COUNTING SORT

Let's recall all the sorting algorithms we have learnt till now...

- Bubble Sort - $O(n^2)$
- Selection Sort - $O(n^2)$
- Insertion Sort - $O(n^2)$
- Merge Sort - O(nlogn)
- Quick sort - O(nlogn)

Can we do better?

Notice that in all the sorting methods we have studied before, we compare the values with each other, and accordingly perform certain operations (swap, insert, etc) to sort the array.

Also note that in all of them, the best (average case) time complexity possible is O(nlogn).

But is a comparison-based algorithm the only way to sort an array? And is there any algorithm with which we can perform sorting in linear time? (Yes, sometimes)

What is Counting Sort?

It is stable, non comparison based, not in-place sorting algorithm. It was introduced by Harold Seward in 1954.

In counting sort, we count the number of elements with distinct key values, and store the counts in an auxiliary count array. Then we apply prefix sum to find the positions of the elements in the sorted array, and accordingly put each element in its correct position.

Here's a visual representation of counting sort

Assumptions

- All elements must be small positive integers (no negative numbers, no floating point numbers) and duplicate values are present

- If the range is [0, k], k must be O(n)

- Ideally, k should be much smaller than n
  It does not make much sense to use counting sort in an array of 10 elements where the range is [0, 10000] because here
  $n+k >>> n\log n$

Algorithm

1. Find the max element in the array (k)
2. Create a count array of size k+1 and initialise it with zeroes
3. Traverse through the input array and each time an element (let's say, x) is encountered, increment the xth index of the count array
4. Find the prefix sum of the count array by adding the value of each index to the value of the previous index
   This cumulative value denotes the position of the element in the sorted array

Algorithm

5. Traverse the input array from right to left
6. As an element is encountered, go to its corresponding index in the count array and decrement it (this gives us the indexof the element, as indexing starts from 0 and the position starts from 1)
7. Place the element in its correct index in the output array
8. Continue till the beginning of the input array is reached
9. Copy the output array into the input array

Sample Code

```c
void countingSort(int arr[], int n)
{
    // find the max element in the array
    // this max element decides the range (0 <= arr[i] <= max)

    int max = -1;

    for (int i = 0; i < n; i++)
        if (arr[i] > max)
            max = arr[i];

    // initialise count array of size max+1 with zeroes

    int *count = (int*) calloc (max+1, sizeof(int));

    // traverse through the given array and increment the count of each encountered element

    for (int i = 0; i < n; i++)
        count[arr[i]]++;

    // cumulatively add the the previous element's count to each element's count
    // subtracting 1 from this value gives us the highest index of each key value in the sorted array

    for (int i = 1; i < max+1; i++)
        count[i] += count[i-1];
```

```
    // traverse the input array from right to left and place each element at its correct position

    int *output = (int*) malloc (n * sizeof(int));

    for (int i = n-1; i >= 0; i--)
    {
        count[arr[i]]--; // we subtract 1 to make it the index, as the count is greater than the index
        by 1
        output[count[arr[i]]] = arr[i]; // essentially output[index] = element
    }

    // copy the output array into the original array

    for (int i = 0; i < n; i++)
        arr[i] = output[i];


    // free the dynamically allocated memory

    free(count);
    free(output);

}
```

```c
#include <stdio.h>
#include <stdlib.h>

void display(int arr[], int n);

void countingSort(int arr[], int n);

int main(void)
{
    int arr[] = {7, 2, 6, 3, 5, 1, 8, 0, 9, 1, 4, 0, 6, 2, 5, 8, 7, 9, 3, 4};
    int n = sizeof(arr) / sizeof(arr[0]);

    printf("Array before sorting: ");
    display(arr, n);

    countingSort(arr, n);

    printf("Array after sorting: ");
    display(arr, n);

}
```

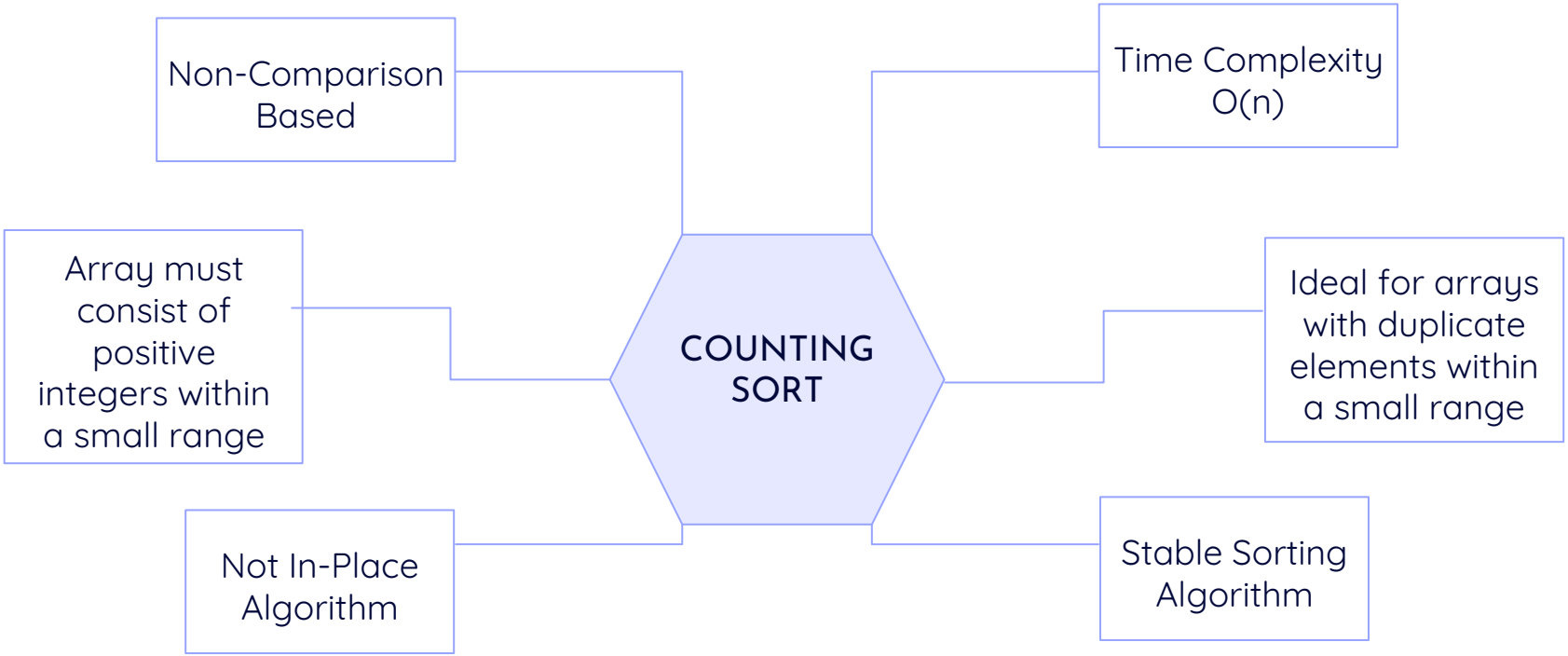
```
Array before sorting: 7 2 6 3 5 1 8 0 9 1 4 0 6 2 5 8 7 9 3 4
Array after sorting: 0 0 1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9
```

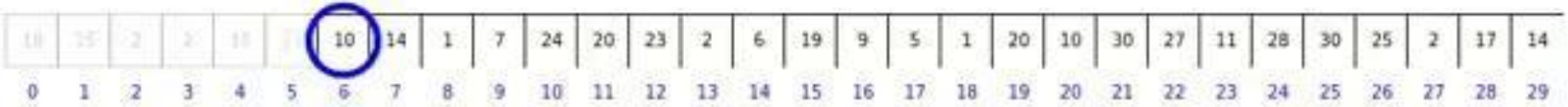
Example Comparison

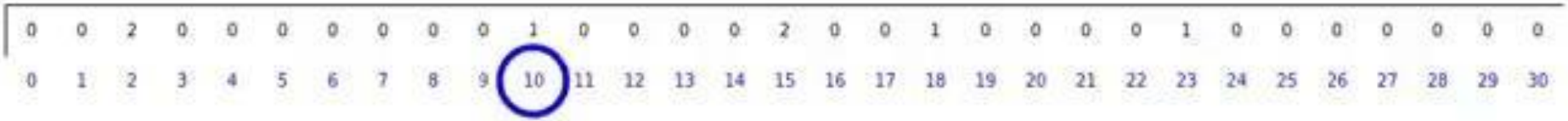
Taking an array of 1000000 random numbers in the range [0, 100]

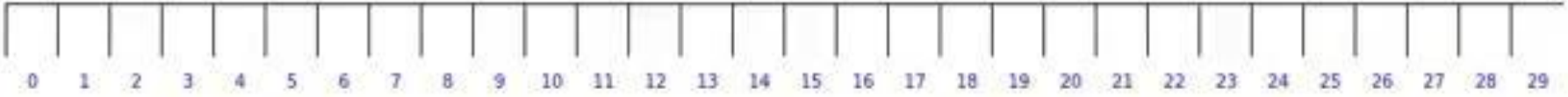
```
Time taken by quick sort: 4.251629114151001

Time taken by counting sort: 0.05409026145935058
```

CHARACTERISTICS OF COUNTING SORT




10 14 1 7 24 20 23 2 6 19 9 5 1 20 10 30 27 11 28 30 25 2 17 14
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29


0 0 2 0 0 0 0 0 0 0 1 0 0 0 0 2 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29

Analysis of Counting Sort

Let us look what were we upto

1. Counting :

In the first step of counting sort, we iterate over the input array to count the frequency of each element.
This takes **O(n) time.**

2. Prefix Sum :

Next, we create a prefix sum array of the frequency array. This takes **O(k) time,** where **k** is the maximum element in the input array.

3. Rearranging :

Finally, we iterate over the input array again, and for each element, we find its position in the output array using the prefix sum array. This takes **O(n) time.**

4. Total Time Complexity :

Adding up the time complexities of the three steps above, we get a total time complexity of **O(n+k),** where **n** is the number of elements in the input array and **k** is the maximum element in the input array.

COMPARISON OF SORTING ALGORITHMS

Name	Best Case	Average Case	Worst Case	Memory Used	Stable	In-Place
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	1	Yes	Yes
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	1	No	Yes
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	1	Yes	Yes
Merge Sort	O(nlogn)	O(nlogn)	O(nlogn)	n	Yes	No
Quick Sort	O(nlogn)	O(nlogn)	$O(n^2)$	1	No	Yes
Counting Sort	O(n+k)	O(n+k)	O(n+k)	n+k	Yes	No