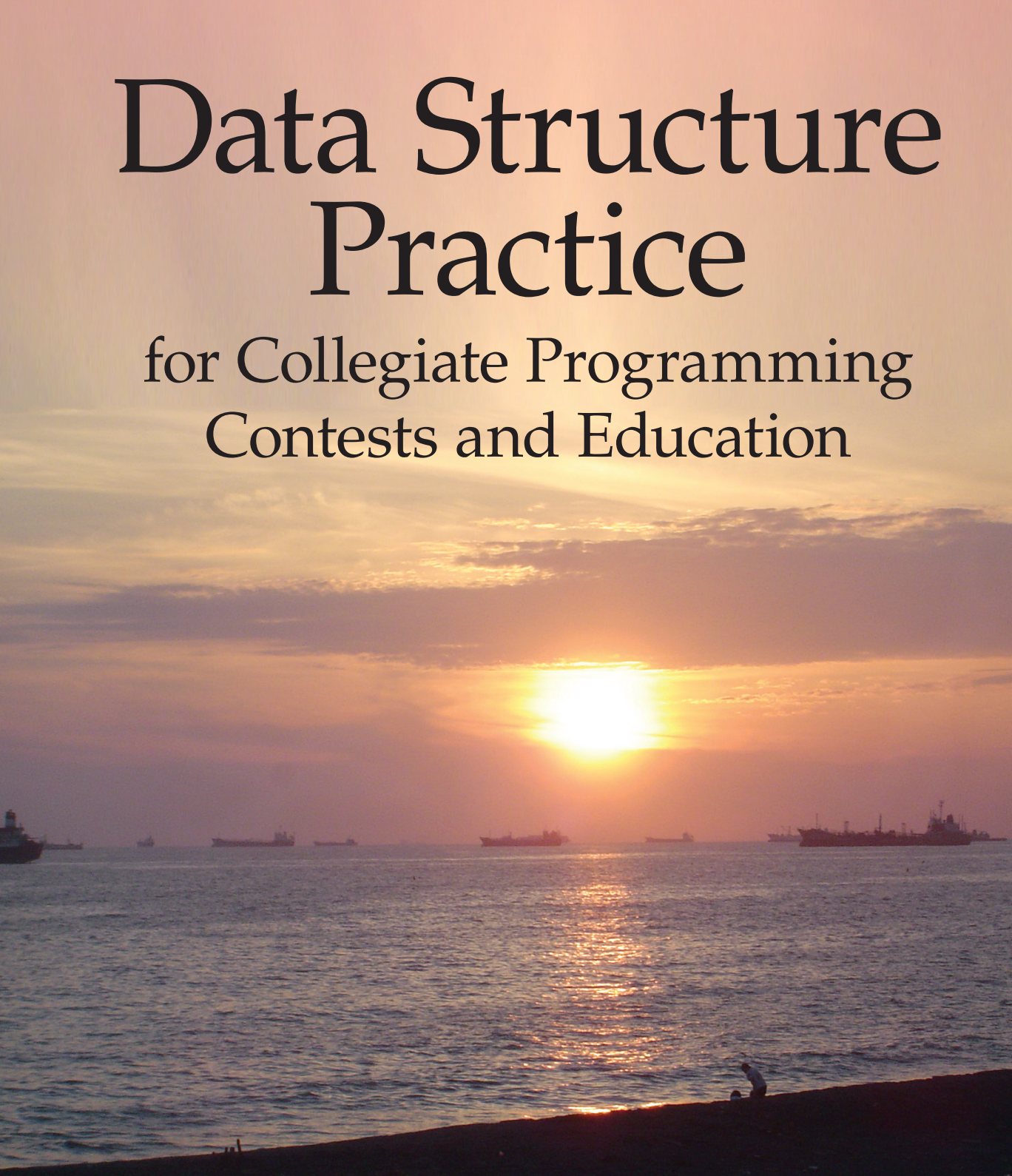
Data Structure Practice

for Collegiate Programming Contests and Education



Yonghui Wu and Jiande Wang

CRC Press
Taylor & Francis Group

Data Structure Practice

for Collegiate Programming Contests and Education

Data Structure Practice

for Collegiate Programming Contests and Education

Yonghui Wu and Jiande Wang


CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Published with arrangement with the original publisher, Beijing Huazhang Graphics and Information Company.

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20160126

International Standard Book Number-13: 978-1-4822-1540-3 (eBook - PDF)

Contents

Preface xiii
Authors xv

SECTION I FUNDAMENTAL PROGRAMMING SKILLS

1 Practice for Simple Computing **3**
1.1 Improving Programming Style 3
1.1.1 Financial Management 4
1.2 Multiple Test Cases 5
1.2.1 Doubles 5
1.2.2 Sum of Consecutive Prime Numbers 7
1.3 Precision of Real Numbers 9
1.3.1 I Think I Need a Houseboat 9
1.4 Improving Time Complexity by Dichotomy 11
1.4.1 Hangover 12
1.4.2 Humidex 14
1.5 Problems 17
1.5.1 Sum 17
1.5.2 Specialized Four-Digit Numbers 17
1.5.3 Quicksum 18
1.5.4 A Contesting Decision 19
1.5.5 Dirichlet's Theorem on Arithmetic Progressions 21
1.5.6 The Circumference of the Circle 22
1.5.7 Vertical Histogram 26
1.5.8 Ugly Numbers 27
1.5.9 Number Sequence 28

2 Simple Simulation **29**
2.1 Simulation of Direct Statement 29
2.1.1 Speed Limit 29
2.1.2 Ride to School 31
2.2 Simulation by Sieve Method 33
2.2.1 Self-Numbers 33
2.3 Construction Simulation 35
2.3.1 Bee 35

2.4 Problems ... 37
2.4.1 Gold Coins ... 37
2.4.2 The 3n + 1 Problem ... 38
2.4.3 Pascal Library ... 39
2.4.4 Calendar ... 41
2.4.5 Manager ... 42

3 Simple Recursion ... 45
3.1 Calculation of Recursive Functions ... 46
3.2 Solving Problems by Recursive Algorithms ... 47
3.2.1 Red and Black ... 48
3.3 Solving Recursive Datum ... 51
3.3.1 Symmetric Order ... 51
3.4 Problems ... 54
3.4.1 Fractal ... 54
3.4.2 Sticks ... 56

SUMMARY OF SECTION I

SECTION II EXPERIMENTS FOR LINEAR LISTS

4 Linear Lists Accessed Directly ... 65
4.1 Application of Arrays 1: Calculation of Dates ... 65
4.1.1 Calendar ... 66
4.1.2 What Day Is It? ... 68
4.2 Application of Arrays 2: Calculation of High-Precision Numbers ... 72
4.2.1 Adding Reversed Numbers ... 74
4.2.2 Very Easy! ... 77
4.3 Application of Arrays 3: Representation and Computation of Polynomials ... 80
4.3.1 Polynomial Showdown ... 81
4.3.2 Modular Multiplication of Polynomials ... 83
4.4 Application of Arrays 4: Calculation of Numerical Matrices ... 86
4.4.1 Error Correction ... 87
4.4.2 Matrix Chain Multiplication ... 89
4.5 Character Strings 1: Storage Structure of Character Strings ... 93
4.5.1 TEX Quotes ... 93
4.6 Character Strings 2: Pattern Matching of Character Strings ... 94
4.6.1 Blue Jeans ... 95
4.6.2 Oulipo ... 99
4.7 Problems ... 101
4.7.1 Moscow Time ... 101
4.7.2 Double Time ... 104
4.7.3 Maya Calendar ... 105
4.7.4 Time Zones ... 107
4.7.5 Polynomial Remains ... 109
4.7.6 Factoring a Polynomial ... 111
4.7.7 What's Cryptanalysis? ... 112

4.7.8 Run-Length Encoding 113
4.7.9 Zipper 114
4.7.10 Anagram Groups 115
4.7.11 English Number Translator 117
4.7.12 Message Decowding 118
4.7.13 Common Permutation 119
4.7.14 Human Gene Functions 120
4.7.15 Palindrome 122
4.7.16 Power Strings 123
4.7.17 Period 123
4.7.18 Seek the Name, Seek the Fame 124
4.7.19 Excuses, Excuses! 125
4.7.20 Product 128
4.7.21 Expression Evaluator 128
4.7.22 Integer Inquiry 129
4.7.23 Super-Long Sums 130
4.7.24 Exponentiation 131
4.7.25 Number Base Conversion 132
4.7.26 Super-Long Sums 134
4.7.27 Simple Arithmetics 134
4.7.28 $a^b - b^a$ 136
4.7.29 Fibonacci Number 137
4.7.30 How Many Fibs 138
4.7.31 Heritage 138

5 Applications of Linear Lists for Sequential Access 141
5.1 Application of Sequence Lists 142
5.1.1 Children 142
5.1.2 The Dole Queue 143
5.2 Application of Stacks 145
5.2.1 Rails 145
5.2.2 Boolean Expressions 151
5.3 Application of Queues 154
5.3.1 A Stack or a Queue? 154
5.3.2 Team Queue 156
5.3.3 Printer Queue 161
5.4 Problems 163
5.4.1 Roman Roulette 163
5.4.2 M*A*S*H 164
5.4.3 Joseph 165
5.4.4 City Skyline 166
5.4.5 Anagrams by Stack 168

6 Generalized List Using Indexes 171
6.1 Solving Problems Using Dictionaries 171
6.1.1 References 172
6.1.2 Babelfish 176

6.2 Solving Problems Using a Hash Table and the Hash Method....179
6.2.1 10-20-30....179
6.3 Problems....185
6.3.1 Spell Checker....185
6.3.2 Snowflake Snow Snowflakes....188
6.3.3 Equations....188

7 Sort of Linear Lists....191
7.1 Using Sort Function in STL....191
7.1.1 Hardwood Species....191
7.1.2 Who's in the Middle?....194
7.1.3 ACM Rank Table....195
7.2 Using Sort Algorithms....197
7.2.1 Flip Sort....197
7.2.2 Ultra-Quicksort....199
7.3 Problems....201
7.3.1 Ananagrams....201
7.3.2 Grandpa Is Famous....202
7.3.3 Word Amalgamation....203
7.3.4 Questions and Answers....205
7.3.5 Find the Clones....206
7.3.6 487-3279....207
7.3.7 Holiday Hotel....209
7.3.8 Train Swapping....210
7.3.9 Unix ls....211
7.3.10 Children's Game....213
7.3.11 DNA Sorting....214
7.3.12 Exact Sum....215
7.3.13 Shellsort....216
7.3.14 Tell Me the Frequencies!....219
7.3.15 Anagrams (II)....219
7.3.16 Flooded!....221
7.3.17 Football Sort....222
7.3.18 Trees....225

SUMMARY OF SECTION II

SECTION III EXPERIMENTS FOR TREES

8 Programming by Tree Structure....231
8.1 Solving Hierarchical Problems by Tree Traversal....231
8.1.1 Nearest Common Ancestor....232
8.1.2 Hire and Fire....236
8.2 Union–Find Sets Supported by Tree Structure....241
8.2.1 Find Them, Catch Them....243
8.2.2 Cube Stacking....246
8.3 Calculation of Sum of Weights of Subtrees by Binary Indexed Trees....248
8.3.1 Apple Tree....250

8.4 Problems 254
8.4.1 Friends 254
8.4.2 Wireless Network 255
8.4.3 War 257
8.4.4 Ubiquitous Religions 259
8.4.5 Network Connections 260
8.4.6 Building Bridges 261
8.4.7 Family Tree 264
8.4.8 Directory Listing 267
8.4.9 Closest Common Ancestors 268
8.4.10 Who's the Boss? 269
8.4.11 Disk Tree 272
8.4.12 Marbles on a Tree 273
8.4.13 This Sentence Is False 275

9 Applications of Binary Trees 281
9.1 Converting Ordered Trees to Binary Trees 281
9.1.1 Tree Grafting 282
9.2 Paths of Binary Trees 285
9.2.1 Binary Tree 285
9.3 Traversal of Binary Trees 287
9.3.1 Tree Recovery 288
9.4 Problems 291
9.4.1 Tree Summing 291
9.4.2 Trees Made to Order 292

10 Applications of Classical Trees 295
10.1 Binary Search Trees 295
10.1.1 BST 296
10.1.2 Falling Leaves 297
10.2 Binary Heaps 301
10.2.1 Windows Message Queue 303
10.2.2 Binary Search Heap Construction 306
10.2.3 Decode the Tree 309
10.3 Huffman Trees 311
10.3.1 Fence Repair 312
10.4 Problems 314
10.4.1 Cartesian Tree 314
10.4.2 Argus 316
10.4.3 Black Box 317
10.4.4 Heap 319
10.4.5 How Many Trees? 320
10.4.6 The Number of the Same BST 322
10.4.7 The Kth BST 325
10.4.8 The Prufer Code 330
10.4.9 Code the Tree 331

SUMMARY OF SECTION III

SECTION IV EXPERIMENTS FOR GRAPHS

11 Applications of Graph Traversal 337
11.1 BFS Algorithm 337
11.1.1 Prime Path 338
11.2 DFS Algorithm 342
11.2.1 House of Santa Claus 342
11.3 Topological Sort 344
11.3.1 Following Orders 345
11.3.2 Sorting It All Out 348
11.4 Connectivity of Undirected Graphs 352
11.4.1 Knights of the Round Table 356
11.5 Problems 362
11.5.1 Ordering Tasks 362
11.5.2 Spreadsheet 363
11.5.3 Genealogical Tree 365
11.5.4 Rare Order 366
11.5.5 Pushing Boxes 367
11.5.6 Basic Wall Maze 373
11.5.7 Firetruck 375
11.5.8 Dungeon Master 377
11.5.9 A Knight's Journey 379
11.5.10 Children of the Candy Corn 381
11.5.11 Curling 2.0 383
11.5.12 Shredding Company 387
11.5.13 Be Wary of Roses 390
11.5.14 Monitoring the Amazon 393
11.5.15 Graph Connectivity 394
11.5.16 The Net 395
11.5.17 The Warehouse 397

12 Algorithms of Minimum Spanning Trees 405
12.1 Kruskal Algorithm 405
12.1.1 Constructing Roads 406
12.2 Prim Algorithm 408
12.2.1 Agri-Net 409
12.3 Problems 412
12.3.1 Network 412
12.3.2 Truck History 413
12.3.3 Slim Span 414
12.3.4 The Unique MST 419
12.3.5 Highways 420

13 Algorithms of Best Paths 423
13.1 Warshall Algorithm and Floyd–Warshall Algorithm 423
13.1.1 Frogger 424
13.1.2 Arbitrage 427

13.2 Dijkstra's Algorithm 430
13.2.1 Toll 431
13.3 Bellman–Ford Algorithm 434
13.3.1 Minimum Transport Cost 435
13.4 Shortest Path Faster Algorithm (SPFA Algorithm) 439
13.4.1 Longest Paths 440
13.5 Problems 443
13.5.1 Knight Moves 443
13.5.2 Big Christmas Tree 444
13.5.3 Stockbroker Grapevine 446
13.5.4 Domino Effect 448
13.5.5 106 miles to Chicago 452
13.5.6 AntiFloyd 454

14 Algorithms of Bipartite Graphs and Flow Networks 457
14.1 Maximum Matching in Bipartite Graphs 457
14.1.1 Conference 458
14.2 Flow Networks 460
14.2.1 Power Network 461
14.2.2 Trash 467
14.3 Problems 470
14.3.1 A Plug for UNIX 470
14.3.2 Machine Schedule 471
14.3.3 Selecting Courses 473
14.3.4 Software Allocation 474
14.3.5 Crime Wave 475
14.3.6 Pigs 477
14.3.7 Drainage Ditches 479
14.3.8 Mysterious Mountain 480

SUMMARY OF SECTION IV

Bibliography 489

Preface

Since the 1990s, the ACM International Collegiate Programming Contest (ACM-ICPC) has become a worldwide programming contest. Every year, more than 10,000 students and more than 1,000 universities participate in local contests, preliminary contests, and regional contests all over the world. In the meantime, programming contests' problems from all over the world can be gotten, analyzed, and solved by us. These contest problems can be used not only for programming contest training, but also for education.

In our opinion, not only a programming contestant's ability, but also a computer student's ability is based on his or her programming knowledge system and programming strategies for solving problems. The programming knowledge system can be summarized as a famous formula: algorithms + data structures = programs. It is also the foundation for the knowledge system of computer science and engineering. Strategies solving problems are strategies for data modeling and algorithm design. When data models and algorithms for problems are not standard, what strategies we should take to solve these problems?

Based on the ACM-ICPC, we published a series of books, not only for systematic programming contest training, but also for better polishing computer students' programming skill, using programming contests' problems: "Data Structure Experiment: For Collegiate Programming Contest and Education," "Algorithm Design Experiment: For Collegiate Programming Contest and Education," and "Programming Strategies Solving Problems" in Mainland China. And the traditional Chinese versions for "Data Structure Experiment: For Collegiate Programming Contest and Education" and "Programming Strategies Solving Problems" were also published in Taiwan.

"Data Structure Practice: For Collegiate Programming Contests and Education" is the English version for "Data Structure Experiment: For Collegiate Programming Contest and Education." There are 4 sections, 14 chapters, and 200 programming contest problems in this book. Section I, "Fundamental Programming Skills," focuses on experiments and practices for simple computing, simple simulation, and simple recursion, for students just learning programming languages. Section II, "Experiments for Linear Lists," Section III, "Experiments for Trees," and Section IV, "Experiments for Graphs," focus on experiments and practices for data structure.

Characteristics of the book are as follows:

1. The book's outlines are based on the outlines of data structures. Programming contest problems and their analyses and solutions are used as experiments. For each chapter, there is a "Problems" section to let students solve programming contests' problems, and hints for these problems are also shown.
2. Problems in the book are all selected from the ACM-ICPC regional and world finals programming contests, universities' local contests, and online contests, and from 1990 to now.

3. Not only analyses and solutions or hints to problems are shown, but also test data for most of problems are provided. Sources and IDs for online judges for these problems are also given. They can help readers better and more easily polish their programming skills.

Therefore, the book can be used not only as an experiment book, but also for systematic programming contests' training.

We appreciate Professors Steven Skiena, Rezaul Chowdhury, C. Jinshong Hwang, Ziliang Zong, Hongchi Shi, and Rudolf Fleischer. They provided us platforms in which English is the native language that improved our manuscript.

We appreciate our students Julaiti Alafate, Zheyun Yao, and Hao Zhang. They finished programs in the book.

The work is supported by the China Scholarship Council.

Online judge systems for problems in this book are as follows:

Online Judge Systems	*Abbreviations*	*Website*
Peking University Online Judge System	POJ	http://poj.org/
Zhejiang University Online Judge System	ZOJ	http://acm.zju.edu.cn/onlinejudge/
UVA Online Judge System	UVA	http://uva.onlinejudge.org/
		http://livearchive.onlinejudge.org/
Ural Online Judge System	Ural	http://acm.timus.ru/
SGU Online Judge System	SGU	http://acm.sgu.ru/

If you discover anything you believe to be an error, please contact us through Yonghui Wu's email: yhwu@fudan.edu.cn. Your help is appreciated.

Yonghui Wu
Jiande Wang

Authors

Yonghui Wu is associate professor at Fudan University. He acted as the coach of Fudan University Programming Contest teams from 2001 to 2011. Under his guidance, Fudan University qualified for Association for Computing Machinery International Collegiate Programming Contest (ACM-ICPC) World Finals every year and won three medals (bronze medal in 2002, silver medal in 2005, and bronze medal in 2010). Since 2012, he has published a series of books for programming contests and education. Since 2013, he has given lectures in Oman, Taiwan, and the United States for programming contest training. He is the chair of the ICPC Asia Programming Contest 1st Training Committee now.

Jiande Wang is a senior high school teacher and a famous coach for the Olympiad in Informatics in China. He has published 24 books for programming contests since the 1990s. Under his guidance, his students have won seven gold medals, three silver medals, and two bronze medals in the International Olympiad in Informatics for China.

FUNDAMENTAL PROGRAMMING SKILLS

I

Programming language is an introductory course of data structures and algorithms. This course enables students to program by programming languages. Programming languages, data structures, and algorithm designs are skills that computer students must polish. Therefore, polishing fundamental programming skills is the first section for this book. There are three chapters in Section I covering

1. Computing
2. Simulation
3. Recursion

These three chapters are not only a review of programming languages, but also an introductory course on data structure.

Chapter 1

Practice for Simple Computing

The pattern of a programming contest problem is input–process–output. A problem for simple computing is a problem whose process is simple. For such a problem, we should only consider optimizing the process and dealing with input and output correctly. The goals of Chapter 1 are as follows:

1. Students master C/C++ or Java programming language.
2. Students become familiar with online judge systems and programming environments.
3. Students begin to learn how to transfer a practical problem into a computing process, implement the computing process by a program, and debug the program to pass all test cases.

"God is in the details." In Chapter 1, problems are relatively simple. We should notice formats of input and output, precision, and time complexity. Therefore, the following topics will be discussed in this chapter:

1. Programming style
2. Multiple test cases
3. Precision of real numbers
4. Improving time complexity by dichotomy

Normally, a complex problem consists of several subproblems for simple computing. "Even the longest journey begins with a single step." Polishing programming skills should begin with solving simple computing problems.

1.1 Improving Programming Style

A program's writing style is not only for its visual sense, but also for examining the program and debugging its errors. A program's style also shows whether its programming idea is clear. It is hard to say which kind of programming style is good, but there are some rules for programing style. They are discussed in the following experiments.

1.1.1 Financial Management

Larry graduated this year and finally has a job. He's making a lot of money, but somehow never seems to have enough. Larry has decided that he needs to get a hold of his financial portfolio and solve his financial problems. The first step is to figure out what's been going on with his money. Larry has his bank account statements and wants to see how much money he has. Help Larry by writing a program to take his closing balance from each of the past 12 months and calculate his average account balance.

Input

The input will be 12 lines. Each line will contain the closing balance of his bank account for a particular month. Each number will be positive and displayed to the penny. No dollar sign will be included.

Output

The output will be a single number, the average (mean) of the closing balances for the 12 months. It will be rounded to the nearest penny, preceded immediately by a dollar sign, and followed by the end of the line. There will be no other spaces or characters in the output.

Sample Input	*Sample Output*
100.00	$1581.42
489.12	
12454.12	
1234.10	
823.05	
109.20	
5.27	
1542.25	
839.18	
83.99	
1295.01	
1.75	

Source: ACM Mid-Atlantic United States 2001.

IDs for online judges: POJ 1004, ZOJ 1048, UVA 2362.

Analysis

The problem's pattern, input–process–output, is very simple: First, the income of 12 months $a[0 .. 11]$ is input by a for statement $for(i = 0; i < 12; i{+}{+})$, and the total income

$$sum = \sum_{i=0}^{11} a[i]$$

is calculated. Then the average monthly income $avg = sum/12$ is calculated. Finally, *avg* is output in accordance with the problem's output format.

Program

```
#include<iostream>                          // Preprocessor Directive
using namespace std;                        // Using C++ Standard Library
int main()                                  // Main function
{
      double avg, sum=0.0, a[12]={0};       // Real variable avg and sum,
and real array a
      int i;                                // Integer variable i
      for(i=0;i<12;i++){                 // Input the income of 12 months a[0..11]
and summation
                cin>>a[i];
                sum+=a[i];
            }
      avg=sum/12;                          // Calculate the average monthly
income
      printf("$%.2f",avg);                 // Output the average monthly
income
      return 0;
   }
```

From the above program, we can get the following:

First, the input and output of the program must meet formats for input and output. In this problem, each input number will be positive and displayed to the penny, and the output will be rounded to the nearest penny, preceded immediately by a dollar sign and followed by an end of the line. If the program doesn't meet formats for input and output, it will be judged as the wrong answer.

Second, a program should be readable. The style of a program should be serration based on a logical level.

Finally, program annotations should be given.

1.2 Multiple Test Cases

The financial management problem (Section 1.1.1) has only one test case. In order to guarantee the correctness of a program, for most problems there are multiple test cases. In some circumstances, the number of test cases is given; in other circumstances, the number of test cases isn't given, but the mark of the input end is given.

1.2.1 Doubles

As part of an arithmetic competency program, your students will be given randomly generated lists of 2–15 unique positive integers and asked to determine how many items in each list are twice

some other item in the same list. You will need a program to help you with the grading. This program should be able to scan the lists and output the correct answer for each one. For example, given the list

1 4 3 2 9 7 18 22

your program should answer 3, as 2 is twice 1, 4 is twice 2, and 18 is twice 9.

Input

The input file will consist of one or more lists of numbers. There will be one list of numbers per line. Each list will contain from 2 to 15 unique positive integers. No integer will be larger than 99. Each line will be terminated with the integer 0, which is not considered part of the list. A line with the single number –1 will mark the end of the file. The example input below shows three separate lists. Some lists may not contain any doubles.

Output

The output will consist of one line per input list, containing a count of the items that are double some other item.

Sample Input	*Sample Output*
1 4 3 2 9 7 18 22 0	3
2 4 8 10 0	2
7 5 11 13 1 3 0	0
–1	

Source: ACM Mid-Central United States 2003.

IDs for online judges: POJ 1552, ZOJ 1760, UVA 2787.

Analysis

There are multiple test cases for the problem. Therefore, a loop statement is used to deal with multiple test cases. The loop enumerates every test case. –1 marks the end of the input. Therefore, –1 is the end condition of the loop. In the loop statement, there are two steps:

1. A loop inputs a test case into array a and accumulates the number of elements n in the test case. 0 marks the end of the test case.
2. A double loop enumerates all pairs of $a[i]$ and $a[j]$ ($0 <= i < n - 1$, $i + 1 <= j < n$) in the test case and determines whether ($a[i]*2 == a[j] \,||\, a[j]*2 == a[i]$) holds.

Program

```
#include <iostream>                    // Preprocessor Directive
using namespace std;                    // Using C++ standard library
int main()                             //Main function
{
     int i, j, n, count, a[20];                //  Integer variables i,
j, n, count and array a
```

```
    cin>>a[0];                          // Input the first element
    while(a[0]!=-1)                    // If it is not the end of input,
input a new test case
    {  n=1;                            // Input array a
        for( ; ; n++)
        {
          cin>>a[n];
          if (a[n]==0) break;
         }
         count=0;        // Determine how many items in each list are
twice some other item
         for (i=0; i<n-1; i++)               // Enumerate all pairs
         {
           for (j=i+1; j<n; j++)
           {
             if (a[i]*2==a[j] || a[j]*2==a[i])    // Accumulation
             count++;
             }
           }
           cout<<count<<endl;              // Output the result
           cin>>a[0];                      // Input the first element of
next test case
    }
    return 0;
  }
```

In this problem, the number of test cases and the size of a test case are unknown. Normally, a double-loop statement is used for the program structure: the outer loop is used to enumerate every test case, and the inner loop is used to deal with a test case.

In some problems, if the size of the test data is larger, all the test cases are dealt with by the same method, and the result area is known, its time complexity can be improved by an offline method. First, all solutions within the specified range are calculated and stored in a constant array. Then the program deals with the constant array directly for each test case. It can avoid duplication of computing.

1.2.2 Sum of Consecutive Prime Numbers

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 10,000, 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	*Sample Output*
2	1
3	1
17	2
41	3
20	0
666	0
12	1
53	2
0	

Source: ACM Japan 2005.

IDs for online judges: POJ 2739, UVA 3399.

Analysis

Because the program needs to deal with consecutive prime numbers for each test case, and the upper limit of prime numbers is 10,000, the offline method can be used to solve the problem.

First, all prime numbers less than 10,001 are obtained and stored in array *prime*[1 .. *total*] in ascending order.

Then we deal with the 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*], is to calculate the sum of consecutive prime numbers. If *cnt* ≥ *n*, then the loop ends, and if *cnt* == *n*, then the number of representations is *ans*++.

When the outer loop ends, *ans* is the solution to the test case.

Program

```
#include <iostream>                    // Preprocessor Directive
using namespace std;                   // Using C++ Standard Library
const int maxp = 2000, n = 10000;      // Set the size of prime array and
the upper limit of  prime numbers
int prime[maxp], total = 0;              // Initialization
```

```
bool isprime(int k)                    // Determine whether k is a
prime number or not
{
    for (int i = 0; i < total; i++)
        if (k % prime[i] == 0)
            return false;
    return true;
}
int main(void)                         // Main Function
{
    for (int i = 2; i <= n; i++)           //  get all prime numbers
less than 10001
      if (isprime(i))
            prime[total++] = i;
    prime[total] = n + 1;
    int m;
    cin >> m;                          // Input the first positive integer
    while (m) {
        int ans = 0;                     // Initialization
        for (int i = 0; m >= prime[i]; i++) { // Enumerate the least
prime number
            int cnt = 0;                 // Calculate the sum of
consecutive prime numbers
            for (int j = i; j < total && cnt < m; j++)
                cnt += prime[j];
            if (cnt == m)              // if cnt==n, then ++ans
                ++ans;
        }
        cout << ans << endl;           // Output the result
        cin >> m;                     // Input the next positive integer
    }
    return 0;
}
```

1.3 Precision of Real Numbers

In some cases, we need to deal with real numbers and real arithmetics to solve problems, such as judging whether two real numbers are equal, and so on. For a programming language, precision of real numbers is limited. And sometime programs are required to meet requirements for accuracy errors of real numbers. If the program can't deal with such details well, it will lead to the wrong answer even though its algorithm is correct.

1.3.1 I Think I Need a Houseboat

Fred Mapper is considering purchasing some land in Louisiana to build his house on. In the process of investigating the land, he learned that the state of Louisiana is actually shrinking by 50 square miles each year, due to erosion caused by the Mississippi River. Since Fred is hoping to live in this house for the rest of his life, he needs to know if his land is going to be lost to erosion.

After doing more research, Fred has learned that the land that is being lost forms a semicircle. This semicircle is part of a circle centered at (0, 0), with the line that bisects the circle being the X

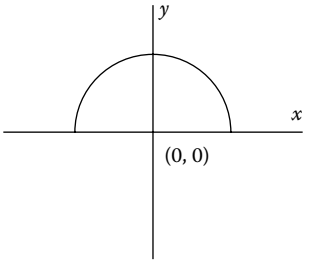


Figure 1.1 The land that is being lost forms a semicircle.

axis. Locations below the X axis are in the water. The semicircle has an area of 0 at the beginning of year 1. (The semicircle is illustrated in Figure 1.1.).

Input

The first line of input will be a positive integer indicating how many data sets will be included (N).

Each of the next N lines will contain the X and Y Cartesian coordinates of the land Fred is considering. These will be floating-point numbers measured in miles. The Y coordinate will be nonnegative. (0, 0) will not be given.

Output

For each data set, a single line of output should appear. This line should take the form of

Property N: This property will begin eroding in year Z.

where N is the data set (counting from 1) and Z is the first year (start from 1) this property will be within the semicircle AT THE END OF YEAR Z. Z must be an integer. After the last data set, this should print out "END OF OUTPUT."

Sample Input	*Sample Output*
2	Property 1: This property will begin eroding in year 1.
1.0 1.0	Property 2: This property will begin eroding in year 20.
25.0 0.0	END OF OUTPUT.

Source: ACM Mid-Atlantic United States 2001.

Note: No property will appear exactly on the semicircle boundary: it will be either inside or outside. This problem will be judged automatically. Your answer must match exactly, including the capitalization, punctuation, and white space. This includes the periods at the ends of the lines. All locations are given in miles.

IDs for online judges: POJ 1005, ZOJ 1049, UVA 2363.

Analysis

The number of test cases n is given. Therefore, a *for* repetition statement is used to deal with all test cases. Each test case contains only X and Y Cartesian coordinates. The ith test case (X_i, Y_i) and the center of the circle (0, 0) constitute the semicircle that will be eroded. Each year 50 square miles

of land is eroded. And the number of years is an integer. When (X_i, Y_i) is in water, the number of years must be the least integer that is greater than

$$\frac{\text{Area of the semicircle}}{50}$$

and function *ceil*(*x*) is used to round up the fare.

Program

```
#include <stdio.h>                    // Preprocessor Directive
#include <math.h>
#define M_PI 3.14159265
int num_props;                       // The number of test cases
float x, y;                           // X and Y Cartesian coordinates
int i;
double calc;                          // The area of the semicircle/50
int years;                            // The number of years
int main( )                        // Main function
{
   scanf("%d", &num_props);        // Input the number of test cases
   for (i = 1; i <= num_props; i++)
   {
     scanf("%f %f", &x, &y);         // Input the i-th test case
     calc = (x*x + y*y)* M_PI / 2 / 50; // Calculate the area of the
semi-circle/50
     years = ceil(calc);
     printf("Property %d: This property will begin eroding in year %d.n",
i, years); //Output
   }
   printf("END OF OUTPUT.n");
}
```

In real arithmetic, sometimes we need to determine whether real number *x* and real number *y* are equal. Using $y - x == 0$ as the condition may result in error of precision. The method avoiding error of precision is to set a constant of precision *delta*. If $|y - x| < delta$, then we can judge *x* and *y* are equal. The hangover problem (Section 1.4.1) shows such an example.

1.4 Improving Time Complexity by Dichotomy

In some cases, the data area of a problem is an ordered interval. Dichotomy is used to divide the interval into two subintervals and then determine if the process of computation is in the left subinterval or the right subinterval. If the solution isn't obtained, then repeat the above steps. For a problem whose time complexity is O(*n*), if dichotomy can be used to solve it, its time complexity can be improved to $O(\log_2(n))$.

Dichotomy is used in many algorithms, such as binary search, recursive halving method, quick sort, merge sort, binary search tree, and segment tree. Among these methods, binary search and recursive halving method are relatively simple algorithms.

The idea for binary search is as follows: Suppose the data area is an interval in ascending order. The search begins by comparing *x* with the number in the middle of the interval. If *x* equals this

number, the search terminates. If x is smaller than the number, then we need only search in the left half; if x is greater than the number, then we need only search in the right half. We repeat the above steps until the search ends.

1.4.1 Hangover

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 $1/2 + 1/3 + 1/4 + \ldots + 1/(n + 1)$ card lengths, where the top card overhangs the second by 1/2, the second overhangs the third by 1/3, the third overhangs the fourth by 1/4, and so on, and the bottom card overhangs the table by $1/(n + 1)$. This is illustrated in Figure 1.2.

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	*Sample Output*
1.00	3 card(s)
3.71	61 card(s)
0.04	1 card(s)
5.19	273 card(s)
0.00	

Source: ACM Mid-Central United States 2001.

IDs for online judges: POJ 1003, UVA 2294.

Analysis

The problem's data area is little. Therefore, first lengths that cards achieve are calculated, and the length is at most 5.20 card lengths. Suppose the *total* is the number of cards and *len*[*i*] is the length

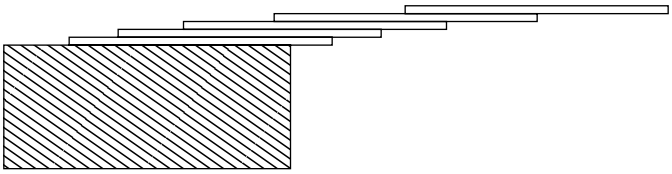


Figure 1.2 A stack of cards overhangs a table.

that i cards achieve. That is, $len[i] = len[i-1] + 1/(i+1)$, where $i \geq 1$ and $len[0] = 0$. 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. Function $zero(x)$ is defined as follows:

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

Initially $len[0] = 0$. Array *len* can be obtained 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 obtained, 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 input. The loop terminates when $x = 0.00$.

The procedure of dichotomy is as follows:

The initial interval $[l, r] = [1, total]$ and $mid = [(l + r)/2]$. If $zero(len[mid] - x) < 0$, then search the right half ($l = mid$); otherwise, search the left half ($r =$ mid). Repeat the above steps in interval $[l, r]$ until $l + 1 \geq r$. r is the minimum number of cards.

Program

```
#include <iostream>          // Preprocessor Directive
using namespace std;         // Using C++ Standard Library
const int maxn = 300;         // Size of array len
const double delta = 1e-8;     // Set the accuracy error
int zero(double x)           // In the area of accuracy error delta, if x
is a negative real number less than 0, then return -1; if x is a positive
real number larger than 0, then return 1; and if x is 0, then return 0.
{
    if (x < -delta)
        return -1;
    return x > delta;
}
int main(void)             // Main Function
{
    double len[maxn];      // Define array len and the length of len
    int total;
    len[0] = 0.0;       // Calculate array len, and len[i] is the length
that i cards achieve
    for (total = 1; zero(len[total - 1] - 5.20) < 0; total++)
        len[total] = len[total - 1] + 1.0 / double(total + 1);
    double x;
    cin >> x;             // Input the first test case x
    while (zero(x)) {       // Using dichotomy to get the minimum number
of cards necessary to achieve an overhang of at least x card lengths.
```

```
        int l, r;
        l = 0;           // Set left pointer l and right pointer r for
the interval
        r = total;
        while (l + 1 < r) {
            int mid = (l + r) / 2;
            if (zero(len[mid] - x) < 0)      // If the middle value is
less than x, then search the right half, else search the left half.
                l = mid;
            else
                r = mid;
        }
        cout << r << " card(s)" << endl;    // Output the minimum number
of cards
        cin >> x;                      //Input the next test case
    }
    return 0;
}
```

Dichotomy can be used not only in a data search, but also in function calculation. Suppose there are variables x_1, x_2, and x_3 and function $x_1 = f(x_2, x_3)$ holds. The recursive halving method can be used to calculate x_3 when x_1 and x_2 are known. The method is as follows:

Halving is to halve the data area of a problem (such as the data area of x_3), and the property of the problem (such as $x_1 = f(x_2, x_3)$) is not changed. Suppose the size of data area for the problem is n. We can first make use of some methods to change the original problem into c subproblems with half of the data area (c is a constant, is related to the problem, and is not related to the data area), and then solve the problem by solving subproblems whose size of data area is $n/2$. Properties for these subproblems are the same as those for the original problem, but the size of the data area for these subproblems is smaller.

Recursion is to repeat the above halving steps. A problem whose size of data area is $n/2$ is changed into c subproblems whose size is $n/4$, and so on. Repeat the above steps until the subproblems can be solved easily.

1.4.2 Humidex

The humidex is a measurement used by Canadian meteorologists to reflect the combined effect of heat and humidity. It differs from the heat index used in the United States in using dew point rather than relative humidity.

When the temperature is 30°C (86°F) and the dew point is 15°C (59°F), the humidex is 34 (note that humidex is a dimensionless number, but the number indicates an approximate temperature in Celsius). If the temperature remains 30°C and the dew point rises to 25°C (77°F), the humidex rises to 42.3.

The humidex tends to be higher than the U.S. heat index at equal temperature and relative humidity.

The current formula for determining the humidex was developed by J.M. Masterton and F.A. Richardson of Canada's Atmospheric Environment Service in 1979.

According to the Meteorological Service of Canada, a humidex of at least 40 causes "great discomfort" and above 45 is "dangerous." When the humidex hits 54, heat stroke is imminent.

The record humidex in Canada occurred on June 20, 1953, when Windsor, Ontario, hit 52.1. (The residents of Windsor would not have known this at the time, since the humidex had yet

to be invented.) More recently, the humidex reached 50 on July 14, 1995, in both Windsor and Toronto.

The humidex formula is as follows:

humidex = temperature + h
$h = (0.5555)*(e - 10.0)$
$e = 6.11*exp\ [5417.7530*((1/273.16) - (1/(\text{dewpoint} + 273.16)))]$

where $exp(x)$ is 2.718281828 raised to the exponent x.

While humidex is just a number, radio announcers often announce it as if it were the temperature, for example, "It's 47° out there … with the humidex." Sometimes weather reports give the temperature and dew point, or the temperature and humidex, but rarely do they report all three measurements. Write a program that, given any two of the measurements, will calculate the third.

You may assume that for all inputs, the temperature, dew point, and humidex are all between –100°C and 100°C.

Input

Input will consist of a number of lines. Each line except the last will consist of four items separated by spaces: a letter, a number, a second letter, and a second number. Each letter specifies the meaning of the number that follows it and will be either T, indicating temperature; D, indicating dew point; or H, indicating humidex. The last line of input will consist of the single letter E.

Output

For each line of input except the last, produce one line of output. Each line of output should have the form:

T number D number H number

where the three numbers are replaced with the temperature, dew point, and humidex. Each value should be expressed rounded to the nearest tenth of a degree, with exactly one digit after the decimal point. All temperatures are in degrees Celsius.

Sample Input	*Sample Output*
T 30 D 15	T 30.0 D 15.0 H 34.0
T 30.0 D 25.0	T 30.0 D 25.0 H 42.3
E	

Source: Waterloo Local Contest, July 14, 2007.

ID for online judge: POJ 3299.

Analysis

Based on the humidex formula *humidex* = *temperature* + h, h is proportional to the dew point. If the dew point and temperature (or humidex) are known, the value of h can be inferred, and the humidex or temperature can be calculated by the humidex formula. If temperature and humidex are known, the recursive halving method can be used to calculate the dew point.

The program sets an initial value 0 to the dew point and enters a loop: the initial value of the increment for dew point is 100; halve the increment value each time as the loop is performed. If the humidex obtained from the formula is larger than the announced humidex, then the value of

the dew point decreases an increment (i.e., $h\searrow$; decrease the humidex to be close to the announced humidex); otherwise, the value of the dew point increases an increment (i.e., $h\nearrow$; increase the humidex to be close to the announced humidex). The repetition condition is the increment value greater than 0.0001. When the loop ends, the dew point is the answer.

Program

```
#include <stdio.h>                  // Preprocessor Directive
#include <math.h>
#include <assert.h>
char a,b;                          //Two characters for Test Mark
double A,B,temp,hum,dew;
double dohum(double tt, double dd){ // Calculate humidex based on
temperature tt and dew point dd
   double e = 6.11 * exp (5417.7530 * ((1/273.16) - (1/(dd+273.16))));
   double h = (0.5555)*(e - 10.0);
   return tt + h;                  // Return humidex
}
double dotemp( ){         // Calculate temperature based on dew point dew
and humidex hum
   double e = 6.11 * exp (5417.7530 * ((1/273.16) - (1/(dew+273.16))));
   double h = (0.5555)*(e - 10.0);
   return hum - h;               // Return temperature
}
double dodew( ){                //Calculate dew point based on
temperature temp and humidex hum
   double x = 0;               // Initialization of dew point and
increment
   double delta=100;
//Loop; Halve the increment value each time the loop is performed. If
humidex gotten from the formula is larger than the announced humidex,
then the value of dew point decrease an increment; otherwise the value of
dew point increase an increment. Repeat the procedure until the increment
value delta<=0.0001.
   for (delta=100; delta>.00001; delta *=.5) {
      if (dohum(temp, x)>hum) x -= delta;
      else x += delta;
   }
   return x;                       //Return dew point x
}
int main( )                              // main function
{      //Loop: each loop inputs two measurements and loop-end condition
is 'E'.

   while (4 == scanf(" %c %lf %c %lf",&a,&A,&b,&B) && a != 'E'){
      temp = hum = dew = -99999;  // Initialization of temperature,
humidex and dew point
      if (a == 'T') temp = A;    // The first measurement is temperate.
      if (a == 'H') hum = A;     // The first measurement is humidex.
      if (a == 'D') dew = A;     // The first measurement is dew point.
      if (b == 'T') temp = B;    // The second measurement is temperate.
      if (b == 'H') hum = B;     // The second measurement is humidex.
      if (b == 'D') dew = B;     // The second measurement is dew point.
```

```
      if (hum == -99999) hum = dohum(temp, dew);// Calculate humidex
based on temperate and dew point.
      if (dew == -99999) dew=dodew( );   // Calculate dew point based on
temperate and humidex.
      if (temp == -99999) temp = dotemp( );   // Calculate temperate
based on humidex and dew point.
      printf("T %0.1lf D %0.1lf H %0.1lfn",temp, dew,hum);  //Output
temperate, dew point and humidex.
   }
   assert(a == 'E');                        //  Loop-end condition is 'E'.
}
```

1.5 Problems

1.5.1 Sum

Your task is to find the sum of all integer numbers lying between 1 and N inclusive.

Input

The input consists of a single integer N that is not greater than 10,000 by its absolute value.

Output

Write a single integer number that is the sum of all integer numbers lying between 1 and N inclusive.

Sample Input	*Sample Output*
–3	–5

Source: ACM 2000, Northeastern European Regional Programming Contest (test tour).

ID for online judge: Ural 1068.

Hint

Based on the summation formula of arithmetic progression $s = 1 + 2 + \ldots N$, if N is an integer larger than 0, then $s = [(1 + N)/2]*N$; otherwise, $s = [(1 - N)/2]*N + 1$.

1.5.2 Specialized Four-Digit Numbers

Find and list all four-digit numbers in decimal notation that have the property that the sum of their four digits equals the sum of their digits when represented in hexadecimal (base 16) notation and also equals the sum of their digits when represented in duodecimal (base 12) notation.

For example, the number 2991 has the sum of (decimal) digits 2 + 9 + 9 + 1 = 21. Since 2991 = 1*1728 + 8*144 + 9*12 + 3, its duodecimal representation is 1893_{12}, and these digits also sum up to 21. But in hexadecimal, 2991 is BAF_{16}, and 11 + 10 + 15 = 36, so 2991 should be rejected by your program.

The next number (2992), however, has digits that sum to 22 in all three representations (including $BB0_{16}$), so 2992 should be on the listed output. (We don't want decimal numbers with fewer than four digits—excluding leading zeros—so that 2992 is the first correct answer.)

Input

There is no input for this problem.

Output

Your output is to be 2992 and all larger four-digit numbers that satisfy the requirements (in strictly increasing order), each on a separate line, with no leading or trailing blanks, ending with a new-line character. There are to be no blank lines in the output. The first few lines of the output are shown below:

Sample Input	*Sample Output*
	2992
	2993
	2994
	2995
	2996
	2997
	2998
	2999
	…

Source: ACM Pacific Northwest 2004.

IDs for online judges: POJ 2196, ZOJ 2405, UVA 3199.

Hint

First, function *calc*(*k*, *b*) is designed to calculate and return the sum of digits of number *k* represented in base *b*. Then every number *i* in [2992 … 9999] is enumerated: if *calc*(*i*, 10) == *calc*(*i*, 12) == *calc*(*i*, 16), then output *i*.

1.5.3 Quicksum

A checksum is an algorithm that scans a packet of data and returns a single number. The idea is that if the packet is changed, the checksum will also change, so checksums are often used for detecting transmission errors, validating document contents, and in many other situations where it is necessary to detect undesirable changes in data.

For this problem, you will implement a checksum algorithm called quicksum. A quicksum packet allows only uppercase letters and spaces. It always begins and ends with an uppercase letter. Otherwise, spaces and letters can occur in any combination, including consecutive spaces.

A quicksum is the sum of the products of each character's position in the packet times the character's value. A space has a value of zero, while letters have a value equal to their position in the alphabet. So, A = 1, B = 2, and so on, through Z = 26. Here are example quicksum calculations for the packets "ACM" and "MID CENTRAL":

ACM: 1*1 + 2*3 + 3*13 = 46
MID CENTRAL: 1*13 + 2*9 + 3*4 + 4*0 + 5*3 + 6*5 + 7*14 + 8*20 + 9*18 + 10*1 + 11*12 = 650

Input

The input consists of one or more packets followed by a line containing only # that signals the end of the input. Each packet is on a line by itself, does not begin or end with a space, and contains from 1 to 255 characters.

Output

For each packet, output its quicksum on a separate line in the output.

Sample Input	*Sample Output*
ACM	46
MID CENTRAL	650
REGIONAL PROGRAMMING	4690
CONTEST	49
ACN	75
A C M	14
ABC	15
BBC	
#	

Source: ACM Mid-Central United States 2006.

IDs for online judges: POJ 3094, ZOJ 2812, UVA 3594.

Hint
Function *value*(*c*) is implemented as follows: if character *c* == '_', then return 0; otherwise, return the corresponding value of *c*: *c* – 'A' + 1.

The process is a loop. Each loop inputs a test case and calculates its *Quicksum*.

First, the location of character *c* and *Quicksum* are initialized 0, and string *s* is initialized NULL. Repeatedly input character *c* and add *c* into *s* until *c* is EOF or '\n'. If *s* is "#," the program ends.

$$Quicksum = \sum_{i=0}^{s.size-1} (i+1) * value(s[i])$$

1.5.4 A Contesting Decision

Judging a programming contest is hard work, with demanding contestants, tedious decisions, and monotonous work—not to mention the nutritional problems of spending 12 hours with only donuts, pizza, and soda for food. Still, it can be a lot of fun.

Software that automates the judging process is a great help, but the notorious unreliability of some contest software makes people wish that something better were available. You are part of a group trying to develop better, open-source, contest management software, based on the principle of modular design.

Your component is to be used for calculating the scores of programming contest teams and determining a winner. You will be given the results from several teams and must determine the winner.

Scoring

There are two components to a team's score. The first is the number of problems solved. The second is penalty points, which reflect the amount of time and incorrect submissions made before the problem is solved. For each problem solved correctly, penalty points are charged equal to the time at which the problem was solved plus 20 minutes for each incorrect submission. No penalty points are added for problems that are never solved.

So if a team solved problem 1 on their second submission at 20 minutes, they are charged 40 penalty points. If they submit problem 2 three times, but do not solve it, they are charged no penalty points. If they submit problem 3 once and solve it at 120 minutes, they are charged 120 penalty points. Their total score is two problems solved with 160 penalty points.

The winner is the team that solves the most problems. If teams tie for solving the most problems, then the winner is the team with the fewest penalty points.

Input

For the programming contest your program is judging, there are four problems. You are guaranteed that the input will not result in a tie between teams after counting penalty points.

```
Line 1: < nTeams >
Line 2: n+1 < Name > < p1Sub > < p1Time > < p2Sub > < p2Time > …
< p4Time >
```

The first element on the line is the team name, which contains no white space. Following that, for each of the four problems, is the number of times the team submitted a run for that problem and the time at which it was solved correctly (both integers). If a team did not solve a problem, the time will be zero. The number of submissions will be at least one if the problem was solved.

Output

The output consists of a single line listing the name of the team that won, the number of problems they solved, and their penalty points.

Sample Input	*Sample Output*
4 Stars 2 20 5 0 4 190 3 220 Rockets 5 180 1 0 2 0 3 100 Penguins 1 15 3 120 1 300 4 0 Marsupials 9 0 3 100 2 220 3 80	Penguins 3 475

Source: ACM Mid-Atlantic 2003.

IDs for online judges: POJ 1581, ZOJ 1764, UVA 2832.

Hint

Suppose the name of the winner is *wname*, the number of problems that winner solved is *wsol*, and the winner's penalty points is *wpt*; the name of the current team is *name*, the number of problems that the current team solved is *sol*, and the current team's penalty points is *pt*. The submission number of the current problem is *sub*, and the time at which of the current problem is solved is *time*.

If the problem is solved (*time* > 0), then we accumulate the number of problems the current team solved (++*sol*) and compute the current team's penalty points *pt* (*pt* += (*sub* – 1)*20 + *time*).

After we deal with a team's case, if the number of problems the current team solved is the most, or the current team and other teams all solved the most number of problems, and the current team is with the fewest penalty points, that is, (*sol* > *wsol* || (*sol* == *wsol* && *wpt* > *pt*)) holds, then the current team is set as winner, and its team name, the number of solved problems, and its penalty points are recorded, that is, *wname* = *name*, *wsol* = *sol*, *wpt* = *pt*.

Obviously, after we deal with all teams' cases, *wname*, *wsol*, and *wpt* are solutions to the problem.

1.5.5 Dirichlet's Theorem on Arithmetic Progressions

If a and d are relatively prime positive integers, the arithmetic sequence beginning with a and increasing by d, that is, a, $a + d$, $a + 2d$, $a + 3d$, $a + 4d$, …, contains infinitely many prime numbers. This fact is known as Dirichlet's theorem on arithmetic progressions, which had been conjectured by Johann Carl Friedrich Gauss (1777–1855) and was proved by Johann Peter Gustav Lejeune Dirichlet (1805–1859) in 1837.

For example, the arithmetic sequence beginning with 2 and increasing by 3, that is,

2, 5, 8, 11, 14, 17, 20, 23, 26, 29, 32, 35, 38, 41, 44, 47, 50, 53, 56, 59, 62, 65, 68, 71, 74, 77, 80, 83, 86, 89, 92, 95, 98, ….

contains infinitely many prime numbers:

2, 5, 11, 17, 23, 29, 41, 47, 53, 59, 71, 83, 89, …

Your mission, should you choose to accept it, is to write a program to find the nth prime number in this arithmetic sequence for given positive integers a, d, and n.

Input

The input is a sequence of data sets. A data set is a line containing three positive integers a, d, and n separated by a space. a and d are relatively prime. You may assume $a \leq 9307$, $d \leq 346$, and $n \leq 210$.

The end of the input is indicated by a line containing three zeros separated by a space. It is not a data set.

Output

The output should be composed of as many lines as the number of the input data sets. Each line should contain a single integer and should never contain extra characters.

The output integer corresponding to a data set a, d, n should be the nth prime number among those contained in the arithmetic sequence beginning with a and increasing by d.

For your information, it is known that the result is always less than 10^6 (1 million) under this input condition.

Sample Input	*Sample Output*
367 186 151	92809
179 10 203	6709
271 37 39	12037
103 230 1	103
27 104 185	93523
253 50 85	14503
1 1 1	2
9075 337 210	899429
307 24 79	5107
331 221 177	412717
259 170 40	22699
269 58 102	25673
0 0 0	

Source: ACM Japan 2006, Domestic.

ID for online judge: POJ 3006.

Hint

A test case consists of integers *a*, *d*, and *n* in an arithmetic sequence, and the end of the input is indicated by a line containing 0 0 0. Therefore, a *while* repetition statement is used for test cases. After the first *a*, *d*, and *n* are input, the program enters the *while*(*a* || *d* || *n*) loop. In the loop body, the steps are as follows:

1. Initialize the number of prime numbers *cnt* 0.
2. Construct an arithmetic sequence with *n* prime numbers through the loop statement *for*(*m* = *a*; *cnt* < *n*; *m* += *d*). The control variable *m* is initialized with *a*, and the loop-continuation condition is *cnt* < *n*. In each loop, if *m* is a prime number, then *cnt*++, and *d* is added to control variable *m*.
3. Output the *n*th prime number *m* – *d*. (Because of the *for* loop, output *m* – *d*.)
4. Input *a*, *d*, and *n* for the next arithmetic sequence.

1.5.6 The Circumference of the Circle

To calculate the circumference of a circle seems to be an easy task—provided you know its diameter. But what if you don't?

You are given the Cartesian coordinates of three noncollinear points in the plane.

Your job is to calculate the circumference of the unique circle that intersects all three points.

Input

The input file will contain one or more test cases. Each test case consists of one line containing six real numbers, $x_1, y_1, x_2, y_2, x_3, y_3$, representing the coordinates of the three points. The diameter of the circle determined by the three points will never exceed 1 million. Input is terminated by the end of the file.

Output

For each test case, print one line containing one real number telling the circumference of the circle determined by the three points. The circumference is to be printed accurately rounded to two decimals. The value of π is approximately 3.141592653589793.

Sample Input	*Sample Output*
0.0 –0.5 0.5 0.0 0.0 0.5	3.14
0.0 0.0 0.0 1.0 1.0 1.0	4.44
5.0 5.0 5.0 7.0 4.0 6.0	6.28
0.0 0.0 –1.0 7.0 7.0 7.0	31.42
50.0 50.0 50.0 70.0 40.0 60.0	62.83
0.0 0.0 10.0 0.0 20.0 1.0	632.24
0.0 –500000.0 500000.0 0.0 0.0 500000.0	3141592.65

Source: Ulm Local Contest 1996.

IDs for online judges: POJ 2242, ZOJ 1090.

Hint

The key to the problem is to find the center of a circle that intersects all three points. Suppose the Cartesian coordinates of three points are (x_0, y_0), (x_1, y_1), and (x_2, y_2), and the Cartesian coordinates of the center of the circle are (x_m, y_m). There are two solutions.

Determinant

Calculate the Cartesian coordinates of the center of the circle that intersects all three points:

$$x_m = \frac{x_1 + x_2}{2} + (y_2 - y_1) * \frac{\begin{vmatrix} y_1 - y_0 & \frac{x_2 - x_0}{2} \\ x_0 - x_1 & \frac{y_2 - y_0}{2} \end{vmatrix}}{\begin{vmatrix} y_1 - y_0 & y_1 - y_2 \\ x_0 - x_1 & x_2 - x_1 \end{vmatrix}}, \quad y_m = \frac{y_1 + y_2}{2} + (x_1 - x_2) * \frac{\begin{vmatrix} y_1 - y_0 & \frac{x_2 - x_0}{2} \\ x_0 - x_1 & \frac{y_2 - y_0}{2} \end{vmatrix}}{\begin{vmatrix} y_1 - y_0 & y_1 - y_2 \\ x_0 - x_1 & x_2 - x_1 \end{vmatrix}}$$

Based on this, the radius of the unique circle

$$r = \sqrt{(x_m - x_0)^2 + (y_m - y_0)^2}$$

and $2\pi r$ is the circumference of the unique circle that intersects all three points: (x_0, y_0), (x_1, y_1), and (x_2, y_2).

Theorem 1.1

The Cartesian coordinates of the center of the unique circle that intersects all three points (x_0, y_0), (x_1, y_1), and (x_2, y_2) are (x_m, y_m).

Proof

Suppose the Cartesian coordinate of the center of a circle is $P = (x_m, y_m)$; the perpendicular bisectors from P to $\overline{AB}$ and $\overline{BC}$ are $\overline{PN}$ and $\overline{PM}$, respectively. The point of intersection of $\overline{PN}$ and $\overline{AB}$ is N, and the point of intersection of $\overline{PM}$ and $\overline{BC}$ is M. Obviously, the Cartesian coordinates of M are $[(x_1 + x_2)/2, (y_1 + y_2)/2]$), and point $(y_2 - y_1, x_2 - x_1)$ is on $\overline{PM}$ (Figure 1.3).

Because $\overline{PM} \perp \overline{BC}$,

$$\frac{y_m - \frac{y_1 + y_2}{2}}{x_m - \frac{x_1 + x_2}{2}} * \frac{y_2 - y_1}{x_2 - x_1} = -1$$

Suppose

$$k = \frac{y_m - \frac{y_1 + y_2}{2}}{x_2 - x_1} = \frac{x_m - \frac{x_1 + x_2}{2}}{y_2 - y_1}$$

Now we need to prove

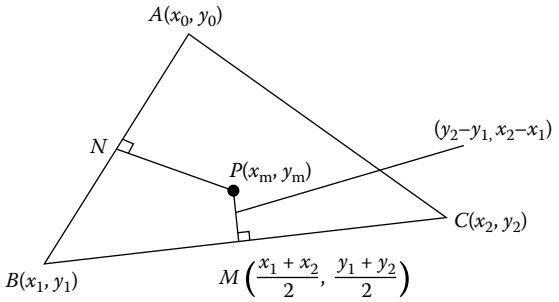


Figure 1.3 Three points and the center of the circle.

$$k = \frac{\begin{vmatrix} y_1 - y_0 & \frac{x_2 - x_0}{2} \\ x_0 - x_1 & \frac{y_2 - y_0}{2} \end{vmatrix}}{\begin{vmatrix} y_1 - y_0 & y_1 - y_2 \\ x_0 - x_1 & x_2 - x_1 \end{vmatrix}} (*)$$

Because $\overline{PN} \perp \overline{AB}$,

$$\frac{y_m - \frac{y_0 + y_2}{2}}{x_m - \frac{x_0 + x_2}{2}} * \frac{y_1 - y_0}{x_1 - x_0} = -1$$

Because

$$x_m = \frac{x_1 + x_2}{2} + (y_2 - y_1) * k,$$

and

$$y_m = \frac{y_1 + y_2}{2} + (x_1 - x_2) * k$$

$$\frac{(x_1 - x_2)k + \frac{y_2 - y_0}{2}}{(y_2 - y_1)k + \frac{x_2 - x_0}{2}} * \frac{y_1 - y_0}{x_1 - x_0} = -1$$

holds. Therefore, (*) holds.

Elementary Geometry

Suppose $a = |\overline{AB}|, b = |\overline{BC}|, c = |\overline{CA}|$, and

$$p = \frac{a + b + c}{2}.$$

Based on Heron's formula,

$$s = \sqrt{p(p-a)(p-b)(p-c)}$$

the formula for calculating the area of a triangle

$$s = \frac{a * b * \sin(\angle ab)}{2}$$

and sine theorem

$$\frac{a}{\sin(\angle bc)} = \frac{b}{\sin(\angle ac)} = \frac{c}{\sin(\angle ab)} = \text{diameter of circumcircle } d$$

we can calculate the diameter of circumcircle $d = (a*b*c)/(2*s)$ and the circumference of circumcircle $l = d*\pi$.

1.5.7 Vertical Histogram

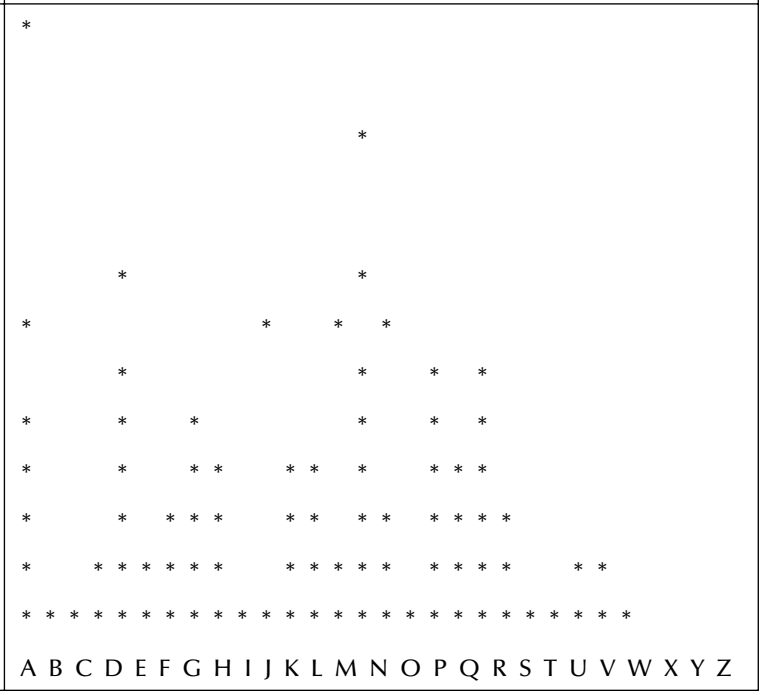
Write a program to read four lines of uppercase (i.e., all CAPITAL LETTERS) text input (no more than 72 characters per line) from the input file and print a vertical histogram that shows how many times each letter (but not blanks, digits, or punctuation) appears in the all-uppercase input. Format your output exactly as shown.

Input

Lines 1–4: Four lines of uppercase text, no more than 72 characters per line.

Output

Lines 1–?: Several lines with asterisks and spaces followed by one line with the uppercase alphabet separated by spaces. Do not print unneeded blanks at the end of any line. Do not print any leading blank lines.

Sample Input	*Sample Output*
THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG. THIS IS AN EXAMPLE TO TEST FOR YOUR HISTOGRAM PROGRAM. HELLO!	

Source: USACO, February 2003, Orange.

ID for online judge: POJ 2136.

Hint

The sequence of drawing a vertical histogram is from top to bottom and left to right. From top to bottom means dealing with every line in the frequency's descending order. From left to right means dealing with letters in the current line in the ordinal number's ascending order.

Suppose *cnt* is the frequency array for letters, where *cnt*[0] is the number of 'A', …, *cnt*[25] is the number of 'Z', and

$$Maxc = \max_{0 \le i \le 25} \{cnt[i]\}$$

that is, *Maxc* is the height of the highest "pillar" in the vertical histogram. The algorithm is as follows:

1. Input a test case and count array *cnt*;
2. Getting the value of *Maxc*;
3. From the highest "pillar" in the vertical histogram, draw the vertical histogram from top to bottom. A repetition statement *for* (int $i = 1$; $i \le Maxc$; i++) implements it as follows:
   - Find the right boundary for the current line. That is, in array *cnt*, from 25 to 0, find the first letter whose serial number is $l1–1$, where $cnt[l1–1] > Maxc–i$.
   - For letters whose serial numbers are in $[0..l1–1]$, if $cnt[j] > Maxc – i$ $(0 \le j \le l1 – 1)$, then output '*_'; otherwise, output '_ _'.
4. Output the last line 'A_B_..... _Z'.

1.5.8 Ugly Numbers

Ugly numbers are numbers whose only prime factors are 2, 3, or 5. The sequence 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, … shows the first 10 ugly numbers. By convention, 1 is included.

Given the integer n, write a program to find and print the nth ugly number.

Input

Each line of the input contains a positive integer n ($n \le 1500$). Input is terminated by a line with $n = 0$.

Output

For each line, output the nth ugly number. Don't deal with the line with $n = 0$.

Sample Input	*Sample Output*
1	1
2	2
9	10
0	

Source: New Zealand 1990, Division I.

IDs for online judges: POJ 1338, UVA 136.

Hint
An offline method is used to solve the problem. The first 1500 ugly numbers are calculated and stored in array $a[1 .. 1500]$.

Suppose the upper limit for the largest ugly number $limit = 1{,}000{,}000{,}000$. The outer loop (control variable i) enumerates multiples of 2. For each time, $i \leftarrow i*2$ is performed. The loop-continuation condition is $i < limit$. The middle loop (control variable j) enumerates multiples of 3. For each time, $j \leftarrow j*3$ is performed. The loop-continuation condition is $i*j < limit$. The inner loop (control variable k) enumerates multiples of 5. For each time, ugly number $i*j*k$ is stored in array a and $k \leftarrow k*5$ is performed. The loop-continuation condition is $i*j*k < limit$.

Then array a is sorted such that $a[x]$ is the xth large ugly number ($1 \leq x \leq 1500$).

1.5.9 Number Sequence

A single positive integer i is given. Write a program to find the digit located in the position i in the sequence of number groups $S_1S_2 \ldots S_k$. Each group S_k consists of a sequence of positive integer numbers ranging from 1 to k, written one after another.

For example, the first 80 digits of the sequence are as follows:

11212312341234512345612345671234567812345678912345678910123456789101112345678910

Input

The first line of the input file contains a single integer t ($1 \leq t \leq 10$), the number of test cases, followed by one line for each test case. The line for a test case contains the single integer i ($1 \leq i \leq 2147483647$).

Output

There should be one output line per test case containing the digit located in the position i.

Sample Input	*Sample Output*
2	2
8	2
3	

Source: ACM Tehran 2002, First Iran Nationwide Internet Programming Contest.

IDs for online judges: POJ 1019, ZOJ 1410.

Hint
First, two functions are implemented. The first function is to calculate the length for the first j groups (i.e., the number of digits for the first j groups) and is stored in an array. The second function is to return the digit located in the position l in a group S_m. Then dichotomy is used to find the group S_n containing the digit located in the position i. Finally, in the group S_n, the digit located in the position i is returned.

Chapter 2

Simple Simulation

In the real world, there are many problems that we can solve by simulating their processes. Such problems are called simulation problems. For these problems, solution procedures and rules are showed in problem descriptions. Programs must simulate procedures or implement rules based on descriptions.

Normally there are two kinds of simulations: stochastic simulation and process simulation.

Problems for stochastic simulation show or imply probabilities. Programmers make use of random functions and round functions to set the random value for a range, making the random value meet the probability as a parameter. Then programmers design the algorithm by simulating the mathematical model. Because of uncertainty, there are fewer problems for stochastic simulation in programming contests.

Problems for process simulation require programmers to design parameters for mathematical models, and to observe changes of states caused by parameters. Programmers design algorithms based on process simulation. Programs depend entirely on authenticity and correctness of the process simulation without any uncertainties.

This chapter focuses on process simulation. There are three kinds of process simulation:

1. Simulation of direct statement
2. Simulation by sieve method
3. Simulation by construction

2.1 Simulation of Direct Statement

For problems for simulation of direct statement, programmers are required to solve them by strictly following rules showed in the problems' descriptions. Programmers must read such problems carefully and simulate processes based on descriptions. A problem for simulation of direct statement gets harder as the number of rules increases. It causes the amount of code to grow and become more illegible.

2.1.1 Speed Limit

Bill and Ted are taking a road trip. But the odometer in their car is broken, so they don't know how many miles they have driven. Fortunately, Bill has a working stopwatch, so they can record their speed and the total time they have driven. Unfortunately, their record-keeping strategy is a

little odd, so they need help computing the total distance driven. You are to write a program to do this computation.

For example, if their log shows

Speed in Miles per Hour	*Total Elapsed Time in Hours*
20	2
30	6
10	7

this means they drove 2 hours at 20 miles per hour, then $6 - 2 = 4$ hours at 30 miles per hour, then $7 - 6 = 1$ hour at 10 miles per hour. The distance driven is then $(2)(20) + (4)(30) + (1)(10) = 40 + 120 + 10 = 170$ miles. Note that the total elapsed time is always since the beginning of the trip, not since the previous entry in their log.

Input

The input consists of one or more data sets. Each set starts with a line containing an integer n, $1 \leq n \leq 10$, followed by n pairs of values, one pair per line. The first value in a pair, s, is the speed in miles per hour, and the second value, t, is the total elapsed time. Both s and t are integers, $1 \leq s \leq 90$ and $1 \leq t \leq 12$. The values for t are always in strictly increasing order. A value of –1 for n signals the end of the input.

Output

For each input set, print the distance driven, followed by a space, followed by the word *miles*.

Sample Input	*Sample Output*
3 20 2 30 6 10 7 2 60 1 30 5 4 15 1 25 2 30 3 10 5 –1	170 miles 180 miles 90 miles

Source: ACM Mid-Central United States 2004.

IDs for online judges: POJ 2017, ZOJ 2176, UVA 3059.

Analysis

This is a simple problem of direct statement. We can simulate the stopwatch's running to compute the total distance driven: if the last total elapsed time in hours is z, the current speed in miles per hour is x, and the current total elapsed time in hours is y, then the current distance driven is $(y - z)^*x$, and we add it to the total distance driven.

Program

```
#include <iostream>                    // Preprocessor Directive
using namespace std;                   // Using C++ Standard Library
int main()                            //Main Function
{
   int n, i, x, y, z, ans;
   // Multiple test cases are dealt with by a while loop statement
   while (cin >> n, n > 0)
   {
      ans = z = 0;
    // Simulate the stopwatch to calculate
      for (i = 0; i < n; i++)             // Input and calculate the
current data set
      {
         cin >> x >> y;               //Input the speed and the total
elapsed time
         ans += (y - z) * x;           // Accumulate the distance driven
         z = y;                       //Record the total elapsed time
      }
      cout << ans << " miles" << endl;  //Output the distance driven for
the current data set
   }
   return 0;
}
```

2.1.2 Ride to School

Many graduate students of Peking University are living on Wanliu Campus, which is 4.5 kilometers from the main campus—Yanyuan. Students in Wanliu have to either take a bus or ride a bike to go to school. Due to the bad traffic in Beijing, many students choose to ride a bike.

We may assume that all the students except "Charley" ride from Wanliu to Yanyuan at a fixed speed. Charley is a student with a different riding habit—he always tries to follow another rider to avoid riding alone. When Charley gets to the gate of Wanliu, he will look for someone who is setting off to Yanyuan. If he finds someone, he will follow that rider, or if not, he will wait for someone to follow. On the way from Wanliu to Yanyuan, at any time if a faster student surpasses Charley, he will leave the rider he is following and speed up to follow the faster one.

We assume the time that Charley gets to the gate of Wanliu is zero. Given the set-off time and speed of the other students, your task is to give the time when Charley arrives at Yanyuan.

Input

There are several test cases. The first line of each case is $N(1 \le N \le 10{,}000)$ representing the number of riders (excluding Charley). $N = 0$ ends the input. The following N lines are information of N different riders, in such format:

$$V_i \text{ [TAB] } T_i$$

V_i is a positive integer ≤ 40, indicating the speed of the ith rider (kilometers per hour). T_i is the set-off time of the ith rider, which is an integer and counted in seconds. In any case, it is ensured that there always exists a nonnegative T_i.

Output

The output is one line for each case: the arrival time of Charley. Round up (ceiling) the value when dealing with a fraction.

Sample Input	*Sample Output*
4 20 0 25 –155 27 190 30 240 2 21 0 22 34 0	780 771

Source: ACM Beijing 2004, Preliminary.

IDs for online judges: POJ 1922, ZOJ 2229.

Analysis

There is no mathematical formula to solve the problem. We can calculate the arrival time of Charley by simulating each student leaving from Wanliu to Yanyuan. For each test case, the time that Charley gets to the gate of Wanliu is zero. From it we calculate the arrival time of each student. Obviously, the earliest arrival time is the arrival time of Charley.

Suppose *min* is the earliest arrival time for the first $i - 1$ riders, the speed of the ith rider is v, and the set-off time of the ith rider is t. Then the time when the ith rider arrives at Yanyuan is $x = t + (4.5*3600)/v$. If $x < min$, then *min* is adjusted as x. Obviously, after the arrival time of all riders is calculated, *min* is the arrival time of Charley.

There is a trap in the test data. If T_i is a negative integer, we should neglect it. It doesn't affect the arrival time of Charley.

Program

```
#include <iostream>                    // Preprocessor Directive
#include <cmath>
using namespace std;                   // Using C++ Standard Library
int main()                            //Main function
{
    const double DISTANCE = 4.50;     //The distance between Yanyuan and
Wanliu
    while(true)                       //A while statement deals with test
cases
    {
        int n;                       //The number of riders except Charley
        scanf("%d", &n);
```

```
        if (n == 0) break;              //Input ends
        double v, t, x, min = 1e100;      //min is initialized 10^100
        for(int i = 0; i < n; ++i)         // A while statement deals
with riders
        {
           scanf("%lf%lf", &v, &t);    // the speed and the set off time
of the i-th rider
// Calculate time x when the i-th rider arrives at Yanyuan. If x<min,
then min is adjusted to x.
           if (t >= 0 && (x = DISTANCE * 3600 / v + t) < min)
           min = x;
        }
        printf("%.0lf\n", ceil(min));     //Output the arrival time of
Charley
    }
    return 0;
}
```

2.2 Simulation by Sieve Method

The simulation by sieve method is to get constraints in the problem description, and such constraints constitute a sieve. And then all possible solutions are put in the sieve to filter out solutions that do not meet constraints from time to time. Finally, solutions settling in the sieve are solutions to the problem. The structure and idea for the simulation by sieve method is concise and clear, but also blind. Therefore, maybe the time efficiency is not good. The key to the simulation by sieve method is to find the constraints. Any errors and omissions will lead to failure. Because filtering rules do not need complex algorithm design, such problems are usually simple simulation problems.

2.2.1 Self-Numbers

In 1949, the Indian mathematician D.R. Kaprekar discovered a class of numbers called self-numbers. For any positive integer n, define $d(n)$ to be n plus the sum of the digits of n. (The d stands for digitadition, a term coined by Kaprekar.) For example, $d(75) = 75 + 7 + 5 = 87$. Given any positive integer n as a starting point, you can construct the infinite increasing sequence of integers n, $d(n)$, $d(d(n))$, $d(d(d(n)))$, …. For example, if you start with 33, the next number is $33 + 3 + 3 = 39$, the next is $39 + 3 + 9 = 51$, the next is $51 + 5 + 1 = 57$, and so on, and you generate the sequence

33, 39, 51, 57, 69, 84, 96, 111, 114, 120, 123, 129, 141, …

The number n is called a generator of $d(n)$. In the sequence above, 33 is a generator of 39, 39 is a generator of 51, 51 is a generator of 57, and so on. Some numbers have more than one generator; for example, 101 has two generators, 91 and 100. A number with no generators is a self-number. There are 13 self-numbers less than 100: 1, 3, 5, 7, 9, 20, 31, 42, 53, 64, 75, 86, and 97.

Input

There is no input for this problem.

Output

Write a program to output all positive self-numbers less than 10,000 in increasing order, one per line.

Sample Input	Sample Output
	1 3 5 7 9 20 31 42 53 64 | a lot more numbers | 9903 9914 9925 9927 9938 9949 9960 9971 9982 9993

Source: ACM Mid-Central United States 1998.

IDs for online judges: POJ 1316, ZOJ 1180, UVA 640.

Analysis

The simulation by sieve method is used to solve the problem. Suppose the sieve is array g, where $g[y] = x$ means y is a number in ascending sequence for x. Based on $d(x) = x +$ the sum of the digits of x, a subprogram *generate_sequence*(x) is to generate the ascending sequence [$d(x)$, $d(d(x))$, $d(d(d(x)))$, …] for x. Suppose x is the generation number for every number in the sequence:

$$g[d(x)] = g[d(d(x))] = g[d(d(d(x)))] = \ldots = x$$

If a number is in ascending sequence for x, it is not a self-number and should be sieved from sieve g. The process will repeat until the generated number ≥1000 or the generated number has been generated before ($g[x] \neq x$). If x has been generated, it is not a self-number.

The algorithm is as follows:

First, $g[i]$ is initialized as i ($1 \leq i \leq 1000$). Then *generate_sequence*(1) … *generate_sequence*(1000) are called to calculate g[1..1000]. Finally, numbers left in the sieve, that is, $g[x]==x$, are self-numbers.