Bogo-sort is Sort of Slow

Max Sherman

June 2013

1 Introduction

In this paper, we provide a rigorous analysis of the runtime of the infamous sorting algorithm "bogo-sort" using only elementary techniques from probability. The material presented draws heavily from *Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms* [2]. We give an overview of the main results proved in [2] and present the details of these results at a level suitable for an audience with no prior experience in the field of probability.

2 Motivation

Before jumping head-first into our analysis, it will be helpful to establish some context in order to motivate the discussion. Bogo-sort is depicted in *The New Hacker's Dictionary* [3]. The entry reads:

> **bogo-sort**: /boh'goh-sort'/ /n./ (var. 'stupid-sort') The archetypical perversely awful algorithm (as opposed to **bubble sort**, which is merely the generic *bad* algorithm). Bogo-sort is equivalent to repeatedly throwing a deck of cards in the air, picking them up at random, and then testing whether they are in order. It serves as a sort of canonical example of awfulness. Looking at a program and seeing a dumb algorithm, one might say "Oh, I see, this program uses bogo-sort." Compare **bogus**, **brute force**, **Lasherism**.

It is an appropriate question to ask why this analysis is performed at all when various other, more performant algorithms for sorting exist. The answer is, of course, that we jump at the opportunity to apply mathematics whenever we are able, even if the "application" serves no discernible purpose. So without further ado, we eagerly begin our understanding of bogo-sort.

3 Key Terms

In order to fully grasp the arguments presented in this paper, the reader is required to be familiar with common definitions and notation from elementary probability. Since we do not assume this knowledge, a brief primer of the relevant terms is provided in the following section. Many of these definitions are taken from [1].

- **Random Variable**
  A random variable is an expression whose value is the outcome of a particular experiment. Just as in the case of other types of variables in mathematics, random variables can take on different values.

  Example. If we roll a standard six-sided die, we can represent its value with the random variable X. Then $X \in \{1, 2, 3, 4, 5, 6\}$.

- **"P" Notation**
  "P" notation is used to represent the probability of an outcome of a random variable, and typically takes the form $P(X \sim a) = p$, where X is a random variable, $\sim$ is some relation, a is a number, and p is the probability that the expression inside the parentheses is true.

  Example. Using the die example from before, since the probability of rolling any of the six numbers on the die is equal, we can write
  $$P(X \leq 4) = \frac{2}{3}$$
  to mean that the probability is 2/3 that a roll of a die will have a value which does not exceed 4.

- **Sample Space**
  The sample space is the set of values that a random variable is allowed to attain. Said differently, it is the set of all possible outcomes. Sample spaces are commonly denoted as Ω.

  Example. For the roll of a single die, $\Omega = \{1, 2, 3, 4, 5, 6\}$, as these are the only values in which a die roll can result.

- **Distribution Function**
  A distribution function is a function that assigns a nonnegative number to an outcome of an experiment such that if $m(x)$ is a distribution function of a random variable, then
  $$\sum_{x \in \Omega} m(x) = 1.$$

  Example. Flipping a fair coin has two outcomes, and typically these outcomes are assigned equal probability. In this case our sample space is given by $\Omega = \{H, T\}$, and $m(H) = m(T) = 0.5$.

- **Expected Value**
  Let X be a numerically-valued random variable with sample space Ω and distribution function $m(x)$. The expected value $E(X)$ is defined by
  $$E(X) = \sum_{x \in \Omega} xm(x).$$

  The expected value represents the value we would probably get by repeating the experiment a sufficient number of times. It is somewhat like the average value in this sense. A fact we will use later about the expected value is its linearity.

  Example. We can compute the expected value of the roll of a single die, denoted by the random variable X. Note that the probability distribution function for each outcome is 1/6. The formula gives
  $$E(X) = \sum_{x \in \{1,2,3,4,5,6\}} \frac{x}{6} = \frac{1}{6} + \frac{2}{6} + \frac{3}{6} + \frac{4}{6} + \frac{5}{6} + \frac{6}{6} = 3.5.$$

  This value is also the average of all of the possible outcomes of a die roll.

- **Independence**
  Two events E and F are independent if $P(E)$ and $P(F)$ are unrelated. Said differently, if E occurs, it has no effect on the probability that F will occur, and vice versa.

  Example. Flipping a coin is an independent event, since the result of any given flip, in no way affects the result of any subsequent flip.

- **Geometric Distribution**
  The geometric distribution is the probability that the first occurrence of success requires k number of independent trials, each with success probability p. If the probability of success on each trial is p, then the probability that the kth trial is the first success is

  $$P(X = k) = (1 - p)^{k-1}p.$$

  The expected value of a geometrically distributed random variable is given by $E[X] = 1/p$.

  Example. Suppose an ordinary die is thrown repeatedly until the first time a 1 appears. We can model this with a geometric distribution on the number of rolls, which is a member of the set $\{1, 2, 3, \ldots\}$, with $p = 1/6$. The expected number of rolls until we produce a 1 is $(1/6)^{-1} = 6$.

- **Big O Notation**
  While it is a slight departure from the flavor of the previous terms, it is also useful to understand big O notation. It is the lingua-franca of algorithm analysis.
  A function $f(n)$ is said to be $O(g(n))$ if $f(n) \leq c \cdot g(n)$ for all $n > n_0$ for some fixed n_0, and some fixed c. So, this notation is a way of assigning upper bounds to functions, or in our case, algorithms. Algorithms are modeled as functions mapping from an input of size n, to the number of "steps" required to terminate.

  Example. A program is said to run in linear time if it is $O(n)$. If we have a program that prints every number from 1 to a supplied input number n, then this program runs in linear time, since it executes n steps, which is proportional to the input, by a multiplicative factor of one.

4 Bogo-sort Implementation

Bogo-sort's implementation is simple, so we provide it in the pseudocode below.

Algorithm 1: Bogo-sort.

```
Input array a[1...n]
while a[1...n] is not sorted do
    randomly permute a[1...n]
end
```

Much like the method of sorting a deck of cards described in the Hacker's Dictionary quote above, we see that bogo-sort takes in an array, checks that it's sorted, and if it isn't, shuffles the elements, then repeats. We describe the pseudocode for the sortedness checking routine, and the random permutation routine below.

Procedure Sorted.
(Returns true if the array is sorted, and false otherwise.)

```
for i = 1 to n − 1 do
    if a[i] > a[i + 1] then
        return false
    end
end
return true
```

There are a few things to note from looking at these algorithms. The first observation one might make is that this algorithm is not guaranteed to terminate. It appears possible that we continue inside the while-loop forever, each iteration being one where the Randomly Permute procedure shuffles the array into another

Procedure Randomly Permute.
(Permutes the array.)

```
for i = 1 to n − 1 do
    j := rand[1 . . . n]
    swap a[i] and a[j]
end
```

unsorted state. Interestingly, the probability of the runtime for the algorithm being infinite is zero. Let T be a random variable denoting the runtime, then by Markov's Inequality

$$P(T \geq t) \leq \frac{E[T]}{t}, \quad t > 0.$$

Later, we will prove that the expected running time, $E[T]$ is finite, so for a program running for t, as t grows large the probability that $T \geq t$ decreases to zero.

Another important factor to note is that inside the Randomly Permute procedure, we just loop through the array once, meaning the procedure is $O(n)$ and therefore linear. Similarly, in the Sorted procedure, we loop through the array from left to right, so Sorted is linear as well.

Finally, notice that the check to see if the array is sorted is performed initially when we begin the algorithm. Because of this, if we are lucky enough to be handed an already-sorted array, bogo-sort will terminate in linear time, and nothing will ever be permuted. In the next section, we detail our plan of attack for analyzing bogo-sort.

5 Plan of Attack

Now that we have built up a strong foundation by establishing relevant terminology, defining bogo-sort, and making several observations about the algorithm, we can discuss how our analysis will proceed.

We will analyze the expected running time for bogo-sort in the standard context that we are given an array $\bar{x} = x_1 x_2 \dots x_n$ containing a permutation of the set of numbers $\{1, 2, \dots, n\}$. To analyze the running time of the algorithm, which is a comparison-based sorting algorithm, we follow the usual convention of "counting on one hand the number of comparisons, and on the other hand the number of swaps". Swaps and comparisons are going to be the two metrics quantifying how long the algorithm runs for. The theorems that follow will prove the expected value for the number of swaps and comparisons that bogo-sort performs under certain restrictions. The expressions obtained will give the expected value in the best, worst, and average cases.

6 Analysis

Our analysis is divided into parts, each analyzing a separate component of bogo-sort.

6.1 Comparisons Required to Check an Array for Sortedness

The first section of the algorithm we analyze is the expected number of comparisons that the Sorted procedure performs every time it is called. This analysis is carried out for two different types of input arrays.

6.1.1 The Basic Case

In the basic case, let the input $\bar{x}$ be an array with n elements, each of which is distinct. Before we state and prove a theorem, we make the following observation.

Observation 1. *If the kth element in the list is the first one which is out of order, the algorithm makes exactly $k-1$ comparisons (from left to right) to detect that the list is out of order.*

Theorem 2. *Assume $\bar{x}$ is a random permutation of $\{1, 2, \ldots, n\}$, and let C denote the random variable counting the number of comparisons carried out in the test whether $\bar{x}$ is sorted. Then*

$$E[C] = \sum_{i=1}^{n-1} \frac{1}{i!} \leq e - 1.$$

Proof. For $1 \leq k < n$, let I_k be the random variable indicating that at least the first k elements in $\bar{x}$ are in order. I_k is an indicator function here, so as an example, if the first 3 elements are sorted, $I_3 = 1$, and if not then $I_3 = 0$. A first observation is that $I_k = 1 \Leftrightarrow C \geq k$. For on one hand, if the first k elements are in order, then at least k comparisons are carried out before the for-loop is exited. On the other hand, if the routine makes a minimum of k comparisons, the kth comparison involves the elements x_k and x_{k+1}, and we can deduce that $x_1 < x_2 < \cdots < x_{k-1} < x_k$. The inequalities are strict here since every element in $\bar{x}$ is distinct, although in general, sortedness allows two elements who are equal to reside next to each other inside $\bar{x}$.

Since this bi-conditional is true, it must be the case that $P[C \geq k] = P[I_k]$. The probability on the right hand side can be computed as

$$P[I_k] = \frac{\binom{n}{k} \cdot (n-k)!}{n!}.$$

The numerator is the product of the number of possibilities to choose k first elements to be in the correct order and the number of possibilities to arrange the remaining $n-k$ elements at the end of the array. The denominator is the total number of arrays of length n. Reducing this fraction, we obtain $P[I_k] = (k!)^{-1}$. By noting that I_k is either 0 or 1, we can directly apply the definition of expected value to get

$$E[C] = \sum_{k>0} P[C \geq k] = \sum_{k>0} P[I_k] = \sum_{k=1}^{n-1} \frac{1}{k!} = \sum_{k=0}^{n-1} \frac{1}{k!} - \frac{1}{0!}.$$

The expression on the far right is the truncated Taylor series for the exponential function evaluated at 1, so we obtain our result. □

Theorem 2 tells us that given an array with distinct elements, we only need a constant number of comparisons on average to check if a large array is sorted, and when n is large, this constant approaches from below $e - 1 \approx 1.72$. In the worst case, we must check the entire array, and so compare $n - 1$ times.

6.1.2 Arrays with Repeated Entries

Now we consider the case when the input array is filled with n numbers that may or may not be distinct.

Theorem 3. *Assume $\bar{x}$ is an array chosen from $\{1, 2, \ldots, n\}^n$ uniformly at random, and let C denote the random variable counting the number of comparisons carried out in the test whether $\bar{x}$ is sorted. Then*

$$E[C] = \sum_{k=1}^{n-1} \binom{n-1+k}{k} \left(\frac{1}{n}\right)^k \sim e - 1.$$

That is, $E[C]$ approaches $e - 1$ as $n \to \infty$.

Proof. The random variable C takes on a value of at least k, for $1 \leq k \leq n-1$, if the algorithm detects that the array is out of order after the kth comparison. In this case $\bar{x}$ takes a form where it starts with an

increasing sequence of numbers of length k, each chosen from $\{1, 2, \ldots, n\}$, and the rest of the array can be filled up arbitrarily. So, the form of $\bar{x}$ can be represented as

$$\underbrace{1^{t_1}2^{t_2}\ldots n^{t_n}}_{k}\underbrace{*\ldots*}_{n-k}$$

where $t_1 + t_2 + \ldots + t_n = k$ and $t_i \geq 0$, for $1 \leq i \leq n$. So to find a form for the expected value, we need to determine how many ways an integer k can be expressed as a sum of n nonnegative integers. To see this, we appeal to intuition with the following argument.

Imagine that there are k pebbles lined up in a row. If we put $n-1$ sticks between the pebbles, we will have partitioned them into n groups of pebbles, each with a nonnegative amount of pebbles. Note that we may have sticks next to each other, meaning some groups of pebbles are empty. Another way to look at this scenario is that there are $n-1+k$ spots, and we must pick $n-1$ of these spots to contain sticks. This is equivalent to choosing k pebbles. Therefore, there are $\binom{n-1+k}{k}$ correct arrangements for the first k elements of our array, and since we have n choices for each of the $n-k$ remaining elements, there are n^{n-k} possible arrangements for the remaining elements. So the number of arrays with the first k elements sorted is the quantity $\binom{n-1+k}{k}n^{n-k}$, and

$$P[C \geq k] = \binom{n-1+k}{k}\left(\frac{1}{n}\right)^k$$

which we obtain by dividing the number of arrays that have at least their first k elements sorted, by the total number of arrays possible in $\{1, 2, \ldots, n\}^n$, which is n^n. Using the same "trick" as in Theorem 2 where we encode the probability that the kth element is sorted with the indicator variable I_k, which is either 1 or 0, we can apply the definition of expected value to get

$$E[C] = \sum_{k=1}^{n-1} P[C \geq k] = \sum_{k=1}^{n-1}\binom{n-1+k}{k}\left(\frac{1}{n}\right)^k \tag{1}$$

$$= \left[\sum_{k=0}^{\infty}\binom{n-1+k}{k}x^k\right] - \left[\sum_{k=n}^{\infty}\binom{n-1+k}{k}x^k\right] - 1 \tag{2}$$

where $x = 1/n$ in the final expression. Next, we consider both of these sums in (2). By elementary calculus on generating functions we have the identity

$$\sum_{k=0}^{\infty}\binom{n-1+k}{k}x^k = \frac{1}{(1-x)^n} \tag{3}$$

which, when we substitute back in $x = 1/n$, the right side becomes $\left(\frac{n}{n-1}\right)^n$. Now we would like to obtain a nice expression for the other sum in (2). Unfortunately this is a little more difficult, but nonetheless manageable. First re-index the sum so that it becomes

$$\sum_{k=n}^{\infty}\binom{n-1+k}{k}x^k = \sum_{k=0}^{\infty}\binom{2n-1+k}{n+k}x^{n+k}.$$

Now we can estimate this sum by noting that

$$\binom{2n-1+k}{k} \leq 2^{2n-1+k}.$$

So we obtain a bound of the form

$$\sum_{k=0}^{\infty}\binom{2n-1+k}{n+k}x^{n+k} \leq \sum_{k=0}^{\infty}2^{2n-1+k}x^{n+k} = 2^{2n-1}x^n\sum_{k=0}^{\infty}(2x)^k.$$

This is a geometric series valid for $x < \frac{1}{2}$, or $n > 2$, so we can substitute in its closed form expression to get

$$\sum_{k=0}^{\infty} \binom{2n-1+k}{n+k} x^{n+k} \leq 2^{2n-1} x^n \frac{1}{1-2x} = \frac{1}{2} \left(\frac{4}{n}\right)^n \frac{n}{n-2}.$$

So using (2) and (3) we can bound $E[C]$ in the following way

$$\left(\frac{n}{n-1}\right)^n - \frac{1}{2}\left(\frac{4}{n}\right)^n \frac{n}{n-2} - 1 \leq E[C] \leq \left(\frac{n}{n-1}\right)^n - 1.$$

Taking limits of both the upper and lower bounds as $n \to \infty$ is an elementary calculation, and we see that both quantities approach $e-1$, so $E[C]$ approaches $e-1$ as n grows large. □

6.2 Computing the Expected Number of Swaps in Bogo-sort

We have already remarked that in the case when $\bar{x}$ is already sorted when it is provided as an input to bogo-sort, the algorithm runs in linear time, and the expected number of swaps is 0. The following theorem deals with the case where the initial array is not already sorted.

Theorem 4. *If S denotes the total number of swaps carried out for an input $\bar{x}$ of length n, where all the elements are distinct, then*

$$E[S] = \begin{cases} 0 & \text{if } \bar{x} \text{ is sorted} \\ (n-1)n! & \text{otherwise.} \end{cases}$$

Proof. In each iteration of the while-loop in bogo-sort, the array is permuted uniformly at random, and this will be done until we eventually permute all of the elements into an order which is increasing. In the case where all the elements are distinct, there is only one such ordering, out of $n!$ possible arrangements. So the probability of being sorted after one shuffle is $(n!)^{-1}$. This behavior is modeled by a geometric distribution, so applying the formula we obtain

$$P[I = i] = \left(\frac{n! - 1}{n!}\right)^i \cdot \frac{1}{n!}$$

where I is a geometrically distributed random variable with probability $p = (n!)^{-1}$ of being the iteration where we produce a sorted array. By the formula given in the previous definition of geometric distributions, this implies that $E[I] = p^{-1} = n!$.

For every iteration, we perform exactly $n-1$ swaps, and these are the only swaps performed by bogo-sort. Let S denote the random variable counting the number of swaps, and we see that $S = (n-1)I$. By linearity of expectation, we get that $E[S] = (n-1)n!$ in the case when $\bar{x}$ is not initially sorted. □

This result lends itself to a more applicable corollary. Recall that we are quantifying the algorithm's performance in terms of the number of swaps and comparisons it performs. The less swaps performed, the better we are doing.

Corollary 6. *If S denotes the total number of swaps carried out for an input $\bar{x}$ of length n, we have*

$$E[S] = \begin{cases} 0 & \text{in the best case,} \\ (n-1)n! & \text{in the worst and average case.} \end{cases}$$

6.3 Computing the Expected Number of Comparisons in Bogo-sort

In this section we count the expected number of comparisons that bogo-sort performs in the best, worst, and average case. In order to reason about this, we will think about the number of iterations bogo-sort makes, denoting this with the random variable I. An iteration includes one instance of permuting the elements of

the input array $\bar{x}$, and then one instance of checking to see if $\bar{x}$ is sorted. Let $\{C_i\}_{i\geq 0}$ be a sequence of random variables where C_i denotes the number of comparisons performed on the ith iteration. Then the total number of comparisons C for an input $\bar{x}$ is given by the sum

$$C = \sum_{i=0}^{I} C_i.$$

To study the expected value of this quantity, we will use Wald's Equation [4]. For convenience, we restate the theorem.

Theorem 7 (Wald's Equation). *Let $\{X_i\}_{0\leq i\leq N}$ be a sequence of independent, identically distributed random variables, and let the number of these random variables N be a random variable. Then*

$$E[X] = E[X_0 + X_1 + \ldots + X_N] = E[X_0] \cdot E[N].$$

This statement of Wald's Equation could be relaxed by removing the need for the variables to be identically distributed, but this allows us to write the right hand side of the equation above as $E[X_i] \cdot E[N]$ for any $0 \leq i \leq N$, since the X_i's in the theorem will correspond to the C_i's in our analysis of bogo-sort, which are each identically distributed. With Wald's Equation firmly in our toolbox, we can compute the expected number of comparisons.

Theorem 8. *Let C denote the number of comparisons carried out by bogo-sort on an input $\bar{x}$ of length n with distinct elements, and let $c(\bar{x})$ denote the number of comparisons needed by the algorithm to check $\bar{x}$ for being sorted. Then*

$$E[C] = \begin{cases} c(\bar{x}) = n-1 & \text{if } \bar{x} \text{ is sorted,} \\ c(\bar{x}) + (e-1)n! - O(1) & \text{otherwise.} \end{cases}$$

Proof. First we address the random variable C_0, since it has a different probability distribution from C_i for $i \geq 1$. Its value is determined by $\bar{x}$, in fact $P[C_0 = c(\bar{x})] = 1$. So by linearity of expectation, $E[C] = c(\bar{x}) + E[\sum_{i=1}^{I} C_i]$. In the case where $\bar{x}$ is initially sorted, the first C_0 comparisons are the only ones performed, so we're done. If $\bar{x}$ is not initially sorted, then sadly we must consider the latter sum, and attempt to produce meaningful results from it.

The random variables $\{C_i\}_{i\geq 1}$ are independent and identically distributed. I is a random variable, so we can apply Wald's Equation to get $E[\sum_{i=1}^{I} C_i] = E[C_1] \cdot E[I]$. After the first shuffle, we have a random permutation of the distinct elements of $\bar{x}$, and we test it for sortedness. Theorem 2 gives the expected value of $E[C_1] = e - 1 - O(\frac{1}{n!})$. We subtract off the $O(\frac{1}{n!})$ term to account for only a finite number of elements in $\bar{x}$ being compared. A consequence of the proof of Theorem 4 gives us that $E[I] = n!$. Multiplying these two quantities together a-la Wald's Equation gives us the desired result. □

Corollary 9. *Let C denote the number of comparisons carried out by bogo-sort on a given input $\bar{x}$ of length n with all its elements distinct. Then*

$$E[C] = \begin{cases} n-1 & \text{in the best case,} \\ (e-1)n! + n - O(1) & \text{in the worst case,} \\ (e-1)n! + O(1) & \text{in the average case.} \end{cases}$$

Proof. In the best case, the input array is already sorted, and so the total number of comparisons performed is $n-1$. In the worst case, $\bar{x}$ is not initially sorted, but we need $n-1$ comparisons to detect this at first. This would happen in the case when the array is completely sorted except for the last element. Applying Theorem 8, we obtain $E[C] = (e - 1 - O(\frac{1}{n!}))n! + n - 1$. For the average case, we apply Theorem 8 in the same way, except we substitute $n-1$ comparisons for the average number of comparisons $c(\bar{x}) = e-1-O(\frac{1}{n!})$ from Theorem 2. □

7 Conclusion

We analyzed the notoriously bad algorithm bogo-sort, and gave several expectations for the running time, quantified in terms of swaps and comparisons, of the algorithm under certain circumstances. These include:

- The expected number of comparisons bogo-sort performs on an array with distinct elements on a given iteration (Theorem 2).
- The expected number of comparisons bogo-sort performs on an array with possibly non-distinct elements on a given iteration (Theorem 3).
- The expected number of swaps bogo-sort performs on an array with distinct elements for its entire duration (Corollary 6).
- The expected number of comparisons bogo-sort performs on an array with distinct elements for its entire duration (Corollary 9).

In the average and worst case, bogo-sort is expected to carry out $O(n \cdot n!)$ swaps, and this is how its runtime is usually stated. In case it has been buried under the formalism of this paper, we dedicate this final part of the conclusion to ensuring that the reader truly grasps how terrible bogo-sort is at sorting.

If we make the generous assumption that bogo-sort performs exactly $n \cdot n!$ operations on an array of length n before terminating, and that each operation takes exactly one nanosecond (one billionth of one second), then how long can we expect to have to wait before an array of 20 elements is sorted? Given our assumptions, bogo-sort will spend 1.5 millennia sorting this array. How awful! [1] For a possibly more helpful metric, the popular sorting algorithm "quicksort" has an average runtime of $O(n \log n)$, and under the same assumptions as before, an array of 20 elements is expected to be sorted in approximately 60 nanoseconds. Stated differently: *bogo-sort is sort of slow.*

So unfortunately, the results of this paper may not wind up being referenced by millions of computer science students after all [2]. However, the methods used to analyze bogo-sort were just direct applications of elementary probability, and we have been offered a lot of "bang for our buck" in the sense that we have been able to prove a lot of results using a very limited pool of knowledge.

[1] An array of 100 elements will sort in approximately 10^{140} millennia. This is *really* bad.
[2] Fingers crossed!

References

[1] Charles M. Grinstead, and J. Laurie Snell, *Introduction to Probability*, 1998.

[2] Hermann Gruber, Markus Holzer, and Oliver Ruepp, *Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms*, 2007.

[3] E. S. Raymond, *The New Hacker's Dictionary*, 1996.

[4] Weisstein, Eric W. "Wald's Equation." From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/WaldsEquation.html`