Kyle Wolf

PSU ID#: 6398

Problem Set #5 Part 2

*Hypothesis:* If given randomly ordered input of size 30, then most of the sorts will finish very quickly except Bogo Sort, because the expected running time of Bogo Sort is O(n!), while all of the other sorts run in polynomial time.

*Input File:* Input with 30 values to be sorted, in a random order.

*Results:*

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Time(s) in ms | 2971 | Unfinished | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |

*Conclusions*: Since sort #2 had to be killed before it was able to complete, and all of the others completed quickly, I conclude that sort #2 is Bogo Sort.

*Hypothesis:* If given an input of size 10 with multiple duplicate inputs with different string values, if the order of these strings are switched in the sort, then the sort is guaranteed to not be stable because they otherwise would not be switched.

*Input File:* Input with 10 values to be sorted with duplicate keys with different string values.

*Results:*

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sort # | 1 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| Stable? | Yes | No | No | Yes | Yes | No | Yes | No | Yes | No |

*Conclusions*: Sort 3, 4, 7, 9, and 11 are unstable. So, these could be Heap, Quick, Randomized Quick, Selection, or Stooge Sort. The remaining tests would then be the other sorts.

*Hypothesis:* While the noticeably poorer runtime of sort #1 in the very first test might indicate it to be Stooge Sort that does not align with the unstable check done before, as then sort #1 should be unstable. So, if I give another randomly ordered input of size 30, but bound the possible values to be no bigger than 10, if sort #1 finishes much quicker, then it must be LSD Radix Sort. This is because its runtime is in part based off the maximum value in the array, which was quite large before.

*Input File:* Input with 30 values to be sorted bounded by 10, in a random order.

*Results:*

|  |  |
| --- | --- |
| Sort # | 1 |
| Time(s) in ms | 0 ms |

*Conclusions*: Since sort #1 has such a short runtime now, I conclude that sort #1 is LSD Radix Sort.

*Hypothesis:* From the remaining unstable sorts, 3,4,7,9, and 11, among them is QuickSort and Randomized QuickSort. Randomized Quicksort is designed to improve time complexity on sorting inputs that are already sorted, which slows QuickSort. Testing with two large inputs, one randomly ordered and one already sorted, if there is a noticeable discrepancy, then this should allow me to identify QuickSort because QuickSort’s runtime will increase for a sorted input.

*Input File:* Input with 10000 values to be sorted, in a random order. Also, input with 10000 values to be sorted, in sorted order.

*Results:*

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Sort # | 3 | 4 | 7 | 9 | 11 |
| Time(s) in ms (Random) | 4 | 6 | 4 | Unfinished | 121 |
| Time(s) in ms  (Sorted) | 181 | 4 | 4 | Unfinished | 89 |

*Conclusions*: Since sort #3 was significantly slowed by an already sorted input compared to a random input, I conclude sort #3 is QuickSort.

*Hypothesis:* Part of the same test can be run to find Stooge Sort. If one sort runs significantly slower for the randomly sorted inputs, then it is Stooge Sort because Stooge Sort has a worst case runtime O(nlog3/log1.5).

*Input File:* Input with 10000 values to be sorted, in a random order.

*Results:*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sort # | 4 | 7 | 9 | 11 |
| Time(s) in ms | 6 | 4 | Unfinished | 121 |

*Conclusions*: Since sort #9 was significantly slower to the point where it didn’t finish, I conclude it is Stooge Sort.

*Hypothesis:* Again, we can use the same test and use runtimes to find Selection Sort. If one sort runs significantly slower for the randomly sorted inputs, but faster than what was found to be Stooge Sort, then it is Selection Sort because Selection Sort has an average case runtime O(n2). Both QuickSorts have the same worst-case runtimes, but if similar runtimes do not appear we can assume they are adhering closer to their average case runtimes.

*Input File:* Input with 10000 values to be sorted, in a random order.

*Results:*

|  |  |  |  |
| --- | --- | --- | --- |
| Sort # | 4 | 7 | 11 |
| Time(s) in ms | 6 | 4 | 121 |

*Conclusions*: Since sort #11 was significantly slower, I conclude it is Selection Sort.

*Hypothesis:* Left with sort #4 and #7, one of which must be Heap Sort, the other Randomized QuickSort. Randomized Quick Sort has the same average case runtime as Heap Sort’s, but a worse worst-case runtime. If forced to sort an input of nothing but duplicates, then Randomized QuickSort will have the worse runtime, because the chosen pivot will be completely unhelpful and force a worst-case runtime.

*Input File:* Input with 10000 values to be sorted, all duplicates.

*Results:*

|  |  |  |
| --- | --- | --- |
| Sort # | 4 | 7 |
| Time(s) in ms | 1 | 85 |

*Conclusions*: Since sort #7 was significantly slower, I conclude it is Randomized QuickSort. I can also therefore conclude that sort #4 is Heap Sort.

*Hypothesis:* Now turn to the undiscovered stable sorts, #5, #6, #8, and #10. If there are two sorts that take significantly longer among these remaining sorts with a large, randomly sorted input, then I might assume that these two are Insertion Sort and Bubble Sort, because Insertion Sort and Bubble Sort both boast the largest worst case runtimes of O(n2). To confirm this, in the best case (an already sorted input), Bubble and Insertion Sort should perform better than Merge and Counting Sort as the two have best cases of O(n) – better than those of Merge and Counting Sort.

*Input File:* Input with 100000 values to be sorted, randomly ordered. Then, input with 100000 values to be sorted, in sorted order.

*Results:*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Sort # | 5 | 6 | 8 | 10 |
| Time(s) in ms  (Random) | 55 | Unfinished | Unfinished | 72 |
| Time(s) in ms  (Sorted) | 52 | 5 | 4 | 59 |

*Conclusions*: Since sort #6 and #8 did not finish with a randomly sorted input but sorted the fastest with an already sorted input, I conclude that these two are Insertion and Bubble Sort. Conversely, sort #5 and #10 are Counting and Merge Sort.

*Hypothesis:* If sort #6 is significantly faster than sort #8 then it is Insertion Sort, because Insertion Sort typically is faster than Bubble Sort. Else, the opposite is true.

*Input File:* Input with 10000 values to be sorted, in random order.

*Results:*

|  |  |  |
| --- | --- | --- |
| Sort # | 6 | 8 |
| Total Heap Space | 287 | 99 |

*Conclusions*: Sort #8 is faster, so I conclude sort #8 is Insertion Sort, and sort #6 is Bubble Sort.

*Hypothesis:* If sort #5, using an input where the range of values is n2 or greater, is slower than sort #10, then it is Counting Sort, because the time complexity of Counting Sort is closer to O(n2) in that case whereas Merge Sort is O(nlogn). Else, the opposite is true.

*Input File:* Input with 10000 values to be sorted, with a bound of 100000000, in random order.

*Results:*

|  |  |  |
| --- | --- | --- |
| Sort # | 5 | 10 |
| Total Heap Space | 9 | 12 |

*Conclusions*: Since sort #10 is slower, I conclude it is Counting Sort. Also, sort #5 must then be Merge Sort.

Sort 1 – LSD Radix Sort

Sort 2 – Bogo Sort

Sort 3 – QuickSort

Sort 4 – Heap Sort

Sort 5 – Merge Sort

Sort 6 – Bubble Sort

Sort 7 – Randomized QuickSort

Sort 8 – Insertion Sort

Sort 9 – Stooge Sort

Sort 10 – Counting Sort

Sort 11 – Selection Sort