*Weijia Sun*

*ws368 | 02/19/2018*

*Assignment2*

Data Structure & Algorithm

Assignment 2

Weijia Sun | ws368

Table:Number of Comparisons – Dataset (Already Sorted)

|  |  |  |
| --- | --- | --- |
| dataset | shell sort | insert sort |
| data0.1024 | 3061 | 1023 |
| data0.2048 | 6133 | 2047 |
| data0.4096 | 12277 | 4095 |
| data0.8192 | 24565 | 8191 |
| data0.16384 | 49141 | 16383 |
| data0.32768 | 98293 | 32767 |

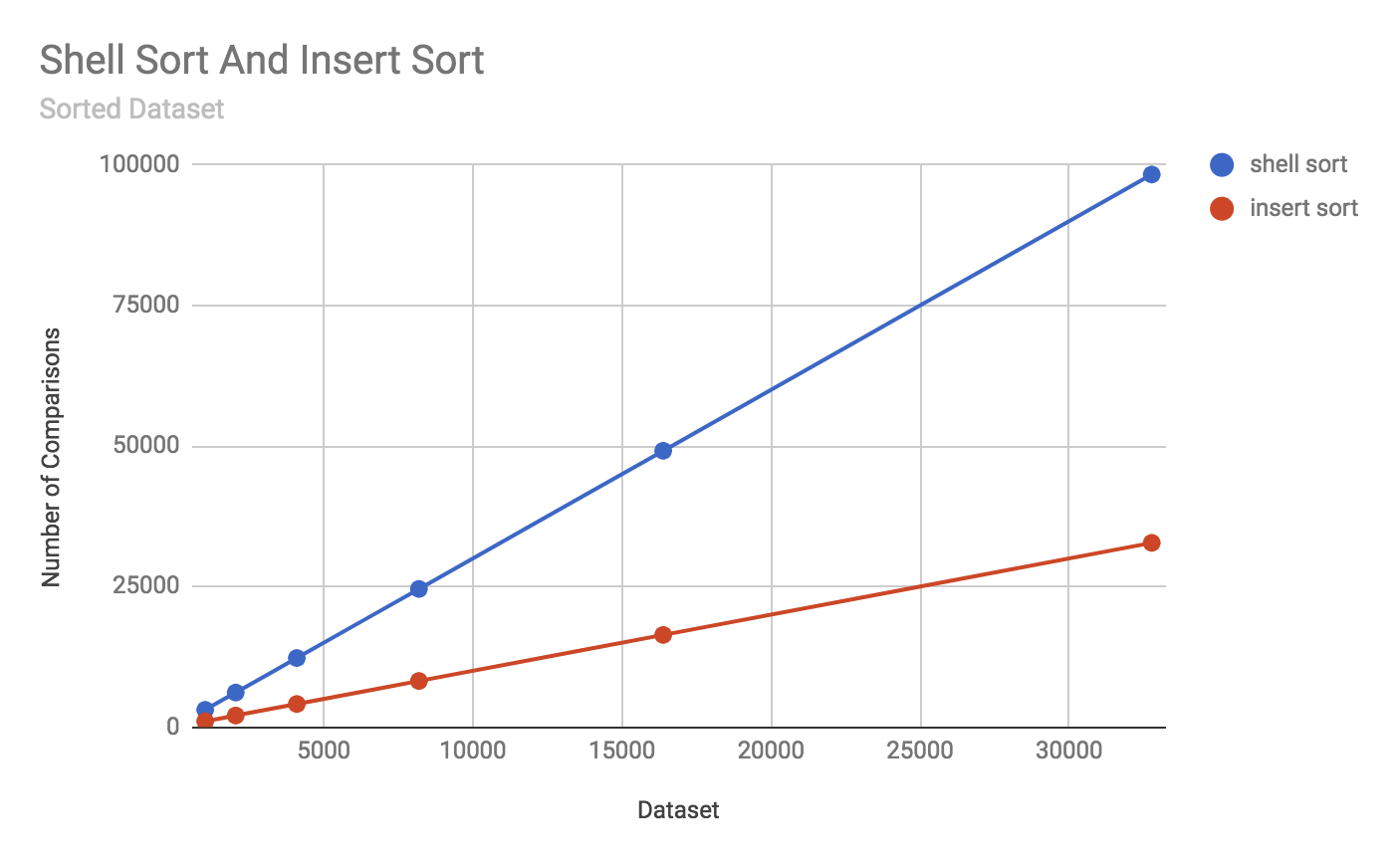
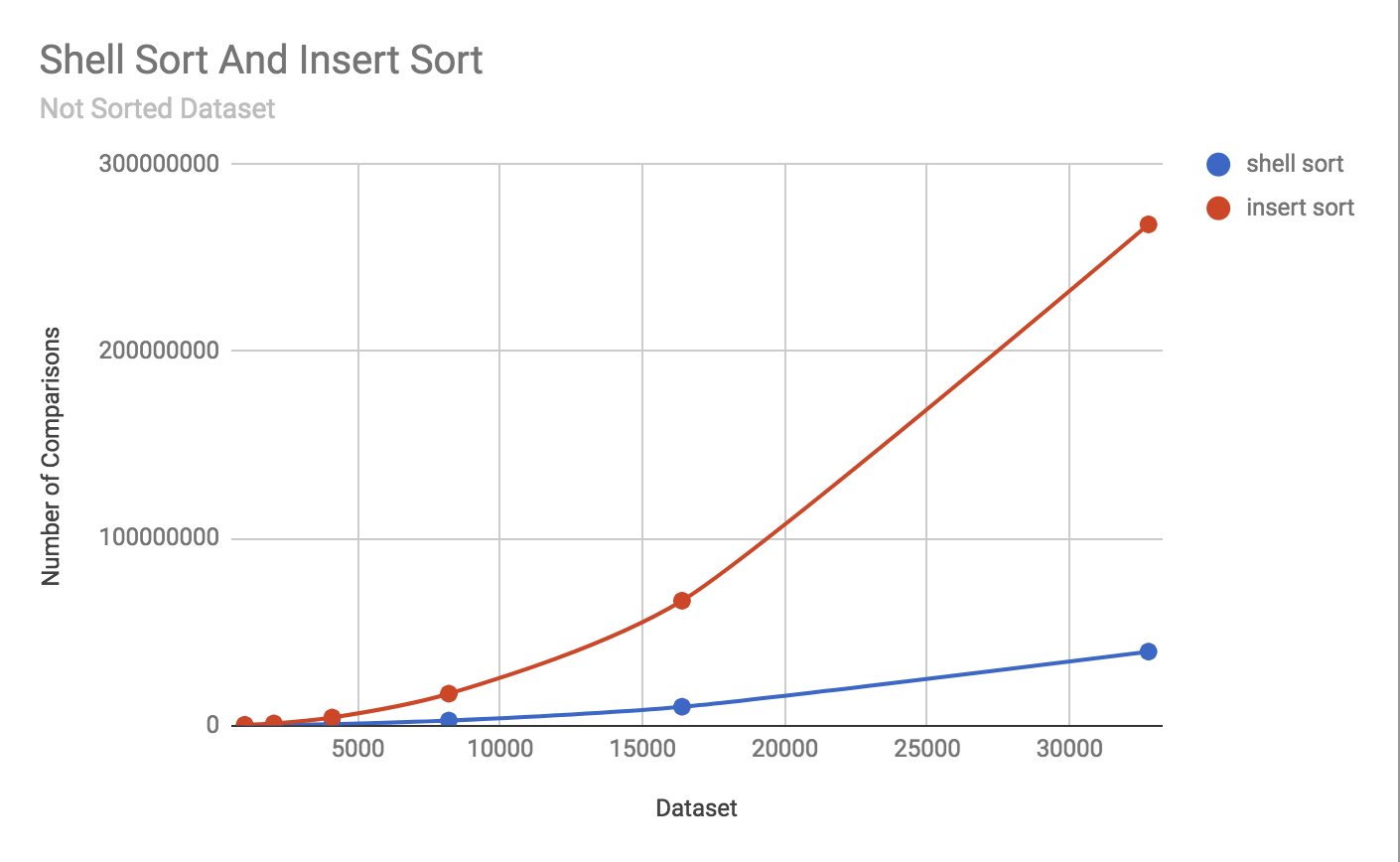


Table:Number of Comparisons – Dataset (Not Sorted)

|  |  |  |
| --- | --- | --- |
| dataset | shell sort | insert sort |
| data1.1024 | 46728 | 265553 |
| data1.2048 | 169042 | 1029278 |
| data1.4096 | 660619 | 4187890 |
| data1.8192 | 2576270 | 16936946 |
| data1.16384 | 9950922 | 66657561 |
| data1.32768 | 39442456 | 267966668 |



**Analyze：**

For the first data set which is already sorted. It is the best case for insertion sort. The number of comparisons is always N-1. But for the shell sort, it has to scan the array for several times when h equals to different value. When h=1, the number of comparisons has already been N-1. So for sorted array, insert sort does better than the shell sort.

For the second dataset which is randomly given. Shell sort does way better than insert sort. Because it efficiently avoid the long-distance 1-sort. It allows elements to move long distance by beginning with large h-value, which can reduce large amounts of disorder quickly. Then the data becomes partially sorted. In this situation, smaller h-sort can solve the problem efficiently. So that’s why shell sort works better than insert sort in random given array.

Table Name: Running Time-Data Size

|  |  |
| --- | --- |
| dataset | Running Time(s) |
| 1024 | 302 |
| 2048 | 514 |
| 4096 | 683 |
| 8192 | 1052 |
| 16384 | 2668 |
| 32768 | 5043 |

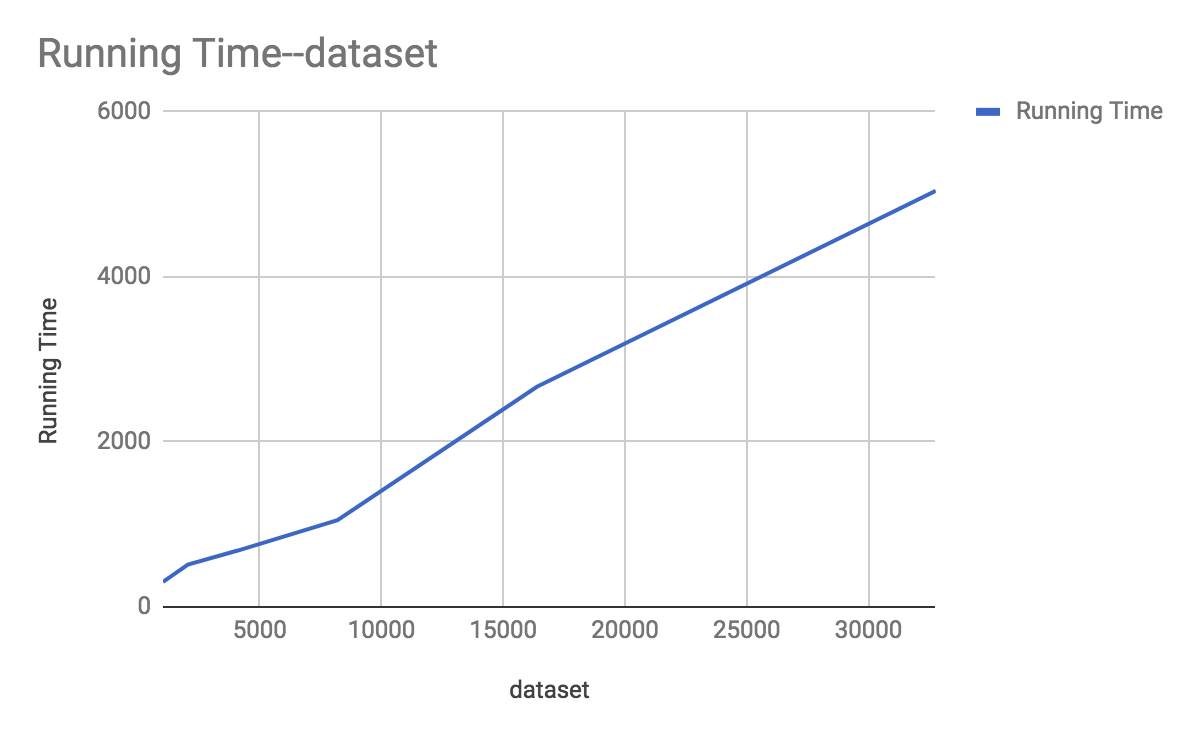


Table Name: Number of inversions – Dataset

|  |  |
| --- | --- |
| Dataset | Number of Inversions |
| data.1024 | 264541 |
| data.2048 | 1027236 |
| data.4096 | 4183804 |
| data.8192 | 16928767 |
| data.16384 | 66641183 |
| data.32768 | 267933908 |

Discuss:

For this question, it just to count the number of the inversions in one array. I use the thought of merge sort to count the number of inversion in complexity of O(nlogn). The key part o the algorithms is as follows:

When merge two subarray, assume i is the index for left subarray, j is the index of the right array. When array[i]>array[j]. For all the element in the left array at the right sort of array[i] is bigger than array[j] . So under this circumstance, there are mid-i+1 pair of inversions. And same for rest of them.

Basically, as you can see from the graph, the time complexity for this algorithm is the same with merge sort. So it is O(nlogn).

The result of the running time is the average of 5 times calculationg.

For this problem, I use the counting sort method. Its’ time complexity is only o(n+r). The n represents the number of elements in the array. The r represents the biggest number in the array. The space complexity is O(r).

But for this question, we can assume we already know the values in the array in advance. So for the counting array ,we can directly create a size 4 array, and use the array record the number of times each value. And use the count array to output the result.

Counting sort is not a comparison based sort, so for the time complexity, it is smaller than any comparison based sort method. So for this problem, I think it is the most efficient sorting method.

Basically, the complexity of the modified counting sort here is O(n). Also, since the array is already sorted, both the bubble sort and the insertion sort perform as O(n) algorithm here, too. So for this question , I think modified counting sort, bubble sort and the insertion sort are the most efficient algorithm.

Table: Number of Comparisons(UB and BU) -- Dataset

|  |  |  |
| --- | --- | --- |
| dataset | number of comparisons(UB) | number of comparisons(BU) |
| data0.1024 | 5120 | 5120 |
| data0.2048 | 11264 | 11264 |
| data0.4096 | 24576 | 24576 |
| data0.8192 | 53248 | 53248 |
| data0.16384 | 114688 | 114688 |
| data0.32768 | 245760 | 245760 |
| data1.1024 | 8954 | 8954 |
| data1.2048 | 19934 | 19934 |
| data1.4096 | 43944 | 43944 |
| data1.8192 | 96074 | 96074 |
| data1.16384 | 208695 | 208695 |
| data1.32768 | 450132 | 450132 |

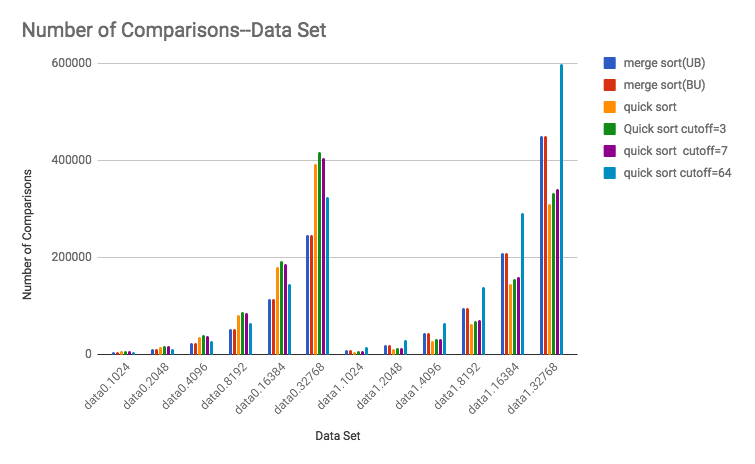
Explain:

As we can see, the number of comparisons needed for UB is same with BU’s. Because when number of element is the power of 2. Both the number of comparisons and the number of array access needed for BU and UB is same, just different at the order.

Table(Number of Comparisons--dataset)

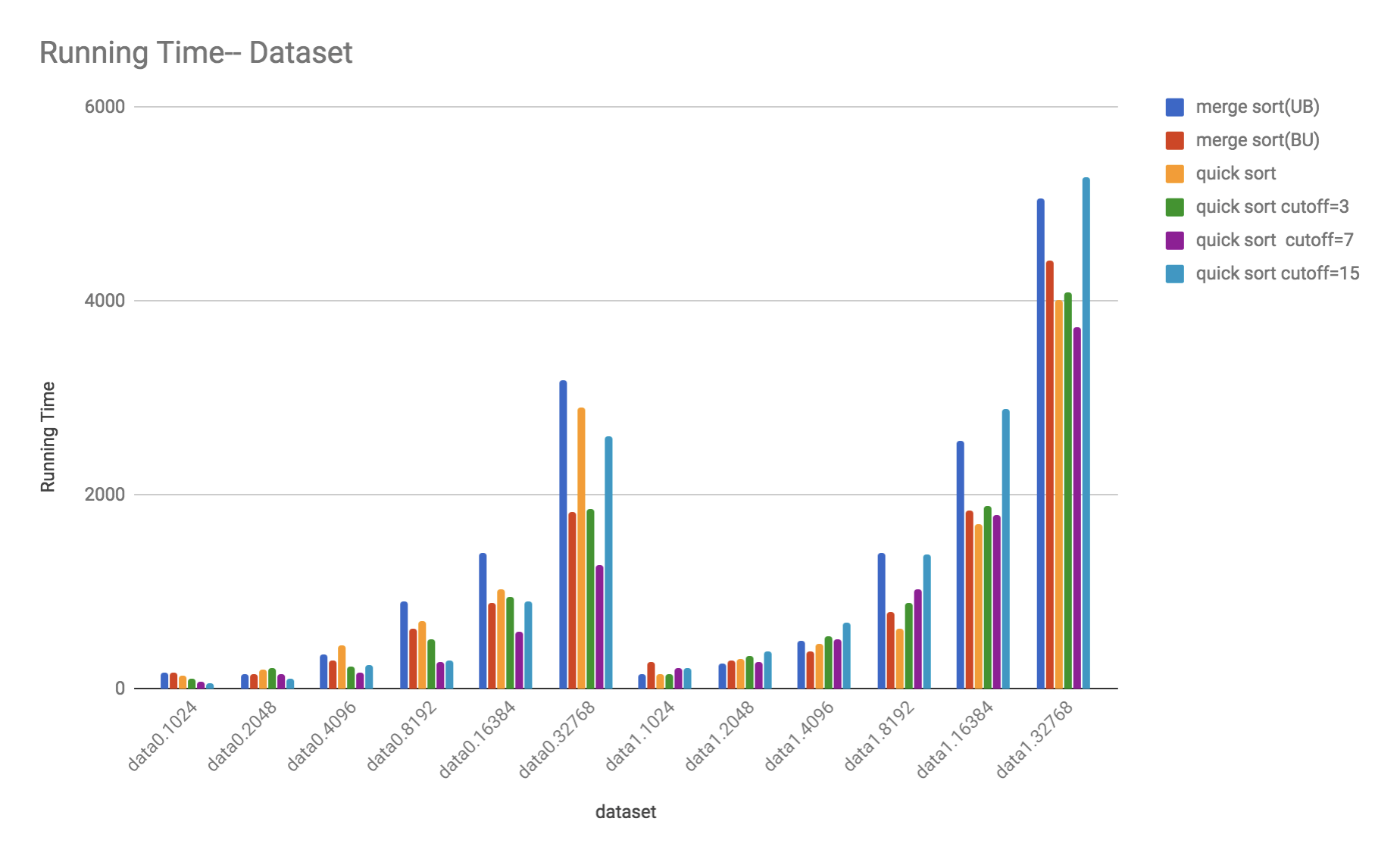
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| dataset | merge sort(UB) | merge sort(BU) | quick sort | Quick sort cutoff=3 | quick sort cutoff=7 | quick sort cutoff=64 |
| data0.1024 | 5120 | 5120 | 7181 | 7948 | 7567 | 5048 |
| data0.2048 | 11264 | 11264 | 16398 | 17933 | 17168 | 12137 |
| data0.4096 | 24576 | 24576 | 36879 | 39950 | 38417 | 28362 |
| data0.8192 | 53248 | 53248 | 81936 | 88079 | 85010 | 64907 |
| data0.16384 | 114688 | 114688 | 180241 | 192528 | 186387 | 146188 |
| data0.32768 | 245760 | 245760 | 393234 | 417809 | 405524 | 325133 |
| data1.1024 | 8954 | 8954 | 6046 | 6716 | 7005 | 15231 |
| data1.2048 | 19934 | 19934 | 12704 | 14033 | 14637 | 29482 |
| data1.4096 | 43944 | 43944 | 28724 | 31399 | 32562 | 65785 |
| data1.8192 | 96074 | 96074 | 63576 | 68956 | 71200 | 139050 |
| data1.16384 | 208695 | 208695 | 145509 | 156274 | 160686 | 290796 |
| data1.32768 | 450132 | 450132 | 310593 | 332125 | 341283 | 598193 |

From this table, we can plot a graph:



Table(Running Time--Dataset)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| dataset | merge sort(UB) | merge sort(BU) | quick sort | Quick sort cutoff=3 | quick sort cutoff=7 | quick sort cutoff=64 |
| data0.1024 | 168 | 173 | 99 | 60 | 101 | 53 |
| data0.2048 | 155 | 150 | 134 | 81 | 181 | 98 |
| data0.4096 | 347 | 286 | 234 | 119 | 287 | 247 |
| data0.8192 | 900 | 627 | 457 | 256 | 557 | 294 |
| data0.16384 | 1407 | 892 | 1009 | 761 | 962 | 896 |
| data0.32768 | 3188 | 1828 | 1960 | 1496 | 2468 | 2613 |
| data1.1024 | 152 | 269 | 106 | 187 | 218 | 87 |
| data1.2048 | 255 | 294 | 324 | 493 | 267 | 764 |
| data1.4096 | 497 | 391 | 740 | 744 | 625 | 450 |
| data1.8192 | 1405 | 793 | 936 | 852 | 1459 | 943 |
| data1.16384 | 2556 | 1840 | 2571 | 2043 | 3141 | 2171 |
| data1.32768 | 5065 | 4425 | 5268 | 3840 | 5974 | 5964 |



For this question, we already know the data0 series data set is already sorted, the data1 series data set is unsorted and is not the worst case. So I don’t shuffle the array in the quicksort.

I compare both the number of comparisons and the running time. Here I use the running time to compare the performance of the runtime complexity. Since the runtime complexity is estimated by the elementary operations of the algorithms, except for the comparisons, it still has the array access to consider. So running time is more considerable way to estimate the runtime complexity here.

As we can see, for quicksort, it performs worse than merge sort(Bottom-up) better than merge sort(Top-Down) at the already sorted array and performs better than merge sort at the random array.

Because in the small array, the insertion sort is quicker than quick sort, since even in a very small subarray, quicksort still always keeps calling itself. So when we set a cutoff value to turn to insertion sort, the algorithm will do better. Also, since the insertion sort is O(n) algorithm at already sorted array, and the quick sort is O(nlogn) at the already sorted array. So the insertion sort performs better at data0 series.

So when I set the cutoff value = 7, it is better than standard quicksort in both case. Also, when cutoff value is set to be smaller(3) or bigger(15), it’s slower than when cutoff=7. So cutoff = 7 is the most suitable value here.

Column2: Merge sort(Bottom-up), because every 4 elements has been sorted.

Column3: Quick sort(Standard, no shuffle), because we can observe that, before word ”navy”, all the element is “smaller” than navy, after “navy”, all the element is bigger than “navy” (alphabetically). So the “navy” is the pivot here.

Column4:Kruth Shuffle: It seems keep implement Kruth Shuffle until word ”silk”.

Column5:Merge sort(top down), because the left half of the array has been sorted, and the each half of the last half of the array has been sorted. So it’s merge sort. But here the number of the array’s element is the power of 2, so it cannot be sure whether it is Bottom up or Top down. Considering we have already make sure the Column2 is Bottom up, so for here, it is top down version.

Column6: Insertion sort, because the invariants for insertion sort is: 1. Entries the left of index fixed and in ascending order. 2. Entries to the right of the index haven’t been seen. Compare to column 8, we can assure column 6 is insertion sort, which up to “teal”

Column7: The current column is an max heap. So it is in an intermediate step of Heap Sort

Column8: Selection sort, because the invariants for selection sort is : 1. Entries the left of index fixed and in ascending order. 2. No entry to the right of index is smaller than any entry to the left of index. So it’s selection sort up to “mint”

Column9:Quicksort(3-way,no shuffle),from the picture ,we can assure that the “navy” is the pivot here. We can tell the difference form the standard Quicksort by the place of the word”plum”. For 3 way quick sort. “plum” is the first word to compare with the pivot “navy”, since it is larger than “navy”. So it exchange with the last word in the column, so “plum” is the last element in the column now.