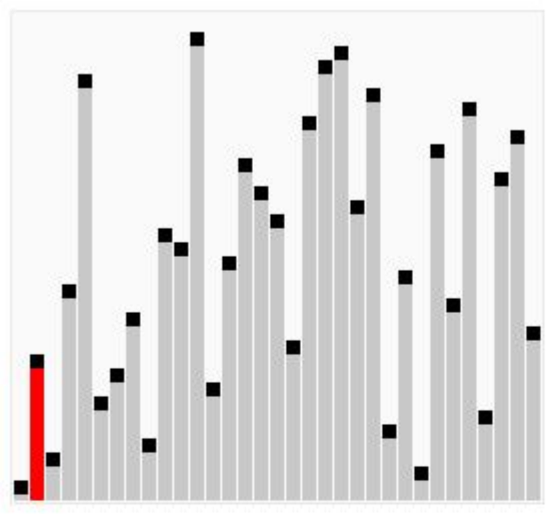
Sort an Array

Wenbo



Data types

Double: 1.35

Integer: 1

Array

Array: my_arr = [1, 3, 5, 7, 9]

Index starts from 1 in Julia!

My_arr[3] = 5

My_arr[1.3] = Error!

For loop

My_arr = [0, 0, 0, 0, 0]

My_arr[1] = 1

My_arr[1] = 2

My_arr[1] = 3

My_arr[1] = 4

My_arr[1] = 5

My_arr = [1, 2, 3, 4, 5]

For loop

```
My_arr = [0, 0, 0, 0, 0]

My_arr[1] = 1

My_arr[1] = 2

My_arr[1] = 3

My_arr[1] = 4

My_arr[1] = 5
```

```
My_arr = [0, 0, 0, 0, 0]

for i = 1:6

        My_arr[i] = i

end
```

My_arr = [1, 2, 3, 4, 5]

Big O notation

For input size = n,

O(n) means the amount of work = n, 2n, 3n, 4n …

O(1) means the amount of work is constant.

O(n^2) means work = n^2, 2n^2, 3n^2 …

Can be used to present **time** and **space** complexity.

In-Place functions

Functions with O(1) space complexity, with an exclamation mark at the end, for example:

Swap!(arr, i, j) swaps the ‘i’-th element and ‘j’-th element of array ‘arr’

My_arr = [1, 2, 3, 4, 5]

Swap!(My_arr, 2, 4)

—> My_arr = [1, 4, 3, 2, 5]

Given an array of integers `nums`, sort the array in ascending order and return it.

You must solve the problem **without using any built-in** functions in `O(nlog(n))` time complexity and with the smallest space complexity possible.

Example 1:

```
Input: nums = [5,2,3,1]
Output: [1,2,3,5]
Explanation: After sorting the array, the positions of some numbers are not changed (for
example, 2 and 3), while the positions of other numbers are changed (for example, 1 and 5).
```

Example 2:

```
Input: nums = [5,1,1,2,0,0]
Output: [0,0,1,1,2,5]
Explanation: Note that the values of nums are not necessairly unique.
```

Constraints:

- $1 <= nums.length <= 5 * 10^4$
- $-5 * 10^4 <= nums[i] <= 5 * 10^4$

- Selection Sort
- Bubble Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Heap Sort
- Counting Sort
- Radix Sort
- Bucket Sort
- Bingo Sort Algorithm
- ShellSort
- TimSort
- Comb Sort
- Pigeonhole Sort
- Cycle Sort
- Cocktail Sort
- Strand Sort
- Bitonic Sort
- Pancake sorting
- BogoSort or Permutation Sort
- Gnome Sort
- Sleep Sort – The King of Laziness
- Structure Sorting in C++
- Stooge Sort
- Tag Sort (To get both sorted and original)
- Tree Sort
- Odd-Even Sort / Brick Sort
- 3-way Merge Sort

Double Loop Sort (Bubble Sort)

Pair-wise comparison

Num comparisons: $C_n^2 = \dfrac{n \times (n-1)}{2 \times 1} = O(n^2)$

Time = O(n^2)

Space = O(1) for in-place implementation

i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	**9**
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	**8**	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	**7**		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	**5**			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	**4**				
	1	1	2	3					
i = 6	0	1	2	**3**					
		1	**2**						

Merge Sort

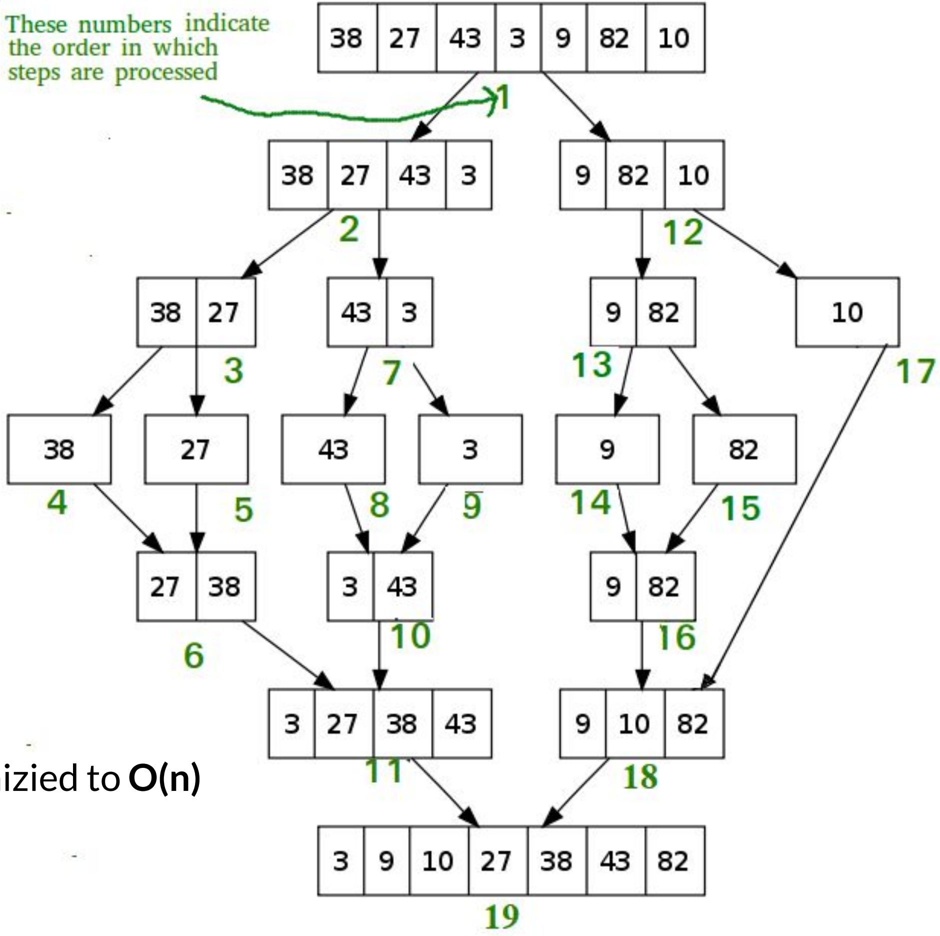
Two phases: Divide and Conquer

Depth for each phase = log(n)

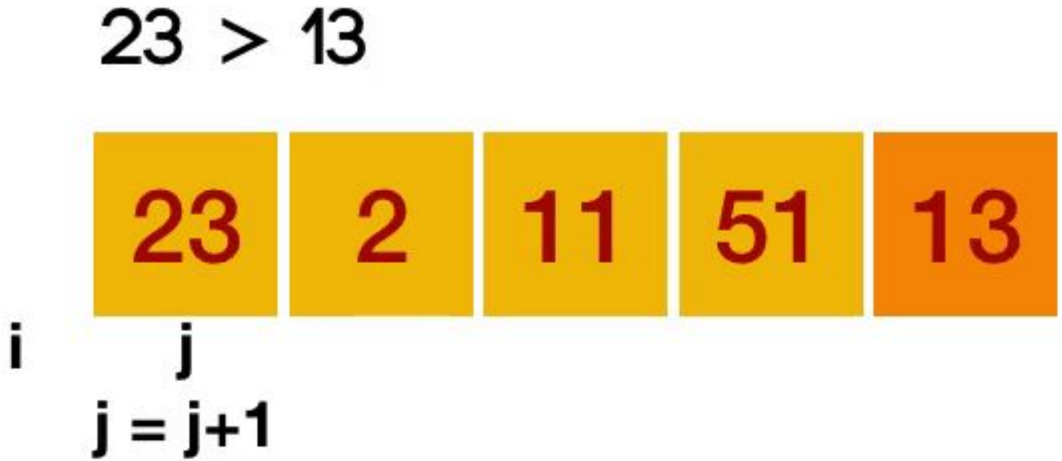
Work for each level = n

So **Time = O(n log(n))**

Space = O(n log(n)) but can be optimizied to **O(n)**



Quick Sort (Still Divide and Conquer)



aaaaaaaaaa|xxxxxxxxxxxxxxxx|bbbbbbbbbb

i j

(<=pivot) (undiscovered) (>pivot)

aaaaaaaaaaaaaaaaaaa|bbbbbbbbbbbbbbbb

j i

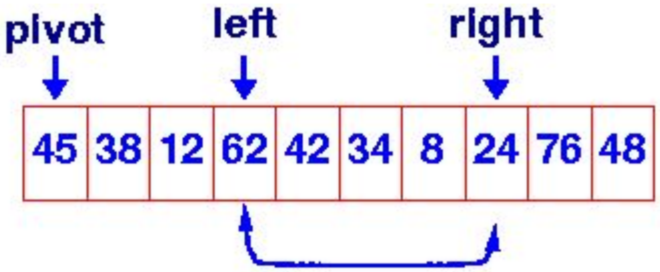
(<=pivot) (>pivot)

Quick Sort (Still Divide and Conquer)

Default: choose the left element as the pivot

Average time complexity: O(n logn)

Space: O(1) by in-place implementation



Quick Sort (Still Divide and Conquer)

Average time complexity: O(n logn)

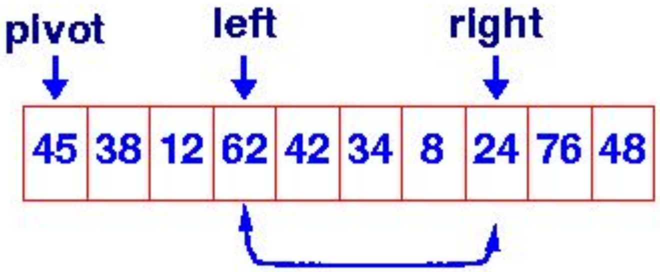
Worst case time complexity: O(n^2)

Space: O(1) by in-place implementation

Worst case, Sort [1,2,3,4,5,....n] in ascending order.

1. Pivot = 1, work = n
2. Pivot = 2, work = n-1
3. ...
4. Pivot = n-2, work = 1

Total work = sum(1:n) = O(n^2)



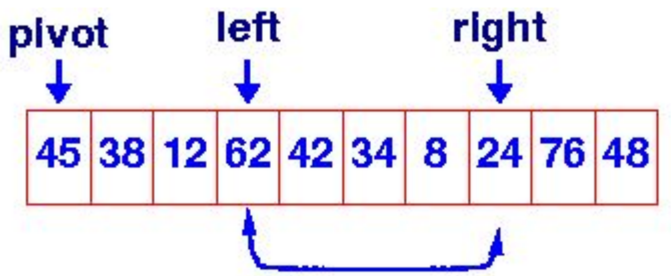
Quick Sort (Still Divide and Conquer)

Average time complexity: O(n logn)

Worst case time complexity: O(n^2)

Space: O(1) by in-place implementation

Worst case, Sort [1,2,3,4,5,....n] in ascending order.



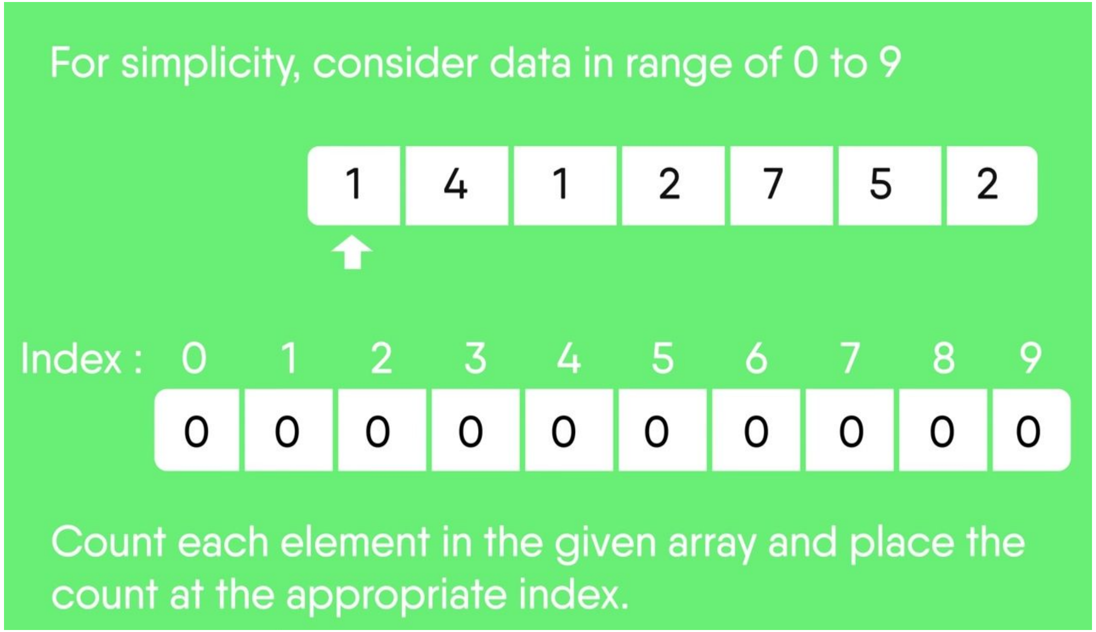
1. Pivot = 1, work = n
2. Pivot = 2, work = n-1
3. ...
4. Pivot = n-2, work = 1

Total work = sum(1:n) = O(n^2)

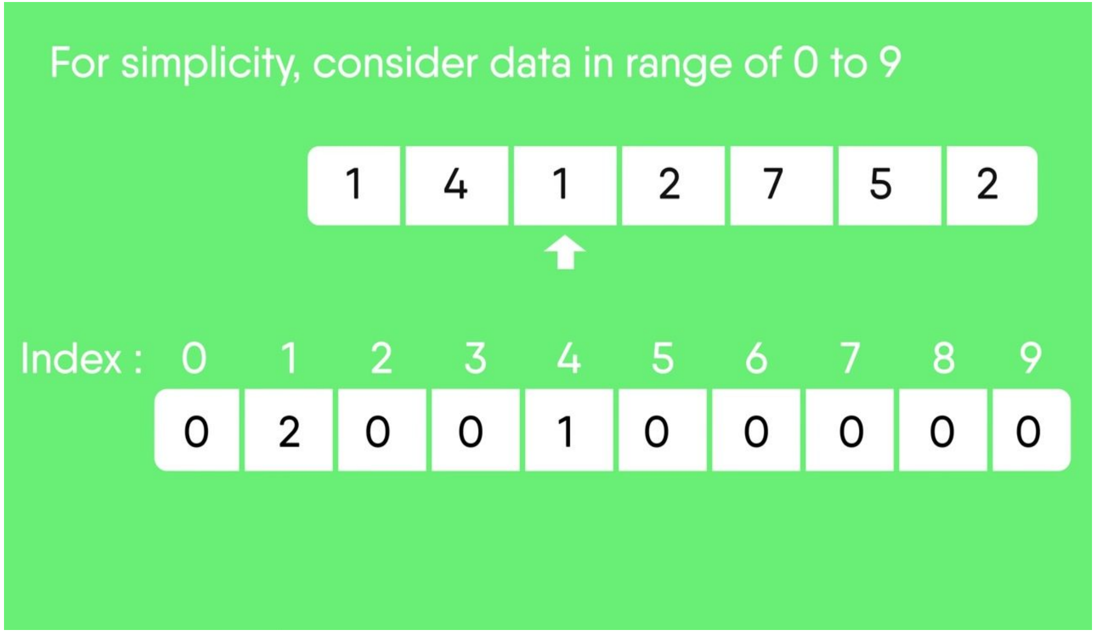
Choose random pivot

Extra: Counting Sort

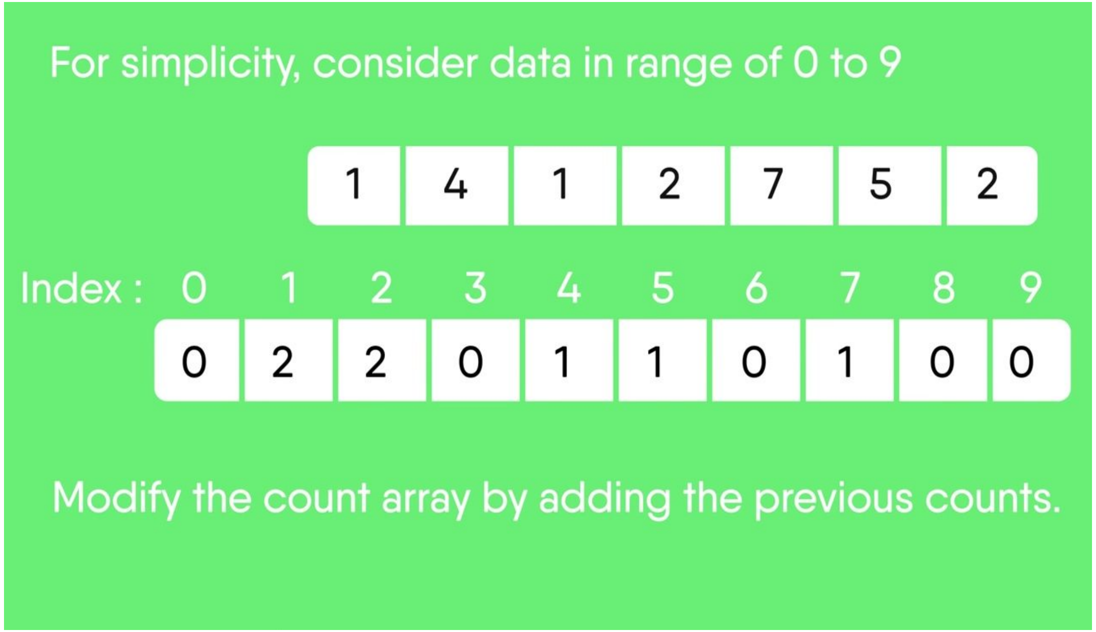
1st pass: Counting



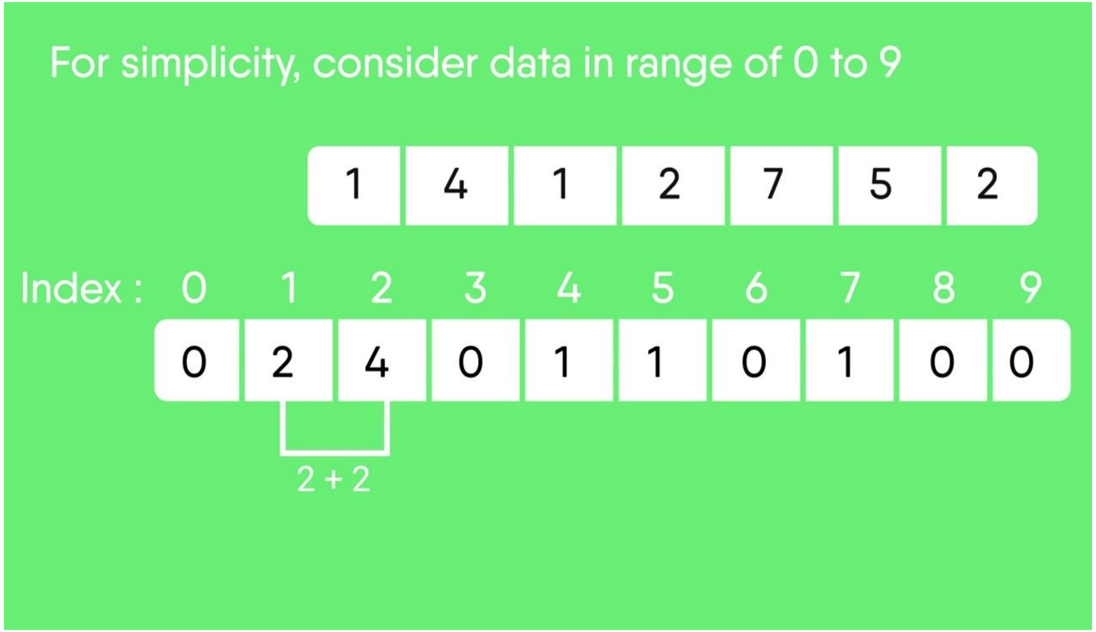
1st pass: Counting



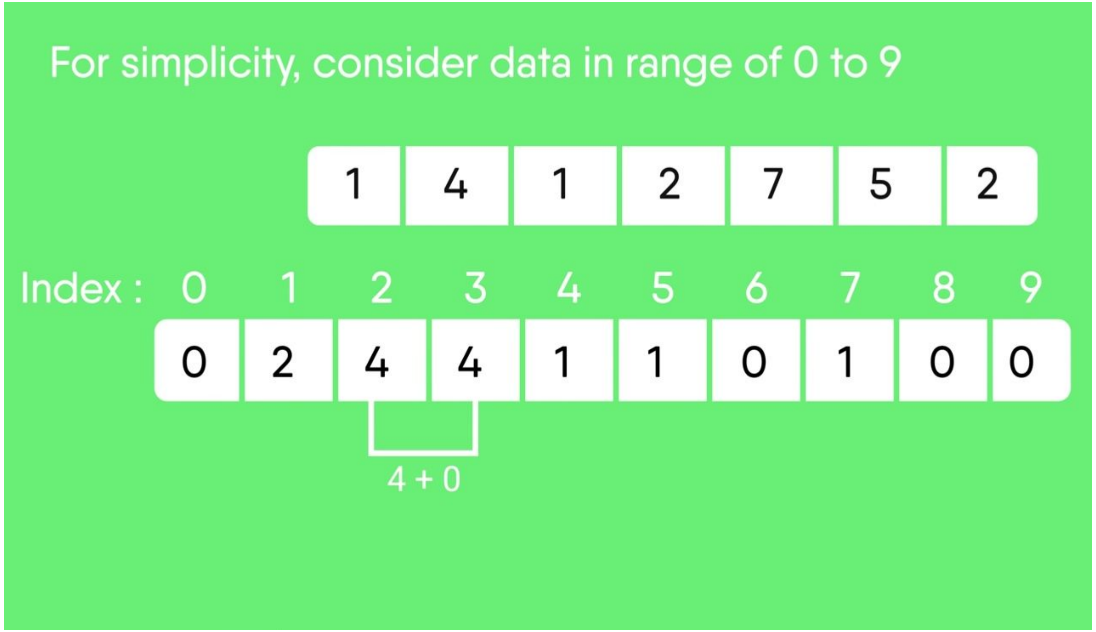
1st pass: Counting



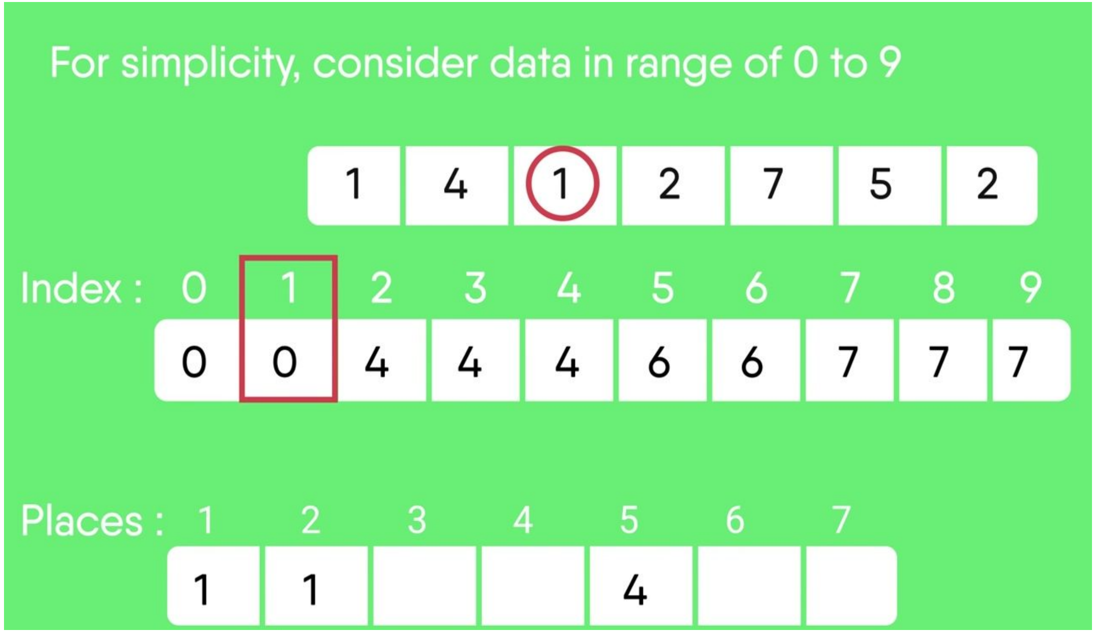
2nd pass: Ranking



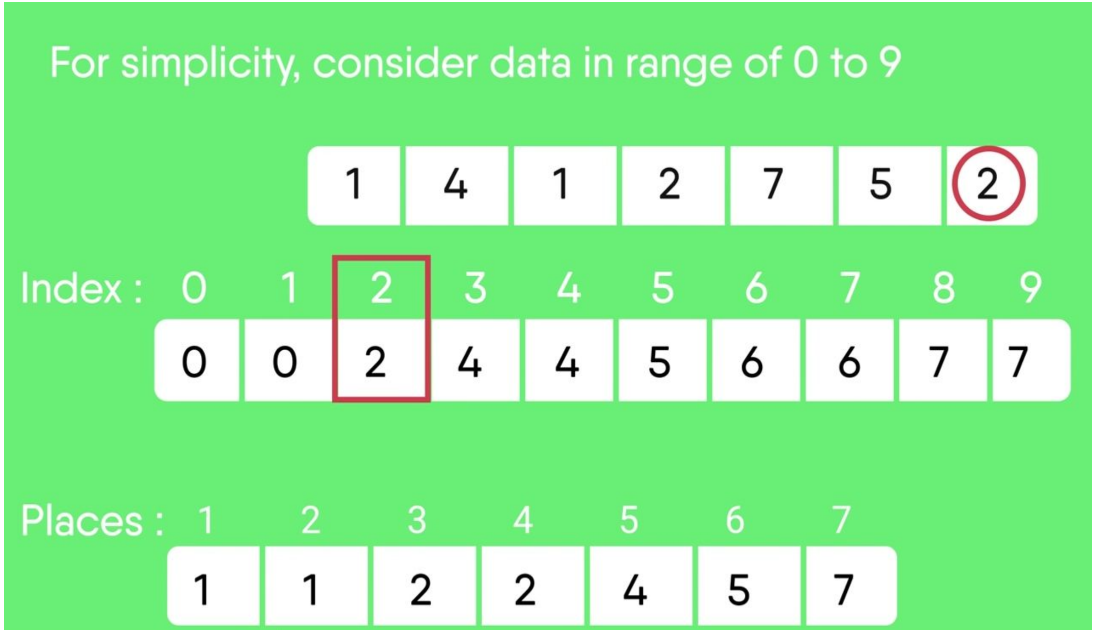
2nd pass: Ranking



3nd pass: Build the output



3nd pass: Build the output



Extra: Counting Sort

Fastest but not space efficient(Heavy load to be deployed on server)

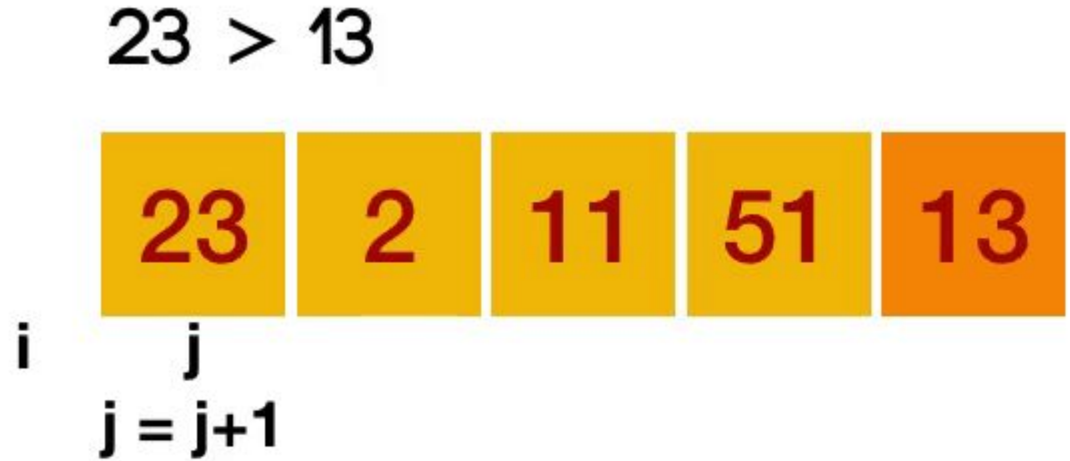
Time: O(n+K)

Space: O(n+K)

Trade off between time and space

Follow Up: A better Quick Sort?

Follow Up: A better Quick Sort?



aaaaaaaaa|**ppppppppp**|xxxxxxxxxxxxxx|bbbbbbbbbb

i j k

(<pivot) (==pivot) (undiscovered) (>pivot)

aaaaaaaaa|**ppppppppp**|bbbbbbbbbb

jk i

(<pivot) (==pivot) (>pivot)

Next Week: Pow(x, n)

Constraints:

- `-100.0 < x < 100.0`
- $-2^{31} <= n <= 2^{31}-1$
- `n` is an integer.
- $-10^4 <= x^n <= 10^4$

Implement pow(x, n), which calculates `x` raised to the power `n` (i.e., x^n).

Example 1:

```
Input: x = 2.00000, n = 10
Output: 1024.00000
```

Example 2:

```
Input: x = 2.10000, n = 3
Output: 9.26100
```

Example 3:

```
Input: x = 2.00000, n = -2
Output: 0.25000
Explanation: 2^-2 = 1/2^2 = 1/4 = 0.25
```