# Darksort: A New Linear Sorting Algorithm

**Blake MacKenzie Burns**

***Abstract*: This is a new paper on the algorithm "Darksort". It is a linear sorting algorithm that operates in O(n) timeand space complexity. It uses advanced data structures to be highly applicable in many Computer Science uses. Comparisons to other linear sorting algorithms are included.**

***Index Terms*: Computer Science, Data Structures, Algorithms, Sorting**

## I. INTRODUCTION

A new sorting algorithm called "Darksort" or "darksort" has been created by me. In this paper you will find the algorithm itself, some direct proofs of run time, and comparison to other sorting algorithms. In general it runs in O(n) time and space complexity. It uses advanced data structures to improve speed in sorting. It is an integer sorting algorithm. Darksort is a new sorting algorithm that is explained in this paper.

## II. BODY TEXT

### A. Model

It takes in an unsorted or sorted array Array and an integer Arraysize that describes the size of the array, as well as max in <maxvariant>.

### B. Proof

*1) Direct-access table darksort (original darksort)*

First I will provide implementation pseudocode (Fig. 1) and then I will provide the direct proof and a brief explanation of the program correctness.

```
DarksortDAT(Array, Arraysize)
 int max = 0
 for i in Arraysize
    if (max < Array[i])
        max = Array[i]
 int array newarray = [0] * (max+1)
 for i in Arraysize
    newarray[Array[i]] += 1
 int count = 0
 for i in (max + 1)
    int var = newarray[i]
    while (var > 0)
        Array[count] = i
        count += 1
        var -= 1
 return Array

DarksortDAT< maxvariant >(Array, Arraysize, max)
 int array newarray = [0] * (max+1)
 for i in Arraysize
    newarray[Array[i]] += 1
 int count = 0
 for i in (max + 1)
    int var = newarray[i]
    while (var > 0)
        Array[count] = i
        count += 1
        var -= 1
 return Array
```

**Figure 1**

As you can tell the direct proof is quite easy, it is simply three for loops where the first and second run in time Arraysize (which is O(n)) (guaranteed to terminate) and then the last for loop which runs in the time of max (which is O(max)) which is the largest value of the array. This is the slowest part. So, in total, it is clear that it runs in O(2n + max) and since max is a constant, it runs in O(2n) therefore O(n) time. Please note that when max is used as a variable, it can be reduced to O(n + max) in runtime as it requires one less for loop as seen in Fig.1 <maxvariant>.

The first for loop initializes the max value, which is the largest size for the array in the DAT. The next for loop creates the direct access table which stores the value of the number of each unique value in the data set (numerical), which is very large in memory size. The last portion is the part which uses this DAT to create a sorted list. All are guaranteed to terminate as they are straightforward for loops, and thus the program terminates.

Manuscript received on April 25, 2018.
**Blake MacKenzie Burns**, Department of Computer Science, University of Toronto, Toronto, Canada, E-mail: blake.burns@gmail.com or blake.burns@mail.utoronto.ca

*2) AVL darksort*

```
DarksortAVL(Array, Arraysize)
 int max = 0
 for i in Arraysize
    if (max < Array[i])
        max = Array[i]
 int array newarray = [0] * (max+1)
 for i in Arraysize
    newarray[Array[i]] += 1
 Create.AVL
 for i in (max + 1)
    int var = newarray[i]
    while (var > 0)
        AVL.insert(i)
        var -= 1
 return AVL
```

**Figure 2**

This provides an example in how darksort can be used to sort the data into a stronger or perhaps more useful way for which its uses are numerous (such as AVL tree storage of data for different uses with log(n) operations and darksort for sorting).

*3) Heap darksort*

```
DarksortHeap(Array, Arraysize)
 int max = 0
 for i in Arraysize
    if (max < Array[i])
        max = Array[i]
 int array newarray = [0] * (max+1)
 for i in Arraysize
    newarray[Array[i]] += 1
 Create.Heap
 for i in (max + 1)
    while (newarray[i] > 0)
        Heap.insert(i)
        newarray[i] -= 1
 return Heap
```

**Figure 3**

This provides an example in how darksort can be used to sort the data into a simple heap which will result in much faster retrievals in data in certain applications such as popping off advertisements to serve them to clients in a near constant fashion (log(n) operations at worst for insert, constant pop).

*4) General Data Structures*

This section will display that darksort can be extended to any data structure, with both the original and <maxvariant> versions (Fig. 4). (General data structure = GDS)

```
DarksortGDS(Array, Arraysize)
 int max = 0
 for i in Arraysize
    if (max < Array[i])
        max = Array[i]
 int array newarray = [0] * (max+1)
 for i in Arraysize
    newarray[Array[i]] += 1
 Create.GDS
 for i in (max + 1)
    while (newarray[i] > 0)
        GDS.insert(i)
        newarray[i] -= 1
 return GDS

DarksortGDS< maxvariant >(Array, Arraysize, max)
 int array newarray = [0] * (max+1)
 for i in Arraysize
    newarray[Array[i]] += 1
 Create.GDS
 for i in (max + 1)
    int var = newarray[i]
    while (var > 0)
        GDS.insert(i)
        var -= 1
 return GDS
```

**Figure 4**

As you can see any general data structure can be used to implement darksort. It works for any data structure that can create and insert and can be modified for others.

## III. THOUGHTS

In general, DarksortDAT is what is most important, but data structure versions can improve space complexity.The space complexity is exactly two times the size of the input array as well as gaps included in the max array finalarray. To be precise it is explainable as such. In an array given of:

A = [10,10,8,6,5,2,3,1]

It uses darksort to sort the array in O(2n + max) assuming max isn't givenwhich results inO(2n + 10) time (max included is n + max), with a space complexity including gaps of O(2n + max) which equates to O(2n + 10). Both of these complexities are in the element of O(n). The data structure variants such as heap or AVL (and others) improve the space complexities of the algorithm significantly in the long term as you can simply use bubble in heap or rebalance AVL trees in O(logn) time to improve space complexity as examples.

The space complexity is large becausedarksort holds gaps for 4, 7, 9(in this example) in newarray which are missing in the data set in memory. This is because newarray holds up to max numbers and although 4, 7, 9 stay as 0 they are still held in space. Array is obviously held in memory as it is dealt with, which gives (n) space, newarray is held which gives (max), and the finalarray is held which yields one more (n) giving (n + max + n == 2n + max). If Array is taken for granted then it is only O(n + max) but without Array in space it gives O(2n + max) space complexity.

The space looks like this for new array:

Space = [1,2,3,4,5,6,7,8,9,10]

Published By:
Blue Eyes Intelligence Engineering
& Sciences Publication Pvt. Ltd.

The returned array will be ascending order using the original darksort algorithm:

A = [1,2,3,5,6,8,10,10]

Most importantly, this is a stable sorting algorithm.It can be changed to descending order by iterating backwards over the newarray in the final for loop.

## IV. COMPARISON TO OTHER LINEAR SORTS

Direct algorithm comparison is left to the reader.

### A. Counting Sort

Counting sort is a sorting algorithm that is similar to darksort in that it uses no comparisons but has a different algorithm entirely. Its input is very similar to darksort. The Big-O time complexity is O(n + k) where n is the number of items and k is the max value [1]. A quick discussion on counting sort: it uses a similar way but note that when max is used as a variable it uses one less for loop and therefore beats it by n (Counting sort: including constants 2n + k, Darksort n + max where max = k). The algorithm darksortis very different compared to counting sort. The algorithm counting sort is available to view at reference [2]. It functions in a completely different way. To be more detailed, Darksort beats it in computation speeddoing less computations by (13n – 3max) Notice if max is equal to 13/3n the speeds are equal and slower if it is larger than 13/3n. As long as there are not too many gaps in the data such as {1, 300, 1600, 2100, 5300, …., 10000}, the max should be less than 13/3n and would be faster in that scenario. {5, 3, 7, 2, 9, 1, 4, 8, 6, 10} is faster with darksort for example by 10n.

### B. Radix sort

Radix sort is a sorting algorithm that sorts using one integer at a time. It can also be used to sort strings [3]. A quick discussion on radix sort: With word sizes such as 64 bit numbers (8 bytes) it depends on the size of the max. Very clearly different from this algorithm as well, and is unique in this respect. Radix sort is (wn) and when word size is 64 bits let us estimate it as (64n). That means as long as k is less than 62n it is at least the same or better than radix sort. Darksort improves on a factor of n with max as a variable so it can have max less than 63n, and still be faster than radix sort, even without distinct keys.

In a practical application such as sort with unique keys both counting sort and Darksort seem superior to radix sort. The algorithm is entirely different from radix sort and radix sort is available from reference [3].

### C. Bucket sort

In bucket sort, the algorithm sorts the items into buckets and then sorts using insertion sort [3][4].Not much needs to be discussed besides providing an additional source on the algorithm for it to be compared by the reader. There is a variation on bucket sort called Groupsort that is not like it at all [5]. It clearly beats bucket sort as it has a worst-case complexity of $O(n^2)$.

### D. Pigeonhole sort

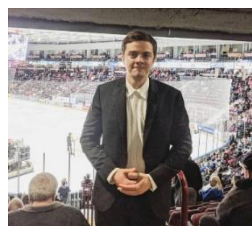
It is a very similar to bucket sort in that it places elements in buckets as well [6]. It is clear that this algorithm is similar to mine, but mine improves on pigeonsort by a couple lines of code. They are very similar but darksort does not require the minimum value in a for loop and has slightly less lines of code overall. Therefore it does beat the pigeonhole sort time complexity in the worst case. In general it wins with less comparisons for min (an if statement and an assignment for min throughout a for loop) and also some addition statements throughout the algorithm that use min. Overall darksort can be substantially faster when implemented properly (by max in computation (not Big-O) in general).

## V. CONCLUSION

Darksort is a unique linear sorting algorithm with superior performance to all other linear sorting algorithms under certain circumstances. One should note that if you have the min, or use a for loop to get it, you can iterate in the last loop from only min to max or max to min depending on how you want it sorted (ascending and descending respectively). This changes the amount of computation in the algorithm and makes it much faster.Also if you can minimize gaps in the data somehow darksort is preferable to all sorting algorithms. Counting sort can have an advantage if max is larger than 13/3n, but in normal circumstances with proper data storage darksort is faster. Radix sort can have an advantage under certain circumstances but given highly distinct keys and large word size darksort is superior with a max within reason. This is a completely original theory that was done without looking into other linear sorting algorithms. It seems that there are no other linear sorting algorithms exactly like it, although counting sort is somewhat similar in concept and pidgeonhole sort, darksort is a unique new theory. It is an integer sorting algorithm only. I am proud that it beats every sorting algorithm in speed, although the space complexity may be larger than others.

## REFERENCES

1. Kazim Ali. "A Comparative Study of Well Known Sorting Algorithms". V. 8, No.1 Jan-Feb 2017, International Journal of Advanced Research in Computer Science pages 1-5.
2. Stijn de Gouw, Frank de Boer, Juriaan Rot. "Proof Pearl: The KeY to Correct and Stable Sorting". V. 53.2, 2014, Journal of Automated Reasoning pages 129-139.
3. PanuHorsmalati. "Comparison of Bucket and Radix sort". https://arxiv.org/abs/1206.3511. V. 1206.3511v1 [cs.DS], 2012, pages 1-10.
4. Paul M. E. Shutler, Seok Woon Kim, Wei Yin Selina Lim. "Analysis of Linear Time Sorting Algorithms". V. 51, Issue 4, Oxford University Press on behalf of The British Computer Society, pages 451–469.
5. Apostolos Burnetas, Daniel Solow, Rishi Agarwal. "An analysis and implementation of an efficient in-place bucket sort". V. 34, 1997. Acta Informatica, pages 687–700.
6. Paul E. Black, "Pigeonhole sort", in Dictionary of Algorithms and Data Structures [online], VredaPieterse and https://www.nist.gov/dads/HTML/pigeonholeSort.html. 2006. Paul E. Blacks eds.



**Blake MacKenzie Burns** is a student at University of Toronto with a Specialist in Computer science and a Minor in Classical Civilization, this paper is his first journal publication, and he is working on some other papers such as papers in robotics, and he is also currently working on a small paper on triangulation and tessellation that he hopes to publish soon. Among his interests in Computer Science are algorithm efficiency, sorting, artificial intelligence and machine learning, robotics, NP hard problems, language creation, and more.

*Published By:*
*Blue Eyes Intelligence Engineering*
*& Sciences Publication Pvt. Ltd.*