BunnySort

A New Sorting Algorithm for Certain Distributions of Data

Sorting Algorithms Are Useful

- Sorting algorithms are used in various fields

    Binary Search, Convex Hull, Dijkstra...

- Comparison sorts have a time complexity of $O(n \log n)$

- Other algorithms have different time complexities

Bucket Sort

- A well-known sorting algorithm uses “buckets”:

(1) Divide the data into buckets.
(2) Sort the individual buckets.
(3) Merge the buckets.

Sorting Each Bucket

Merge Sort & Counting Sort

- Merge Sort is used to *merge* small chunks of data
- Counting Sort works well with a *narrow* range of data

Our Algorithm

1. Divide the data into buckets

2. Analyze each bucket to see which algorithm between Counting Sort and Merge Sort performs well

Which Algorithm is Faster?

- Counting Sort works in $O(n + k)$
- Merge Sort works in $O(n \log n)$

- So, we have the equation $k = a \cdot n \cdot \log n + b \cdot n + c$

- After several linear regressions, the boundary condition was:

$$k = 2.68 \cdot n \cdot \log n - 0.73 \cdot n + 199$$

Results

Table 2. Running times of various algorithms on $U(1, k)$

n	k	Our method	Tim Sort	Merge sort	Quick sort
10^5	10^9	0.101	0.312	0.425	0.175
10^5	10^6	0.166	0.234	0.324	0.133
10^5	10^3	0.033	0.228	0.321	0.270
10^3	10^9	0.007	0.001	0.002	0.001

Time Complexity of Our Method

- 1. $O(n+B)$
- 2. $O\left(\min\left(\frac{k}{B}+a_i, a_i \log a_i\right)\right)$
- 3. $O(n \log B)$

→ $B \approx \sqrt{k}$

Other Algorithms

- Merge sort was too slow -> Quick sort
- Counting sort was too slow -> Radix sort

Results

2.3. Method 2 (Counting sort + Quick sort)

First of all, the classical quicksort algorithms perform poorly on reversely sorted data, so we chose RandQS, which is a variation that chooses the pivot randomly. We tested our algorithm on the same data as Table 2 and 3. The running times were averaged over 100 runs.

Table 4. Running times of various algorithms on $U(1, k)$

n	k	Our Method	Tim sort	Merge sort	RandQS
10^5	10^9	0.147	0.229	0.400	0.368
10^5	10^6	0.162	0.183	0.327	0.298
10^5	10^3	0.026	0.181	0.330	0.136
10^3	10^9	0.95e-3	1.23e-3	2.62e-3	2.03e-3

Table 5. Running times of various algorithms on $N(0, \sigma^2)$

n	σ	Our Method	Tim sort	Merge sort	RandQS
10^5	10^8	0.154	0.218	0.387	0.347
10^5	10^5	0.091	0.194	0.334	0.300
10^5	10^2	0.026	0.213	0.360	0.124
10^3	10^8	0.95e-3	1.57e-3	2.03e-3	2.47e-3

2.3. Method 3 (Radix sort + Quick sort)

Because counting sort could only sort integer data, we replaced it with radix sort. We tested our algorithm on the same data as Table 2 and 3. The running times were averaged over 100 runs.

Table 6. Running times of various algorithms on $U(1, k)$

n	k	Our Method	Tim sort	Merge sort	RandQS
10^5	10^9	0.116	0.174	0.319	0.287
10^5	10^6	2.698	0.188	0.334	0.304
10^5	10^3	0.131	0.178	0.320	0.130
10^3	10^9	0.85e-3	1.05e-3	2.33e-3	1.87e-3

Table 7. Running times of various algorithms on $N(0, \sigma^2)$

n	σ	Our Method	Tim sort	Merge sort	RandQS
10^5	10^8	0.151	0.213	0.381	0.344
10^5	10^5	0.392	0.224	0.402	0.366
10^5	10^2	0.095	0.210	0.363	0.125
10^3	10^8	1.05e-3	1.02e-3	2.29e-3	1.71e-3

Dividing into buckets

Parameterized Distributions

- We assumed that every given distribution is a union of three normal distributions. Here is an example:

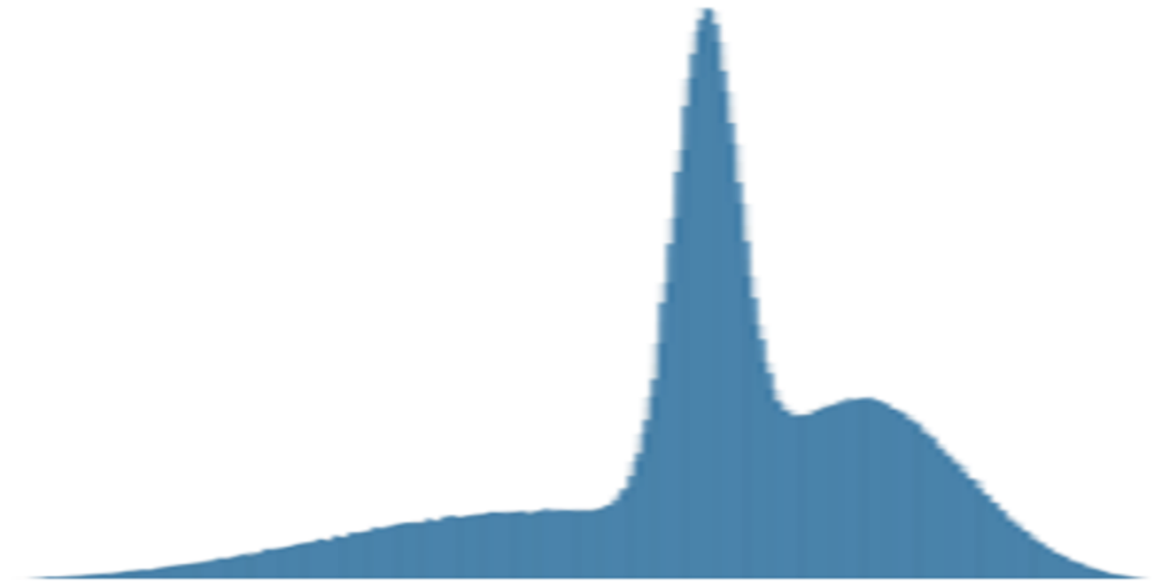

Figure 1. A concatenation of three normal distributions

Neural Networks

- Next, we used a neural network to predict the index of each element, when the list is fully sorted.

In other words, we tried to estimate the CDF

Adding a Sample Model

- This sample model takes 512 elements from the list and predicts the CDF

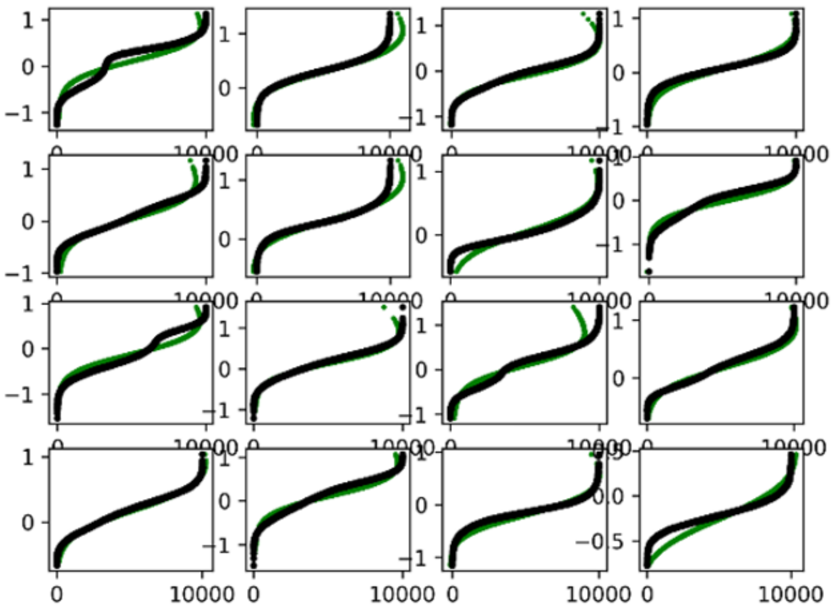

Figure 2. The estimate (green) / ground truth (black) for the model of section 3.4

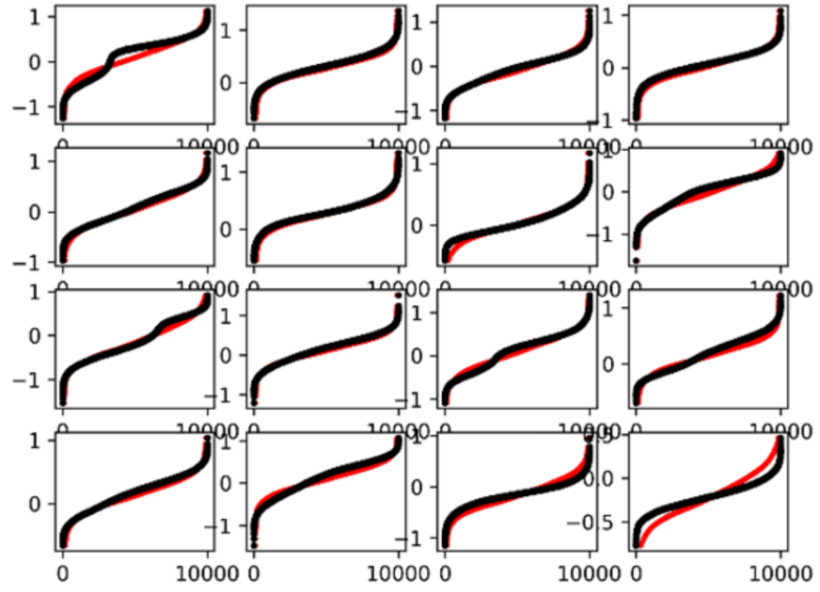

Figure 3. The estimate (red) / ground truth (black) for the model of section 3.5

Results

Table 11. Sorting time taken for each data size and algorithm in seconds. Averaged over 5 iterations.

	10^7	$5 \cdot 10^7$	10^8	$5 \cdot 10^8$
Simple	1.203	4.722	10.258	51.963
Softmax	2.129	10.373	24.078	111.67
Hybrid	1.365	5.456	14.819	57.012
Np.sort	4.516	23.591	62.342	> 300

Results

- We measured the time for each step:

Table 12. Running time of algorithm for each model in seconds; time is cumulative. Data size is 100 000 000.

	Simple	Softmax	Hybrid
Model Evaluation	2.941	16.787	4.912
Bucketing	7.489	20.934	9.652
Sorting Buckets	11.684	23.517	14.353
Sort all	11.712	23.547	14.387

C/C++

Available Algorithms

- In C++, RandQS was the best among comparison sorts

→ Counting sort + RandQS

Counting Sort + RandQS

- We implemented the same thing in C++

Table 14. Running times of various algorithms on $N(10^7, 10^{10})$

Merge sort	RandQS	Introsort	Our Method
6.367	1.003	1.080	0.955

Cumulative Distribution Functions

- We decided to put each element in each bucket according to the CDF

Predicting CDFs

- We first sorted linearly distributed data, since we already know its CDF.

- Also, we fixed the number of buckets to $B = \frac{n}{8}$.

The number of buckets is now range-proof

Results

Table 15. Running times of various algorithms on $U(1, k)$

n	k	Our Method	RandQS	Merge sort	Introsort
10^6	10^9	6.44e-2	7.93e-2	3.91e-1	8.86e-2
10^6	10^6	7.23e-2	8.55e-2	4.21e-1	9.53e-2
10^6	10^3	5.80e-2	5.42e-2	3.89e-1	7.34e-2

Neural Networks

- We exported the weights and implemented the same thing in C++

We are now using a neural network to estimate the CDF

Results

- *Table 17. Running times of various algorithms on a concatenation of three normal distributions (see Figure 1, section 3.2)*

	$2^{20} \cdot 10$	$2^{20} \cdot 50$	$2^{20} \cdot 100$	$2^{20} \cdot 300$
Hybrid	0.6546	3.6405	8.0628	24.4638
RandQS	0.7846	4.4308	9.6221	29.3900

Binary Search

- We also improved the `std::lower_bound` function in C++

Table 16. Running times of various algorithms on $U(1, k)$

n	k	Our Method	C++ standard library
10^6	10^3	1.42e-7	2.29e-7
10^6	10^6	2.68e-7	3.00e-7
10^6	10^9	2.71e-7	3.00e-7

Thank you