Evaluating Sorting Algorithms with Varying Data Sortedness

Wei-Tse Kao, I-Ju Lin

May 2nd, 2023

Why sorting is important?

Background

Outline

- Sorting and Sortedness
- (K, L) Sortedness Matrix
- (K, L) Sorting Algorithm
- Benchmark - BoDS

Why sorting is important? Benefits of Sorting

- Faster read
- Better performance in index designed structure
- Easier data analysis

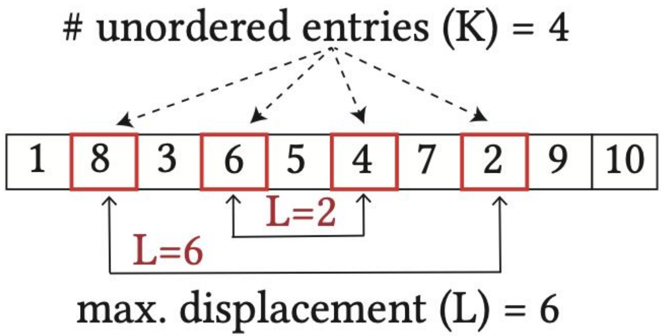
Sortedness

Refers to the degree to which the data is ordered

Degree of Sortedness

(K, L)-Sortedness Metric

- *K: the number of the elements* are out of place
- *L*: the maximum positional displacement of the out-of-order elements



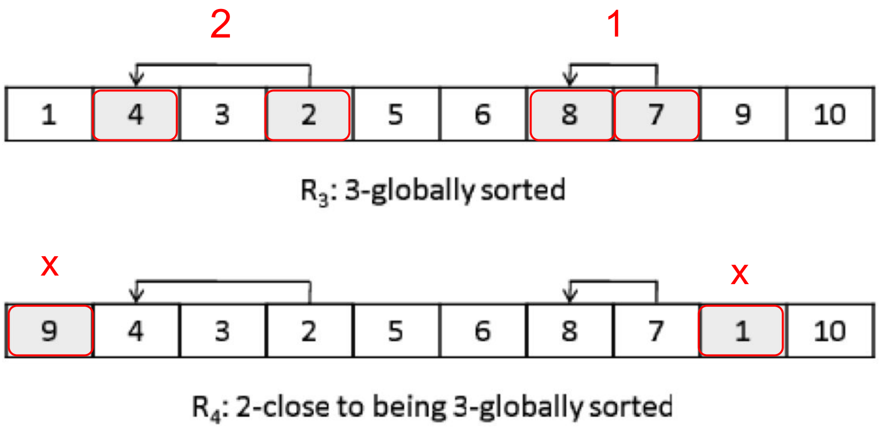
Define Near-Sorted Index

- **K-close** to being sorted:

  The size of unordered indices set is **smaller or equals** to K.

- **L-globally** sorted:

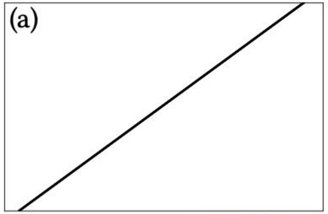
  The distance between the locations of any two unsorted tuples is always **smaller** than L.



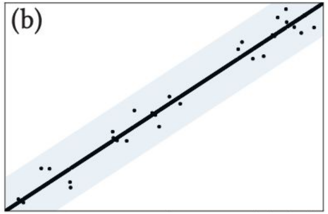
Workloads respective to (K, L)

X-axis: position of entry in data.

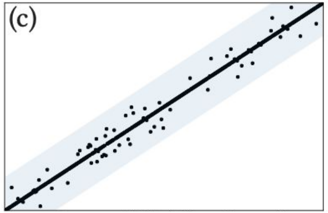
Y-axis: entry-value.



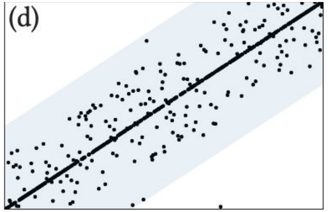
(a) K=0%, L=0%



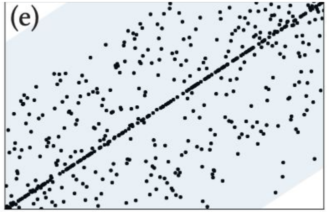
(b) K=10%, L=10%



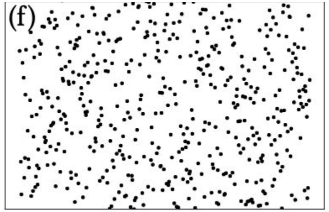
(c) K=20%, L=10%



(d) K=50%, L=25%

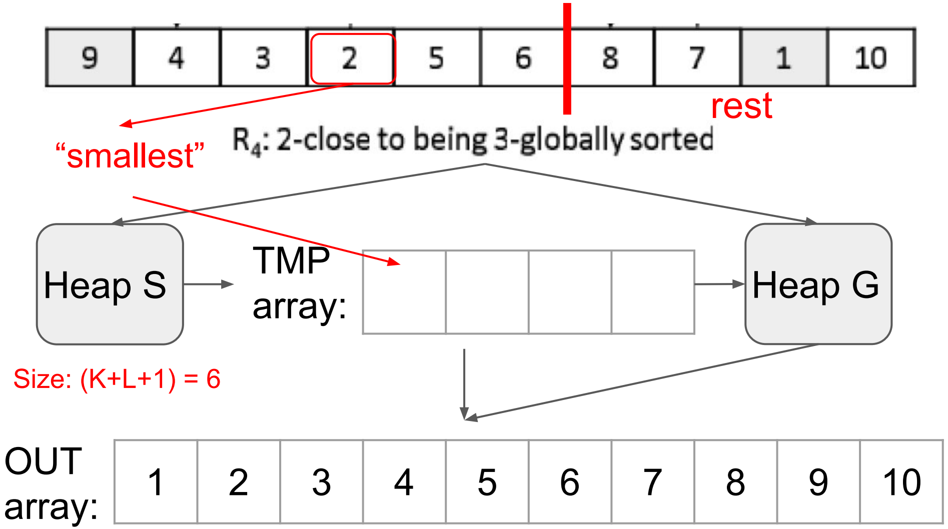


(e) K=100%, L=50%



(f) K=100%, L=100%

(K, L) Sort by Binary Min Heap



Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap

Best Case:	Worst Case:	Memory:	Stable:
O(n)	O(n log(n))	O(n)	Yes

where heap extractMin() and insert() takes O(log n).

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

BoDS

Benchmark on Data Sortedness

Data generator producing data respective to specific values of the (K, L)-sortedness metric.

https://github.com/BU-DiSC/bods

BoDS: Benchmark on Data Sortedness

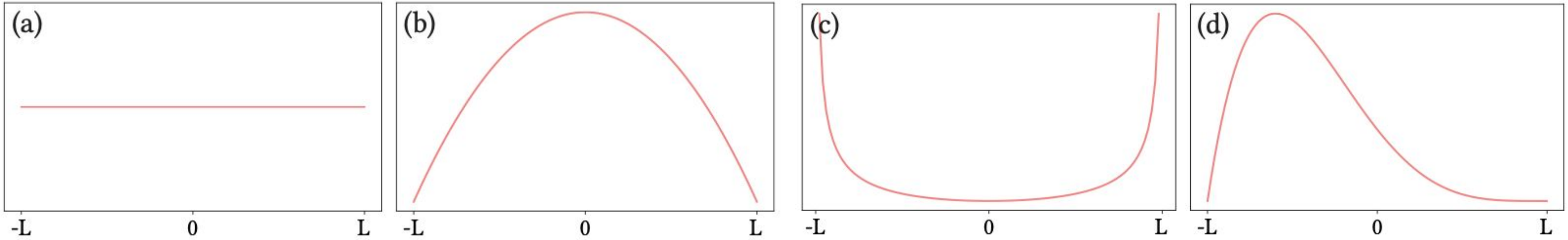
Input parameters:

- *K: proportion(%) of the elements* are out of place
- *L*: *proportion(%)* maximum positional displacement of the out-of-order elements
- N: number of entries
- α, β: parameter to regulate distribution map of indexes
- P: size of payload

https://github.com/BU-DiSC/bods

Selection of α, β

When α=1, β=1, the displacements are uniformly distributed



(a) α=1, β=1 (b) α=2, β=2 (c) α=0,5, β=0.5 (d) α=2, β=5

Probability distribution of **Beta-distribution** map bounded between [−L, L]

Experiment: 5min

Experiment

Evaluating Sorting Algorithms with

1. Sorted index workload
2. Unsorted index workload
3. Partially sorted index workload

E-Work Life Scale - Pearson Test

(green)	statisically significant
(grey)	not statisically significant

n = 42
α = 0.05

P-value	calories	steps	stress	heart rate
Organisational Trust	0.032	0.369	0.102	0.026
Flexibility	0.537	0.208	0.841	0.068
Worklife Interference	0.009	0.925	0.066	0.026
Effectiveness & Productivity	0.085	0.484	0.247	0.000
E-Work Life Scale	0.038	0.518	0.173	0.002

BoDS: Benchmark on Data Sortedness

10,000,000 = 10^7
entries to generate

10% * 10^7 = 10^6
is the max displacement

Key beta-distribution
with α=1, β=1.

./sortedness_data_generator **-N 10000000** **-K 10** **-L 10** -o ./created_data.txt -S 0 -a 1 -b 1 -P 0

10% * 10^7 = 10^6
entries is out of place

Random Seed: 0

Payload size: 0 bytes

https://github.com/BU-DiSC/bods

BoDS: Benchmark on Data Sortedness

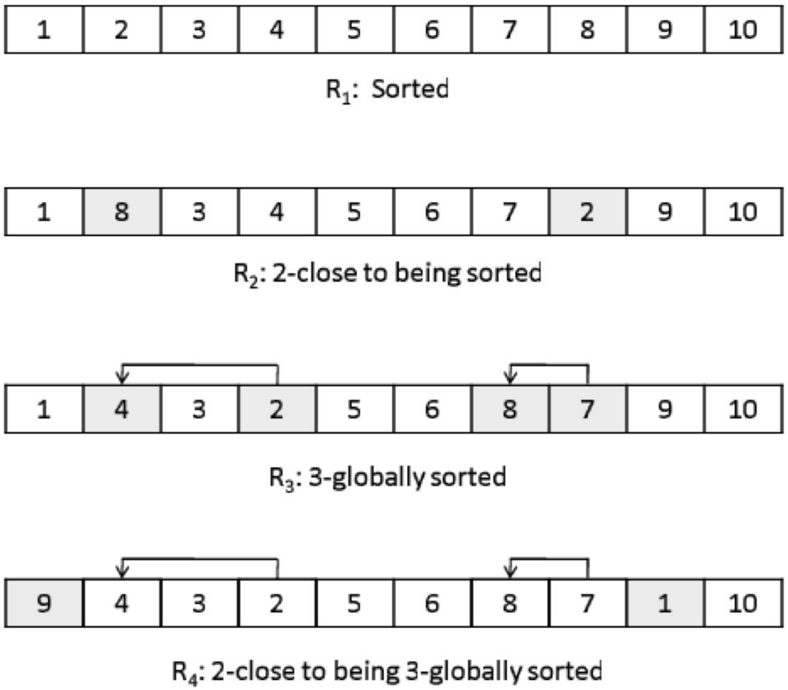
K	L
100	1
50	1
25	1
10	1
5	1
1	1

K	L
1	5
1	10
1	25
1	50
1	100
100	100

```
for ((k=0; k<=100; k+=10)); do
   for ((l=0; l<=100; l+=10)); do
       OUTPUT="./workloads/createdata_10M_K"${k}"_L"${l}".txt"
       ./sortedness_data_generator -N 10000000 -K $k -L $l -o $OUTPUT -S 0 -a 1 -b 1 -P 0
   done
done
```


https://github.com/BU-DiSC/bods

Sorted and nearly-sorted relations



Sagi Ben-Moshe, Yaron Kanza, Eldar Fischer, Arie Matsliah, Mani Fischer, and Carl Staelin. 2011. Detecting and exploiting near-sortedness for efficient relational query evaluation, ICDT '11. https://doi.org/10.1145/1938551.1938584

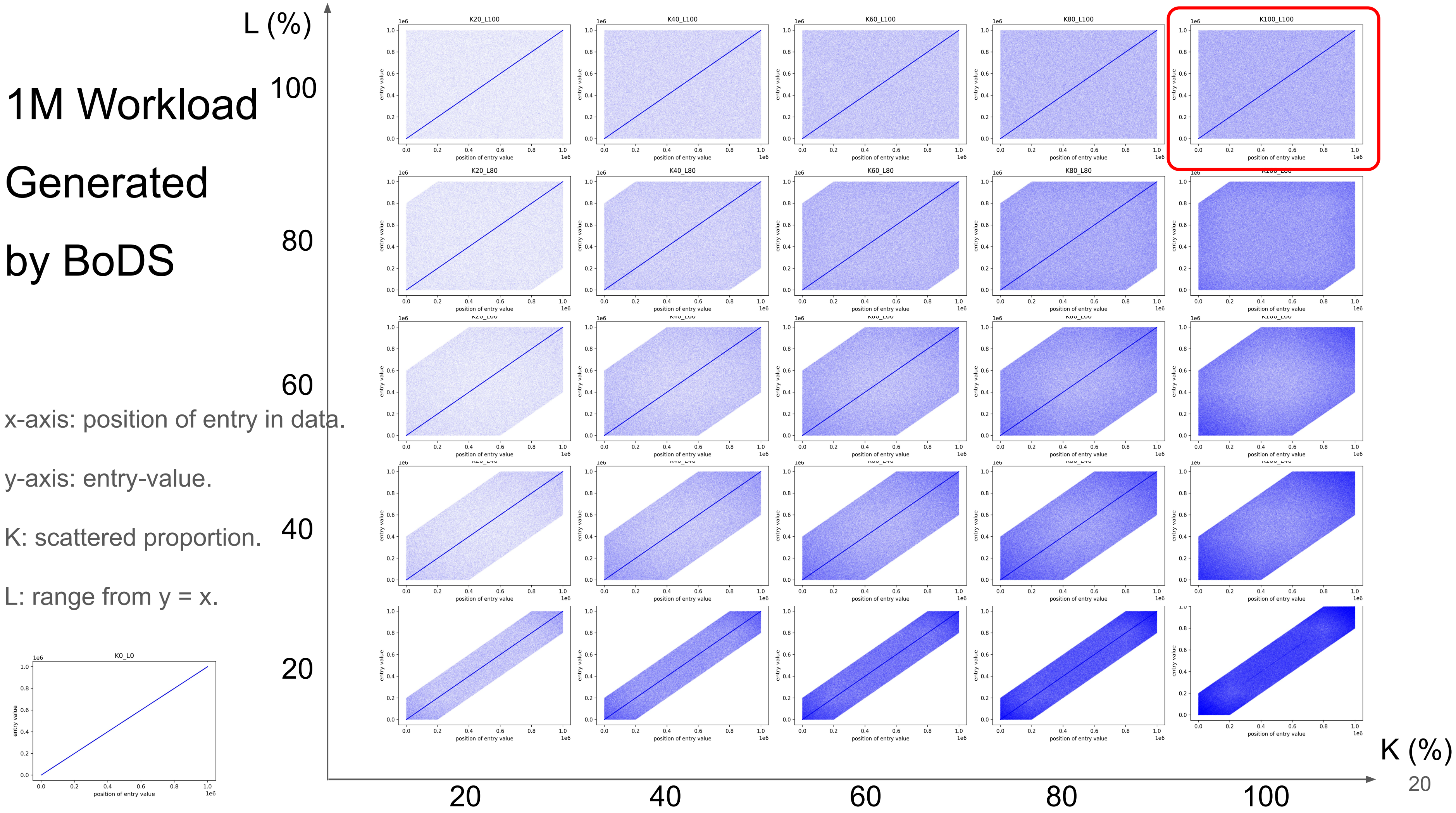
1M Workload Generated by BoDS

x-axis: position of entry in data.

y-axis: entry-value.

K: scattered proportion.

L: range from y = x.



Setup: Implementation and Solution Approach

- Windows Subsystem for Linux
- Intel® Core™ i9-11900H@2.5 GHz
    - 24M Cache
    - 8 Cores
- Two 16GB of RAM
- C++ libraries:
  algorithm, chrono, climits, cstdlib, fstream, iostream, string.

Experiment Interface

File path of the input workload generated by BoDS

Sorting algorithm to use

Divisor for L when using kl_sort:
Estimated L =10^7 * 1% / **100** = 10^3

./main.out ./created_data_K50_L1.txt ./result.csv kl_sort 100 100

File path to store output

Divisor for K when using kl_sort:
Estimated K =10^7 * 50% / **100** = 5 *10^4

Example Output

./main.out ./created_data_K50_L1.txt ./result.csv kl_sort 100 100

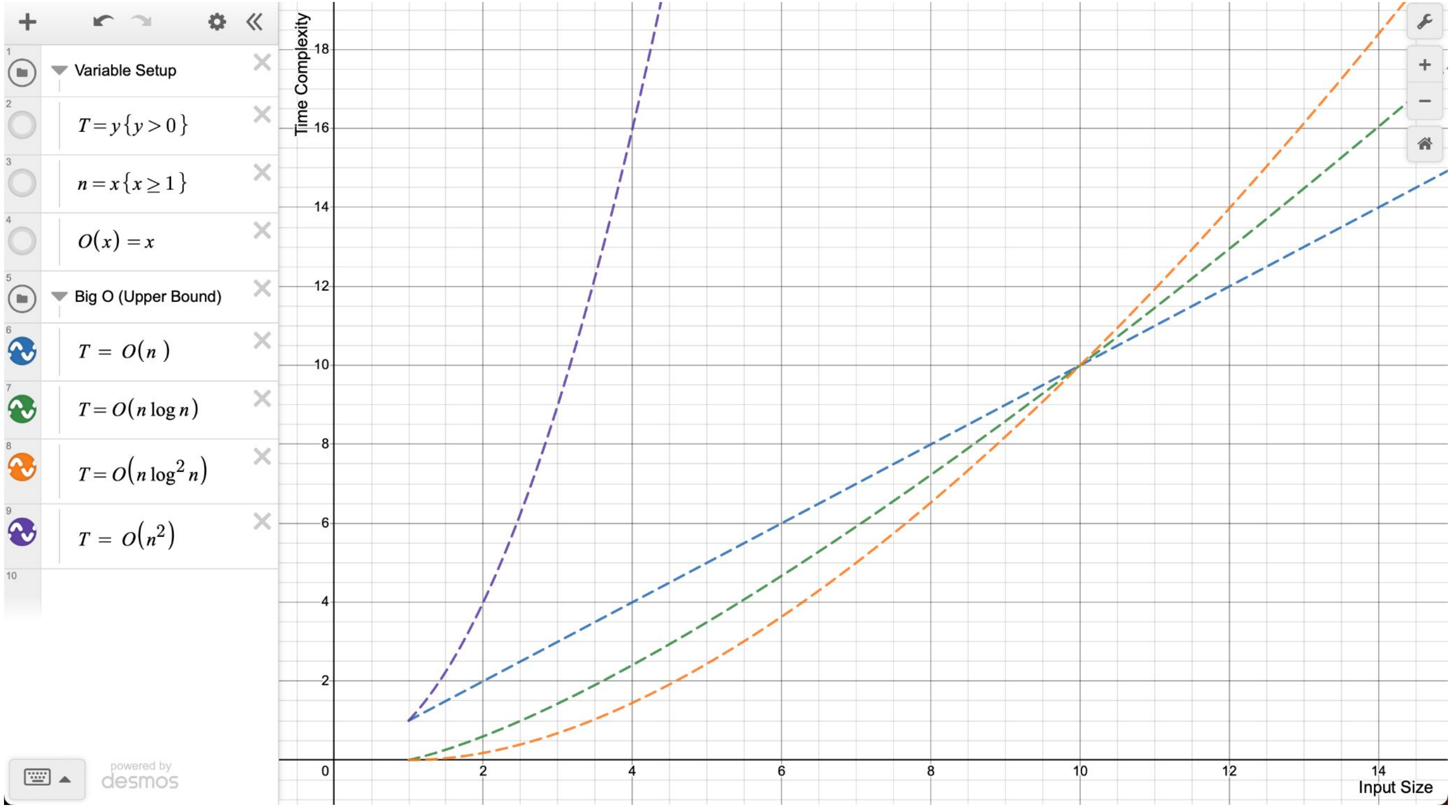
K	K_DIV	L	L_DIV	ALGORITHM	DURATION (ns)
50	**100**	**1**	**100**	**kl_sort**	**46342235**
25	100	1	100	kl_sort	35101784
10	100	1	100	kl_sort	31461449
5	100	1	100	kl_sort	33199069

Rows in ./result.csv

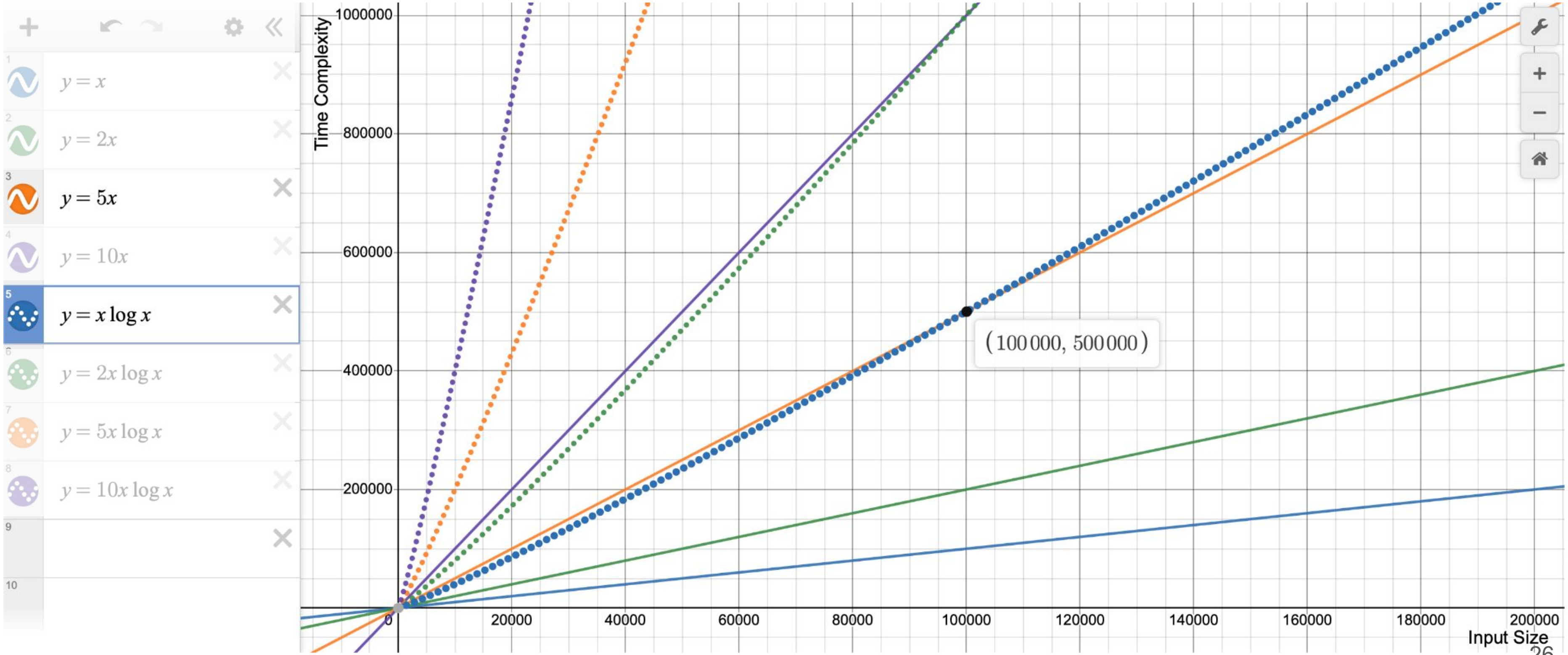
Sorting Algorithm Baselines

	Best Case	Worst Case	Memory	Stable	10^6 Sorted Index (seconds)	10^6 Unsorted Index (seconds)
KL Sort	O(n log(n))	O(n log(n))	O(n)	Yes	0.000008	3.859853
Insertion Sort	O(n)	$O(n^2)$	O(1)	Yes	0.055174	288.114436
Quick Sort	O(n log(n))	$O(n^2)$	O(log(n))	No	0.535943	0.772779
std::stable_sort	O(n log(n))	$O(n \log^2(n))$	O(n)	Yes	0.700985	1.284057
TimSort	O(n)	O(n log(n))	O(n)	Yes	0.961878	1.464594
Merge Sort	O(n log(n))	O(n log(n))	O(n)	Yes	1.272614	1.999205
Radix Sort	O(n)	O(n)	O(n)	Yes	2.083822	0.628858
~~**Selection Sort**~~	$\cancel{O(n^2)}$	$\cancel{O(n^2)}$	$\cancel{O(n^2)}$	~~No~~	~~531.778953~~	~~516.02257~~

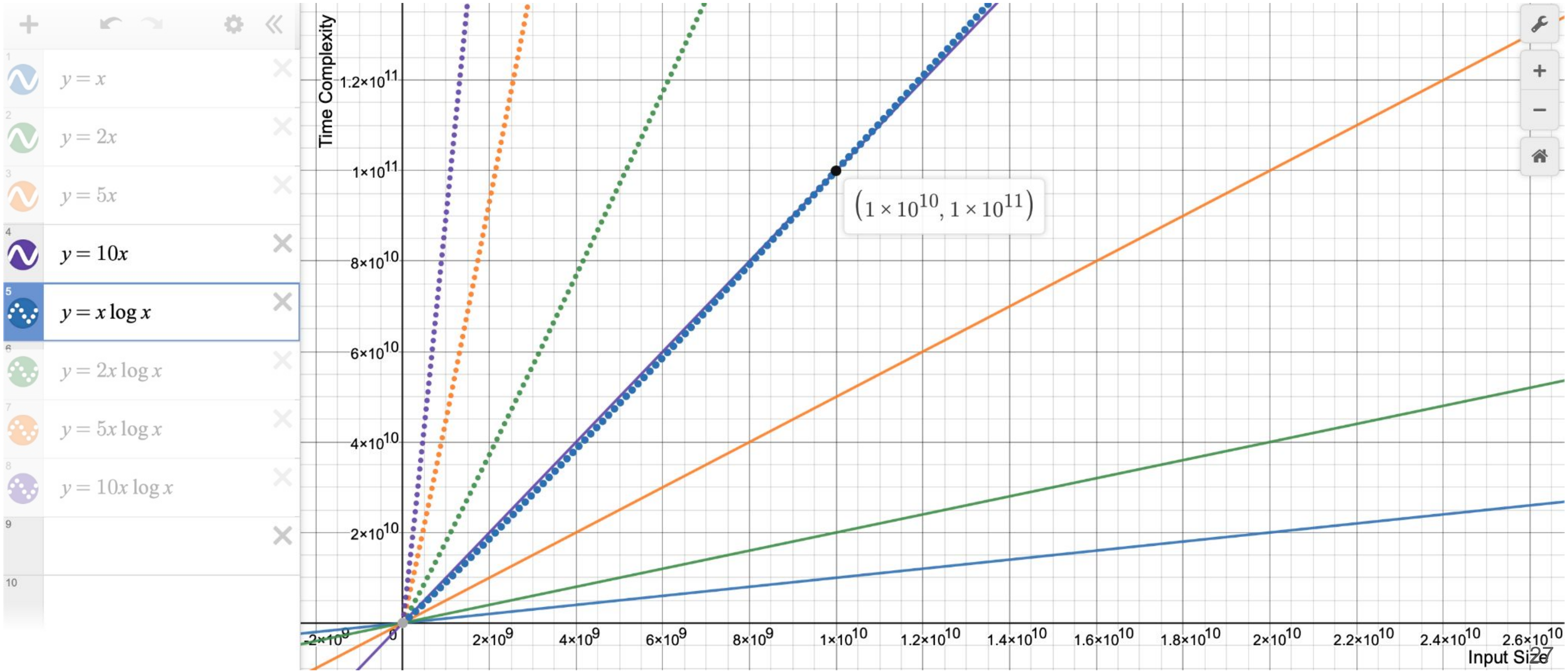
Big-O Analysis



Big-O Analysis



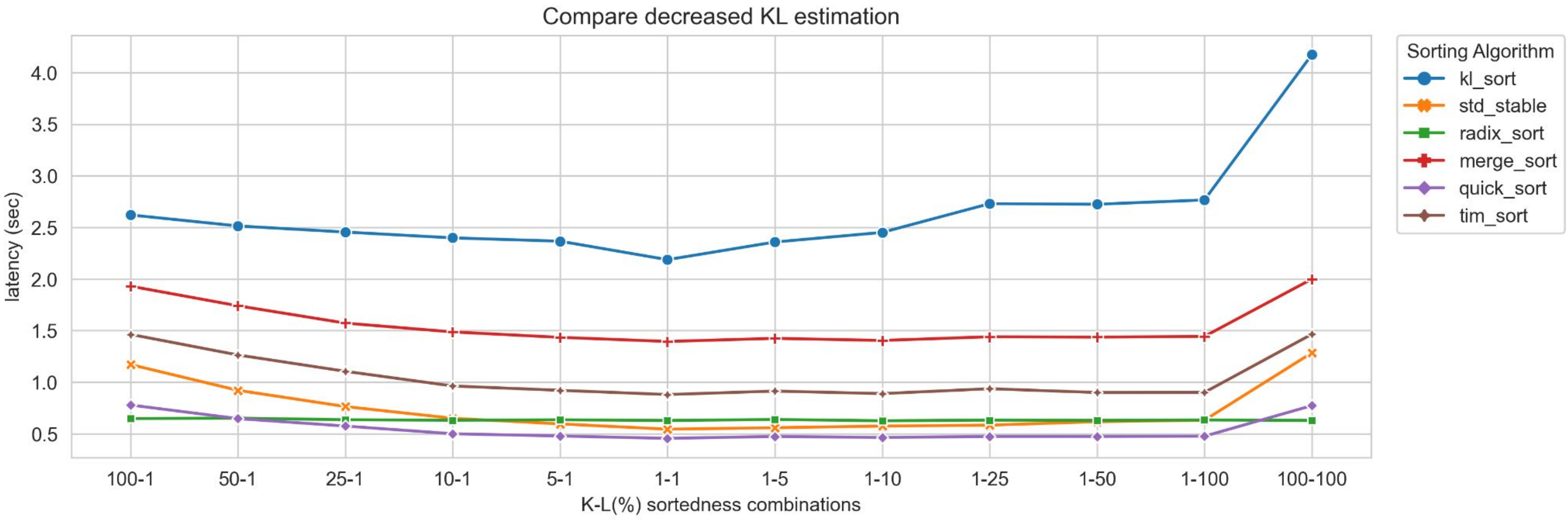
Big-O Analysis



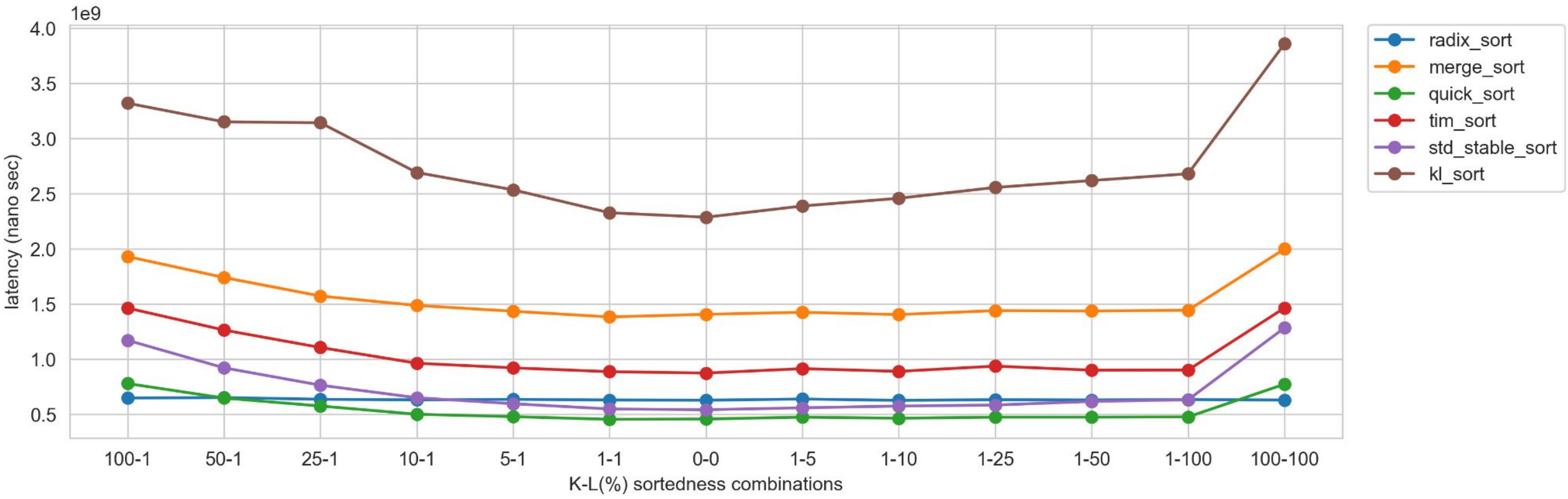
Sorting Algorithm Baselines

	Best Case	Worst Case	Memory	Stable	10^6 Sorted Index (seconds)	10^6 Unorted Index (seconds)
Radix Sort	O(n)	O(n)	O(n)	Yes	2.083822	0.628858
Quick Sort	O(n log(n))	$O(n^2)$	O(log(n))	No	0.535943	0.772779
std::stable_sort	O(n log(n))	$O(n \log^2(n))$	O(n)	Yes	0.700985	1.284057
TimSort	O(n)	O(n log(n))	O(n)	Yes	0.961878	1.464594
Merge Sort	O(n log(n))	O(n log(n))	O(n)	Yes	1.272614	1.999205
KL Sort	O(n)	O(n log(n))	O(n)	Yes	0.000008	3.859853
~~**Insertion Sort**~~	~~O(n)~~	$\cancel{O(n^2)}$	~~O(1)~~	~~Yes~~	~~0.055174~~	~~288.114436~~
~~**Selection Sort**~~	$\cancel{O(n^2)}$	$\cancel{O(n^2)}$	$\cancel{O(n^2)}$	~~No~~	~~531.778953~~	~~516.02257~~

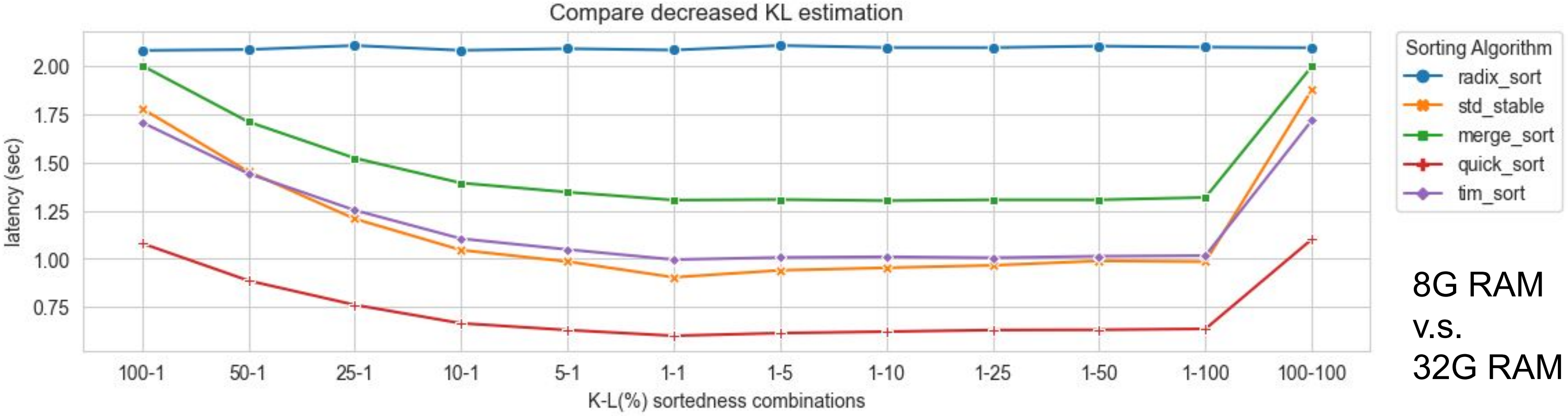
Performance of Algorithms on various Sortedness



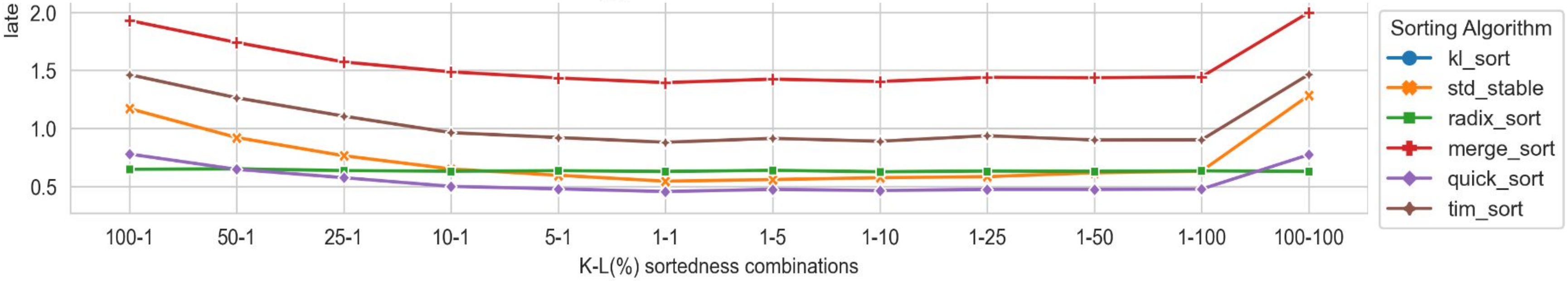
Performance of Algorithms on various Sortedness: K/1, L/1

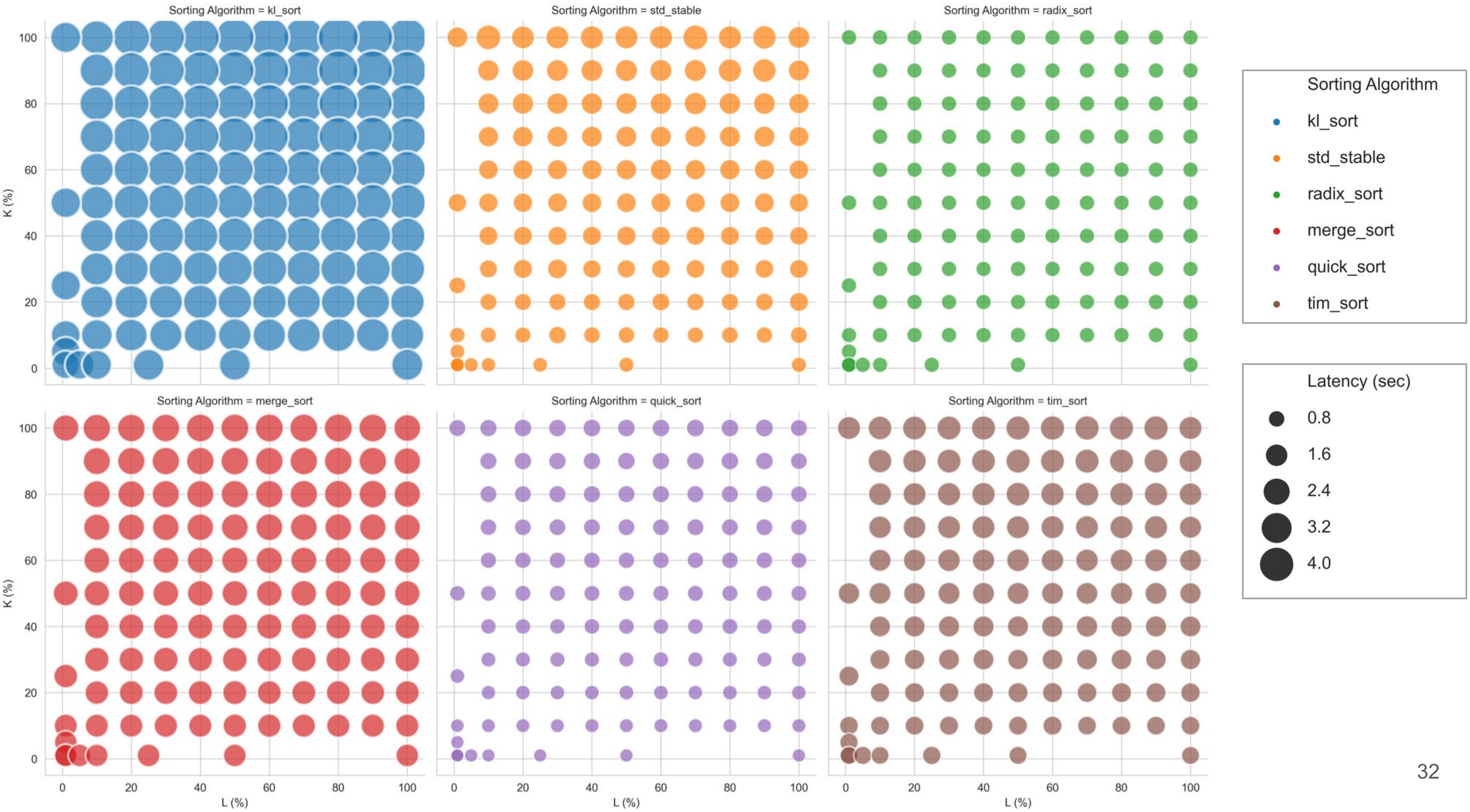


Performance of Algorithms on various Sortedness:

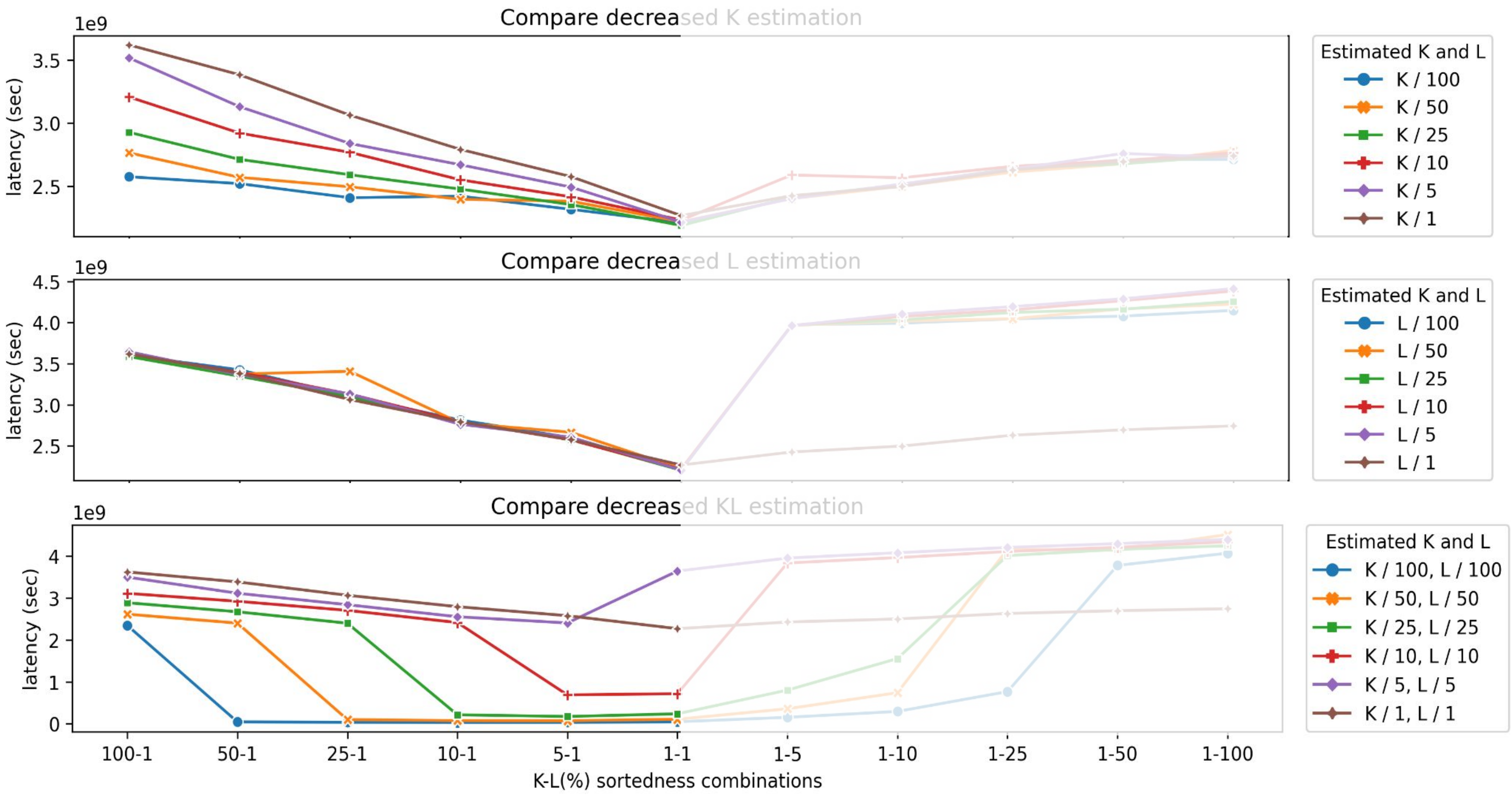


8G RAM
v.s.
32G RAM

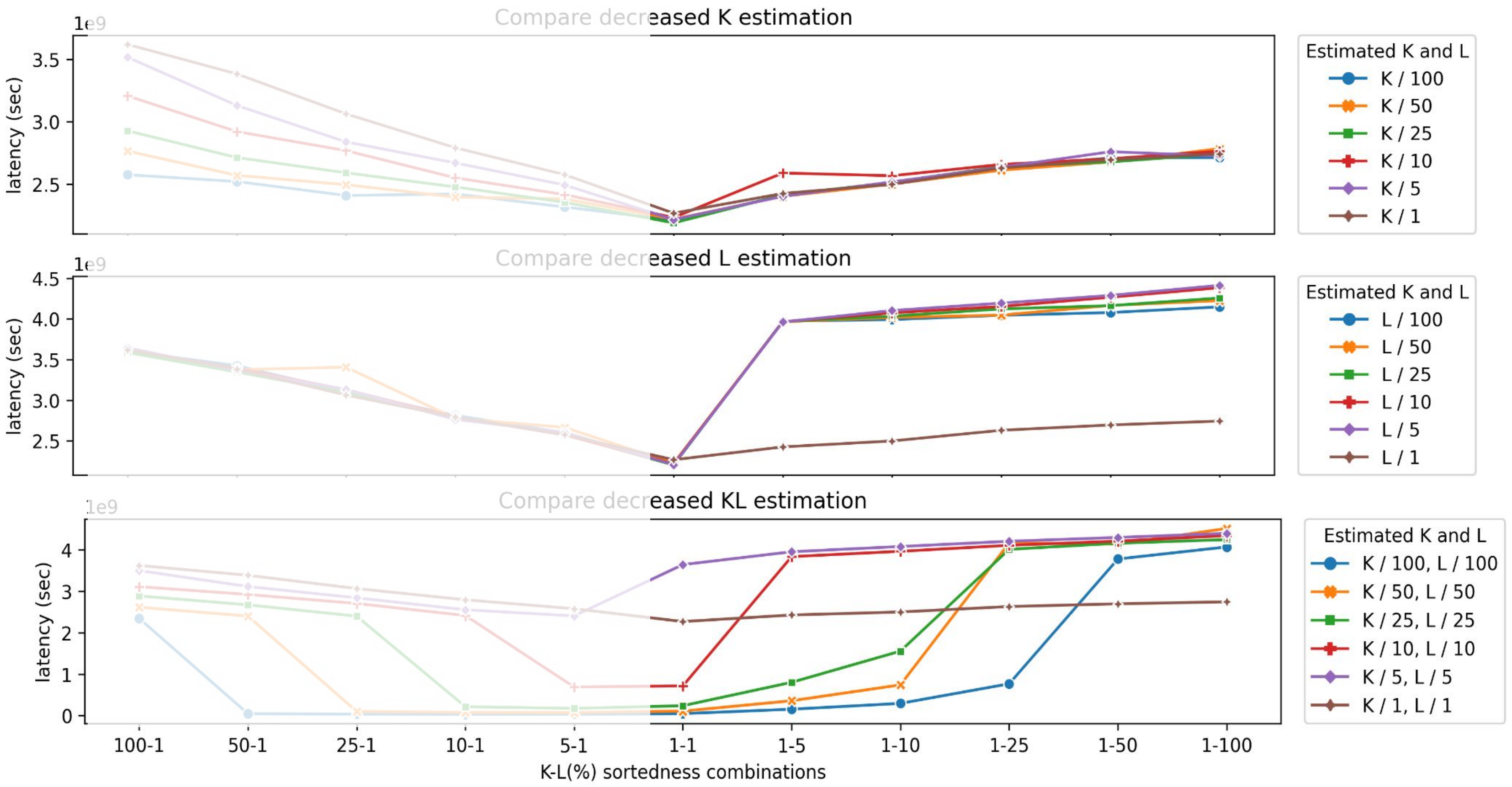



Sorting Algorithm = kl_sort
Sorting Algorithm = std_stable
Sorting Algorithm = radix_sort
Sorting Algorithm = merge_sort
Sorting Algorithm = quick_sort
Sorting Algorithm = tim_sort
K (%)
L (%)
0
20
40
60
80
100
Sorting Algorithm
kl_sort
std_stable
radix_sort
merge_sort
quick_sort
tim_sort
Latency (sec)
0.8
1.6
2.4
3.2
4.0

Estimate a Lower K or L



Estimate a Lower K or L



Conclusion: 3min

Conclusion

- **K** is a crucial factor in all experiments.
- **Quick sort** outperforms under nearly-sorted data.
- (K, L)-sort works best with a **shrunken K from BoDS** under nearly-sorted data.

Remaining experiments and analysis

- **Coefficient analysis** over sorting algorithms' big-o complexity.
- Generate scrambled workload with std::shuffle().

- Experiments on **nearly-sorted L** from 0.0001% to 1%.
- Implement **(K, L) estimation** by exponential search for minimal ones without failure.
- Provide the tradeoff analysis on (K, L) estimation methods.

- At least **three trials** on each experiments.
- Implement additional sorting algorithms: **K-sort**, **Spreadsort**, other stable sortings.
- **Space complexity** analysis and the actual memory footprint record.

Expected results

- Big-O complexity with coefficient on the highest rank for each sorting algorithm.
- Clarify the difference between std::stable_sort() and mergesort.
- Prove that **K-sort** is suitable for nearly sorted workload,
  while **radix sort** and **quick sort** is for general usage.
- Show the tradeoff between time and space among sorting algorithms.
- Provide hyper-parameters for tuning K estimation.
- Draw all figures with standard deviation shadow.
- Add args names for API to allow out of order commends

Interesting & Challenging Experience

1. Knowing the exploration and design process, how can we find a state-of-the-art **adaptive index sorting algorithm** that beats all baselines?
2. More questions after this project:
    a. How to estimate L in a wild field?
    b. Would the (K, L) estimation takes longer than the saved sorting time?
    c. If the total (K, L) estimation and sorting time is longer than baselines, (K, L) sortedness might not be a useful benchmark.

Advice on the technical aspect

1. Use **int** data type to avoid machine-level optimization in C++ compiler.
2. Use **Linux** system to allow clearing on RAM and swap files.
3. Use mixed workloads including writing operations (inserts, updates, deletes).
4. Use self-craft or multi generators to **decouple the dependency on libraries**.
5. Charging or not for the hardware device affects the performance a lot.

Evaluating Sorting Algorithms with Varying Data Sortedness

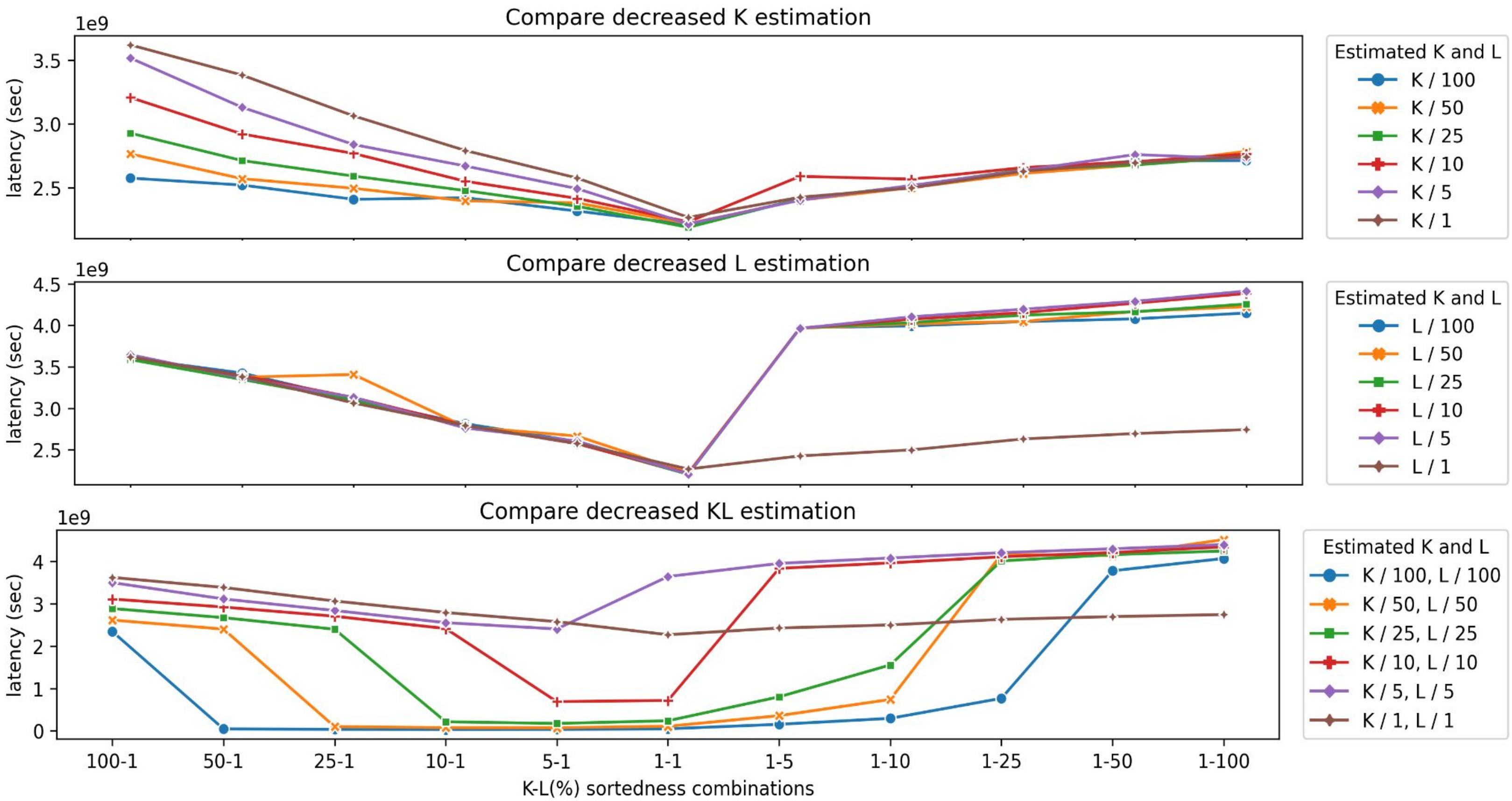
Wei-Tse Kao, I-Ju Lin

May 2nd, 2023

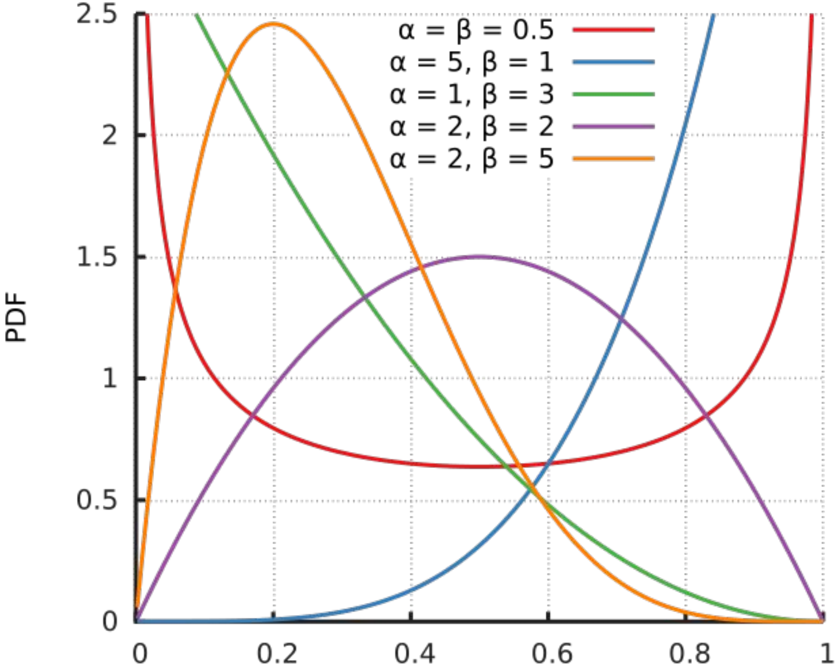
Related Class Sessions

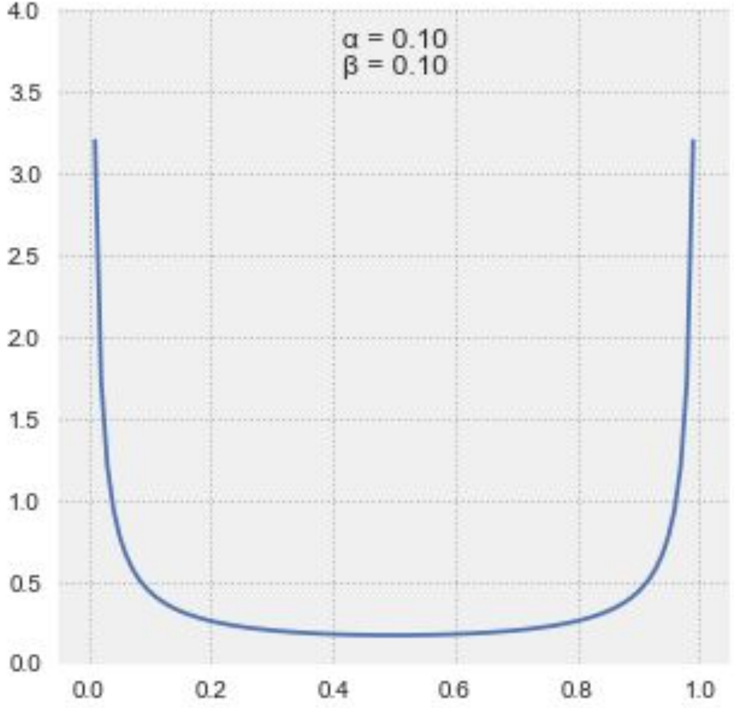
- Class 12: Adaptive Radix Trees (student presentation S2)
  - The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases, ICDE 2013. (R2)
- Class 13: Adaptive Indexing & Cracking (student presentation S3)
  - Adaptive Adaptive Indexing, ICDE 2018. (Technical Question T3)
  - Self-organizing Tuple Reconstruction in Column-stores, SIGMOD 2009.
- Class 14: Guest Lecture on Sortedness-Aware Indexing: Aneesh Raman
  - Indexing for Near-Sorted Data, ICDE 2023.
  - BoDS: A Benchmark on Data Sortedness, TPCTC 2022.
- Class 23: Guest Lecture on Learned Index: Ryan Marcus
  - LSI: A Learned Secondary Index Structure, SIGMOD 2022.
  - Benchmarking Learned Indexes, VLDB 2021.

Result: Estimate a Lower K or L



Beta Distribution from Wikipedia





Workload Generator

Algorithm 1: Generate (K, L, B)-sorted keys

Input: Fully sorted array arr, $N \geq 0; K \geq 0; L \geq 0; B(\alpha, \beta)$, $num\_tries1 > 0, num\_tries2 > 0$
Output: (K, L, B)-sorted array arr

```
Sources ← Generate_Sources(N, K) ;                        /* using Algorithm 2 */
dest <>;                                                  /* set of destinations */
left <>;                                                  /* set of left out sources */
for x ∈ Sources do
    while num_tries1 > 0 do
        r ← Pick_dest(N, K, x, B);                        /* using Algorithm 3 */
        num_tries1 ← num_tries1 − 1;
        if r ∈ dest or r ∈ Sources;                       /* destination already used */
         then
            if num_tries1 == 0;          /* retrials exhausted, moving r to leftovers */
             then
                insert r to left;
            end
            continue;
        else
            insert r in dest;
            swap arr[x] with arr[r];
            break;
        end
    end
end
for x ∈ left;                                  /* randomized re-attempt for leftovers */
 do
    while num_tries2 > 0 do
        r ← Pick_dest(N, K, x, B);                        /* using Algorithm 3 */
        num_tries2 ← num_tries2 − 1;
        if r ∈ dest or r ∈ Sources;                       /* destination already used */
         then
            continue;
        else
            insert r in dest;
            swap arr[x] with arr[r];
            remove x from left;
            break;
        end
    end
end
Perform_Brute_Force(arr, left, dest, L) ;                 /* using Algorithm 4 */
```

Reimplementation

Indexing for Near-Sorted Data

Aneesh Raman, *Boston University*, aneeshr@bu.edu

Subhadeep Sarkar, *Boston University*, ssarkar1@bu.edu

Matthaios Olma, *Microsoft Research*, maolma@microsoft.com

Manos Athanassoulis, *Boston University*, mathan@bu.edu

***Abstract*—Indexing in modern data systems facilitates efficient query processing when the selection predicate is on an indexed key. As new data is ingested, indexes are gradually populated with incoming entries. In that respect, *indexing can be perceived as the process of adding structure* to incoming, otherwise unsorted data. Adding structure, however, comes at a cost. Instead of simply appending the incoming entries, we insert them into the index. If the ingestion order matches the indexed attribute order, the ingestion cost is entirely redundant and can be avoided altogether (e.g., via bulk loading in a B^{+}-tree). However, classical tree index designs do not benefit when incoming data comes with an implicit ordering that is *close to* being sorted, but *not* fully sorted.**

In this paper, we study how indexes can exploit *near-sortedness*. Particularly, we identify *sortedness as a resource* that can accelerate index ingestion. We propose a new sortedness-aware (SWARE) design paradigm that combines *opportunistic bulk loading*, *index appends*, *variable node fill and split factors*, and an *intelligent buffering scheme*, to optimize ingestion and read queries in a tree index in the presence of near-sortedness.

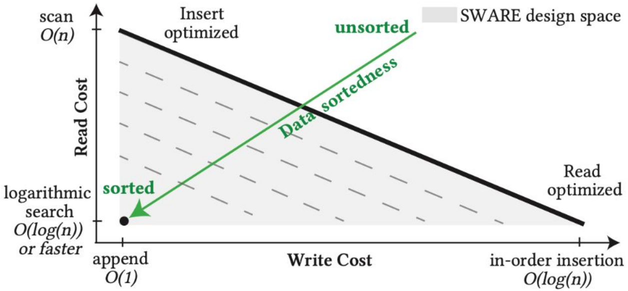


Fig. 1: State-of-the-art indexing and data organization techniques pay a higher write cost in order to store data as sorted (or, in general, more organized) and offer efficient reads. Since the goal of indexing is to store the data as sorted, we ideally expect that ingesting *near-sorted* data would be more efficient, which is not the case. We introduce the SWARE meta-design that offers better performance as data exhibit higher degree of sortedness.

the figure). On the other extreme, if read queries are infrequent

https://github.com/BU-DiSC/sware

Is this true?

Choice of Sorting Algorithm. To reduce the cost of reads, we sort the buffer after every flush. Ideally, we want the sorting cost to be minimal to attain the maximum benefits of the SWARE paradigm. While any sorting algorithm that leverages data sortedness (e.g., TimSort [44], Replacement Selection Sort [34]) can be used, here we consider three sorting algorithms: (i) *quicksort*, as it is common and has minimal space requirements, (ii) (K, L)-*adaptive sorting* [7], as it aggressively takes into account pre-existing data sortedness with $O(K + L)$ space usage, and (iii) *mergesort* (specifically, the C++ standard library `std::stable_sort`), as it maintains relative order of duplicate values with $O(n)$ space usage. Because we need to maintain the relative order of duplicates, we are constrained between mergesort and (K, L)-adaptive sorting. Our experimental analysis shows that for low data-sortedness, mergesort outperforms (K, L)-adaptive sorting (in fact, (K, L)-adaptive sorting fails for significantly high values of K or L). However, for $K < 20\%$ or $L < 5\%$, their performance is similar, and we opt for (K, L)-adaptive sorting due to its **smaller space requirements** $(K + L < n)$ [7]. So, when the estimated (meta-data) values are $K < 20\%$ or $L < 5\%$ of the buffer size we employ (K, L)-adaptive sorting while using `std::stable_sort`), otherwise.

In

Aneesh Raman
Boston University
aneeshr@bu.edu

Abstract—Indexing in moder
query processing when the sele
key. As new data is ingested, ind
incoming entries. In that respec
process of adding structure to in
Adding structure, however, co
appending the incoming entrie
If the ingestion order matches
ingestion cost is entirely redund
(e.g., via bulk loading in a B+-t
designs do not benefit when inc
ordering that is *close to* being
In this paper, we study
sortedness. Particularly, we ide
can accelerate index ingestion
aware (SWARE) design parad
bulk loading, *index appends*, *v*
and an *intelligent buffering sc*
read queries in a tree index in

(K, L) Sort by Binary Min Heap

R:

9	4	3	2	5	6	8	7	1	10

R_4: 2-close to being 3-globally sorted

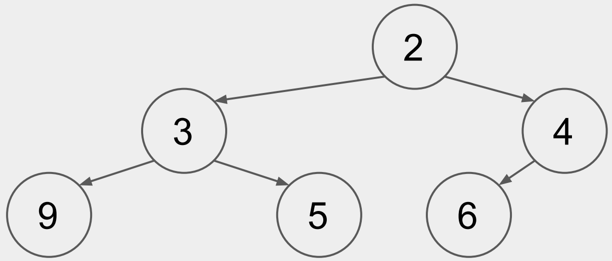
k = 2, l = 3.

Heap S

Heap G

Insert the first (k+l+1) = 6 tuples into S:

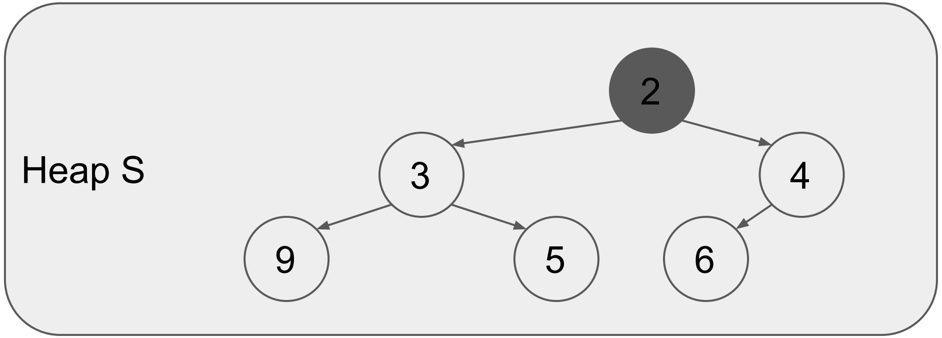
Heap S



Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k+\ell+1$ tuples $(R[1], \ldots, R[k+\ell+1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S|+1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap



i_write = 1;

for i_read = |S| + 1 = 7 to n = 10:

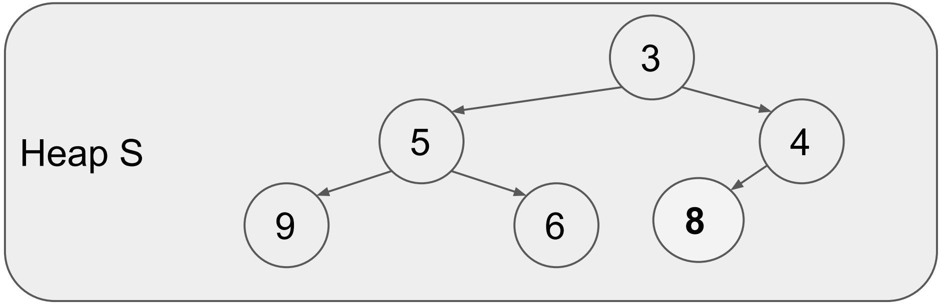
    i_read = 7;

    last_written = 2;

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap



i_read = 7, last_written = 2, i_write = 1;

Write last_written = 2 to TMP[i_write] = TMP[1]

TMP:									
	2								

9	4	3	2	5	6	8	7	1	10

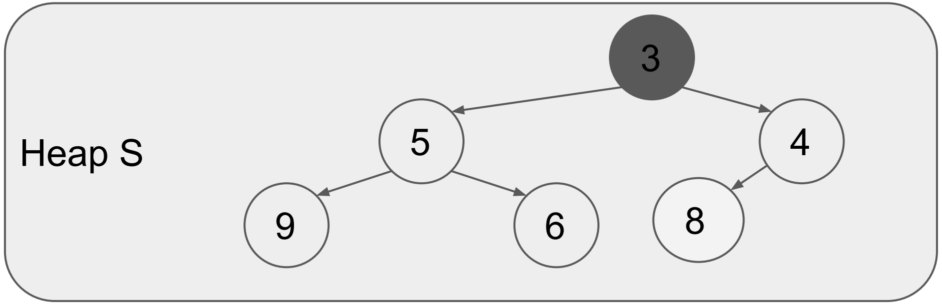
R[i_read] = R[7] = 8

If (R[i_read] = 8) >= (last_written = 2) then
Insert R[i_read] = 8 into S;

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap



i_read = 8, last_written = 3, i_write = 2;

Write last_written = 3 to TMP[i_write] = TMP[3]

TMP:

2	3								

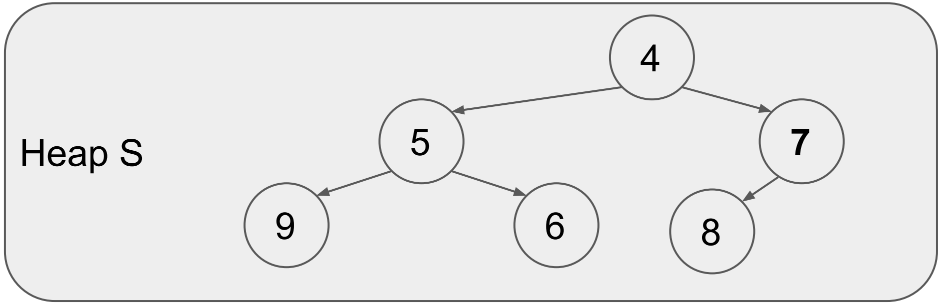
9	4	3	2	5	6	8	7	1	10

R[i_read] = R[8] = 7

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap



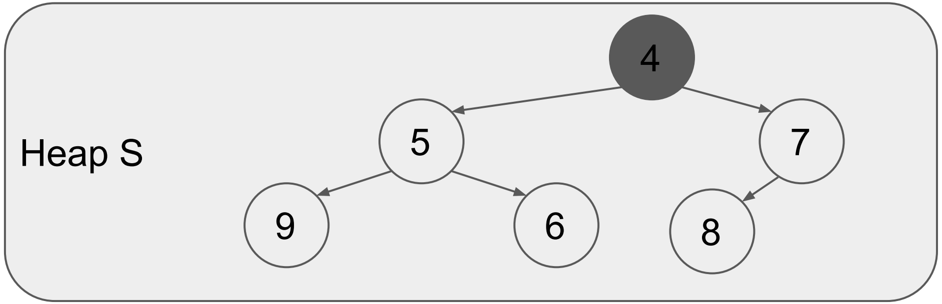
If (R[i_read] = 7) >= (last_written = 3) then

Insert R[i_read] = 7 into S;

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap



i_read = 9, last_written = 4, i_write = 3;

Write last_written = 4 to TMP[i_write] = TMP[3]

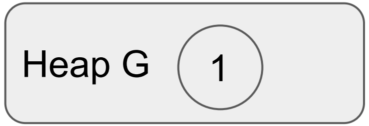
TMP:

2	3	4							

9	4	3	2	5	6	8	7	1	10

else

Insert R[i_read] = 1 into G;

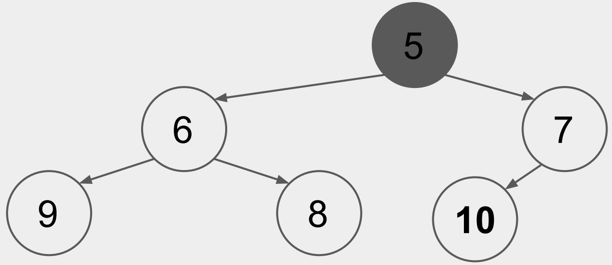


Algorithm 1 (Sorts a (k,ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k+\ell+1$ tuples $(R[1],\dots,R[k+\ell+1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S|+1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap

Heap S



i_read = 10, last_written = 5, i_write = 4;

Write last_written = 5 to TMP[i_write] = TMP[4]

TMP:										
	2	3	4	5						

9	4	3	2	5	6	8	7	1	10

If (R[i_read] = 10) >= (last_written = 5) then

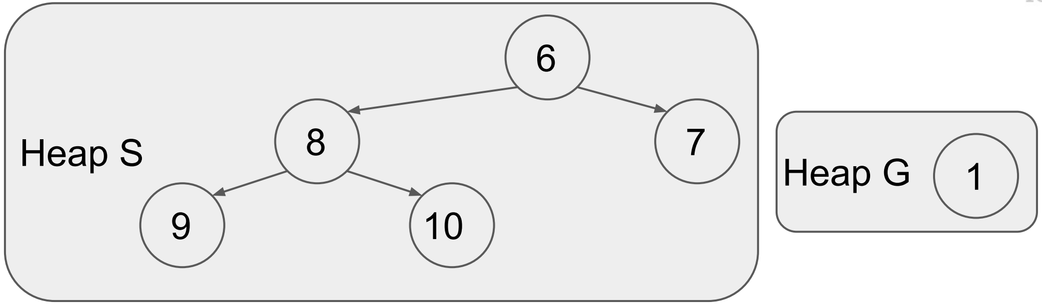
Insert R[i_read] = 10 into S;

end for

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap



Append all tuples in S to TMP in sorted order;

TMP:										
	2	3	4	5	**6**	**7**	**8**	**9**	**10**	

i_write = 1;

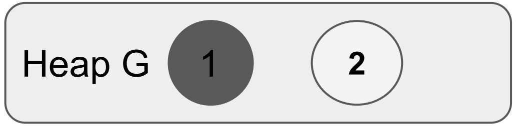
for i_read = 1 to (n - |G| = 10 - 1 = 9):

    i_read = 1;

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap



i_read = 1, x = 1, i_write = 1;

else

Write (x = 1) to OUT[i_write] = OUT[1]

Remove (x = 1) from G

Add (TMP[1] = 2) to G

TMP:

2	3	4	5	6	7	8	9	10	

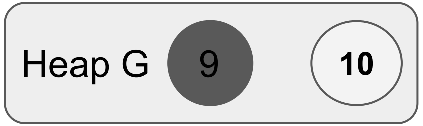
OUT:

1									

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k+\ell+1$ tuples $(R[1], \ldots, R[k+\ell+1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S|+1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap



Similarly till the end of loop:

i_read = 9, x = 9, i_write = 9;

else

Write (x = 9) to OUT[i_write] = OUT[9]

Remove (x = 9) from G

Add (TMP[9] = 10) to G

end for

TMP:	2	3	4	5	6	7	8	9	**10**	

OUT:	1	2	3	4	5	6	7	8	9	

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k+\ell+1$ tuples $(R[1], \ldots, R[k+\ell+1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S|+1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order

(K, L) Sort by Binary Min Heap

Append all tuples in G to OUT in sorted order.

OUT:										
	1	2	3	4	5	6	7	8	9	**10**

Best Case:	Worst Case:	Memory:	Stable:
O(n log(n))	O(n log(n))	O(n)	Yes

, where heap extractMin() and insert() takes O(log n).

Algorithm 1 (Sorts a (k, ℓ)-nearly sorted relation R.)

create two binary heaps S, G
insert the first $k + \ell + 1$ tuples $(R[1], \ldots, R[k + \ell + 1])$ into S
$i_{write} \leftarrow 1$
for $i_{read} = |S| + 1$ to n **do** {first pass}
  if $S = \emptyset$ **then**
    FAIL
  end if
  $last\_written \leftarrow \min\{x \in S\}$
  write $last\_written$ to $TMP[i_{write}]$
  $S \leftarrow (S \setminus \{last\_written\})$
  $i_{write} \leftarrow i_{write} + 1$
  if $R[i_{read}] \geq last\_written$ **then**
    insert $R[i_{read}]$ into S
  else
    insert $R[i_{read}]$ into G
  end if
end for
append all tuples in S to TMP, in sorted order
$i_{write} \leftarrow 1$
for $i_{read} = 1$ to $n - |G|$ **do** {second pass}
  $x \leftarrow \min\{y \in G\}$
  if $x > TMP[i_{read}]$ **then**
    write $TMP[i_{read}]$ to $OUT[i_{write}]$
  else
    write x to $OUT[i_{write}]$
    $G \leftarrow (G \setminus \{x\}) \cup \{TMP[i_{read}]\}$
  end if
  $i_{write} \leftarrow i_{write} + 1$
end for
append all tuples in G to OUT, in sorted order