Baby-Step Giant-Step Algorithms for the Symmetric Group

Eric Bach*
bach@cs.wisc.edu

Bryce Sandlund*
sandlund@cs.wisc.edu

University of Wisconsin - Madison
1210 W Dayton St.
Madison, WI 53706

ABSTRACT

We study discrete logarithms in the setting of group actions. Suppose that G is a group that acts on a set S. When $r, s \in S$, a solution $g \in G$ to $r^g = s$ can be thought of as a kind of logarithm. In this paper, we study the case where $G = S_n$, and develop analogs to the Shanks baby-step / giant-step procedure for ordinary discrete logarithms. Specifically, we compute two sets $A, B \subseteq S_n$ such that every $\sigma \in S_n$ can be written as a product ab of elements $a \in A$ and $b \in B$. Our deterministic procedure is close to optimal, in the sense that A and B can be computed efficiently and $|A|$ and $|B|$ are not too far from $\sqrt{n!}$ in size. We also analyze randomized "collision" algorithms for the same problem.

Keywords

Symmetric group, group actions, discrete logarithm, collision algorithm, computational group theory.

1. INTRODUCTION

Collision algorithms have been used to obtain polynomial (typically square root) speedups since the advent of computer science. Indeed, there are even collision "algorithms" in the world of analog measurement [9]. Most collision algorithms exploit time-space tradeoffs, arriving at a quicker algorithm by storing part of the search space in memory and utilizing an efficient lookup scheme. One of the most famous of these collision-style methods is Shanks's baby-step giant-step procedure for the discrete logarithm problem [17].

Traditionally, the discrete logarithm problem is the problem of finding an integer k such that $b^k = g$, where b and g are elements of a finite cyclic group of order n and b is a generator (has order n). Then there is exactly one $k \in \{1, \ldots, n\}$ such that $b^k = g$. Shanks's baby-step giant-step algorithm then writes $k = im + j$ with $m = \lceil \sqrt{n} \rceil$ and $0 \leq i, j \leq m$ and looks for a collision in the equation:

$$g(b^{-m})^i = b^j.$$

*Research supported by NSF: CCF-1420750

ISSAC '16, July 19–22, 2016, Waterloo, ON, Canada

DOI: http://dx.doi.org/10.1145/2930889.2930930

By precomputing values of b^j (or b^{-mi}) and storing them in a hash table, a collision can be found in $O(\sqrt{n})$ time and $O(\sqrt{n})$ space, recovering the solution k.

Various extensions of the baby-step giant-step algorithm have been developed, mostly focusing on discrete logarithm problems in groups that are important to cryptography. For the classic problem, Pollard [14] contributed two elegant methods that also exploit collision, but use very little space. (They have yet to be rigorously analyzed, in their original form.) For more information about these algorithms, and more efficient methods that apply to specific groups of an arithmetic nature, we refer to surveys by McCurley [11] and Teske [18].

Ideas similar to the baby-step giant-step algorithm have been used on 0-1 integer programming problems. (This seems to be folklore.) Suppose we want to solve $Ax = b$, where x is a 0-1 column vector. If we let x_1 and x_2 be half-length column vectors, and split A down the central column into A_1 and A_2, we can use collision to solve $A_1 x_1 = b - A_2 x_2$. (Here, we exploit not a group structure, but rather, the associative law for matrix-vector multiplication.) For a recent application, see [5].

In this paper, we focus on a different discrete log generalization that can be stated as follows.

DEFINITION 1.1. *Suppose that G is a finite group that acts on a set S. Denote by r^g the action of $g \in G$ on $r \in S$. Then, given elements $r, s \in S$, the group action discrete logarithm problem is the problem of finding a g such that $r^g = s$.*

The first step beyond brute force search for this problem is to design an analog to the Shanks method. We will find appropriate splitting sets $A, B \subseteq G$ so that for any s in the orbit $r^G = \{r^g : g \in G\}$, we have $r^{ab} = s$ for some $a \in A$ and $b \in B$. A match in the two sets

$$\{r^a : a \in A\} \text{ and } \{s^{b^{-1}} : b \in B\}$$

recovers the solution $g = ab$.

In this work, we treat two situations. The first is one of maximum generality: we know almost nothing about the structure of G, and can only work with it by applying it to elements of S. The second is maximally specific: G is the symmetric group S_n. In neither case do we assume any particular knowledge about the orbit of r or s.

For general groups (the first case), we analyze randomized methods that achieve square root speedups when compared to the naive approach of exhaustive search. For the important second case, we develop deterministic algorithms that

utilize the structure of S_n. These algorithms have close to square root complexity.

To motivate our model and concentration on S_n, we give several applications that fit into our framework.

2. APPLICATIONS

We first show how group actions lead to an unconventional algorithm for the graph isomorphism problem (GI). Let S be the set of adjacency matrices for graphs on n vertices. The symmetric group S_n acts on S, via $M^g = P_g M P_g^{-1}$. (P_g is just the permutation matrix for g.) In this case, the group action discrete logarithm problem is exactly graph isomorphism: given adjacency matrices M and N, find $g \in S_n$ to make $M^g = N$. Using our results, we arrive at a deterministic graph isomorphism procedure with run time about $\sqrt{n!}$.

Although there are much faster algorithms than this, they are either conceptually involved [1, 2], cannot guarantee efficient performance in all cases [3], or both [12]. The baby-step giant-step algorithm, on the other hand, gives an immediate proof that exhaustive search through permutations is not the best method for graph isomorphism.

There are a variety of other GI-related problems that also fit in the discrete log symmetric group action framework. In particular, hypergraph isomorphism and equivalence of permutation groups via conjugation can both be formulated as symmetric group actions. Furthermore, the latter problem has no known moderately exponential ($\exp(n^{1-c})$ for $c > 0$) algorithm [2].

We further note that in cryptography, the solution of iterated block ciphers [10] is closely related to a group-action discrete logarithm problem in a symmetric group. Our approach may also be useful in computational Galois theory, specifically, in computing the splitting field of a polynomial [13].

Because our approach is orbit-oblivious, our framework is very general. In some problems, however, algorithms aware of orbit restrictions may benefit significantly by reducing the number of $g \in G$ to consider. This is true, for example, in the graph isomorphism problem. An orbit-sensitive GI algorithm can utilize the fact that a vertex can only be mapped to another vertex of the same degree, whereas our approach will test every permutation.

On the other hand, there seem to be problems where one cannot exploit such orbit restrictions. We will now develop such an example.

Our problem will rely on the existence of fully homomorphic encryption schemes. Such schemes were only recently discovered, and utilize the presumed difficulty of the learning with errors problem [6]. We can explain the basic idea of homomorphic encryption as follows.

Denote by $\mathcal{E}(x)$ the result of encrypting bitstring x into ciphertext. Then in a fully homomorphic encryption scheme, we may convert any function f to a function f' on ciphertext, such that $\mathcal{E}(f(x)) = f'(\mathcal{E}(x))$. This leads to the following problem:

DEFINITION 2.1. *Let G be a finite group that acts on a set S. Then G acts on $\{\mathcal{E}(s) : s \in S\}$ via the induced action $\mathcal{E}(r)^g = \mathcal{E}(r^g)$. Then, given $\mathcal{E}(r)$ and $\mathcal{E}(s)$, the obfuscated group action discrete logarithm problem is the problem of finding a $g \in G$ such that $\mathcal{E}(r)^g = \mathcal{E}(s)$.*

The obfuscated group action discrete log problem is a special case of the general version. In this problem, we are not given the decryption keys, r, s, or S. We may only work with the ciphertexts of S and the action of G on these ciphertexts. Thus, we must utilize orbit-oblivious approaches.

Furthermore, observe that this problem can be applied to any group G. In particular, this suggests there are actions over the symmetric group such that there is no easily discernible relation between the action of S_n on $\{1, 2, \ldots, n\}$ and the set S.

It is worth noting that in the above setting, no backtrack search or other heuristical approaches will yield an efficient solution in practice. In the case where backtracking can yield an efficient solution, however, the worst case running time of such backtrack approaches is typically exponential. Additionally, their performance can vary from instance to instance and is often hard to predict. Although there are Monte Carlo procedures that can be used to estimate the runtime on a particular instance, such methods exhibit large variance. It is therefore of interest to develop techniques with provable running times.

For the algorithm designer, realization of such a baby-step giant-step algorithm for group actions reduces to efficient construction of the sets A and B. In the remainder of this paper, we discuss various methods for this.

3. A RANDOMIZED APPROACH

In this section our approach is general enough to work over any group G where random elements can be generated efficiently. In this setting, an obvious idea is to simply pick k random elements of G for the set A and k random elements of G for the set B. Then the probability a particular $g \in G$ is present in AB will depend on the value of k. For ease of notation, let $m = |G|$. We have:

PROPOSITION 3.1. *Suppose we pick k random elements of G **without** replacement for the set A and likewise for B. Then the probability a particular $g \in G$ is present in AB satisfies:*

$$Pr[g \in AB] \geq 1 - e^{-k^2/m}.$$

PROOF. Observe that $g \notin AB$ precisely when each $b \in B$ avoids the set $\{a^{-1}g : a \in A\}$. The probability of this event is

$$\left(1 - \frac{k}{m}\right)\left(1 - \frac{k}{m-1}\right)\cdots\left(1 - \frac{k}{m-k+1}\right),$$

which is at most $(1 - k/m)^k$. Rewriting the exponent and utilizing that $(1 - x/n)^n \leq e^{-x}$ gives us

$$\Pr[g \notin AB] \leq \left(\left(1 - \frac{k}{m}\right)^m\right)^{k/m} \leq e^{-k^2/m},$$

from which the claim follows. □

By setting $k = \Theta(\sqrt{m})$, we can make the probability that g is present in AB constant. Our analysis, however, assumed sampling without replacement. If we simply sample with replacement and redraw when a duplicate is found, it is not hard to see that as long as we are sampling $o(m)$ elements, the number of extra draws is $O(1)$ in expectation.

Note that with this approach, there will always be a non-zero probability that some group elements are missing in AB, which will lead to one-sided error in our algorithm.

Namely, if no g is found where $r^g = s$, the randomized procedure only gives probabilistic evidence that no g exists. Furthermore, checking for missing elements of G in AB takes $O(|G|)$ time. While this would only need to be done once and work for any set G acts on, it is prohibitively expensive.

This leads us to ask for a deterministic algorithm to construct the sets A and B. This question inherently asks about structure of the group G that can be exploited, similarly to the original Shanks method for the traditional discrete logarithm. Therefore, we will focus on the special case when $G = S_n$.

4. BACKGROUND ON GROUPS

Our deterministic algorithm will rely on some elementary group theory. For this, we state a few necessary results and definitions.

All groups in this paper will be finite. If K is a subgroup of G, we write $K \leq G$, and $K < G$ if the containment is proper. We do not assume that K is normal in G.

When $K \leq G$, its left cosets are the $|G|/|K|$ sets gK with $g \in G$. Right cosets are defined similarly.

Note that the set of left (or right) cosets of K in G forms a partition of G. Thus if we have a subgroup K of G, we can take $B = K$ and A to be a set of elements of G such that each left coset of K in G is represented in A. Then every element of G will be present in AB.

In group theory, a minimal perfect set A of this kind is called a transversal.

DEFINITION 4.1 (TRANSVERSAL). *A left (right) transversal T of a subgroup K of G is a set of elements of G such that each left (right) coset of K in G has exactly one representative in T. Thus, T is a minimal set of coset representatives of K in G.*

To make this definition clear, we give the following:

EXAMPLE 4.2. *Let $G = \mathbb{Z}_{n^2}$ and let K be the unique subgroup of G with n elements. Then $|G : K| = n$. If $G = \{0, \ldots, n^2 - 1\}$, $K = \{0, n, 2n, \ldots, (n-1)n\}$. One transversal of K in G is $T = \{0, 1, 2, \ldots, n-1\}$.*

For this example, K and T are exactly the sets of giant steps and baby steps that the Shanks algorithm would use. However, transversals are not unique; for example, we could have taken T to be any complete set of representatives modulo n.

Combinatorially, a subgroup B and its left transversal A form a perfect splitting set for G, in the sense that every $g \in G$ is uniquely of the form ab, for $a \in A$ and $b \in B$. Perfect splitting sets need not be subgroups, as we could always replace A by Ax and B by $x^{-1}B$, choosing $x \in G$ at will.

Ideally, these splitting sets have cardinality exactly $\sqrt{|G|}$. However, $n!$ is never a square for $n > 1$, so such perfect splitting sets cannot exist for $G = S_n$. Therefore, we will either have to tolerate duplicated products (as we did in the last section), or look for set sizes close to, but not exactly matching, $\sqrt{n!}$.

We now give some concepts that provide a "data structure" for working with permutation groups:

DEFINITION 4.3 (BASE). *Let G act on Ω. A base B for G is an ordered subset of Ω (i.e. a list) with the following property: the only element of G that stabilizes everything in B is the identity.*

For our purposes $G = S_n$, which stabilizes no element of $\Omega = \{1, \ldots, n\}$. So, we will use $B := [1, 2, \ldots, n]$; that is, $B = \Omega$ with the natural ordering of the integers. We could choose to not include any single integer from B, since the action of a permutation on the missing integer can be inferred via its action on the other elements; however, it will be easier to describe the transversal algorithm with a more complete base, and no loss of efficiency will be incurred. A base provides a convenient form to represent elements of G:

DEFINITION 4.4 (BASE IMAGE). *If $g \in G$ and $B = [\beta_1, \beta_2, \ldots, \beta_k]$ is a base for G, then $B^g := [\beta_1^g, \ldots, \beta_k^g]$ is called the base image of g (relative to B).*

Recall that β_i^g means the result of applying the group element g to the object $\beta_i \in \Omega$.

With the base $B := [1, 2, \ldots, n]$, the base image gives the typical vector notation of a permutation. The base image B^g uniquely determines the element $g \in G$.

5. BIDIRECTIONAL COLLISIONS

In recent work [15], David Rosenbaum described a general framework for applying collision algorithms to isomorphism problems. Potential isomorphisms are represented by paths in a tree. Given objects X and Y, we choose one set of paths that exhaust all levels halfway down (the remainder can be chosen arbitrarily), and apply these to X. Similarly, for Y, we choose one path that goes halfway down, extend it in all possible ways, and then apply each of these to Y. If we denote the set of transformations applied to X as C and to Y as D, then, borrowing the group action notation, this framework finds some $c \in C$ and $d \in D$ so that

$$X^c = Y^d$$

whenever there exists some g such that

$$X^g = Y.$$

Thus by taking $A = C$ and $B = D^{-1}$, where by D^{-1} we denote the set of inverses of all elements of D, this approach produces sets A and B with $AB = G$.

To split the search space for permutations, we choose a k with $1 \leq k \leq n$. The set C will consist of $(n)_k$ (n pick k) permutations. Each permutation sends $1, 2, \ldots, k$ to all possible k-tuples in $\{1, 2, \ldots, n\}$. We can choose arbitrarily where to send $k+1, \ldots, n$.

To make D, choose an image tuple for the first k elements arbitrarily (not moving them at all will do) and then extend with all $(n-k)!$ possible suffixes.

Since $|C| = (n)_k$ and $|D| = (n-k)!$, the counting functions for these two sets are not interchangable. However, we can try to balance the values of $(n)_k$ and $(n-k)!$. Since $(n)_k \cdot (n-k)! = n!$, finding a k such that $(n)_k \approx (n-k)!$ will, at least approximately, minimize $(n)_k + (n-k)!$, the cost of the collision algorithm. We will show that the "right" value of k is roughly, but not exactly, $n/2$. Surprisingly, the performance of the algorithm is very sensitive to k.

PROPOSITION 5.1. *The method presented above finds $A, B \subset S_n$ such that $AB = S_n$ and $\max(|A|, |B|) = O(n^{1/2}\sqrt{n!})$.*

PROOF. Let x be the positive real solution to $x! = \sqrt{n!}$ (we use $x!$ as an abbreviation for $\Gamma(x+1)$). Stirling's approximation says that

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n.$$

Use this on both variables in the defining relation for x, and then take logarithms of both sides. We can then apply "bootstrapping" [7] and show that

$$x = \frac{n}{2}\left(1 + O\left(\frac{1}{\log n}\right)\right).$$

Since $x!$ increases for $x \geq 1$, there is an integer m such that

$$(m-1)! \leq x! = \sqrt{n!} \leq m!.$$

Choose k so that $n-k$ (either m or $m-1$) is the closest integer to x. This will cause one of our sets to be larger than its "ideal" value $\sqrt{n!}$, and we must estimate this disparity. Let

$$n - k = x + \alpha, \qquad \text{with } |\alpha| \leq 1/2.$$

Recall that $(x+\alpha)! \sim x!x^\alpha$ as $x \to \infty$, in the sense that the limiting ratio is 1. Using this, and our asymptotic expression for x, we have

$$(n-k)! = (x+\alpha)! \sim x!x^\alpha \sim \sqrt{n!}\left(\frac{n}{2}\right)^\alpha.$$

Similarly,

$$(n)_k = \frac{n!}{(n-k)!} \sim \frac{n!}{\sqrt{n!}(n/2)^\alpha} = \sqrt{n!}(n/2)^{-\alpha}.$$

The bound on $|\alpha|$ gives

$$\max(|A|,|B|) = \max((n)_k,(n-k)!) = O\left(n^{1/2}\sqrt{n!}\right).$$

□

Probably, the factor $n^{1/2}$ is best possible. Examination of numerical data shows that α varies irregularly within the interval $(-1/2, 1/2)$, and we see no reason why this behavior should not continue. In particular, there is likely to be an infinite sequence of n's on which α's limiting value is $1/2$.

It is also interesting to compare our procedure to one that splits the permutation vectors exactly in half, as Rosenbaum initially recommends [15, p. 193]. This splitting amounts to taking $k = n/2$, and as a consequence of Stirling's formula,

$$(n)_{n/2} = \Theta(n^{-1/4}2^{n/2}\sqrt{n!})$$

when n is even. Therefore, the "overhead" for exact splitting is exponential in n. It is not hard to see this intuitively, by considering a decision tree for generating permutations. Each of the "large" branching factors $n-i$, $0 \leq i < n/2$, is roughly twice as large as its "small" counterpart $n/2-i$. By choosing the best splitting fraction for each n, we have reduced this overhead to a small power of n.

Finally, we note that this approach indeed employs a subgroup and a corresponding transversal of that subgroup. The set D fixes the first k elements and permutes the remaining $n-k$ elements amongst themselves in all possible ways. So D is simply S_{n-k}. Because of this, $D^{-1} = D$. Then, since $CD^{-1} = CD = S_n$, C is a left transversal of S_{n-k} in S_n.

6. A BETTER SUBGROUP

We can improve the results of the last section by using a different subgroup and its corresponding transversal. Intuitively, we can expect to get better results for the following reason: in bidirectional collision detection, we start with a fixed-length code for group elements and seek a good "halfway" point in the code words. This approach provides only n subgroups to choose from. To improve it, we can select our subgroup from a larger set.

A better choice of subgroup is the centralizer of a product of $\lfloor n/2 \rfloor$ disjoint two-cycles. Recall that the centralizer of an element $h \in G$ is the set $\{g \in G \mid gh = hg\}$, and is always a subgroup [4]. We have:

LEMMA 6.1. *The centralizer of a product of $\lfloor n/2 \rfloor$ disjoint two-cycles, denoted H, is a subgroup of S_n. We have*

$$|H| = \Theta(n^{1/4}\sqrt{n!})$$

if n is even and

$$|H| = \Theta(n^{-1/4}\sqrt{n!})$$

if n is odd. When n is odd, H is the same subgroup as it would be for $n-1$. When n is even, write $a = (1\ 2)(3\ 4)\ldots(n-1\ n)$, and so $H = C_{S_n}(a) = \{g \in S_n \mid ga = ag\}$.

PROOF. When n is odd, no permutation σ that moves the integer n can commute with a, because $a\sigma$ and σa will always move n in different directions. Therefore, when n is odd, the group is the same as for $n-1$.

Now, the centralizer of a in S_n, $C_{S_n}(a)$, has order:

$$|C_{S_n}(a)| = \frac{|S_n|}{\text{Size of conjugacy class of } a}.$$

See [4] for a proof. The size of the conjugacy class of a is determined by its cycle type [4]. The number of possible permutations with $\lfloor n/2 \rfloor$ disjoint two-cycles is given by:

$$\text{Size of conjugacy class of } a = \frac{n!}{2^{\lfloor n/2 \rfloor}\lfloor n/2 \rfloor!}.$$

This can be explained as follows. First, take all possible permutations on n items. Then, since the ordering of pairs within two-cycles does not matter, we can divide by $2^{\lfloor n/2 \rfloor}$. Furthermore, how the two-cycles are arranged does not matter, so we can further divide by $\lfloor n/2 \rfloor!$. Then:

$$|C_{S_n}(a)| = 2^{\lfloor n/2 \rfloor}\lfloor n/2 \rfloor!\ .$$

We can transform this into the desired result. We will show the algebra for the case that n is even, and then make a small change for when n is odd.

When n is even we have (by Stirling's approximation):

$$2^{n/2}(n/2)! \sim \sqrt{\pi n}\left(\frac{n}{e}\right)^{n/2},$$

$$\sqrt{n!} \sim \sqrt[4]{2\pi n}\left(\frac{n}{e}\right)^{n/2},$$

so

$$|C_{S_n}(a)| = 2^{n/2}(n/2)! \sim \sqrt[4]{\frac{\pi}{2}}n^{1/4}\sqrt{n!}.$$

When n is odd, a similar argument yields

$$|C_{S_n}(a)| = 2^{(n-1)/2}((n-1)/2)! \sim \sqrt[4]{\frac{\pi}{2}}n^{-1/4}\sqrt{n!}.$$

We therefore see when n is even, $|H|$ is $\Theta(n^{1/4}\sqrt{n!})$, and when n is odd, $|H|$ is $\Theta(n^{-1/4}\sqrt{n!})$. □

The subgroup H gives an improvement on bidirectional collision detection by a factor of $\Theta(n^{1/4})$. However, we also must have that H is efficiently enumerable. By Lemma 6.1, when n is odd, the group is the same as for $n-1$. Thus we need only concern ourselves with the even case:

LEMMA 6.2. *Let n be even. The centralizer of $a = (1\ 2)(3\ 4)\dots(n-1\ n)$ in S_n, denoted H, is generated by:*

$$H = \langle (1\ 2), (1\ 3)(2\ 4), (1\ 3\dots\ n-1)(2\ 4\dots\ n)\rangle.$$

PROOF. Any products of the disjoint two-cycles of a will commute with a. The two-cycle $(1\ 2)$ is present in the generating set, and by conjugating $(1\ 2)$ by powers of $(1\ 2\dots\ n-1)(2\ 4\dots\ n)$, we may obtain the two-cycles $(3\ 4), (5\ 6), \dots, (n-1\ n)$.

Furthermore, we may permute any of the disjoint two-cycles themselves by composing them with an appropriate transformation permutation.

We give the following homomorphism $\varphi : S_{n/2} \to S_n$: For each of the disjoint cycles of $\sigma \in S_{n/2}$, map $(\tau_1\ \tau_2\dots\ \tau_k)$ to

$$(2\tau_1 - 1\ 2\tau_2 - 1\dots\ 2\tau_k - 1)(2\tau_1\ 2\tau_2\dots\ 2\tau_k).$$

This map is clearly a homomorphism because all we have done is replaced each cycle with two cycles, one of which maps τ_i to $2\tau_i - 1$ and the other τ_i to $2\tau_i$. Since the two cycles will never share the same integers (one contains odd integers, the other even), we have effectively only relabeled σ, and thus the group operations are preserved under φ.

Furthermore, φ represents the necessary mapping to permute the two-cycles of a, since it is swapping both integers within a cycle with another cycle. Then, since

$$S_{n/2} = \langle (1\ 2), (1\ 2\dots\ n/2)\rangle,$$

the transformations that permute the two-cycles of a are generated by

$$\langle \varphi((1\ 2)), \varphi((1\ 2\dots\ n/2))\rangle,$$

which equals

$$\langle (1\ 3)(2\ 4), (1\ 3\dots\ n-1)(2\ 4\dots\ n)\rangle.$$

Note that any of the products of the disjoint two-cycles of a can be composed with any of the permutations of the disjoint two-cycles of a to form a new element in the centralizer. Thus the transformations described amount to $2^{n/2}(n/2)!$ unique elements. This is the size of the centralizer, so we know we have accounted for every element. □

With generators, we can run a closure algorithm to produce all the elements of H. This would achieve an $O(n|H|)$ time enumeration, but is slightly wasteful, because for each element of H, we must compose it with all generators before we can ensure all elements have been constructed. We can avoid this factor of 3 overhead by explicitly generating the elements of H themselves.

Note that the proof of Lemma 6.2 shows that H is isomorphic to the wreath product $C_2 \wr S_{n/2}$, and gives an explicit way to construct H given this property. We utilize this in the following algorithm:

Algorithm 1 Enumeration of the elements of H

1. Generate all possible products of the disjoint two-cycles of $a = (1\ 2)(3\ 4)\dots(n-1\ n)$; call the set X.
2. Generate all elements of $S_{n/2}$; call the set Y.
3. Apply the homomorphism φ described in the proof of Lemma 6.2 to each element of Y, resulting in a new set, Y'.
4. Compose every element of X with every element of Y' and return this set.

Before analyzing the algorithm, we make a few statements regarding our model of computation. For the purposes of this paper, we will charge $O(1)$ space for each integer and $O(1)$ time for accessing elements of an array. This hides logarithmic factors in runtime and space bounds. We will additionally charge space for output, though a note on this will be made after the proof of Lemma 7.4.

In this model of computation, a permutation can be represented in vector (base image) form using space $O(n)$ and two permutations can be composed in $O(n)$ time. Further note that we may choose to represent permutations as lists of arrays representing cycle notation, and in section 7, we will represent them as "partial base images" in a linked list or array. All representations can be converted to and from each other in $O(n)$ time.

We then have:

LEMMA 6.3. *Algorithm 1 correctly enumerates the elements of H in time $O(n|H|)$ and space $O(n|H|)$.*

PROOF. For correctness, Lemma 6.2 shows that the product set $XY' = H$.

For runtime, observe that a backtracking procedure can enumerate through all binary strings on $n/2$ bits in time $O(n2^{n/2})$. In practice, this can be done particularly efficiently by incrementing an integer and accessing its binary representation. Then for each binary string, we can concatenate the corresponding two-cycle of a for every set bit. If we represent elements in cycle notation, this concatenation can be done in $O(1)$ time. Thus we can construct the set X in $O(n2^{n/2})$ time.

To construct the set Y, note that a backtracking procedure can also enumerate every permutation of $S_{n/2}$ in time $O(n(n/2)!)$ (see [8] for the simple procedure). Applying the homomorphism φ can then be done in $O(n)$ time per element, producing all of Y' in overall $O(n(n/2)!)$ time.

There are $2^{n/2}(n/2)!$ elements in the product set XY', and each can be computed by composing an element of X with an element of Y' in $O(n)$ time per element. Thus the elements of H can be enumerated in $O(n|H|)$ time.

Regarding space, note that the most space used is in representing the output. We represent each $h \in H$ in $O(n)$ space, and so the space complexity is $O(n|H|)$. □

7. CONSTRUCTING TRANSVERSALS

We now consider finding a transversal of H in S_n. Although finding transversals can be complicated and require backtrack search through the parent group G [8], we can take advantage of having $G = S_n$.

We first give the following definition:

DEFINITION 7.1. *Let $B = [\beta_1, \ldots, \beta_k]$ be a base. Define a partial ordering $\prec$ on elements of Ω by taking $\beta_i \prec \beta_{i+1}$ for all $1 \leq i < k$, and $\beta_i \prec \alpha$, for every $\alpha \in \Omega$ not present in B. We extend this to base images by saying that for $g, h \in G$, $B^g \prec B^h$ if g precedes h in the lexicographic ordering on the base vectors.*

For our purposes, this is the natural ordering of the integers in B. Furthermore, this defines lexicographical ordering in the typical manner for base images B^g and B^h, $g, h \in G$.

Our transversal construction will exploit the following lemma.

LEMMA 7.2. *Let $K < G \leq S_n$ and let $B = [\beta_1, \ldots, \beta_k]$ be a base of G. Then $g \in G$ is the $\prec$-least element of its coset gK if and only if β_j^g is the $\prec$-least element of its orbit in $K_{\beta_1^g, \ldots, \beta_{j-1}^g}$ for $1 \leq j \leq k$.*

In this lemma, $K_{\beta_1^g, \ldots, \beta_{j-1}^g}$ denotes the subgroup of K consisting of the elements that fix each of the elements listed as subscripts. (When $j = 1$, this subgroup is just K.)

Although the result has been known since the work of Charles Sims, we present a proof in the interest of being self-contained. This lemma can be found (without proof) in [8, p. 115].

PROOF. Suppose that g satisfies the property given in Lemma 7.2. We must show g is $\prec$-least in gK. Write:

$$g = [\alpha_1, \alpha_2, \ldots, \alpha_k].$$

Now suppose there exists some $h \in gK$ such that $h \prec g$. Write:

$$h = [\gamma_1, \gamma_2, \ldots, \gamma_k].$$

Since $h \in gK$, we can write $h = gk$ for some $k \in K$. Therefore we can think of h as applying some element k to g. Now, since $h \prec g$, there must be a first index j such that $\gamma_j \prec \alpha_j$; so $\gamma_i = \alpha_i$ for all $1 \leq i < j$. Then k must stabilize $\alpha_1, \ldots, \alpha_{j-1}$. Since $\gamma_j \prec \alpha_j$, k must map α_j to γ_j, therefore γ_j and α_j are in the same orbit in $K_{\alpha_1, \ldots, \alpha_{j-1}}$. But by assumption, α_j is the $\prec$-least such element in its orbit in $K_{\alpha_1, \ldots, \alpha_{j-1}}$. Therefore the element h cannot exist and so g is minimal in gK.

In the other direction, suppose g does not satisfy the property in Lemma 7.2. Let j be the first index such that α_j is not $\prec$-least in its orbit in $K_{\alpha_1, \ldots, \alpha_{j-1}}$. Then there must be some $k \in K$ that stabilizes $\alpha_1, \ldots, \alpha_{j-1}$ and maps α_j to some η such that $\eta \prec \alpha_j$. Then $gk \prec g$. □

We can apply Lemma 7.2 to find base images that satisfy the property required to be $\prec$-least elements in their respective cosets. However, these base images might not necessarily correspond to elements of G if G is an arbitrary permutation group. Here we take advantage of the fact $G = S_n$. Every base image that satisfies Lemma 7.2 corresponds to some permutation on $\{1, 2, \ldots, n\}$, which is necessarily an element of S_n. Thus if it easy enough to find orbits after stabilizing in H, a transversal of H in S_n can be calculated efficiently. Below we have a useful fact about the structure of H.

LEMMA 7.3. *Denote by H_n our subgroup H for $G = S_n$. When n is even, we have that $(H_n)_{(\beta)} \cong H_{n-2}$. That is, H_n stabilized by any $\beta \in \Omega$ is isomorphic to H_{n-2}. When n is odd, $H_n = H_{n-1}$, so the result carries through for any $\beta \in \Omega$, $\beta \neq n$. If $\beta = n$, then $(H_n)_{(\beta)} = H_n$.*

PROOF. For odd n, the proof of Lemma 6.1 shows that H_n already stabilizes the integer n. Now, assume n is even. It should be clear that H is symmetric around any $\beta \in \Omega$, both because H is the centralizer of $a = (1\ 2)(3\ 4)\ldots(n-1\ n)$, where each $\beta \in \Omega$ plays the same role, and also by the proof of Lemma 6.2. We therefore can assume without loss of generality that $\beta = 1$.

Now, again observing the proof of Lemma 6.2, we can look at permutations σ that do not move 1 and have the property that $a\sigma = \sigma a$. Clearly we cannot use (1 2) as a product in any such σ, and additionally permuting the first cycle will require that 1 is not stabilized in σ. It is then clear that:

$$\begin{aligned} & C_{S_n}((1\ 2)(3\ 4)\ldots(n-1\ n))_{(1)} \\ = \quad & C_{S_n}((3\ 4)(5\ 6)\ldots(n-1\ n)). \end{aligned}$$

Again, by the symmetry in H, we have for even n and any $\beta \in \Omega$, $(H_n)_{(\beta)} \cong H_{n-2}$. □

This leads to the following algorithm to compute a transversal of H in $G = S_n$:

Algorithm 2 Transversal of H in $G = S_n$

If n is even:

1. Recursively obtain a transversal for $n - 2$. As a base case, if $n = 2$, let $T = \{[1, 2]\}$.
2. Increase every integer in the base images returned for $n - 2$ by 2.
3. Prepend 1 to the base images.
4. Put 2 in all possible locations after 1 in each base image, increasing the number of base images by a factor of $n - 1$, and return this set.

If n is odd:

1. Obtain a transversal for $n - 1$.
2. Put n in all possible locations in each base image returned for $n-1$, increasing the number of base images by a factor of n, and return this set.

By design, the algorithm finds the transversal consisting of the $\prec$-least element in each coset. We then have:

LEMMA 7.4. *Algorithm 2 correctly computes a transversal of H in $G = S_n$ in time $O(n|G : H|)$ and space $O(n|G : H|)$.*

PROOF. We first prove correctness. We use Lemma 7.2 to find the $\prec$-least element of each coset of H in G. From the list of generators, it is clear that if n is even, all objects of Ω are in the same orbit in H, and if n is odd, the two orbits of Ω are $\{1, 2, \ldots, n-1\}$ and $\{n\}$. Focusing on the even case, the first base element of every transversal base image will be 1, since it is the $\prec$-least of $\{1, 2, \ldots, n\}$. Furthermore, when we stabilize 1, by Lemma 7.3, we are left with a smaller instance of the same problem, with one caveat; the base element 2 is in its own orbit, and so can exist in any position after 1. We then see we can compute the transversal efficiently by recursively computing a solution to the smaller problem and then inserting 2 in all possible positions after 1. As a base case we have $H_2 = S_2$, so the transversal is $T = \{[1, 2]\}$. For

the odd case, we now must simply place n in every possible location within the entire base image of every transversal element for $n-1$.

We now prove runtime. Ignoring the recursion, in the even case the algorithm must add 2 to each integer of the base images and place 2 in all possible locations after 1. In the odd case the algorithm must do similarly but with n. Both cases must also copy the new transversal elements into the updated list T. With appropriate data structures, this can easily be accomplished in time $O(n \cdot \textit{number of transversal elements})$. Since the number of transversal elements is $|G:H| = \frac{|G|}{|H|}$, we then have the following recurrence for the runtime, when n is even:

$$T(n) = T(n-2) + O(n^{3/4}\sqrt{n!}).$$

This implies

$$\begin{aligned} T(n) &= O(n^{3/4}\sqrt{n!}) + O\left(\sum_{\substack{2 \le i \le n-2 \\ i \text{ even}}} i^{3/4}\sqrt{i!}\right) \\ &\le O(n^{3/4}\sqrt{n!}) + O\left(n^{7/4}\sqrt{(n-2)!}\right) \\ &= O(n^{3/4}\sqrt{n!}). \end{aligned}$$

When n is odd, we can first compute the transversal for $n-1$ in time $O((n-1)^{3/4}\sqrt{(n-1)!}) = O(n^{1/4}\sqrt{n!})$. We can then add n in the necessary locations and produce output in time $O(n \cdot n^{1/4}\sqrt{n!}) = O(n^{5/4}\sqrt{n!})$. So the runtime when n is odd is $O(n^{5/4}\sqrt{n!})$. Both of these runtimes are $O(n|G:H|)$.

Regarding space, note that the most space used is in representing the final output. Representing each permutation takes $O(n)$ space, so the space cost is $O(n|G:H|)$. □

As a further remark, we note the $O(n|G:H|)$ runtime can be improved by a factor of $O(n)$ if instead of writing the output, it is iterated. By iterating, we mean cycling through the transversal, rather than writing each element of the transversal to a list. In this same model, we can make the space bound as little as $O(n)$.

Let us consider the even n case, and the same logic applies to the case of odd n. Assume we can cycle through the output at the level of recursion for $n-2$. For each output given by $n-2$, prepending 1 and inserting 2 in all positions after 1 can be done in $O(1)$ time per insertion if the base images are represented as integers in a linked list. Furthermore, we must only maintain $O(1)$ bits of information per level. Therefore, since there are $O(n)$ levels of recursion, we do amortized constant amount of work per element of the transversal, resulting in a runtime that is improved by a factor of $O(n)$ and a space bound that is simply $O(n)$.

This same logic applies to enumerating the elements of H, though we omit the argument for brevity. In general, the collision framework for our discrete logarithm problem will have each $s \in S$ taking at least $O(n)$ space and $O(n)$ time to hash into a dictionary, so this observation will only bring a slight improvement in the space bound for our problem in certain cases.

Finally, we note that the property exploited in computing the transversal was that every base image found directly via stabilizers and orbits is necessarily an element of the parent group G if $G = S_n$. This observation can be combined with black-box orbit, stabilizer, and base changing methods as discussed in [8] to compute a transversal of an arbitrary permutation group $K < S_n$ efficiently. We will not discuss the details here, nor give exact asymptotic guarantees. For more information, consult the transversal algorithms discussed in Holt's book.

8. THE MAIN RESULT

We have the following theorem:

THEOREM 8.1. *We can compute sets A and B such that $AB = S_n$ with $\max(|A|,|B|) = O(n^{1/4}\sqrt{n!})$. The computation can be done deterministically in time and space $O(n^{5/4}\sqrt{n!})$.*

PROOF. Let H be the subgroup of S_n specified in Lemma 6.1. It has size $O(n^{1/4}\sqrt{n!})$ when n is even and $O(n^{-1/4}\sqrt{n!})$ when n is odd. Using Algorithm 1, it can be enumerated deterministically in time and space $O(n|H|)$. Furthermore, by employing Algorithm 2, its transversal can be found and enumerated deterministically in time and space $O(n^{3/4}\sqrt{n!})$ when n is even and $O(n^{5/4}\sqrt{n!})$ when n is odd. This means we can find sets of permutations A and B deterministically such that every $\sigma \in S_n$ can be represented as a product ab, $a \in A$ and $b \in B$ in time $O(n^{5/4}\sqrt{n!})$. □

This lemma implies that the group action discrete logarithm problem in the symmetric group can be solved deterministically in roughly $O(n^{5/4}\sqrt{n!})$ time. The specifics depend on the ability to hash or compare elements of S generated for the collision procedure. We note that while randomization may be used in the analysis of such hashing functions, the algorithm itself will always produce correct results.

In comparison, the computation of sets A and B by the randomized approach will take time $O(n\sqrt{n!})$ and by bidirectional collision detection time $O(n^{1.5}\sqrt{n!})$. In this sense, our result shows the difference between randomized and deterministic algorithms for this problem can be made as little as $O(n^{1/4})$. It is unclear if this can be eliminated entirely, but $O(n^{1/4})$ is so small when compared to $O(\sqrt{n!})$ that the improvement would be very minor. In particular, since $3^4 = 81$ and $\sqrt{81!} \doteq 2.4 \times 10^{60}$, for practical values of n, $n^{1/4}$ will not exceed 3.

Furthermore, in computing (an upper bound of) the expected number of permutations needed to achieve various reliability percentages, we note that our deterministic approach uses fewer permutations for reliability $> 80\%$ for $n < 12$, and $> 90\%$ for all other reasonable values of n. Thus due to the simplicity of Algorithm 1 and Algorithm 2, we consider our approach a viable practical alternative to the randomized one.

9. ACKNOWLEDGEMENTS

We thank Derek Holt, Gene Cooperman, and László Babai for correspondence on computing transversals in permutation groups. We additionally thank the stackexchange community for their input on subgroup choices and the anonymous reviewers for their constructive feedback.

10. REFERENCES

[1] L. Babai, W. M. Kantor, E. M. Luks. Computational complexity and the classification of finite simple groups. In *Proc. 24th Ann. Symp. Found. Comp. Sci.*, 1983, pp. 162-171.

[2] L. Babai. Graph isomorphism in quasipolynomial time. Manuscript, 2016. http://arxiv.org/pdf/1512.03547v2.pdf.

[3] J. Cai, M. Fürer, N. Immerman. An optimal lower bound on the number of variables for graph identification. *Combinatorica*, v. 12, 1992, pp. 389-410.

[4] D. Dummit, R. Foote. *Abstract Algebra.* Wiley, 2004.

[5] C. J. Etherington, M. W. Anderson, E. Bach, J. T. Butler, and P. Stănică. A parallel approach in computing correlation immunity up to six variables. *Int. J. Found. Comp. Sci.*, to appear.

[6] C. Gentry. Fully homomorphic encryption using ideal lattices. In *Proc. 41st ACM Symp. on Theory of Computing.* 41, 2009, pp. 169-178.

[7] D. H. Greene and D. E. Knuth. *Mathematics for the Analysis of Algorithms.* Birkhäuser, 1982.

[8] D. Holt. *Handbook of Computational Group Theory.* Chapman & Hall, 2005.

[9] A. Kwan. Vernier scales and other early devices for precision measurement. *American J. Physics*, v. 79, 2011, pp. 368-373.

[10] R. C. Merkle and M. E. Hellman, On the security of multiple encryption. *Comm. ACM*, v. 24, 1981, pp. 465-467.

[11] K. S. McCurley, The discrete logarithm problem. *Proc. Symp. Appl. Math.*, v. 42, 1990, pp. 49-73.

[12] B. McKay, A. Piperno. Practical graph isomorphism, II. *J. Symb. Comp.*, v. 60, 2014, pp. 94-112.

[13] S. Orange, G. Renault, K. Yokoyama. Computation schemes for splitting fields of polynomials. In *Proc. 2009 Int. Symp. on Symbolic and Algebraic Computation*, 2009, pp. 279-286.

[14] J. M. Pollard. Monte Carlo methods for index computation (mod p). *Math. Comp.*, v. 32, 1978, pp. 918-924.

[15] D. Rosenbaum. *Quantum Computation and Isomorphism Testing.* Dissertation, U. Washington, 2015.

[16] A. Seress. *Permutation Group Algorithms.* Cambridge Univ. Press, 2003.

[17] D. Shanks. Class number, a theory of factorization and genera. *Proc. Symp. Pure Math.*, v. 20, 1969, pp. 415-440.

[18] E. Teske, Square-root algorithms for the discrete logarithm problem (a survey). In *Public Key Cryptography and Computational Number Theory*, de Gruyter, 2001, pp. 283-301.