AS SORT

Rohith PR

Department of Information Science and Engineering

RNS Institute of Technology, Bengaluru

rohithpr@outlook.com

***Abstract*: A new stable comparison-based sorting algorithm called AS Sort has been described in this paper. This algorithm makes use of the advantages of both arrays and lists to achieve a best case efficiency of O(n) and a worst case efficiency of O(n ln(n)). This algorithm requires O(n) extra space. It sorts elements in almost O(n) time when elements are partially in ascending or descending order which are common cases.**

Keywords- sorting; building ordered sequences;

I. INTRODUCTION

Arrays have the advantage that their elements can be accessed in O(1) time but adding an element at the start of the array will force us to move all the existing elements. The size of the array must also be known in advance. These drawbacks can be overcome by making use of linked lists. Adding elements to the start of the list is easy but adding it to the end of the list is an O(n) operation unless a pointer to the last node of the list has been stored somewhere. A HEADER which is an array and LISTs which are linked lists are used to get the advantage of both an array and a list. A new algorithm that uses these data structures has been described in this paper.

II. RELATED WORK

A. Preprocessing input data

Cormen et al. [1] describe a method to build a min-heap in O(n) time. By applying this to the input array we can reduce the total number of lists that will be created.

B. Merging sorted lists

Knuth [2] describes a multiway merge algorithm which merges k sorted lists in O(n ln(k)) time where n is the total number of elements. Knowing this, we build multiple sorted lists with O(n ln(n)) comparisons and merge them using the multiway merge algorithm to get a completely sorted array.

III. DATA STRUCTURES USED

LIST: Linked list. These are ordered sequences.
The operations that can be performed on them are:
1. insert: Adds an element to the START of the list in O(1) time.
2. append: Adds an element to the END of the list in O(1) time. This operation can be done in O(1) time as the header stores a pointer to the last element of the list.
3. pop: Removes the first element of the list.

HEADER: It is an array of structures. The number of structures that will be needed in the worst case is equal to floor(n/4)+1. Let us assume that the header is infinitely long, for now.

Each structure has the following items:
1. START: First element (or position) of a LIST.
2. START_P: Pointer to the first element of a LIST.
3. END: Last element (or position) of a LIST.
4. END_P: Pointer to the last element of a LIST.
5. VALID: Boolean indicating whether or not a LIST is in use.

The header has the following attributes and methods:
1. LENGTH: The number of LISTs that the header is currently pointing to.
2. update(START_P): Pointer to the first element of a list is passed to the function. LENGTH is incremented and the structure items of the newly valid structure are updated.

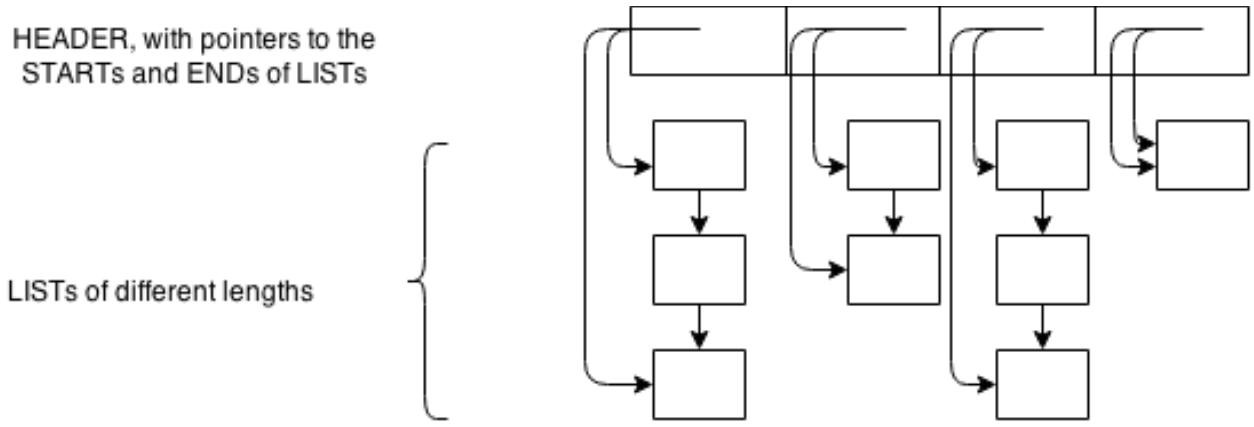

Figure 1: Data structures used

```
algorithm AS_SORT(input_array):
        for each element in input_array:
                BUILD_SEQ(element)
        end for
        output_array = merge()
end AS_SORT

algorithm BUILD_SEQ(ELE):
        if HEADER is empty:
                create a LIST with ELE as its only element and update the HEADER
                return
        for i = 0 to HEADER.LENGTH:
                if ELE >= HEADER[i].END:
                        append ELE to HEADER[i]
                        return
                else if ELE < HEADER[i].START:
                        insert ELE to HEADER[i]
                        return
        end for
        create a LIST with ELE as its only element and update() the HEADER
        return
end BUILD_SEQ
```

IV. ALGORITHM

The sorting process starts by looping over each element of the input array and calling the function BUILD with the current element as the parameter.

Working of BUILD: We loop over all the existing lists to check if the current element can be added to any one of them. If it cannot be added to any one of the existing lists we simply create a new list and update the HEADER.
Checking works as per the following rules:
1. If the element is greater than or equal to the last element of the list we append the element at the end of the list and return.
2. If the element is lesser than the first element of the list we insert the element to the start of the list and return.
3. If the element has not been added to the current list we simply move on to the next list.

A. Random case: Let the input array be: [4, 1, 8, 5, 4, 9, 3, 5, 4, 0]

Initially HEADER = []
BUILD(4): [[4]] # New list is created
BUILD(1): [[1, 4]] # 1 < 4
BUILD(8): [[1, 4, 8]] # 8 > 4
BUILD(5): [[1, 4, 8], [5]] # 5 > 1 and 5 < 8 so next list
BUILD(4): [[1, 4, 8], [4, 5]]
BUILD(9): [[1, 4, 8, 9], [4, 5]]
BUILD(3): [[1, 4, 8, 9], [3, 4, 5]]
BUILD(5): [[1, 4, 8, 9], [3, 4, 5, 5]]
BUILD(4): [[1, 4, 8, 9], [3, 4, 5, 5], [4]]
BUILD(0): [[0, 1, 4, 8, 9], [3, 4, 5, 5], [4]]
We now have 3 sorted lists with 5, 4 and 1 elements in them.

B. Worst case: One of the worst case inputs for this algorithm has been identified as the array of the form:
[min, max, min+1, max-1, min+2, max-2, ...]
Example for worst case: [1, 10, 2, 9, 3, 8, 4, 7, 5, 6]
HEADER after each BUILD is as follows:

BUILD(1): [[1]]
BUILD(10): [[1, 10]]
BUILD(2): [[1, 10], [2]]
BUILD(9): [[1, 10], [2, 9]]
BUILD(3): [[1, 10], [2, 9], [3]]
BUILD(8): [[1, 10], [2, 9], [3, 8]]
BUILD(4): [[1, 10], [2, 9], [3, 8], [4]]
BUILD(7): [[1, 10], [2, 9], [3, 8], [4, 7]]
BUILD(5): [[1, 10], [2, 9], [3, 8], [4, 7], [5]]
BUILD(6): [[1, 10], [2, 9], [3, 8], [4, 7], [5, 6]]
We now have 5 sorted lists with 2 elements each i.e. n/2 lists with 2 elements each.

Analysis for worst case:
The basic operation is comparing if the element can be inserted into a given list.
For the first two BUILDs the basic operation is performed once.
For the third and fourth BUILDs the basic operation is performed twice.
For the fifth and sixth BUILDs the basic operation is performed thrice.
...
For the (n-1)th and nth BUILDs the basic operation is performed (n/2) times.

Therefore, BUILD performs O(n) comparisons.
As BUILD is called n times, the worst case efficiency class of the sorting algorithm is $O(n^2)$.

C. Best case: There are many combinations of input_arrays that qualify as the best case. One of these combinations is an array that is in descending order. Example for best case: [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

The HEADER, after each BUILD is as follows:
BUILD(10): [[10]]
BUILD(9): [[9, 10]]
BUILD(8): [[8, 9, 10]]
BUILD(7): [[7, 8, 9, 10]]
BUILD(6): [[6, 7, 8, 9, 10]]
BUILD(5): [[5, 6, 7, 8, 9, 10]]
BUILD(4): [[4, 5, 6, 7, 8, 9, 10]]
BUILD(3): [[3, 4, 5, 6, 7, 8, 9, 10]]
BUILD(2): [[2, 3, 4, 5, 6, 7, 8, 9, 10]]
BUILD(1): [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
We now have one sorted list with n elements.

Analysis for best case:
Each time BUILD is called, the basic operation is performed once. Therefore the best case efficiency class of the algorithm is O(n).

So far, the best case efficiency is O(n) and the worst case efficiency is $O(n^2)$.

V. IMPROVING BUILD

By observing the HEADER for various cases we find the following:
If the HEADER is [[a..f], [b..e], [c..d]] then a < b < c < d < e < f.
That is:
1. STARTs of all the lists are in ascending order .. (P1)
2. ENDs of all the lists are in descending order .. (P2)
3. STARTs of each list is lesser than the ENDs of all lists .. (P3)

A simple example of the above is the final state of the HEADER for the worst case input and it can be verified for any input. By making use of the above properties and a modified version of binary search BUILD can be rewritten to improve its efficiency to O(ln(n)).

The properties of the lists that were described above hold true even for lists built using BUILD_BINARY. BUILD_BINARY performs O(ln(n)) comparisons. The worst case efficiency class of the sorting algorithm is O(n ln(n)).

```
algorithm BUILD_BINARY(ELE):
        if HEADER is empty:
                create a LIST with ELE as its only element and update the HEADER
                return
        low = 0
        high = HEADER.LENGTH – 1
        while low <= high:
                mid = (low + high) / 2
                if mid == 0:
                        if ELE >= HEADER[mid].END:
                                append ELE to HEADER[mid]
                                return
                        else if ELE < HEADER[mid].START:
                                insert ELE to HEADER[mid]
                                return
                        else:
                                low = mid + 1
                else if ELE >= HEADER[mid].END:
                        if ELE <= HEADER[mid-1].END:
                                append ELE to HEADER[mid]
                                return
                        else:
                                high = mid – 1
                else if ELE < HEADER[mid].START:
                        if ELE >= HEADER[mid-1].START:
                                insert ELE to HEADER[mid]
                                return
                        else:
                                high = mid – 1
                else:
                        low = mid + 1
        end while
        create a LIST with ELE as its only element and update the HEADER
        return
end BUILD_BINARY
```

Number of lists								
n	Without heapify				With heapify			
	Ascending	Descending	Worst case	Random	Ascending	Descending	Worst case	Random
2^3	1	1	4	2.523	1	2	2	2.024
2^6	1	1	32	8.669	1	8	4	6.823
2^9	1	1	256	28.047	1	15	7	21.757
2^12	1	1	2048	84.611	1	21	10	65.433
2^15	1	1	16384	247.246	1	27	13	191.201
2^18	1	1	131072	709.233	1	33	16	542.788

Table 1. Number of lists used for n elements
(Averages of a 1000 trials each for random cases)

VI. MERGING THE LISTS

All the lists that we have built so far are sorted. These sorted lists can be merged by using the multiway merge algorithm [2] in O(n ln(k)) time, where k is the number of lists. As k < n in all cases, the efficiency class of the sorting algorithm remains unaffected.

VII. NUMBER OF BEST CASE INPUTS

n distinct elements can be arranged in n! ways in an array. Of these, $2^{(n-1)}$ arrangements qualify as the best case input. Any input_list that can be built by following the steps mentioned below qualify as the best case input and will have just one sorted list to be merged

1. Consider a sorted list that has n distinct elements.
2. Pop any number from this list and place it in the first position in the input_array. We now have two lists: left_list and right_list to the left and right of the element that was popped.
3. Elements from the left_list can be popped only from the right end. Elements from the right_list can be popped only from the left end.
4. Pop elements from the left and right_list randomly (but follow the previous rule) and fill the input_array from left to right.

It can be verified that the number of input_arrays that can be built using the above procedure is $2^{(n-1)}$, which is the number of best case inputs for AS Sort.

VIII. STABILITY

AS Sort is a stable sorting algorithm. Care should be taken to ensure that if the left and right nodes have the same value, while merging, then the left node must be promoted as the element in the left list would have been added before the element in the right node.

IX. FURTHER IMPROVEMENT

NOTE: This is applicable only if stability is not required. It also affects the number of best case inputs.

By using heapify [1] and building the HEADER from the last element of the input_array we can ensure that the HEADER's maximum length is floor(n/4)+1. Since stability is no longer an issue, BUILD_BINARY can be modified so that elements can be inserted into a list even if its value is equal to the START of the list. This reduces the number of comparisons done in the BUILD function and while merging. Refer table 1 for details.

X. RESULT

Algorithms such as Bubble Sort and Insertion Sort do very well if the input is close to the best case but have an efficiency of $O(n^2)$ in the worst case. Algorithms such as Heap Sort and Merge Sort have an efficiency of O(n ln(n)) for all cases of inputs and perform much better than Insertion Sort for worst and average cases. AS Sort sits comfortably in between by having an efficiency class of O(n) in the best case and O(n ln(n)) in the worst case. It is well suited to add a few elements into a list that is already sorted. At the same time we can sort completely random elements in very good time. This eliminates the need for implementing two algorithms for the said tasks. Since AS Sort can be a stable sorting algorithm, if required, so it can be used instead of Quick Sort in situations that require stability.

XIII. REFERENCES

1. Cormen, Thomas et al.: Introduction to Algorithms, Third Edition, section 6.3
2. Knuth, Donald: The Art of Computer Programming, Volume III - Sorting and Searching, section 5.4.1