

# Mastermind Report

รัชมน ราชเวียง  
(65070501027)

ธิชาวน์ พิทิสมบูรณ์  
(65070501028)  
คณัสส์ สุวรรณรัตน์  
(65070501068)

รอชีน มุเก็ม  
(65070501047)  
ธันวา ใจสูงเนิน  
(63070501095)

สุวิจักขณ์ เรียวแรงบุญญา  
(65070501058)

## 1 บทนำ

### 1.1 ที่มาและความสำคัญ

เกม Mastermind เป็นเกมที่เกี่ยวกับการแก้ปริศนารหัสลับ โดยผู้เล่นจะต้องพยายามสืบสานคำใบ้ที่ได้มา หากพยายามแล้วจะทำให้ได้คำใบ้ที่บ่งบอกว่าคำทายนั้นใกล้เคียงกับรหัสของคำตอบเท่าใด ซึ่งผลลัพธ์และประสิทธิภาพของเกมนั้นสามารถวัดได้จากจำนวนครั้งที่ใช้ในการพยายาม หากพยายามถูกด้วยจำนวนครั้งในการพยายามที่น้อย จะหมายถึงการแก้ปัญหาด้วยประสิทธิภาพ ผู้เล่นจะต้องใช้เหตุและผลต่าง ๆ ในการคาดเดารหัสเหล่านั้นหรืออาจจะพึงการเสียงดวงในการหาคำตอบได้ ซึ่งสุดท้ายแล้วการใช้เหตุผลและตรรกะจะต้องเปลี่ยนไปตามสถานการณ์และเสียงดวง ทำให้หัววิธีที่ดีสุดได้ยาก

ทางผู้จัดทำจึงสนใจที่จะหัววิธีการแก้ปัญหาเกม Mastermind ที่ดีที่สุด โดยสามารถเขียนและแสดงผลเป็นโปรแกรมภาษาซีได้ เพื่อเป็นการสาธิตการพยายามรหัสสี พร้อมกับการวัดค่าจำนวนครั้ง การพยายามเฉลี่ยจากการหัสรส์ที่เป็นไปได้ทั้งหมด และเวลาที่ใช้ ซึ่งวิธีหลัก ๆ ที่ใช้คือ Genetic Algorithm ในการพยายามรหัสลับ เนื่องด้วยประสิทธิภาพที่มีแนวโน้มใช้การพยายามน้อย และมีหลากหลาย Parameter ที่สามารถปรับแต่งได้ และจะนำไปเปรียบเทียบกับการใช้ Permutation ในการพยายามผล เพื่อให้เห็นความแตกต่างในแต่ละวิธีการแก้ปัญหา Mastermind ทั้งด้านจำนวนการพยายามที่ใช้ และด้านเวลาที่ใช้ในการประมวลผล

### 1.2 วัตถุประสงค์

- เพื่อศึกษาและเปรียบเทียบหัววิธีการแก้ปัญหาเกม Mastermind ที่ดีที่สุด
- เพื่อศึกษาการเขียน Genetic Algorithm และ Permutation ในภาษาซี
- เพื่อเป็นการฝึกประยุกต์ใช้ความรู้อัลกอริทึมที่ได้ศึกษามาใน การแก้ปัญหาในชีวิตประจำวัน

### 1.3 เครื่องมือการดำเนินงาน

- คอมพิวเตอร์
- โปรแกรม Code::Blocks
- โปรแกรม Core Temp
- อุปกรณ์ทำความสะอาด เช่น พัดลม เครื่องปรับความเย็น

### 1.4 ประโยชน์ที่คาดว่าจะได้รับ

- เพื่อศึกษาและเปรียบเทียบหัววิธีการแก้ปัญหาเกม Mastermind ที่ดีที่สุด
- เพื่อศึกษาการเขียน Genetic Algorithm และ Permutation ในภาษาซี
- เพื่อเป็นการฝึกประยุกต์ใช้ความรู้อัลกอริทึมที่ได้ศึกษามาใน การแก้ปัญหาในชีวิตประจำวัน

## 2 เอกสารอ้างอิง

ในบทนี้อธิบายถึงทฤษฎีและเอกสารอ้างอิงที่เกี่ยวข้องเพื่อใช้เป็นแนวทางสำหรับการดำเนินงาน โดยประกอบด้วยเนื้อหาดังนี้

### 2.1 เกี่ยวกับ Mastermind

Mastermind เป็นเกมแก้ไขหารรหัสลับ

#### 2.1.1 Objective of Mastermind

เกม Mastermind มีเป้าหมายคือการพยายามหารหัสลับให้ได้เร็วที่สุด

#### 2.1.2 Setup for Mastermind

โดยการเตรียมเล่นเกม Mastermind มีขั้นตอนดังนี้

- จะมีการเลือกมา 2 ฝ่ายคือ คนสร้างรหัสลับ (Codemaker) และคนไขรหัสลับ (Codebreaker)
- จะมีการทำหมอดเวลาว่าจะต้องเข้ารหัสให้ได้ภายในกี่รอบ
- จะมีการทำหมอดจำนวนตัวรหัสลับที่คนไขรหัสลับที่ต้องแก้

#### 2.1.3 How to Play Mastermind

- Codemaker จะทำการสร้างรหัสลับออกมาโดยคนสร้างรหัสลับโดยจะสามารถใช้สีอะไรก็ได้ และสามารถใช้สีซ้ำได้ ภายในจำนวนที่ได้กำหนดไว้
- Codebreaker จะเล่นทำการไขรหัสลับ โดยจะพยายามและตำแหน่งของรหัส
- Codemaker จะทำการตรวจสอบผลลัพธ์มาดังนี้
  - โดยจะให้ หมุดสีดำ เมื่อรหัสมีสีและตำแหน่งที่ถูกต้อง
  - จะให้ หมุด สีขาว เมื่อรหัสมีสีถูกต้อง
  - ไม่ให้ หมุดสี เมื่อรหัสไม่มีสีถูกต้อง

โดย Codemaker ไม่จำเป็นต้องใส่หมุดเรียงเพื่อความท้าทาย

- Codebreaker จะนำข้อมูลที่ได้มา ทายรหัสลับต่อไป

### 2.2 Efficient solutions for Mastermind using genetic algorithms

ในเอกสารฉบับนี้ทางผู้เขียนได้นำเสนอวิธีการใช้ Genetic Algorithm ใน การแก้ปัญหาเกม Mastermind ซึ่งเป้าหมายหลักที่ต้องการสังเกตคือจำนวนครั้งเฉลี่ยที่จะต้องใช้ในการพยายามห้าม โดยที่จะเฉลี่ยมาจากจำนวนเกม ใจกลางของไอเดียใน algorithm นี้คือการสร้างเซตคัดเลือกรูปแบบการพยายามแต่ละรุ่น (generation) ที่รวมเก็บเข้ามา คุณภาพของแต่ละตัวจะตัดสินจากการเทียบกับรูปแบบการพยายามในเซตคัดเลือกนั้น ๆ ซึ่งเมื่อมีมีสิ่อยู่ 6 สีและซองที่ใส่อยู่ 4 ช่อง จะทำให้มีรูปแบบการใส่ทั้งหมด  $6^4$  หรือคือ 1296 คำตอบในการประเมินคุณภาพ ถ้ามีจำนวนสีที่เลือกได้และรูปแบบมากกว่านี้ จะมีชุดข้อมูลที่เกิดขึ้นมาใหม่โดยการสุ่มเลือกขึ้นมา ซึ่งทางเอกสารได้ทดลองและพบว่าวิธีการนี้สามารถหาถูกได้เร็วกว่าวิธีการอื่น ๆ อย่างเห็นได้ชัด ไม่ว่าจะกำหนดซ่องหรือสีให้มากขึ้นก็ตาม

#### 2.3 Mastermind Five Guess Algorithm

Donald Knuth's five guess algorithm เป็น algorithm ที่แสดงให้เห็นว่า codebreaker สามารถแก้หารหัสลับในภายใน 5 turn หรือ น้อยกว่า โดยใช้ algorithm ที่ค่อยทำการลดจำนวนคำตอบที่เป็นไปได้ และจะทำงานตามนี้

- สร้างเซตคำตอบ 5 ที่เป็นไปได้ทั้งหมด (1296 คำตอบ)
- ทำการเริ่มพยายามด้วย 1122
- ดูการแสดงผลที่ได้กลับมาว่ามี หมุดสีดำ หมุดสีขาว กี่หมุด
- ถ้า Codemaker ตอบกลับมาด้วย หมุดสีดำ 4 หมุด คือเกมจบ แล้วหยุดการทำงาน
- หาก ยังไม่ใช่คำตอบให้ทำการลบคำตอบ ที่ไม่เป็นไปตามการแสดงผลที่ได้มาก่อนหน้านี้

2.3.6 ใช้วิธี minimax เพื่อหาค่าที่จะใช้ทายต่อไป โดยค่าที่จะใช้ทายจะไม่ได้หากใน เขตคำตอบ S แต่เป็นค่าทั้งหมดที่เป็นไปได้ โดยการหาคำตอบที่จะโดนตัดทิ้ง ตามจำนวนหมุดสีขาวและดำ โดยคำตอบที่จะทำการทายเป็น ค่าที่เป็นไปได้ที่ต่ำที่สุดที่ จะถูกตัดออกจากคำตอบ

โดยใน 1 รอบผ่านคำตอบที่ไม่ได้ซ้ำ 1296 คำตอบ และจะส่ง ค่า hit count ตามความเป็นไปได้ของหมุดสีดำและขาว และทำการสร้างเขตคำตอบที่มีค่า max score ที่น้อยที่สุด แล้วทำการเลือกคำตอบ จากเขตคำตอบก่อนหน้านี้มาหนึ่งตัว แล้วทำการทายด้วยค่านั้นกับสมาชิกของเขต S ที่เป็นไปได้

Knuth จะทำการเลือกคำตอบที่จะใช้ทายเป็นตัวเลข Knuth ได้แสดงตัวอย่างใน บาง Case ที่ไม่มี สมาชิก ของ S ในค่ามาก score สุดที่ทำการทาย ทำให้ไม่สามารถชนะได้ในรอบต่อไป แต่จำสำหรับการแข่งขันภายนอกใน 5 รอบ

### 2.3.7 กลับไปทำงานในขั้นตอนที่ 3

## 2.4 Multiplication Principle (กฎการคูณ)

กฎการคูณ (Multiplication Principle) เป็นรูปแบบการหาคำตอบของวิธีการทั้งหมดอย่างหนึ่ง หากมีลำดับขั้นในการเลือกแบ่งออกเป็น k ขั้นตอน โดยที่การเลือกนั้นมีความต่อเนื่องกันซึ่ง

- ขั้นตอนที่ 1 สามารถเลือกได้  $n_1$  วิธี
- ขั้นตอนที่ 2 สามารถเลือกได้  $n_2$  วิธี
- ขั้นตอนที่ 3 สามารถเลือกได้  $n_3$  วิธี
- ขั้นตอนที่ k สามารถเลือกได้  $n_k$  วิธี

จะสามารถคำนวณจำนวนวิธีทั้งหมดในการเลือก k ขั้นตอนได้ ทั้งหมด  $n_1 \times n_2 \times n_3 \times \dots \times n_k$  วิธี

## 2.5 วิธีเรียงสับเปลี่ยน (Permutation)

วิธีเรียงสับเปลี่ยน หมายถึง การจัดเรียงอันดับสิ่งของโดยถือเอาอันดับเป็นสำคัญ เช่น การเรียงสับเปลี่ยนตัวอักษร 3 ตัว คือ A, B และ C นำมาจัดเรียงอันดับทั้งหมดได้เป็น ABC, ACB, BAC, CAB, CBA ซึ่งจะเห็นได้ว่า วิธีเรียงสับเปลี่ยนตัวอักษรทั้ง 3 ตัวนี้ มี 6 วิธี

การจำแนกวิธีทั้งหมดในการเรียงสับเปลี่ยนจะนำกฎเกณฑ์ เป็นต้นเกี่ยวกับการนับมาใช้ โดยถือเสมอว่าการจัดอันดับแต่ละอันดับเป็นการทำงานอย่างหนึ่ง เช่น การจัดเรียงตัวอักษร 3 ตัว ข้างต้นเป็นการทำงาน 3 อย่าง คือ

การจัดตัวอักษรในตำแหน่งที่ 1 มี 3 วิธี (A หรือ B หรือ C) ในแต่ละวิธีสามารถจัดตัวอักษรใน ตำแหน่งที่ 2 ได้อีก 2 วิธี (ตัวอักษรที่เหลือ) และในแต่ละวิธีของการจัดตัวอักษรในตำแหน่งที่ 1 และตำแหน่งที่ 2 จะจัดตัวอักษรในตำแหน่งที่ 3 ได้อีก 1 วิธี จำนวนวิธีทั้งหมดในการจัดอันดับตัวอักษรที่แตกต่างกัน จึงเท่ากับ  $3 * 2 * 1 = 6$  วิธี (ตามกฎข้อ 3)

จำนวนวิธีเรียงสับเปลี่ยนสิ่งของ n สิ่ง ซึ่งแตกต่างกันทั้งหมด โดยจัดทีละ r สิ่ง เอียนแทนด้วย  $P_{n,r}$  จากกฎข้อที่ 4 จะได้ วิธีเรียงสับเปลี่ยนสิ่งของที่ไม่แตกต่างกันทั้งหมด

$$P_{n,r} = \frac{n!}{(n-r)!}$$

$$P_{n,n} = n!$$

Figure 1: หนังสือบางเล่มอาจใช้สัญลักษณ์อื่นๆ แทน  $P_{n,r}$  เช่น  $nPr$ ,  ${}^nP_r$  หรือ  $P(n, r)$

สิ่งของที่ไม่แตกต่างกันทั้งหมดหรือสิ่งของที่เหมือนกันเป็นกลุ่ม ๆ เช่น มีร่องอยู่ 10 ผืน เป็นร่องสีแดง 2 ผืน สีเขียว 3 ผืน และสีเหลือง 5 ผืน ถือได้ว่าเป็นสิ่งของ 3 กลุ่ม ในแต่ละกลุ่มมีสิ่งของที่เหมือนกัน เมื่อนำสิ่งของทั้งหมดมาสับเปลี่ยนกัน สิ่งของที่เหมือนกันจะไม่ทำให้เกิดวิธีใหม่ จำนวนวิธีเรียงสับเปลี่ยนจึงน้อยกว่าวิธีเรียงสับเปลี่ยนสิ่งของที่แตกต่างกันทั้งหมด

โดยทั่วไปถ้ามีสิ่งของอยู่  $n$  สิ่งในจำนวนนี้มี  $n_1$  สิ่งที่เหมือนกันเป็นกลุ่มที่หนึ่ง มี  $n_2$  สิ่งที่เหมือนเป็นกลุ่มที่สอง มี  $n_3$  สิ่งที่เหมือนเป็นกลุ่มที่สาม... และมี  $n_k$  สิ่งที่เหมือนกันเป็นกลุ่มที่  $k$  โดยที่  $n_1 + n_2 + n_3 \dots + n_k = n$  ในวิธีเรียงสับเปลี่ยนของสิ่งของ  $n$  สิ่งที่แตกต่างกันทั้งหมด จะมีวิธีเรียงสับเปลี่ยน  $n!$  วิธี แต่ในกรณีที่มีของเหมือนกันเป็นกลุ่ม ๆ จำนวนวิธีจะน้อยลงไป คือในจำนวน  $n!$  วิธีดังกล่าว จะรวมวิธีเรียงสับเปลี่ยนที่ไม่แตกต่างกันได้ถึง  $n_1!$  วิธี สำหรับของกลุ่มแรกที่เหมือนกัน และ  $n_2!$  วิธี สำหรับของกลุ่มที่สองที่เหมือนกัน... และ  $n_k!$  สำหรับของกลุ่มที่  $k$  ที่เหมือนกัน ดังนั้น จึงมีวิธีเรียงสับเปลี่ยนที่สามารถเห็นความแตกต่างกันของสิ่งของทั้ง  $n$  สิ่งได้ เท่ากับ  $n! / n_1! n_2! n_3! \dots n_k!$  วิธี

### 3 วิธีการดำเนินงาน

การทดลองนี้ใช้ปัญหานักสืบในการเขียน code วิธีการแก้ปัญหานี้ในเกม Mastermind ด้วยภาษา C ด้วยวิธี Genetic algorithm หาค่า parameter ที่เหมาะสมที่สุดและเปรียบเทียบกับ วิธี permutation อื่นๆในการแก้ปัญหาดังกล่าว

1. เครื่องมือที่ใช้ในการดำเนินงาน
2. การเตรียมการดำเนินงาน
3. การดำเนินงาน
4. แผนการเก็บรวบรวมข้อมูล
5. การวิเคราะห์ข้อมูลที่ได้รับจากการดำเนินงาน

#### 3.1 เครื่องมือการดำเนินงาน

##### 3.1.1 คอมพิวเตอร์

1. Device specifications
  - คอมพิวเตอร์รุ่น ASUS TUF Gaming F15 FX507ZM FX507ZM
  - Processor : AMD Ryzen 7 6800H with Radeon Graphics
  - Graphics card : NVIDIA GeForce RTX 3060 Laptop GPU
  - Wireless card : MediaTek Wi-Fi 6 MT27921 Wireless LAN Card
  - Storage : SSD/EMMC 1 476 GB, INTEL SSDPEKNU512GZ
  - Memory : Total 16 GB
  - System type : 64-bit operating system, x64-based processor

## 2. Windows specifications

- Edition : Windows 11 Home Single Language
- Version : 22H2
- OS build : 22621.2715
- Experience : Windows Feature Experience Pack 1000.22677.1000.0

```
1 void genAllCase(int now){
2     if(now == CodeLength){
3         for(int i=0;i<CodeLength;i++){
4             allCase[caseNum][i]=per[i];
5         }
6         caseNum++;
7         return;
8     }
9
10    for(int i=1;i<=ColorsNum;i++){
11        per[now] = i;
12        genAllCase(now+1);
13    }
14 }
```

### 3.1.2 โปรแกรมสำหรับดำเนินงาน

1. Code::Blocks
2. Core Temp 1.18.1

### 3.1.3 อุปกรณ์ทำความสะอาด

1. พัดลมโน๊ตบุ๊ก

2. เครื่องปรับอากาศ

### 3.1.4 เอกสารประกอบการดำเนินงานเกี่ยวกับโปรแกรมที่ใช้ และข้อมูลเกี่ยวกับ Genetic algorithm และ เกม Mastermind

## 3.2 การเตรียมการดำเนินงาน

### 3.2.1 เขียนโค้ดการแก้ปัญหาในเกม Mastermind ด้วยเป็นภาษา C ด้วยวิธี Bruteforce Permutation ที่จะทายรหัสจนกว่าเจอรหัสที่ถูกต้อง

1. Function genAllCase สำหรับการสร้างรหัสทั้งหมดที่เป็นไปได้จากความยาวของรหัสและจำนวนสีที่กำหนดเพื่อนำไปทายรหัสจนกว่าเจอรหัสที่ถูกต้อง
2. Function checkCode สำหรับการตรวจรหัสที่ทายกับรหัสลับโดยจะมีวิธีการดังนี้

- พิจารณารหัสที่ทายกับรหัสลับมีสีใดที่ตรงกันบ้างเมื่อเจอสีที่ตรงกันและสีของรหัสลับซึ่งนั้นยังไม่ตรงกับสีอื่นในรหัสที่ทายก่อนหน้าให้เพิ่มค่าหมุดสีขาวขึ้นไป 1

Figure 2: Function createCase

- พิจารณาในทำແນ່ງເຕີຍວັນຂອງຮ້າສ້າງສອງວ່າມີສີຕະຈົກນໍ້າໂມ່ໄຫ້ໃຊ້ໃຫ້ເພີ່ມຄ່າໜຸດສື່ດຳຂຶ້ນໄປ 1
- ສຸດທ້າຍລັບຄ່າໜຸດສື່ຂາວດ້ວຍຄ່າຂອງໜຸດສື່ດຳແລະຄືນຄ່າໜຸດສື່ຂາວ ອູ້ສື່ດຳທີ່ເປັນ global variable

```
1 int whitePeg,blackPeg;
2 int foundedPosition[CodeLength];
3 void checkCode(int guess[],int code[]){
4
5     whitePeg=0,blackPeg=0;
6     memset(foundedPosition, 0, (CodeLength) * sizeof(int));
7
8     for(int i=0;i<CodeLength;i++){
9         for(int j=0;j<CodeLength;j++){
10            if(guess[i]==code[j] && foundedPosition[j]!=1 ){
11                foundedPosition[j]=1;
12                whitePeg++;
13                break;
14            }
15        }
16    }
17    for(int i=0;i<CodeLength;i++){
18        if(guess[i]==code[i]){
19            blackPeg++;
20        }
21    }
22    whitePeg-=blackPeg;
23 }
```

Figure 3: Function checkCode()

### 3.2.2 เขียนโค้ดการแก้ปัญหาในเกม Mastermind ด้วยเป็นภาษา C ด้วยวิธี Permutation ที่อ้างอิงจาก Knuth algorithm

1. Function `createCase` สำหรับการสร้างรหัสทั้งหมดที่เป็นไปได้จากความยาวของรหัสและจำนวนสีที่กำหนด

```

1 void createCase(int now){
2     if(now == CodeLength){
3         for(int i=0;i<CodeLength;i++){
4             possibleCode[caseNum][i]=per[i];
5         }
6         caseNum++;
7         return;
8     }
9
10    for(int i=1;i<=ColorsNum;i++){
11        per[now] = i;
12        createCase(now+1);
13    }
14}

```

Figure 4: Function `createCase`

2. Function `checkCode` สำหรับการตรวจรหัสที่พยายามหักลับโดยจะมีวิธีการดังนี้

- พิจารณารหัสที่พยายามหักลับมีสีใดที่ตรงกันบ้างเมื่อเจอสีที่ตรงกันและสีของรหัสลับซึ่งนั้นยังไม่ตรงกับสีอื่นในรหัสที่พยายามหักลับให้เพิ่มค่าหมุดสีขาวขึ้นไป 1
- พิจารณาในตำแหน่งเดียว กันของรหัสทั้งสองว่ามีสีตรงกันหรือไม่ หากใช้ให้เพิ่มค่าหมุดสีดำขึ้นไป 1
- สุดท้ายลบค่าหมุดสีขาวด้วยค่าของหมุดสีดำและคืนค่า หมุดสีขาว หรือ สีดำที่เป็น global variable

```

1 int calculateChance(int newGuess){
2
3     int score[caseNum];
4     for(int i=0;i<caseNum;i++){
5         if(possibleCode[i][0]==-1) continue;
6         score[i]=-1;
7         memset(pegScore, -1, 100 * sizeof(int));
8         for(int j=0;j<caseNum;j++){
9             if(possibleCode[j][0]==-1) continue;
10
11            checkCode(possibleCode[i],possibleCode[j]);
12            int peg = (blackPeg>10)+(whitePeg);
13            if(pegScore[peg] == -1) pegScore[peg]=1;
14            else pegScore[peg]++;
15        }
16        for(int j=0;j<10;j++){
17            if(pegScore[j]==-1) continue;
18            score[i] = max(pegScore[j],score[i]);
19        }
20    }
21    int minScore = 2e9,minIndex;
22    for(int i=0;i<caseNum;i++){
23        if(possibleCode[i][0]==-1) continue;
24        if(score[i]<minScore){
25            minScore = score[i];
26            minIndex = i;
27        }
28    }
29    memcpy(newGuess, possibleCode[minIndex], CodeLength * sizeof(int));
30}

```

Figure 5: Function `checkCode()`

3. Function `removeCase` และ `cutCase` สำหรับการนำรหัสที่กำหนดออกจากชุดรหัสที่เป็นไปได้ซึ่งใน function `cutCase` จะทำการนำรหัสที่มีผลลัพธ์หมุดต่างจากผลลัพธ์ของหมุดที่ไม่ตรงกันออก

```

1 void removeCase(int remove[]){
2
3     for(int i=0;i<caseNum;i++){
4         int ch=1;
5         for(int j=0;j<CodeLength;j++){
6             if(possibleCode[i][j]!=remove[j]){
7                 ch=0;
8             }
9         }
10        if(ch==1){
11            possibleCode[i][0]=-1;
12        }
13    }
14}

```

Figure 6: Function `removeCase`

```

1 void cutCase(int guess,int B,int W){
2     for(int i=0;i<caseNum;i++){
3         if(possibleCode[i][0]==-1) continue;
4         checkCode(possibleCode[i],guess);
5         if(blackPeg != B || whitePeg != W){
6             possibleCode[i][0]=-1;
7         }
8     }
9 }

```

Figure 7: Function `cutCase`

4. Function `calculateChance` สำหรับการคำนวณจำนวนรหัสที่สามารถนำออกได้มากที่สุดหากเลือกรหัสนี้เป็นการพยายามครั้งถัดไปเป็นคชนวนและเลือกคชนวนที่มีค่าน้อยที่สุดในการพยายามครั้งถัดไป

```

1 int calculateChance(int newGuess){
2
3     int score[caseNum];
4     for(int i=0;i<caseNum;i++){
5         if(possibleCode[i][0]==-1) continue;
6         score[i]=-1;
7         memset(pegScore, -1, 100 * sizeof(int));
8         for(int j=0;j<caseNum;j++){
9             if(possibleCode[j][0]==-1) continue;
10
11            checkCode(possibleCode[i],possibleCode[j]);
12            int peg = (blackPeg>10)+(whitePeg);
13            if(pegScore[peg] == -1) pegScore[peg]=1;
14            else pegScore[peg]++;
15        }
16        for(int j=0;j<10;j++){
17            if(pegScore[j]==-1) continue;
18            score[i] = max(pegScore[j],score[i]);
19        }
20    }
21    int minScore = 2e9,minIndex;
22    for(int i=0;i<caseNum;i++){
23        if(possibleCode[i][0]==-1) continue;
24        if(score[i]<minScore){
25            minScore = score[i];
26            minIndex = i;
27        }
28    }
29    memcpy(newGuess, possibleCode[minIndex], CodeLength * sizeof(int));
30}

```

Figure 8: Function `calculateChance`

5. เริ่มแรกจะใช้รหัส {1,1,2,2} ในการทายครั้งแรกโดยในแต่ละครั้งที่ทำการทายให้นำรหัสนั้นออกจากชุดรหัสที่เป็นไปได้แล้วหลังจากการทายให้ทำการนำรหัสที่เป็นไปไม่ได้ออกจากชุดรหัสเมื่อผลลัพธ์จากการทายครั้งล่าสุดเป็นตัวอักษร แล้วจึงไปหารหัสที่จะทายต่อไป

```

1 int guess[CodeLength] = {1,1,2,2};
2
3 printf("Code : ");
4 for(int i=0;i<CodeLength;i++){
5     printf("%d ",code[i]);
6 }
7 printf("\n");
8
9 int turn=1;
10 while(1){
11
12     removeCase(guess);
13
14     printf("%d Guess : ",turn);
15     for(int i=0;i<CodeLength;i++){
16         printf("%d ",guess[i]);
17     }
18     checkCode(guess,code);
19     printf(" => %d %d\n",blackPeg,whitePeg);
20     if(blackPeg == 4){
21         printf("Found\n",turn);
22         //printf(" %d\n",turn);
23         cnt+=turn;
24         break;
25     }
26     cutCase(guess,blackPeg,whitePeg);
27     calculateChance(guess);
28     turn++;
29 }
30 }
```

Figure 9: Function main

### 3.2.3 เขียนโค้ดการแก้ปัญหาในเกม Mastermind ด้วยเป็นภาษา C ด้วยวิธี Genetic algorithm

1. ค่า parameter ทั้งหมดที่ใช้ในการแก้ปัญหาเกม Mastermind ด้วย Genetic algorithm โดยจะมีดังนี้

- ColorsNum จำนวนสีทั้งหมด
- CodeLength ความยาวของรหัสลับ
- MaxPopSize จำนวนประชากรในแต่ละ generation
- MaxGen จำนวน Generation ทั้งหมดในการใช้ Genetic algorithm
- CrossOverChance โอกาสในการเกิด Crossover
- SwapMutateChance, ResetMutateChance และ InverseMutateChance โอกาสที่จะเกิด mutation แบบ Swap, Random และ Inversion ตามลำดับ

```

1 #define ColorsNum 6
2 #define CodeLength 4
3
4 #define MaxPopSize 60
5 #define MaxGen 100
6
7 #define CrossOverChance 50
8 #define SwapMutateChance 10
9 #define ResetMutateChance 3
10 #define InverseMutateChance 2
```

Figure 10: ค่า parameter ของ Genetic algorithm

2. Function checkCode สำหรับการตรวจรหัสที่ทายกับรหัสลับโดยจะมีวิธีการดังนี้

- พิจารณารหัสที่ทายกับรหัสลับมีสีใดที่ตรงกันบ้างเมื่อเจอสีที่ตรงกันและสีของรหัสลับของนั้นยังไม่ตรงกับสีอื่นในรหัสที่ทายก่อนหน้าให้เพิ่มค่าหมุดสีขาวขึ้นไป 1
- พิจารณาในตำแหน่งเดียวกันของรหัสทั้งสองว่ามีสีตรงกันหรือไม่หากใช้ให้เพิ่มค่าหมุดสีดำขึ้นไป 1
- สุดท้ายลบค่าหมุดสีขาวด้วยค่าของหมุดสีดำและคืนค่า หมุดสีขาว หรือ สีดำที่เป็น global variable

```

1 int whitePeg,blackPeg;
2 int foundedPosition[CodeLength];
3 void checkCode(int guess[],int code[]){
4
5     whitePeg=0,blackPeg=0;
6     memset(foundedPosition, 0, (CodeLength) * sizeof(int));
7
8     for(int i=0;i<CodeLength;i++){
9         for(int j=0;j<CodeLength;j++){
10             if(guess[i]==code[j] && foundedPosition[j]!=1 ){
11                 foundedPosition[j]=1;
12                 whitePeg++;
13                 break;
14             }
15         }
16     }
17
18     for(int i=0;i<CodeLength;i++){
19         if(guess[i]==code[i]){
20             blackPeg++;
21         }
22     }
23     whitePeg-=blackPeg;
24 }
25 }
```

Figure 11: Function checkCode

3. วิธีการ Crossover แบบ Single point Crossover ที่จะทำการสุ่มจุดฯหนึ่งของรหัสแล้วจึงให้รหัสของลูกที่เกิดจากการ Crossover ในตำแหน่งที่อยู่ระหว่างจุดที่สุ่มมาเป็นรหัสของแม่และหลังจุดที่ถูกสุ่มมาเป็นรหัสของพ่อ

```

1 int chance = rand() % 100;
2 if(chance < crossChance){
3     int point = rand() % CodeLength;
4     for(int i=0;i<CodeLength;i++){
5         if(i < point){
6             son[i]=mommy[i];
7         }
8         else{
9             son[i]=daddy[i];
10        }
11    }
12 }
```

Figure 12: Code วิธีการ Single point Crossover

4. วิธีการ Crossover แบบ Two-point Crossover ที่จะทำการสุ่มจุดสองจุดของรหัสแล้วจึงให้รหัสของลูกที่เกิดจากการ Crossover ในตำแหน่งที่อยู่ระหว่างจุดที่สุ่มมาเป็นรหัสของพ่อนอกเหนือจากนั้นเป็นรหัสของแม่

```

1 int chance = rand() % 100;
2 if(chance < crossChance){
3     int point1 = rand() % CodeLength;
4     int point2 = rand() % CodeLength;
5     if(point1 > point2){
6         swap(&point1,&point2);
7     }
8     for(int i=0;i<CodeLength;i++){
9         if(i < point1){
10             son[i]=mommy[i];
11         }
12         else if (i >= point1 && i < point2){
13             son[i]=daddy[i];
14         }
15         else{
16             son[i]=mommy[i];
17         }
18     }
19 }
```

Figure 13: Code วิธีการ Two-point Crossover

5. วิธีการ Crossover แบบ Uniform Crossover ที่จะทำการสุ่มในทุกตำแหน่งรหัสว่าจะเลือกใช้รหัสของพ่อหรือแม่

```

1 for(int i=0;i<CodeLength;i++){
2     int chance = rand() % 100;
3     if(chance < crossChance){
4         son[i]=mommy[i];
5     }
6     else{
7         son[i]=daddy[i];
8     }
9 }
```

Figure 14: Code วิธีการ Uniform Crossover

6. Function randMutate ซึ่งเป็นวิธีการ Mutate แบบ Random ที่จะทำการสุ่มตำแหน่งที่จะทำการ mutate แล้วจึงสุ่ม ณ ตำแหน่งนั้นใหม่

```

1 int resetChannce = ResetMutateChance;
2 void randMutate(int code[]){
3     int chance = rand() % 100;
4     if(chance < resetChannce){
5         int randPoint = rand() % CodeLength;
6         int randNumber = (rand() % ColorsNum)+1;
7         code[randPoint] = randNumber;
8     }
9 }
```

Figure 15: Function randMutate

7. Function swapMutate ซึ่งเป็นวิธีการ Mutate แบบ Swap ที่จะทำการสุ่มตำแหน่งมาสองตำแหน่งที่จะทำการสลับกัน

```

1 int swapChannce = SwapMutateChance;
2 void swapMutate(int code[]){
3     for(int i=0;i<CodeLength;i++){
4         int chance = rand() % 100;
5         if(chance < swapChannce){
6             int x = rand() % CodeLength ,y = rand() % CodeLength;
7             swap(&code[x],&code[y]);
8         }
9     }
10 }
```

Figure 16: Function swapMutate

8. Function inversionMutate ซึ่งเป็นวิธีการ Mutate แบบ Inverse ที่จะทำการสุ่มตำแหน่งมาสองตำแหน่งที่ทำการ reverse ในช่วงสองตำแหน่งนั้น

```

1 int inverseChannce = InverseMutateChance;
2 void inversionMutate(int code[]){
3     int chance = rand() % 100;
4     if(chance < inverseChannce){
5         int start = rand() % CodeLength;
6         int end = rand() % CodeLength;
7
8         if (start > end) {
9             swap(&start,&end);
10        }
11
12         while (start < end) {
13             swap(&code[start], &code[end]);
14             start++;
15             end--;
16         }
17     }
18 }
```

Figure 17: Function inversionMutate

9. Function genNewPop สำหรับการสร้างประชากรใหม่ จากประชากรเดิมโดยจะนำประชากรเดิม 2 ชุดมาสร้างเป็นประชากรใหม่โดยนำมา Crossover และ mutate ตามลำดับ

```

1 int selMutate[MutateMethod] = {1,1,1};
2 void genNewPop(CODE population[]){
3     for (int i = 0; i < MaxPopSize; i += 2) {
4
5         crossOver(population[i].code, population[i + 1].code, population[i].code);
6         crossOver(population[i + 1].code, population[i].code, population[i + 1].code);
7
8         if(selMutate[0] == 1){
9             randMutate(population[i].code);
10            randMutate(population[i+1].code);
11        }
12        else if(selMutate[1] == 1){
13            swapMutate(population[i].code);
14            swapMutate(population[i + 1].code);
15        }
16        else if(selMutate[2] == 1){
17            inversionMutate(population[i].code);
18            inversionMutate(population[i + 1].code);
19        }
20
21         population[i].fitness = 0;
22         population[i + 1].fitness = 0;
23     }
24 }
```

Figure 18: Function genNewPop

10. Function fitnessCal สำหรับคำนวนค่า fitness ในประชากรแต่ละชุดโดยจะทำการตรวจสอบหัสดูนั้นกับชุดรหัสและผลลัพธ์หมุนทุกตัวที่ถูกทำในครั้งที่ผ่านมา โดยใช้ function checkCode โดยค่า fitness คือผลรวมของผลต่างระหว่างหมุดสีขาวและสีดำของการทำลายครั้งนั้นกับผลลัพธ์จากการตรวจสอบที่ได้กล่าวข้างต้น

```

1 void fitnessCal(CODE pop[],CODE trail[],int trailCnt){
2
3     for(int i=0;i<MaxPopSize;i++){
4         pop[i].fitness=0;
5         for(int j=0;j<trailCnt;j++){
6             checkCode(pop[i].code,trail[j].code);
7             pop[i].fitness += abs(trail[j].black-blackPeg)+abs(trail[j].white-whitePe
8         }
9     }
10 }
11 }
```

Figure 19: Function fitnessCal

11. Function addEligible สำหรับเก็บประชากรที่มีค่า fitness เป็น 0 ที่ซึ่งจะไม่ได้เดิมกับรหัสลับมากที่สุดหากประชากรนั้นไม่ได้อยู่ในชุดประชากรนั้นที่ถูกเลือกอยู่แล้ว

```

1 int addEligible(CODE Eligibles[],CODE pop[],int eCnt){
2     for(int i=0;i<MaxPopSize;i++){
3         if(eCnt == MaxPopSize)
4             break;
5         if(pop[i].fitness == 0 && isDupe(pop[i].code,Eligibles,eCnt) == -1){
6             Eligibles[eCnt]=pop[i];
7             eCnt++;
8         }
9     }
10    return eCnt;
11 }
```

Figure 20: Function addEligible

12. Function popSelectTournament ที่เป็นวิธีการเลือกประชากรที่ใช้ในรุ่นต่อไปโดยวิธี Tournament Selection โดยจะทำการสุ่มเลือกประชากรตามจำนวนขนาดของ Tournament ที่กำหนดไว้แล้วเลือกประชากรที่มีค่า Fitness น้อยที่สุดมาเลือกเป็นประชากรชุดถัดไป

```

1 int tourSize = TournamentselectionSize;
2 void popSelectTournament(CODE pop[]){
3
4     int cntSel=0,mark[MaxPopSize];
5     CODE selected[MaxPopSize];
6
7     while(cntSel < MaxPopSize){
8         CODE competitor[tourSize];
9
10        for(int i=0;i<tourSize;i++){
11            int randSel = rand() % MaxPopSize;
12            competitor[i] = pop[randSel];
13        }
14        CODE winner = competitor[0];
15
16        for(int i=1;i<tourSize;i++){
17            if(competitor[i].fitness < winner.fitness){
18                winner = competitor[i];
19            }
20        }
21        selected[cntSel++] = winner;
22    }
23
24    for(int i=0;i<MaxPopSize;i++){
25        copyArr(pop[i].code,selected[i].code);
26        pop[i].fitness=selected[i].fitness;
27    }
28 }
29 }
```

Figure 21: Function popSelectTournament

13. Function popSelectRouletteWheel ที่เป็นวิธีการเลือกประชากรที่ใช้ในรุ่นต่อไปโดยวิธี Proportional Roulette Wheel Selection โดยจะทำการสุ่มเลือกประชากรโดยที่โอกาสในการสุ่มจะขึ้นอยู่กับค่า fitness ในแต่ละชุดยิ่งค่า fitness น้อยโอกาสในการถูกเลือกยิ่งมาก

```

1 void popSelectRouletteWheel(CODE pop[]) {
2
3     double totalFitness = 0.0;
4     for (int i = 0; i < MaxPopSize; i++) {
5         totalFitness += pop[i].fitness;
6     }
7
8     double probabilities[MaxPopSize];
9     if (totalFitness == 0.0) {
10         for (int i = 0; i < MaxPopSize; i++) {
11             probabilities[i] = 1.0 / MaxPopSize;
12         }
13     } else {
14         for (int i = 0; i < MaxPopSize; i++) {
15             probabilities[i] = 1.0 - (pop[i].fitness / totalFitness);
16         }
17     }
18
19     CODE selected[MaxPopSize];
20     int cntSel = 0;
21
22     while (cntSel < MaxPopSize) {
23         double r = ((double)rand() / RAND_MAX);
24         double cumulativeProbability = 0.0;
25
26         for (int i = 0; i < MaxPopSize; i++) {
27             cumulativeProbability += probabilities[i];
28             if (r <= cumulativeProbability) {
29                 selected[cntSel++] = pop[i];
30                 break;
31             }
32         }
33     }
34
35     for (int i = 0; i < MaxPopSize; i++) {
36         copyArr(pop[i].code, selected[i].code);
37         pop[i].fitness = selected[i].fitness;
38     }
39 }
```

Figure 22: Function popSelectRouletteWheel

14. Function popSelectRanking ที่เป็นวิธีการเลือกประชากรที่ใช้ในรุ่นต่อไปโดยวิธี Ranking Selection โดยจะทำการสุ่มเลือกประชากรโดยที่โอกาสในการสุ่มจะขึ้นอยู่กับอันดับในการเรียงประชากรจากน้อยไปมากยิ่งอันดับสูงโอกาสในการถูกเลือกยิ่งสูง

```

1 void popSelectRanking(CODE pop[]) {
2
3     int rank[MaxPopSize];
4     int totalRank = 0;
5
6     for (int i = 0; i < MaxPopSize; i++) {
7         int count = 0;
8         for (int j = 0; j < MaxPopSize; j++) {
9             if (pop[j].fitness < pop[i].fitness) {
10                 count++;
11             }
12         }
13         rank[i] = count + 1;
14         totalRank += rank[i];
15     }
16
17     double probabilities[MaxPopSize];
18     for (int i = 0; i < MaxPopSize; i++) {
19         probabilities[i] = (double)rank[i] / totalRank;
20     }
21
22     CODE selected[MaxPopSize];
23     int cntSel = 0;
24
25     while (cntSel < MaxPopSize) {
26
27         double r = ((double)rand() / RAND_MAX);
28         double cumulativeProbability = 0.0;
29
30         for (int i = 0; i < MaxPopSize; i++) {
31             cumulativeProbability += probabilities[i];
32             if (r <= cumulativeProbability) {
33                 selected[cntSel++] = pop[i];
34                 break;
35             }
36         }
37     }
38
39     for (int i = 0; i < MaxPopSize; i++) {
40         copyArr(pop[i].code, selected[i].code);
41         pop[i].fitness = selected[i].fitness;
42     }
43 }
```

Figure 23: Function popSelectRanking

15. Function reNewPop ที่เป็นวิธีการเลือกประชากรโดย  
จะนำประชากรที่มีค่า fitness ที่เป็น 0 มาเป็นประชากร  
ใหม่และทำการสุ่มประชากรใหม่ให้ครบกับจำนวนประชากร  
สูงสุด

```

1 void reNewPop(CODE primePop[], CODE pop[], int primeCnt){
2
3     for(int i=0;i<primeCnt;i++){
4         pop[i]=primePop[i];
5     }
6
7     for(int i=primeCnt-1;i<MaxPopSize;i++){
8         randNum(pop[i].code);
9     }
10 }
11 }
```

Figure 24: Function reNewPop

16. Function selPrimePop ที่เป็นการเก็บประชากรที่มีที่มีค่า  
fitness ที่เป็น 0 และทำการเปลี่ยนค่าในชุดประชากรที่ถูก  
เลือกหากมีประชากรที่มีค่า fitness ที่เป็น 0 ซ้ำและเปลี่ยน  
ค่าเป็นประชากรใหม่ที่ถูกสุ่มขึ้นมา

```

1 int selPrimePop(CODE pop[], CODE prime[], CODE Eligibles[], int eCnt){
2     int primeCnt=0;
3     for(int i=0;i<MaxPopSize;i++){
4         if(pop[i].fitness == 0){
5             prime[primeCnt]=pop[i];
6             primeCnt++;
7         }
8     }
9
10    for(int i=0;i<primeCnt;i++){
11        int dupeIndex = isDupe(prime[i].code,Eligibles,eCnt);
12        if(dupeIndex != -1){
13            randNum(Eligibles[dupeIndex].code);
14        }
15    }
16    return primeCnt;
17 }
```

Figure 25: Function selPrimePop

17. Function geneticAlgo ที่เป็น genetic algorithm ที่จะ<sup>ทำการสร้างประชากรที่เป็นรหัสที่ใกล้เคียงกับรหัสมากที่สุด  
ออกมายอดจะมีวิธีดังนี้</sup>

- เริ่มแรกสุดสุ่มชุดประชากรตามจำนวนที่กำหนดไว้
- ในแต่ละรุ่นของประชากรจะใช้ function  
genNewPop ที่มีการ Crossover และ mutation

- ใช้ function fitnessCal ที่คำนวนหาค่า fitness ของ  
ประชากรแต่ละตัว
- ใช้ function selPrimePop ที่จะเลือกประชากรที่มี  
fitness เป็น 0 และหากไม่มีประชากรที่มีค่า fitness  
เป็น 0 ให้จับการ คำนวนในรุ่นนี้และไปคำนวนในรุ่น  
ถัดไป
- ใช้ function addEligible ที่จะเลือกประชากรที่มี  
fitness เป็น 0 ลงในชุดประชากรที่ถูกเลือก
- ใช้ function selection ในแต่ละวิธีในการสร้าง  
ประชากรใหม่ในรุ่นถัดไป

- Function จะหยุดการทำงานเมื่อ คำนวนถึงรุ่น  
สุดท้ายหรือจำนวนชุดประชากรที่ถูกเลือกเกินจำนวน  
ที่ได้กำหนดไว้

```

1 int selMethod = 1;
2 int genericAlgo(CODE Eligibles[],CODE trail[],int trailCnt,int mode){
3
4     CODE population[MaxPopSize];
5
6     for(int i=0;i<MaxPopSize;i++){
7         randNum(population[i].code);
8     }
9
10    int genCnt = 1,eCnt = 0,chEmpty = 0;
11    while(genCnt <= MaxGen/mode && eCnt < MaxPopSize*mode){
12
13        genNewPop(population);
14
15        fitnessCal(population,trail,trailCnt);
16
17        CODE primePopulation[MaxPopSize];
18        int primeCnt=0;
19
20        primeCnt = selPrimePop(population,primePopulation,Eligibles,eCnt);
21
22        if(primeCnt == 0){
23            genCnt++;
24            continue;
25        }
26
27        eCnt = addEligible(Eligibles,population,eCnt);
28
29        switch(selMethod){
30            case 1: reNewPop(primePopulation,population,primeCnt);
31            break;
32            case 2: popSelectRanking(population);
33            break;
34            case 3: popSelectRouletteWheel(population);
35            break;
36            case 4: popSelectTournament(population);
37            break;
38        }
39        genCnt++;
40    }
41
42    return eCnt;
43 }
44 }
```

Figure 26: Function geneticAlgo

18. Function main เริ่มแรกจะทายรหัสที่เป็น {1,1,2,2} แล้วทำการเก็บหมุดสีขาวและคำที่ได้จากการทายมาใช้ใน genetic algorithm หลังจากนั้นจะใช้ประชากรล่าสุดในชุดประชากรที่ถูกเลือกจากการใช้ Genetic algorithm และหากในชุดประชากรที่ถูกเลือกไม่มีประชากรเลยให้ใช้ Genetic algorithm ใหม่ด้วย generation ที่หารด้วย 2 และขนาดของประชากรที่คูณ 2 และในการเลือกคำตอบในการทายหากประชากรล่าสุดถูกทายไปแล้วให้ใช้ประชากรถัดไปแทนจนกว่าประชากรที่เลือกจะไม่เป็นประชากรที่ถูกทายไปแล้ว

```

1 while(1){
2     CODE E[MaxPopSize];
3     int lastCode=0;
4
5     checkCode(guess,code);
6
7     printf("#%d Guess : ",guessCnt+1);
8     printArr(guess,CodeLength);
9     printf("\nBlack: %d White: %d\n\n",blackPeg,whitePeg);
10
11    if(blackPeg == CodeLength){
12        printf("Found it!!! in %d times\n",guessCnt+1);
13        break;
14    }
15
16    copyArr(guessTrial[guessCnt].code,guess);
17    guessTrial[guessCnt].black = blackPeg;
18    guessTrial[guessCnt].white = whitePeg;
19
20    guessCnt++;
21
22    lastCode = generaticAlgo(E,guessTrial,guessCnt,1);
23
24    while(lastCode == 0){
25        lastCode = generaticAlgo(E,guessTrial,guessCnt,2);
26    }
27    lastCode--;
28
29    int dupeIdx = isDupe(E[lastCode-1].code,guessTrial,guessCnt);
30
31    while(dupeIdx != -1 && lastCode != 0){
32        lastCode--;
33        dupeIdx = isDupe(E[lastCode].code,guessTrial,guessCnt);
34    }
35
36    copyArr(guess,E[lastCode].code);
37 }
```

Figure 27: Function main

### 3.3 การดำเนินงาน

- 3.3.1 กำหนดเกม Mastermind ของห้องสมบูรณ์โดยจะทำการทดสอบเกม Mastermind ที่รหัสมีความยาว 4 ตัวและมีสีที่ไม่ซ้ำกัน 6 สี
- 3.3.2 ทดสอบและวัดผลในโปรแกรมโดยจะหาค่าเฉลี่ยในการทายรหัสลับทั้งหมดที่เป็นไปได้ของโค้ดวิธีการแก้ปัญหาเกม Mastermind ด้วยวิธี brute force permutation
- 3.3.3 ทดสอบและวัดผลในโปรแกรมโดยจะหาค่าเฉลี่ยในการทายรหัสลับทั้งหมดที่เป็นไปได้ของโค้ดวิธีการแก้ปัญหาเกม Mastermind ด้วยวิธีที่อ้างอิงจาก Knuth algorithm โดยให้ครั้งแรกเป็นการทายรหัส 1,1,2,2
- 3.3.4 ทดสอบและวัดผลในโปรแกรมโดยจะหาค่าเฉลี่ยในการทายรหัสลับทั้งหมดที่เป็นไปได้ของโค้ดวิธีการแก้ปัญหาเกม Mastermind ด้วยวิธี Genetic algorithm ทั้งหมด 100 รุ่นและแต่ละรุ่นจะมีจำนวนประชากรและขนาดของประชากรที่ถูกเลือกเป็น 60 รวมไปถึงให้ครั้งแรกเป็นการทายรหัส 1,1,2,2 และจะมีการเปลี่ยนแปลงในแต่ละการทดสอบดังนี้

- เปลี่ยนวิธีการและโอกาสในการ Crossover ที่แต่ละโอกาส 0 25 50 75 100 เปอร์เซ็นต์โดยจะควบคุมให้โอกาสการเกิด mutation แบบ Random, Swap และ Inversion เป็น 3 10 และ 2 ตามลำดับอีกทั้งในการ Selection เพื่อเลือกประชากรในรุ่นถัดไปจะใช้วิธี Random
- เปลี่ยนวิธีการและโอกาสในการ Mutation เป็น Random ,Swap และ Inversion ที่แต่ละโอกาส 0 25 50 75 100 เปอร์เซ็นต์ โดยจะสามารถมีการ mutation มากกว่า 1 วิธีที่มีโอกาสแตกต่างกันได้โดยจะควบคุมวิธีการ Crossover เป็นแบบ Uniform ที่โอกาสเป็น 50 เปอร์เซ็นต์ และการ Selection ที่จะใช้วิธี Random

3. เปลี่ยนวิธีการ Selection เป็น Random ,Ranking, Proportional roulette และ Tournament ที่มีขนาด 5 15 25 35 45 55 โดยจะควบคุมให้โอกาสการเกิด mutation แบบ Random, Swap และ Inversion เป็น 3 10 และ 2 ตามลำดับอีกทั้งในการ Crossover จะใช้วิธี Uniform ที่โอกาสเป็น 50 เปอร์เซนต์
4. เปลี่ยน parameter ทั้งหมดเป็นวิธีที่ดีที่สุดและโอกาสการเกิดโดยอ้างอิงจากข้อมูลที่ทำการวิเคราะห์จากการทดสอบข้างต้นเพื่อนำไปเปรียบเทียบกับวิธีการแก้ปัญหาแบบ Bruteforce permutation และ วิธีที่อ้างอิงจาก Knuth algorithm
- 3.3.5 เตรียมสภาพแวดล้อมให้เหมาะสมเพื่อใช้ในการทดสอบระยะเวลาที่ใช้ในการประมวลผลของทั้งสามโปรแกรม**
- ใช้คอมพิวเตอร์เครื่องเดียวเพื่อให้ผลลูกอกมาคลาดเคลื่อนน้อยที่สุด
  - รีสตาร์ทคอมพิวเตอร์ทุกครั้งที่รันโปรแกรม
  - ควบคุมอุณหภูมิเครื่องให้อยู่ที่ประมาณ 55 - 60 องศาเซลเซียส
- 3.3.6 ทดสอบและวัดผลกระทบระยะเวลาที่ใช้ในการประมวลผลโปรแกรมทั้งสาม**
- 3.3.7 นำข้อมูลที่ได้ไปทำการวิเคราะห์และสรุปผลการทดลอง**
- 3.4 แผนการเก็บรวบรวมข้อมูล**  
นำโค้ดที่เตรียมไว้ไปประมวลผลในคอมพิวเตอร์ที่เตรียมไว้ โดยแบ่งออกเป็นสองเคสเพื่อเก็บค่าตัวแปรที่ต่างกัน ดังนี้
- 3.4.1 เป็นการทดลองเพื่อหาค่าเฉลี่ยจำนวนครั้งที่ใช้ในการทายเพื่อแก้ปัญหาเกม Mastermind โดยในแต่ละกรณีจะทำการทดลองทั้งหมด 10 ครั้ง
- 3.4.2 เป็นการทดลองเพื่อหาระยะเวลาที่ใช้ในการประมวลผลเพื่อแก้ปัญหาเกม Mastermind โดยในแต่ละกรณีจะทำการทดลองทั้งหมด 10 ครั้ง
- 3.5 การวิเคราะห์ข้อมูลที่ได้รับจากการดำเนินงาน**
- 3.5.1 การวิเคราะห์และเปรียบเทียบข้อมูลค่าเฉลี่ยที่ใช้ในการทายเกม Mastermind ในแต่ละแบบของโค้ดวิธีการแก้ปัญหาโดยใช้ Genetic algorithm
- 3.5.2 การวิเคราะห์และเปรียบเทียบข้อมูลค่าเฉลี่ยที่ใช้ในการทายเกม Mastermind ระหว่างโค้ดวิธีการแก้ปัญหาโดยใช้ Genetic algorithm ,โค้ดวิธีการแก้ปัญหาโดยใช้ permutation ที่อ้างอิงจาก Knuth algorithm และ โค้ดวิธีการแก้ปัญหาด้วยวิธี bruteforce permutation
- 3.5.3 การวิเคราะห์และเปรียบเทียบข้อมูลระยะเวลาที่ใช้ในการประมวลผลระหว่างโค้ดวิธีการแก้ปัญหาโดยใช้ Genetic algorithm ,โค้ดวิธีการแก้ปัญหาโดยใช้ permutation ที่อ้างอิงจาก Knuth algorithm และ โค้ดวิธีการแก้ปัญหาด้วยวิธี bruteforce permutation

## 4 ผลการดำเนินงาน

### 4.1 เปรียบเทียบประสิทธิภาพใน Genetic Algorithm

#### 4.1.1 เปรียบเทียบจากการ Mutation

Random	Swap	Inverse	Average	Min	Max
0%	0%	0%	4.6289	1	7
0%	0%	25%	4.6427	1	7
0%	0%	50%	4.6605	1	7
0%	0%	75%	4.6134	1	7
0%	0%	100%	4.6327	1	7
0%	25%	0%	4.6983	1	7
0%	25%	25%	4.6728	1	7
0%	25%	50%	4.6242	1	7
0%	25%	75%	4.6227	1	7
0%	25%	100%	4.6497	1	7
0%	50%	0%	4.6690	1	7
0%	50%	25%	4.6582	1	7
0%	50%	50%	4.6651	1	8
0%	50%	75%	4.6404	1	7
0%	50%	100%	4.6443	1	7
0%	75%	0%	4.6682	1	7
0%	75%	25%	4.6242	1	7
0%	75%	50%	4.6481	1	7
0%	75%	75%	4.6713	1	7
0%	75%	100%	4.6759	1	7
0%	100%	0%	4.6435	1	7
0%	100%	25%	4.6644	1	7
0%	100%	50%	4.6896	1	7
0%	100%	75%	4.6458	1	7
0%	100%	100%	4.6474	1	8
25%	0%	0%	4.6289	1	7
25%	0%	25%	4.6713	1	7
25%	0%	50%	4.6173	1	7
25%	0%	75%	4.5826	1	7
25%	0%	100%	4.6196	1	7
25%	25%	0%	4.6211	1	7
25%	25%	25%	4.6196	1	7
25%	25%	50%	4.6381	1	7
25%	25%	75%	4.6119	1	7
25%	25%	100%	4.5841	1	7
25%	50%	0%	4.6273	1	7
25%	50%	25%	4.6157	1	7
25%	50%	50%	4.5918	1	7
25%	50%	75%	4.6150	1	8
25%	50%	100%	4.5988	1	7
25%	75%	0%	4.6613	1	7
25%	75%	25%	4.5965	1	8
25%	75%	50%	4.6657	1	7
25%	75%	75%	4.6096	1	8
25%	75%	100%	4.6404	1	7
25%	100%	0%	4.6705	1	8
25%	100%	25%	4.6119	1	7
25%	100%	50%	4.6673	1	7
25%	100%	75%	4.6026	1	7
25%	100%	100%	4.5525	1	7
50%	0%	0%	4.6690	1	8

Table 1: จำนวนครั้งที่ใช้ในการทายเฉลี่ยเมื่อปรับ Random% เป็น 0 และปรับ Parameter อื่นๆ

Swap%	Inverse%	จำนวนครั้งเฉลี่ย	Min	Max
0	0	4.62285	1	7
0	25	4.64707	1	8
0	50	4.64551	1	8
0	75	4.64205	1	8
0	100	4.6449	1	8
25	0	4.64522	1	8
25	25	4.63671	1	8
25	50	4.64722	1	8
25	75	4.65216	1	8
25	100	4.64237	1	8
50	0	4.64222	1	8
50	25	4.65164	1	8
50	50	4.64507	1	8
50	75	4.64445	1	8
50	100	4.63564	1	8
75	0	4.64775	1	8
75	25	4.64174	1	8
75	50	4.63412	1	8
75	75	4.65224	1	8
75	100	4.64552	1	8
100	0	4.64112	1	8
100	25	4.63332	1	8
100	50	4.643	1	8
100	75	4.64166	1	8
100	100	4.63927	1	8

Figure 28: ตัวอย่างผลการทดสอบการเปรียบเทียบ Mutation

Table 2: จำนวนครั้งที่ใช้ในการทายเฉลี่ยเมื่อปรับ Random% เป็น 25 และปรับ Parameter อื่นๆ

Swap%	Inverse%	จำนวนครั้งเฉลี่ย	Min	Max
0	0	4.61027	1	7
0	25	4.6297	1	8
0	50	4.62701	1	8
0	75	4.61225	1	8
0	100	4.61142	1	8
25	0	4.62661	1	8
25	25	4.60663	1	8
25	50	4.61451	1	8
25	75	4.61869	1	8
25	100	4.61914	1	8
50	0	4.60778	1	8
50	25	4.62746	1	8
50	50	4.61805	1	8
50	75	4.6142	1	8
50	100	4.60749	1	8
75	0	4.61326	1	8
75	25	4.62253	1	8
75	50	4.61852	1	7
75	75	4.61488	1	8
75	100	4.60795	1	8
100	0	4.60579	1	8
100	25	4.61364	1	8
100	50	4.62963	1	8
100	75	4.61151	1	8
100	100	4.61572	1	8

Table 3: จำนวนครั้งที่ใช้ในการทายเฉลี่ยเมื่อปรับ Random% เป็น 50 และปรับ Parameter อื่นๆ

Swap%	Inverse%	จำนวนครั้งเฉลี่ย	Min	Max
0	0	4.62338	1	8
0	25	4.62107	1	8
0	50	4.60269	1	7
0	75	4.61041	1	9
0	100	4.60647	1	8
25	0	4.62092	1	7
25	25	4.61349	1	7
25	50	4.61581	1	7
25	75	4.61837	1	8
25	100	4.61212	1	8
50	0	4.61527	1	8
50	25	4.61273	1	8
50	50	4.61442	1	7
50	75	4.6087	1	8
50	100	4.60959	1	8
75	0	4.60788	1	8
75	25	4.61264	1	8
75	50	4.61226	1	8
75	75	4.6183	1	8
75	100	4.61581	1	8
100	0	4.60509	1	7
100	25	4.62717	1	8
100	50	4.61976	1	8
100	75	4.61906	1	8
100	100	4.61026	1	7

Table 4: จำนวนครั้งที่ใช้ในการทายเฉลี่ยเมื่อปรับ Random% เป็น 75 และปรับ Parameter อื่นๆ

Swap%	Inverse%	จำนวนครั้งเฉลี่ย	Min	Max
0	0	4.61449	1	7
0	25	4.62484	1	8
0	50	4.61205	1	8
0	75	4.6088	1	8
0	100	4.62615	1	8
25	0	4.61134	1	8
25	25	4.6017	1	8
25	50	4.6145	1	8
25	75	4.61034	1	8
25	100	4.61821	1	8
50	0	4.60284	1	8
50	25	4.61335	1	8
50	50	4.61566	1	8
50	75	4.61157	1	8
50	100	4.6155	1	8
75	0	4.6115	1	8
75	25	4.61729	1	8
75	50	4.61791	1	7
75	75	4.60448	1	8
75	100	4.6051	1	8
100	0	4.60787	1	8
100	25	4.62075	1	8
100	50	4.60958	1	8
100	75	4.60603	1	8
100	100	4.62238	1	8

Table 5: จำนวนครั้งที่ใช้ในการทายเฉลี่ยเมื่อปรับ Random% เป็น 100 และปรับ Parameter อื่นๆ

Swap%	Inverse%	จำนวนครั้งเฉลี่ย	Min	Max
0	0	4.62762	1	8
0	25	4.61187	1	8
0	50	4.61002	1	8
0	75	4.61845	1	7
0	100	4.61752	1	8
25	0	4.61332	1	8
25	25	4.60881	1	8
25	50	4.60418	1	8
25	75	4.61419	1	8
25	100	4.61072	1	8
50	0	4.61311	1	8
50	25	4.62006	1	8
50	50	4.61303	1	8
50	75	4.62345	1	8
50	100	4.60601	1	8
75	0	4.60679	1	8
75	25	4.62469	1	8
75	50	4.61606	1	8
75	75	4.61197	1	9
75	100	4.619	1	7
100	0	4.60726	1	8
100	25	4.61066	1	8
100	50	4.60717	1	7
100	75	4.61157	1	8
100	100	4.61597	1	8

จากตารางผลการทดลอง จำนวนครั้งที่ใช้ในการพยายามเพื่อแก้ไขปัญหาเกม Mastermind ด้วยวิธี Genetic algorithm เมื่อมีการเปลี่ยนค่า parameter การ mutation ในแบบต่างๆ จะสามารถวิเคราะห์ได้ดังนี้

ในกรณีที่โอกาสการเกิด Random mutation เป็น 75 %, Swap mutation 25 % และ Inversion mutation เป็น 25 % จะมีค่าเฉลี่ยที่จำนวนครั้งที่ใช้ในการพยายามที่มากที่สุดเป็น 7 ครั้ง ซึ่งเป็นค่าที่น้อยที่สุดจากการทดลองทั้งหมด

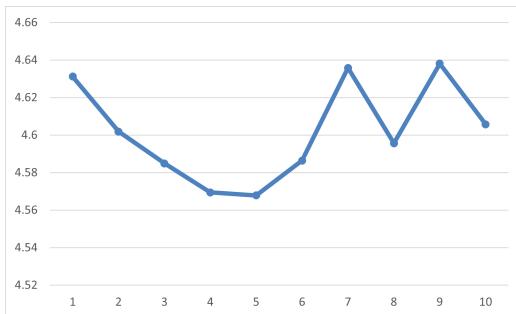


Figure 29: กราฟแสดงผลการทดสอบทั้งหมด 10 ครั้ง ในกรณีที่โอกาสการเกิด Random mutation เป็น 75 %, Swap mutation 25 % และ Inversion mutation เป็น 25 %

แต่ทั้งนี้ทั้งนั้นก็อาจจะไม่ใช่ค่า parameter ที่ดีที่สุด เนื่องจากเมื่อลองพิจารณาในการทดลองแต่ละครั้งจะพบว่า ค่าเฉลี่ยที่ได้ในแต่ละครั้งจะไม่มีความใกล้เคียงจากกราฟ และในที่ค่า parameter นั้น ๆ ก็มีค่าเฉลี่ยที่ใกล้เคียงไม่ต่างกันมากนัก ซึ่งอาจจะเป็น เพราะในวิธีการ genetic algorithm ที่มีการเขียนขั้นมา มีการสุ่มค่าขั้นมาในหลาย ๆ ขั้นตอนไม่ว่าจะเป็นในการสร้างประชากรรุ่นแรก การ crossover mutation หรือแม้กระทั่ง selection ที่ยังมีโอกาสในการเกิดหรือการสุ่มค่าขั้นมา

4.6335	1	7
4.5949	1	8
4.6665	1	7
4.5792	1	7
4.5779	1	7
4.6611	1	7
4.6294	1	7
4.6343	1	7
4.6611	1	7
4.6566	1	7
4.6273	1	7
4.5965	1	7
4.6628	1	7
4.6296	1	8
4.6619	1	7
4.6696	1	7
4.6196	1	7
4.6543	1	7
4.6289	1	7
4.6250	1	7
4.6327	1	7
4.5779	1	8
4.5764	1	7
4.6235	1	7
4.6420	1	7
4.5988	1	7

```

Avg 100 : 4.6162
Process returned 0 (0x0)   execution time : 483.718 s
Press any key to continue.

```

Figure 30: ภาพผลลัพธ์การทดสอบเฉลี่ย 100 ครั้ง ในกรณีที่โอกาสการเกิด Random mutation เป็น 75 %, Swap mutation 25 % และ Inversion mutation เป็น 25 % เมื่อทำการทดสอบเข่นเดิมที่โอกาสการเกิด Random mutation เป็น 75 %, Swap mutation 25 % และ Inversion mutation เป็น 25 % ถ้าครั้งแต่ทดสอบเพิ่มเป็น 100 ครั้งจะพบว่าค่าเฉลี่ยจะมีค่าไม่เหมือนเดิมจะได้เป็น 4.6162 และมีจำนวนครั้งที่ใช้ในการพยายามที่มากที่สุดเป็น 8 ครั้ง ซึ่งจะไม่ใกล้เคียงกับการทดสอบในตอนแรกทั้งในส่วนของค่าเฉลี่ยในการพยายามและจำนวนครั้งการพยายามที่ใช้มากที่สุด

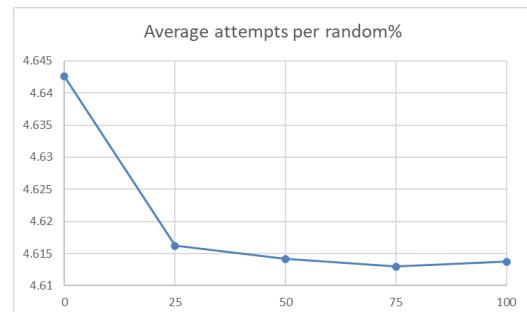


Figure 31: กราฟแสดงค่าเฉลี่ยของจำนวนครั้งในการพยายามของแต่ละ random % อย่างไรก็ตาม หากปรับเปลี่ยนตัวการ Mutation แบบ Random ไว้เป็น 0 % จะทำให้จำนวนการพยายามเฉลี่ยมีค่าเพิ่มขึ้นในช่วง 4.64 ซึ่งต่างจากการปรับเปลี่ยนตัวเป็นรูปแบบอื่นที่จะทำให้ได้ค่าเฉลี่ยอยู่ในช่วง 4.61 ถึง 4.62

#### 4.1.2 เปรียบเทียบจากการ Crossover

```
C:\Users\User\Downloads> python crossover.py
Using single point crossover
Percent    Average      Min      Max
0%        4.6057       1        7
25%       4.5756       1        7
50%       4.6165       1        7
75%       4.6373       1        8
100%      4.5748       1        7

Using Double point crossover
Percent    Average      Min      Max
0%        4.5957       1        7
25%       4.6049       1        7
50%       4.5895       1        7
75%       4.6188       1        7
100%      4.6366       1        7

Using Uniform point crossover
Percent    Average      Min      Max
0%        4.6057       1        7
25%       4.5903       1        7
50%       4.6258       1        7
75%       4.5980       1        7
100%      4.6312       1        7

-----
Process exited after 127.4 seconds with return value 0
Press any key to continue . . .

```

Figure 32: ตัวอย่างผลการทดสอบการเปรียบเทียบ Crossover

Table 6: ผลการทดลองเปรียบเทียบการทายเฉลี่ยจากการ Crossover และเบอร์เซ็นต์ที่ใช้

Percent (%)	Single	Double	Uniform
0	4.61419	4.60139	4.61327
25	4.61853	4.62268	4.60981
50	4.60586	4.60804	4.60201
75	4.61936	4.61735	4.60519
100	4.60101	4.61326	4.60633

Table 7: ผลการทดลองเปรียบเทียบการทายมากที่สุดจากการ Crossover และเบอร์เซ็นต์ที่ใช้

Percent (%)	Single	Double	Uniform
0	8	7	8
25	8	7	8
50	8	8	8
75	8	8	8
100	8	8	8

เมื่อพิจารณาค่าเฉลี่ยที่ดีที่สุดในแต่ละแบบของการ crossover จะได้ว่าการ crossover แบบ single point ที่โอกาสการเกิด 100 % จะได้จำนวนเฉลี่ยที่ใช้ในการทายเป็น 4.6101 จำนวนครั้งที่ทายมากสุดคือ 8 แบบ double point ที่โอกาสการเกิด 0 % จะได้จำนวนเฉลี่ยที่ใช้ในการทายเป็น 4.60139 จำนวนครั้งที่ทายมากสุดคือ 7 และแบบสุดท้ายแบบ uniform ที่โอกาสการเกิด 50 % จะได้จำนวนเฉลี่ยที่ใช้ในการทายเป็น 4.60201 จำนวนครั้งที่ทายมากสุดคือ 8 ซึ่งสามารถสรุปได้ว่าวิธี crossover ที่ดีที่สุดในการทดลองนี้คือแบบ single point crossover ที่โอกาสการเกิด

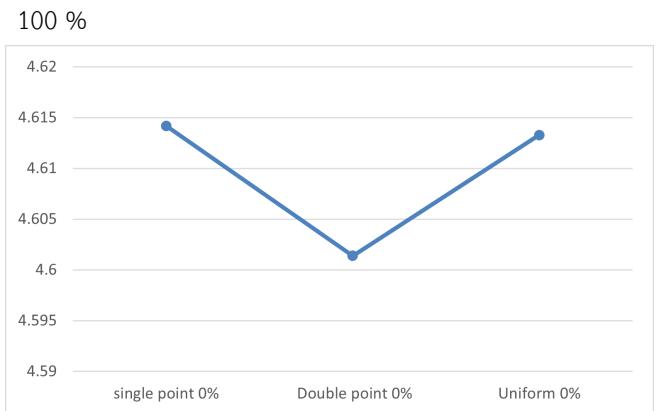


Figure 33: กราฟแสดงค่าเฉลี่ยของจำนวนครั้งในการทายของแต่ละวิธีการ Crossover ที่โอกาสในการเกิดเป็น 0 %

เมื่อพิจารณาที่โอกาสการเกิด 0 % ของ single point และ uniform ของ crossover ที่มีค่าเฉลี่ยเป็น 4.61419 และ 4.61327 ซึ่งมีความใกล้เคียงกันแต่เมื่อพิจารณาค่าเฉลี่ยของ crossover แบบ double point ที่มีค่าเฉลี่ยเป็น 4.60139 ซึ่งมีความต่างกับสองค่าก่อนหน้าเป็นอย่างมากและอาจจะสรุปได้ว่าในการทดลองนี้วิธีที่ดีที่สุดอาจจะไม่ใช่ single point crossover ที่โอกาสการเกิด 100 % อย่างที่ได้สรุปข้างต้นซึ่งเหตุผลก็เหมือนกับที่กล่าวไว้ในกรณีของ mutation เลยก็คือมีการสูญค่ามากเกินไปทำให้ค่าเฉลี่ยในการทายแต่ละรอบไม่เกิดความเสถียร

#### 4.1.3 เปรียบเทียบจากการ Selection

```
C:\Users\User\Downloads> x + v
Using Random selection
    Average   Min   Max
    4.5849   1     7

Using Ranking selection
    Average   Min   Max
    4.6258   1     7

Using Roulette wheel selection
    Average   Min   Max
    4.6065   1     7

Using Tournament selection
Tour size      Average   Min   Max
5              4.7569   1     8
15             4.7137   1     8
25             4.7392   1     8
35             4.7446   1     9
45             4.7153   1     9
55             4.7508   1     8

Process exited after 285.6 seconds with return value 0
Press any key to continue . . .

```

Figure 34: ตัวอย่างผลการทดสอบการเปรียบเทียบ Selection

Table 8: ผลการทดลองเปรียบเทียบการทายเฉลี่ยจากวิธีการ Selection ที่ใช้ (ไม่รวม Tournament)

วิธีการ	จำนวนครั้งเฉลี่ย	จำนวนครั้งมากที่สุด
Random	4.61907	8
Ranking	4.62864	8
Roulette wheel	4.60772	8

Table 9: ผลการทดลองเปรียบเทียบการทายเฉลี่ยจากวิธีการ Selection แบบ Tournament โดยเปรียบเทียบ Toursize

Toursize	จำนวนครั้งเฉลี่ย	จำนวนครั้งมากที่สุด
5	4.71597	9
15	4.72577	9
25	4.71682	11
35	4.71259	10
45	4.71012	9
55	4.70024	10

จากตารางผลการทดลอง จำนวนครั้งที่ใช้ในการทายเพื่อแก้ไขปัญหาเกม Mastermind ด้วยวิธี Genetic algorithm เมื่อมีการเปลี่ยนค่า parameter การ selection ในแบบต่างๆ จะสามารถวิเคราะห์ได้ดังนี้

เมื่อพิจารณาค่าเฉลี่ยที่ดีที่สุดในแต่ละแบบของการ selection จะได้ว่าการ selection แบบ Random จะได้จำนวนเฉลี่ยที่ใช้ในการทายเป็น 4.61907 จำนวนครั้งที่ทายมากสุดคือ 8 แบบ Ranking จะได้จำนวนเฉลี่ยที่ใช้ในการทายเป็น 4.62864 จำนวนครั้งที่ทายมากสุดคือ 8 และแบบสุดท้ายแบบ Tournament ที่มีขนาด 55 จะได้จำนวนเฉลี่ยที่ใช้ในการทายเป็น 4.70024 จำนวนครั้งที่ทายมากสุดคือ 10 ซึ่งสามารถสรุปได้ว่าวิธี selection ที่ดีที่สุดในการทดลองนี้คือแบบ Roulette wheel selection

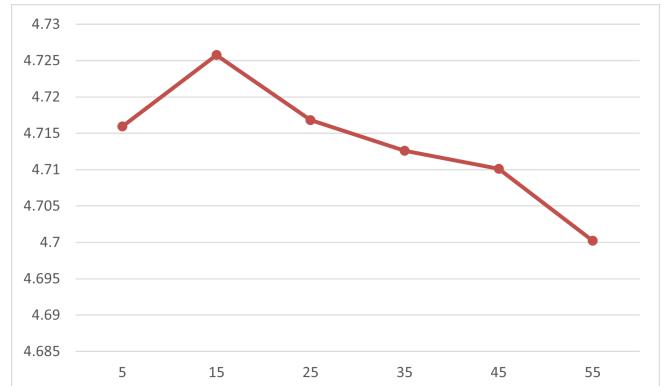


Figure 35: กราฟแสดงค่าเฉลี่ยของจำนวนครั้งในการทายของวิธีการ Tournament Selection โดยเปรียบเทียบกับ Toursize ที่ใช้

เมื่อพิจารณาผลการทดลองการ Tournament selection ในแต่ละขนาดของ Tournament และนำมาทำเป็นกราฟจะพบว่า เมื่อขนาดของ Tournament เพิ่มขึ้นจำนวนเฉลี่ยที่ใช้ในการทายมีแนวโน้มที่จะลดลง

จากการทดลองทั้งหมดข้างต้นสามารถสรุปค่า parameter ที่ดีที่สุดของทั้งการ Mutation ,Crossover และ Selection โดยจะทดสอบหากค่าเฉลี่ยจำนวนครั้งที่ใช้ในการทายเพื่อแก้ไขปัญหาเกม Mastermind และระยะเวลาที่ใช้ในการประมวลผล ซึ่งจะใช้ Parameter ดังนี้

- Single point crossover ที่โอกาสการเกิด 100
- Swap mutation, Random mutation และ Inversion Mutation ที่โอกาสการเกิดเป็น 25% 75% 25% ตามลำดับ
- Proportional Roulette Wheel Selection

## 4.2 เปรียบเทียบกับวิธี Bruteforce permutation และ Knuth algorithm

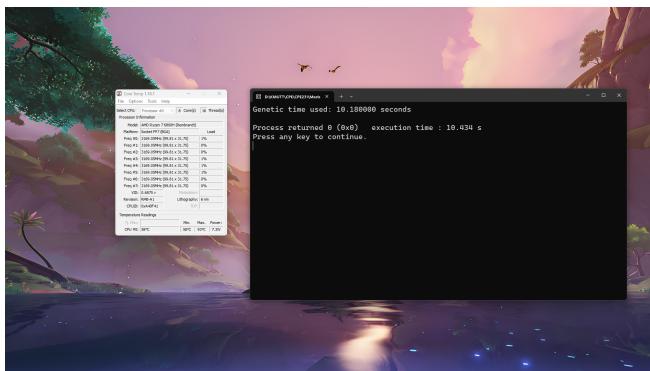


Figure 36: ภาพตัวอย่างผลการทดสอบระยะเวลาการประมวลผล genetic algorithm

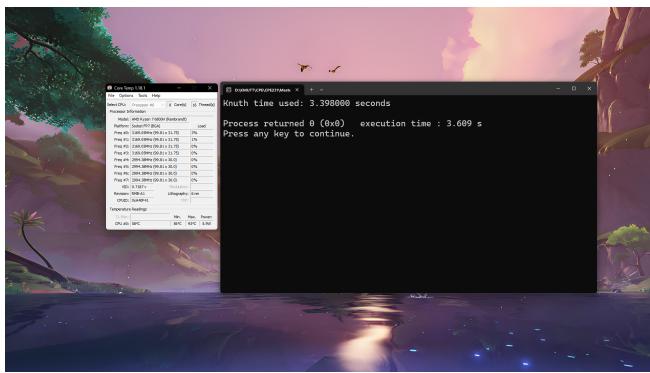


Figure 37: ภาพตัวอย่างผลการทดสอบระยะเวลาการประมวลผล knuth algorithm

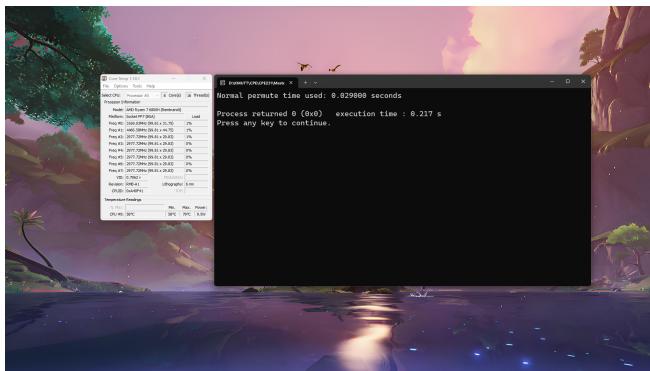


Figure 38: ภาพตัวอย่างผลการทดสอบระยะเวลาการประมวลผล bruteforce permutation

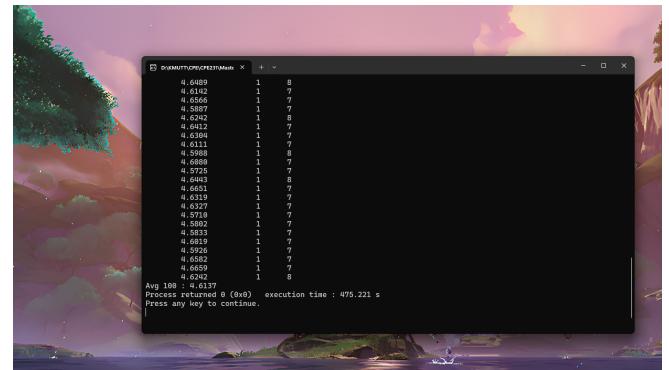


Figure 39: ภาพตัวอย่างผลการทดสอบค่าเฉลี่ยจำนวนครั้งที่ใช้ในการหาทายของ genetic algorithm

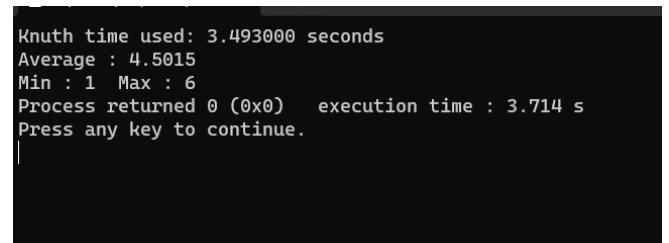


Figure 40: ภาพตัวอย่างผลการทดสอบค่าเฉลี่ยจำนวนครั้งที่ใช้ในการหาทายของ knuth algorithm

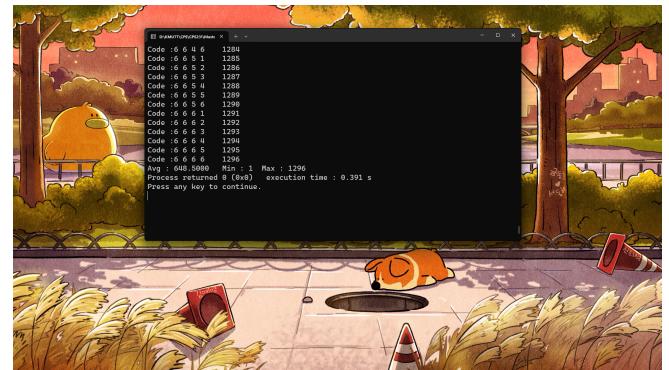


Figure 41: ภาพตัวอย่างผลการทดสอบค่าเฉลี่ยจำนวนครั้งที่ใช้ในการหาทายของ bruteforce permutation

Table 10: ผลการทดลองเปรียบเทียบการทายเฉลี่ยและระยะเวลาระหว่าง Genetic Algorithm, Knuth Algorithm และ Bruteforce Algorithm

วิธี	จำนวนครั้งเฉลี่ย	เวลาที่ใช้เฉลี่ย	Max
Genetic Algorithm	4.6137	10.9277	8
Knuth Algorithm	4.5015	3.3454	6
Bruteforce Algorithm	648.5	0.028	1296

จากตารางผลการทดลอง ระยะเวลาที่ใช้ในการประมาณผลการทายเพื่อแก้ไขปัญหาเกม Mastermind ด้วยวิธี Bruteforce permutation, วิธี Genetic algorithm, และ Knuth algorithm ที่มีค่าเฉลี่ยจำนวนครั้งที่ใช้ในการทายอยู่ที่ 0.028, 10.9277 และ 3.3454 วินาทีตามลำดับซึ่งสามารถสรุปได้ว่า Bruteforce permutation มีระยะเวลาประมาณน้อยที่สุดเมื่อเทียบกับทั้ง 2 วิธีเนื่องจากการทำงานมีเพียงแค่สร้างรหัสที่เป็นไปได้ทั้งหมด และพิจารณาในทุกรหัสที่เป็นไปได้ทั้งหมดว่าเป็นรหัสลับหรือไม่ ในขณะที่ Genetic algorithm หรือ Knuth algorithm ที่จะมีการทำงานที่ซับซ้อนกว่าไม่ว่าจะเป็นการคำนวณค่า fitness ของ genetic algorithm หรือ หารหัสที่สามารถตัดรหัสที่เป็นไปได้ออกให้ได้มากที่สุดของ Knuth algorithm

จากตารางผลการทดลอง จำนวนครั้งที่ใช้ในการทายเพื่อแก้ไขปัญหาเกม Mastermind ด้วยวิธี Bruteforce permutation, วิธี Genetic algorithm, และ Knuth algorithm ที่มีค่าเฉลี่ยจำนวนครั้งที่ใช้ในการทายอยู่ที่ 64.85, 4.6137 และ 4.5015 ครั้ง ตามลำดับ อีกทั้งเมื่อพิจารณาจำนวนครั้งมากที่สุดที่เป็น 1296, 8 และ 6 ครั้ง ซึ่งสามารถสรุปได้ว่า Knuth algorithm มีจำนวนการครั้งในการทายน้อยที่สุด ตามด้วย Genetic algorithm และ Bruteforce permutation ที่เลือกทายในทุกรหัสที่เป็นไปได้ทั้งหมดจนกว่าจะพบรหัสที่ถูกต้องต่างกับ Genetic algorithm หรือ Knuth algorithm ที่มีการพิจารณาจากผลลัพธ์ที่ได้จากการทายในครั้งก่อนๆ

และจากข้อมูลในการทดสอบทั้งสามวิธีในการแก้ปัญหาเกม Mastermind ยังจะพบอีกว่าในวิธีการแก้ปัญหาแบบ Bruteforce permutation และ Knuth algorithm จะใช้จำนวนครั้งในการทายเท่าเดิมถ้ารหัสลับเป็นชุดเดิมไม่เหมือนกับวิธี Genetic algorithm ที่มีการสุ่มค่าขึ้นมาในหลายชั้นตอนที่ทำให้หารหัสลับเป็นรหัสเดียวกันจำนวนครั้งในการทายก็อาจจะไม่เท่ากันเสมอ

## 5 สรุปผลการทดลอง

จากการทดลองสร้างโค้ดเพื่อแก้ไขปัญหาเกม Mastermind ด้วย Genetic algorithm ค้นพบว่าการปรับ Parameter ต่าง ๆ จะส่งผลต่อประสิทธิภาพของการแก้ปัญหาการทดลองได้สำรวจและทดสอบค่า Parameter ต่าง ๆ เช่น Selection ,Crossover, Mutation เพื่อหาวิธีที่ดีที่สุดในการแก้ปัญหาในเกม Mastermind โดยมีการทดสอบการทำงานของอัลกอริทึมจำนวนหลายครั้งเพื่อทดสอบความสามารถในการค้นหาสำหรับการแก้ปัญหาเกม Mastermind จนได้ผลลัพธ์ที่ดีเด่นดังให้เห็นว่าการปรับ

Parameter สามารถส่งผลต่อประสิทธิภาพได้ดีที่สุดคือ

- จำนวนประชากรในแต่ละ generation เป็น 60
- จำนวนกลุ่มประชากรที่ถูกเลือกมีได้สูงสุด 60
- จำนวน Generation ทั้งหมดในการใช้ Genetic algorithm 100
- ใช้ Single point crossover ที่โอกาสการเกิด 100%
- ใช้ Swap mutation, Random mutation และ Inversion Mutation ที่โอกาสการเกิดเป็น 25% 75% 25% ตามลำดับ
- ใช้ Proportional Roulette Wheel Selection เป็น Parent selection method
- Termination criteria คือเมื่อ กลุ่มประชากรที่ถูกเลือกเต็ม หรือ Generation สูงสุด หรือ พบรูปแบบที่ถูกต้อง

และเมื่อทำการเปรียบเทียบ Genetic algorithm กับวิธีการแก้ปัญหานี้ในการแก้ปัญหาเกม Mastermind คือวิธีการ Bruteforce permutation และ Knuth algorithm ก็จะได้ผลลัพธ์ที่ว่า Genetic algorithm ที่ parameter ข้างต้นจะมีประสิทธิภาพในการแก้ปัญหาได้ดีกว่าไม่ว่าจะเป็นด้านจำนวนครั้งที่ใช้ในการ tahy หรือระยะเวลาที่ใช้ในการประมวลผล Genetic algorithm จะมีประสิทธิภาพดีกว่า Bruteforce permutation แต่น้อยกว่า Knuth algorithm อย่างไรก็ตาม จากผลการทดสอบที่ไม่เสียเวลาระหว่าง parameter ดังกล่าวข้างต้นก็อาจจะไม่ใช่ค่า parameter ที่ดีที่สุดสำหรับ Genetic algorithm ในการแก้ปัญหาเกม Mastermind ที่มีการสุ่มในหลายชั้นตอนซึ่ง การทดลองนี้ได้เป็นการเรียนรู้ที่มีคุณค่าในการเข้าใจการทำงานของ Genetic algorithm ในการแก้ปัญหา Mastermind และเป็นการสะท้อนถึงความสำคัญของการปรับแต่ง parameter เพื่อให้ได้ผลลัพธ์ที่เหมาะสมในการแก้ปัญหาในโลกจริงได้อย่างมีประสิทธิภาพ

## References

- [BGL09] Lotte Berghman, Dries Goossens, and Roel Leus. Efficient solutions for mastermind using genetic algorithms. *Computers Operations Research*, 36(6):1880–1885, 2009.
- [CHU23] CHULATUTOR. สรุปเนื้อหา ความน่าจะเป็น คืออะไร พร้อมตัวอย่างข้อสอบ, Nov 2023.
- [mag14] magisterrex. Mastermind game rules, Jul 2014.
- [อ18] K. อรุณ. บทที่ 2 ความน่าจะเป็น (probability), 2018.