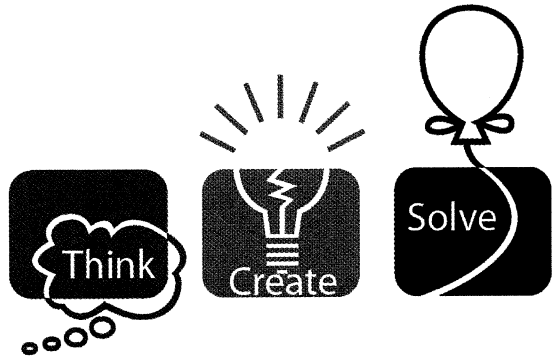
University of Central Florida



2014 (Fall) Local Programming Contest

Problems

Problem#	Difficulty Level	Filename	Problem Name
1	Easy	vowel	Vowel Count
2	Easy	soccer	Soccer Standings
3	Easy	frog	Jumping Frog
4	Medium	sushi	Fujiyama Thursday
5	Medium	email	Chain Email
6	Medium	microwave	Faster Microwaving
7	Medium	dirty	Dirty Plates
8	Medium	chocolate	Chocolate Fix
9	Hard	shop	Shopping Spree
10	Hard	fact	Factorial Products
11	Hard	lucky	Super Lucky Palindromes
12	Hard	ucf	Under Construction Forever

Call your program file: filename.c, filename.cpp, or filename.java
Call your input file: filename.in

For example, if you are solving Fujiyama Thursday:

Call your program file: sushi.c, sushi.cpp, or sushi.java
Call your input file: sushi.in

UCF Local Programming Contest — August 30, 2014

Vowel Count

filename: `vowel`
(*Difficulty Level:* `Easy`)

Dr. Orooji noticed that his name has more vowels than consonants. Since he likes meeting people like himself, he has asked you to write a program to help him identify such names.

The Problem:

Given a name, determine whether or not it has more vowels than consonants. Assume the vowels are "aeiou".

The Input:

The first input line contains a positive integer, *n*, indicating the number of names to check. The names are on the following *n* input lines, one name per line. Each name starts in column 1 and consists of 1-20 lowercase letters (assume the name will not contain any other characters).

The Output:

Output each name on a separate line as it appears in the input followed by a 1 (one) or 0 (zero) on the next line indicating whether or not it has more vowels than consonants. Follow the format illustrated in Sample Output.

Sample Input:

```
4
ali
arup
travis
orooji
```

Sample Output:

```
ali
1
arup
0
travis
0
orooji
1
```

UCF Local Programming Contest — August 30, 2014

Soccer Standings

filename: `soccer`
(*Difficulty Level:* `Easy`)

In a soccer match, a team either earns a win, tie or loss. A win is worth 3 points, a tie is worth 1 point, and a loss is worth 0 points. Unfortunately, due to poor record-keeping, some leagues have only saved the number of total matches played and the number of points each team has earned. One of these leagues has asked you to write a program to recreate the possible combinations of wins, ties and losses for certain teams in the league.

The Problem:

Given the number of games played by a soccer team in a season and the number of points earned by the team, list each possible combination of wins, ties and losses that the team could have gotten to achieve the given total points.

The Input:

The first input line contains a positive integer, *n*, indicating the number of teams for which you are to reconstruct possible records. The teams' information are on the following *n* input lines, one team per line. Each of these lines contains two space separated integers: g $(0 < g \leq 100)$, and p $(0 \leq p \leq 300)$, representing the number of games played and the total points earned by the team, respectively. It is guaranteed that there is at least one possible combination of wins, ties and losses that is consistent with the given information for each team.

The Output:

For each team, first output header info with the following format:

```
Team #k
Games: g
Points: p
Possible records:
```

where k is the team number (starting with 1), g is the number of games, and p is the total points earned. Following the above header info, output the possible records, each on a separate line with the format:

```
w-t-l
```

where w is the number of wins, t is the number of ties and l is the number of losses. Print these by descending order of wins.

Leave a blank line after the output for each team. Follow the format illustrated in Sample Output.

Sample Input:

```
3
6 10
1 3
4 4
```

Sample Output:

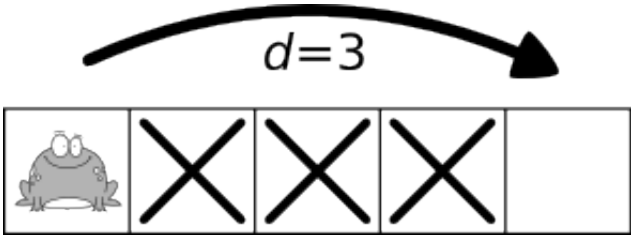
```
Team #1
Games: 6
Points: 10
Possible records:
3-1-2
2-4-0

Team #2
Games: 1
Points: 3
Possible records:
1-0-0

Team #3
Games: 4
Points: 4
Possible records:
1-1-2
0-4-0
```

UCF Local Programming Contest — August 30, 2014

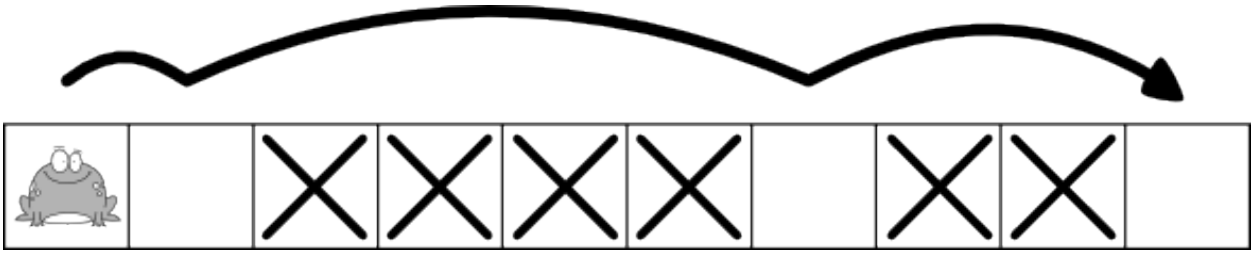
Jumping Frog

filename: `frog`
(*Difficulty Level:* `Easy`)

Freddy the frog is trying to go get a bite to eat. The problem is Freddy lives far away from all the restaurants he likes. Freddy is a capable frog, able to jump over a large number of cells. Unfortunately, on a given day, Freddy may be too hungry to jump very far. On a given day, he can only jump over at most *d* cells (note that he can also jump over fewer cells).



Another complication for Freddy is the path he jumps across is always under construction. Some of the cells are blocked off! Freddy doesn't want to land in a cell under construction but is allowed to jump over them.



Freddy starts off in the first cell and must travel to the last cell where his destination restaurant resides. He would like to know if he can reach the last cell and how quickly he can reach it. Freddy always jumps towards his destination.

The Problem:

Given a description of cells, determine the minimum number of jumps Freddy can make to reach the restaurant.

The Input:

The first input line contains a positive integer, *n*, indicating the number of days to check. Each day is represented by two lines. The first line of each day contains two integers, *c* ($2 \le c \le 50$) and *d* ($0 \le d \le 50$), representing (respectively) the number of cells in the path from Freddy's home to the restaurant (including his home and the restaurant) and the maximum number of cells Freddy can jump over in a single jump on that day. The second line is a string consisting of *c* characters containing only '`.`' and '`X`' characters where '`.`' represents that the cell is okay for

Freddy to occupy and 'X' represents that the cell is blocked by construction. The first and last characters of the string represent Freddy's home and the restaurant, respectively; these two locations will never be blocked.

The Output:

For each day, first output the heading "`Day #`*d*", where *d* is the day number, starting with 1. Then, print the input values exactly as they appear in the input. Following the header info, output the minimum number of jumps it takes Freddy to reach the restaurant. If it is not possible to reach the restaurant on a given day, print the number 0 instead.

Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

Sample Input:

```
4
8 3
.XX.X.X.
3 50
...
8 1
...XX...
10 4
..XXXX.XX.
```

Sample Output:

```
Day #1
8 3
.XX.X.X.
2

Day #2
3 50
...
1

Day #3
8 1
...XX...
0

Day #4
10 4
..XXXX.XX.
3
```

UCF Local Programming Contest — August 30, 2014

Fujiyama Thursday

filename: `sushi`
(*Difficulty Level:* `Medium`)

During the past year, the UCF programming team members started a weekly dinner at Fujiyama Sushi. Usually, the programming team members wait for everyone to arrive to start eating, but as a wise programming team coach from the north of campus always says: "regionals is coming." Heeding this ominous warning, the programming team members need to spend more time to prepare. To spend more time practicing, the programming team members will start eating as they arrive. However, they will still wait until everyone is done eating to leave Fujiyama.

Each car the team members are taking holds exactly four people. As each car may take a different amount of time to arrive at Fujiyama and each team member may have different eating speeds, it is important to assign team members to cars in a careful manner. Any team member can be assigned to any car as all team members can drive any of the cars.

The Problem:

Given the time it takes for cars to arrive and eating speeds of the team members, determine the minimum amount of time needed for all the team members to finish eating if the team members are assigned to cars optimally.

The Input:

The first input line contains a positive integer, n, indicating the number of trips to Fujiyama Sushi. Each trip is represented by three lines. The first line for each trip contains a single integer, c $(1 \le c \le 50)$, representing the number of cars going to Fujiyama. The second line contains c integers, d_i $(1 \le d_i \le 45)$, representing the time (in minutes) it takes for the i^{th} car to arrive to Fujiyama. The third line contains $4*c$ integers, t_j $(1 \le t_j \le 75)$, representing the time (in minutes) it takes for the j^{th} team member to finish eating.

The Output:

For each trip, first output the heading "`Trip #d:`", where `d` is the trip number, starting with 1. Then, output a single integer representing the minimum amount of time (in minutes) for all the team members to finish eating. Follow the format illustrated in Sample Output.

Sample Input:

```
3
1
40
1 2 3 4
2
10 20
5 6 3 4 8 9 1 2
3
15 20 20
10 10 10 10 20 20 20 20 30 30 30 30
```

Sample Output:

```
Trip #1: 44
Trip #2: 24
Trip #3: 45
```

UCF Local Programming Contest — August 30, 2014

Chain Email

filename: `email`
(*Difficulty Level:* `Medium`)

A chain email is an email that people receive and then forward to all of their friends. This sort of email is very common amongst elderly people, who have notably bad memories. Elderly people's memories are so bad that if they ever receive a chain email they will forward it to all of their contacts. This can become very problematic when elderly people continually send the same email to each other. For instance, if two people have each other in their contacts and if either of them receive a chain email they will continually send the email to each other back and forth forever. Email companies are worried that this will result in a massive amount of storage loss on their servers and have asked you to determine if a specific person were to start a chain email, who would receive that email forever.

The Problem:

Given each elderly person's contacts and which elderly person will be starting a chain email, determine who will be indefinitely receiving emails.

The Input:

The first line of the input is a positive integer, *n*, indicating the number of scenarios that your program will have to analyze. Following this will be the description of each scenario. The first line of each scenario will have two single-space-separated integers, p ($1 \leq p \leq 50$), indicating the number of people who use the email service and, s ($1 \leq s \leq p$), indicating the source of the chain email, where each person is labeled from 1 to p. Following this will be a single line with the names of all of the people, from person 1 to person p, who use the email service, each separated by exactly one space. All names will contain alphabetic characters only and be between 1 and 19 characters (inclusive) in length. Following this will be p lines. The i^{th} line will describe the contact list of the i^{th} person. This description will consist of an integer, m ($0 \leq m < p$), indicating the number of contacts this person has, followed by the 1-based index of each of the contacts, each separated by exactly one space. It's guaranteed that no one will contain themselves as a contact.

The Output:

The first line of the output for each scenario should be "`Chain Email #`*d*`:`", where *d* is the scenario number, starting with 1. Following this should be a line containing the names of all of the people who will infinitely receive chain emails, assuming that everyone continually forwards the email to all of their contacts. Each name should be followed by a space. List these contacts in the order that they appear in the input. If no one will infinitely receive chain emails, then print "`Safe chain email!`" instead.

Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

Sample Input:

```
3
3 1
James Sarah John
2 2 3
2 1 3
2 1 2
3 1
James Sarah John
2 2 3
0
0
6 3
Ali Matt Glenn Sumon Arup Chris
2 3 5
0
1 4
1 1
1 2
2 5 4
```

Sample Output:

```
Chain Email #1:
James Sarah John

Chain Email #2:
Safe chain email!

Chain Email #3:
Ali Matt Glenn Sumon Arup
```

UCF Local Programming Contest — August 30, 2014

Faster Microwaving

filename: `microwave`
(*Difficulty Level:* `Medium`)

Chris likes getting his food quickly so he cooks a lot using the microwave. However, he is frustrated by how microwave makers seem not to communicate well with makers of microwavable foods. For example, many microwaves have a "popcorn" button, but most microwave popcorn instructions say "Do not use the 'popcorn' button." To avoid any chance of ruining his food, Chris uses only timed cooking, entering the time to cook each item in minutes and seconds (in MM:SS format) on the numbered buttons of the microwave. Since there is one digit per button, he has to press one digit at a time and move his finger between different digits, which is tedious and annoying because he's hungry. One nice thing is that there is no button for the ":" as the microwave always interprets the last 2 digits as seconds, and inserts the ":" appropriately—however, it does not enforce a restriction that the last two digits are 59 seconds or less; if Chris presses 1, then 9, then 0 for a time of "1:90" it will cook for 1 minute and 90 seconds, which is the same as 2 minutes and 30 seconds.

Chris would like to be able to enter the cooking times more quickly. He notices that it takes 1 "moment" (a unit of time just under half a second) to press each digit's button firmly, and it also takes 1 moment (the same unit of time) to move his finger away from one digit's button to find the button for a different digit. Therefore, to enter a time "4:00" takes 4 moments in total—one to press "4", one to move from "4" to "0", one to press "0", and one to press "0" again immediately, without having to find the button. It also takes 4 moments to enter "4:45" (press 4, then 4 again, then move from 4 to 5, then press 5), but it takes 5 moments to enter "4:30" (4, then move, then 3, then move, then 0).

After some experimentation, Chris devises the following plan to enter faster cooking times that are reasonably close to the recommended times in the cooking instructions for each item:

1. Based on the microwavable item type, consider using a range of proposed cooking times that are each within a small percent above or below the recommended cooking time. For example, using 10% with a recommended cooking time of 2 minutes and 30 seconds (2:30), the proposed cooking times would be the range of times from 2:15 to 2:45 inclusive.
2. Find the sequence of digits (buttons) that takes the lowest total moments to enter out of *any* of the proposed cooking times in the range.
3. If there are multiple sequences of digits that have the same lowest total moments, choose the sequence that yields an actual cooking time that is closest to the recommended cooking time.

Chris has verified that the above plan always results in a unique answer so you may assume so.

The Problem:

Given the recommended cooking time for a microwavable item, and a percent to use for the range of proposed cooking times, output the sequence of digits that Chris should press in order to start the microwave as fast as possible, according to his plan.

The Input:

The first input line contains a positive integer, *n*, indicating the number of microwavable items for which cooking times should be converted. The first line for each food item contains only the recommended cooking time, which consists of exactly 5 characters in the format *MM*:*SS* with 2-digit minutes $MM\ (00 \le MM \le 20)$ and 2-digit seconds $SS\ (SS \in \{00, 15, 30, 45\})$. The recommended cooking time will be at least 00:15 (15 seconds). The second line contains only an integer $p\ (2 \le p \le 10)$, which is the percent of the recommended cooking time that defines the range of lower and higher proposed cooking times.

The Output:

For each recommended cooking time in the input, first output the heading "`Case #d: `", where *d* is the test case number, starting with 1. Then, print the exact digits that should be pressed, in the order they should be pressed. Follow the format illustrated in Sample Output.

Note that the seconds are always integers and the time must be "within" the percent range. For example, for a recommended cooking time of 00:45 and 10%, the range of proposed cooking times is 41 seconds to 49 seconds (45 ± 4, because 40 and 50 are not within 10% of the recommended time).

Sample Input:

```
3
01:30
4
00:30
10
06:00
8
```

Sample Output:

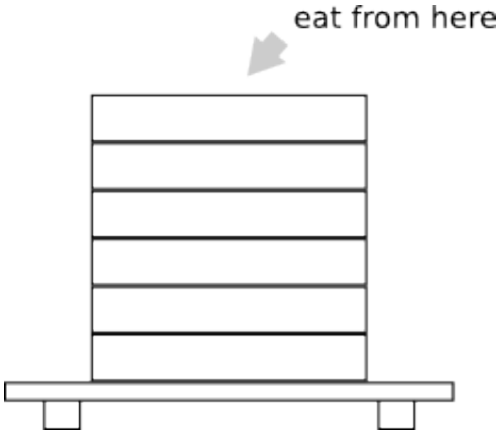
```
Case #1: 88
Case #2: 33
Case #3: 555
```

UCF Local Programming Contest — August 30, 2014

Dirty Plates

filename: `dirty`
(*Difficulty Level:* `Medium`)

Lazy Louie loves to eat but he adamantly hates cleaning. That is why he has chosen to delay cleaning for as long as possible. Being an avid inventor, Louie created an invention to help increase his laziness. That invention is the two sided plate! Two sided plates can be stacked like coins for easy storage. They also have the interesting feature that you may eat from either side of the plate.



Lazy Louie lacks cabinet space. That is why he stores his two sided plates directly on his dining table. When enjoying a meal he simply eats off the topmost plate, although he will only eat off the topmost surface if it is clean. When the plates are stacked, if a dirty side of the plate touches the clean side of another plate then the grime of that plate transfers to the clean plate making that side of the plate dirty. The side that was originally dirty stays dirty.

Since Louie is stacking plates on his table, if a dirty side of a plate touches the table then the grime will transfer to the table making the table dirty. If the table is dirty already and it touches a clean side of a plate then that plate's side becomes dirty. Note that the table remains dirty. Also, both sides remain dirty when two dirty sides, either the table or plates, touch.

The Problem:

Louie would like to know the maximum number of meals that can be eaten before any cleaning is done. Louie is given the number of plates he already has of three kinds: plates that are clean on both sides, plates that are clean on one side while dirty on the other, and plates that are dirty on both sides. Before eating his first meal he can stack these plates in any way he likes on the table. After eating each meal the topmost plate becomes dirty. Between meals Louie can rearrange the plates in any way he likes. Louie is allowed to change the order of the plates on the stack and change which side is facing up but they must remain a stack of plates after being rearranged.

The Input:

The first input line contains a positive integer, n, indicating the number of eating scenarios to analyze. The next n lines contain the description of the plates. Each line contains three integers, c, s, and d, ($0 \leq c \leq 100$; $0 \leq s \leq 100$; $0 \leq d \leq 100$) representing (respectively) the number of plates with both sides clean, the number of plates with one side clean and the number of completely dirty plates.

The Output:

For each scenario, first output the heading "`Scenario #d:`", where d is the scenario number, starting with 1. Then, print the maximum number of times a meal can be eaten before Louie has to clean. Follow the format illustrated in Sample Output.

Sample Input:

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

Sample Output:

```
Scenario #1: 2
Scenario #2: 3
Scenario #3: 2
Scenario #4: 4
```

UCF Local Programming Contest — August 30, 2014

Chocolate Fix

filename: `chocolate`
(*Difficulty Level:* `Medium`)

Your cousin recently bought a game that involves arranging truffles on a 3 x 3 grid. Eager to one up him, you decide to write a program to solve the puzzles automatically, so that he doesn't have a chance in figuring out the puzzles against you!

The game comes with nine truffles, which are used in each puzzle. Each truffle is either vanilla, strawberry or chocolate (VSC) and the shape of each truffle is either square, round or triangular (SRT). There is exactly one truffle of each combination of attributes and all of them must be used. In each puzzle, you must place each of the nine truffles on the 3 x 3 board. For convenience, label the squares 1, 2 and 3 on the first row, from left to right, 4, 5 and 6 on the second row, and 7, 8 and 9 on the third row as illustrated below:

1	2	3
4	5	6
7	8	9

For each puzzle, you are given a list of clues. Each clue represents partial information about a subsection of the board. For example, the clue below means that there is some 3 x 2 window with the square strawberry truffle in its upper right corner:

```
__ SS
__ __
__ __
```

There are only two such possible 3 x 2 windows on a 3 x 3 board. Thus, this clue essentially conveys the fact that the square strawberry truffle must appear in either square 2 or square 3 of the board since clues cannot be rotated:

__	SS	
__	__	
__	__	

	__	SS
	__	__
	__	__

Another example of a clue is as follows:

```
__ _S
__ RC
__ T_
```

This clue also represents a 3 x 2 window of the board. It indicates that in the right column of the window there must be a strawberry truffle (of some shape) on the top row, the round chocolate truffle in the middle row of the column and a triangular truffle (of some flavor) on the bottom

row of the column. Due to the window size, we can ascertain that this must be true of either middle or rightmost column of the game board.

The Problem:

Given a set of clues that uniquely specifies a solution to a chocolate puzzle, determine that solution.

The Input:

The first input line contains a positive integer, *n*, indicating the number of puzzles to solve. The puzzles follow. The first line of each puzzle will contain a single positive integer, *c (1 ≤ c ≤ 10)*, representing the number of clues for that puzzle. The clues will follow. The first line of each clue will contain two space separated integers: *x (1 ≤ x ≤ 3)* and *y (1 ≤ y ≤ 3),* representing the number of rows and columns, respectively, for the clue window. The next *x* lines will contain the clue, with each element in the clue represented by 2 characters. The first character will come from the set `{'_','S','R','T'}`, indicating the shape of that element and the second character will come from the set `{'_','V','S','C'}`, indicating the flavor of that element. Note that the underscore character simply means that that attribute is not fixed. Each of these *x* lines will contain *y* codes; the codes separated by exactly one space.

The Output:

The first line of the output for each puzzle should be "`Puzzle #`*p*`:`", where *p* is the puzzle number, starting with 1. On the next three lines, output the puzzle solution, using the same codes used in the clues. Note that since there is a unique solution to each puzzle, no underscore characters can be in any valid solution.

Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

Sample Input:

```
3
4
3 3
TC __ SS
__ __ __
__ TV __
2 3
__ SC __
RV __ SV
3 3
__ __ __
__ RC __
__ __ __
2 3
__ __ __
TS __ RS
```

```
5
2 3
__ __ __
__ __ RC
2 2
__ RS
SC __
2 2
SV TC
__ __
3 2
TV __
__ __
__ RV
3 2
__ TS
__ __
__ __
3
3 2
_C R_
_C __
S_ _C
1 2
TC _V
3 2
_V __
S_ S_
T_ _V
```

Sample Output:

```
Puzzle #1:
TC SC SS
RV RC SV
TS TV RS

Puzzle #2:
TV RS TS
SC SV TC
SS RV RC

Puzzle #3:
TV TC RV
SS SC RS
TS SV RC
```

UCF Local Programming Contest — August 30, 2014

Shopping Spree

filename: `shop`
(*Difficulty Level:* `Hard`)

You've won a shopping spree with a very peculiar rule. The items you are allowed to take are in consecutive order, indexed 1 through *n*. You may select any subset of these items subject to the following constraint:

> For each index *k* of items chosen for the subset
> at most half of the items with indexes 1 through *k* may be in the subset

For example, if the item with index 10 is chosen for the subset, then your selected subset can contain at most half of the items with index 1 through 10. Similarly, if the item with index 2 is chosen for the subset, then your selected subset can contain at most half of the items with index 1 through 2. Note that "half" is an integer value so half of 10 and 11 are both 5. The only exception to the constraint is that if the item with index 1 is chosen for the subset, you can select 1 item and not zero (to be fair).

The Problem:

Given a list of the dollar values of items, I_1, I_2, … I_n, in the shopping spree, determine the maximum value you can obtain from the shopping spree subject to the above constraint.

The Input:

The first line of the input is a positive integer, *n*, indicating the number of shopping sprees that your program will have to analyze. Following this will be the descriptions of each shopping spree. Each shopping spree will be described on a single line. The first value on each of these lines will be a single positive integer, *s (s ≤ 500)*, representing the number of items for the shopping spree. The next *s* space-separated positive integers will be the values for the shopping spree items in dollars, in order. Each of these values will be less than or equal to 10^6.

The Output:

For each shopping spree, first output the heading "`Spree #d:` ", where *d* is the spree number, starting with 1. Then, print a single integer equal to the maximum value, in dollars, that can be obtained for that shopping spree. Follow the format illustrated in Sample Output.

Sample Input:

```
2
5
1 2 3 4 5
3
12 2 4
```

Sample Output:

```
Spree #1: 9
Spree #2: 12
```

UCF Local Programming Contest — August 30, 2014

Factorial Products

filename: `fact`
(*Difficulty Level:* `Hard`)

Factorial is just a game of multiplications. Formally, it can be defined as a recurrence relation:

Fact (0) = 1
Fact (n) = n * Fact(n-1), for all integers n > 0

This problem is all about multiplications, more and more multiplications. It is a game of multiplications of factorials.

The Problem:

You will be given three lists of numbers: A, B and C. You have to take the factorials of all the numbers in each list and multiply them to get ProFact(A), ProFact(B), ProFact(C). Then report which product is the largest.

For example, consider the lists A = {2, 4, 7}, B = {0, 1, 9} and C = {2, 3, 5, 5}. Then,

ProFact(A) = 2! * 4! * 7! = 241, 920
ProFact(B) = 0! * 1! * 9! = 362, 880
ProFact(C) = 2! * 3! * 5! * 5! = 172,800

So, the largest product for this example is ProFact(B).

The Input:

The first input line contains a positive integer, *n*, indicating the number of test cases. Each test case consists of four input lines. The first line consists of three positive integers providing, respectively, the size for the lists A, B and C. The last three lines contain, respectively, the elements (non-negative integers) in lists A, B and C.

All the values in the input file will be less than 2,501.

The Output:

For each test case, output "`Case #`*t*`: `*h*" in a line, where *t* is the case number (starting with 1) and *h* is the list name with the highest product. If two or three lists are tied for the highest product, print "`TIE`". Follow the format illustrated in Sample Output.

Assume that, if the pairwise product values differ, then the relative difference of these products will differ by at least 0.01% of the largest product.

Sample Input:

```
3
3 3 4
2 4 7
0 1 9
2 3 5 5
2 2 2
2 3
3 2
2 2
3 3 3
1 3 5
2 4 6
1 4 7
```

Sample Output:

```
Case #1: B
Case #2: TIE
Case #3: C
```

UCF Local Programming Contest — August 30, 2014

Super Lucky Palindromes

filename: `lucky`
(*Difficulty Level:* `Hard`)

Lucky numbers are positive integers composed only of the digits '4' and '7'. For example, 47477 and 777 are lucky numbers while 457 and 1232 are not.

Super lucky numbers have the following additional properties:

- They are a lucky number themselves
- Number of digits in them is a lucky number
- The number of '4's or the number of '7's in them is a lucky number (or both counts are lucky numbers).

A palindrome is an integer that reads the same forwards and backwards. For example, 547745 and 343 are palindromes while 74 and 12345 are not. A super lucky palindrome is a positive integer that is both a super lucky number and a palindrome.

The Problem:

Given a number *k*, print the k^{th} smallest super lucky palindrome.

The Input:

The first input line contains a positive integer, *n*, indicating the quantity of numbers to check. Each of the next *n* lines contains a single integer, *k* $(1 \leq k \leq 10^{18})$.

The Output:

For each query, first output the heading "`Query #d:` ", where *d* is the query number, starting with 1. Then, for the value *k* given in the query, print the k^{th} smallest super lucky palindrome. Follow the format illustrated in Sample Output.

Sample Input:

```
5
1
2
3
5
100
```

Sample Output:

```
Query #1: 4444
Query #2: 7777
Query #3: 4444444
Query #4: 4747474
Query #5: 444444447447444444444444444444744744444444
```

UCF Local Programming Contest — August 30, 2014

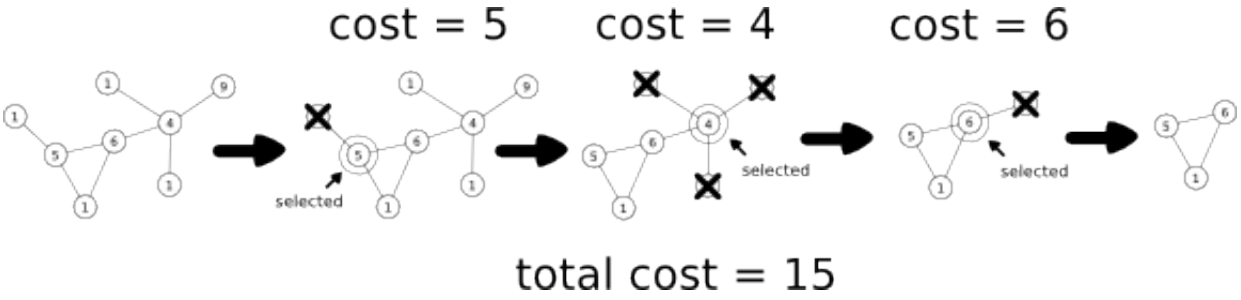
Under Construction Forever

filename: `ucf`
(*Difficulty Level:* `Hard`)

UCF is now the nation's second largest university in student population. To meet the demands of a growing student body, the university is always constructing new buildings. One of the issues is the campus needs many new buildings but has limited space. To help make room for new buildings, a new restructuring method is being used!

Each day (until the end of restructuring) a building is selected for "reconstruction". During reconstruction, all buildings that are connected to the selected building *and only to the selected building* are combined into (torn down and then added as new parts to) the selected building. Every building has a given cost associated with applying this reconstruction operation; when a building is selected and other buildings are combined with it, the cost associated with the selected building is the cost for combining the selected group of buildings.

For example, in the building layout below, the building with cost 5 is selected in Day 1. The building with cost 1 (crossed out in the picture in Day 1) is the only building connected to the selected building and this building is connected to no other building. These two buildings are combined and the cost of the operation is 5 (the cost of the selected building). In Day 2, the building with cost 4 is selected (hence the cost 4 for combining) and in Day 3, the building with cost 6 is selected. No more buildings can be selected after Day 3 since every building is connected to more than one building. The total cost for this sequence of combinations is therefore 5+4+6= 15.



The Problem:

Given the building connections and the costs associated with refactoring the buildings, determine the minimum possible number of buildings that will be left when refactoring is complete, the minimum cost of achieving this minimum size and the number of ways to achieve this minimum cost with minimum size. Two ways are considered different if on the i^{th} day the building being reconstructed is different or the number of days to complete the reconstruction differs. Since the number of ways to reconstruct the university can be quite large, print the result modulo 1,000,000,007.

The Input:

The first input line contains a positive integer, *n*, indicating the number of restructurings to perform. Each restructuring will contain multiple lines. The first line contains two integers, b $(1 \le b \le 500)$ and c $(1 \le c \le 2000)$, representing (respectively) the number of buildings and the number of connections between them. The next line contains b integers, w_i $(1 \le w_i \le 500)$, representing the cost of performing a reconstruction on the i^{th} selected building. The next c lines each contain two integers, x_i and y_i $(1 \le x_i \le b,\ 1 \le y_i \le b,\ x_i \ne y_i)$, representing two identifiers for buildings that connect. There will be at most one direct connection between any two buildings. Also, every pair of buildings will be directly or indirectly connected.

The Output:

For each restructuring, first output the heading "`Case #d:` ", where *d* is the test case number, starting with 1. Then, print three integers separated by a single space: the minimum number of buildings left, the minimum cost for achieving this configuration, and the number of ways to achieve the minimum cost with minimum size. Follow the format illustrated in Sample Output.

Sample Input:

```
3
3 2
1 2 3
3 1
3 2
8 7
80 4 3 2 1 90 5 80
1 2
2 3
3 4
4 5
5 6
4 7
7 8
8 8
1 5 6 1 1 4 1 9
1 2
2 3
3 4
4 2
3 6
6 7
6 5
6 8
```

Sample Output:

```
Case #1: 1 3 1
Case #2: 1 15 28
Case #3: 3 15 3
```