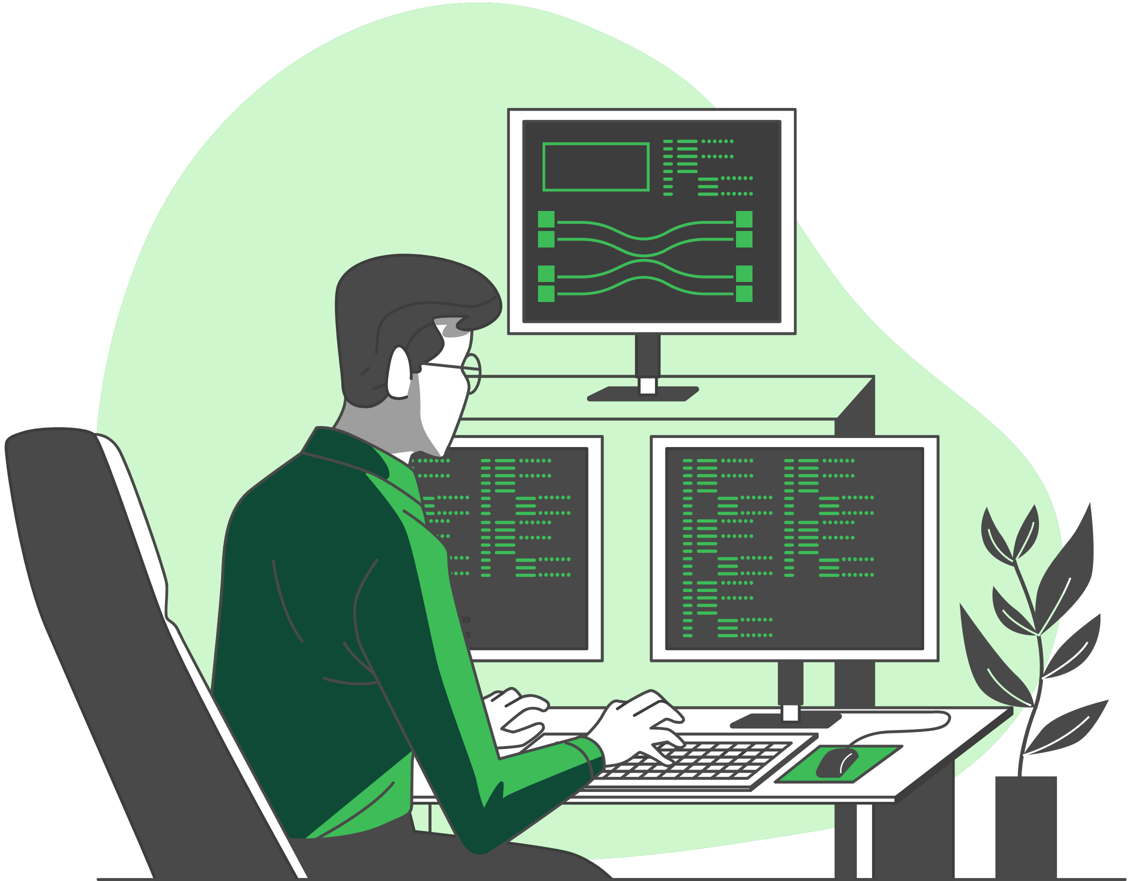
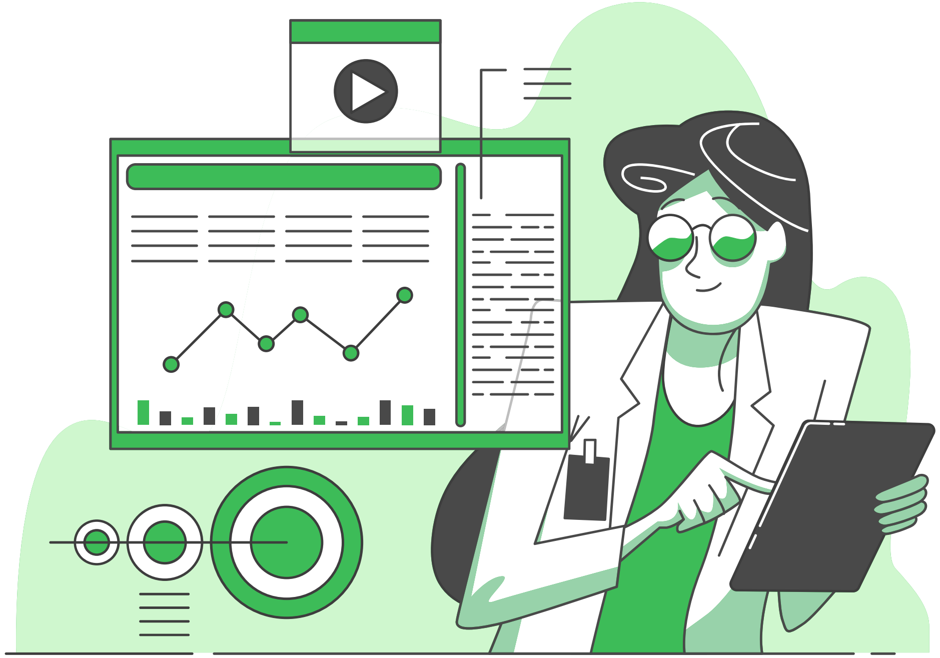


Linear Sorting

CS 251 - Data Structures and Algorithms

Note:
Slides complement the discussion in class

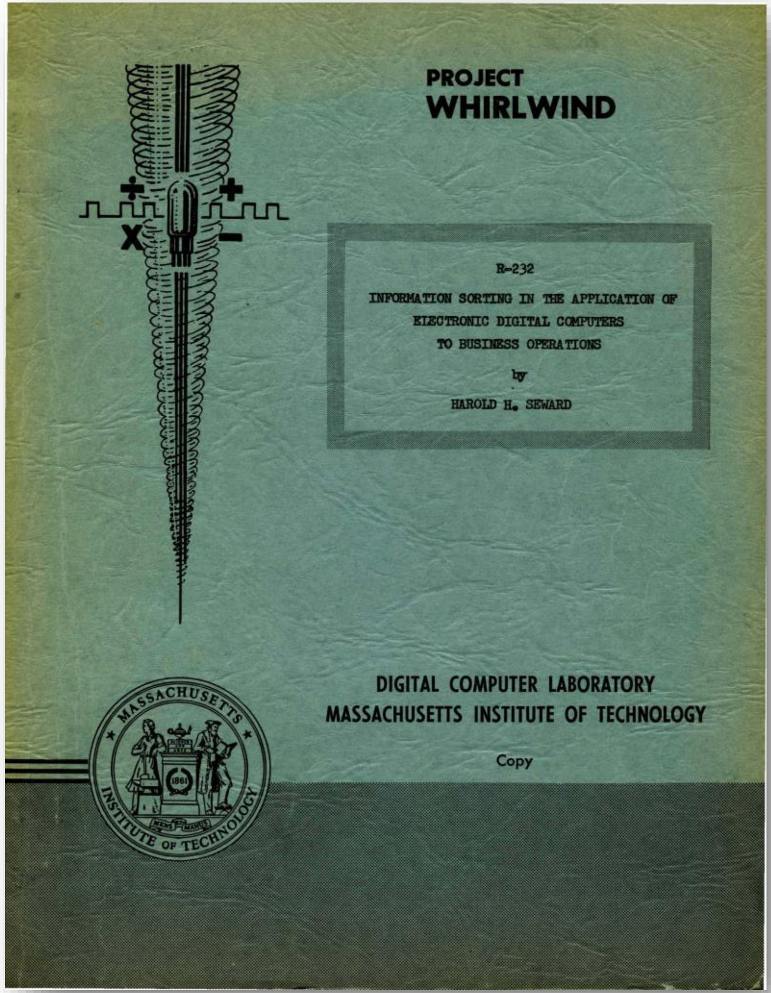
Table of Contents



01
Counting Sort

Why comparing when we can just count?

Harold H. Seward. "Information Sorting in the Application of Electronic Digital Computers to Business Operations." Master's thesis, MIT, 1954.


PROJECT WHIRLWIND

R-232

INFORMATION SORTING IN THE APPLICATION OF ELECTRONIC DIGITAL COMPUTERS TO BUSINESS OPERATIONS

by

HAROLD H. SEWARD

DIGITAL COMPUTER LABORATORY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Copy

https://www.cs.cornell.edu/courses/JavaAndDS/files/R-232_Info_Sorting_in_the_Applic_Electronic_Computers_Busin_Ops_May54.pdf

Just count?

*“**Counting sort** assumes that each of the n input elements is an integer in the range 0 to k, for some [positive] integer k. When $k \in O(n)$, then [the algorithm] runs in $\Theta(n)$.”*

So, how do we sort an array without comparing its elements?

Step 1: Array C keeps the number of occurrences for each element in A.

Step 2: Count the occurrences of each item in A. Use A[i] as the indices of C.

Step 3: Accumulate the count values in C from left to right.

Step 4: Use values in C to determine the final index for each element in A.

Step 5 (optional): Copy the elements from B to A if they must be in the original array.

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

	0	1	2	3	4	5	6	7
A	(2, A)	(5, C)	(3, H)	(0, B)	(2, X)	(3, B)	(0, X)	(3, D)

k = 5

Step 1: Array C keeps the number of occurrence for each element in A.

	0	1	2	3	4	5
C	0	0	0	0	0	0

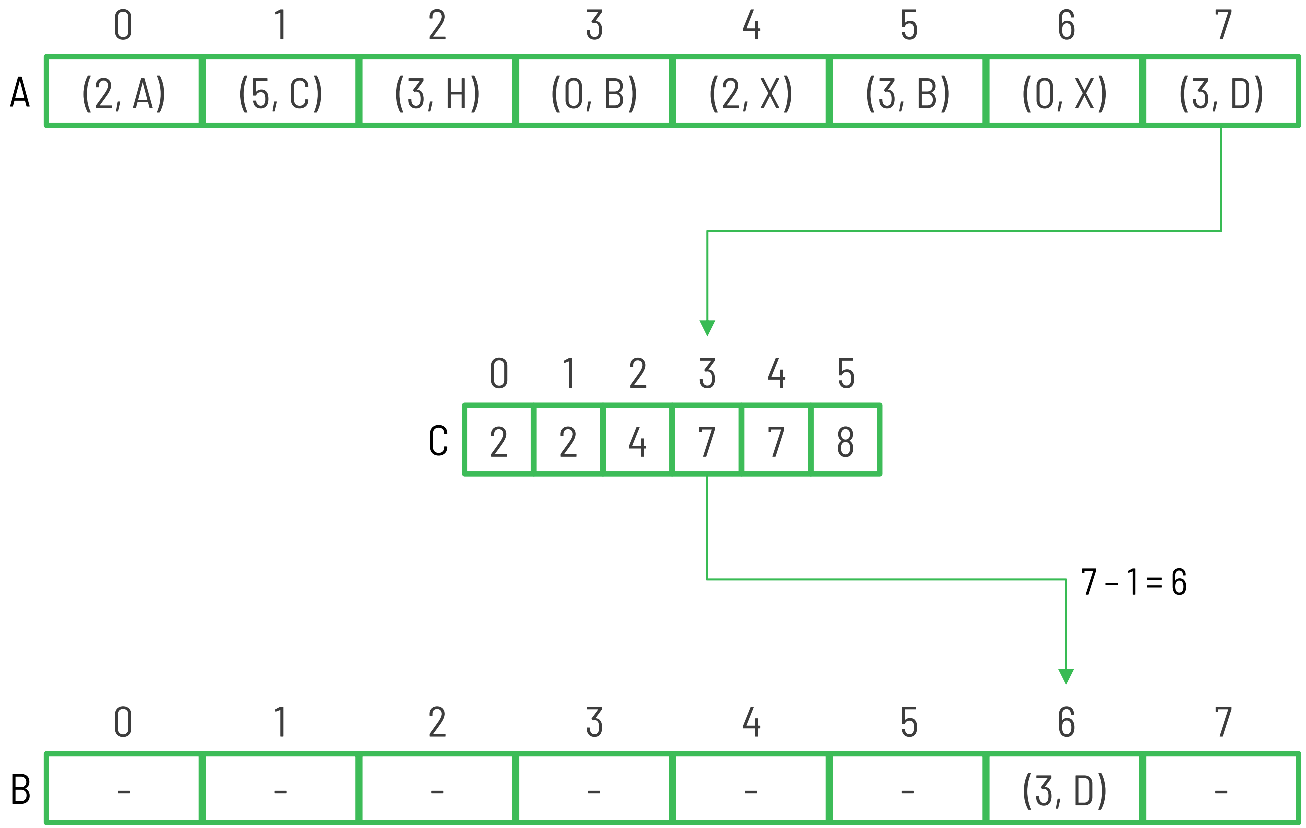
Step 2: Count the occurrences of each item in A. Use A[i] as the indices of C.

	0	1	2	3	4	5
C	2	0	2	3	0	1

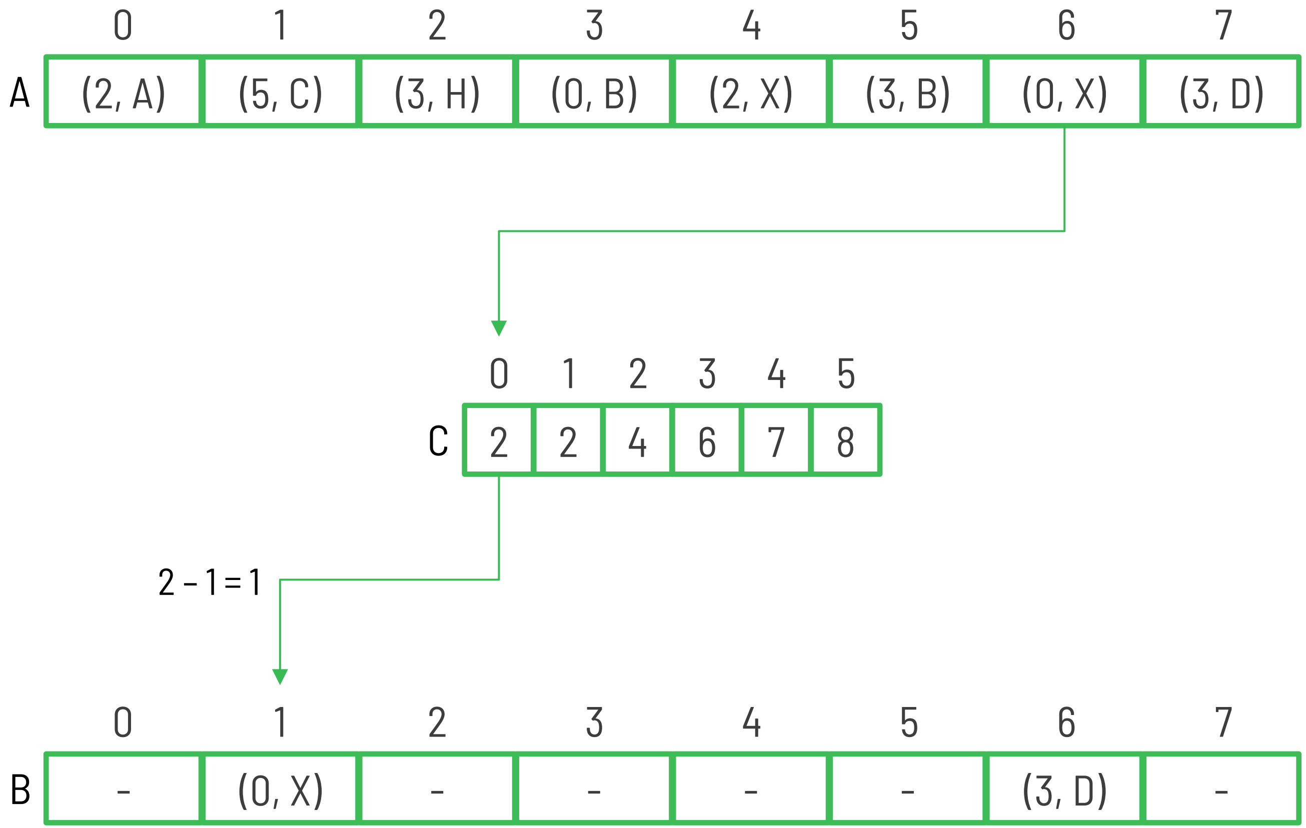
Step 3: Accumulate the count values in C from left to right.

	0	1	2	3	4	5
C	2	2	4	7	7	8

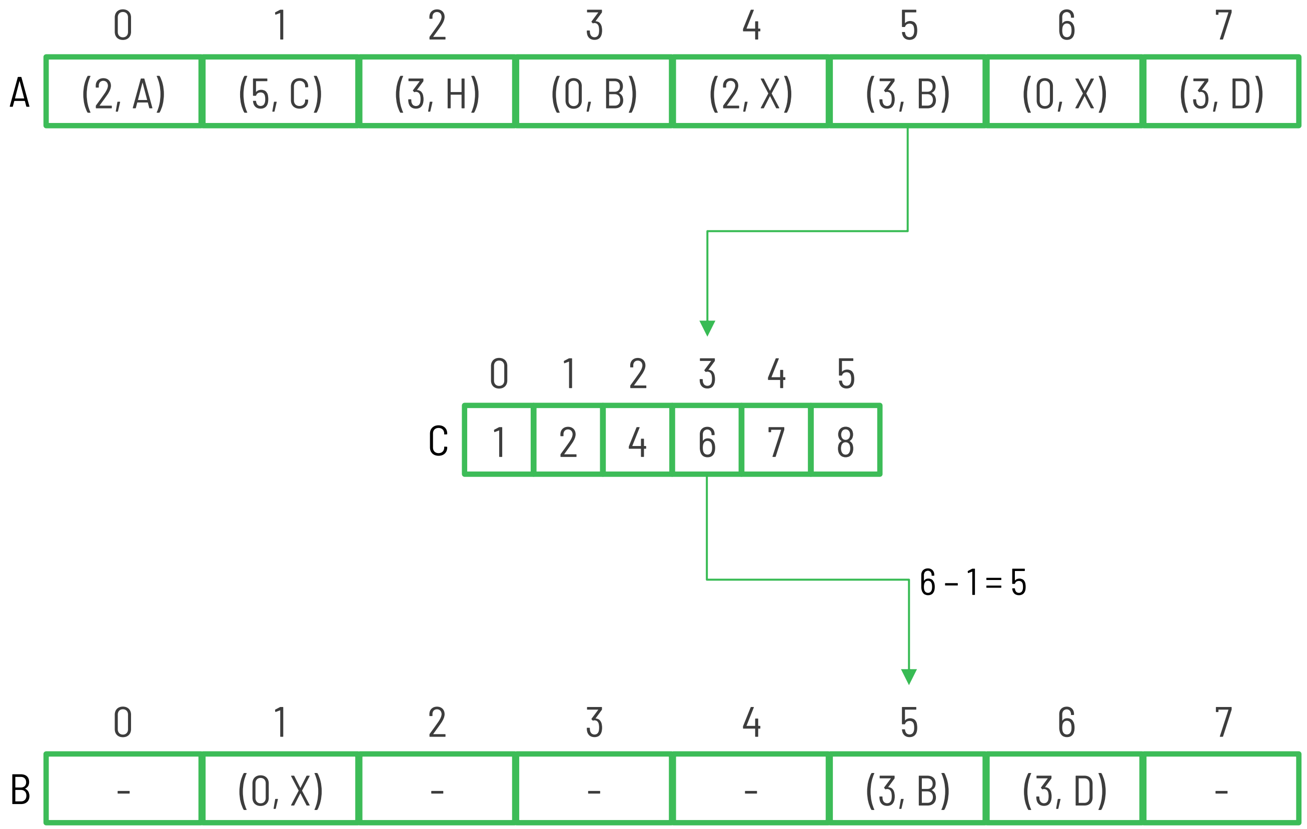
Step 4: Use values in C to determine the final index for each element in A.



Step 4: Use values in C to determine the final index for each element in A.

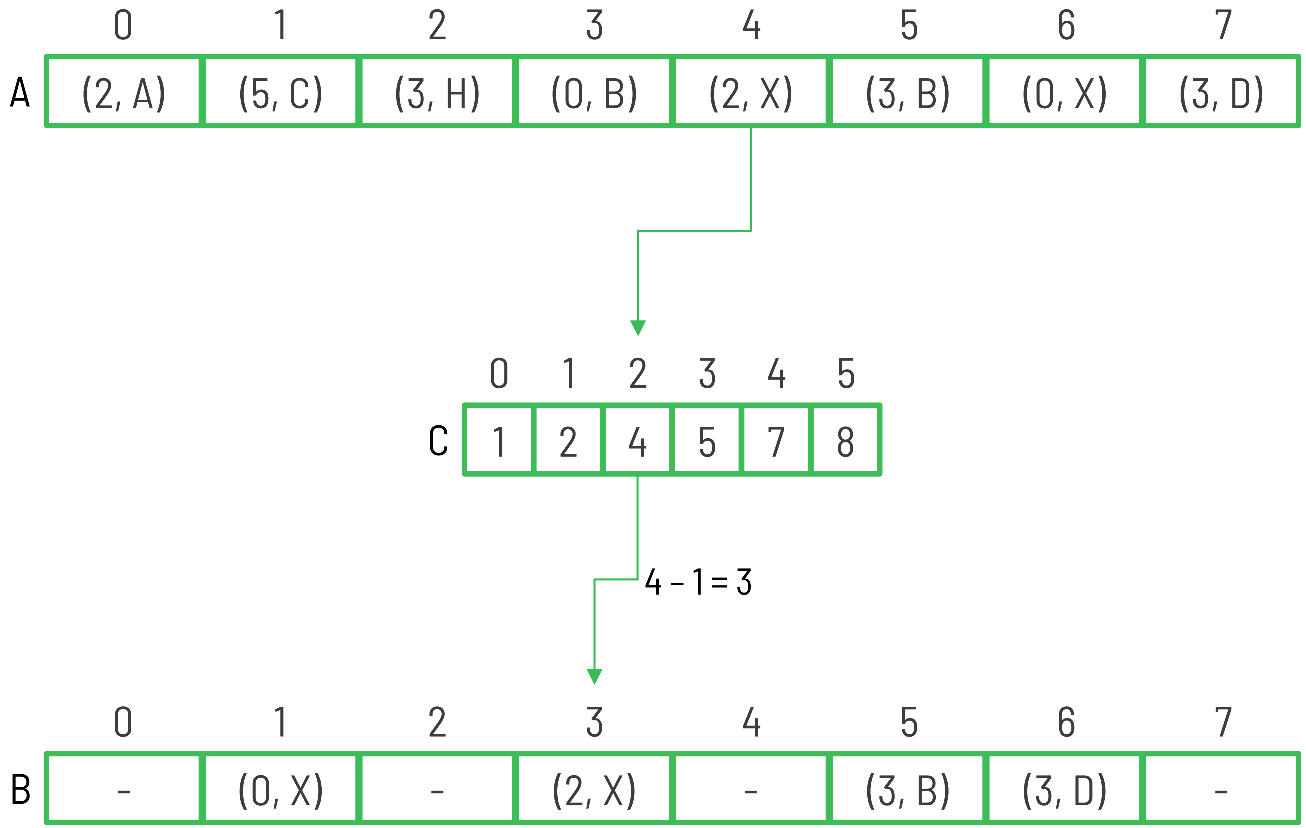


Step 4: Use values in C to determine the final index for each element in A.



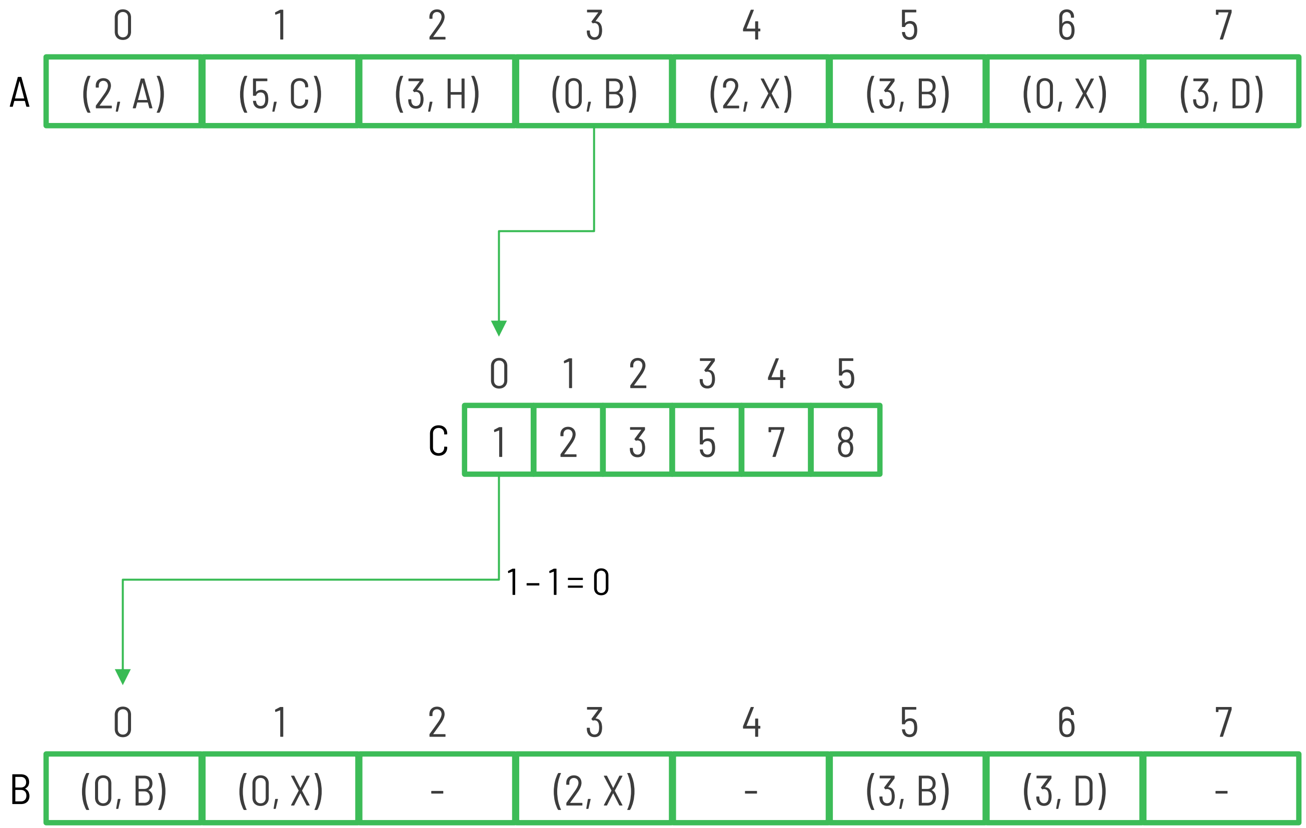
0 1 2 3 4 5 6 7
A
(2, A) (5, C) (3, H) (0, B) (2, X) (3, B) (0, X) (3, D)
0 1 2 3 4 5
C
1 2 4 6 7 8
6 – 1 = 5
0 1 2 3 4 5 6 7
B
- (0, X) - - - (3, B) (3, D) -

Step 4: Use values in C to determine the final index for each element in A.



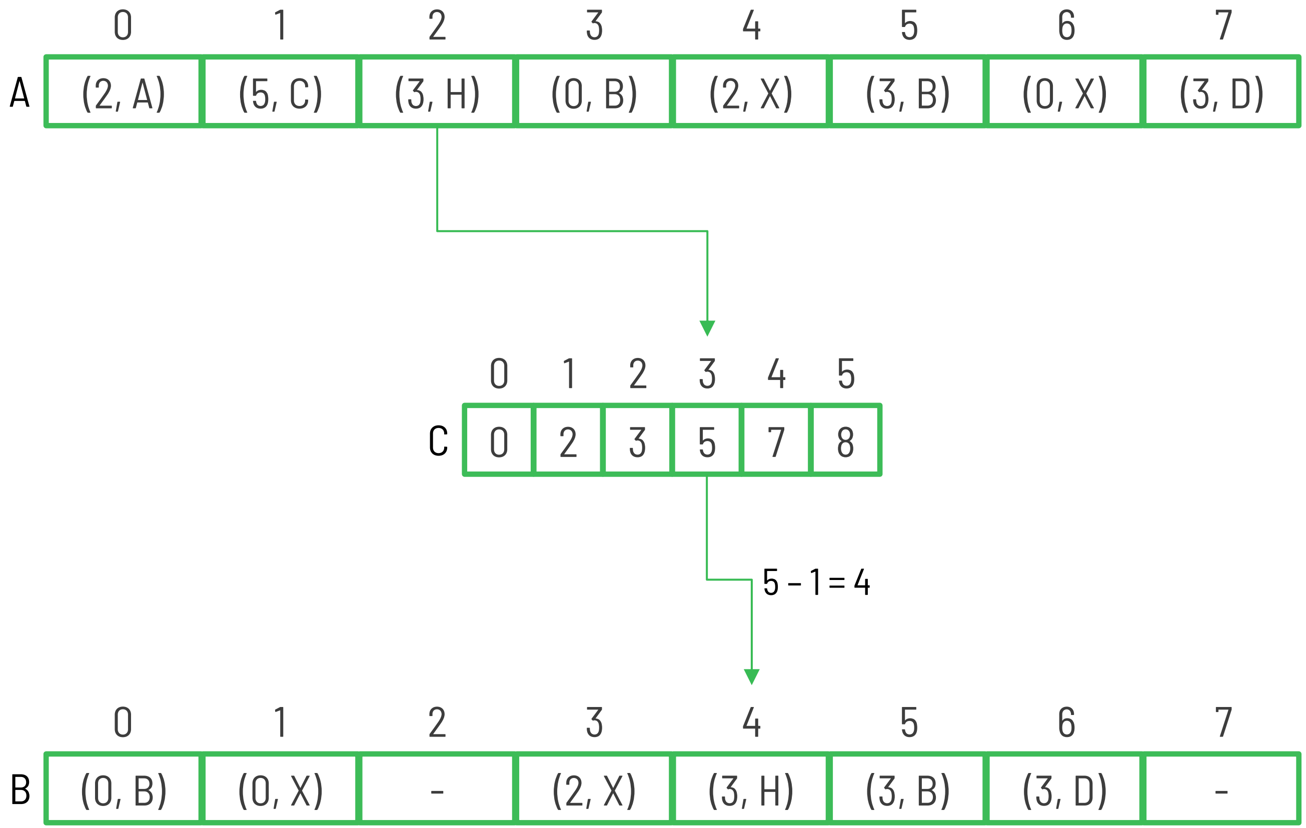
A
0 1 2 3 4 5 6 7
(2, A) (5, C) (3, H) (0, B) (2, X) (3, B) (0, X) (3, D)
C
0 1 2 3 4 5
1 2 4 5 7 8
4 – 1 = 3
B
0 1 2 3 4 5 6 7
- (0, X) - (2, X) - (3, B) (3, D) -

Step 4: Use values in C to determine the final index for each element in A.

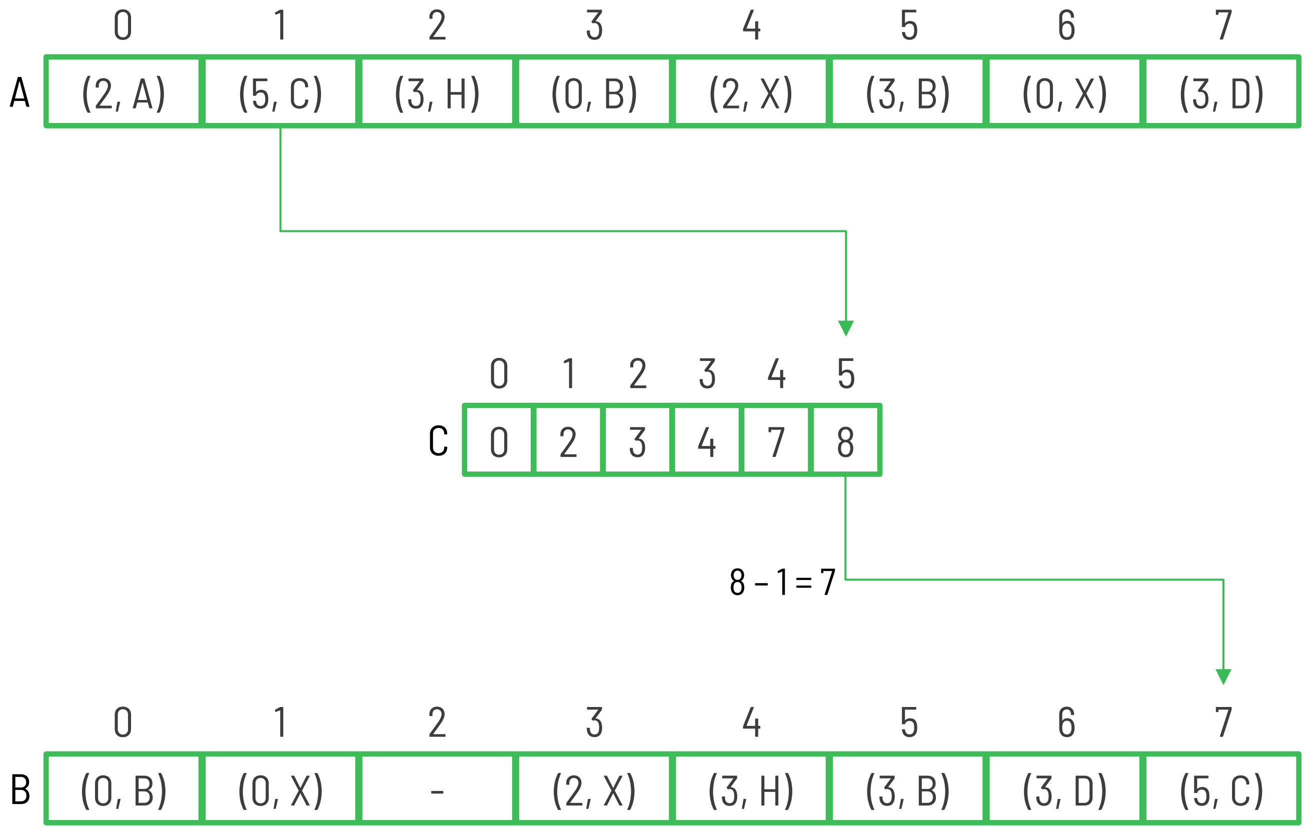


0 1 2 3 4 5 6 7
A
(2, A) (5, C) (3, H) (0, B) (2, X) (3, B) (0, X) (3, D)
0 1 2 3 4 5
C
1 2 3 5 7 8
1 – 1 = 0
0 1 2 3 4 5 6 7
B
(0, B) (0, X) - (2, X) - (3, B) (3, D) -

Step 4: Use values in C to determine the final index for each element in A.

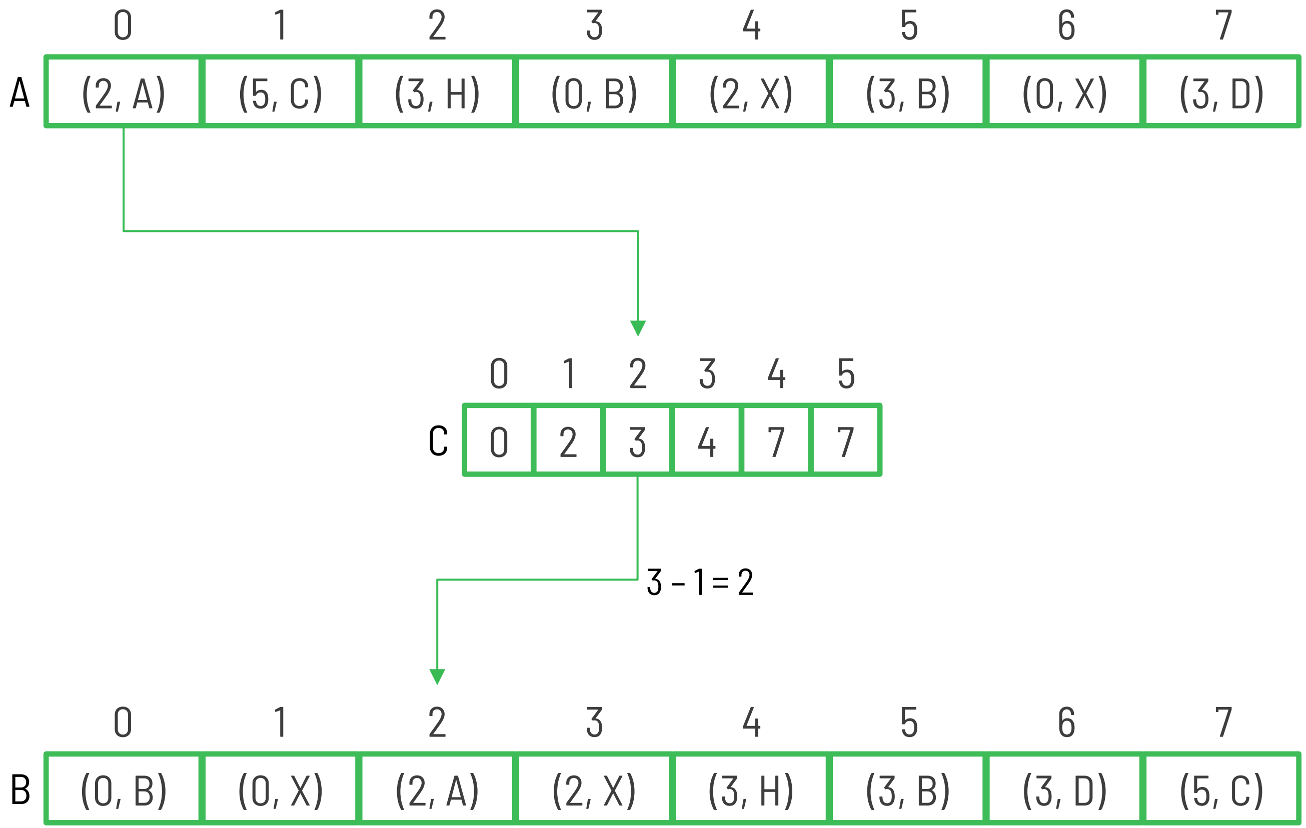


Step 4: Use values in C to determine the final index for each element in A.



A
0 1 2 3 4 5 6 7
(2, A) (5, C) (3, H) (0, B) (2, X) (3, B) (0, X) (3, D)
C
0 1 2 3 4 5
0 2 3 4 7 8
8 – 1 = 7
B
0 1 2 3 4 5 6 7
(0, B) (0, X) - (2, X) (3, H) (3, B) (3, D) (5, C)

Step 4: Use values in C to determine the final index for each element in A.



Runtime complexity: $O(n+k)$
Warning: Check your k

Space complexity: $\Theta(n+k)$

Stable? Yes, since we populate array B backwards.

Step 1: $O(k)$
Step 2: $O(n)$
Step 3: $O(k)$
Step 4: $O(n)$

```
algorithm countingsort(A:array, k:$\mathbb{Z}^+$)

  let C be an array of length k+1
  fill C with 0s

  let n be the size A

  for i from 0 to n-1 do
     C[A[i]] ← C[A[i]] + 1
  end for

  for i from 1 to k do
     C[i] ← C[i] + C[i-1]
  end for

  let B be an array of size n

  for i from n-1 to 0 by -1 do
     B[C[A[i]] - 1] ← A[i]
     C[A[i]] ← C[A[i]] - 1
  end for

  return B

end algorithm
```

Proof of Correctness (Insights)

Initial Counting Step: The algorithm starts by counting the occurrence of each item in the input array A and stores these counts in an auxiliary array C. For each item i, $C[i]$ accurately represents the total number of elements with item i in A. This step ensures that we have the exact count of each item.

Cumulative Count Adjustment: The next step transforms C into a cumulative count array. After this adjustment, for any item i, $C[i]$ represents the total number of elements with items less than or equal to i. This transformation is crucial because $C[i]$ now indicates the position just after the last occurrence of i in the sorted array B. It effectively tells us where a new occurrence of i should be placed in B, guaranteeing each element's correct position based on its item.

Placement and Stability: The placement step iterates over the input array A from last to first. For each element $A[i]$, it places $A[i]$ into the output array B at the position indicated by $C\left[A[i]\right] - 1$, and then decrements $C\left[A[i]\right]$. This decrementing step is critical: it updates $C\left[A[i]\right]$ to point to the next free position for a potential previous occurrence of the same item, ensuring stability. By iterating from last to first in A, we guarantee that elements with the same item are placed in B in the same order they appear in A, thus maintaining stability.

Proof of Correctness

Correct Final Position: After the cumulative count adjustment, $C[i]$ indicates the correct final position for the next element with item i (considering 0-based indexing). Placing each element $A[j]$ into $B\left[C\left[A[j]\right] - 1\right]$ and decrementing $C\left[A[j]\right]$ ensures that elements are placed in their correct position, preserving the sorted order.

Stability: By iterating backwards through the input array A and using a decrementing index from C for placement in B, we ensure that when two items have the same value, the one encountered later in the backward traversal (which was originally placed later in A) is placed later in B. This maintains the original relative order of equal items, proving the algorithm's stability.

Completeness: Since every item in A is placed into B exactly once, and C is correctly decremented to manage duplicate items, all items from A are accounted for in B, ensuring the output array is a complete and accurate sorted version of A.

02
Bucket Sort

Yes, use buckets!

Think About This

Longest word in an English dictionary*:
pneumonoultramicroscopicsilicovolcanoconiosis (45 characters).

Smallest word? There are some words made up of a single character.

Consider a list of n English words stored as Strings.

Think about an algorithm to sort the words by **length**. Assume the length of a String is $O(1)$.

The algorithm should take $O(n)$. Is that possible?

* There exist longer words, but those do not appear in major dictionaries.

Solution

Algorithm:

Define an array A of 45 linked lists.

For each word w in the input list: get its length l and insert w in list at $A[l-1]$.

Join the 45 lists. This results in all words sorted by length.

Analysis:

Traversing the input words: $O(n)$

Inserting w in list at $A[l-1]$: $O(1)$

Joining two lists: $O(1)$
All 45 joins in $O(1)$ (it's always 45)

Runtime complexity: $O(n)$

Bucket Sort

Bucket Sort assumes an input array A of n floating-point numbers ranging from 0 to 1 (exclusive). The original algorithm distributes the items of the array into n buckets (i.e., lists).

Each bucket is then sorted individually, either using a different sorting algorithm or by recursively applying the bucket sort algorithm.

It is particularly useful when the input is uniformly distributed over a range.

Bucket sort can be seen as a generalization of counting sort; while counting sort creates a bucket for each possible value, bucket sort creates a finite number of buckets for a range of values.

Bucket Sort

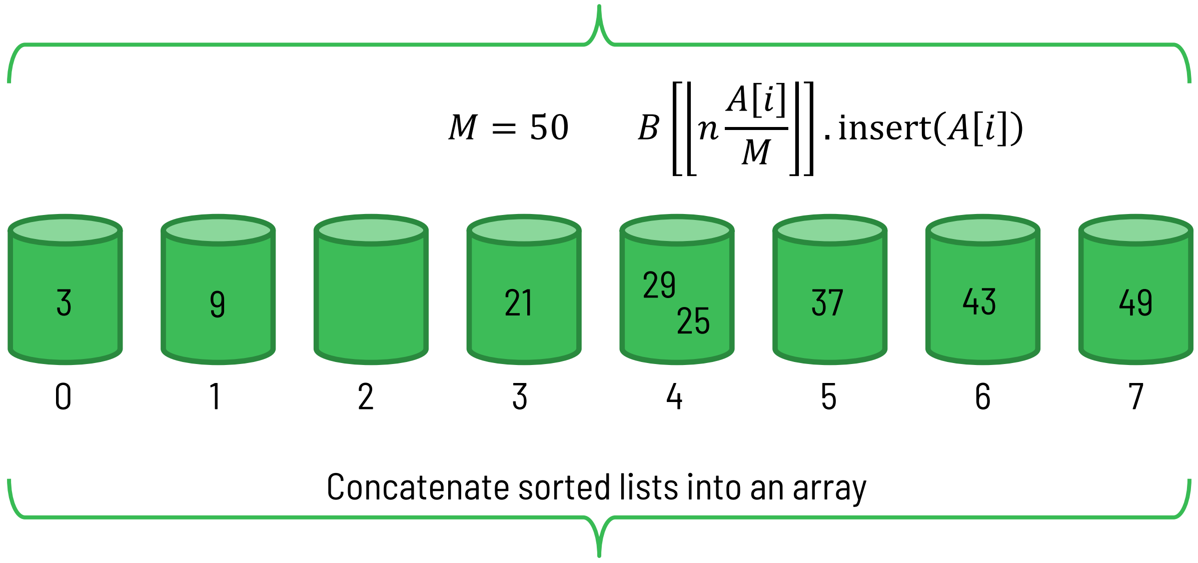
```
algorithm bucketsort(A:array) → array
   let n be the size of A
   let B be an array of n empty lists
   let M be the max key in A plus 1

   for i from 0 to n-1 do
      j ← floor(n * (A[i] / M))
      B[j].insert(A[i])
   end for

   for i from 0 to n-1 do
      sort the list B[i]
   end for

   concatenate the sorted lists into a
   single array and return it
end algorithm
```

Note: Bucket Sort assumes every $A[i]$ to be in range $[0, 1)$ instead of an integer number.

Bucket Sort Example

[29, 25, 3, 49, 9, 37, 21, 43]

$M = 50 \qquad B\left[\left\lfloor n\frac{A[i]}{M}\right\rfloor\right].\mathrm{insert}(A[i])$



Concatenate sorted lists into an array

[3, 9, 21, 25, 29, 37, 43, 49]

Note: Bucket Sort assumes every $A[i]$ to be in range $[0, 1)$ instead of an integer number.

Bucket Sort with k buckets

```
algorithm bucketsort(A:array, k:ℤ⁺) → array
   let n be the size of A
   let B be an array of k empty lists
   let M be the max key in A plus 1

   for i from 0 to n-1 do
      j ← floor(k * (A[i] / M))
      B[j].insert(A[i])
   end for

   for i from 0 to k-1 do
      sort the list B[i]
   end for

   concatenate the sorted lists into a
   single array and return it
end algorithm
```

Note: Bucket Sort assumes every $A[i]$ to be in range $[0, 1)$ instead of an integer number.

Bucket Sort Runtime

Runtime depends on: n (input size), k (number of buckets), how the elements are distributed among the buckets, and the method for sorting the buckets.

Worst case? $O(n^2)$ (e.g., all elements in a single bucket sorted with an awful sorting algorithm)

Best case? $O(n)$ (e.g., one element per bucket, all sorted in a single bucket)

Average case? $O\left(n + \frac{n^2}{k} + k\right)$

***One key per bucket?** $O(n + k)$

*One key per bucket ≠ One element per bucket

Proof of Correctness (Insights)

Correct Distribution to Buckets: Given an array A with n elements that are uniformly distributed across the range $[0,1)$, we distribute elements into n buckets, $B[0 \dots n-1]$, where each bucket $B[i]$ corresponds to a specific range of values. By design, if $A[j]$ is placed into bucket $B[i]$, then any element $A[k]$ placed into $B[i+1]$ must be greater than $A[j]$. This is ensured by the distribution step, where an element $A[j]$ is placed into bucket $B[\lfloor nA[j] \rfloor]$, mapping the element's value directly to its position in the sorted array, based on its proportion within the total range.

Correct Sorting within Buckets: Each bucket $B[i]$ contains elements from A that fall within the range i/n to $(i+1)/n$. Since the elements within each bucket $B[i]$ are independently sorted, we ensure that the relative order of elements within each bucket is correct according to the sorting criterion.

Stability within Buckets (if applicable): If the sorting algorithm used for individual buckets is stable, then relative order among equal elements is preserved within each bucket. This is particularly important when sorting complex objects that have been grouped by a specific key.

Proper Concatenation Preserves Sorted Order: Upon concatenation, the elements from buckets $B[0]$ to $B[n-1]$ are combined in order. Given that each bucket $B[i]$ was sorted independently, and buckets are concatenated in their index order, the final array is in non-descending order. This is because elements in bucket $B[i]$ are always less than or equal to elements in $B[i+1]$, and within each bucket, elements are in sorted order.

Proof of Correctness

Invariant 1: After the distribution step, for any two elements $A[j]$ and $A[k]$ where $j < k$, if $A[j]$ and $A[k]$ are in the same bucket, then $A[j]$ will be placed before $A[k]$ after sorting that bucket. If they are in different buckets, $A[j]$ will be in a bucket with a lower index than $A[k]$'s bucket.

Invariant 2: After sorting all buckets, the elements within each bucket are in ascending order.

Invariant 3: Concatenating the buckets in order results in a sorted sequence because the ranges of the buckets ensure that all elements in bucket $B[i]$ are less than those in bucket $B[i+1]$, and within each bucket, elements are sorted.

03

Radix Sort

Sort by digits

Radix Sort

Radix Sort is a **non-comparative** sorting algorithm that sorts integers (and other data types that can be represented as **integers**, such as strings) by processing **individual digits**.

It operates on the principle of sorting numbers first by their least significant digits and progressively moving towards the most significant digit. Radix Sort utilizes a stable sorting algorithm (like Counting Sort) as a subroutine to sort the elements based on each digit.

Radix Sort

```
algorithm radixsort(A:array, d:$\mathbb{Z}^+$)
   for i from 0 to d-1 do
      use a stable sort to sort A on digit i
   end for
end algorithm
```

Note: For the number 932, digit 0 is 2, digit 1 is 3, and digit 2 is 9.

Counting Sort by Digit

```
algorithm countingsortbydigit(A:array, d:ℤ≥0)
   let C be an array of length 10
   fill C with 0s
   let n be the size A
   D ← 10^d

   for i from 0 to n-1 do
      digit ← floor(A[i] / D) mod 10
      C[digit] ← C[digit] + 1
   end for

   for i from 1 to 9 do
      C[i] ← C[i] + C[i-1]
   end for

   let B be an array of size n

   for i from n-1 to 0 by -1 do
      digit ← floor(A[i] / D) mod 10
      B[C[digit] - 1] ← A[i]
      C[digit] ← C[digit] - 1
   end for

   return B
end algorithm
```

Note: For the number 932, digit 0 is 2, digit 1 is 3, and digit 2 is 9.

Example: radixsort([187, 383, 370, 146, 387, 427, 442, 94, 470, 320], 3)

Calling countingsortbydigit([187, 383, 370, 146, 387, 427, 442, 94, 470, 320], **0**)
returns [370, 470, 320, 442, 383, 94, 146, 187, 387, 427]

Calling countingsortbydigit([370, 470, 320, 442, 383, 94, 146, 187, 387, 427], **1**)
returns [320, 427, 442, 146, 370, 470, 383, 187, 387, 94]

Calling countingsortbydigit([320, 427, 442, 146, 370, 470, 383, 187, 387, 94], **2**)
returns [94, 146, 187, 320, 370, 383, 387, 427, 442, 470]

$$\left\lfloor \frac{94}{100} \right\rfloor \bmod 10 = 0$$

[94, 146, 187, 320, 370, 383, 387, 427, 442, 470]

Example: Sorting 3D tuples

Original sequence (3, 5, 2) (3, 1, 8) (8, 3, 4) (3, 1, 6)

i = 0 (3, 5, **2**) (3, 1, **8**) (8, 3, **4**) (3, 1, **6**)

↓

i = 1 (3, **5**, 2) (8, **3**, 4) (3, **1**, 6) (3, **1**, 8)

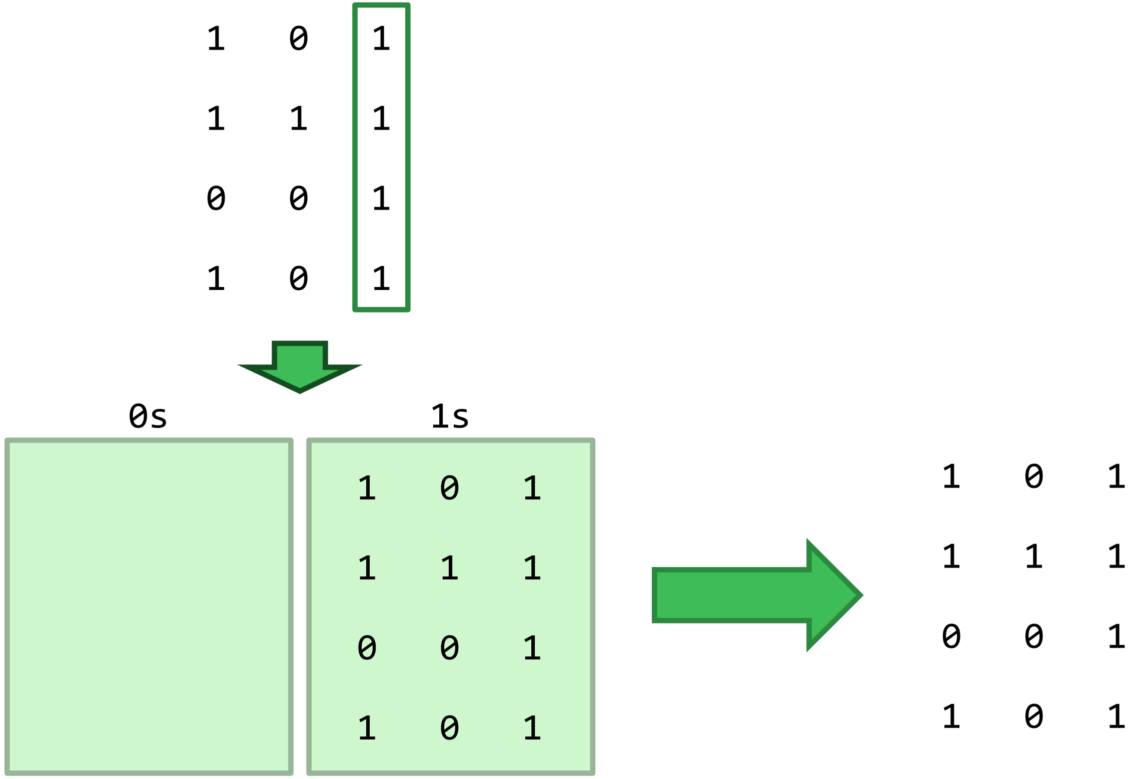
↓

i = 2 (**3**, 1, 6) (**3**, 1, 8) (**8**, 3, 4) (**3**, 5, 2)

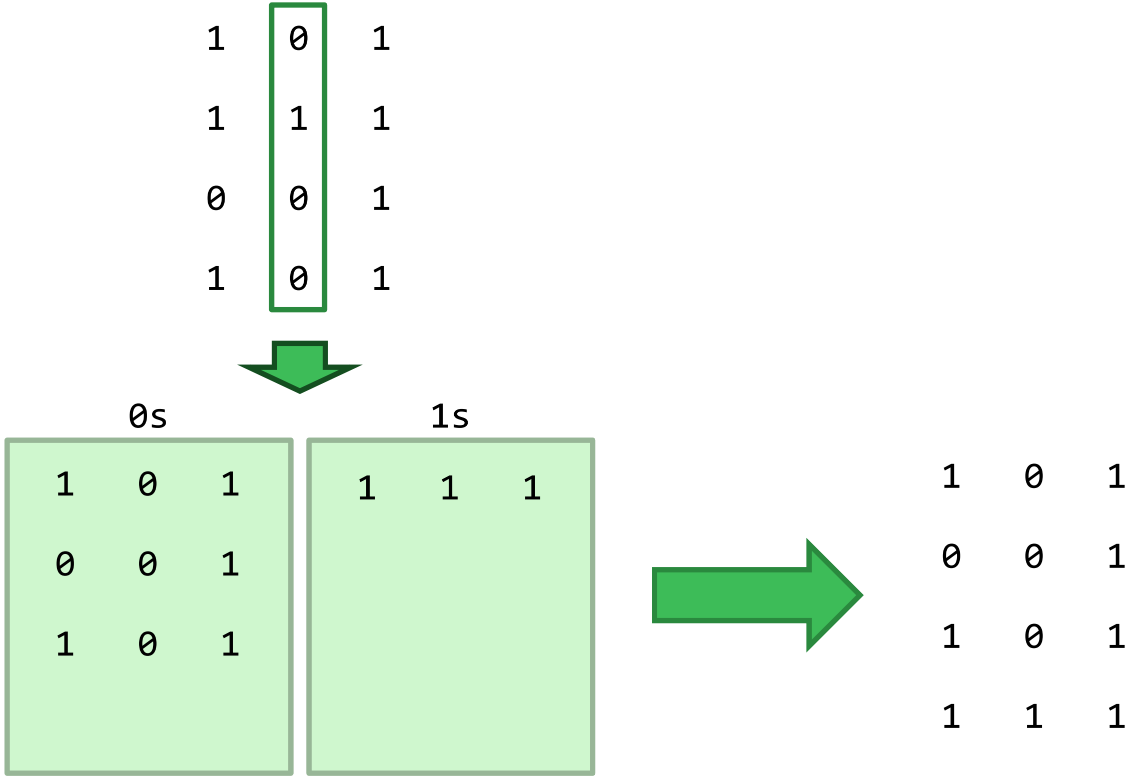
↓

(3, 1, 6) (3, 1, 8) (3, 5, 2) (8, 3, 4)

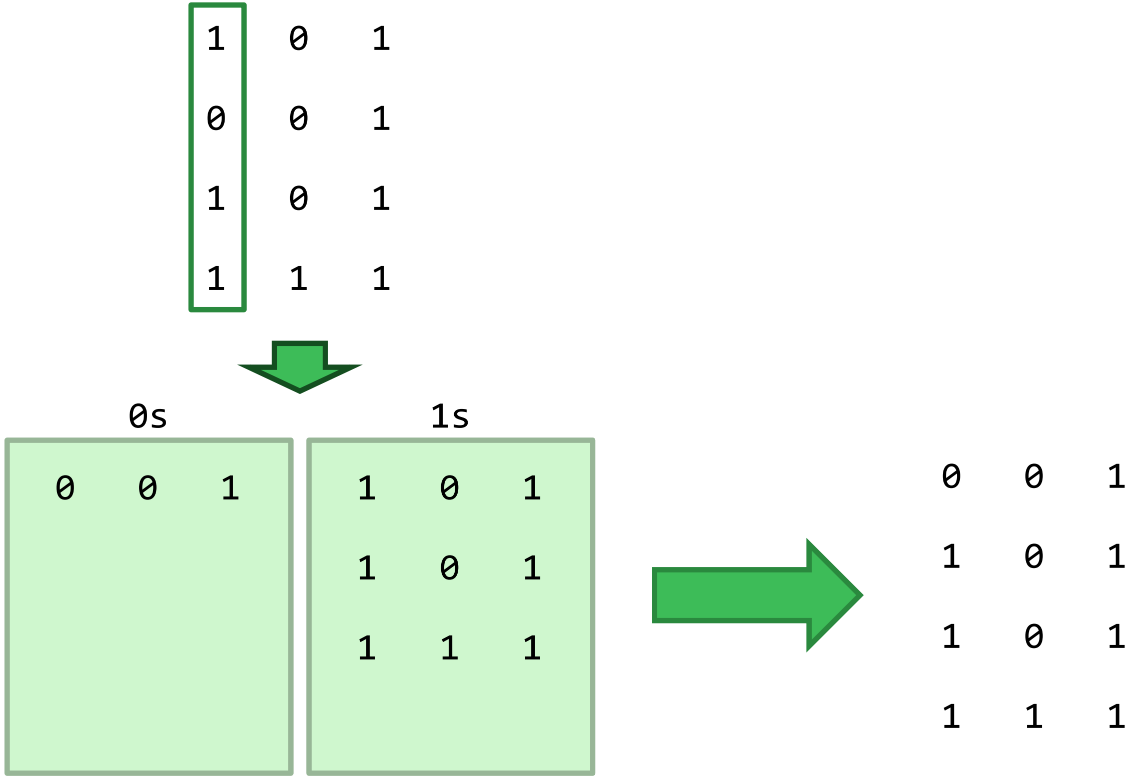
Example: Sorting binary numbers using Bucket Sort (2 buckets, one for 0s and one for 1s) as the stable sort



Example: Sorting binary numbers using Bucket Sort (2 buckets, one for 0s and one for 1s) as the stable sort



Example: Sorting binary numbers using Bucket Sort (2 buckets, one for 0s and one for 1s) as the stable sort



Radix Sort for Binary Numbers

Input: Sequence of n b-bit integers (e.g., $X = x_{b-1} \dots x_1 x_0$).

Treat each integer like a tuple with b dimensions and sort with radix sort.

The key range (for buckets) is [0, 1]. So, $k = 2$.

Runtime: $O\big(b(n+2)\big) = O(n)$ if b is constant.

Conclusion: We can sort a sequence of 32-bit integers in linear time.

Radix Sort Analysis

Time Complexity: $O(nd)$, where n is the number of elements in the input array, and d is the maximum number of digits in the largest number. This efficiency holds because the algorithm iterates over each digit of each number and uses a linear-time sorting algorithm (e.g., Counting Sort) for sorting by digits.

Space Complexity: $O(n + k)$, where k is the radix or base of the number system used. In the case of decimal numbers, $k = 10$. The space complexity accounts for the storage needed for the auxiliary array B and the count array C in Counting Sort. The space complexity changes if used a different stable sort.

Stability: Radix Sort is stable if the sorting algorithm used for sorting digits is stable. This stability is crucial for ensuring that the relative order of numbers with the same digits in earlier passes is preserved in later passes.

Applicability: Radix Sort is most effective for sorting integers or strings where the length of the numbers (in terms of digits) or the strings is relatively uniform. Its performance can significantly surpass comparison-based sorting algorithms for large datasets with a relatively small key space.

Radix Sort Correctness (Insights)

Stable Sorting of Digits: Radix Sort processes numbers by sorting them based on their digits, starting from the **least significant digit (LSD)** and moving towards the **most significant digit (MSD)**. The correctness of Radix Sort is contingent upon the stability of the digit sorting algorithm used at each step—meaning that if two elements have the same digit in the current position being sorted, their relative order is preserved from the input array to the output array for that digit-sorting phase.

Correct Ordering by Significance of Digits: By sorting numbers based on their digits from least significant to most significant, Radix Sort ensures that at any stage of the sorting process, the sorted order of numbers reflects their order considering all previously sorted digits. This approach builds up the correct overall order incrementally, digit by digit.

Radix Sort Correctness

Step 1: Stability Ensures Correct Relative Order

Assume that we use a stable sorting algorithm (e.g., Counting Sort) to sort the elements based on their current digit.

Given two elements x and y in the array such that $x < y$ and they have the same digit in the position currently being sorted, stability ensures that x will still precede y after the sorting of this digit is complete.

This preservation of relative order is crucial when sorting by subsequent digits. If x and y have the same higher-order digits but differ in a lower-order digit, sorting by the lower-order digit correctly determines their relative order, and this order will not be disturbed by subsequent sorts on higher-order digits due to stability.

Radix Sort Correctness (cont.)

Step 2: Correct Ordering by Significance of Digits

By starting from the least significant digit and moving towards the most significant digit, each stage of sorting builds upon the correctly sorted order of the previous stages.

After sorting by the least significant digit, numbers are grouped by that digit, with ties (numbers with the same least significant digit) still in their original order due to stability.

Sorting by the next digit (and each digit thereafter) rearranges the numbers so that within each group of the newly sorted digit, the groups formed by the previous digit's sort are preserved and correctly ordered. This is because the sorting algorithm is stable and because any two numbers with the same digit in the current position have already been correctly ordered relative to each other based on their lower-order digits.

After the final digit has been sorted, all numbers are correctly ordered because their relative order reflects the correct precedence of all their digits.

Radix Sort Correctness (cont.)

Invariant: After sorting based on digit i, the array is correctly sorted considering only the i least significant digits. This invariant holds for each digit-sorting phase.

Completion: When the final digit-sorting phase is complete (after sorting by the most significant digit), the array is sorted considering all digits. Since numbers with differing digits are correctly ordered by the significance of their differing digit, and numbers with identical digits are correctly ordered by the stability of the sort, the entire array is sorted.

Done!

Do you have any questions?