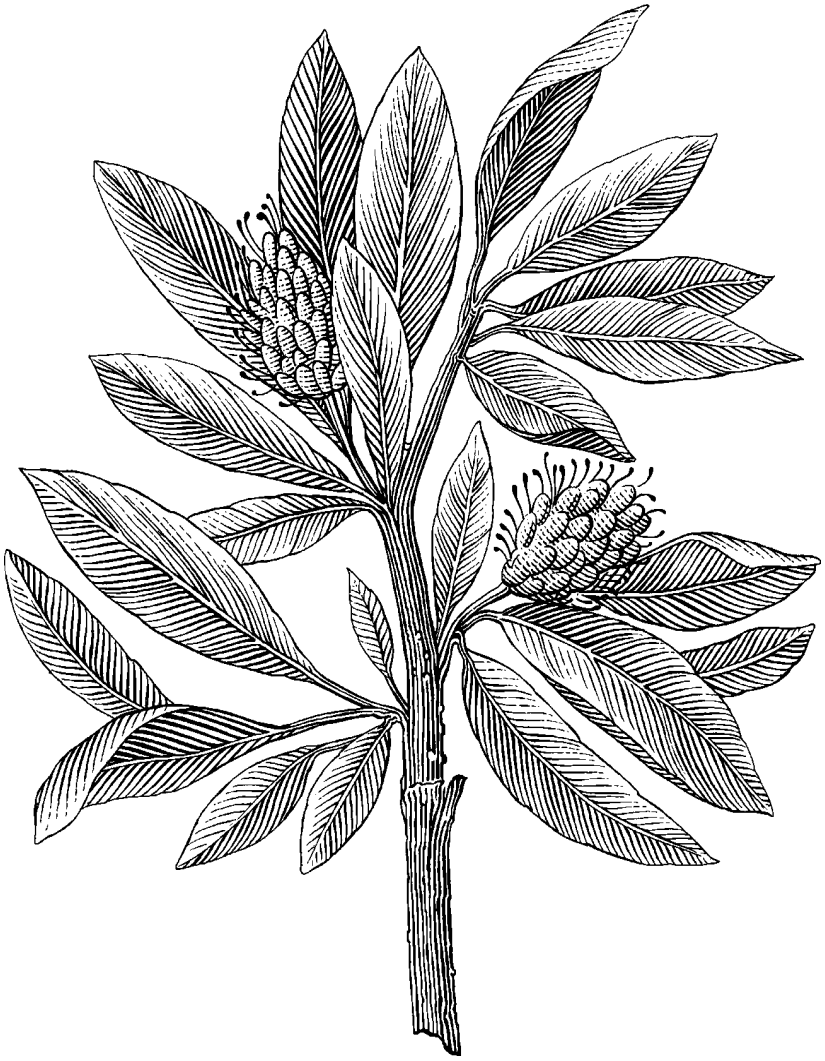
Linnéuniversitetet
Kalmar Växjö

Report

Assignment 3

1DV516



Supervisor: Morgan Ericsson
Semester: Autumn 2023

Author1: Katarina Simakina
Email es225hi@student.lnu.se

Author2: Zejian Wang
Email zw222bb@student.lnu.se

Table of Contents

Task 2

In this task we use hash table from the problem1 to store information about vehicles. We have created the Class which is called Car. This class has the following attributes: plate number, year, color and make. Hash function has the following attributes plateNumber, colour, year and make.

To make experiments to analyze how our hash function works we did the following:

Firstly, we made the function which creates randomly plate numbers.

Secondly, we can enter the number of vehicles to insert to the hash table. We write the number of vehicles should be inserted and our function randomly insert vehicles.

Thirdly, we implemented the capability to delete a specified number of vehicles from the hash table. The vehicles to delete were chosen randomly. This step allowed us to observe how the hash table accommodates removals and how it responds to subsequent insertions.

To assess the performance of our hash function, we examined the following metrics:

Number of Conflicts: This metric tracks the number of times a collision occurred during insertions.

Number of the Offset: The offset value signifies how many steps (i.e., linear probing) were taken to resolve collisions during insertions.

Load Factor: We calculated the load factor, which is the ratio of the current number of elements in the hash table to the table's total capacity. This value provides insights into how efficiently the table uses its space. It is a measure that represents how full a hash table is, indicating the ratio of the number of elements currently stored in the table to the total number of available buckets or slots in the table. A lower load factor (close to 0) indicates that the hash table has many empty buckets relative to the number of elements. This leads to inefficient memory usage as there are many unused slots. A higher load factor (close to 1) means that the hash table is approaching its capacity limit. This can lead to more collisions, longer lookup times, and potentially reduced performance.

Table 1

Hash Table Size	Number of Vehikels inserted	Number of the conflicts	Number of the offsets	Load Factor
444	232	64	112	0.52
600	280	70	115	0.46
607	511	217	692	0.84
628	424	140	335	0.67
919	500	132	254	0.54
1000	500	125	219	0.50
1709	641	118	170	0.37
1979	688	121	191	0.34

From the table 1 we can do the following conclusions:

1.Inserting the number of elements that fill the table more than 50%, then the number of conflicts and offsets have been increased.

For example, hash table has size 444; inserting vehicles is 232; number of conflicts is 64 and number offsets 112. Load factor is 0.52 which is **not** high which says that a table using quadratic probing is no more than 0.5 then quadratic probing is guaranteed to find a slot for any inserted item.

However, if it is size of the hash table is 628; inserting vehicles is 424; number of conflicts is 140 and number offsets 335. We see that number of conflicts and offsets were increased as it was filled the hash table more than 50 %. Load factor is 0,67 which tells us that hash table is approaching its capacity limit. This results in increased conflicts and offsets, indicating that collisions become more frequent and require more steps to resolve.

Our conclusion that the number of conflicts and offsets increases when the table is more than 50% full. This emphasizes the importance of maintaining a balanced load factor to minimize conflicts and ensure efficient collision resolution. A load factor closes to 0.5 ensures efficient memory usage and minimizes the occurrence of conflicts and offsets. Thus, the importance of selecting an appropriate table is crucial thing. It should be based on the number of elements will plan to insert. Ensuring that the table is not overfilled can prevent performance degradation.

2. In our experiments, we used hash tables with prime numbers as their sizes like 919 or 1979. We also employed a technique called quadratic probing. This combination offers a significant advantage in ensuring that new elements can always be inserted. This advantage is backed by Theorem 5.1 (Weiss, 2012), which tells us that when we use quadratic probing with a prime-sized table, we can always add new elements, provided the table is at least half empty. In simple terms, it significantly reduces the risk of not finding a place to put something new because prime-sized tables provide more evenly distributed spots to handle clashes.

These prime-sized tables consistently showed better load factors, often reaching a state of being half full. For example, the 919-sized table had a load factor of 0.54, and the 1979-sized table had a load factor of 0.34. So, prime-sized tables make it easier to add new stuff even when the table is somewhat full.

When it comes to conflicts, prime-sized tables performed better. For instance, the 919-sized

table had 132 conflicts, while the 444-sized table had 64 conflicts.

As well in our experiments showed that prime-sized tables had fewer offsets. For example, the 1979-sized table had 191 offsets, whereas the 600-sized table had 115 offsets, indicating a smoother process for handling clashes.

Prime-sized tables consistently maintained balanced load factors, which is good for using space efficiently. The 1979-sized table had a load factor of 0.34, while the 607-sized table had a load factor of 0.84.

In the Table2 we have added the function delete vehicles. Our experiments were based on chain operations such insert, delete and again insert. It helps to observe metrics.

Table 2.

Hash Table Size	Number of Vehicles inserted (first time)	Number of Vehicles deleted	Number of Vehicles inserted (second time)	Number of the conflicts	Number of the offsets	Load Factor
444	232	30	50	94	187	0.56
600	280	66	55	107	165	0.44
607	511	125	123	334	1079	0.83
628	424	321	322	399	668	0.67
919	500	378	379	382	504	0.54
1000	500	399	400	355	381	0,50
1709	641	599	600	395	318	0.37
1979	688	455	456	310	297	0.34

In the experiments which presented in Table2 we used the size of tables exactly the same like in the Table1. However, we did the following operations: insert numbers of the vehicles, delete numbers of the vehicles and again insert. It is worth noting we deleted and inserted approximately the same number of vehicles.

Firstly, load factor did not change (the same like in Table 1).
In Table 2, the load factors remained consistent with Table 1, suggesting that the overall capacity utilization of the hash tables didn't significantly change when using a combination of insertions and deletions. This consistency reflects the balanced distribution of elements within the tables.

Secondly, we made function inserting vehicles after deleting vehicles and we observe that the number of the conflicts and offsets have been increased. For example (Table 2), size of the table 444: the number of the conflicts 94, the number of the offsets 187. Comparing values with the Table 1: size of the table 444: the number of the conflicts 64, the number of the offsets 112. Size of the table 1709: the number of the conflicts 395, the number of the offsets 318. Comparing values with the Table 1: size of the table 1709: the number of the conflicts 118, the number of the offsets 170.

The reason of it that when an element is deleted, the slot (or bucket) is marked as "deleted" but not actually cleared. Marked slots (deleted buckets) are not off-limits for new insertions; they are used as part of the collision resolution strategy, but having too many of them can lead to

performance issues.
When new elements are inserted after deletions, and the hash function maps them to locations that are marked as "deleted," the new elements are placed in these marked slots.

This mechanism allows for efficient collision resolution. However, it leads to more conflicts and offsets when many marked slots exist, especially if a table becomes more than 50% full. In this scenario, the table is at risk of experiencing performance issues as probing may take longer to find an empty slot.

Task5

In this task, quick sort has been implemented and experiments has been done to determine at which recursion depth Heap sort or Insert sort should be swapped to from Quick sort. To conduct the experiment and present results more straightforward and precise, three classes have been created instead of only one Quick Sort class. The class QuickSort is pure quick sort when doing the sorting on the various arrays, so that no "depth" parameter in the sort method, this class's function is only used to do the comparison of time cost among all sorting classes. The class QuickToHeapSort has the "depth" parameter in the sorting method at which a certain depth the sorting will be swapped from quick sort to heap sort. The class QuickToInsertSort also has the "depth" parameter in the sorting method at which a certain depth the sorting will be swapped from quick sort to insert sort.

In our experiment, we have sorted various sizes of arrays, from size of 10000 to 100000, the arrays are randomly generated numbers within the range of array sizes. We sorted the arrays with different recursion depths from 0 to 30. At each depth, a array has been sorted 10 times in different swapping sorting method and then get the average value to be more precise. As result data is lot, only a part and necessary of it presented in table below, which we can see some findings from it.

Table 3.

Array size	Recursion Depth	Pure quick sort time(ns)	Pure insert sort time(ns)	Pure heap sort time(ns)	Quick to Heap sort time(ns)	Quick to Insert sort time(ns)
10000	*4*	514940	3899210	715730	648710	835310
	5	520570	3835640	709010	630150	567330
	6	513660	3798000	697010	697010	437360
	7	522330	3798440	703650	596330	397320
	8	515570	3733090	705930	591560	**391790**
	9	517460	3787660	680190	586100	417900
	10	665860	3767720	690550	558460	417040
	11	519200	3816670	720970	563100	444960
	12	529130	3690360	689820	544250	455210
	13	526870	3781870	717490	557930	485850
	14	539190	4123940	762070	581810	548250
	15	520360	3608300	682780	523490	503090
	16	512270	3776750	692080	524150	536080
	17	539020	4137500	701850	532850	539040
20000	*10*	1178050	15933710	1587050	1357710	1040720
	11	1123200	15554320	1557180	1317990	952050
	12	1139520	15457460	1538900	1282370	**944900**
	13	1125290	15534980	1592650	1265960	955810
	14	1154200	15583390	1549830	1260440	973480
40000	*10*	2238910	65597120	3168940	2659850	1964780
	11	2214720	65980850	3159210	2601040	1895320
	12	2256670	65981450	3185020	2523160	**1854870**
	13	2296520	66081000	3198750	2519830	1962840
60000	*11*	6551260	214748364	9229240	7461060	5666630
	12	6301210	214748364	8638740	6846200	**5068840**
	13	5934040	214748364	9338530	6966850	5072600
	14	6608050	214748364	9834360	7237350	5749710
80000	*11*	7559450	214748364	12662460	9406070	7674730
	12	7485500	214748364	12310260	8784470	6839410

	13	7713200	214748364	12264630	8870830	**6619460**
	14	8302960	214748364	12356200	8714660	7493760
10000	*12*	9247840	214748364	12843010	9598330	8210360
	13	8922160	214748364	12838880	9620790	7271470
	14	8273840	214748364	12584750	9472110	**7253260**
	15	8247420	214748364	12734310	9063060	7313520

We can see from the above that as the array size increases, the sorting time for all algorithms also increases, which is expected as sorting larger datasets takes more time. Also we can draw from it that the pure quick sort always cost much less time than pure insert and pure heap sort time. We can also observe that when recursion depth is low, the pure quick sort still cost less time than quick to heap sorting and quick to insert sorting, take example from the table when the array size is 10000 and depth is less and equal than 5.

As the depth increases, quick to heap sorting and quick to insert sorting cost lesser time. At a certain depth, it costs less time than pure quick sorting and it has the least time when at a certain depth for example as the table shows above when array size is 1000 and depth is 8. And then quick to insert and quick to heap sorting time start to increase until a certain depth that they are almost the same as the pure quick sorting, as the depth is too large that exceeds the number of all recursions that the quick algorithm would not swap to heap or insert at all.

We can also observe that when the depth is set small, the quick to heap sorting costs less time than quick to insert sort, for example as the table shows when the array size is 10000 and the depth is less and equal than 4, however as the depth increases, it takes more time for quick to heap sort than for quick to insert sort, when the depth is 8, the time cost difference is the largest and the time cost for quick to insert sorting is the least, hence generally quick to insert sort is faster than quick to heap sort and pure quick sort.

We made some math calculation and boldly made a consumption about what the optimal depth should be to do the swap from the results table above shows. The calculation for example, when the array size is 10000, the optimal depth is 8, 8 = round(Log_e10000) - 1, and all the calculations are below:

Table 4.

Array size	Optimal depth	Calculation
10000	8	round(Log_e10000) - 1
20000	12	round(Log_e20000) + 2
40000	12	round(Log_e40000) + 1
60000	12	round(Log_e60000) + 1
80000	13	round(Log_e80000) + 2
100000	14	round($Log_e100000$) + 2

From the mathematical analysis, if we ignore the constant in the calculation and the round function, we can deduce that the optimal depth is log_en, n is the size of input array.

All in all, when doing the quick sort on a array, when the recursion depth is log_en, it should be swapped to insert sort.

Task 6

For this task, we have implemented both the iterative and recursive versions of the Merge Sort algorithm. Our implementation allows us to enter the size of the array which is generated with random values. We have measured the execution time of both the iterative and recursive versions of Merge Sort and present the results in the following table:

Table 5

Number of the Run	Size of the array	Iterative version of Mergesort (milliseconds)	Recursive version of Mergesort (milliseconds)
1	1000	1	2
2	2000	1	5
3	5000	2	18
4	10000	3	76
5	15000	5	145
6	22000	6	207
7	37000	11	563
8	56999	11	823
9	89888	14	1201
10	100000	15	1495

From the Table5 we made the following conclusions:

The iterative version of Mergesort consistently exhibits faster execution times compared to the recursive version. For instance, with an array size of 2000, the iterative version takes only 1 millisecond, while the recursive version requires 5 milliseconds.

As the size of the array increases, the time required for execution of both versions of Mergesort also increases. For instance, when the array size is 56999, the iterative version takes 11 milliseconds, while the recursive version takes 823 milliseconds.

The reason why the recursive version takes more time than the iterative version is related to the overhead associated with function calls and the repeated splitting of the array. In the recursive version, the algorithm divides the array into sub-arrays multiple times, leading to a deeper function call stack and more memory usage. This extra bookkeeping and memory management result in slower performance for larger array sizes.

In contrast, the iterative version directly manages the merging process and uses a loop structure to divide and merge the arrays. This approach reduces the overhead associated with function calls and memory management, leading to faster execution times, especially for larger arrays.

Moreover, when choosing between the recursive and iterative versions of Mergesort, it is essential to consider factors such as array size and specific performance needs. While the time difference may be minimal for small arrays, as array size grows, the iterative version becomes a more efficient and practical option.

Task 7

For this task we have implemented four different gap sequence:

Function ShellSort1()
This function implements the ShellSort algorithm using the standard ShellSort gap sequence. The gap sequence is initially set to half of the array's length and is iteratively reduced by dividing it by 2 in each step until it becomes 1. This sequence is used for sorting the array.

Function ShellSort2()
This function implements the ShellSort algorithm using a gap sequence defined in an array a as {9, 5, 3, 2, 1}. It specifically employs these gap values for sorting the array.

Function ShellSortCiura()
This function implements the ShellSort algorithm using Ciura's gap sequence, which is defined as {701, 301, 132, 57, 23, 10, 4, 1}. It applies Ciura's gap values to perform the sorting.

Function ShellSortTokuda()
This function implements the ShellSort algorithm using Tokuda's gap sequence. Tokuda's Gap Sequence is generated using the formula where h starts at 1 and is iteratively updated by 2.25 * h + 1 until it is less than or equal to the array's size. This sequence is used for sorting the array.

Table 6

№	Size of the array	parameters	ShellSort1()	ShellSort2()	ShellSortCiura()	ShellSorTokuda()
1	1000	Time (milliseconds)	3	0	0	0
		Number of swaps	7518	31343	6465	6332
		Number of the comparisons	15040	31343	6465	6332
2	7000	Time (milliseconds)	2	7	0	8
		Number of swaps	75090	1411596	67930	56792
		Number of the comparisons	148479	1411596	67930	56792
3	15000	Time (milliseconds)	2	22	0	0
		Number of swaps	202465	6527428	198623	127122
		Number of the comparisons	374701	6527428	198623	127122

4	26987	Time (milliseconds)	8	32	0	8
		Number of swaps	410896	20617595	484965	236813
		Number of the comparisons	747091	20617595	484965	236813
5	44444	Time (milliseconds)	8	72	8	8
		Number of swaps	723367	55516489	1099349	388296
		Number of the comparisons	1321825	55516489	1099349	388296
6	987654	Time (milliseconds)	176	36248	562	169
		Number of swaps	32610125	1422189388	369215463	9004854
		Number of the comparisons	49845912	1422189388	369215463	9004854

In accordance the result for the Table6 we can make the following conclusions:

1. **Time performance**

The most of the time to make sorting it took for ShellSort2().
Comparing the function ShellSort2() and ShellSort1(): more quickly make sorting ShellSort1().
The observed performance differences between ShellSort1() and ShellSort2() can be attributed to the gap sequences they use. We have analyzed it and made the following conclusions:

Gap Sequences:
ShellSort1(): It uses the standard ShellSort gap sequence, which initially sets the gap to half of the array's length and progressively reduces the gap size until it becomes 1. This results in a versatile sequence that adapts to the array size.
ShellSort2(): It uses a fixed gap sequence defined as {9, 5, 3, 2, 1}. These specific gap values might not be as effective as the dynamic sequence in ShellSort1() for all array sizes.

Adaptability:
ShellSort1() adjusts the gap according to the array size, ensuring that the elements are efficiently rearranged. This adaptability allows it to perform well in a wide range of scenarios.
ShellSort2() uses a fixed sequence that might **not** be suitable for all array sizes. If the chosen gaps in the sequence do not align well with the data, it can result in more complex sorting operations, leading to increased execution times.

Time for sorting:
The timing data from the Table shows that ShellSort1() generally outperforms ShellSort2() across different array sizes, supporting the notion that the dynamic gap sequence of ShellSort1() is more effective in these the particular experiments.

We made conclusion that the adaptability of the gap sequence in ShellSort1() allows it to perform well in a variety of scenarios. This adaptability is crucial for efficient sorting, especially when dealing with arrays of varying sizes or with different distribution characteristics. On the other hand, ShellSort2() employs a fixed gap sequence that might not work optimally in all cases, resulting in slower sorting times.

Time Performance of Ciura's and Tokuda's Gap Sequences: ShellSort with Ciura's Gap Sequence and ShellSort with Tokuda's Gap Sequence consistently exhibit better performance across different array sizes as these sequences are well-suited for practical applications.

Comparison between Ciura and Tokuda: When comparing the two more efficient sequences, Tokuda's Gap Sequence generally performs faster than Ciura Gap Sequence as observed in our data. This suggests that Tokuda's Gap Sequence is more effective for the specific dataset and sorting operations in our experiments.

2. **Number of the swaps and comparisons.**

The significant number of swaps and comparisons in ShellSort2() has a noticeable impact on the time required to perform the sorting. This can be explained by understanding how these operations affect the efficiency of the sorting algorithm:

Swaps involve exchanging elements within the array. Each swap operation is time-consuming because it requires multiple memory read and write operations.
ShellSort2() involves a much larger number of swaps compared to ShellSort1(). For instance, in the second experiment, the number of swaps in ShellSort2() is substantially higher (1411596 swaps) compared to ShellSort1() (148479 swaps).
The additional swaps in ShellSort2() contribute to its longer execution time. As the number of swaps increases, so does the time it takes to sort the array.

Comparisons are essential in determining the order of elements. More comparisons usually lead to longer sorting times.
As well, ShellSort2() involves a significantly higher number of comparisons than ShellSort1(), which directly affects its time performance.

Comparing ShellSortCiura() and ShellSorTokuda().

Comparing these two shell sorting, we have concluded that ShellSorTokuda() makes more quickly sorting than ShellSortCiura(). If we increased array than time for performance this operation has been increased. However, it needs not much time to perform it.
For example, if array is 44444 (run number 4): time performance for both sorting is 8 milliseconds; if we increase array, it is 987654 (run number 5): from this run we see that performance time has been increased for both sorting: ShellSortCiura() is 562 milliseconds and ShellSorTokuda() is 169 milliseconds. So, ShellSorTokuda() makes sorting more quickly.
The reason why ShellSorTokuda() out performs ShellSorTokuda() in the following aspects:

Number of swaps and comparisons.
Comparing ShellSorTokuda() outperforms ShellSortCiura() we noticed that numbers

of swaps and comparisons more for ShellSortCiura(). For example, in the run number 5: ShellSortCiura() has 1 099 349 swaps and comparisons which is greater than for ShellSorTokuda() which has 388 296 swaps and comparisons. The reason of it we see in the folloing aspects:

Gap sequence efficiency: The efficiency of the gap sequence used by ShellSortCiura() and ShellSorTokuda() also plays a role in their performance.
ShellSortCiura() uses Ciura's gap sequence, which is known to be efficient for practical applications and has been extensively tested. It is designed to strike a balance between adaptive behavior and efficiency.

ShellSorTokuda(), on the other hand, uses Tokuda's gap sequence, which is optimized for minimizing the average number of comparisons required to sort an array. This sequence has been shown to be effective for a wide range of array sizes.

The choice of an efficient gap sequence can have a significant impact on the number of comparisons and swaps required, ultimately affecting the sorting time.

In conclusion, ShellSort1() generally outperforms ShellSort2() in terms of time performance, thanks to its adaptability and dynamic gap sequence.
ShellSorTokuda() demonstrates better performance than ShellSortCiura() due to the more optimized gap sequence.