Gray Houston
ghousto
Project 2

Analysis Questions

1. Both the best case and the worst case running time is O(nd). To start, the inner loop will execute in O(d) time, as the program loops over the next d elements to find the smallest element to insert. Next, the outer loop will iterate n times and therefore run in O(n) time. Because the inner loop will run proportionally to the outer loop, the overall running time is O(nd). Because the inner loop will always run in O(d) time (and can never run for shorter), both the best case and the worst case will be O(nd) time.

2. The best and worst case running time is O(nd logd). To start, the creation of each small min-heap will take log(d) because it uses the bottom-up heapify method and there will be d heaps created. Therefore, creating all of the heaps will run in O(d logd). Deleting each element from a heap will run in O(n) time. Therefore, the algorithm will run at O(nd logd) time. This implementation runs in this time because it is in-place.

3. If T(n) is the time that it takes for my implementation of Mergesort to run, then

$$
\begin{aligned}
T(n) &= 2 * T(n/2) + K \qquad \text{where K is some constant locality value} \\
&= 2^2 * T(n/2^2) + K(1 + 2) \\
&= 2^3 * T(n/2^3) + K(1 + 2 + 2^2) \\
\ldots &= 2^K * T(n/2^K) + K(1 + 2 + 2^2 + \ldots + 2^{K-1}) \\
&= 2^K * T(n/2^K) + K * n \\
&= K * n
\end{aligned}
$$

   Therefore, when d is set to a constant value, my Mergesort will have an asymptotic O(n) running time.

4. As the data below shows, as the data sizes increase from 10^3 to 10^6, Lmerge grows the slowest, Lselection is just slightly slower, and Lheap is noticeably slower. These results confirm the asymptotic performance of the algorithms. Lmerge runs in O(n logd) time, Lselection runs in O(nd) time, and Lheap runs in O(nd logd). O(n) is slightly less than O(nd) for small levels of d. Therefore, it makes sense that Lselection is close to the speed of Lmerge. Furthermore, O(nd logd) greatly less than both of the other two asymptotic times. Therefore, it makes sense that it runs the slowest in the data.

   Furthermore, that data of 10^6 as d increases confirms the asymptotic performances of the locality aware algorithms. For **Lselection**, running time significantly increases as d increases because its asymptotic performance is O(nd). Linear growth with d would cause huge increases in running time for this algorithm. For **Lmerge**, the increase in running time from an increase in d decreases as d gets larger. Because this algorithm theoretically runs in O(n logd), the running time of this function should grow logarithmically with d. Even though the data for Lmerge is not the best, the data still seems to support this growth function. Finally, **Lheap** should grow linearithmically as d grows because it runs in O(nd logd). However, the data doesn't support this model. However, this could be explained by random variation. It is noticeable that both Lmerge and Lheap have abnormal results for 10^6 data. The data likely doesn't accurately represent the asymptotic growth because of the small number of samples taken.

5. As the data shows, in regards to Heapsort, for all 5 values of d, Lheap run about 5 times faster than the normal Heap. This is likely because Lheap grows proportional to O(d logd) as d increases and Heap is independent of changes in d. Because O(d logd) is such a small value of d compared with values of n in 10^6 range, there would be practically no change in performance for either sorting algorithm as d changes.

   In regards to Mergesort, excluding the first locality value, the data shows increases in d don't greatly change the running time of Lmerge but it does increase the running time of the normal Merge. This increase for Merge is likely because as d increases, the number of comparisons that occur during the merge increases, causing the running time to increase. However, in Lmerge uses my own "Relevant Range" algorithm (which I describe in the code), it utilizes the locality to determine the relevant range of elements that actually needs to be merged. This way, changes in d don't greatly affect the running time.

   In regards to Selection Sort, the data shows Selection is affected by the changes in d while increases in d cause direct increases in Lselection's running time. This makes sense because Selection will always run $O(n^2)$ comparisons, regardless of the locality of the data. However, because Lselection runs asymptotically to O(nd), every increase in d would cause a linear increase in the running time of Lselection.

   Furthermore, the data confirms that Quicksort will run on average at about O(n logn) time. It's average running times are above the guaranteed O(n logn) of Mergesort (likely due to the implementation) and the running times are greatly varied, as the running time depends on the selection of the pivot.

6. For an array of size n, when d=0, that means that each element is at most 0 indexes away from its final position. When there are 0 indices between an element and its final position, it means that the element must be in its final position. Therefore, d=0 for an array means that the array is already sorted.

7. Of all of the locality sorting algorithms implemented, the only stable sort was LMerge. First, selection sort's and heapsort's original algorithms were not stable, so their locality-based implementations would not have been. Mergesort remained stable because the only difference between the locality version and the normal version were the bounds used for the merge. Because changing the bounds of the merge doesn't impact stability, the locality Mergesort is stable.

8. We were not asked to implement a locality version of Insertion Sort because it is already locality aware. As the element being sorted moves left into the sorted region, it moves closer and closer to its final destination. If there is some d such that d >= the maximum distance between an element and its final position, because Insertion Sort already moves elements to the left towards their final positions, Insertion Sort will never iterate a single element more than d times. Therefore, Insertion Sort already handles the case of locality.

9. Even if data with locality d significantly less than n was used with bubble sort, it would still run in $O(n^2)$ time. Because the array is already partially sorted, there would likely just be less exchanges. However, the number of comparisons will remain the same.

10. The method that could be used to generate data that isn't sorted is quicksort because of its partition. Each partition is a logical separation of the data. Furthermore, the final position of each element within the partition is within the partition. Therefore, the locality of a set of a data is the size of the partition. Using this information, you could take any array, even if it isn't sorted, and form a non-sorted array that has the locality condition.

    To create the array with a specified locality, call quicksort as normal until a partition size is less than or equal to the locality parameter. Quicksort wouldn't be called on this partition (of size <= d). This would act as a base case in the recursion. Because the locality of an element in the quicksort algorithm is the size of its partition, stopping when partitions are less than d would create data that possesses the locality condition.

Data

Question 4

Lselection

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^3	5	2.4314	1.5782	1.798	1.8506	1.8293	1.8975
10^3	15	2.0637	2.6698	2.6326	2.6277	2.2228	2.44332
10^3	25	2.9506	2.6295	2.8226	2.8795	3.1451	2.88546
10^3	35	3.0663	3.4888	2.6835	3.3355	2.9465	3.10412
10^3	45	3.9527	3.8219	3.1322	3.7492	3.7468	3.68056
10^5	5	12.283	11.6393	10.7387	10.5307	11.9092	11.42018
10^5	15	14.4971	14.3714	14.4162	14.4467	20.5195	15.65018
10^5	25	16.7368	15.8205	17.0148	15.9904	16.9688	16.50626
10^5	35	19.1087	19.8543	19.0878	19.2599	19.2743	19.317
10^5	45	21.8598	21.7228	21.8714	22.8612	21.9722	22.05748
10^6	5	43.4964	29.0851	34.3877	35.5694	36.2339	35.7545
10^6	15	64.1366	61.9486	63.8755	62.1407	62.3913	62.89854
10^6	25	96.0541	124.8162	102.1384	91.0496	91.1718	101.04602
10^6	35	162.435	149.7188	156.5328	149.5086	149.4877	153.53658
10^6	45	239.7399	187.1239	239.0426	229.4462	196.1264	218.2958

Quick

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^3	5	2.0945	2.3385	7.7524	3.5066	2.6399	3.66638
10^3	15	3.2563	3.4076	2.0055	2.9561	5.122	3.3495
10^3	25	1.9542	2.2157	2.3236	4.2346	3.309	2.80742
10^3	35	3.2481	2.4975	2.2331	4.9037	2.3964	3.05576
10^3	45	2.3765	2.5339	2.5282	1.9404	2.1663	2.30906
10^5	5	38.9786	34.7101	34.969	28.3453	32.372	33.875
10^5	15	32.5535	20.4418	37.5928	25.7454	22.826	27.8319
10^5	25	29.8504	33.5623	30.952	44.5501	25.6562	32.9142
10^5	35	37.1164	46.3867	32.5334	32.0246	43.0306	38.21834
10^5	45	33.4047	29.6223	27.3228	38.6458	34.4754	32.6942
10^6	5	110.7811	132.6342	77.063	82.7869	84.8106	97.61516
10^6	15	83.0275	106.0832	82.372	100.1096	81.8074	90.67994
10^6	25	92.3026	80.1889	89.4204	109.6833	95.0349	93.32602
10^6	35	116.9366	96.2967	90.4509	108.4123	92.3925	100.8978
10^6	45	87.4839	90.807	114.1022	106.8524	104.7672	100.80254

Lmerge

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^3	5	1.2636	1.1975	1.1751	1.6125	1.6156	1.37286
10^3	15	1.7843	1.6933	2.1744	2.0701	1.9385	1.93212
10^3	25	2.0029	1.4796	2.0764	2.4884	2.7719	2.16384
10^3	35	1.9885	2.0471	2.212	2.2777	1.9234	2.08974
10^3	45	2.3785	4.3236	4.6692	2.4235	2.6933	3.29762
10^5	5	7.4659	7.9975	10.7647	9.8089	7.2928	8.66596
10^5	15	12.9017	23.6662	11.1453	12.2908	11.387	14.2782
10^5	25	12.3269	15.7397	11.2833	12.9204	23.5155	15.15716
10^5	35	13.1536	11.5482	15.2546	16.7216	13.7473	14.08506
10^5	45	13.898	19.6708	14.7297	15.4244	15.6842	15.88142
10^6	5	31.9828	28.0445	28.032	33.9435	37.7059	31.94174
10^6	15	56.1221	73.826	70.8796	69.5951	47.6862	63.6218
10^6	25	88.8719	65.3468	76.4642	61.2427	83.1351	75.01214
10^6	35	102.6382	98.8838	93.8551	114.6293	100.563	102.11388
10^6	45	90.937	86.2921	69.0651	101.936	80.5708	85.7602

Lheap

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^3	5	3.5413	2.4062	2.8249	2.5719	2.6633	2.80152
10^3	15	2.5358	2.7487	3.4211	2.9359	2.4518	2.81866
10^3	25	2.1591	2.3283	2.6955	1.9449	1.9035	2.20626
10^3	35	3.615	2.8936	3.921	2.1751	3.085	3.13794
10^3	45	2.4223	2.2099	2.3048	2.3007	2.3252	2.31258
10^5	5	21.8162	24.9235	17.9624	31.5759	20.1982	23.29524
10^5	15	19.7322	21.3543	22.6377	26.5269	29.6791	23.98604
10^5	25	17.7651	23.507	21.5047	26.3493	21.1318	22.05158
10^5	35	21.3205	31.2981	17.4573	20.256	20.6867	22.20372
10^5	45	14.6861	26.4941	23.3274	27.9988	21.93	22.88728
10^6	5	50.4596	70.1708	77.7549	55.9615	72.4877	65.3669
10^6	15	86.4977	95.0508	74.4402	78.3961	80.2962	82.9362
10^6	25	87.5956	94.3287	99.5553	84.8543	81.1195	89.49068
10^6	35	95.713	74.3697	80.8941	66.9856	65.0122	76.59492
10^6	45	62.1169	69.9989	57.604	61.0158	59.6741	62.08194

Question 5

Heap

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^6	5	218.5874	237.1036	262.6604	304.945	288.3105	262.32138
10^6	15	247.4676	241.7818	282.0429	287.5211	280.0997	267.78262
10^6	25	258.5698	327.7222	240.2086	186.2124	244.4883	251.44026
10^6	35	281.6952	218.2441	189.988	181.0077	226.2678	219.44056
10^6	45	242.5596	204.4353	250.6984	207.3909	220.7575	225.16834

Lheap

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^6	5	58.3833	58.0385	61.2857	50.0106	52.778	56.09922
10^6	15	60.4401	61.9961	58.982	68.1868	60.136	61.9482
10^6	25	52.41	61.5825	59.3773	59.8253	59.3378	58.50658
10^6	35	72.1669	62.4481	63.5646	62.2402	59.646	64.01316
10^6	45	60.3182	59.7293	75.0318	57.4313	52.8256	61.06724

Merge

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^6	5	48.2363	48.8416	45.9869	57.7066	43.5008	48.85444
10^6	15	69.0863	73.7313	67.5788	71.0158	69.2211	70.12666
10^6	25	94.0812	94.7049	99.0958	115.4012	101.3646	100.92954
10^6	35	96.3237	117.6681	93.0315	124.638	114.3348	109.19922
10^6	45	94.0236	116.4133	145.4479	178.6501	167.4026	140.3875

Lmerge

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^6	5	42.0729	38.289	35.2284	26.8566	49.5891	38.4072
10^6	15	71.0455	87.387	76.6613	73.7254	66.0112	74.96608
10^6	25	87.7832	81.8738	65.6178	79.0992	66.8582	76.24644
10^6	35	80.2349	80.8973	69.4187	85.5731	75.4049	78.30578
10^6	45	94.7354	77.4757	78.2067	105.5224	73.0546	85.79896

Quick

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^6	5	82.4484	83.8157	85.6472	81.2236	75.6609	81.75916
10^6	15	93.0316	104.5385	122.9876	117.995	136.6053	115.0316
10^6	25	142.4112	135.774	114.8235	125.0608	119.5602	127.52594
10^6	35	141.8662	117.7044	115.6224	123.1255	132.5758	126.17886
10^6	45	220.0304	127.0301	130.7272	137.0135	147.7819	152.51662

selection

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^5	5	6655.066	6689.243	6777.34	6878.974	6713.04	6742.73258
10^5	15	6688.241	6725.019	6751.622	6676.048	6689.611	6706.10812
10^5	25	6674.864	6665.388	6672.132	6663.848	6689.29	6673.10442
10^5	35	6654.788	6671.552	6771.809	6686.777	6678.152	6692.61562
10^5	45	6871.619	6736.916	6825.618	6661.026	6712.642	6761.56422

Lselection

Size	Locality	First Run	Second Run	Third Run	Fourth Run	Fifth Run	Average Run Time
10^5	5	11.2956	18.7959	11.2766	11.5729	10.7045	12.7291
10^5	15	19.7304	13.977	18.6652	13.0688	18.9097	16.87022
10^5	25	20.9076	15.6906	15.9479	17.1019	15.781	17.0858
10^5	35	19.7132	21.7881	19.127	19.1344	20.179	19.98834
10^5	45	21.429	21.4612	22.6601	21.3879	21.4025	21.66814