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 number of Sundays that fall on the first of a month over a period of 1200 months AKA 100 years from January 1st 1901 - December 31st 2000.

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,

9           7: 31, 8: 31, 9: 30, 10: 31, 11: 30, 12: 31}

10          this.dayCounter = 6

11          this.sundayCounter = 0

12          this.monthCounter = 1

13          this.currentYear = 1901

14          this.endYear = 2001

15

16

17      def checkLeapYear(this):

18          '''If the year is a leap year, change the number of days in

19          February from 28 days to 29 days.'''

20          if (this.currentYear % 4 == 0) and (this.currentYear % 100 != 0 or

21          this.currentYear % 400 == 0):

22              this.monthToDayDict[2] = 29

23          else:

24              this.monthToDayDict[2] = 28

25

26

27      def checkDayCounterExceedsMonth(this):

28          return this.dayCounter > this.monthToDayDict[this.monthCounter]

29

30

31      def rollOverDayCounter(this):

32          '''Take the current dayCounter modulo the number of days in

33          the month. The end result is the first sunday in the next month.'''

34          this.dayCounter %= this.monthToDayDict[this.monthCounter]

35          this.monthCounter += 1

36

37

38      def checkMonthCounterExceedsYear(this):

39          return this.monthCounter > 12

40
```

```python
41
42      def checkForFirstSunday(this):
43          if this.dayCounter == 1:
44              this.sundayCounter += 1
45
46
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   Zellers Congruence

3.1   Introduction

In 1882 Julius Christian Johannes Zeller published an equation to calculate what day falls on any given date [1]. 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.

3.2   Explanation Of Maths

Theorem 3.1 (Zellers Congruence). Here is the 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 01/01/1967 (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.

Corollary 3.1.1. This is a modified form of Zellers Congruence which was taken from this paper [2] which also explains Theorem 3.1 in much more (and much better) detail than in this paper.

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

where h is the day of the week ($0 =$ Saturday), q the day of the month, m is the month ($3 =$ March, $14 =$ February) and Y is the current year.

3.3 The Algorithm

```python
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
```

```
34          this.firstOfMonth = 0
35          this.day = 1
36
37
38      def adjustDate(this) -> None:
39          if this.month > 14:
40              this.month = 3
41              this.zellerYear += 1
42          if this.month == 12:
43              this.currentYear += 1
44
45
46      def getFirstOfMonth(this) -> None:
47          monthCode = (13 * (this.month + 1)) // 5
48          this.firstOfMonth = this.day + monthCode + this.zellerYear +
49          (this.zellerYear // 4) - (this.zellerYear // 100) + (this.zellerYear // 400)
50          this.firstOfMonth %= 7
51
52
53      def checkForFirstSunday(this) -> None:
54          if this.firstOfMonth == 1:
55              this.sundayCount += 1
56
57
58      def incrementMonth(this, step: int = 1) -> None:
59          this.month += step
60
61
62      def calculateDaysOnFirstOfMonth(this) -> int:
63          '''calculates all Sundays that fall on the first of a month
64          between this.currentYear and this.endYear.'''
65
66          while this.currentYear  < this.endYear:
67
68              this.getFirstOfMonth()
69
70              this.checkForFirstSunday()
71
```

```
72              this.incrementMonth()
73
74              this.adjustDate()
75
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)

4   14 Unique Gregorian Years

4.1   The Gregorian Calendar

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 [3].

4.2   Explanation Of 'Unique' Years

Because the number of weeks in a non-leap year (AKA a 'common' 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 and sum the number of Sundays that fall on the first of the those unique years months.

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 2004 (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.

4.3   The First Algorithm

```
1   from time import time_ns
2
3   class archive19:
4
5
6       def __init__(this):
7           '''Their are 14 unique years in the Gregorian calendar,
8           meaning there are 14 configurations in which the days of
9           the week occupy the same numerical days in a year. The way
10          to distinguish between unique years is whether it is a leap
11          year and what day of the week it starts with.
12
13          :param this.yearLookup:
```

```
14              This is a dictionary whose key values represent the
15              days of the week (0 = saturday). Their corresponding
16              values are the number of sundays that fall on the first
17              of each month in that year.
18          :param this.leapYearLookup:
19              This is a dictionary whose key values represent the
20              days of the week (0 = saturday). Their corresponding
21              values are the number of sundays that fall on the first
22              of each month in that leap year.
23          :param this.yearLookupIndex:
24              The index into each yearLookup dictionary. The value
25              of this variable corresponds to the first day in a year
26              (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.
34          :param this.endYear:
35              The year to end calculations on.
36              '''
37
38          this.yearLookup = {0: 1, 1: 2, 2: 2, 3: 2, 4: 1, 5: 3, 6: 1}
39          this.leapYearLookup = {0: 1, 1: 3, 2: 2, 3: 1, 4: 2, 5: 2, 6: 1}
40          this.yearLookupIndex = 3
41          this.sizeOfWeek = 7
42          this.sundayCount = 0
43          this.currentYear = 1901
44          this.endYear = 2001
45
46
47      def calculateSundaysInYear(this) -> None:
48          if this.checkLeapYear():
49              this.sundayCount += this.leapYearLookup[this.yearLookupIndex]
50              this.yearLookupIndex += 2
51          else:
```

```
52            this.sundayCount += this.yearLookup[this.yearLookupIndex]
53            this.yearLookupIndex += 1
54
55
56    def updateLookupIndex(this) -> None:
57        '''If the lookup index, which corresponds to the days
58        of the week, exceeds the number of days in a week it wraps
59        around the week again.'''
60        this.yearLookupIndex %= this.sizeOfWeek
61
62
63    def checkLeapYear(this) -> bool:
64        return (this.currentYear % 4 == 0) and (this.currentYear % 100 != 0 or
65        this.currentYear % 400 == 0)
66
67
68    def incrementYear(this, step: int = 1) -> None:
69        this.currentYear += step
70
71
72    def calculateSolution(this):
73        '''calculates all days that fall on the first of a month
74        between this.currentYear and this.endYear'''
75
76        while this.currentYear < this.endYear:
77
78            this.calculateSundaysInYear()
79
80            if this.yearLookupIndex >= this.sizeOfWeek:
81                this.updateLookupIndex()
82
83            this.incrementYear()
84
85        return this.sundayCount
86
87
88 if __name__ == "__main__":
89
```

```
90    startTime = time_ns()
91    archiveSolution = archive19()
92    noOfDays = archiveSolution.calculateSolution()
93    stopTime = time_ns()
94    print(noOfDays, "\nTime Taken (ns): " + str(stopTime - startTime))
```

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

5 The 28 And 400 Year Solar Cycle

5.1 A Simple(ish) Sequence

There is a property of the sequence of unique years in the Julian calendar - they have a period of 28 years. This is not completely accurate in the Gregorian calendar, as years that are divisible by 100 but not 400 (1900, 1700, etc) disrupt this cycle. Despite these 'disruptive' years the exact sequence of unique years repeats every 400 years, hence the 400 year Gregorian solar 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 starting from January 1st 1901.

The number of Sundays that fall on the first of each month in this 28 year cycle is exactly 48 (calculated with the help of our algorithm in Section 4.3). So if we take the quotient of $\frac{100}{28}$ and multiply it by 48 we will get the the number of Sundays that fall on the first of a month within $n - (n \mod 28)$ years. Then we can calculate the remaining years within the last 28 year cycle using our lookup algorithm in Section 4.3.

5.2 The Algorithm

```
1  from time import time_ns
2
3  class archive19:
4
5
6      def __init__(this):
7          '''Their are 14 unique years in the Gregorian calendar,
8          meaning there are 14 configurations in which the days of
9          the week occupy the same numerical days in a year. The way
```

```
10          to distinguish between unique years is whether it is a leap
11          year and what day of the week it starts with.
12
13          :param this.yearLookup:
14              This is a dictionary whose key values represent the
15              days of the week (0 = saturday). Their corresponding
16              values are the number of sundays that fall on the first
17              of each month in that year.
18          :param this.leapYearLookup:
19              This is a dictionary whose key values represent the
20              days of the week (0 = saturday). Their corresponding
21              values are the number of sundays that fall on the first
22              of each month in that leap year.
23          :param this.yearLookupIndex:
24              The index into each yearLookup dictionary. The value
25              of this variable corresponds to the first day in a year
26              (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)
65          this.quotient = numberOfYears // 28
66          this.currentYear += (this.quotient * 28)
67          this.quotient *= 48
68
69
70      def updateLookupIndex(this) -> None:
71          '''If the lookup index, which corresponds to the days
72          of the week, exceeds the number of days in a week it wraps
73          around the week again.'''
74          this.yearLookupIndex %= this.sizeOfWeek
75
76
77      def checkLeapYear(this) -> bool:
78          return (this.currentYear % 4 == 0) and (this.currentYear % 100
79          != 0 or this.currentYear % 400 == 0)
80
81
82      def incrementYear(this, step: int = 1) -> None:
83          this.currentYear += step
84
85
```

```
86    def calculateSolution(this):
87        '''calculates all days that fall on the first of a month
88        between this.currentYear and this.endYear'''
89
90        this.getQuotientResult()
91
92        while this.currentYear < this.endYear:
93
94            this.calculateSundaysInYear()
95
96            if this.yearLookupIndex >= this.sizeOfWeek:
97                this.updateLookupIndex()
98
99            this.incrementYear()
100
101        return this.sundayCount + this.quotient
102
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, so the number of calculations has a very low upper limit. Note: because this calculates the 28 year cycle instead of the 400 year cycle, it will not work across 'disruptive' years (1900, 1700, 2100, etc).

6   Bibliography

References

[1] C. Zeller, "Die grundaufgaben der kalenderrechnung auf neue und vereinfachte weise gelöst," German, *Württembergische Vierteljahrshefte für Landesgeschichte*, vol. 5, pp. 313–314, 1882. DOI: 10.53458/wvjh.v5i.14389.

[2] R. Sivaraman, "Determining day of given date mathematically," *Mathematics and Statostics*, vol. 8, no. 5, pp. 590–595, 2020. DOI: 10.13189/ms.2020.080514.

[3] J. C. Kolecki, D. Mazza, and T. M. Scott. "Calendar calculations." (2001), [Online]. Available: https://www.grc.nasa.gov/www/k-12/Numbers/Math/Mathematical_Thinking/calendar_calculations.htm. (accessed: 07.08.2025).