Distribution Sorting Algorithms

> As theories could have proved that comparison based sort can not do better than O(nlogn), what comes to geeks' minds is how can we do better if we do not reply on comparison?

Where there is a will, there is a way. At the same time, "where there is a gain, there is a pain". If we want to do better than `O(nlogn)`, we must sacrifice something. If we do not use comparison, we need to know some other information about the numbers then we can line them up.

In this post, we will take a look at the most widely known non-comparison based sorting algorithms set - **Distribution Sort**, which can break the `O(nlogn)` boundary.

Counting Sort

> Numbers between a specific range.

Assume we have `n` numbers within `start` to `end`, i.e. `start <= n[i] <= end` for all `i` in `0 ... n - 1`. Here `start` and `end` CAN be negative.

```java
public void countingSort(int[] nums, int start, int end) {
    int[] counting = new int[end - start + 1];

    // Count occurrences for all numbers
    // The index is the number subtracted by start
    // So we have: 0 <= nums[i] - start <= end - start
    for (int num : nums) {
        counting[num - start]++;
    }

    // Calculate sums in counting array
    // Then for each value in counting, we know
    // how many numbers is less or equal to the number at the index
    for (int i = 1; i < counting.length; i++) {
        counting[i] += counting[i - 1];
    }

    // Arrange positions according to counts
    int[] output = new int[nums.length];
    for (int num : nums) {
        output[counting[num - start] - 1] = num;
        counting[num - start]--;
    }

    // Put numbers back to nums
    for (int i = 0; i < nums.length; i++) {
        nums[i] = output[i];
    }
}
```

Source Codes

- **Time Complexity**: `O(n + k)`, where `n` is the length of numbers to be sorted, while `k = end - start + 1` is the value range.
- **Space Complexity**: `O(n + k)`
- Counting sort is efficient if the range of **input data is NOT significantly greater than the number of objects to be sorted (i.e. `n >> k` is the best)**. If we have a huge gap of values, say 1 to 10M, but we only got 5 numbers. You would want to create a sparse counting array take 10M space.
- Traverse from back to front when arranging positions, we can make the sorting **stable**.

Radix Sort

> Sort digit by digit. Small number of digits preferred.

Radix sort is to sort numbers digit by digit, usually from least significant digit (LSD) to most significant digit (MSD). And take use of *counting sort* or other stable sort for each iteration.

Java

```java
private int getMax(int[] nums) {
    int res = Integer.MIN_VALUE;
    for (int num : nums) {
        res = Math.max(res, num);
    }
    return res;
}

private void countingSort(int[] nums, int exp) {
    int[] output = new int[nums.length];
    int[] count = new int[10];

    for (int num : nums) {
        count[(num / exp) % 10]++;
    }

    for (int i = 1; i < count.length; i++) {
        count[i] += count[i - 1];
    }

    for (int i = nums.length - 1; i >= 0; i--) {
        output[count[(nums[i] / exp) % 10] - 1] = nums[i];
        count[(nums[i] / exp) % 10]--;
    }

    for (int i = 0; i < nums.length; i++) {
        nums[i] = output[i];
    }
}

public void radixSort(int[] nums) {
    int max = getMax(nums);
    for (int exp = 1; max / exp > 0; exp *= 10) {
        // Counting Sort for each digit
        countingSort(nums, exp);
    }
}
```

Source Codes

- **Time Complexity**: `O(nk)`, `k` is the number of digits. When digits is much less than the number of digits, radix sort can be a good choice.
- CAN be used to sort negative numbers. How? Treat the sign as a special digit!

Bucket Sort

> Input numbers are uniformly distributed over a range.

Think about a scenario when we want to sort a large set of floating point number in range from 0.0 to 1.0. We talked about counting sort which can address issue with value range, however, it can do nothing with floating point numbers as we need to use the number value as indices. Here bucket sort comes if the numbers are fair uniformly distributed.

Java

```java
public void bucketSort(double[] nums) {
    // 1. Create n buckets
    List<List<Double>> buckets = new ArrayList<>();
    for (int i = 0; i < nums.length; i++) {
        buckets.add(new ArrayList<>());
    }

    // 2. Distribute elements into different buckets
    for (double num : nums) {
        int bucketNum = (int)num * nums.length;
        buckets.get(bucketNum).add(num);
    }

    // 3. Sort elements in each bucket
    for (List<Double> bucket : buckets) {
        Collections.sort(bucket);
    }

    // 4. Concatenate all buckets
    int i = 0;
    for (List<Double> bucket : buckets) {
        for (double num : bucket) {
            nums[i] = num;
            i++;
        }
    }
}
```

Source Codes

- **Time Complexity**: If we take step 3 as `O(1)` for each bucket (uniformly distributed, so there is only one element in each bucket in average), we have overall time complexity to be `O(n)`.
- Bucket sort is **more a way to handling distributing related problems** than simply a sorting algorithm. If we meet a problem with gap between problem no more than k, find the maximum gap... *Making buckets* is a good direction to go.