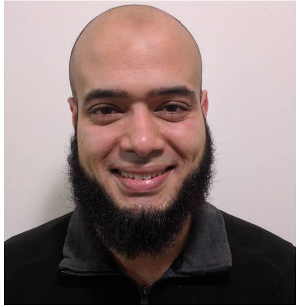
Algorithms

Count Sort

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / Msc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)



The Sorting problem (v2)

- Given an array of N numbers in the range [0-500], order them from small to large
  - Input: [9, 3, 10, 9, 5, 3, 90, 9]
  - Output: [3, 3, 5, 9, 9, 9, 10, 90]
- Problem-solving tip: Whenever solving a problem that has some **specific** properties relative to a **general** problem, you must think if these properties can allow a better or a different algorithm
- What is special here? Values have **a small range**!
  - While we can apply the general algorithms (insertion sort, merge sort, etc), can we utilize such extra constraints?
  - Take 10 minutes to think about an approach

Count Sort

- Recall: any comparison-based sorting algorithm must make at least $\Omega(n\log n)$ comparisons to sort the input array
- Count sort is an algorithm that **doesn't use comparisons** to decide the order!
- Here is a hint: Compute the frequency of the input array:
  - Input: [9, 3, 10, 9, 5, 3, 90, 9]
  - Frequency: [3 ⇒ 2], [5 ⇒ 1], [90 ⇒ 1], [10 ⇒ 1], [9 ⇒ 3]
  - Develop an efficient approach to sort the numbers based on this frequency info

Procedure

- Compute the maximum value in the array
  - Input: [9, 3, 10, 9, 5, 3, 90, 9] ⇒ 90
- Create an array of **91** values and compute the frequency of the values
  - Arr[[3] = 2, arr[5] = 1, arr[9] = 3, arr[10] = 1, arr[90] = 1
- Iterate from 0 to max and if any value has a frequency just spread them in the array
  - arr[0], Arr[1], Arr[2]: has zeros ignore them
  - arr[3] = 2:   put 3 twice in the output: [3, 3]
  - arr[5] = 1:   put 5 once in the output: [3, 3, 5]
  - arr[9] = 3:   put 9 three times in the output: [3, 3, 5, 9, 9, 9]
  - arr[10] = 1:   put 10 once in the output: [3, 3, 5, 9, 9, 9, 10]
  - arr[90] = 1:   put 10 once in the output: [3, 3, 5, 9, 9, 9, 10, 90]
- Code it! Analyze it

- Analyze it

```
vector<int> countSort(vector<int> &array) {
    // Find the largest element of the array
    int size = array.size(), mxValue = array[0];
    for (int i = 1; i < size; ++i)
        if (array[i] > mxValue)
            mxValue = array[i];

    // Compute Frequency
    vector<int> count(mxValue+1);   // zeros
    for (int i = 0; i < size; ++i)
        count[array[i]] += 1;

    // Put the values back to the array
    int idx = 0;
    for (int i = 0; i <= mxValue; ++i) {
        for (int j = 0; j < count[i]; ++j, ++idx)
            array[idx] = i;
    }
    return array;
}
```

Analysis

- Let K = the max value in an array of N integers
- Clearly this is O(K) space: the frequency array
- The first and the second loop are O(N)
- The first impression about the nested loops is order O(NK), but this is wrong.
- Clearly we loop K steps. But the total sum of the internal loop is simply the array elements
- Hence: time complexity is O(N+K)
- Worst case: If K is $\sim N^2$, the time complexity is now $O(N^2)$. Don't use this algorithm unless you are aware about the min/max values of the array

Properties

- The previous algorithm is NOT stable. We already lost values by completely depending on the frequency
- Also it is not adaptive, as the processing flow is the same regardless of the data
- Also it is not practical for online processing, as we have to compute the whole output array from the beginning
- This is NOT an in-place implementation of the algorithm as it requires extra space

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."