Why would you sort when you know where things approximately belong?

Peter Taraba *

February 1, 2025

Abstract

We introduce statistical sorting, which is close to $O(n)$ time complexity on average (as it really is $O(n \log \log n)$ complexity), and compare it with mainstream C++ std (standard template library) sort, which has on average $O(n \log n)$ time complexity. We show it on three different distributions - uniform, Gauss and an additional one. The statistical sorting is usually called nested bucket sorting, but unlike standard bucket sorting which divides into constant number of m buckets, to achieve $O(n \log \log n)$ complexity, nesting and variable number of buckets of size $\sqrt{n}$ is needed.

1 Introduction

Currently, the most popular algorithm for sorting is the quick sort algorithm, which has $O(n \log n)$ time complexity on average [1] and in the worst case $O(n^2)$. This publication introduces statistical sorting which is closer to $O(n)$ time complexity (it really is $O(n \log \log n)$ complexity). In C++, the std::sort() function is implemented using the Intro Sort Algorithm, which combines three standard sorting algorithms: insertion sort, quick sort, and heap sort [3]. On average, they all have $O(n \log n)$ time complexity [4].

In section 2, we introduce the statistical algorithm, in section 3 we show simple implementation in C++, in section 4 we compare it with C++ std sort, in section 5 we talk about simple parallelization of the statistical algorithm and finally in section 6, we draw conclusions and discuss future work.

*Berkeley, CA 94709

2 Stat Sort Algorithm

The simple idea behind statistical sorting is to go through the vector, find its minimum and maximum value (which is $O(n)$), and then split the vector into many tiny ones, which will be sorted either by the same algorithm (in case size of the vector is more than a chosen threshold; for this publication, we chose threshold 10) or by standard sorting algorithms (C++ std sort, etc). For this publication, we chose the number of vectors we split the initial vector of size s as $\sqrt{s}$. This was satisfactory to achieve better time complexity performance (closer to $O(n)$) than other sorting algorithms, which usually have on average $O(n \log n)$ time complexity. We show this on several examples - uniform distribution, Gaussian distribution ([2]) as well as one additional distribution. As the implementation of the statistical sorting is very short in C++, instead of describing it in too many details, we provide the entire algorithm in the following section. The algorithm is also known as nested bucket sorting, but instead of a constant number of buckets m, we subdivide the vectors into variable size $\sqrt{n}$ buckets, where n is length of the vectors.

3 Stat Sort Algorithm Implementation in C++

C++ implementation of the basic statistical sorting is as followed:

```
void statsort(vector<double> &vec, double min, double max)
{
    int memsize = sqrt(vec.size());
    std::vector<double> *myvectors =
        new std::vector<double>[memsize];

    for(long int i=0;i<vec.size();i++)
    {
        long int where = (int)((vec[i]-min) *
            (memsize + 0.0) / (max-min));

        myvectors[where].push_back(vec[i]);
    }

    for(long int i=0;i<memsize;i++)
    {
        if(myvectors[i].size()>10)
        {
            statsort(myvectors[i], min+(i+0.0)*(max-min)/memsize,
```

```
                    min+(i+1.0)*(max-min)/memsize);
            }
            else
            {
                std::sort (myvectors[i].begin(), myvectors[i].end());
            }
        }

        vec.clear();
        for(long int i=0;i<memsize;i++)
        {
            for(int j=0;j<myvectors[i].size();j++)
            {
                vec.push_back(myvectors[i][j]);
            }
        }
        delete[] myvectors;
}

void statsort(vector<double> &vec)
{
        double min, max;

        min = std::numeric_limits<double>::max();
        max = std::numeric_limits<double>::lowest();

        for(long int i=0;i<vec.size();i++)
        {
            if(min>vec[i])
            {
                min = vec[i];
            }
            if(max<vec[i])
            {
                max = vec[i];
            }
        }

        max += 0.0001 * (max-min);
        statsort(vec, min, max);
}
```

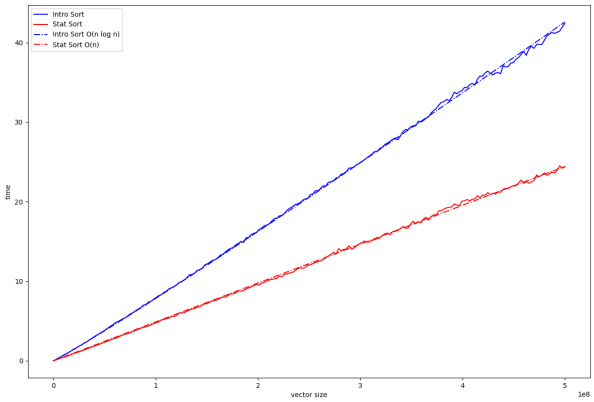

Figure 1: C++ std sort (blue) vs C++ stat sort (red) for uniform distribution vector. Full lines are measured times, the dashed blue line is $O(n \log n)$ for C++ std sort, and the dashed red line is $O(n)$ for C++ stat sort determined by residuals from linear regression fit.

4 Std Sort vs Stat Sort

Measured times of std sort and stat sort are in figure 1, as well as theoretical $O(n \log n)$ and $O(n)$ time complexity determined by linear regression and minimum residual.

The reason why statistical sorting is $O(n)$ for $n = n_1 n_2$ is because of

$$O(n_1 n_2 \log n_2) \approx O(n_1 n_2) = O(n),$$

as $\log n_2$ is approximately constant for small n_2. In case larger n would be measured (over 500000000), it could become again $O(n \log n)$ (without nesting). For our measurements, we can't go over 500000000 due to memory limitations of most linux users (16 GB of RAM memory). As we do recursion in the sorting algorithm, we can show $O(n)$ time complexity as followed:

$$O(n^{1/2}(n^{1/2} \log n^{1/2})),$$

and as $n^{1/2} \log n^{1/2}$ can be sorted again with recursion, we get

$$O(n^{1/2} n^{1/4}(n^{1/4} \log n^{1/4})),$$

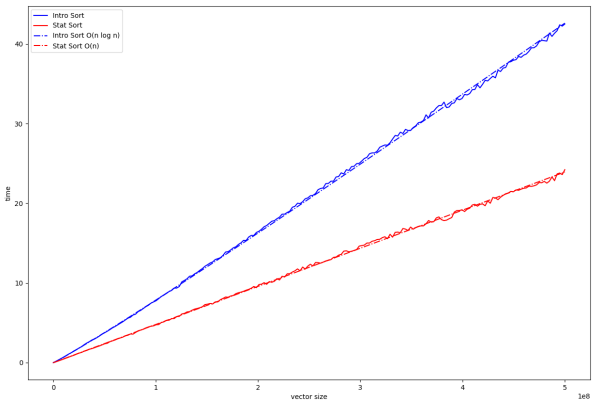

Figure 2: C++ std sort (blue) vs C++ stat sort (red) for Gaussian distribution vector. Full lines are measured times, the dashed blue line is $O(n \log n)$ for C++ std sort, and the dashed red line is $O(n)$ for C++ stat sort determined by residuals from linear regression fit.

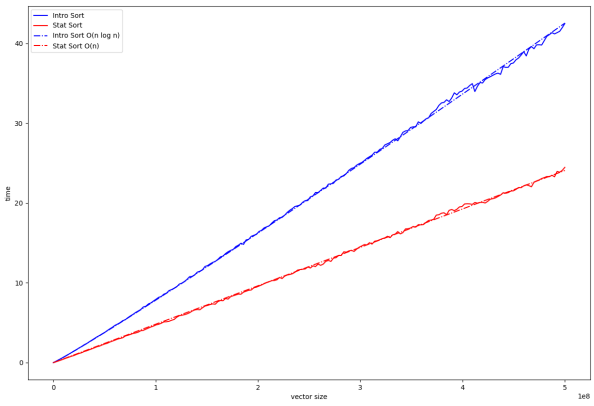

Figure 3: C++ std sort (blue) vs C++ stat sort (red) for an additional distribution vector. Full lines are measured times, the dashed blue line is $O(n \log n)$ for C++ std sort, and the dashed red line is $O(n)$ for C++ stat sort determined by residuals from linear regression fit.

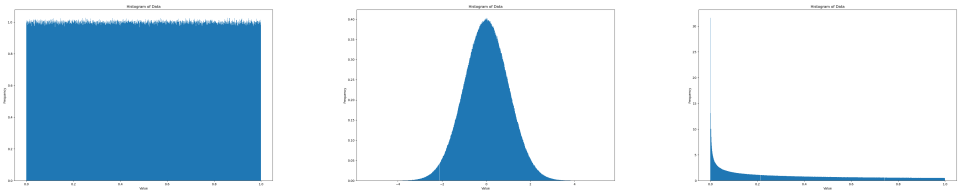

Figure 4: Three distributions tested. From left - uniform, Gaussian, and an additional one.

and using recursion again and again will get us

$$O(n^{1/2}n^{1/4}n^{1/8}...) = O(n).$$

So theoretically, using recursion we should get close to $O(n)$ even for vectors beyond 500000000.

If the constant m would be used for the amount of buckets, we could write

$$km^l \approx n,$$

where l is number of nestings, n is size of original vector and k is a treshold, which we do not nest anymore. From the equation we get $l \approx \log n$ and because of additional fact that we have to go l times through vector of size n, we would end up again with complexity $O(n \log n)$.

Because we use non-constant size of buckets, we can write

$$k^{2^l} \approx n.$$

From this we get $l \approx \log \log n$. Hence the final complexity is $O(n \log \log n)$, which is closer to $O(n)$ than $O(n \log n)$.

Table 1 contains selected measurements for time execution and table 2 contains residuals for determining time complexity of both compared algorithms. These are for uniform distribution, but similar results can be seen for Gaussian distribution in figure 2 and an additional distribution in figure 3. All three distributions can be seen in figure 4.

The entire code for the plots in figure 1, 2, 3, 4 and tables 1 and 2 can be found on github [5].

Table 1: Times for Quick & Stat Sort for various vector sizes

size	Quick Sort time	Stat Sort time
10000	1.15760485641658E-08	0
10060050	8.27546318760142E-06	5.12730912305415E-06
100510502	9.20717575354502E-05	5.54861180717126E-05
201011005	0.000190925929928198	0.000110023145680316
301511507	0.000290057876554783	0.000171666666574311
402012010	0.000397928241000045	0.000232824073464144
500000000	0.00049090277752839	0.00028194444894325

Table 2: Determining time complexity of Quick & Stat Sort through residuals

Time complexity	Quick Sort residual	Stat Sort residual
$O(n)$	27.80140662	7.98515784
$O(n \log n)$	13.80724414	11.03394399

5 Parallelization

As the initial vector is split into multiple vectors, all the smaller vectors can be sorted in parallel, which makes the statistical sorting algorithm easily parallelizable. As this is quite trivial software engineering work, we will not provide further details in this work, but code for parallelization can be found again on github [6].

6 Discussion & Future Work

We introduced statistical sorting and showed from measurements and also by intuition, that its complexity is closer $O(n)$ than $O(n \log n)$, which is better than any currently known sorting algorithm on average. On average, all known sorting algorithms have complexity at least $O(n \log n)$. We also compared it with C++ implementation of std sort, which has on average $O(n \log n)$ time complexity. We compared statistical and std sort algorithms on three different vector distributions - uniform, Gaussian and an additional

one. We also explained why theoretically choosing variable number of buckets ($\sqrt{n}$) is needed to achieve better performance.

Further work should concentrate on determining best possible threshold for recursion as well what algorithm should be used for sorting tiny vectors (if there is better possibility than C++ std sort). Further work should also look into what happens with vectors of sizes beyond ones measured in this publication as we showed only intuition beyond size of vector 500000000.

Acknowledgement

The authors would like to thank Dutch born people of Amsterdam, Netherlands for pleasant working environment in spring of 2023, when idea for statistical sorting was sparked. The authors would like to thank Grammarly for improving English in this work.

References

[1] Press, William H. and Teukolsky, Saul A. and Vetterling, William T. and Flannery, Brian P. (2007), Numerical Recipes 3rd Edition: The Art of Scientific Computing, Cambridge University Press, USA

[2] Carl F. Gauss (1823), Theoria combinationis observationum: erroribus minimis obnoxiae, Gottingae, H. Dieterich, Print

[3] Geek Writers, std::sort() in C++ STL, `https://www.geeksforgeeks.org/sort-c-stl/`

[4] Wikipedia Writers, Sorting algorithm, `https://en.wikipedia.org/wiki/Sorting_algorithm`

[5] Peter Taraba, Statistical sorting algorithm, `https://github.com/peta78/Sorting/tree/main/sorting_publication`

[6] Peter Taraba, Statistical sorting algorithm, `https://github.com/peta78/Sorting`