

University of Richmond

## UR Scholarship Repository

---

Honors Theses

Student Research

---

2020

### Almost Difference Sets in 2-Groups

Xin Yutong

*University of Richmond*

Follow this and additional works at: <https://scholarship.richmond.edu/honors-theses>



Part of the [Mathematics Commons](#)

---

#### Recommended Citation

Yutong, Xin, "Almost Difference Sets in 2-Groups" (2020). *Honors Theses*. 1526.

<https://scholarship.richmond.edu/honors-theses/1526>

This Thesis is brought to you for free and open access by the Student Research at UR Scholarship Repository. It has been accepted for inclusion in Honors Theses by an authorized administrator of UR Scholarship Repository. For more information, please contact [scholarshipprepository@richmond.edu](mailto:scholarshipprepository@richmond.edu).

# Almost Difference Sets in 2-Groups

A HONOR THESIS\* PRESENTED  
BY  
YUTONG XIN  
TO  
THE DEPARTMENT OF MATHEMATICS  
UNIVERSITY OF RICHMOND  
RICHMOND, VIRGINIA  
APRIL 30, 2020

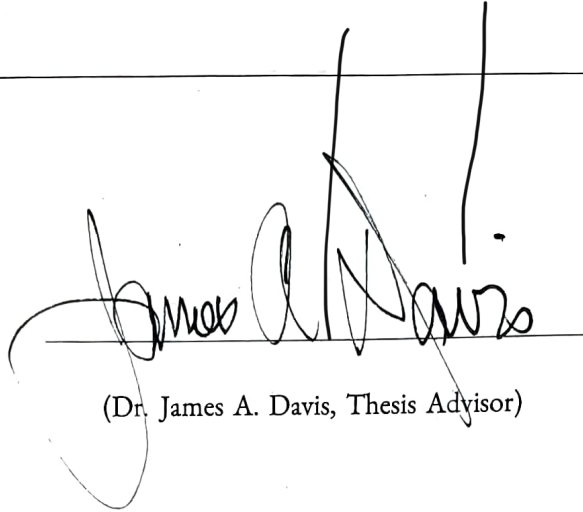
---

\*Under the direction of Dr. James A. Davis


---

The signatures below, by the thesis advisor, the departmental reader, and the honors coordinator for mathematics, certify that this thesis, prepared by Yutong Xin, has been approved, as to style and content.

---




(Dr. James A. Davis, Thesis Advisor)



---

(Dr. Della Dumbaugh, Departmental Reader)



---

(Dr. Van Nall, Honors Coordinator)

## Almost Difference Sets in 2-Groups

### ABSTRACT

Difference sets have been studied for decades due to their applications in digital communication, cryptography, algebra, and number theory. More recently, mathematicians have expanded their focus to the field of almost difference sets. Almost difference sets have similar functionalities with difference sets, yet with more potential of finding new constructions. In this paper I will introduce the definitions, properties, and applications of difference sets and almost difference sets, and discuss our effort and results in the exploration of almost difference sets in cyclic and non-cyclic groups.

# Contents

1	INTRODUCTION	2
1.1	Difference Sets . . . . .	3
1.2	Theoretical and Real-world Significance of Difference Sets . . . . .	4
1.3	Almost Difference Set . . . . .	6
2	EXPLORING STRUCTURES OF ADS	9
3	SOFTWARE TOOL	20
3.1	Stage 1 . . . . .	21
3.2	Stage 2 . . . . .	23
3.3	Stage 3 . . . . .	24
4	FUTURE WORK	27
	REFERENCES	30
	APPENDIX A APPENDIX	31

# 1

## Introduction

Difference sets and almost difference sets, with their special structural properties, are considered quite useful and powerful for both theory and application. In this paper we will discuss where difference sets and almost difference sets can be applied, the existence and non-existence of almost difference sets for certain groups we studied, analysis of algorithms used for our searches of almost difference sets, as well as some immediate results of applying our algo-

rithms. All groups in this paper will be written multiplicatively. Our ultimate objective is to provide construction methods for new almost difference sets.

We start with the definition of difference set.

## 1.1 DIFFERENCE SETS

**Definition 1.1.1.** A  $(v, k, \lambda)$  difference set (DS) is a  $k$ -element subset  $D$  of a finite group  $G$  of order  $v$ , such that for all non-identity elements  $g$  in  $G$ ,  $|\{(d_1, d_2) \in D^2 \mid g = d_1d_2^{-1}\}| = \lambda$ .

The parameters  $v, k, \lambda$  satisfy the following relationship:

**Proposition 1.1.2.** *If  $D$  is a  $(v, k, \lambda)$  difference set in  $G$ , then*

$$k(k - 1) = \lambda(v - 1)$$

*Proof.* Let  $G$  be a group, and  $D$  be a  $(v, k, \lambda)$  difference set of  $G$ . Recall that all  $g \in G$  can be expressed  $\lambda$  times as  $d_1, d_2 \in D$  such that  $g = d_1d_2^{-1}$ , for  $d_1, d_2 \in D$ .

Since  $|D| = k$ , there are  $k$  choices for  $d_1$ , and because  $d_2 \neq d_1$ , there are  $k - 1$  remaining choices for  $d_2$ . Hence, the total number of differences equals  $k(k - 1)$ .

On the other hand, since  $|G| = v$ , there are  $v - 1$  non-identity elements of  $G$ , and since each of these elements is covered  $\lambda$  times by the result of taking all the differences, we have that the number of differences is  $\lambda(v - 1)$ .

Therefore,  $k(k - 1) = \lambda(v - 1)$ . ■

Given a group  $G$ , a difference set  $D$  of  $G$ , and  $a \in G$ , we call  $aD = \{ad \mid d \in D\}$  a translate of  $D$ , where  $a$  is an element in  $G$ . If we let the elements of  $G$  be the points and we let the trans-

lates of  $G$  be the blocks, then the points and the blocks form a symmetric design. (See Beth et al. for details on designs.)

**Example 1.1.3.** The subset  $D = \{x, x^2, x^4\}$  is a  $(7, 3, 1)$  difference set of the group  $C_7 = \langle x \mid x^7 = 1 \rangle$ . Each element in  $C_7$  can be expressed in exactly  $\lambda = 1$  way by taking the difference of two elements in  $D = \{x, x^2, x^4\}$ , and  $3 \cdot (3 - 1) = 1 \cdot (7 - 1)$  satisfies proposition 1.1.2.

Additionally, the above difference set  $\{x, x^2, x^4\}$  and all its corresponding translates of  $C_7$ ,  $\{1, x, x^3\}$ ,  $\{x^2, x^3, x^5\}$ ,  $\{x^3, x^4, x^6\}$ ,  $\{x^4, x^5, 1\}$ ,  $\{x^5, x^6, x\}$ ,  $\{x^6, 1, x^2\}$ , can be visualized as the lines on the Fano plane (see Figure 1), with each line containing the three elements in the translates of  $D$ .

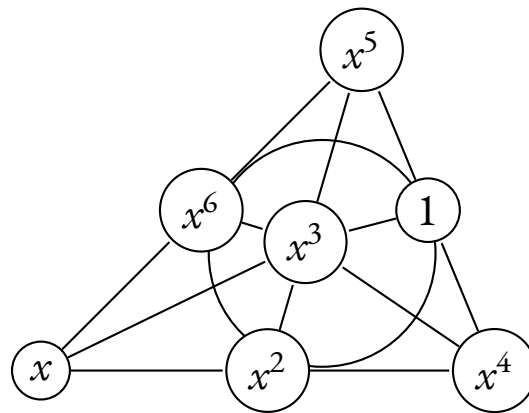


Figure 1.1: Fano plane

## 1.2 THEORETICAL AND REAL-WORLD SIGNIFICANCE OF DIFFERENCE SETS

Difference sets are useful in the realms of both theory and application.

Theoretically, difference sets are closely intertwined with algebra and number theory. Facts borrowed from these fields can be combined to produce beautiful difference sets<sup>15 4</sup>. For one



example, Davis and Jedwab discussed the theory of building blocks in regards to the character of a group<sup>8</sup>. Another example is the Mann Test which can be dated back to Mann (1964)<sup>13</sup> and later strengthened by Jungnickel and Pott (1988)<sup>11</sup> and Arasu, Davis, Jungnickel, and Pott (1990)<sup>1</sup>, where ring and group theory are applied to generate an important non-existence criterion for difference sets. Jungnickel and Pott (1999)<sup>15</sup> derived several corollaries from the Mann Test concerning existence conditions for planar difference sets, using the tools of elementary number theory, particularly the law of quadratic reciprocity.

The real-world significance of difference set can be found in applications such as sequences, coding theory, and design theory. Difference sets can be employed to construct signal sequences applicable in digital communication<sup>12</sup>; they have also been used to construct error-correcting codes with good performance<sup>3</sup>, and even were taken one step further to generate quantum stabilizer codes that help with communication of quantum information<sup>14</sup>. Difference sets can also assist in constructing symmetric and quasi-symmetric designs with the symmetric difference properties<sup>10</sup>. In fact, a difference set can be viewed equivalently to a symmetric design, with regular automorphism group<sup>4</sup>.

Driven by these motivations, the study of difference set constructions dates back to the early 1900s. There are a number of different families of difference sets: Paley(1933), Lehmer(1953), and Hall(1956) contributed to several constructions of cyclotomic difference sets; Whiteman discovered the twin-prime difference sets and their construction in 1962; McFarland constructed a new difference set family in 1972, which was later modified by Dillon in 1985 and Spence in 1977. Menon found a new family, Menon difference sets, in 1960 and 1962, and Dillon, again, proposed a new construction of Menon difference sets in 1974 and 1975. In 1997 Davis and Jedwab came up with a recursive construction that unified Hadamard,

McFarland and Spence difference sets. This construction yielded a new Davis-Jedwab family, which dealt with all abelian groups where such difference sets are known to exist<sup>7</sup>. This result were later extended by Chen<sup>5</sup>. For a long time, the Paley-Hadamard difference sets, constructed by Paley in 1933 utilizing nonzero quadratic residues of finite field, were thought to be the only example of skew Hadamard difference sets. But, Ding and Yuan found a new example in 2006 using permutation polynomials, and Ding, Pott, and Wang discovered another new example using Dickson polynomial in 2013. Another noteworthy family are Singer difference sets discovered by Singer in 1938. They are the family with the most distinct approaches of constructions. For this Singer difference sets, we are able to find the HKM (Helleseth, Kumar, Martinsen) construction(2001), the Lin(1998) construction, the Maschietti construction(1998), the Dillon-Dobbertin construction(2004), the Gordon-Mills-Welch construction(1962), and the No construction(2004). Additionally, Arasu and Player(2003), and Cao(2007) also contributed new constructions of Singer difference sets<sup>9</sup>.

However, with more of these constructions being produced, the remaining options are becoming limited and hence it is getting increasingly hard to come up with new constructions. As a result, the concept of an almost difference set arose<sup>6</sup>.

### 1.3 ALMOST DIFFERENCE SET

**Definition 1.3.1.** A  $(v, k, \lambda, t)$  almost difference set (ADS) is a subset that contains  $k$  elements of a group  $G$  of order  $v$ , such that for  $t$  non-identity elements  $g \in G$ , there are exactly  $\lambda$  different pairs of  $d_1, d_2 \in D$ ,  $d_1 \neq d_2, g = d_1 d_2^{-1}$ , and for the remaining  $v - t - 1$  non-identity elements  $g' \in G$ , there are exactly  $\lambda - 1$  different pairs of  $d'_1, d'_2 \in D$ ,  $d'_1 \neq d'_2, g' = d'_1 d'^{-1}_2$ .

The parameters  $v, k, \lambda, t$  satisfy the following relationship:

**Proposition 1.3.2.** *If  $D$  is a  $(v, k, \lambda, t)$  almost difference set, then*

$$k(k-1) = \lambda t + (\lambda - 1)(v - t - 1)$$

*Proof.* Let  $G$  be a group, and  $D$  be a  $(v, k, \lambda, t)$  almost difference set of  $G$ .

Following similar reasoning to the proof in proposition 1.1.2, we can conclude the total number of differences is equal to  $k(k-1)$  where  $k = |D|$ .

On the other hand, since  $|G| = v$ , there are  $v-1$  non-identity elements of  $G$ , and  $t$  of these elements are covered  $\lambda$  times, while the rest of the  $v-t-1$  elements are covered  $\lambda-1$  times, by the result of taking all the differences. Thus, we have that the number of the differences is  $\lambda t + (\lambda - 1)(v - t - 1)$ .

Therefore,  $k(k-1) = \lambda t + (\lambda - 1)(v - t - 1)$ . ■

**Example 1.3.3.**  $D = \{(1, 1), (1, x), (x, 1), (x, x), (x^2, 1), (x^4, x), (x^6, 1), (x^7, 1), (x^9, 1), (x^{12}, x), (x^{13}, 1), (x^{14}, x)\}$  is a  $(32, 12, 5, 8)$  almost difference set of the group  $C_{16} \times C_2 = \langle (x, y) \mid x^{16} = y^2 = 1 \rangle$ . Eight non-identity elements in  $C_{16} \times C_2$  can be expressed in exactly 5 ways, and the remaining  $32 - 1 - 8 = 23$  non-identity elements of the group can be expressed in exactly 4 ways by taking the difference of two elements in  $D$ . Additionally,  $12 \cdot (12 - 1) = 5 \cdot 8 + 4(32 - 8 - 1)$  satisfies the relationship among the parameters.

Similar to difference sets, ADS can also be applied in coding theory and sequences. They can be used to generate good constant-weight codes<sup>2</sup>. For some specific cyclic almost difference sets, their characteristic sequence can produce optimal auto-correlation codes, which is very useful in radar. Moreover, almost difference sets can be applied to generate functions with high nonlinearity, which is of importance in cryptography<sup>2</sup>. Based on what mathematicians

have discovered so far, due to almost difference sets' less strong structure than difference sets, it is relatively difficult to derive general theory for almost difference sets.

However, mathematicians did successfully discover certain constructions<sup>9</sup>. In the past mathematicians have managed to produce constructions of almost difference sets utilizing the power of cyclotomy, certain types of difference sets, planar functions, and the Gordon-Mills-Welch construction of difference set<sup>9</sup>.

Since the study of almost difference sets is relatively new compared with that of difference sets, we believe there is more potential in the exploration of new almost difference set constructions. We start by building a programming software to automate the process of almost difference set generation, aimed at detecting new patterns in the differences sets of cyclic and non-cyclic abelian groups, from which we work towards insights into new constructions.

# 2

## Exploring Structures of ADS

Example 1.1.3 provided a small example of a difference set. We start this section with small examples of difference sets and almost difference sets.

We explored constructions of difference sets for  $C_8 \times C_2$ ,  $C_4 \times C_2 \times C_2$ ,  $C_4 \times C_4$ , and  $C_2^4$ . These groups were previously known to have difference sets, but finding them manually still requires time. Hence we built a program to automatically generate difference sets. The

program did find a (16, 6, 2) difference set for each of the groups mentioned above.

The following is a short list of difference set examples for each  $G$ :

- $G = C_8 \times C_2 = \langle x, y \mid x^8 = y^2 = 1 \rangle$ :

$$D = \{(1, 1), (1, y), (x, 1), (x^2, 1), (x^5, 1), (x^6, y)\}$$

$$= \langle y \rangle \cup x \langle x^4 \rangle \cup x^2 \langle x^4 y \rangle$$

- $G = C_4 \times C_2 \times C_2 = \langle x, y, z \mid x^4 = y^2 = z^2 = 1 \rangle$ :

$$D = \{(1, 1, 1), (1, 1, z), (1, y, 1), (x, 1, 1), (x^2, 1, 1), (x^3, y, z)\}$$

$$= \langle x^2 \rangle \cup y \langle yz \rangle \cup x \langle x^2 yz \rangle$$

- $G = C_4 \times C_4 = \langle x, y \mid x^4 = y^4 = 1 \rangle$ :

$$D = \{(1, 1), (1, y), (1, y^2), (x, 1), (x^2, y), (x^3, y^2)\}$$

$$= y \langle x^2 \rangle \cup \langle y^2 \rangle \cup x \langle x^2 y^2 \rangle$$

- $G = C_2^4 = \langle x, y, z, w \mid x^2 = y^2 = z^2 = w^2 = 1 \rangle$ :

$$D = \{(1, 1, 1, 1), (1, 1, 1, w), (1, 1, z, 1), (1, y, 1, 1), (x, 1, 1, 1), (x, y, z, w)\}.$$

$$= x \langle xy \rangle \cup z \langle zw \rangle \cup \langle xyzw \rangle$$

From above, one important thing to notice is the difference sets in all the non-cyclic abelian groups consist of cosets of subgroups of order 2. This structure is desirable as we can view each coset as a hyperplane. Take the group  $C_8 \times C_2$  as an example, where the three hyperplanes are  $H_1 = \langle y \rangle, H_2 = \langle x^4 \rangle, H_3 = \langle x^4 y \rangle$ . If we define  $\chi : C_8 \times C_2 \rightarrow \mathbb{C}$ , such that  $\sum_{b \in H_i} \chi(b) \neq 0, \sum_{b \in H_j} \chi(b) = 0$  for  $j \neq i$ . For example, define  $\chi(x) = i, \chi(y) = -1$ . We then have  $\sum_{b \in H_2} \chi(b) = 1 + (\chi(x))^4 = 2, \sum_{b \in H_1} \chi(b) = 1 + \chi(y) = 1 - 1 = 0, \sum_{b \in H_3} \chi(b) = 1 + \chi(x)^4 \chi(y) = 1 - 1 = 0$ . Note  $\sum_{i=1 \dots 3} \sum_{b \in H_i} \chi(b) = 2 + 0 + 0 = 2(1) = \lambda(1)$ . We looked for comparable structure in ADSs.

Next we proceeded to the cyclic group  $C_{16}$ . As previously known, there does not exist a  $(16, 6, 2)$  difference set for  $C_{16}$ . Hence, we want to try to find the almost difference set  $(16, 5, 2, 5)$  for  $C_{16}$ , if it exists. To help with that, more features were added to the program such that it supports the auto-generation of ADS's. With the help of the program we found in total 38  $(16, 5, 2, 5)$  ADS for  $C_{16}$ . After filtering out the translates we had 17 ADS left.

We then tried to expand the group size even larger: we hoped to explore ADS in  $C_{32}$  and  $C_{64}$ . For  $C_{32}$  we were looking for  $(32, 12, 5, 8)$  ADS, and for  $C_{64}$  we were trying to find  $(64, 27, 12, 9)$  ADS. Yet the order of  $C_{64}$  plus the query size for ADS were so large that the program eventually ran out of time. Hence we decided to start with smaller  $k$ , and gradually try to expand. By varying the size of  $k$ , the program generated results listed in the next page. Note for each  $G$ , where  $|G| = v$ , we do not need to make queries for ADS with  $k > \frac{v}{2}$ . If an ADS with order  $k$ , denoted as  $D$ , exists for  $G$  with order  $v$ , taking the complement of  $D$ ,  $G \setminus (D \cup \{1\})$  yields another ADS with order  $v - 1 - k$ . In other words, an ADS with order  $v - 1 - k$  if and only if an ADS with order  $k$  exists. Hence, there is no need to look for ADS with order  $k > \frac{v}{2}$ .

$C_{16}$  :

ADS	whether exists	example ADS if exists
(16, 5, 2, 5)	yes	$\{1, x, x^2, x^5, x^8\}$
(16, 6, 2, 15)	no	
(16, 7, 3, 12)	yes	$\{1, x, x^2, x^3, x^5, x^8, x^{12}\}$
(16, 8, 4, 11)	yes	$\{1, x, x^2, x^3, x^4, x^7, x^9, x^{12}\}$

$C_{32}$  :

ADS	whether exists	example ADS if exists
(32, 7, 2, 11)	yes	$\{1, x, x^2, x^4, x^8, x^{13}, x^{18}\}$
(32, 8, 2, 25)	yes	$\{1, x, x^2, x^4, x^7, x^{13}, x^{17}, x^{25}\}$
(32, 9, 3, 10)	yes	$\{1, x, x^2, x^3, x^5, x^8, x^{14}, x^{18}, x^{25}\}$
(32, 10, 3, 28)	no	
(32, 11, 4, 17)	yes	$\{1, x, x^2, x^3, x^4, x^7, x^9, x^{13}, x^{17}, x^{22}, x^{25}\}$
(32, 12, 5, 8)	yes	$\{1, x, x^2, x^3, x^4, x^7, x^9, x^{14}, x^{15}, x^{19}, x^{23}, x^{26}\}$
(32, 13, 6, 1)	no	
(32, 14, 6, 27)	no	
(32, 15, 7, 24)	yes	$\{1, x, x^2, x^3, x^4, x^5, x^7, x^8, x^{12}, x^{15}, x^{17}, x^{21}, x^{23}, x^{26}, x^{27}\}$
(32, 16, 8, 23)	no	



$C_{64}$ :

ADS	whether exists	example ADS if exists
(64, 9, 2, 9)	yes	$\{1, x, x^2, x^5, x^{14}, x^{16}, x^{34}, x^{42}, x^{59}\}$
(64, 10, 2, 27)	yes	$\{1, x, x^2, x^4, x^7, x^{11}, x^{17}, x^{25}, x^{37}, x^{49}\}$
(64, 11, 2, 27)	yes	$\{1, x, x^2, x^4, x^7, x^{14}, x^{32}, x^{40}, x^{44}\}$
(64, 12, 3, 6)	no	
(64, 13, 3, 30)	yes	$\{1, x, x^2, x^3, x^5, x^{10}, x^{15}, x^{21}, x^{35}, x^{39}, x^{43}, x^{52}, x^{58}\}$
(64, 14, 3, 56)	no	
(64, 15, 4, 21)	yes	$\{1, x, x^2, x^3, x^4, x^8, x^{17}, x^{28}, x^{33}, x^{36}, x^{43}, x^{45}, x^{48}, x^{54}, x^{58}\}$
(64, 16, 4, 51)	no	
(64, 17, 5, 20)	yes	$\{1, x, x^2, x^3, x^4, x^6, x^9, x^{14}, x^{15}, x^{21}, x^{23}, x^{31}, x^{38}, x^{41}, x^{45}, x^{49}, x^{54}\}$
(64, 18, 5, 54)	unknown	
(64, 19, 6, 27)	yes	$\{1, x, x^2, x^3, x^4, x^5, x^8, x^{12}, x^{14}, x^{17}, x^{23}, x^{27}, x^{34}, x^{40}, x^{41}, x^{46}, x^{48}, x^{51}, x^{56}\}$
(64, 20, 7, 2)	unknown	
(64, 21, 7, 42)	yes	$\{1, x, x^2, x^3, x^4, x^6, x^7, x^{14}, x^{15}, x^{22}, x^{23}, x^{29}, x^{34}, x^{39}, x^{41}, x^{44}, x^{47}, x^{51}, x^{53}, x^{57}, x^{62}\}$

$C_4 \times C_4$ :

ADS	whether exists	example ADS if exists
(16, 5, 2, 5)	yes	$\{(1, 1), (1, y), (1, y^2), (x, 1), (x^2, 1)\}$
(16, 6, 2, 15)	yes	$\{(1, 1), (1, y), (1, y^2), (x, 1), (x^2, 1), (x^3, y^2)\}$
(16, 7, 3, 12)	yes	$\{(1, 1), (1, y), (1, y^2), (x, 1), (x, y), (x^2, y), (x^3, y^3)\}$
(16, 8, 4, 11)	no	

$C_8 \times C_2$ :

ADS	whether exists	example ADS if exists
(16, 5, 2, 5)	yes	$\{(1, 1), (1, y), (x, 1), (x^2, 1), (x^4, 1)\}$
(16, 6, 2, 15)	yes	$\{(1, 1), (1, y), (x, 1), (x^2, 1), (x^5, 1), (x^6, y)\}$
(16, 7, 3, 12)	no	
(16, 8, 4, 11)	yes	$\{(1, 1), (1, y), (x, 1), (x, y), (x^2, 1), (x^3, 1), (x^4, y), (x^6, y)\}$

$C_4 \times C_2 \times C_2$ :

ADS	whether exists	example ADS if exists
(16, 5, 2, 5)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (x, 1, 1), (x^2, 1, 1)\}$
(16, 6, 2, 15)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (x, 1, 1), (x^2, 1, 1), (x^3, y, z)\}$
(16, 7, 3, 12)	no	
(16, 8, 4, 11)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (x, 1, 1), (x, 1, z), (x, y, 1), (x^2, 1, 1), (x^3, y, z)\}$

$C_2 \times C_2 \times C_2 \times C_2$ :

ADS	whether exists	example ADS if exists
(16, 5, 2, 5)	yes	$\{(1, 1, 1, 1), (1, 1, 1, w), (1, 1, z, 1), (1, y, 1, 1), (x, 1, 1, 1)\}$
(16, 6, 2, 15)	yes	$\{(1, 1, 1, 1), (1, 1, 1, w), (1, 1, z, 1), (1, y, 1, 1), (x, 1, 1, 1), (x, y, z, w)\}$
(16, 7, 3, 12)	no	
(16, 8, 4, 11)	no	

$C_{16} \times C_2:$ 

ADS	whether exists	example ADS if exists
(32, 7, 2, 11)	yes	$\{(1, 1), (1, y), (x, 1), (x^2, 1), (x^3, y), (x^7, y), (x^{11}, 1)\}$
(32, 8, 2, 25)	yes	$\{(1, 1), (1, y), (x, 1), (x^2, 1), (x^3, y), (x^5, y), (x^8, y), (x^{12}, y)\}$
(32, 9, 3, 10)	yes	$\{(1, 1), (1, y), (x, 1), (x^2, 1), (x^3, y), (x^9, y), (x^{11}, 1), (x^{12}, y), (x^{13}, 1)\}$
(32, 10, 3, 28)	no	
(32, 11, 4, 17)	yes	$\{(1, 1), (1, y), (x, 1), (x, y), (x^2, 1), (x^3, 1), (x^5, 1), (x^7, 1), (x^8, y), (x^{11}, y), (x^{13}, 1)\}$
(32, 12, 5, 8)	yes	$\{(1, 1), (1, y), (x, 1), (x, y), (x^2, 1), (x^4, y), (x^6, 1), (x^7, 1), (x^9, 1), (x^{12}, y), (x^{13}, 1), (x^{14}, y)\}$
(32, 13, 6, 1)	no	
(32, 14, 6, 27)	yes	$\{(1, 1), (1, y), (x, 1), (x, y), (x^2, 1), (x^3, 1), (x^4, 1), (x^6, y), (x^8, 1), (x^{10}, y), (x^{11}, 1), (x^{12}, y), (x^{13}, 1), (x^{13}, y)\}$
(32, 15, 7, 24)	no	
(32, 16, 8, 23)	yes	$\{(1, 1), (1, y), (x, 1), (x, y), (x^2, 1), (x^2, y), (x^3, 1), (x^4, 1), (x^5, 1), (x^6, y), (x^8, y), (x^9, 1), (x^{11}, y), (x^{12}, 1), (x^{12}, y), (x^{14}, y)\}$

$C_8 \times C_4:$ 

ADS	whether exists	example ADS if exists
(32, 7, 2, 11)	yes	$\{(1, 1), (1, y), (1, y^2), (x, 1), (x^2, 1), (x^3, y), (x^4, 1)\}$
(32, 8, 2, 25)	yes	$\{(1, 1), (1, y), (1, y^2), (x, 1), (x^2, y), (x^4, 1), (x^5, y^3), (x^7, y)\}$
(32, 9, 3, 10)	yes	$\{(1, 1), (1, y), (1, y^2), (x, 1), (x^2, y), (x^2, y), (x^3, y^2), (x^4, 1), (x^5, y^3)\}$
(32, 10, 3, 28)	no	
(32, 11, 4, 17)	yes	$\{(1, 1), (1, y), (1, y^2), (1, y^3), (x, 1), (x^2, 1), (x^3, 1), (x^4, 1), (x^5, y^3), (x^6, y^2), (x^7, y)\}$
(32, 12, 5, 8)	yes	$\{(1, 1), (1, y), (1, y^2), (x, 1), (x, y), (x^2, 1), (x^2, y), (x^3, y^2), (x^4, 1), (x^5, 1), (x^5, y^2), (x^7, y^3)\}$
(32, 13, 6, 1)	no	
(32, 14, 6, 27)	no	
(32, 15, 7, 24)	yes	$\{(1, 1), (1, y), (1, y^2), (1, y^3), (x, 1), (x, y), (x^2, 1), (x^2, y), (x^3, 1), (x^4, y^2), (x^5, y), (x^5, y^3), (x^6, 1), (x^6, y^3), (x^7, y)\}$
(32, 16, 8, 23)	yes	$\{(1, 1), (1, y), (1, y^2), (1, y^3), (x, 1), (x, y), (x^2, 1), (x^2, y), (x^2, y^2), (x^3, 1), (x^4, y^2), (x^4, y^3), (x^5, 1), (x^5, y^2), (x^6, 1), (x^7, y)\}$

 $C_4 \times C_4 \times C_2:$ 

ADS	whether exists	example ADS if exists
(32, 7, 2, 11)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (1, y^2, 1), (x, 1, 1), (x, y, z), (x^2, 1, 1)\}$
(32, 8, 2, 25)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (x, 1, 1), (x, y, 1), (x, y^2, z), (x^2, y^3, z), (x^3, y, z)\}$
(32, 9, 3, 10)	no	
(32, 10, 3, 28)	no	
(32, 11, 4, 17)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (1, y, z), (1, y^2, 1), (x, 1, 1), (x, y, 1), (x, y^2, z), (x^2, 1, 1), (x^2, y^2, 1), (x^3, y^3, z)\}$
(32, 12, 5, 8)	no	
(32, 13, 6, 1)	no	
(32, 14, 6, 27)	no	
(32, 15, 7, 24)	no	
(32, 16, 8, 23)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (1, y, z), (1, y^2, 1), (1, y^2, z), (x, 1, 1), (x, 1, z), (x, y, 1), (x, y^2, 1), (x^2, 1, 1), (x^2, y, 1), (x^2, y^2, 1), (x^2, y^3, z), (x^3, 1, z), (x^3, y^3, z)\}$

$$C_8 \times C_2 \times C_2:$$

ADS	whether exists	example ADS if exists
(32, 7, 2, 11)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (x, 1, 1), (x^2, 1, 1), (x^3, 1, z), (x^4, y, 1)\}$
(32, 8, 2, 25)	no	
(32, 9, 3, 10)	no	
(32, 10, 3, 28)	no	
(32, 11, 4, 17)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (1, y, z), (x, 1, 1), (x^2, 1, 1), (x^3, 1, 1), (x^4, 1, z), (x^5, y, 1), (x^6, y, z), (x^7, 1, z)\}$
(32, 12, 5, 8)	no	
(32, 13, 6, 1)	no	
(32, 14, 6, 27)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (x, 1, 1), (x, 1, z), (x, y, 1), (x^2, 1, 1), (x^2, y, z), (x^3, 1, 1), (x^3, 1, z), (x^4, y, z), (x^5, y, z), (x^6, 1, 1), (x^6, y, 1)\}$
(32, 15, 7, 24)	no	
(32, 16, 8, 23)	yes	$\{(1, 1, 1), (1, 1, z), (1, y, 1), (1, y, z), (x, 1, 1), (x, 1, z), (x, y, 1), (x^2, 1, 1), (x^2, 1, z), (x^3, y, 1), (x^4, 1, 1), (x^4, y, 1), (x^5, 1, z), (x^6, 1, 1), (x^6, y, z), (x^7, 1, 1)\}$

$$C_4 \times C_2 \times C_2 \times C_2:$$

ADS	whether exists	example ADS if exists
(32, 7, 2, 11)	yes	$\{(1, 1, 1, 1), (1, 1, 1, w), (1, 1, y, 1), (1, y, 1, 1), (x, 1, 1, 1), (x, y, z, w), (x^2, 1, 1, 1)\}$
(32, 8, 2, 25)	no	
(32, 9, 3, 10)	no	
(32, 10, 3, 28)	no	
(32, 11, 4, 17)	no	
(32, 12, 5, 8)	no	
(32, 13, 6, 1)	no	
(32, 14, 6, 27)	no	
(32, 15, 7, 24)	no	
(32, 16, 8, 23)	yes	$\{(1, 1, 1, 1), (1, 1, 1, w), (1, 1, z, 1), (1, 1, z, w), (1, y, 1, 1), (1, y, 1, w), (x, 1, 1, 1), (x, 1, 1, w), (x, 1, z, 1), (x, y, 1, 1), (x^2, 1, 1, 1), (x^2, 1, z, 1), (x^2, y, 1, 1), (x^2, y, z, w), (x^3, 1, 1, w), (x^3, y, z, 1)\}$

Observing our results, there are several things that worth noticing.

First of all, we tried to group the elements in the ADS to look for potential structural patterns. It would be very ideal if structures similar to the grouping of hyperplanes in difference sets could exist, in which case there would be a systematic way to construct a set of ADS's.

However, we did not succeed in our attempt to detect such patterns, but such exploration is certainly worthwhile to be continued in the future.

For groups with same order  $v$ , and same query range for the ADS order  $k$ , as the group is broken into more components, more nonexistence results showed up. Take  $v = 32$  as an example. Both  $C_{16} \times C_2$  and  $C_8 \times C_4$  had 3 ADS queries returned empty. For  $C_4 \times C_4 \times C_2$  and  $C_8 \times C_2 \times C_2$ , the number of empty query results increased to 6. While when we had 4 components for group  $C_4 \times C_2 \times C_2 \times C_2$ , only 2 out of 10 queries yields valid ADS's. Such a result was intuitively reasonable, as the choice of elements for ADS became less flexible when the number of components increased, but we should try to find mathematical reasoning to justify this trend.

Another interesting observation is, based on the results we have so far, an ADS with smallest size seemed to have exist for all groups we have explored in. (By "smallest," I meant if the size of ADS is strictly smaller than the "smallest" ADS, there would be elements in the group that could never be covered.) We need to do more searches to see whether this result still remains true, and look for an explanation if this is the case.

I'd also like to point to the result for  $C_4 \times C_2 \times C_2 \times C_2$ . The ADS's only existed for  $k = 7$  and  $k = 16$ , but no ADS was found for any of the value of  $k$  between  $[8, 15]$ . We could try to find out whether this uncommon pattern was a coincidence, or whether we could explain it by using our algebra tools.

# 3

## Software tool

In order to explore patterns in ADS, I developed a software tool, which automates our ADS generating process and saves a significant amount of time in searching and checking the validity of ADS.

The software takes user input about ADS parameters, and returns the ADS in query, if one exists. Initially the software can find all possible ADS, filter out the translates, and return the



remaining ADS, but we suppressed that feature for our current purpose of the study, since we are primarily interested in the existence question.

There have been 3 stages for the development of our software tool. In the first stage the basic features were built up; in the second stage, some currently unnecessary procedures were cut off to increase efficiency; in the third stage the code was made more consistent, and the data structure used was modified such that the time required for a successful search was significantly reduced. In short, the stage 1 software is more complete, yet the stage 2 and 3 software is more efficient.

### 3.1 STAGE 1

The software is composed of 2 main classes, one for searching in multi-dimensional, non-cyclic abelian group (ex.  $C_4 \times C_4 = \{(x, y) \mid x^4 = 1, y^4 = 1\}$ ), while the other one for searching in cyclic group (ex.  $C_{16}$ ). The latter is a subclass of the former.

For data representation, in the non-cyclic ADS generator, each element of the group is represented as an integer array, and hence an ADS is stored as a set of integer arrays; in the one-dimensional ADS generator, each element can simply be stored in an integer, so as a result, an ADS is stored as a set of integers.

With the basic skeleton and data structure used in mind, we now proceed to describe the algorithm:

The program receives input containing the parameters  $(v, k, \lambda, t)$  and whether the group cyclic. After checking whether the parameters satisfy Proposition 1.1.2, the program starts generating and initializing a counter for all elements of the group  $G$ , and then begins searching for difference sets. It first generates all possible subsets  $S$  of  $G$  with size  $k$ , then for all  $d_i, d_j$  in

each subset  $S$ , it computes  $d_i d_j^{-1}$ , meanwhile it keeps track of how many times each element of  $G$  is covered by taking  $d_i d_j^{-1}$  in each  $S$ . After all such differences are computed from elements in  $S$ , if exactly  $t$  elements in  $G$  are covered exactly  $\lambda$  times, the program knows  $S$  is a valid *ADS* and adds it into the result list, otherwise it removes  $S$ . After all such subsets are processed, the program filter out all translates and multiples, after which it returns the resulting list containing all valid *ADS*.

Based on the algorithm, we were able to analyze the time and space complexity of running the software. Assume a search for  $(v, k, \lambda, t)$  *ADS*, and the dimension of  $G$  is  $d$ . The time complexity is  $O(k^2 \binom{v}{k})$ : there are in total  $v$  elements in  $G$ , and  $k$  elements for each subset generated, hence  $\binom{v}{k}$  subsets generated. For each of the sets,  $O(k^2)$  computations need to be done. Therefore, it takes  $O(k^2 \binom{v}{k})$  time for the entire algorithm, and since  $v > k$ , we can express the approximate time complexity as  $O(n^k)$ . The space complexity is  $O(dk \binom{v}{k})$ , since each element takes  $d$  space, each *ADS* candidate has  $k$  elements, and there are  $\binom{v}{k}$  *ADS* candidates generated. Similarly, we can express it as  $O(dn^k)$ .

At its current stage, the software has guaranteed completeness: it is able to generate all the *ADS* for a given group  $G$ . It also has the feature of a filter. As a result, all *ADS* returned are not translates and multiples of each other, and the union of the translates and multiples of each *ADS* cover all the possible  $(v, k, \lambda, t)$  *ADS* for  $G$ .

Nevertheless, there are also certain restrictions. In order to achieve as much completeness as possible, the algorithm is too time and space intensive, quickly running out of time or space when the size of the given group  $G$  gets slightly larger.

### 3.2 STAGE 2

For stage 2, we focused on the cyclic groups, so our improvement was done entirely on the part of the software corresponding to the cyclic groups. The classes and data structures used are the same as the software in Stage 1.

The program starts with reading input, validity checking, and generating and initializing a counter for the elements of  $G$ . It then looks for valid ADS based on the following procedure: start with an empty set  $S$ , and fix the element  $x_0 = 1$  to be in  $S$ . Next, try expanding the size of  $S$  by adding  $x_1, x_2, \dots \in G$  one at a time. In each expansion, when adding  $x_i$ , compute the difference between each element  $x_i$  and each element  $d_i$  that is already in  $S$ , and update the counter for each newly generated element by taking the difference. If the newly generated element is already covered more than  $\lambda$  times, the program throws  $x_i$  out, and tries to include the next element of  $G$ ,  $x_{i+1}$ , repeating the same procedure described above.

When the size of  $S$  reaches  $k$ , and no elements in  $G$  have yet been covered more than  $\lambda$  times, it means  $S$  is a valid ADS, so  $S$  will be added to the result list. The program immediately stops and return the ADS generated after one such instance is found.

Suppose we are searching for a  $(v, k, \lambda, t)$  ADS, and the dimension of  $G$  is  $d$ . In the worst scenario, the time upper bound for the algorithm is still  $O(n^k)$  in theory. However, since the algorithm stops in the middle, it avoids a lot of meaningless searches, and thus it overall runs about 3 times faster in practice than in Stage 1. In regards to space, the program needs to store all elements in  $G$ , and the current expanding ADS candidates. Elements in  $G$  take  $O(vd)$  space, and the expanding ADS candidates have max size  $k$  and hence take  $O(kd)$  space, where  $k \leq v$ . Hence the space complexity is  $O(vd)$ .

In the current stage, since the program immediately returns the ADS if one exists, we are able to know whether there exists an ADS for the input group much more efficiently, as we do not need to wait until the program runs a complete search through all possible subsets of size  $k$ .

The following is a timing comparison between Stage 1 and Stage 2 in finding ADS from cyclic groups. The timing results are expressed in nanoseconds.

For group  $C_{16}$ ,  $(16, 5, 2, 5)$  ADS:

Stage 2: 69509 ns

Stage 1: 234269 ns

Speed-up: 3.370

For group  $C_{16} \times C_2$ ,  $(32, 12, 5, 8)$  ADS:

Stage 2: 24075868 ns

Stage 1: java.lang.OutOfMemoryError: Java heap space

Despite the relatively significant improvement of efficiency, the speed is still not ideal. If there does not exist an ADS for a given group  $G$ , the programs still needs to do a checking for all the ADS candidates, before it terminates and concludes the nonexistence of an ADS. This process still takes a lot of time.

### 3.3 STAGE 3

The main improvement in this stage is focused on non-cyclic groups, which we accomplished by modifying how the data is stored. I combined the two classes and ended up with only one class supporting the search in both single and multiple dimensional groups. Previously, the

non-cyclic group was not supported in stage 2, because in order to generate the ADS more efficiently, the software needs to keep track how many times each of the element of  $G$  is already covered, which requires the use of a HashMap. In the HashMap, each element would be used as a different key, while the number of times each element is covered would be the corresponding value. For example, when  $g \in G$  is generated for the first time by  $d_1 d_2^{-1} = g$ , the corresponding key-value pair associated with  $g$  in the HashMap would be  $(g, 1)$ ; when  $g$  is generated by  $d_3 d_4^{-1} = g$  the second time, the key-value pair is updated to  $(g, 2)$ . In previous stages each element of  $G$  was stored as an Integer Array, if  $G$  is non-cyclic. For instance, an element  $(x, y, z), x, y, z \in \mathbb{Z}$  was stored as  $[x|y|z]$ . However, Integer Array would not yield desirable results if used directly as keys of the HashMap. That was why the algorithm in Stage 2 could not be applied to non-cyclic groups. In the current stage, each component of an element is represented as a Character. Elements of the group  $G$  are represented as a String, which is a concatenation of Characters. Since Strings function normally as keys in HashMap, the previous problem was solved and hence the speed-up version of the algorithm in Stage 2 could now be applied to non-cyclic groups.

The algorithm stays unchanged compared to Stage 2, and hence the time and space complexity is same as in Stage 2 (but the software in Stage 2 had not yet supported non-cyclic groups).

Compared to Stage 1, due to the change of data structure and the algorithm, there is a significant speed up. The following is the timing result for two specific non-cyclic groups:

For group  $C_8 \times C_2, (16, 5, 2, 5)$  ADS:

Stage 3: 101142 ns

Stage 1: 338540 ns

Speed-up: 3.347

For group  $C_{16} \times C_2$ , (32, 12, 5, 8) *ADS*:

Stage 3: 124211003 ns

Stage 1: java.lang.OutOfMemoryError: Java heap space

However, as shown above, regardless of our large boost in speed, when the group size increases from 16 to 32 by a factor of 2, the time used increases by a factor of 1228.085, which is still huge. If there exists an approach where the time complexity actually changes, this deficiency could potentially be improved.

Additionally, similar to Stage 2, only 1 *ADS* is returned instead of all, which indeed suffices for our current focus.

# 4

## Future Work

We've built up the automatic tools for generating the existence results for cyclic and non-cyclic groups of reasonable size. We have also applied the tools to get some existence results of varied size of almost difference sets for both cyclic and non-cyclic groups. There are some immediate next steps we could take to further extend this current research.

For one thing, we could look for patterns in the current existence results. For the current

groups with reasonable size, it's still possible for us to analyze the structure, compared to the harder situation when the group size gets much larger. If we successfully find a pattern in the smaller group, we could try to apply it to larger groups, and potentially work towards new constructions of almost difference sets. Groups with order  $2^n$ ,  $n \in \mathbb{Z}$  are a fertile ground for exploration. We are currently able to determine existence results of groups of order 16, 32, 64 and we would love to know more about groups of order 128, 256, 512, and so on.

Another direction is to further upgrade the software tool. The software of the current version supports generating existence results for relatively large groups, yet it would be even more ideal if it could support listing all possible almost difference sets given an input group in query. This could give us more examples to explore, and potentially provide us with more insights into the ADS structures.

We would also expand our focus to non-abelian groups. For now all the groups in which we have tried searching for ADS ( $C_{32}$ ,  $C_{64}$ ,  $C_4 \times C_4$ ,  $C_{16} \times C_2$ , etc.) are abelian. If we start some new searches to non-abelian groups, such as a semidirect product groups for instance, there could be results yielding new insights for us.



## References

- [1] Arasu K., Davis J., Jungnickel D., Pott A. (1990). A note on intersection numbers of difference sets. *European Journal of Combinatorics*, 11(2), 95 – 98.
- [2] Arasu K., Ding C., Helleseth T., Kumar P. V., Martinsen H. M. (2001). Almost difference sets and their sequences with optimal autocorrelation. *IEEE Transactions on Information Theory*, 47(7), 2934–2943.
- [3] Assmus E. F., Key J. D. (1992). *Designs and their Codes*. Cambridge Tracts in Mathematics. Cambridge University Press.
- [4] Beth T., Jungnickel D., Lenz H. (1999). *Design Theory*. Cambridge University Press.
- [5] Chen, Y. Q. (1999). On a family of covering extended building sets. *Designs, Codes and Cryptography*, 17, 69–72.
- [6] Davis J. (1992). Almost difference sets and reversible divisible difference sets. *Archiv Der Mathematik*, 59(2), 595–602.
- [7] Davis J., Jedwab J. (1997). A unifying construction for difference sets. *Journal of Combinatorial Theory, Series A*, 80(1), 13 – 78.
- [8] Davis J., Jedwab J. (1999). A unified approach to difference sets with  $\gcd(v, n) > 1$ .
- [9] Ding, C. (2014). *Codes from Difference Sets*. WORLD SCIENTIFIC.
- [10] Ding C., Munemasa A., Tonchev V. (2019). Bent vectorial functions, codes and designs. *IEEE Transactions on Information Theory*, PP, 1–1.
- [11] Jungnickel D., Pott A. (1988). Two results on difference sets. *Coll. Math. Soc.*, (pp. 325–330).
- [12] Ma, S. (2012). Difference sets and sequences. *Bulletin of the Malaysian Mathematical Sciences Society. Second Series*, 35.

- [13] Mann, H. B. (1964). Balanced incomplete block designs and abelian difference sets. *Illinois J. Math.*, 8(2), 252–261.
- [14] Nguyen, M. D. (2018). New constructions of quantum stabilizer codes based on difference sets. *Symmetry*, 10, 655.
- [15] Pott A., Kumaran V., Helleseth T. , Jungnickel D. (1999). *Difference Sets, Sequences and their Correlation Properties*, volume 542 of *Nato Science Series C*. Springer Netherlands.



## Appendix

The first part consists of the code in Stage 1, where queries in both cyclic and non-cyclic groups are supported, but the speed is relatively slow.

In a query for ADS, the program does the following:

- `readInput` (line 19) called by constructor to read inputs from user, initialize all fields and do validity check. We want to ensure the input parameters yield valid difference sets.
- `getAllGroupElements` (line 204) gets called to generate all group elements  $g \in G$ , from which we will select all possible  $k$ -subsets as ADS candidates.
- `getADSCandidates` (line 72) gets called to generate all ADS candidates, by trying to get all possible combinations to form a subset of size  $k$ , with the assistance of the recursive method `getNext()` (line 85). WE applied backtracking search in this step.
- `getADS` (line 103) gets called to select from the candidates the valid ADS, by calling `isADS` (line 156) to do a checking for each candidate. The set of all valid ADS's are returned at the end of this method.
- `printADS` (line 225) gets called to print out all valid ADS's

```

1 import java.util.*;
2 public class ADSGenerator {
3     // fields
4     protected int dimension; // dimension of the space in concern
5     protected int[] array; // array holding info for R_i in index i
6     protected int groupOrder; // num elements in entire group
7     protected int numElemLambda; // number of elements to be covered for timeCovered times
8     protected int lambda; // cover numElemCovered elements timesCovered times
9     protected int ADSOrder; // ADS order
10
11     /* constructor
12     */
13     public ADSGenerator() {
14         this.readInput();
15     }
16
17     /* readInput from user, error checking before initializing all fields
18     */
19     public void readInput() throws IllegalArgumentException {
20         Scanner in = new Scanner(System.in);
21         System.out.println("input the dimension you want to search for");
22         dimension = in.nextInt();
23         if (dimension < 1) {
24             throw new IllegalArgumentException("dimension should be positive");
25         }
26
27         array = new int[dimension];
28         groupOrder = 1;
29         for (int i = 0; i < array.length; i++) {
30             // read in n in Z_n, and store n in each index
31             System.out.println("input next n for Z_n");
32             array[i] = in.nextInt();
33             if (array[i] < 1) {
34                 throw new IllegalArgumentException("n should be positive");
35             }
36             groupOrder *= array[i];
37         }
38
39         System.out.println("input lambda");
40         lambda = in.nextInt();
41
42         System.out.println("input number of elements to cover lambda times");
43         numElemLambda = in.nextInt();
44
45         System.out.println("input ADS size in query");
46         ADSOrder = in.nextInt();

```

```

47
48     this.validADSPreCheck();
49
50 }
51
52 public void validADSPreCheck() throws IllegalArgumentException {
53     if (lambda < 1)
54         throw new IllegalArgumentException("lambda should be positive");
55
56     if (numElemLambda < 0) {
57         throw new IllegalArgumentException("input numElem should be non-negative");
58     }
59
60     if (ADSOrder < 0) {
61         throw new IllegalArgumentException("ADS size should be non-negative");
62     }
63
64     if (ADSOrder * (ADSOrder - 1) !=
65         numElemLambda * lambda + (groupOrder - 1 - numElemLambda) * (lambda - 1)) {
66         throw new IllegalArgumentException("invalid input");
67     }
68 }
69
70 /* generate all possible sets of order ADSOrder
71 */
72 public Set<List<int[]>> getADSCandidates(int[][] allGroupElem) {
73     List<int[]> currSet = new ArrayList<>();
74     int firstElemInd = 0;
75     currSet.add(allGroupElem[firstElemInd]);
76     firstElemInd++;
77
78     Set<List<int[]>> allSets = new LinkedHashSet<>();
79     getNext(firstElemInd, ADSOrder - 1, allGroupElem, currSet, allSets);
80     return allSets;
81 }
82
83 /* recursive method to help generate next ADS candidate
84 */
85 private void getNext(int currInd, int numLeft, int[][] allGroupElem,
86     List<int[]> currSet, Set<List<int[]>> allSets) {
87     if (numLeft == 0) {
88         List<int[]> newSet = new ArrayList<>(currSet);
89         allSets.add(newSet);
90     } else if (allGroupElem.length - currInd >= numLeft) {
91         // attach current elem to currSet
92         for (int i = currInd; i < allGroupElem.length; i++) {

```

```

93     currSet.add(allGroupElem[i]);
94     getNext(i + 1, numLeft - 1, allGroupElem, currSet, allSets);
95     currSet.remove(allGroupElem[i]);
96 }
97 }
98 }
99
100 /* go through the candidates, and return a set of candidates that are
101 * actually ADS
102 */
103 public Set<List<int[]>> getADS(Set<List<int[]>> candidates,
104     int[][] allGroupElements) {
105     Set<List<int[]>> adsSet = new LinkedHashSet<>();
106     for (List<int[]> candidate : candidates) {
107         if (this.isADS(candidate, allGroupElements)) {
108             adsSet.add(candidate);
109         }
110     }
111
112     return adsSet;
113 }
114
115 /* convert int[] to String
116 */
117 protected String encode(int[] entry) {
118     String encoding = "";
119     for (int i : entry) {
120         encoding = i + encoding;
121     }
122     return encoding;
123 }
124
125 /* check whether the input is ADS
126 */
127 public void isInputADS() {
128     Scanner in = new Scanner(System.in);
129     System.out.println("want to check ads? y/n");
130     String ans = in.next();
131     while (ans.equals("y")) {
132         System.out.println("input the ADS candidate, each entry a line" +
133             ", numbers in each entry separated by space");
134         List<int[]> candidate = new ArrayList<>();
135         for (int order = 0; order < ADSOrder; order++) {
136             int[] entry = new int[dimension];
137             for (int dim = 0; dim < dimension; dim++) {
138                 entry[dim] = in.nextInt();

```

```

I39     }
I40     candidate.add(entry);
I41     }
I42
I43     boolean res = this.isADS(candidate, getAllGroupElements());
I44     if (res) {
I45         System.out.println("input is ADS");
I46     } else {
I47         System.out.println("input is not ADS");
I48     }
I49     System.out.println("continue? y/n");
I50     ans = in.next();
I51 }
I52 }
I53
I54 /* check whether a candidate is really an ADS
I55 */
I56 private boolean isADS(List<int[]> candidate, int[][] allGroupElements) {
I57     int lambdaCoverCounter = 0;
I58     Map<String, Integer> coverCounter = new HashMap<>();
I59     // initialize the time of coverage to 0 for all elements
I60     for (int[] elem : allGroupElements) {
I61         // convert each element into String
I62         coverCounter.put(encode(elem), 0);
I63     }
I64
I65     // go through the candidate set, do corresponding subtractions, and
I66     // keep updating how many times each elem is covered, and increment lambda
I67     // counter if an elements gets covered for lambda times
I68     for (int[] first : candidate) {
I69         for (int[] second : candidate) {
I70             // first - second
I71             if (first != second) {
I72                 int[] diffArr = new int[dimension];
I73                 for (int digit = 0; digit < dimension; digit++) {
I74                     diffArr[digit] = (first[digit] + array[digit] - second[digit]) % array[digit];
I75                 }
I76
I77                 // if already reach lambda times
I78                 String encodingOfDiffArr = encode(diffArr);
I79                 int currCount = coverCounter.get(encodingOfDiffArr);
I80                 if (currCount == lambda) {
I81                     //System.out.println(encodingOfDiffArr + " covered " + (lambda + 1) + " times");
I82                     return false;
I83                 }
I84             }
I85         }
I86     }

```

```

185         if (currCount == lambda - 1) {
186             lambdaCoverCounter++;
187         }
188         coverCounter.put(encodingOfDiffArr, currCount + 1);
189     }
190     if (lambdaCoverCounter > numElemLambda) {
191         //System.out.println("more than numElemLambda covered lambda times");
192         return false;
193     }
194 }
195 }
196 }
197 return true;
198 }
199
200 /* returns all group elements
201 * [r]: each element
202 * [c]: the number at each dimension of each element
203 */
204 public int[][] getAllGroupElements() {
205     int[] divides = new int[dimension];
206     divides[dimension - 1] = 1;
207     for (int i = dimension - 2; i >= 0; i--) {
208         divides[i] = divides[i + 1] * array[i + 1];
209     }
210
211     // / last digit, mod curr digit
212     int[][] allElem = new int[groupOrder][dimension];
213     for (int i = 0; i < groupOrder; i++) {
214         for (int dim = 0; dim < dimension; dim++) {
215             allElem[i][dim] = (i / divides[dim]) % array[dim];
216         }
217     }
218
219     return allElem;
220 }
221
222
223 /* print out all the ADS
224 */
225 public void printADS(Set<List<int[]>> ads) {
226     for (List<int[]> set : ads) {
227         System.out.print("(");
228         for (int[] entry : set) {
229             System.out.print(Arrays.toString(entry) + " ");
230         }

```



```

231     System.out.println("");
232 }
233 }
234
235 /* filter out the multiples by removing them from the hashSet
236  *
237 */
238 public Set<List<int[]>> filterMultiples(Set<List<int[]>> ads) {
239     System.out.println("in filter");
240     HashMap<String, List<int[]>> hm = new LinkedHashMap<>();
241     // put each ads into the hm, the key is the string concatenation of
242     // all its bits
243     for (List<int[]> eachADS : ads) {
244         String encoding = "";
245         for (int[] entry : eachADS) {
246             encoding += encode(entry);
247         }
248         hm.put(encoding, eachADS);
249     }
250     System.out.println("hm created w/ size " + hm.size());
251
252     int increment = 1;;
253     int multiple = 2;
254     if (groupOrder % 2 == 0) {
255         increment++;
256         multiple++;
257     }
258
259     List<String> encodingsToBeRemoved = new ArrayList<>();
260     System.out.println("generating encodings to be removed");
261     for (String encoding : hm.keySet()) {
262         if (!encodingsToBeRemoved.contains(encoding)) {
263             for (int i = multiple; i < groupOrder; i += increment) {
264                 String multEncoding = encodeMultiple(i, hm.get(encoding));
265                 encodingsToBeRemoved.add(multEncoding);
266             }
267         }
268     }
269
270     System.out.println("generated encodings to be removed w/ size " +
271         encodingsToBeRemoved.size() + ", now removing");
272     for (String multEncoding : encodingsToBeRemoved) {
273         hm.remove(multEncoding);
274     }
275
276     return (new LinkedHashSet<List<int[]>>(hm.values()));

```

```

277 }
278
279 /* get multiple encoding
280 */
281 public String encodeMultiple(int multiply, List<int[]> currentSet) {
282     String encoding = "";
283     List<int[]> multipleSet = new ArrayList<>();
284
285     for (int[] currEntry : currentSet) {
286         int[] newEntry = new int[dimension];
287         for (int digit = 0; digit < dimension; digit++) {
288             newEntry[digit] = currEntry[digit] * multiply % array[digit];
289         }
290         multipleSet.add(newEntry);
291     }
292     for (int i = dimension - 1; i >= 0; i--) {
293         final int index = i;
294         Collections.sort(multipleSet, (a, b) -> a[index] - b[index]);
295     }
296     for (int[] newEntry : multipleSet) {
297         encoding += encode(newEntry);
298     }
299
300     return encoding;
301 }
302 }

```

**Listing A.1:** Stage 1 - non-cyclic

The second part consists of the code in Stage 2, where only query in cyclic is supported, but the group order  $v$  supported increased due to the speed up in the algorithm.

In a query for ADS, the program does the following:

- OneDimADSGenerator (line 12) gets called to initialize all fields, by calling the constructor of its super class.
- getAllGroupElements (defined and implemented in the super class) gets called to generate all group elements  $g \in G$ , from which the elements of ADS candidates will be selected from
- getADSCandidates (line 49) gets called to find a potential ADS candidate, by recursively calling getNext() (line 64) which uses the backtrack approach. After the inclusion of each candidate element, the program throws the element out if either  $\exists g \in G$ , such that  $g$  is covered more than  $\lambda$  times, or there are already  $t + 1$  elements that are covered

$\lambda$  times. Otherwise, the program keeps adding elements until the size of ADS candidate set reaches  $k$ . When such a valid ADS  $D$  is found, the method returns  $D$ .

- `getADS` (line 132) gets called to return the single valid ADS  $D$  just found. This method is not functionally necessary, but is implemented here for the sake of the inheritance relationship between the parent and child class.
- `printADS` (line 35) gets called to print the ADS found.

```
1 import java.util.*;
2
3 public class OneDimADSGenerator extends ADSGenerator {
4     // additional field
5     int[] coverTimesCntr;
6
7     // keep track of all elements that are generated lambda times by curr ads
8     Map<String, int[]> lambdaElemTracker;
9
10    /* constructor
11     */
12    public OneDimADSGenerator() {
13        super();
14        coverTimesCntr = new int[this.groupOrder];
15        lambdaElemTracker = new HashMap<>();
16    }
17
18
19    /* another constructor
20     */
21    public OneDimADSGenerator(int groupOrder, int ADSOrder, int lambda, int numElemLambda) {
22        this.dimension = 1;
23        array = new int[dimension];
24        this.groupOrder = groupOrder;
25        array[0] = this.groupOrder;
26        this.lambda = lambda;
27        this.numElemLambda = numElemLambda;
28        this.ADSOrder = ADSOrder;
29        this.validADSPreCheck();
30
31        coverTimesCntr = new int[this.groupOrder];
32        lambdaElemTracker = new HashMap<>();
33    }
34
35    public void printADS(Set<List<int[]>> ads) {
```

```

36     for (List<int[]> set : ads) {
37         System.out.print("[");
38         String encoding = "";
39         for (int[] entry : set) {
40             System.out.print(entry[0] + " ");
41             encoding += encode(entry);
42         }
43         System.out.println("] - " + Arrays.toString(lambdaElemTracker.get(encoding)));
44     }
45 }
46
47 /* generate all possible sets of order ADSOrder
48 */
49 public Set<List<int[]>> getADSCandidates(int[][] allGroupElem) {
50     //System.out.println("in one-dim getADSCandidates");
51     List<int[]> currSet = new ArrayList<>();
52     int firstElemInd = 0;
53     currSet.add(allGroupElem[firstElemInd]);
54     firstElemInd++;
55
56     Set<List<int[]>> allSets = new LinkedHashSet<>();
57     getNext(firstElemInd, ADSOrder - 1, allGroupElem, currSet, allSets, 0);
58     //System.out.println(allSets.size());
59     return allSets;
60 }
61
62 /* recursive method to help generate next ADS candidate
63 */
64 private void getNext(int currInd, int numLeft, int[][] allGroupElem,
65                     List<int[]> currSet, Set<List<int[]>> allSets,
66                     int numLambdaCovered) {
67     if (numLeft == 0) {
68         List<int[]> newSet = new ArrayList<>(currSet);
69         allSets.add(newSet);
70
71         // add all the elements that are covered lambda times to the hashmap
72         // tracker
73         int i = 0;
74         int[] elem = new int[this.numElemLambda];
75         //System.out.println("set just added, and numLmbdaCover " + numLambdaCovered);
76         for (int j = 0; j < coverTimesCntr.length; j++) {
77             if (coverTimesCntr[j] == lambda) {
78                 elem[i++] = j;
79             }
80         }
81         String encoding = "";

```

```

82     for (int[] entry : newSet) {
83         encoding += encode(entry);
84     }
85     lambdaElemTracker.put(encoding, elem);
86     return;
87
88 } else if (allGroupElem.length - currInd >= numLeft) {
89     List<Integer> differences = new ArrayList<>();
90     // attach current elem to currSet
91     int i = currInd;
92     // add new to set, update ctr
93     boolean stop = false;
94     for (int[] num : currSet) {
95         int diff1 = (num[0] - allGroupElem[i][0] + groupOrder) % groupOrder;
96         int diff2 = (allGroupElem[i][0] - num[0] + groupOrder) % groupOrder;
97         differences.add(diff1);
98         differences.add(diff2);
99         coverTimesCtr[diff1]++;
100        coverTimesCtr[diff2]++;
101        if (coverTimesCtr[diff1] == lambda) {
102            numLambdaCovered++;
103        }
104        if (coverTimesCtr[diff2] == lambda && diff2 != diff1) {
105            numLambdaCovered++;
106        }
107
108        if (coverTimesCtr[diff1] > lambda ||
109            coverTimesCtr[diff2] > lambda ||
110            numLambdaCovered > this.numElemLambda) {
111            // stop with this num since rules are broken
112            stop = true;
113            break;
114        }
115    }
116    if (!stop) {
117        currSet.add(allGroupElem[i]);
118        for (int k = i + 1; k < groupOrder; k++) {
119            getNext(k, numLeft - 1, allGroupElem, currSet, allSets, numLambdaCovered);
120            if (allSets.size() > 0)
121                return;
122        }
123        currSet.remove(allGroupElem[i]);
124    }
125
126    for (int diff : differences) {
127        coverTimesCtr[diff]--;

```

```

128     }
129 }
130 }
131
132 public Set<List<int[]>> getADS(Set<List<int[]>> candidates,
133     int[][] allGroupElements) {
134     return candidates;
135 }
136 }

```

**Listing A.2:** Stage 2 - cyclic

The last part consists of the code in Stage 3, where queries in both cyclic and non-cyclic groups, and large group order  $v$  are supported, due to a modification in data structures used. In a query for ADS, the program does the following:

- ADSGen (line 19) constructor gets called, and it calls readInput (line 60) to read input from user, do parameters validity check, and initialize all fields.
- getADS (line 127) gets called to find potential ADS candidate, by calling getNext() (line 146) which uses the backtrack approach, similar to in Stage 2. When a valid ADS  $D$  is found, the method returns  $D$ .
- printADS (line 111) gets called to print the ADS found

```

1 import java.util.*;
2 public class ADSGen {
3     // fields
4     protected int dimension; // dimension of the space in concern
5     protected int[] dim; // array holding info for R_i in index i
6     protected int v; // v = |G|
7     protected int t; // number of elements to be covered for lambda times
8     protected int lambda; // t elements covered LAMBDA times
9     protected int k; // ADS order
10
11     private final int A = (int)'A';
12
13     // keep track of times each elem is covered
14     protected Map<String, Integer> counter;
15     protected List<String> tElements;
16
17     /*constructor
18     */
19     public ADSGen() {

```

```

20     this.readInput();
21
22     this.initCntr();
23     for (String s: counter.keySet()) {
24         System.out.println(s + ", " + counter.get(s));
25     }
26
27     // elements that are covered lambda times
28     tElements = new ArrayList<>();
29 }
30
31 private void initCntr() {
32     counter = new LinkedHashMap<>();
33
34     Queue<String> q1 = new LinkedList<>();
35     Queue<String> q2 = new LinkedList<>();
36     q1.add("");
37
38     for (int d = 0; d < dimension; d++) {
39         while (!q1.isEmpty()) {
40             String s = q1.poll();
41
42             // for each str, attach next dim
43             for (int i = 0; i < dim[d]; i++) {
44                 q2.add(new String(s + (char)((int)'A' + i)));
45             }
46         }
47         q1 = q2;
48         q2 = new LinkedList<>();
49     }
50
51     while (!q1.isEmpty()) {
52         counter.put(q1.poll(), 0);
53     }
54 }
55
56 /* readInput from user, error checking before initializing all fields
57 * returns an array containing [dimension, dim[0]..[n - 1], v, k, lambda,
58 * t]
59 */
60 private void readInput() throws IllegalArgumentException {
61     Scanner in = new Scanner(System.in);
62     System.out.println("input format:");
63     System.out.println("dimension\n" +
64         "n1 n2 n3 ... n_dimension\n" +
65         "k");

```

```

66
67     dimension = in.nextInt();
68     if (dimension < 1) {
69         throw new IllegalArgumentException("dimension should be positive");
70     }
71
72     dim = new int[dimension];
73     v = 1;
74     for (int i = 0; i < dim.length; i++) {
75         // read in n in Z_n, and store n in each index
76         dim[i] = in.nextInt();
77         if (dim[i] < 1) {
78             throw new IllegalArgumentException("n should be positive");
79         }
80         v *= dim[i];
81     }
82
83     k = in.nextInt();
84     if (k > v) {
85         throw new IllegalArgumentException("k should be smaller than v");
86     }
87     lambda = findLambda(v, k);
88     t = k * (k - 1) - (lambda - 1) * (v - 1);
89
90     // print out all inputs
91     for (int i = 0; i < dim.length - 1; i++) {
92         System.out.print("Z_" + dim[i] + " * ");
93     }
94     System.out.println("Z_" + dim[dim.length - 1]);
95
96     System.out.printf("ads: (%s, %s, %s, %s)\n", v, k, lambda, t);
97 }
98
99 /* given v and k, find corresponding lambda
100 */
101 private int findLambda(int v, int k) {
102     int l = 0;
103     while ((v - 1) * l < k * (k - 1)) {
104         l++;
105     }
106     return l;
107 }
108
109 /* print out all ADS
110 */
111 public void printADS(Set<List<String>> ads) {

```



```

I12     for (List<String> set : ads) {
I13         System.out.print("{ ");
I14         for (String str: set) {
I15             System.out.print(" ");
I16             for (char c : str.toCharArray()) {
I17                 System.out.print((int)(c - 'A') + " ");
I18             }
I19             System.out.print(" ");
I20         }
I21         System.out.println("}");
I22     }
I23 }
I24
I25 /* find ADS with order k by using backtrack
I26 */
I27 public Set<List<String>> getADS() {
I28     List<String> elemOfG = new ArrayList<>(this.counter.keySet());
I29     List<String> currSet = new ArrayList<>();
I30     Set<List<String>> allSets = new LinkedHashSet<>();
I31
I32     int firstElemInd = 0;
I33
I34     // fix 0 in the set
I35     currSet.add(elemOfG.get(firstElemInd));
I36     firstElemInd++;
I37
I38     getNext(firstElemInd, k - 1, elemOfG, currSet, allSets, 0);
I39
I40     System.out.println("ADS size: " + allSets.size());
I41     return allSets;
I42 }
I43
I44 /* recursive method to help generate next ADS candidate
I45 */
I46 private void getNext(int currInd, int numLeft, List<String> elemOfG,
I47                     List<String> currSet, Set<List<String>> allSets,
I48                     int numLambdaCovered) {
I49
I50     if (numLeft <= 0) {
I51         allSets.add(new ArrayList<>(currSet));
I52
I53         // put the t elements into record
I54         for (String diff : counter.keySet()) {
I55             if (counter.get(diff) == lambda) {
I56                 tElements.add(diff);
I57

```

```

158         return;
159     }
160 }
161 } else if (elemOfG.size() - currInd >= numLeft) {
162     int i = currInd;
163     boolean stop = false;
164     List<String> differences = new ArrayList<>();
165     for (String prevElem: currSet) {
166         String newDiff1 = new String();
167         String newDiff2 = new String();
168         for (int d = 0; d < dimension; d++) {
169             char diff1Char = (char)((((int)(prevElem.charAt(d) - elemOfG.get(i).charAt(d)) +
170                 dim[d]) % dim[d] + A);
171             char diff2Char = (char)((((int)(elemOfG.get(i).charAt(d) - prevElem.charAt(d)) +
172                 dim[d]) % dim[d] + A);
173
174
175             newDiff1 += diff1Char;
176             newDiff2 += diff2Char;
177         }
178         differences.add(newDiff1);
179         differences.add(newDiff2);
180
181
182         int count1 = counter.put(newDiff1, counter.get(newDiff1) + 1) + 1;
183         int count2 = counter.put(newDiff2, counter.get(newDiff2) + 1) + 1;
184
185         if (count1 == lambda) {
186             numLambdaCovered++;
187         }
188
189         if (!newDiff1.equals(newDiff2) && count2 == lambda) {
190             numLambdaCovered++;
191         }
192
193         if (count1 > lambda || count2 > lambda || numLambdaCovered > this.t) {
194             stop = true;
195             break;
196         }
197     }
198     if (!stop) {
199         currSet.add(elemOfG.get(i));
200         for (int m = i + 1; m <= v - (numLeft - 1); m++) {
201             getNext(m, numLeft - 1, elemOfG, currSet, allSets, numLambdaCovered);
202             if (allSets.size() > 0)
203                 return;

```

```
204     }
205     currSet.remove(elemOfG.get(i));
206 }
207
208     for (String diff: differences) {
209         counter.put(diff, counter.get(diff) - 1);
210     }
211 }
212 }
213 }
```

**Listing A.3:** Stage 3