

# USING RECURSION TO SOLVE THE PILL PROBLEM

*Keith Brandt*  
*Mathematics and Physics*  
*Rockhurst University*  
*Kansas City, MO 64110*  
*keith.brandt@rockhurst.edu*

*Kaleb Waite*  
*Mathematics and Physics*  
*Rockhurst University*  
*Kansas City, MO 64110*  
*waitek@rockhurst.edu*

## ABSTRACT

A probability question regarding the distribution of whole and half pills in a bottle is studied. Four different recursive implementations are used to investigate this question, and the performance of each implementation is measured in terms of the trade-off between computing time and storage required.

## INTRODUCTION TO PROBLEM

Suppose you have a bottle of whole pills and you would like a daily dose of half a pill. Each day, remove a pill from the bottle. If it is a whole pill, break it in half, swallow one half, and return the other half to the bottle. If it is a half pill, simply swallow it. Over time, you will have a mix of whole and half pills in the bottle. This note addresses the following question: On the  $n$ -th day, what is the probability that the pill you remove from the bottle is whole?

A recurrence relation for the pill problem is described and implemented. Simply applying the recurrence relation is quite easy, but the many calculations—with all their redundancies—allow the problem to be solved for only very small values of  $n$ . Redundant calculations can be avoided by storing computed values and retrieving them when necessary. This paper explores a few different data structures as storage devices and discusses their strengths and weaknesses in solving the pill problem. The material in this note is accessible to anyone who is familiar with recursion and standard data structures such as arrays, lists, and trees. (For background, see [1] and [2].) It can be used for demonstrations and student projects in classes such as data structures, computer algorithms, and discrete mathematics.

## The Recurrence Relation

Consider a bottle that contains  $w$  whole pills and  $h$  half pills. Define  $P_n(w,h)$  to be the probability that the pill removed on day  $n$  is whole. If there are no whole pills in the bottle, then  $P_n$  would be zero for all  $n$ . If there are no half pills in the bottle, the first pill chosen will be whole, so consider the bottle on the next day with one fewer whole pill and one half pill. To determine  $P_1$ , the probability on the first day, compare the number of whole pills to the total number of pills in the bottle. In general, what happens on one day is given in terms of what happened on the previous day. The function  $P_n(w,h)$  is described by the recurrence relation

$$P_n(0, h) = 0$$

$$P_1(w, h) = \frac{w}{w+h}$$

$$P_n(w, 0) = P_{n-1}(w-1, 1)$$

$$P_n(w, h) = \left(\frac{w}{w+h}\right)P_{n-1}(w-1, h+1) + \left(\frac{h}{w+h}\right)P_{n-1}(w, h-1).$$

For example, to calculate the probability of getting a whole pill on the third day when the initial number of pills are 2 whole and 1 half, proceed as follows:

$$P_3(2,1) = \frac{2}{3}P_2(1,2) + \frac{1}{3}P_2(2,0) = \frac{2}{3}\left(\frac{1}{3}P_1(0,3) + \frac{2}{3}P_1(1,1)\right) + \frac{1}{3}P_1(1,1) = \frac{2}{3}\left(0 + \frac{2}{3} \cdot \frac{1}{2}\right) + \frac{1}{3} \cdot \frac{1}{2} = \frac{7}{18}.$$

### The Pill Tree

A binary tree called the pill tree is used to help visualize the recursion. Its paths from root to leaf describe all possible configurations of pills in the bottle. The root of the pill tree is the ordered pair  $(w, h)$ . Its left subtree, which corresponds to the case when a whole pill is chosen, has root  $(w-1, h+1)$ . Its right subtree corresponds to the case when a half pill is chosen and has root  $(w, h-1)$ . In cases with only whole or half pills, no branching occurs and the node will have only one subtree. For example, the pill tree for 2 whole pills and 1 half pill is given below:

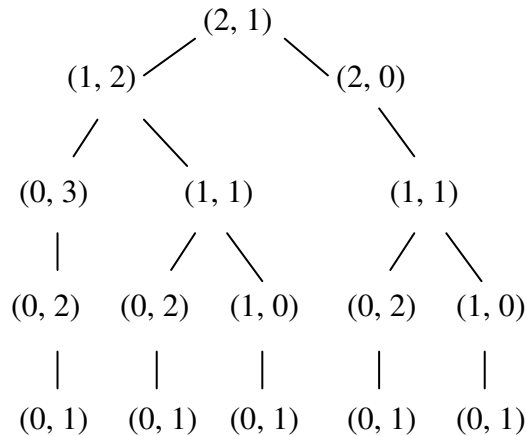


Figure 1-Example Pill Tree

### SOLUTION METHODOLOGY

A Java program is developed to explore the properties of this recursive function. The program inputs the initial whole and half pill counts and outputs the probability that a whole pill will be removed from the bottle on each successive day. It also outputs the time it takes to complete the computation and the number of times the recursive function is called. The program went through four main phases of development. Each phase was increasingly effective at solving the pill problem in the “reasonable timeframe” for computation, which was defined as a maximum of 12 hours on the testing computer (a Dell Inspiron 6000 with a 1.6 Ghz Pentium M processor and 512MB RAM).

## Pure Recursion

The first phase of development involves entering the recursive function as it is presented above without modification. Advantages to this solution are its simplicity and use of very little system memory (~5MB). The drawback of this method is that the time required to follow the recursion increases exponentially as the number of pills increases. Under this method, the program can solve the pill problem for up to 21 whole pills (and no half pills). This method demonstrates how essential it is to optimize algorithms that are computationally demanding. While the development of this solution method is easy to follow, it fails to be efficient.

## Array

The second phase of development is motivated by the poor performance of recursion alone. For example, in the simple case of starting with 2 whole pills and 1 half pill given in the introduction (Figure 1), all calculations corresponding to the subtree with root (1,1) would be repeated. A 3-dimensional array of doubles (floating point numbers) is used to store values once they are computed. Each time a probability value of the function for a given (day, whole, half) combination is computed, it is stored in the array at position `Array[day][whole][half]`. Thus, when the function is called, it first checks the array to find if the value has already been computed. If so, the value is returned and further recursive calls are avoided.

With this approach, the time it takes to compute the probability values for all days drops to negligible amounts, but because arrays (especially multi-dimensional ones) require allocation of large amounts of RAM (memory), the method is confined by the maximum size of the array. The limits on the amount of RAM available to the program cap this method's capacity at 253 whole pills (with extended default Java heap size of 256 MB). This is the fastest method found because of the ease of array lookups, but 75-90% of the memory allocated for the array is not used. Only about 10-25% of all possible (day, whole, half) combinations are actually computed during execution. Although this approach is a significant improvement over the first, its inefficient use of memory suggests that other storage devices should be considered.

## List

The third phase of development integrates the space-saving concept of combining the location information (day, whole, half) and the probability  $P_{\text{day}}(\text{whole, half})$  into a single value so that storage and retrieval is based on one value rather than four and the problem of empty memory slots is avoided. This is accomplished by storing the following value in memory:

$$\text{Stored Value} = \text{day} * 1,000,000 + \text{whole} * 1,000 + \text{half} + P_{\text{day}}(\text{whole, half}).$$

The scope of this formula is limited to 3-digit inputs because, for example, if called with 1,000 half-pills, the program would interpret the input as 1 whole pill. The input to this method produces a number of the form `dddd,www,hhh.ppp`, where `dddd` is the day, `www` is the number of whole pills, `hhh` is the number of half pills, and `ppp` is the probability for that function call. As the program computes these values, they are added to an unsorted list, with each value added at the head of the list. When the program calls the recursive

function, it first searches through the list from head to tail to see if the value dddd,www,hhh (with the probability truncated) is present. As the number of functions computed increases, the search times also increase proportionally.

This implementation can solve the pill problem for roughly 400 initial whole pills. This method's main advantage is saving on RAM required, as the list implemented has very minimal memory wasted on overhead functionality, and all of the memory allocated is used. The drawback to this method is that while memory is being used efficiently, searching for values in the list is time consuming (since search times increase linearly as the number of pills increases).

## Tree

The final phase of development brought one more data structure into play in order to minimize the search times that limited the previous phase. A self-balancing binary search tree using Java's TreeMap class [3] was created to store the probability values indexed with an integer in the dddd,www,hhh format. Because the binary search tree allowed search times to be reduced from linear complexity to logarithmic complexity, the execution time of this method was much faster and successfully solved the largest possible problem under the current encoding scheme: 999 initial whole pills. This program gives the full solution—from day 1 until the pills run out—for this case in less than an hour. The method could also be extended to solve larger problems by using an expanded format to encode the day, whole, and half values. However, it was decided that 999 pills was more than sufficient for the original goal of computing the probability function for a realistic number of pills found in a bottle. The algorithm used in the final version of the program is shown in pseudocode below:

```
method Calculate (whole pills, half pills, day)
  SET DayWholeHalf to day*1,000,000 + whole*1,000+half
  IF DayWholeHalf is in tree
    RETURN probability at DayWholeHalf
  ELSEIF whole is 0
    RETURN 0
  ELSEIF day is 1
    RETURN whole/(whole+half)
  ELSEIF half is 0
    SET probability to CALL Calculate(whole-1,1,day-1)
  ELSE
    SET probability to whole/(whole+half) * CALL Calculate(whole-1,
    half+1,day-1) + half/(whole+half)* CALL Calculate(whole,half-1,day-1)
  ENDIF
  INSERT probability into tree with key DayWholeHalf
  RETURN probability
END METHOD
```

## RESULTS

Figure 2 describes the resultant shape of the probability curve with no initial half pills. As seen in the figure, the probabilities decrease over time as more of the whole

pills are consumed and finally drops to zero on the last day. The probabilities at the beginning of the trial are decreasing at a higher rate than those later because there are more whole pills available.

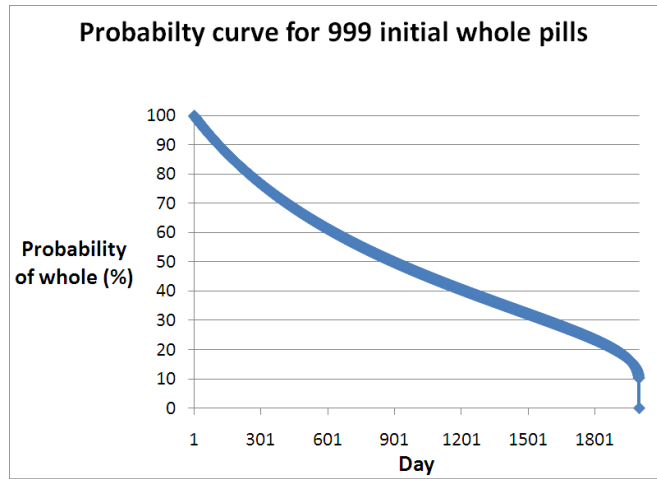


Figure 2-Probability Curve

Figure 3 compares the relative savings in overhead between the number of calls to the recursive function when no storage device is used and when the probabilities are stored. Both graphs represent the same initial condition of 21 whole pills. It takes about 150 billion function calls (and many hours of computation) to determine the probability on the last day without using data storage, while it takes about 450 function calls (and just seconds) to find the probability using the storage methods described in this paper.

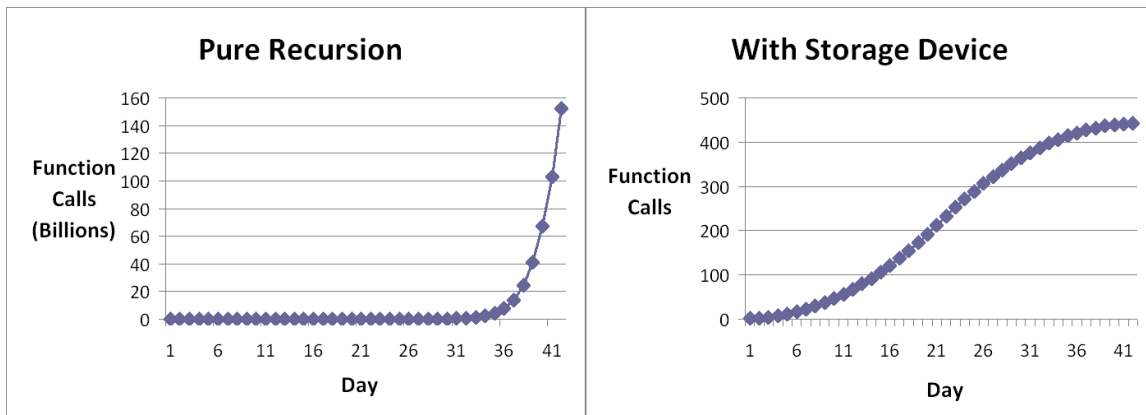


Figure 3-Comparison of Recursion Depths.

## CONCLUSIONS

The main theme of this work was to explore the properties of recurrence relations and to identify the trade-offs between storage limits and computation limits. In many applications, the more storage space is available, the less computational power is needed and vice-versa. Further exploration of this trade-off could be done by modifying the parameters of the initial project statement. For example, the pill could be split into three pieces instead of halves. Another possible topic of exploration would be to expand the coding scheme to allow for larger solutions (e.g. dddddd,www,hhhh). If the method

were to be expanded in this way, caution should be used to ensure that the primitive data type can store values in the trillions place. Also, non-recursive approaches could be considered. Ultimately, the pill problem holds much more potential for exploration and is suitable for a variety of student projects. Copies of all programs are available from the authors upon request.

## **REFERENCES**

- [1] Ford, W., Topp, W., *Data Structures with Java*, Upper Saddle River, NJ: Pearson Education, Inc., 2005.
- [2] Gersting, J., *Mathematical Structures for Computer Science: A Modern Treatment of Discrete Mathematics*, 5<sup>th</sup> ed., New York, NY: W. H. Freeman and Company, 2003.
- [3] Sun Microsystems Inc., Java 2 Platform Standard Edition 5.0 API Documentation, 1994, <http://java.sun.com/j2se/1.5.0/docs/api/java/util/TreeMap.html>, retrieved November 18, 2008.