Marcus Mann

Theodore Walsh

28 January 2022

APMA 3100-004

Project 1: An Exploration of Counting Methods

On my honor as a student, I have neither given nor received aid on this assignment.

Theodore Walsh

Marcus Mann

1.

a) (astronomy, calculus, French, history) , (astronomy, calculus, French, literature) , (astronomy, calculus, French, organic chemistry) , (astronomy, calculus, French, physics) , (astronomy, calculus, history, literature) , (astronomy, calculus, history, organic chemistry) , (astronomy, calculus, history, physics) , (astronomy, calculus, literature, organic chemistry) , (astronomy, calculus, literature, physics) , … (history, literature, organic chemistry, physics)

b) (astronomy, calculus, French, history), (astronomy, calculus, history, French), (astronomy, history, calculus, French), (astronomy, history, French, calculus), (astronomy, French, history, calculus), (astronomy, French, calculus, history),

(calculus, astronomy, French, history), (calculus, astronomy, history, French), …

(French, astronomy, history, calculus),…

(history, French, astronomy, calculus),…

(astronomy, literature, organic chemistry, physics),…

c) The provided code counts the book orders as the iterator checks between for loops eliminates all duplicate combinations.

d)

import java.util.ArrayList;  
import java.util.List;

List<String> classes = new ArrayList<>();  
classes.add("astronomy");  
classes.add("calculus");  
classes.add("French");  
classes.add("history");  
classes.add("literature");  
classes.add("organic chemistry");  
classes.add("physics");  
  
int orders = 0;  
for (int i=1; i<=classes.size(); i++) {  
 for (int j=1; j<=classes.size(); j++) {  
 if (i == j) {  
 break;  
 }  
 for (int k=1; k<=classes.size(); k++) {  
 if ((i == k) || (j == k)) {  
 break;  
 }  
 for (int l=1; l<=classes.size(); l++) {  
 if ((l == i) || (l == j) || (l==k)) {  
 break;  
 }  
 orders++;  
 if (orders <= 20) {  
 System.*out*.println(classes.get(l - 1) + ", " + classes.get(k - 1) + ", " + classes.get(j - 1) + ", " + classes.get(i - 1));  
 }  
 }  
 }  
 }  
}  
  
System.*out*.println("Total orders: " + orders);

astronomy, calculus, French, history

astronomy, calculus, French, literature

astronomy, calculus, history, literature

astronomy, French, history, literature

calculus, French, history, literature

astronomy, calculus, French, organic chemistry

astronomy, calculus, history, organic chemistry

astronomy, French, history, organic chemistry

calculus, French, history, organic chemistry

astronomy, calculus, literature, organic chemistry

astronomy, French, literature, organic chemistry

calculus, French, literature, organic chemistry

astronomy, history, literature, organic chemistry

calculus, history, literature, organic chemistry

French, history, literature, organic chemistry

astronomy, calculus, French, physics

astronomy, calculus, history, physics

astronomy, French, history, physics

calculus, French, history, physics

astronomy, calculus, literature, physics

Total orders: 35

2.

If a requested password requires 8 digits of which 3 must be zeroes and the remaining 5 must be ones, we can represent the positions of the zeroes in the password array as solutions to the problem. For instance, the password string 00011111 can be represented as 123. Using Java, with 3 nested for- loops that skip over duplicate entries where one loop’s iterator equals another’s, that is, a zero cannot be in the same ith position multiple times, the solutions to the problem can be represented by the following code:

int counter = 0;  
System.*out*.print("{ ");  
for (int i=1; i<=8; i++) {  
 for (int j=1; j<=8; j++) {  
 if (j == i) {  
 break;  
 }  
 for (int k=1; k<=8; k++) {  
 if (k == i || k == j) {  
 break;  
 }  
 System.*out*.print("(" + k + "" + j + "" + i + ") ");  
 counter++;  
 if (counter %10 == 0) {  
 System.*out*.println();  
 }  
 }  
 }  
}  
System.*out*.println("}");  
System.*out*.println("Total passwords: " + counter);

This results in the 56 values that indicate the location of the zeroes in the password.

{ (123) (124) (134) (234) (125) (135) (235) (145) (245) (345)

(126) (136) (236) (146) (246) (346) (156) (256) (356) (456)

(127) (137) (237) (147) (247) (347) (157) (257) (357) (457)

(167) (267) (367) (467) (567) (128) (138) (238) (148) (248)

(348) (158) (258) (358) (458) (168) (268) (368) (468) (568)

(178) (278) (378) (478) (578) (678) }

Total passwords: 56

If this problem were to be expanded to a 30-digit password that required 13 0’s and 17 1’s, adapting the previous solution would be problematic as each of the 13 nested loops would need to be tested against each other to prevent 0’s ending in duplicate positions. This seems rather labor intensive and antithetical to the programming mantra. A recursive solution is probably possible though admittedly I could not discover how to make it work correctly. Programming the math directly was also problematic as it involves calculating the value of 30! which is many orders of magnitude larger than what can naturally be represented by a 64-bit integer, so Java’s BigInteger Class was used which cleverly turns the large value into an array of single-digit values.

import java. math.BigInteger;

int length = 30;  
int zeroes = 13;  
System.*out*.println("Total passwords: " + *passwords*(length, zeroes));

public static BigInteger passwords(int length, int zeroes) {  
  
 BigInteger numerator = BigInteger.*valueOf*(length);  
 for (int i=length-1; i > (length-zeroes); i-- ) {  
 numerator = numerator.multiply(BigInteger.*valueOf*(i));  
 }  
  
 BigInteger denominator = BigInteger.*valueOf*(zeroes);  
 for (int i=zeroes-1; i>0; i--) {  
 denominator = denominator.multiply(BigInteger.*valueOf*(i));  
 }

return numerator.divide(denominator);  
 }

Total passwords: 119759850

With the issue of the integer size solved, the values of the numerator and denominator were calculated using the formula for combinations, since each 0 is not unique, where The numerator is calculated first by only calculating the values of to reduce the number of calculations required and the overall size of the numerator. Next the denominator Is calculated, and then the two are divided using BigInteger’s appropriate method to get the solution of 119,759,850 passwords.

Problem 3:

1. We decided to approach this by multiplying the amount of combinations of zero positions (denoted by the index, starting at 0, with 7 being the max for our 8 digit number) and multiplying that by the number of arrangements of the rest of the 5 unique digits.
   * To find the count of all of the different positions of zero, we simply find because order of the positions of zero don’t matter.
   * To find the number of arrangements of the rest of the numbers, we can calculate it as Text

     Description automatically generated with medium confidence .
   * Taking the product of both both, we get: A picture containing logo

     Description automatically generated
   * Listing the first twenty values by hand:
     1. 00012345
     2. 00012346
     3. 00012347
     4. 00012354
     5. 00012356
     6. 00012357
     7. 00012364
     8. 00012365
     9. 00012367
     10. 00012374
     11. 00012375
     12. 00012376
     13. 00012435
     14. 00012436
     15. 00012437
     16. 00012453
     17. 00012456
     18. 00012457
     19. 00012463
     20. 00012465

def count\_passwords(num\_of\_digits: int = 8, possible\_value: int = 7) -> int:  
 total: int = 0  
 slots = possible\_value + 1  
 z\_count = 0  
 # iterate through  
 cur\_zero\_digits: List[int] = [0] \* 3  
 cur\_digits: List[int] = [0] \* 5  
 # skip over previously chosen number because we are looking at combinations and order doesn't matter.  
 for z\_1 in range(0, num\_of\_digits):  
 cur\_zero\_digits[0] = z\_1  
 for z\_2 in range(z\_1 + 1, num\_of\_digits):  
 cur\_zero\_digits[1] = z\_2  
 for z\_3 in range(z\_2 + 1, num\_of\_digits):  
 cur\_zero\_digits[2] = z\_3  
 # check for duplicates, because sets aren't allowed to contain duplicates.  
 if len(cur\_zero\_digits) != len(set(cur\_zero\_digits)):  
 # skip this one  
 continue  
 for d\_1 in range(1, slots):  
 cur\_digits[0] = d\_1  
 for d\_2 in range(1, slots):  
 cur\_digits[1] = d\_2  
 for d\_3 in range(1, slots):  
 cur\_digits[2] = d\_3  
 for d\_4 in range(1, slots):  
 cur\_digits[3] = d\_4  
 for d\_5 in range(1, slots):  
 cur\_digits[4] = d\_5  
 # check for duplicates, because sets aren't allowed to contain duplicates.  
 if len(cur\_digits) != len(set(cur\_digits)):  
 # skip this iteration  
 continue  
 total += 1  
 value: List[int] = [d\_1, d\_2, d\_3, d\_4, d\_5]  
 value.insert(z\_1, 0)  
 value.insert(z\_2, 0)  
 value.insert(z\_3, 0)  
 if total <= 100:  
 print(reduce(lambda a, b: str(a) + str(b), value))  
 return total

The program will iterate through the combinations of the positions of zeros, and for each combination iterate through all the permutations of possible arrangements of distinctly unique remaining digits, and merge the zeroes into the 5 generated 1-7 digits at the respective indices.  
  
the first 100:

00012345

00012346

00012347

00012354

00012356

00012357

00012364

00012365

00012367

00012374

00012375

00012376

00012435

00012436

00012437

00012453

00012456

00012457

00012463

00012465

00012467

00012473

00012475

00012476

00012534

00012536

00012537

00012543

00012546

00012547

00012563

00012564

00012567

00012573

00012574

00012576

00012634

00012635

00012637

00012643

00012645

00012647

00012653

00012654

00012657

00012673

00012674

00012675

00012734

00012735

00012736

00012743

00012745

00012746

00012753

00012754

00012756

00012763

00012764

00012765

00013245

00013246

00013247

00013254

00013256

00013257

00013264

00013265

00013267

00013274

00013275

00013276

00013425

00013426

00013427

00013452

00013456

00013457

00013462

00013465

00013467

00013472

00013475

00013476

00013524

00013526

00013527

00013542

00013546

00013547

00013562

00013564

00013567

00013572

00013574

00013576

00013624

00013625

00013627

00013642

The final count: 141120, which matches up with our calculation.

1. To calculate for arbitrary n number of zeros and m number of digits for values 1-9, we can use the expression

Text

Description automatically generated with low confidence

4. The following function will be used in all three parts, and will be fed in as the “counts” variable, as you will see in the

if \_\_name\_\_ == "\_\_main\_\_":

portion of the code.

# Problem 4. A 5-letter password is to be constructed from the following letters: A, B, C, D, and E and the password  
# must contain exactly 3 of the letters. (Obviously at least one of them will need to be used more than once to  
# create a 5-letter password.)  
  
# Because the instructions for Problem 4 never said anything about points being deducted for using a collection based  
# solution, I generate all the passwords and only filter based off the exactly 3 distinct characters requirement,  
# which is reused through all 3 parts of Problem 4.  
def generate\_all\_passwords() -> List[Dict[str, Union[str, int]]]:  
 ret: List[Dict[str, Union[str, int]]] = []  
 letters = ['A', 'B', 'C', 'D', 'E']  
 # iterate through all permutations of passwords  
 for a in letters:  
 for b in letters:  
 for c in letters:  
 for d in letters:  
 for e in letters:  
 # construct the string from the different permutations of the letters.  
 pwd: str = f"{a}{b}{c}{d}{e}"  
 # Create a dictionary that has a mapping of the characters to the count of the characters  
 # in the password. This will be used in other filters.  
 counts: Dict[str, Union[int, str]] = {}  
 # loop through the characters in the string  
 for i in pwd:  
 # if this string isn't in the counts dictionary, then add it to the dictionary and set it   
 # to 1. otherwise, increment by 1.  
 if i in counts:  
 counts[i] += 1  
 else:  
 counts[i] = 1  
 # only return a password if there are exactly 3 distinct characters. sets aren't allowed to have  
 # duplicate values, so when we check set(pwd), it will deduplicate the string's characters.  
 if len(counts.keys()) == 3:  
 # Store word in counts dictionary for later use  
 counts['word'] = pwd  
 # add to array to later return, and go to next iteration  
 ret.append(counts)  
 return ret

This is then run at the bottom:

if \_\_name\_\_ == "\_\_main\_\_":  
 print(f"The answer to 3.b is {count\_passwords()}")  
 pwds = generate\_all\_passwords()  
 print(f"The answer to 4.a = {count\_4\_a(generate\_all\_passwords())}")  
 print(f"The answer to 4.b = {count\_4\_b(pwds)}")  
 print(f"The answer to 4.c = {count\_4\_c(pwds)}")

for part a:

# 2 A's, 2 B's, 1 C  
def count\_4\_a(counts: List[Dict[str, int]]) -> int:  
 ret: int = 0  
 # loop through all of the words that made it through the first round of filtration in generate\_all\_passwords()  
 for c in counts:  
 # if A, B, and C are in the dictionary, and if the counts of them are their respective correct counts,  
 # then increment the counter by 1.  
 if "A" in c.keys() and "B" in c.keys() and "C" in c.keys() and c["A"] == 2 and c["B"] == 2 and c["C"] == 1:  
 ret += 1  
 return ret

For part b:

# Suppose A,B, and C are the letters selected and there are no other restrictions.  
# How many passwords exist in this scenario?  
def count\_4\_b(counts: List[Dict[str, int]]) -> int:  
 ret: int = 0  
 # loop through all the words that made it through the first round of filtration in generate\_all\_passwords()  
 for c in counts:  
 # If A, B, and C are selected:  
 if "A" in c.keys() and "B" in c.keys() and "C" in c.keys():  
 ret += 1  
 return ret

For part c:

# Now suppose we don’t put any restrictions on the letters chosen. (It could be A,B, and C, or it could be A, C, and D,  
# or it could be B, D, and E, etc.) How many different passwords exist in this scenario?  
def count\_4\_c(counts: List[Dict[str, int]]) -> int:  
 # we already matched the filter with our original filter in generate\_all\_passwords(), so return the length.  
 return len(counts)

When we run, we get the following results:

Text

Description automatically generated