An introduction to bogoceptionsort and its performance compared to ordinary bogosort

Emil Sitell

Abstract

This paper introduces the bogoceptionsort. The bogoception tackles one of the major problems with bogosort, namely its relative efficiency at sorting very short array. This problem stems from the deterministic nature of the source code itself. In bogoceptionsort the source code for bogosort is bogosorted and then compiled, this then runs and checks if the array is sorted. If the array isn't sorted then the source code is scrambled again. The bogoception sort performs similarly to bogosort for large n but excels at smaller n and need orders of magnitudes more iterations for small n.

1 Introduction

Through the ages hundred upon hundred algorithms for sorting lists and alike have been discovered, studied and perfected. From the naïve approach of bubble sort and insert sort to the more complex quicksort and radix sort, data scientist the world over search for better algorithms. One of the major breakthroughs in the field came when the revolutionizing bogosort was discovered, it combined beauty and elegancy with ehmm.. speed. To this day it's one of the few algorithms with best case sorting of $\mathcal{O}(n)$. Many esteemed researchers have optimized the algorithm since its inception but in this paper one major new discovery is revealed and studied, the bogoceptionsort.

2 The algorithm

In ordinary bogosort the array is shuffled and checked, if the array is sorted then the algorithm returns, otherwise the array is shuffled again. See code snippet below:

```
is_sorted = False
while not is_sorted:
    random.shuffle(shuffled_array)
    for i in range(len(shuffled_array) - 1):
        if shuffled_array[i] > shuffled_array[i + 1]:
            raise UserWarning
        if i == len(shuffled_array) - 2:
            is_sorted = True
```

For smaller arrays this makes it very easy to accidentally sort the array in only a couple of tries. To combat this the bogoception sort starts by bogosorting the code for the bogosort. In this paper the scramble is limited to shuffling the order of the rows. The code is then converted to a string and executed with pythons `excec`. If the code doesn't return a sorted list, the code is shuffled again and it tries again, see code snippet below. This includes both cases when an exceptions is raised as well as if the code is executed correctly but the underlying bogosort fails to sort the array. You can never be certain the code was right so might as well shuffle again.

```
is_sorted = False
while not is_sorted:
    random.shuffle(code)
    try:
        exec(formatted_code,
            {"bogo_test": unsorted, "random": random})
        is_sorted = True
    except:
        pass
```

3 Evaluation

By initially bogosorting the code, the algorithm essentially performs as $bogosort(n+a)$ where a is the number of lines in the original bogosort, in this case 8. Therefore it needs thousands of iterations to sort an array even with $n = 2$. See figure 1 to see the averaged performance for different values of n.

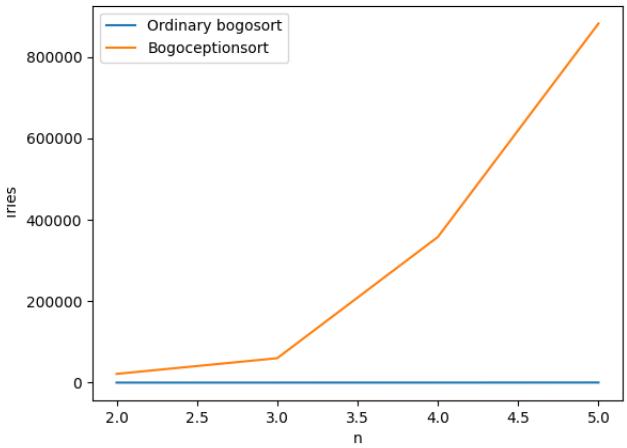

Figure 1: Iterations needed to sort an array of size n for bogosort and bogoseptionsort.

As seen i figure 1 the iterations needed for bogoception sort quickly grows very large. In theory when n gets larger $lim_{n\to\infty} n + a \approx n$ which implies that bogoception sort converges to ordinary bogosort. Due to the factorial nature of bogosorts complexity it is theorized that this would still need a large n before they become comparable. Due to the inefficie... *hark* technical limitations the research team was not able to verify this empirically.

4 Further studies and improvements

The algorithm shows great prospect of becoming one of the major players for sorting arrays, especially smaller arrays when the execution time is of utmost importance. The algorthm described in the paper is bogoception sort of degree 1, further studies into harnessing the power recursive programming the concept could easily be extended to bogosorting the bogoceptionsorts code. This could then be repeated until desired performance is reached.