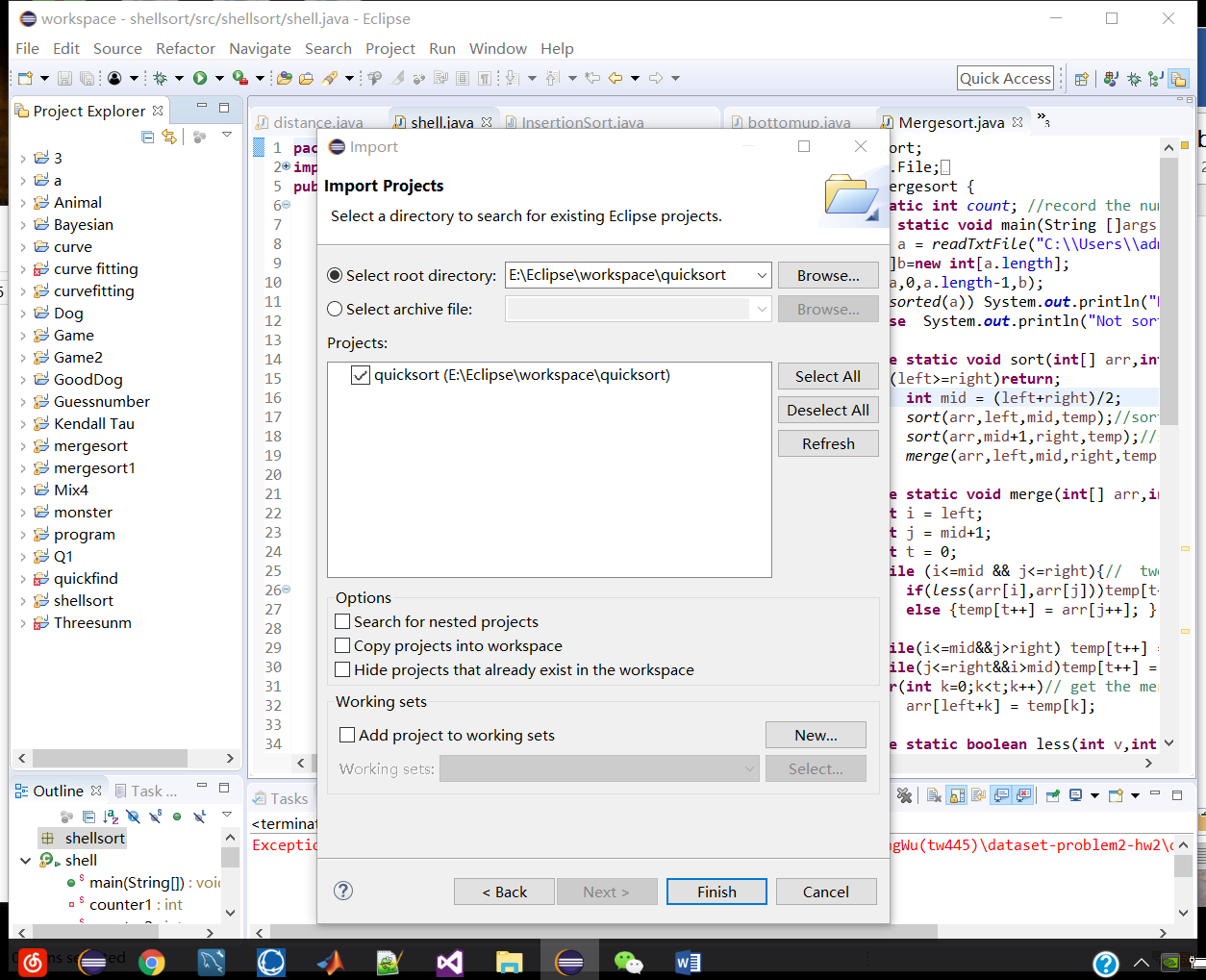
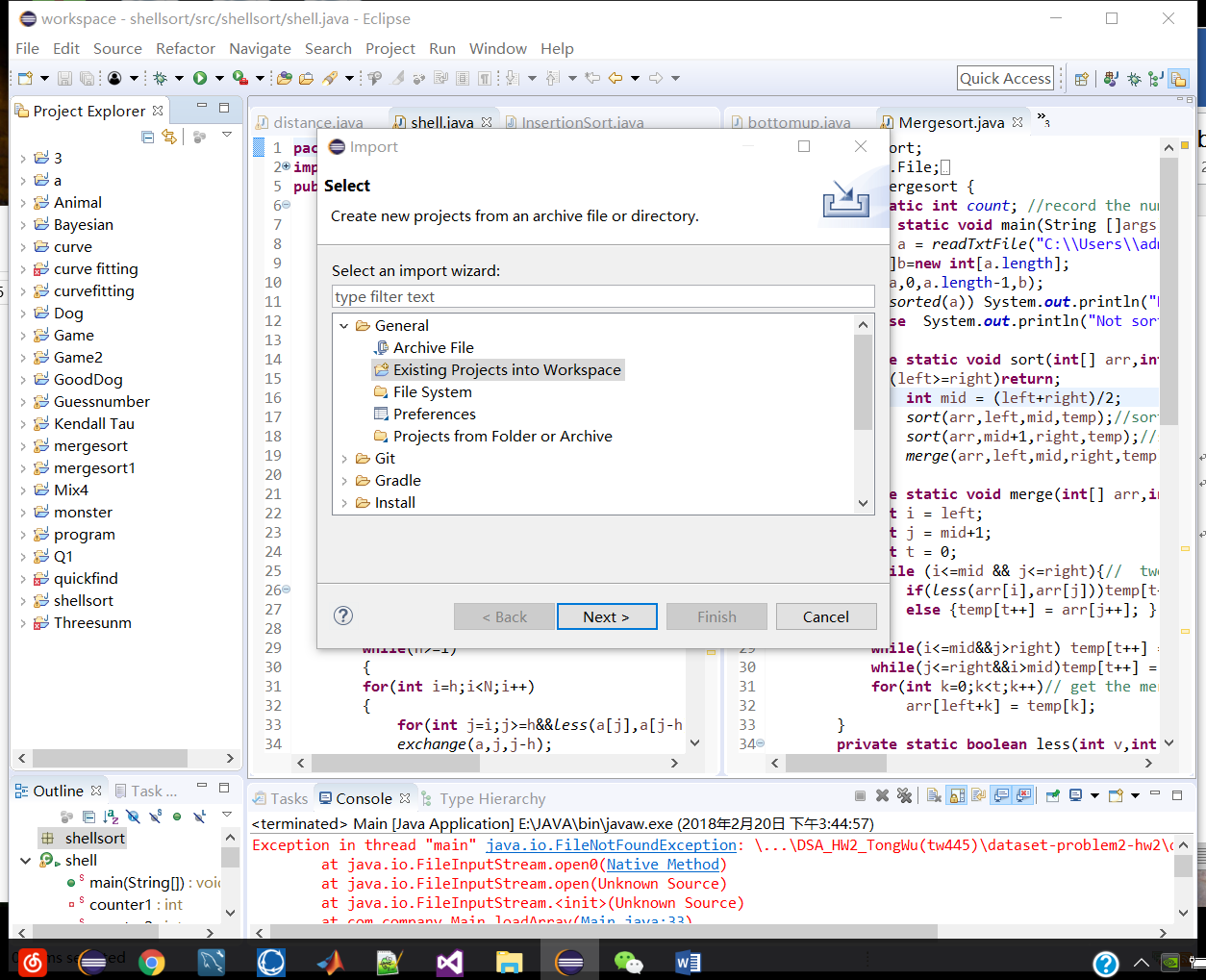
**HW2**

**Xinyu Lyu(xl422) 182007396**

**Instructions:**

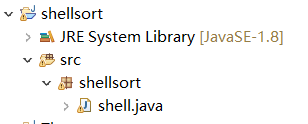
Language: java IDE: Eclipse Java Oxygen No extra library.

If you test with Eclipse IDE, you can simply import the project folder under the Q1/2/4/5 folder into the IDE. Shown below:

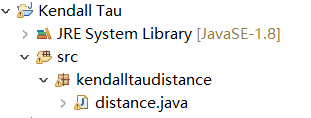


Or if you want to build new project. Make sure the project and the package names are the same with the names shown below. And all the .java files you need are under the corresponding src folders.

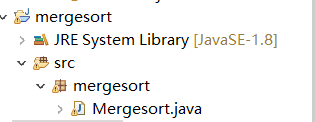
**Q1**: Build the new project like the Figure.1 below. In the readTxtFile(String filePath) function, input the right address of test file for both shell-sort and insertion sort.

Figure.1

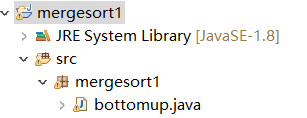
**Q2**: Build the new project like the Figure.2 below. In the readTxtFile(String filePath) function, input the right address of test file.

Figure.2

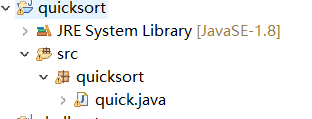
**Q4**: Build the new project like the Figure.3 below. In the readTxtFile(String filePath) function, input the right address of test file.

Figure.3

Build the new project like the Figure.4 below. In the readTxtFile(String filePath) function, input the right address of test file.

Figure.4

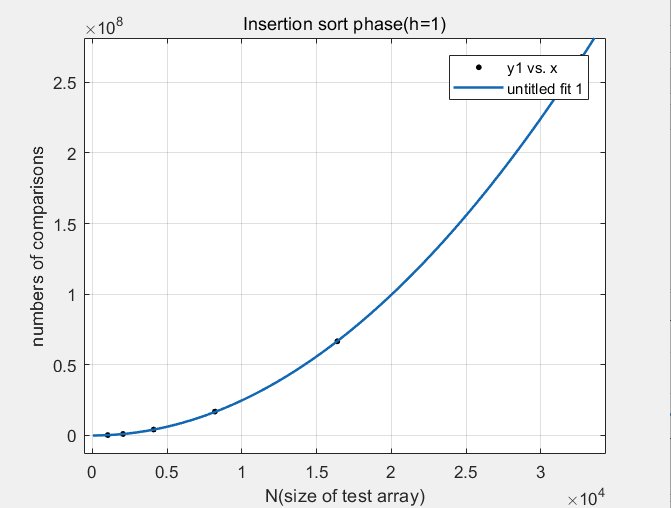
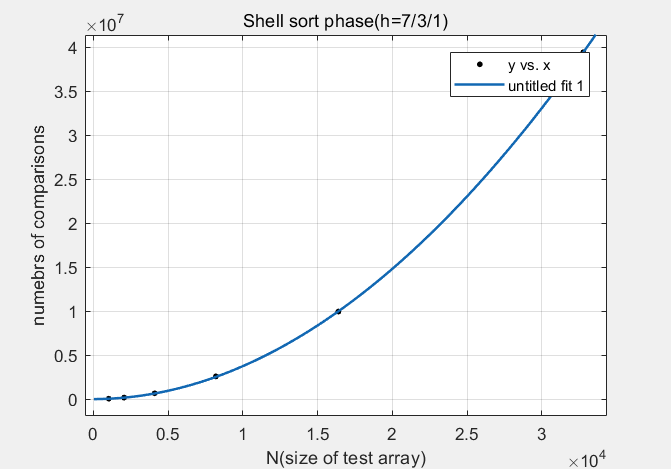
**Q5**. Build the new project like the Figure.5 below. In the readTxtFile(String filePath) function, input the right address of test file. And you can change the value of cut\_off .

Figure.5

**1. Implement Shellsort which** **reverts to insertion sort. (Use the increment**

**sequence 7, 3, 1). Create a table or a plot for the total number of comparisons made in the sorting the data for both cases (insertion sort phase and shell sort phase). Explain why Shellshort is more effective than Insertion sort in this case.**

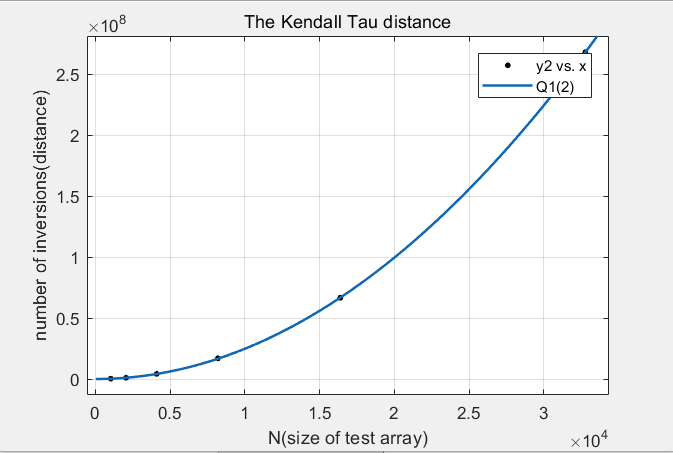
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 |
| Shell sort phase comparisons(h=7…3…1) | 46728 | 169042 | 660619 | 2576270 | 9950922 | 39442456 |
| Insertion sort phase comparisons (h=1) | 265553 | 1029278 | 4187890 | 16936946 | 66657561 | 267966668 |

For insertion sort, it is very slow when sorted such a big dataset. Because, it can only exchange the adjacent elements, and move the element from one end to the other one by one. The worst case is that the smallest element moves from the end of the dataset to the start of the dataset which needs N-1 movements to put it at right place. The shell-sort can change the nonadjacent element to partly sort the array. Then the partly sorted array is more effective for insertion sort/shell sort (h=1) which avoid the worst case. In this case, when h=7, every seven elements in the array is partly sorted. Then h=3, we make every 3 elements sorted which is more effective based on h=7 partly sorted. Therefore, the array is partly sorted every three elements which is more effective for shell sort h=1/insertion sort. However, if you sort the whole array directly use the insertion sort it will be very slow, for it only can exchange the adjacent elements one by one.

**2.** **The Kendall Tau distance is a variant of the "number of inversions" we discussed in class. It is defined as the number of pairs that are in different order in two** **permutations. Write an efficient program that computes the Kendall Tau distance in less than quadratic time on average. Plot your results and discuss. Use the dataset provided here. Note: data0.\* for convenience is an ordered set of numbers (in powers of two). data1.\* are shuffled data sets of**

**sizes (as given by "\*").**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 |
| Kendall Tau distance | 264541 | 1027236 | 4183804 | 16928767 | 66641183 | 267933908 |

The process of sorting is just the process of eliminating the inversions. For, the merge sort, the procedure of merging is exactly the process of eliminating the inversions. Therefore, when we do merge sort we can get the numbers of inversions. In the merge sort, before every merge process, we can count the inversions. For example, if the element a[j] in the second sub array is smaller than the element a[i] in the first array, the number of inversions in this case is the number of elements in the first sub array bigger or equal to a[i]. And those elements’ indexes are all bigger than i. It is very easy to count. For merge sort the complexity is O(nlogn) which is all about the comparisons in merge process. And it is less than the square of n.

**3. Create a data set of 8192 entries which has in the following order:** **1024 repeats of 1, 2048 repeats of 11, 4096 repeats of 111 and 1024 repeats of 1111. Write a sort algorithm that you think will sort this set "most"effectively. Explain why you think so.**

We can use the counting sort to effectively sort such a data set. For the raw array A, we record the number of elements equal to or less than A[i]. Then we can obtain the right place for it. For example, for the raw array {2,5,3,0,2,3,0,3}, we find out that there are 8 elements less or equal to the 5(including itself), so the element 5 should be placed at 8th place of the array A (the place for A[7]). In this case, we first traverse the array and record the number of elements less or equal to 1111, 111, 11, 1. Then we can find the right place for them to store. For 1111, there are 8192 elements equal or less than it, so we place it at a[8191]. Then we update the number of elements equal or less than 1111. When we meet another 1111, or the element less than 1111 we just place it at the right place. And the complexity is O(N) And the **pseudocode** shows below:

Counting-sort(A,B,k)// A[1~n] is the raw array, B[1~n] is the array to store the sorted elements, k is the maximal element in the dataset

let C[0..k]be a new array//C[0~k] store counting numbers less than or equal to the element in A

for i=0 to k

C[i]=0 //N

for j=1 to A.length

C[A[j]]=C[A[j]]+1//C[i] contains the counting number equal to the element A[j] or i// 2N

for i=1 to k

C[i]=C[i]+C[i-1]//C[i] contains the counting number less than or equal to the element i//2N

for j=A.length down to 1

B[C[A[j]]]=A[j]

C[A[j]]=C[A[j]]-1

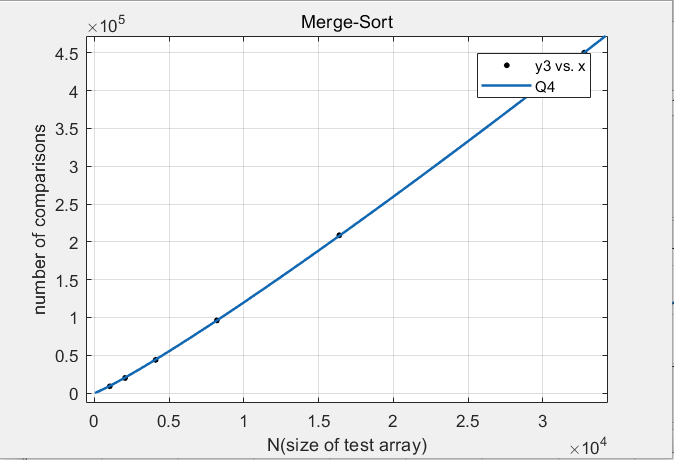
**Complexity analysis**: The initialization of the count array, and the second for loop which performs a prefix sum on the count array, each iterate at most k + 1 times and therefore take O(k) time. The other two for loops, and the initialization of the output array, each take O(n) time. Therefore, the time for the whole algorithm is the sum of the times for these steps, O(n + k).---cited from <https://en.wikipedia.org/wiki/Counting_sort>.

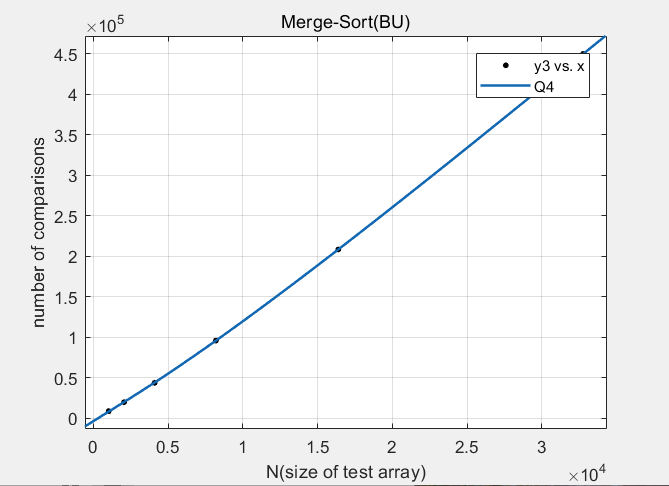
**4. Implement the two versions of MergeSort that we discussed in class. Create a**

**table or a plot for the total number of comparisons to sort the data (using data**

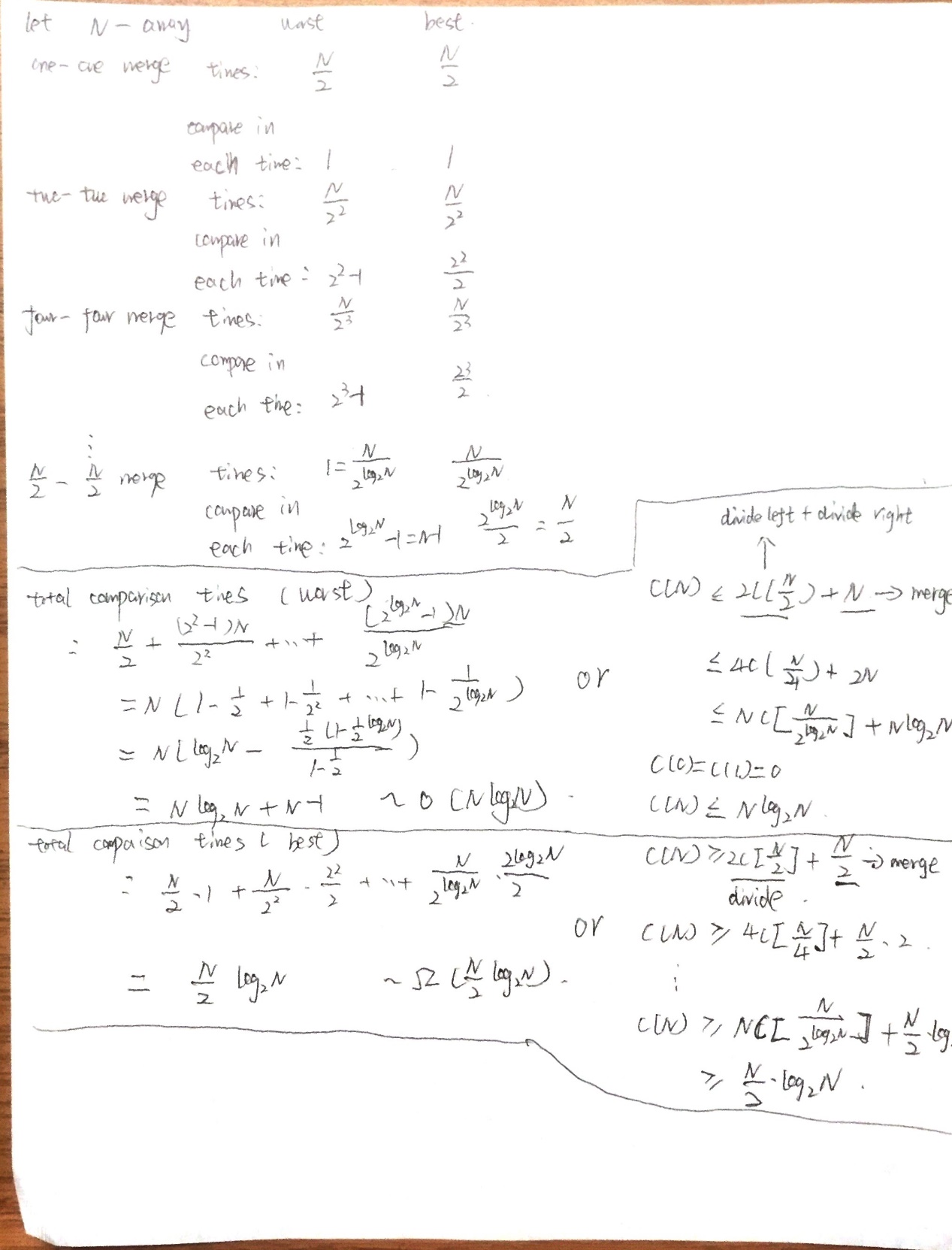
**set here) for both cases. Explain.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 |
| Merge sort Comparisons | 8954 | 19934 | 43944 | 96074 | 208695 | 450132 |
| Merge sort(BU) Comparisons | 8954 | 19934 | 43944 | 96074 | 208695 | 450132 |





The comparison number for both Mergesort and Mergesort BU are the same. Because these two kinds of sort are the same in nature. For mergesort, it divides the whole array into sub arrays with only one element by the recursion function sort to divide the left sub array and the right sub array. With only one element in the sub array, we can reasonably think it as the sorted array. Then we begin to merge them. With the recursion call of function sort, we recursively merge the sub array. Like such recursive sequence: one-element-array merge one-element-array to two-element-array, one merge one to two-element-array, two-element-array merge two-element-array to four. Therefore, for the worst case in one-one merge you need compare 1 time, two-two merge 3 times… N/2-N/2 merge N-1 times. Assume the N-array, you need N/2 times one-one merge, N/4 times two-two merge… 1 time N/2-N/2 merge. For Mergesort BU, we can see every element as each sub array of the whole array. Therefore, we needn’t to divide it again. We only need to merge size by size. First do one-one merge. Then two-two merge. We can clearly see that the only difference in such two merge is sequence of divide and merge. They have the same number of merge process which means they have same amount of comparisons when sorting the same array. And the specific proof of the complexity from N/2logN~NlogN is shown in the picture below.



**5. Implement Quicksort using** **median-of-three to determine the partition element. Compare the performance of Quicksort with the Mergesort implementation and dataset from Q4.** **Is there any noticeable difference when you use N=7 as the cut-off to insertion sort.** **Experiment if there is any value of "cut-off to insertion" at which the performance inverts.**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1024 | 2048 | 4096 | 8192 | 16384 | 32768 |
| Quicksort Comparisons | 10517 | 24768 | 52916 | 114618 | 256106 | 538506 |
| Quicksort(cut off=7) Comparisons | 10039 | 23884 | 51184 | 110887 | 248800 | 523534 |
| Merge sort Comparisons | 8954 | 19934 | 43944 | 96074 | 208695 | 450132 |
| Merge sort(BU) Comparisons | 8954 | 19934 | 43944 | 96074 | 208695 | 450132 |

Comparing the performance of Quicksort with the Mergesort, the comparisons of merge sort is a little less than the quicksort, but with the same order of magnitude with the quicksort as O(nlogn).

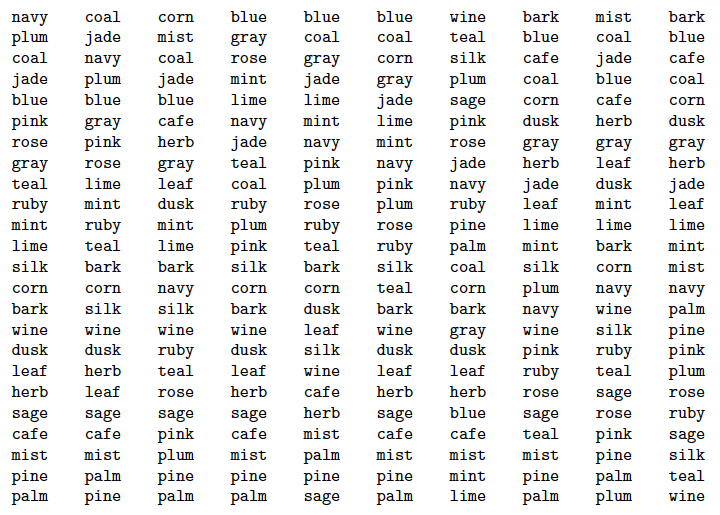
With Cut off =7, number of comparisons of Quicksort(cut off) is a little less than the Quicksort without cut off. However, not very noticeable about 3% reduction.

Test file : data1.32768.txt

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cut off | 0 | 3 | 5 | 7 | 9 | 11 | 15 | 16 | 18 |
| Quicksort comparisons | 523671 | 524204 | 542859 | 531309 | 532852 | 526889 | 523508 | 530212 | 549853 |
| Quicksort(cut off) comparisons | 523671 | 509052 | 52]]]6692 | 516614 | 521108 | 519358 | 526129 | 536159 | 562502 |

When the cut off to insertion is above 15, the performance of quicksort inverts, because the number of comparisons of quicksort(cut off) is larger than the quicksort(without cut off).

**6. View the following Data Set here. The column on the left is the original input of strings to be sorted or shuffled; the column on the extreme right are the string in sorted order; the other columns are the contents at some intermediate step during one of the 8 algorithms listed below. Match up each algorithm under the corresponding column. Use each algorithm exactly once: (1)Knuth shuffle (2) Selection sort(3) Insertion sort (4) Mergesort(top-down)(5)Mergesort (bottom-up) (6) Quicksort (standard, no shuffle) (7) Quicksort (3-way,no shuffle) (8) Heapsort.**



|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| original | Merge bottom-up | Quicksort  (standard) | Knuth shuffle | Merge top-down | Insertion  sort | Heap  sort | Selection sort | Quicksort  (3-way) | Sorted |