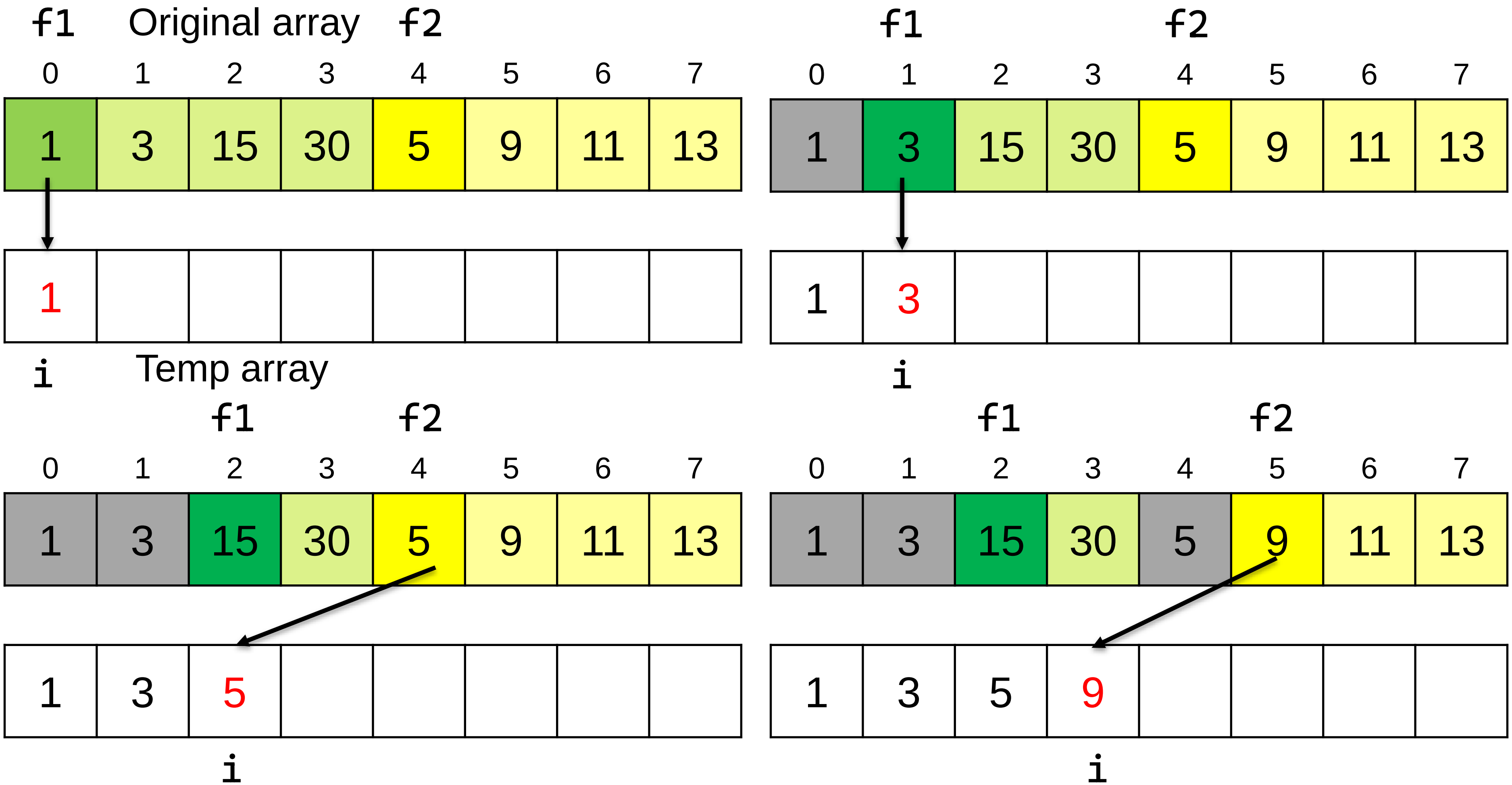
- Example



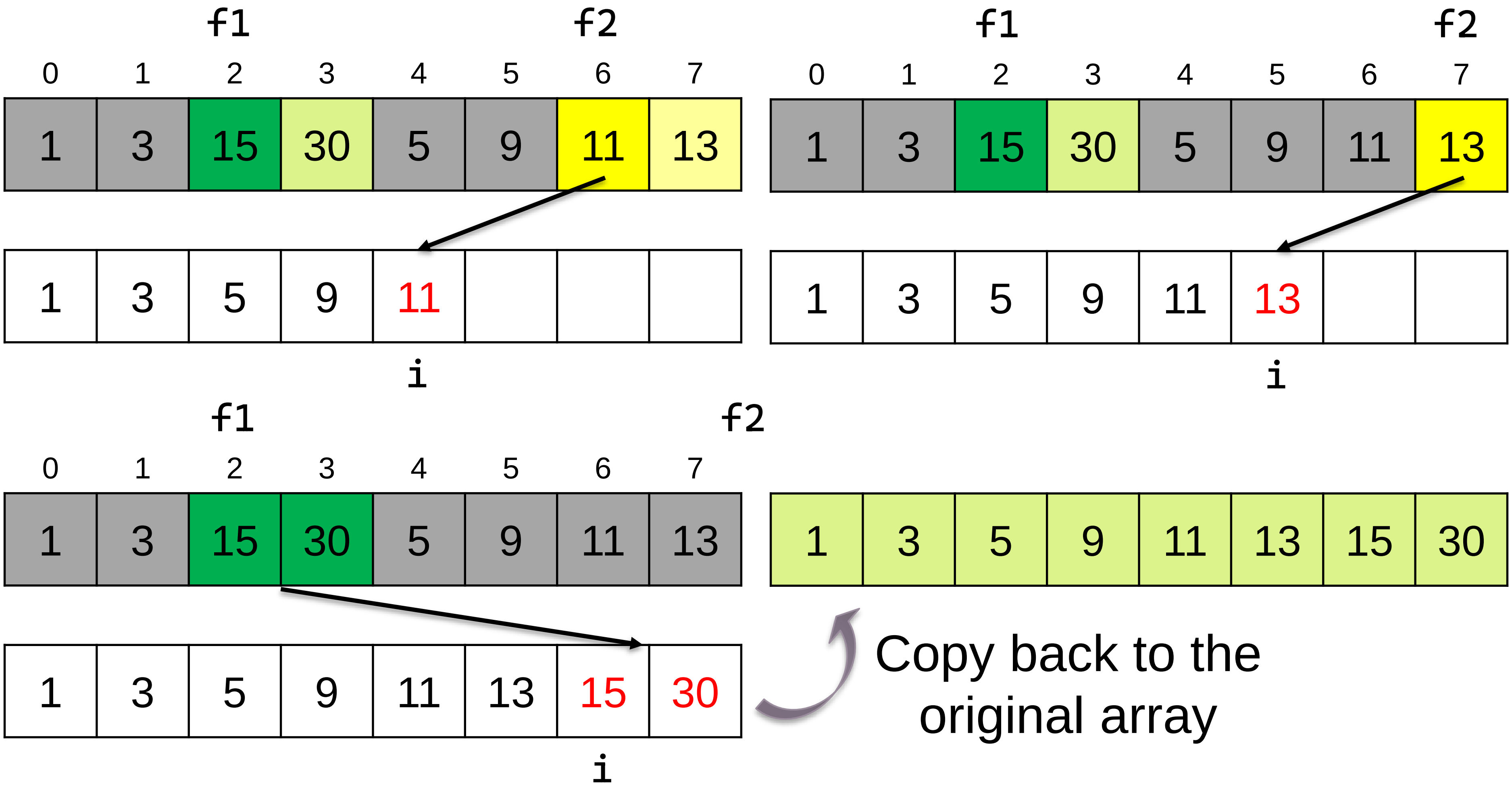
Merge Sort - Illustration



Merge Sort – Merge Steps



Merge Sort – Merge Steps



Merge Sort – Implementation

```
void MergeSort(int arr[], int first, int last)
{
   if (first >= last)
      return;
   int mid = (first + last) / 2;
   MergeSort(arr, first, mid);
   MergeSort(arr, mid + 1, last);
   Merge(arr, first, mid, last);
}
```

Merge Sort – Implementation

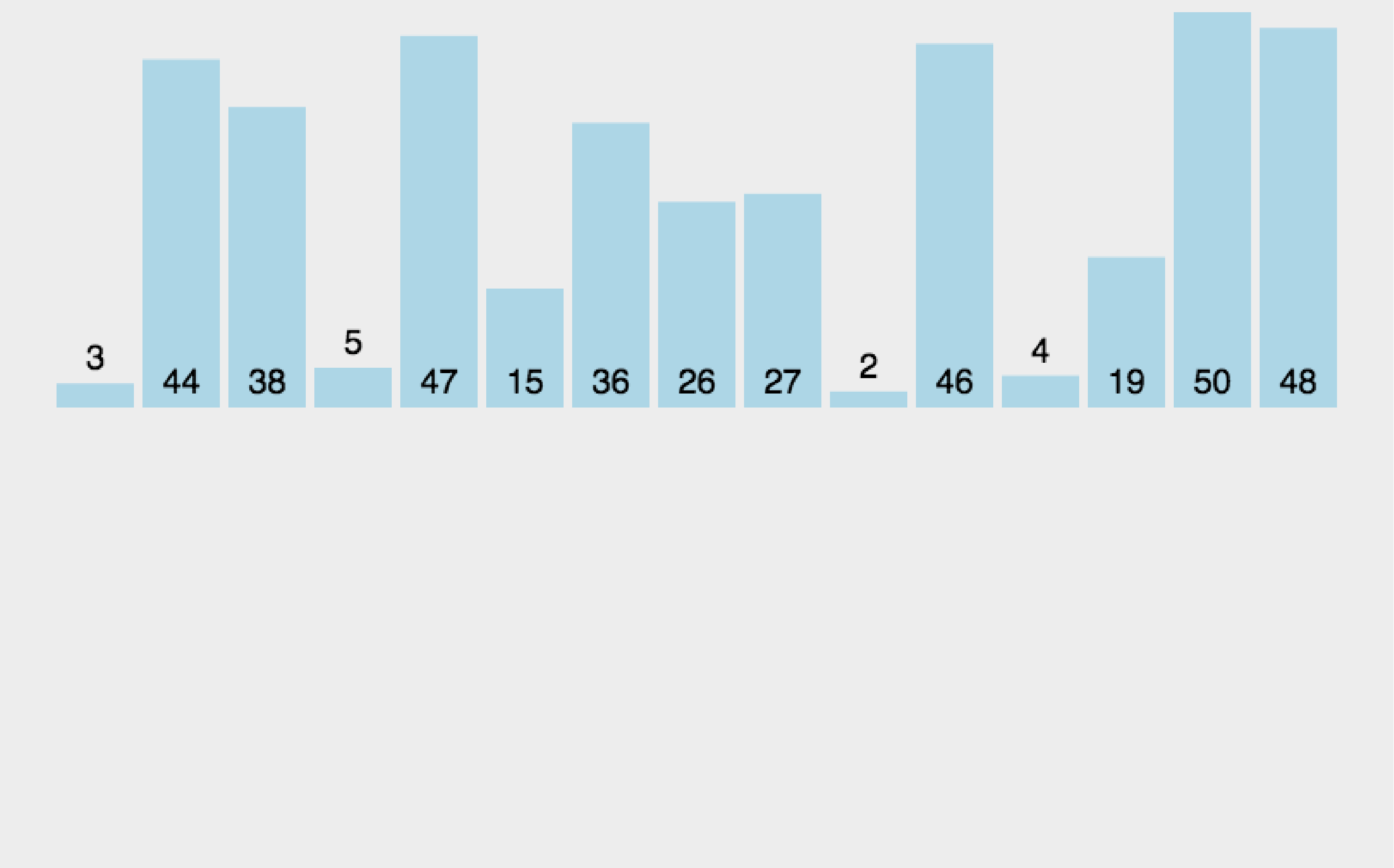
```
void Merge(int arr[], int first, int mid, int last)
{
  //Temporary array to merge 2 sub array: a[first...mid] and a[mid+1...last-1]
  int* tmp_arr = new int[last + 1];
  int f1 = first, l1 = mid;
  int f2 = mid + 1, l2 = last;
  int i = first;

  while ((f1 <= l1) && (f2 <= l2)) {
     if (arr[f1] < arr[f2]) {
        tmp_arr[i] = arr[f1];
        f1++;
     }
     else {
        tmp_arr[i] = arr[f2];
        f2++;
     }
     i++;
  }
```

Merge Sort – Implementation

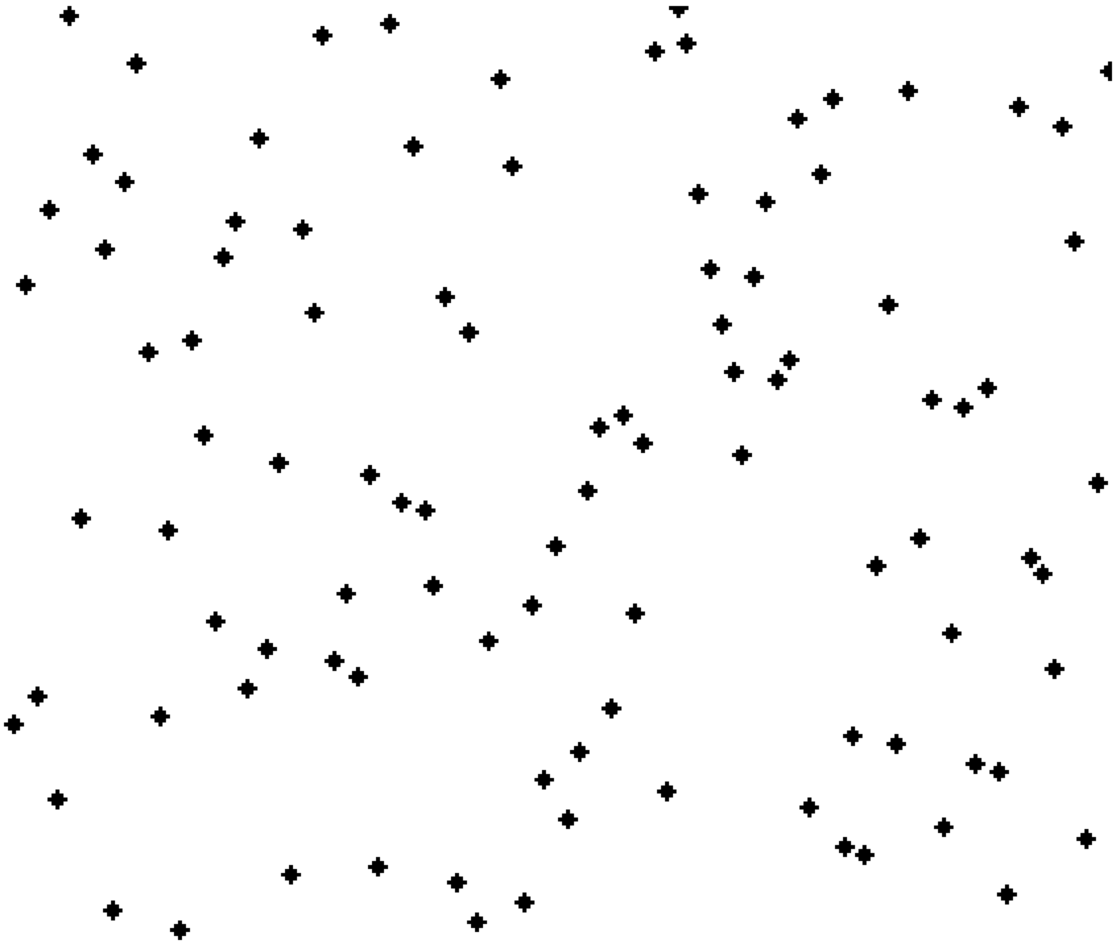
```
  //At this step, one sub array has no item left
  while (f1 <= l1) { //left subarray has items
     tmp_arr[i] = arr[f1];
     f1++;
     i++;
  }
  while (f2 <= l2) { //right subarray has items
     tmp_arr[i] = arr[f2];
     f2++;
     i++;
  }
  //Copy back to the original array arr
  for (i = first; i <= last; i++)
     arr[i] = tmp_arr[i];
  delete[] tmp_arr;
}
```

Merge Sort – Visualization



[Click here to view animation]

Merge Sort – Visualization



[Click here to view animation]

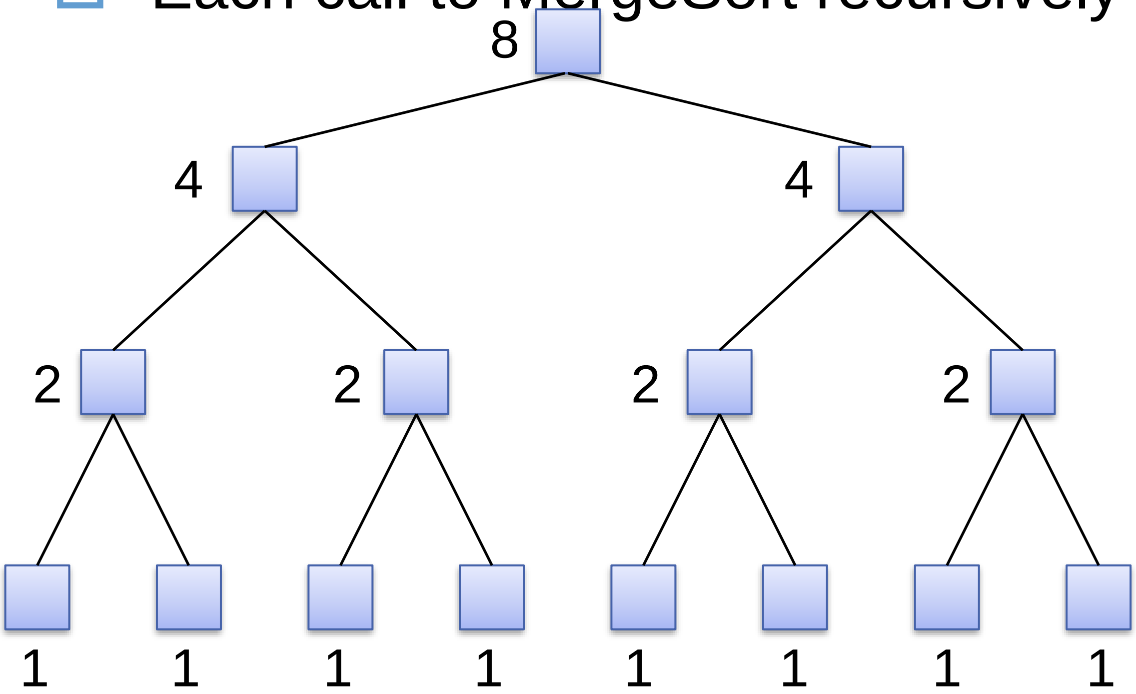
Merge Sort - Analysis

- Most of the effort in merge sort algorithm is in the merge step.
- If the total number of items in the two arrays to be merged is *n*, then the number of comparisons in the worst case is $n-1$
- Number of data moves:
  - Original array → Temporary array: $n-1$
  - Temporary array → Original array: $n-1$
- Total number of major operations of merge step:

$$3 \times (n-1)$$

Merge Sort - Analysis

- Each call to MergeSort recursively calls itself twice



Level 0: Mergesort 8 items

Level 1: 2 calls to Mergesort 4 with items each

Level 2: 4 calls to Mergesort with 2 items each

Level 3: 8 calls to Mergesort with 1 item each

- How many levels of recursive calls to Mergesort?

$\log_2 n$ or $1+\log_2 n$

Merge Sort - Analysis

- Level 0: $3 \times (n - 1)$ operations
- Level 1: $3 \times \left(\frac{n}{2} - 1\right) \times 2$ operations
- Level 2: $3 \times \left(\frac{n}{4} - 1\right) \times 4$ operations
- Level m: $3 \times \left(\frac{n}{2^m} - 1\right) \times 2^m$ operations
- In both worst case and average cases, merge sort algorithm requires about:

$$3 \times n \times (1 + \log_2 n) - C \quad \text{operations}$$

Merge Sort – Pros and Cons

- Pros:
  - 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
  - Require additional storage during merging operation

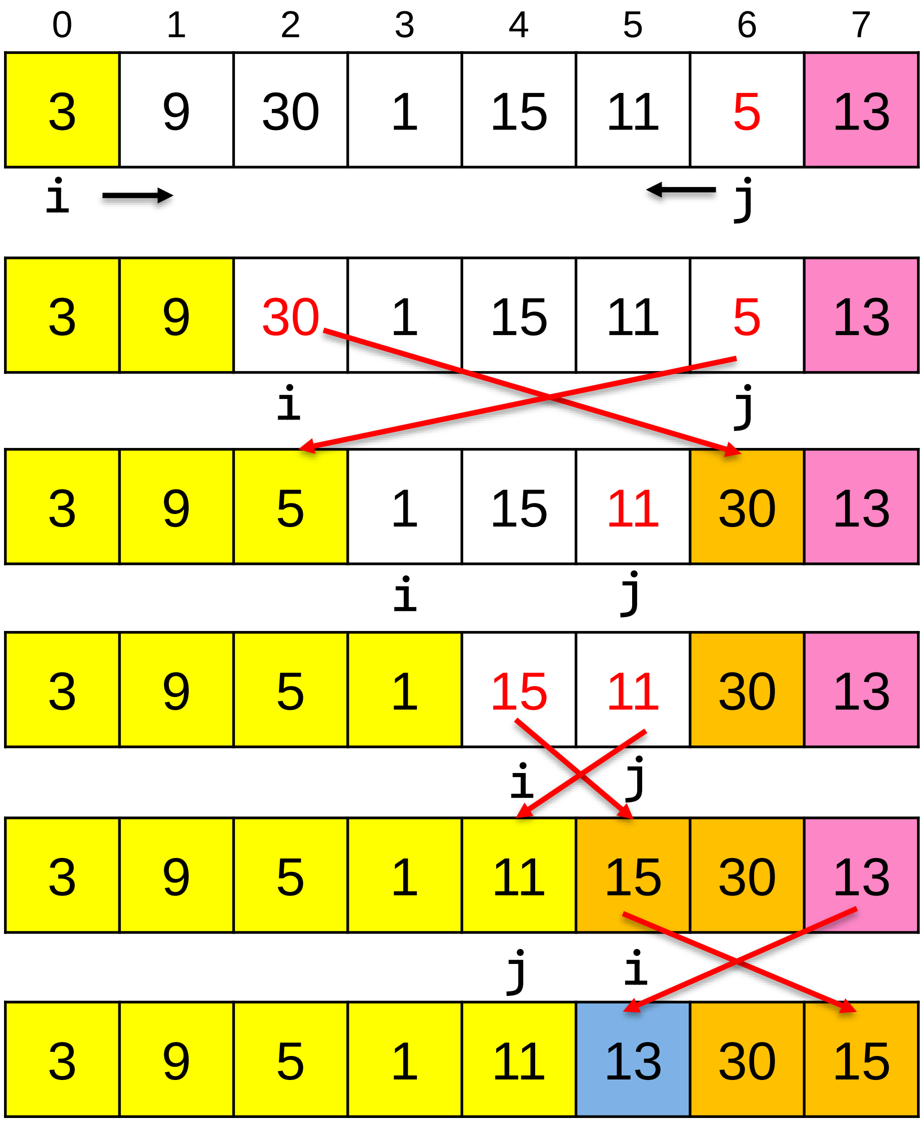
Quick Sort – Idea

- Quick sort is a divide and conquer sorting algorithm
- Divide step:
  - Choose an item `A[p]` (known as pivot) and partition the items of A[i…j] into two parts
    - Items that are smaller than `A[p]`
    - Item that are greater than or equal to `A[p]`
  - Put `A[p]` to the correct position (in the final sorted array)
  - Recursively sort the two parts `A[i…p-1]` and `A[p+1…j]`
- Conquer step:
  - Do nothing!

Quick Sort – Idea

- How to choose pivot:
  - The first element `A[0]`
  - The last element `A[j-1]`
  - The middle element `A[(j-1)/2]`
  - Randomly
  - …

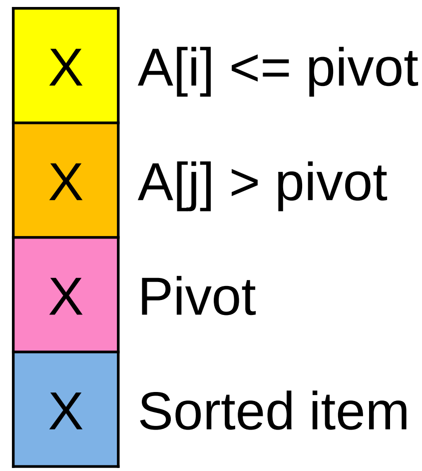
Quick Sort - Illustration



An inversion occurs:

- A[j] < pivot
- A[i] > pivot

→ Swap A[i], A[j]



Finally, swap A[i] and pivot

→ pivot is at correct position

Quick Sort - Illustration

0	1	2	3	4	5	6	7
3	9	5	1	11	13	30	15
i	→		←	j			
3	9	5	1	11	13	30	15
1	9	5	3	11	13	30	15
1	3	5	9	11	13	30	15
1	3	5	9	11	13	30	15
1	3	5	9	11	13	30	15
1	3	5	9	11	13	15	30

Quick Sort – Implementation

```
void QuickSort(int arr[], int low, int high)
{
   if (low >= high)
      return;
   int p = Partition(arr, low, high);
   QuickSort(arr, low, p - 1);
   QuickSort(arr, p + 1, high);
}
```

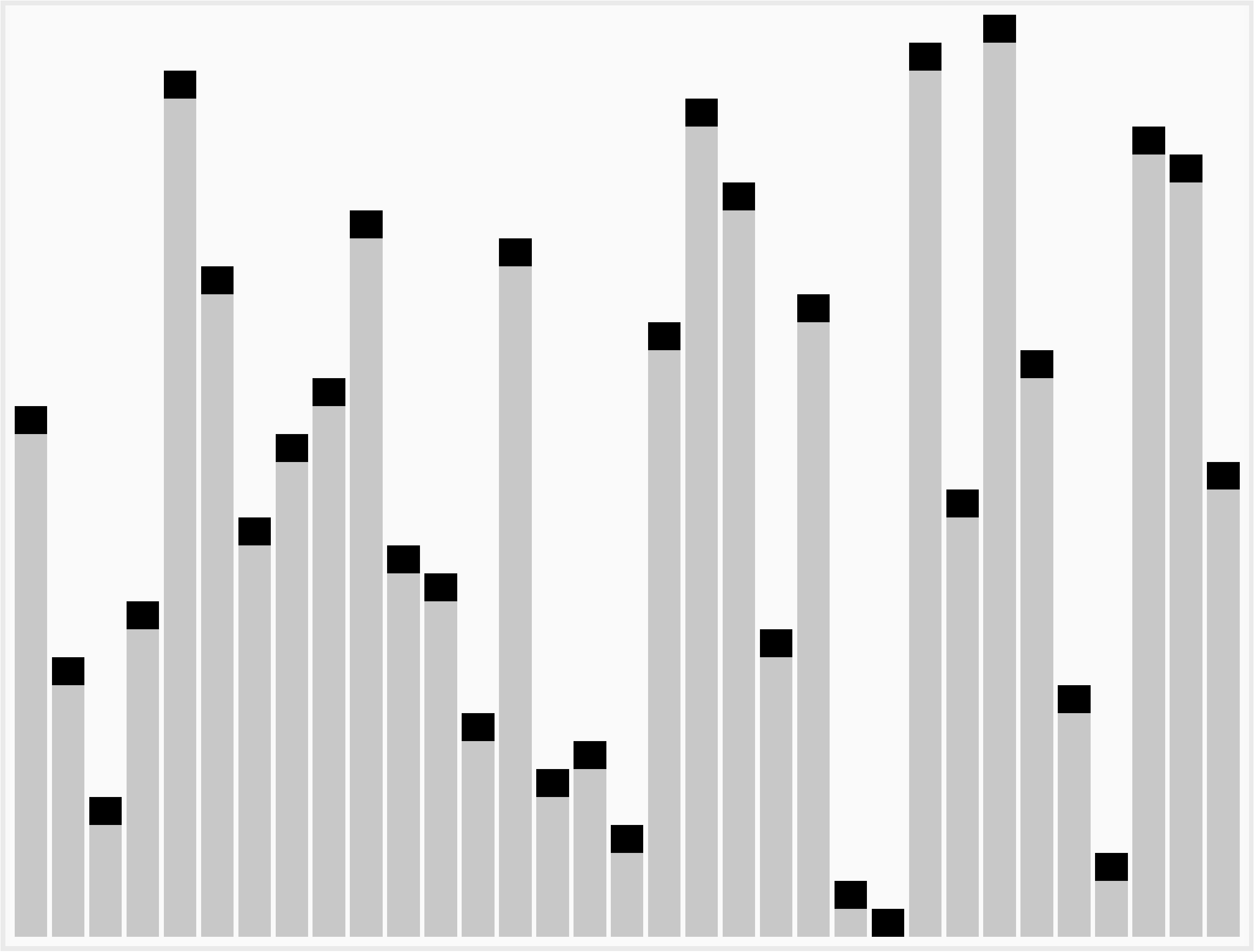
Quick Sort – Implementation

```
int Partition(int arr[], int i, int j)
{
   int pivot = j; //pivot is the last element
   j--;//traverse the array from i to j-1
   while (i <= j) {
      while (arr[i] < arr[pivot] && i <= j) i++;
      while (arr[j] > arr[pivot] && j >= i) j--;
      if (i <= j) //find the inverse-elements pair (a[i]>a[pivot], a[j] <a[pivot])
      {
         Swap(arr[i], arr[j]);
         i++;
         j--;
      }
   }
   Swap(arr[i], arr[pivot]); //move the pivot to the correct position of array
   return i; //arr[i] is now in the correct position
}
```

Quick Sort – Implementation

- Quick sort is a divide and conquer sorting 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*
    - Item that are greater than or equal to *p*
  - Recursively sort the two parts
- Conquer step:
  - Do nothing!

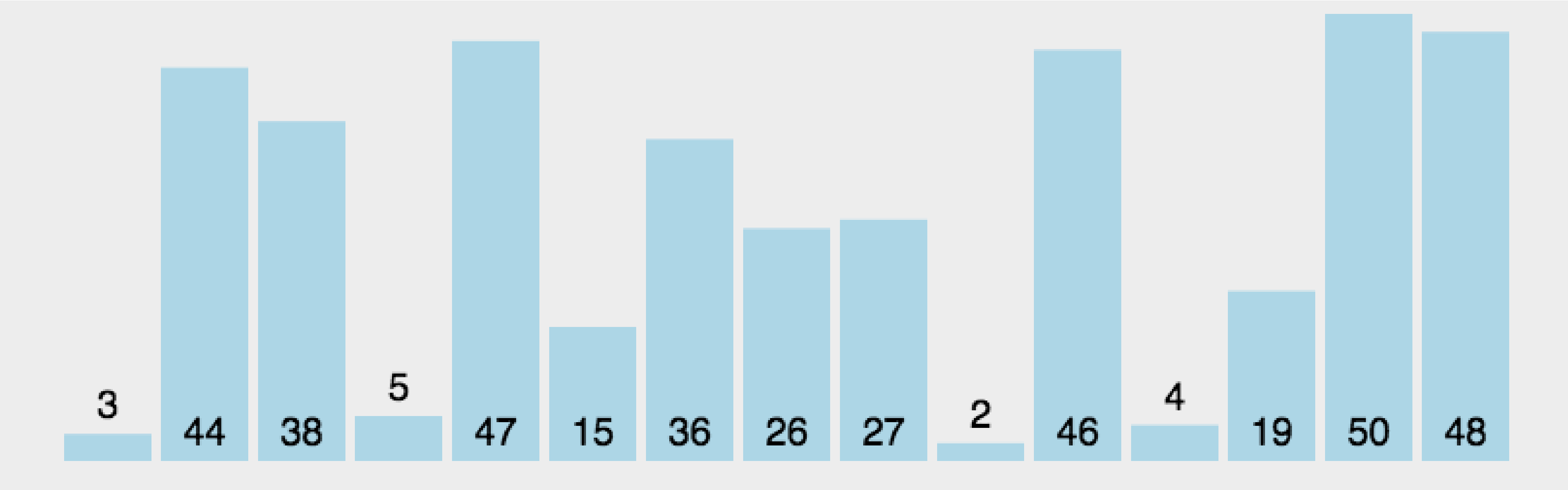
Quick Sort – Visualization



Click here to view animation

Quick Sort – Visualization

- Another version of Quick Sort:
  - Pivot it the first element
  - Different way to partition 2 sub arrays at each step



[Click here to view animation]

Quick Sort – Pros and Cons

- Pros:
  - The performance in average case is far better than its worst case
  - In most situation, quick sort is better than merge sort
- Cons:
  - In the worst case, quick sort is significant slower than merge sort.
  - What is the worst case of quick sort?
    - Pivot is the largest/smallest element

Exercises

1. Show Merge Sort, Quick Sort algorithms step by step when applied to the following array:

   A = {15, 39, 26, 47, 33, 10, 25}
2. Rewrite `MergeSort` to sort an array decreasingly
3. Rewrite the `partition` function so that:
   a) Pivot is the first element
   b) Pivot is the middle element

PROPERTIES OF SORTING ALGORITHMS

Properties of Sorting Algorithms

- Stability:
  - A sorting algorithm is stable if it preserves the relative order of equal elements in the input array
- Time complexity:
  - Depends on the number of comparisons/swaps they make.
- Space complexity:
  - In-place sort: do not use extra memories
- Adaptivity:
  - Reduce the complexity if the array is nearly sorted
- Comparison or not

SORTING WITH DIFFERENT COMPARISON CRITERIA

Different Comparison Criteria

- Sorting an array based on different keys
- Example: sort an array of students according to:
  - Age
  - Height
  - Score
  - Weight
  - …

Sorting function in C++

- The qsort() function in C++ sorts a given array in ascending order using Quicksort algorithm.
- Protorype:

```
void qsort (void* base, size_t num,size_t size,
        int (*compare)(const void*,const void*));
```

- Syntax:

```
qsort(A, n, sizeof(int), compare);
```

Pointer to the array to sort

Number of elements

Size of each element in bytes

Comparison function

qsort – compare 2 integers

```
int compare(const void* x, const void* y)
{
    int a = *(const int*)x;
    int b = *(const int*)y;
    if (a > b)
        return 1;
    else if (a < b)
        return -1;
    return 0;
}
```

qsort – compare 2 strings

```
int compare_str(const void* x, const void* y)
{
   const char* s1 = *(const char**)x;
   const char* s2 = *(const char**)y;
   return strcmp(s1, s2);
}
```

Exercise

1. Write a program to allow user to choose the criterion for sorting a list of students (by name, age, height, weight, gpa)

→ Hint: *use function pointer to call the corresponding compare function in the qsort function.*


Q&A