Testing the Birthday Problem Experimentally

Eliot Ball

December 2, 2010

1 Introduction

> "It doesn't matter how beautiful your theory is, it doesn't matter how smart you are. If it doesn't agree with experiment, it's wrong." - Richard Feynman

The Birthday Problem is the question in probability of wether, in a given group of people, one or more people share the same birthday. It is considered obvious that with a group of 366 people (or 367 in a leap year) two people are gauranteed to have the same birthday, because there are not enough days for each person to have their own. Somewhat counterintuitively, however, the probability is 99% with just 57 people, a group size not uncommon in real life.

It is my aim to use simple computer scripts written in the Python programming language to generate a large number of example cases and demonstrate that the mathematics behind the result of the birthday problem correlates with reality.

2 The Mathematics Behind the Birthday Problem

Firstly the problem should be defined in an unambiguous way. The length of a year is taken to be 365 days exactly, and it is taken that, for each person, they are equally likely to be born on each day of the year. Furthermore, the birthdays of each person are taken to be independent from any other persons. Factors such as twins are ignored for the purpose of the birthday problem.

It is suggested that the best way to tackle the problem is by finding the probability that none of the people in the room share the same birthday, as this may be simpler and will by definition yield the probability of two people sharing a birthday.

In the case of a group of one person, the probability of sharing a birthday is evidently 0, as there is no second person with whom to share a birthday. The probability for two people is also simple to deduce. The first person takes one day, leaving 364 free for another person to have a different birthday. The probability of selecting one of these days at random is therefore $\frac{364}{365}$. Continuing with this process gives 363 days available for the third person to choose, and hence a probability of $\frac{363}{365}$ of them achieving this at random. Generally then, where $\bar{B}(n)$ is the probability of two people in a group of n people not sharing a birthday:

$$\bar{B}(n) = \frac{366-1}{365} \times \frac{366-2}{365} \times \frac{366-3}{365} \times \cdots \times \frac{366-n}{365}$$

By mulitplying the individual probabilities it can be seen that

$$\bar{B}(n) = \frac{365 \times 364 \times 363 \times \cdots \times (366-n)}{365^n}$$

Noting that the top part of the fraction is a factorial sequence, we can simplify to

$$\bar{B}(n) = \frac{365!/(365-n)!}{365^n}$$

By subtracting the proabability of no two people sharing the same birthday we arrive at the probability of two people in a given group of size n sharing the same birthday, $B(n)$

$$\bar{B}(n) = \frac{365!}{365^n(365-n)!} \Rightarrow B(n) = 1 - \frac{365!}{365^n(365-n)!}$$

This formula works fine for values of n up to and including 365, but testing for 366 people yields a division by comlex infinity, which is unpleasant to deal with. Thankfully, the formula can be manipulated to provide

a more usable version. Multiplying the numerator and denominator of the fraction by $n!$ and splitting it up allows the binomial coefficient to be recognised:

$$\begin{aligned} B(n) &= 1 - \frac{n! \times 365!}{365^n \times n! \times (365-n)!} \\ B(n) &= 1 - \frac{n!}{365^n} \times \frac{365!}{n! \times (365-n)!} \\ B(n) &= 1 - \frac{n!}{365!}\binom{365}{n} \end{aligned}$$

This formula returns the same result but $n > 365$ can be handled without infinity becoming involved.

3 Implementing a Simulation

The approach used will be that of generating a large number of example groups for each group size, and checking how many of these have shared birthdays. In order to achieve reasonable reliability, the number of groups required for each size will be several thousand.

3.1 Choice of Programming Language

The Python programming language will be used because of its easily readable and understandable syntax. A disadvantage of this language, for a simulation such as this, is that, due to it being an interpreted language, it is not as fast as a natively compiled language such as C, although the speed difference should not be dramatic enough to make running the simulation in Python impractical, especially if the necessary steps are taken to optimise the Python source code.

3.2 Random Number Generation

Random number generation will be handled by the included Python module `random`. This uses a Mersenne Twister algorithm for generating pseudorandom numbers. This algorithm is widely recognised as one of the best PRNGs available. The generator needs a seed value, and the system clock is used by default. The first two lines of the simulation program take care of this:

```
import random
random.seed()
```

All of the random numbers needed by this program will be integers $1 \leq n \leq 365$ (to represent a day of the year) so it will be useful to have a function that allows us to easily obtain these later on in the program:

```
def RandDay():
  return random.randint(1, 365)
```

This function calls the `randint(lower, upper)` function from the `random` module with the necessary values for a day of the year and returns the result. The bounds are inclusive for this function.

3.3 Generating a Random Group of n People

The next function needed is one that generates a group of n people, all with random birthdays. In fact, the only thing being generated is n random birthdays; no other information about each person is needed for the purposed of the simluation. The function looks like this:

```
def MakeGroup(n):
  group = []
  for i in range(n):
    group.append(RandDay())
  return group
```

This function takes one parameter n, the number of people to be included in the group. It creates an empty list and then iterates through the number of people, generating a random day and appending this to the list. At the end, the list is returned.

3.4 Checking a Group for Paired Birthdays

The most important and complicated function is the one to check a group for any repeated birthdays.

```
def CheckForPair(group):
  for a in range(len(group)):
    for b in range(a + 1, len(group)):
      if group[a] == group[b]:
        return True
  return False
```

This function iterates over the whole group, and for each member of the group, it iterates over all of the members after the current member, checking if any have the same birthday value. If one is found, the function is stopped and the value `True` is returned. If the end of the function is reached then no pair has been found, and the value `False` is returned.

3.5 Preparing the Simulation Code

The simuation code is rather simple. For each group size $1 < n \leq 365$ twenty-five thousand random example groups will be generated and tested for two people sharing birthdays. The number of groups with two people sharing birthdays will be counted as they are found and then a percentage will be calculated. This should apporoximately replicate the results found using the probability function found above. The results will be output as comma separated values to allow easy manipulation in Excel afterward, and three values will be presented: the size n, the number of groups out of 25000 that contained paired birthdays, and the time taken for the simulation of that group size, in seconds.

```
import time

for n in range(2, 366):
  starttime = time.clock()
  count = 0
    for i in range(25000):
      if CheckForPair(MakeGroup(n)) == True:
        count += 1
        print str(n) + "," + str(count) + "," + str(round(time.clock() - starttime, 2))
```

The results from this were copied into a CSV document which was opened in Excel and an XLSX file was then saved. The `time` module was used to determine how long each simulation took.

3.6 Running the Simulation

The simulation ran for 1 hour, 7 minutes and 6.11 seconds. The results were provided in the format expected, and there were no problems with the environment such as memory issues, the operating system hanging or crashing, or the simulation pausing when the computer went into standby mode. The very largest group sizes incurred about twenty seconds of computation for all 25000 groups, while the first several group size simulations were completed in well under a second each.

4 Analysing the Results

The first step in analysing the results is to derive some more data from the data obtained through the simulation. The most important of these is the value of the proportion of the groups for a given size n that contained pairs of birthdays. This value is gained by simply dividing the number of groups by 25000. The second value is the expected proportion, by the formula shown previously. Neither Excel nor Python were able to deal with the large integers involved in the factorial functions, so an approximation formula had to be used:

$$B(n) = 1 - e^{-n(n-1)/(2\times 365)}$$

This approximation relies on the Taylor Series and the inaccuracy is negligible for the purposes of validating our results. The Excel formula for this:

```
=1-EXP(-A2*(A2-1)/(2*365))
```

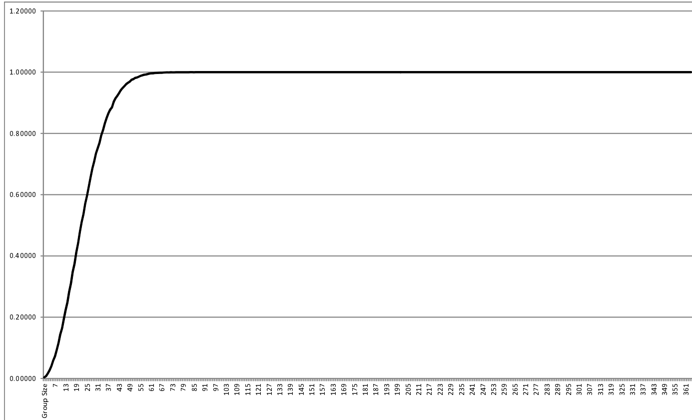
With this the absolute value of the difference was calculated to give an idea of how close the simulation was to the real probabilities.

4.1 The Results

The results table is very large so I have not included it. It can be obtained here:

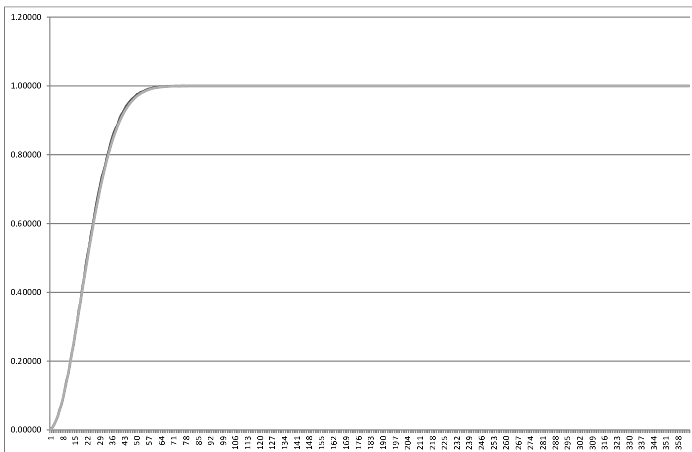
http://eliotball.com/files/BirthdayProblemResults.xlsx (Microsoft Office Excel 2007 or later will be required to view this file)

4.2 Relationship Between Group Size and Proportion of Paired Birthdays



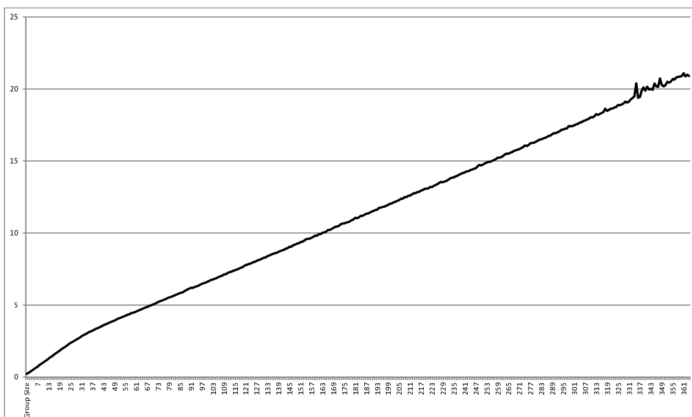
Predictably, the proportion of groups containing shared birthdays grows with group size, but it is striking from the graph how quickly it grows, reaching almost certainty at around 60 members.

4.3 Similarity of Experiment Results to Expected Probability



The experimentally obtained proportion of groups with pairs and the expected probability given by the approximate formula above agree very strongly, as is evident from the graph. Therefore we can conclude that the mathematics behind the Birthday Problem are accurate and reliable, if a little unbelievable. The average difference between the approximate formula and the result of each simulation was just 0.00091.

4.4 Running Time of Each Group Size Simulation



The growth of the running time with group size is broadly linear, suggesting that the computational complexities of the functions to generate and check each group are all $O(n)$. While this seems obvious for the random group generation, it seems a little counter-intuitive for the function to check if there are pairs in a group. A little investigation is in order:

Let $D(g)$ be true if the set g contains any duplicates.

For each element of g we check each of the subsequent elements to see if they are equal.

Therefore for different sizes of g:

$$
\begin{aligned}
D(1) &= \bot \\
D(2) &= g_0 \leftrightarrow g_1 \\
D(3) &= (g_0 \leftrightarrow g_1 \vee g_0 \leftrightarrow g_2) \vee (g_1 \leftrightarrow g_2) \\
D(4) &= (g_0 \leftrightarrow g_1 \vee g_0 \leftrightarrow g_2 \vee g_0 \leftrightarrow g_3) \vee (g_1 \leftrightarrow g_2 \vee g_1 \leftrightarrow g_3) \vee (g_2 \leftrightarrow g_3)
\end{aligned}
$$

The number of comparisons can be seen to have this closed form, where $S(n)$ is the number of comparisons taken for a group of size n:

$$S(n) = \frac{n^2 - n}{2}$$

This shows that the computational complexity of the function `CheckForPair` is $O(n^2)$. It seems reasonable to conclude, then, that the time taken for random number generation, which grows linearly, is so much greater than the time taken for comparison, that the exponential growth of checking for paired birthdays is not apparent in the graph.

The fluctuation at the end of the program is likely to be due to the PC being used for other tasks during this time, while it had been left to work soley on the simulation for about an hour previous to this.

5 The Full Program Source Code

```
import random
random.seed()

def RandDay():
  return random.randint(1, 365)

def MakeGroup(n):
  group = []
  for i in range(n):
    group.append(RandDay())
  return group

def CheckForPair(group):
  for a in range(len(group)):
    for b in range(a + 1, len(group)):
      if group[a] == group[b]:
        return Truer
  return False

import time

for n in range(2, 366):
  starttime = time.clock()
  count = 0
  for i in range(25000):
    if CheckForPair(MakeGroup(n)) == True:
      count += 1
      print str(n) + "," + str(count) + "," + str(round(time.clock() - starttime, 2))
```

6 The Output of the Simulation

The output data has been reformatted to allow it to be reprinted in a reasonable amount of space.

(2, 83, 0.23) (3, 185, 0.3) (4, 406, 0.38) (5, 671, 0.47) (6, 1008, 0.56) (7, 1457, 0.65) (8, 1811, 0.73) (9, 2321, 0.85) (10, 2906, 0.92) (11, 3581, 1.01) (12, 4098, 1.1) (13, 4865, 1.19) (14, 5576, 1.29) (15, 6209, 1.38) (16, 7067, 1.46) (17, 7761, 1.57) (18, 8697, 1.66) (19, 9349, 1.75) (20, 10303, 1.83) (21, 11040, 1.93) (22, 11989, 2.02) (23, 12767, 2.1) (24, 13422, 2.19) (25, 14264, 2.28) (26, 14914, 2.37) (27, 15654, 2.44) (28, 16429, 2.51) (29, 17151, 2.59) (30, 17693, 2.66) (31, 18368, 2.74) (32, 18787, 2.83) (33, 19233, 2.91) (34, 19837, 2.97) (35, 20280, 3.03) (36, 20831, 3.12) (37, 21251, 3.16) (38, 21645, 3.23) (39, 21944, 3.29) (40, 22133, 3.36) (41, 22581, 3.4) (42, 22868, 3.47) (43, 23081, 3.52) (44, 23301, 3.6) (45, 23549, 3.65) (46, 23711, 3.7) (47, 23870, 3.76) (48, 24024, 3.82) (49, 24138, 3.87) (50, 24228, 3.92) (51, 24383, 3.98) (52, 24432, 4.06) (53, 24536, 4.1) (54, 24562, 4.15) (55, 24626, 4.2) (56, 24697, 4.25) (57, 24742, 4.32) (58, 24795, 4.35) (59, 24810, 4.43) (60, 24844, 4.46) (61, 24885, 4.5) (62, 24913, 4.55) (63, 24907, 4.61) (64, 24935, 4.67) (65, 24944, 4.72) (66, 24950, 4.77) (67, 24956, 4.83) (68, 24959, 4.87) (69, 24974, 4.94) (70, 24981, 4.98) (71, 24984, 5.03) (72, 24980, 5.08) (73, 24993, 5.14) (74, 24991, 5.21) (75, 24984, 5.27) (76, 24995, 5.31) (77, 24998, 5.36) (78, 24996, 5.41) (79, 24995, 5.47) (80, 24998, 5.53) (81, 24998, 5.57) (82, 24998, 5.62) (83, 24999, 5.67) (84, 25000, 5.73) (85, 25000, 5.78) (86, 24998, 5.84) (87, 25000, 5.86) (88, 25000, 5.93) (89, 25000, 6.01) (90, 25000, 6.08) (91, 25000, 6.13) (92, 25000, 6.2) (93, 25000, 6.19) (94, 25000, 6.26) (95, 25000, 6.3) (96, 25000, 6.34) (97, 25000, 6.41) (98, 25000, 6.48) (99, 25000, 6.52) (100, 25000, 6.56) (101, 25000, 6.62) (102, 25000, 6.67) (103, 25000, 6.75) (104, 25000, 6.77) (105, 25000, 6.83) (106, 25000, 6.87) (107, 25000, 6.94) (108, 25000, 7.0) (109, 25000, 7.03) (110, 25000, 7.12) (111, 25000, 7.14) (112, 25000, 7.22) (113, 25000, 7.28) (114, 25000, 7.31) (115, 25000, 7.37) (116, 25000, 7.41) (117, 25000, 7.46) (118, 25000, 7.51) (119, 25000, 7.58) (120, 25000, 7.62) (121, 25000, 7.7) (122, 25000, 7.78) (123, 25000, 7.81) (124, 25000, 7.87) (125, 25000, 7.89) (126, 25000, 7.97) (127, 25000, 8.0) (128, 25000, 8.07) (129, 25000, 8.13) (130, 25000, 8.16) (131, 25000, 8.23) (132, 25000, 8.29) (133, 25000, 8.31) (134, 25000, 8.4) (135, 25000, 8.44) (136, 25000, 8.51) (137, 25000, 8.55) (138, 25000, 8.6) (139, 25000, 8.63) (140, 25000, 8.7) (141, 25000, 8.75) (142, 25000, 8.8) (143, 25000, 8.84) (144, 25000, 8.92) (145, 25000, 8.95) (146, 25000, 9.05) (147, 25000, 9.05) (148, 25000, 9.16) (149, 25000, 9.19) (150, 25000, 9.26) (151, 25000, 9.29) (152, 25000, 9.37) (153, 25000, 9.4) (154, 25000, 9.46) (155, 25000, 9.57) (156, 25000, 9.59) (157, 25000, 9.61) (158, 25000, 9.67) (159, 25000, 9.73) (160, 25000, 9.82) (161, 25000, 9.81) (162, 25000, 9.92) (163, 25000, 9.93) (164, 25000, 10.01) (165, 25000, 10.04) (166, 25000, 10.09) (167, 25000, 10.21) (168, 25000, 10.21) (169, 25000, 10.28) (170, 25000, 10.35) (171, 25000, 10.43) (172, 25000, 10.45) (173, 25000, 10.49) (174, 25000, 10.6) (175, 25000, 10.66) (176, 25000, 10.67) (177, 25000, 10.72) (178, 25000, 10.75) (179, 25000, 10.8) (180, 25000, 10.89) (181, 25000, 10.94) (182, 25000, 11.06) (183, 25000, 11.03) (184, 25000, 11.08) (185, 25000, 11.19) (186, 25000, 11.19) (187, 25000, 11.26) (188, 25000, 11.34) (189, 25000, 11.36) (190, 25000, 11.41) (191, 25000, 11.49) (192, 25000, 11.53) (193, 25000, 11.6) (194, 25000, 11.62) (195, 25000, 11.75) (196, 25000, 11.76) (197, 25000, 11.81) (198, 25000, 11.83) (199, 25000, 11.9) (200, 25000, 11.94) (201, 25000, 12.03) (202, 25000, 12.05) (203, 25000, 12.13) (204, 25000, 12.17) (205, 25000, 12.23) (206, 25000, 12.28) (207, 25000, 12.38) (208, 25000, 12.38) (209, 25000, 12.49) (210, 25000, 12.49) (211, 25000, 12.58) (212, 25000, 12.6) (213, 25000, 12.67) (214, 25000, 12.76) (215, 25000, 12.77) (216, 25000, 12.84) (217, 25000, 12.86) (218, 25000, 12.94) (219, 25000, 12.99) (220, 25000, 13.06) (221, 25000, 13.08) (222, 25000, 13.09) (223, 25000, 13.19) (224, 25000, 13.19) (225, 25000, 13.27) (226, 25000, 13.33) (227, 25000, 13.39) (228, 25000, 13.48) (229, 25000, 13.55) (230, 25000, 13.52) (231, 25000, 13.58) (232, 25000, 13.62) (233, 25000, 13.69) (234, 25000, 13.8) (235, 25000, 13.84) (236, 25000, 13.86) (237, 25000, 13.93) (238, 25000, 13.98) (239, 25000, 14.05) (240, 25000, 14.11) (241, 25000, 14.17) (242, 25000, 14.2) (243, 25000, 14.28) (244, 25000, 14.29) (245, 25000, 14.36) (246, 25000, 14.4) (247, 25000, 14.46) (248, 25000, 14.49) (249, 25000, 14.62) (250, 25000, 14.73) (251, 25000, 14.69) (252, 25000, 14.75) (253, 25000, 14.83) (254, 25000, 14.89) (255, 25000, 14.94) (256, 25000, 14.94) (257, 25000, 15.01) (258, 25000, 15.06) (259, 25000, 15.12) (260, 25000, 15.23) (261, 25000, 15.23) (262, 25000, 15.27) (263, 25000, 15.34) (264, 25000, 15.46) (265, 25000, 15.51) (266, 25000, 15.5) (267, 25000, 15.58) (268, 25000, 15.61) (269, 25000, 15.7) (270, 25000, 15.74) (271, 25000, 15.8) (272, 25000, 15.82) (273, 25000, 15.9) (274, 25000, 15.96) (275, 25000, 16.08) (276, 25000, 16.04) (277, 25000, 16.11) (278, 25000, 16.24) (279, 25000, 16.25) (280, 25000, 16.27) (281, 25000, 16.34) (282, 25000, 16.41) (283, 25000, 16.48) (284, 25000, 16.51) (285, 25000, 16.56) (286, 25000, 16.61) (287, 25000, 16.66) (288, 25000, 16.74) (289, 25000, 16.77) (290, 25000, 16.87) (291, 25000, 16.93) (292, 25000, 16.93) (293, 25000, 17.0) (294, 25000, 17.06) (295, 25000, 17.17) (296, 25000, 17.18) (297, 25000, 17.25) (298, 25000, 17.25) (299, 25000, 17.43) (300, 25000, 17.41) (301, 25000, 17.43) (302, 25000, 17.47) (303, 25000, 17.54) (304, 25000, 17.58) (305, 25000, 17.66) (306, 25000, 17.69) (307, 25000, 17.78) (308, 25000, 17.81) (309, 25000, 17.88) (310, 25000, 17.92) (311, 25000, 18.03) (312, 25000, 18.02) (313, 25000, 18.09) (314, 25000, 18.25) (315, 25000, 18.2) (316, 25000, 18.26) (317, 25000, 18.33) (318, 25000, 18.39) (319, 25000, 18.63) (320, 25000, 18.48) (321, 25000, 18.54) (322, 25000, 18.63) (323, 25000, 18.64) (324, 25000, 18.71) (325, 25000, 18.74) (326, 25000, 18.89) (327, 25000, 18.87) (328, 25000, 18.92) (329, 25000, 18.99) (330, 25000, 19.12) (331, 25000, 19.06) (332, 25000, 19.12) (333, 25000, 19.28) (334, 25000, 19.37) (335, 25000, 19.49) (336, 25000, 20.39) (337, 25000, 19.39) (338, 25000, 19.45) (339, 25000, 19.89) (340, 25000, 20.11) (341, 25000, 19.88) (342, 25000, 20.16) (343, 25000, 19.96) (344, 25000, 20.01) (345, 25000, 19.94) (346, 25000, 20.38) (347, 25000, 20.18) (348, 25000, 20.15) (349, 25000, 20.73) (350, 25000, 20.29) (351, 25000, 20.18) (352, 25000, 20.26) (353, 25000, 20.5) (354, 25000, 20.44) (355, 25000, 20.51) (356, 25000, 20.69) (357, 25000, 20.65) (358, 25000, 20.81) (359, 25000, 20.84) (360, 25000, 20.86) (361, 25000, 20.9) (362, 25000, 21.09) (363, 25000, 20.87) (364, 25000, 20.99) (365, 25000, 20.9)