EE3980 Algorithms

Project. Comparing O(n) and O(nlgn) Sorts

By 105061212 王家駿

1. Introduction

In the previous homework, we implemented sorting algorithms with the time complexity of O(n), like counting sort or radix sort, as well as sorting algorithms with the time complexity of O(n*logn), such as heap sort. However, in the conclusion we had before, the linear sort was not faster than the heap sort in all test cases. Now, we would reuse these sorting algorithms to check if the conclusion still holds for a larger input data size. In addition, we would also try other sorting algorithms with the same complexity, and figure out that whether they are more efficient than what we used. In the end, we would analyze the pros and cons of those algorithms, and try to find out the reason to cause the differences.

2. Implementation

2.1. readInput

For the input data set, there contains a pile of floating numbers. If we store the numbers into the type of float or double, we would face the problem that it’s not precise enough, since it could only use the binary representation for the floating-point

number. When the number contains the information with many digits below the decimal point, we might not precisely represent the number by using floating point.

Using the floating number representation to store the numbers seems not a good idea. So, we run another program to find out that the range of the numbers. It follows that the input data are at the range about $+10^{10} \sim -10^{10}$, with the precision to at most 10^{-10}. Thus, we could assume that the input data contains at most 20 digits, 10 for the integer part and 10 for the decimal part.

Hence, instead of storing the number in the type of floating number, we would use an array of integer, where an integer represents one digit, and an extra digit at top to stand for the sign. The digit is one if the number is positive, and zero if it's negative. Because the positive numbers are always larger than the negative number, at the steps of sorting, the extra digit could determine the order of the positive and negative numbers by just comparing the first digit.

Thus, for a number, we would use an integer array with 21 integers to store it. The first integer stands for the sign, 2nd to 11th digits stands for the integer part, and the rest represents the decimal part.

For each number, we read the digit one by one, and store them in the array by fitting the format we set. As for the exponential number, we add additional statements to handle the situation, by simply shifting the digits depending on the exponent value.

After the reading process, all number had been stored into the array with the format, and then we could sort the numbers by using the format easily.

2.2. reverseNegative

```
1.  Algorithm reverseNegative(arr)          // reverse order of negative part
2.  {
3.      if arr[1][0] = 0 then return;       // all positive, nothing to do
4.
5.      for i := 1 to N do {
6.          if arr[i][0] = 1 then           // find the first positive number
7.              neg := i - 1 and break;
8.      }
9.
10.     if not find then neg := N;          // all negative
11.
12.     for i := 1 to neg / 2 do {
13.         swap arr[i] and arr[neg+1-i];   // swap the numbers
14.     }
15. }
```

After the sorting process, the numbers were sorted in increasing order. However, we encode the negative numbers with zero for the first digit. It’s not a problem in separating the positive and negative part, while in the negative part, the value is larger when the unsigned part is smaller. It would let the order become decreasing in the

negative part. Thus, we need to reverse the order in the negative part after the sorting was completed.

The algorithm would reverse the order in the negative part. At first, it would iterate through the array to find the position of the largest negative number. Then, for the negative part, we swap the first element and the last element, the second and the last second, and so on. After the process, we could get the whole negative part reversed.

For the time complexity, both the iteration and the swapping take $O(n)$ times since each element was swapped at most once. Thus, the total time complexity is $O(n)$. And because we only use the algorithm after the sorting, which is exactly once, the process would not affect the overall time complexity of the sorting algorithm.

For the space complexity, we only need an extra number array t to store the value during swapping, so the space complexity is $O(1)$.

2.3. Method of value comparison

In the comparison-based sorting algorithm, we would compare the value of numbers to decide the arrangement. Now we get an array of integer instead, so we could just compare the values digit by digit from the first one. Since we encode the sign as 0/1 in advance, the comparison doesn't have to be revised to fit the sign.

2.4. Sorting algorithms

In this homework, we use sorting algorithms with two different time complexities: $O(n)$ and $O(n * \log n)$. And we also tried to find the algorithm with the highest efficiency based on the same complexity.

For the $O(n)$ sort, we had known that there are *counting sort*, *bucket sort*, and *radix sort*. The counting sort would perform well when there are only few possible values, but in our case, there are many such values ($\cong 10^{21}$). The bucket sort would perform well if the element is sparse referring to the buckets, and it's difficult to determine the number of buckets since the input data size is quite large. Thus, it obvious that the radix sort is more efficient above the three sorting algorithms.

For the $O(n * \log n)$ sort, the quick sort has the worst-case time complexity of $O(n^2)$. While for the heap sort and the merge sort, it is difficult to tell which one is more efficient.

Thus, we would implement radix sort, heap sort, and the merge sort, and compare the time efficiency of the three algorithms.

2.4.1. Radix Sort

```
1. Algorithm RadixSort(A, d)
2. {
```

```
3.      for i := max_string_length to 1 step -1 do {
4.          Sort array A by digit i using CountingSort;
5.      }
6.  }
```

With the input data stored in the array, we use radix sort to sort it in increasing order. Radix sort applies sorting algorithm with respect to the certain digit from the LSB to MSB, and the sorting algorithm has to be stable. Thus, when we sort the higher digit, the lower digit we have sorted would not change its order if two of the higher digits are same. That is, the two numbers would arrange in the increasing order.

In our implementation, we use the counting sort for the stable sorting algorithm. And then we seem each integer in the array as a digit to be sorted, so we apply counting sort from the 21st index of the array (LSB) to the first (MSB). The result would be all the number arranged in increasing order.

```
1.  Algorithm CountingSort(A,B,n,k)
2.  {
3.      // Initialize C to all 0
4.      for i := 1 to k do {
5.          C[i] := 0;
6.      }
7.
8.      // Count number elements in C[A[i]]
9.      for i := 1 to n do {
10.         C[A[i]] := C[A[i]] + 1;
11.     }
12.
```

```
13.     // C[i] is the accumulate number of elements
14.     for i := 1 to k do {
15.         C[i] := C[i] +C[i-1];
16.     }
17.
18.     // Store sorted order in array B
19.     for i := n to 1 step -1 do {
20.         B[C[A[i]]] := A[i];
21.         C[A[i]] := C[A[i]]-1;
22.     }
23. }
```

We use the counting sort as the stable sorting algorithm in the radix sort. First, we construct an array B with size N to store the sorting result, and an array C with size 10 to store the number of each element.

At the next step, we iterate through the array, and record the number of times each number appears in C. Then, we make the C array be the accumulation of the number of letters by adding the value from C[1] to C[i-1]. Thus, after the additions through the whole array, the content in the array C would be (not strictly) increasing.

At the final step, because we had known how many times each number appears in the array, the only thing we have to do is place the number to array B directly. For a number i, we would find C[i] to get the accumulation of index, and the index is where we should put the number in B. And then minus the accumulation by one, which indicates that the next number with the same digit would be place before the previous

one. We keep executing this step until the iteration goes to the end, and it also means every component in array A was arranged well.

Finally, we copy the content in array B to A and finish the sorting of the digit.

The time complexity is $O(n)$, and the space complexity is $O(n)$.

2.4.2. Heap Sort

```
1.  Algorithm Heapify(A, root, n)
2.  {
3.      t:=root; j:=root*2;
4.      while j<=n do{
5.          if(j < n && A[j] < A[j+1]){          // if rchild > lchild
6.              j:=j+1;                          // j is rchild
7.          }
8.          if(t > A[j]){                        // if root > children
9.              break;                           // done
10.         }
11.         else{
12.             A[j/2]:=A[j];                   // place child to parent
13.             j:=j*2;                         // j is child, keep finding
14.         }
15.     }
16.     A[j/2]:=t;                              // place the root node
17. }
18.
```

The heapify function replaces the root node to rearrange the heap to the form of maxheap. We found the place where the root node should be inserted from the top, and keep tracing down the children. When tracing to the node j, if the value of the

root node is larger than the children, we place the root node at node j to form a maxheap. Otherwise, we keep finding by assign j to the child node which is the larger until finding a right place for the root node.

The loop stops when j > n, and the j doubles itself after each iteration step. So, the loop would execute at most $\log n$ times. Thus, the time complexity of heapify once is $O(\log n)$.

```
1.  Algorithm HeapSort(A, n)
2.  {
3.      for i:=n/2 to 1 step -1 do{
4.          Heapify(A, i, n);           // heapify all the subtrees
                                           to be a max heap
5.      }
6.
7.      for i:=n to 2 step -1 do{       // repeat n-1 times
8.          t:=A[i];                    // swap the first and the last
9.          A[i]:=A[1];
10.         A[1]:=t;
11.         Heapify(A, 1, i-1);         // make A[1:i-1] be a max heap
12.     }
13. }
```

This heap sort algorithm rearranges the array in increasing order by using the heap sort. At first, we heapify the subtrees with the nodes which have at least one child to make the heap become a maxheap. At each iteration step, we swap the root node, whose value is the largest, with the last node of the array. Then we heapify the tree A[1:i-1], due to the last i nodes were already well-arranged, so the largest element

would be placed at the top of the heap, and could be swapped for the next iteration. After the iteration goes to the end, the whole array is well-sorted.

For the two loops of this algorithm, the iterations go almost through the array. Thus, the time complexities are $O(n * \log n)$, which had timed the complexities of heapify.

For the space complexity, expect the array being sorted, we only need a temporary t to during the sorting (in-place sorting), and the heapify function is also an in-place function. So, the space complexity must be $O(n)$ (the array length).

2.4.3. Merge Sort

```
1.  Algorithm mergeSort(A, low, high)       // stable merge sort
2.  {
3.      if low < high then {
4.          mid := (low + high) / 2;        // find mid
5.          mergeSort(A, low, mid);         // merge sort for first half part
6.          mergeSort(A, mid + 1, high);    // merge sort for second half part
7.          merge(A, low, mid, high);       // merge two parts
8.      }
9.  }
```

In the merge sort algorithm, we use the divide-and-conquer method to sort the array in increasing order. If there are still items in the array, the merge sort algorithm would keep dividing the array into two halves from the middle, and merge the two arrays in an increasing order. The recursion call would keep executing until there is

only one element in the array. Since we would obtain the array arranged in increasing order after the merge function, we could get the array also arranged in increasing order after every step of merge sort function. Thus, we whole array will be well arranged after the function call of mergeSort(A, 1, N), which means that the entire array is sorted.

```
1.  Algorithm merge(A, low, mid, high)      // merge two arrays
2.  {
3.      h := low; i := low; j := mid + 1;   // initialize
4.
5.      while h <= mid and j <= high do {   // store smaller one to B
6.          if A[h] <= A[j] then {          // A[h] is smaller
7.              B[i] := A[h];
8.              h := h + 1;
9.          }
10.         else {                          // A[j] is smaller
11.             B[i] := A[j];
12.             j := j + 1;
13.         }
14.         i := i + 1;
15.     }
16.
17.     if h > mid then {                   // second half remaining
18.         for k := j to high do {
19.             B[i] := A[k];
20.             i := i + 1;
21.         }
22.     }
23.     else {                              // first half remaining
24.         for k := h to mid do {
25.             B[i] := A[k];
26.             i := i + 1;
27.         }
28.
```

```
29.     for k := low to high do             // copy B to A
30.         A[k] := B[k];
31. }
```

In the merge function, we would merge two arrays together in increasing order. The two input arrays had been arranged well at first, so we could know that the smallest item is one of the first elements in the two arrays. We record the value at the top of a new array.

Then for the second smallest, we could pop the smallest element we had found, and the second smallest is one of the first elements in the two revised arrays. We could repeat this method until one of the arrays is empty, and it means that the elements in the remaining array are all larger than those we record. So, we could just put the remaining elements in the end of the new array. Finally, all elements were well arranged in one array in increasing order.

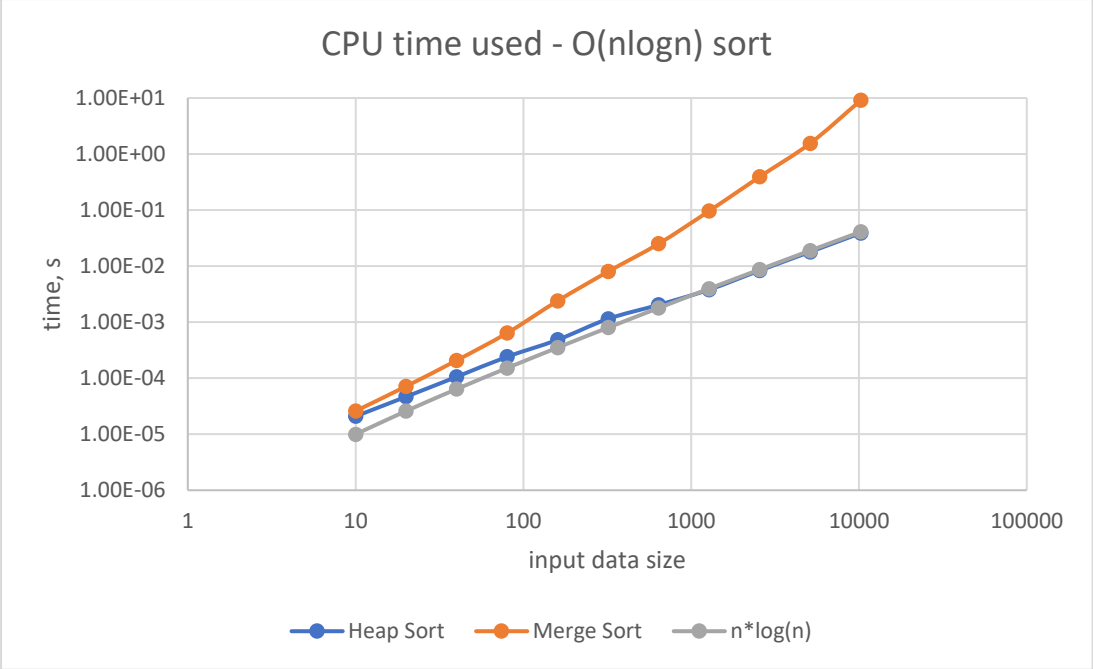
The time complexity of merge sort is $O(n * \log n)$, and the space complexity is $O(n)$.

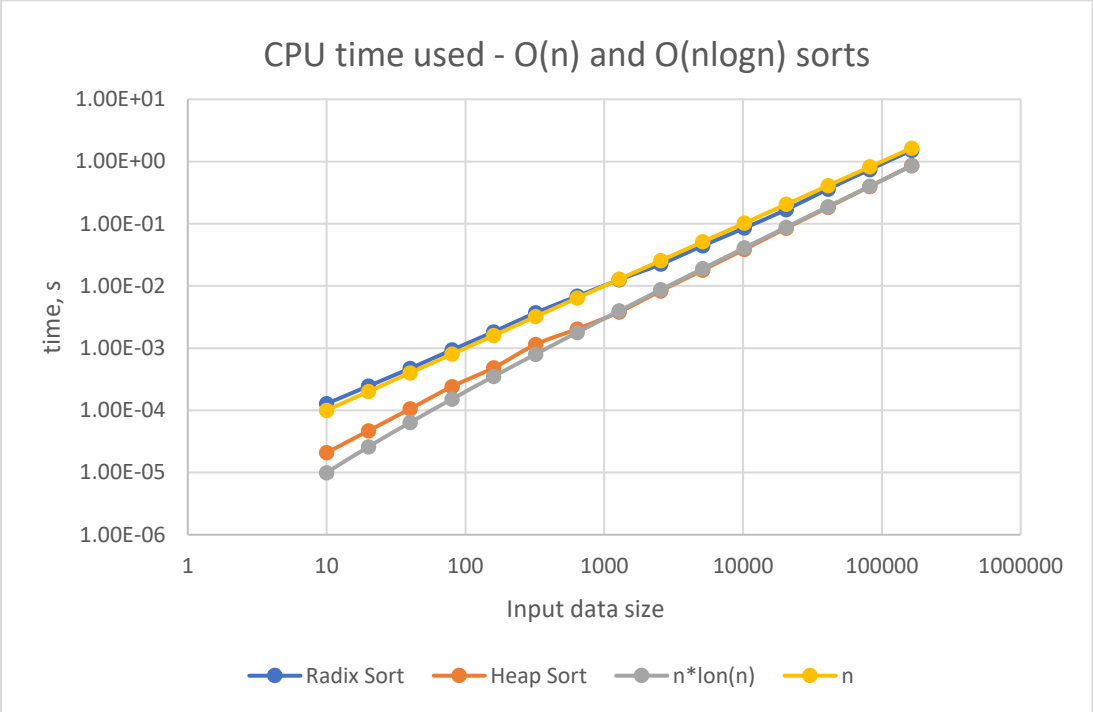
3. Executing results

We run the testing data from s1.dat to s15.dat with different input data size for 10 times, and record the average CPU time used.

Input data size	Radix Sort	Heap Sort	Merge Sort
10	127.8 μ s	20.89 μ s	25.92 μ s
20	245.3 μ s	46.71 μ s	71.60 μ s
40	472.4 μ s	105.7 μ s	207.5 μ s
80	936.2 μ s	240.8 μ s	639.4 μ s
160	1.831 ms	484.6 μ s	2.395 ms
320	3.694 ms	1.145 ms	7.986 ms
640	6.833 ms	2.037 ms	25.07 ms
1280	12.56 ms	3.804 ms	96.27 ms
2560	22.37 ms	8.348 ms	393.1 ms
5120	44.21 ms	17.96 ms	1.553 s
10240	85.25 ms	38.88 ms	9.071 s
20480	168.2 ms	84.71 ms	-
40960	360.6 ms	183.5 ms	-
81920	744.4 ms	397.1 ms	-
163840	1.512 s	864.4 ms	-

・Merge sort would take a lot of time for large input data size.





4. Result analysis and conclusion

From the first graph, we could observe that the heap sort has a trend of n*log(n), which is same as our estimation. However, although the merge sort works well for

small input data size, the CPU time used rises significantly after the data size grows above 100. This doesn't match the analysis we got before.

The reason why merge sort runs so slow might be the fact that the algorithm mostly bases on recursion function calls. Each step in the merge sort would invoke another two merge sort functions unless it reaches the terminal condition. For a large input data size, the number of function calls would rise rapidly, and the function call would consume much time than basic statement or iteration. Thus, the efficiency of the merge sort would be highly affected by the recursions. So, we would use the heap sort instead of merge sort as the O(n*logn) sort.

From the second graph, we could observe that both the time complexities of two sorting algorithms match our estimation, O(n) and O(n*logn). However, for all input data case, the radix sort is slower than the heap sort in spite of the face that it has the lower time complexity.

To explain the phenomenon, one reason might be that at each counting sort step in the radix sort, we have to construct two new arrays, and apply assignment through the whole array at least four times. These steps might make the radix sort slower than the heap sort in a small input data size.

Another reason might be that at each counting sort step in the radix sort, we move almost all elements in the array. As for the heap sort, we would only move the element in the tree-traversal path until finding the proper position to place the root node at each iteration step. Hence, the difference in number of moving the elements might cause the efficiency difference between the two sorting algorithms.

Yet, since the heap sort has a faster growing trend, it may become slower with regard to the radix sort if the input size keeps growing up.

In our data structure, we use an array of integers to store each number, and one integer to store a digit. However, since the digit could only have values from 0 to 9, it seems a waste of space to allocate an integer to store the digit. Thus, we could use other variable type which takes fewer spaces than integer, such as short integer. The method could reduce the real space used to some extent.