Sorting the Slow Way: An Analysis of Perversely Awful Randomized Sorting Algorithms

Hermann Gruber[1] and Markus Holzer[2] and Oliver Ruepp[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität München,
Oettingenstraße 67, D-80538 München, Germany
email: gruberh@tcs.ifi.lmu.de
[2] Institut für Informatik, Technische Universität München,
Boltzmannstraße 3, D-85748 Garching bei München, Germany
email: {holzer,ruepp}@in.tum.de

Abstract. This paper is devoted to the "Discovery of Slowness." The archetypical perversely awful algorithm bogo-sort, which is sometimes referred to as Monkey-sort, is analyzed with elementary methods. Moreover, practical experiments are performed.

1 Introduction

To our knowledge, the analysis of perversely awful algorithms can be tracked back at least to the seminal paper on pessimal algorithm design in 1984 [2]. But what's a perversely awful algorithm? In the "The New Hacker's Dictionary" [7] one finds the following entry:

> **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**.

Among other solutions, the formerly mentioned work contains a remarkably slow sorting algorithm named *slowsort* achieving running time $\Omega\big(n^{\log n/(2+\epsilon)}\big)$ even in the best case. But the running time, still being sub-exponential, does not improve (i.e., increase) in the average case, and not even in the worst case. On the contrary, the analysis of bogo-sort carried out here shows that this algorithm, while having best-case expected running time as low as $\mathrm{O}(n)$, achieves an asymptotic expected running time as high as $\Omega(n \cdot n!)$ already in the average case. The pseudo code of bogo-sort reads as follows:

Algorithm 1 Bogo-sort

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

The test whether the array is sorted as well as the permutation of the array have to be programmed with some care:

```
1: procedure sorted: {returns true if the array is sorted and false otherwise}
2: for i = 1 to n − 1 do
3:    if a[i] > a[i + 1] then
4:       return false
5:    end if
6: end for
7: return true
8: end procedure
```

```
1: procedure randomly permute: {permutes the array}
2: for i = 1 to n − 1 do
3:    j := rand[i...n]
4:    swap a[i] and a[j]
5: end for
6: end procedure
```

The second algorithm is found, e.g., in [5, p.139]. Hence the random permutation is done quickly by a single loop, where **rand** gives a random value in the specified range. And the test for sortedness is carried out from left and right.

In this work we present a detailed analysis, including the exact determination of the expected number of comparisons and swaps in the best, worst and average case. Although there are some subtleties in the analysis, our proofs require only a basic knowledge of probability and can be readily understood by non-specialists. This makes the analysis well-suited to be included as motivating example in courses on randomized algorithms. Admittedly, this example does not motivate coursework on *efficient* randomized algorithms. But the techniques used in our analysis cover a wide range of mathematical tools as contained in textbooks such as [4].

We will analyze the expected running time for bogo-sort under the usual assumption that we are given an array $\overline{x} = x_1 x_2 \dots x_n$ containing a permutation of the set of numbers $\{1, 2, \dots, n\}$. In a more abstract fashion, we are given a list containing all elements of a finite set S with $|S| = n$ and an irreflexive, transitive and antisymmetric relation $\sqsubset$. 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. An immediate observation is that the algorithm isn't guaranteed to terminate at all. However, as we will prove that the *expectation* of the running time T is finite, we see by Markov's inequality

$$\mathbb{P}[T \geq t] \leq \frac{\mathbb{E}[T]}{t} \quad , \text{for } t > 0,$$

that the probability of this event equals 0. There are essentially two different initial configurations: Either the list $\overline{x}$ is initially sorted, or it is not sorted. We

have to make this distinction as the algorithm is smart enough to detect if the given list is initially sorted, and has much better running time in this case. This nice built-in feature also makes the running time analysis in this case very easy: The number of total comparisons equals $n-1$, and the total number of swaps equals zero, since the while-loop is never entered.

We come to the case where the array is not initially sorted. Note that the first shuffle yields a randomly ordered list, so the behavior of the algorithm does no longer depend on the initial order; but the number of comparisons before the first shuffle depends on the structure of the original input.

2 How long does it take to check an array for sortedness?

2.1 The basic case

We make the following important

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.*

This motivates us to study the running time of the subroutine for detecting if the list is sorted on the average:

Theorem 2. *Assume $\overline{x}$ 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 $\overline{x}$ is sorted. Then*

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

Proof. For $1 \leq k < n$, let I_k be the random variable indicating that (at least) the first k elements in $\overline{x}$ are in order. 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 left. 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 $x_1 < x_2 < \cdots < x_{k-1} < x_k$.

Thus, we have also $\mathbb{P}[C \geq k] = \mathbb{P}[I_k]$. This probability computes as

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

The numerator is the product of the number of possibilities to choose k first elements to be in correct order and the number of possibilities to arrange the remaining $n-k$ elements at the end of the array, and the denominator is just the total number of arrays of length n. Reducing this fraction, we obtain $\mathbb{P}[I_k] = \frac{1}{k!}$. As the range of C is nonnegative, we obtain for the expected value of C:

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

And it is a well-known fact from calculus that the last sum appearing in the above computation is the partial Taylor series expansion for e^x at $x = 1$. □

Wasn't that marvelous? Theorem 2 tells us that we need only a constant number of comparisons on the average to check if a large array is sorted, and for n large enough, this number is about $e - 1 \approx 1.72$. Compare to the worst case, where we have to compare $n - 1$ times.

2.2 A detour: Random arrays with repeated entries

In a short digression, we explore what happens if the array is filled not with n distinct numbers. At first glance we consider the case when n numbers in different multiplicities are allowed. Then we have a look at the case with only two distinct numbers, say 0 and 1. In the former case the expected number of comparisons remains asymptotically the same as in the previous theorem, while in the latter the expected number of comparisons jumps up dramatically.

Theorem 3. *Assume $\overline{x}$ 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 $\overline{x}$ is sorted. Then*

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

Proof. The random variable C takes on a value of at least k, for $1 \leq k \leq n-1$, if the algorithms detects that the array is out of order after the kth comparison. In this case $\overline{x}$ is of the form that it starts with an increasing sequence of numbers chosen from $\{1, 2, \ldots, n\}$ of length k, and the rest of the array can be filled up arbitrarily. Thus, the form of $\overline{x}$ can be illustrated as follows:

$$\underbrace{1^{t_1}2^{t_2}\ldots n^{t_n}}_{k}\underbrace{*\ldots*}_{n-k} \quad \text{with } t_1 + t_2 + \ldots + t_n = k \text{ and } t_i \geq 0, \text{ for } 1 \leq i \leq n.$$

Hence we have to determine how many ways an integer k can be expressed as sum of n non-negative integers. Image that there are k pebbles lined up in a row. Then if we put $n - 1$ sticks between them we will have partitioned them into n groups of pebbles each with a non-negative amount of marbles. So we have basically $n - 1 + k$ spots, and we are choosing $n - 1$ of them to be the sticks—this is equivalent to choosing k marbles. Therefore the number of arrays of this form is $\binom{n-1+k}{k} n^{n-k}$, and $\mathbb{P}[C \geq k] = \binom{n-1+k}{k} \left(\frac{1}{n}\right)^k$, as there is a total of n^n arrays in $\{1, 2, \ldots, n\}^n$. But then

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

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

Next let us consider both infinite sums in more detail. By elementary calculus on generating functions we have for the first sum

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

which in turn gives $(\frac{n}{n-1})^n$ because $x = \frac{1}{n}$ and by juggling around with double fractions. It remains to consider the second sum. Shifting the index n places left gives us a more convenient form for the second sum:

$$\sum_{k=0}^{\infty} \binom{2n-1+k}{k+n} \cdot x^{k+n} = x^n \sum_{k=0}^{\infty} \binom{2n-1+k}{k+n} \cdot x^k \tag{4}$$

Doesn't look that bad. As the coefficients of this power series are binomial coefficients, there might be quite a good chance that this sum can be expressed as a (generalized) hypergeometric function. In general, a hypergeometric function is a power series in x with $r+s$ parameters, and it is defined as follows in terms of rising factorial powers:

$$F\left[\begin{array}{c} a_1, a_2, \ldots, a_r \\ b_1, b_2, \ldots, b_s \end{array} \middle| x \right] = \sum_{k\geq 0} \frac{a_1^{\overline{k}} a_2^{\overline{k}} \ldots a_r^{\overline{k}}}{b_1^{\overline{k}} b_2^{\overline{k}} \ldots b_s^{\overline{k}}} \cdot \frac{x^k}{k!}.$$

In order to answer this question we have to look at the ratio between consecutive terms—so let the notation of the series be $\sum_{k\geq 0} t_k \cdot \frac{x^k}{k!}$ with $t_0 \neq 0$. If the term ratio t_{k+1}/t_k is a rational function in k, that is, a quotient of polynomials in k of the form

$$\frac{(k+a_1)(k+a_2)\ldots(k+a_r)}{(k+b_1)(k+b_2)\ldots(k+b_s)}$$

then we can use the ansatz

$$\sum_{k\geq 0} t_k \cdot \frac{x^k}{k!} = t_0 \cdot F\left[\begin{array}{c} a_1, a_2, \ldots, a_r \\ b_1, b_2, \ldots, b_s \end{array} \middle| x \right].$$

So let's see whether we are lucky with our calculations. As $t_k = \binom{2n-1+k}{k+n} \cdot k!$, the first term of our sum is $t_0 = \binom{2n-1}{n}$, and the other terms have the ratios given by

$$\frac{t_{k+1}}{t_k} = \frac{(2n+k)!(k+n)!(n-1)!(k+1)!}{(n+k+1)!(n-1)!(2n-1+k)!k!} = \frac{(k+2n)(k+1)}{(k+n+1)},$$

which are rational functions of k, yeah .... Thus, the second sum equals

$$\begin{aligned} \sum_{k=0}^{\infty} \binom{2n-1+k}{k+n} \cdot x^k &= \binom{2n-1}{n} \cdot F\left[\begin{array}{c} 2n, 1 \\ n+1 \end{array} \middle| x \right] \\ &= \frac{1}{2} \cdot \binom{2n}{n} \cdot F\left[\begin{array}{c} 2n, 1 \\ n+1 \end{array} \middle| x \right], \end{aligned} \tag{5}$$

because $\binom{2n-1}{n} = \frac{(2n-1)!}{n!(n-1)!} = \frac{n}{2n} \cdot \frac{(2n)!}{n!n!} = \frac{1}{2} \cdot \binom{2n}{n}$. This looks much nicer, and it's even a Gaussian hypergeometric function, i.e., $r = 2$ and $s = 1$. What about a closed form for $F(\,1, 2n; n+1 \mid x\,)$? Supercalifragilisticexpialidoceous[1] .... That's fresh meat for the Gosper-Zeilberger algorithm. Next the fact

$$2S_x(n)x(x-1)(2n-1) + nS_x(n-1) = 0,$$

where $S_x(n)$ is the indefinite sum $\sum_{k=-\infty}^{\infty} \frac{(2n)^{\overline{k}}(1)^{\overline{k}}}{(n+1)^{\overline{k}}} \cdot \frac{x^k}{k!}$, can be easily verified with a symbolic computation software at at hand.[2] Hence the sum is Gosper-Zeilberger summable. Ah, ... Maybe it's worth a try to check whether the original sum given in Equation (1) is Gosper-Zeilberger summable as well. Indeed, with a similar calculation as above we obtain

$$(x-1)S_x(n) + S_x(n-1) = 0,$$

where $S_x(n)$ now equals $\sum_{k=-\infty}^{\infty} \binom{n-1+k}{k} \cdot x^k$. That's even nicer than above. Since we don't remember all details of the Gosper-Zeilberger algorithm by heart, we peek into a standard book like, e.g., [4]. Wow, ... our sum (with slight modifications) from Equation (1) is already "solved"—[4, page 236]: The recurrence for the definite sum $s_x(n) = \sum_{k=0}^{n-1} \binom{n-1+k}{k} \cdot x^k$—note that $\mathbb{E}[C] = s_{1/n}(n) - 1$—reads as

$$s_x(n) = \frac{1}{1-x}\left(s_x(n-1) + (1-2x)\binom{2n-3}{n-2} \cdot x^{n-1}\right).$$

Because $s_x(1) = 1$, we can solve the recurrence and obtain

$$s_x(n) = \frac{1}{(1-x)^{n-1}} + (1-2x)\sum_{k=1}^{n-1} \binom{2k-1}{k-1} \cdot \frac{x^k}{(1-x)^{n-k}}. \tag{6}$$

[1] According to Pamela L. Travers' "Mary Poppins" it is a very important word everybody should know—see, e.g., [6]:

Jane: Good morning, father. Mary Poppins taught us the most wonderful word.
Michael: Supercalifragilisticexpialidocious.
George W. Banks: What on Earth are you talking about? Superca - Super - or whatever the infernal thing is.
Jane: It's something to say when you don't know what to say.
George W. Banks: Yes, well, I always know what to say.

[2] The actual computation is done by Maple's hsum-package as follows:

```
> read "hsum10.mpl";
    Package "Hypergeometric Summation", Maple V - Maple 10
   Copyright 1998-2006, Wolfram Koepf, University of Kassel
> sumrecursion(hyperterm([1, 2*n], [n + 1], x, k), k, S(n));
      2 (2 n + 1) (x - 1) x S(n + 1) + (n + 1) S(n) = 0
```

Here `S(n)` plays the role of $S_x(n)$. Moreover, we have shifted the index n one to the right to obtain the above mentioned recurrence.

Unfortunately this "closed form" is more complicated than the original sum.[3] So we are happier with Equation (1) as a solution.

What about the asymptotic behaviour for $x = \frac{1}{n}$ and growing n. For both Equations (5) and (6) taking limits is no fun at all, in particular for the respective second terms! But still we are lucky, because it is not too hard to give an estimate for $x^n \sum_{k=0}^{\infty} \binom{2n-1+k}{k+n} \cdot x^k$ from Equation (4) by noting that $\binom{2n-1+k}{k+n} \leq 2^{2n-1+k}$. So this sum is upper-bounded by a geometric series: $x^n 2^{2n-1} \sum_{k=0}^{\infty} (2x)^k = x^n 2^{2n-1} \frac{1}{1-2x}$, which is valid for $x < 1/2$. For $n > 2$, we can plug in $x = 1/n$, and get $\sum_{k=n}^{\infty} \binom{2n-1+k}{k+n} (1/n)^{n+k} \leq \frac{1}{2} \left(\frac{4}{n}\right)^n$, and this even holds for $n \geq 2$. Thus we have

$$\left(\frac{n}{n-1}\right)^n - \frac{1}{2}\left(\frac{4}{n}\right)^n - 1 \leq \mathbb{E}[C] \leq \left(\frac{n}{n-1}\right)^n - 1. \tag{7}$$

Since $\left(\frac{4}{n}\right)^n$ tends to 0 as n grows and $\left(\frac{n}{n-1}\right)^n \sim e$, we see that $\mathbb{E}[C]$, the expectation of C, is asymptotically $e - 1$. □

The behavior of (the analytic continuations of) these two functions is compared in Figure 1. We turn to the binary case, which again turns out to be easier.

Theorem 4. *Assume $\overline{x}$ is an array chosen from $\{0,1\}^n$ uniformly at random, and let C denote the random variable counting the number of comparisons carried out in the test whether $\overline{x}$ is sorted. Then*

$$\mathbb{E}[C] = 3 - (2n+4)2^{-n} \sim 3.$$

Proof. Assume $k \in \{1, 2, \ldots, n-2\}$. If C takes on the value k, then the algorithm detects with the kth comparison that the array is out of order. Thus $\overline{x}$ must be of a special form: it starts with a number of 0s, then follows a nonempty sequence of 1s, which is again followed by a 0 at index $k+1$. The rest of the array can be filled up arbitrarily with zeroes and ones. This can be illustrated as follows:

$$\underbrace{0\ldots0}_{\ell}\underbrace{1\ldots1}_{k-\ell>0}0\underbrace{*\ldots*}_{n-k-1}.$$

Counting the number of arrays of this form, we obtain: $\sum_{\ell=0}^{k-1} 2^{n-k-1} = k2^{n-k-1}$, and $\mathbb{P}[C = k] = k\left(\frac{1}{2}\right)^{k+1}$, as there is a total of 2^n arrays in $\{0,1\}^n$.

The remaining case is that the number of comparisons equals $n-1$. In this case, either the array is sorted, or x has the following form:

$$\underbrace{0\ldots0}_{\ell}\ \underbrace{1\ldots1}_{n-1-\ell>0}\ 0.$$

[3] Our detour on hypergeometric functions was not useless because by combining Equations (2), (5), and (6) and evaluating at $x = \frac{1}{2}$ results in the quaint hypergeometric identity $\binom{2n}{n} F\left[\begin{matrix} 2n, 1 \\ n+1 \end{matrix} \middle| \frac{1}{2}\right] = 2^{2n}$, for integers $n \geq 2$.

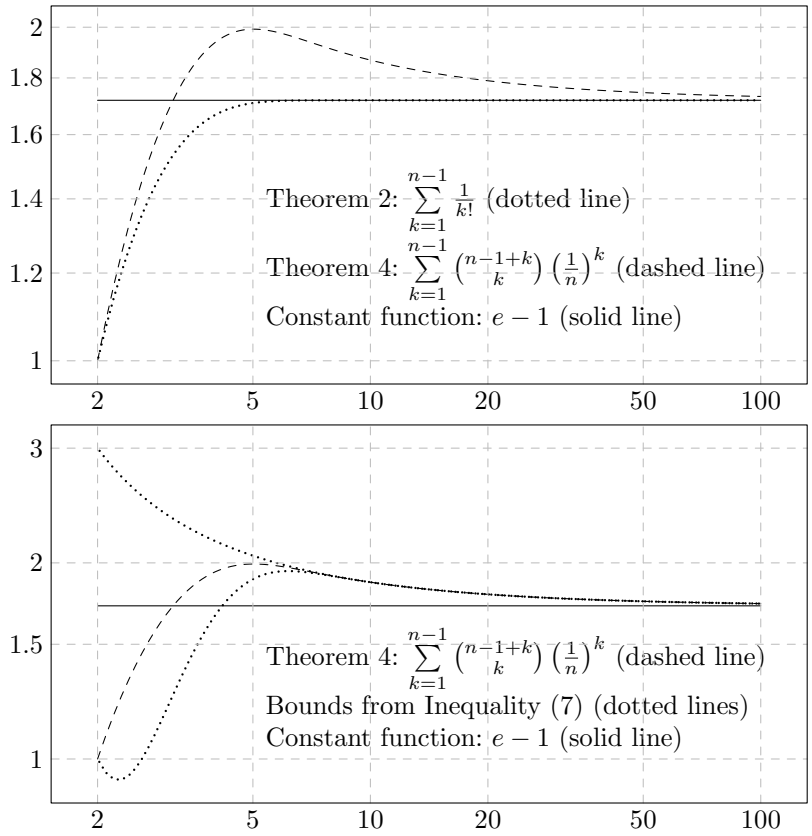

Fig. 1. The functions on the number of expected number of comparisons from Theorems 2 and 3 compared with the constant $e-1$.

The set $\{0,1\}^n$ contains exactly $n+1$ sorted arrays, and the number of arrays of the second form clearly equals $n-1$. Thus we have $\mathbb{P}[C=n-1]=2n2^{-n}$. Now we are ready to compute the expected value as

$$\mathbb{E}[C]=\sum_{k=1}^{n-2} k^2\left(\frac{1}{2}\right)^{k+1}+(n-1)\mathbb{P}[C=n-1]=\frac{1}{2}\left(\sum_{k=1}^{n-2} k^2x^k\right)+(2n^2-2n)2^{-n},$$

for $x=\frac{1}{2}$. Next, the fact

$$(x-1)^3\cdot\sum_{k=1}^{m} k^2x^k=m^2x^{m+3}-2m(m-1)x^{m+2}+(m+1)^2x^{m+1}-x^2-x$$

can be easily verified if we have a symbolic computation software at hand. Then we briskly compute $\sum_{k=1}^{n-2} k^2\left(\frac{1}{2}\right)^k=6-(4n^2+8)2^{-n}$, and finally get it: $\mathbb{E}[C]=3-(2n+4)2^{-n}$. □

We can use a similar approach to determine the expected value in the setup where the array is drawn uniformly at random from all arrays with a fixed number of zeroes, but it apparently cannot be expressed in a neat form as above. As we feel that ugly expressions are outside the scope of this conference, we refuse to further report on this here.

2.3 The expected number of swaps in bogo-sort

When computing the expected number of iterations in bogo-sort, we concentrate on the case where the input $\overline{x}$ is not sorted; for the other case it equals 0, because of the intelligent design of the algorithm. In each iteration, the array is permuted uniformly at random, and we iterate until we hit the ordered sequence for the first time. As the ordered sequence is hit with probability $\frac{1}{n!}$ in each trial, the number of iterations I is a random variable with

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

That is, I is a geometrically distributed random variable with hitting probability $p = \frac{1}{n!}$, and $\mathbb{E}[I] = p^{-1} = n!$

In each iteration, the array is shuffled; and a shuffle costs $n-1$ swaps. As the algorithm operates kind of economically with respect to the number of swaps, these are *the only* swaps carried out while running the algorithm. If S denotes the random variable counting the number of swaps, we have $S = (n-1) \cdot I$. By linearity of expectation, we derive:

Theorem 5. *If S denotes the total number of swaps carried out for an input $\overline{x}$ of length n, we have*

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

Corollary 6. *Let S denote the number of swaps carried out by bogo-sort on a given input $\overline{x}$ of length n. Then*

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

2.4 The expected number of comparisons in bogo-sort

Now suppose that, on input $\overline{x}$, we iterate the process of checking for sortedness and shuffling eternally, that is we do not stop after the array is eventually sorted. We associate a sequence of random variables $(C_i)_{i \geq 0}$ with the phases of this process, where C_i counts the number of comparisons before the ith shuffle. Recall

the random variable I denotes the number of iterations in bogo-sort. Then the total number of comparisons C in bogo-sort on input $\overline{x}$ is given by the sum

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

Wait ... This is a sum of random variables, where the summation is eventually stopped, and the time of stopping is again a random variable, no? No problem. We can deal with this rigorously.

Definition 7. *Let $(X_i)_{i\geq 1}$ be a sequence of random variables with $\mathbb{E}[X_i] < \infty$ for all $i \geq 1$. The random variable N is called a* stopping time *for the sequence $(X_i)_{i\geq 1}$, if $\mathbb{E}[N] < \infty$ and, $\mathbb{1}_{(N\leq n)}$ is stochastically independent from $(X_i)_{i>n}$, for all n.*

For the concept of stopping times, one can derive a useful (classical) theorem, termed Wald's Equation. For the convenience of the reader, we include a proof of this elementary fact.

Theorem 8 (Wald's Equation). *Assume $(X_i)_{i\geq 1}$ is a sequence of independent, identically distributed random variables with $\mathbb{E}[X_1] < \infty$, and assume N is a stopping time for this sequence. If $S(n)$ denotes the sum $\sum_{i=0}^{n} X_i$, then*

$$\mathbb{E}[S(N)] = \mathbb{E}[X_1] \cdot \mathbb{E}[N].$$

Proof. We can write $S(n)$ equivalently as $\sum_{i=1}^{\infty} X_i \cdot \mathbb{1}_{(N\geq i)}$ for the terms with $i > N$ are equal to zero, and the terms with $i \leq N$ are equal to X_i. By linearity of expectation, we may write $\mathbb{E}[S(n)]$ as $\sum_{i=1}^{\infty} \mathbb{E}[X_i \cdot \mathbb{1}_{(N\geq i)}]$. Next, observe that X_i and $\mathbb{1}_{(N\geq i)}$ are stochastically independent: Since N is a stopping time, X_i and $\mathbb{1}_{(N\leq i-1)}$ are independent. But the latter is precisely $1 - \mathbb{1}_{(N\geq i)}$. Thus we can express the expectation of $X_i \cdot \mathbb{1}_{(N\geq i)}$ as product of expectations, namely as $\mathbb{E}[X_i] \cdot \mathbb{E}[\mathbb{1}_{(N\geq i)}]$. And finally, as the X_i are identically distributed, we have $\mathbb{E}[X_i] = \mathbb{E}[X_1]$. Putting these together, we get

$$\mathbb{E}[S(n)] = \sum_{i=1}^{\infty} \mathbb{E}[X_1] \cdot \mathbb{E}[\mathbb{1}_{(N\geq i)}] = \sum_{i=1}^{\infty} \mathbb{E}[X_1]\mathbb{P}[N \geq i] = \mathbb{E}[X_1] \cdot \mathbb{E}[N],$$

as $\mathbb{E}[\mathbb{1}_{(N\geq i)}] = \mathbb{P}[N \geq i]$ and $\mathbb{E}[N] = \sum_{i=1}^{\infty} \mathbb{P}[N \geq i]$. □

Now we have developed the tools to compute the expected number of comparisons:

Theorem 9. *Let C denote the number of comparisons carried out by bogo-sort on an input $\overline{x}$ of length n, and let $c(\overline{x})$ denote the number of comparisons needed by the algorithm to check $\overline{x}$ for being sorted. Then*

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

Proof. The random variable C_0 has a probability distribution which differs from that of C_i for $i \geq 1$, but its value is determined by $\overline{x}$, that is $\mathbb{P}[C_0 = c(\overline{x})] = 1$. By linearity of expectation, $\mathbb{E}[C] = c(\overline{x}) + \mathbb{E}[\sum_{i=1}^{I} C_i]$. For the latter sum, the random variables $(C_i)_{i\geq 1}$ are independent and identically distributed. And I is indeed a stopping time for this sequence because the time when the algorithm stops does not depend on future events. Thus we can apply Wald's equation and get $\mathbb{E}[\sum_{i=1}^{I} C_i] = \mathbb{E}[C_1] \cdot \mathbb{E}[I]$. After the first shuffle, we check a random array for being sorted, so with Theorem 2 and the following remark holds $\mathbb{E}[C_1] = e - 1 - \mathrm{O}\big(\frac{1}{n!}\big)$. The left inequality follows by an easy induction. And recall from Section 2.3 that $\mathbb{E}[I] = n!$. □

Corollary 10. *Let C denote the number of comparisons carried out by bogo-sort on a given input $\overline{x}$ of length n. Then*

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

Proof. In the best case, the input array $\overline{x}$ is already sorted, and thus the total number of comparisons equals $n-1$. In the worst case, $\overline{x}$ is not initially sorted, but we need $n-1$ comparisons to detect this. Putting this into Theorem 9, we obtain $\mathbb{E}[C] = \big(e - 1 - \mathrm{O}\big(\frac{1}{n!}\big)\big)\, n! + n - 1$. For the average case, recall in addition that $c(\overline{x}) = e - 1 - \mathrm{O}\big(\frac{1}{n!}\big)$ holds for an average input $\overline{x}$ by Theorem 2. □

3 Variations and optimizations

3.1 A variation: bozo-sort

We can generalize the template of repeated testing and shuffling by using other shuffling procedures than the standard shuffle. For instance, the set of transpositions, or swaps, generates the symmetric group S_n. Thus one can think of the following variation of bogo-sort, named bozo-sort: After each test if the array is ordered, two elements in the array are picked uniformly at random, and swapped. The procedure is iterated until the algorithm eventually detects if the array is sorted.

Algorithm 2 Bozo-sort

```
1: Input array a[1 ... n]
2: while a[1 ... n] is not sorted do
3:     randomly transpose a[1 ... n]
4: end while
```

We note that this specification is ambiguous, and two possible interpretations are presented in pseudo-code:

```
1: procedure rand. transpose:
   {swaps two elements chosen
   independently}
2: i := rand[1 ... n]
3: j := rand[1 ... n]
4: swap a[i] and a[j]
5: end procedure
```

```
1: procedure rand. transpose:
   {swaps a random pair }
2: i := rand[1 ... n]
3: j := rand[1 ... i − 1, i + 1, ... n]
4: swap a[i] and a[j]
5: end procedure
```

We refer to the variant on the left as bozo-sort and to the right variant as bozo-sort$^+$. Note the apparent difference to bogo-sort: This time there are permutations of $\overline{x}$ which are not reachable from $\overline{x}$ with a single exchange, and indeed there are inputs for which the algorithm needs at least $n-1$ swaps, no matter how luckily the random elements are chosen.

We conclude that the respective process is not stateless. But it can be suitably modeled as a finite Markov chain having $n!$ states. There each state corresponds to a permutation of $\overline{x}$. For bozo-sort$^+$, transition between a pair of states happens with probability $1/\binom{n}{2}$ if the corresponding permutations are related by a transposition. The expected hitting time of the sorted array on n elements for this Markov chain was determined using quite some machinery in [3]. Translated to our setup, the relevant result reads as:

Theorem 11 (Flatto/Odlyzko/Wales). *Let S denote the number of swaps carried out by bozo-sort$^+$ on an input $\overline{x}$ of length n. Then*

$$\mathbb{E}[S] = n! + 2(n-2)! + \mathrm{o}((n-2)!)$$

in the average case.

The expected number of swaps in the best case is clearly 0, but we do not know it in the worst case currently. The expected number of comparisons is still more difficult to analyze, though it is easy to come up with preliminary upper and lower bounds:

Theorem 12. *Let C denote the number of comparisons carried out by bozo-sort$^+$ on an input $\overline{x}$ of length n. Then*

$$n! + 2(n-2)! + \mathrm{o}((n-2)!) \leq \mathbb{E}[C] \leq (n-1)n! + 2(n-1)! + \mathrm{o}((n-1)!)$$

in the average case.

Proof. We can express the number of comparisons as a sum of random variables as in Section 2.4: If I denotes the number of iterations on an input $\overline{x}$ chosen uniformly at random, and C_i the number of iterations before the ith swap, then the total number C of comparisons equals $C = \sum_{i=0}^{I} C_i$. Obviously $1 \leq C_i \leq n-1$, and thus $\mathbb{E}[S] \leq \mathbb{E}[C] \leq (n-1)\mathbb{E}[S]$ by linearity of expectation. □

The results obtained in Section 2.4 even suggest that the expected total number of comparisons on the average is as low as $\mathrm{O}(n!)$. This would mean that the running time of bogo-sort outperforms (i.e. is higher than) the one of

bozo-sort on the average. In particular, we believe that bozo-sort has the poor expected running time of only $O(n!)$ in the average case. Compare to bogo-sort, which achieves $\Omega(n \cdot n!)$.

Conjecture 13. For arrays with n elements, the expected number of comparisons carried out by bozo-sort$^+$ is in $\Theta(n!)$ in the average case, as n tends to infinity.

3.2 Comments on optimized variants of bogo-sort

Though optimizing the running time seems somewhat out of place in the field of *pessimal* algorithm design, it can be quite revealing for beginners in both fields of optimal *and* pessimal algorithm design to see how a single optimization step can yield a dramatic speed-up. The very first obvious optimization step in all aforementioned algorithms is to swap two elements only if this makes sense. That is, before swapping a pair, we check if it is an inversion: A pair of positions (i, j) in the array $a[1 \ldots n]$ is an *inversion* if $i < j$ and $a[i] > a[j]$. This leads to optimized variants of bogo-sort and its variations, which we refer to as bogo-sort$_{\text{opt}}$, bozo-sort$_{\text{opt}}$, and bozo-sort$^+_{\text{opt}}$, resp. As there can be at most $\binom{n}{2}$ inversions, this number gives an immediate upper bound on the number of swaps for these variants—compare, e.g., to $\Omega(n \cdot n!)$ swaps carried out by bogo-sort. It is not much harder to give a similar upper bound on the expected number of iterations. As the number of comparisons during a single iteration is in $O(n)$, we also obtain an upper bound on the expected total number of comparisons:

Lemma 14. *The expected number of iterations (resp. comparisons) carried out by the algorithms bogo-sort$_{\text{opt}}$, bozo-sort$_{\text{opt}}$, and bozo-sort$^+_{\text{opt}}$ on a worst-case input $\overline{x}$ of length n is at most $O\big(n^2 \log n\big)$ (resp. $O\big(n^3 \log n\big)$).*

Thus a single optimization step yields *polynomial* running time for all of these variants. The proof of the above lemma, which is based on the coupon collectors' problem, is elementary and well-suited for education. A very similar fact is shown in [1], so the details are omitted. Besides, variations of bozo-sort based on this optimization have been studied in [1]: A further optimization step is to run the procedure sorted only after every nth iteration, which results in the algorithm *guess-sort*, designed and analyzed in the mentioned work.

4 Experimental results

We have implemented the considered algorithms in C and have performed some experiments. The source code as well as the test scripts are available on request by email to one of the authors. The experiments were conducted on our lab pool, roughly 10 PCs AMD Athlon XP 2400+ and Intel Pentium 4 CPU 3.20 GHz with 3 to 4 GB RAM. It took quite some time to collect our results, but this was no problem, since the lab courses start in late February and the PCs were idle anyway. The results are shown in Figure 2, for the number swaps and comparisons for the bogo-sort and both bozo-sort variants. For the values

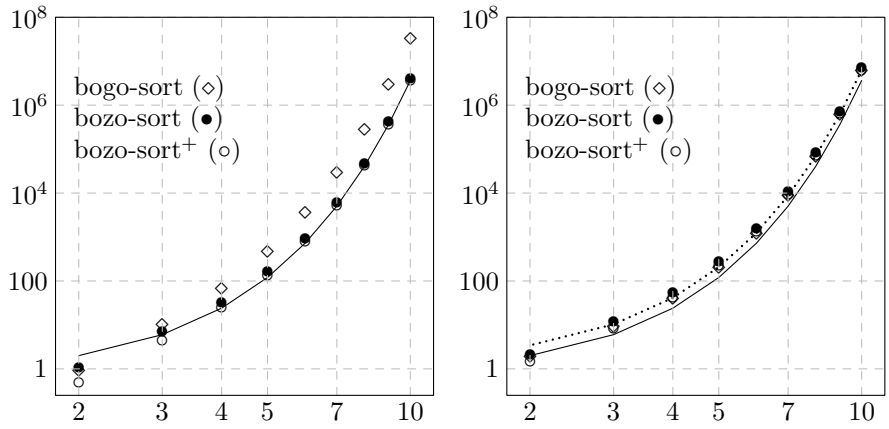

Fig. 2. Expected number of swaps (left) and comparisons (right) for the three considered randomized sorting algorithms—both axes are logarithmically scaled. The factorial function is drawn as a solid line, while the factorial times $(e-1)$ is drawn as dotted line.

$n = 2, 3, \ldots, 6$ all $n!$ permutations were sorted more than 1000 times. For the remaining cases $n = 7, 8, 9, 10$ only $6! \cdot 1000$ randomly generated permutations were sorted. The average values depicted in the diagrams nicely fit the theoretical results. Moreover, our conjecture on the number of comparisons carried out by bozo-sort$^+$ is supported by the given data. We can also conclude from the data that in practice bogo-sort outperforms, i.e., is slower than, the bozo-sort variants w.r.t. the number of swaps by a linear factor, whereas all variants perform equally good w.r.t. the number of comparisons. This is somehow counter-intuitive since one may expect at first glance that the bozo-sorts are slower.

Finally, we have evaluated the performance of optimized variants of bogo-sort and bozo-sort empirically on the same data-set as described above. The data in Figure 3 suggests that the upper bounds on the expected running time we obtained are probably not sharp and can be improved. In particular, we experience that the optimized variant of bogo-sort performs considerably less comparisons than the appropriate counterparts bozo-sort$_{\text{opt}}$ and bozo-sort$^+_{\text{opt}}$.

5 Conclusions

We contributed to the field of pessimal algorithm design with a theoretical and experimental study of the archetypical perversely awful algorithm, namely bogo-sort. Remarkably, the expected running time in terms of the number of swaps and comparisons can be determined exactly using only elementary methods in probability and combinatorics. We also explored some variations on the theme: In Section 2.2, we determined the number of comparisons needed to detect sortedness on the average in different setups. And in Section 3, we introduced two variants of bogo-sort which are based on random transpositions. The analysis of

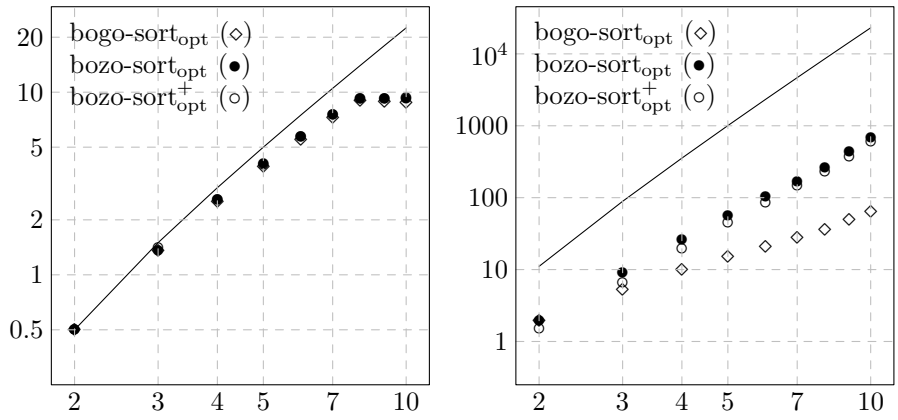

Fig. 3. Expected number of swaps (left) and comparisons (right) for the three considered optimized randomized sorting algorithms—both axes are logarithmically scaled. Both, the function $\frac{1}{4}n(n-1)$ (left), which is the number of expected inversions, and $n^4 \log n$ (right) are drawn as solid lines.

these variants seems to bear far more difficulties. There our results essentially rely on a technical paper on random walks on finite groups. Quite opposed, we showed that the expected running time becomes polynomial for all variants by a simple optimization. We contrasted our theoretical study with computer experiments, which nicely fit the asymptotic results already on a small scale.

6 Acknowledgments

Thanks to Silke Rolles and Jan Johannsen for some helpful discussion on random shuffling of cards, and on the structure of random arrays of Booleans, resp.

References

1. T. Biedl, T. Chan, E. D. Demaine, R. Fleischer, M. Golin, J.A. King, and J. I. Munro. Fun-sort—or the chaos of unordered binary search. *Discrete Applied Mathematics*, 144(3):231–236, 2004.
2. A. Broder and J. Stolfi. Pessimal algorithms and simplexity analysis. *SIGACT News*, 16(3):49–53, 1984.
3. L. Flatto, A. M. Odlyzko, and D. B. Wales. Random shuffles and group presentations. *Annals of Probability*, 13(1):154–178, 1985.
4. R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, 1994.
5. D. E. Knuth. *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, 1981.
6. Walt Disney's Mary Poppins. VHS Video Cassette, 1980. [Walt Disney Home Video 023-5].
7. E. S. Raymond. *The New Hacker's Dictionary*. MIT Press, 1996.