Lecture 1: Programming Skills

Yonghui Wu

Fudan University & Stony Brook SUNY

yhwu@fudan.edu.cn

- Fundamental Programming Skills
  - (1) Computing
  - (2) Simulation
  - (3) Recursion
- Calculation of High Precision Numbers

Compting

- Off-line Method
- Precision of Real Numbers & Improving Time Complexity by Dichotomy

Sum of Consecutive Prime Numbers

- **Source: ACM Japan 2005**
- **IDs for Online Judge: POJ 2739, UVA 3399**

- Some positive integers can be represented by a sum of one or more consecutive prime numbers. How many such representations does a given positive integer have? For example, the integer 53 has two representations 5 + 7 + 11 + 13 + 17 and 53. The integer 41 has three representations 2 + 3 + 5 + 7 + 11 + 13 , 11 + 13 + 17 , and 41 . The integer 3 has only one representation, which is 3. The integer 20 has no such representations. Note that summands must be consecutive prime numbers, so neither 7 + 13 nor 3 + 5 + 5 + 7 is a valid representation for the integer 20. Your mission is to write a program that reports the number of representations for the given positive integer.

- **Input**
- The input is a sequence of positive integers each in a separate line. The integers are between 2 and 10000, inclusive. The end of the input is indicated by a zero.

- **Output**
- The output should be composed of lines each corresponding to an input line except the last zero. An output line includes the number of representations for the input integer as the sum of one or more consecutive prime numbers. No other characters should be inserted in the output.

- **Sample Input**
- 2
- 3
- 17
- 41
- 20
- 666
- 12
- 53
- 0

- **Sample Output**
- 1
- 1
- 2
- 3
- 0
- 0
- 1
- 2

- Analysis
- Because the program need to deal with consecutive prime numbers in each test case, and the upper limit of prime numbers is 10000, off-line method can be used to solve the problem.

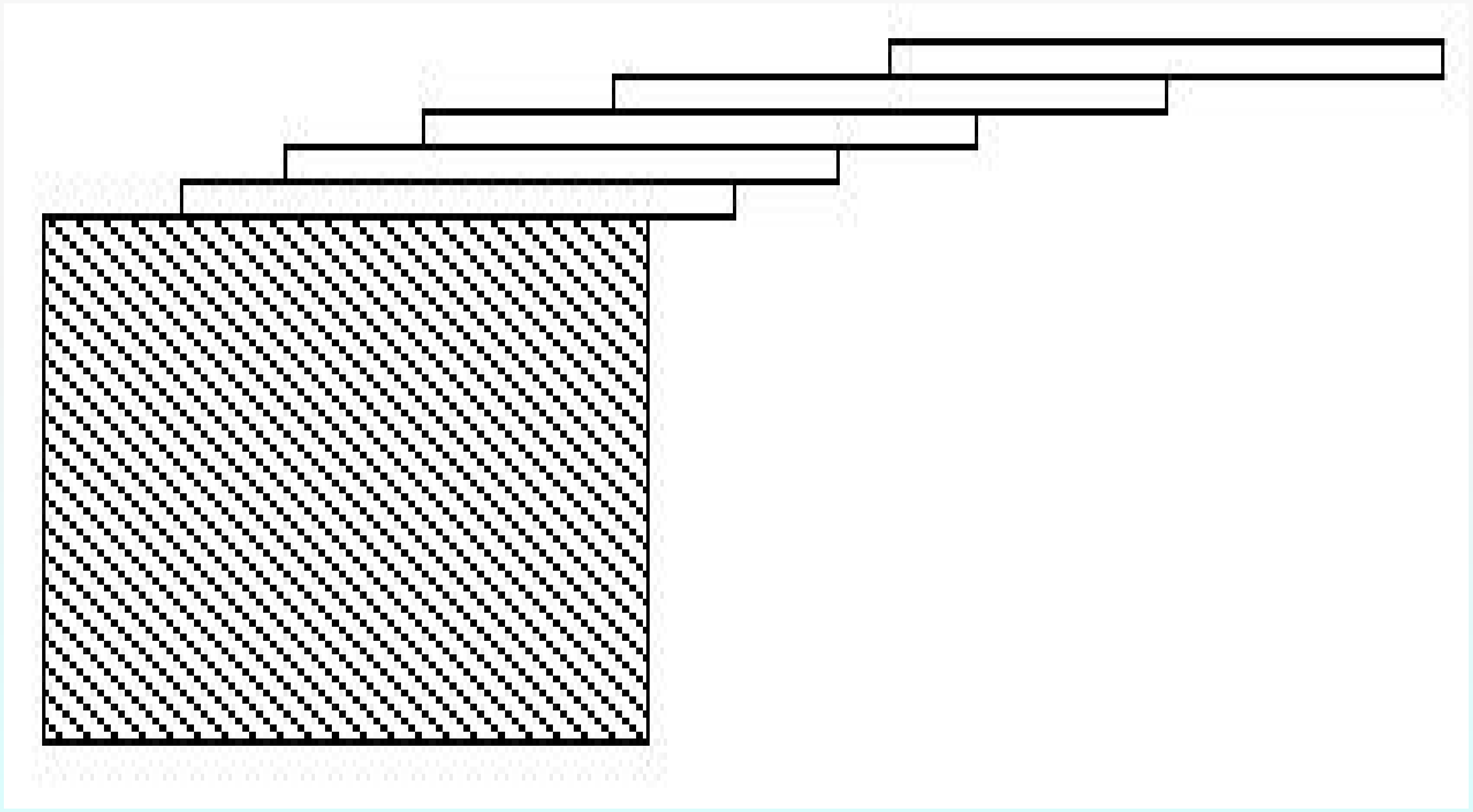
- Firstly, all prime numbers less than 10001 are gotten, and these prime numbers are stored in array $prime[1..\ total]$ in ascending order.

- Then we deal with test cases one by one:
  - Suppose the input number is *n*; the sum of consecutive prime numbers is *cnt*; the number of representations for *cnt=n* is *ans*.
  - A double loop is used to get the number of representations for *n*:
    - The outer loop *i*: for (int *i*=0; *n*>=*prime*[*i*]; *i*++) enumerates all possible minimum *prime*[*i*];
    - The inner loop *j*: for (int *j*=*i*; *j* < *total* && *cnt*<*n*; *j*++) *cnt* += *prime*[*j*]; to calculate the sum of consecutive prime numbers. If *cnt*≥*n*, then the loop ends; and if *cnt*==*n*, then the number of representations *ans*++.
  - When the outer loop ends, *ans* is the solution to the test case.

Hangover

- Source: ACM Mid-Central USA 2001
- IDs for Online Judge: POJ 1003, UVA 2294

- How far can you make a stack of cards overhang a table? If you have one card, you can create a maximum overhang of half a card length. (We're assuming that the cards must be perpendicular to the table.) With two cards you can make the top card overhang the bottom one by half a card length, and the bottom one overhang the table by a third of a card length, for a total maximum overhang of 1/2 + 1/3 = 5/6 card lengths. In general you can make n cards overhang by 1/2 + 1/3 + 1/4 + ... + 1/(*n* + 1) card lengths, where the top card overhangs the second by 1/2, the second overhangs tha third by 1/3, the third overhangs the fourth by 1/4, etc., and the bottom card overhangs the table by 1/(*n* + 1).

- This is illustrated in the figure below.
- 

- **Input**
- The input consists of one or more test cases, followed by a line containing the number 0.00 that signals the end of the input. Each test case is a single line containing a positive floating-point number *c* whose value is at least 0.01 and at most 5.20; *c* will contain exactly three digits.

- **Output**
- For each test case, output the minimum number of cards necessary to achieve an overhang of at least *c* card lengths. Use the exact output format shown in the examples.

- **Sample Input**
- 1.00
- 3.71
- 0.04
- 5.19
- 0.00

- **Sample Output**
- 3 card(s)
- 61 card(s)
- 1 card(s)
- 273 card(s)

- **Analysis**
- The problem's data area is little. Therefore firstly lengths that cards achieve are calculated, and the length is at most 5.20 card lengths. Suppose *total* is the number of cards, *len*[*i*] is the length that *i* cards achieve. That is, *len*[*i*] = *len*[*i* - 1] + 1/(*i*+1). Obviously array *len* is in ascending order.

- Because elements of *len* and *x* are real numbers, the accuracy error must be controlled. Suppose *delta*=1e-8, and function *zero*(*x*) marks *x* is a positive real number, a negative real number, or a zero, and is defined as follow:

$$zero(x) = \begin{cases} 1 & x > delte \\ -1 & x < -delte \\ 0 & otherwise \end{cases}$$

- Initially *len*[0]=0. Array *len* can be gotten through the following loop:
- for(*total*=1; *zero*(*len*[*total*-1]-5.20)<0; *total*++)
- *len*[*total*]=*len*[*total*-1]+1.0/double(*total*+1);

- After array *len* is gotten, the program inputs the first test data *x* and enters the loop of *while* (*zero*(*x*)). In each loop dichotomy is used to get the minimum number of cards necessary to achieve an overhang of at least *x* card lengths, and then the next test data *x* is inputted. The loop terminates when *x*=0 .00.

- The procedure of dichotomy is as follow:
- The initial interval [*l*, *r*]=[1, *total*] and

$$\min = \left\lfloor \frac{l+r}{2} \right\rfloor$$

- If *zero*(*len*[*mid*] - *x*) < 0, then search the right half (*l*=*mid*); otherwise search the left half (*r*=*mid*). Repeat above steps in interval [*l*, *r*] until *l*+1≥*r*. And *r* is the minimum number of cards.

Calculation of High Precision Numbers

- For programming languages, range and precision for integer and real are usually limited. If problem requires high precision more than the ranges, we can only solve such high precision problems by programming. For high precision number, there are two fundamental problems:
    - Representation of high precision numbers;
    - Fundamental calculations of high precision numbers;

Representation of high precision numbers

- A high precision number can be represented by an array: Numbers are separated by decimal digits, and each decimal digit is sequentially stored into an array. In the program implementation, firstly a string is used to store a number data, and a character stores a digit. Then the string is conversed to the corresponding decimal number and stored into an array.

- For a long positive integer, the program is as follow:
- int *a*[100]={0}; // Array *a* is used to store a long positive integer, one digit is stored in one element. Initial values are 0.
- int *n*; // *n* is the number of digits for the long integer
- string *s*; // String *s* is used to receive the integer
- cin>>*s*; // Input the integer into *s*
- *n*=*s*.length( ); // Calculate the number of digits
- for (*i*=0; *i*<*n*; *i*++) // Array *a* stores the integer from right to left, and one element stores one digit.
- *a*[*i*]=s[*n*-*i*-1]-'0';

Fundamental calculations of high precision numbers

- Fundamental calculations of high precision numbers are ‘+’, ‘-‘, ‘*’, and ‘/’.

- Addition and Subtraction of High Precision Numbers
  - Rules for addition and subtraction of high precision numbers are the same as rules of arithmetic addition and subtraction. In programs, addition of high precision numbers needs to carry, and subtraction of high precision numbers needs to borrow.

- Suppose x and y are two non-negative high precision integers, and $n1$ is the number of digits of x, and $n2$ is the number of digits of y. x and y are stored in array a and array b in above format. The program segment for addition of x and y is as follow:
- for (*i*=0; *i*<(*n*1>*n*2 ? *n*1 : *n*2); *i*++){ Addition of two integers whose numbers of digits are $n1$ and $n2$ respectively

```
            a[i]=a[i]+b[i];          // Bitwise addition
            if ( a[i]>9 ) {          //  Carry
                  a[i]=a[i]-10;
                  a[i+1]++;
            }
    }
```

- Suppose *x* and *y* are two non-negative high precision integers ($x>y$), and *n* is the number of digits of *x*. *x* and *y* are stored in array *a* and array *b* in above format. If $x<y$, then *a* and *b* exchange each other, and take a negative after the subtraction. The program segment subtraction is as follow:

```
for (i=0; i<n; i++) {
        if (a[i]>=b[i])
              a[i]= a[i]- b[i];
        else          // Borrow
      { a[i]= a[i]+10-b[i];
              a[i+1]--;
        }
}
```

- Multiplication and Division of High Precision Numbers
  - Based on addition and subtraction of high precision numbers, algorithms for multiplication and division of high precision numbers are given.

Adding Reversed Numbers

- **Source: ACM Central Europe 1998**
- **IDs for Online Judge: POJ 1504, ZOJ 2001, UVA 713**

- The Antique Comedians of Malidinesia prefer comedies to tragedies. Unfortunately, most of the ancient plays are tragedies. Therefore the dramatic advisor of ACM has decided to transfigure some tragedies into comedies. Obviously, this work is very hard because the basic sense of the play must be kept intact, although all the things change to their opposites. For example the numbers: if any number appears in the tragedy, it must be converted to its reversed form before being accepted into the comedy play.

- Reversed number is a number written in arabic numerals but the order of digits is reversed. The first digit becomes last and vice versa. For example, if the main hero had 1245 strawberries in the tragedy, he has 5421 of them now. Note that all the leading zeros are omitted. That means if the number ends with a zero, the zero is lost by reversing (e.g. 1200 gives 21). Also note that the reversed number never has any trailing zeros.

- ACM needs to calculate with reversed numbers. Your task is to add two reversed numbers and output their reversed sum. Of course, the result is not unique because any particular number is a reversed form of several numbers (e.g. 21 could be 12, 120 or 1200 before reversing). Thus we must assume that no zeros were lost by reversing (e.g. assume that the original number was 12).

- **Input**
- The input consists of N cases. The first line of the input contains only positive integer N. Then follow the cases. Each case consists of exactly one line with two positive integers separated by space. These are the reversed numbers you are to add.

- **Output**
- For each case, print exactly one line containing only one integer - the reversed sum of two reversed numbers. Omit any leading zeros in the output.

- **Sample Input**
- 3
- 24 1
- 4358 754
- 305 794

- **Sample Output**
- 34
- 1998
- 1

- Analysis
  - Suppose
    - $Num[0][0]$ stores the length of the first addend, and the first addend is stoted in $Num[0][1..Num[0][0]]$;
    - $Num[1][0]$ stores the length of the second addend, and the second addend is stored in $Num[1][1..Num[1][0]]$;
    - $Num[2][0]$ stores the length of the sum; and the sum is stored in $Num[2][1..Num[2][0]]$.

- The algorithm is as follow.
  - Firstly, input the first addend and the second addend, delete zeros which numbers end with. The two addends are stored into $Num[0]$ and $Num[1]$. Then chang them into reversed numbers.
  - Secondly, two reversed numbers $Num[0]$ and $Num[1]$ are added. Then their reversed sum $Num[2]$ are outputed. Note that the reversed number never has any trailing zeros.