**Assignment 2**

**Akhil Patil | ap1604**

**Q1 ->**

**Table1 : Number of Comparisons – data0.\* (Sorted Dataset)**

|  |  |  |
| --- | --- | --- |
| **Dataset** | **Shell Sort** | **Insertion 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 |

**Table2: Number of Comparisons – data1.\* (Unsorted Dataset)**

|  |  |  |
| --- | --- | --- |
| **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 |

**Analysis:**

For the sorted datasets, it is the best case for insertion sort where it makes N-1 comparisons and 0 exchanges. But in the case of shell sort it must go through the whole array for different values of h. when h is 1 shell sort scans the whole array where it has to make N-1 comparisons. From the graph we can see that shell sort requires almost more double number of comparisons than Insertion sort. So, in sorted dataset case Insertion sort performs better than shell sort.

For the Unsorted datasets, shell short out performs insertion sort because it avoids the long-distance sorting when h=1, for the higher vales of h, long distance disorder is fixed, and array becomes partially sorted. While for Insertion sort it has scan through the whole array and place the element at correct place in sort half by comparing it to each element in sorted half array. So, the number of comparisons required for insertion sort are way more than shell sort. In this case shell sort performs better than insertion sort.

**Q2 ->**

**Table3 : Running Time-Data Size**

|  |  |
| --- | --- |
| **Dataset** | **Running Time(μs)** |
| 1024 | 360 |
| 2048 | 489 |
| 4096 | 738 |
| 8192 | 1542 |
| 16384 | 3549 |
| 32768 | 7182 |

**Table4 : 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 |

**Analysis:**

To count number of inversions in an array I used recursive merge sort to sort and count the number of inversions with the cost of O(NlogN). For getting the number of inversions we first divide the array in two parts left and right and compare elements in right and left subarrays. When array[left] > array[right] , then there are (mid – i+1) pairs of inversions.

From the plot we can interpret that running time and size of datasets are directly related. i.e. increase in dataset size increases the time like merge sort . Also, we can say that as number of inversion pairs increase , time also increases. Time complexity for this algorithm is O(NlogN).

**Q3 ->**

For this question, I have implemented the counting sort method to sort the numbers of the array. The regular counting sort method sorts the elements at the cost of O(n +r). n is the size of array and r is the largest number in the array.

But in this scenario, we have advance knowledge of the elements of the array . So, I created a enhanced version of counting sort which first calculates the number of times the elements are repeated like 1 is repeated for 1024 times and stores it in a temp array. Then we create the array from the smallest element to the size which we counted, and another number appends the array for the count of occurrences.

This implementation costs O(N) which is same as the bubble and insertion sort. So, For this question I think Bubble sort, insertion sort and enhanced version of counting sort are the efficient algorithms for sorting.

**Q4 - >**

**Table5 : Number of Comparisons(UB and BU) – Sorted Dataset**

|  |  |  |
| --- | --- | --- |
| **Datasets** | **Number of Comparisons(UB)** | **Number of Comparisons(BU)** |
| data0.1024 | 25600 | 25600 |
| data0.2048 | 56320 | 56320 |
| data0.4096 | 122880 | 122880 |
| data0.8192 | 266240 | 266240 |
| data0.16384 | 573440 | 573440 |
| data0.32768 | 1228800 | 1228800 |

**Table6 : Number of Comparisons(UB and BU) – Unsorted Dataset**

|  |  |  |
| --- | --- | --- |
| **Datasets** | **Number of Comparisons(UB)** | **Number of Comparisons(BU)** |
| data1.1024 | 44770 | 25600 |
| data1.2048 | 99670 | 56320 |
| data1.4096 | 219720 | 122880 |
| data1.8192 | 480370 | 266240 |
| data1.16384 | 1043475 | 573440 |
| data1.32768 | 2250660 | 1228800 |

**Analysis:**

From the Table5 we can say that the number of comparisons in Best case i.e. for sorted dataset required by both the implementations of merge sort one with recursive(UB) and the iterative methods(BU) are same with time complexity of O(1/2 \* NlogN)

For unsorted datasets, from the table6 we can observe that Bottom Up approach requires the same amount of comparisons as for the sorted arrays because the size is same as for the datasets. While in case of Top bottom recursive approach requires more comparisons than that of sorted dataset. Here the complexity of both is O(NlogN).

From the above results we can say that both the approaches top-bottom and bottom-Up performs same for best case and Bottom up implementation of merge sort performs better in Average case than top-bottom approach.

**Q5 ->**

**Table7 - Number of Comparisons—both datasets**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **dataset** | **merge sort(UB)** | **merge sort(BU)** | **quick sort** | **Quick sort cutoff=3** | **quick sort cutoff=7** | **quick sort cutoff=64** |
| data0.1024 | 25600 | 25600 | 7692 | 7948 | 7563 | 5048 |
| data0.2048 | 56320 | 56320 | 17421 | 17933 | 17164 | 12137 |
| data0.4096 | 122880 | 122880 | 38926 | 39950 | 38413 | 28362 |
| data0.8192 | 266240 | 266240 | 86031 | 88079 | 85006 | 64907 |
| data0.16384 | 573440 | 573440 | 188432 | 192528 | 186383 | 146188 |
| data0.32768 | 1228800 | 1228800 | 409617 | 417809 | 405520 | 325133 |
| data1.1024 | 44770 | 25600 | 6486 | 6764 | 7042 | 15231 |
| data1.2048 | 99670 | 56320 | 13577 | 14151 | 14789 | 29864 |
| data1.4096 | 219720 | 122880 | 30473 | 31647 | 32945 | 66439 |
| data1.8192 | 480370 | 266240 | 67101 | 68956 | 71895 | 140654 |
| data1.16384 | 1043475 | 573440 | 152514 | 157224 | 162165 | 293073 |
| data1.32768 | 2250660 | 1228800 | 324659 | 333953 | 344084 | 601975 |

**Table8 - (Running Time—Both Dataset)**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **dataset** | **merge sort(UB)** | **merge sort(BU)** | **quick sort** | **Quick sort cutoff=3** | **quick sort cutoff=7** | **quick sort cutoff=64** |
| data0.1024 | 229 | 235 | 120 | 90 | 76 | 43 |
| data0.2048 | 194 | 228 | 128 | 130 | 178 | 80 |
| data0.4096 | 354 | 736 | 228 | 178 | 237 | 313 |
| data0.8192 | 597 | 703 | 454 | 494 | 464 | 330 |
| data0.16384 | 1076 | 1193 | 951 | 572 | 889 | 811 |
| data0.32768 | 2131 | 2795 | 1612 | 1085 | 1938 | 1462 |
| data1.1024 | 395 | 421 | 120 | 194 | 195 | 90 |
| data1.2048 | 759 | 360 | 232 | 437 | 253 | 195 |
| data1.4096 | 1344 | 710 | 538 | 583 | 559 | 489 |
| data1.8192 | 1405 | 1322 | 1122 | 1117 | 1148 | 986 |
| data1.16384 | 2884 | 3076 | 2284 | 2199 | 2963 | 2151 |
| data1.32768 | 6263 | 6376 | 4910 | 4715 | 5495 | 4041 |

**Analysis:**

To compare merge sort and quick sort I am using number of comparisons made in the execution and the time of execution. Now since the dataset of data0 is sorted and data1 is unsorted , but it is not the worst case of descending, so I am avoiding the shuffling of array in quicksort.

From Table7 , we can see that quick sort has a smaller number of comparisons made to sort both the datasets. Also, from table8 we can say that the running time is less than merge sort. So, overall quick sort performs better based on time and number if comparisons.

For sorted dataset insertion sort performs better in smaller sub arrays while quick sort calls recursively for sorting in small arrays. So, when cutoff value is set , the algorithm does better. Insertion takes O(n) for the sorted array while quick sort takes O(nlogn). So the insertion sort performs better at data0 series. So, when the cutoff value is set to 7, algorithm performs better than without cutoff because of the use of insertion sort for smaller arrays. And as we increase the value of cutoff value like in this case, I have done cutoff value of 64 which gets the best results among all other algorithms in every case. So, Higher cutoff value gets better results.