Better Sorts

110191019 YU-TING CHUNG

Introduction

In this homework, I implemented six different sorting algorithms to compare their efficiency and analyze their performance. The algorithms include Heap Sort, Merge Sort, an improved version of Merge Sort, Quick Sort, an improved version of Quick Sort, and Randomized Quick Sort. Specifically, the improved versions of Merge Sort and Quick Sort adopt a hybrid approach that incorporates insertion sort for small subarrays to enhance practical performance. Additionally, Randomized Quick Sort introduces randomness in pivot selection to avoid worst-case time complexity.

To analyze the performance of these algorithms, I tested them using the same string-based datasets provided in Homework 1. Each dataset consists of a specified number of strings to be sorted. I measured the CPU time taken by each algorithm to sort the datasets and compared the results with their theoretical time and space complexities. This helped evaluate the efficiency and practical performance differences among the sorting methods.

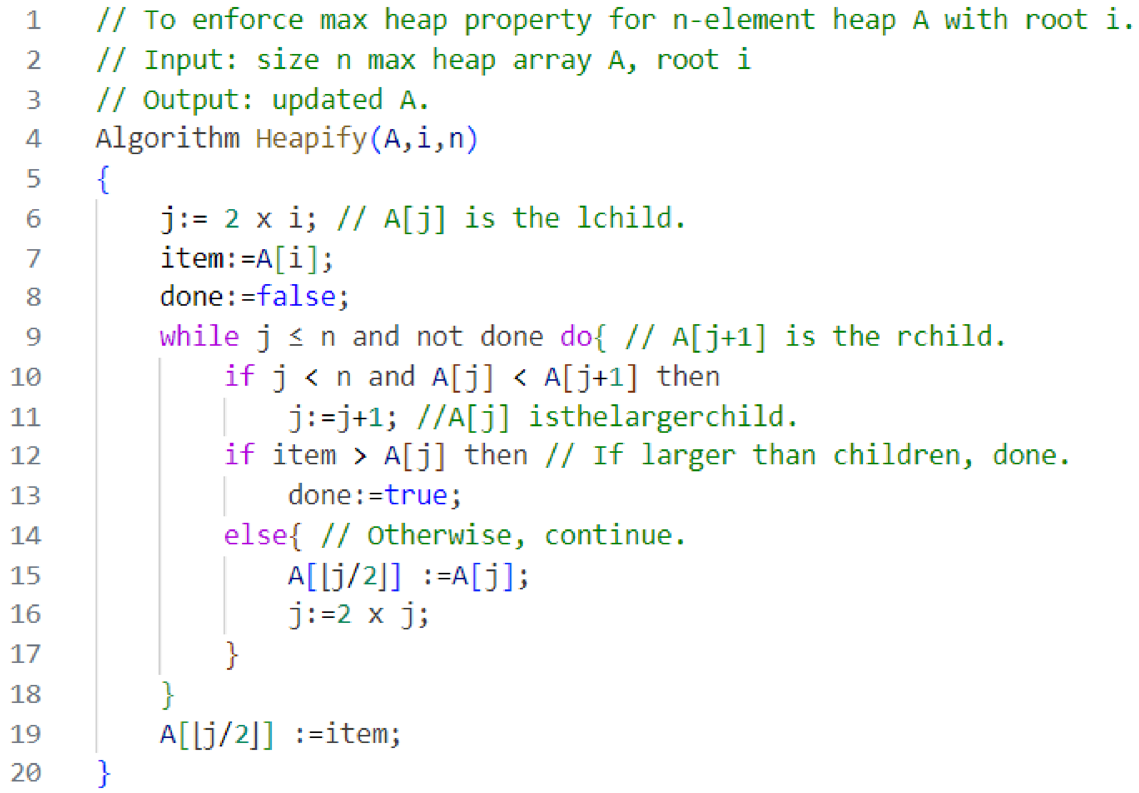
The Sorting Problem

The sorting problem involves arranging a list of elements in a specific order, typically ascending or descending. These elements are not limited to numbers; they can also be strings or other comparable data types.

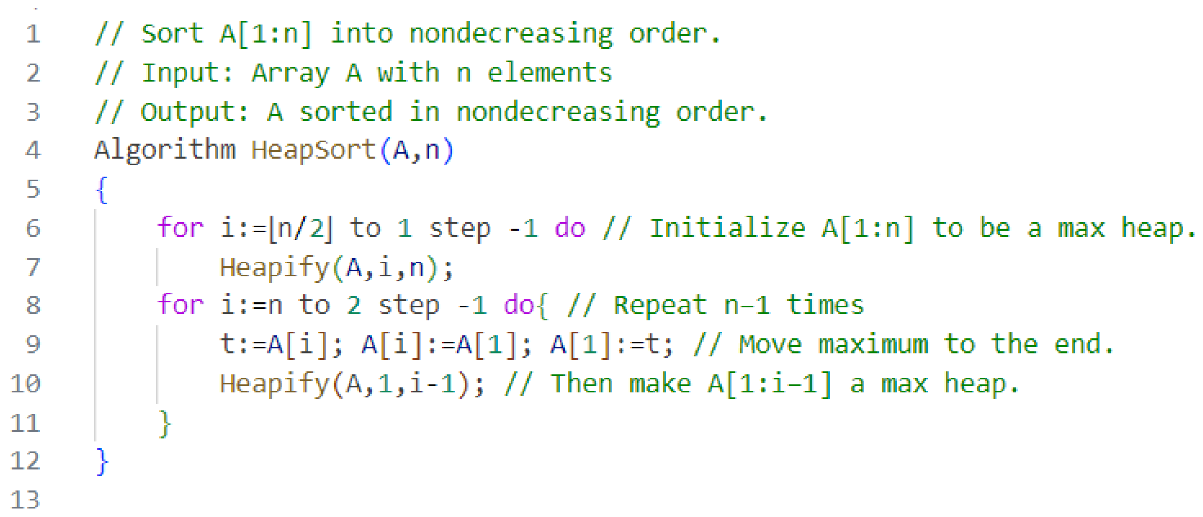
Approach

- Heap Sort

1. Algorithm:



```
// To enforce max heap property for n-element heap A with root i.
// Input: size n max heap array A, root i
// Output: updated A.
Algorithm Heapify(A,i,n)
{
    j:= 2 x i; // A[j] is the lchild.
    item:=A[i];
    done:=false;
    while j ≤ n and not done do{ // A[j+1] is the rchild.
        if j < n and A[j] < A[j+1] then
            j:=j+1; //A[j] isthelargerchild.
        if item > A[j] then // If larger than children, done.
            done:=true;
        else{ // Otherwise, continue.
            A[⌊j/2⌋] :=A[j];
            j:=2 x j;
        }
    }
    A[⌊j/2⌋] :=item;
}
```



```
// Sort A[1:n] into nondecreasing order.
// Input: Array A with n elements
// Output: A sorted in nondecreasing order.
Algorithm HeapSort(A,n)
{
    for i:=⌊n/2⌋ to 1 step -1 do // Initialize A[1:n] to be a max heap.
        Heapify(A,i,n);
    for i:=n to 2 step -1 do{ // Repeat n-1 times
        t:=A[i]; A[i]:=A[1]; A[1]:=t; // Move maximum to the end.
        Heapify(A,1,i-1); // Then make A[1:i-1] a max heap.
    }
}
```

The algorithm operates by first transforming the input array into a max heap to ensure that the largest element is always at the root. Then, the largest element is swapped with the last element and removed from the heap, while the remaining elements are restructured to maintain the heap property. This process is repeated until all elements are sorted.

2. Proof of correctness:

At the start of the algorithm, the input array is first transformed into a max heap. This ensures that the largest element is always at the root of the heap. In each step of the sorting process, the root element which is the largest is swapped with the last unsorted element in the array, and the heap size is reduced by one. The heap property is then restored by performing the Heapify operation on the new root.

Since the Heapify function ensures that the heap property is maintained at every step, and since each extraction step correctly places the next largest element in its correctly sorted position, the array is fully sorted when the heap size is reduced to one. Given that every element is correctly placed in each iteration, heap sort always produces the correct sorted order. Hence, proved.

3. Time and space complexities:

Time complexity:

Heap Sort consists of two main steps:

- Building the heap: O (N log N)

  Heap Sort uses the top-down approach to build the heap by inserting each element. Each insertion takes O (log N) time, and with n elements, the total time complexity is O (N log *N)*.

- Heapify and extraction: O (N log *N)*

  After extracting the root, the heap needs to be adjusted using heapify, which takes O (log N) time. The following table explains why each data entry can move up to ($\log_2$ N) levels.

Number of layers	Number of nodes/ Number of data points	Number of layers
1	$2^0 = 1$	$\log_2 2$
2	$2^0+2^1 = 3$	$\log_2 4$
3	$2^0+2^1+2^2 = 7$	$\log_2 8$
…	…	…
k	$2^0+2^1+2^2+\ldots+2^{k-1}=2^k-1$	$\log_2 2^k$
	n	$\log_2 n$

Since this process is repeated n times, the total time complexity is

O (n log *n*).

Overall Time Complexity: O (n log *n*) + O (n log *n*) = O (n log *n*)

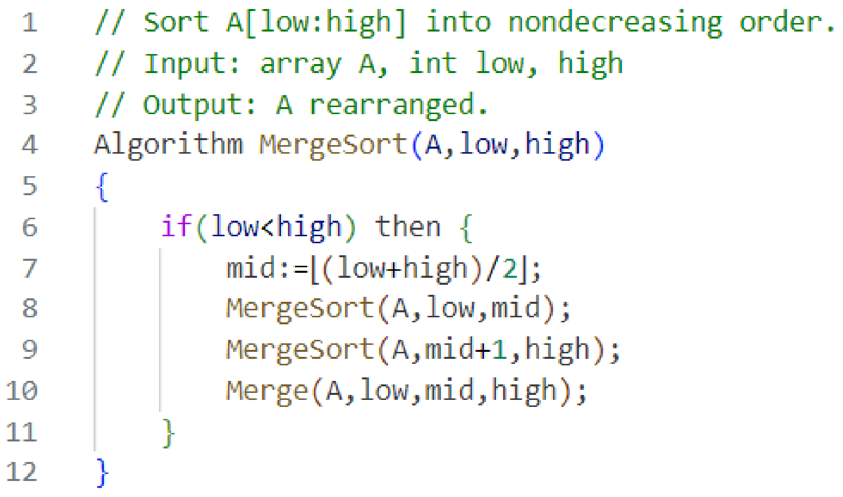
Space complexity: O (n)

5 integers (i, j, done, n, item), and the space that we need to store the array

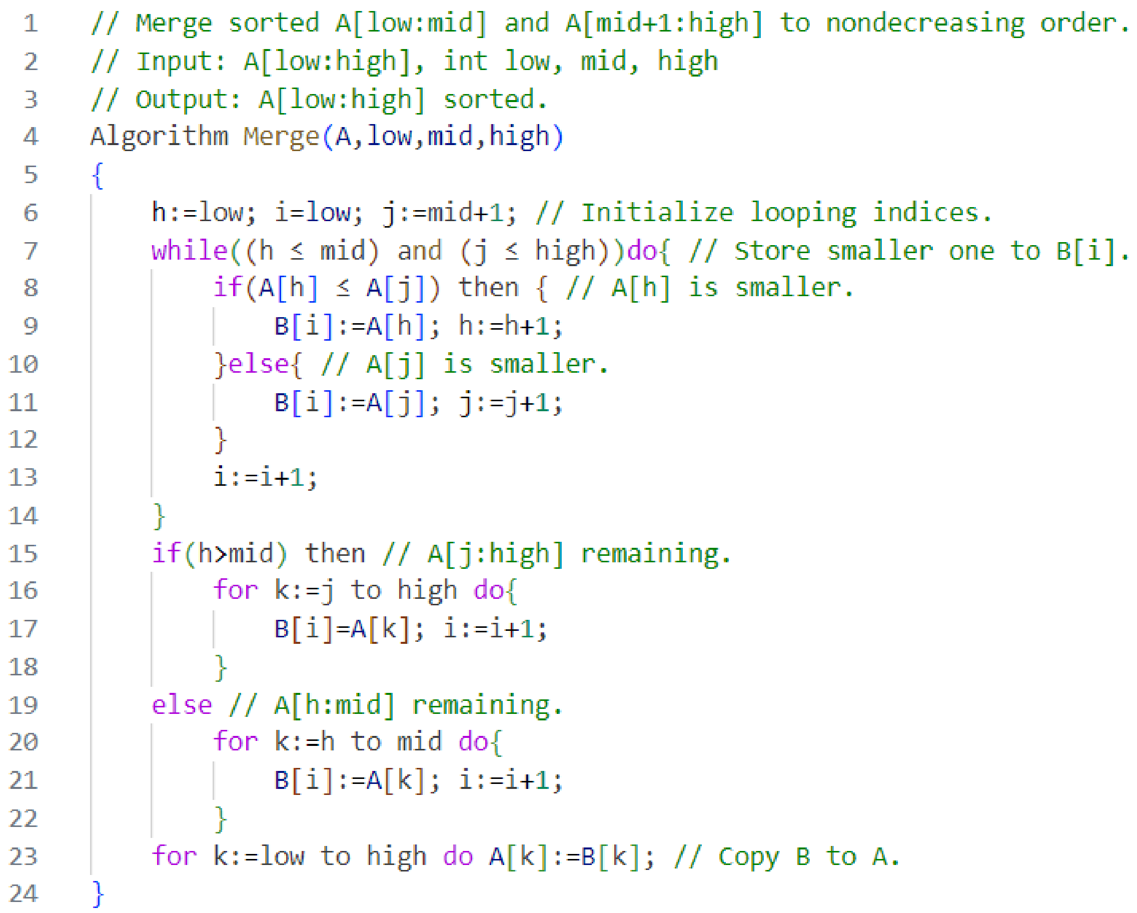
is n, so the space complexity is O(n).

- Merge Sort

1. Algorithm:



```
// Sort A[low:high] into nondecreasing order.
// Input: array A, int low, high
// Output: A rearranged.
Algorithm MergeSort(A,low,high)
{
    if(low<high) then {
        mid:=⌊(low+high)/2⌋;
        MergeSort(A,low,mid);
        MergeSort(A,mid+1,high);
        Merge(A,low,mid,high);
    }
}
```



```
// Merge sorted A[low:mid] and A[mid+1:high] to nondecreasing order.
// Input: A[low:high], int low, mid, high
// Output: A[low:high] sorted.
Algorithm Merge(A,low,mid,high)
{
    h:=low; i=low; j:=mid+1; // Initialize looping indices.
    while((h ≤ mid) and (j ≤ high))do{ // Store smaller one to B[i].
        if(A[h] ≤ A[j]) then { // A[h] is smaller.
            B[i]:=A[h]; h:=h+1;
        }else{ // A[j] is smaller.
            B[i]:=A[j]; j:=j+1;
        }
        i:=i+1;
    }
    if(h>mid) then // A[j:high] remaining.
        for k:=j to high do{
            B[i]=A[k]; i:=i+1;
        }
    else // A[h:mid] remaining.
        for k:=h to mid do{
            B[i]:=A[k]; i:=i+1;
        }
    for k:=low to high do A[k]:=B[k]; // Copy B to A.
}
```

Merge Sort is a sorting algorithm based on the divide and conquer strategy. The array is split into two halves. This process continues recursively until each part contains only one element, as a single element is inherently sorted. Each of the divided subarrays is sorted. Since each subarray reaches a size of one, the merging begins after that. Two sorted subarrays are combined into a single sorted array. This is done by comparing the smallest elements of each subarray and repeatedly choosing the smallest to place in the new array.

2. Proof of correctness:

When the array length is 0 or 1, it is already sorted. Merge Sort directly returns the array, thus correct.

Assume that Merge Sort can correctly sort all arrays of size ≤ k. Let A be an array of size k+1. The algorithm splits A into two subarrays, L and R, where the lengths of L and R are both ≤ k. The algorithm recursively sorts L and R, and by the inductive hypothesis, they are sorted correctly.

3. Time and space complexities:

Time complexity: O (n lg*n*)

Let T (n) be the computing time of merge sort applying to a data set of n elements, then when n=1, T(n)= a and a is a constant. When n>1,

2T(n/2) +c·n, and c is a constant. If $n=2^k$, then

$$T(n)=2\ (2T(n/4))\ +c\cdot n/2)\ +c\cdot n$$

$$=4T(n/4)\ +2c\cdot n$$

$$=4\ (2T(n/8)\ +c\cdot n/4)\ +2c\cdot n$$

$$=2^kT\ (1)\ +k\cdot c\cdot n$$

$$=a\cdot n + c\cdot n\cdot \lg n$$

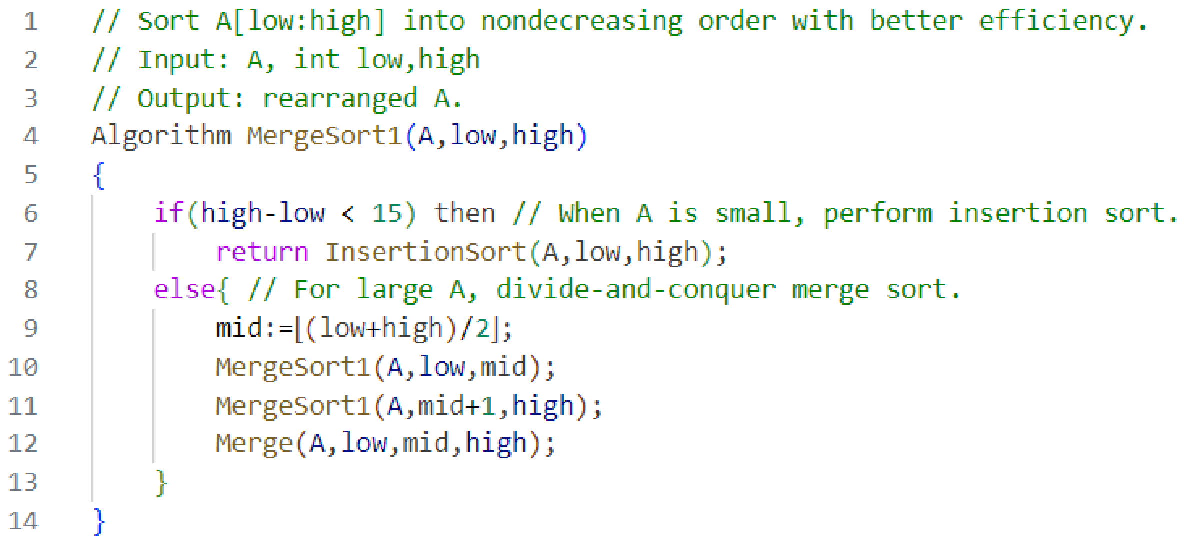
If $2^k < n \le 2^{k+1}$, then $T(n) \le T(2^{k+1})$. Therefore, $T(n)=O(n \lg n)$.

Space complexity: O(n)

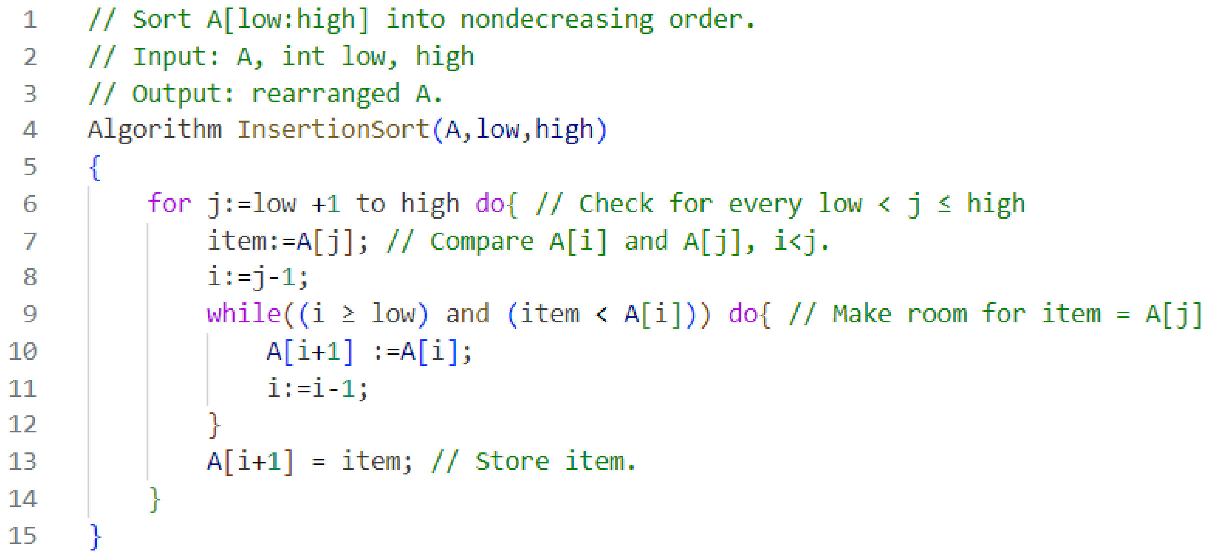
9 integers (A, B, h, i, j, k, low, high, mid), and the space that we need to store the array is n, so the space complexity is O(n).

- Merge Sort 1

1. Algorithm:



```
// Sort A[low:high] into nondecreasing order with better efficiency.
// Input: A, int low,high
// Output: rearranged A.
Algorithm MergeSort1(A,low,high)
{
    if(high-low < 15) then // When A is small, perform insertion sort.
        return InsertionSort(A,low,high);
    else{ // For large A, divide-and-conquer merge sort.
        mid:=⌊(low+high)/2⌋;
        MergeSort1(A,low,mid);
        MergeSort1(A,mid+1,high);
        Merge(A,low,mid,high);
    }
}
```



```
// Sort A[low:high] into nondecreasing order.
// Input: A, int low, high
// Output: rearranged A.
Algorithm InsertionSort(A,low,high)
{
    for j:=low +1 to high do{ // Check for every low < j ≤ high
        item:=A[j]; // Compare A[i] and A[j], i<j.
        i:=j-1;
        while((i ≥ low) and (item < A[i])) do{ // Make room for item = A[j]
            A[i+1] :=A[i];
            i:=i-1;
        }
        A[i+1] = item; // Store item.
    }
}
```

Merge Sort 1 is an enhanced version of Merge Sort. When the number of data is less than 15, Insertion Sort will be used. When the number of data is larger than 15, Merge Sort will be used.

Insertion Sort is an algorithm that divides the array into a sorted region and an unsorted region. It extracts elements from the unsorted region and inserts them into their correct position within the sorted region.

2. Proof of correctness:

Insertion Sort

When the array length is 0 or 1, it is already sorted. Insertion Sort directly returns the array, thus correct.

Let A be an array of size k+1. The (k+1)-th element is compared sequentially to the elements in the sorted portion. Each comparison ensures the relative order remains intact. Once the (k+1)-th element is placed correctly, the entire array is sorted.

Merge Sort

When the array length is 0 or 1, it is already sorted. Merge Sort directly returns the array, thus correct.

Assume that Merge Sort can correctly sort all arrays of size ≤ k. Let A be an array of size k+1. The algorithm splits A into two subarrays, L and R, where the lengths of L and R are both ≤ k. The algorithm recursively sorts L and R, and by the inductive hypothesis, they are sorted correctly.

Since both insertion and merge sort are correct, the merge sort 1 is correct.

3. Time and space complexities:

Time complexity:

Insertion Sort

$T(n)= [n(n+1) /2] -1=O(n^2)$.

The worst-case time complexity of insertion sort is $O (n^2)$

The best case of time complexity of insertion sort is $\Theta(n)$

Since MergeSort1 only uses Insertion Sort when $n < 15$, it will be executed at most at the bottom level of the recursion tree.

The cost of Insertion Sort becomes O (1) level (handling at most 14 elements in constant time with a cost of $O(14^2)$).

In total, there will be $O(n/15)$ small subarrays that utilize Insertion Sort.

Therefore, the time complexity of insertion sort in Merge Sort 1 is

$O\ ((n/15) \cdot 14^2) = O(n)$

Merge Sort

In the recursion splitting phase, the algorithm divides the array. Due to the recursive layers created, time complexity is O (log n).

The algorithm combines the subarrays for merging all elements across a single level, so the time complexity is O(n).

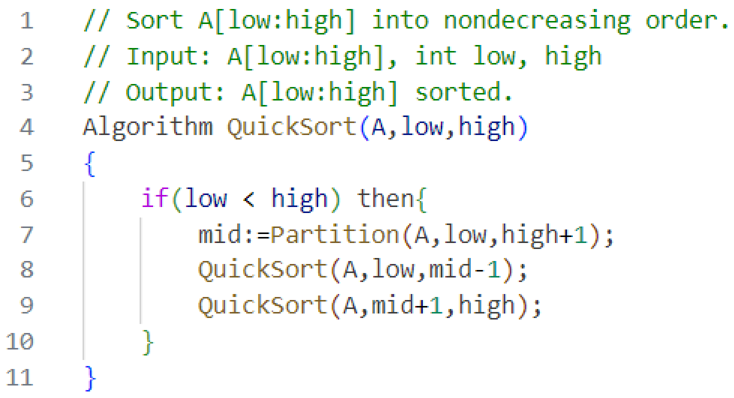
When combined, the total time complexity for MergeSort1 remains O (n log *n*), which is the same as the original Merge Sort.

Space complexity:

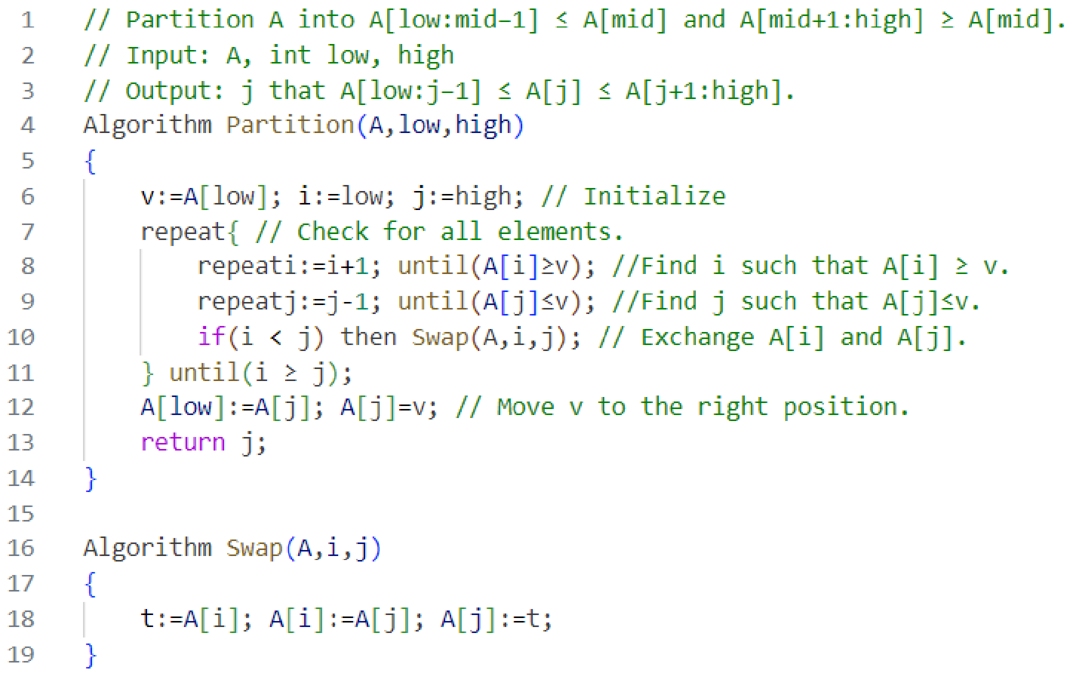
9 integers (A, B, h, i, j, k, low, high, mid), and the space that we need to store the array is n, so the space complexity is O(n).

- Quick Sort

1. Algorithm:



```
// Sort A[low:high] into nondecreasing order.
// Input: A[low:high], int low, high
// Output: A[low:high] sorted.
Algorithm QuickSort(A,low,high)
{
    if(low < high) then{
        mid:=Partition(A,low,high+1);
        QuickSort(A,low,mid-1);
        QuickSort(A,mid+1,high);
    }
}
```



```
// Partition A into A[low:mid-1] ≤ A[mid] and A[mid+1:high] ≥ A[mid].
// Input: A, int low, high
// Output: j that A[low:j-1] ≤ A[j] ≤ A[j+1:high].
Algorithm Partition(A,low,high)
{
    v:=A[low]; i:=low; j:=high; // Initialize
    repeat{ // Check for all elements.
        repeati:=i+1; until(A[i]≥v); //Find i such that A[i] ≥ v.
        repeatj:=j-1; until(A[j]≤v); //Find j such that A[j]≤v.
        if(i < j) then Swap(A,i,j); // Exchange A[i] and A[j].
    } until(i ≥ j);
    A[low]:=A[j]; A[j]=v; // Move v to the right position.
    return j;
}

Algorithm Swap(A,i,j)
{
    t:=A[i]; A[i]:=A[j]; A[j]:=t;
}
```

Quick Sort is a sorting algorithm that uses the divide and conquer strategy. It begins by selecting a pivot element from the array. The array is then partitioned so that all elements smaller than the pivot are moved to its left, and all elements greater than or equal to the pivot are moved to its right. After this, Quick Sort is recursively applied to the left and right subarrays to ensure the entire array is sorted.

2. Proof of correctness:

If the array has 0 or 1 element, then it is already sorted, and Quick Sort correctly returns.

Assume that Quick Sort correctly sorts any array of size ≤ *k*. For an array of size (k+1), Quick Sort first selects a pivot element and partitions the array into two subarrays. The left subarray contains elements smaller than the pivot and the right subarray contains elements greater than or equal to the pivot. The pivot is placed in its correctly sorted position. Quick Sort recursively sorts both the left and right subarrays correctly. Finally, the algorithm combines the two subarrays with the pivot results in a fully sorted array. Thus, we can know Quick Sort correctly sorts arrays of any finite size.

3. Time and space complexities:

Time complexity:

Average case

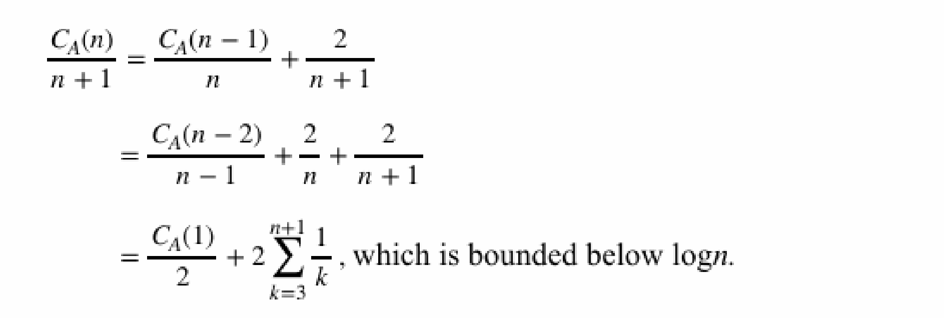
A general equation is:

$$C_A(n) = n + 1 + \frac{1}{n}\sum_{k=1}^{n}(C_A(k-1) + C_A(n-k))$$

where C_A(n) is the comparisons at top layer, and C_A(k-1), C_A(n-k) are comparisons from the left half and right half. As we know C_A (0) =C_A (1) =0 and expand the sigma, we can get

nC_A (n) = n(n−1) + 2 (C_A(1) + C_A (2) +⋯+ C_A (n)).

We replace n with n - 1 and subtract the two equations and we can get:



$$\frac{C_A(n)}{n+1} = \frac{C_A(n-1)}{n} + \frac{2}{n+1}$$

$$= \frac{C_A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$= \frac{C_A(1)}{2} + 2\sum_{k=3}^{n+1}\frac{1}{k}\text{, which is bounded below } \log n.$$

So, we can get C_A (n) ≤ 2(n−1) log(n−2) =O (n $\log n$) =O (n $\log n$)

Worst-case

At the highest level, the Partition function is invoked with (n+1) comparisons. At the next level, the worst-case scenario involves one of the partitions with (n-1) elements and (n) comparisons.

Therefore, the time complexity is O(n^2)

Space complexity:

Best case

In an ideal scenario, where each pivot divides the array into two equal halves, the recursion depth is proportional to log n. Thus, the space complexity is O (log n).

Average case

The recursion depth remains around log n, so the space complexity is also

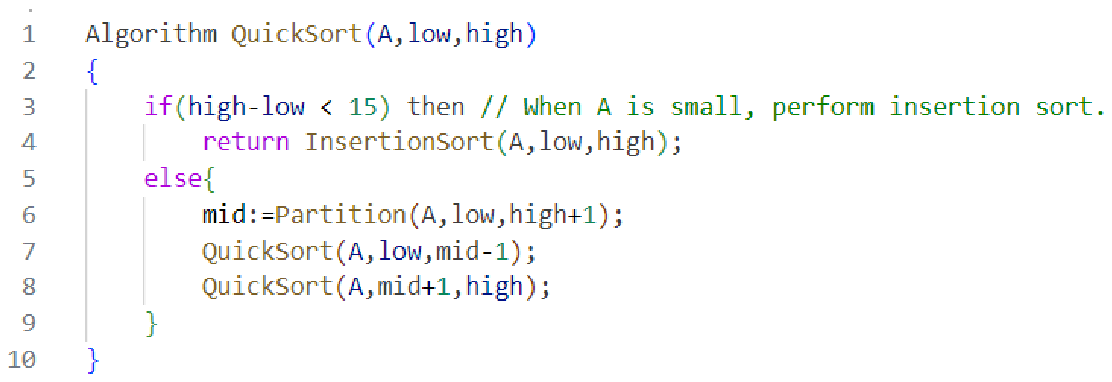
O (log n).

Worst-case

If the pivot selects the smallest or largest element, the depth of recursion equals the size of the array. It results in O(n) space complexity due to the stack frames created for every recursive call.

- Quick Sort 1

1. Algorithm:



```
Algorithm QuickSort(A,low,high)
{
    if(high-low < 15) then // When A is small, perform insertion sort.
        return InsertionSort(A,low,high);
    else{
        mid:=Partition(A,low,high+1);
        QuickSort(A,low,mid-1);
        QuickSort(A,mid+1,high);
    }
}
```

Quick Sort 1 is an enhanced version of Quick Sort. When the number of data is less than 15, Insertion Sort will be used. When the number of data is larger than 15, Merge Sort will be used.

2. Proof of correctness:

In Merge Sort 1, we have proven the correctness of Insertion Sort, and in the analysis of Quick Sort, we have also demonstrated the correctness of Quick Sort. Since both algorithms are proven to be correct, we can conclude that

Quicksort 1 is also correct.

3. Time and space complexities:

Time complexity:

Insertion sort

From the analysis of Merge sort 1 above, we can know the best case of time complexity of Insertion sort is O(n). The average time complexity is O(n^2), and the worst-case time complexity is O(n^2).

Quick sort

From the analysis of Quick sort above, we can know the best case of time complexity of Quick sort is O (n $\log n$). The average time complexity is

O (n $\log n$), and the worst-case time complexity is O(n^2).

Overall Time Complexity:

Best Case: O (n $\log n$)

Average Case: O (n $\log n$)

Worst Case: O(n^2)

Space Complexity

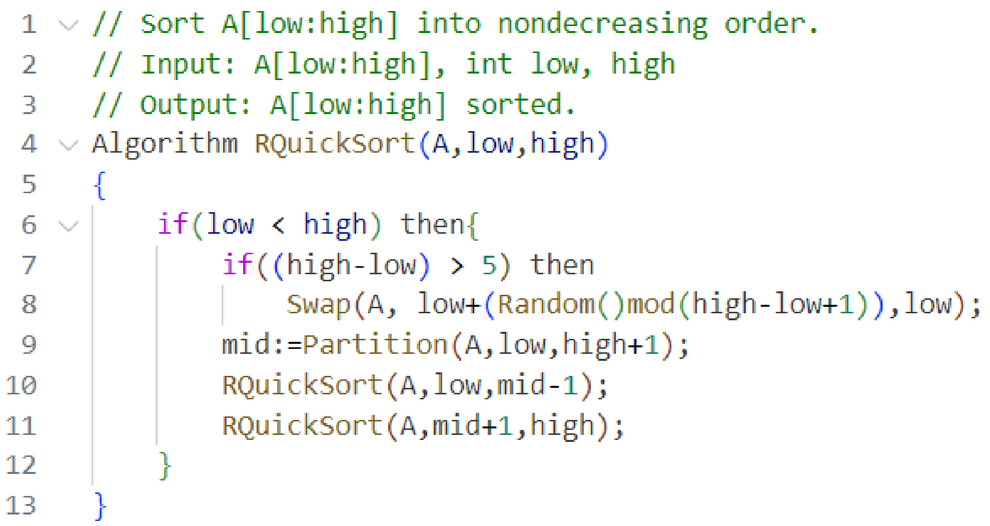
Best Case: O (log n)

Average Case: O (log n)

Worst Case: O(n)

- Randomized Quick Sort

1. Algorithm:



```
// Sort A[low:high] into nondecreasing order.
// Input: A[low:high], int low, high
// Output: A[low:high] sorted.
Algorithm RQuickSort(A,low,high)
{
    if(low < high) then{
        if((high-low) > 5) then
            Swap(A, low+(Random()mod(high-low+1)),low);
        mid:=Partition(A,low,high+1);
        RQuickSort(A,low,mid-1);
        RQuickSort(A,mid+1,high);
    }
}
```

Randomized Quick Sort is a variation of the standard Quick Sort algorithm that improves performance by introducing randomness in the selection of the pivot element. Instead of always choosing a fixed position (e.g., the first or last element) as the pivot, it randomly selects an element within the current subarray and swaps it to the front before partitioning.

2. Proof of correctness:

If size > 5, it picks a random index r ∈ [low, high] and swaps A[low] with A[r].

Then it calls Partition(A, low, high + 1). All elements < pivot are on the left, and all elements ≥ pivot are on the right. Pivot is placed in its correct final position. Let mid be the index of the pivot, and then it **recursively** sorts left subarray (A[low...mid-1]) and right subarray (A[mid+1..high])

Since the pivot is in its correctly sorted position, left and right parts are also correctly sorted. All elements in A[low...mid-1] ≤ A[mid] ≤ A[mid+1..high], so the entire array A[low..high] is sorted after the recursive calls.

3. Time and space complexities:

Time complexity:

The Partition function takes O(n) time in each recursive call because it scans the entire subarray once to partition it.

Best case

The pivot always splits the array into two equal halves, so recursion depth is $\log_2 n$ and partitioning at each level takes O(n).

The time complexity is $T(n)=2T(n/2) + O(n) \Rightarrow O(n \log n)$

Average case

With random pivot selection, the partitions are fairly balanced on average.

Therefore, the time complexity is $T(n)=2T(n/2) +O(n)\Rightarrow O(n \log n)$

Worst case

If the pivot always ends up as the smallest or largest element, such as one side has n−1 elements, and the other has 0, and recursion depth becomes n. Therefore, the time complexity is $T(n)=T(n-1) +O(n)\Rightarrow O(n^2)$

Space Complexity

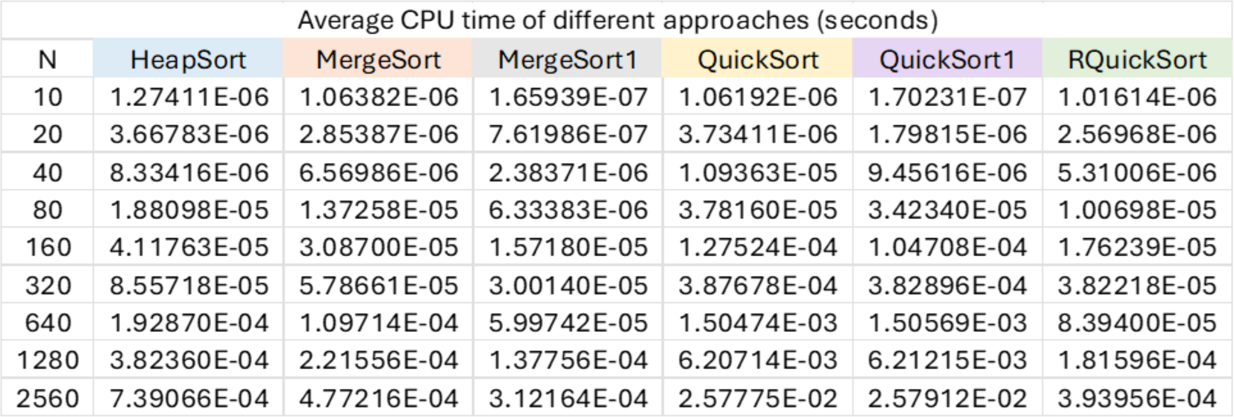
Best Case: $O(\log n)$

Average Case: $O(\log n)$

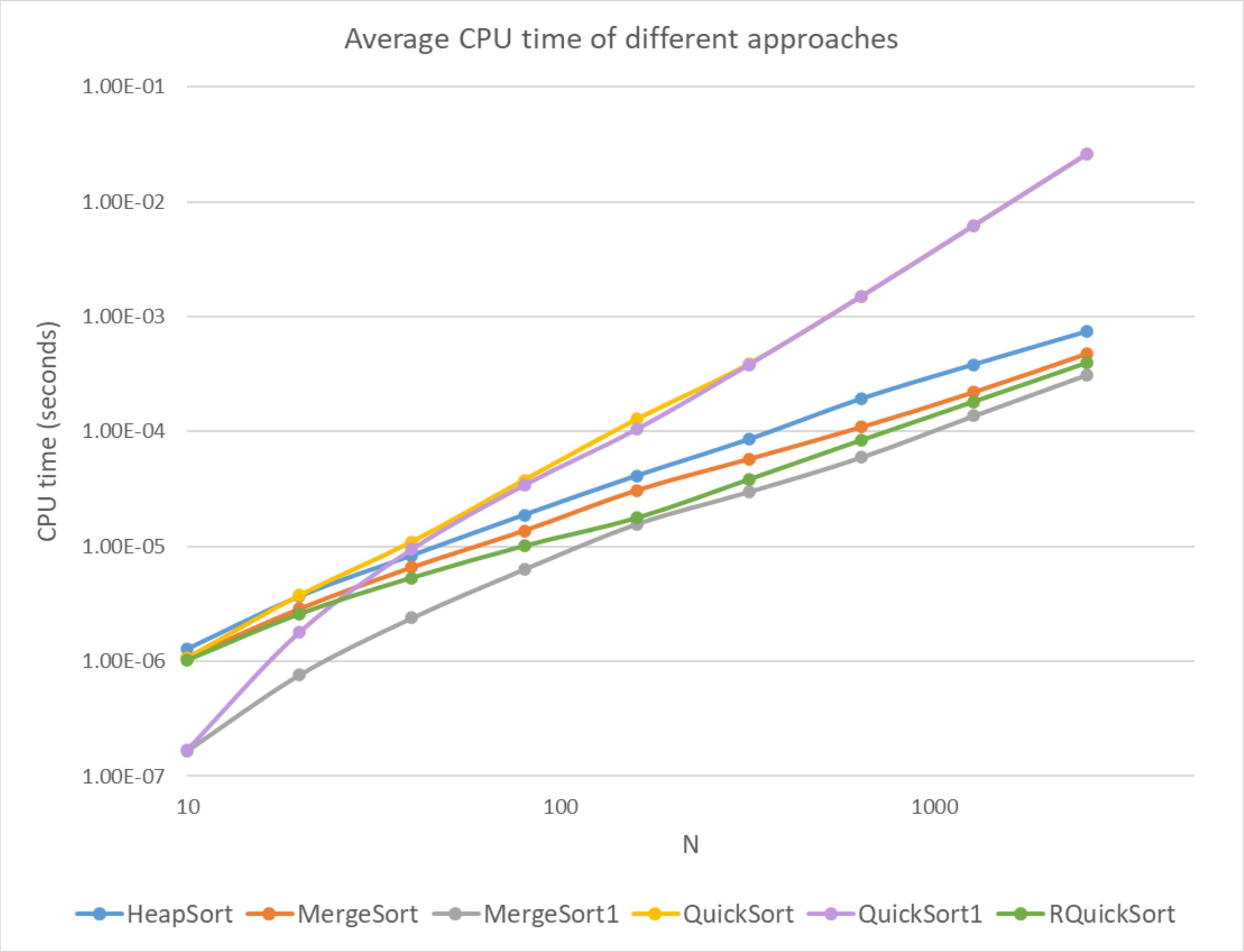
Worst Case: $O(n)$

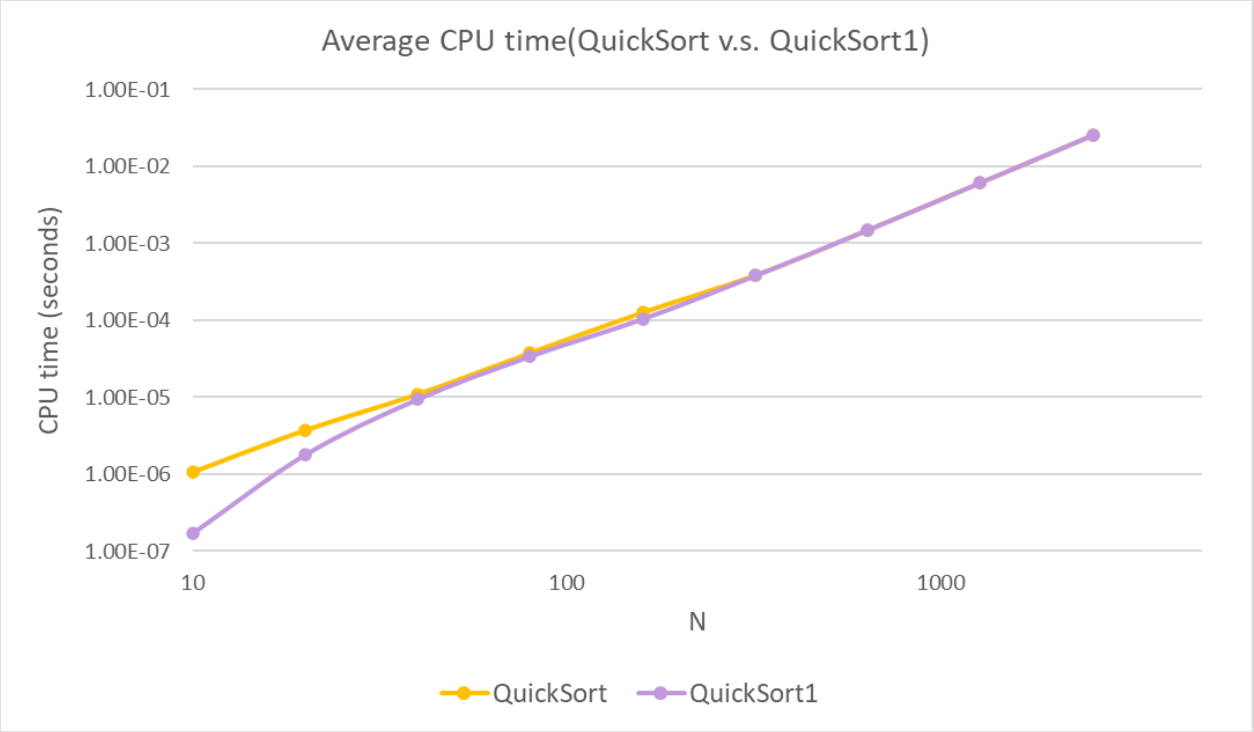
Results and observations

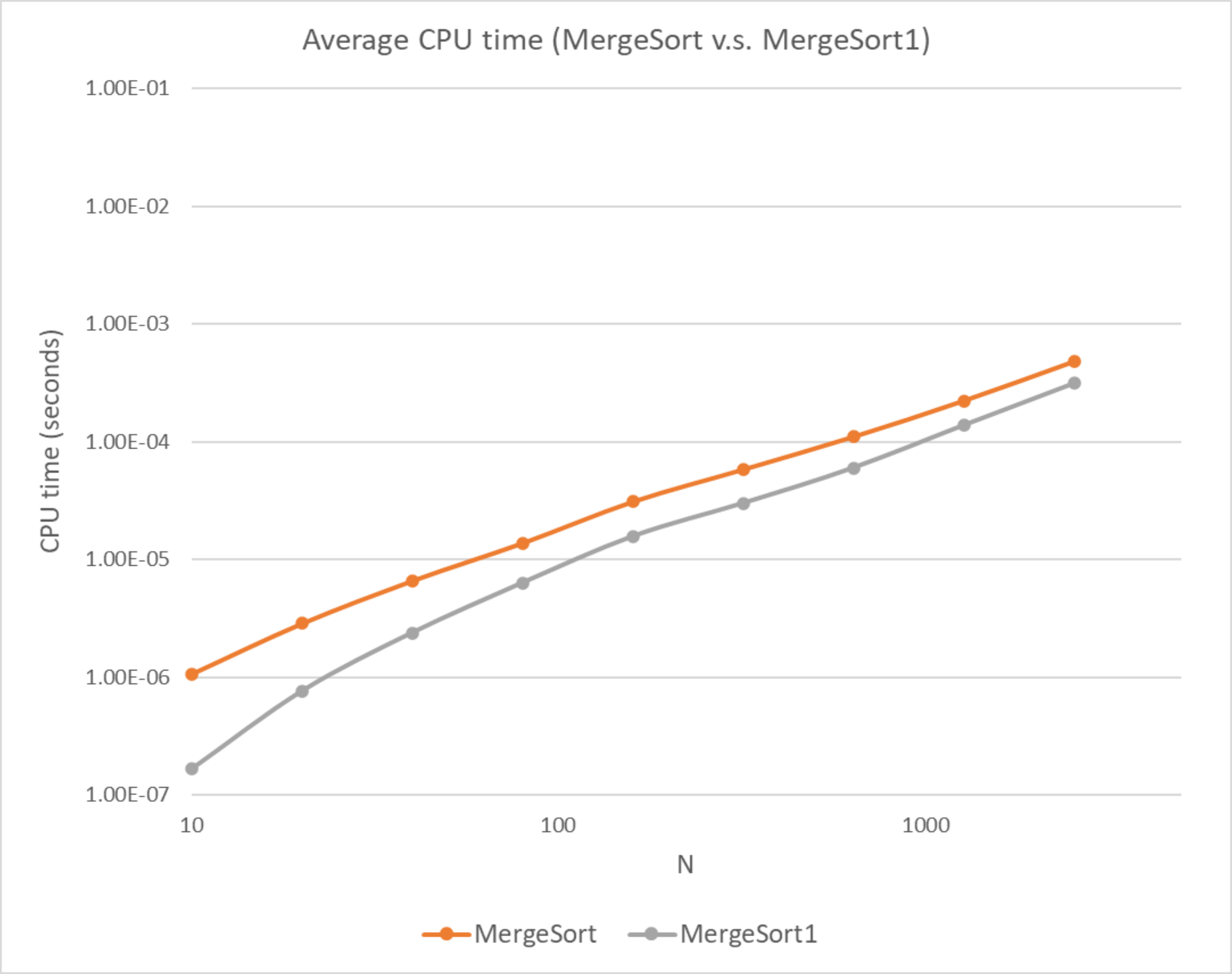
1. Results



Average CPU time of different approaches (seconds)						
N	HeapSort	MergeSort	MergeSort1	QuickSort	QuickSort1	RQuickSort
10	1.27411E-06	1.06382E-06	1.65939E-07	1.06192E-06	1.70231E-07	1.01614E-06
20	3.66783E-06	2.85387E-06	7.61986E-07	3.73411E-06	1.79815E-06	2.56968E-06
40	8.33416E-06	6.56986E-06	2.38371E-06	1.09363E-05	9.45616E-06	5.31006E-06
80	1.88098E-05	1.37258E-05	6.33383E-06	3.78160E-05	3.42340E-05	1.00698E-05
160	4.11763E-05	3.08700E-05	1.57180E-05	1.27524E-04	1.04708E-04	1.76239E-05
320	8.55718E-05	5.78661E-05	3.00140E-05	3.87678E-04	3.82896E-04	3.82218E-05
640	1.92870E-04	1.09714E-04	5.99742E-05	1.50474E-03	1.50569E-03	8.39400E-05
1280	3.82360E-04	2.21556E-04	1.37756E-04	6.20714E-03	6.21215E-03	1.81596E-04
2560	7.39066E-04	4.77216E-04	3.12164E-04	2.57775E-02	2.57912E-02	3.93956E-04


Average CPU time of different approaches
1.00E-01
1.00E-02
1.00E-03
1.00E-04
1.00E-05
1.00E-06
1.00E-07
CPU time (seconds)
10
100
1000
N
HeapSort
MergeSort
MergeSort1
QuickSort
QuickSort1
RQuickSort


Average CPU time(QuickSort v.s. QuickSort1)
1.00E-01
1.00E-02
1.00E-03
1.00E-04
1.00E-05
1.00E-06
1.00E-07
CPU time (seconds)
10
100
1000
N
QuickSort
QuickSort1



2. Observation

From the graph, we can see that the merger sort 1 is the fastest because it is an optimized sorting algorithm that improves efficiency by switching to Insertion Sort when the size of the subarray is smaller than 15. Using Insertion Sort for smaller datasets not only reduces recursion depth but also minimizes memory overhead.

We can also observe that although Quick sort 1 is also switching to Insertion Sort when the size of the subarray is smaller than 15, it is the slowest. The inefficiency of the partition function (such as excessive comparisons) could result in increased computational overhead. Additionally, excessive recursion and data movement might lead to higher actual CPU time consumption.

In addition, we can see that although both are optimized using Insertion Sort, Quick sort1 ultimately perform similarly to the original Quick sort, while Merge sort1 consistently outperforms the original Merge sort.

It is because the recursion depth of Quick sort is highly dependent on the distribution of the data. If the pivot is chosen poorly, it may result in skewed partitions that degrade performance. Even with Insertion Sort handling subarrays smaller than 15, the overall speed is still largely determined by the structure of the recursive calls.

The merge operation (copying to array B and then copying back) imposes a relatively large overhead for small datasets. Using Insertion Sort in such cases can completely avoid the two copying steps to save time. Even for large N, the merging step always involves N copies. The use of Insertion Sort reduces this burden for small subarrays, and the cumulative effect becomes substantial.

Conclusion

1. Time and space complexities comparison:

Time complexity:

	HeapSort	MergeSort	MergeSort1	QuickSort	QuickSort1	RQuickSort
Best	O (n lg *n*)	O (n lg *n*)	O (n lg *n*)	O (n lg *n*)	O (n lg *n*)	O (n lg *n*)
Average	O (n lg *n*)	O (n lg *n*)	O (n lg *n*)	O (n lg *n*)	O (n lg *n*)	O (n lg *n*)
Worst	O (n lg *n*)	O (n lg *n*)	O (n lg *n*)	O(n^2)	O(n^2)	O(n^2)

Space complexity:

	HeapSort	MergeSort	MergeSort1	QuickSort	QuickSort1	RQuickSort
Best	O (n)	O (n)	O (n)	O (lg *n*)	O (lg *n*)	O (lg *n*)
Average	O (n)	O (n)	O (n)	O (lg *n*)	O (lg *n*)	O (lg *n*)
Worst	O (n)	O (n)	O (n)	O(n)	O(n)	O(n)

2. QuickSort1's performance is hindered by the inefficiency of the partition function, excessive comparisons, and its dependency on data distribution, so there is minimal improvement over the original Quicksort.

3. MergeSort1 consistently outperforms the original Merge Sort because it minimizes the overhead of merging operations, while QuickSort1's overall performance remains dictated by its recursive structure.