QUINSERTION: A HYBRID SORTING TECHNIQUE

Pavi Saraswat
Department of IT
Amity University, Sector-125
Noida India

Anurag Yadav
Department of IT
Amity University, Sector-125
Noida India

Abstract—Sorting is the basic problem in computer science. In this we have to rearrange data like numerical or alphabetical either in ascending or in descending order. There are many sorting algorithms in literature like Merge sort, Heap sort, Quick sort, Insertion sort, Smooth sort and many more. There is a drawback in quick sort that its complexity is O(n^2) when data is already in sorted manner. To overcome this problem we are introducing an algorithm called Quinsertion sort which is combination of quick sort and insertion sort.

Keywords— Sorting Algorithm, Quick Sort, Insertion Sort, Data Structures, Hybrid Approach, Time Complexity

I. INTRODUCTION

Sorting has been a thoughtful area for the algorithmic researchers. And a lot of resources are invested to put forward a more working sorting algorithm. For this reason many existing sorting algorithms were experiential in terms of the efficiency of the algorithmic complexity [1]. Quicksort [2] was observed to be both cost-effective and efficient. A lot of algorithms are very well recognized for sorting the unordered lists. Most significant of them are Heap sort, Bubble sort, Quicksort, and Insertion sort [3]. Efficient sorting is significant to optimize the utilization of other algorithms that necessitate sorted lists to work in the approved manner; it is also often in producing human-readable output [4]. The insertion sort is used in combination with quick sort as the time complexity of the insertion sort for best case is O(n) which is linear. Insertion sorting algorithm is another important algorithm, used for sorting small lists. But the study shows that the EIS is more competent, in theory, logically, and practically as compared to the real insertion sorting algorithm and also good for sorting larger lists. Sorting has been well thought-out as an elementary problem in the study of algorithms, that due to many reasons:

- The need to sort information is intrinsic in many applications.
- Algorithms often use sorting as a key subroutine and competent sorting is important to optimize the utilization of new algorithms that necessitate sorted lists to work appropriately.

II. QUICK SORT

An array having large number of elements with a random order needs to be ordered in an ascending or descending manner. Quick sort can deal with this problem by using two key ideas: The first idea of Quick sort is that the problem can be divided and solved. The problem can be broken into smaller arrays, and those smaller arrays can be sorted easily. This is done by choosing an element from the array as pivot element, and reordering the array so that all elements which are smaller than the pivot appear before the pivot and all elements which are larger than the pivot come after it. This process is called partitioning process. Second, by partitioning the array into two sub parts, then partitioning those two parts recursively into arrays of single elements, two already sorted arrays can be concatenated on either side of the chosen pivot element into one array, as an array with one element is already sorted. Refer to the following pseudocode:

```
Input -- A array of n elements, left -- left sub array from the pivot, right -- right sub array from the pivot

Output -- array A sorted in ascending order
1. proc quicksort(A, left, right)
2. if right > left
3.  pivot = random(A)
4.  newPivot = partition(A, left, right, pivot)
5.  quicksort(A, left, newPivot - 1)
6.  quicksort(A, newPivot + 1, right)
7.  return A
```

Fig. 1. Pseudocode for Quick Sort Algorithm

The output should satisfy 2 major conditions:
- The output is a combination, or reordering, of the input.
- The output is in non decreasing order.

The rest of the paper is organized as follows section II contains Quick sort, Section III contains Insertion sort, section IV contains proposed Quinsertion sort Algorithm, section V contains Results and section VI contains Conclusion.
As the pseudocode shows, after a new pivot element is randomly chosen (lines 3-4), the array is broken up into a left half and a right half and sorted recursively (lines 5-6). In the partition algorithm, the two sub arrays are compared to determine how they should be arranged (lines 12-15). In the following examples, using the given input, the partitioning of the array (Figure 3) and how the sub arrays are concatenated back into a sorted array (Figure 4) are illustrated.

Inputs – A: array of $n$ elements (33 55 77 11 66 88 22 44)
Output – array A in ascending order

III. INSERTION SORT


Here is a pseudo code of this algorithm:

```plaintext
Algorithm InsertionSort(A[0..n-1])
// Input: An array A[0..n-1] with n elements
// output: An array A[0..n-1] sorted in ascending order
for(i=1;i<n-1;i++)
{
    t=A[i];
    j=i-1;
    While(j >= 0 && t < A[j])
    {
        j=j-1;
    }
    A[j+1] = t;  // Insert element in proper place
}
```

As the example shows, array A is broken in half based on the pivot until they are in arrays of only a single element, then those single elements are concatenated together until they form a single sorted array in ascending order.
A. Number of comparisons

The basic operation of this algorithm is the key comparison. In this algorithm, the number of key comparisons depends on the nature of the input. The worst case occurs, when input of an array are of reverse order. The number of key comparisons for such an input is

\[ C(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} (i - 1) - 0 + 1 = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2) \]

B. Number of movements

The number of key movements is

\[ M(n) = 1 + 2 + 3 + \ldots + (n-1) = \frac{n(n-1)}{2} \]

Since, the number of keys that needs to be moved in i-th iteration is i.

C. Number of swaps

The total number of swaps when the array are of reverse order, S(n) = 0

IV. QUINSERTION ALGORITHM

As we discussed above that quick sort is having a drawback that its performance complexity is O(n^2) when data entered is already in sorted manner so it’s the problem of sorting algorithm to minimize this problem we are introducing a algorithm called quinsertion sort which is combination of quick sort and insertion sort. As it is the property of insertion sort that when data is already in sorted manner it will only perform n comparisons which is exponentially less than of quick sort so we can combine these two algorithms which can result in better complexity of quick sort.

Here is a code sample of Quinsertion sort algorithm in C#.

class QuinsertionSort
{
    public void Quinsertion_Sort(int[] list, int start, int end)
    {
        if (start < end)
        {
            // This is where we switch to Insertion Sort!
            if ((end - start) < 9)
            {
                this.InsertionSort(list, start, end + 1);
            }
            else
            {
                int part = this.partition(list, start, end);
                this.Quinsertion_Sort(list, start, part - 1);
                this.Quinsertion_Sort(list, part + 1, end);
            }
        }
    }

    public void InsertionSort(int[] list)
    {
        for (int x = start + 1; x < end; x++)
        {
            int val = list[x];
            int j = x - 1;
            while (j >= 0 && val < list[j])
            {
                list[j + 1] = list[j];
                j--;
            }
            list[j + 1] = val;
        }
    }

    private int partition(int[] list, int leftIndex, int rightIndex)
    {
        int left = leftIndex;
        int right = rightIndex;
        int pivot = list[leftIndex];
        while (left < right)
        {
            if (list[left] < pivot)
            {
                left++;
                continue;
            }
            if (list[right] > pivot)
            {
                right--;
                continue;
            }
            int tmp = list[left];
            list[left] = list[right];
            list[right] = tmp;
            left++;
        }
        return left;
    }

    // Here in this code we break the array of inputs around the size of 9 because according to literature if we use hybrid approaches then the insertion sort gives the best result at this size only.
Quick sort is a recognized sorting algorithm proposed by Hoare [5] that on average basis makes O(nlogn) comparisons to sort n items. However it makes $\Theta(n^2)$ comparisons in its worst case. Quick sort is a comparison sort and, in proficient implementations, is not a stable sort. Stable sorting algorithms maintain the relative order of records which have equal keys. This means, a sorting algorithm is called stable if there are two items X and Y with the equal key values and with X present before Y in the given list then X will come before Y in the final sorted list [6, 7]. As mentioned in [8], there are three well known divide-and-conquer approaches for sorting a sub array A[p..r]:-

- Divide: The array A[p..r] is divided into two sub arrays (non empty), A[p..q] And A[q+1..r] such that every element of A[p..q] is less than or equal to every element of A[q+1..r]. q is computed as a part of partitioning procedure.
- Conquer: Now sub arrays A[p..q] and A[q+1..r] are sorted recursively.
- Combine: Since the sub arrays are sorted, no effort is required to combine them, and the entire array A is sorted now.

The code of the Quick sort algorithm has two parts. The first part is a method named quicksort, which places the pivot at its correct position in a single pass.

The steps of Quick sort [2, 9] are:

- Pick an element as pivot from the given list.
- Arrange the list as all elements less than to pivot is at one side and greater elements are at others. Elements equal to pivot can come to either side because Quick Sort is not stable. After this, the pivot is at its final position. This is called the partition procedure.
- Now recursively sort the two sub-lists of elements.

The QUINSERTION algorithm improves the way Quick sort divides the given array. Quicksort moves the pivot to its correct position and then divides the array into two sub parts, and recursively follows the same procedure, until it reaches the base case. Unlike this, the QUINSERTION algorithm divides the array into three parts (arrays), positive, negative, and frequent elements, and swaps each element to its correct position in a single pass.

Table -1 Running time of quicksort and Quinsertion sort

<table>
<thead>
<tr>
<th>Size of input</th>
<th>Running time of quick sort</th>
<th>Running time of Quinsertion sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>75ms</td>
<td>46ms</td>
</tr>
<tr>
<td>2000</td>
<td>78ms</td>
<td>49ms</td>
</tr>
<tr>
<td>5000</td>
<td>89ms</td>
<td>58ms</td>
</tr>
</tbody>
</table>

Table 1 shows the average running time for quinsertion and quick sort for different size of inputs.

VI. CONCLUSION

The quinsertion sort is an enhancement to quick sort which is trying to resolve the problem of quick sort which is that it takes maximum time complexity to sort already sorted data. Quick sort takes O(n^2) performance complexity for its worst case that is to sort already sorted data and quinsertion takes around O(n) performance complexity to sort already sorted data.
VII. REFERENCE


