Computer Science Competition
Invitational B 2025
Programming Problem Set

I. General Notes

1. Do the problems in any order you like. They do not have to be done in order from 1 to 12.

2. All problems have a value of 60 points.

3. There is no extraneous input. All input is exactly as specified in the problem. Unless specified by the problem, integer inputs will not have leading zeros. Unless otherwise specified, your program should read to the end of file.

4. Your program should not print extraneous output. Follow the form exactly as given in the problem.

5. A penalty of 5 points will be assessed each time that an incorrect solution is submitted. This penalty will only be assessed if a solution is ultimately judged as correct.

II. Names of Problems

Number	Name
Problem 1	Akash
Problem 2	Amy
Problem 3	Dante
Problem 4	Girish
Problem 5	Howard
Problem 6	Hulin
Problem 7	Julia
Problem 8	Nicolas
Problem 9	Pranav
Problem 10	Ruth
Problem 11	Sarac
Problem 12	Varsha

1. Akash

Program Name: Akash.java **Input File: None**

Your friend Akash has taken a liking to prime numbers. He recently learned about the Sieve of Eratosthenes, a method for determining prime numbers up to a given limit. The method works as follows: start with 2, and remove all items that are multiples of 2 up to the limit. Then move to the next number in the list (3 in this case) and repeat the process. The remaining numbers must be prime! Akash is curious as to what this would look like for numbers up to 20, so you write a program to output that for him.

Input: None

Output: Output the list after each step, only output if a value is removed. Numbers are separated by a single space.

Sample input: None

Sample output:

```
2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
2 3 5 7 9 11 13 15 17 19
2 3 5 7 11 13 17 19
```

2. Amy

Program Name: Amy.java **Input File: amy.dat**

Amy has recently become interested in password cracking, specifically the amount of time it takes for a machine to guess a given password when brute-force guessing. So far, she has been doing the calculations by hand for fun, but Amy knows you're pretty good at programming and has asked for your help. Amy would like you to write a program to determine the amount of time it would take a given machine to crack a password.

Input: The first input line will contain a number N (1 <= N <= 100) denoting the number of lines to follow.
Each following line of input will contain a string S (1 <= S.length <= 100) being the password, and a value G (1 <= G <= 10^20) being the number of guesses the machine can make every second. Passwords will only ever contain lowercase English characters and numbers. The computer will guess completely random, legal, passwords of the length of S.

Output: Output the amount of time in seconds it would take the given machine to crack the provided password.

Sample input:
```
2
cats 100
cats1 100
```

Sample output:
```
16796
604661
```

3. Dante

Program Name: Dante.java **Input File: dante.dat**

Dante is a new kid at your school, and he LOVES prime numbers. You are playing a game with him lately, where he gives you two numbers, and you determine how many prime numbers are in between these numbers (not inclusive).

Input: The input will begin with an integer, `n` (`0 < n <= 1000`), denoting the number of test cases to follow. Each test case will consist of two integers, separated by a space, `a` and `b` (`0 < a < b <= 100,000`), denoting the beginning and end of the range of numbers you are to search through to find out how many prime numbers lie within this range.

Output: For each test case, output the number of prime numbers between `a` and `b`.

Sample input:

```
3
4 15
10 14
8 11
```

Sample output:

```
4
2
0
```

4. Girish

Program Name: Girish.java **Input File: girish.dat**

You have another new professor, Dr. Girish. He loves lists. He loves lists so much, he wants you to find special statistics about his lists. He will give you an unknown number of lists of integers, and your job is to find the median average of the lists, the average of the sums of each of the lists, and the integer that occurs most frequently in the lists, without occurring in all of the lists.

Input: The input will consist of an unknown number of lists of integers. Each line will contain a list of integers, all separated by spaces. The test cases will be concluded with a line containing only 10 − characters.

Output: Output three lines, each containing one of the statistics requested. The first line should be in the format "`Median Mean: `", followed by the median of the average values of all the lists, formatted to two decimal places. The second line should be in the format "`Mean Sum: `", followed by the average of all the sums of each of the lists, formatted to two decimal places. The third line should be in the format "`Mode Kinda: `", followed by the integer occurring most in all of the lists, but not occurring in every list.

Sample input:

```
1 2 3 4 5
5 5 5 4 3
3 4 5 2 2
5 4 3 2 1
1 2 3 5 1
----------
```

Sample output:

```
Median Mean: 3.00
Mean Sum: 16.00
Mode Kinda: 2
```

5. Howard

Program Name: Howard.java **Input File: howard.dat**

You are the newest member of the basketball team at your school. Professional player Howard Dwight is coming to help coach and take you guys to the next level. One of the big things he wants to do is to start incorporating "Advanced Statistics" into your schemes as a team. The first thing he's going to do is sort the players based on their true shooting percentage, and if two players have the same true shooting, then sort by their assist to turnover ratio. True shooting percentage can be calculated using the following formula: (variables defined in input section below)

$$TS = \frac{PTS}{(2 * (FGA + 0.44 * FTA))}$$

Assist to Turnover ratio can be calculated simply by dividing the number of assists by the number of turnovers.
You have been assigned the job of sorting the players, and choosing the best player at each position to create the ideal starting lineup. Remember to sort by players with the largest true shooting first, and then if any are equal sort by players with the higher assist to turnover ratio (`APG / TPG`).

Input: The input will begin with an integer, `n` (`0 < n <= 1000`), denoting the number of players to follow. Each player will appear on a single line with the following fields in the order listed below, all separated by spaces:

- `Number` – The number on a player's jersey, an integer value.
- `Name` – The player's last name, a string.
- `Position` – A string denoting which position this player plays.
  - Will be `PG`, `SG`, `SF`, `PF`, or `C`
- `PTS` – Points per game, a floating point value (maximum 3 decimal places).
- `APG` – Assists per game, a floating point value (maximum 3 decimal places).
- `FGA` – Field goals attempted per game, a floating point value (maximum 3 decimal places).
- `FTA` – Field goals attempted per game, a floating point value (maximum 3 decimal places).
- `TPG` – Turnovers per game, a floating point value (maximum 3 decimal places).

Output: Output the best player to be in the starting lineup for each position, in the order `PG`, `SG`, `SF`, `PF`, `C`. Each position should be listed on its own line, followed by a colon and a space, followed by the number of the player, a period, a space, and the last name of the player. There is always guaranteed to be at least one player for each position, and no players will have equal true shooting AND assist to turnover ratio.

Sample input:

```
8
12 Armstrong PG 18.5 7.8 9.9 3.4 2.7
15 Devens PG 17.6 8.0 10.0 3.3 1.4
13 Harden SG 35.6 7.6 24.3 10.5 4.5
34 O'Neal C 28.5 1.5 17.3 12.6 1.8
9 Caruso PG 8.5 3.4 5.4 0.8 0.5
2 Doncic SG 34.7 9.5 27.5 8.4 3.9
23 James SF 28.5 10.1 19.4 7.3 2.4
35 Durant PF 31.4 3.4 19.6 9.3 2.3
```

Sample output:

```
PG: 12. Armstrong
SG: 13. Harden
SF: 23. James
PF: 35. Durant
C: 34. O'Neal
```

6. Hulin

Program Name: Hulin.java **Input File: hulin.dat**

Your good friend Hulin has recently started a new job stocking groceries at the local mega-corporation LostLo. Hulin complains to you (and her boss) that he is micromanaging the shifts and as a result, there are too many people working at the same time. Her boss, however, does not believe her! Since there are so many employees, she has decided to give you everyone's schedules provided you will write a program to determine the largest number of people working at the same time.

Input: The first input line will contain a single value N (1 <= N <= 100) denoting the number of schedules to follow. Each following input line will contain a list of 3 space-separated pairs of time values (B, E) (1 <= B < E <= 86400) denoting the start and stop time of someone's shift (inclusive) during a day.

Output: The largest number of people working at a given time.

Sample input:

```
5
1 100 500 1900 3000 4999
200 499 1901 3219 40000 86400
300 400 1542 1782 24509 53332
2399 13259 21511 24992 59929 59999
6150 10019 12512 59900 70359 82122
```

Sample output:

```
3
```

7. Julia

Program Name: Julia.java **Input File: julia.dat**

Julia, your neighbor, has recently become obsessed with prime numbers. She wants to know how many prime numbers exist within the first 100 numbers. She knows you're a good programmer, so she asks you for help with this problem. You, being the great friend you are, know Julia will ask follow-up questions past the first 100 so decide to write a program to find the number of prime values before any given value.

Input: The first input line will contain a value N (1 <= N <= 100) that denotes the number of input lines that follow. Each additional input line will contain a single value V (1 <= V <= 10^7) being the number Julia has asked about.

Output: Output the number of prime values less than or equal to the queried value V.

Sample input:

```
3
100
1000
10000
```

Sample output:

```
25
168
1229
```

8. Nicolas

Program Name: Nicolas.java **Input File: nicolas.dat**

It's February, which is when Texas gets the most snow, which is no different for Nicolas on this cold morning. Thankfully, Nicolas lives within walking distance of his school; however, unfortunately for him, the roads and sidewalks are rather icy. This makes it so that, when Nicolas starts walking in a certain direction, he is forced to continue moving in that direction until he bumps into something stationary. Thankfully for him, Nicolas is rather tough and is not concerned with getting hurt from running into said objects; however, he is concerned with minimizing the number of times he bonks into something, even if it means walking (really sliding on the ice) a little further. Help Nicolas in planning the path that minimizes the number of times he needs to run into a stationary object on his way to school.

Input: The first line of input will consist of a single integer n, $1 \leq n \leq 10^3$, denoting the number of different places that Nicolas needs to determine a route to. The next n test cases will each consist of a single line denoting two space separated integers, r and c, $2 \leq r, c \leq 2 \cdot 10^2$, denoting the number of rows and columns, respectively, in the map that Nicolas is basing his routes off of. The next r lines will consist of c characters, each character will be among one of the following:

- '`.`' : Denotes a walkable space that Nicolas can slide on.
- '`#`' : Denotes a stationary object that Nicolas can bump into.
- '`S`' : Denotes Nicolas' starting position. There will be a single spot labeled with an '`S`'.
- '`E`' : Denotes the location that Nicolas is attempting to reach. There will be a single spot labeled with an '`E`'.

Output: For each of Nicolas' n routes, print out a string denoting the series of directions that Nicolas should move in to minimize the number of stationary objects that he has to run into. This string will consist of the characters '`N`', '`S`', '`E`', and '`W`' denoting North, South, East, and West. If there are multiple such paths that minimize this, then print out the one that is lexicographically smallest. It is also guaranteed that there will be some path from where Nicolas is starting and where he intends to end at. You may also assume that there is an understood border of stationary objects on the spaces outside of the map provided to you.

Sample input:

```
3
5 6
S###.E
.#.#.#
.#.#.#
......
#..#..
5 10
S.........
..########.
#..#######.
##..######.
###..E....
3 5
..S..
.###.
..E..
```

Sample output:

```
SESWNE
ESW
ESW
```

9. Pranav

Program Name: Pranav.java **Input File: pranav.dat**

Your math professor is not having a good time lately. He has given the class an assignment to determine the best strategy to win the "Pranav" game, a game he made up that is conveniently named after himself. You will be given a line of "pots" of gold, each pot worth a given number of points. Each round of the game, you select the "pot" on either end of the line, take it and add it to your total, and then your opponent does the same. Determine the best possible score you can get, and if that number will win you the game (the goal is to have a larger total than your opponent). You can assume that your opponent will play optimally.

Example: 5, 3, 7, 10
Answer: Winner 15

You choose 10, opponent chooses 7, you choose 5, opponent chooses 3: (10 + 5) = 15 > (7 + 3) = 10

Input: The input will begin with an integer, `n` (`0 < n <= 1000`), denoting the number of test cases to follow. Each test case will consist of an unknown number of space-separated integers denoting the "pots" of gold the game is to be played with. There will always be an even number of "pots".

Output: For each best possible score, if you are the winner, output the string "`Winner`", if you are the loser, output the string "`Loser`", otherwise output "`Tie`". After that string, output a space, followed by the best possible score you could get.

Sample input:

```
3
10 7 3 5
10 10 10 10 10 10
8 15 3 7
```

Sample output:

```
Winner 15
Tie 30
Winner 22
```

10. Ruth

Program Name: Ruth.java **Input File: ruth.dat**

You and your friend Ruth are at the beach, building sandcastles. You've both been randomly piling sand, but you notice something interesting. Your sand castles connect to form a wall, with differing heights between each castle. Ruth says she wonders how much sand can fit in the empty spaces, and starts measuring the height of the walls. Once home, you have the heights of the walls and must write a program to calculate the amount of sand that can fit the space.

Input: The first input line will contain a single value N (1 <= N <= 100) denoting the number of input lines that follow. Each additional input line will contain some number M (1 <= M <= 10^7) of space-separated integers
H (1 <= H <= 20) denoting the height of a sandcastle wall at that position. You may assume that each sandcastle wall (or lack thereof) is exactly 1 unit wide.

Output: Output for each test case, a single value V denoting the units of sand that could be filled in. Note that you are not trying to make a wall of a single height, but simply fill in gaps between walls.

Sample input:

```
2
3 0 2 0 4
2 1 5 3 1 0 4
```

Sample output:

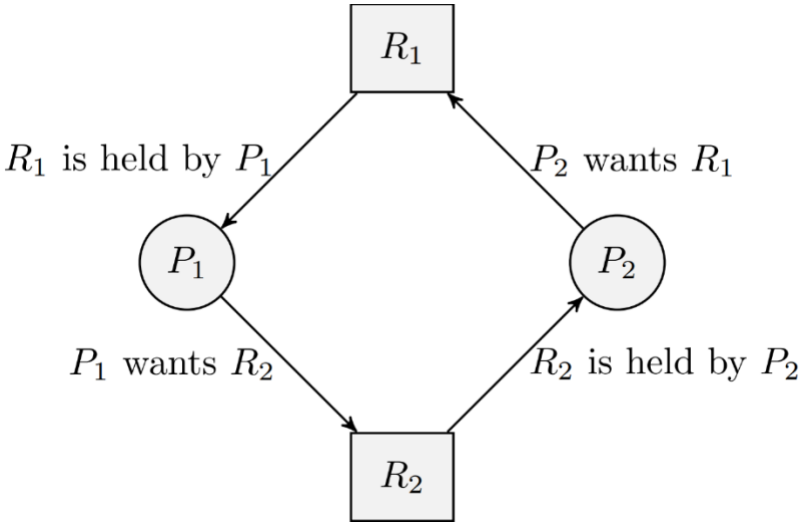
```
7
9
```

11. Sarac

Program Name: Sarac.java **Input File: sarac.dat**

Sarac is learning about Mutual Exclusion and Deadlocks which are common problems in Multi-Threaded programs and an issue that Operating System Engineers are constantly forced to address. For a resource to be considered "Mutually Exclusive", then at most one process or program is allowed to access that resource at any single point in time. In other words, no two processes can share the resource at the same time, but multiple processes may hold the resource independently over a period of time.

Since a single process may be interested in holding multiple such shared resources that need to be accessed mutually exclusively at a single instance of time, it is possible for a "Deadlock" to occur. Consider the example graph below – since both P_1 and P_2 are waiting on a resource held by the other, and neither is able to release their resource until they have obtained all resources they are interested in, a Deadlock has formed since neither process can continue.



One such way of modeling these systems is through a structure known as a Resource Allocation Graph (RAG). This is a bipartite graph where a directed edge from a resource R, to a process P, exists if and only if resource R is held by process, P. Similarly, there exists a directed edge from a process P, to a resource R, if and only if process P is interested in holding resource R. No other edges exist in the graph outside of these two types. Using this model, Sarac is able to detect the occurrence (or absence) of Deadlocks in a given system. Given a RAG, help Sarac detect the occurrence or absence of a Deadlock in the RAG.

Input: The first line of input will consist of a single integer n, $1 \leq n \leq 20$, denoting the number of test cases to follow. Each test case will begin with a line denoting two space-separated integers V, $4 \leq V \leq 10^5$, denoting the number of vertices and E, $2 \leq E \leq 10^5$, denoting the number of edges in the RAG. The next line will consist of V space-separated strings denoting the labels of V vertices in the graph, each of which will either be in the form of "`Pi`" or "`Ri`", where `i` is some integer $1 \leq i \leq V$. The next line will consist of E space-separated edges, each of which will either be in the form of "`Pi->Ri`" indicating that `Pi` is interested in holding `Ri`, or "`Ri->Pi`" indicating that `Ri` is held by `Pi`. It is guaranteed that all labels among the edges exist among the V vertices provided, and all edges adhere to the rules described above.

Output: For each of Sarac's n requests, either print the string "`Deadlock free; all is well`" if the RAG does not contain a Deadlock or the string "`Deadlock exists; not good...`" if the RAG contains a Deadlock.

~ ***Sample input and output on next page*** …~

~ ***Sarac continued*** …~

Sample input:

```
4
4 4
P1 P2 R1 R2
R1->P1 P1->R2 R2->P2 P2->R1
4 3
P1 P2 R1 R2
R1->P1 P1->R2 P2->R1
5 4
P1 P2 P3 R1 R2
R1->P1 R2->P2 P3->R1 P3->R2
7 9
R1 P1 R2 P2 R3 P3 R4
R1->P1 R2->P2 R3->P3 P1->R2 P1->R3 P2->R1 P2->R3 P3->R1 P3->R4
```

Sample output:

```
Deadlock exists; not good...
Deadlock free; all is well
Deadlock free; all is well
Deadlock exists; not good...
```

12. Varsha

Program Name: Varsha.java **Input File: varsha.dat**

You are playing Minecraft, and a new kind of ore, Varsha, has been added in the latest update. You, being the expert Minecrafter you are, have given yourself a mission. You are setting out to find the largest "vein" of Varsha in a given chunk. A "vein" is a contiguous blob of a certain ore, where every block of that ore in that blob is directly contiguous to another block of the same ore. Blocks can only be contiguous in 6 directions, up, down, left, right, forward and backward.

Input: The input will begin with an integer, `n` (`0 < n <= 1000`), denoting the number of test cases to follow. Each test case will begin with 3 integers, `d`, `r`, and `c`, denoting the number of levels, rows, and columns in the given chunk to be scanned. The following `r` lines will each contain `c` characters denoting all the characters in the first level of the chunk. All `d` levels will be given in this manner, with `r` lines of `c` characters denoting each level.
The chunks will be made up of the following characters:

- `#` – Denotes a dirt block.
- `*` – Denotes a diamond ore block.
- `V` – Denotes a Varsha ore block.
- `@` – Denotes a stone block.
- `o` – Denotes a redstone ore block.
- `%` - Denotes a gravel block.

Output: For each chunk, output the size of the largest "vein" of Varsha you find.

Sample input:

```
2
2 5 6
###ooV
#oooVV
**oV#o
***Voo
%%VV##
####VV
###V%%
#oooo%
V*oo%%
VVoo%%
3 3 3
VoV
###
#V#
**V
*VV
###
###
VV#
###
```

Sample output:

```
5
6
```