**DSA Homework 2 Report**

Haocong Wang

[mw814@scarletmail.rutgers.edu](mailto:mw814@scarletmail.rutgers.edu)

**Question 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| File Name | Insertion | Shell | Runtime of Insertion | Runtime of Shell |
| data0.1024 | 1023 | 3061 | 0.000561237 | 0.000735998 |
| data0.2048 | 2047 | 6133 | 0.000982046 | 0.001269102 |
| data0.4096 | 4095 | 12277 | 0.002062082 | 0.002754688 |
| data0.8192 | 8191 | 24565 | 0.004192114 | 0.004617929 |
| data0.16384 | 16383 | 49141 | 0.007228851 | 0.009440184 |
| data0.32768 | 32767 | 98293 | 0.016254902 | 0.018826962 |
| data1.1024 | 265553 | 46768 | 0.072683096 | 0.01025176 |
| data1.2048 | 1029278 | 169081 | 0.299160957 | 0.040673971 |
| data1.4096 | 4187890 | 660673 | 1.303462029 | 0.183189869 |
| data1.8192 | 16936946 | 2576322 | 4.897849083 | 0.603111029 |
| data1.16384 | 66657561 | 9950984 | 19.38217998 | 2.316880941 |
| data1.32768 | 267966668 | 39442505 | 78.57995605 | 9.297394037 |

As we can see from the figures and plots, when the data is sorted (data0.\*), insertion sort is more effective. This makes sense because when the data is sorted, the insertion sort only takes (N-1) times of comparison while the shell sort need to compare (3\*N-11) times of comparison.

However, when the data is shuffled, things are different. The shell, obviously, is much more effective than the insertion sort. Insertion sort works well on short sequences or partly sorted sequences, but shell sort first divides a long shuffled sequence into several short sequences and sort each part with insertion sort, which is more effective than using insertion sort all the way.

图片包含 屏幕截图

描述已自动生成图片包含 屏幕截图

描述已自动生成

**Question 2**

|  |  |  |
| --- | --- | --- |
| File size (data1) | Running time | Distance |
| 1024 | 0.004781723 | 264541 |
| 2048 | 0.009749889 | 1027236 |
| 4096 | 0.025924206 | 4183804 |
| 8192 | 0.045995951 | 16928767 |
| 16384 | 0.09845376 | 66641183 |
| 32768 | 0.216032982 | 267933908 |

After searching online on the definition of the Kendall Tau distance, I found that the Kendall Tau distance in this question is actually a sorting question, in which the distance is the number of comparisons. This is because the data0.\* are all sorted. To calculate the distance effectively, I chose the merge sort algorithm and count the comparison times. As we can see from the plot, the O(N) of the runtime is almost linear. According to what we have learned from the course, the runtime cost should be O(N\*logN).

图片包含 文字, 地图

描述已自动生成图片包含 文字

描述已自动生成

**Question 3**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| data0 | comp in merge sort | comp in BUmer | runtime in merge sort | runtime in BUmer |
| 1024 | 5120 | 5120 | 0.005332947 | 0.00449419 |
| 2048 | 11264 | 11264 | 0.00886488 | 0.007612944 |
| 4096 | 24576 | 24576 | 0.020756006 | 0.019711971 |
| 8192 | 53248 | 53248 | 0.040574312 | 0.035650015 |
| 16384 | 114688 | 114688 | 0.086947918 | 0.077246189 |
| 32768 | 245760 | 245760 | 0.19887805 | 0.177154779 |
| data1 | comp in merge sort | comp in BUmer | runtime in merge sort | runtime in BUmer |
| 1024 | 8954 | 8954 | 0.004723072 | 0.004546881 |
| 2048 | 19934 | 19934 | 0.011224985 | 0.01044488 |
| 4096 | 43944 | 43944 | 0.045725822 | 0.038417816 |
| 8192 | 96074 | 96074 | 0.048655987 | 0.04848671 |
| 16384 | 208695 | 208695 | 0.097422123 | 0.105820894 |
| 32768 | 450132 | 450132 | 0.237993002 | 0.218292952 |

As we can see, the number of comparisons, no matter it is bottom-up merge sort or regular merge sort, is always the same for each dataset. This is because the data size is power of 2, so the comparison times are always the same. If the size is not power of 2, the situation will be different.

The runtime cost, however, differs a little bit. The bottom-up merge sort is faster than the regular merge sort because the bottom-up merge sort does not use recursion, which usually takes much time in the regular merge sort. The difference of runtime cost in this question is not obvious. I think if the data size is not power of 2, the difference will be clearer.

**Question 4**

In this question, I tried merge sort, insertion sort and bubble sort. All of them take O(N) runtime cost, but the bubble sort is the slowest. The merge sort is very quick and effective. The dataset is already sorted, so the merge sort basically just checks if the data is sorted recursively. The insertion sort is also effective. This is also because the data is sorted. It only takes (N-1) times of comparisons if we use insertion sort to sort this data.

**Question 5**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| data0 | merge sort | quick sort | cutoff = 7 | cutoff = 10 | cutoff = 15 |
| 1024 | 0.004071 | 0.00274611 | 0.00286198 | 0.00308084 | 0.00326514 |
| 2048 | 0.00863194 | 0.00610089 | 0.00616002 | 0.0061872 | 0.00662899 |
| 4096 | 0.01794934 | 0.01549697 | 0.01305509 | 0.01382279 | 0.01385593 |
| 8192 | 0.04331112 | 0.02855492 | 0.02789497 | 0.02806401 | 0.03001714 |
| 16384 | 0.07899594 | 0.06079388 | 0.05968618 | 0.06327868 | 0.06398082 |
| 32768 | 0.17245698 | 0.12750006 | 0.11651373 | 0.12713385 | 0.13007784 |
| data1 | merge sort | quick sort | cutoff = 7 | cutoff = 10 | cutoff = 15 |
| 1024 | 0.00521278 | 0.00335383 | 0.00294471 | 0.00310802 | 0.00327802 |
| 2048 | 0.00861716 | 0.007231 | 0.00661778 | 0.00690818 | 0.00715184 |
| 4096 | 0.0233109 | 0.01764297 | 0.015378 | 0.01549888 | 0.01656508 |
| 8192 | 0.0366292 | 0.03300571 | 0.03183627 | 0.03272414 | 0.03293109 |
| 16384 | 0.09173727 | 0.07215309 | 0.06971598 | 0.07360077 | 0.07505202 |
| 32768 | 0.20467806 | 0.14385796 | 0.14204168 | 0.14216709 | 0.14427805 |

In this question, I compared the performance, which is the runtime cost, among merge sort, quick sort and quick sort with cut-off. There is no doubt that the quick sort, with or without cut-off, is faster than the merge sort. This is because the merge sort uses recursive method, which takes much time when running the algorithm.

When comparing the performance between the quick sort and the quick sort with cut-off, I found that when the data is a sorted short sequence, quick sort is faster than quick sort with cut-off. As the data grows larger, quick sort with cut-off becomes better than quick sort. When the data is shuffled, quick sort with cut-off is also faster.

As for the number of cut-off, I get the result from the figure that in this question, the cut-off in the range from 7 to 10 is a good choice. Even if some of them are slower than quick sort, the difference is very small. For the cut-off that is larger than 10, the runtime cost is obviously larger, especially in the small dataset.

图片包含 文字, 地图

描述已自动生成图片包含 文字, 地图

描述已自动生成图片包含 文字, 地图

描述已自动生成图片包含 文字, 地图

描述已自动生成

**Question 6**

**Col.2:** Mergesort (bottom-up). Every four elements are sorted.

**Col.3:** Quicksort (standard, no shuffle). Every element before “navy” is smaller than it while every element after is bigger.

**Col.4:** Knuth shuffle. All the elements before “silk” is shuffled and the other are in place.

**Col.5:** Mergesort (top-down). The first half of the column is sorted and both halves of the last half of the column are sorted respectively.

**Col.6:** Insertion sort. All the elements before “teal” are sorted, but they are different from the last column and other elements remain in place.

**Col.7:** Heap sort. This column looks like a max heap, so I think it is a heap sort.

**Col.8:** Selection sort. All the elements before “mint” are sorted and are the same of the last column while other elements remain in place.

**Col.9:** Quicksort (3-way, no-shuffle). Every element before “navy” is smaller than it while every element after is bigger. The last one is “plum”, which is first element that is bigger than “navy”. In the quick sort with 3-way, it is exchanged with the arr[hi] and get decrement.