Theoretical and Experimental comparison of sorting algorithms

Alexe Luca Spataru
24.04.2019
West University of Timişoara

Abstract

This paper compares 20 sorting algorithms mainly on factors such as memory, stability, speed, computational idea, practical use. Here you will find algorithms from the upper to lower bound of sorting and not find an explanation of how they work, except a brief one of those that are not widely known.

1 Introduction

Sorting is a fundamental problem in computer science, it is the core for understanding and designing algorithms, for this reason, it is studied at introductory courses on algorithms. Many algorithmic ideas developed as an output of solving the problem of sorting, because of that it is a well-studied topic, currently, there is a wide range of sorting algorithms and it is unlikely that some new and more efficient ones will be discovered. Every algorithmic book dives deeply into sorting as it is a natural operation in real-life so that more formal concepts of algorithmic analysis can be introduced neatly, but there is also **THE BOOK** [6], which could not go unmentioned in every topic that touches sorting, that has a volume totally dedicated to sorting where the author gives a complete and comprehensive overview of the problem.

Its importance comes in many circumstances, whether it is faster search, finding the frequency of every element in a collection, finding equal elements in more collections or simply organizing your data so that it is not randomly

spread and has a pattern. In order to gain the full benefits of a sorting algorithm, we need to know which is the most suited for our will and this is why we need to compare them. As the will of every human is different this paper compares more sorting algorithms than usual, not only restricting itself to the fastest or popular ones but pointing out the full range of advantages, restrictions, and ideas behind the algorithms. The paper is more concerned about eluding the pros and cons rather than explaining the implementation or proof of complexity since all of that can be found in other works, however, you will find some explanation on algorithms which are not taught or are assumed to not be known.

The structure consists of two main parts: Theoretical Comparison and Experimental Comparison. The Theoretical part consists of a comparison of complexity in different cases, computational idea and its origin, popularity, applicability, etc., whereas the Experimental one tries to prove what was stated in theory and also mentions details that were ignored. The algorithms that will be compared are distributed in 4 categories, most often the algorithms will be compared only within their category, for reasons which will be mentioned later. Because we are talking about sorting, the order the categories will be discussed is sorted increasingly and decreasingly so that it ends where it started.

But if one wants to sort the sorting algorithms then one must find the most inefficient one and a significant part of this paper is dedicated to solving this problem. Which is important because some people have the wrong opinion of what is inefficient and which algorithm is the most inefficient one. Or even worse they ignore and limit themselves to only efficient ones, which is rational for them but irrational if they see the reasons for their importance in this paper.

2 Theoretical Comparison

The approach of comparing sorting algorithms by distributing them into categories is giving the benefit of not going messy, by dividing algorithms into one group and comparing the groups separately the concepts would be better memorized and more comprehensible. To respect this approach one must put in one category the algorithms that are similar between them and different from others so that any statement which holds for the whole group must hold for each individual algorithm and vice versa. In our case the classes

of algorithms that will be compared are:

1. undesirable amount of time to wait, around $O(n!), O(n^n), O(\infty)$
2. long time to wait, around $O(n^2), O(n^3)$
3. short time to wait, $O(nlogn)$
4. sometimes very short time to wait, around $O(n)$

In the first category are algorithms close to a more combinatorial nature, because the information for output is taken from the whole set rather from individual elements of it. The fact that these type algorithms are very isolated does exempt us from the burden of comparing with the rest and focusing on a more inside-oriented comparison. The second category fit algorithms that have the genuine paradigms of sorting by exchange, selection, insertion. The third one includes algorithms that improve, by using the Divide and Conquer paradigm, or use, as subroutines in specific cases, the algorithms from the second category, and this is a reason to compare the two separate groups. The last class contains sorting methods that use some distribution techniques to overcome the lower bound of comparison-based algorithms. Because such algorithms claim to be faster than previous ones, we should accord more attention to the distinction between these two.

2.1 Retarded Sort

The reason why we should be interested in slow algorithms is for the purpose of doing something that has no intention to be applied, just for humoring pleasure or even for the reason of going outside the usual, not narrowing our ideas. Besides that, you may find some practical applications like finding methods that must never be used in algorithm design or simply finding something you must never spend time with, thus you will spend time correctly in the future. If this did not motivate you then you can read the paper Pessimal Algorithms and Simplexity Analysis [2] which gives an original overview of this subject with more such algorithms and really strong citations, here you will also find new paradigms for designing reluctant algorithms and a real-life application of them: When the boos sends you to sort something out in Paris. But before diving in, I want to ask the question

of what is inefficient, an algorithm that does your job but will take you a lot of time or the one that will **never** do your job?

If it is the latter then every non-sorting algorithm is the worst sorting algorithm, even the act of doing nothing. Even worse than these algorithms would be the ones that spoil your job, make it harder or even destroy it. Of course, there exist such ones which are intended just for a laugh at how far stupidity could go. Some examples are Stalin Sort, the one that finds the elements which is out of order and deletes it (actually in the end you get a sorted list but not your desired one), Sleep Sort that at every element waits for some time and then displays it, 1stYearComputerScienceStudent Sort that is just a recursive call without a base case, NotMyProblem Sort, ObsessiveCompulsive Sort, Harakiri Sort, Tornado Sort etc., these can go as far as your imagination wants, even some of the names I invented myself. If we consider the option that the worst sorting algorithm is the one that never does your job, we would get a big number of odd retarded algorithms with trivial operations and we would never find the most inefficient one. Then let us consider the other option, that is an algorithm that halts with a sorted permutation of the initial array.

It is often considered that the most inefficient algorithm is Bogo Sort that randomly picks a permutation, checks if it is sorted, if not repeat, else stop. In the end, you will get a sorted list but it halts **only** when you get a sorted list. The problem is that if the algorithm does not stop then it will never do your job. If we want to put it in one of these two subcategories then we have to be sure if it works or not, in our case it halts or not, and by sure we mean 100% sure. Here is a more detailed analysis of this problem inspired by the "Infinite Monkey Theorem"[5] :

Bogo Sort has two main subroutines, $shuffle(A)$ that permutes the array and $check(A)$ that outputs True if sorted. If $check(A)$ outputs False then $shuffle(A)$ is called again. We assume that the initial array $A = [a_1, a_2, ..., a_n]$ has only distinct elements $a_1 \neq a_2 \neq ... \neq a_n$, because that is the worst scenario when we have duplicates our possibility of getting a sorted array is bigger thus the chance of halting. Everytime the probability of not getting a sorted array is $1 - \frac{1}{n!}$, the probability of not getting a sorted array after two tries is $(1 - \frac{1}{n!})^2$, the probability of not getting a sorted array, say P, after k tries is

$$P_k = (1 - \frac{1}{n!})^k.$$

What we want is a k such that $P_k = 0$, because in this situation the algorithm

will halt. By doing these steps:

$$(1 - \frac{1}{n!})^k = 0$$

$$1 - \frac{1}{n!} = 0$$

$$n = 1$$

we get to the conclusion that whatever the k the algorithm will halt only when the size of A is 1, which is obvious and trivial. However we also would get 0 if we allow an infinite number of tries:

$$\lim_{k\to\infty}(1 - \frac{1}{n!})^k = 0,\ because\ (1 - \frac{1}{n!}) < 1.$$

When k is finite there still is a possibility that we will not get a sorted array and for very big k this possibility is very small but never 0 as it is shown in this table with some examples (the boxes from the center are the values P_k, (0) means very close to 0 but not 0 and analogously (1)):

n \ k	10	100	1 000	10 000	1 bil.
2	0.0009765625	$7.88 \cdot 10^{-31}$	$9.332 \cdot 10^{-302}$	(0)	(0)
5	0.9197	0.433083	0.00023	$4.54 \cdot 10^{-37}$	(0)
10	(1)	0.99997	0.99972	0.99724	$2.09 \cdot 10^{-120}$

As P_k approaches infinity it gets closer to zero meaning more likely to halt, and in reality, an event whose probability is infinitesimally small is unlikely to happen but not impossible. We can come to the conclusion that we can be **absolutely sure** Bogo Sort will give us a sorted array only if $|A| = 1$ or we give it an infinite number of tries, but what does that mean? Does it mean that in order to halt I would have to wait an eternity? And when it halts I will stop waiting and therefore the eternity will end? Suppose it will not run an eternity is it still the worst sorting algorithm? There is a possibility that you will get your sorted array even after the first try, so if you gamble your life would you trust a dice with $n!$ faces? I would suggest that Bogo Sort sits in between the two categories mentioned before since the true behavior of it depends on mother Randomness, also its behavior varies on fixed input as

we will see in the experimental part of the paper where we will also try to answer previous questions.

There is another approach of designing a good 'bad algorithm' introduced by Miguel A. Lerma [7] which also mentions the cons of Bogo Sort and finds large lower bounds and breaks the upper bounds of sorting in the following way. Given a list of integers on length n, compute all the possible permutations and store them in another list, then use a sorting algorithm, preferably a bad one like bubble sort, to sort the permutations lexicographically. The first element in the list of permutations is the sorted initial array. The lower bound for this algorithm is $\Omega((n!)^2)$ but we can do worse, what if instead of using bubble sort we would use the just specified algorithm to sort the list of permutations, meaning generating a list of permutations of lists that contain permutations of the given integers and then sort them lexicographically with bubble sort. This algorithm would have the complexity $\Omega(((n!)!)^2)$ which is worse than the previous, and if we do it again we would get an even worse algorithm. You can observe that the inefficiency is based on recursion levels, and the algorithm with recursion level $x + 1$ is worse than one with x, and since the numbers are infinite there is no algorithm that bounds every other one. This already seems too much and certainly, nobody will ever use such an algorithm, but Lerma did miss a slight detail that allows us to perform worse.

A permutation of a set is just a bijective map into the same set. The bijective property makes the number of possibilities $n \cdot (n-1) \cdot (n-2) \cdot (n-3) ... \cdot 2 \cdot 1$ or in a more concise way $n!$. If we remove the bijective property, making it a cartesian product instead of a permutation, we would get a better(worse actually) result which is n^n, but we would also get multiple sorted lists with numbers that repeat themselves not just our desired sorted list. In this way, we need a subroutine that checks if the generated list contains all the elements in the initial one. We can do this in n^2 time: for every element in the generated list search linearly in the copy of the initial list, if found delete it and return True, else return False. The time complexity for this is $O((n^n)^2 + (n^n)^2)$, the first $(n^n)^2$ for "bubble-sorting" the cartesian product and the second one for the checking routine. In the list of sorted products the first one that has all the elements in the initial array is the desired array. And of course, as in the previous case, we can add recursive depth to this algorithm making it unimaginable stupid.

2.2 Amateur Sort

After the worst of all come the ones that are poorly designed, in a way that they compute more than necessary. Such algorithms can sort more elements than the previous ones, but are not used in real-life problems and are useless on big data. One can call the ideas behind such algorithms poor, that come only to an inexperienced programmer, however, there are some that perform badly but are not straightforward like bubble/insertion/selection sort. By looking at these algorithms a programmer would know what to avoid and where to perform.

One example of a sorting algorithm that performs worse than bubble sort is stooge sort that has the following instructions:

- Stop if the size is 1
- If the value at the start is larger than the value at the end then swap.
- Repeat for first $^2/_3$ of the list
- Repeat for the last $^2/_3$ of the list
- Repeat again for the first $^2/_3$

The complexity for stooge sort in the worst case is $O(n^{log_{1.5}3}) = O(n^{2.7095...})$. It is clear that it performs awfully bad and that its operations are forced or somehow fake, despite the fact that in the end, you obtain a sorted list.

Another algorithm whose steps seem superfluous is pancake sort, that is easy to understand if you imagine the elements as a stack of pancakes, where you put a spatula between or at the beginning, then you reverse the stack from the spatula to the end. You can sort the elements by finding the maximum, reversing from its index to the end, then reversing the whole list to the index your array is already sorted. It is of no practical use but has a puzzling nature. The complexity is $O(n^2)$ and it can be interpreted as a derived form of selection sort that does multiple swaps instead of just one.

Bubble sort is a classic example of a bad algorithm, its popularity and origin were studied by Owen Astrachan [1]. One of the reason for its popularity is the ease to implement and to remember, but from there also comes the potential of misusing it on a large data set. However, bubble sort's idea of exchanging elements is seen in other algorithms like comb sort

or cocktail sort, which do not have an impressive performance but maintain a computational idea good for educational purposes.

Cocktail sort, as you can guess from the name, differs in the way you go through the list. In bubble sort, you go bottom to top repeatedly, whereas in the other you are oscillating top bottom, bottom top. Comb sort, designed by Wlodzimierz Dobosiewicz[3], has the same idea as bubble sort except for the gap between elements that are compared, in the latter consequent elements are compared meaning the $gap = 1$, wherein comb sort you start with the gap equal to the size of the list and then shrink it by a factor, which showed the best performance on 1.3. To be more clear here is a sequence of gaps:

- n
- $\frac{n}{1.3}$
- $\frac{n}{1.3 \cdot 1.3}$
- $\frac{n}{1.3 \cdot 1.3 \cdot 1.3}$
- ...
- 1

You can see that the last loop of comparisons is similar to bubble sort. Both algorithms are better than bubble sort but still maintain the worst case of $O(n^2)$, comb sort has a very fast $\Omega(nlogn)$ but on average it performs $\Omega(n^2/2^p)$, where p is the number of increments. Cocktail sort actually has some similarities with selection sort, instead of finding the minimum bottom up only, it finds the maximum bottom up and the minimum when it passes the list top bottom. Cocktail sort is sometimes called bidirectional selection sort.

Famous for its simplicity to implement selection sort is useless on big data but can be improved using a heap data structure for finding the max/min logarithmically instead for time-consuming linear search, thus creating an algorithm that does not fit in this category. Another algorithm that can be improved is insertion sort where instead of searching linearly for the place an element to be inserted you use binary search, thus another algorithm emerges called binary insertion sort. It is better than initial insertion sort in the average case but the time complexity in the worst case is still $O(n^2)$ because of the number of swaps required to put every element at the correct location.

2.3 Pretty Good Sort

Sorting algorithms that are of practical use are the ones that are fast and efficient for any type of data, used to solve real-life problems. Such Algorithms are heap sort, merge sort, quick sort that sort in $nlogn$ time which is really useful but every one of these algorithms has its pros and cons.

Quick sort [4] is the fastest from these three but its performance depends on how the list is arranged and how you choose the pivot. Ironically the worst case scenario is when the list is already sorted, in this case, it performs n^2 comparisons. Merge sort whose performance does not depend on the arrangement of the list and maintains $nlogn$ on every case, requires additional memory. Heap sort that also maintains $nlogn$ for every case is not a stable algorithm, meaning that the relative order of similar elements changes, and in a real-world implementation, it performs slower than Merge/Quick Sort due to the constant factor that is ignored in the theoretical analysis. For these reasons, some programming languages use a hybrid sorting algorithm for their standard library.

For example, Swift and C++ use introspective sort, usually called intro sort, that has both fast average-case performance and optimal worst-case performance of $nlogn$. It mainly uses the best parts of quick and heap sort by applying the quick's partitioning until a specified recursive depth is reached, known as $2 \cdot logn$. If that depth is reached then it performs heap sort on the chunk resulted from partitioning. In this way, intro sort contains the speed of quick sort and also does not allow to reach the worst-case of it or to a stack overflow caused by recursion. It also uses insertion sort, that is the best on small inputs, if the partition size is too small where quick sort performs badly.

Another hybrid algorithm is tim sort that is used in the standard libraries of Python and JAVA, It combines merge and insertion sort in a creative way. Every list of integers is contained of increasing and decreasing sublists, for example $L = [3, 8, 21, 32, 9, 5, 3, 10]$ contains the following sublists $L_1 = [3, 8, 21, 32]$ $L_2 = [9, 5, 3]$ $L_3 = [10]$. If you reverse the lists that are decreasing you transform it into an increasing one, and if you get a small list like L_3 then use insertion sort to put in another sublist. After doing this process till the end you get sorted chunks of the original list, and the only thing you have to do is merge them. The time complexity of this algorithm is n for the best case and $nlogn$ for both average and worst case and it is a stable algorithm.

2.4 Fast but Furious

Every comparison sorting algorithm cannot do better than $nlogn$ and if we eliminate the comparison operation we can acquire the complexity of n, where we sort the data distributively. Unfortunately, it is the lower bound of sorting because you have to get the information about every number in the list. Even if such algorithms are the fastest they are very limited to the type of data they can sort or require additional memory. Applying such algorithms on the data you do not know is very risky and a better approach would be to use the $nlogn$ algorithm.

Counting sort which is the most known example of such algorithms that actually performs well only when the range of the numbers is not great because of its time complexity of $O(n + r)$ where r is the range. It also requires additional memory of size r for the process of counting. It is used as a subroutine in radix sort, which overcomes the problem of memory but does not maintain the same speed, because it does counting sort repeatedly. If you sort numbers with 4 digits then radix will perform 4 counting sorts, from this comes its complexity of $O((n + d) \cdot d)$, where d is the number of digits and $n + d$ is the counting sort subroutine. Another such algorithm is bucket sort that requires memory but performs fast only on specific data from a specific range.

A variation of bucket sort that is less efficient is pigeonhole sort mainly useful on data where the size and the range of the integers are approximately the same. The main similarity with bucket sort is that instead of having buckets where you store elements from a particular range, you have only a hole for each element, and you put more elements in a hole if they are the same. It is also similar to counting sort except that you put the actual elements in the auxiliary array whereas in the other you count their position. It has a time complexity of $O(n + r)$ where r is the range. Pigeonhole sort has a principle behind it that originates from math, which is similar to the idea of distributing elements from these sorting algorithms. For that reason, they somehow imitate linearity by trying to avoid comparing elements.

3 Experimental Comparison

All the experiments were performed on my Lenovo IdeaPad with an Intel i5-8250U 1.60 GHz and 8GB of RAM. I wrote the code in python which you can find on my GitHub [8]. Even though python is not the fastest language the actual comparison of algorithms matter rather than the language. However, you will find that in one case C++ was used, for the purpose of proving a point rather than comparing it with python, which is obviously slower. The data on which the algorithms will be compared varies on cases and is self-generated then written in a file, where it latter will be read. In the code, you will not find the data, but you will find the programs that were used to generate and I encourage to run and test the results on your system. The results will be shown as a table, inside whose are values that represent seconds.

3.1 Speed indeed

To prove the speed and also limited practical use, I will sort the elements obtained from this data: generated ages of persons, a small size list. It seems that the type of data that represent ages is very convenable for such algorithms because the maximum age can be 150. In this situation, we can use every algorithm in this category but it is not advisable to use pigeonhole sort and you can see the reason if you see the results.

Alg. / Size	Counting	Radix	Bucket	Pigeonhole
100 000	0.015	0.064	0.023	191.57
500 000	0.071	0.293	0.115	...
1 000 000	0.143	0.588	0.224	...
5 000 000	0.720	2.973	1.206	...
10 000 000	1.440	6.062	2.40	...
50 000 000	7.149	31.359	12.29	...
100 000 000	MemoryError	MemoryError	MemoryError	...

If we would round up the results you would see clearly the linearity as the size of the list grows and also that algorithms can be compared based on

their first results. Pigeonhole performed very bad because of the number of similar elements in the data, as in real life it is hard to put a lot of pigeons in one cage. The results of counting sort really showed that it is fast and if I would not be limited on memory it would sort a billion elements in circa 2 minutes and 20 seconds, bucket sort despite being a little slower actually performed well and radix performed the worst from these three. Nevertheless, it still maintained linearity as the size grew and if it was a *nlogn* algorithm that showed 0.064 seconds on a list of size 100 000 then at the size 50 000 000 it would stop at around 288 seconds not at 31.36 how radix performed, and in the next experiment radix will prove why it is superior in some case.

Suppose you do not know the form and size of the input you will have, but you know it will contain only integers. Assume in this list are two elements 8589934592 (which is equivalent of 8GB) and 2. It seems an easy task if we just compare them but we are not allowed to do that. If we use any sorting algorithm besides radix it will build an additional array of size 8589934590 or 858993459 buckets and if 8GB is the only memory you have then you could not perform the task, but radix could easily do it since you need memory only for the digits of the first and second integer. However, as every algorithm from this category, its performance is limited to only specific cases and you have to take risks when you want to use it on a data set whose organization you do not know in advance, this is why comparison-based algorithms come in handy.

3.2 Don't rave, be safe

Previously discussed algorithms are not universal and one example where they cannot perform is when you have real numbers, except for bucket sort if you implement the buckets correspondingly, that is because you are not allowed to compare, but count or sort by digits or anything else. To prove the usefulness of comparison algorithms I am going to sort this kind of data, moreover, I will also compare the sorting algorithms based on an already sorted list to see the purpose of algorithms in the standard libraries. In case of tim sort I used the python simple sort() and of intro sort the C++ STL's sort() because these are the best implementation found. Here are the first results:

Alg. / Size	Merge	Quick	Heap	Intro	Tim
50 000	0.221	0.097	0.152	0.010	0.014
100 000	0.457	0.190	0.333	0.020	0.024
500 000	2.695	1.301	2.516	0.106	0.159
1 000 000	5.680	3.190	5.353	0.227	0.411
5 000 000	33.96	21.03	33.15	1.265	3.182
10 000 000	69.27	49.05	83.41	2.795	7.384
50 000 000	...	...	...	14.606	50.48

From the results, you can form two groups, the usual algorithms, and the hybrid ones. It does not matter who won in the hybrid group because C++ is a lot faster than python, the purpose of this experiment was to prove that other people worked hard to optimize sorting and that the best sorting algorithm is the one made by them. In case of performance of the others, heap sort did well until the size of the list grew and that may be because of the constant factor, merge sort had a reasonably good performance and quick sort is the fastest from these three but here is why it is not always the best.

When we try to quick sort, with the pivot as the first element, an already sorted list of 1000 elements a RuntimeError is raised because maximum recursion depth was exceeded, but this limit can be manually set using the sys library. We will just sort this kind of lists with quick sort and the very ugly bubble sort. Here are the results:

Algorithm / Size	Bubble	Quick
1000	0.036	0.066
5000	0.591	1.229
10 000	2.473	4.951
15 000	5.333	11.03
25 000	14.81	30.66

3.3 The rabbit and the turtle

Any data we give to such algorithms they will still perform badly, to prove this lists with random integers were given to sort, whose size increases until reasonable amount of time passed and the algorithm counts as inefficient. Moreover, we will also try to see what algorithm between binary insertion sort and insertion sort is the fastest. Here are the result from the first row of tests:

Algorithm / Size	Stooge	Pancake	Bubble	Cocktail
500	2.988	0.010	0.015	0.017
1000	8.863	0.031	0.051	0.050
5000	...	0.565	0.993	0.990
10 000	...	2.536	4.009	4.003
15 000	...	6.145	9.021	9.076
25 000	...	17.53	25.37	25.50
50 000	...	...	...	...

The second row:

Algorithm / Size	Selection	Insertion	Binsertion	Comb
1000	0.036	0.044	0.011	0.002
5000	0.480	0.743	0.213	0.015
10 000	1.960	3.065	1.002	0.032
15 000	4.603	6.641	2.393	0.053
25 000	13.53	18.89	7.172	0.073
50 000	57.57	76.78	33.90	0.142
100 000	...	...	171.38	0.323
500 000	...	...	...	3.984
1 000 000	...	...	...	9.796

Comb sort performed better than binary insertion sort because it deals with the small elements at the end of the list by just swapping with another element at the beginning whereas in the other you need to perform multiple swaps until you get to the correct position. However, binary insertion sort really outperformed insertion sort and the obvious reason for that is binary search. But is it better on a nearly sorted array? Here are the results of insertion sort:

Size \ Algorithm	Insertion
100 000	0.015
500 000	0.074
1 000 000	0.156
5 000 000	0.755
10 000 000	1.493
50 000 000	7.471

However the results depend on the range of how the list is nearly sorted. Now as the turn goes to binary insertion sort, I will actually start with 5000 elements and then go to 100 000, at the same range.

Size \ Algorithm	Binsertion
5000	0.213
10 000	0.926
25 000	6.456
50 000	26.82
100 000	133.5

As in the case of quick sort you will not always get to sort an already sorted list, but if we compare two algorithms in general then we have to take into account every case, and in these cases, the advantages binary insertion and quick had previously, drag them down to inefficiency.

3.4 Hold or Halt?

I am very restricted here with my computing power due to the fast-growing function of factorial and super-exponential and because of that, I will limit myself to the first level of recursion which I think is enough to convince you how slow these algorithms are. For example, computing a list with just 4 integers with the recursion level 2 would take approximately $24!^2$ operations with the factorial algorithm and with the exponential $(4^{4^4})^2 = 4^{512}$ operations. To make this practical I will sort a list of 2 integers [2,1] then I will add one more until my patience ends. I start first with the factorial algorithm then I will move to the super-exponential one. Here are the results:

List \ Algorithm	Factorial	Exponential
[2,1]	0.0002896	0.0007243
[3,2,1]	0.0004336	0.0010099
[4,3,2,1]	0.0006673	0.0463752
[5,4,3,2,1]	0.0096621	6.2405209
[6,5,4,3,2,1]	0.3994884	1396.1882
[7,6,5,4,3,2,1]	16.878021	...
...	...	...

I hope that you are convinced enough to admit that these are very slow algorithms and that you wish to proceed to the flaws of bogo sort. The "randomness" in our case is a pseudo-random generator installed in python's library which is Mersenne twister, currently, there are no pure random generators. Its period is $2^{19937}-1$ which is very long and certainly we will die when our computers would reach this one. But suppose we do not, then if after $2^{19937}-1$ tries bogo sort does not halt means it will never halt because the random sequences will repeat themselves. Another flaw is that its behavior varies on fixed input, which means it is a more randomized algorithm rather than a sorting algorithm, from there comes the name bogus which means fake. We saw that as the number of tries grows the probability P_k goes to 0 (but never 0) and eventually the probability of terminating would grow to 1 (but never 1). This dilemma did not allow us to answer the question if it halts or not. I think you should base your answers on the following results, here I run multiple attempts on a single list, also I keep track of the number of tries k. I include also the lists that were ignored in the theoretical analysis, the ones with repeated elements.

List \ k	Smallest	Average	Biggest
[3,2,1]	$6.5 \cdot 10^{-5}, k = 1$	0.00010, $k = 20$	0.00012, $k = 29$
[5,4,3,2,1]	$6.31 \cdot 10^{-5}, k = 1$	0.00019, $k = 62$	0.000365 $k = 108$
[7,6,5,4,3,2,1]	$0.00011, k = 60$	0.00447, $k = 2631$	0.00852, $k = 5002$
[9,8,7,6,5,4,3,2,1]	0.00027, $k = 134$	0.3080, $k = 148067$	0.5980,$k = 275114$
[11,10,9,...,3,2,1]	0.576, $k = 214892$	6.7883, $k = 2559910$	16.5705, $k = 6468303$
[5,5,4,4,...,1,1]	0.00133, $k = 507$	0.6068, $k = 242292$	1.3411, $k = 479867$
[4,4,4,...,1,1,1]	0.1884, $k = 61170$	1.724, $k = 545399$	5.0068, $k = 1687653$

As you see when we have duplicates the algorithm performs better for the

reason we discussed earlier, also it obviously terminates and the final probability where it did for lists with distinct elements is:

	n	k	P_k
[3,2,1]	3	29	0.005
[5,4,3,2,1]	5	108	0.405
[7,6,5,4,3,2,1]	7	5002	0.370
[9,8,7,6,5,4,3,2,1]	9	275114	0.468
[11,10,9,...,3,2,1]	11	6468303	0.850

You see that when $n = 11$ we were lucky and the probability was reasonable for the algorithm to continue, in contrast when $n = 3$ we reached a really long "unsorted spree". So, the actual inefficiency of bogo sort only depends on chance, you could get a sorted array after the first try or after a billion. Even though this algorithm is awfully bad I do not think it is the worst.

4 Conclusion

There does not exist such algorithm as the best, there are just those that suit best to your data, implementation or programming language. Everything involves a tradeoff, as in the case on *nlogn* algorithms, you lose speed but you get universality and less memory usage, in case of quick sort, you get speed but you have to take some risks, in case of binary insertion sort you get speed at the average case but inefficiency if the data is almost sorted. The "real" sorting is done with chunks of classical algorithms, smart people worked in teams to optimize sorting, perhaps the best algorithm is the one created by them.

We have to take into account many factors when describing an algorithm as inefficient. If an algorithm is inefficient because it will never fulfill a task then other algorithms created for other purposes, where they are efficient, and any other algorithm, however retarded it may be, are considered inefficient. If an algorithm is considered inefficient because it takes a lot of time then we would have to consider its lower bound which represents minimum expected time, because if the running time is small then the algorithms does not count as inefficient.

We have to study sorting algorithms at the university not to use them in everyday life but to develop our algorithmic thinking, and some of them

are good for that purpose. If we limit ourselves only to the algorithms we need, the ones that are the most efficient, we neglect those that may be developed to create a totally new one, which may or may not be efficient but can lead to an expansion of our ideas about sorting. It would be great if there would be a map where branches of algorithms connect in relations and hierarchies, the same as linguistics does with languages.

References

[1] O. Astrachan. Bubble sort: an archaeological algorithmic analysis. In *ACM SIGCSE Bulletin*, volume 35, pages 1–5. ACM, 2003.

[2] A. Broder and J. Stolfi. Pessimal algorithms and simplexity analysis. *ACM SIGACT News*, 16(3):49–53, 1984.

[3] W. Dobosiewicz. An efficient variation of bubble sort. *Information Processing Letters*, 11(1):5–6, 1980.

[4] C. A. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[5] R. Isaac. The pleasures of probability. *Springer Science & Business Media*, pages 48–50, 2013.

[6] D. E. Knuth. *The art of computer programming: sorting and searching*, volume 3. Pearson Education, 1997.

[7] M. A. Lerma. How inefficient can a sort algorithm be? *CoRR*, abs/1406.1077, 2014. URL `http://arxiv.org/abs/1406.1077`.

[8] S. A. Luca. My code. URL `https://github.com/lukaqwe/ThAndExpCompOfSortAlg`.