BunnySort: A New Sorting Algorithm for Certain Distributions of Data

Seungwon-Kim[1], Eun-ho Choi[1] and Min-geon Jeong[1]

[1]Seoul Science High School, Seoul, Republic of Korea

couterattack00000@gmail.com, chldmsgh695@gmail.com, geon4096@gmail.com

Abstract – Sorting algorithms have been an integral part of computer science. It has been over a hundred years since the first sorting algorithm was introduced. In this paper, we introduce a new sorting algorithm that takes advantage of various sorting algorithms and neural networks in Python 3 and C/C++11.

Keywords: Sorting, Algorithm, Neural Network, Sample, Distribution

1. Introduction

It is proven that the time complexity of comparison sorts cannot be lower than $O(nlog\ n)$. Other algorithms perform better than that in some cases. In this paper, we take advantage of various sorting algorithms and neural networks to outperform classical algorithms. We first test the idea in Python 3, and then implement it in C/C++11. Sections 2 and 3 were done in Python 3. Section 4 was done in C/C++11.

2. Classical Approaches

2.1. Theoretical Approaches

We can very easily prove the lower bound of the time complexity of comparison sorts. Let n be the amount of data. There are $n!$ possible initial states, so there are $n!$ nodes in the decision tree. According to Stirling's approximation, $O(\log n!) = O(n \log n)$, which is the lower bound of the time complexity of comparison sorts.

It's trivial that the time complexity of the counting sort algorithm is $O(n + k)$, where k is the range of data. We omit the proof for this one.

We already know the time complexities of both algorithms, so we know when the two algorithms would take the same time:

$$an \log n = k + bn + c \quad (1)$$

If $k < an \log n - bn - c$ then the counting sort algorithm is faster. Though several tests and linear regressions, we found the coefficients a, b, and c, with $R^2 > 0.999$. Similarly, we can also get the boundary conditions for radix sort and quick sort. Here are three boundary conditions of several algorithms.

Table 1. Boundary conditions of several algorithms

Counting sort-Merge sort	$k = 2.68n \log n - 0.73n + 199$
Counting sort-Quicksort	$k = 0.257n \log n + 3.52n - 121$
Radix sort-RandQS (base 16)	$\log k = 3.484 \log n - \frac{258.72}{n}$

Also, the counting sort algorithm only works if the data consists of only non-negative integers. If we have negative integers, we can make them positive by adding a constant to all of the data.

2.2. Method 1 (Counting sort + Merge sort)

We divided the entire data into $\sqrt{k}$ buckets (where k is the range of data), and then sorted each bucket with either counting sort or merge sort. We used the equations in Table 1 to find out which was faster. The i-th bucket would contain data in the range $[i\sqrt{k}, (i+1)\sqrt{k}]$. We tested out algorithm over two distributions of data: the uniform distribution and normal distribution. The running times were averaged over 100 iterations.

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

Table 3. Running times of various algorithms on $N(0, \sigma^2)$

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

Our method could take advantage of the counting sort algorithm when k was small. However, making too many buckets to reduce the number of k would result in a longer running time. This was about the same when σ was small in the normal distribution.

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

2.4. 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

Since we used radix sort in our method this time, we decided to compare our algorithm with radix sort. The running times were averaged over 100 runs.

Table 8. Running times of various algorithms on $U(1, k)$

n	k	Our method	Radix sort
10^5	10^9	0.116	0.190
10^5	10^6	2.698	0.134
10^5	10^3	0.131	0.074
10^3	10^9	0.85e-3	1.35e-3

Table 9. Running times of various algorithms on $N(0, \sigma^2)$

n	σ	Our method	Radix sort
10^5	10^8	0.151	0.265
10^5	10^5	0.392	0.182
10^5	10^2	0.095	0.081
10^3	10^8	1.05e-3	1.36e-3

Our method only worked better than radix sort when k was relatively large to n. So, it would be better to use a combination of radix sort and RandQS when k is large, and counting sort and RandQS elsewise.

2.5. Time complexity of the method

To further analyze our method, we calculated the time complexity of our method. Our method can be divided into 3 steps:

(a) Dividing the data into $B \approx \sqrt{k}$ buckets.
(b) Sorting each bucket.
(c) Merging the buckets.

Let a_i be the number of elements in the i-th bucket. Then we can calculate the time complexity of each step:

(a) $O(n + B)$
(b) $O(\min(\frac{k}{B} + a_i, a_i \log a_i))$
(c) $O(n \log B)$

Let's look at step (b). Since the second derivative of $x \log x = \frac{1}{x} > 0$, using the merge sort algorithm may be slower if the data is concentrated in a bucket. In that case, our method would use the counting sort algorithm to sort that bucket. Conversely, if there are few data in a bucket, our method would use the merge sort algorithm. This suggests that our method would be fast if a large portion of the data is concentrated in a narrow range, just like the normal distribution or the Erlang distribution. To be more specific, we list some widely used distributions that our method will work well on:

(a) Binomial distribution, with extreme p
(b) Hypergeometric distribution, with extreme K/N
(c) Boltzmann distribution, with large λ
(d) Geometric distribution, with large p
(e) Poisson distribution, with small λ
(f) Normal distribution, with small σ^2

We can determine the ideal size of B to optimize our method. It is obvious that the ideal size of B is proportional to $\sqrt{k}$, because of the B and $\frac{k}{B}$ terms in the time complexity. (Ignore the $\log B$ term) We just must experimentally determine the ideal size of B by each probability distribution. In this research, we decided that $\sqrt{k}$ was the ideal size.

2.6. Hardware Information

Data from Table 2 to 9 were generated with a PC with the following hardware.

Table 10. Hardware information

CPU	12th Gen Intel i7-1260P
Base speed	2.1GHz
Cores	12
Logical processors	16
RAM	16GB

3. Neural Network Approaches

3.1. Approximating Cumulative Distribution Functions

We trained an MLP (Multi-Layer Perceptron) model to predict the cumulative distibution function of any given array, so that the model could predict the index of an element when the array is completely sorted. The algorithm would look like:

(a) Run the MLP model through every element.
(b) Assign every element to an empty array according to the index given in (a).
(c) Perform an adaptive sort.

Adapative sort are sorts that perform well in already sorted arrays; examples are insertion sort and timsort. We also calculated the time complexity of each step:

(a) $O(n)$
(b) $O(n)$ best case; $O(n^2)$ worst case
(c) $O(n)$ best case; $O(n^2)$ worst case, depending on how much it is sorted

The important part is to avoid getting $O(n^2)$ in steps (b) and (c). To do that, our model must be accurate. For further work on this avenue, see for instance [3].

3.2. Parameterized Distributions

However, in many applications, we must sort data that has many different distributions. In most cases though, the data still follows a pattern, such as being normally distributed with different mean and standard variance. We call this a parameterized distribution. We develop the method of using two models; one model takes input a sample from the data to sort, and finds a representation of the distribution parameters; the second model is as in 3.1, but accepts the input from the first model and uses that information to predict the CDF(Cumulative Distribution Function) for accurately. For example, in the above case, the parameters would be the mean and the standard variance. We use the setup of a concatenation of normal distributions; this is likely to be close to practice since a lot of data are normally distributed, and this case is complex enough to be interesting. We thus take 3 equal sized normal distributions, with each mean and standard variance $-0.5 \le \mu \le 0.5, 0.1 \le \sigma \le 0.3$. Here is what the data would look like:

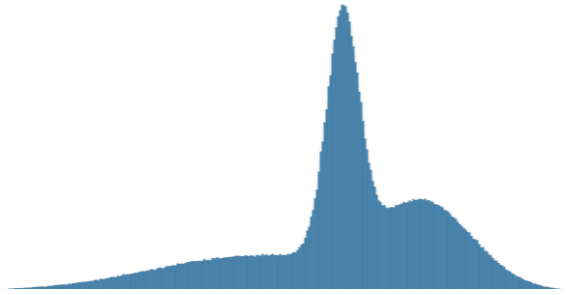


Figure 1. A concatenation of three normal distributions

When using the predicted indices to sort data, collisions can happen, which can be difficult to resolve. Thus we used bucket sorting again; we use the predicted indicies to only determine the bucket the element goes to, and sort each bucket with a classical sorting algorithm; after concatenating all buckets in order, an adaptive sort can be

performed to ensure that all elements are correctly sorted.

3.3. Method 4 (Multi-Layer Perceptron)

We first tested our idea without a sample model, using a dense feed-forward neural network. The model had 3 dense layers with 16, 4, 1 nodes each. We trained the model with 1 million normally distributed data and achieved an average error of about $0.01n$. This still means we get $O(n^2)$ in step (c). Also, the overhead of step (a) was too large, since we used a dense neural network. So, we modified the model and decided to test the model through large n's.

3.4. Method 5 (Multi-Layer Perceptron Redux)

We then added a sample model, also a multilayer perceptron. This model took a sample of size 512, and had a hidden dimension of dimension 80, and a depth of dimension 4, resulting in an representation of dimension 8. We can afford this model to be big, since the time it takes to generate the representation is constant (since the sample size is constant), and thus negligable to running the CDF estimator. The CDF estimator, i.e. the model that predict the index, had a dimension of 16, and a depth of 3. We trained the model on data generated of size 250 000, with a batch size of 512 was trained for 50 000 times, repeating 10 times on each of the 5 000 data generated. the loss was the mean squared error of the model output with the actual index.

3.5. Method 6 (Discretized Output)

We developed another method to predict the index, in a different way, inspired by output layers of transformers. We divided [0, 1] into 48 equal intervals, and and we trained the output model (still a MLP) to predict the probability that the element being read was in each bucket. This was done by making the output dimension 48 and taking the softmax of the output. The loss was taken to be the KL-divergence of the ground truth (the delta distribution of where the element actually is) from the predicted distribution. To predict the index, we dot producted the middle value of each interval with the softmax probabilities. This was also trained in the same way, with the same amount of the data. All other details of the sample and estimator models were same.

3.6. Method 7 (Hybrid Loss)

After plotting the output of the model of 3.4, we discovered that the index predicted by the model "overshot" at the extreme indices (see figure 2). In order to make the model more accurately predict the index, we developed a hybrid loss. Using the idea of KL divergence again, the model output can be thought of estimating the CDF; we took differences with respect to the sorted indices so that we could get the PDF. This can also be done with the "ground truth"; the CDF obtained by taking the sorted indices and diving by the data size. Then we took the KL divergence, and combined it with the MSE loss.

3.7. Results

We first sample 10 000 data following the distribution of section 3.2 16 times and compare the CDF with the model's prediction. In the below diagrams, the x axis is the index, while the y axis is the values of the elements.

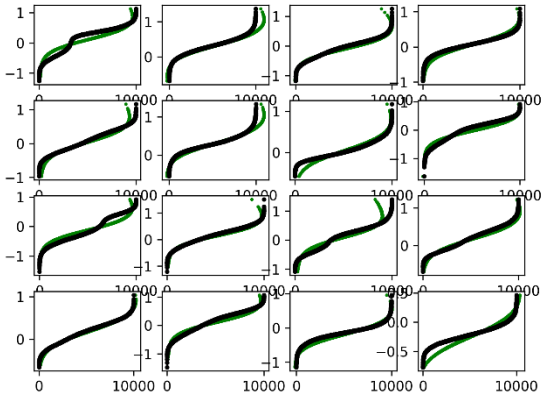


Figure 2. The estimate (green) / ground truth (black) for the model of section 3.4

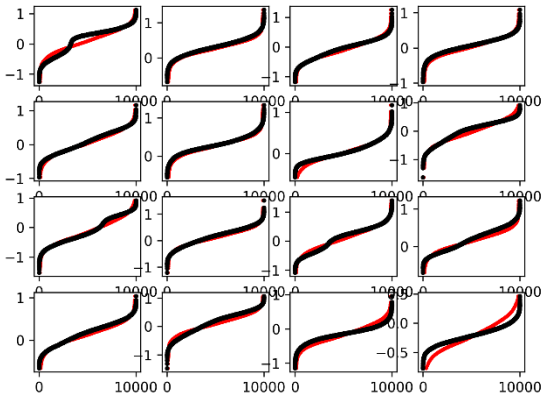


Figure 3. The estimate (red) / ground truth (black) for the model of section 3.5

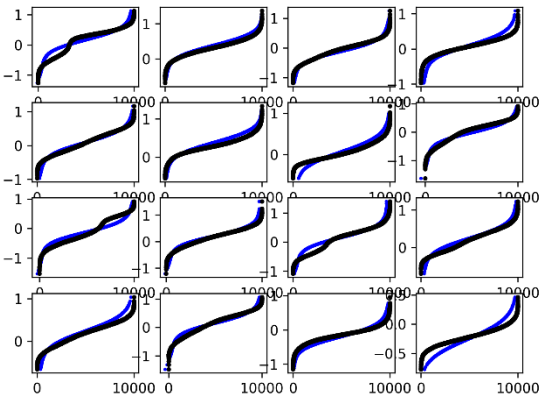


Figure 4. The estimate (blue) / ground truth (black) for the model of section 3.6

The sorting time was evaluated a nd compared with np.sort, for arrays of size $10^7, 5 \cdot 10^7, 10^8, 5 \cdot 10^8$, distributed in the same way as Section 3.2. We call the model of section 3.4 *simple*, of section 3.5 *softmax*¸of section 3.6 *hybrid*. The algorithm used was that of 3.2. The number of buckets was experimentally determined to be optimal at the value of 16, in the range of data size tested.

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

We thus improved upon the classical algorithm of np.sort by up to 300%. To understand the time used by each step of the algorithm in 3.2, we timed each step for a single run of data size 10^8.

Table 12. Running time of various algorithms 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

In contrast to the different accuracies of the models expressed in the figures 2 to 4, the time used for model evaluation is the most important factor in each model's running time. This may be because the bucket size is small, and thus most inaccuracies are ignored. Regardless of the model used, the remaning steps of the algorithm take roughly the same amount of time.

3.8. Hardware Informtion

The library used was JAX and equinox; the Adam optimizer was used with a learning rate of 10^{-4}. All training were performed on data generated with a size of 250 000, and at each iteration a batch size was used of 512, with 10 iterations for a single array of generated data. In total, 5 000 data was generated and used, equal to 50 000 iterations. The training was done on a RTX 3090. The evaluation was done on the following hardware:

Table 13. Hardware information. No GPU was used.

CPU	12th Gen Intel i5-1235U
Base speed	1.3GHz
Cores	10
Logical processors	12
RAM	16GB

4. C/C++ Implementation

4.1. Introduction

Section 2 and 3 were done in Python 3, and we implemented it in C/C++11. Since RandQS and Introsort(C++ standard sort) are the best comparison sorts in C++, we decided to only use the combination of counting sort and RandQS, which is the same as section 2.3.

4.2. Method 2 Revisited

Everything is the same as section 2.3, except the entire thing is written in C++. Here are the results. The running times were averaged over 10 iterations.

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

4.3. Approximating Cumulative Distribution Functions

If we could estimate the CDF of the data, we could divide the data into buckets according to the CDF. Also, we could make relatively small number of buckets, because we can *ensure* the data will be roughly sorted. First, we sorted linearly distributed data because the CDF is easy to estimate. Later, we expand this to unknown distributions, using a neural network to estimate the CDF.

4.4. Sorting Linearly Distributed Data

We first sorted linearly distributed data since we already know its CDF. We divided the data into $B = \frac{n}{8}$ buckets, and then put the data into each bucket according to its size, so that each bucket is roughly the same size. Next, we sorted each bucket with RandQS. After that, we have B sorted buckets. Because the buckets are already sorted, we used insertion sort to merge the buckets. We tested our algorithm over the same data as table 2. The running times were averaged over 100 iterations.

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

If we "approximate" the CDF then the number of buckets is irrelevant to the range of data, which makes the algorithm more range-proof. Also, if k isn't too small, we could approximate the CDF accurately enough to put each element in the right bucket. However, if k is too small the insertion sort would have to swap a lot of elements, which resulted in longer running times.

4.5. Binary Search

If we could approximate the CDF of some data, we could improve the speed of binary search on that data. We improved the speed of std::lower_bound, which is a binary search function in C++. We empirically decided that the error range of the estimated CDF is about $n^{\frac{1}{3}}$. We used this to perform a binary search on an array of size $n = 10^7$. Here are the results. The running times were averaged over 10^7 iterations.

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

Because we approximated the CDF, it is enough to search near the expected index. And the error is small enough(small than $O(\sqrt{n})$) to outperform the C++ standard library.

4.6. Neural Networks

We investigate whether the speedups of section 3.7 apply to lower-level, fine-tuned applications.

We exported the weights of the MLP in section 3.4 and section 3.6 manually. The MLP was implemented in C11 and OpenCL[1], with CLBlast[2] as the BLAS (Basic Linear Algebra Subprograms) backend. A custom API was provided for use in C/C++ applications. Custom kernels for copying vectors and running the gelu activation were made. Implementing this neural network must be done carefully; in particular, the first layer of the estimator can be accelerated. This is because the inputs of the first layer are same except for the last dimension (which is the element of the actual input array). The other inputs are generated by the sample model, which is same for a sample of the same array. Let A be the weights of the first layer, and $A = [A'|a]$, where a is the last column vector. Each i-th input can be seen as $\begin{bmatrix} l \\ x_i \end{bmatrix}$, where l is the latent vector from the sample model, and x_i is the i-th element of the input array. Finally, let b be the bias. Then the i-th output is given as

$$A'l + ax_i + b \qquad (2)$$

Thus, we may tile the output buffer of the first layer with $A'l + b$, and then do a rank-1 update with $a\,x^T$ where x is the input array seen as a column vector.

Next, we must use the predicted indices of the Neural Network model to bucket and sort the array. The pseudocode for the algorithm used to do this is down below; we essentially used this approach for both the Python and the C/C++ case.

```
   function BunnySort(arr[1…n], b)
    count[1…b] // count the number of elements
in bucket
    pred[1…n] // predicted indices
    out[1…n] // output array

    //predicts the distribution
    latents = SampleNN(arr)

    for i in [1…n] do
        pred[i] = EstimatorNN(latents, arr[i])
        count[floor(pred[i] * b)]++
    for i in [2…b] do
        count[i] += count[i – 1] // accumulates
the bucket indices
    for i in [1…n] do
        ind = floor(pred[i] * b)
        count[ind]--
        out[count[ind]] = arr[i]
    for i in [1…n] do
        end = (n if i == n else count[i+1])
        Sort(out[count[i]…end])
    AdaptiveSort(out[1…n])
    return out
```

The adaptive sorting algorithm must be sufficiently fast on roughly sorted arrays; we use insertion sort in this case. As we are not using external GPUs, we use the main RAM to store our buffers. The simple model however performed inferior on all our tests. This was because the model tended to under/overpredict the indices of the elements at the tails. The hybrid model suffered much less from this problem, and yielded the performance below.

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

The RandQS algorithm was highly optimized, and used insertion sort when the number of elements was less than 16. This algorithm performed better than all other classical algorithms we implemented on almost all data. The running times of each step in the sorting method (see pseudocode of section 4.6) for a single run are outlined below.

Table 18. Running times of each step of the algorithm in section 4.6 (see also Table 12), on a data size of $2^{20} \cdot 100$.

	Model	Count	Bucketing	Sort each	Sort
Time	2.94	0.64	1.83	1.74	0.10

The machine where the tests of table 17 and 18 were run on is the same as in table 13, section 3.8.

5. Conclusion

The algorithm in section 3.4 and 4.6, implemented in both Python (see section 3.4) and C/C++ (see section 4.6) outperformed other sorting algorithms on the distributions it was trained on. We decided to name the two methods "BunnySort". The "bu" is from buckets, and "nn" from neural networks.

The methods of BunnySort, in particular predicting indices of elements, also can be used to improve the performance of binary searches, especially on offline queries where the model predictions may be rapidly computed. This is a path we plan to further explore.

6. Discussion

To improve our method in Python 3, we could consider parallelizing the process of merging buckets. Also, we could adjust the number of buckets according to n and k. (A neural network would help) Also, it is important to know when the algorithm in section 2.3 outperforms that of section 2.4. Then we could further improve the algorithm.

To improve our method in C/C++11, first we may try to improve the model evaluation speed. We currently implement the model by synchronizing after each layer. However, as the estimator model size is small, with a hidden dimension of 16, which allows the whole model to be executed on a single thread.

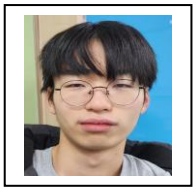
References

[1] J. E. Stone, D. Gohara and G. Shi, (2010) OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. In Computing in Science & Engineering, vol. 12, no. 3, pp. 66-73. Published in IEEE.

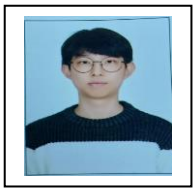
[2] Nugteren, C. (2018). CLBlast: A tuned OpenCL BLAS library. In Proceedings of the International Workshop on OpenCL (pp. 1-10).

[3] Zhu, X., Cheng, T., Zhang, Q., Liu, L., He, J., Yao, S., & Zhou, W. (2019). NN-sort: Neural network based data distribution-aware sorting. arXiv preprint arXiv:1907.08817.

Seung-Won Kim
He is a student in Seoul Science High School. He is interested in computer science and mathematics.



Eun-Ho Choi
He is a student in Seoul Science High School. He is interested in neural networks and mathematics.



Min-Geon Jeong
He is a student in Seoul Science High School. He is interested in algorithms.