**Homework 2 - Pratik Mistry : Spring DSA 2020 (pdm79)**

**Solution 1:**

Results:

1. Time Complexity for Shell Sort Phase and Insertion Sort Phase

|  |  |  |  |
| --- | --- | --- | --- |
| Shell Sort All The Way | Insertion Sort (h=1) | Data Set | |
| Time (ms) | Time (ms) |
| 1.41 | 0.46 | 1024 | dataset0.\* |
| 3.05 | 0.95 | 2048 |
| 6.14 | 2 | 4096 |
| 12.15 | 4.03 | 8192 |
| 23.3 | 8.42 | 16384 |
| 50.09 | 16.46 | 32768 |
| 45.28 | 332.6 | 1024 | dataset1.\* |
| 177.46 | 1177.15 | 2048 |
| 710.59 | 4528.88 | 4096 |
| 3209.29 | 19738.74 | 8192 |
| 13721.08 | 78297.7 | 16384 |
| 42303.87 | 295337.98 | 32768 |

1. Complexity Count for Shell Sort Phase and Insertion Sort Phase

|  |  |  |  |
| --- | --- | --- | --- |
| Shell Sort All The Way | Insertion Sort (h=1) | Data Set | |
| Complexity Count | Complexity Count |
| 3061 | 1023 | 1024 | dataset0.\* |
| 6133 | 2047 | 2048 |
| 12277 | 4095 | 4096 |
| 24565 | 8191 | 8192 |
| 49141 | 16383 | 16384 |
| 98293 | 32767 | 32768 |
| 46768 | 265564 | 1024 | dataset1.\* |
| 169081 | 1029283 | 2048 |
| 660673 | 4187899 | 4096 |
| 2576322 | 16936958 | 8192 |
| 9950984 | 66657566 | 16384 |
| 39442505 | 267966675 | 32768 |

Graphs:

1. Time Complexity v/s Dataset0
2. Time Complexity v/s Dataset1

1. Complexity Count v/s Dataset0
2. Complexity Count v/s Dataset1

Explanation:

From the results and graphs of Time Complexity (T (n)) or Complexity Count (O (n)), below are the inferences clearly drawn:

* In **dataset0.\*** since all the elements are already sorted, the time complexity or complexity count for Shell Sort (Insertion Sort Phase) i.e. when h=1 behaves very well because when h=1 it is Insertion Sort and thus complexity in case of sorted elements is **linear** i.e. **O (n)**.
* While the complexity for the shell sort all the way for h=7, 3, 1 is not linear. It is almost **O (n) = nlog(n)** i.e. **linearithmic** since the data is sorted
* In **dataset1.\*** since elements are not sorted, the time complexity or complexity count for Shell Sort (Insertion Sort Phase) i.e. when h=1 does not behave good as it is Insertion sort and the complexity would be nearly **O (n) = n^2** i.e. **quadratic** in worst case.
* While the complexity for the shell sort all the way for h=7, 3, 1 behaves well compared to Insertion Sort Phase with complexity of **O (n) = nlog(n)** i.e. **linearithmic** in average case.
* Thus, for sorted datasets Insertion Sort Phase behaves well compared to Shell Sort all the way while its reverse in case for unsorted datasets.

**Solution 2:**

Results: Complexity Count for Kendall Tau Distance between Sophisticated and Brute Force Methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Complexity Count  (Sophisticated) | Complexity Count  (Brute Force) | Kendall Tau Distance  (Sophisticated) | Kendall Tau Distance  (Brute Force) | Dataset |
| 10240 | 523776 | 264541 | 264541 | 1024 |
| 22528 | 2096128 | 1027236 | 1027236 | 2048 |
| 49152 | 8386560 | 4183804 | 4183804 | 4096 |
| 106496 | 33550336 | 16928767 | 16928767 | 8192 |
| 229376 | 134209536 | 66641183 | 66641183 | 16384 |
| 491520 | 536854528 | 267933908 | 267933908 | 32768 |

Graphs:

1. Complexity Count v/s Dataset : Brute Force Method
2. Complexity Count v/s Dataset : Sophisticated Method

Explanation:

* Kendall Tau Distance between two arrays has some conditions: 1. Array sizes must be same. 2. Array elements must be same.
* Since, we are using array1 as dataset0.\* and array2 as dataset1.\*, and dataset0.\* is already **sorted** then Kendall Tau Distance is number of inversions in dataset1.\* i.e. array2
* To find the inversions as Kendall Tau Distance, I tried using Brute Force Method, but the complexity is **O (n) = n^2** i.e. **quadratic** as seen in graphs and results above.
* Thus, to find the inversions using sophisticated method, I have used **Merge Sort** for array2 as sophisticated method and the complexity is nearly **O (n) = nlogn** i.e. **linearithmic**.
* Thus sophisticated method by using merge sort to find inversions i.e. Kendall Tau distance behaves well and can be seen in the graphs and results above.

**Solution 3:**

Results:

1. Time Complexity for Merge Sort (Top Down) and Merge Sort (Bottom Up)

|  |  |  |  |
| --- | --- | --- | --- |
| Top Down | Bottom Up | Data Set | |
| Time (ms) | Time (ms) |
| 15.87 | 16.36 | 1024 | dataset0.\* |
| 73.72 | 35.45 | 2048 |
| 83.58 | 123.78 | 4096 |
| 175.96 | 191.42 | 8192 |
| 343.58 | 338.64 | 16384 |
| 733.3 | 761.29 | 32768 |
| 21.02 | 17.56 | 1024 | dataset1.\* |
| 38.02 | 37.55 | 2048 |
| 102.44 | 130.77 | 4096 |
| 178.35 | 191.62 | 8192 |
| 382.42 | 386.32 | 16384 |
| 809.21 | 939.9 | 32768 |

1. Complexity Count for Merge Sort (Top Down) and Merge Sort (Bottom Up)

|  |  |  |  |
| --- | --- | --- | --- |
| Top Down | Bottom Up | Data Set | |
| Complexity Count | Complexity Count |
| 10240 | 10240 | 1024 | dataset0.\* |
| 22528 | 22528 | 2048 |
| 49152 | 49152 | 4096 |
| 106496 | 106496 | 8192 |
| 229376 | 229376 | 16384 |
| 491520 | 491520 | 32768 |
| 10240 | 10240 | 1024 | dataset1.\* |
| 22528 | 22528 | 2048 |
| 49152 | 49152 | 4096 |
| 106496 | 106496 | 8192 |
| 229376 | 229376 | 16384 |
| 491520 | 491520 | 32768 |

Graphs:

1. Time Complexity v/s Dataset0
2. Time Complexity v/s Dataset1
3. Complexity Count v/s Dataset0
4. Complexity Count v/s Dataset1

Explanation:

* As seen from the results and graphs above, we can say that the Complexity Count i.e. number of comparisons for both type of Merge Sort – Top Down and Bottom Up are equal since we are counting comparisons in the “merge” operations.
* But, we can see that the time complexity in case of Merge Sort – Top Down approach is better than Bottom Up approach. The only reason is “recursion” which helps in caching the data in the memory while it creates virtual tree in it. Thus, cache locality helps improve time for Top Down approach as seen in graph and results of time complexity
* The only problem in Merge Sort – Top Down method is that the memory may get exhausted in case of higher values of N (i.e. millions of records) due to recursive calls and thus Bottom Up approach would help.
* The complexity for both types of Merge Sort is nearly same and it is **O (n) = nlogn i.e. linearithmic**.

**Solution 4:**

Results: Complexity Count for the Algorithm

|  |  |
| --- | --- |
| Complexity Count | Dataset |
| 8191 | 8192 |

Explanation:

* Since the dataset to be generated of 8192 elements is consecutive orders of 1, 11, 111 and 1111 which is already sorted, there are many sorting algorithms that have linear time complexity. I have implemented **Insertion Sort** which has complexity **O (n) = n** i.e. **linear** since dataset is already sorted.
* We can even implement Merge Sort and Bubble Sort which would have linear complexity but would need some improvements/addition to their standard algorithms.
* For Merge Sort, we can check condition of first/lowest element in second array with highest/largest element in first array before merging which would infer that the two sub-arrays are already sorted.
* For Bubble sort, we can add a flag which will be set to True or 1 every time the data is swapped and flag to False or 0 before entering the inner loop for comparison.
* Since, there is no modifications required for existing standard insertion sort algorithm, it can be used effectively to solve this given problem as it have **O (n) = n** i.e. **linear** complexity as seen in table above.

**Solution 5:**

**Part 1:** Performance Comparison between Merge Sort (both types), Quick Sort (Median of 3) and Quick Sort (Cutoff to Insertion Sort with N=7) for the datasets given

Results:

1. Complexity Count for Comparison between Merge Sort (both types), Quick Sort (Median of 3) and Quick Sort (Cutoff to Insertion Sort with N=7)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Quick Sort (Cutoff: N=7) | Quick Sort (Median of 3) | Merge Sort (Top Down) | Merge Sort  (Bottom Up) | Data Set | |
| Complexity Count | Complexity Count | Complexity Count | Complexity Count |
| 7567 | 7693 | 10240 | 10240 | 1024 | dataset0.\* |
| 17168 | 17422 | 22528 | 22528 | 2048 |
| 38417 | 38927 | 49152 | 49152 | 4096 |
| 85010 | 86032 | 106496 | 106496 | 8192 |
| 186387 | 188433 | 229376 | 229376 | 16384 |
| 405524 | 409618 | 491520 | 491520 | 32768 |
| 6747 | 5686 | 10240 | 10240 | 1024 | dataset1.\* |
| 14023 | 11903 | 22528 | 22528 | 2048 |
| 31125 | 26978 | 49152 | 49152 | 4096 |
| 67132 | 58748 | 106496 | 106496 | 8192 |
| 151849 | 134924 | 229376 | 229376 | 16384 |
| 320151 | 286896 | 491520 | 491520 | 32768 |

1. Time Complexity for Comparison between Merge Sort(both types), Quick Sort(Median of 3) and Quick Sort(Cutoff to Insertion Sort with N=7)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Quick Sort (Cutoff: N=7) | Quick Sort (Median of 3) | Merge Sort (Top Down) | Merge Sort (Bottom Up) | Data Set | |
| Time (ms) | Time (ms) | Time (ms) | Time (ms) |
| 4.2 | 5.59 | 15.87 | 16.36 | 1024 | dataset0.\* |
| 10.89 | 12.4 | 73.72 | 35.45 | 2048 |
| 43.12 | 48.62 | 83.58 | 123.78 | 4096 |
| 72.45 | 91.73 | 175.96 | 191.42 | 8192 |
| 144.65 | 183.23 | 343.58 | 338.64 | 16384 |
| 227.61 | 258.18 | 733.3 | 761.29 | 32768 |
| 7.14 | 7.11 | 21.02 | 17.56 | 1024 | dataset1.\* |
| 26.78 | 16.07 | 38.02 | 37.55 | 2048 |
| 49.17 | 33.91 | 102.44 | 130.77 | 4096 |
| 119.31 | 83.67 | 178.35 | 191.62 | 8192 |
| 203.89 | 177.65 | 382.42 | 386.32 | 16384 |
| 362.87 | 348.84 | 809.21 | 939.9 | 32768 |

Graphs:

1. Complexity Count v/s Dataset0 for Merge Sort(both types), Quick Sort(Median of 3) and Quick Sort(Cutoff to Insertion Sort with N=7)
2. Complexity Count v/s Dataset1 for Merge Sort(both types), Quick Sort(Median of 3) and Quick Sort(Cutoff to Insertion Sort with N=7)
3. Time Complexity v/s Dataset0 for Merge Sort (both types), Quick Sort(Median of 3) and Quick Sort(Cutoff to Insertion Sort with N=7)
4. Time Complexity v/s Dataset1 for Merge Sort(both types), Quick Sort(Median of 3) and Quick Sort(Cutoff to Insertion Sort with N=7)

Explanation:

From the results and graphs of Time Complexity (T (n)) or Complexity Count (O (n)), below are the inferences clearly drawn:

* In **dataset0.\*** since all the elements are already sorted, the time complexity or complexity count for merge sort (both types) are nearly similar as described in Solution3 too. While, quick sort with median of 3 method, performs better than merge sort. Moreover, quick sort with median of 3 along with Cutoff to Insertion Sort (N=7) performs little better than quick sort.
* The main reason why quick sort with cutoff to insertion sort behaves better than quick sort is because the elements are already sorted and insertion sort on sorted array takes linear time.
* In **dataset1.\*** since the elements are unordered/not sorted the performances vary for all the sorting algorithms. Merge Sort (both types) perform nearly same but quick sort with median of 3 performs better than merge sort. Also, quick sort with cutoff to insertion sort (N=7) behaves better than merge sort but a bit low in performance compared to quick sort with median of 3.
* The reason why quick sort with cutoff to insertion sort behaves bad in dataset1.\* i.e. unsorted array compared to quick sort with median of 3 is because insertion sort in unsorted array can take quadratic time in worst case. Thus, quick sort with median of 3 performs better and in each recursive call the median of 3 helps in placing the median in almost correct position and reducing overhead while partitioning and sorting.

**Part 2:** Performance Comparisons between Quick Sort (Median of 3) and Quick Sort (Cutoff to Insertion Sort for different values of N)

Results:

1. Cutoff Analysis for Different Values of Cutoff i.e. N for Dataset0.32768

|  |  |  |
| --- | --- | --- |
| dataset0.32768 | | |
| Time (ms) | Complexity Count | Quick Sort Category |
| 258.18 | 409618 | Quick Sort (Median of 3) |
| 319.14 | 417809 | Quick Sort (Cutoff to Insertion = 5) |
| **292.97** | **417809** | **Quick Sort (Cutoff to Insertion = 6)** |
| 227.61 | 405524 | Quick Sort (Cutoff to Insertion = 7) |
| 299.56 | 405520 | Quick Sort (Cutoff to Insertion = 8) |
| 335.06 | 405520 | Quick Sort (Cutoff to Insertion = 9) |
| 472.37 | 405520 | Quick Sort (Cutoff to Insertion = 10) |

1. Cutoff Analysis for Different Values of Cutoff i.e. N for Dataset1.32768

|  |  |  |
| --- | --- | --- |
| dataset1.32768 | | |
| Time (ms) | Complexity Count | Quick Sort Category |
| 348.84 | 286896 | Quick Sort (Median of 3) |
| 430.13 | 286896 | Quick Sort (Cutoff to Insertion = 1) |
| **416.94** | **291522** | **Quick Sort (Cutoff to Insertion = 2)** |
| 362.08 | 306126 | Quick Sort (Cutoff to Insertion = 4) |
| 487.59 | 315736 | Quick Sort (Cutoff to Insertion = 6) |
| 362.87 | 320151 | Quick Sort (Cutoff to Insertion = 7) |
| 713.04 | 324436 | Quick Sort (Cutoff to Insertion = 8) |

Explanation:

For analyzing the performance in quick sort with different values of cutoff, I have considered one dataset each for sorted and unsorted dataset i.e. dataset0.32768 and dataset1.32768. Also, I have not considered time complexity i.e. physical clock time into consideration because the time may change each time the program runs.

* For dataset0.32768 since the data is already sorted, the higher the cutoff value better would be the performance. The cutoff values N = 7,8,9 and so on perform better compared to the Quick Sort Median of 3 method. But for **N = 6** cutoff value, the performance reverts/degrades compared to Quick Sort median of 3 method as seen in table 1 above.
* For dataset1.32768 since the data is not sorted, the lower the cutoff value better would be the performance. Only cutoff value N=1 performs same as that of quick sort median of 3. While, the performance reverts/degrades compared to Quick Sort median of 3 when cutoff value is **N=2** and higher as seen in table 2 above.

**Solution 6:**

Explanation:

**Column 1:** Original Array

**Column 2:** This is the intermediate step of **Merge Sort (Bottom Up method)** as we can see that every 4 elements are sorted which is the result after sorting with size 2.

**Column 3:** This is the intermediate step of **Quicksort (standard, no shuffle)** as we can see that the first element “navy” is placed correctly at the position same as final position and elements to the left/above are lower than right/below of “navy”. Moreover, to differentiate between standard and 3-way quick sort, we can see that the next higher element to “navy” i.e. “plum” in original array is replaced with the smallest element from the right/end i.e. “mist” after the first iteration and so on.

**Column 4:** This is the intermediate step of **Knuth Shuffle** algorithm as we can see that all the elements before “silk” are shuffled while elements after it are still in place.

**Column 5:** This is the intermediate step of **Merge Sort (Top Down method)** as can see that the first 12 elements are sorted i.e. up to “teal” while the for the next 12 elements half of them are sorted i.e. up to “wine” and the next half is also sorted starting from “café”. Thus, next step would be merging and sorting the second half of the column i.e. merging two subarrays of 6 elements (sorted) each.

**Column 6:** This is the intermediate step of the **Insertion Sort** algorithm as we can see that up to “teal” the data is sorted and this is what Insertion sort does. In each iteration it sorts the earlier part of array i.e. the left sub array.

**Column 7:** This is the intermediate step of **Heap Sort** algorithm in max-heap form where the largest element “wine” is placed on top of tree/array as root and then followed by two children – “teal” and “silk” less than root and then further these children having their children – “plum”, “sage”, “pink” and “rose” less than them and so on.

**Column 8:** This is the intermediate step of **Selection Sort** as we can see that the top most element is the smallest element and this is what Selection sort does. In each iteration it finds the smallest array and sorts the array accordingly as visible until “mint”.

**Column 9:** This is intermediate step of **Quick Sort (3-way, no shuffle)** as we can see that “navy” selected as pivot is placed at correct position with respect to final array and also we can see that during partitioning while it traverses from top/left to bottom/right, it places the elements greater than pivot to the end of array and move the end pointer above 1. Similarly, it swaps pivot with the smaller elements to the top/left of it. This is clearly visible that the elements greater than “navy” – “plum”, ”pink”, ”rose” are moved to the bottom of array in the order of its encounter while moving from left/top to right/bottom of array. Also, “mist”, ”coal”, ”jade”, etc. are swapped with navy while traversing.

**Column 10:** Sorted Array