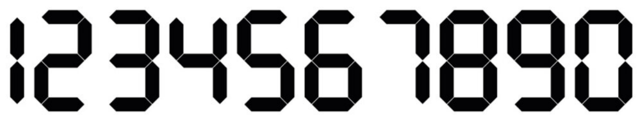
Mathematical Modeling Samples

Andrew Attilio

Contents

1 Clock Energy Optimization

Problem. On a digital number display, digits consist of small illuminated line segments. The energy consumption of the display depends on the number of small line segments switched on or off as time elapses. For example, when a 3 changes to a 4, two line segments are switched off and one is switched on, which means 3 switch operations. During a full cycle of 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, this adds up to a total of 30 switch operations.



If we cycle through the numbers in some different order, the number of switch operations could be reduced. Find the minimum number of switch operations that could be achieved in a full cycle, and give an example for a possible order of digits.

Solution. We will model the problem with Python.

Our program starts by storing each number as a 7-digit binary string. Each digit represents a line segment on the clockface, from top-to-bottom and left-to-right. For example:

```
zero = '1110111'        five = '1101011'
one = '0010010'         six = '1101111'
two = '1011101'         seven = '1010010'
three = '1011011'       eight = '1111111'
four = '0111010'        nine = '1111011'
```

Using a built-in function in python, we generate a list of all possible permutations of the set $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ and store this list of permutations in the variable all_permutations.

The meat of the code is below, with nested for loops. The code has comments, but let's pre-analyze it here. The first for loop iterates through each possible permutation. Once we are in a permutation, the next for loop iterates through each digit of the permutation. It compares the first digit to a 'nil' value of '0000000' since we are starting from a blank clock. Then, the 3rd and final for loop compares each line segment of the current_num with the next_num. If the numbers don't match, then we add one to the current_count which represents the total energy usage for this permutation. This is repeated for each number in the permutation.

Once we break out of these nested for loops, we compare the total energy consumption to the current record holder (note that we initialize the current_record with an impossibly high number, so it is sure to be beaten). If the current permutation has a new record, then it is recorded, along with the permutation that acheived the record.

Lastly, we reset the current_count to 0 and make the clock-face 'blank' by assigning it the 'nil'. The program then repeats the above process for every possible permutation (all 10! of them).

```
current_record = 1000
record_holder = ''

current_count = 0
current_num = nil

for perm in all_permutations: # for each possible sequence
    for digit in perm: # go through each number in sequence
        next_num = numbers[digit] # store the switch values for next number
        for i in range(7): # for each switch
            if current_num[i] != next_num[i]: # if they are not equal
                current_count += 1 # add 1 to the count for this perm
        current_num = next_num # next number is now 'current'

        """ If new record, update """
    if current_count < current_record:
        current_record = current_count
        record_holder = perm

        """ Reset stats for next perm """
    current_count = 0
    current_num = nil
```

After running the program (we omitted some lines above for brevity's sake), we receive this output:

```
(1, 7, 3, 2, 0, 8, 6, 5, 9, 4)
Record: 16
Runtime: 27.23 seconds
```

We found that the sequence (1, 7, 3, 2, 0, 8, 6, 5, 9, 4) generated the lowest energy consumption, which was 16.

2 Factorial Summation

Problem. Determine the value of $2\sqrt{1*2*3+2*3*4+3*4*5+...+1998*1999*2000}$.

Proof. After playing around with the numbers in the radicand, a pattern clearly reveals itself:

$$2\sqrt{(3!/0!)+(4!/1!)+(5!/2!)+...+(2000!/1997!)}$$

This can be formalized into:

$$2\sqrt{\sum_{i=3}^{2000}\frac{n!}{(n-3)!}}$$

This looks like a job for Python. This code models the problem and prints out the answer to the console:

```
def factorial_summation():
    total_sum = 0

    for i in range (3, 2000):
        ith_sum = i * (i-1) * (i - 2)
        total_sum += ith_sum

    return (2 * (total_sum)**(1/2))

print(factorial_summation())
```

Running this code results in an output of 39994001. □

3 Jelly Beans

Problem. In this problem, we are exploring the most probable outcome after a jelly bean experiment. The experiment proceeds as follows:

- Stir together 2023 red jelly beans and 2023 green jelly beans.
- Pull 3 jelly beans out of the jar at a time.
- If they are all the same color, eat all 3 jelly beans.
- If there are 2 of one color, eat 1 of the majority color and put the other 2 jelly beans back in the jar.

After following this process, there are 4 distinct ways for the experiment to end:

a) No beans.

b) One bean (either color is equally probable).

c) Two beans of the same color (either color is equally probable).

d) One red and one green.

Solution. Calculating the probabilities of hundreds of consecutive draws seemed impossibly complicated, so I coded a Python program that simulates the stated problem, drawing three beans at a time until no more than 2 remain.

The code for my bean_jar_experiment() function is listed below. There are two global variables, red_count and green_count. The bean_jar_experiment() continuously calls the draw_beans function until there are 2 or less beans left in the virtual jar. Each instance of draw_beans generates three random numbers, each of which corresponds to either a red or green bean. Depending on the results of the 3 randomly-generated beans, the function subtracts the proper amount of beans from red_count or green_count, simulating consumption of those beans. The draw_count is initialized to 0, and I simulated each red bean as -1 for the draw_count and each green bean as $+1$. For example, the following line represents consuming three red beans, given that the draw_count is equal to -3:

```
case (-3): # all red
        red_count -= 3
```

The following is the full code for the program:

```python
red_count = 2023
green_count = 2023

def bean_jar_experiment():
    while (red_count + green_count) > 2:
        draw_beans(red_count, green_count)

    print('Red: ', red_count)
    print('Green: ', green_count)

def draw_beans(red, green):
    global red_count
    global green_count

    draw_count = 0 # if < 0: red, if > 0: green
    for i in range(3):
        random_int = randint(1, (red + green))
        if random_int <= red:
            draw_count -= 1
        else:
            draw_count += 1
    match draw_count:
        case (-3): # all red
            red_count -= 3
        case (-1): # 2 red, 1 green
            red_count -= 1
        case (1): # 2 green, 1 red
            green_count -= 1
        case (3): # all green
            green_count -= 3
```

Great, but this would be most useful by running the program hundreds, thousands, or millions of times and collecting all of the results. Let's do that.

I created a dictionary for the program that stores the results of each occurrence of the bean_jar_experiment function. The mappings are based on the 4 cases $\{a, b, c, d\}$ defined in the intro section of this paper. The initial dictionary is:

```
results = {'a': 0, 'b': 0, 'c': 0, 'd': 0}
```

Each time the program runs, it will add 1 to the corresponding case result. I ran the program 1000 times, then 10000 times, then 100000 times. Here are the results:

```
RESULTS FROM 10000 EXPERIMENTS
{'a': 1057, 'b': 3335, 'c': 2885, 'd': 2723}
Runtime: 72.10 seconds

RESULTS FROM 100000 EXPERIMENTS:
{'a': 10258, 'b': 33810, 'c': 28834, 'd': 27098}
Runtime: 754.85 seconds
```

The results are consistent. According to the Python program, **case b is most likely to occur**, which corresponds to ending with either one green or one red bean in the jar. **Case a is the least likely outcome**, which is the case in which we end with no beans.

Let's consider the data from running the experiment 100000 times. The probabilities of each case are:

- $a)$ No beans = **10.3%**
- $b)$ One bean = **33.8%**
- $c)$ Two beans of the same color = **28.8%**
- $d)$ One red and one green = **27.1%**