Countinsort

Florian Rehm

March 14, 2022

Abstract

When it comes to sorting elements - in this case numerical values - most of the time we lack a lot of information to reach a good overall performance. When we know the maximum key value in our list of elements as well as the number of elements we can employ an implementation of Countinsort to get them sorted. It basically counts the occurences of each key value and exploits these counts as pointers to their final position. Deeper analysis shows that in general the runtime is ***linear*** *upon the number of elements but we can easily produce edge-cases where the overall runtime explodes.*

Contents

1 Problem and Motivation

Sorting elements topologically is a problem that we are facing in many different situations. A popular example would be sorting contestants in a leaderboard. Depending on the application we want to - and sometimes even must - aim for the best runtime complexity. We do not want to wait another season to determine the winner of a football-match, do we? Since we have to touch every element at least once we can already tell that we have $\Omega(n)$ as our lower boundary in sorting. One way to reach that runtime is an implementation of **Countingsort**.

2 Solution

2.1 Overview

In order for **Countingsort** to work we assume that our input only consists of numbers; more specifically positive integer values. The reason for this comes clear in ***Phase 2***. In total the algorithm consists of three phases.

2.2 Phase 1 *(Counting-phase)*

This phase is pretty straight forward. We simply go over all our elements and count their occurence. For that we introduce a counting array ***Counts*** which employs a length of **1...k** where **k** is the maximum value a key employs in our input. **Side note:** Even if unknown finding the maximum key only takes at most $\mathbf{n} \in \mathbf{O}(\mathrm{n})$ steps.

2.3 Phase 2 *(Pointing Phase)*

Now that we know how many times each key exists in our array we need to determine the final position of each key. The idea here is going to be that our ***Counts*** array already holds the repsecive keys in a sorted order. The only thing missing is the respective offsets in the sorted array.

Let $k_1, k_2, ..., k_n = \mathbf{k}$ be the possible keys going by ***Counts*** then ***Counts***(k_i) represents the remaining numbers of elements of value k_i with $1 \leq i \leq \mathbf{n}$. After ***Phase 1*** these represent the exact counts for each key. For simplicity lets assume that every key k_i exists at least once in our input. The lowest key k_1 obviously sits at the very beginning of the final array. The first element representing the the second lowest key k_2 sits at the next position after the last element of k_1. This pattern applies for every pair of key k_i, k_j with $k_i < k_j$.

To retreive a list of final positions after ***Phase 1*** we need to produce...

$$\textbf{\textit{Counts}}(k_i) = \sum_{i=1}^{i-1} \textbf{\textit{Counts(i)}}$$

The fact that we potentially have no occurences for some of these keys does nothing to the overall idea of this pointers adjustment because they will have an initial value of zero and therefore do nothing in the sums portraied above. Recursively this whole setup boils down to...

$$\textbf{\textit{Counts(i)}} = \textbf{\textit{Counts(i)}} + \textbf{\textit{Counts(i-1)}}$$

... beginning from the lowest key going through to the highest key.

2.4 Phase 3 *(Sorting Phase)*

The final part of the algorithms is copying the elements to the respective positions that we now hold in our ***Counts*** array. For that part we go over our input elements another time and for each element we retreive their final position from ***Counts*** . Finally we copy the element over to the retreived postion in a array we call ***Result*** of length **n**. This yeilds the following pattern...

$$\boldsymbol{Result[Counts(x)]} = \boldsymbol{x}$$
$$\boldsymbol{Counts(x)} = \boldsymbol{Counts(x)} - 1$$

2.5 Pseudocode(s)

3 Implementations

Python:

```python
# Finding a maximum among n elements in any
# language should not be worth any commenting really
def find_maximum(a):
    maximum = a[0]
    counter = 0
    for x in a:
        if x > maximum:
            maximum = x
        counter += 1
    return counter, maximum
# Same applies for finding the minimum
def find_minimum(a):
    minimum = a[0]
    counter = 0
    for x in a:
        if x < minimum:
            minimum = x
        counter += 1
    return counter, minimum


def advanced_counting_sort(a, high_key=None):
    # Phase 0: Normalizing for negative elements
    num_elements, low_key = find_minimum(a)
    for i in range(0, num_elements):
        a[i] -= low_key

    if not high_key:
        _, high_key = find_maximum(a)

    high_key -= low_key

    # Phase 1: Counting occurencies of each key
    counts = [0 for i in range(0, high_key + 1)]
    for i in range(0, num_elements):
        counts[a[i]] += 1

    # Phase 2: Adjusting result array pointers
    for i in range(1, high_key + 1):
        counts[i] += counts[i - 1]  # Counts-array becomes a
    pointer array

    # Phase 3: Backwardly filling up the result
    #          array with the input elements
    result = [None for x in a]
    for i in range(num_elements - 1, -1, -1):
        element = a[i]
        a[i] += low_key # Reverting normalization
        result_pointer = counts[element]
        result[result_pointer - 1] = element + low_key
        counts[element] -= 1

    return num_elements, result
```

This code represents and implementation in Python. It is called 'advanced Countingsort' because it can already deal with negative values. The basic concept however is the same since we only need to normalize the values of our input data to zero which is a linear projection that persists topology.