Huskysort

R C HILLYARD*, College of Engineering, Northeastern University, USA
YUNLU LIAOZHENG†, College of Engineering, Northeastern University, USA
SAI VINEETH K R‡, College of Engineering, Northeastern University, USA

Much of the copious literature on the subject of sorting has concentrated on minimizing the number of comparisons and/or exchanges/copies. However, a more appropriate yardstick for the performance of sorting algorithms is based on the total number of array accesses that are required (the "work"). For a sort that is based on divide-and-conquer (including iterative variations on that theme), we can divide the work into linear, i.e. $O(N)$, work and linearithmic, i.e. $O(N \log N)$, work. An algorithm that moves work from the linearithmic phase to the linear phase may be able to reduce the total number of array accesses and, indirectly, processing time. This paper describes an approach to sorting which reduces the number of expensive comparisons in the linearithmic phase as much as possible by substituting inexpensive comparisons. In Java, the two system sorts are dual-pivot quicksort (for primitives) and Timsort for objects. We demonstrate that a combination of these two algorithms can run significantly faster than either algorithm alone for the types of objects which are expensive to compare. We call this improved sorting algorithm Huskysort.

CCS Concepts: • Theory of computation → Sorting and searching; Data structures and algorithms for data manipulation; • Software and its engineering → Data types and structures; • Computer systems organization → Multicore architectures;

Additional Key Words and Phrases: Sorting, Quicksort, Timsort, Insertion sort, Complexity, Comparison Sorting

ACM Reference Format:
R C HILLYARD, Yunlu Liaozheng, and Sai Vineeth K R. 2020. Huskysort. ACM Trans. Graph. 37, 4, Article 111 (August 2020), 9 pages. https://doi.org/10.1145/3366666.3367534

1 INTRODUCTION

With the vast increase in the size of the data world, efficient computing algorithms play a crucial role in processing data. From the earliest days of computing, sorting has been one of the most studied areas of research even though it appears to be a straightforward and familiar topic. The rise of data volume and complexity has made clear a need for new approaches to traditional sorting techniques. Within the realm of comparison-sort algorithms, many aspects of the algorithm can be significant: stability, extra memory, partitioning, merging, exchanging (swapping), copying, and, of course, the comparison itself.

Even though sorting appears to be a settled area of research, yet new improvements are still being made: [Jugé [n.d.]; McIlroy 1993; Peters 2002; Yaroslavskiy 2009]. There is, therefore, an abundance of sorting techniques available. Nevertheless, there is still room for improved versions in regard to computation, memory, time, and space complexity. In a recent paper [Zhang et al. 2016], the authors conclude that quicksort is indeed the fastest general-purpose sorting algorithm, especially when appropriate care is taken to avoid the quadratic worst-case with several other improvements (e.g. dual-pivot quicksort)[Yaroslavskiy 2009] over the traditional version [Hoare 1961]. However, quicksort does not perform as well as some variations of merge sort, especially Timsort [Peters 2002], when the dataset is partially sorted. Timsort, a hybrid stable sorting algorithm derived from merge sort, is now available in the Python and Java libraries as the default object-sorting utility.

We determined to test the efficacy of combining quicksort with Timsort for object types that are costly to compare. Such types include String, LocalDateTime, BigInteger, BigDecimal, and many user-defined types of this kind.

We review existing methods of sorting in Section 2 and go through the Huskysort algorithm in Section 3. Implementation options and their likely impact are explored in Section 4. We describe our Test Cases in Section 5 and present results in Section 6, with outcomes summarized in Section 7.

2 BACKGROUND

Timsort is ideally suited to the partially ordered data which is common in the real-world business applications. For this use case it is easily the fastest standard algorithm. It is less well suited, however, for another use case, which is unordered data. Huskysort is designed specifically for this use case: unordered data consisting of objects which are relatively expensive to compare.

Before talking about our approach let’s look at insertion sort and Timsort in more detail. The key feature of both of these methods is that the number of comparisons (and swaps, in the case of insertion sort) is proportional to the number of inversions. Because insertion sort is an "elementary" sort (it reduces the complexity of the problem by iteratively partitioning by one element at a time), it is only suitable for small arrays. It traverses the array, moving each new element encountered into its proper place in the ordered partition, which therefore grows by one element in each iteration until the whole array is ordered.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Permissions requests should be addressed to: permissions@acm.org. For questions about reproducing figures, contact permissions@acm.org. 

ACM Trans. Graph., Vol. 37, No. 4, Article 111. Publication date: August 2020.
Timsort is a variation of bottom-up merge sort which takes advantage of existing ordered runs. Because of this, it is very efficient for partially-ordered arrays. Additionally, like merge sort, Timsort is stable. Timsort’s worst case is as $O(N + N \log p)$, where $p$ is the number of runs [Auger et al. 2018].

One of the authors (Hillyard) had set a question in a mid-term exam (Fall, 2018) about a hypothetical algorithm that created a 64-bit hash code for a date-time object and then used quicksort to sort the array. Afterward, it became apparent that it was not such a crazy idea after all. A quick implementation showed that, indeed, we could see an improvement over the system sort (Timsort) for sorting objects. The essence of the algorithm lies in the fact that quicksort can sort an array of (64-bit) longs very quickly. 64 bits affords sufficient room to encode many objects uniquely, but even if the coding is not unique, post-sorting an array of objects which are quasi-ordered can be very fast using Timsort. The key innovation of the algorithm was developed by Liaozheng: swap the objects at the same time as swapping the longs.

3 ALGORITHM

3.1 Overview

The strategy of Huskysort (see Flow 1) is as follows (assuming that the dataset is presented as an array $x_s$ of type $X$): see Algorithm 1

![Fig. 1. Huskysort Flow](image1)

Step 1: a Create a long[] array ($longs$) of the same size as $x_s$;
b for each element $x$ of $x_s$, set the corresponding element of $longs$ to $x.huskyCode()$.

Step 2: Sort array $longs$ using quicksort (in practice, we have found that Introstart [MUSSER 1997] works well) in such a way that when elements $i, j$ of $longs$ are exchanged, then elements $i, j$ of $x_s$ are also exchanged; See Flow 2

Step 3: Finally, when quicksort is finished and, if necessary, we run Timsort on the array $x_s$ to ensure that its elements are truly in order.

For the details of the husky-coding itself, see Algorithm 4

3.2 Discussion of Husky Encoding

Step 1b requires the definition of a 64-bit hash function (huskyCode) whose properties are as follows: (refer to Algorithm 2)

- If $x = y$, then $x.huskyCode() = y.huskyCode()$;
- If $x > y$, then:
  - either $x.huskyCode() <= y.huskyCode()$ (with probability $p$)
  - or $x.huskyCode() > y.huskyCode()$ (with probability $1 - p$);

We interpret this rule as follows: if $x$ and $y$ are the same ($x.equals(y)$ in Java), their huskyCode should be the same. If $x > y$, we normally expect that $x.huskyCode() > y.huskyCode()$, i.e. the sort on the husky-coded longs will not invert $x$ and $y$. However, when $x > y$ while $x.huskyCode() < y.huskyCode()$, then an inversion will be introduced. And, when $x > y$ and $x.huskyCode() = y.huskyCode()$,
then an inversion may be introduced because quick-sort is not stable [Xiang 2011]

For a successful encoding, the value of \( p \), the probability of a potential inversion, should be less than some critical probability, \( p_{crit} \). Leaving a more detailed discussion of \( p_{crit} \) until later, we note that, for some domains of \( X \), we can definitely assert that \( p = 0 \). We refer to this as a “perfect” encoding. In such a case, we can skip step 3 of the algorithm because, when \( p = 0 \), there will be no inversions remaining after quick-sorting according to the huskyCodes. An example of such a domain is date-time values with a resolution of one nano-second over a period of 128 years. For strings of English case-independent letters (no punctuation or numbers), i.e. 5-bit ASCII, words of 12 (or fewer) characters in length can also be coded uniquely. For case-dependent strings (6-bit ASCII), we can encode perfectly if the lengths are all 10 characters or less. In step 3, it might be acceptable to use insertion sort instead of Timsort. In any event, assuming that the huskyCode is as accurate as required, there will be very few inversions (elements out of place) remaining after the quicksort phase. This implies that the second sort can be performed in very close to linear time. The results have been very satisfactory. For example, taking an array of 100,000 English words chosen randomly from a corpus of 100,000 words [Goldhahn et al. 2012], we have found, for example in one run, that Huskysort takes a normalized time of 0.93, compared with 1.00 for quicksort and 1.45 for Timsort (For further data, please refer to table 2.)

### 3.3 Discussion of \( p_{crit} \)

When \( p > 0 \), there is a finite probability of any pair of elements remaining inverted after the first sorting pass. For an array of \( N \) randomly ordered elements, the average number of inversions before sorting is \( X = N(N - 1)/4 \). The number of inversions remaining after the first pass is simply \( pX \). And, for Timsort or insertion sort, the time to re-sort the array will be \( t = k(N + pX) \) where \( k \) is some system-dependent constant. The total time to sort the array is made up of these three time-consuming steps where \( k_1, k_2, k_3 \) are system- and algorithm-dependent constants:

**Step 1:** \( T_1 \) is the time to encode the elements:

\[
T_1 = k_1N
\]

**Step 2:** \( T_2 \) is the (average) time to quick-sort the huskyCode array while also swapping the element array itself:

\[
T_2 = 2k_2N\ln N
\]

**Step 3:** \( T_3 \) is the time to Timsort the element array (only required if \( p > 0 \)):

\[
T_3 = k_3(N + pX)
\]

Provided that the total time \( T = T_1 + T_2 + T_3 \) is less than the time required to use Timsort on the original element array, we have a win. The actual values of \( p_{crit} \) are very much dependent on the machine running the sort. We have not calculated a typical value for \( p_{crit} \) but, even for the worst-case: Chinese characters using UTF8 encoding, we still find very significant gains. Please refer to table 4

### 3.4 Explanation of Working

So, why does this work? Well, first of all, we observe that 64-bit word length in today’s computers is the norm. This allows for an encoding which has a low, possibly zero, value of \( p_{crit} \).

Even though a Unicode string can encode only the first 3 15/16ths characters in 64 bits, this is extremely helpful in getting the elements close to their final order.

Here, we may compare the method of Huskysort with that of ShellSort [Sedgewick 1996]

ShellSort acts exactly like a series of insertion sorts but, whereas each compare/swap in insertion sort is limited to resolving one inversion only, so in ShellSort, \( h \) inversions can be resolved in one compare/swap (where we are \( h \)-sorting the array). So, Shellsort’s primary mechanism is to try to fix as many inversions as possible before taking the subsequent step.

In the same way, Huskysort fixes as many inversions as possible before doing the final step. Note that it isn’t necessary for the encoding to be perfect. But it should be good.

Another way of looking at it is that running quicksort or merge sort (or their variations) will require \( O(N\log N) \) comparisons. Merge sort, like insertion sort, is linear when the number of inversions is small and grows with \( O(N) \). But merge sort does use extra memory. But values such as 64-bit integers (int or long), doubles, can be compared extremely quickly in modern machines—far quicker than the time required to follow a reference (pointer) to an object in memory. In Java, this distinction is described by contrasting primitives with objects. Primitives can be quick-sorted very fast and of course, long is primitive.

Exchanging (swapping), on the other hand, takes the same amount of time whatever type you are sorting because swapping an object reference is just the same as swapping any 64-bit word. The other reason why quicksort is so fast is that it is cache-friendly. So, we perform a fast quicksort on the array of longs. The only difference between the Husky version of quicksort and standard quicksort is that the swap operation must actually do two swaps: one of the longs and one of the object references. But, again, swapping is fast because elements of both arrays (i.e. both longs and refs) have good chances of being cached.

The extra time (and space) required to perform the encoding upfront, which is \( O(N) \), is more than offset by the faster comparisons of longs as opposed to comparisons of objects.

After the first pass (using quicksort), we can easily determine whether our encoding was perfect—in which case, there will be no remaining inversions. We do this for non-sequence classes via a simple constant-time lookup. For sequences (in practice, these are always Strings), we check each sequence’s coding perfection (according to its length), and as soon as we find a sequence that cannot be encoding perfectly, we stop checking.

If the coding was perfect, we simply skip the final pass.

Given the notion of hashing that the algorithm uses, it would seem natural to call is Hash Sort. Unfortunately, that name was already taken for quite a different algorithm [Gilreath 2004].
Algorithm 1: HuskySort.sort

Input: xs as an array of object X which extends Comparable<X>
1. coding = huskyCoder.huskyEncode(xs);
2. longs = coding.longs;
3. Introsort(xs, longs, 0, longs.length, 2 * floor_lg(xs.length));
4. if coding.perfect then
5. return;
6. Arrays.sort(xs);

Algorithm 2: huskyEncode

Input: xs as an array of object X
Output: A new Coding object
1. result = new long[xs.length];
2. for i = 0 to xs.length do
3. result[i] = huskyEncode(xs[i]);
4. return new Coding(result, perfect());

Algorithm 3: huskyEncode (for character sequence)

Input: xs as an array of CharSequence X
Output: A new Coding object
1. isPerfect = true;
2. result = new long[xs.length];
3. for i = 0 to xs.length do
4. X x = xs[i];
5. if isPerfect then
6. isPerfect = perfectForLength(x.length());
7. result[i] = huskyEncode(xs[i]);
8. return new Coding(result, isPerfect());

Algorithm 4: stringToLong

Input:
str as String,
maxLength as the max length can be encoded;
bitWidth as how many bits will one char be encoded into;
mask as a modifier for different encoding;

Output: A new Coding object
1. length = Math.min(str.length(), maxLength);
2. padding = maxLength - length;
3. result = 0;
4. if mask == 0 then
5. for i = 0 to length do
6. result = result * bitWidth | str.charAt(i);
7. else
8. for i = 0 to length do
9. result = result * bitWidth | str.charAt(i) & mask;
10. result = result * bitWidth * padding;
11. return result;

4 IMPLEMENTATION

4.1 Data Source

The sources for the Strings (English and Chinese) are retrieved from Leipzig Corpora Collection [Goldhahn et al. 2012].

In each case, we take data from the appropriate sentences.txt file and break it up into strings as shown in code 1.

The sample of data can be seen here 3

Fig. 3. Strings Data Example

Example of English strings:
words = [String[22865][1158]
0 = "Consider"
1 = "the"
2 = "extras"
3 = "not"
4 = "usually"
5 = "included"
6 = "with"
7 = "any"
8 = "FREE"
9 = "car"
10 = "offer"
11 = "CDV"
12 = "LDV"
13 = "For"
14 = "noise"
15 = "which"
16 = "is"
17 = "persistent"
18 = "but"
19 = "occurs"
20 = "en"
21 = "such"
22 = "an"
23 = "irregular"
24 = "basis"
25 = "that"
26 = "case"
27 = "officer"
28 = "unable"
29 = "to"

Example of Chinese strings:
words = [String[24017][1388]
0 = "Aaron"
1 = "如果您指的是"
2 = "过滤效果"
3 = "Brikhorst"
4 = "Accept"
5 = "Ranges"
6 = "bytes"
7 = "Last"
8 = "Modified"
9 = "Fr"
10 = "Mom"
11 = "Sat"
12 = "Acid"
13 = "Alkaline"
14 = "Exhaust"
15 = "Fal"
16 = "System"
17 = "Solvent"
18 = "Treatment"
19 = "General"
20 = "一般气体稀能贝托系统"
21 = "制备中产生的一股有害气体"
22 = "DMF-单甲基"
23 = "fumarate"
24 = "也即氢酸二甲酯"
25 = "Adobe"
26 = "表示让全行师生能享有最新的"
27 = "Creative"
28 = "Suite"
29 = "软便与技术"
4.2 System Environment

The configurations of the system that is used for implementation is shown in figure 4. The Java Virtual Machine running is 1.8.0_152.

4.3 Implementation of Algorithm

- Huskysort (top level)

```java
public void sort(final X[] xs) {
    // NOTE: we start with a random shuffle
    // This is necessary if we might be sorting
    // a pre-sorted array. Otherwise, we usually don’t need it.
    if (mayBeSorted) Collections.shuffle(Arrays.asList(xs));
    // NOTE: First pass where we code to longs
    // and sort according to those.
    final Coding coding = huskyCoder.huskyEncode(xs);
    final long[] longs = coding.longs;
    introSort(xs, longs, 0, longs.length, 2 * floor_lg(xs.length));
    // NOTE: Second pass (if required) to fix
    // any remaining inversions.
    if (coding.perfect) {
        return;
    }
    Arrays.sort(xs);
}
```

- Huskysort (Husky Encoding)

```java
default Coding huskyEncode(X[] xs) {
    long[] result = new long[xs.length];
    for (int i = 0; i < xs.length; i++) result[i] = huskyEncode(xs[i]);
    return new Coding(result, perfect());
}
```

- Unicode to Long:

```java
private static long unicodeToLong(final String str) {
    return stringToLong(str, MAX_LENGTH_UNICODE, BIT_WIDTH_UNICODE, MASK_UNICODE) >>> 1;
}
```

- String To Long:

```java
private static long stringToLong(final String str, final int len, final int bitWidth, final int mask) {
    long result = 0;
    for (int i = 0; i < len; i++) {
        result |= (str.charAt(i) & mask) << (bitWidth * i);
    }
    return result;
}
```

- Ascii to Long:

```java
private static long asciiToLong(final String str) {
    return stringToLong(str, MAX_LENGTH_ASCII, BIT_WIDTH_ASCII, MASK_ASCII);
}
```

- Constants:

```java
private static final int BITS_LONG = 64;
private static final int BIT_WIDTH_UNICODE = 16;
private static final int MAX_LENGTH_UNICODE = BITS_LONG / BIT_WIDTH_UNICODE;
private static final int MASK_SHORT = 0xFFFF;
private static final int MASK_UNICODE = MASK_SHORT;
```

Here are the VM options in use:

- `-XX:ReservedCodeCacheSize=240m`
- `-XX:+UseCompressedOops`
- `-XX:UpdateNonSafepointSSTable`
- `-XX:SoftRefNANO=8`
- `-XX:NewGenMinSurvivorSize=50`
- `-Dsun.io.useCanonCaches=false`
- `-Djava.net.preferIPv4Stack=true`
- `-Dhttp.acute.hunleng_disabledSchemes=""`
- `-Djava.io.tmpdir=C:\Users\Anam\Java\error\error\java_error\java_error\temp ordinance`

Listing 1. Data Source

| System used: | MacBook Pro |
|--------------|-------------|
| Model Name:  | MacBookPro14,3 |
| Processor Name: | Quad-Core Intel Core i7 |
| Processor Speed: | 2.8 GHz |
| Number of Processors: | 1 |
| Total Number of Cores: | 4 |
| L2 Cache (per Core): | 256 KB |
| L3 Cache: | 6 MB |
| Hyper-Threading Technology: | Enabled |
| Memory: | 16 GB 2133 MHz LPDDR3 |
| Boot ROM Version: | 428.0.0.0.0 |
| SMC Version (system): | 2.404 |

Fig. 4. System Environment

Listing 2. Husky Overview

- HuskyOverview

```java
default public Coding(long[] longs, boolean perfect) {
    this.longs = longs;
    this.perfect = perfect;
}
```

Listing 3. HuskyEncode

- Class to combine the long codes for an array of objects with a determination of coding perfection.

```java
public class Coding {
    public Coding(long[] longs, boolean perfect) {
        this.longs = longs;
        this.perfect = perfect;
    }
    public final long[] longs;
    public final boolean perfect;
}
```

Listing 4. HuskyEncode

- Unicode to Long:

```java
private static long unicodeToLong(final String str) {
    return stringToLong(str, MAX_LENGTH_UNICODE, BIT_WIDTH_UNICODE, MASK_UNICODE) >>> 1;
}
```

Listing 5. unicodeToLong

- String To Long:

```java
private static long unicodeToLong(final String str, final int len, final int bitWidth, final int mask) {
    long result = 0;
    for (int i = 0; i < len; i++) {
        result |= (str.charAt(i) & mask) << (bitWidth * i);
    }
    return result;
}
```

Listing 6. stringToLong

- Ascii to Long:

```java
private static long asciiToLong(final String str) {
    return stringToLong(str, MAX_LENGTH_ASCII, BIT_WIDTH_ASCII, MASK_ASCII);
}
```

Listing 7. asciiToLong

- Constants:

```java
private static final int BITS_LONG = 64;
private static final int BIT_WIDTH_UNICODE = 16;
private static final int MAX_LENGTH_UNICODE = BITS_LONG / BIT_WIDTH_UNICODE;
private static final int MASK_SHORT = 0xFFFF;
private static final int MASK_UNICODE = MASK_SHORT;
```
5 TEST CASE AND ANALYSIS

Let’s try to do an analysis of Huskysort’s performance. The first thing to note is that, like merge sort, Huskysort requires $O(N)$ extra space. This is because, in addition to the array of object references, we must have an equal-length array of longs (the huskyCode values).

Secondly, Huskysort is not stable unless using the merge sort variation. How many comparisons and swaps does Huskysort require, on average? It has been shown for dual-pivot quicksort [Nebel] that these values are (approximately):

- Comparisons: $1.9N \ln N$;
- Swaps: $0.6N \ln N$.

Our experiments [refer to Quicksort Analysis table] show that each comparison requires 2 array accesses (although, because one comparator is always a pivot value, this number is effectively closer to 1. Each swap requires 4 array accesses, although again, we will be swapping elements that have already been accessed. The effective number of (normalized) array accesses is, in total, approximately 4 rather than the expected $3.8 + 2.4$. That is to say, the coefficient of $N \ln N$ is approximately 4. The analysis represented by the table is for dual-pivot quicksort. However, in Huskysort, each swap also must swap the object references. These will not, in general, have been pre-fetched and so the effective total normalized array accesses ($A$) is approximately:

$$A = 4 + 0.6 \times 4 = 6.4$$

Additionally, Huskysort requires $N$ huskyCode constructions. Each of these requires $k + 1$ array accesses where $k$ is proportional to the number of fields contributing to the huskyCode. For a string, that will be the number of characters in the string (except that this is limited to a relatively small number according to the encoding). For ASCII strings, it is nine. For Unicode strings, it is four.

However, keep in mind that this is linear and, while it can’t be completely ignored, it doesn’t contribute to the overall growth of the complexity. Once the Huskysort is complete, we will assume that there are $pN$ remaining inversions, where $p$ is the probability of an idiosyncratic (self-inflicted) inversion. Again, while we should not ignore it completely, it is linear. The number of array access for the comparisons.swaps to fix these will be $(2 + j + 4)pN$, where $j$ is the number of fields to be compared to get a result ($j$ is usually smaller than $k$).

So, the total number of array accesses ($A$) for Husky-sorting an array of length $N$ is:

$$A = (6.4N + (2 + j + 4)p + k + 1)N$$

Timsort would be hard to analyze in the general case and so we instead analyze merge sort, which should be approximately the same number of array accesses in the average case.

For merge sort, the analysis is much simpler: the number of comparisons for a randomly ordered array of length $N$ is $N \log_2 N$. Merge sort does not use swaps, per se, but instead uses copies. Assuming that we optimize the extra copying, there will be $N$ copies to get the algorithm started and $N$ per level. Thus, the number of copies will be $N \log_2 N$. Each of these copies requires two array accesses while each comparison requires $2 + j$ (described above) array accesses.

The total number of array accesses ($A$) for merge sort is, therefore:

$$A = (4 + j)N \log_2 N + 2N$$

Given that $\log_2 N = 1.44N$, and assuming that:

- $j = 4$ corresponding to the comparison of two strings that start with the same character but differ in the second character; and
- $k = 7$ and $p = 0.1$; then

the totals ($A_m$ for merge sort and $A_h$ for Huskysort) come to:

$$A_m = 11.5N \ln N + 2N$$

$$A_h = 6.4N \ln N + 9N$$

The consequence of this is that, as expected, the linear phase of Huskysort is relatively high for small $N$ but that is soon swapped by the saving of comparisons in the linearithmic phase. The break-even point is actually a very small value of $N$: four. As $N$ increases, we expect the advantage of Huskysort to increase further. This is generally borne out by the benchmarks. Please refer to tables 1 and 2.

Since the final pass of Huskysort will be cleaning up a relatively small number of inversions, does it matter whether we use insertion sort or Timsort? It turns out that for arrays which are smaller than about 50,000 Strings, it makes barely any difference at all. Indeed, insertion sort is sometimes faster. However, as the size of the array increases beyond this value, Timsort begins to win out over insertion sort. Please refer to table 3.

6 RESULTS AND OBSERVATIONS

6.1 Benchmarks

- Comparison of sort times for Huskysort, dual-pivot quicksort, system sort. Please refer to table 2.
- Time vs Size ($N$) for Huskysort, Please refer to figure 5.
Table 1. Theoretical comparisons

| N       | N ln N  | merge sort | Huskysort |
|---------|---------|------------|-----------|
| 4       | 6       | 72         | 72        |
| 1,000   | 6,908   | 81,439     | 55,075    |
| 1,000,000 | 13,815,511 | 160,878,371 | 101,149,455 |
| 1,000,000,000 | 20,723,265,837 | 240,317,557,125 | 147,224,183,132 |

Table 2. Relative sort times for Huskysort, dual-pivot quicksort, system sort

| N     | Huskysort | DP quicksort | system sort |
|-------|-----------|--------------|-------------|
| 1000  | 0.60      | 1.00         | 0.84        |
| 2000  | 0.78      | 1.00         | 1.19        |
| 4000  | 0.81      | 1.00         | 1.23        |
| 8000  | 0.77      | 1.00         | 1.27        |
| 16000 | 0.74      | 1.00         | 1.24        |
| 32000 | 0.74      | 1.00         | 1.24        |
| 64000 | 0.85      | 1.00         | 1.29        |
| 125000| 0.76      | 1.00         | 1.51        |
| 250000| 0.92      | 1.00         | 1.32        |
| 500000| 0.97      | 1.00         | 1.42        |
| 1000000| 0.95     | 1.00         | 1.45        |
| 2000000| 0.93     | 1.00         | 1.54        |
| 4000000| 0.91     | 1.00         | 1.64        |

Table 3. Timsort or insertion sort

| N      | Husky with Tim | Husky with Insertion |
|--------|----------------|----------------------|
| 1000   | 0.14           | 0.14                 |
| 2000   | 0.31           | 0.30                 |
| 4000   | 0.63           | 0.64                 |
| 8000   | 1.46           | 1.56                 |
| 16000  | 3.24           | 3.19                 |
| 32000  | 6.56           | 6.48                 |
| 64000  | 16.85          | 19.41                |
| 125000 | 47.74          | 54.90                |
| 250000 | 168.21         | 167.90               |
| 500000 | 259.37         | 373.43               |
| 1000000| 602.08         | 1202.61              |
| 2000000| 1234.90        | 3129.23              |
| 4000000| 2423.62        | 11018.19             |

- Comparison of Huskysort with system sort for numeric types (Integer, Double, Long, BigInteger, BigDecimal): Please refer to figure 6 for results.
- Comparison of Huskysort with system sort for (String types–English Words, Chinese Words): Please refer to figures 7, 8 for results.
- Comparison of Huskysort with system sort for Tuples: Please refer to figure 9 for results.

6.2 Summary

Huskysort’s target application is the sorting of randomly ordered arrays of objects on the JVM (Java Virtual Machine). Let’s eliminate the non-use-cases first:

- Partially-ordered arrays: that’s to say, arrays with only a linear number of inversions as would be the case when adding a relatively small number of records to a large existing index. For such use cases, Timsort (the Java system sort for objects) would be more suitable.
Arrays of primitives: in Java, a clear distinction exists between primitive types (char, byte, short, int, float, long, double) and objects. Dual-pivot quicksort (the Java system sort for primitives) is already the fastest algorithm for sorting such types. Thus, Huskysort is best for any object array where there’s no expectation of partial ordering. Here is a summary of typical improvements in performance over Timsort: Please refer to table 4.

7 CONCLUSION
We have demonstrated that we can shift work from the linearithmic phase of sorting to the linear phase (both the prelude and the postlude) and effect an overall reduction in array accesses and thus processing time. When sorting objects rather than primitives, Huskysort is always faster than dual-pivot quicksort. It is faster than Timsort for arrays which are not partially ordered. It is especially fast for Unicode character strings.

ACKNOWLEDGMENTS
To Darshan Dedhia, Akshay Bhusare, Kartik Kumar for their help with this project.

REFERENCES
Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. 2018. On the Worst-Case Complexity of Timsort. CoRR abs/1805.08612 (2018). arXiv:1805.08612 http://arxiv.org/abs/1805.08612
William F. Gilreath. 2004. Hash sort: A linear time complexity multiple-dimensional sort algorithm. CoRR cs.DS/0408040 (2004). http://arxiv.org/abs/cs.DS/0408040
Dirk Goldhahn, Thomas Eckart, and Uwe Quasthoff. 2012. Building Large Monolingual Dictionaries at the Leipzig Corpora Collection: From 100 to 200 Languages. In Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC’12) (23-25), Nicoletta Calzolari, Khalid Choukri, Thierry Declerck, Mehmet Uğur Doğan, Bente Maegaard, Joseph Mariani, Asuncion Moreno, Jan Odijk, and Stelios Piperidis (Eds.). European Language Resources Association (ELRA), Istanbul, Turkey.
C. A. R. Hoare. 1961. Algorithm 64: Quicksort. Commun. ACM 4, 7 (July 1961), 321. https://doi.org/10.1145/366622.366644
Vincent Jugé. [n.d.]. Adaptive Shivers Sort: An Alternative Sorting Algorithm. 1639–1654. https://doi.org/10.1137/1.9781611975994.101
Peter McIlroy. 1993. Optimistic Sorting and Information Theoretic Complexity. In Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (Austin, Texas, USA) (SODA ’93). Society for Industrial and Applied Mathematics, USA, 467–474.
DAVID R. MUSSER. 1997. Introspective Sorting and Selection Algorithms. Software: Practice and Experience 27, 8 (1997), 983–993. https://doi.org/10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#
Table 4. Improvements Summary

| Data Type | Percentage of Improvement |
|-----------|--------------------------|
| Strings of English words (4,000–500,000 elements) | 33-41 |
| Strings of Chinese words (4,000–500,000 elements) | 31-52 |
| Tuples (20,000–500,000 elements) | 14-19 |
| Integers (20,000–500,000 elements) | 31-41 |
| Doubles (20,000–500,000 elements) | 25-38 |
| Longs (20,000–500,000 elements) | 31-42 |
| BigIntegers (20,000–500,000 elements) | 35-40 |
| BigDecimals (20,000–500,000 elements) | 36-38 |

APPENDIX

A.1 Discussion of coding accuracy

It is not essential that coding is perfect for Huskysort to work well. We have experimented with deliberately throwing off the encoding quite drastically. If the probability of one of these deliberate mis-codings is less than approximately 25 percent, Huskysort can still save time. The inference that we can take from this is that it is not critical that encodings are all very good. We can be confident that we are still saving time up to about one in four errors.

These deliberate errors were tested for Bytes and Integers but not the other generic types.

A.2 Handling of partially-sorted arrays

Not surprisingly, Huskysort is best suited for sorting random arrays. The benefits of partially-sorted arrays are enjoyed most by the merge sort family of algorithms (including Timsort). However, we have studied the efficacy of a modified merge sort (it is required to manage both the objects and the longs) as the first pass of the Huskysort algorithm. For random English String inputs, the merge-sort variation improves on system sort by approximately 20 percent whereas the normal intro-sort-based Huskysort improves on the system sort by more like 40 Percent.

However, it is not recommended to use Huskysort when arrays are partially ordered. In such cases, the merge-sort variation of Huskysort will take anywhere between one and one half times and four times as long as Timsort.