Chapter 2 Practice for Simulation Problems

Yonghui Wu

School of Computer Science, Fudan University

yhwu@fudan.edu.cn

Wechat: 13817360465

- 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 or rules are showed in problem descriptions. Programs must simulate procedures or implement rules based on descriptions.

Chapter 2 Practice for Simulation Problems

- 2.1 Simulation of Direct Statement
- 2.2 Simulation by Sieve Method
- 2.3 Construction Simulation

2.1 Simulation of Direct Statement

- For problems for simulation of direct statement, programmers are required to solve these problems strictly implementing 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 up and get more illegible.

- Two kinds of simulations of direct statement:
  - simulations based on a sequence of instructions
  - simulations based on a sequence of time intervals

2.1.1 The Hardest Problem Ever

- Source: ACM South Central USA 2002
- IDs for Online Judges: POJ 1298, ZOJ 1392, UVA 2540

- Julius Caesar lived in a time of danger and intrigue. The hardest situation Caesar ever faced was keeping himself alive. In order for him to survive, he decided to create one of the first ciphers. This cipher was so incredibly sound, that no one could figure it out without knowing how it worked.
- You are a sub captain of Caesar's army. It is your job to decipher the messages sent by Caesar and provide to your general. The code is simple. For each letter in a plaintext message, you shift it five places to the right to create the secure message (i.e., if the letter is 'A', the cipher text would be 'F'). Since you are creating plain text out of Caesar's messages, you will do the opposite:
- Cipher text: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
- Plain text: V W X Y Z A B C D E F G H I J K L M N O P Q R S T U
- Only letters are shifted in this cipher. Any non-alphabetical character should remain the same, and all alphabetical characters will be upper case.

- **Input**
- Input to this problem will consist of a (non-empty) series of up to 100 data sets. Each data set will be formatted according to the following description, and there will be no blank lines separating data sets. All characters will be uppercase.
- A single data set has 3 components:
- Start line - A single line, "START"
- Cipher message - A single line containing from one to two hundred characters, inclusive, comprising a single message from Caesar
- End line - A single line, "END"
- Following the final data set will be a single line, "ENDOFINPUT".
- **Output**
- For each data set, there will be exactly one line of output. This is the original message by Caesar.

Analysis

- The problem is solved by strictly implementing the rule in the problem description.
- The rule creating plain text out of Caesar's messages is as follow.
  - A letter in the plain text='A'+(A letter in the cipher text-'A'+21)% 26.

2.1.2 Rock-Paper-Scissors Tournament

- Source: Waterloo local 2005.09.17
- IDs for Online Judges: POJ 2654, UVA 10903

- Rock-Paper-Scissors is game for two players, A and B, who each choose, independently of the other, one of *rock, paper,* or *scissors*. A player chosing *paper* wins over a player chosing *rock*; a player chosing *scissors* wins over a player chosing *paper*; a player chosing *rock* wins over a player chosing*scissors*. A player chosing the same thing as the other player neither wins nor loses.

A tournament has been organized in which each of n players plays k rock-scissors-paper games with each of the other players - $k\frac{n(n-1)}{2}$ games in total. Your job is to compute the *win average* for each player, defined as $\frac{w}{w+l}$ where w is the number of games won, and l is the number of games lost, by the player.

- **Input**
- Input consists of several test cases. The first line of input for each case contains $1 \le n \le 100$, $1 \le k \le 100$ as defined above. For each game, a line follows containing p_1, m_1, p_2, m_2. $1 \le p_1 \le n$ and $1 \le p_2 \le n$ are distinct integers identifying two players; m_1 and m_2 are their respective moves ("rock", "scissors", or "paper"). A line containing 0 follows the last test case.
- **Output**
- Output one line each for player 1, player 2, and so on, through player n, giving the player's win average rounded to three decimal places. If the win average is undefined, output "-". Output an empty line between cases.

Analysis

- This is a problem for simulation of direct statement.
- In the problem description,
  - a player chosing *paper* wins over a player chosing *rock*;
  - a player chosing *scissors* wins over a player chosing *paper*;
  - and a player chosing *rock* wins over a player chosing*scissors*.
  - A player chosing the same thing as the other player neither wins nor loses.

- A tournament has been organized in which each of n players plays k rock-scissors-paper games with each of the other players
- $k\frac{n(n-1)}{2}$ games in total.

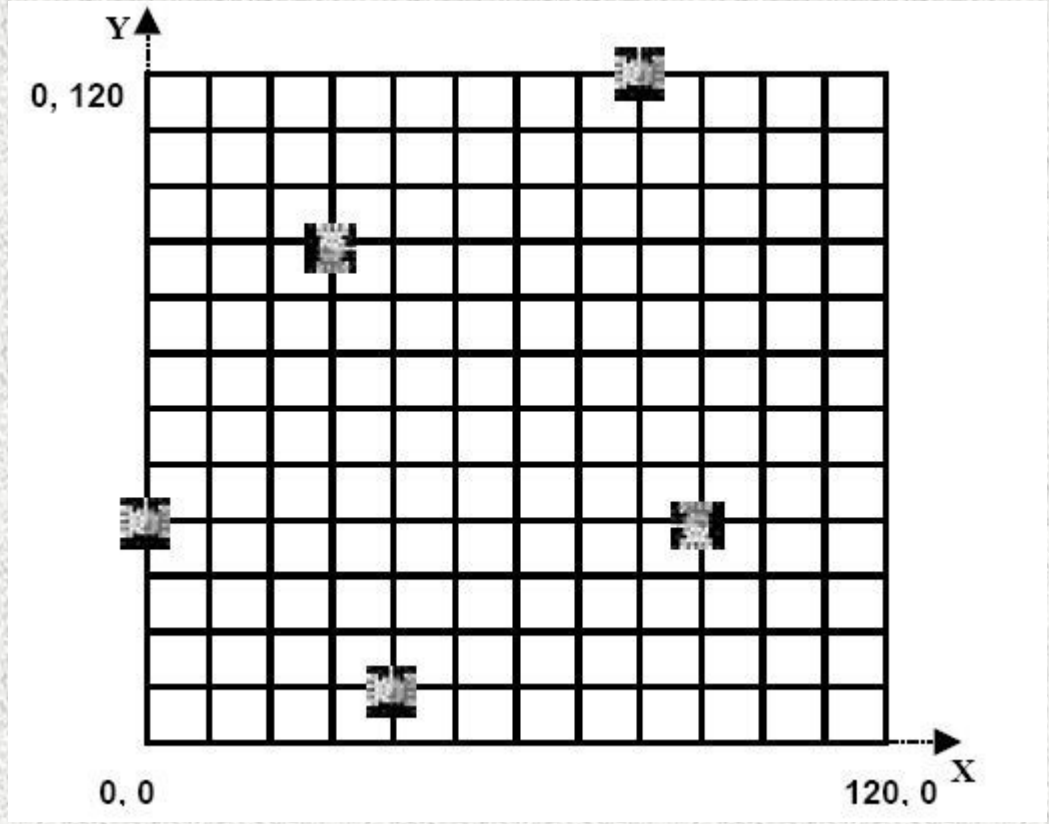
- For each test case, cases are input one by one; and for for each player, the number of games won and the number of games lost are accumulated.
- Finally, the win average for each player is calculated.
  - For a player, if the number of games won and the number of games lost are all 0, then the win average is undefined;
  - else the win average for the player is $\frac{w}{w+l}$

2.1.3 Robocode

- Source: ACM Beijing 2005
- IDs for Online Judges: POJ 2729， UVA 3466

- Robocode is an educational game designed to help learn Java. The players write programs that control tanks fighting with each other on a battlefield. The idea of this game may seem simple, but it takes a lot of effort to write a winning tank's program. Today we are not going to write an intelligent tank, but to design a simplified robocode game engine.

- Assuming the whole battlefield is 120*120 (pixel). Each tank can ONLY move in vertical and horizontal direction on the fixed path (There are paths every 10 pixels in the battlefield in both vertical and horizontal direction. In all there are 13 vertical paths and 13 horizontal paths available for tanks, as shown in Figure 1). The shape and size of the tank are negligible and one tank has (x, y) $(x, y \in [0, 120])$ representing its coordinate position and α ($\alpha \in \{0, 90, 180, 270\}$) representing its facing direction ($\alpha = 0, 90, 180$ or 270 means facing right, up, left or down respectively). They have a constant speed of 10 pixels/second when they move and they can't move out of the boundary (they will stop moving, staying in the direction that they are currently facing, when touching any boundary of the battlefield). The tank can shoot in the direction it's facing whether it's moving or still. The shot moves at the constant speed 20 pixel/second and the size of the shot is also negligible. It will explode when it meets a tank on the path. It's possible for more than one shot to explode in the same place if they all reach a tank at the exact same time. The tank being hit by the explosion will be destroyed and will be removed from the battlefield at once. A shot exploding or flying out of the boundary will also be removed.


Y
0, 120
X
0, 0
120, 0

- When the game begins, all the tanks are stopped at different crosses of the vertical and horizontal paths. Given the initial information of all the tanks and several commands, your job is to find the winner -- the last living tank when all the commands are executed (or omitted) and no shot exists in the battlefield (meaning that no tank may die in the future).

- **Input**
- There are several test cases. The battlefield and paths are all the same for all test cases as shown in Figure 1. Each test case starts with integers N ($1 \le N \le 10$) and M ($1 \le M \le 1000$), separated by a blank. N represents the number of the tanks playing in the battlefield, and M represents the number of commands to control the tanks' moving. The following N lines give the initial information (at time 0) of each tank, in the format:
- *Name x y α*
- The Name of a tank is consisted of no more than 10 letters. x, y, α are integers and $x, y \in \{0, 10, 20, ..., 120\}$, $\alpha \in \{0, 90, 180, 270\}$. Each field is separated by a blank.

- The following *M* lines give commands in such format:
- *Time Name Content*
- Each field is separated by a blank. All the commands are giving in the ascending order of *Time* ($0 \leq Time \leq 30$), which is a positive integer meaning the timestamp when the commands are sent. *Name* points out which tank will receive the command. The *Content* has different types as follows:

MOVE	When receiving the command, the tank starts to move in its facing direction. If the tank is already moving, the command takes no effect.
STOP	When receiving the command, the tank stops moving. If the tank is already stopped, the command takes no effect.
TURN *angle*	When receiving the command, the tank changes the facing direction α to be $((\alpha + angle + 360) \bmod 360)$, no matter whether it is moving or not. You are guaranteed that $((\alpha + angle + 360) \bmod 360) \in \{0, 90, 180, 270\}$. TURN command doesn't affect the moving state of the tank.
SHOOT	When receiving the command, the tank will shoot one shot in the direction it's facing.

- Tanks take the corresponding action as soon as they receive the commands. E.g., if the tank at (0, 0), $\alpha = 90$, receives the command MOVE at time 1, it will start to move at once and will reach (0, 1) at time 2. Notice that a tank could receive multiple commands in one second and take the action one by one. E.g., if the tank at (0, 0), $\alpha = 90$, receives a command sequence of "TURN 90; SHOOT; TURN -90", it will turn to the direction $\alpha = 180$, shoot a shot and then turn back. If the tank receives a command sequence of "MOVE; STOP", it will keep still in the original position.

- Some more notes you need to pay attention:
- If a tank is hit by an explosion, it will take no action to all the commands received at that moment. Of course, all the commands sent to the already destroyed tank should also be omitted.
- Although the commands are sent at discrete seconds, the movement and explosions of tanks and shots happen in the continuous time domain.
- No two tanks will meet on the path guaranteed by the input data, so you don't need to consider about that situation.
- All the input contents will be legal for you.
- A test case with $N = M = 0$ ends the input, and should not be processed.

- **Output**
- For each test case, output the winner's name in one line. The winner is defined as the last living tank. If there is no tank or more than one tank living at the end, output "NO WINNER!" in one line.

Analysis

- The problem is a simulation problem based on a sequence of time intervals.
  - For each command, $0 \leq Time \leq 30$ (seconds), and states may be changed after the last command is sent. Therefore Robocode must be simulated at most 45 seconds.
  - If a tank at (0, 0) shoots at a tank at (0, 1), and the tank at (0, 1) moves to the tank at (0, 0), then the moving tank is shot after it moves 10/3 pixels, and after 0.5 seconds.
  - The map should be enlarged 6 times, and states should be simulated every 1/6 seconds.

Analysis

- Attributes for tanks and shots are as follows:
  - positions,
  - directions,
  - move (or stop),
  - removed (or unremoved).

Analysis

- Since *Time* 0, commands are processed one by one.
- If the timestamp when the current command is sent is t_2, and the timestamp when the last command is sent is t_1, states from t_1 to t_2 must be simulated.

Then attributes for the tank receiving the current command are set:

✧ If the command is "MOVE" command, the tank receiving the command moves in its facing direction;

✧ If the command is " STOP " command, the tank receiving the command stops moving;

✧ If the command is "SHOOT" command and the tank receiving the command isn't removed, then a shot is added, and its attributes are same as attributes of a tank, except move;

If the command is TURN *angle*, then the tank receiving the command adjusts its facing direction as (the original number of direction + ($\frac{angle}{90}$ % 4) +4) % 4, where the number of direction is $\frac{angle}{90}$.

- After all commands are processed, states are simulated 15 seconds continuously.
- Finally, the number of last living tanks is calculated. If all tanks are removed, or more than one tank live, then output "NO WINNER!"; else output the last living tank.

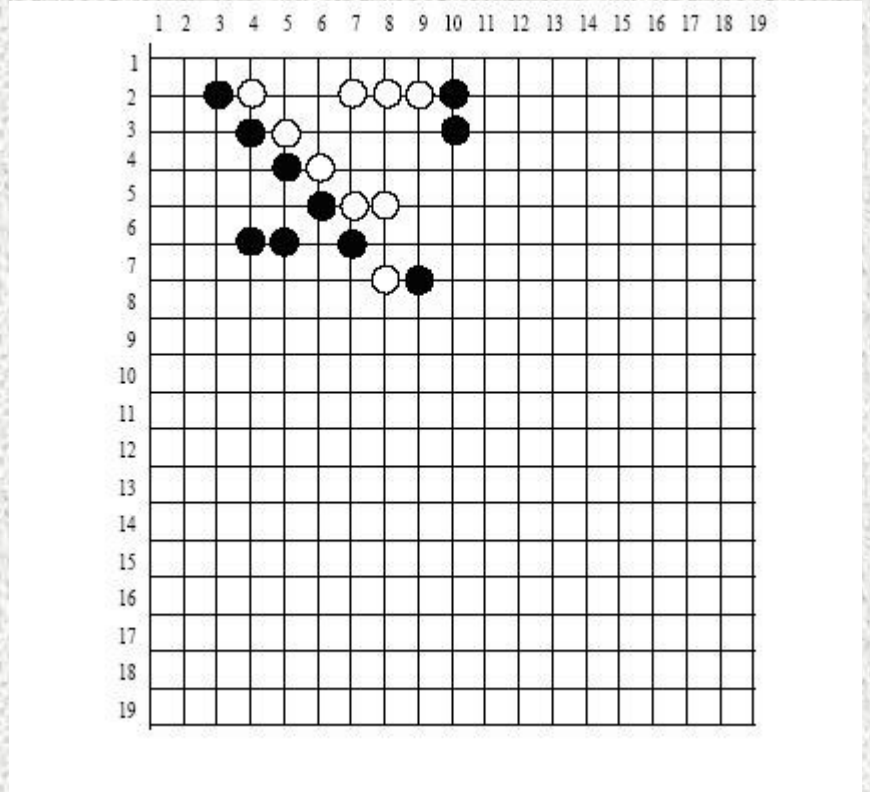
2.2 Simulation by Sieve Method

- In some problems, constraints are given in descriptions. And these constraints constitute a sieve. All possible solutions are put on the sieve to filter out solutions which do not meet constraints. Finally, solutions settling on the sieve are solutions to the problem. The method solving such problems are called simulation by sieve method. The structure and idea for simulation by sieve method is concise and clear, but it is also blindly. Therefore, its time efficiency maybe is not good. The key to 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 The Game

- **Source: ACM Tehran Sharif 2004 Preliminary**
- **IDs for Online Judge: POJ 1970, ZOJ 2495**

- A game of Renju is played on a 19×19 board by two players. One player uses black stones and the other uses white stones. The game begins in an empty board and two players alternate in placing black stones and white stones. Black always goes first. There are 19 horizontal lines and 19 vertical lines in the board and the stones are placed on the intersections of the lines.
- Horizontal lines are marked 1, 2, ..., 19 from up to down and vertical lines are marked 1, 2, ..., 19 from left to right.

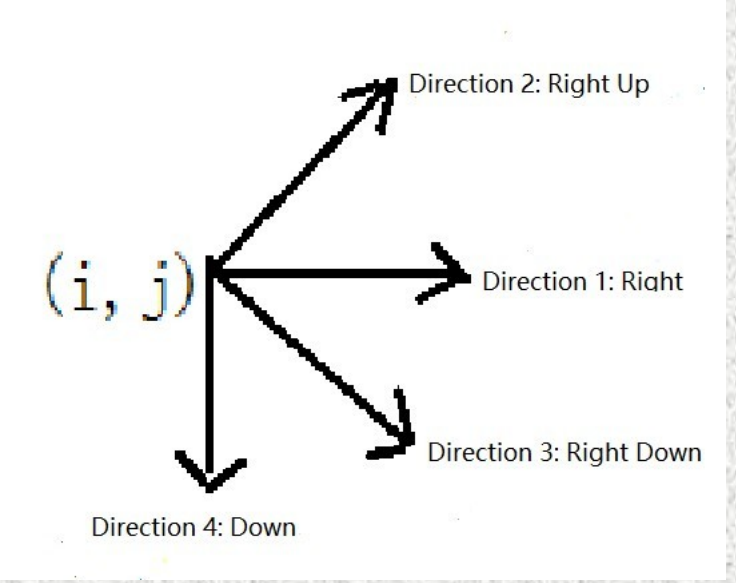

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19

- The objective of this game is to put five stones of the same color consecutively along a horizontal, vertical, or diagonal line. So, black wins in the above figure. But, a player does not win the game if more than five stones of the same color were put consecutively.
- Given a configuration of the game, write a program to determine whether white has won or black has won or nobody has won yet. There will be no input data where the black and the white both win at the same time. Also there will be no input data where the white or the black wins in more than one place.

- **Input**
- The first line of the input file contains a single integer t ($1 \leq t \leq 11$), the number of test cases, followed by the input data for each test case. Each test case consists of 19 lines, each having 19 numbers. A black stone is denoted by 1, a white stone is denoted by 2, and 0 denotes no stone.
- **Output**
- There should be one or two line(s) per test case. In the first line of the test case output, you should print 1 if black wins, 2 if white wins, and 0 if nobody wins yet. If black or white won, print in the second line the horizontal line number and the vertical line number of the left-most stone among the five consecutive stones. (Select the upper-most stone if the five consecutive stones are located vertically.)

Aanlysis

- Initially all stones on the 19×19 board constitute a sieve. Every stone is scanned from top to down and from left to right. If there is a stone at position (*i*, *j*), its adjacent stones in direction *k* is analyzed (0≤*k*≤3, 0≤*i*, *j*≤18).



- The objective of this game is to put five stones of the same color consecutively along a horizontal, vertical, or diagonal line. Therefore, the constraint conditions wining a game are as follows.
  - The number at position (*i*, *j*) is different with the number at the adjacent position in the opposite direction for direction *k*;
  - From (*i*, *j*) and along direction *k*, 5 positions are in the board;
  - From (*i*, *j*) and along direction *k*, numbers at 5 continuous positions are same, and the number at the 6th position is different, or the 6th position is out of the board;
- If the above constraint conditions hold, the stone at position (*i*, *j*) wins the game. If 4 directions are examined and the above constraint conditions don't hold, the stone at position (*i*, *j*) is filtered out.
- If all stones are filtered out, nobody wins the game.

2.3 Construction Simulation

- Construction simulation is a kind of relatively complex simulation method. It requires a mathematical model to represent and solve a problem.
- We need to design parameters of the model, and calculate simulation result.
- Because such mathematical models represent objects and their relationships accurately, the efficiencies are relatively high.

2.3.1 Packets

- Source: ACM Central Europe 1996
- IDs for Online Judges: POJ 1017, ZOJ 1307, UVA 311

- A factory produces products packed in square packets of the same height *h* and of the sizes 1*1, 2*2, 3*3, 4*4, 5*5, 6*6. These products are always delivered to customers in the square parcels of the same height *h* as the products have and of the size 6* 6. Because of the expenses it is the interest of the factory as well as of the customer to minimize the number of parcels necessary to deliver the ordered products from the factory to the customer. A good program solving the problem of finding the minimal number of parcels necessary to deliver the given products according to an order would save a lot of money. You are asked to make such a program.

- **Input**
- The input file consists of several lines specifying orders. Each line specifies one order. Orders are described by six integers separated by one space representing successively the number of packets of individual size from the smallest size 1*1 to the biggest size 6*6. The end of the input file is indicated by the line containing six zeros.
- **Output**
- The output file contains one line for each line in the input file. This line contains the minimal number of parcels into which the order from the corresponding line of the input file can be packed. There is no line in the output file corresponding to the last ``null'' line of the input file.

Analysis

- The simulation problem is solved by construction method.
- The greedy method is also used. Packets are packed in parcels in descending order by size.
- Because parcels' size is 6*6, each packet whose size is 4*4, 5*5, or 6*6 is packed in a parcel.

- The strategy is as follow.
  - A packet whose size is 6*6 is packed in a parcel.
  - A packet whose size is 5*5 is packed in a parcel. Packets whose size are 1*1 are packed into the remaining space for the parcel.
  - A packet whose size is 4*4 is packed in a parcel. Packets whose size are 2*2 are packed into the remaining space for the parcel. If there is no packet whose size is 2*2, packets whose size are 1*1 are packed into the remaining space for the parcel.
  - 4 packets whose size are 3*3 are packed in a parcel.

The algorithm is as follow.

Suppose the number of packets whose size are $i*i$ is a_i $(1 \le i \le 6)$.

The number of parcels in which packets whose size are 6*6, 5*5, 4*4, and 3*3 are packed is $M = a_6 + a_5 + a_4 + \left\lceil \frac{a_3}{4} \right\rceil$.

The number of packets whose size is 2*2 and which can be pack in above M parcels is $L_2 = a_4 * 5 + u[a_3 \bmod 4]$, where $u[0]=0$, $u[1]=5$, $u[2]=3$, $u[3]=1$. If there are remaining packets whose size is 2*2 ($a_2 > L_2$), the remaining packets are packed in new $\left\lceil \frac{a_2 - L_2}{9} \right\rceil$ parcels. And $M += \left\lceil \frac{a_2 - L_2}{9} \right\rceil$.

The number of packets whose size is 1*1 and which can be pack in above M parcels is $L_1 = M*36 - a_6*36 - a_5*25 - a_4*16 - a_3*9 - a_2*4$. If there are remaining packets whose size is 1*1 ($a_1 > L_1$), the remaining packets are packed in new $\left\lceil \frac{a_1 - L_1}{36} \right\rceil$ parcels.

And $M += \left\lceil \frac{a_1 - L_1}{36} \right\rceil$.

Obviously, M is the minimal number of parcels.