Algorithms and Data Structures

6th Lecture: Sort II

Yutaka Watanobe, Jie Huang, Yan Pei, Wenxi Chen,
Qiangfu Zhao, Deepika Saxena, Doan Hoang Le

University of Aizu

Last Updated: 2024/01/01

Outline

- Quick Sort
- Counting Sort

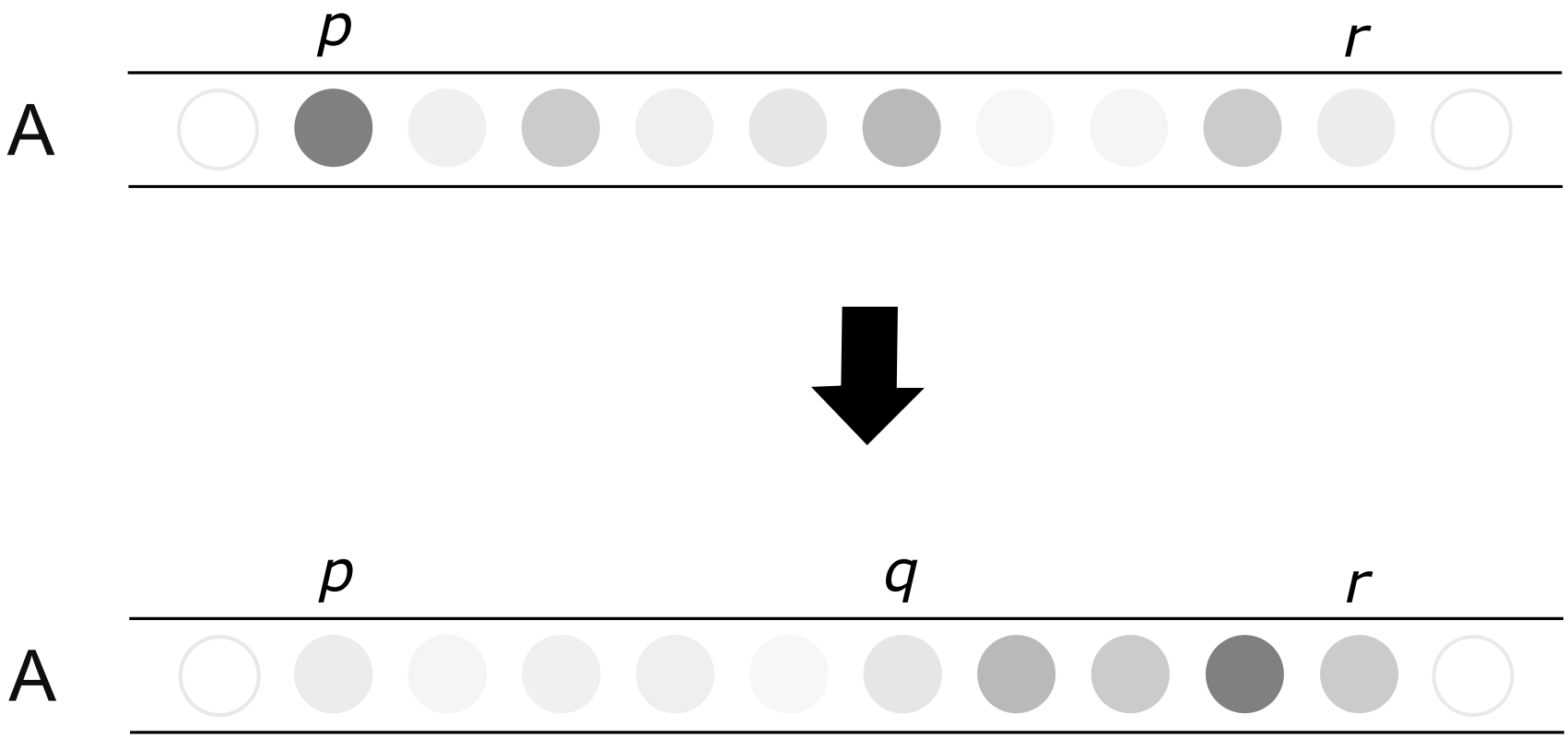
Divide and Conquer for Quick Sort

- Quick Sort is based on the divide and conquer paradigm. Here is the three-step divide and conquer process for sorting a subarray $A[p..r]$.
  1. **[Divide]** Partition (rearrange) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ such that each element of $A[p..q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1..r]$. Compute the index q as part of this partition procedure.
  2. **[Conquer]** Sort the two subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to Quick Sort.
  3. **[Combine]** Since the subarrays are sorted in place, no work is needed to combine them: the entire array $A[p..r]$ is now sorted.

Quick Sort

```
quickSort(A, p, r)
   if p < r
       q = partition(A, p, r)
       quickSort(A, p, q-1)
       quickSort(A, q+1, r)
```

Partition

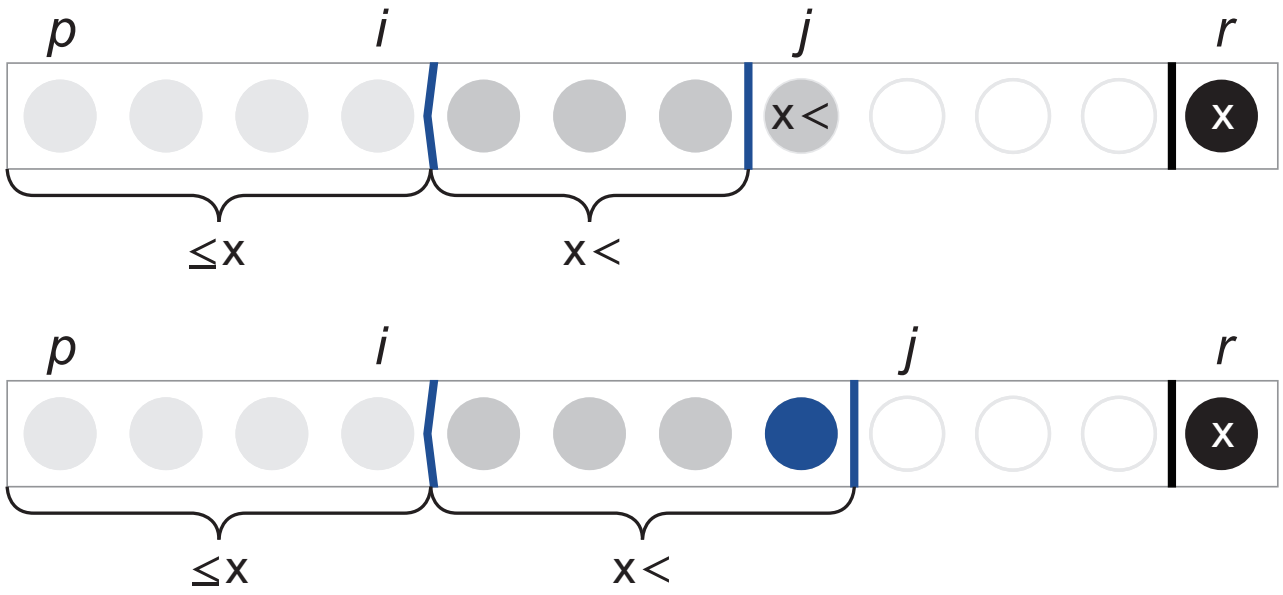


Partition Algorithm

```
partition(A, p, r)
    x = A[r]
    i = p-1
    for j = p to r-1
        if A[j] <= x
            i = i+1
            exchange A[i] and A[j]
    exchange A[i+1] and A[r]
    return i+1
```

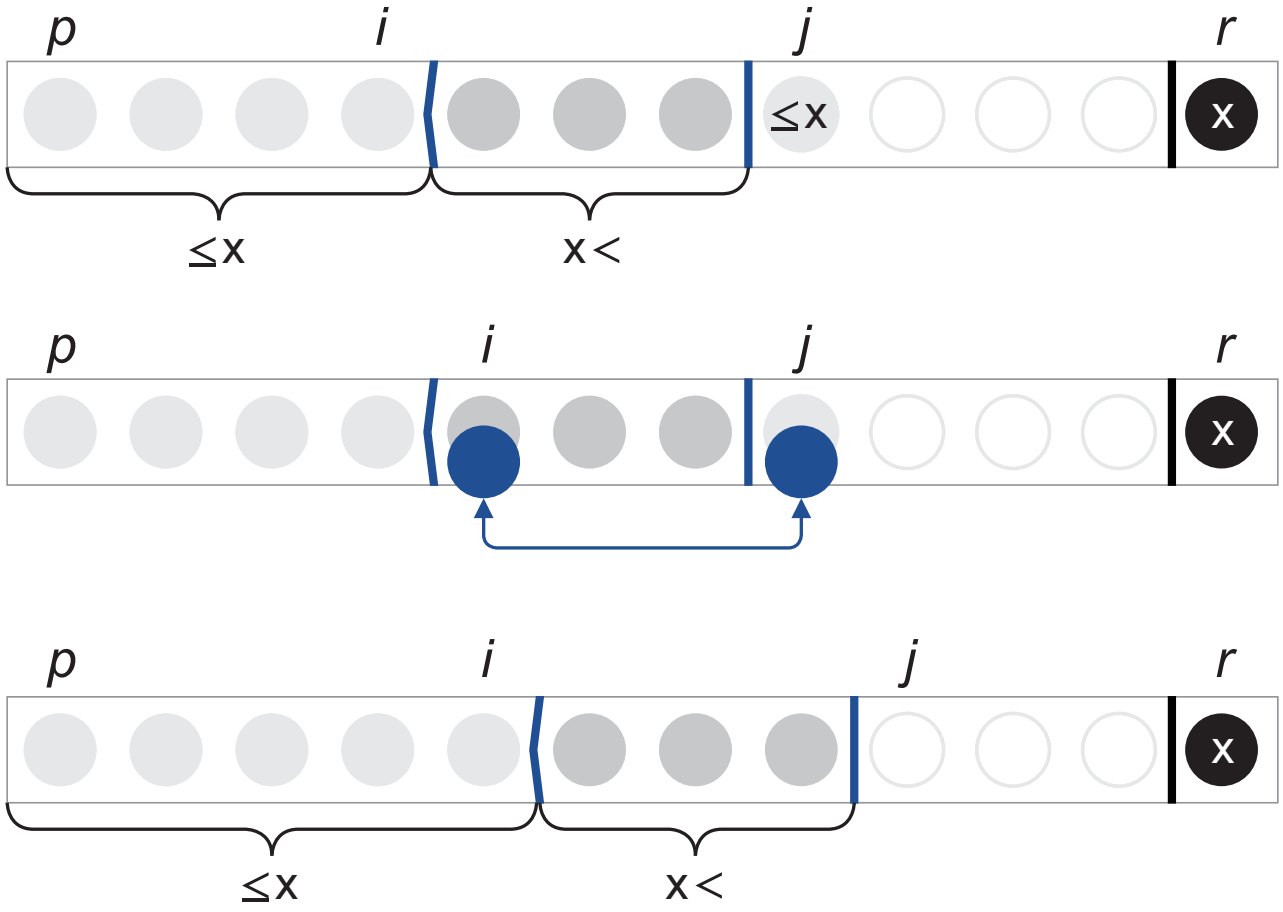
- To sort an entire array A with n elements, we should call `quickSort(A, 0, n-1)`

Partition (Case 1)



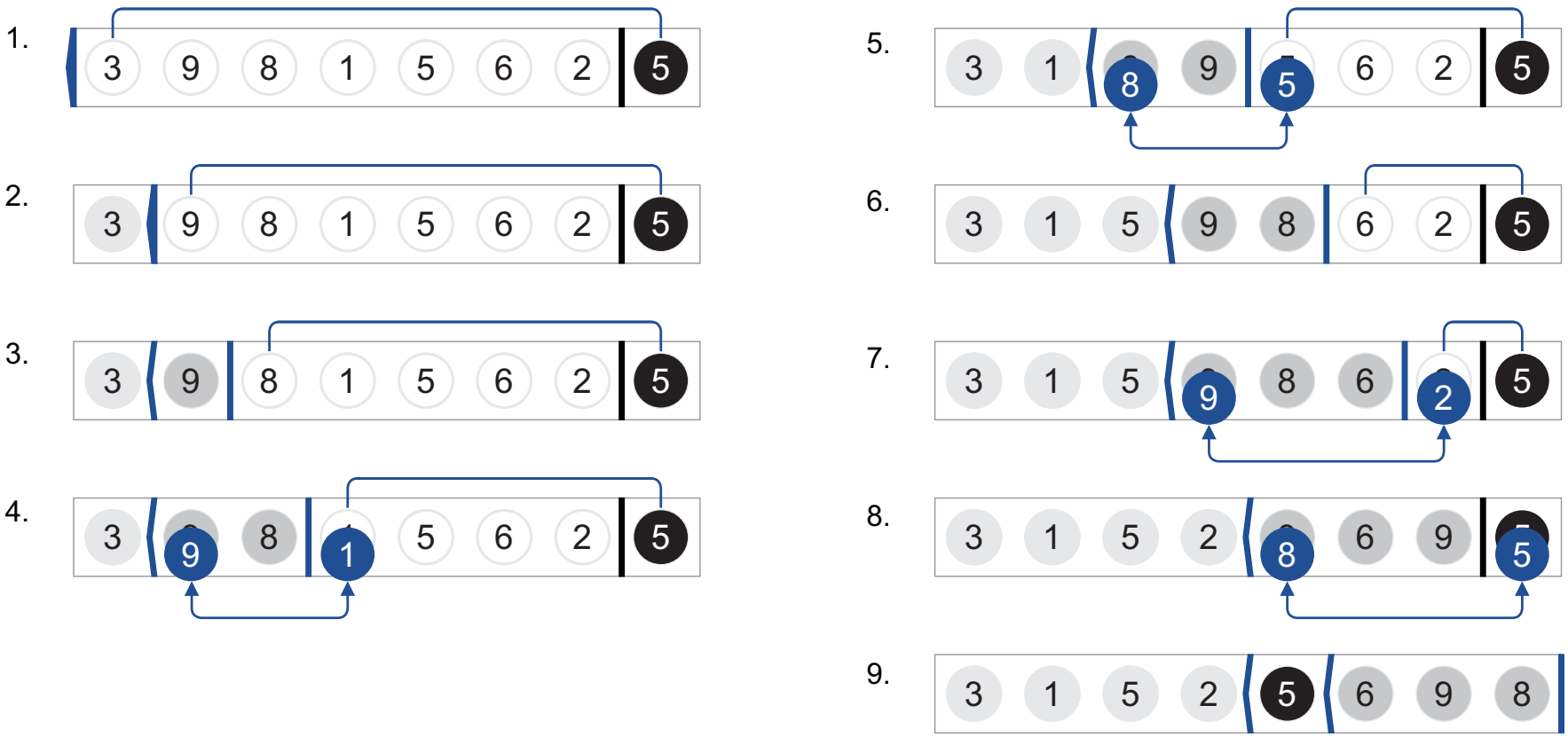
If $x < A[j]$, the only action is to increment j.

Partition (Case 2)



If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented.

Partition


1.
3 9 8 1 5 6 2 5
2.
3 9 8 1 5 6 2 5
3.
3 9 8 1 5 6 2 5
4.
3 9 8 1 5 6 2 5
5.
3 1 8 9 5 6 2 5
6.
3 1 5 9 8 6 2 5
7.
3 1 5 9 8 6 2 5
8.
3 1 5 2 8 6 9 5
9.
3 1 5 2 5 6 9 8

Partition (1)

1.

Partition (2)

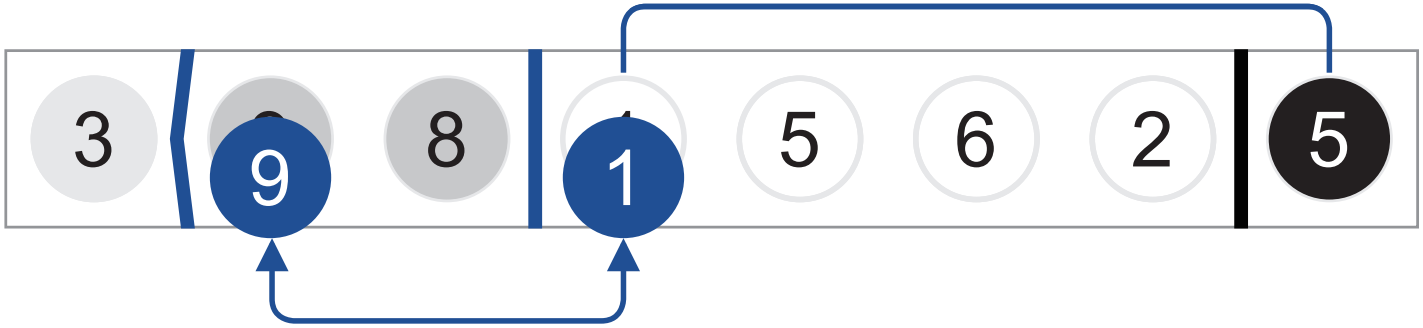
2.

Partition (3)

3.

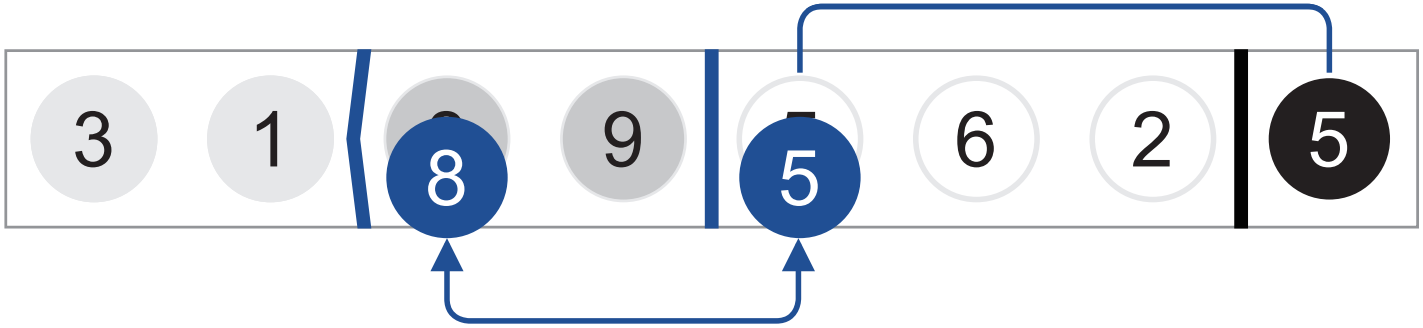
Partition (4)

4.



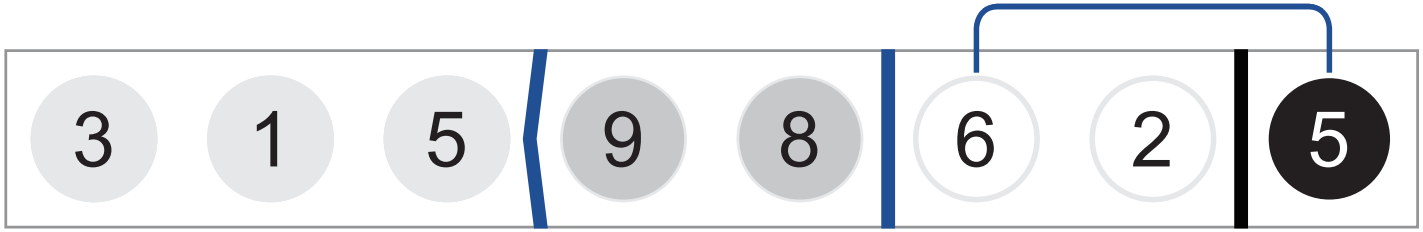
Partition (5)

5.



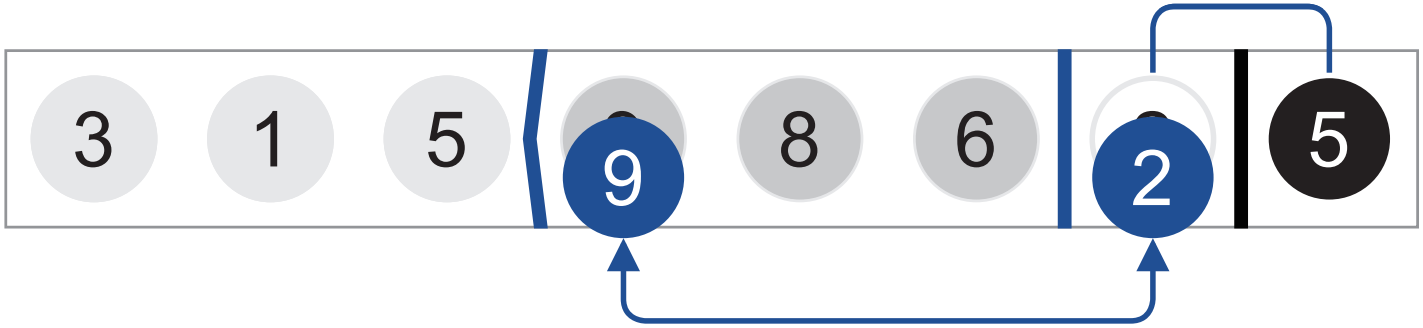
Partition (6)

6.



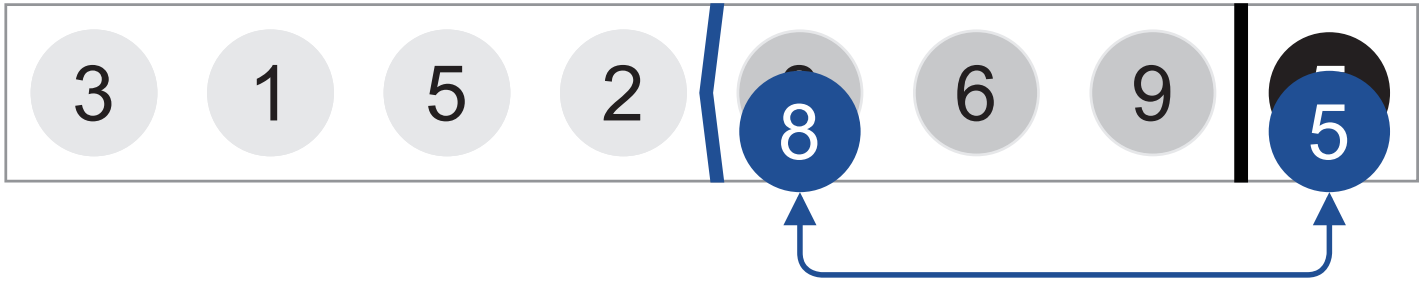
Partition (7)

7.



Partition (8)

8.



Partition (9)

9.

Quick Sort

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
13 19 9 5 12 8 7 4 21 2 5 3 14 6 11
1 Partition(A, 0, 14)
9 5 8 7 4 2 5 3 6 11 19 12 14 21 13
2 Partition(A, 0, 8)
7 Partition(A, 10, 14)
5 4 2 5 3 6 7 9 8
12 13 14 21 19
3 Partition(A, 0, 4)
6 Partition(A, 6, 8)
8 Partition(A, 12, 14)
2 3 5 5 4
7 8 9
14 19 21
4 Partition(A, 2, 4)
4 5 5
5 Partition(A, 3, 4)
5 5

Quick Sort (1)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	19	9	5	12	8	7	4	21	2	5	3	14	6	11	
1						Partition(A, 0, 14)									
9	5	8	7	4	2	5	3	6	11	19	12	14	21	13	

Quick Sort (2)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	19	9	5	12	8	7	4	21	2	5	3	14	6	11	

1 `Partition(A, 0, 14)`

9	5	8	7	4	2	5	3	6	11	19	12	14	21	13

2 `Partition(A, 0, 8)`

5	4	2	5	3	6	7	9	8

Quick Sort (3)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	19	9	5	12	8	7	4	21	2	5	3	14	6	11	

1 `Partition(A, 0, 14)`

9	5	8	7	4	2	5	3	6	11	19	12	14	21	13

2 `Partition(A, 0, 8)`

5	4	2	5	3	6	7	9	8

3 `Partition(A, 0, 4)`

2	3	5	5	4

Quick Sort (4)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	19	9	5	12	8	7	4	21	2	5	3	14	6	11	

1 `Partition(A, 0, 14)`

9	5	8	7	4	2	5	3	6	11	19	12	14	21	13

2 `Partition(A, 0, 8)`

5	4	2	5	3	6	7	9	8

3 `Partition(A, 0, 4)`

2	3	5	5	4

4 `Partition(A, 2, 4)`

4	5	5

Quick Sort (5)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
13 19 9 5 12 8 7 4 21 2 5 3 14 6 11
1
Partition(A, 0, 14)
9 5 8 7 4 2 5 3 6 11 19 12 14 21 13
2
Partition(A, 0, 8)
5 4 2 5 3 6 7 9 8
3
Partition(A, 0, 4)
2 3 5 5 4
4
Partition(A, 2, 4)
4 5 5
5
Partition(A, 3, 4)
5 5

Quick Sort (6)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
13 19 9 5 12 8 7 4 21 2 5 3 14 6 11
1
Partition(A, 0, 14)
9 5 8 7 4 2 5 3 6 11 19 12 14 21 13
2
Partition(A, 0, 8)
5 4 2 5 3 6 7 9 8
3
Partition(A, 0, 4)
6
Partition(A, 6, 8)
2 3 5 5 4
7 8 9
4
Partition(A, 2, 4)
4 5 5
5
Partition(A, 3, 4)
5 5

Quick Sort (7)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
13 19 9 5 12 8 7 4 21 2 5 3 14 6 11
1 Partition(A, 0, 14)
9 5 8 7 4 2 5 3 6 11 19 12 14 21 13
2 Partition(A, 0, 8)
7 Partition(A, 10, 14)
5 4 2 5 3 6 7 9 8
12 13 14 21 19
3 Partition(A, 0, 4)
6 Partition(A, 6, 8)
2 3 5 5 4
7 8 9
4 Partition(A, 2, 4)
4 5 5
5 Partition(A, 3, 4)
5 5

Quick Sort (8)

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
13 19 9 5 12 8 7 4 21 2 5 3 14 6 11
1 Partition(A, 0, 14)
9 5 8 7 4 2 5 3 6 11 19 12 14 21 13
2 Partition(A, 0, 8)
7 Partition(A, 10, 14)
5 4 2 5 3 6 7 9 8
12 13 14 21 19
3 Partition(A, 0, 4)
6 Partition(A, 6, 8)
8 Partition(A, 12, 14)
2 3 5 5 4
7 8 9
14 19 21
4 Partition(A, 2, 4)
4 5 5
5 Partition(A, 3, 4)
5 5

Performance and Stability of Quick Sort

- Worst-case partitioning: $O(n^2)$
- Best-case partitioning: $O(n \log n)$
- Balanced partitioning: $O(n \log n)$
- Quick Sort is not a stable sort algorithm because it can exchange two distant elements.

Sorting in Linear Time

- We have now introduced several algorithms that can sort n numbers in $O(n \log n)$ time.
- Merge Sort achieve this upper bound in the worst case, Quick Sort achieves it on average.
- There are several algorithms - that run in linear time.

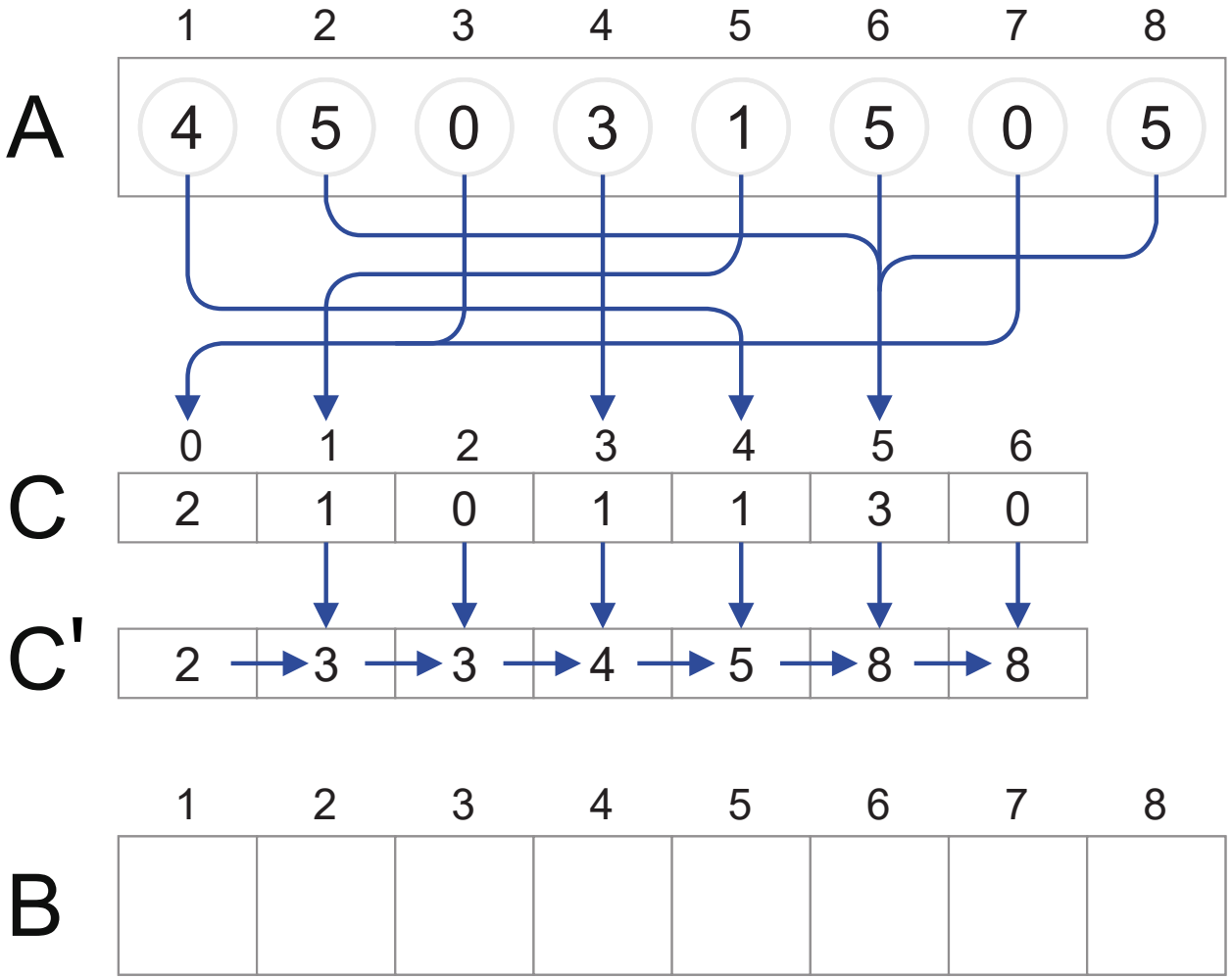
Counting Sort

- Counting Sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.
- The basic idea of counting sort is to determine, for each input element x, the number of elements less than or equal to x. This information can be used to place element x directly into its position in the output array.
- For example, if there are 17 elements less than or equal to x, then x belongs in output position 17.
- This scheme must be modified slightly to handle the situation in which several elements have the same value, since we don't want to put them all in the same position.

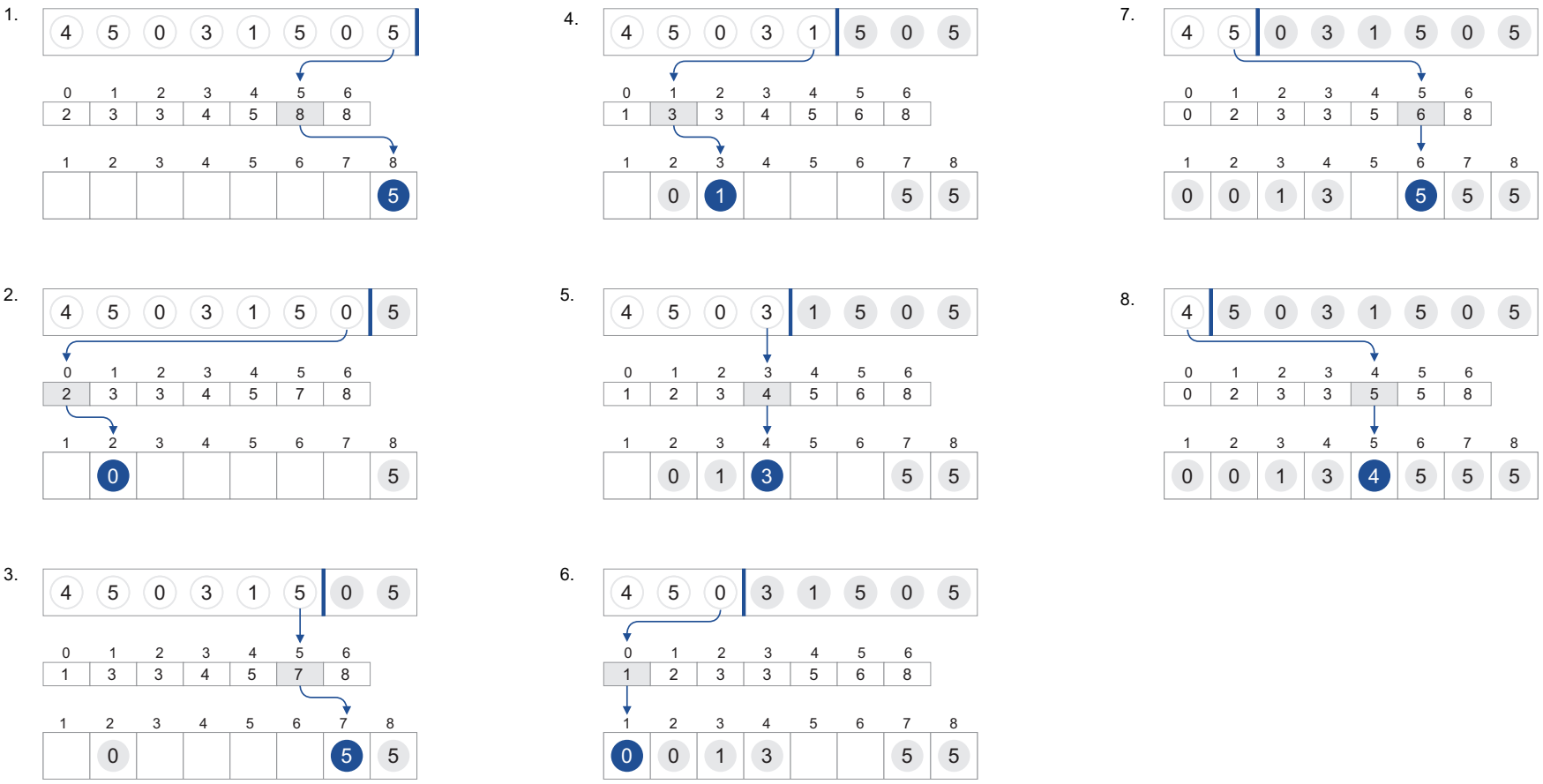
Counting Sort

```
countingSort(A, B, k)
    for i = 0 to k
        C[i] = 0
    for j = 1 to n
        C[A[j]] = C[A[j]]+1
    /* C[i] now contains the number of elements equal to i.*/
    for i = 1 to k
        C[i] = C[i] + C[i-1]
    /* C[i] now contains the number of elements
       less than or equal to i */
    for j = n downto 1
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]]-1
```

Counting Sort (Init)

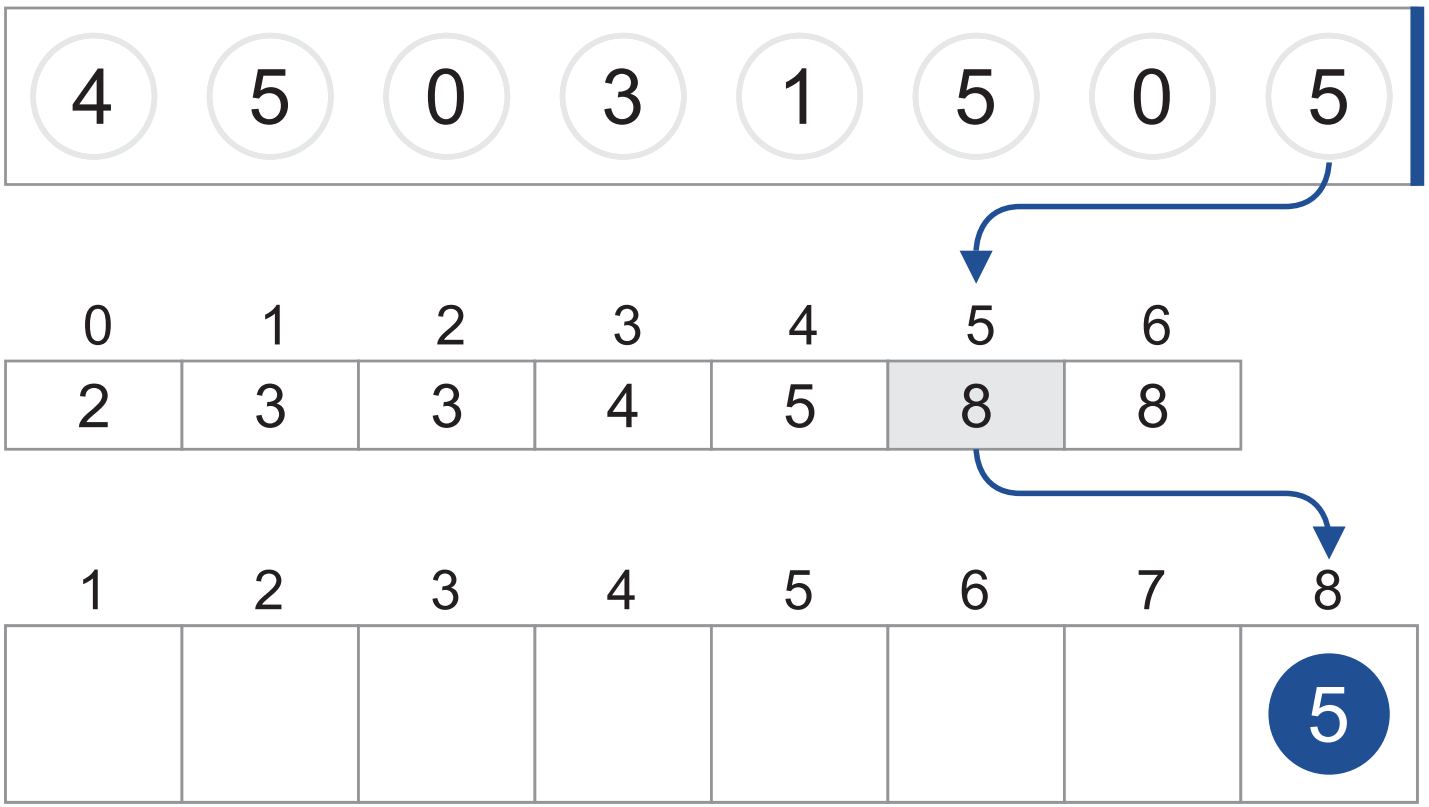


Counting Sort



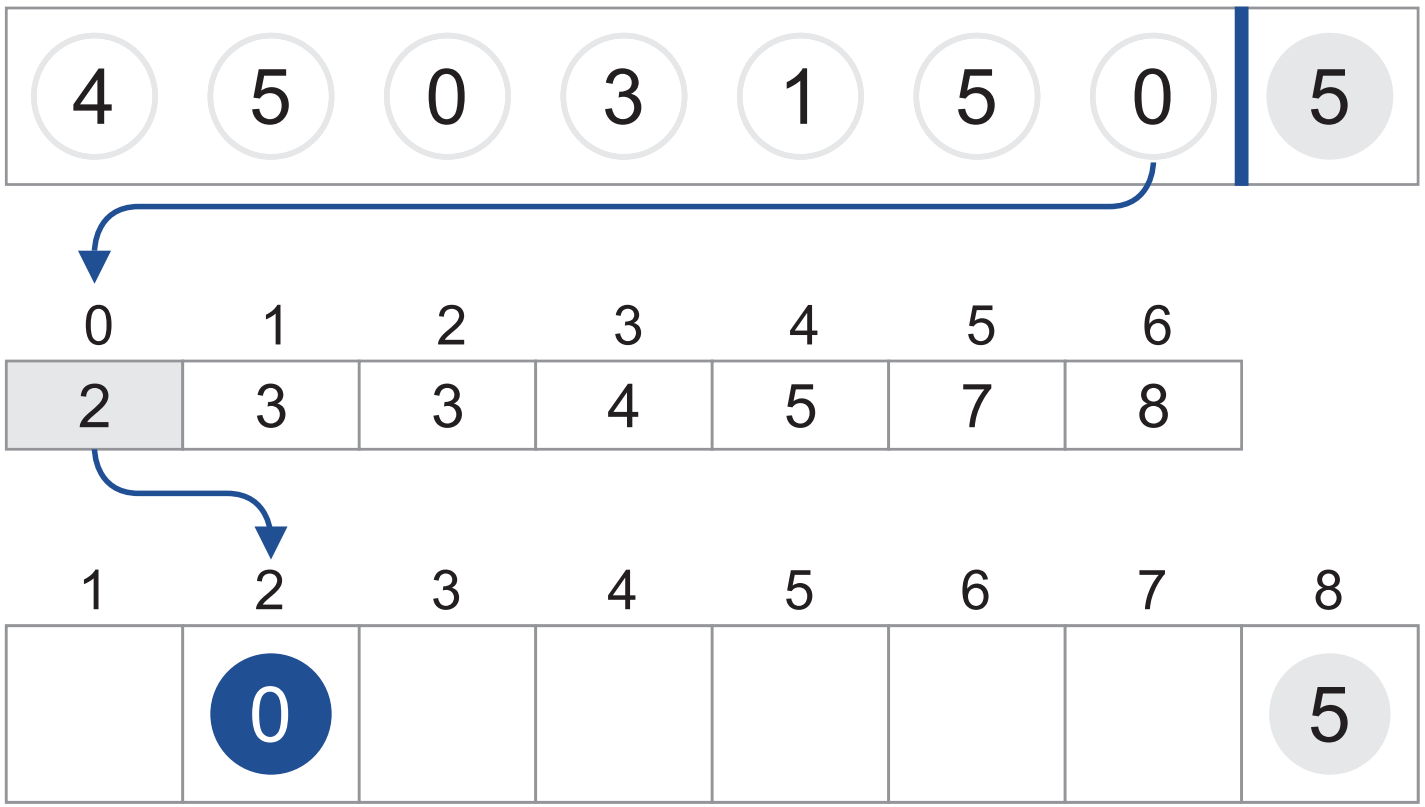
Counting Sort (1)

1.



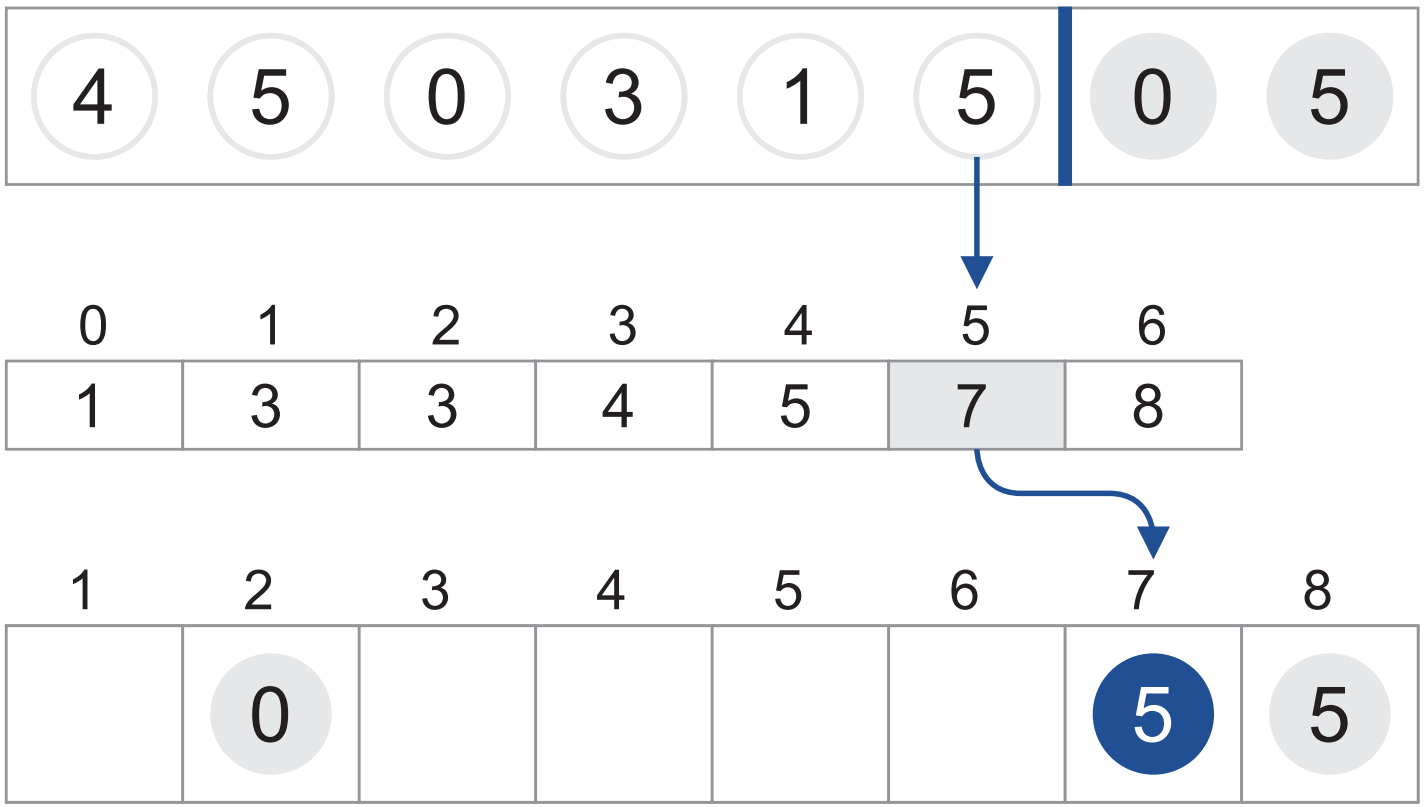
Counting Sort (2)

2.



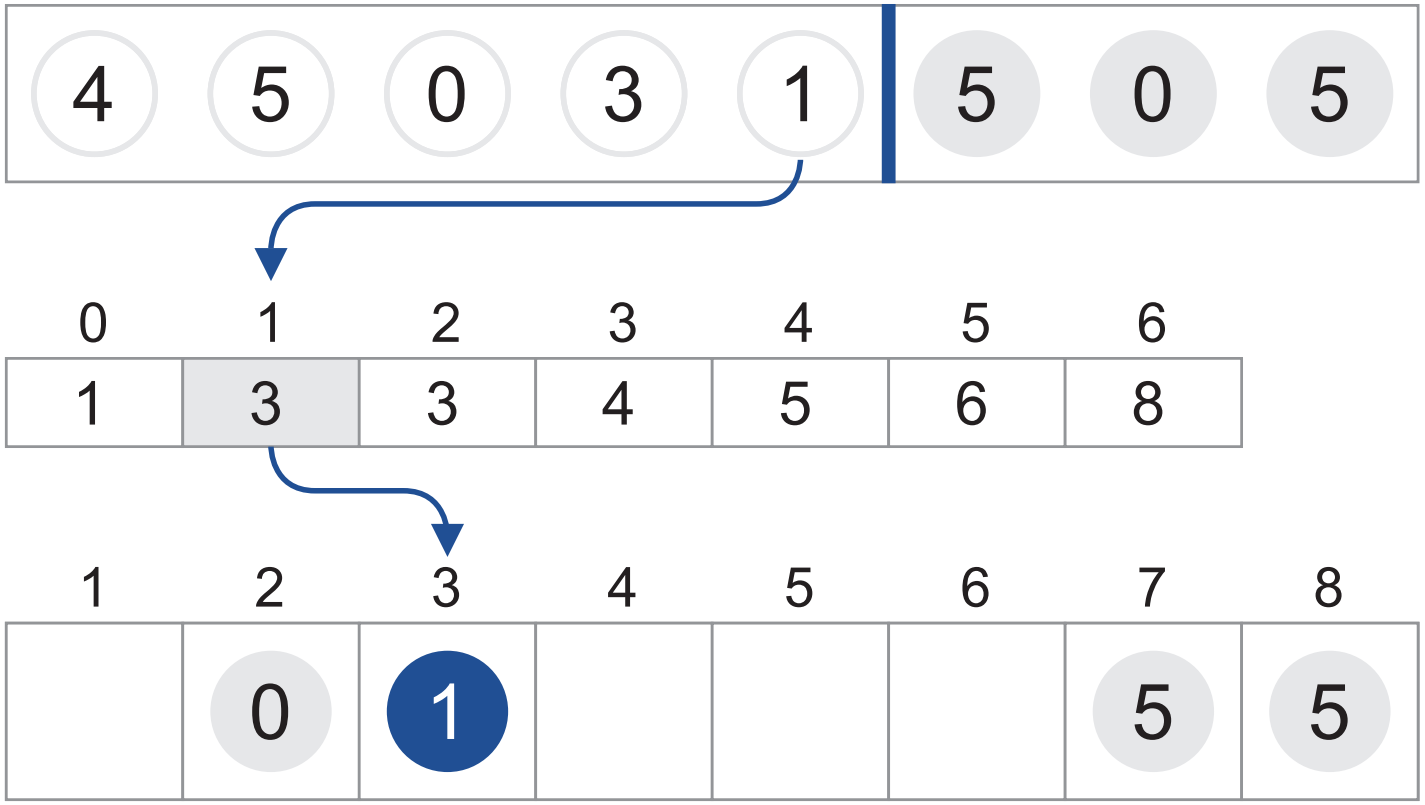
Counting Sort (3)

3.



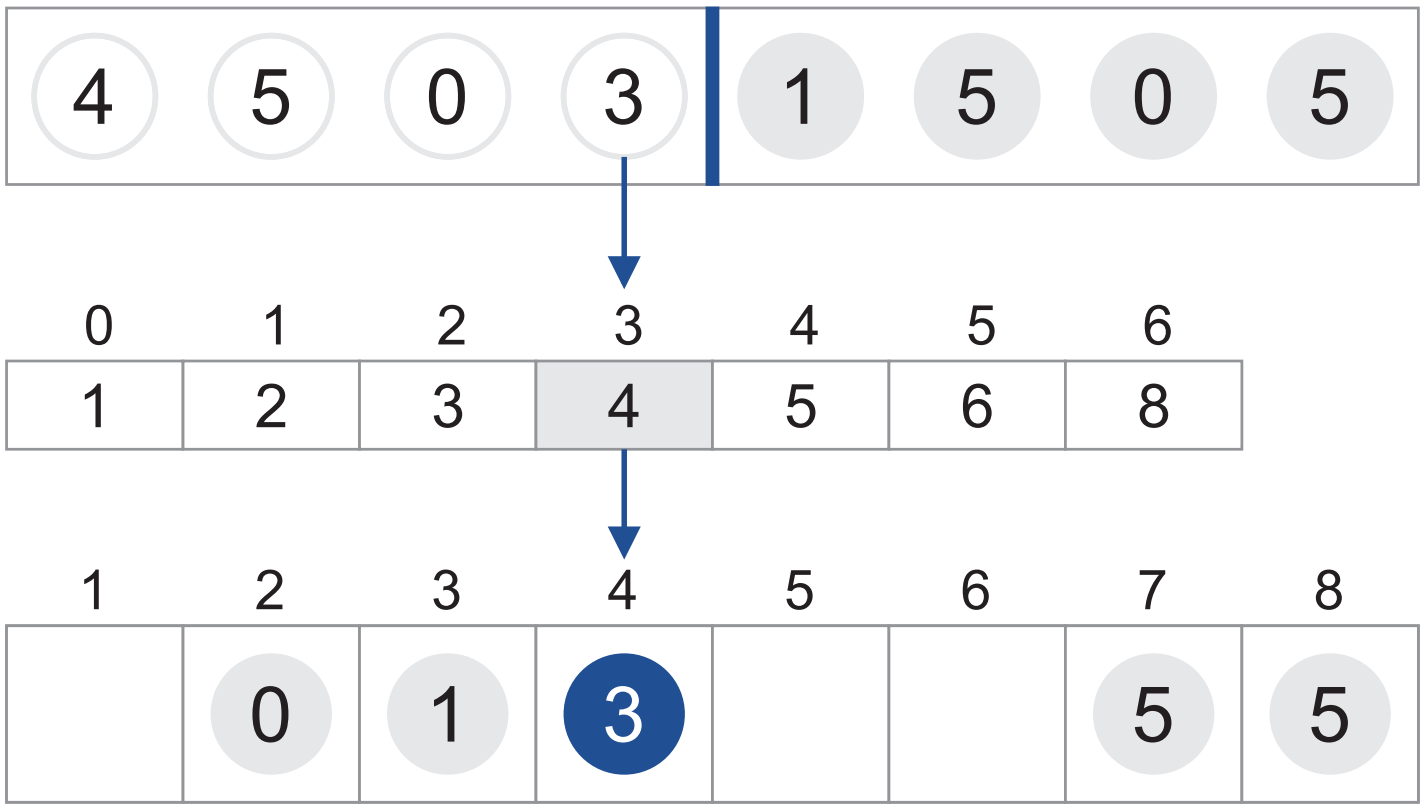
Counting Sort (4)

4.



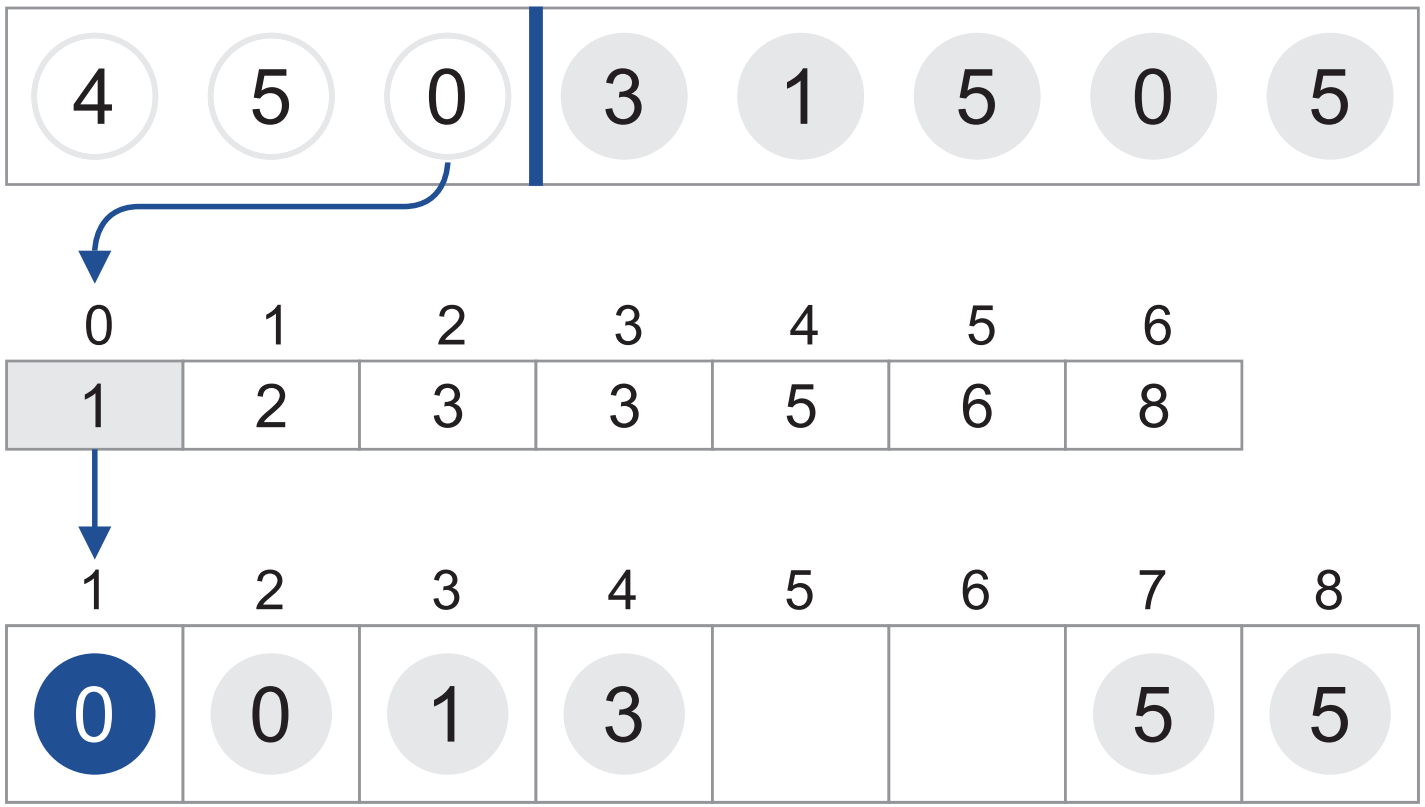
Counting Sort (5)

5.



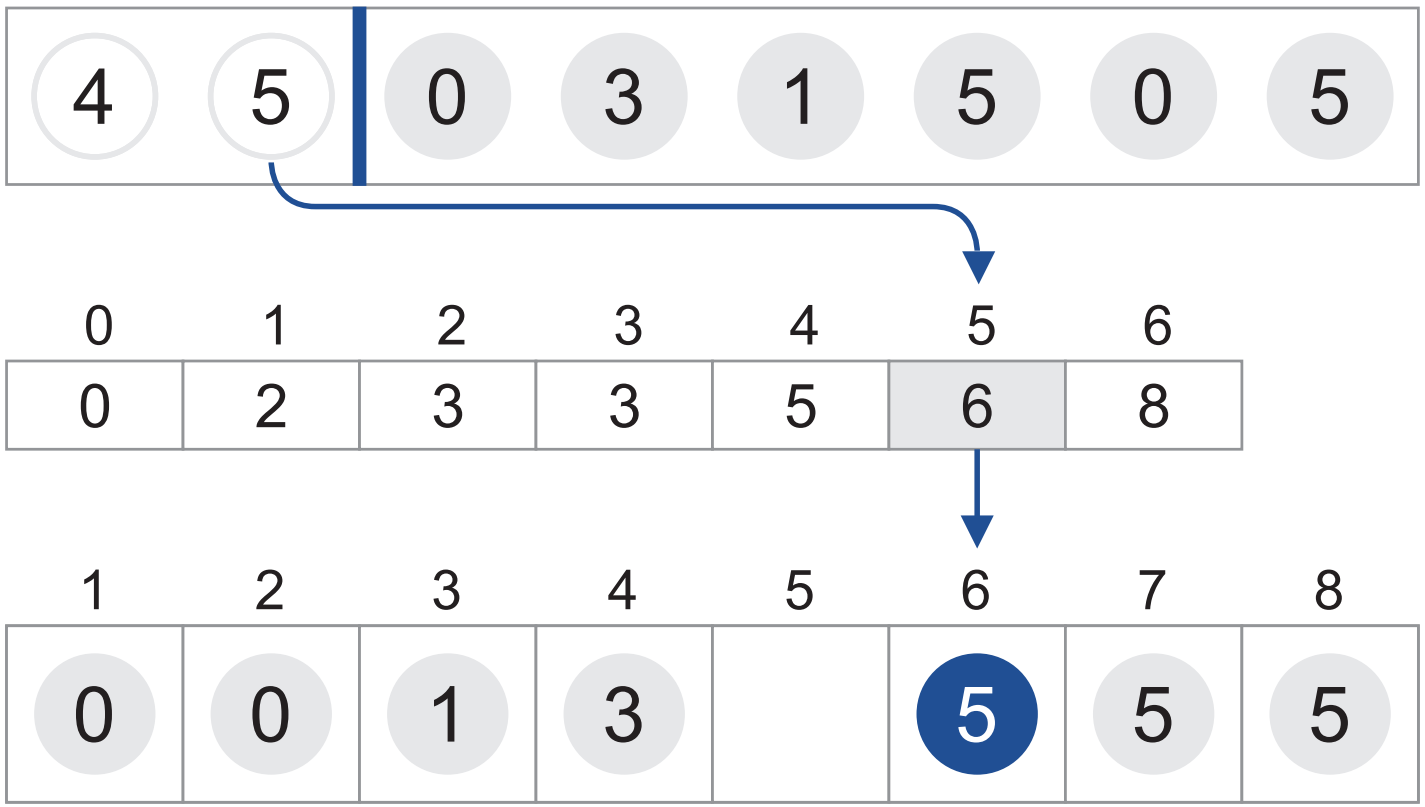
Counting Sort (6)

6.



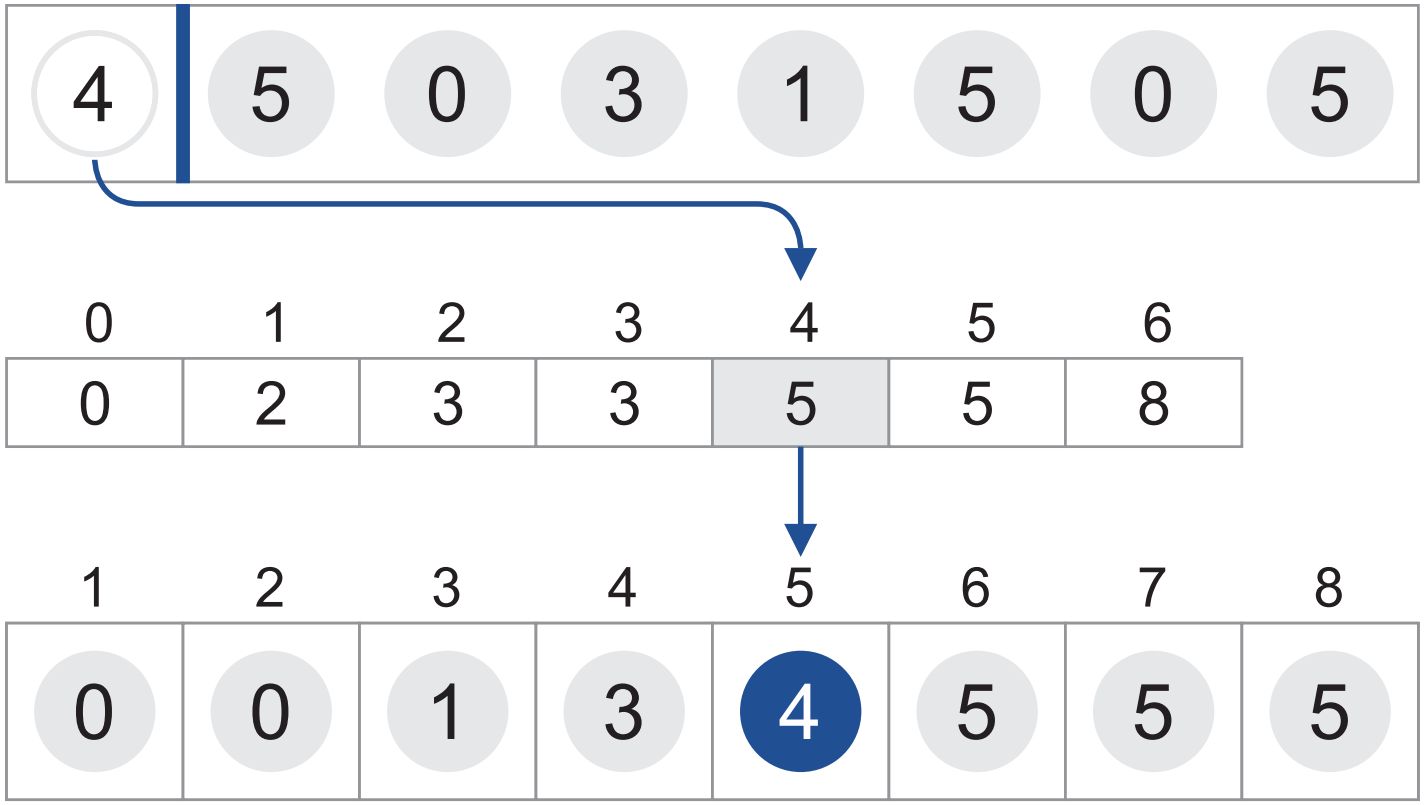
Counting Sort (7)

7.



Counting Sort (8)

8.



Performance and Stability of Counting Sort

- Worst-case: $O(n)$
- Memory space: $O(k)$
- Counting Sort is a stable sort algorithm.

Reference

1. Introduction to Algorithms (third edition), Thomas H.Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. The MIT Press, 2012.