Just Sort

Sathish Kumar Vijayakumar
Chennai, India
satthhishkumar@gmail.com

***Abstract*—Sorting is one of the most researched topics of Computer Science and it is one of the essential operations across computing devices. Given the ubiquitous presence of computers, sorting algorithms utilize significant percentage of computation times across the globe. In this paper we present a sorting algorithm with average case time complexity of O(n) without any assumptions on the nature of the input data.**

Keywords—sorting; time complexity; space complexity;

I. INTRODUCTION

Sorting is not only the most researched and analyzed topic among researchers but also the most prevalent and ever present algorithm for every introductory computer science classes . As a result of its ubiquitous nature the audience it has attracted in solving it is humongous. The best worst case complexity achieved by any comparison based sorting algorithm so far is O(*nlogn*), like quicksort, merge sort, heap sort etc, and non-comparison based sorting algorithm is O(*dn*), for radix sort where *d* is the key size. In this paper I have developed a novel , stable, non-comparison sorting algorithm of average case time complexity O(*n*) using Combinatorics and the divide and conquer paradigm. For the ease of understanding and analysis, the whole algorithm, called Just Sort , is presented in its naive form, having separate sub routines for each of the base cases. The core subroutine of this algorithm is based on Combinatorics, a branch of Mathematics, and uses the Hash Table data structure .Hence it is termed as the Combinatorial Hash Sort sub routine.

II. MATHEMATICAL BACKGROUND

Given *n* unique elements, the problem of sorting can be restated mathematically as finding the only permutation ,where the *n* elements are arranged in ascending order, out of *n!* permutations which contains all *n* elements in various orders. Thus for a sorting problem we have *n!-1* incorrect outputs and only one correct output in which the elements are in sorted order. Unlike permutations, there is only one combination which consists of all these *n* elements. For a set of *n* unique numbers whose range is *k*, the sorting problem can be solved partially by identifying a combination out of 2^{k+1} combinations, which contains all *n* unique numbers. We are deriving and exploiting a constraint relationship between the set of n unique elements and 2^{k+1} combinations in our approach.

As mentioned before, the heart of this algorithm is based on the knowledge derived from Combinatorics. According to Combinatorics the total number of possible combinations of *n* distinct items is given by 2^n. Let there be, for instance, 5 numbers from 0 to 4. The total number of possible combinations of these 5 numbers is 32 and it is listed in Table I where the presence or absence of a number in a combination is represented in binary digit and the numbers in column headers are in sorted order. 1 denotes its presence and 0 denotes its absence in that particular combination. The decimal equivalents of these binary numbers represented by these combinations are given under the Sum column of Table I. It is easy to note that these binary numbers represented in decimal number system are continuous in nature and it is unique for each sequence. In other words, the presence or absence of a number in a sequence is reflected in its decimal sum *S*. Let b_0, b_1, b_2, b_3 and b_4 be the bits of these binary numbers from least to most significant bit, its decimal sum is given by

$$S = b_0\times2^0 + b_1\times2^1 + b_2\times2^2 + b_3\times2^3 + b_4\times2^4$$

In general the Sum of j^{th} row is given by

$$S_j = \sum_{i=0}^{n-1} b_i^j\, 2^i \qquad (1)$$

Where n is the number of elements in our collection and b_i^j is the i^{th} bit of the binary number corresponding to the j^{th} row of Table I.

We are storing the numbers in an array which are present in each combination from Least Significant Bit to Most Significant Bit, i.e. from b_0 to b_4, only if they are present i.e. if $b_i=1$ or its face value in binary sequence is *1,* while maintaining its order from LSB to MSB. For instance the first combination has no elements and the tenth combination has 1 and 4, which is stored in an array in the exact order. We are storing this knowledge in the form of a two dimensional array and we are denoting it as Sort Table(*sorttable*) ,where the outer array is indexed by the respective decimal sum S_j of the j^{th} combination calculated by (1). For instance, the 10th row and 32nd row of the sort table are denoted as

sorttable [9] = {1,4}

sorttable [31] = {1,2,3,4,5}

Where the term within square brackets denote the Sum column of Table I and its corresponding elements present are represented as an array within curly braces.

Equation (1) can be restated as

$$S_j = \sum_{i=0}^{n-1} b_i^j \cdot w_i \qquad (2)$$

Where $w_i = 2^i$ is the weight or place value of that element in the binary sequence of the j^{th} combination or the contribution of i^{th} bit towards the sum S_j. For the scenario considered above, the value of n would be 5, as there are 5 elements the weight values w_i are stored in an array C_W of size *5.*

$C_W = \{ 1 , 2 , 4 , 8 , 16 \}$

TABLE I Combination sequences of 5 elements and its decimal sum.

Four(b_4)	Three(b_3)	Two(b_2)	One(b_1)	Zero(b_0)	Sum
0	0	0	0	0	0
0	0	0	0	1	1
0	0	0	1	0	2
0	0	0	1	1	3
0	0	1	0	0	4
0	0	1	0	1	5
0	0	1	1	0	6
0	0	1	1	1	7
0	1	0	0	0	8
0	1	0	0	1	9
0	1	0	1	0	10
0	1	0	1	1	11
0	1	1	0	0	12
0	1	1	0	1	13
0	1	1	1	0	14
0	1	1	1	1	15
1	0	0	0	0	16
1	0	0	0	1	17

1	0	0	1	0	18
1	0	0	1	1	19
1	0	1	0	0	20
1	0	1	0	1	21
1	0	1	1	0	22
1	0	1	1	1	23
1	1	0	0	0	24
1	1	0	0	1	25
1	1	0	1	0	26
1	1	0	1	1	27
1	1	1	0	0	28
1	1	1	0	1	29
1	1	1	1	0	30
1	1	1	1	1	31

TABLE II SORT TABLE

Index	Array Elements
0	{}
1	{ 0 }
2	{ 1 }
3	{ 0 , 1 }
4	{ 2 }
5	{ 0 , 2 }
6	{ 1 , 2 }
7	{ 0 , 1 , 2 }
8	{ 3 }
9	{ 0 , 3 }
10	{ 1 , 3 }
11	{ 0 , 1 , 3 }
12	{ 2 , 3 }
13	{ 0 , 2 , 3 }
14	{ 1 , 2 , 3 }
15	{ 0 , 1 , 2 , 3 }
16	{ 4 }
17	{ 0 , 4 }
18	{ 1 , 4 }
19	{ 0 , 1 , 4 }
20	{ 2 , 4 }
21	{ 0 , 2 , 4 }
22	{ 1 , 2 , 4 }
23	{ 0 , 1 , 2 , 4 }
24	{ 3 , 4 }
25	{ 0 , 3 , 4 }

26	{ 1 , 3 , 4 }
27	{ 0 , 1 , 3 , 4 }
28	{ 2 , 3 , 4 }
29	{ 0 , 2 , 3 , 4 }
30	{ 1 , 2 , 3 , 4 }
31	{ 0 , 1 , 2 , 3 , 4 }

This knowledge can be used for sorting in the following way. For instance if we are to sort an array *{ 4 , 3 , 2 }* we can traverse the array and add the corresponding weights w_i to compute *S* cumulatively which will be equal to 28 at the end of our traversal and we can get the corresponding sequence of numbers from the *sorttable* whose index *S* is equal to 28 i.e. *S[28]* which has the elements *{ 2 , 3, 4 }* which is sorted. Thus we just need to traverse these elements in order to sort it.

If the numbers to be sorted are not within the interval of [0 , 4], for instance 14, 13 and 12, but its range lies in the interval [0 , 4], we could consider the modulo 5 operation of the numbers(as the order of *sortable* α used is 5) and use the result as indices to store the numbers in an array B of size 5 (which equals α), which holds a constraint that the value of an index is directly proportional to the value of the number stored in that index and hence the number stored in lesser value of index is always lesser than the number stored in greater value of index .Consequently, in the above scenario 14 will be stored in index B[4] , 13 in B[3] and 12 in B[2].We calculate *S* from the value of indices by fetching the corresponding values from the array C_W (i.e. C_W[2],C_W[3] and C_W[4]) and then we could retrieve the sorted indices from the *sorttable* whose index is *S(equals to 28 in the above scenario)* , from which we could retrieve the numbers in sorted order, i.e. from the indices {2 , 3 , 4 } of array B, which corresponds to *S=28* in the *sorttable*.

Thus we could partition the numbers to be sorted such that the range of each partition is less than the range of the *sortable* and then we could sort the elements using *sortable* by using the method elucidated above.

For a sortable of order $\alpha = 10$, the table has 1024 entries and the values of C_W array is as follows

$$C_W = \{ 1 , 2 , 4 , 8 , 16 , 32 , 64 , 128 , 256 , 512 \}$$

We are defining the Order of a Sort Table α as the number of unique elements present in it and TABLE I is of the order 5 as it has 5 unique elements. We are using this combinatorial knowledge in the form of two dimensional array denoted as *sorttable* , as a pre-computed constant in our sorting algorithm. The computation time of this Sort Table of order 5 is infinitesimal, obviously, and we need not even bother computing it and it can be integrated into the libraries of programming languages. For the ease of analysis, going forward we use a Sort Table of order (α) 10 in our algorithm, which has elements from 0 to 9, inclusive, and it takes 1024 computations which is executed under 6 milliseconds in Java Standard Edition 8.

III. Just Sort Overview

We use the knowledge derived above in our sorting algorithm using divide and conquer paradigm by partitioning the numbers into buckets whose size is equal to the Order of Sort Table α. Let *A[n]* denote an array of *n* elements and *R* be the range of the numbers in it. The primary data structure used by this sorting algorithm is Hash table. The size of the Hash table β is determined by the range *R* of the numbers to be sorted and the Order of the Sort Table α .Each entry of the Hash table represents a bucket .Therefore the size of the Hash table β is equal to the number of buckets. We divide the Range *R* into *k* buckets of size α. Thus each entry of the Hash Table *H* corresponds to a bucket. We define a suitable hash function involving α and place each element of the array in its corresponding bucket.

$$\textit{Size of Hash Table} = \textit{Range/Order of the Sort Table}$$

$$\beta = {}^{R}/_{\alpha}$$

The hash function $f(A[i])$ is given by

$$f(A[i]) = A[i]/\alpha \qquad (4)$$

where the operator '/ ' in (4) denotes integer division and *A[i]* is the i^{th} element of array *A* and the above division is integer division and it just gives the quotient excluding the remainder.

It is obvious that the space complexity of this algorithm is dependent on the range of the numbers to be sorted. In order to minimize the space complexity we first separate the numbers into groups, where every number in a group has same number of digits. If necessary we further divide the groups as per our space constraints. We then sort each group separately and then combine them in the end.

Algorithm 1. Range Reduction

```
Input  : A [ 1 ... n ]
Output : Array of arrays(or list of lists) B

1:  i = 0
2:  k = 0
3:  While i ≠ n
4:      k = A [i] / 10
5:      Add A [i] to B based on k value
6:      i++
7:  end while
8:  return B
```

While we are traversing each element of the array , we identify the corresponding bucket using the hash function given by (4) and we calculate the sum $S_{f(A[i])}$ for that bucket simultaneously by adding the respective weight $w_{f(i)}$ to that bucket's sum $S_{f(A[i])}$, where $f(i)$ is given by the number modulo α in (5) .

$$f(i) = A[i] \bmod \alpha \qquad (5)$$

We use this sum $S_{f(A[i])}$ to retrieve the corresponding array from the *sorttable* whose values denotes the sorted indices of the elements present in that particular bucket. We define the term ***Active Buckets*** as those entries or indices of the Hashtable which contains at least one element in it. While traversing the numbers, we insert the active buckets, i.e. the bucket which contains at least one element in it, into a list denoted as *keys*. We then sort *keys* using the algorithm *keyssort*. It is evident that the total number of active buckets is always lesser than or equal to the total number of element in the array *A[n]* and as a result the time complexity of sorting the *keys*, which is a list of active buckets, is always lesser than or equal to the time complexity of sorting *A[n]*. Thus we are placing the numbers to be sorted into its corresponding buckets while computing the sum $S_{f(A[i])}$ for every active bucket simultaneously during the traversal and we are sorting *keys*, which is the list of active buckets, using *keyssort* algorithm recursively, which internally calls the algorithm *Combinatorial hashsort*. We traverse the list *keys* whose elements are now sorted, i.e. it contains active buckets in sorted order, and with the help of the sum $S_{f(A[i])}$ calculated for each active bucket, we retrieve the sorted indices for each active bucket present in *keys* using the algorithm *Combinatorial hashsort_d,* and from those indices we are retrieving the numbers present in that active bucket in sorted order and storing it in a list. We then merge these lists which contains the elements in sorted order.

This algorithm comprises of three subroutines which are called internally by the main algorithm JUST SORT.

A. *Keysort*

This subroutine sorts the active buckets recursively and its output consist of a list which consists of the active buckets in sorted order. The input of this sub routine consists of unique elements as it is evident that the buckets are unique and it calls the Combinatorial Hash Sort sub routine internally.

B. *Combinatorial Hash Sort*

The input of this subroutine consists of unique numbers whose range is always lesser than or equal to the order of the *sorttable α* and it is invoked by Keysort subroutine. Thus the maximum number of elements passed as its input never exceeds the order of the *sorttable α.* This is the key subroutine of this algorithm and its input elements are traversed and its sum S is computed and then the sorted indices are retrieved from the *sortable.* This algorithm exploits the key idea discussed in section II.

C. *Combinatorial Hash Sort_d*

This subroutine is similar to the above sub routine but it handles duplicate elements in its input. This is invoked by the main algorithm to sort the elements in each active bucket.

Each entry of the hash table is associated with a list whose head and tail pointers are maintained so as to merge two list in *O(1)* time. Other list operations like Insert are also completed in *O(1)* time. The order of the *sortable* used in the following sections is 10.

IV. Correctness

The line by line explanation of this algorithm is as follows

A. *Just Sort*

In lines 1 to 3, the size of the hash table H is determined. We are maintaining an array, C_0, of size k whose purpose is to track the presence of active buckets followed by its insertion into the list of active buckets L and by default all its entries are initialized to 0. Once an element is insert into L at line 7, the entry of C_0 corresponding to that particular active bucket is assigned a value of 1 in line 8. The input array *A[1...n]* is traversed in lines 4 to 11. The variable *key* in line *5* denotes

Algorithm 2. Just Sort

```
Input : A [ 1 ... n ] , Hashtable  H [ 0 ... k ] , C_0 [ 0 ... k ] ,
        List L , max , min
Output : List result
1:  k = max-min
2:  start = min/10
3:  k = k/10 +1
4:  for i = 1 to A.length
5:    key = (A[i]/10) – start //we are shifting index range of H
6:    if  C_0[key] = 0
7:        LIST-INSERT (L , key )
8:        C_0[key] = 1
9:    end if
10:   INSERT (H, key ,A[i] )
11: end for
```

```
12: if  n>= k/10
13: traverse H and call combinatorialhashsort_d( H[temp.value] ) for all keys
14: else
15:    keys = keysort (L, k ) // keys are  sorted
16:   temp =  keys.head
17:   while  temp ≠ null
18:      templist = combinatorialhashsort_d( H[temp.value] )
19:      result = LIST-MERGE (result, templist)
20:      temp = temp.next
21:   end while
22: End if else
```

the hash function given by equation (4) and it denotes a bucket. The range is shifted by subtracting the variable *start* from *key* so as to shift the range towards zero. In line 6 the value of $C_0[key]$ is checked and if it is equal to 0, which indicates the absence of a bucket for that *key* value in list *L*, that particular *key* value is inserted into list *L* which contains all active buckets. In line 8 the value of $C_0[key]$ is set to 1 and it indicates that the bucket denoted by its index (*key* value) is added to List *L*. The element *A[i]* is inserted into the hash table *H* in line 8, at the entry denoted by *key*. In line 15 the *Keysort* algorithm is called internally and its output is assigned to the variable *keys* which is a sorted list of active buckets, i.e. the active indices of the hash table H in sorted order. In lines 17 to 21, for each node of list L, the *Combinatorialhashsort_d* sub routine is called internally and the output list is merged with the *result* list which gives the sorted elements of *A[1…n]*.

B. *Keysort*

This algorithm sorts the list of active buckets L using *Combinatorialhashsort* sub routine by portioning the list *L* into buckets of size α similar to *Just Sort* and it recursively calls itself until the range is lesser than 10, which is the order of the *sortable* α and then the recursive calls stop. Its input consists of list *L* and its range *k*. If the value of *k* is lesser than or

Algorithm 3. Key Sort

Input : *Hashtable H [0 ... k] , C [0 ... k] ,*
List L , L_s , k ,
Output : *List result*

```
1:  if  k  >  9
2:     k = k/10 +1
3:    temp = L.head
4:    while temp ≠ null
5:        key = temp.value  / 10
6:        if C[key] = 0
7:           LIST-INSERT ( $L_s$ , key)
8:          C[key] = 1
9:        end if
10:      INSERT ( H , key , temp.value )
11:      temp = temp.next
12:   end while
13:   $L_s$ = keysort ( $L_s$ , k ) //$L_s$ is now sorted
14: else
15:   $L_s$ = combinatorialhashsort( $L_s$ )// $L_s$ is now sorted
```

```
16:     return $L_s$
17: end if  else
18: temp = $L_s$.head
19: while temp $\neq$ null
20:    templist = combinatorialhashsort( H[ temp.value ] )
21:    result = LIST-MERGE( result , templist )
22:    temp = temp.next
23: end while// result now has elements of L in sorted order
24: return result
```

equal to 9, then lines 2 to 13 are skipped and *Combinatorialhashsort* sub routine is called directly as in line 15, else the value of *k* is updated in line 2 and List *L* is traversed in lines 4 to 12. Similar to *Just Sort* a variable *C [0 ... k]* is maintained to keep track of active buckets and the the active buckets are inserted into the list L_s. The value of *A[i]* is inserted into the hash table *H [0 ... k]*. L_s at lines 13 and 15 has elements in sorted order. In lines 18 to 23 the list L_s is traversed and for each entry the *Combinatorialhashsort* sub routine is called and the resulting list is merged with the list *result*. At line 24 the *result* list is returned which consists of the active buckets in sorted order.

C. *Combinatorial HashSort*

This subroutine traverses its input elements in lines 2 to 7. The range of the input elements is always less than 10, as α is equal to 10. The *key* value is computed using equation (5) in line 4 and it is inserted accordingly in Hashtable *H [0 ... 10]* which is an array of size 10 in this sub routine. The sum value is calculated by the variable *C* cumulatively. The sorted active indices of *Hashtable H [0 ... 10]* are retrieved in line 8 from *sortable* and assigned to the variable *ind* and the elements of *H [0 ... 10]* are populated into the list *result* in sorted order in lines 9 to 11. This *result* list is returned in line 12. Thus the elements are traversed, its decimal sum is computed for the combination of input list *L* and the elements are populated in sorted order using the index array returned by the *sortable[C]*.

Algorithm 4. Combinatorial HashSort

Input : *Hashtable H [0 ... 10] , C_w [10] , List L*
Output : *List result*

```
1:  C = 0
2:  temp = L.head
3:  While temp $\neq$ null
4:       key = temp.value mod 10
5:      C = C + $C_w$[ key ]
6:      H [ key ] = temp.value
7:      temp=temp.next
8:  end while
9:  ind = sorttable [ C ] // ind array contains sorted indices
10: for i = 0 to ind.length
11:      LIST-INSERT ( result , H[ind[i]] )
12: end for // result contains elements of L in sorted order
13: return result
```

D. *Combinatorial HashSort_d*

This subroutine is similar to the above sub routine except that it handles duplicate elements. This is called by the main algorithm and each entry of the hash table *H [0 ... 10]* contains a list with its head and tail pointers. The array C_W is a precomputed constant and it values are given below

$$C_W = \{ 1 , 2 , 4 , 8 , 16 , 32 , 64 , 128 , 256 , 512 \}$$

The elements are placed initially in its respective buckets using the hash function given by equation (4). The list of active buckets, i.e. *L* is populated when an element is inserted into a bucket for the first time. The active buckets are sorted by the *Keysort* sub routine and it returns a list *keys* which contains elements of *L* in sorted order. The list *keys* returned by *Keysort* is traversed and the elements in those active buckets are sorted using *Combinatorialhashsort_d* and the results are merged into list *result* which contains the elements of the array *A[1...n]* in sorted order.

Algorithm 5. Combinatorial HashSort_d

Input : *Hashtable H [0 ... 10] ,*C_1 *[10] ,* C_w *[10] , List L*
Output : *List result*

```
1:  C = 0
2:  temp = L.head
3:  while temp ≠ null
4:    key = temp.value mod 10
5:    if $C_1$[key] = 0
6:      C = C + $C_w$[key]
7:      $C_1$[key] = 1
8:    end if
9:    INSERT ( H , key , temp.value )
10:   temp=temp.next
11: end while
12: ind = sorttable [ C ] // ind array now contains sorted indices of keys
13: for i = 1 to ind.length
14:   if H[ind[i]] contains floats, discretize it and call Just Sort internally and then retransform it to get
      sorted float values.
15:   result = LIST-MERGE ( result , H[ind[i]] )
16: end for // now result contains elements of L in sorted order
17: return result
```

V. ASYMPTOTIC ANALYSIS

Let us analyze the algorithm asymptotically starting from smaller subroutines or the internally used subroutines.

A. *Combinatorial HashSort_d*

Let the input list L of the subroutine contain *n* elements. It is evident that lines 1, 2, 4, 5, 6, 7, 8, 9 and 10 has time complexity of *O(1)*. The while loop in line 3 is executed once for all n elements of list *L* , which contains *n* elements, and therefore its complexity is *O(n)*. The complexity of lines 12 and 14 are also *O(1)*. The length of the array *ind* is always lesser than or equal to *n*. For instance let the list *L* has input elements 11, 11, 12, 16, 11, 19, 19, 17 and 14. Then after line 11, the hash table *H*, has the above numbers stored at the indices 1, 2, 6, 9 and 7 which is retrieved in sorted order in line 12. Thus the worst case complexity of the for loop at line 13 is *O(n)* where the list *L* has no duplicate elements. As a result the overall worst case complexity of this sub routine is *O(n).*

B. *Combinatorial HashSort*

Similar to the above analysis, let the input list consist of *n* elements. The time complexity of the lines 1, 2, 4, 5, 6, 7, 9 and 11 is *O(1).* The while loop at line 3 is executed *n* times and hence its complexity is *O(n).*It is evident that the array *ind* has *n* elements always and therefore the complexity of lines 10 to 12 is *O(n).* Thus the overall worst case time complexity of this sub routine is *O(n).*

C. *Key Sort*

Lines 1, 2, 3, 5, 6, 7, 8, 9, 10 and 11 has time complexity of *O(1).* At line 13 a recursive call is made. Let the input list *L* consist of *n* elements and the list L_s consists of n_s elements. It is evident that on successive recursive calls, the number of elements in list *L* of the successive recursion will be lesser than *n* as a result of partitioning the elements into buckets and the number of input elements will be lesser than or equal to 10 upon termination of the recursion. Hence we can say that the time complexity of this recursion will be lesser than or equal to *O(nlog k)* . The time complexity of line 15 is *O(n)* . Lines 18, 21 and 22 has the time complexity of *O(1).* The List data structure we use here has head and tail pointers and as a result the merge operation in line 21 is completed in constant time. The while loop at line 19 traverses all the active buckets denoted by list L_s and for each bucket it calls Combinatorial Hashsort subroutine. Let *n* be the number of elements in list L_s and let the *n* elements be partitioned into active buckets ranging from 0 to *k* . Let n_i be the number of elements in the i^{th} active bucket. Then *n* is given by the following equation.

$$n = \sum_{i=0}^{k} n_i \qquad (6)$$

For each iteration of the while loop the Combinatorial Hashsort subroutine is called which has time complexity of *O(*n_i*)* and by aggregate analysis the time complexity of the while loop at line 19 is *O(n).* Thus the overall worst case time complexity of this subroutine is *O(nlog k).*

D. *Just Sort*

Lines 1, 2, 3, 5, 6, 7, 8 and 10 has time complexity of *O(1).* The list data structure we use in this algorithm has head and tail pointers and so the insertion is done in constant time. Let the array A contain *n* elements. The for loop at line 4 has *n* iterations and hence its time complexity is *O(n).*Line 12 checks for the condition and if *n>=k/10* then we traverse Hashtable *H* of size *k/10* and proceed to sort directly whose complexity is *O(n)* as *n* is greater than or equal to *k/10.* Line 15 calls *keysort* subroutine and its worst case complexity is *O(nlog k).* Lines 16, 19 and 20 has time complexity of *O(1).* Again ,the List data structure we use here has head and tail pointers and as a result the merge operation in line 19 is completed in constant time. As array *A* has *n* elements, these elements are partitioned into the buckets represented by the entries of the hash table. The active buckets are inserted into list *L* at line 7 and this list is sorted by the *keysort* subroutine in line 15. As a result, he list *keys* in line 15 contains all the active buckets list *L* in sorted order. The sum of the elements present in all the active buckets is equal to *n.* The while loop at line 17 iterates over all the active buckets contained by the list *keys* which are in sorted order. Thus by aggregate analysis the time complexity of the while loop at line 17 is *O(n)* and hence the overall worst case time complexity of this subroutine is *O(nlog k).*

While the overall time complexity of this algorithm varies between *O(n) and O(nlogk)* , the space complexity of this algorithm is *O(k)* where *k* is the range of the numbers to be sorted. Based on the optimum and affordable value of memory, which varies greatly based on system configuration, the input array *A* can be partitioned into different segments of same or different range and each segment can be sorted independently and the sorted lists could be merged. In this way, the space complexity of this algorithm can be reduced to a desired value and also it can be used in distributed environment using parallelism and also external sorting can be used.

VI. Average case analysis

Lets us analyse the average case complexity of this algorithm by considering the distribution of inputs. The time complexity of this algorithm is O(n) when n is greater than k/10. We are calculating the average case complexity pessimistically by limiting the range of n(number of inputs elements) from 0 to k. The worst case complexity is O(n log k) if n is lesser than k/10 and it is O(n) if n is greater than or equal to k/10. For a given range k , excluding duplicates, there are 2^k possible inputs. Thus the sample set S for our input for a given range is 2^k. Let us consider two scenarios X and Y.

X- all possible combinations of S where n< k/10

Y - all possible combinations of S where n>= k/10

Let C denote the average computational cost of this algorithm. Then

$$C = P(X) * (n \log k) + P(Y) * (n)$$
$$P(X) = \sum_{I=1}^{\frac{K}{10}-1} kC_i \;/\; 2^k$$
$$P(Y) = \sum_{I=K/10}^{K} kC_i \;/\; 2^k$$

Where kC_i is the combinatorial notation. The curve for kC_i is bell shaped curve and it can be easily proved by mathematical induction that P(X) is lesser than 0.01. In fact when k is 10 , P(X) is 0.0097. And the value of P(X) reduces drastically to $0.017/2^{50}$ for k=100 and so on. Hence P(X) is infinitesimal and can be neglected.

Thus

$$C = P(Y) * (n)$$

Thus the average case complexity is *O(n).* In the above scenario all the events are considered to be likely and have equal probability.

VII. Adaptations

This algorithm can be easily applied to strings using its ASCII values and if the input consists of negative and positive values, values can be separated and then sorted and can be combined finally. Reversal of the *sort table* entries leads to the output being in descending order which can be used for sorting negative values. Based on the availability of memory space, the input elements can be partitioned range wise , sorted and then combined. In this way, the performance can be tuned as per the configuration of the system.

VIII. Performance evaluation

This algorithm is implemented in java, using *sort table* of order 10 and evaluated using the dataset T40I10D100K [2]. The list and hash table data structures are implemented using arrays. The sorting algorithms to be compared with this algorithm are Quick sort, Merge sort and Heap sort. The performance is measured in terms of execution time in Milli seconds.

TABLE III PERFORMANCE EVALUATION

Algorithm	**Execution time (ms)**
Just Sort	158
Quick Sort	250
Merge Sort	406
Heap Sort	765

From the above data it is clear that Just sort is nearly 37% faster than Quick Sort, which is the fastest among the three popular state of the art algorithms. The performance is measured in a system with the configuration of Intel i5 2nd generation desktop processor @ 3.00 GHz and 8 GB RAM.

IX. CONCLUSION

In this paper we have proposed a novel sorting algorithm and have elucidated and analyzed its correctness .Last but not the least, the name of this sorting algorithm points to the adjective form of the word ‘Just’.

X. APPENDIX

A. Extensive Mathematical Background of Sort Table , Sorting and its input data.

REFERENCES

[1] Cormen, Thomas H., Leiserson, C. E., Rivest, R. L., & Stein, C. Introduction to algorithms. MIT press, 2009.

[2] http://fimi.cs.helsinki.fi