Prime Digit Sum

Tae Young Kevin Shin

Wofford College  
429 North Church Street  
Spartanburg, SC 29303-3663  
+1 (864) 540-1327

shint@email.wofford.edu

**ABSTRACT**

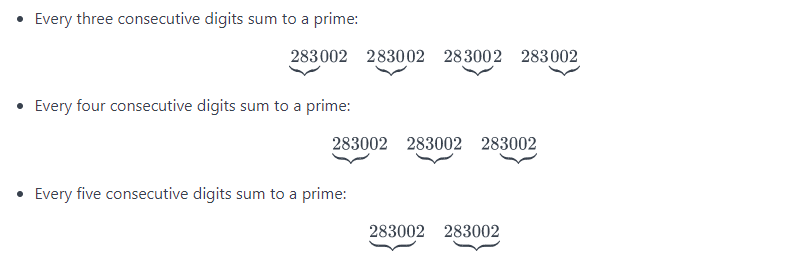
In this paper, we describe the dynamic programming and memoization algorithm to find prime digit sum.

**Keywords**

Brute Force, Dynamic Programming, Memoization,

# Introduction

In this paper, we come across any random number from which we seek to find whether the sum of three, four, and five consecutive digits of the given number respectively match with prime numbers. This rule is called **Chloe’s rules**. [2] Under *q* number of queries, each occurrence of query contains an integer value of *n* denoting the number of digits. With regard to constraints, 1≤ *q* ≤ 2 x 104, such that *q* represents the number of queries and 1≤ *n* ≤ 4 x 105, such that n represents the number of digits to be tested in each query. The final output from each query should be the number of units that satisfy **Chloe’s rules**. For example, according to the following picture, if the given n is 6, then, we are supposed to figure out whether all 6 digits positive integers have the following property. To be specific, we need to check whether sum of each three, four, and five consecutive digits is prime or not.



# Background

As this problem’s solution results from several different cases of functions, we should break down this problem into several small sub functions serving their own unique purposes, known as Dynamic Programming approach. In addition, as sub functions share the recurrent feature of calculating the sum of consecutive digits, we can also utilize the method called Memoization, an optimization technique used primarily to speed up computer programs by storing the results of repeated functions calls and returning cached result when the same inputs occur again.

# Algorithm

In general, there are eight major blocks of functions that we need to construct.

**3.1 is\_prime(int n)**

//input: integer value n

//output: True if n is prime, False otherwise.

//example: is\_prime(37) → True, is\_prime(80) → False

**3.2 split\_digit(n)**

//input: integer value n

//output: vector containing each digit of integer value n in a reverse order

//example: if n = 283002, then, it returns vector that holds {2,0,0,3,8,2}

**3.3 sum3\_prime(n)**

//input: integer value n

//output: return True if all three consecutive digit sum is prime, False otherwise.

//example: sum3\_prime(133) → True, sum3\_prime(1332) → False

**3.4 sum4\_prime(n)**

//input: integer value n

//output: return True if all four consecutive digit sum is prime, False otherwise.

//example: sum4\_prime(101101) → True, sum4\_prime(101122) → False

**3.5 sum5\_prime(n)**

//input: integer value n

//output: return True if all five consecutive digit sum is prime, False otherwise.

//example: sum5\_prime(101101) → True, sum5\_prime(101122) → False

**3.6 Chloe’s rules(d)**

//input: integer value d which stands for the number of digits that are to be tested

//output: return True if the number of d digits satisfies is prime, False otherwise.

//example: sum5\_prime(101101) → True, sum5\_prime(101122) → False

**3.7 main\_function()**

//input: number of queries, number of digits that are to be tested

//output: return number of cases which satisfy **Chole’s rules** on each query

//example: if q = 1, d = 6. Given the one number of query, there are 95 numbers of 6 digit numbers that satisfy sum3\_prime(n), sum4\_prime(n), sum5\_prime(n).

## Pseudocode

1 // Find the number of units whose all three, four, and five consecutive digits sum are prime numbers

2 // Input: first line <- the number of queries(*q*),

subsequent *q* lines <- the number of digits of input(*n*)

3 // Output: subsequent *q* lines <- the number of units that satisfy Chloe’s rules

4 **first function: is\_prime(n)**

5 n <- int(n)

6 if n = 0 or n = 1:

7 return False:

8 for i = 2 to sqrt(n) + 1

9 if n % i = 0:

10 return False

11 return True

12 temp = []

13 **second function: is\_three\_digits\_sum\_prime(n):**

14 for i = 0 to len(n-2)

15 seg <- int(n[i]) + int(n[i+1]) + int(n[i+2])

16 temp.append(seg)

17 if is\_prime(seg) = False:

18 return False

19 return True

20 **third function: is\_four\_digits\_sum\_prime(n):**

21 for i = 0 to len(n-3)

22 temp[i] = temp[i] + int(n[i+3])

23 if is\_prime(temp[i]) = False

24 return False

25 return True

26 **fourth function: is\_five\_digits\_sum\_prime(n):**

27 for i = 0 to len(n-4)

28 temp[i] = temp[i] + int(n[i+4])

29 if is\_prime(temp[i]) = False

30 return False

31 return True

32 **fifth function: free\_temp\_function(n):**

33 int \*temp = new temp[n];

34 delete [] temp;

35 **sixth function: final\_digits(n):**

36 lst = []

37 for i = 10n-1 to 10n -1

38 i = str(i)

39 x <- is\_three\_digits\_sum\_prime(i)

40 y <- is\_four\_digits\_sum\_prime(i)

41 z <- is\_five\_digits\_sum\_prime(i)

42 if x,y,z are all True:

43 lst.append(i)

44 delete temp

45 return len(lst)

46 **last function: main\_function()**

47 q <- first\_line\_input\_value(“Enter the number of queries: ”)

48 for i = 0 to q-1:

49 n <- input (“Enter the number of digits : ”)

50 return final\_digits(n)

## An Example

Let us suppose that we put **1 for *q* and 6 for *n***to begin with. This means that we want to see just one test query and hope to find all 6 digit numbers that satisfies **Chole’s rules.** In other words, for-loop under sixth function will iterate through 100000 to 999999.

Meanwhile, the number, **‘665002’**, for example, is the one that satisfies **Chloe’s rules.** First and foremost, as this number is 6 digits, we can apply this number into second, third, and fourth function which identifies whether all sums of consecutive three, four, and five digits are prime numbers.

In other words, if we only focus on three consecutive digit sum as segments, we can get **6+6+5**, **6+5+0**, **5+0+0**, and **0+0+2** sequentially, which are equivalent to **17, 11, 5, and 2** according to the algorithm from second function and first function. **17, 11, 5,** and **2** are stored in ***temp*** array to be utilized when third and fourth function is to be calculated. They are all prime numbers as seen. **Temp array = [17,11,5,2]**

With the same principle, **6+6+5+0, 6+5+0+0,** and **5+0+0+2**, which are **17, 11, and 7** according to the algorithm from third function. In fact, the first 17 (6+6+5+0) is calculated from adding the fourth index element of **0 to 6+6+5**, which is equivalent to index 3 of this number position. That is because, when we calculate the sum of consecutive four numbers, we already calculated first three on second function. Therefore, we dynamically keep those data as cache and utilize it efficiently rather than reiterating our unnecessary calculation. The ***temp*** array is to be updated now as **[17,11,7,2].** This rule applies for the rest of two results, **11 and 7.** It turns out that those three numbers, **17, 11, and 7** are all prime numbers as seen.

Last but not least, if we group five numbers sequentially, **6+6+5+0+0,** and **6+5+0+0+2** are 17 and 13 respectively, all of which are also prime numbers. First 17 is also calculated by adding fifth element of the given number **‘665002’** to **17** which is the first index element from the previous ***temp*** array. So does rest of the element in ***temp***array got updated with the same rule. So, ***temp*** array is to be updated as **[17,13,7,2]**. Therefore, **665002** does satisfy **Chole’s rules** as all elements in ***temp*** array are prime numbers. Finally, we need to delete and empty out all the elements of our ***temp*** array in order to test subsequent number within for-loop in **sixth function.**

## Time Complexity

When this program is prompted to run, seven different functions start to interactively run together in the form of bottom-up implementation. Even though each function takes part in their unique role and performance independently, they end up getting intertwined one another very dynamically. To be specific, every function from first one to fifth one which takes care of discerning whether the given number is a prime number or not or three, four, or five consecutive sums are prime or not has at least single for-loop which takes about **O(n),** a linear time. However, the true time complexity is contingent on the input query size *q* and input digit size *n* since the major portion of iteration takes place inside final\_digits function where the number of iteration is the difference between 10n-1 to 10n -1. Fortunately, since transitory elements in ***temp*** array help save unnecessary addition of consecutive digits, we can avoid exponentially large times of calculation. On each case of *n* digit number, all Chole’s sub functions from first to fifth are subordinately implemented. The final time complexity would be *q* times the difference between 10n-1 to 10n -1 times O(n), linear time, which ends up being expressed as **q times O(n2)** depending on the input size *n.*

# ConclusionS

Dynamic Programming is very powerful and systematic in terms of splitting one clustered schema into different roles of sub-functions. Memoization which temporarily stores data which was already calculated from previous implementation contributes to this entire algorithm’s process efficiently by diminishing unnecessary and repetitive addition. Brute Force algorithm approach from which two previous algorithm got derived suggests very straightforward and categorial strategy. However, in terms of space and time complexity, Brute Force algorithm in this case does not appear to be the best solution due to the length of implementation and space size taken from this approach.

# REFERENCES

1. Levitin Anany. 2012. *Introduction to The Design & Analysis of Algorithms*. 3rd ed. Boston: Pearson Education, 2012. Print.
2. Sogo, K. (n.d.). 2016. Prime Digit Sums. HackerRank. (September 2016). Retrieved May 10, 2018 from https://www.hackerrank.com/contests/world-codesprint-8/challenges/prime-digit-sums
3. Miller, Bradley M., and David L. Ranum. 4.12 Dynamic Programming – Problem Solving with Algorithms and Data Structures. Franklin Beedle, 22 Sept. 2013. Web.16.Apr 2018.http://interactivepython.org/runestone/static/pythonds/Recursion/DynamicProgramming.html
4. Saumont, Pierre-Yves. "Do it in Java 8: Automatic memoization - DZone Performance."Dzone.com. N.p., 30 Sept. 2014. Web. 16 Apr. 2018.