KING ABDULAZIZ UNIVERSITY
FACULTY OF COMPUTING AND INFORMATION TECHNOLOGY
CPCS 223-Analysis & Design of Algorithms
Sprint 2020

Compare between two algorithms using Empirical Analysis

Mona Mohammed Hafez
1780735

Content:

1 Introduction:

We have two ways for analyzing any algorithm, mathematically or empirically. Both algorithms nonrecursive and recursive, can be analyzed mathematically. Though these techniques can be applied successfully to many simple algorithms, not all.

In fact, even some seemingly simple algorithms have proved to be very difficult to analyze with mathematical precision and certainty(average case for quicksort).To avoid this complexity we use empirical analysis.

Empirical analysis gives us an idea of how well a given algorithm will perform in a specific situation, it is also important in comparing two algorithms which may or may not have the same order of complexity – when would we use one and not the other.

In this report we use empirical analysis to analyze two algorithms first one is a brute-force algorithm to check whether all the elements in a given array of *n* elements are distinct, the second algorithm uses quicksort to sort the list before checking the uniqueness of elements in a list with other operations.

2 Empirical Analysis of Algorithms:

2.1 The Experiment Purpose:

The purpose of this experiment is to compare the efficiency of 2 approaches to check the uniqueness of elements in a list, the first one uses brute-force algorithm with an unordered list, the second one uses a preordered list using quicksort. then Decide which algorithm is faster and compare results with the theoretical assertion about the algorithm’s efficiency.

2.2 The Efficiency Metric:

We have two ways to measure the efficiency of an algorithm:

1- insert counter into a program implementing the algorithm to count the number of times the algorithm's basic operation is executed.
2- time the program implementing the algorithm using some functions such as console. time then console.timeEnd in javascript.

The first way is straightforward and machine independent. we should only mindful of the possibility that the basic operation is executed in several places in the program and that all its execution needs to be accounted for. Also, we should test the program each time when we modify it and ensure that we solve the problem and have the correct count for the basic operation.

For algorithms that depend just on the size n we always have the same result when we run the program but for algorithms that depend on both the size and the instance, we can run the program several times then take the average for the results. In this case, we don’t care about the machine performance we focus on the algorithm itself so it's clear that we can use the counter to analyze the two algorithms and compare them. for our purpose, it's not useful to use runtime as a metric because we don’t focus on the performance of a specific machine on this algorithm.

For the reasons above I decided to use counter to count the number of times the algorithm's basic operation is executed.

2.3 input sampling strategy:

For input sampling, we will start with a small size 2000 and increase it by a constant value (each time add 1000 to the previous size) until we have ten sizes end with size 11000 (the sizes will be 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 11000). The reason behind choosing these sizes in this pattern is that ten points are enough to give a clear graphical representation and does not slow down the browser used to test the efficiency of the algorithm. In this experiment, we have two algorithms and each one has different cases worst, average and best. The range for the input will be 0x9, 0xff, 0xffffeee, 0xffffeeee depend on the algorithm to be studied and the cases for each one.

Brute-force algorithm depends on both size and instance(all element will be unique or will have some elements have the same value). For the worst case use the range 0xffffeeee in a method called fill that uses Math.random() To fill the list with a unique value for the sizes we previously identified. For the average case use the range 0xffffeee to generate a list that may contain two elements that have the same value. Because the best case is not important when we study the efficiency of any algorithm, we will ignore it.

Presorted algorithm uses the quick sort algorithm which depends on the size and value of the partition element (we use random index for the partition so we can't achieve the worst case so we can use a small range to have a list with the same element so each time the partition element will divide the list into a list with no element and another list with the same element minus the partition element). For the worst case use small range such as 0x9 to have same elements with the same value in the list for the average case I use range 0xee that generate

Several instances of the same size are included in the sample, the averages or medians of the observed values for each size should be computed and investigated instead of or in addition to individual sample points.

2.4 Prepare and implementation algorithms:

We have two different algorithms to solve the same problem and use another algorithm to sort the list in the Presorted algorithm and another function to fill the list randomly with a specific range:

- brute-force unique elements algorithm:
  (from textbook ch2 section 2.3 page 89)
- Presort Element uniqueness algorithm:
  (from textbook ch6 section 6.1 page 203)
- Quick sort algorithm used by Presort Element uniqueness algorithm:
  (From Dr. Muhammed Al-hashimi website sortcomp.js) with the other method used by this algorithm such as fill method

Tools used in this experiment:

- JavaScript language: using to implement the code .

- Firefox browser: using to run the code and show the output on the console .
- Microsoft Excel: using to represent result data (see 2.6) .

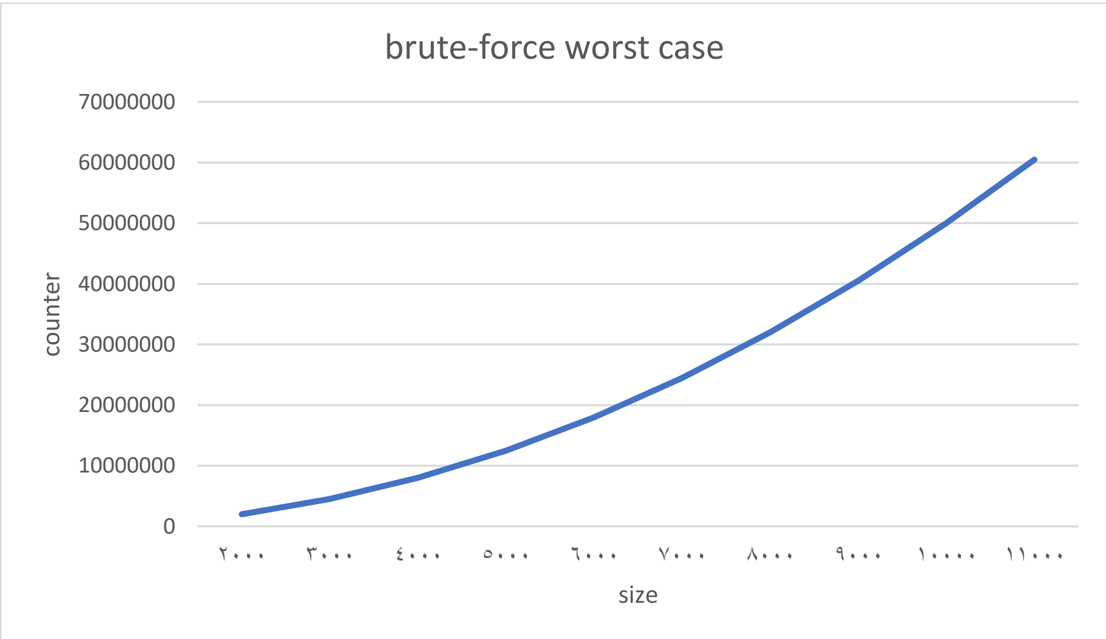
2.5 Generate a sample of input

Using the fill method to generate random numbers with a specific range for each case as we describe in 2.3 . Each run for the program will generate the input depend on which case we want to test, so we have to change the range and call the right function for each case. See the comments in the code

2.6. Data Observed from Running the Algorithm:

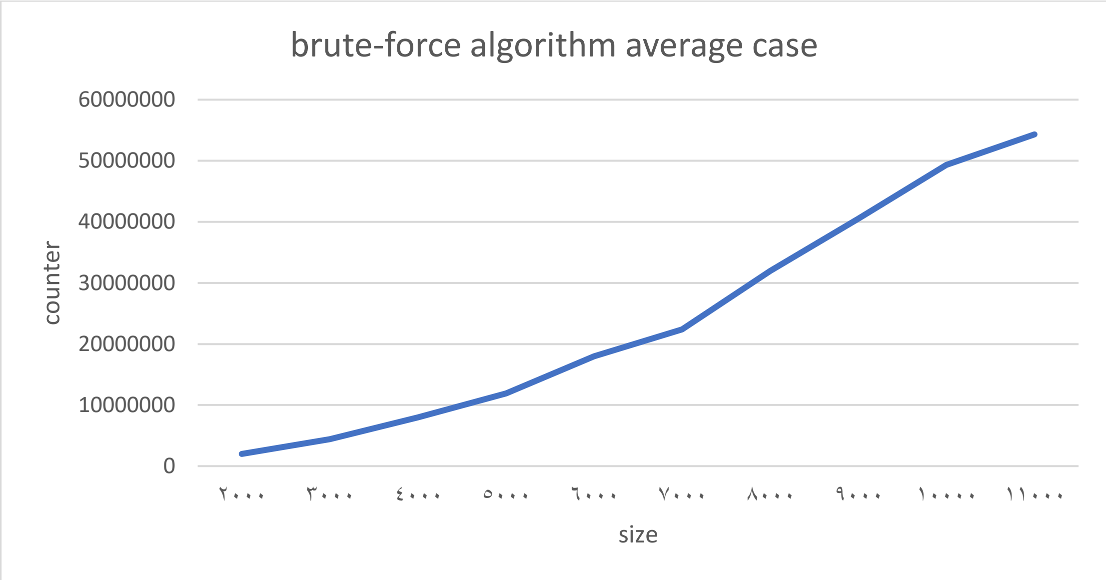
size	1	2	3	4	5	6	7	8	9	10	average
2000	1999000	1999000	1999000	1999000	1999000	1999000	1999000	1999000	1999000	1999000	1999000
3000	4498500	4498500	4498500	4498500	4498500	4498500	4498500	4498500	4498500	4498500	4498500
4000	7998000	7998000	7998000	7998000	7998000	7998000	7998000	7998000	7998000	7998000	7998000
5000	12497500	12497500	12497500	12497500	12497500	12497500	12497500	12497500	12497500	12497500	12497500
6000	17997000	17997000	17997000	17997000	17997000	17997000	17997000	17997000	17997000	17997000	17997000
7000	24496500	24496500	24496500	24496500	24496500	24496500	24496500	24496500	24496500	24496500	24496500
8000	31996000	31996000	31996000	31996000	31996000	31996000	31996000	31996000	31996000	31996000	31996000
9000	40495500	40495500	40495500	40495500	40495500	40495500	40495500	40495500	40495500	40495500	40495500
10000	49995000	49995000	49995000	49995000	49995000	49995000	49995000	49995000	49995000	49995000	49995000
11000	60494500	60494500	60494500	60494500	60494500	60494500	60494500	60494500	60494500	60494500	60494500

Table1: result for the brute-force algorithm worst case (counter)



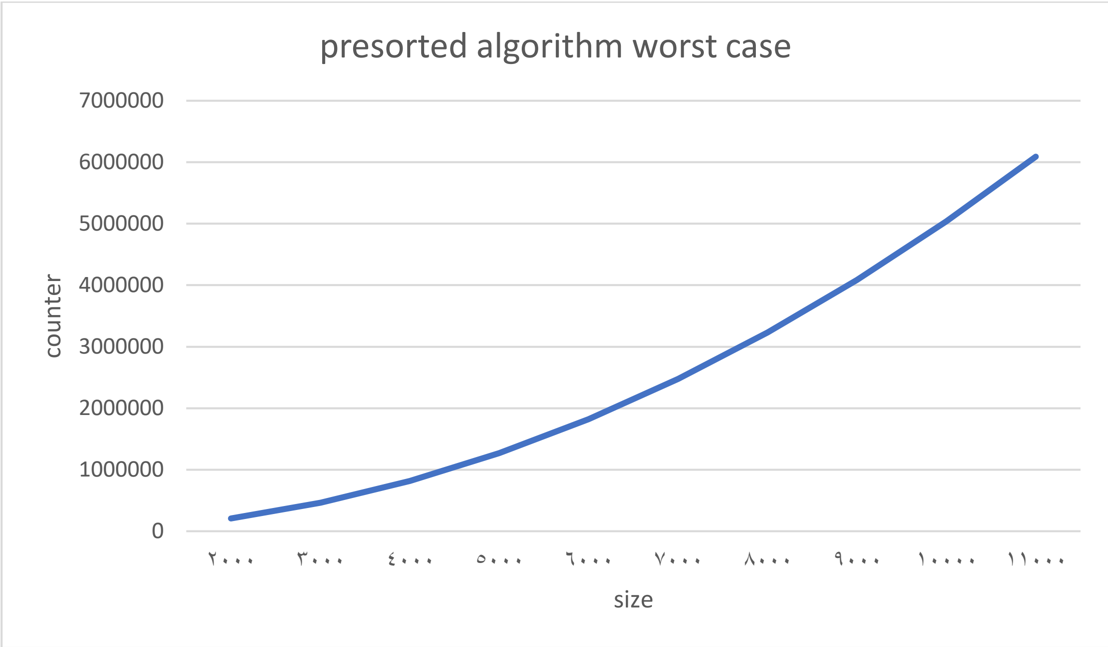
size	1	2	3	4	5	6	7	8	9	10	average
2000	1999000	1999000	1999000	1999000	1999000	1999000	1999000	1999000	1999000	1999000	1999000
3000	4498500	4498500	4498500	4498500	4498500	4498500	4498500	4498500	4498500	3373769	4386026.9
4000	7998000	7998000	7998000	7998000	7998000	7998000	7998000	7998000	7998000	7998000	7998000
5000	12497500	12497500	12497500	12497500	6355916	12497500	12497500	12497500	12497500	12497500	11883341.6
6000	17997000	17997000	17997000	17997000	17997000	17997000	17997000	17997000	17997000	17997000	17997000
7000	24496500	24496500	24496500	24496500	24496500	24496500	24496500	24496500	4255220	23742177	22396939.7
8000	31996000	31996000	31996000	31996000	31996000	31996000	31996000	31996000	31996000	31996000	31996000
9000	40495500	40495500	40495500	40495500	40495500	40495500	40495500	40495500	40495500	40495500	40495500
10000	49995000	49995000	49995000	49995000	49995000	43319987	49995000	49995000	49995000	49995000	49327498.7
11000	60494500	60494500	60494500	54122821	60494500	5071383	60494500	60494500	60494500	60494500	54315020.4

Table2: result for the brute-force algorithm average case (counter)



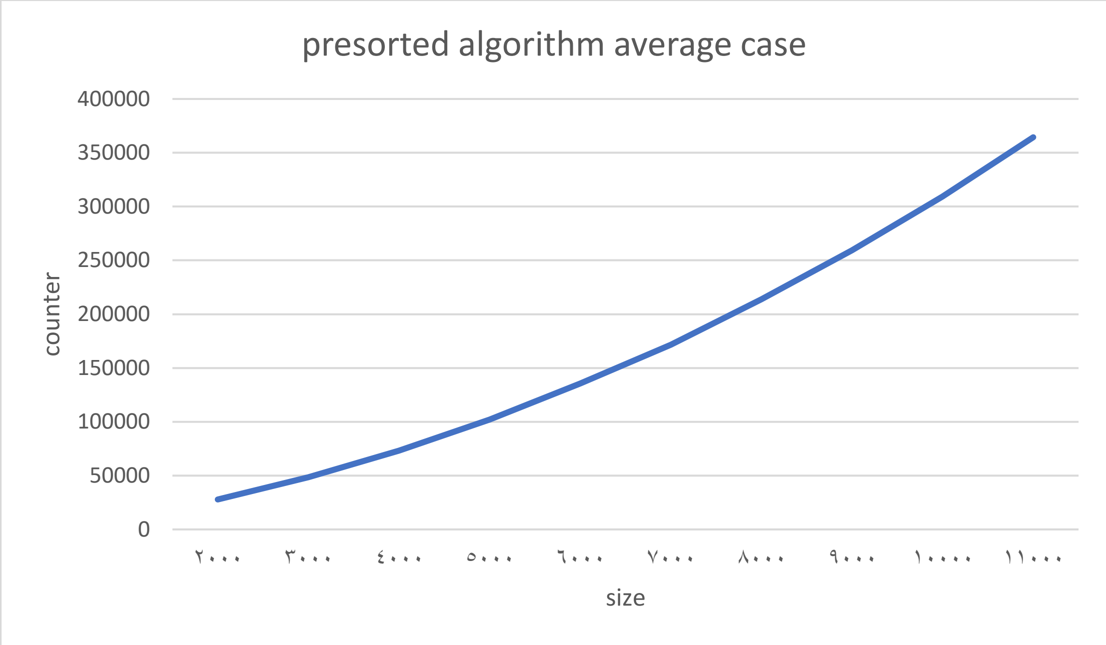
size	1	2	3	4	5	6	7	8	9	10	average
2000	207495.6	207775	207906.5	207598.9	206648.4	206711.9	207732.6	207740.1	207451.1	209171.4	207623.15
3000	461214.1	463174.8	461749.8	460320.3	462861.9	460102.1	462474.7	462063.8	462478.8	462653.3	461909.36
4000	814854.9	815843.1	815152.4	815454.4	815954.9	815388.2	814572.5	815325.1	817662.6	813613.5	815382.16
5000	1271617	1268038.2	1269093.7	1269884.5	1268422.3	1268007	1268386	1270542.9	1269859.5	1271109.3	1269496.01
6000	1821821	1821168.3	1822621.7	1820330.9	1820807.6	1822321.2	1823494	1825030.7	1825411.5	1826136.5	1822914.34
7000	2476654	2480238.4	2479531.5	2478948.2	2477986.4	2475945.6	2477835	2479193.8	2478664.9	2481403.8	2478640.12
8000	3231818	3232206.7	3230209.3	3234267	3228852.4	3228176.6	3226931	3231961.4	3227930.7	3238655.7	3231100.88
9000	4088865	4083169.9	4084640.5	4080013.6	4080585	4081030.5	4086311	4082868.7	4081127.9	4085449.3	4083406.13
10000	5032650	5033596	5039964.7	5036953	5039816.9	5036964	5043677	5039498	5031108.4	5038637.6	5037286.49
11000	6087617	6090285.6	6094517.3	6091352.3	6087475.7	6095703.3	6089998	6093107.5	6090849.4	6086091.1	6090699.63

Table3: result for the presorted algorithm worst case (counter)



size	1	2	3	4	5	6	7	8	9	10	average
2000	27349.4	27965.5	27867.1	28114.2	27920.1	27990.1	28004.5	27675.5	27601.4	27627.4	27811.52
3000	47918.4	48510.8	50069.8	47882.7	48740.5	48793.9	49479.7	47550.3	47947.4	47680.8	48457.43
4000	73461.3	72678.8	72431.4	73970.7	73374.4	73442.1	73277.4	73845.4	73756.3	72816.9	73305.47
5000	103991.4	101543.6	101996.9	100654.7	100601.2	100831.4	102384.6	104610.3	101689.2	102996.3	102129.96
6000	136631	136209.2	133539.3	135083.2	135592.1	134906	138006.5	137213.4	134725.1	134128.9	135603.47
7000	171577.5	170936.9	171302.1	169714.1	169937.6	174263.7	172225.4	169326.5	172986.5	172472	171474.23
8000	213769.7	210456.6	210446.5	216827.3	217049.6	210596.3	215669.9	217881.8	212264.5	214735	213969.72
9000	257944.9	258983.9	261092.8	259156.3	259587.1	258974.8	259502.5	258548.1	260314.3	260144.5	259424.92
10000	311323.6	307587.9	305895.8	309214.5	306017.5	312640.2	311786	310940.7	308375.4	309440	309322.16
11000	359569.4	365525.2	370259.3	362559.6	361937.3	363119.3	363858.8	367190.1	366430.4	363691.2	364414.06

Table 4: result for the presorted algorithm average case (counter)



size	1	2	3	4	5	6	7	8	9	10	average
2000	8.795	8.1595	8.143	8.13	8.35	8.983	9.1395	8.525	8.559	8.4815	8.526549998
3000	18.227	19.793	18.38	21.888	18.935	18.8215	17.048	17.2555	18.0955	17.4635	18.5907
4000	31.202	33.2805	31.902	31.2035	30.396	30.8515	29.895	30.1425	30.8375	30.2005	30.99110001
5000	47.162	46.983	46.686	47.557	46.579	47.439	49.7125	47.8695	46.9235	46.801	47.37125
6000	66.9075	68.611	68.8665	68.3825	69.4705	77.4815	71.7995	68.56	68.4855	68.381	69.69454999
7000	91.7885	90.8775	91.272	95.398	96.4625	91.892	92.32	95.7595	97.9765	95.623	93.93695
8000	130.3355	124.604	113.7605	115	112.118	111.9305	112.5395	113.988	130.3935	115.58	118.02495
9000	141.8765	142.7425	151.3335	146.9595	146.398	141.093	140.3835	141.1225	141.275	143.333	143.6517
10000	174.0665	173.1905	173.5045	174.018	173.595	173.8355	172.886	175.104	174.2345	172.698	173.71325
11000	201.8335	257.878	203.3595	198.908	199.2735	209.8685	207.2435	203.732	204.8645	204.202	209.1163

Table5: result for the brute-force algorithm worst case (time)

size	1	2	3	4	5	6	7	8	9	10	average
2000	9.9285	7.5535	7.1475	9.82300001	9.5195	8.2165	8.1365	7.7915	6.8475	7.1365	8.210050003
3000	16.6795	17	16.5465	15.6485	17.401	16.3895	16.1465	15.7185	15.67	16.6415	16.38414998
4000	27.8595	27.739	28.0425	27.284	7.191	27.952	28.2415	28.7125	29.216	28.7855	26.10235
5000	44.085	42.706	43.863	43.357	43.972	44.8615	46.202	43.667	44.019	41.5015	43.82340001
6000	58.4455	61.5325	59.259	61.413	58.3245	58.5565	58.43	46.029	58.179	58.516	57.86849999
7000	24.814	90.9965	95.073	36.7685	83.5575	79.3785	79.8275	81.4695	79.829	79.358	73.1072
8000	103.3695	103.049	107.7755	105.639	108.0065	127.1275	134.21	75.5445	102.229	102.4195	106.937
9000	131.6985	131.1265	129.9605	40.2620001	128.2535	129.433	128.348	128.223	129.99	127.6075	120.49025
10000	157.987	25.231	157.23	37.948	163.177	157.4185	156.5435	157.0865	150.326	157.227	132.01745
11000	194.4615	151.4935	91.3315	190.117	192.307	19.8335	193.3245	160.509	191.2285	190.2845	157.48905

Table6: result for the brute-force algorithm average case (time)

size	1	2	3	4	5	6	7	8	9	10	average
2000	2.469	0.8605	0.9815	1.1375	1.0295	1.082	1.487	1.117	0.994	0.7425001	1.190049991
3000	1.3855	1.5225	1.2405	1.49700006	1.6575	1.499	1.281	1.5075	1.471	1.3095	1.437100014
4000	2.6715	2.1955	2.2285	2.23699999	1.961	2.468	2.4035	2.2055	2.5555	2.4465001	2.337250011
5000	3.272	3.7	3.038	3.27200003	3.547	4.4065	3.2155	3.3935	3.5645	2.8850001	3.429400013
6000	4.3955	4.059	3.929	4.60900003	4.735	4.344	4.193	4.188	4.2089999	4.043	4.27045
7000	5.272	5.257	5.374	4.93949996	4.8045	5.4565	5.5135	5.5015	5.305	5.7985	5.322199995
8000	6.68	6.9449999	7.016	6.904	7.054	7.2124999	7.069	6.8365	7.2324999	7.4525	7.04019998
9000	8.825	8.905	9.2055	7.51999998	8.1355	7.4375	8.7005	7.6894999	7.876	7.3034999	8.159799983
10000	9.781	9.6970001	9.9425	11.69	9.2975	10.9095	9.2975	9.9155	10.7785	9.5585	10.08674999
11000	12.1555	12.1685	11.7775	11.115	10.5205	10.6375	13.115	10.9415	11.535	11.309	11.52749999

Table7: result for the presorted algorithm worst case (counter)

size	1	2	3	4	5	6	7	8	9	10	average
2000	1.17	0.842	0.3085	0.2405	0.2525	0.2355	0.2745	0.232	0.2345	0.4335	0.42235
3000	0.461	0.3845	0.531	0.307	0.518	0.435	0.3835	0.341	0.5305	0.6915	0.4583
4000	0.5615	0.726	0.507	0.668	0.525	0.6465	0.489	0.6315	0.516	0.6325	0.5903
5000	0.694	0.77	0.6875	0.663	0.6965	0.6705	0.6855	0.645	0.71	0.6075	0.68295
6000	0.871	0.9815	0.981	1.08	0.838	1.0445	1.0265	0.9215	0.9275	0.83	0.95015
7000	0.998	1.125	1.074	0.866	0.9535	1.0845	1.0385	1.077	0.966	1.149	1.03315
8000	1.1815	1.2515	1.0655	1.698	1.3815	1.2115	2.077	1.3255	1.185	1.337	1.371400001
9000	1.7075	1.3615	1.3915	1.211	1.4075	1.4945	1.2445	1.392	2.1465	1.4245	1.4781
10000	1.569	1.733	1.3555	1.442	1.9375	1.4855	1.502	1.9335	1.5715	1.622	1.61515
11000	2.112	1.495	1.717	1.8295	1.941	1.9875	2.0265	2.1025	2.118	2.892	2.0221

Table8: result for the presorted algorithm average case (time)

2.7 Analyze the data obtained:

From the previously observed data, the following can be inferred:

We can compute the ratios M(2n)/M(n) and see how the counter reacts to doubling of its input size. As we discussed in Section 2.2 from the textbook, the ratios determine the behavior of algorithms in one of the basic efficiency classes. such ratios should change only slightly for logarithmic algorithms and most likely converge to 2, 4, and 8 for linear, quadratic, and cubic algorithms, respectively—to name the most obvious and convenient cases.

- Table1: the data represent the worst case for the brute force algorithm. by computing the ratios C(2n)/C(n) when we double its input size we have:
  $\frac{C(8000)}{C(4000)} = 4.000500125$, $\frac{C(4000)}{C(2000)}$ =4.0010005,

$\frac{C(6000)}{C(3000)} = 4.000666889$

The function increases fourfold then it should be quadratic
C_{worst} (n)=$n^2 \in \Theta(n^2)$.

- Table2: the data represent the average case for the brute force algorithm. by compute the ratios M(2n)/M(n) when we double its input size we have:
  $\frac{C(8000)}{C(4000)} = 4.000500125$, $\frac{C(4000)}{C(2000)}$ =4.0010005,
  $\frac{C(6000)}{C(3000)}$ =4.103258008,
  The function increases fourfold then it should be quadratic
  C_{avg} (n)=$n^2 \in \Theta(n^2)$.
- Table3: the data represent the worst case for the presorted algorithm. by compute the ratios M(2n)/M(n) when we double its input size we have:
  $\frac{C(8000)}{C(4000)}$ =3.962682823, $\frac{C(4000)}{C(2000)}$ =3.927221796,

  $\frac{C(6000)}{C(3000)}$ =4.000666889,

  The function increases fourfold then it should be quadratic
  C_{worst} (n)= $n^2 \in \Theta(n^2)$.
- Table4: the data represent the average case for the presorted algorithm. by compute the ratios M(2n)/M(n) when we double its input size we have:
  $\frac{C(8000)}{C(4000)} = 2.9923631$, $\frac{C(4000)}{C(2000)} = 2.397656$,

  $\frac{C(6000)}{C(3000)} = 2.0732053$, $\frac{C(10000)}{C(5000)} = 2.3649608$,

  The function increases slightly more than twofold then it should be linearithmic $n \log_2 n$

  C_{avg} (n)= $n \log_2 n \in \Theta(n \log_2 n)$.

We can use the same steps for the time metric which leads to the same result.

2.8 Comparison between algorithms:

As a conclusion from the last step we have the following:

Brute force algorithm:
- C_{worst} (n)=n^2.
- C_{avg} (n)=n^2.

Presorted algorithm:
- C_{worst} (n)=n^2.
- C_{avg} (n)= $n \log_2 n$.

There is A convenient method for comparing the orders of growth of two specific functions is based on computing the limit of the ratio of two functions t(n) is the presorted algorithm, g(n) is the brute-force algorithm.
Three principal cases may arise:

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 \text{ implies that t (n) has a smaller order of growth than g(n)} \\ \text{c implies that t (n) has the same order of growth as g(n)} \\ \infty \text{ implies that t (n) has a larger order of growth than g(n)} \end{cases}$$

Using excel find the limit for the sizes (2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, 11000)

size	worst
2000	1
3000	1
4000	1
5000	1
6000	1
7000	1
8000	1
9000	1
10000	1
11000	1

1 is constant implies that both presorted algorithms and brute-force algorithms have the same order of growth for the worst case.

size	average
2000	0.005483
3000	0.00385
4000	0.002991
5000	0.002458
6000	0.002092
7000	0.001825
8000	0.001621
9000	0.00146
10000	0.001329
11000	0.00122

The result 0 implies that the presorted algorithm has a smaller order of growth than the brute-force algorithm for the average case.

3 conclusion:

Empirical analysis used when it is difficult to analyze with mathematical precision. we apply the steps of the empirical analysis and found the time efficiency for two algorithms using the counter as metric and collect physical time (will have the same result). Using a counter for comparing the efficiency is more accurate and have stable result independent on the machine we use.

The result matches the mathematical result discussed in the textbook as follow:

Brute force algorithm:

- C_{worst} (n)=n^2.
- C_{avg} (n)=n^2.

Presorted algorithm:

- C_{worst} (n)=n^2.
- C_{avg} (n)= $n \log_2 n$.

The comparison between the two algorithms :

Presorted algorithm is much better in the average case($n \log_2 n$) than brute-force one (n^2), but in the worst case they both have the same order of growth (n^2) because the worst case in Presorted algorithm is rear to happen that makes the Presorted algorithm a good choice.