DETERMINING THE PROPER SORTING ALGORITHM BASED ON THE GIVEN DATA

PARHAM HOUSHMAND, YASAMIN VAZIRI
MAHYA MOTTAGHI & ELHAM SOLEIMANI

SHARIF UNIVERSITY OF TECHNOLOGY

July 2022

CONTENTS

1 ABSTRACT

The cost of using sorting algorithms may have a considerable difference in that finding the proper algorithm reduces time and space and improves sorting efficiency. There are linear sorting algorithms that need to be preprocessed before they can be used. The quick sort algorithm also needs the data to be unsorted and sometimes according to the structured data we have (heap, linked list) they need another special algorithm (heap and merge sort) also sometimes the data is very voluminous, and We have to use an algorithm that performs in place sorting. According to these interpretations, we perform linear pre-processing on the primary data to find the proper sorting algorithm with the features we mention from each sorting algorithm.

keywords:algorithm, sorting, data, input, proper sorting algorithm, time complexity

2 INTRODUCTION

As there are different sorting algorithms, we know they have bold differences with each other and we are going to give a short summary about them. But the main reason of this paper is to construct the proper sorting algorithm with given input which has the minimum time complexity no matter what the size of input is.Our algorithm firstly checks each number behind the head whether it is ascending or descending, also checks whether the number is negative or not, then it finds the largest length of the number and the largest number, after that it checks its inversion and then it finds whether the inversion is larger than nlog n or not. Note that for radix and count sort the ascending order should not neither be less than nlog n nor negative.

3 METHODS

1. different methods of sorting
2. different techniques
3. Analysis of sorting techniques
4. proper sorting algorithm analysis

3.1 different sorts time complexity

1. Bubble sort and Insertion sort
   The average and worst case time complexity of both is n^2 while the best time complexity is n. This happens when the array is already sorted, instead the worst case is when the array is completely reversed. and as a use, these algorithms work better for the smaller data and insertion sort is of order of n when the array is sorted.
2. Merge sort
   Merge sorts best, average and worst case time complexity is$n \log n$ and again similarly independent of distribution of data. for this algorithm we can say it works the best when the array is reversed sort
3. Heap sort
   like merge sort, its best, average and worst case time complexity is $n \log n$ which is independent of distribution of data. heap sort's property is that the space is of order of 1.

4. Quick sort
   It is a divide and conquer approach with recurrence relation:
   $$T(n) = T(k) + T(n-k-1) + cn$$
   Its worst case is when the array is sorted or reverse sorted, the partition algorithm divides the array in two sub arrays with 0 and $n-1$ elements. Therefore,
   $$T(n) = T(0) + T(n-1) + cn$$
   Solving this we get:
   $$T(n) = O(n^2)$$
   On an average, the partition algorithm divides the array in two subarrays with equal size. Therefore,
   $$T(n) = 2T(\frac{n}{2}) + cn$$
   and finally
   $$T(n) = O(nlogn)$$
   the algorithm is of order of n^2 when the array is revered. Non-comparison based sorting:
   In non-comparison based sorting, elements of array are not compared with each other to find the sorted array.

5. Radix sort
   Best, average and worst case time complexity for radix sort is nk where k is the maximum number of digits in elements of array.

6. Count sort
   For this algorithm best, average and worst case time complexity is $n+k$ where k is the size of count array.

3.2 common techniques

1. in place & out place technique
   A sorting technique is in place if it does not use any extra memory to sort the array. Among the comparison based techniques discussed, only merge sort is outplaced technique as it requires an extra array to merge the sorted sub arrays. Among the non-comparison based techniques discussed, all are outplaced techniques. Counting sort uses a counting array and bucket sort uses a hash table for sorting the array.
2. on line & off line technique
   A sorting technique is considered Online if it can accept new data while the procedure is ongoing i.e. complete data is not required to start the sorting operation. Among the comparison based techniques discussed, only Insertion Sort qualifies for this because of the underlying algorithm it uses i.e. it processes the array (not just elements) from left to right and if new elements are added to the right, it doesn't impact the ongoing operation.
3. Stable & unstable technique
   A sorting technique is stable if it does not change the order of elements with the same value. Out of comparison based techniques, bubble sort, insertion sort and merge sort are stable techniques. Selection sort is unstable as it may change the order of elements with the same value. For example, consider the array $4, 4, 1, 3$. In the first iteration, the minimum element found is 1 and it is swapped with 4 at 0th position. Therefore, the order of 4 with respect to 4 at the 1st position will change. Similarly, quick sort and heap sort are also unstable. Out of non-comparison based techniques, Counting sort and Bucket sort are stable sorting techniques whereas radix sort stability depends on the underlying algorithm used for sorting.

3.3 sorting techniques analysis

1. When the array is almost sorted, insertion sort can be preferred.
2. When order of input is not known, merge sort is preferred as it has worst case time complexity of $n \log n$ and it is stable as well.
3. When the array is sorted, insertion and bubble sort gives complexity of n but quick sort gives complexity of n^2.

3.4 Constructed Algorithm

firstly it checks the inversion number of the array then it checks that if it is ascending or descending and if it has negative element and it finds max value and max value digits then it checks if number of inversions are less than nlogn and number of ascending and descending numbers are equal to the length of the array or the length of the array is (0,20] the we sort with insertion. if it is only descending (reversed sorted)we sort with merge sort and if it is completely ascending we use insertion. now if it doesn't contain any negative number we check if the max value is less than nlogn we use count sort and if max number of digits is less than nlogn we return radix sort.if none of these algorithms were not the answer we choose between quick and merge sort. if number of ascending and descending numbers are less than n we use quick sort, else we return merge sort.
The python code of the algorithm would be like:

```python
from math import log


def pyme(array: list):
   """
   This function is the implementation of the algorithm.
   It takes an array as input and returns the best sorting algorithm.
   """
   """
   The complexity of this algorithm is O(n log n).
   we can improve the algorithm by using the following optimizations:
   1. We can ignore the inversions because we use it for determining the best
       sorting algorithm on small arrays
   and we can ignore it for large arrays.
   2. because we use the max_value and max_number_of_digits to determine the
       best sorting algorithm for arrays with
   no negative numbers, we can ignore them for arrays with negative numbers and
       we can just find
   the max number of digits by the number that has the max value.
   """
   is_asc, is_desc, has_negative, max_value, max_number_of_digits =
       __analyse(array) # Analyse the array for
   # finding attributes of the array
   inversions = inversion_count(array) # Count the inversions for Insertion Sort
   n_log_n = len(array) * log(len(array), 2) # Calculate the n log n
   if inversions < n_log_n or (is_asc == len(array) - 1 and is_desc ==
       len(array) - 1) or \
         20 >= len(array) > 1 or is_asc == len(array) - 1:
      return "Insertion Sort" # If the inversions are less than n log n or
      # the array is sorted in ascending order or if the array is small or all
          keys are the same we use Insertion Sort
   if is_desc == len(array) - 1:
      return "Merge Sort" # If the array is sorted in descending order we use
          Merge Sort
      # by test, it works better than others
   if not has_negative: # count sort and radix sort can only be used if the
       array has no negative numbers
      if max_number_of_digits < log(len(array), 2) and max_number_of_digits *
          len(array) < max_value + len(array) \
            and max_number_of_digits * len(array) < n_log_n / 2:
         return "Radix Sort" # for the array with bigger rang than linear and
             the number of digits is smaller
         # than the biggest number of the array we use Radix Sort
      if max_value < len(array) * log(len(array), 2):
         return "Count Sort" # if the max value is small count sort is better
             than radix sort
   if is_asc < len(array) * 3 / 5 and is_desc < len(array) * 3 / 5:
      return "Quick Sort" # Quick Sort is better than Merge Sort for arrays
          with a lot of inversions
      # and if the array is sorted in ascending order and descending order we
          use other sorts.
      # it works better on big arrays.
   else:
      return "Merge Sort"
      # merge sort needs more space than quick sort and merge is better for
          small arrays.
      # merge sort need to copy array every time and quick sort need to copy
          array only once.
   # we didn't use heap sort because it is slower than merge sort and quick sort
   # bubble sort is just a simple sorting algorithm that doesn't use any
       optimizations.
   # bubble sort is easier to understand and it is just handy to use but by
       test it works slower than Insertion.


def __analyse(array):
   is_asc, is_desc, has_negative, max_value, max_number_of_digits = 0, 0,
       False, 0, 0
   temp_asc = 0
   temp_desc = 0
   for i in range(len(array) - 1):
      if array[i] < 0:
         has_negative = True
      if array[i] >= array[i + 1]:
         temp_asc = max(temp_asc, is_asc)
         is_desc = temp_desc + 1
      else:
         temp_asc = temp_asc + 1
         is_desc = max(temp_desc, is_desc)
      max_value = max(array[i], max_value)
      max_number_of_digits = max(len(str(array[i])), max_number_of_digits)
   is_asc = max(is_asc, temp_asc)
   is_desc = max(is_desc, temp_desc)
   max_value = max(array[len(array) - 1], max_value)
   max_number_of_digits = max(len(str(array[len(array) - 1])),
       max_number_of_digits)
   return is_asc, is_desc, has_negative, max_value, max_number_of_digits


def inversion_count(array):
   """
   This function is the implementation of the inversion algorithm.
   It takes an array as input and returns the number of inversions.
   """
   if len(array) <= 1:
      return 0
```

```
    mid = len(array) // 2
    left = array[:mid]
    right = array[mid:]
    return inversion_count(left) + inversion_count(right) + merge(left, right)


def merge(left, right):
    count = 0
    i = 0
    j = 0
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            i += 1
        else:
            count += len(left) - i
            j += 1
    return count
```

the order of the algorithm is: O(n log n)
for more information check the Git hub link.

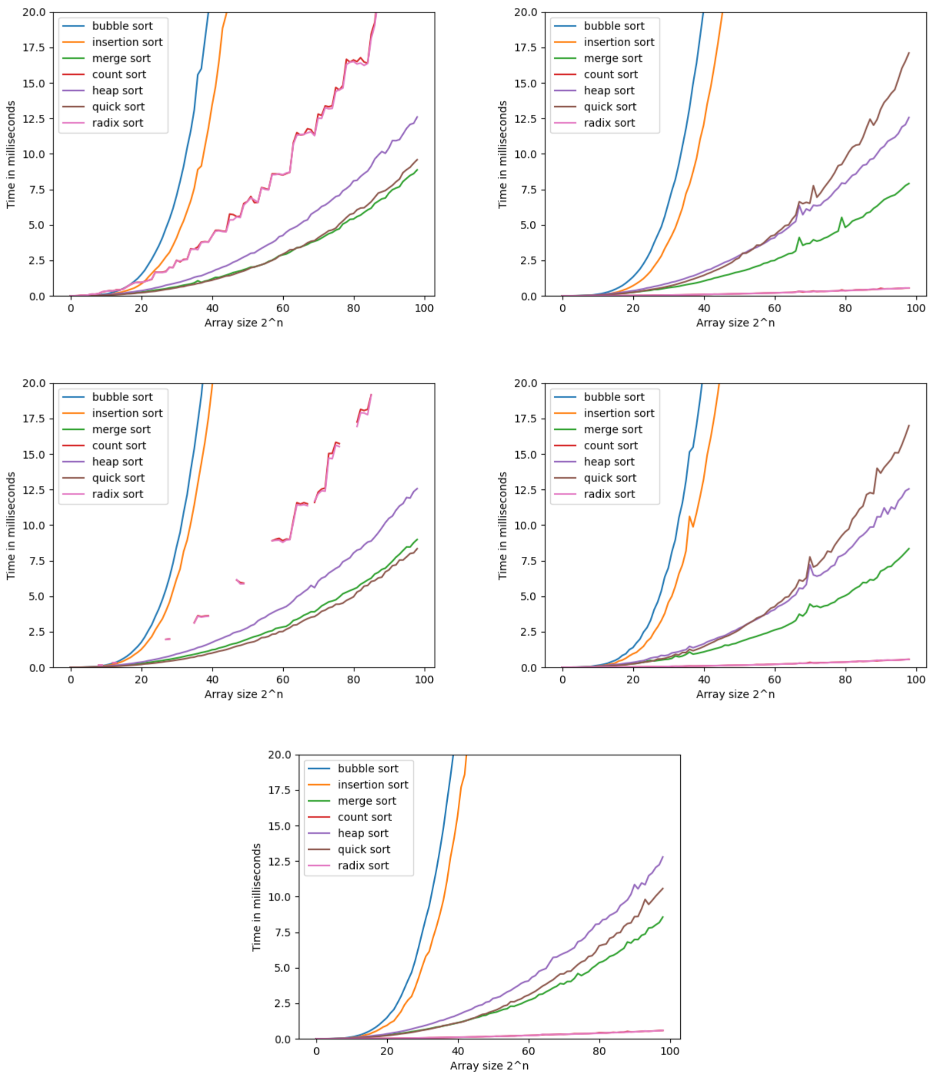

Figure 1: Results of testing in diffrenet conditions

To explain the error of the algorithm, due to the graphs above, this might be because of the CPU, the input, data streaming type, amount of RAM conflict etc. effect the coefficients of the order which are not shown in the order, so that the efficiency will get worse (see figures).
Also we can say that our algorithm's correctness average is between 70 to 80 % that we discuss later

4 RESULTS AND DISCUSSION

we theoretically made a method to understand the proper sorting algorithm and that was almost efficient.
the algorithm checks different factors in the given data and will return the answer.
we also use a test generator for making different tests and try it on our method and compare it with different sorting algorithms to see how well does it work.As an average, it's efficiency is between 70 to 80 %.
To see complete tests visit Git hub
see the table below:

factor	efficiency
sorted numbers	73 %
small numbers	81 %
reversed numbers	87 %
negative numbers	75 %
large numbers	70 %
duplicate numbers	81 %
duplicate negative numbers	78 %

The second column is the average of doing 100 tests on the algorithm And as you can see the least efficiency is for large numbers and the most is for duplicate numbers. For more studies it's good to make the method more efficient with a linear order that works the best for every kind of input.

5 REFERENCES

Geeksforgeeks.org
Introduction to Algorithms clrs

TEAM CONTRIBUTION :
Parham Houshmand:25 %
Yasamin Vaziri:25 %
Mahya Mottaghi:25 %
Elham Soleimani:25 %
all participated in discussing problems,paper writing and coding.