Lecture 8

Linear Time Sorting

Md. Asif Bin Khaled

How Fast Can We Sort?

Up until now we have only encountered comparison based sorting algorithm which means it only uses comparisons to determine the relative order of elements.

The best worst-case running time that we've seen for comparison sorting is, **O(nlgn).** Is this the best we can do**?**

Decision trees can help us answer this question.

Lower Bounds for Sorting

In a comparison sort, we use only comparisons between elements to gain order information about an input sequence $\langle a_1, a_2, \ldots, a_n \rangle$. That is, given two elements a_i and a_j, we perform one of the tests $a_i < a_j$, $a_i \leq a_j$, $a_i = a_j$, $a_i \geq a_j$, or $a_i > a_j$ to determine their relative order. We may not inspect the values of the elements or gain order information about them in any other way.

Now, we assume without loss of generality that all the input elements are distinct. Given this assumption, comparisons of the form $a_i = a_j$ are useless,
so we can assume that no comparisons of this form are made.

Lower Bounds for Sorting Continued

We also note that the comparisons $a_i < a_j$, $a_i \leq a_j$, $a_i \geq a_j$, and $a_i > a_j$ are all equivalent in that they yield identical information about the relative order of a_i and a_j . We therefore assume that all comparisons have the form $a_i \leq a_j$.

The Decision Tree Model

– – –

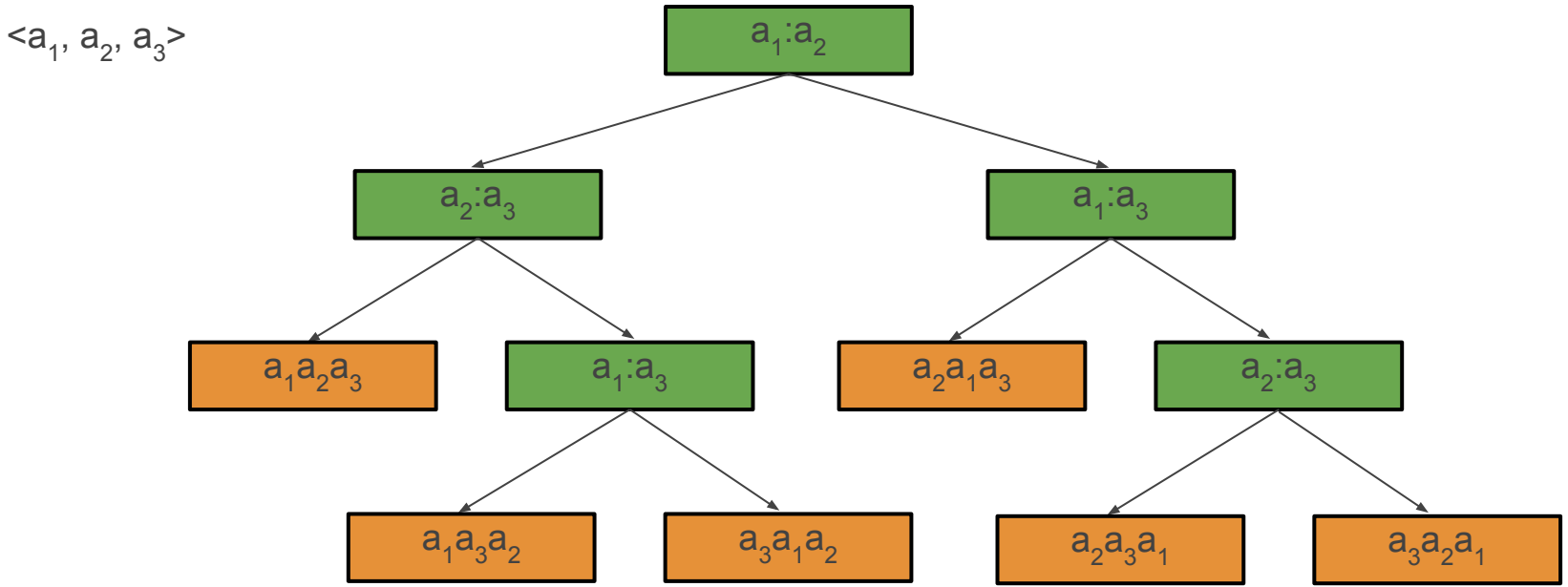
Sort $\langle a_1, a_2, \ldots, a_n \rangle$

Each internal node is labeled i:j for $i, j \in \{1, 2, \ldots, n\}$.

The left subtree shows subsequent comparisons if $a_i \leq a_j$.

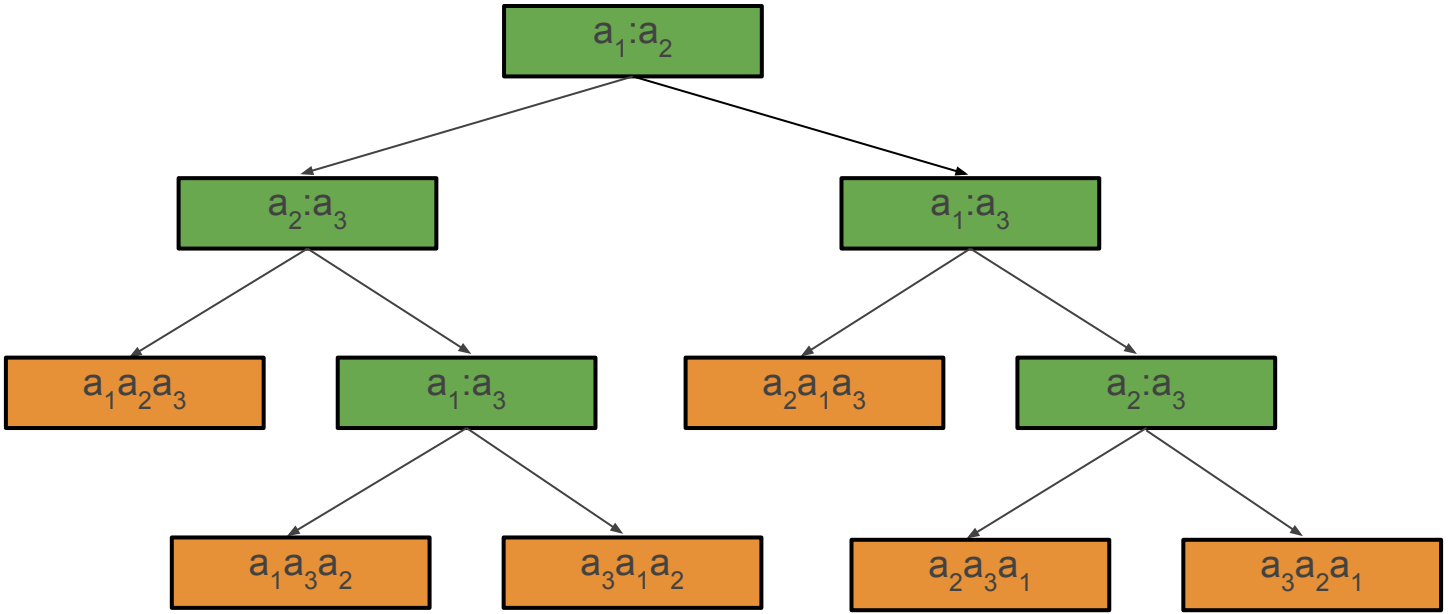
The right subtree shows subsequent comparisons if $a_i \geq a_j$.

The Decision Tree Model Continued



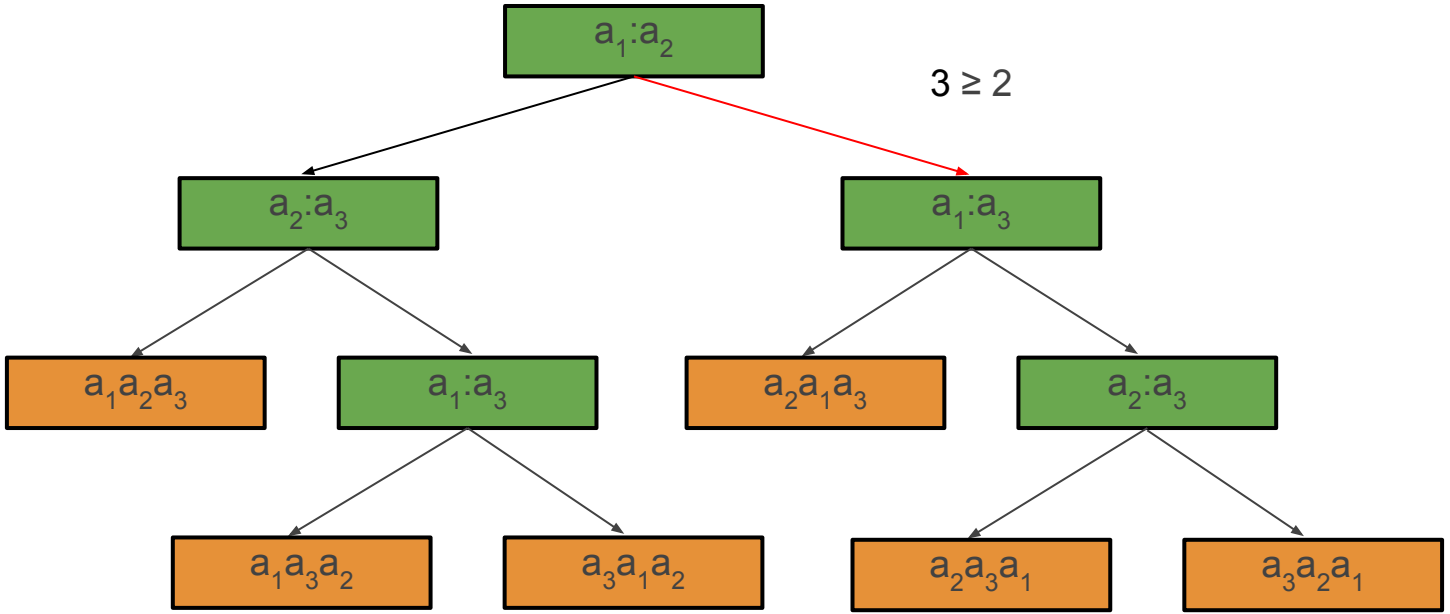
The Decision Tree Model Continued

<3, 2, 5>



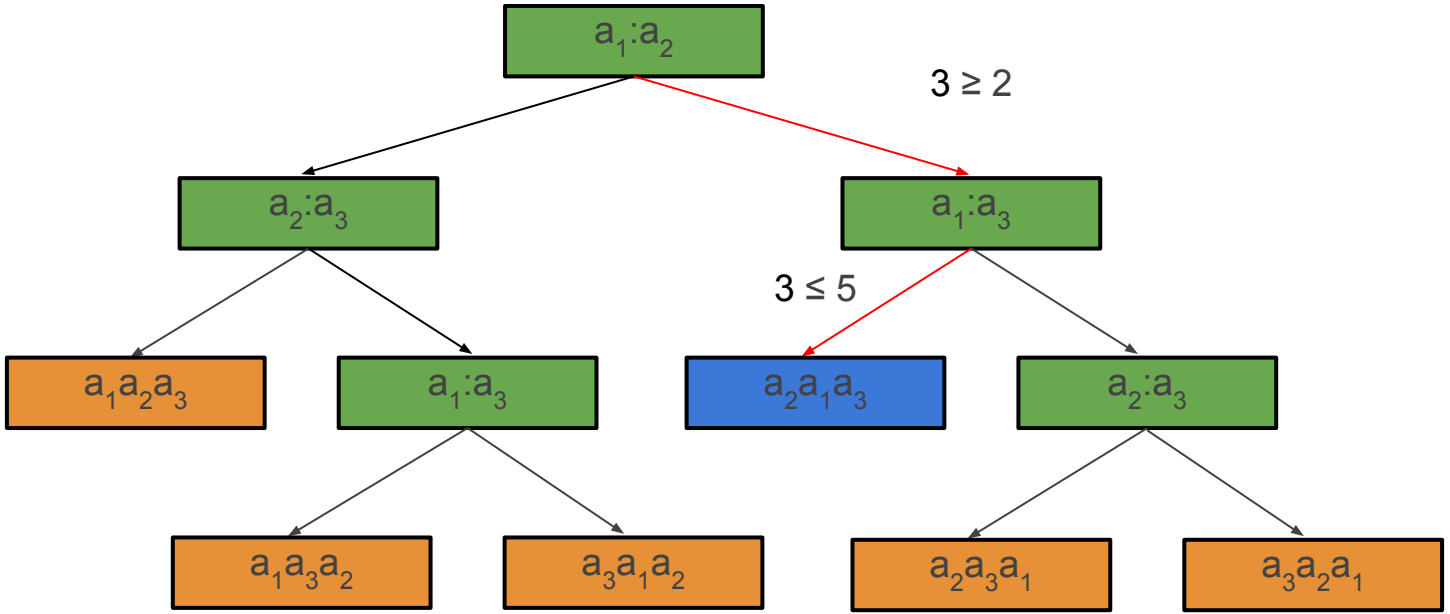
The Decision Tree Model Continued

<3, 2, 5>



The Decision Tree Model Continued

<3, 2, 5>



The Decision Tree Model Continued

A decision tree can model the execution of any comparison sort:

1. One tree for each input size n.
2. View the algorithm as splitting whenever it compares two elements.
3. The tree contains the comparisons along all possible instruction traces.
4. The running time of the algorithm = the length of the path taken.
5. Worst-case running time = height of tree.

A Lower Bound For The Worst Case

The length of the longest simple path from the root of a decision tree to any of its reachable leaves represents the worst-case number of comparisons that the corresponding sorting algorithm performs. Consequently, the worst-case number of comparisons for a given comparison sort algorithm equals the height of its decision tree.

A lower bound on the heights of all decision trees in which each permutation appears as a reachable leaf is therefore a lower bound on the running time of any comparison sort algorithm.

A Lower Bound For The Worst Case Continued

– – –

Theorem: Any decision tree that can sort n elements must have height **Ω(nlgn)**.

Proof:

Consider a decision tree of height h with l reachable leaves corresponding to a comparison sort on n elements. Because each of the n! permutations of the input appears as some leaf, we have n! ≤ l. Since a binary tree of height h has no more than 2^h leaves, we have,

$$\mathbf{n! \leq l \leq 2^h}$$

A Lower Bound For The Worst Case Continued – – –

We can write,

$2^h \geq n!$

$h \lg_2 2 \geq \lg_2 n!$ [taking logarithms]

$h \geq \lg_2 n!$

$h \geq \lg_2 n*(n-1)*(n-2)...n/2...1$

$h \geq \lg_2 n+\lg_2(n-1)+\lg_2(n-2)+...+\lg_2 n/2+...+\lg_2 1$ [$\lg_b a*b = \lg_b a + \lg_b b$]

$h \geq \lg_2 n+\lg_2(n-1)+\lg_2(n-2)+...+\lg_2 n/2$

$h \geq \lg_2 n/2+\lg_2 n/2+\lg_2 n/2+...+\lg_2 n/2$

$h \geq n/2\lg_2 n/2$

$h = \mathbf{\Omega(n \lg n)}$

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. When k = O(n), the sort runs in Θ(n) time.

Counting sort determines, for each input element x, the number of elements less than x. It uses this information to place element x directly into its position in the output array. For example, if 17 elements are less than x, then x belongs in output position 18. We must modify this scheme slightly to handle the situation in which several elements have the same value, since we do not want to put them all in the same position.

Counting Sort Continued

In the code for counting sort, we assume that the input is an array A[1...n], and thus A.length = n. We require two other arrays: the array B[1...n] holds the sorted output, and the array C[0...k] provides temporary working storage.

Counting Sort Pseudocode

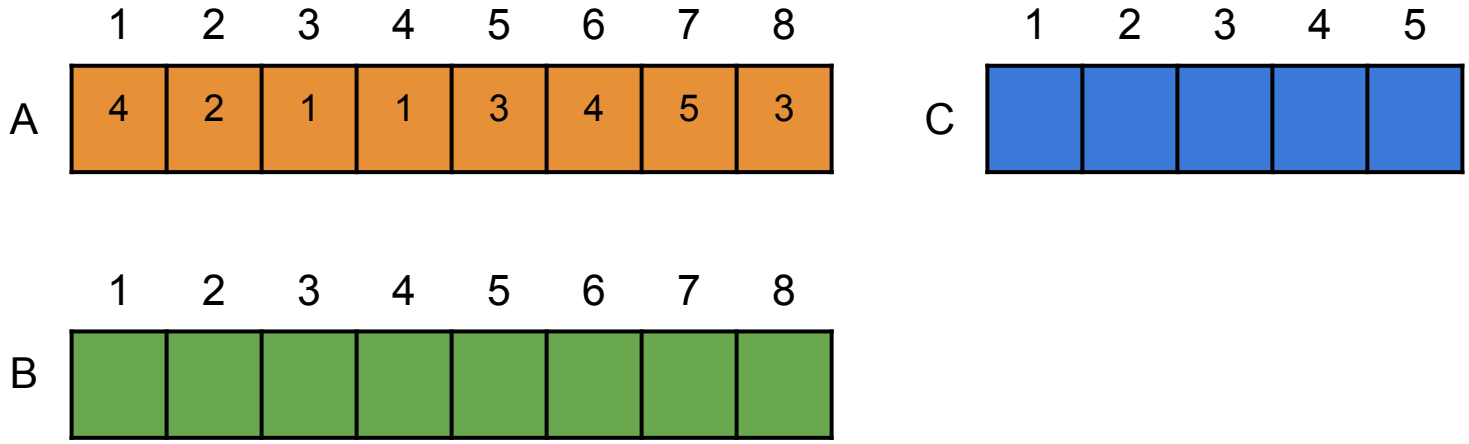
```
Counting_Sort(A, B, k):
    let C[0...k] be a new array
    for i = 0 to k:
        C[i] = 0
    for j = 1 to A.length:
        C[A[j]] = C[A[j]] + 1
    // C[i] now contains the number of elements equal to i.
    for i = 1 to k:
        C[i] = C[i] + C[i - 1]
```

Counting Sort Pseudocode

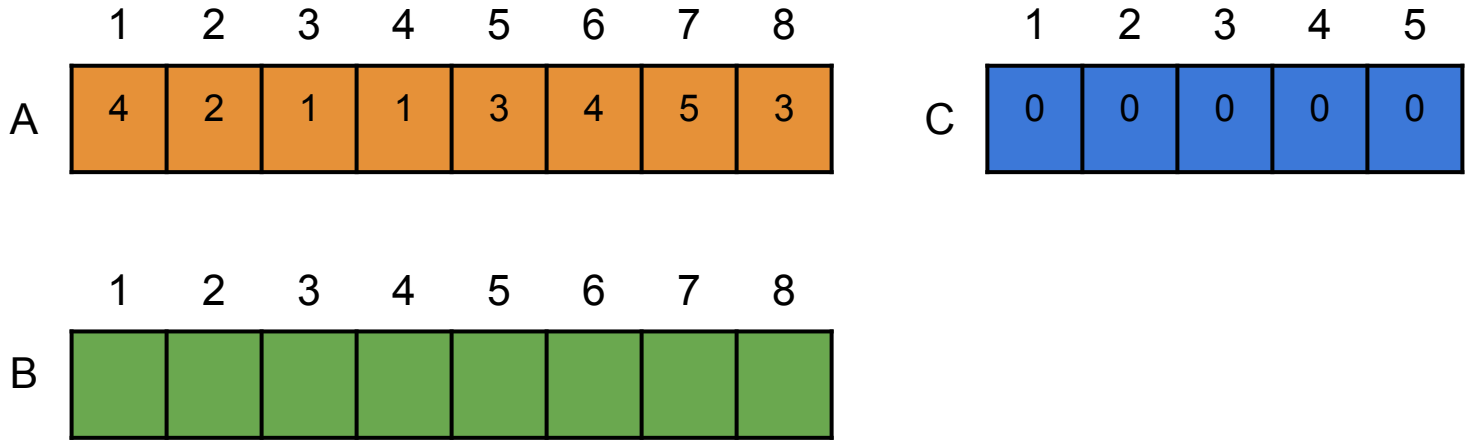
– – –

```
// CŒi now contains the number of elements less than or equal to i.
for j = A.length downto 1:
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] - 1
```

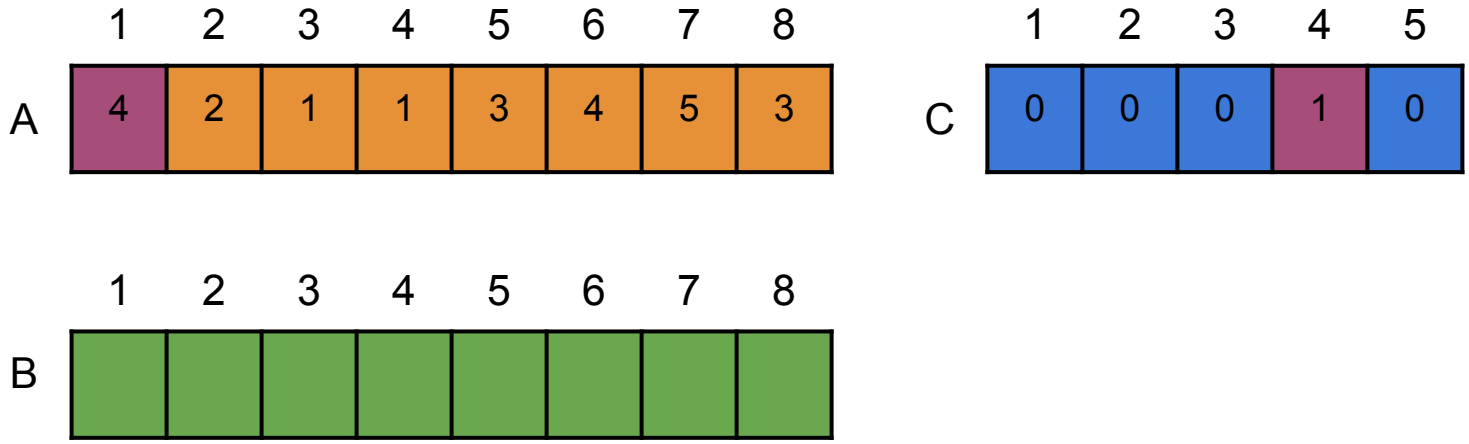
Simulation of Counting Sort



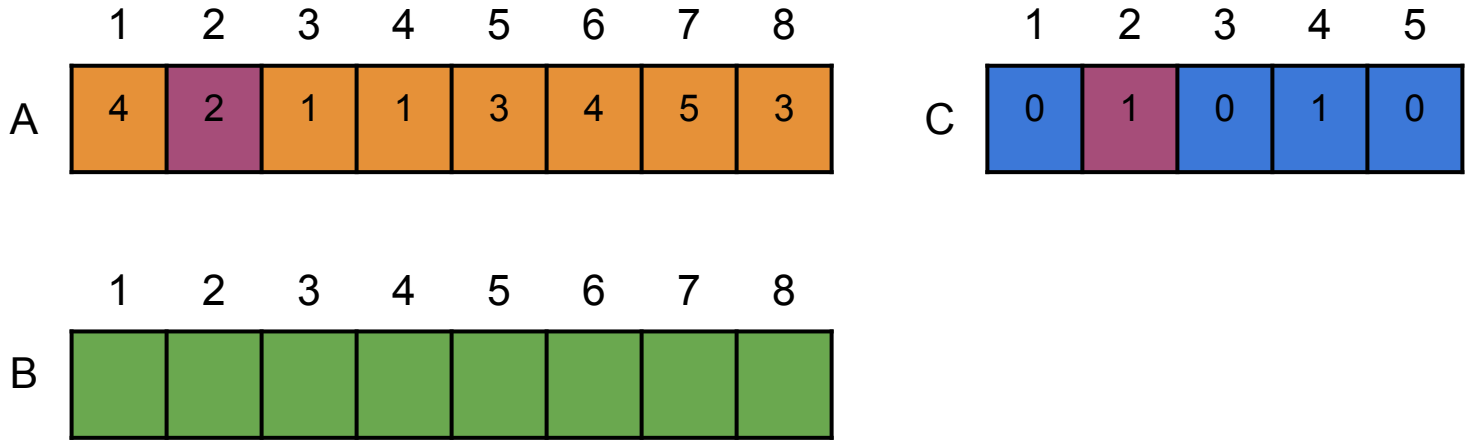
Simulation of Counting Sort



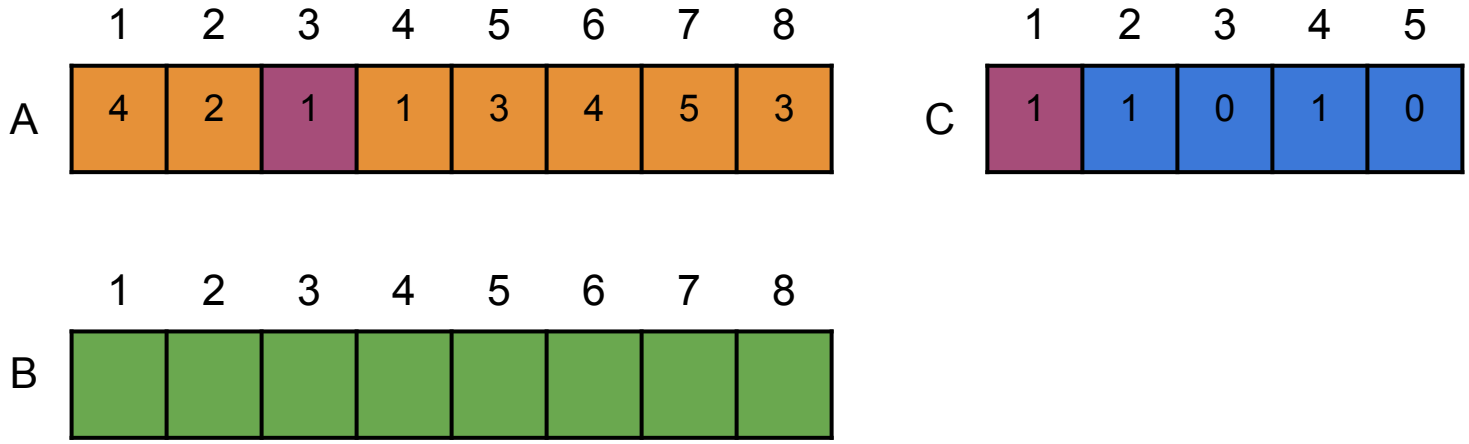
Simulation of Counting Sort



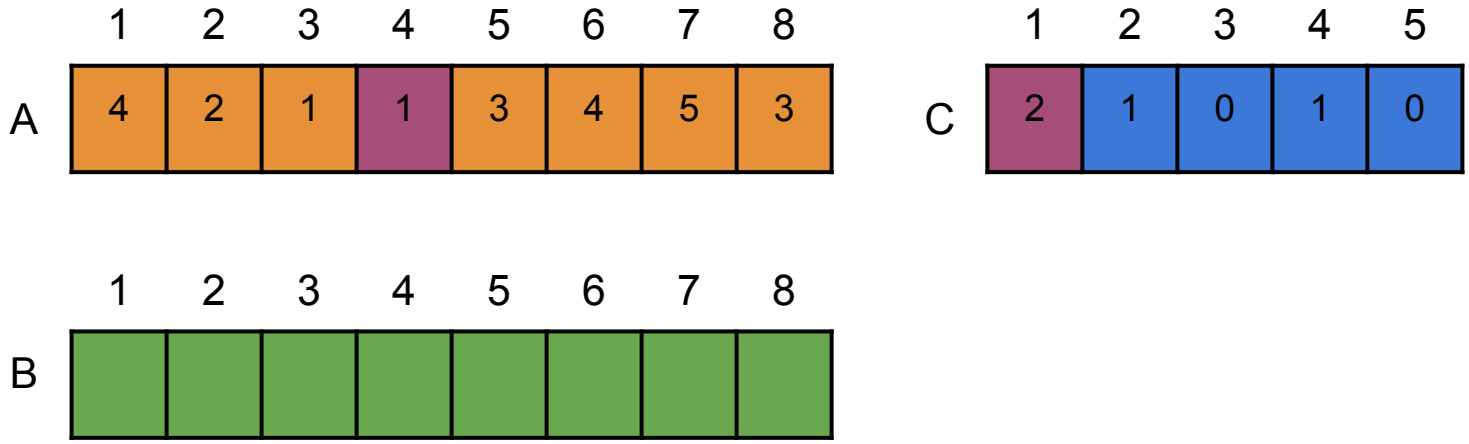
Simulation of Counting Sort



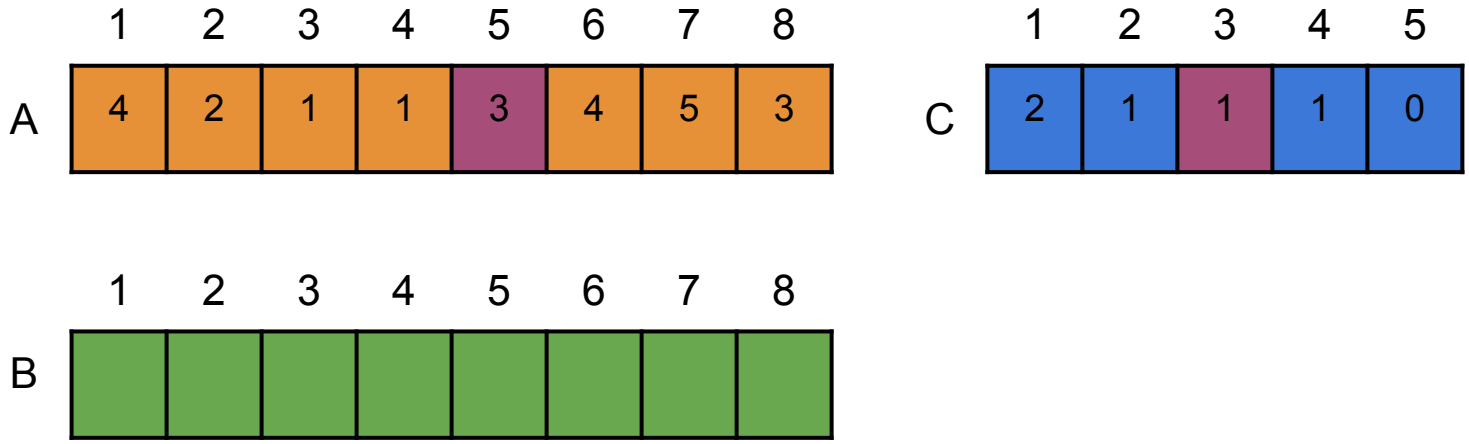
Simulation of Counting Sort



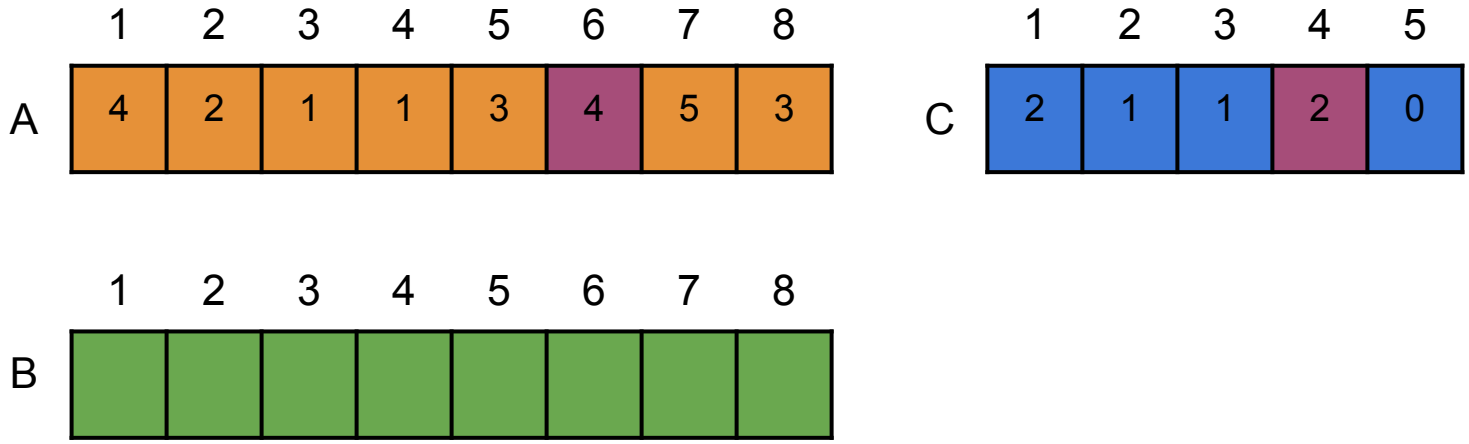
Simulation of Counting Sort



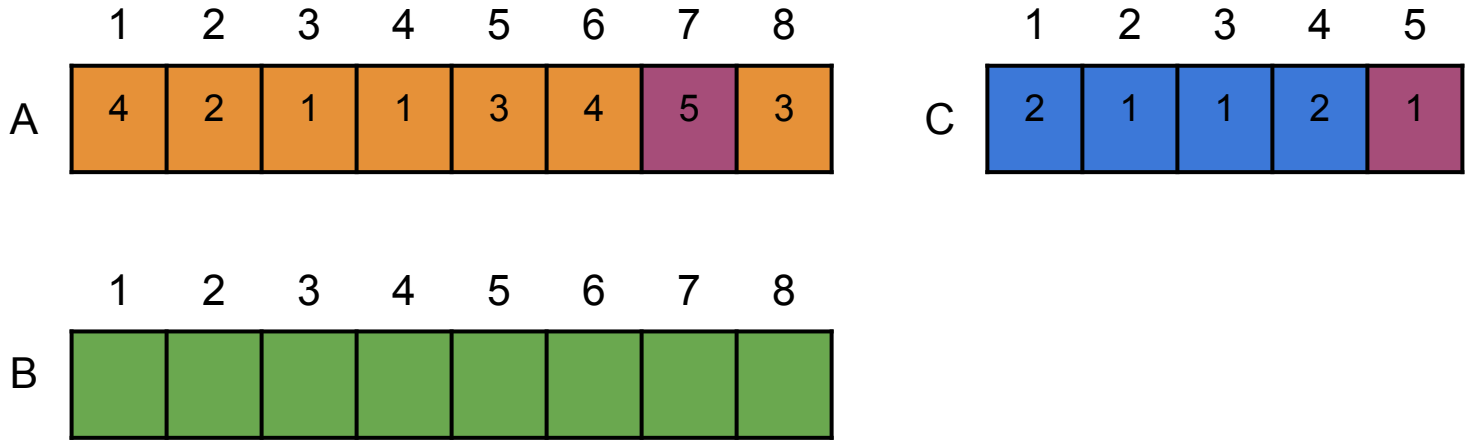
Simulation of Counting Sort



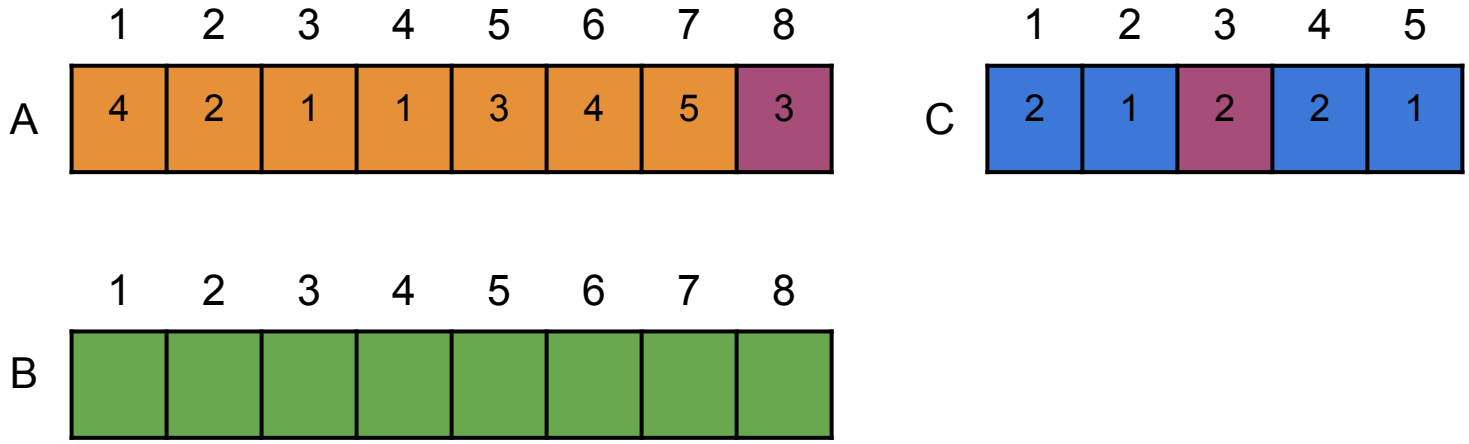
Simulation of Counting Sort



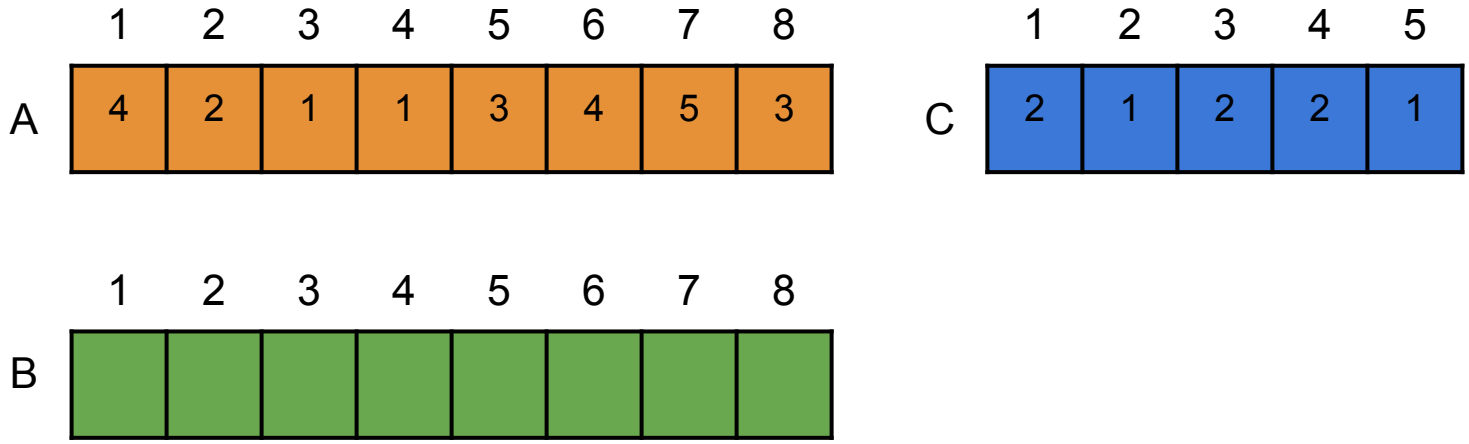
Simulation of Counting Sort



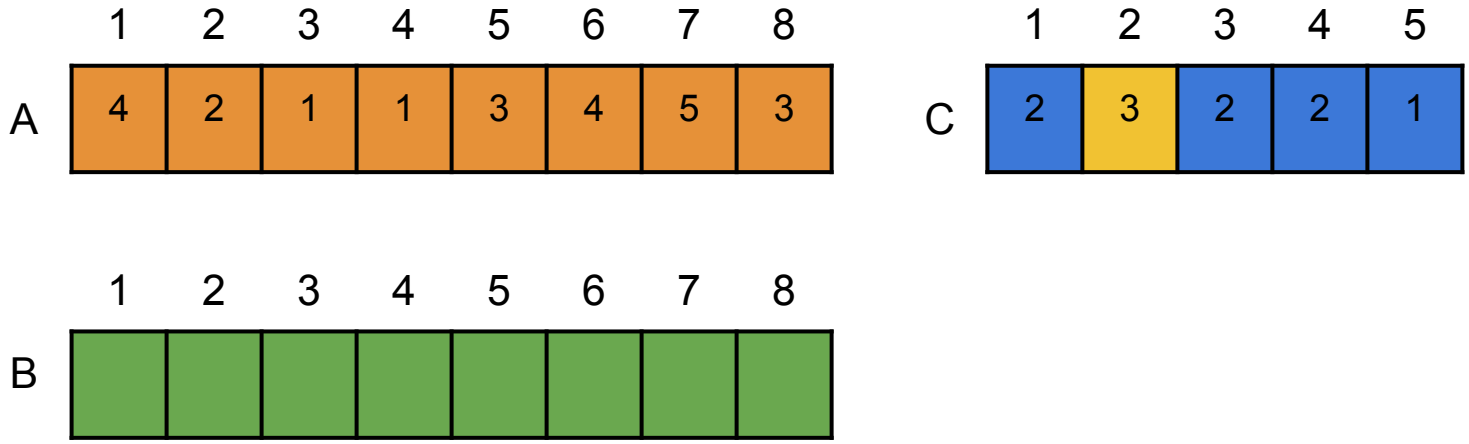
Simulation of Counting Sort



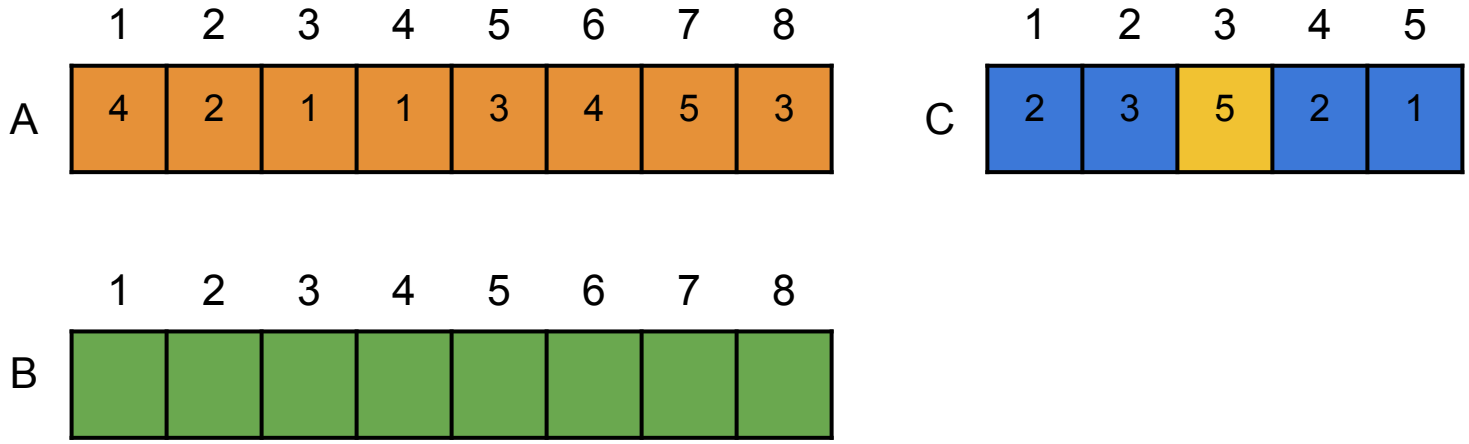
Simulation of Counting Sort



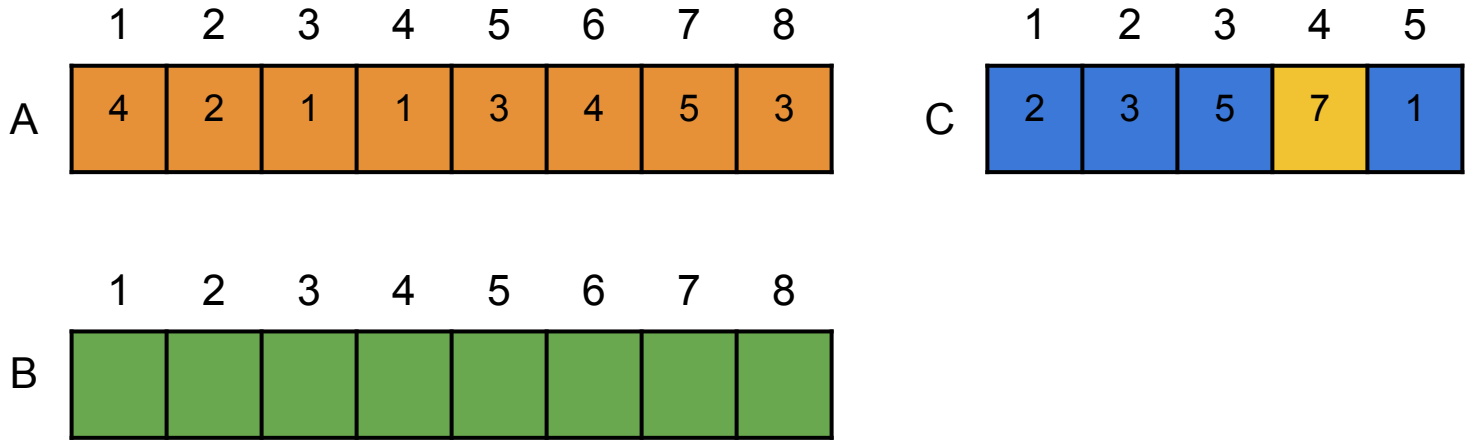
Simulation of Counting Sort



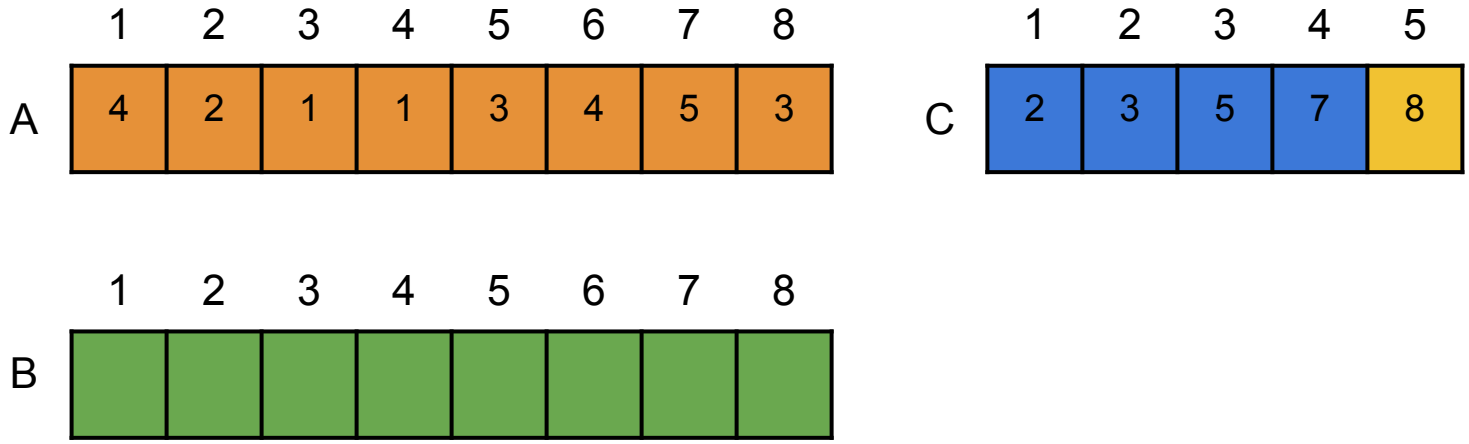
Simulation of Counting Sort



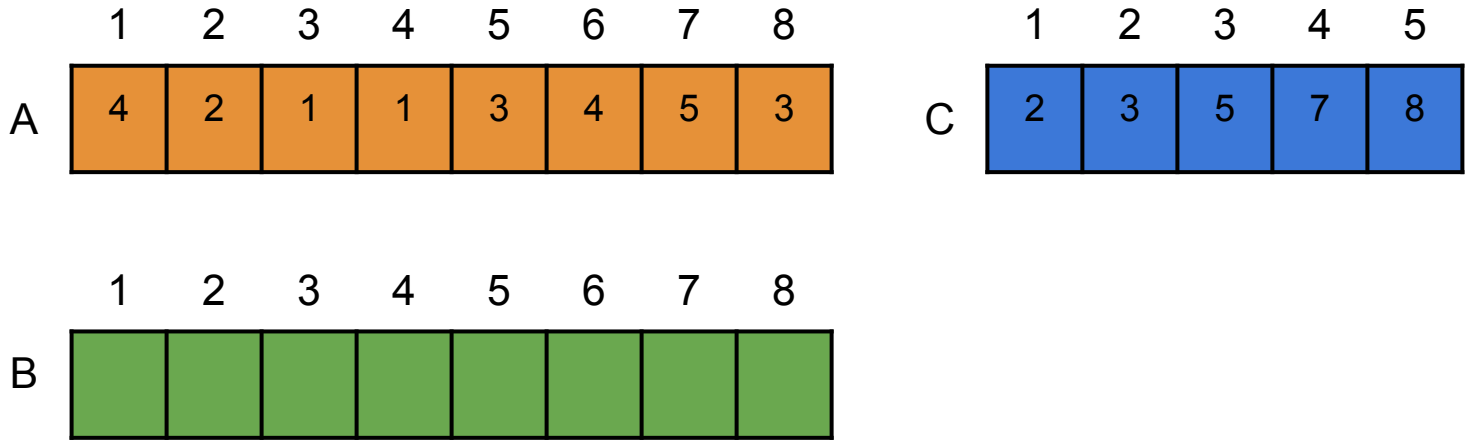
Simulation of Counting Sort



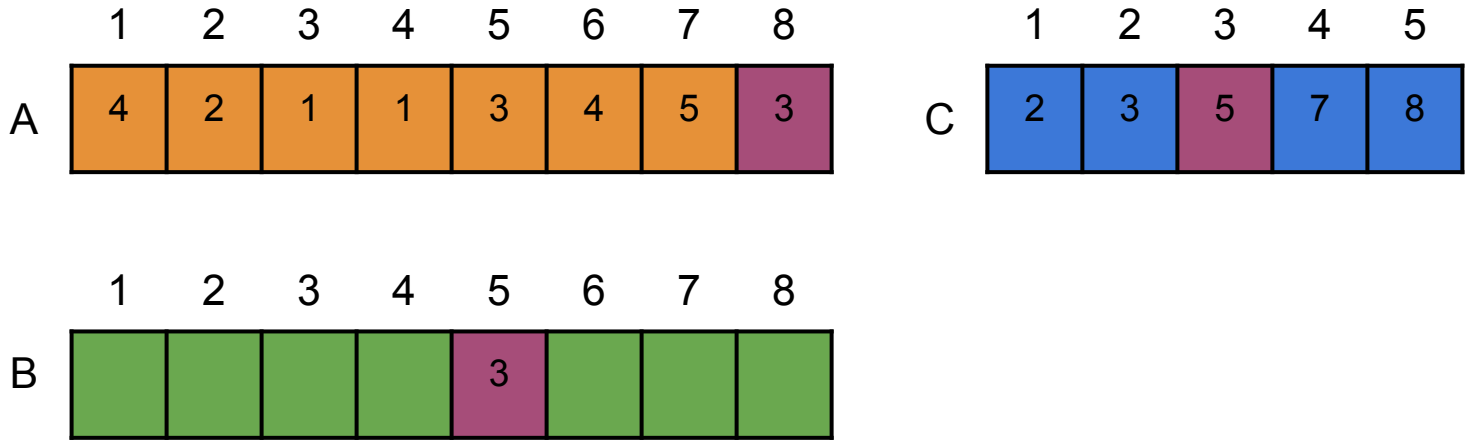
Simulation of Counting Sort



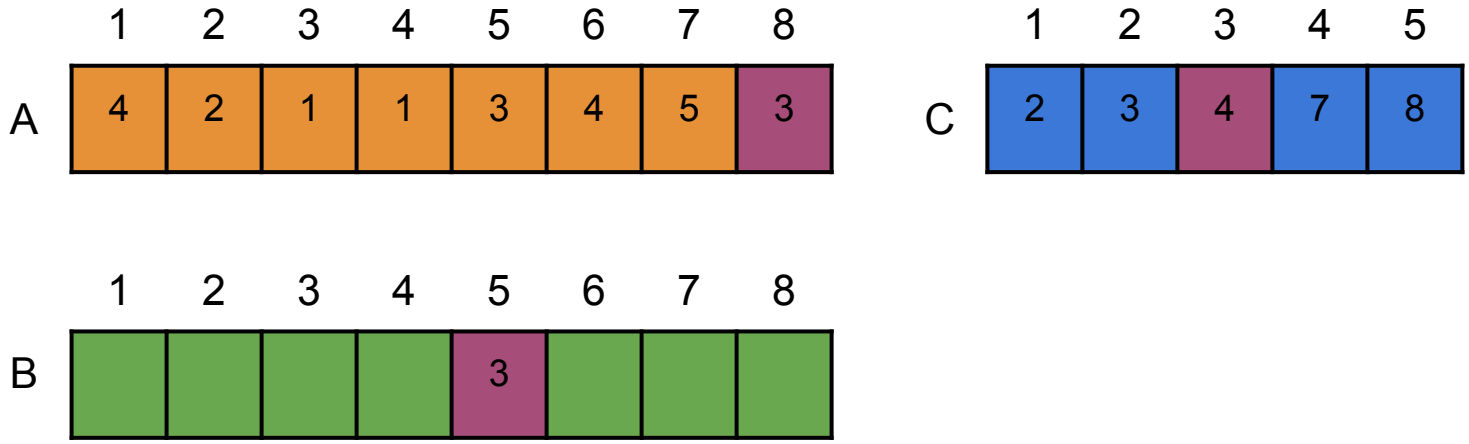
Simulation of Counting Sort



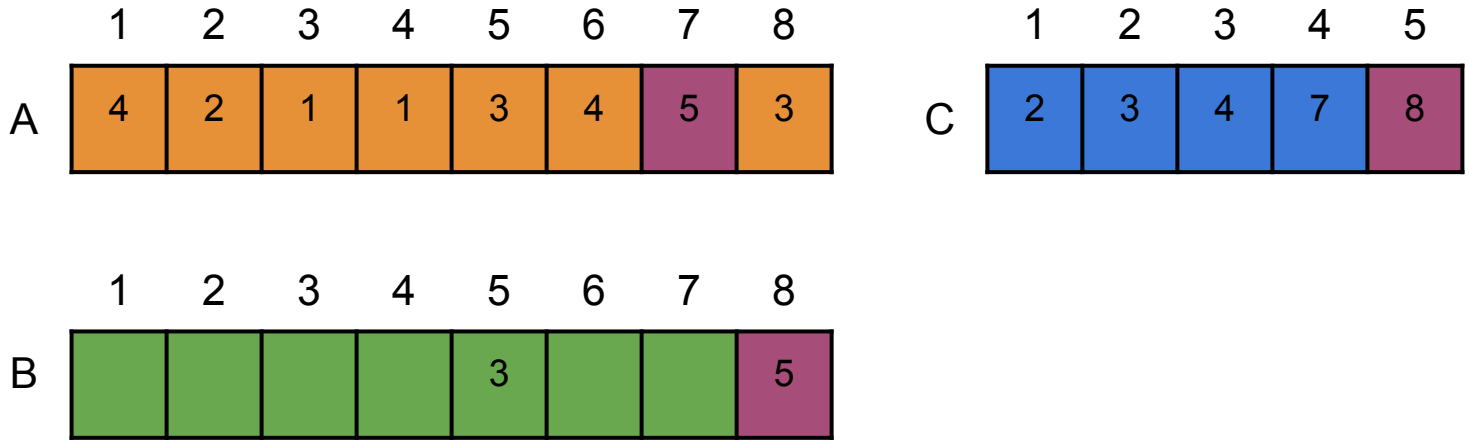
Simulation of Counting Sort



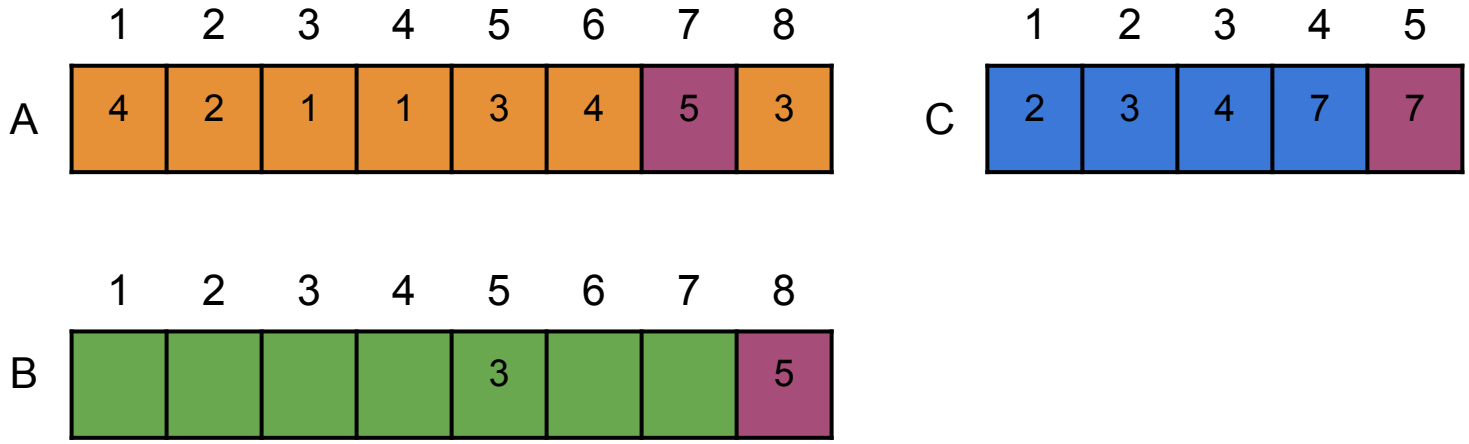
Simulation of Counting Sort



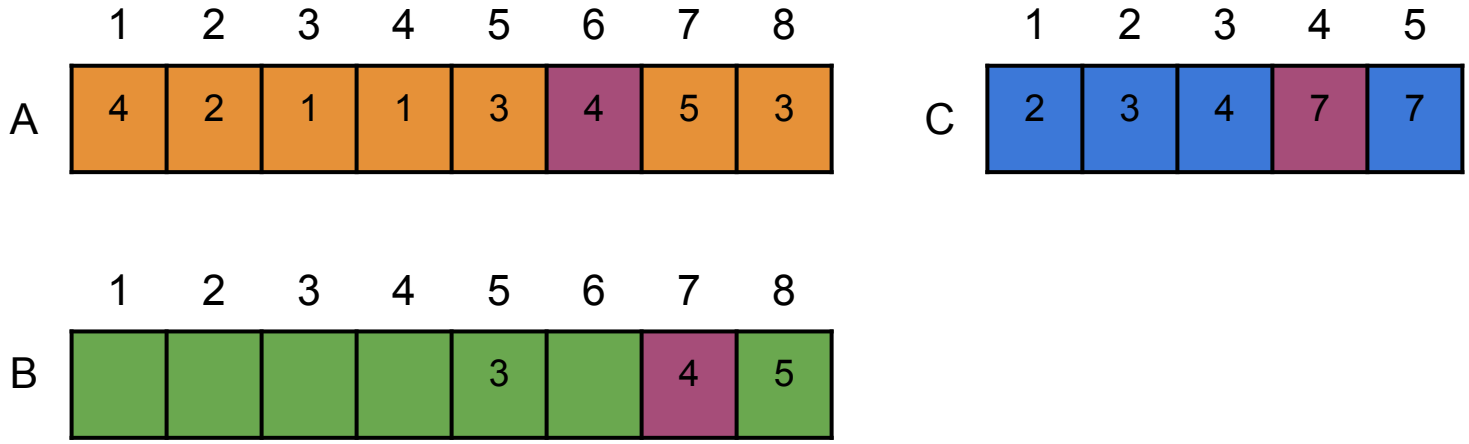
Simulation of Counting Sort



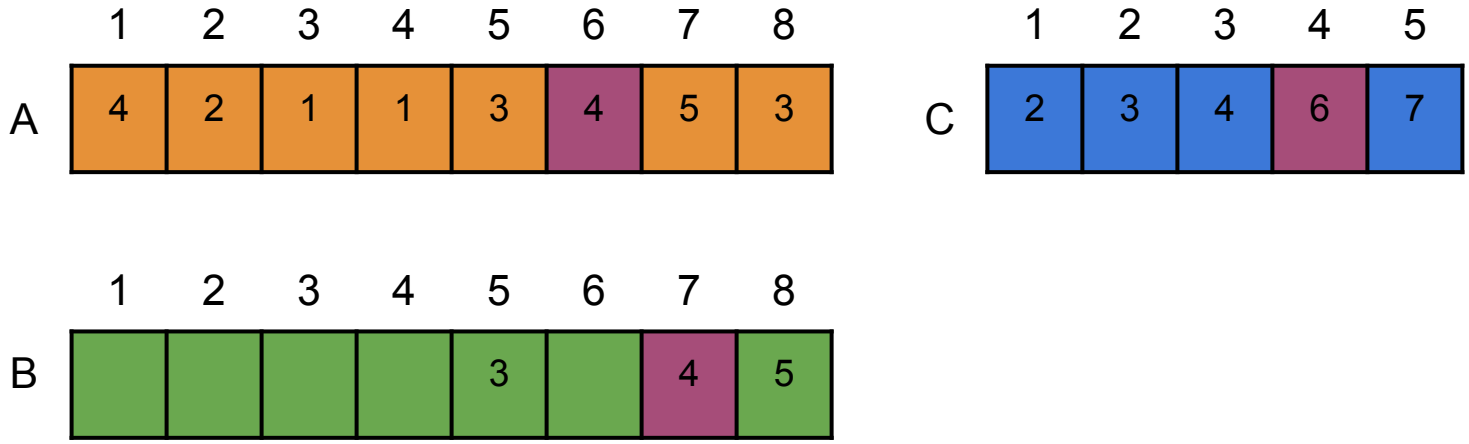
Simulation of Counting Sort



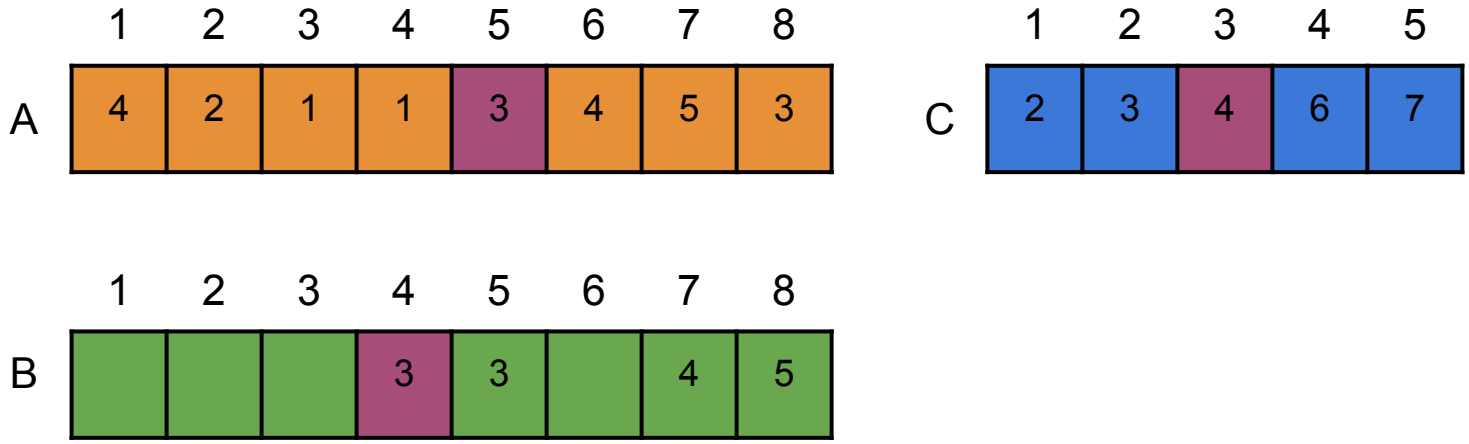
Simulation of Counting Sort



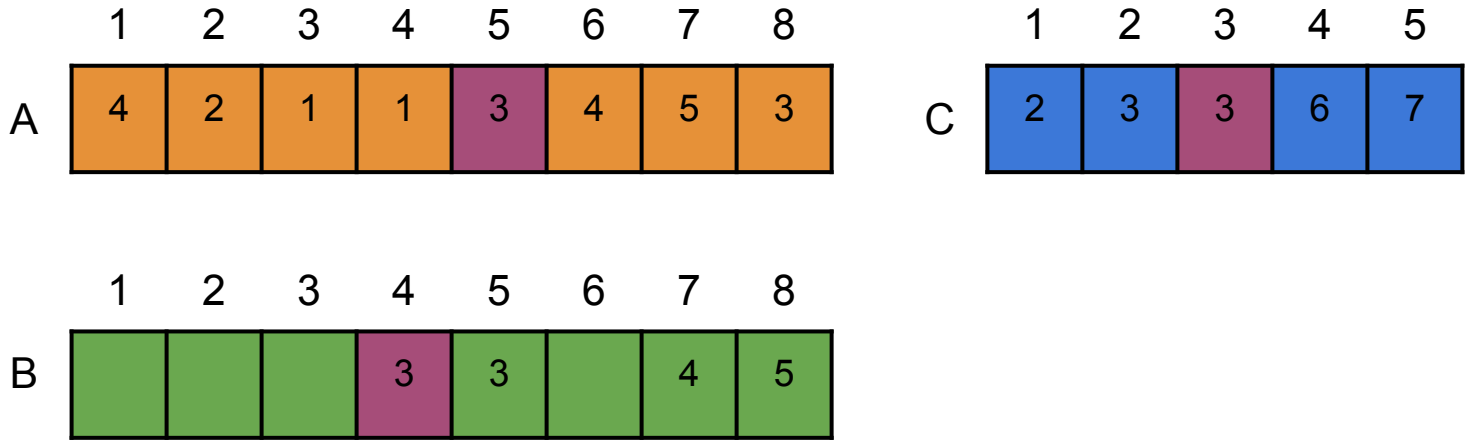
Simulation of Counting Sort



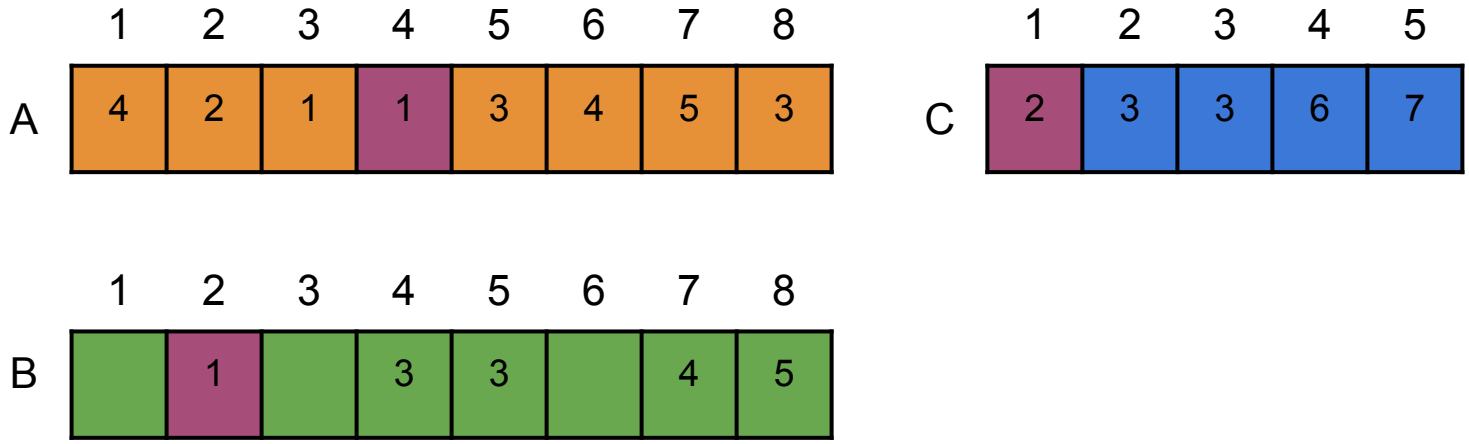
Simulation of Counting Sort



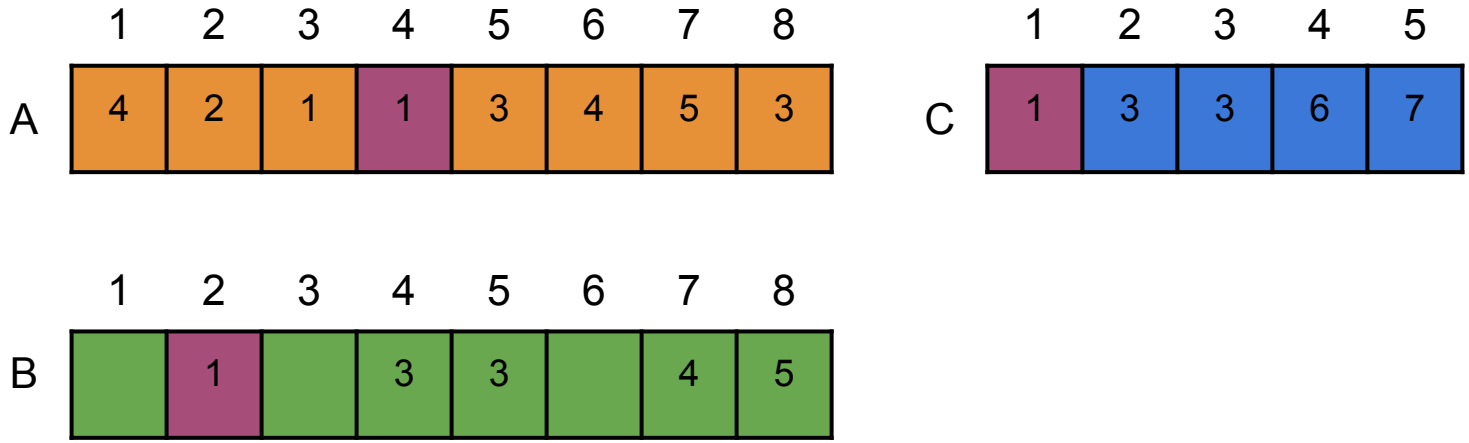
Simulation of Counting Sort



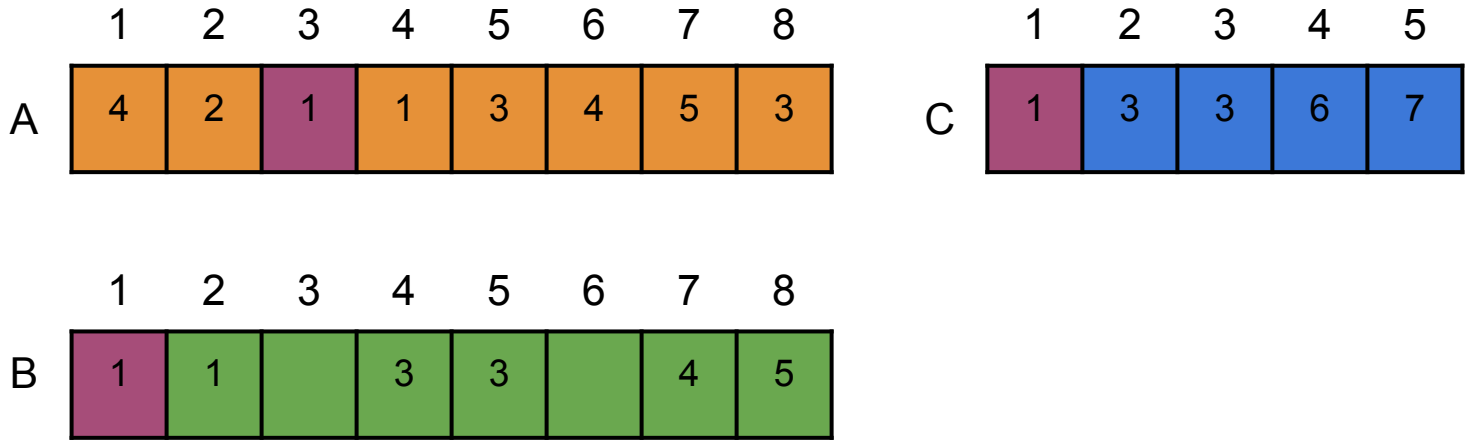
Simulation of Counting Sort



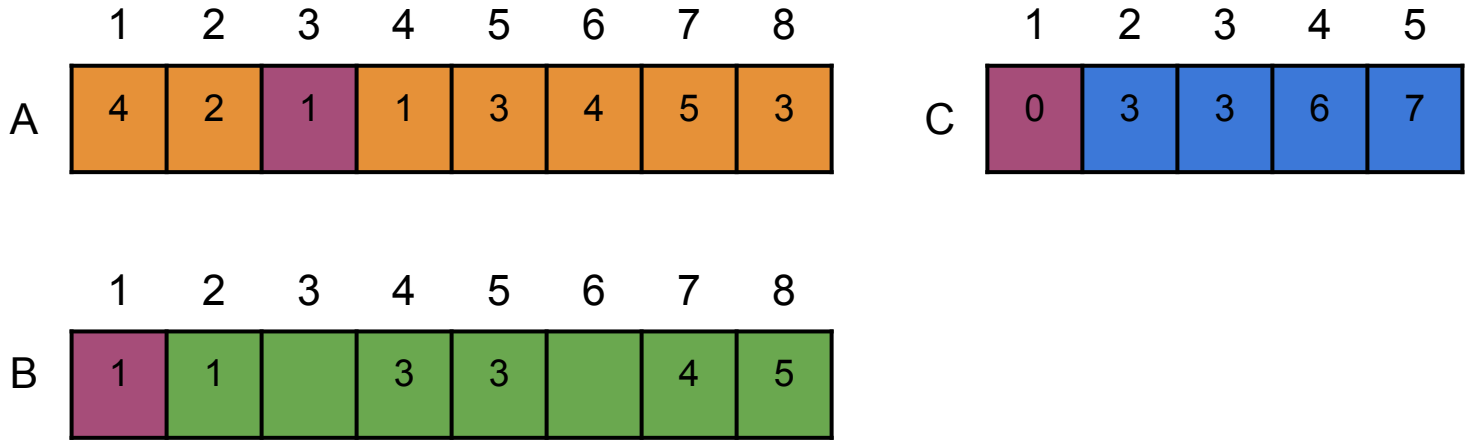
Simulation of Counting Sort



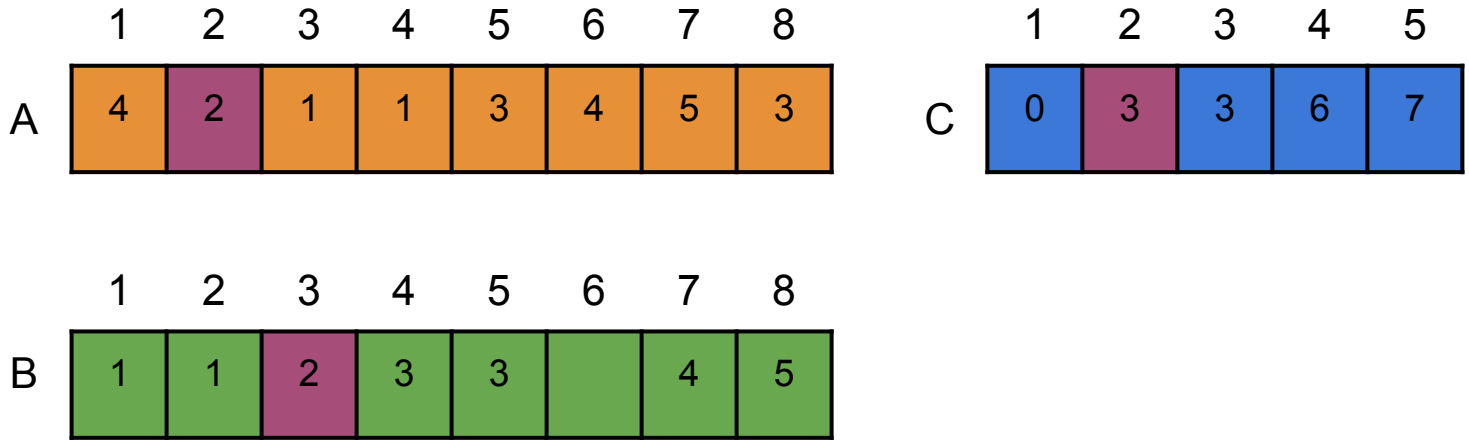
Simulation of Counting Sort



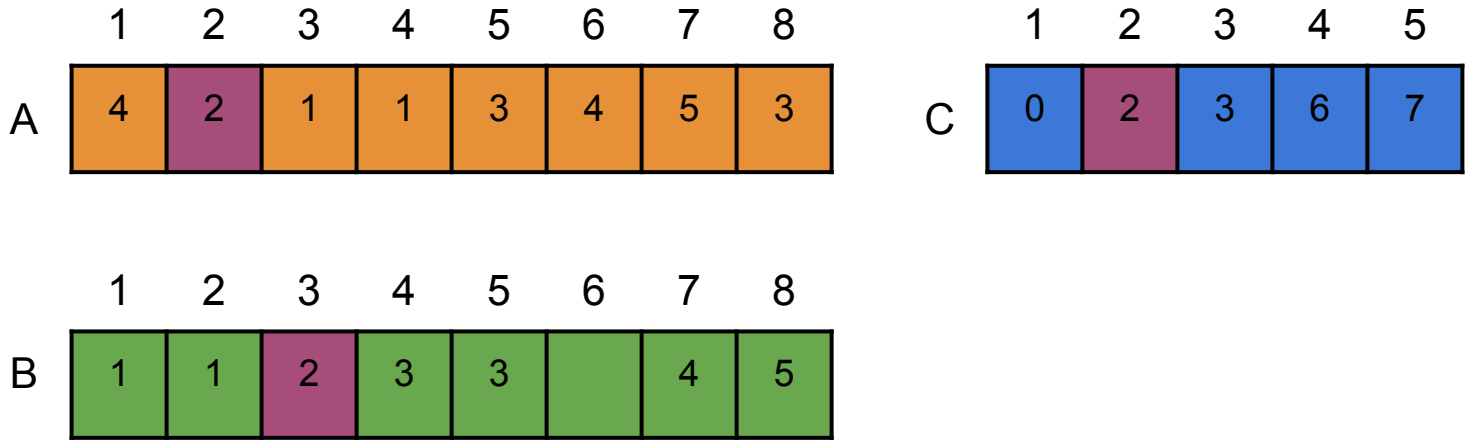
Simulation of Counting Sort



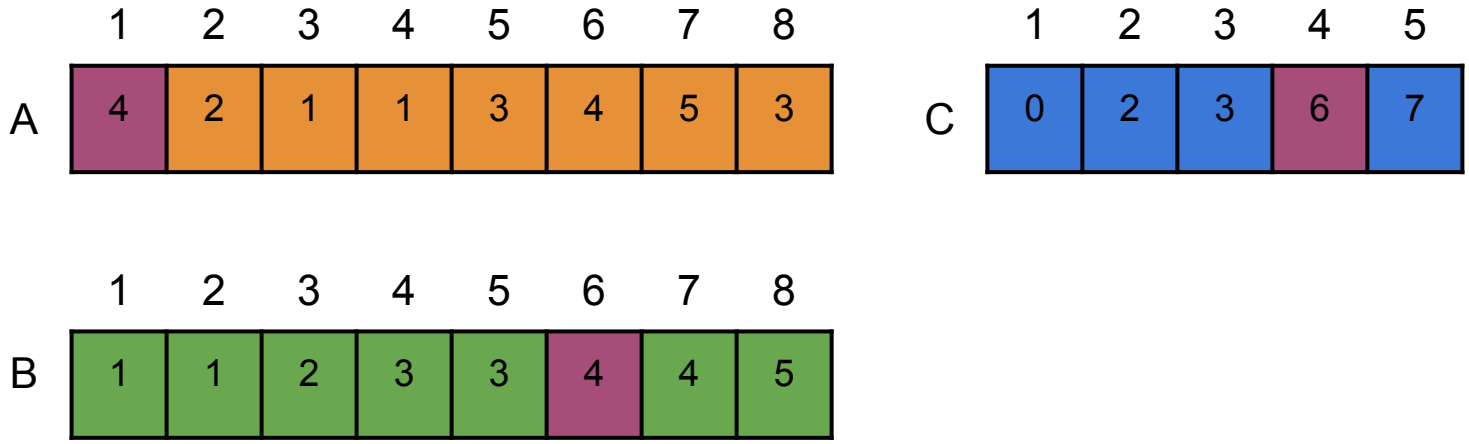
Simulation of Counting Sort



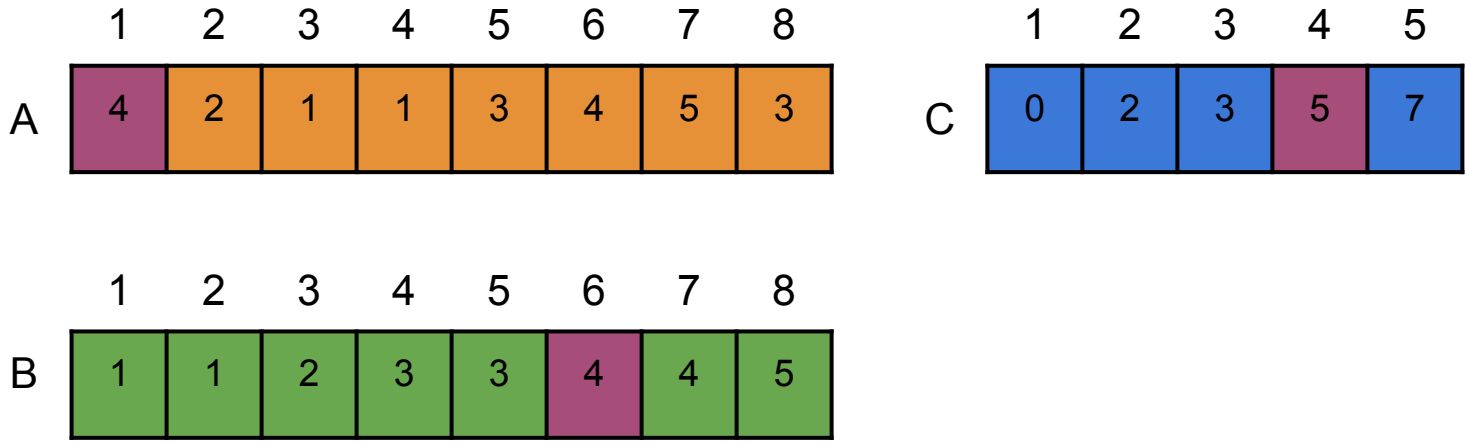
Simulation of Counting Sort



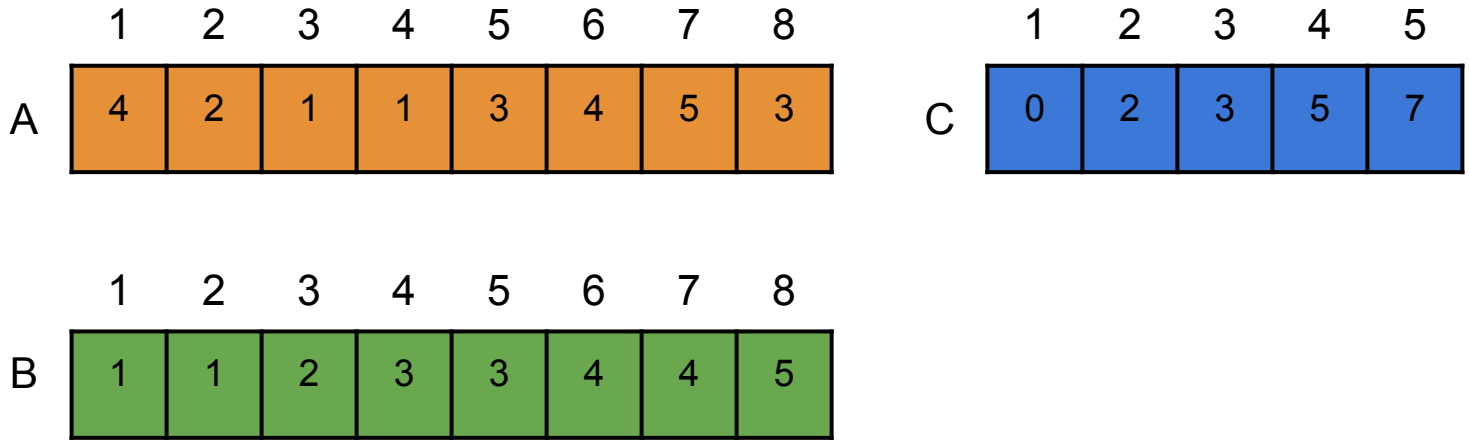
Simulation of Counting Sort



Simulation of Counting Sort



Simulation of Counting Sort



Analysis of Counting Sort Continued

– – –

If k = O(n), then counting sort takes (n) time. But, normally sorting takes Θ(nlgn) time!

So, where's the fallacy?

1. Comparison sorting takes (n lg n) time.
2. Counting sort is not a comparison sort.
3. In fact, not a single comparison between elements occurs!

Analysis of Counting Sort

– – –

for i = 0 to k:

    C[i] = 0

Θ(k)

for j = 1 to A.length:

    C[A[j]] = C[A[j]] + 1

Θ(n)

for i = 1 to k:

    C[i] = C[i] + C[i - 1]

Θ(k)

for j = A.length downto 1:

    B[C[A[j]]] = A[j]

    C[A[j]] = C[A[j]] - 1

Θ(n)

So,

    Θ(n+k)

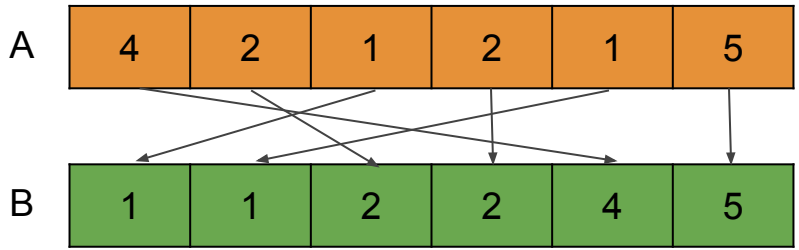
Stable Sorting

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

Counting sort is a stable sort as it preserves the input order among equal elements. Some other Sorting Algorithms are stable by nature, such as Bubble Sort, Insertion Sort, Merge Sort,

Stable Sorting

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted. Counting sort is a stable sort as it preserves the input order among equal elements. Some other Sorting Algorithms are stable by nature, such as Bubble Sort, Insertion Sort, Merge Sort,



Radix Sort

Radix sort is the algorithm used by the card-sorting machines you now find only in computer museums.

Digit-by-digit Sorting.

Sort on least-significant digit first with auxiliary stable sort

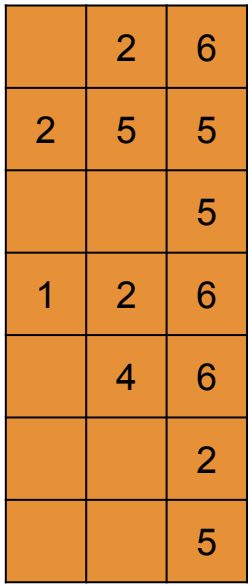
Radix Sort Pseudocode

– – –

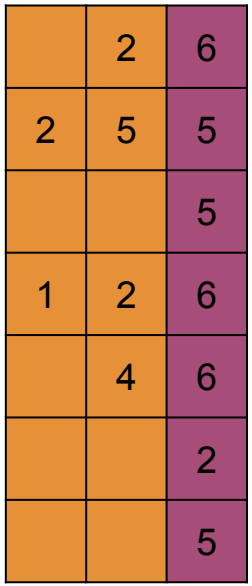
Radix_Sort(A, d):

```
    for i = 1 to d:
        use a stable sort to sort array A on digit i
```

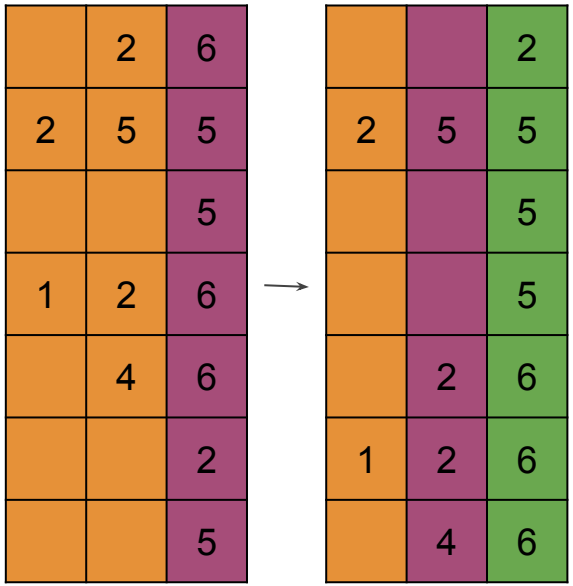
Simulation of Radix Sort



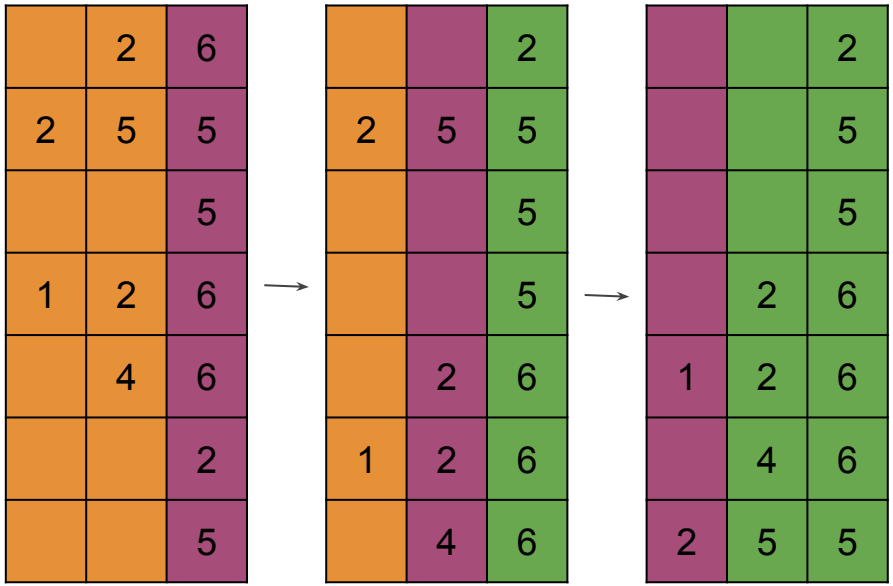
Simulation of Radix Sort



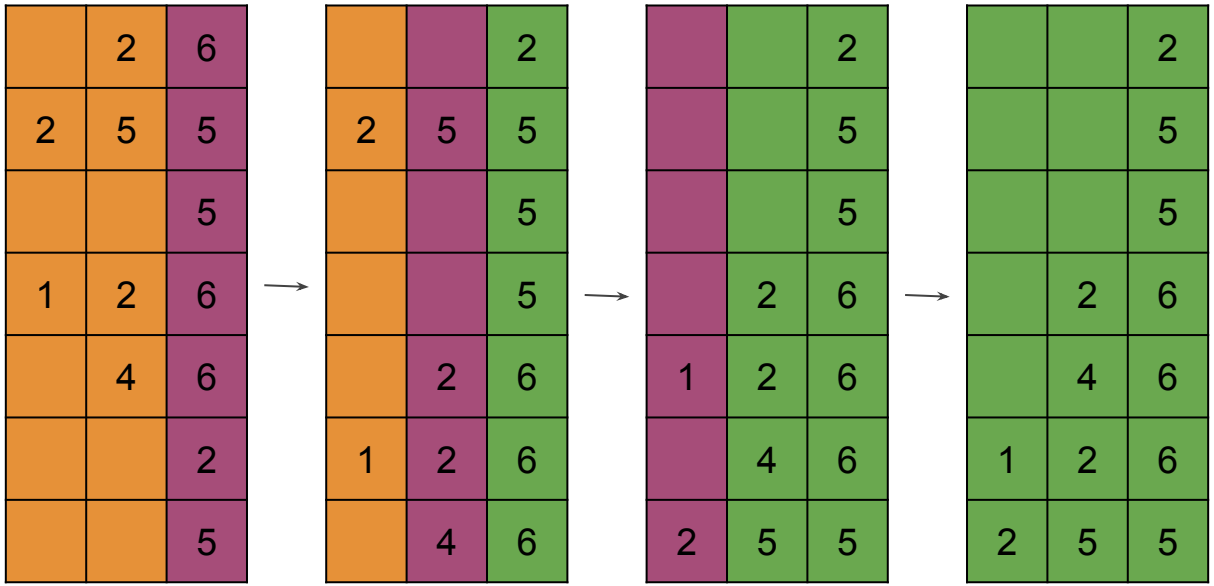
Simulation of Radix Sort



Simulation of Radix Sort



Simulation of Radix Sort



Complexity Analysis of Radix Sort

We know that counting sort is an auxiliary stable sort used. Sort n computer words of b bits each. Each word can be viewed as having b/r base-2^r digits.

For example: We take 32-bit word

If r = 8 then b/r = 4 passes of counting sort on base-2^8 digits;

or

If r = 16 then b/r = 2 passes of counting sort on base-2^{16} digits.

So, How many passes should we make?

Complexity Analysis of Radix Sort Continued

– – –

Counting sort takes (n + k) time to sort n numbers in the range from 0 to k – 1.

If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n + 2^r)$ time. Since there are b/r passes, we have

$$T(n,b) = \Theta(b/r(n + 2^r))$$

We need to choose r to so that we can minimize T(n, b). Increasing r means fewer passes, but as r >> lgn, the time grows exponentially.

Complexity Analysis of Radix Sort Continued – – –

Minimize T(n, b) by differentiating and setting to 0.

Or

Just observe that we don't want $2^r >> n$, and there's no harm asymptotically in choosing r as large as possible subject to this constraint.

Choosing r = lgn implies T(n, b) = Θ(b n/lg n). For numbers in the range from 0 to $n^d – 1$, we have b = dlgn radix sort runs in Θ(dn) time.