ProjectEuler Archive 19: Gregorian calendar calculations

YarrowForAlgernon

August 2025

Contents

1 Challenge Description

Challenge Name: *Counting Sundays*
Challenge Description:

You are given the following information, but you may prefer to do some research for yourself.

1 Jan 1900 was a Monday. Thirty days has September, April, June and November. All the rest have thirty-one, Saving February alone, Which has twenty-eight, rain or shine. And on leap years, twenty-nine. A leap year occurs on any year evenly divisible by 4, but not on a century unless it is divisible by 400.

How many Sundays fell on the first of the month during the twentieth century (1 Jan 1901 to 31 Dec 2000)?

2 Naive Solution

2.1 The First Thing That Comes To Mind

We need to calculate the time span of 100 years. This is 1200 months, of which we need to tabulate which have Sunday as their first day.

We know that April, June, September and November have 30 days, and all the rest except February have 31. We can calculate what day the first of each month will be through modular arithmetic. The first Sunday of 1901 is the 6th of January, which we will calculate all subsequent dates from.

The first thought I had was to set a 'day counter' variable to 7, and continually increment it by 7 to keep it pointing to Sunday. Once the day counter exceeds the number of days in that month, we take its modulus as the number of days in that month. This way, we can traverse each month while still having the day counter point to Sunday. This means we will have to loop through all 5,200 weeks in the 100 year period, but provides a straightforward way to brute force the answer.

2.2 The Algorithm

```
1 from time import time_ns
2
3 class archive19:
4
5
6     def __init__(this):
7
8         this.monthToDayDict = {1: 31, 2: 28, 3: 31, 4: 30, 5: 31, 6: 30,
```

```
        7: 31, 8: 31, 9: 30, 10: 31, 11: 30, 12: 31}
        this.dayCounter = 6
        this.sundayCounter = 0
        this.monthCounter = 1
        this.currentYear = 1901
        this.endYear = 2001


    def checkLeapYear(this):
        '''If the year is a leap year, change the number of days in
        February from 28 days to 29 days.'''
        if (this.currentYear % 4 == 0) and (this.currentYear % 100 != 0 or
        this.currentYear % 400 == 0):
            this.monthToDayDict[2] = 29
        else:
            this.monthToDayDict[2] = 28


    def checkDayCounterExceedsMonth(this):
        return this.dayCounter > this.monthToDayDict[this.monthCounter]


    def rollOverDayCounter(this):
        '''Take the current dayCounter modulo the number of days in
        the month. The end result is the first sunday in the next month.'''
        this.dayCounter %= this.monthToDayDict[this.monthCounter]
        this.monthCounter += 1


    def checkMonthCounterExceedsYear(this):
        return this.monthCounter > 12


    def checkForFirstSunday(this):
        if this.dayCounter == 1:
            this.sundayCounter += 1


```

```
47      def incrementWeek(this):
48          this.dayCounter += 7
49
50
51      def incrementYear(this):
52          this.currentYear += 1
53          this.monthCounter = 1
54
55
56      def calculateSolution(this):
57          '''calculates all sundays that fall on the first of a month
58          in the years between 1901 and 2001.'''
59
60          while this.currentYear  < this.endYear:
61
62              this.incrementWeek()
63
64              if this.checkDayCounterExceedsMonth():
65                  this.rollOverDayCounter()
66
67              if this.checkMonthCounterExceedsYear():
68                  this.incrementYear()
69                  this.checkLeapYear()
70
71              this.checkForFirstSunday()
72
73          return this.sundayCounter
74
75
76  if __name__ == "__main__":
77
78      startTime = time_ns()
79      archiveSolution = archive19()
80      noOfDays = archiveSolution.calculateSolution()
81      stopTime = time_ns()
82      print(noOfDays, "\nTime Taken (ns): " + str(stopTime - startTime))
```

This algorithm has a time complexity of O(n).

3 Efficient solutions

Each solution discussed will be (relatively) faster than the last.

3.1 Zellers congruence

3.1.1 Introduction

In 1882 Julius Christian Johannes Zeller published an equation to calculate what day falls on any given date [2]. Using this method we can compute the beginning of each month and determine if it is a Sunday. This allows us to loop month by month rather than week by week.

3.1.2 Explanation of Maths

Theorem 3.1. Here is the original formula Zeller published for the Gregorian calendar:

$$h = q + \lfloor \frac{26(m+1)}{10} \rfloor + k + \lfloor \frac{k}{4} \rfloor - 2e \mod 7 \tag{3.1}$$

Where h is the day of the week (0 = Tuesday), q is the day of the month, m is the number of the month in the year, J is the number of the century, K is the year of that century and e is $J/4$

There are two things to note here. The first is that January and February are computed as the 13th and 14th months of the previous year, for example 01/1/1967 (DD/MM/YYYY) is interpreted as 01/13/1966.

The second is that Zeller notes that the divisions were actually taking the quotient, which would lead to integers. This is why we add floor symbols around all divisions, as taking the quotient is equivalent to taking the floor of the fractions.

Theorem 3.2. We will modify this equation into a more convenient form by keeping track of less variables:

$$h = q + \lfloor \frac{26(m+1)}{10} \rfloor + Y + \lfloor \frac{Y}{4} \rfloor - \lfloor \frac{Y}{100} \rfloor + \lfloor \frac{Y}{400} \rfloor \mod 7 \tag{3.2}$$

where Y is the current year.

3.1.3 The Algorithm

```
from time import time_ns

class archive19:


    def __init__(this):
        '''
        :param this.sundayCount:
            The number of Sundays that fall on the first of the
            months between this.currentYear and this.endYear.
        :param this.month:
            The month used in calculation of Zellers congruence
            (February = 14, March = 3).
        :param this.zellerYear:
            The year used in calculation of Zellers congruence, which
            is the previous year if the current month is January
            or February.
        :param this.currentYear:
            The current Year which we start calculations on.
        :param this.endYear:
            The year to end calculations on.
        :param this.firstOfMonth:
            An integer value between 0-6, which correspond to the
            days of the week (0 = >Saturday).
        :param this.day:
            The day to use in calculation of zellers congruence.
            '''

        this.sundayCount = 0
        this.month = 13
        this.zellerYear = 1900
        this.currentYear = 1901
        this.endYear = 2001
        this.firstOfMonth = 0
        this.day = 1


```

```
    def adjustDate(this) -> None:
        if this.month > 14:
            this.month = 3
            this.zellerYear += 1
        if this.month == 12:
            this.currentYear += 1


    def getFirstOfMonth(this) -> None:
        monthCode = (13 * (this.month + 1)) // 5
        this.firstOfMonth = this.day + monthCode + this.zellerYear +
        (this.zellerYear // 4) - (this.zellerYear // 100) + (this.zellerYear // 400)
        this.firstOfMonth %= 7


    def checkForFirstSunday(this) -> None:
        if this.firstOfMonth == 1:
            this.sundayCount += 1


    def incrementMonth(this, step: int = 1) -> None:
        this.month += step


    def calculateDaysOnFirstOfMonth(this) -> int:
        '''calculates all Sundays that fall on the first of a month
        between this.currentYear and this.endYear.'''

        while this.currentYear  < this.endYear:

            this.getFirstOfMonth()

            this.checkForFirstSunday()

            this.incrementMonth()

            this.adjustDate()

```

```
76          return this.sundayCount
77
78
79  if __name__ == "__main__":
80
81      startTime = time_ns()
82      archiveSolution = archive19()
83      noOfDays = archiveSolution.calculateDaysOnFirstOfMonth()
84      stopTime = time_ns()
85      print(noOfDays, "\nTime Taken (ns): " + str(stopTime - startTime))
```

The time complexity of this algorithm is O(n)

3.2 Properties Of The Gregorian Calendar

3.2.1 14 Unique Gregorian Years

The Gregorian calendar was introduced as a replacement to the Julian calendar. This was done in order to consistently calculate when Easter occurred and more closely simulating the rotation of the earth around the sun (or as then believed, the suns rotation around the earth). As the Julian calendar stated every four years was a leap year, which was on average 356.25 days a year. While close, the real number of days is closer to 365.2422, and the Gregorian calendar changed its calculations to reduce the number of leap years, though it is still off from the tropical year by roughly 26 seconds every year [1].

Because the number of weeks in a non-leap year is exactly 52, every year will start and end with the same day. This means the rest of the days between the first and last days can be immediately known, as they progress in a predictable pattern (Monday -¿ Tuesday -¿ Wednesday, etc). Because there are only 7 days in a week, there can only be 7 unique non-leap years where the days of the week all fall on the same numerical days of the year.

Leap years have 52 weeks and 1 day, which means they follow a slightly different pattern but also only have 7 unique forms, as we only need to know what day of the week the 1st of January is to calculate the rest of the year. All together, the Gregorian calendar has only 14 'unique' years.

The only two variables needed to differentiate between unique years are the day the year starts on, and whether it is a leap year. Because the number of days in each month stays consistent across years this way, we can store the configurations of each

'unique' year and match them to each year from 1901 to 2000. This way, we can loop exactly n times by matching each year to its respective unique year.

My initial thought was to use Zellers congruence to calculate the start of each year but this is not actually necessary. These unique configurations progress in a predictable pattern. If the current year is a common year the day the next year begins with is the following day the current year starts with, and two days over if the current year is a leap year. For example, if 2002 (common year) begins with a Tuesday, 2003 will begin with a Wednesday. If 2024 (leap year) begins with Thursday, 2005 will begin with Saturday.

Given that we know that 1st January 1900 is a Monday, we can tell what day all subsequent years will start with.

This way, we no longer need to loop year by year using Zellers congruence. Using a dictionary of what unique year corresponds to the number of Sundays that fall on the 1st of its months, we can loop n times.

3.2.2 The First Algorithm

```
1
2  from time import time_ns
3
4  class archive19:
5
6
7      def __init__(this):
8          '''Their are 14 unique years in the Gregorian calendar,
9          meaning there are 14 configurations in which the days of
10         the week occupy the same numerical days in a year. The way
11         to distinguish between unique years is whether it is a leap
12         year and what day of the week it starts with.
13
14         :param this.yearLookup:
15             This is a dictionary whose key values represent the
16             days of the week (0 = saturday). Their corresponding
17             values are the number of sundays that fall on the first
18             of each month in that year.
19         :param this.leapYearLookup:
20             This is a dictionary whose key values represent the
21             days of the week (0 = saturday). Their corresponding
```

```
22              values are the number of sundays that fall on the first
23              of each month in that leap year.
24          :param this.yearLookupIndex:
25              The index into each yearLookup dictionary. The value
26              of this variable corresponds to the first day in a year
27              (0 = saturday).
28          :param this.sizeOfWeek:
29              A week has seven days.
30          :param this.sundayCount:
31              A counter of all sundays that fall on the first of a month
32              between this.currentYear and this.endYear.
33          :param this.curentYear:
34              The current year.
35          :param this.endYear:
36              The year to end calculations on.
37              '''
38
39          this.yearLookup = {0: 1, 1: 2, 2: 2, 3: 2, 4: 1, 5: 3, 6: 1}
40          this.leapYearLookup = {0: 1, 1: 3, 2: 2, 3: 1, 4: 2, 5: 2, 6: 1}
41          this.yearLookupIndex = 3
42          this.sizeOfWeek = 7
43          this.sundayCount = 0
44          this.currentYear = 1901
45          this.endYear = 2001
46
47
48      def calculateSundaysInYear(this) -> None:
49          if this.checkLeapYear():
50              this.sundayCount += this.leapYearLookup[this.yearLookupIndex]
51              this.yearLookupIndex += 2
52          else:
53              this.sundayCount += this.yearLookup[this.yearLookupIndex]
54              this.yearLookupIndex += 1
55
56
57      def updateLookupIndex(this) -> None:
58          '''If the lookup index, which corresponds to the days
59          of the week, exceeds the number of days in a week it wraps
```

```
60          around the week again.'''
61          this.yearLookupIndex %= this.sizeOfWeek
62
63
64      def checkLeapYear(this) -> bool:
65          return (this.currentYear % 4 == 0) and (this.currentYear % 100 != 0 or
66          this.currentYear % 400 == 0)
67
68
69      def incrementYear(this, step: int = 1) -> None:
70          this.currentYear += step
71
72
73      def calculateSolution(this):
74          '''calculates all days that fall on the first of a month
75          between this.currentYear and this.endYear'''
76
77          while this.currentYear < this.endYear:
78
79              this.calculateSundaysInYear()
80
81              if this.yearLookupIndex >= this.sizeOfWeek:
82                  this.updateLookupIndex()
83
84              this.incrementYear()
85
86          return this.sundayCount
87
88
89  if __name__ == "__main__":
90
91      startTime = time_ns()
92      archiveSolution = archive19()
93      noOfDays = archiveSolution.calculateSolution()
94      stopTime = time_ns()
95      print(noOfDays, "\nTime Taken (ns): " + str(stopTime - startTime))
```

The time complexity of this algorithm is O(n).

3.2.3 The 28 and 400 year solar cycle

There is a property of the sequence unique years in the Gregorian calendar - they have a period of 28 years. This is not completely accurate, as years that are divisible by 100 by not 400 (1900, 1700, etc) disrupt this cycle. However, for our case we do not need to take this into consideration as 1901 to 2000 includes no 'disruptive' years. This allows for an unbroken 28 year cycle.

The number of Sundays that fall on the first of each month in this cycle is 48. So if we take the quotient of $\frac{100}{24}$ and multiply it by 48 we will get the first part of our answer. Then we can calculate the remaining years using our lookup algorithm.

3.2.4 The Second Algorithm

```
from time import time_ns

class archive19:


    def __init__(this):
        '''Their are 14 unique years in the Gregorian calendar,
        meaning there are 14 configurations in which the days of
        the week occupy the same numerical days in a year. The way
        to distinguish between unique years is whether it is a leap
        year and what day of the week it starts with.

        :param this.yearLookup:
            This is a dictionary whose key values represent the
            days of the week (0 = saturday). Their corresponding
            values are the number of sundays that fall on the first
            of each month in that year.
        :param this.leapYearLookup:
            This is a dictionary whose key values represent the
            days of the week (0 = saturday). Their corresponding
            values are the number of sundays that fall on the first
            of each month in that leap year.
        :param this.yearLookupIndex:
            The index into each yearLookup dictionary. The value
            of this variable corresponds to the first day in a year
            (0 = saturday).
```

```
27          :param this.sizeOfWeek:
28              A week has seven days.
29          :param this.sundayCount:
30              A counter of all sundays that fall on the first of a month
31              between this.currentYear and this.endYear.
32          :param this.curentYear:
33              The current year which we start calculations on.
34          :param this.endYear:
35              The year to end calculations on.
36          :param this.quotient:
37              The quotient of the number of years to iterate over divided
38              by 28 (28 year cycle), multiplied by the number of
39              Sundays that fall on the first of the months inside the 28
40              year cycle (48).
41              '''
42
43          this.yearLookup = {0: 1, 1: 2, 2: 2, 3: 2, 4: 1, 5: 3, 6: 1}
44          this.leapYearLookup = {0: 1, 1: 3, 2: 2, 3: 1, 4: 2, 5: 2, 6: 1}
45          this.yearLookupIndex = 3
46          this.sizeOfWeek = 7
47          this.sundayCount = 0
48          this.currentYear = 1901
49          this.endYear = 2001
50          this.quotient = 0
51
52      def calculateSundaysInYear(this) -> None:
53          if this.checkLeapYear():
54              this.sundayCount += this.leapYearLookup[this.yearLookupIndex]
55              this.yearLookupIndex += 2
56          else:
57              this.sundayCount += this.yearLookup[this.yearLookupIndex]
58              this.yearLookupIndex += 1
59
60      def getQuotientResult(this):
61          '''Updates the value this.quotient to the number of Sundays
62          that fall on the first of each month in the 28 year cycles
63          between the currentYear and endYear.'''
64          numberOfYears = (this.endYear - this.currentYear)
```

```
        this.quotient = numberOfYears // 28
        this.currentYear += (this.quotient * 28)
        this.quotient *= 48


    def updateLookupIndex(this) -> None:
        '''If the lookup index, which corresponds to the days
        of the week, exceeds the number of days in a week it wraps
        around the week again.'''
        this.yearLookupIndex %= this.sizeOfWeek


    def checkLeapYear(this) -> bool:
        return (this.currentYear % 4 == 0) and (this.currentYear % 100
        != 0 or this.currentYear % 400 == 0)


    def incrementYear(this, step: int = 1) -> None:
        this.currentYear += step


    def calculateSolution(this):
        '''calculates all days that fall on the first of a month
        between this.currentYear and this.endYear'''

        this.getQuotientResult()

        while this.currentYear < this.endYear:

            this.calculateSundaysInYear()

            if this.yearLookupIndex >= this.sizeOfWeek:
                this.updateLookupIndex()

            this.incrementYear()

        return this.sundayCount + this.quotient

```

```
103
104 if __name__ == "__main__":
105
106     startTime = time_ns()
107     archiveSolution = archive19()
108     noOfDays = archiveSolution.calculateSolution()
109     stopTime = time_ns()
110     print(noOfDays, "\nTime Taken (ns): " + str(stopTime - startTime))
```

The time complexity of this algorithm is O(1), as the amount of loops will never exceed 28.

4 Bibliography

References

[1] Joseph C. Kolecki. *Calendar Calculations*. URL: https://www.grc.nasa.gov/www/k-12/Numbers/Math/Mathematical_Thinking/calendar_calculations.htm. (accessed: 07.08.2025).

[2] Christian Zeller. "Die Grundaufgaben der Kalenderrechnung auf neue und vereinfachte Weise gelöst". German. In: *Württembergische Vierteljahrshefte für Landesgeschichte* 5 (1882), pp. 313–314. DOI: 10.53458/wvjh.v5i.14389.