CutShort: A Hybrid Sorting Technique

[1]Harshit Yadav ,[2]Shraddha Naik,[3]Urvashi Jalan
Department of Computer Science and Engineering.
Manipal Institute of Technology
Manipal,Karnataka,India,576104.
[1]harshit591@outlook.com
[2]shraddhanaik1995@gmail.com
[3]urvashijalan60@gmail.com

*Abstract--***There are many sorting techniques which were developed over hundreds of years and also optimized to reduce the complexity for worst and average cases. This paper presents a technique that can be used to optimize the sorting algorithms, named CutShort. The name 'CutShort' signifies cutting of original array into shorter pieces. In this technique, the input array is divided into several subarray of shorter lengths based on bit-count, somewhat similar to the Bucket sort concept. Here each sub-arrays contains those elements which can be represented by same number of bits. We have tested this proposed technique on random samples of large input array and the results are very satisfactory. This technique reduces the complexity of worst and average case by a significant factor depending on the number of sub-arrays formed. This method can be used as a preprocessing technique and can be more beneficial if implemented in the processor as a subroutine. It can be used to optimize the runtime of various sorting algorithms.**

Keywords- CutShort, BitCount, BitBand, Insertion sort, optimal sorting, hybrid sort.

I. Introduction

Sorting is the important and basic feature of many of the algorithm used in the computer. It is important because it improves the time complexity of searching, insertion and deletion algorithm in many data structure for example as in binary search. From the school days, we learned that any number having more number of digits is always greater than a number of less digits. This basic concept of our number system tells us that the two numbers can be compared on the basis of number of digits in them. The comparison between the elements of an array can also be made using the same concept, which means it provides an approach for sorting an array.

The traditional insertion sort [2] whose best case complexity time is O (n) and for average case O(n log n) . But in worst case, the complexity becomes O (n^2) because each insertion takes O(n) time and when compared to that of average case, it emerges to be high. Later, Bender, Farach and Martin [3] developed an optimised version of insertion sort that reduces the worst case complexity to O(n.log n).

The sorting technique uses the optimal sorting [4] method such as optimized version of insertion sort [3] , which helps in providing the worst case complexity to be O(n log n). Thus, the complexity of implementing CutShort cannot be more than O(n log n).

The organization of our paper is as follow: Section 2 presents an algorithm in detail. Section 3 summarizes the result ans discussion with detailed description of experimental setup. Conclusion is given in section 4. Finally the references used in this work are given at last.

II. CutShort: Algorithm

The working of this technique can better be understood if we divide its working into following steps:

1. Basis step
2. Range definition step
3. Repositioning step
4. Sub-array sorting step

We have discussed all these steps one by one in a chronologial order below.

1. Basis Step

: basis step of this technique uses BitCount which takes an integer value as argument and returns the minimum number of bits actually required to represent that number in binary form. Output of BitCount operation on some numbers is shown in Table1 below:

Table1: Output of BitCount Operation

Number	Binary Representation	(O/P of BitCount)
12	1010	4
53	110101	6
500	111110100	9
9999	10011100001111	14

Using this operation, we find the number of integers in the input array which requires the same number of bits and store these numbers in an array called *bitband*. In this bitband array, zero indexed cell is only reserved for containing the number of zeros in input array. The remaining entries of this table are filled up by the following steps:

a) Allocate an array, called bitband, of size 32 or 64 depending on system architecture.
b) Initialize each element of bitband to 0.
c) Take an element from the array and perform BitCount() operation on it.
d) The value returned by the bitcount() operation is used as the index for bitband and the value at that index is incremented by 1.
e) Repeat the step 3 and 4 for each element in the array.

For example consider the following input array:

0	1	2	3	4	5	6	7	8	9
15	10	19	49	13	2	7	4	1	3

10	11	12	13	14	15	16	17	18	19
10	20	17	15	46	16	53	0	5	9

Fig 1: Input Array

In this array, there is only 1 integer requires one bit (only 1), 2 integers require two bits(only {2, 3}), 3 integers require three bits (only {4, 5, 7}), 6 integers require four bits (only {9, 10, 10, 13, 15, 15}), 4 integers require five bits(only {16, 17, 19, 20}), and 3 integers require six bits (only {46, 49, 53}) for their binary representation. So the content of the bitband array will be as:

0	1	2	3	4	5	6
1	1	2	3	6	4	3

Fig 2: Bitband Array

2. *Range definition Step*

The range refers to the lower and upper bound position of the sub-array having only elements that have same BitCount. It can be cumulatively calculated by using the following procedure:

a) Start from the index k = 1
b) Add the preceding element's value (K-1) with the value at index k in BitBand and store it at the same index k.
c) Increment k by 1.
d) Repeat the steps until the last element is reached in the BitBand

example, the ranges for the sub-array can be defined as, element 0 will have the range [0,1), element {1} will have the range [1,2), elements {2,3} will have the range [2,4), element {4,5,7} will have the range [4,7), element {9,10,10,13,15,15} will have the range [7,13), element {16,17,19,20} will have the range [13,17), element {46,49,53} will be in the range [17,20). The range values calculated by the above procedure are stored in the bitband array as:

0	1	2	3	4	5	6
1	2	4	7	13	17	20

Fig 3: Resultant Bitband Array

3. *Repositioning step*

his phase, the elements are re-positioned in their specific ranges, that is, the elements are placed inside a range for a sub-array having elements with same BitCount without using any extra array for storing the elements. Repositioning can be done by following procedure:

a) Create an array of same size as BitBand.
b) Initialize each element with the value 0.
c) Starting from index 0,that is, k=0
d) Perform the bitcount() operation on the value in the array at index k and store it in a variable say 'count'
e) Finding the appropriate position of the element
   pos = bitband[count]+ bitmap[count]
f) If the element at index k is in the specific range of the sub-array it belongs to, then increment k by 1.
   K = k+1
g) Otherwise, swap the elements at index k and index 'pos'
h) Also increment the bitmap element at index 'count'

bitmap[count] = bitmap[count] + 1

i) Repeat the steps 4, 5 and 6 until k reaches the last element of the input array.

For example, after re-positioning the elements, the original input array would be:

0	1	2	3	4	5	6	7	8	9
0	**1**	**2**	**3**	**7**	**4**	**5**	**15**	**10**	**13**

10	11	12	13	14	15	16	17	18	19
10	**15**	**9**	**19**	**20**	**17**	**16**	**49**	**46**	**53**

Fig 4: Resultant Input array after Repositioning

4. Sub-arrays Sorting Step

Now each sub-array can be sorted using any of the sorting techniques with the help of BitBand array that defining the ranges of the sub-arrays. Generally in this situation Insertion sort is used because it provides the linear order time complexity for the smaller sublist. In our experimental setup we have also used this sorting as a post processing step. Procedure is as follows:

a) Starting from index, k = 2 (as first array will contain only zeroes)
b) Sort the sub-array ranging from bitband[k-1] to bitband[k]
c) Sort (array, bitband[k-1], bitband[k])
d) Repeat step 2 until k reaches the last of the bitband.

After applying these steps the input array contains the sorted list of elements as shown in figure 5 given below:

0	1	2	3	4	5	6	7	8	9
0	**1**	**2**	**3**	**4**	**5**	**7**	**9**	**10**	**10**

10	11	12	13	14	15	16	17	18	19
13	**15**	**15**	**16**	**17**	**19**	**20**	**46**	**49**	**53**

Fig 5: Sorted result in input array

III. Results and Discussion

In the experimental setup the proposed methodology is tested in Borland C++ compiler version 5.5 with IDE Codeblock version 10.05. All these software are installed on a machine having Intel Core i3-4005U 1.70Ghz processor with RAM 4GB DDR3 and Windows 10 Enterprise operating system.

Each test case contains one lakh elements and these test cases are divided into three categories to check effectiveness of our proposed techniques in all dimesions:

a) Random cases: In this case the input array contains all randomly choosen elements.
b) Worst cases: In this, all the elements of the array belongs to a single range (2^i to 2^i+1). Here i is any positive integer i.e. all elements lies in the single bucket.
c) Favourable cases: In this, the total number of elements in the array can be equally divided into 3 or more ranges(upto 31 or 61 depending on the architecture) iteratively. One bucket is reserved for the zeros.

The comparative results of the proposed algorithm is shown in the following bar charts:

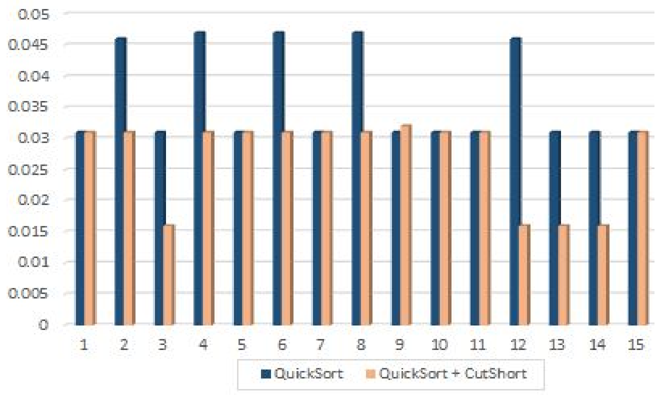

Fig 6: Comparison b/w quick sort and CutShort + Quick sort

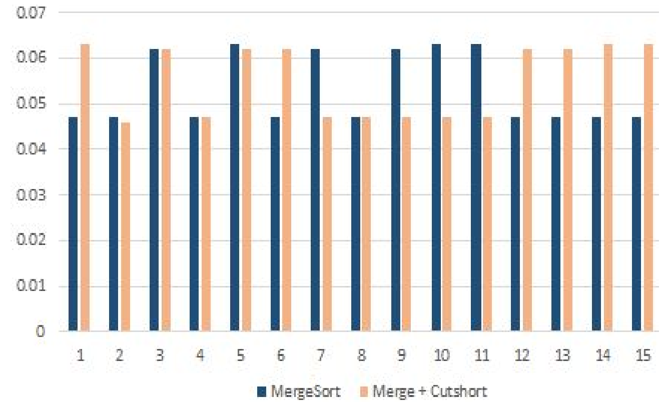

Fig 7: Comparison b/w Merge sort and CutShort + Merge sort

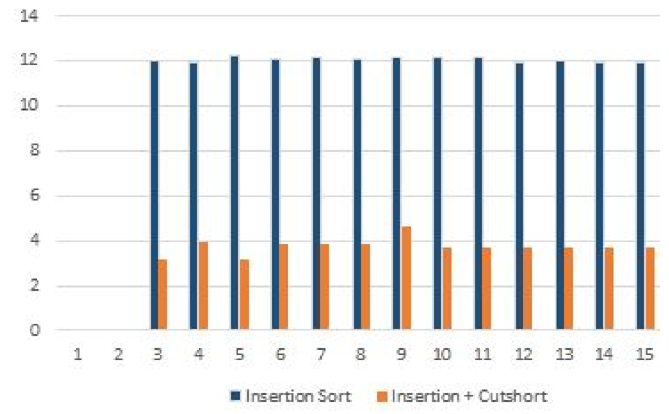

Fig 8: Comparison b/w Insertion sort and CutShort + Insertion sort

The complexity of sorting technique is considered in different cases i.e. best, average and worst cases. Since this technique uses another optimal sorting method to sort the sub-arrays and most sortings have optimal time complexity O(n.log n).

Best Case:

The best case occurs when the array is divided into several sub arrays of almost equal length. Thus, the length of input array is reduced to dmax times and processing time reduces by *n log(dmax)*, here dmax is the maximum number of sub-arrays obtained from original array.

n1 = n2 = n3 = n4 ….

where n1 + n2 + n3 + n4 + ….. = n

T(n) = O(n1.log n1 +n2.log n2 + n3.log n3 + ..)

T(n) = O((n1.log n1) * dmax)

T(n) = O((n/ dmax) log (n/dmax) * dmax)

T(n) = O(n log (n/dmax))

T(n) = O(n log (n) – n log (dmax))

Worst Case:

The worst case occurs when all the elements belongs to a single sub-array and the worst case complexity for optimised sorting method is O(n log n). So, in worst case, the time complexity will remain the same O(nlog n).

T(n) = O(n.log n)

Average Case:

In the average case, consider that this technique have divided the array in atleast d different sections and on combining the factor d with the worst case complexity of traditional and optimal sorting techniques of O(n.log n), we find T(n) = O(n log n/d)

T(n) = O(n log (n) – n log(d))

where d is the number of resulting sub-arrays.

Amortized analysis of Cut Short Algorithm

We used the Aggregate Analysis to calculate the Amortized cost of this algorithm. In this method we find an upper bound on the total sequence of n operations. The average cost per operation is called the Amortized cost.

- To read file, we require constant amount of time i.e. O(1).
- Bitmap array takes O (n logn) time as it has to convert n decimal values into binary numbers and get the bit count.
- The BitBand array takes constant time to rearrange the numbers with same BitCount in 1 group i.e. O(1).
- The quick sort algorithm takes O (n logn) in worst case.

Total time taken = O (1+nlogn+1+nlogn)

= O (nlogn)

Amortized cost Analysis:

Total no of operations:

- n decimal numbers to be converted into binary numbers would involve n arithmetic operations.
- To rearrange these n numbers would require n swaps in worst case.
- Quick sort involves the partition function which is the heart of the algorithm and it requires to n operations to be performed.

Total no of operation = n + n + n

= 3n

Amortized cost = Total time taken / Total no of operations

= O (n logn/3n)

= O (logn)

IV. Conclusion

Complexity and Cost analysis of the Cutshort has been studied thoroughly in the following report

References

[1] Thomas H. Coreman, Charles E. Leiserson and Ronald L. Rivest, *Introduction to Algorithms,* MIT Press, Third edition, 2009.

[2] Traditional insertion algorithm – en.wikipedia/wiki/ Insertion_sort, three lines implementation and five-lines optimized version by Jon Bentley (1999), Programming Pearls. Addison-Wesley Professional.

[3] Bender, Michael A; Farach-Colton, Martín; Mosteiro, Miguel (2006), Insertion Sort is O(n log n) SUNYSB; http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.60.3758

[4] Link: en.wikipedia.org/wiki/Sorting_ algorihm

RCT Lee, SS Tseng, RC Chang and YT Tsai, *Introduction to the Design and Analysis of Algorithms*, Mc Graw Hill, 2005