

Merge Sort

- Merge sort is a **divide-and-conquer** algorithm that sorts an array by **repeatedly dividing it in half** and **merging the sorted sub-arrays**.
- The time complexity of merge sort is $O(n \log n)$, where n is the number of elements in the array.
 - Making it more efficient than other sorting algorithms with a **quadratic time** complexity (i.e., bubble sort and insertion sort.)
 - The polynomial equation whose highest **degree is two** is called a quadratic equation.
 - Form of Quadratic equation
 - $ax^2 + bx + c = 0$; where a, b, c are real numbers and $a \neq 0$.

Merge Sort is a **stable sort**?

Merge sort is a **stable sort**, meaning that it preserves the relative order of elements with equal keys.

- if two elements have the same key value, their **relative order in the sorted array** will be the same as their **relative order in the original unsorted array**

Example

- Sort an array of people with their **ages as the key** (i.e., by age)
 - a stable sort will keep people of the same age in the same order as they were in the original array.
 - In contrast, an unstable sort may change the relative order of elements with the same key value.

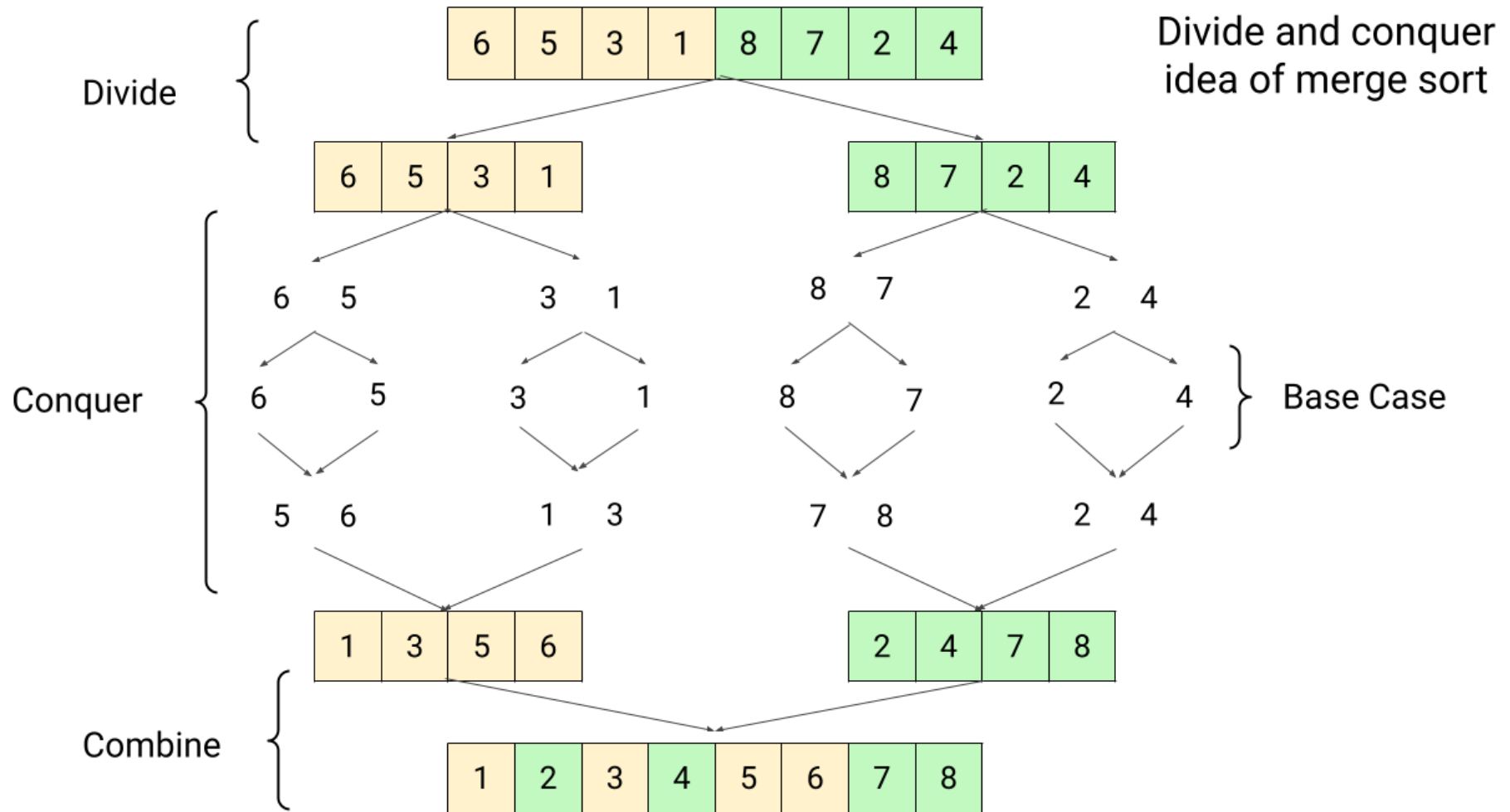
Merge Sort is a divide-and-conquer algorithm?

Merge sort is a sorting algorithm that works by

- **dividing** an array into smaller subarrays,
- sorting each subarray,
- merging the sorted subarrays back together to form the final sorted array.

1. **Divide:** If S has zero or one element, return S immediately; it is already sorted. Otherwise (S has at least two elements), remove all the elements from S and put them into two sequences, S_1 and S_2 , each containing about half of the elements of S ; that is, S_1 contains the first $\lfloor n/2 \rfloor$ elements of S , and S_2 contains the remaining $\lceil n/2 \rceil$ elements.
2. **Conquer:** Recursively sort sequences S_1 and S_2 .
3. **Combine:** Put back the elements into S by merging the sorted sequences S_1 and S_2 into a sorted sequence.

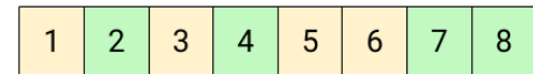
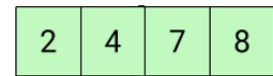
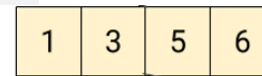
Merge Sort is a divide-and-conquer algorithm?



Merge or Combine Operation

```
1 def merge(left, right):
2     i = j = 0
3     merged = []
4     while i < len(left) and j < len(right):
5         if left[i] <= right[j]:
6             merged.append(left[i])
7             i += 1
8         else:
9             merged.append(right[j])
10            j += 1
11    while i < len(left):
12        merged.append(left[i])
13        i += 1
14    while j < len(right):
15        merged.append(right[j])
16        j += 1
17    return merged
18 arr1 = [1, 3, 5, 6]
19 arr2 = [2, 4, 7, 8]
20 sorted_arr = merge(arr1, arr2)
21 print(sorted_arr)
```

Combine



[1, 2, 3, 4, 5, 6, 7, 8]

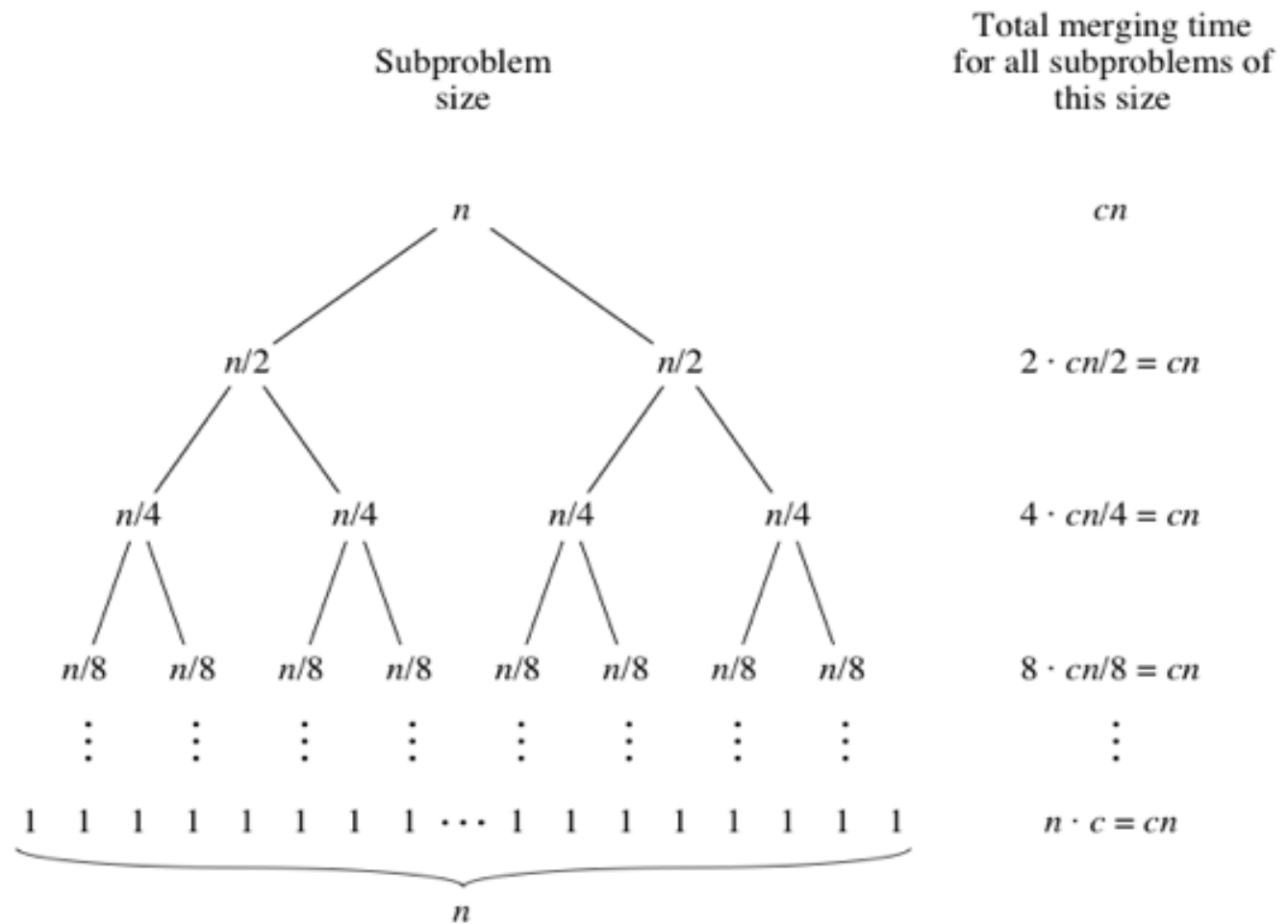
```
1 def merge_sort(arr):
2     if len(arr) <= 1:
3         return arr
4     else:
5         middle = len(arr) // 2
6         left_half = merge_sort(arr[:middle])
7         right_half = merge_sort(arr[middle:])
8         return merge(left_half, right_half)
9
10 def merge(left, right):
11     i = j = 0
12     merged = []
13     while i < len(left) and j < len(right):
14         if left[i] <= right[j]:
15             merged.append(left[i])
16             i += 1
17         else:
18             merged.append(right[j])
19             j += 1
20     while i < len(left):
21         merged.append(left[i])
22         i += 1
23     while j < len(right):
24         merged.append(right[j])
25         j += 1
26     return merged
27 arr = [5, 4, 3, 2, 1]
28 sorted_arr = merge_sort(arr)
29 print(sorted_arr) # [1, 2, 3, 4, 5]
```

Complexity of Merge Sort

We assume that we're sorting a total of n elements in the entire array.

1. The divide step takes constant time, regardless of the subarray size. After all, the divide step just computes the midpoint q of the indices p and r . Recall that in big- Θ notation, we indicate constant time by $\Theta(1)$.
2. The conquer step, where we recursively sort two subarrays of approximately $n/2$ elements each, takes some amount of time, but we'll account for that time when we consider the subproblems.
3. The combine step merges a total of n elements, taking $\Theta(n)$ time.

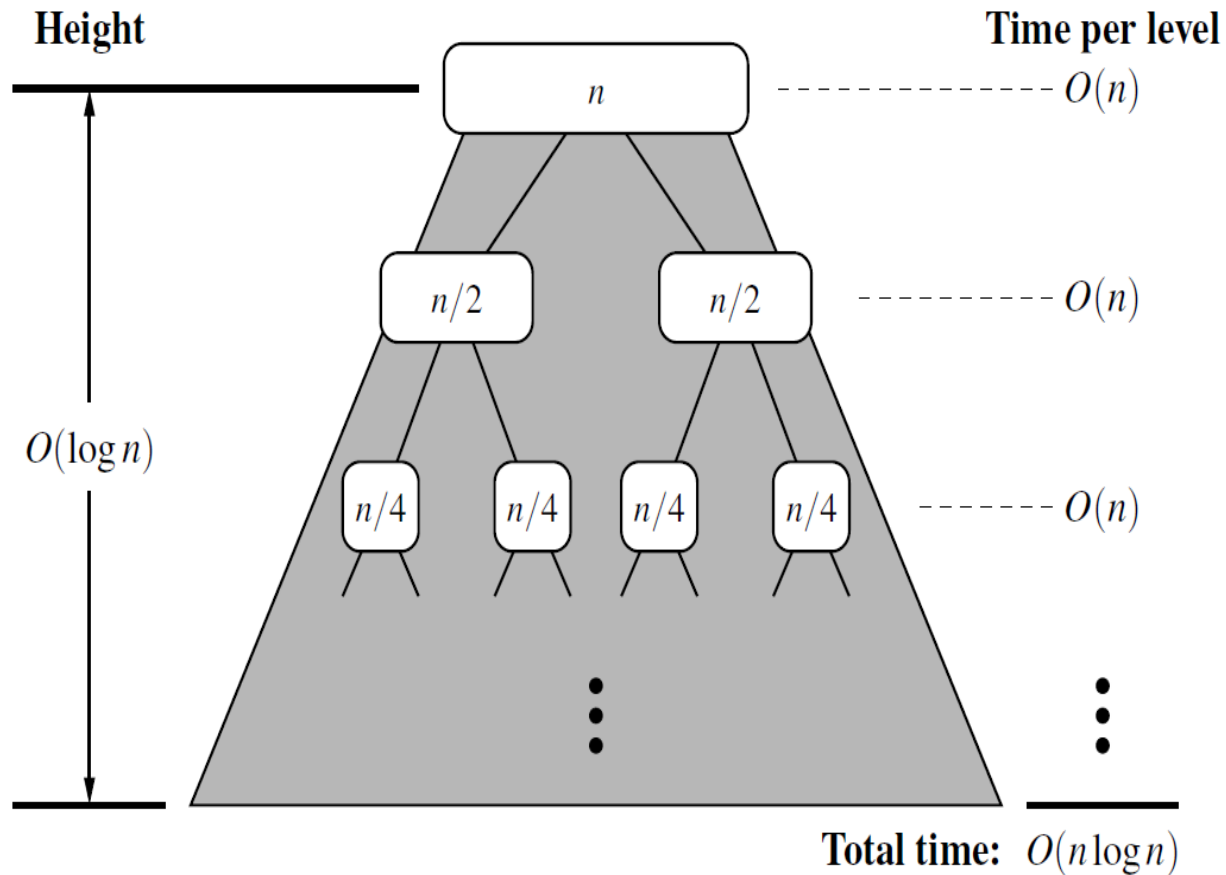
Complexity of Merge Sort



- As the subproblems get smaller, the number of subproblems doubles at each "level" of the recursion, but the merging time halves.
- The doubling and halving cancel each other out, and so the total merging time is cn at each level of recursion.

So time complexity of merge sort = Cost sum at each level = $O(\log n) \cdot O(n) = O(n \log n)$

Complexity of Merge Sort



So time complexity of merge sort = Cost sum at each level = $O(\log n) * O(n) = O(n \log n)$

Complexity of Merge Sort

Merge sort space complexity depends on the extra space used by the merging process and the size of recursion call stack used by the recursion.

- Space complexity of merging process = $O(n)$
- Space complexity for recursion call stack = Height of merge sort recursion tree = $O(\log n)$.
- Space complexity of merge sort algorithm = $O(n) + O(\log n) = O(n)$

As we have seen here, space complexity is dominated by the extra space used by the merging process.

Application of Merge Sort

- To sort linked lists in $O(n \log n)$ time:
- Inversion count problem
 - the inversion count problem tells how many pairs need to be swapped in order to get a sorted array. Merge sort works best for solving this problem.
- Merge sort is an external sorting technique.
 - if we have data of 1GB but the available RAM size is 500MB, then we will use merge sort.
 - When the data being sorted is too large to fit in the primary memory (often RAM) of a computing device and must instead reside in the slower external memory (usually a hard drive).
 - Whenever we have an input size larger than the RAM size, we use merge sort. Thus, merge sort is very well suited for larger datasets.

Drawbacks of Merge

- Merge sort is not a space-efficient algorithm. It makes use of an extra $O(n)$ space.
- In the case of smaller input size, merge sort works slower in comparison to other sorting techniques.
- If the data is already sorted, merge sort will be a very expensive algorithm in terms of time and space.
 - This is because it will still traverse the whole array and perform all the operations.

Quick sort Algorithm - Tony Hoare in 1960

- Tony Hoare developed the quicksort algorithm in the early 1960s while working at the National Physical Laboratory in the United Kingdom.
 - Merge sort was one of the sorting algorithms that existed at the time
- In the case of quicksort, the algorithm uses an **in-place partitioning** technique,
 - which means it doesn't require any additional memory allocation during the sorting process.
- The space complexity of both merge sort and quicksort is $O(n)$ and the stack space used during the recursion is $O(\log n)$ - which is considered as constant space.
- Quick sort is not a **stable** algorithm

Quick sort Algorithm - Tony Hoare in 1960

- Quick Sort follows a recursive algorithm.
- It uses the idea of divide and conquer approach.
- It divides the given array into **two sections (i.e. two subarray)** using a partitioning element called as **pivot**.

How is the division performed?

- All the elements to the **left side** of pivot are **smaller** than pivot.
- All the elements to the **right side** of pivot are **larger** than pivot.
- Then, sub arrays are **sorted separately** by applying quick sort algorithm **recursively**.

```

def quicksort(A, low, high):
    # Base case: if the subarray has 0 or 1 element, it is already sorted
    if low < high:
        # Partition the subarray and get the partition point
        p = hoare_partition(A, low, high)
        # Recursively sort the left subarray
        quicksort(A, low, p)
        # Recursively sort the right subarray
        quicksort(A, p + 1, high)

def hoare_partition(A, low, high):
    # Choose the first element as the pivot
    pivot = A[low]
    # Initialize the left pointer just before the first element
    i = low - 1
    # Initialize the right pointer just after the last element
    j = high + 1
    # Continue until the pointers cross
    while True:
        # Increment the left pointer until an element greater than the pivot is found
        i += 1
        while A[i] < pivot:
            i += 1
        # Decrement the right pointer until an element less than the pivot is found
        j -= 1
        while A[j] > pivot:
            j -= 1
        # If the pointers have crossed, the partition is complete
        if i >= j:
            return j
    # Swap the elements at the left and right pointers
    A[i], A[j] = A[j], A[i]

```

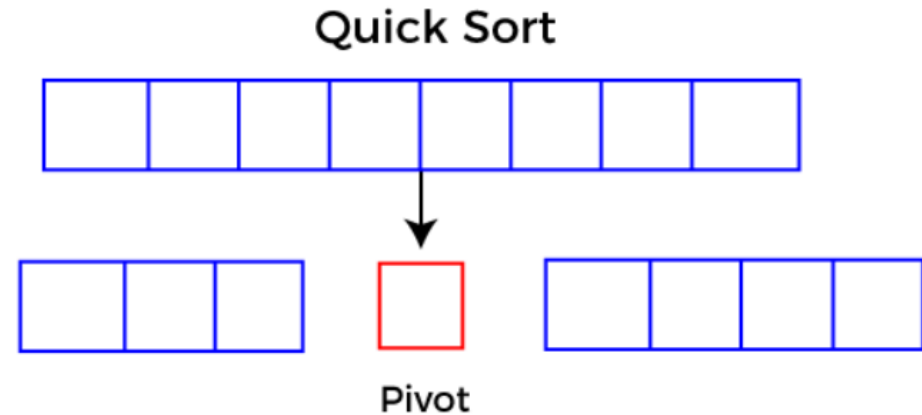
Test

```

A = [3, 8, 2, 5, 1, 4, 7, 6]
quicksort(A, 0, len(A) - 1)
print(A) # [1, 2, 3, 4, 5, 6, 7, 8]

```

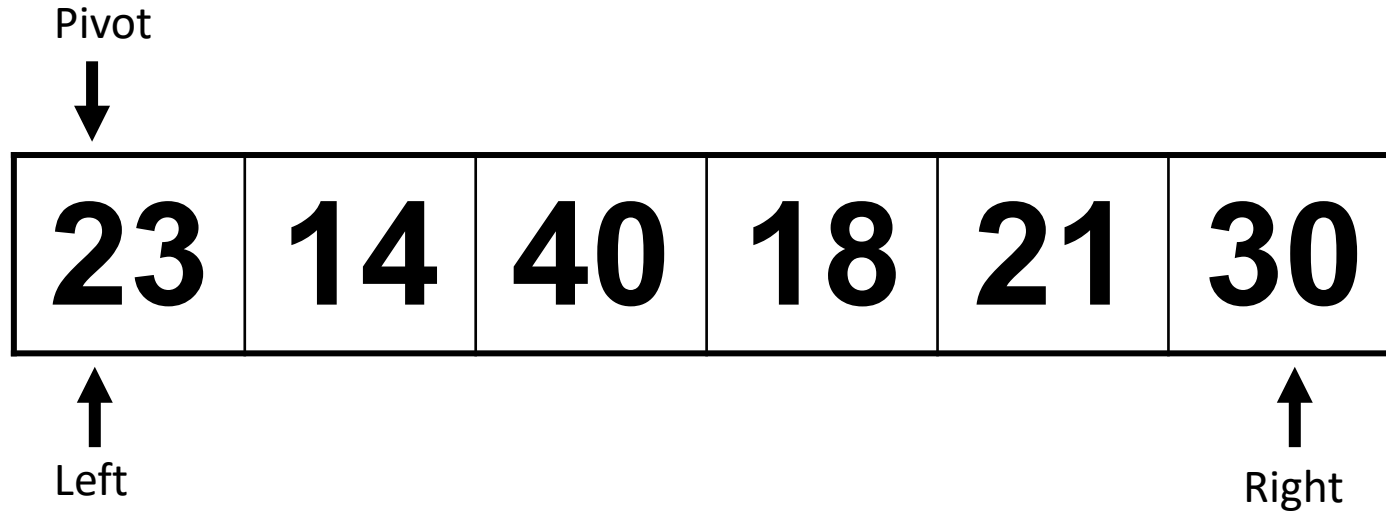
Quick Sort – Worked example



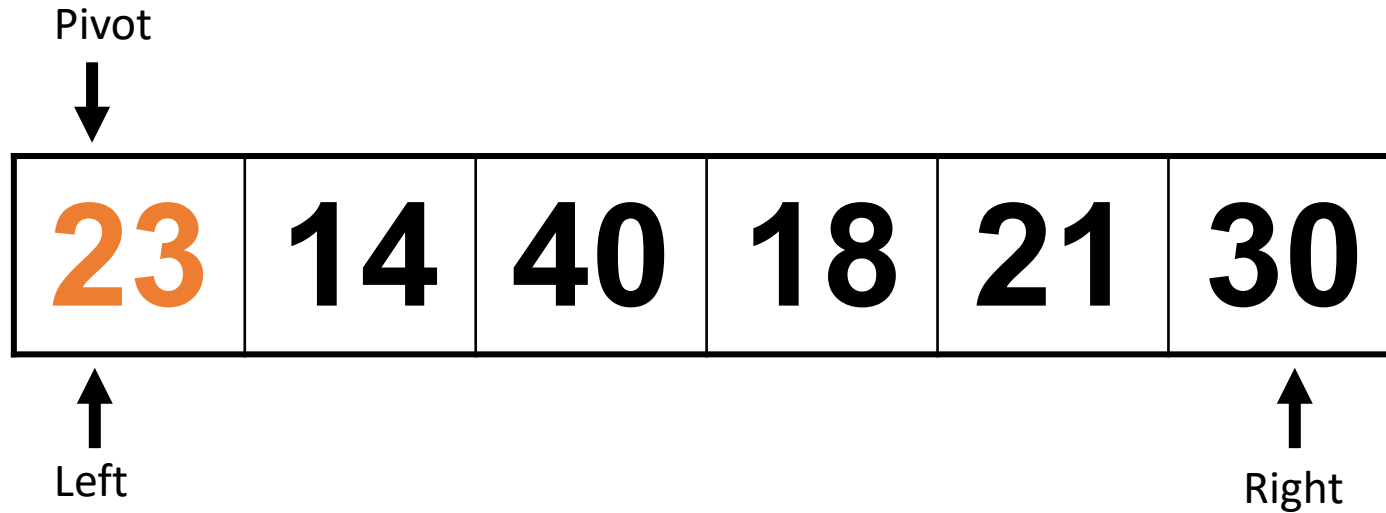
- Pivot can be random, i.e. select the random pivot from the given array.
- Pivot can either be the rightmost element of the leftmost element of the given array.
- Select median as the pivot element.

23	14	40	18	21	30
-----------	-----------	-----------	-----------	-----------	-----------

Quick Sort – Worked example

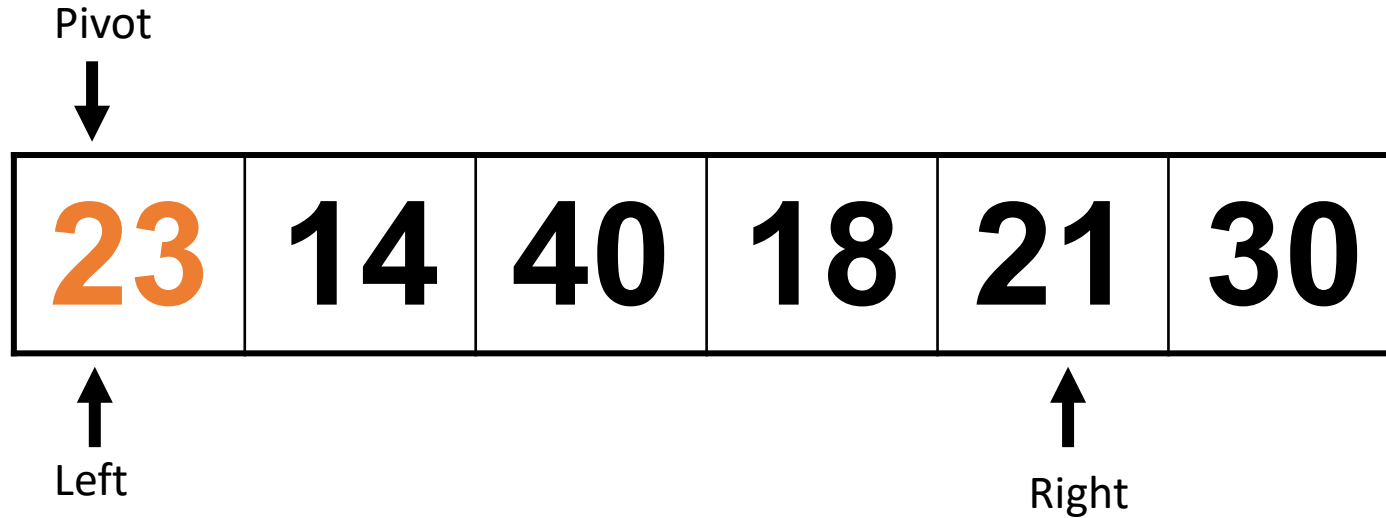


Quick Sort – Worked example



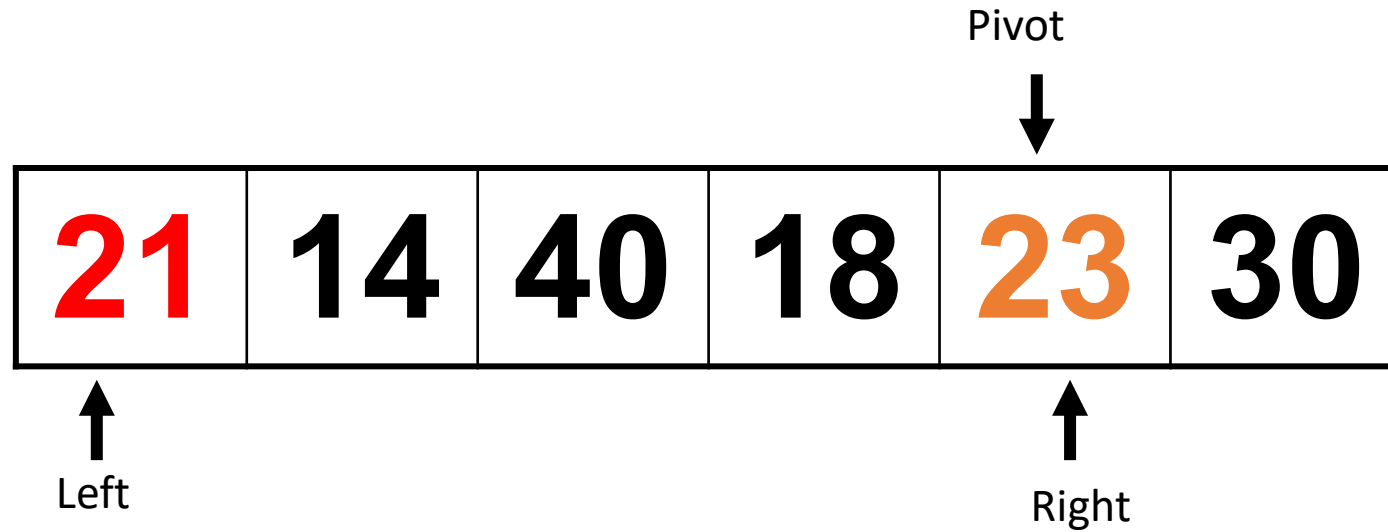
- The pivot is at left, so algorithm starts from right and move towards left.
- Now, $A[\text{pivot}] < A[\text{right}]$, so the right pointer moves forward one position towards left

Quick Sort – Worked example

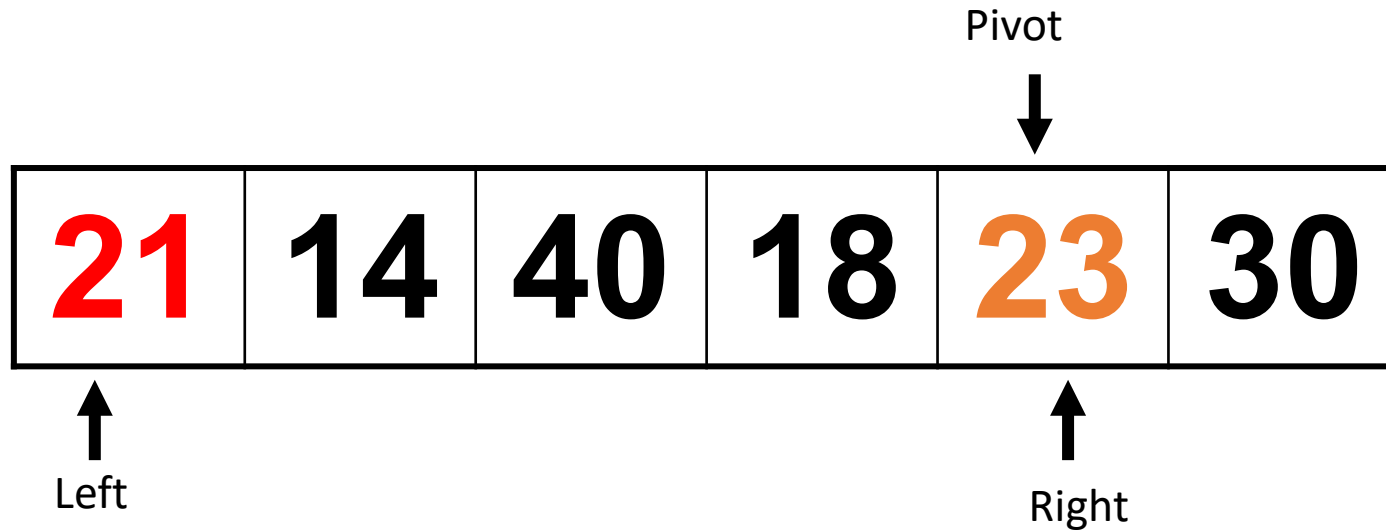


- Now, $A[\text{pivot}] > A[\text{right}]$, swap $A[\text{pivot}]$ with $A[\text{right}]$, and then pivot moves to **right**

Quick Sort – Worked example

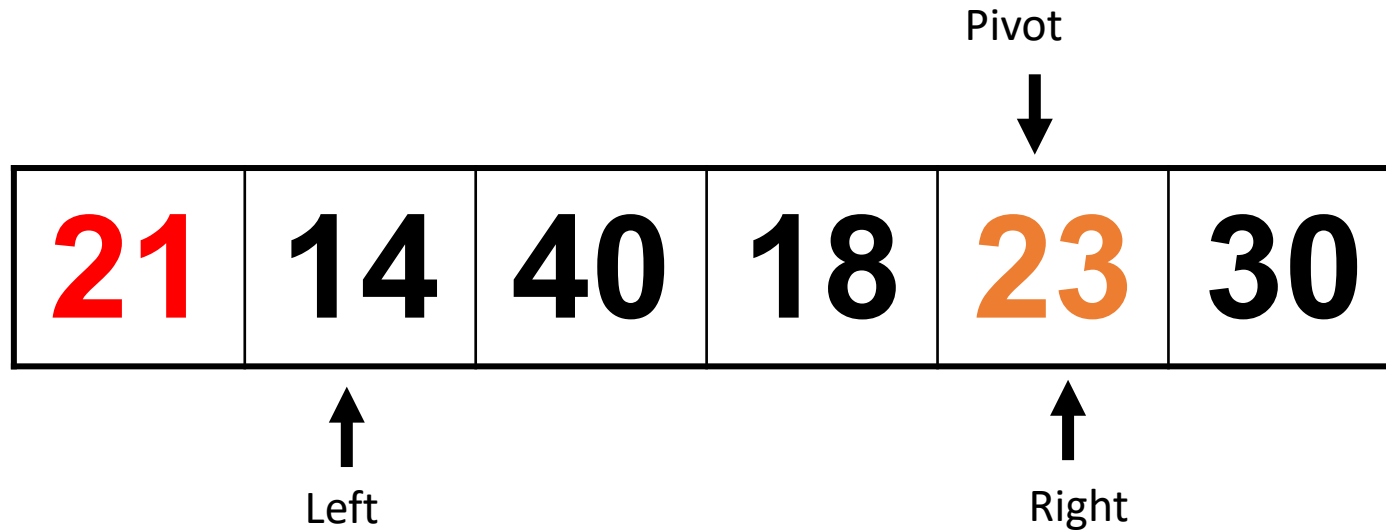


Quick Sort – Worked example



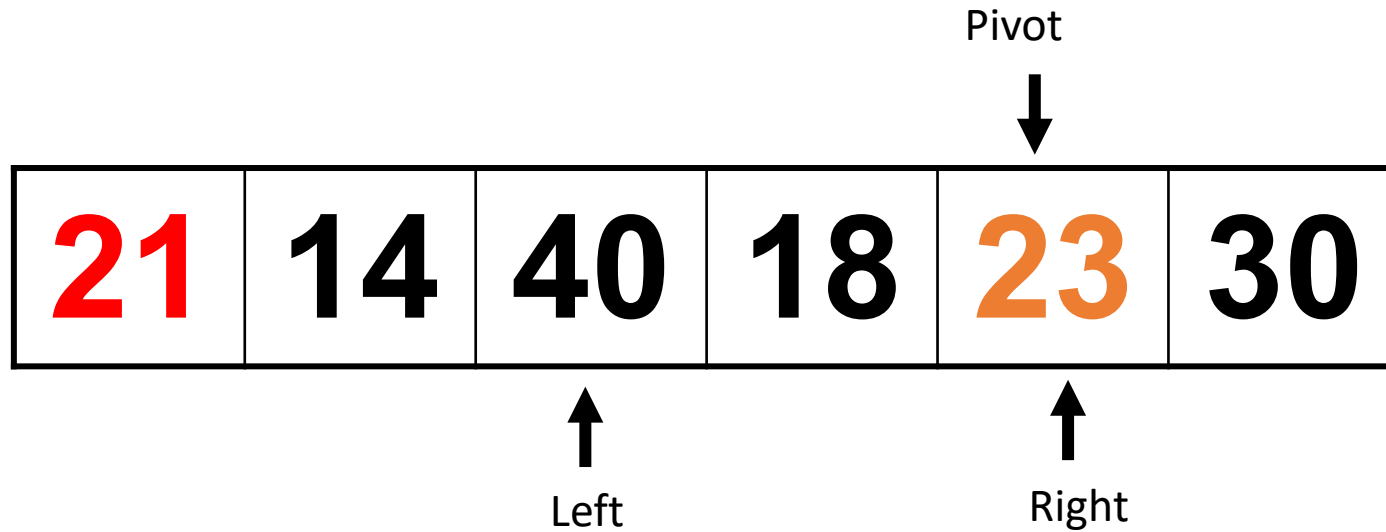
As pivot is at right, so algorithm starts from left and moves to right.

Quick Sort – Worked example

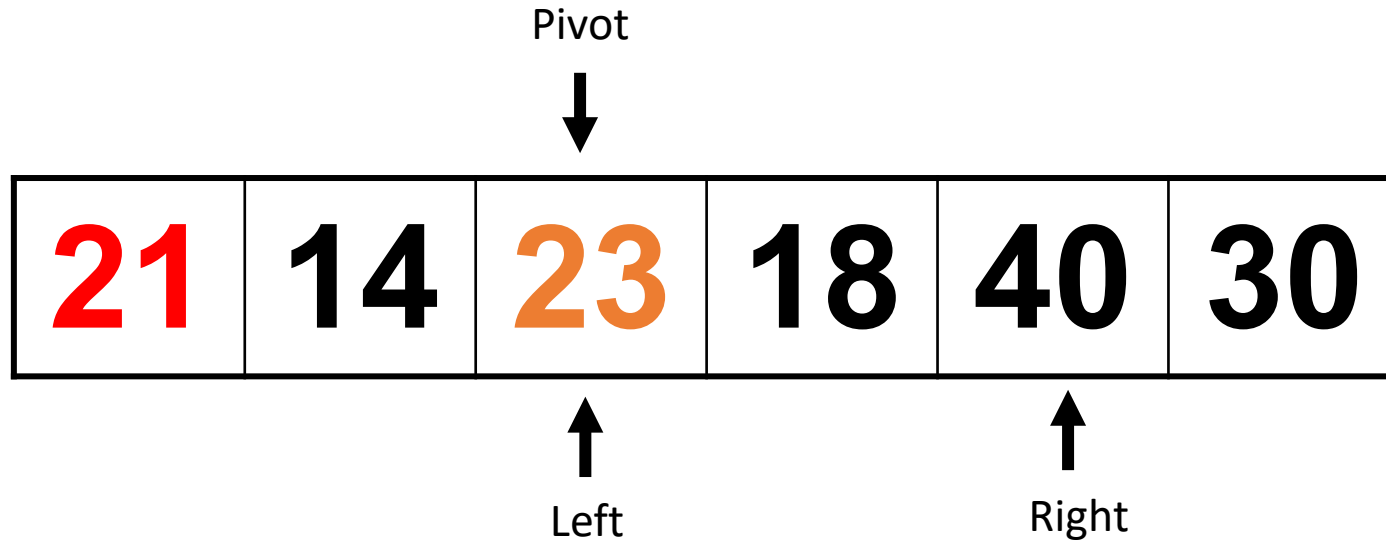


As $A[\text{pivot}] > A[\text{left}]$, so the left pointer moves one position to right

Quick Sort – Worked example



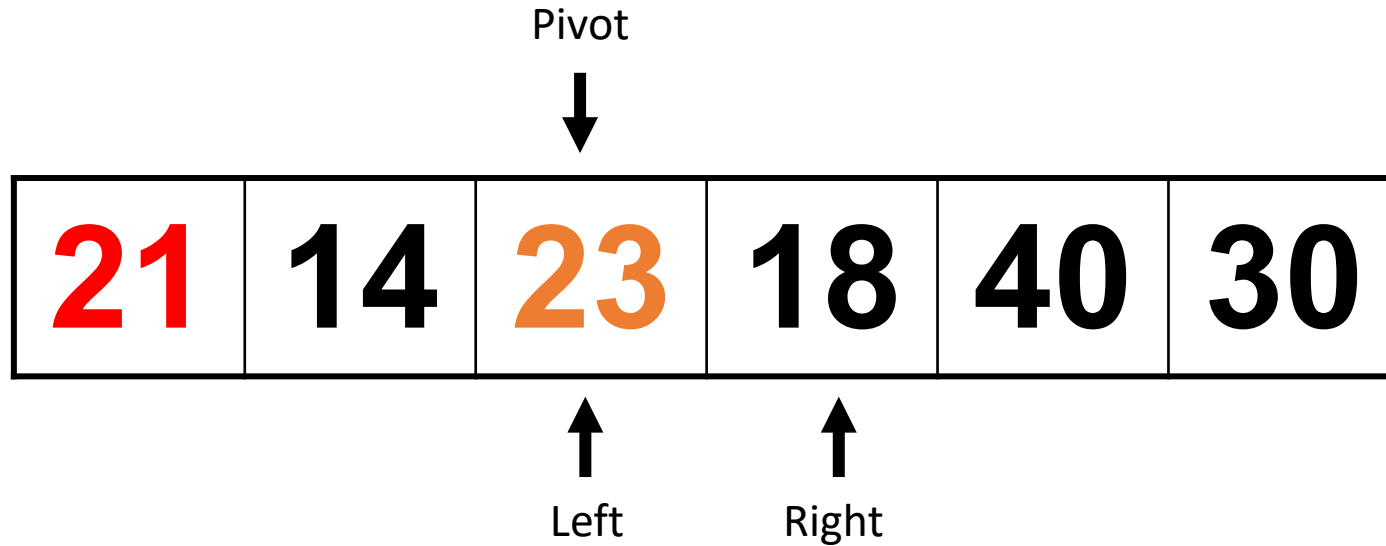
Quick Sort – Worked example



As $A[\text{pivot}] < A[\text{left}]$, so swap and pivot is at left

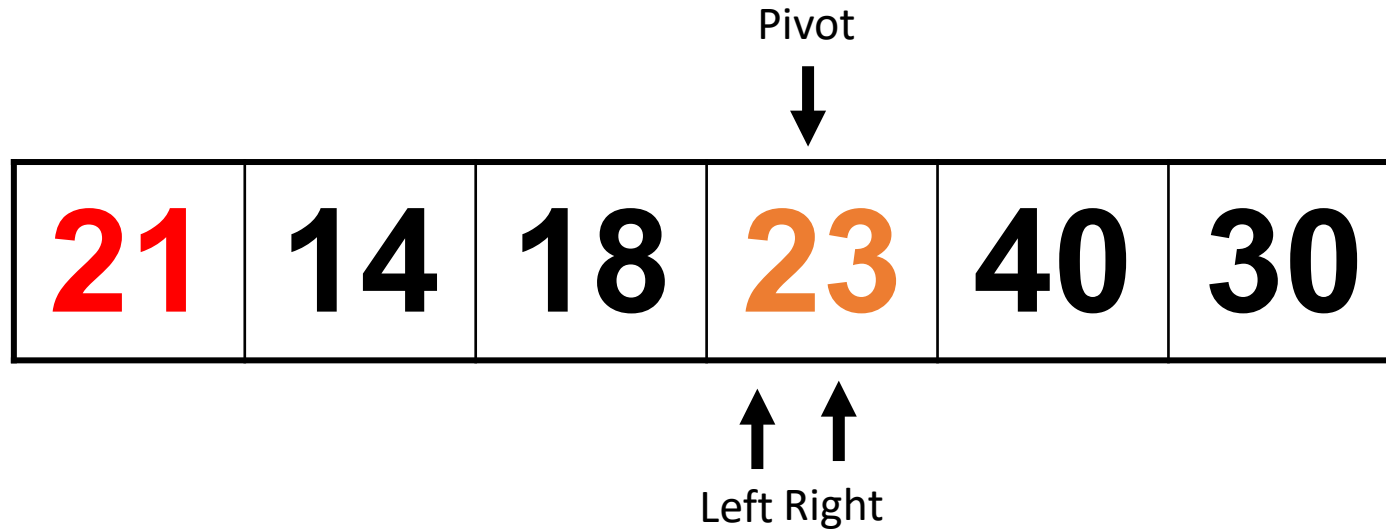
Since, pivot is at left, so algorithm starts from right, and move to left.

Quick Sort – Worked example



As $A[\text{pivot}] < A[\text{right}]$, right moves one position to left

Quick Sort – Worked example



As $A[\text{pivot}] > A[\text{left}]$, left moves one position to right

```

def quicksort(A, low, high):
    # Base case: if the subarray has 0 or 1 element, it is already sorted
    if low < high:
        # Partition the subarray and get the partition point
        p = hoare_partition(A, low, high)
        # Recursively sort the left subarray
        quicksort(A, low, p)
        # Recursively sort the right subarray
        quicksort(A, p + 1, high)

def hoare_partition(A, low, high):
    # Choose the first element as the pivot
    pivot = A[low]
    # Initialize the left pointer just before the first element
    i = low - 1
    # Initialize the right pointer just after the last element
    j = high + 1
    # Continue until the pointers cross
    while True:
        # Increment the left pointer until an element greater than the pivot is found
        i += 1
        while A[i] < pivot:
            i += 1
        # Decrement the right pointer until an element less than the pivot is found
        j -= 1
        while A[j] > pivot:
            j -= 1
        # If the pointers have crossed, the partition is complete
        if i >= j:
            return j
    # Swap the elements at the left and right pointers
    A[i], A[j] = A[j], A[i]

```

Test

```

A = [3, 8, 2, 5, 1, 4, 7, 6]
quicksort(A, 0, len(A) - 1)
print(A) # [1, 2, 3, 4, 5, 6, 7, 8]

```

Quick sort – Complexity

- **$O(n)$ operations** are required to find the location of an element that splits the array into two parts, .
 - every element in the array is compared to the partitioning element.
- After the division, each section is examined separately.
- If the array is split approximately in half (**which is not usually!**), then there will be $\log_2 n$ splits.
- Therefore, total comparisons required are
$$f(n) = n \times \log_2 n = O(n \log_2 n).$$

Quick sort – Worst Case Complexity

- Quick Sort is sensitive to the order of input data.
- It gives the worst performance when elements are already in the ascending order.
 - It then divides the array into sections of **1 and (n-1)** elements in each call.
 - Hence, **(n-1)** divisions in all.
- The total comparisons required are
$$f(n) = n \times (n-1) = O(n^2).$$
- Making it worst than merge sort and heap sort in terms of worst-case complexity.
- It is not a **stable sort** i.e., the order of equal elements may not be preserved.

Quick sort – Worst Case Complexity

- Quick sort is called quick because it has a much faster **average case** performance compared to other sorting algorithms like merge sort.
- While it does have a worst-case time complexity of $O(n^2)$ if the pivot is chosen poorly, in practice the pivot is often chosen randomly which results in a much faster average case performance of $O(n \log n)$.
- Additionally, quick sort has a smaller constant factor than merge sort, which means it can be faster in practice even when the input is already sorted or nearly sorted.