Lab 4 - A Mathematical Card Trick

Luciano Giannini, Ryan Pepin, Mike Emmett
LC 207: Mathematical Experiments in Computer Science
(Dated 11/13/2019)

Contents

Introduction

In this lab we were introduced to an ordinary magic trick that turned out to be more mathematics than magic. This trick allows for a person to pick five random cards, and then the “magician” ,with the help of his assistant, picks one card to give back to the player and then guesses what card it is. This trick illustrates the use of combinatorics and permutations of different combinations of numbers in relations to the randomly selected cards. In this lab we explored how this card trick is actually performed, and different ways it can be implemented.

The idea of permutations is used by the “helper” to communicate the missing card’s informations to the “magician” without anyone knowing. Once the card is given back to the player the “helper” orders the remaining cards in a way that the “magician” can decrypt the hidden cards suite and value. Combinatorics was implemented to determine what the permutation for a desired card should be. We used these two ideas to determine how to play the game with 4 cards and 6 cards, and how these new versions change the permutations and combinatorics principles used. When exploring how to use different amounts of cards for the trick, we also explored how the size of the deck plays a role in the trick.

Program 1: Encoding and Decoding Permutations

Objective: Write a program that helps the magician and accomplice encode and decode *n*! Permutations.

The program begins by asking the user if they want to encode or decode.

Selecting encode will ask the user for the number of elements (n), and which permutation(m). It will run the perm method seen in FIG 1.2 and print out the correct permutation.

FIG 1.1: Encode Method

This method just asks the user for the number of elements and which permutation. Then passes both to the perm method

```
public void encode()
{
    Scanner input = new Scanner(System.in);
    System.out.println("How many elements do you want?");
    int n = input.nextInt();
    System.out.println("Which permutation?");
    int m = input.nextInt();
    int p = m;
    perm(n,m);

}//end encode method
```

```
public void perm(int n, int p)
{

    // initialize permutation
    int[] a = new int[n];
    for (int i = 0; i < n; i++)
        a[i] = i+1;

    // print permutations
    for(int i = 1; i < p; i++)
    {
        nextPerm(a);
    }
    print(a);
}
```

FIG 1.2: Perm Method

This creates an array of size n and puts in the first permutation of n in. For example, when n = 4, the array will have 1,2,3,4 in it. This array will be passed in the nextPerm() method to get the next permutation. This will continue until the loop is done.

```
Do you want to (D)ecode or (E)ncode?
E
How many elements do you want?
4
Which permutation?
8
2 1 4 3
```

FIG 1.3: Output of Encode Method

Selecting decode will ask the user for the number of elements(n), and the permutation. It will run the nextPerm method seen in FIG 1.7 and tell the user which permutation the input is.

```
public void decode()
{
   Scanner input = new Scanner(System.in);
   System.out.println("How many elements are there?");
   int n = input.nextInt();
   System.out.println("Enter the permutation:");
   String permu = input.nextLine();
   permu = input.nextLine();
   int [] permutation = new int[n];
   int [] permutationdemo = new int[n];
   for(int i = 0; i <n; i++)
      permutation[i] = Character.getNumericValue(permu.charAt(i));
   for(int i = 0; i <n; i++)
      permutationdemo[i] = i+1;
   int count = 1;
   while(!isEqual(permutation,permutationdemo))
   {
      permutationdemo = nextPerm(permutationdemo);

      count++;
   }
   System.out.println("Permutation: " + permu + " is the "+ count+ "th permutation of "+n+" elements");


}
```

FIG 1.4: Decode Method

This method creates two arrays, one that holds in input from the user and another one that starts from the first permutations of n elements. The while loop is executed until the two arrays holding the permutation from the user and the one that is traversing through the next permutation are equal. Also, we are keeping track of the number of times it does through the while loop so that the user knows what number permutation it is.

```
public boolean isEqual(int[] array1, int[]array2)
{
   for(int i = 0; i < array1.length;i++)
   {
      if(array1[i]!=array2[i])
         return false;


   }
   return true;
```

FIG 1.5: isEqual Method

```
Do you want to (D)ecode or (E)ncode?
D
How many elements are there?
4
Enter the permutation:
2143
Permutation: 2143 is the 8th permutation of 4 elements
```

FIG 1.6: Output of Decode Method

The encode and decode methods depend on a method called nextPerm. We used the pseudocode from https://codesciencelove.wordpress.com/2013/07/15/finding-the-nth-lexicographic-permutation-of-0123456789/ which was given to us.

- First, the method finds the largest index k in the array such that a[k] < a[k+1].
  - If such an index cannot be found, the array cannot be permuted lexicographically.
- Next, we need to find the largest index i in the array such that a[k] < a[i].
  - This value must exist if there is a value k because of a[k] < a[k+1].
- The program will then swap the values at a[k] and a[i].
- Lastly, the program reverses all values in the array from index k+1 to the end.
- The method returns a permuted version of the array it is given as a parameter.

```
public int[] nextPerm(int[] permPassed)
{
   int k = 0;
   int length = permPassed.length;

   for (k = length-2; k >= 0; k--)
      if (permPassed[k] < permPassed[k+1])
         break;
   if (k == -1)
      return null;

   int j = length-1;
   while (permPassed[k] >permPassed[j])
      j--;
   swap(permPassed, j, k);

   for (int r = length-1, s = k+1; r > s; r--, s++)
      swap(permPassed, r, s);

   return permPassed;
}//end method
```

FIG 1.7: nextPerm Method

Program 2: The Trick With 28 Cards

Objective: To write a program that simulates the simple card trick using 28 cards. Must be able to perform Ralph's job of setting up the cards in the permutation of the lowest card of the 5. Has to also perform Shai's job and be able to recover the hidden card when presented with the permutation.

The program begins by asking the user which job it must do in the trick.

For (C)onstructing the trick, we first create the random hand of numbers 1-28. The lowest card is taken to be stored in lo.

```
public static void construct()
{
int [] a = new int [28];
for (int i = 1;i<a.length;i++)
  a[i] = i;
int [] hand = hand(a,5);
int lo = getCard(hand);
//System.out.println(lo);
int [] left = reorder(hand,lo);
LexicographicPermutation1 trick = new Lexic
int [] key = trick.encode(lo);
int [] order = order(key,left);
for (int i = 0;i<order.length;i++)
  System.out.print(order[i]+" ");
}
```

```
public static int getCard(int [] a)
{
  int lo = 28;
  for (int i=0;i<a.length;i++)
  {
    if (a[i]<lo)
      lo = a[i];
  }
  return lo;
}
```

We then use the LexicographicPermutation1 class to find and return the permutation of 1234 corresponding to the value from lo. This permutation is taken and used to reorder the remaining 4 cards in the correct order so they can be presented to the user. The user can then perform the trick and guess the remaining card.

LexicographicPermutation1

```
public int [] perm(int n, int p, int [] a)
{
int [] card = new int [p];
   // initialize permutation
   for (int i = 0; i < n; i++)
      a[i] = i+1;

   if (p == 1)
     return a;
   for(int i = 1; i < p; i++)
   {
      card = nextPerm(a);
      //print(a);
   }
   return card;
}
public static void print(int[] a) {
   for (int i = 0; i < a.length; i++)
      System.out.print(a[i]+" ");
   System.out.println();
}
public int [] encode(int lo)
{
   int n = 4;
   int [] b = new int [n];
   int [] p = perm(n,lo,b);

return p;
}//end encode method
```

Ordering the Cards

```
public static int[] order(int [] key, int [] left)
{
  int [] max = new int[4];
  int [] order = new int[4];
  for (int k = 3;k>=0;k--)
  {
    int m = 0;
  for (int i = 0;i<4;i++)
  {
    if (left[i]>max[k])
    {
      max[k] = left[i];
      m = i;
    }
  }
  left[m]=-1;
  }
```

The (R)ecover option performs Shai's job. The user must input 4 numbers in an order and judging by that order, the program returns the value of the hidden card (lo).

recover()

```
public static void recover()
{
  Scanner input = new Scanner(System.in);
  int [] perm = new int [4];
  System.out.println("Enter four cards to find the hidden card!")
  perm[0] = input.nextInt();
  perm[1] = input.nextInt();
  perm[2] = input.nextInt();
  perm[3] = input.nextInt();
  int [] order = permOrder(perm);
  LexicographicPermutation1 trick = new LexicographicPermutation1
  int permutation = trick.decode(4,order);
  System.out.println(permutation);
}
```

permOrder(int [])

```
public static int [] permOrder(int [] a)
{
  int [] count = new int [4];
  for (int i = 0;i<count.length;i++)
  {
    count[i] = 1;
    for (int j = 0;j<a.length;j++)
    {
      if (a[i]>a[j])
        count[i]++;
    }
  }
  return count;
}
```

The permOrder(int []) method takes the hand of 4 cards and creates an array of 1, 2, 3, and 4 with 4 in the location of the largest card and 1 in the location of the smallest card in the hand. Then, using the LexicographicPermutation1 class, we find the permutation of 1234 that corresponds to that found in permOrder(int []) and return that, revealing the hidden card.

LexicographicPermutation1 decode()

```
public int decode(int size, int [] a)
{
   int n = size;

   int [] permutation = a;
   int [] permutationdemo = new int[n];
'     for(int i = 0; i <n; i++)
'        permutation[i] = Character.getNumericValue
   for(int i = 0; i <n; i++)
      permutationdemo[i] = i+1;
   int count = 0;
   while(!isEqual(permutation,permutationdemo))
   {
      permutationdemo = nextPerm(permutationdemo);

      count++;
   }
   //System.out.println("Permutation: " + permu +
 return count;

}
```

Extending Our Working Strategy to Other Cases

Lower Bounds on the Number of Cards Possible – Based on Our Working Strategy

1. If we do the trick by letting the person choose 4 cards and the accomplice shows an ordered subset of three of them, then what is the maximum number of cards for which our working strategy works? Note: You should think of the deck as having three suits.
    - The maximum number of cards for which the strategy works is 15.
2. Same question for 3 cards? 2 cards?
    - Three cards would have a maximum of 6 cards
    - Two cards would have a maximum of 3 cards
3. Let's go in the other direction. If we let the person choose 6 cards and the accomplice shows an ordered subset of 5, then how large a deck can our working strategy handle?
    - If 6 cards were chosen then the maximum number of cards would be 245.
4. Generalize your results above for the case where n is the number of cards chosen, and the accomplice chooses an ordered subset of n-1.
    - If n is the number of cards chosen, then the maximum size of the deck is 2(n-1)!+(n-1).

Extending Our Combinatorial Arguments to Other Cases

Upper Bounds on the Number of Cards – Based on a Combinatorial Idea

5. The upper bound for $n = 5$ is 124 cards. What are the upper bounds for $n = 2, 3, 4$?

    - n = 2 is 3 cards
    - n = 3 is 8 cards
    - n = 4 is 24 cards

6. Write a formula for the upper bound U in terms of n.

    - U = n! + (n-1)

7. Write formulas for the number of ways to choose n cards from U, and for the number of ways to order n-1 cards from U. Prove that these are equal.

    - Choose n cards to form U:
        - ❑ C(U,n) - the total number of ways to choose n cards from U is U choose n
    - Ways to order n-1 cards from U:
        - ❑ N!
    - Due to a combinatorial identity, these are equal. Because of a set of n elements, each sub-set of size U produces a complementary sub-set of size n-U, obtained by taking the n-U elements not in sub-set size U: sub-set n-U= sub-set U. Since these sections are in one-to-one correspondence, they must be equal in number.

Counting the Huge Number of Possible Strategies

Now let's analyze the possibility of other strategies. In practice, a strategy needs to be easily decodable, but theoretically, a strategy is no more or less than a list of what the accomplice should do in every possible situation.

8. Assume we are using the largest size deck of cards possible. Let *n* be the number of cards chosen, where the accomplice chooses an ordered subset of *n*-1. Calculate formulas in terms of *n*, for:

- The number of ways the contestant can choose his cards.
    - The number of ways the contestant can choose his cards is n! + C(n-1,n)
- The number of different permutations available to the accomplice for each of the contestant's choices. (For $n = 5$, there are 120 possibilities).
    - The number of different permutations is n!
- The number of different possible *tables*. A *table* is a list of permutations (each length *n*-1), one permutation for each of the possible sets of *n* cards in the contestant's hand. (For $n = 5$, this equals $120^{225150024}$).
    - The number of different possible tables is $n!^{C(n!+n-1,n)}$

Program/Exercise 3

a. Write a program to fill in the table of 20 (6 choose 3) slots (123, 124, 125, ..., 456) for six numbers *using our working strategy*. If you prefer, you can do this by hand.

Working Strategy:

A-B -->	B-A -->	B-C -->	C-B -->	A-C -->	C-A

123 12	124 21	125 52	126 61	134 14	135 15	136 16	145 54	146 41	156 16
234 23	235 32	236 63	245 25	246 26	256 65	345 34	346 43	356 36	456 45

b. If we try to use our working strategy for a deck of 7 cards, verify that the strategy is unsuccessful. (Hint: it fails after 123, 124, 125). Fill in the table for 7 cards successfully yourself by hand, using any strategy that works.

Working Strategy:

B-A -->	A-B -->	C-A -->	A-C -->	B-C -->	C-B

We used Program 4/5 to fill in the table.

123 21	124 12	125 51	126 61	127 71
134 31	135 13	136 16	137 17	145 41
146 14	147 47	156 15	157 57	167 67
234 32	235 23	236 62	237 72	245 42
246 24	247 27	256 52	257 25	267 26
345 43	346 34	347 73	356 53	357 35
367 63	456 54	457 45	467 64	567 65

Program 4/5: Generate a Table for 8 Cards

Objective: Write a program using a depth-first search that searches for a strategy for n = 3 and a deck of 8 cards, until you generate a legal table.

Hints Given: Note, that your program, due to massive backtracking, *may* take a very long time. In fact, with the ordering I suggested above, (ab, ac, bc, ba, ca, cb), you should expect to see your program hang up for hours in the large search space. Nobody knows what makes one ordering backtrack and another sail through. There may be some elegant mathematical theorem that predicts which ordering finds a clean 1-1 function, but no one has yet discovered such a theorem. Meanwhile, your program can check empirically.

Pseudocode:

```
Boolean findstrategy(i)  // fills an array with pairs from index i through 55

if i==56 then return true; // you filled up 0 through 55 so you are done
Let xyz be the triple in slot i;
for j =1 to 6
{  Try the next ordered pair from the triple xyz;   Call this a_b;
        If a_b has not yet been used, then
                {
                   assign a_b to slot i;  remember that a_b has been used  // Use a separate Boolean array
                   if findstrategy(i+1) return true;
                    // if false, then the call is backtracking and we continue in the loop to the next pair
                    remember that a_b is no longer being used;
                }
}
 return false;  //  you tried all 6 pairs and failed with all of them, so backtrack
```

For this program to work we created two new object classes: CardBoolean and CardCodes. CardBoolean holds a boolean used and a 2-digit integer code. CardCodes holds a 3-digit integer xyz and a 2-digit integer code.

```
public class CardBoolean
{
   boolean used;
   int code;
   public CardBoolean()
   {
      used = false;
      code = 0;
   }
   public CardBoolean(boolean used, int code)
   {
      this.used = used;
      this.code = code;
   }
   public CardBoolean(int code)
   {
      this.code = code;
   }
   public boolean getUsed()
   {
      return this.used;
   }
   public void setCode(int c)
   {
      this.code = c;
   }
}
```

```
public class CardCodes
{
   public int xyz;
   public int code;
   public CardCodes()
   {
      xyz = 0;
      code = 0;
   }
   public CardCodes(int xyz)
   {
      this.xyz = xyz;
   }
   public CardCodes(int xyz, int code)
   {
      this.xyz = xyz;
      this.code = code;
   }
   public int getCode()
   {
      return this.code;
   }
   public void setXYZ(int x)
   {
      xyz = x;
   }
}
```

FIG 4.1: Main Method

```java
public class cards8
{
   public static CardBoolean[] used;
   public static CardCodes[] codes;
   public static int[] order;

   public static void main(String[]args)
   {
      Scanner input = new Scanner(System.in);
      System.out.println("Enter the number of cards");
      int numCards = input.nextInt();   //                   (*__*)
      //order = new int[] {12,13,23,21,31,32};//this one does not work (>--3 3)
      //order = new int[] {21,13,12,31,23,32};//this one does not work ( >___>)
      order = new int[] {21,12,31,13,23,32}; //this one works YEAAA!!!
      //order = new int[] {21,12,31,13,32,23}; //this one works YEAAA!!!
      Filltable(numCards);
      Boolean(numCards);
      findStrat(0);
   }
```

This program starts off by asking the user for the number of cards being used. Then we use one of the ordering sets to see if the program can create a full table for the number of cards given. Then we call the Filltable passing in the number of cards to the method. Which fills the codes array will all of the 3-digit permutation (e.g. 123,124,125,126,...,678). Then we call Boolean passing in the number of cards and this method fills the used array will all of the possible 2-digit codes (e.g. 12,13,14,15,...,87). Finally we call the findStrat(0) method, which creates the finished table with all of the codes for each of the three digit permutations.

```java
public static boolean findStrat(int f)
{

   if(f == codes.length)
   {
      cards8.printTable();
      return true;
   }
   else
   {
      for(int i = 0; i< order.length;i++)
      {
         int xyz = codes[f].xyz;
         int a = xyz/100;
         int b = (xyz-a*100)/10;
         int c = xyz%10;
         switch(order[i])
         {
            case 12:
               {
                  if(used(a,b) == false)
                  {  codes[f].code = (a*10) + b;
                     change(a,b);
                     if(findStrat(f+1))
                        return true;
                     else
                     {
                        change(a,b);
                        codes[f].code = 0;
                     }
                  }
                  break;
               }
```

This is the findStrat method that we created from the pseudo code that was given to use. We are going to be using recursion and backtracking to get the answers that we need for the problem. So to start we created base case that would stop the program once the table is full. We made it then print out the table and return true to exit the method. If the program doesn't go into the if statement then it will enter the else statement. Remember the order array that we created in the main, the for loop that we have here is based off of it directly. Then the program gets the three digit permutation from the index of f in codes. Then we make each digit an integer by itself. I have a switch statement to switch on the ordering of the array. For example in the order array the second integer in it is 12, which means we take int a and b to try use for the code of that three digit number. The program first checks if that two digit code has not been used, and if not we set the code value to a*10 + b which gives us a two digit code to use and we change the boolean in the used array to true for the value of a*10 + b. From here we call findStrat(f+1) to continue the recursion. If that call comes back false then we change the boolean in the used array back to false and the code value of the three digit to 0. Then the for loop continues where it left off which is where we see the backtracking being done.

```
Enter the number of cards
8
123   21
124   12
125   51
126   61
127   71
128   81
134   31
135   13
136   16
137   17
138   18
145   41
146   14
147   47
148   48
156   15
157   57
158   58
167   67
168   68
178   78
234   32
235   23
236   62
237   72
238   82
245   42
246   24
247   27
248   28
256   52
257   25
258   85
267   26
268   86
278   87
345   43
346   34
347   73
348   83
356   53
357   35
358   38
367   63
368   36
378   37
456   54
457   45
458   84
467   64
468   46
478   74
567   65
568   56
578   75
678   76
```

The output of this program gives a full table of all of the different permutations for 8 cards each three digit number has a two digit code.

Program 6: The Intuitive Method With 8 Cards

Objective: Write a program to implement the intuitive practical method for n = 3, and print out the resulting 56 entry table.

The concept that we are going to be using for this program is to use a hide method that will pick out one of the cards to hide using the code system, that we created in program 4-5. As seen in program 4-5 we created a table using backtracking and an ordering array to get that values that we would use as the code for each 3 digit number.

FIG 6.1: Main Method

```
public static void main(String[] args)
{
    Scanner input = new Scanner(System.in);
    int[] array = FillTable.createTable(8);
    for(int i = 0; i < array.length;i++)
    {
        int xyz = array[i];
        int a = xyz/100;
        int b = (xyz-a*100)/10;
        int c = xyz%10;
        HideCard(a,b,c);
    }
}
```

In this program we start by creating a table that holds all 56 different permutations of 8 cards. These permutations are created in fillTable method and stored linearly in the array. Next we want to take a look at each of the permutations that are in the array, so we made a for loop that looks at what is stored in the array and we save that to an int. We want to split the three digit number into three separate single digit ints. Once we do this we call the HideCard method that will return two cards and the order of them in a certain permutation. The two cards that are left is the code to find the hidden card. We continue to do this until the full table of 56 number have their own codes.

The HideCard method has some really cool math in itself. In the paper where it is titled "A 'memorizable' or 'simply computed' strategy" it talks about how if you add up all the card values and mod it by the number of cards then you are left with a number from 0 to 4. Which it directly related to the index of the array that we created storing the values of the card. From the main method we pass in a, b, and c, which are put in to an array one for each index and then we sort this. To find which card to hide we add the total and mod by 3, which gives use a 0,1, or 2. Then we find the code if we take that number out of the array, and return the two other numbers in an ordered pair. This ordered pair is permutation of three digit number which tells the user what the hidden card number is.

<u>Output Table for Program 6</u>

123	2 3	235	2 5	368	6 3
124	1 4	236	3 2	378	7 8
125	1 2	237	3 7	456	6 5
126	2 6	238	2 8	457	7 4
127	1 7	245	2 4	458	5 4
128	2 1	246	4 6	467	6 4
134	1 3	247	2 7	468	8 6
135	3 5	248	4 2	478	8 4
136	1 6	256	6 2	567	7 6
137	3 1	257	5 2	568	8 5
138	3 8	258	5 8	578	7 5
145	1 5	267	6 7	678	8 7
146	4 1	268	8 2		
147	4 7	278	7 2		
148	1 8	345	4 5		
156	5 6	346	3 6		
157	7 1	347	4 3		
158	5 1	348	4 8		
167	6 1	356	5 3		
168	6 8	357	5 7		
178	8 1	358	8 3		
234	3 4	367	7 3		

```
public static void HideCard(int z, int z1, int z2)
{
   Scanner input = new Scanner(System.in);
   int[] cards = new int[3];
   int HiddenCard;
   cards[0] = z;
   cards[1] = z1;
   cards[2] = z2;
   Arrays.sort(cards);
   int a = cards[0];
   int b = cards[1];
   int c = cards[2];
   int f = a+b+c;
   int position = (f%3);
   HiddenCard = cards[position];
   int[] cardss = new int[2];
   int counter = 0;
   for(int i = 0; i < 3; i++)
   {
      if(cards[i] != HiddenCard)
      {
         cardss[counter] = cards[i];
         counter++;
      }
   }
```

For each of the three digit permutations each of them has the two cards that are shown to the user, the other one is the hidden card. So if we loop at code 8-6 the hidden card is the 4.

Program 7: The Intuitive Method With 124 Cards

Objective: Write a program that implements this intuitive method for *n* = 5 and 124 cards.

a. The user inputs five numbers, and the program returns an ordered subset of four numbers. You can do this program by looking at the 5 cards, determining the card to be hidden. Loop through the 124 cards and store the possible candidates in an array (there are 24 of these, including the hidden card. Look up the index of the hidden card in the array and find the permutation of that index (as in program 1). Use that permutation to order the 4 non-hidden cards. For example, given the 5 numbers 23, 27, 59, 87, and 93. You would hide 93, and since 93 is the 18th of the possible cards that could be hidden, you would exhibit the 18th permutation of the remaining 4 numbers, namely 59, 87, 27, 23.

FIG 7.1: Main Method

```
public static void main(String[]args)
{
   Scanner input = new Scanner(System.in);
   String choice  = "";
   System.out.println("Do you want to (H)ide or (F)ind a card?");
   while((!choice.equals("H")) && (!choice.equals("F")))
   {
      choice = input.nextLine();

   }

   if(choice.equals("H"))
   {
      HideCard();
   }
   else
   {
      System.out.println(FindCard());
   }
}
```

The HideCard method in this program is relatively similar to that of program 6 of which it adds up all 5 numbers that are given then mod by 5 to get a remainder, which is the index of the array that stores the number that is going to be hidden. This then will find the correct permutation to order the rest of the numbers in so we can show the user 4 cards and hide 1. The order of the 4 cards is a permutation seen in program 1(ex. 1 2 3 4 or 4 3 1 2 etc.) These permutations are relative to the four cards. For example if we have the cards 54, 32, 89, 90, ad 111, the hidden card would be 54 and the output order will be 89 111 32 90 giving it the permutation of 2 4 1 3.

```
Do you want to (H)ide or (F)ind a card?
H
Please enter the 5 cards in any order
54
32
89
90
111
89 111 32 90
```

b. The user inputs an ordered set of 4 cards, and the program returns the missing fifth card. You can do this by looping through the 124 cards and determining which 24 cards can be the missing card. Store these cards in an array. Then determine which permutation is indicated by the ordering of the cards. Use the number of permutations to pull out the hidden card from the array.

```
public static int FindCard()
{
   Scanner input = new Scanner(System.in);
   int[] cards = new int[4];
   int[] sortedCards = new int[4];
   int[]permutation = new int[4];
   int permutation1;
   System.out.println("Please enter the 4 cards in the same order seen");
   for(int i = 0; i < 4;i++)
   {
      cards[i] = input.nextInt();
      sortedCards[i] = cards[i];
   }
   Arrays.sort(cards);
   for(int i = 0 ; i < 4; i++)
   {
      for(int j =0;j<4;j++)
      {
         if(sortedCards[j] == cards[i])
         {
            permutation[i] = j+1;
         }
      }
   }
   permutation1 = LexicographicPermutation.calculate(permutation, 4);
```

This FindCard method asks the user to enter in the 4 cards in the order they see them, which we created and array to store each of these values. We also create another array that is going to get sorted and used as a comparison tool to get a permutation of the set of four numbers. Once we get a permutation of the set of numbers we can pass it in the calculate method from program 1 to get the number permutation is. Then the program creates another array that will store the available numbers that could be candidates for the hidden number of that permutation. Once we find all of the available values for the given permutation we then look inside of this array at the permutation number, minus 1, that we calculated from the calculate method and return that number located in the array.

What We Liked About This Lab

In this lab, we were able to see how some card tricks work mathematically and how the user is able to create a full table of a working strategy for a more simple card trick. This trick was really interesting on how it works and how we can show it off to friends and family. This lab linked very well with Discrete Math and the combinatorics topics we have been going over recently. Also, if we had any questions about how the card trick works or the combinatorics that was used in this LC lab we were able to ask the professor. The programs in the lab weren't the simplest but, we were able to work through them adding in simple print lines so we can see what the problem was.

What We Did Not Like About This Lab

The main problems our group encountered was the wording of some of the questions were poor. We spent a while trying to grasp exactly what to do for program 4-5 but eventually after reading over the code multiple times and after coming up with something that somewhat worked, we were able to get rid of all the bugs and find the solution. We also had some trouble working through program 7 because the trick was taught to us on the first day of the lab, which once it came time to code the program we had to read through the paper and the powerpoint once again to understand the trick in its entirety.

What We Learned From This Lab

Overall, this lab gave us a better understanding of combinatorics, permutations, and backtracking. The card trick used the choosing idea from discrete math of n choose m and how to exactly find the value of the missing card from program 7. The backtracking was a bit confusing at first but in the end it works. We learned how to perform the card trick with 124 cards and able to understand each part of the trick.