2005 Consortium for Computing Sciences in Colleges
Programming Contest
Saturday, November 12th
Lenoir-Rhyne College
Hickory, North Carolina

There are six (6) problems in this packet. Each team member should have a copy of the problems. These problems are NOT necessarily sorted by difficulty. You may solve them in any order. Each team will receive a football in the color indicated below upon solving a problem correctly.

Remember input/output for the contest will be from `stdin` to `stdout`. `stderr` as well as any external file output will be ignored.

Have Fun & Good Luck! ☺

Problem 1. The Real Jones Average *(purple football)*

Problem 2. Sudoko! *(blue football)*

Problem 3. Waiting for Soarin' *(orange football)*

Problem 4. The New CCSC Election *(pink football)*

Problem 5. David's Electrical Outlets *(green football)*

Problem 6. Rotating my Graphic *(yellow football)*

Problem 1

The Real Jones Average

Professor Jones has discovered that an average isn't always the best indicator of how his students have done on his exams. He is interested in developing a program that will display the *median* value of his students on his exams.

The median value isn't impacted as much by outlier scores, those scores that lie very far below or above the rest of the class distribution. To calculate it, you merely find the middle value in an ordered distribution of the scores. If there are an even number of scores, it is the average of the two scores adjacent to the middle location. For example, if there are 26 scores, the median is the average of the thirteenth and fourteenth scores.

Input
The input will be one or more lines. Each line will begin with a single nonnegative integer value representing a student's exam score on a scale from 0 up to 100. The end of input will be triggered by a negative 32-bit integer.

Output Corresponding to Sample Input
The output will be one line consisting of a single number, the median of the students' scores. Include exactly one digit after a decimal point followed by a % sign.

Sample Input
```
87
96
85
25
42
41
100
82
85
86
78
79
-1
```

Sample Output
```
83.5%
```

Problem 2

Sudoku!

Sudoku is the number-placing game that is taking the world by storm! Have you played yet? Each Sudoku game consists of a 9 x 9 matrix that consists of nine 3 x 3 submatrices, and the rules of the game are simple. The board has some initial entries, for example:

	7		5	2		8		
	8			1	6		4	
2		1						7
6			8			7	9	
	4	9		6		2	3	
	1	8			9			5
1						6		9
	9		6	8			7	
		6		5	7		2	

and the player is to enter digits from 1 to 9 in the blank spaces, satisfying the following:

- Every row must contain one of each digit
- Every column must contain one of each digit
- Every 3 x 3 submatrix must contain one of each digit.

For example, in the above board, the 8 in column one can only be placed in the bottom row (row 9). It can't be placed rows 1-6 of column one, because each of these is contained in a submatrix that already has an 8. Row 7 of column one is occupied. It can't be placed in row 8, because that row already has an 8 in column 5. Placing the 8 in any row in column one other than the last would be a wrong move. Each Sudoku game has a unique solution that can be reached logically without guessing.

Your job is to write a program that reads Sudoku boards and determines whether or not any wrong moves have been made.

Input

Input consists of a nonnegative 32-bit integer `n` on a line by itself, denoting the number of problem instances, followed by `n` Sudoku boards. Each board consists of a blank line, followed by 9 lines, with each line containing 9 symbols. Each symbol is either a digit from 1 to 9, or a period.

Output

For each problem instance, your program prints one of

- `ok so far` if the board has no mistakes, but is not complete
- `there is an error somewhere` if the board has at least one mistake
- `you got it!` if the board has no mistakes, and is complete

Sample Input

```
4

.7.52.8..
.8..16.4.
2.1.....7
6..8..79.
.49.6.23.
.18..9..5
1.....6.9
.9.68..7.
..6.57.2.

.7.52.8..
.8..16.4.
2.1.....7
6..8..79.
.49.6.23.
.18..9..5
1.....6.9
.9.68..7.
8.6.57.2.

.7.52.8..
.8..16.4.
2.1.....7
6..8..79.
.49.6.23.
.18..9..5
1.....6.9
89.68..7.
..6.57.2.

974523816
583716942
261498357
625834791
749165238
318279465
157342689
492681573
836957124
```

Output Corresponding to Sample Input

```
ok so far
ok so far
there is an error somewhere
you got it!
```

Problem 3

Waiting for Soarin'

May 5th of this year (05/05/05) marked the 50th anniversary of the opening of Disneyland in California. In honor of this special anniversary, the Disney theme parks around the world in Anaheim, Orlando, Paris, Tokyo, and the newest park in Hong Kong are joining together for the Happiest Celebration on Earth. This eighteen month celebration continues through October 5th of next year.

This past May at Epcot in Orlando a brand new and amazing attraction called Soarin' opened which simulates hang gliding over California. Guests truly feel like they are "flying" over landmarks in San Francisco, Palm Springs, and Disneyland itself. Each of the Disney parks prides itself in using the latest technology to enhance their guests' experience. One feature that helps guests plan their day is an electronic display board at the front of the park as well as outside all of the popular rides indicating exactly how much standby time a guest can expect to wait before getting on the ride.

You are asked to write a program to calculate the standby waiting time for a given snapshot of the queue for Soarin' during a park day. You will also calculate the next boarding time for those who enter the queue at that given time. Soarin' can seat up to 360 guests spread over four 90-seat theatres. The attraction itself is five minutes long plus five minutes for the cast members to de-board guests from the hang glider theatre guests and allow new ones to board. The 360 guests at the front of the queue before the ride starts are allowed to board.

All rides start on the hour or at the 10, 20, 30, 40, or 50 minute mark past the hour. If the park opens at 9:00 AM, the very first ride on Soarin' will be at 9:10 AM. It will then run for a total of 72 times throughout the day over the next twelve hours, and can seat at most 360 guests every 10 minutes. This means a maximum of 25,920 guests can be serviced in a park day. The final ride of the day is always at park closing at 9:00 PM, and Epcot will not allow a guest to enter a standby line if their wait time places them at a time later than park closing.

During peak periods when the park is crowded, the standby wait time can extend for hours. On the Fourth of July weekend this past summer, the standby line for Soarin' stretched well outside the front of the Land pavilion where the ride is housed by early in the morning.

Input

The input set starts with a positive 32-bit integer `n` representing the number of snapshots. This is followed by `n` snapshots each of which is made up of one line with two tokens separated by a single space. The first token represents the snapshot time in military time in the format `hhmm`, and the second token is a nonnegative integer representing the total number of guests currently waiting in the standby line. All times will be a valid park operating time after 9:00 AM in the morning and before 9:00 PM in the evening.

You may assume that if the snapshot time matches a start time, all guests have already boarded for that time and started Soarin'. Also, the total number of guests will never exceed the maximum seating of guests on the ride from the time of the snapshot through closing.

Output
For each snapshot, your program should print out one line containing the snapshot number, the time that it was made, and the number of guests currently in line using the format shown below. Use a comma appropriately after the thousands position of the number of guests. All times should print in the format `hh::mm` along with `AM` or `PM`.

If the ride is full until park closing, print a single line containing the string `No Further Boarding of New Passengers Today`. Otherwise, print two lines. The first displays the current standby waiting time in hours and minutes, and the second displays the current boarding time.

Use the singular for `guests`, `hours`, or `minutes` where appropriate. Include a blank line after each snapshot.

Sample Input
```
5
1124 2160
1309 2009
2059 1
0930 24840
0930 25
```

Output Corresponding to Sample Input
```
Snapshot #1 at 11:24 AM with 2,160 guests in line
Standby Time Approximately 1 hour, 6 minutes
Your Boarding Time is Now 12:30 PM

Snapshot #2 at 01:09 PM with 2,009 guests in line
Standby Time Approximately 0 hours, 51 minutes
Your Boarding Time is Now 02:00 PM

Snapshot #3 at 08:59 PM with 1 guest in line
Standby Time Approximately 0 hours, 1 minute
Your Boarding Time is Now 09:00 PM

Snapshot #4 at 09:30 AM with 24,840 guests in line
No Further Boarding of New Passengers Today

Snapshot #5 at 09:30 AM with 25 guests in line
Standby Time Approximately 0 hours, 10 minutes
Your Boarding Time is Now 09:40 AM
```

Problem 4

The New CCSC Election

Due to massive confusion on the last election for CCSC officers, the supervisors appointed a committee to recommend changes in the CCSC election process. The committee's first choice was to change to an "instant runoff" system in which each voter ranks all candidates in a race, specifying 1st, 2nd, 3rd, etc. choices for candidates. This has the advantage of saving the consortium money, since only one election is required – there is no need for both a primary and a general election. However, focus groups of the consortium voters disliked the idea of having to rank all candidates.

The voter focus groups preferred listing head-to-head contests between every possible pairing of candidates for an office. In this model, for four candidates A, B, C and D, the ballot lists the following pairings:

- A vs. B, A vs. C, A vs. D
- B vs. C, B vs. D
- C vs. D

The voter marks his/her preference for each of these "forced choice" scenarios.

This model rapidly becomes unwieldy as the race grows beyond four candidates. A five-candidate race would have ten head-to-head pairings. Therefore, if more than four candidates qualify for a race, a primary election will be held in which the top four candidates by plurality vote will advance to the general election. The resulting head-to-head general election will not exceed six pairings.

In the order they are to be applied, the rules for winning the general election are:

1. If a candidate beats every opponent in head-to-head matchups, the candidate wins.
2. Add up the total votes for a candidate in all the head-to-head races. Drop the candidate with the lowest number of votes. If there is a tie, remove all candidates with that total. If there are any candidates remaining, remove the contests not involving them and start over with step 1.
3. Declare a tie.

Input

The first line of input is a positive 32-bit integer `n` representing the number of races. This is followed by `n` series of races each consisting of two lines. Each race series starts with the name of the office, an alphanumeric string on one line. The next line will be the votes for each candidate in that race. There are three possible formats: 1, 3, or 6 pairs of numbers. Each number will be a nonnegative 32-bit integer separated by a blank. Each line will be at most 80 characters long.

n_{12} n_{21}

n_{12} n_{21} n_{13} n_{31} n_{23} n_{32}

n_{12} n_{21} n_{13} n_{31} n_{14} n_{41} n_{23} n_{32} n_{24} n_{42} n_{34} n_{43}

where n_{ij} is the number of votes for candidate *i* against candidate *j* in that head-to-head contest.

Output
Print two lines for each contest. Print the name of the office on the first line. Print the candidate number of the winner of the general election on the second line. If there is a tie, print the candidate numbers involved in the tie, sorted by candidate number (lowest first) and separated by one blank.

Sample Input
```
5
secretary
100 50 100 50 50 100 100 40 100 45 100 50
vice president
445 445 445 445 445 445 445 445 445 445 445 445
treasurer
12287 13876
president
127 99 54 67 184 157
registrar
1 0 1 0 99 100
```

Output Corresponding to Sample Input
```
secretary
1
vice president
1 2 3 4
treasurer
2
president
2
registrar
1
```

Problem 5

David's Electrical Outlets

David has just moved into a new apartment in Hickory's Claremont Historic District. The apartments in this old neighborhood date back to the day before people had electricity in their homes. Because of this, David's apartment has only one single wall outlet, so David can only power one of his electrical appliances at a time. David likes to watch TV as he works on his computer, and to listen to his HiFi system (on high volume) while he vacuums, so using just the single outlet is not an option. Actually, he wants to have all his appliances connected to a powered outlet, all the time. The answer, of course, is power strips, and David has some old ones that he used in his old apartment. However, that apartment had many more wall outlets, so he is not sure whether his power strips will provide him with enough outlets now.

Your task is to help David compute how many appliances he can provide with electricity, given a set of power strips. Note that without any power strips, David can power one single appliance through the wall outlet. Also, remember that a power strip has to be powered itself to be of any use.

Input
Input will start with a single integer n where $1 \leq n \leq 20$, indicating the number of test cases to follow. This will be followed n lines, each describing a test case.

Each test case starts with an integer k where $1 \leq k \leq 10$, indicating the number of power strips in the test case. This will be followed, on the same line, by k integers separated by single spaces, $O_1\ O_2\ \ldots\ O_k$, where $2 \leq O_i \leq 10$, indicating the number of outlets in each power strip.

Output
Output one line per test case, with the maximum number of appliances that can be powered.

Sample Input
```
3
3 2 3 4
10 4 4 4 4 4 4 4 4 4 4
4 10 10 10 10
```

Output Corresponding to Sample Input
```
7
31
37
```

Problem 6

Rotating my Graphic

A plain portable grey map (PGM) file is a very simple form of a digital image file. One of the nice things about plain PGM files is that they can be edited with a standard text editor, even though it can be tedious. Unfortunately, they are an older format so most Internet browsers and many image viewers today don't understand them, they either complain about plug-ins or they display PGM images as text files. We have an image viewer that does handle them properly but what we want to do is modify them. Your job is to write a program that reads in a plain PGM image from a file, rotates the image 90^{o} clockwise, and writes out the new image.

Input and Output

The format of plain PGM files is simple. The first word in the file is a line containing the "magic number" which is the word `P2`. This may be followed by lines containing comments. Comments in PGM files are lines that start with the `#` (also known as a pound sign, splat, or octothorpe) and continue to the end of the line.

The next two words in the file are 32-bit integers representing the number of columns and the number of rows, in that order. The fourth word is the MAXVAL, a value in the range between 0 and 255 inclusively. This defines the maximum possible grey value for a pixel, 0 being black, and MAXVAL is white with everything else a shade of grey. All comments must occur before the line containing the number of columns and the number of rows.

Then there are a set of "column times row" pixel values which are ASCII numbers in the range 0 to MAXVAL.

For this situation we will have the following requirements.

- There will be no blank lines in the file and every line will start with a character at the left margin.
- The magic number will be alone on the first line of the file.
- Any comments will follow that.
- The column size and row size values will be on the first line following any comments. The values will be separated by one whitespace character and there will be nothing else on the line.
- The MAXVAL will be alone on the next line of the file.
- The pixel values will follow. Each value will be separated from the next by a single whitespace character and each row will contain one value for each column of the image.
- There will be no extra lines in the file.

Here is a sample of what a plain PGM might look like. Note that the number of columns does not have to be equal to the number of rows.

Sample Input

```
P2
#any comments go here and start with a pound sign.
#there can be multiple comment lines here but NOT in the rest of
#the file.
#the next line will have the number of columns then the number of
#rows
17 7
255
37 44 37 37 44 37 44 44 44 37 44 44 48 48 48 44 37
34 33 34 37 33 34 37 48 44 44 44 48 44 44 44 44 53
44 44 37 44 53 53 49 44 41 37 41 41 33 33 29 26 26
29 29 33 37 37 44 48 53 53 57 53 57 61 64 69 69 81
81 77 81 81 73 77 72 48 37 33 33 33 37 37 48 44 33
22 22 25 22 22 14 14 14 14 22 22 22 22 22 22 22 22
22 22 26 22 22 22 22 18 22 18 22 18 18 18 22 26 22
```

Output Corresponding to Sample Input

```
P2
#any comments go here and start with a pound sign.
#there can be multiple comment lines here but NOT in the rest of
#the file.
#the next line will have the number of columns then the number of
#rows
7 17
255
22 22 81 29 44 34 37
22 22 77 29 44 33 44
26 25 81 33 37 34 37
22 22 81 37 44 37 37
22 22 73 37 53 33 44
22 14 77 44 53 34 37
22 14 72 48 49 37 44
18 14 48 53 44 48 44
22 14 37 53 41 44 44
18 22 33 57 37 44 37
22 22 33 53 41 44 44
18 22 33 57 41 48 44
18 22 37 61 33 44 48
18 22 37 64 33 44 48
22 22 48 69 29 44 48
26 22 44 69 26 44 44
22 22 33 81 26 53 37
```