UCF Programming Team Practice
September 10, 2016
Developmental Team
Individual Problem Set

Problem Name	Filename
Ant Tunneling	ant
Calculator Games	calc
Exact Change	change
Driving in Circles	circles
Strings with Same Letters	letters
Push Button Lock	lock
Candy Combinations	morecombos
Shuffle'm Up	shuffle
Upwards	upwards
Jimmy's Perfect Vacation	vacation

Ant Tunneling

Filename: `ant`

The evil ants of UCF have taken your teacher hostage in order to force you to help them with a problem. They are trying to interconnect their ant hills, so that there is some way to get from any ant hill to any other ant hill using only tunnels they have dug. They are quite smart ants, so they have already mapped out all the possible tunnels they can dig to connect their ant hills and assigned each a cost. However, before they get to digging, they demand your help to determine which set of tunnels they should choose to dig to minimize the total cost while still connecting all of their ant hills (directly or indirectly). A pair of ant hills is considered connected if there exists a series of tunnels connected end-to-end that form a path between each of the hills. The ants can start digging from any ant hill, but in the end all pairs of hills must have a path of fully dug tunnels connecting them.

The Problem:

Given the cost of all possible tunnels to connect pairs of ant hills, determine the minimum total cost to dig a tunnel network that connects all ant hills.

The Input:

The first line will contain a positive integer, n, the number of campuses for which the ants want you to solve this problem (UCF has many satellite campuses). Following will be descriptions of n campuses. Each campus description starts with a line containing two integers, h ($1 \leq h \leq 100$) and t ($0 \leq t \leq h * (h - 1) / 2$), representing the number of ant hills and the number of tunnels the ants can possibly dig, respectively. Each ant hill is assigned a unique number from 1 to h. The following t lines will describe the possible tunnels. Each tunnel is described by three integers, x, y ($1 \leq x < y \leq h$) and d ($1 \leq d \leq 1{,}000$), representing the two ant hills this tunnel connects and the cost of digging this tunnel to connect them, respectively. For each campus, each possible pair of ant hills will have at most one tunnel between them.

The Output:

For each campus, the output should start with "`Campus #c: `" where *c* is the number of the campus in the input (starting at 1). Follow this with either a single integer which is the minimum sum of difficulty costs to dig tunnels in a matter that connects all ant hills (directly or indirectly) or with "`I'm a programmer, not a miracle worker!`" if there is no way to dig tunnels to connect them all.

Sample Input:

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

Sample Output:

```
Campus #1: 5
Campus #2: I'm a programmer, not a miracle worker!
Campus #3: 6
```

Calculator Games

Filename: `calc`

The Problem

You are given a calculating device that has a single display screen that displays non-negative integers less than 1 billion (upto 9 digits) and three buttons. The three buttons perform the following functions on the current value, x, on the display screen:

(1) Add 1
(2) Multiply by 3
(3) Integer Division by 2

You are given a starting positive integer n < 100 and your goal will be to figure out the minimum number of button presses necessary to reach all of the other positive integers less than 100. Your program should simply output the maximum of all of these numbers, namely, the most number of button presses needed to reach any particular value. Remember that since the display doesn't allow any numbers one billion or greater, no intermediate calculation can arrive at a result equal to or greater than one billion.

Input Format

The first line will contain a single positive integer, n < 100, specifying the number of input cases. Each input case will have a single positive integer, k < 100, on a line by itself representing the starting value for that case.

Output Format

For each test case, on a line by itself, output the desired maximum number of button presses to reach any of the numbers from 1 to 99 for that case.

Sample Input

```
3
1
73
99
```

Sample Output

```
10
11
12
```

UCF Local Contest — September 1, 2012

Exact Change

filename: `change`

Whenever the UCF Programming Team travels to World Finals, Glenn likes having the exact amount of money necessary for any purchase, so that he doesn't have to count and receive change. Of course, most countries don't have many different denominations of coins, so Glenn creates different "packages" with him, each with some particular amount of money, in cents. Glenn would like to know which amount of money (in cents), is the smallest that he can't pay for exactly, with some combination of his packages.

The Problem:

Given a list of positive integers, determine the smallest integer that can't be represented as the sum of some subset of the integers on the list.

The Input:

The first input line contains a single positive integer, n $(1 < n \le 100)$, indicating the number of sets of coin packages to evaluate. Each of the n input sets follows. The first line of each input set contains only an integer, c $(1 \le c < 31)$, representing the number of different packages of coin for that input set. The following line contains exactly c positive integers, each separated by a single space, representing the value of each of the c packages in cents. The sum of these c integers is guaranteed not to exceed 10^9. Note that the package values are not necessarily distinct, i.e., there may be multiple packages with the same value.

The Output:

For each set of packages, first output "`Set #i: `" where `i` is the input data set number, starting with 1. Follow this with a single positive integer, the smallest value that can't be represented as a sum of the values of a subset of the packages given. Note that a package value can be used at most once in a subset unless there are multiple packages with that value (if there are m occurrences of a package value, up to m occurrences of that value can be used in a subset). Leave a blank line after the output for each test case. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
3
6
12 8 1 2 4 100
3
1 2 3
6
3 1 3 2 3 3
```

Sample Output:

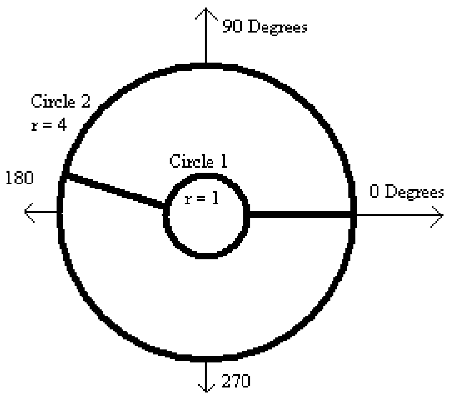
```
Set #1: 28

Set #2: 7

Set #3: 16
```

Driving in Circles

Filename: `circles`

The University of Central Florida (UCF) has a unique layout that makes it more interesting to drive through. Specifically, the roads form circles around the Student Union, with radial cross streets connecting circles. The whole setup can be disorienting for the delivery trucks that need to bring in packages to the center of UCF, since most drivers are not used to driving in circles when they already know where they are going. Luckily, no street or road on campus is one-way so drivers avoid that confusion.

As a new driver for a delivery company that includes UCF and other similarly arranged locations, Ali is finding that it can be difficult to figure out the fastest way to get to a delivery with all the circles. Determined to minimize the distance he has to travel to get to a delivery point, Ali decides that he'll make an application for his new smart phone to help him out. You just put in the layout of the roads and streets and it tells you how far you have to travel. Who knows, maybe there will be a market for just such an app!



The Problem:

Given a description of the map as well as starting and ending locations, determine the shortest trip.

If you need a value for π, use the one provided by your environment or 3.141592653589793 if one is not provided.

The Input:

The first line of the input will contain a positive integer, t ($t \leq 100$), giving the number of trips to follow. Each trip begins with a line containing the integers, c ($1 \leq c \leq 10$) and r (c-1 $\leq r \leq 20$), where c is the number of circular roads that surround a single center point and r is the number of radial streets connecting the circular roads. The next line will contain c integer circle radii in strictly increasing order, each separated by a single space.

The next r lines will each contain two integers, separated by a single space, describing a radial street, d ($1 \leq d < c$) and a ($0 \leq a < 360$). d is the one-based index of the starting circular road and represents connecting circular road d with next larger circular road d+1. a is an integer angle in degrees in relation to the center of all circular roads, with 0 degrees pointing directly to the East, and the angle increasing in the counter-clockwise direction (90 degrees would be North, etc.).

All circular roads will be connected by radial streets, and it will always be possible to reach one circular road from another. Radial streets will only connect to valid circular roads. No road or street will lie on top of another road.

After the r radial street descriptions, there will be four more integers on a single line, *sc sa fc fa* ($1 \leq sc$, $fc \leq c$ and $0 \leq sa$, $fa < 360$), denoting the starting and ending locations. *sc* is the one-based index of the starting circular road, and *sa* is the angle of the starting point compared to the center of the circular road. *fc* and *fa* are the destination circular road and the angle of the point on the destination circular road, respectively. Again, angles are in degrees and given in the same orientation as the radial street descriptions (0 degrees is East, 90 is North, etc.).

The Output:

For each trip, print a line containing only the shortest distance from the starting point to the ending point, traveling along circular roads and radial cross streets. This distance should be output rounded to two decimal points (for example, 2.435 should be rounded to 2.44 and 2.934 should be rounded to 2.93).

Sample Input:

```
2
1 0
1
1 0 1 180
2 2
1 4
1 0
1 170
2 180 1 90
```

Sample Output:

```
3.14
5.09
```

H. Strings with Same Letters

A professor assigned a program to his class for which the output is a string of lower-case letters. Unfortunately, he did not specify an ordering of the characters in the string, so he is having difficulty grading student submissions. Because of that, he has requested that ICPC teams help by writing a program that inputs pairs of strings of lower-case letters and determines whether or not the strings have the same letters, though possibly in different orders. Note that repeated letters are important; the string "abc" and "aabbbcccc" are not viewed as having the same letters since the second one has more copies of each letter.

Input:

Input to be processed will be pairs of lines containing nonempty strings of lower-case letters. All input will be valid until the end of file indicator. End of file will be signaled by two lines that contain just the word "END" in upper case. No input line will be longer than 1,000 characters.

Output:

Report whether pairs of strings have the same letters or not. Follow the format exactly: "`Case`", a space, the case number, a colon and one space, and the result given as "`same`" or "`different`" (lower-case, no punctuation). Do not print any trailing spaces.

Sample Input

```
testing
intestg
abc
aabbbcccc
abcabcbcc
aabbbcccc
abc
xyz
END
END
```

Sample Output

```
Case 1: same
Case 2: different
Case 3: same
Case 4: different
```

D • Push Button Lock

The *Frobozz Magic Lock Company* is in the business of manufacturing *push button* style combination door locks. A push button door lock consists of a number of push buttons **B**, (1 ≤ **B** ≤ 11), labeled "1" through "**B**". The lock is opened by pressing the correct sequence of button combinations and then turning the doorknob. If the sequence of presses is correct, the door *magically* opens.

A *combination* consists of 1 or more buttons being pressed simultaneously. A *sequence* consists of a series of combinations. A sequence must have at least one combination. Once a button has been used in a combination, it may not be used again in the same sequence. In addition, it is not necessary to use all the buttons in a sequence. For example, for **B**=8:

(1-2-3)(4)(7-8)

is a valid sequence with 3 combinations (1-2-3), (4), and (7-8). Note that buttons 5 and 6 are not used in this sequence.

(1-2-3)(2-4)(5-6)

is not a valid sequence, since button 2 appears in 2 combinations (1-2-3) and (2-4).

The CEO of Frobozz, *J. Pierpont Flathead*, wants you to write a program that determines the number of valid sequences possible for given values of **B**. The program must be able to process a list of lock orders (datasets) from customers and generate a report showing the order number, the value of **B**, and the number of valid sequences possible. This list will always contain at least one dataset, but no more than 100 datasets.

Input

The first line of input contains a single integer **N,** (1 ≤ **N** ≤ 100), representing the number of datasets that follow. Each dataset consists of a single line of data containing a single integer **B**, which is the number of buttons for the lock.

Output

For each dataset, display the dataset number, a blank, the value **B**, a blank, and the number of valid sequences.

Sample Input	Sample Output

3
3
4
3
	1 3 25
	2 4 149
	3 3 25

Reference Materials:

J. Pierpont Flathead

Candy Combinations

Filename: morecombos

You have won a contest where you get to choose several bags of candy out of a larger set of bags. Luckily, you know the contents of each bag. Your goal is to maximize the number of different candies you receive.

The Problem

Given the contents of several bags of candies, and a limit to the number of those bags you can choose, determine the maximum number of unique candies you can receive.

The Input

The first line of the input file will contain a number, *n* $(1 \leq n \leq 100)$, representing the number of contests you have to evaluate. The first line of each contest will contain two positive integers, *b* $(1 \leq b \leq 20)$ and *k* $(1 \leq k \leq b)$, representing the number of bags from which to choose and the maximum number of bags that you are allowed to choose, respectively, for this contest. The next *b* lines will contain information about the contents of each bag, respectively. The first value on each of these lines will be an integer *m* $(1 \leq m \leq 50)$, representing the number of candies in the corresponding bag. The following *m* integers will represent the number of each of the candies in the bag. Each of these integers will be in between 1 and 31, inclusive.

The Output

For each case, output the maximum number of unique candies (which will never be more than 31) you can obtain by choosing the specified number of bags.

Sample Input

```
2
3 2
4 1 1 1 1
2 2 3
2 4 5
4 1
10 6 5 5 5 4 6 5 5 5 4
4 1 2 3 4
5 2 2 2 3 3
6 7 7 1 1 3 3
```

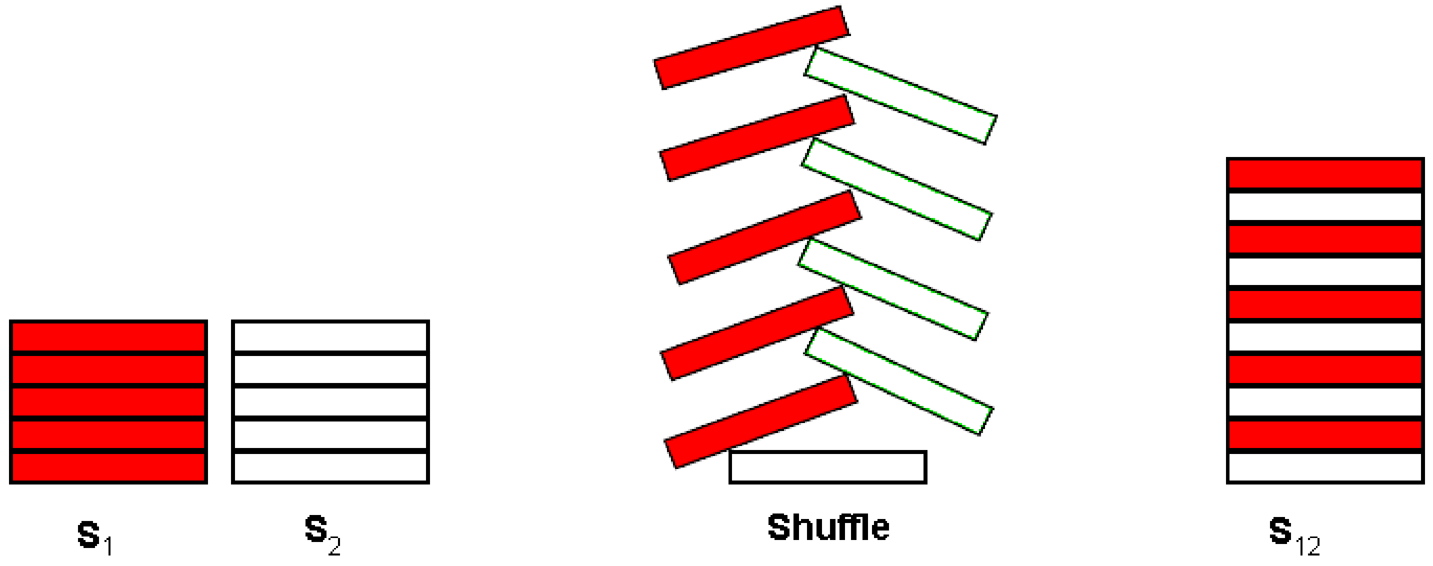
Sample Output

```
4
4
```

C • Shuffle'm Up

A common pastime for poker players at a poker table is to shuffle stacks of chips. Shuffling chips is performed by starting with two stacks of poker chips, $\mathbf{S_1}$ and $\mathbf{S_2}$, each stack containing **C** chips. Each stack may contain chips of several different colors.

The actual shuffle operation is performed by interleaving a chip from $\mathbf{S_1}$ with a chip from $\mathbf{S_2}$ as shown below for **C**=5:



The single resultant stack, $\mathbf{S_{12}}$, contains 2***C** chips. The bottommost chip of $\mathbf{S_{12}}$ is the bottommost chip from $\mathbf{S_2}$. On top of that chip, is the bottommost chip from $\mathbf{S_1}$. The interleaving process continues taking the 2nd chip from the bottom of $\mathbf{S_2}$ and placing that on $\mathbf{S_{12}}$, followed by the 2nd chip from the bottom of $\mathbf{S_1}$ and so on until the topmost chip from $\mathbf{S_1}$ is placed on top of $\mathbf{S_{12}}$.

After the shuffle operation, $\mathbf{S_{12}}$ is split into 2 new stacks by taking the bottommost **C** chips from $\mathbf{S_{12}}$ to form a new $\mathbf{S_1}$ and the topmost **C** chips from $\mathbf{S_{12}}$ to form a new $\mathbf{S_2}$. The shuffle operation may then be repeated to form a new $\mathbf{S_{12}}$.

For this problem, you will write a program to determine if a particular resultant stack $\mathbf{S_{12}}$ can be formed by shuffling two stacks some number of times.

Input

The first line of input contains a single integer **N**, ($1 \le \mathbf{N} \le 1000$) which is the number of datasets that follow.

Each dataset consists of four lines of input. The first line of a dataset specifies an integer **C**, ($1 \le \mathbf{C} \le 100$) which is the number of chips in each initial stack ($\mathbf{S_1}$ and $\mathbf{S_2}$). The second line of each dataset specifies the colors of each of the **C** chips in stack **S**$_1$, starting with the bottommost chip. The third line of each dataset specifies the colors of each of the **C** chips in stack **S**$_2$ starting with the bottommost chip. Colors are expressed as a single uppercase letter (**A** through **H**). There are no blanks or separators between the chip colors. The fourth line of each dataset contains 2*__C__ uppercase letters, (**A** through **H**), representing the colors of the desired result of the shuffling of **S**$_1$ and **S**$_2$ zero or more times. The bottommost chip's color is specified first.

Output

Output for each dataset consists of a single line that displays the dataset number (1 though **N**), a space, and an integer value which is the *minimum* number of shuffle operations required to get the desired resultant stack. If the desired result can not be reached using the input for the dataset, display the value negative 1 (**-1**) for the number of shuffle operations.

Sample Input	Sample Output
2	1 2
4	2 -1
AHAH	
HAHA	
HHAAAAHH	
3	
CDE	
CDE	
EEDDCC	

Upwards

Filename: upwards

In a previous contest hosted by Kyle, the question was posed whether or not an input word was an “upword.” In order to be such a word, all of its letters have to appear in alphabetical order, with no repeated letters. For example, “act” is an “upword”, but “cat” and “deep” are not. Arup really likes these words and wants you to investigate them further. In addition, he’s added a new definition. The original “upward” is a level-0 upward. In order to be a level-1 upward, all the letters in the word have to be in alphabetical order and the gap between consecutive letters must contain at least one letter. Thus, “ace” is a level-1 upward, but “hit” is not. In general, we can define a level-k upward to be a word in alphabetical order where the gap between consecutive letters must contain at least k letters.

Now that Arup has made his definition, he wants to know of all the level-k upwards of exactly n letters, which one is of a particular rank, r, given values for k, n and r. The rank of a word in the list is based on alphabetical order. For example, of all level-1 upwords of length 3, “ace” has rank 1 (it’s the first) and “act” has rank 16.

The Problem

Given a level, k, number of letters, n, and a positive integer rank, r, find the rth word in alphabetical order of all the level-k upwords of length n.

The Input

The first line of the input file will contain a single positive integer, c $(c \le 100)$, representing the number of input cases. The input cases follow, one per line. Each of these lines will have the integers, k $(0 \le k \le 24)$, n $(2 \le n \le 26)$ and r $(1 \le r \le 10000)$, respectively, separated by spaces. Note that r is bounded by 10000, so even if a very large number of words exist, no more than 10000 will have to be generated. Also, r is guaranteed to be less than or equal to the total number of k-level upwords with exactly n letters.

The Output

For each case, output the correct string for the query, in all lowercase letters.

Sample Input

```
2
1 3 16
0 25 24
```

Sample Output

```
act
abdefghijklmnopqrstuvwxyz
```

Jimmy's Perfect Vacation

Filename: vacation

Jimmy and his family are planning a vacation. They all love amusement parks, and they always try to go on every ride that the park has. Unfortunately, they only have a certain amount of time in the day. Not only does timing make things complicated, some of the paths between two rides have been blocked off for parades, construction, and mothers tending to crying babies. To make matters even worse, Jimmy is very sensitive to rides and throws up every time he gets off, so he never wants to go near a ride he has already ridden. They could write out by hand to figure out the fastest way to visit each ride, but Jimmy has a good friend (you)! Help Jimmy and his family find the fastest way to visit all of the rides. Of course, they will be going on a lot more vacations, so your program should be able to handle multiple vacations.

The Problem:

Given each ride the amusement park has, as well as a list of blocked paths between two rides, find the fastest time it will take for Jimmy and his family to visit each one. Assume that he and his family walk at 1 meter per second, and they can always travel to their destination in a straight line assuming that the path is not blocked. Rides always take a constant time of 120 seconds per ride (they always visit the parks during the off-season so the waiting time is negligible).

The Input:

Input will begin with a single integer, *n*, which is the number of different parks Jimmy and his family have planned to visit. For each park, there will be two non-negative integers, *r* and *b,* which are the number of rides and blocked paths respectively ($r < 11$, $b < r^2$). On the following *r* lines are two integers, *x* and *y*, denoting the Cartesian coordinates of the ride (in meters). The rides are numbered from 1 to *r* for simplicity, and no two rides will have the same coordinates. After these *r* lines are *b* lines, each giving two integers *i* and *j* stating that the path between ride *i* and ride *j* is blocked, and therefore, the path from *j* to *i* is blocked as well. All coordinates will be non-negative integers less than 1,000, and Jimmy and his family always start their visit at the entrance, which is at the origin, (0,0). There will never be a blocked path that includes the entrance, and you may never revisit the entrance (they consider that exiting the park and make you buy another ticket!). Assume there is a path between every pair of rides unless it was blocked.

The Output:

For each vacation, first output a header, on a line by itself, stating "`Vacation #`*`k`*`:`" where *`k`* is the number of the vacation, starting with 1. If Jimmy can and his family can ride every ride, output a new line "`Jimmy can finish all of the rides in` *`s`* `seconds.`" where *s* is the number of seconds rounded to the nearest thousandth of a second it takes to ride every ride (as an example of rounding, a time of 100.5455 would be rounded up to 100.546, whereas a time of 100.5454 would be rounded down to 100.545). If Jimmy and his family cannot ride every ride, output one line saying "`Jimmy should plan this vacation a different day.` " instead. Output a single blank line after the output for each vacation.

Sample Input:

```
3
4 0
0 2
2 2
2 4
4 4
5 4
10 10
12 35
64 60
3 7
100 857
1 2
1 3
1 4
1 5
5 2
0 5
0 10
0 20
0 50
0 25
3 4
1 2
```

Sample Output:

```
Vacation #1:
Jimmy can finish all of the rides in 488.000 seconds.

Vacation #2:
Jimmy should plan this vacation a different day.

Vacation #3:
Jimmy can finish all of the rides in 670.000 seconds.
```