Alba Ballarà Sala

Camilla Bocciolone

Assignment 3

Ex 2

We want to test how well our hash function works so we conduct an experiment where we add to our hash table 5000 different vehicles. The size of the table is 10007 because it is recommended to use the closest prime number to the double of the elements you want to add.

In our class, the vehicles have four attributes: plate number, year of manufacture, color and brand. We do two experiments, in the first one, the vehicles we add have very similar information, only changing the plate number and all the other attributes are the same. In the second one, all the attributes are random for every vehicle we add. the color is chosen randomly form this list: `("blue", "green", "red", "black", "gray", "white")` and the brand form this one: `("volvo", "audi", "mercedes", "tesla")`. The year is a random one form 2000 to 2023 and the plate number is random from 1000 to 9999. Result of both experiments:

EXPERIMENT 1: Only changing plate number

Average number of collisions: 1

EXPERIMENT 2: Different attributes

Average number of collisions: 0

As we can see, only one collision occurs in the first experiment and 0 in the second one as an average. This means that our hash function is effective because very few collisions occur, which means that very rarely the result given by the hash function points to the same slot. We produce hash codes that are evenly distributed across the hash table.

Ex 5

We conducted an experiment to see which one is the recommended depth to swap from quicksort to insert sort or heapsort in arrays of sizes 100000, 200000, 400000, 600000, 800000, 1000000. We created an array of depths with the values 10, 12, 15, 17, 20, 22, 25, 27, 30, 32 that indicate the depths at which we will stop sorting the array with quick Sort and swap to both Heap Sort and Insert Sort. For each size we

used 10 random arrays to be sure the results will not depend on the input but will be an average representation of the situation when swapping at the given depth. We obtain the following results:

size	Depth 10	Depth 12	Depth 15	Depth 17	Depth 20	Depth 22	Depth 25	Depth 27	Depth 30	Depth 32
10^5	1678293	1134272	1263394	1166345	1179057	1212881	1159260	1155976	1139179	1153036
10^5	1170331	1122237	1127619	1153604	1199226	1312803	1110066	1116589	1113388	1095027
2*10^5	5652618	3165559	2326085	2836126	2718369	2788819	2730901	2748986	2816442	2731993
2*10^5	2439501	2412107	2389988	2589734	2732036	2627660	2705216	2674342	2609728	2587650
4*10^5	15764658	7922911	4986128	4783161	5020246	5145358	5518597	5179951	5199394	5165056
4*10^5	5149720	5299382	4926748	4925714	4969806	5305786	5267770	5049561	4986197	4994727
6*10^5	32265127,	15384502,	8921291,	7432245,	7572165,	7892398,	8091491,	8673974,	8008971	8178348
6*10^5	8883548,	8568935,	8010050,	7583515	7680043,	8424131,	8104313,	7713505,	8247597,	8245306
8*10^5	76420954,	31533709,	13026696,	11108999	10131183,	10761666,	11277742	11620070	12463332,	10887917
8*10^5	11503642,	10679335,	10411214,	10478151,	10431517,	11412134,	10558215	12187255,	11339858,	10962234
10^6	240285292,	100528015,	31287326,	17974718,	12860445,	12848365,	13204455,	13345874	13616782,	13444953
10^6	13747061,	13683646,	13155348,	13613804,	13372538,	12873035,	13621443,	12852808	13361613,	13041256

where the results are reported in nanoseconds and refer to time taken in average to sort the array of the size reported in the first column swapping to insert sort if yellow row or heap sort if white row at the depth reported on the upper row.

We can observe that as the depth increases, the time taken to sort the array decreases initially because Quicksort is supposed to be a quicker sorting algorithm than insert sort or heap sort.

However, as the depth continues to increase, there is some points different for each size where the Quicksort's recursion and partitioning results not so good in terms of performance. At this depths, switching to Insertsort or heap sort for the smaller

subarrays gives a better performance in fact the subarrays are so small that they can be simply sorted by heapsort and insertsort.

In general we can say that the swapping point looks similar for both heapsort and insertsort. The swapping point increases with the size and this helps us also in some cases in which the swapping point is not so evident. We could say that the swapping point is: 12 for size 100000, 15 for size 200000, 17 for size 400000, around 19 for size 600000, 21 for size 800000 and 23 for size 1000000. Finally by confronting the time taken by the sortIS and sortHS swapping to insert sort is generally better.

Ex 6

The time complexity for the sort and iter_sort methods should be both O(n log n), where "n" is the number of elements in the array.

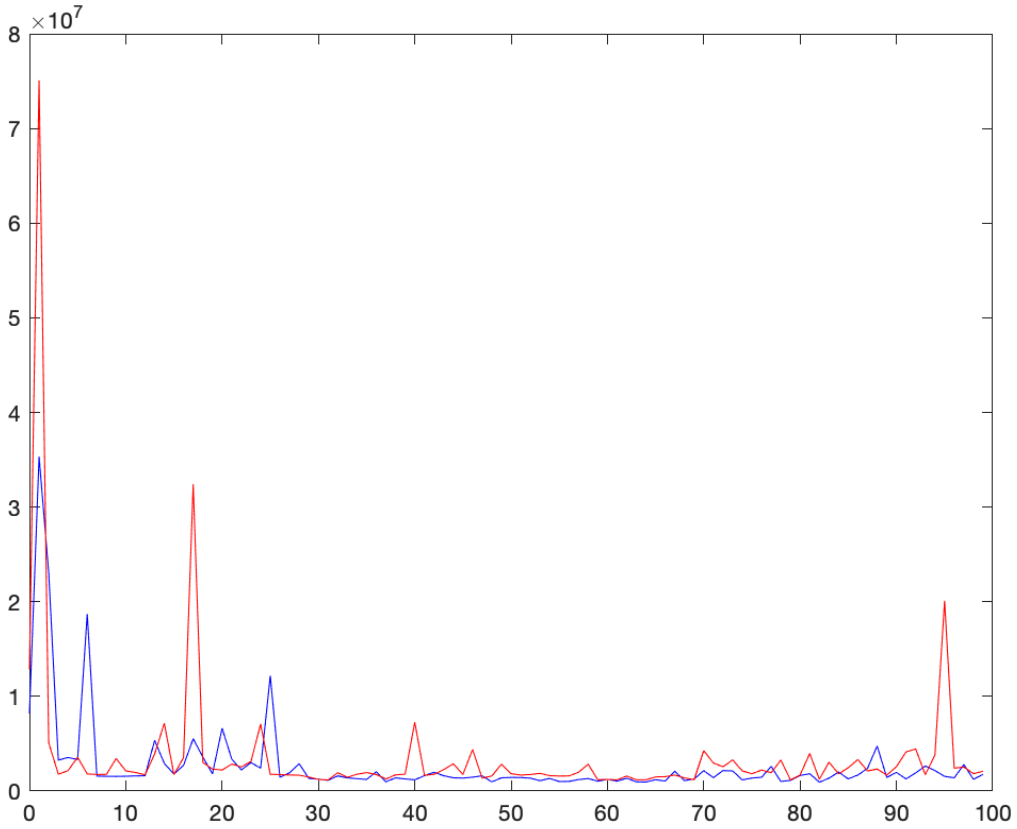
In fact both the recursive sort method and the iterative iter_sort method follow the same algorithm, you divide the array into smaller arrays until you reach arrays of size 1 (base case). This division is done in a way that each level of the recursion (in the recursive method) or each iteration (in the iterative method) divides the array into halves.

The merge step is O(n) at each level of recursion or iteration, and since there are log(n) iterations (because you are dividing the array in half each time), the overall time complexity for both methods remains O(n log n).

So since the algorithm is basically the same just written in a different way the time complexity should be the same and also the time taken by the two sorting to sort 100000 arrays of size 10000.

From our code we find in fact that the time taken to sort 100000 arrays of size 10000 is pretty similar but it is higher for the recursive call . We could explain that by saying that it is due to recursive overhead: The recursive merge sort implementation may introduce some additional overhead due to function calls and maintaining a call stack.

We notice as well some jumps and little differences that could be due to background processes, system load, and resource availability.



where red is recursive time and blue is iterative time and both are reported in nanoseconds

yiter=[8196071, 35324427, 23097155, 3269261, 3548629, 3334888, 18694121, 1582436, 1559601, 1550770, 1566276, 1601533, 1614799, 5350096, 2921200, 1796077, 2718587, 5526354, 3598597, 1821804, 6640496, 3384242, 2229831, 3007349, 2419361, 12182442, 1466664, 1922928, 2890788, 1336859, 1258304, 1136496, 1599279, 1393632, 1313873, 1220873, 2048142, 980538, 1396075, 1281052, 1192672, 1625669, 1983439, 1622545, 1405274, 1371960, 1466340, 1609877, 971640, 1400375, 1442196, 1429219, 1365590, 1098518, 1339382, 1010346, 1022448, 1216043, 1325249, 1039390, 1244320, 1042099, 1356132, 976875, 954314, 1202245, 1052064, 2102515, 1110852, 1272001, 2162246, 1414192, 2168268, 2122195, 1180164, 1374550, 1458654, 2599837, 1019605, 1129811, 1650918, 1831223, 938405, 1363921, 2008607, 1285638, 1697822, 2396515, 4757407, 1435891, 1975640, 1282747, 1919583, 2648215, 2158837, 1557755, 1399034, 2795005, 1241063, 1761664]

mean iter= 90253897182/100000

yrec=[12863228, 75074363, 5106915, 1782629, 2156670, 3569925, 1810557, 1756193, 1763672, 3444880, 2129862, 1967323, 1713804, 3917440, 7165690, 1749506, 3528705, 32410761, 3024044, 2322132, 2224551, 2851971, 2547285,

3131386, 7080645, 1772114, 1747045, 1695221, 1679800, 1484563, 1223050, 1174888, 1938319, 1457366, 1777756, 1950000, 1753636, 1278823, 1706768, 1787258, 7269554, 1662119, 1775287, 2266852, 2877811, 1754119, 4384437, 1360511, 1592875, 2838883, 1836552, 1688456, 1751191, 1870838, 1626431, 1586951, 1597523, 1985174, 2837052, 1198279, 1227666, 1181024, 1589010, 1193329, 1192215, 1497523, 1520343, 1696599, 1392483, 1205759, 4258630, 2967797, 2568784, 3288862, 2129655, 1829278, 2217922, 1931765, 3272615, 1185529, 1691922, 3968905, 1240003, 3051047, 1820554, 2461489, 3330555, 2123090, 2329403, 1679583, 2574850, 4137948, 4449393, 1726634, 3818055, 20099281, 2426959, 2536284, 1834812, 2087558]

mean rec= 111087358649/100000

Ex 7

We considered 3 versions of shellsort + insertion sort. Insertion sort is in fact the base of shellsort with h constant and equal to 1. For the shellsort we used:

the basic Shell gap:

$$1, 2, \ldots, \left\lfloor \frac{N}{4} \right\rfloor, \left\lfloor \frac{N}{2} \right\rfloor$$

the Hibbard sequence:

1,3,7,15, ...

the Knutt sequence:

1,4,13,40,121, …

results:

To sort using the instertion sort on 100 arrays it took in mean 33069904 nanoseconds

To sort using the basic shellsort on 100 arrays it took in mean 3421642 nanoseconds

To sort using the Hibbard sort on 100 arrays it took in mean 1415305 nanoseconds

To sort using the Knuth sort on 100 arrays it took in mean 1703531 nanoseconds

As expected all the gaps sequence we used resulted always in a sorted array that is because every gap sequence that contains 1 gives a correct sort (as this makes the final pass an ordinary insertion sort);

As expected if we compare the time taken to sort 100 arrays of random sizes and with random values the time taken by insertionsort to sort them is the biggest, secondly there is the time taken by the basic Shellsort gap and lastly the time taken by Hibbard and Knutt sequence. This is because the Hibbard and Knutt Shellsort do O(n^3/2) comparisons in order to sort an array of size n, while the basic Shellsort has an average-case time complexity that's better than O(n^2) and the worst-case time complexity can be O(n^2) and the insertionsort does O(n²) comparisons.