CSCI 269 – Fall 2014 Handout: Follow The Rules
Computational Problem Solving
Michael H. Goldwasser
Saint Louis University Tuesday, 9 September 2014

1 Follow the Rules

For today's practice, I have chosen a theme of problems that I consider the "follow the rules" variety. While these are not trivial problems to solve, solutions do not require knowledge of any advanced data structures or algorithms.

The particular problems I'm selecting are all taken from recent years in our Mid-Central regional. I have intentionally listed the problem in decreasing order of team's success rate; you can decide whether you wish to start at the top or bottom of this list. The judge's data for all of these problems are loaded onto turing so that you can use our submit script to test your solutions. (See `http://cs.slu.edu/~goldwasser/icpc/resources/` for details and `http://cs.slu.edu/~goldwasser/cgi-bin/icpc` for links to problem statements.)

shortname	Problem	contest	success	notes
mad	Mad Scientist	2010 Mid-Central	90%	
lru	LRU Caching	2012 Mid-Central	67%	
juggle	Jugglefest	2012 Mid-Central	60%	
round	Round Robin	2013 Mid-Central	60%	
sokoban	Sokoban	2011 Mid-Central	40%	2D grid
crash	Crash and Go(relians)	2011 Mid-Central	35%	geometry
bridges	The Bridges of San Motchi	2008 Mid-Central	22%	

Problem B: Mad Scientist

Source file: `mad.{c, cpp, java}`
Input file: `mad.in`

A mad scientist performed a series of experiments, each having *n* phases. During each phase, a measurement was taken, resulting in a positive integer of magnitude at most *k*. The scientist knew that an individual experiment was designed in a way such that its measurements were monotonically increasing, that is, each measurement would be at least as big as all that precede it. For example, here is a sequence of measurements for one such experiment with $n=13$ and $k=6$:

1, 1, 2, 2, 2, 2, 2, 4, 5, 5, 5, 5, 6

It was also the case that *n* was to be larger than *k*, and so there were typically many repeated values in the measurement sequence. Being mad, the scientist chose a somewhat unusual way to record the data. Rather than record each of *n* measurements, the scientist recorded a sequence *P* of *k* values defined as follows. For $1 \leq j \leq k$, $P(j)$ denoted the number of phases having a measurement of *j* or less. For example, the original measurements from the above experiment were recorded as the *P*-sequence:

2, 7, 7, 8, 12, 13

as there were two measurements less than or equal to 1, seven measurements less than or equal to 2, seven measurement less than or equal to 3, and so on.

Unfortunately, the scientist eventually went insane, leaving behind a notebook of these *P*-sequences for a series of experiments. Your job is to write a program that recovers the original measurements for the experiments.

Input: The input contains a series of *P*-sequences, one per line. Each line starts with the integer *k*, which is the length of the *P*-sequence. Following that are the *k* values of the *P*-sequence. The end of the input will be designated with a line containing the number *0*. All of the original experiments were designed with $1 \leq k < n \leq 26$.

Output: For each *P*-sequence, you are to output one line containing the original experiment measurements separated by spaces.

Example Input:	**Example Output:**

`6 2 7 7 8 12 13`
`1 4`
`3 4 4 5`
`3 0 4 5`
`5 2 2 4 7 7`
`0`
	`1 1 2 2 2 2 2 4 5 5 5 5 6`
	`1 1 1 1`
	`1 1 1 1 3`
	`2 2 2 2 3`
	`1 1 3 3 4 4 4`

Problem F: LRU Caching

Source file: `lru.{c, cpp, java}`
Input file: `lru.in`

When accessing large amounts of data is deemed too slow, a common speed up technique is to keep a small amount of the data in a more accessible location known as a *cache*. The first time a particular piece of data is accessed, the slow method must be used. However, the data is then stored in the cache so that the next time you need it you can access it much more quickly. For example, a database system may keep data cached in memory so that it doesn't have to read the hard drive. Or a web browser might keep a cache of web pages on the local machine so that it doesn't have to download them over the network.

In general, a cache is much too small to hold all the data you might possibly need, so at some point you are going to have to remove something from the cache in order to make room for new data. The goal is to retain those items that are more likely to be retrieved again soon. This requires a sensible algorithm for selecting what to remove from the cache. One simple but effective algorithm is the Least Recently Used, or LRU, algorithm. When performing LRU caching, you always throw out the data that was least recently used.

As an example, let's imagine a cache that can hold up to five pieces of data. Suppose we access three pieces of data—A, B, and C. As we access each one, we store it in our cache, so at this point we have three pieces of data in our cache and two empty spots (Figure 1). Now suppose we access D and E. They are added to the cache as well, filling it up. Next suppose we access A again. A is already in the cache, so the cache does not change; however, this access counts as a use, making A the most recently used. Now if we were to access F, we would have to throw something out to make room for F. At this point, B has been used least recently, so we throw it out and replace it with F (Figure 2). If we were now to access B again, it would be exactly as the first time we accessed it: we would retrieve it and store it in the cache, throwing out the least recently used data—this time C—to make room for it.

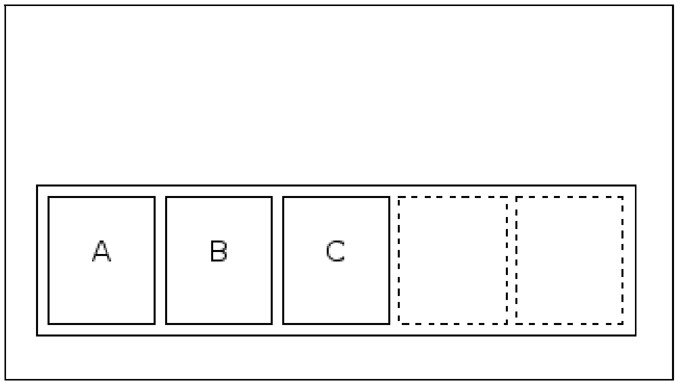


Figure 1: Cache after A, B, C

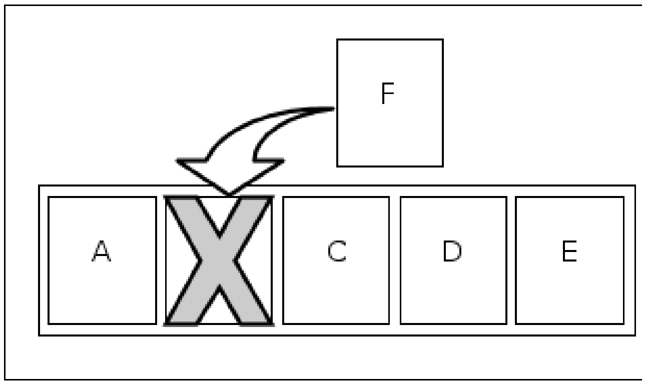


Figure 2: Cache after A, B, C, D, E, A, F

Your task for this problem is to take a sequence of data accesses and simulate an LRU cache. When requested, you will output the contents of the cache, ordered from least recently used to most recently used.

Input: The input will be a series of data sets, one per line. Each data set will consist of an integer *N* and a string of two or more characters. The integer *N* represents the size of the cache for the data set ($1 \leq N \leq 26$). The string of characters consists solely of uppercase letters and exclamation marks. An upppercase letter represents an access to that particular piece of data. An exclamation mark represents a request to print the current contents of the cache.

For example, the sequence *ABC!DEAF!B!* means to acces A, B, and C (in that order), print the contents of the cache, access D, E, A, and F (in that order), then print the contents of the cache, then access B, and again print the contents of the cache.

The sequence will always begin with an uppercase letter and contain at least one exclamation mark.

The end of input will be signaled by a line containing only the number zero.

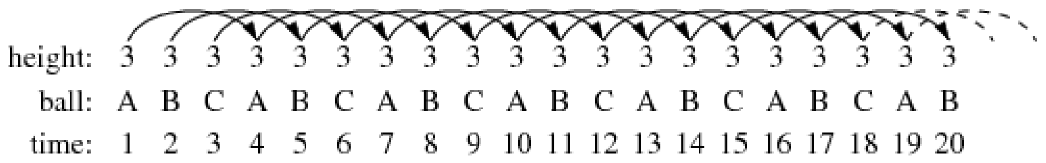
Output: For each data set you should output the line "Simulation *S*", where *S* is 1 for the first data set, 2 for the second data set, etc. Then for each exclamation mark in the data set you should output the contents of the cache on one line as a sequence of characters representing the pieces of data currently in the cache. The characters should be sorted in order from least recently used to most recently used, with least recently occuring first. You only output the letters that are in the cache; if the cache is not full, then you simply will have fewer characters to output (that is, do not print any empty spaces). Note that because the sequence always begins with an uppercase letter, you will never be asked to output a completely empty cache.

Example input:	**Example output:**
`5 ABC!DEAF!B!`	`Simulation 1`
`3 WXWYZ!YZWYX!XYXY!`	`ABC`
`5 EIEIO!`	`CDEAF`
`0`	`DEAFB`
	`Simulation 2`
	`WYZ`
	`WYX`
	`WXY`
	`Simulation 3`
	`EIO`

Problem G: Jugglefest

Source file: `juggle.{c, cpp, java}`
Input file: `juggle.in`

Many people are familiar with a standard 3-ball juggling pattern in which you throw ball A, then ball B, then ball C, then ball A, then ball B, then ball C, and so on. Assuming we keep a regular rhythm of throws, a ball that is thrown higher into the air will take longer to return, and therefore will take longer before the next time it gets thrown. We say that a ball thrown to height *h* will not be thrown again until precisely *h* steps later in the pattern. For example, in the standard 3-ball pattern, we say that each ball is thrown to a height of 3, and therefore thrown again 3-steps later in the pattern. For example, ball A that we throw at time 1 of the process will be next thrown at time 4.

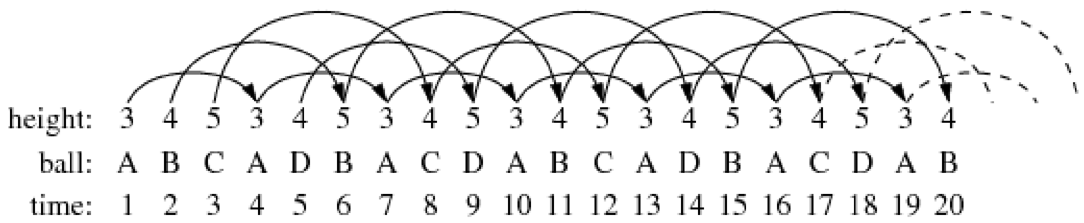

height: 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
ball: A B C A B C A B C A B C A B C A B C A B
time: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

By convention, we label the first ball thrown as A, and each time we introduce a new ball into the pattern, we label it with the next consecutive uppercase letter (hence B and then C in the classic pattern).

There exist more complex juggling patterns. Within the community of jugglers, a standard way to describe a pattern is through a repeating sequence of numbers that describe the height of each successive throw. This is known as the ***siteswap*** notation.

To demonstrate the notation, we first consider the "3 4 5" siteswap pattern. This describes an infinite series of throws based on the repeating series "3 4 5 3 4 5 3 4 5 ...". The first throw the juggler makes will be to a height of 3, the second throw will be to a height of 4, the third throw to a height of 5, the fourth throw to a height of 3 (as the pattern repeats), and so forth.

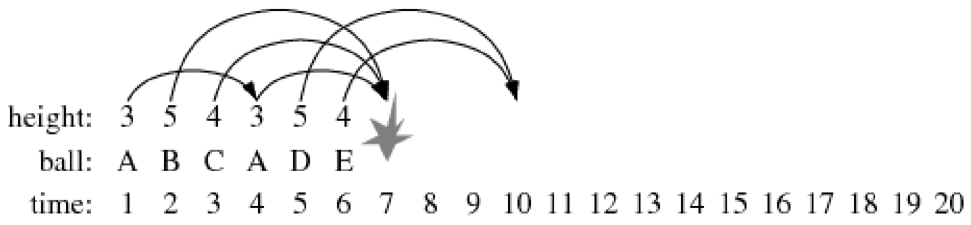
While the siteswap pattern describes the heights of the throws, the actual movement of individual balls does not follow as obvious a pattern. The following diagram illustrates the beginning of the "3 4 5" pattern.


height: 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 4 5 3 4
ball: A B C A D B A C D A B C A D B A C D A B
time: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

The first throw is ball A, thrown to a height of 3, and thus ball A is not thrown again until time 4. At time 2, we must make a throw with height 4; since ball A has not yet come back, we introduce a second ball, conventionally labeled B. Because ball B is thrown at time 2 with a height of 4, it will not be thrown again until time 6. At time 3, we introduce yet another ball, labeled C, and throw it to height 5 (thus it will next be thrown at time 8). Our next throw, at time 4, is to have height 3. However, since ball A has returned (from its throw at time 1), we do not introduce a new ball; we throw A. At time 5, we are to make a throw with height 4, yet we must introduce a new ball, D, because balls A, B, and C are all still up in the air. (Ball D is the last ball to be introduced for this particular pattern.) The juggling continues with ball B being thrown to height 5 at time 6, and so on.

The "3 4 5" siteswap pattern works out nicely. It happens to be a 4-ball pattern, because after introducing ball D, the juggler can now continue until his or her arms get tired. Unfortunately, not all siteswap sequences are legitimate!

Consider an attempt to use a siteswap pattern "3 5 4". If we were only interested in making six throws, everything works well. But a problem arises at time 7, as shown in the following diagram.



Ball B was thrown at time 2 with a height of 5. Therefore, it should get its next turn to be thrown at time 7. However, ball C was thrown at time 3 with a height of 4, and so it too should get its next turn at time 7. (To add insult to injury, ball A gets thrown at time 4 with height of 3, also suggesting it get its next turn at time 7.) What we have here is a problem, resulting in a lot of balls crashing to the ground.

Input: Each line represents a separate trial. It starts with the number *1* $\leq$ *P* $\leq$ *7* which represents the period of the repeating pattern, followed by *P* positive numbers that represent the throw heights in the pattern. An individual throw height will be at most 19. The input is terminated with a single line containing the value 0.

Output: For each pattern, output a single line describing the first 20 throws for the given pattern, if 20 throws can be legally made. Otherwise, output the word `CRASH`. You need not be concerned with any crashes due to balls landing strictly after time 20.

Example input:	**Example output:**

```
3 3 4 5
1 3
3 3 5 4
5 7 7 7 3 1
0
```
	```
	ABCADBACDABCADBACDAB
	ABCABCABCABCABCABCAB
	CRASH
	ABCDEEDABCCBEDAADCBE
	```

Problem B: Round Robin

Source file: `round.{c, cpp, java}`
Input file: `round.in`

Suppose that N players sit in order and take turns in a game, with the first person following the last person, to continue in cyclic order. While doing so, each player keeps track of the number of turns he or she has taken. The game consists of rounds, and in each round T turns are taken. After a round, the player who just had a turn is eliminated from the game. If the remaining players have all had the same number of turns, the game ends. Otherwise, they continue with another round of T moves, starting with the player just after the one who was most recently eliminated.

As an example, assume we begin a game with N=5 and T=17, labeling the players in order as A, B, C, D, and E, with all counts initially zero.

Player	A	B	C	D	E
Count	0	0	0	0	0

Beginning with A, 17 turns are taken. B will have taken the last of those turn, leaving our counts as follows:

Player	A	B	C	D	E
Count	4	4	3	3	3

Suppose that after *every* 17 turns, the player who just had a turn is eliminated from the game. All remaining players then compare their counts. If all of those counts are equal, everyone has had a fair number of turns and the game is considered completed. Otherwise, they continue with another round of 17 moves starting with the player just after the one who was most recently eliminated.

Continuing with our example, B will leave the game, and the next turn will be for C.

Player	A	C	D	E
Count	4	3	3	3

After 17 more turns starting with C, we find that A, D and E received 4 turns, while C received 5 turns, including the last:

Player	A	C	D	E
Count	8	8	7	7

Then C leaves, and since the remaining counts are not all the same, a new round beings with D having the next turn.

Player	A	D	E
Count	8	7	7

The next 17 turns start with D and end with E.   A adds 5 turns, while D and E add 6:

Player	A	D	E
Count	13	13	13

Then E leaves.

Player	A	D
Count	13	13

At this point, notice that the two remaining players have the same count of 13. Therefore, the game ends. (We note that E's count was irrelevant to the decision to end the game.)

Input: The input will contain one or more datasets. Each dataset will be described with a single line containing two integers, *N* and *T*, where *N* ($2 \le N \le 100$) is the initial number of players, and *T* ($2 \le T \le 100$) is the number of turns after which the player with the most recently completed turn leaves. Following the last dataset is a line containing only 0.

Output: There is one line of output for each dataset, containing two numbers, *p* and *c*. At the time the game ends *p* is the number of players that remain in the game and *c* is the common count they all have.

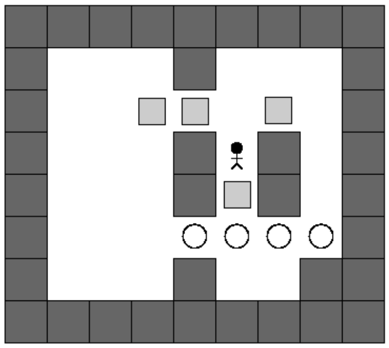
Example input:	**Example output:**

```
5 17
7 10
90 2
0
```
	```
	2 13
	5 3
	45 1
	```

ACM Mid-Central Programming Competition 2013

Problem G: Sokoban

Source file: `sokoban.{c, cpp, java}`
Input file: `sokoban.in`



Soko-ban is a Japanese word for a warehouse worker, and the name of a classic computer game created in the 1980s. It is a one-player game with the following premise. A single worker is in an enclosed warehouse with one or more boxes. The goal is to move those boxes to a set of target locations, with the number of target locations equalling the number of boxes. The player indicates a direction of motion for the worker using the arrow keys (up, down, left, right), according to the following rules.

- If the indicated direction of motion for the worker leads to an empty location (i.e., one that does not have a box or wall), the worker advances by one step in that direction.
- If the indicated direction of motion would cause the worker to move into a box, and the location on the other side of the box is empty, then both the worker and the box move one spot in that direction (i.e., the worker pushes the box).
- If the indicated direction of motion for a move would cause the worker to move into a wall, or to move into a box that has another box or a wall on its opposite side, then no motion takes place for that keystroke.

The goal is to simultaneously have all boxes on the target locations. In that case, the player is successful (and as a formality, all further keystrokes will be ignored).

The game has been studied by computer scientists (in fact, one graduate student wrote his entire Ph.D. dissertation about the analysis of sokoban). Unfortunately, it turns out that finding a solution is very difficult in general, as it is both NP-hard and PSPACE-complete. Therefore, your goal will be a simpler task: simulating the progress of a game based upon a player's sequence of keystrokes. For the sake of input and output, we describe the state of a game using the following symbols:

Symbol	**Meaning**
`.`	`empty space`
`#`	`wall`
`+`	`empty target location`
`b`	`box`
`B`	`box on a target location`
`w`	`worker`
`W`	`worker on a target location`

For example, the initial configuration diagrammed at the beginning of this problem appears as the first input case below.

Input: Each game begins with a line containing integers *R* and *C*, where 4 ≤ *R* ≤ 15 represents the number of rows, and 4 ≤ *C* ≤ 15 represents the number of columns. Next will be *R* lines representing the *R* rows from top to bottom, with each line having precisely *C* characters, from left-to-right. Finally, there is a line containing at most 50 characters describing the player's sequence of keystrokes, using the symbols `U`, `D`, `L`, and `R` respectively for up, down, left, and right. You must read that entire sequence from the input, even though a particular game might end successfully prior to the end of the sequence. The data set ends with the line `0 0`.

We will guarantee that each game has precisely one worker, an equal number of boxes and locations, at least one initially misplaced box, and an outermost boundary consisting entirely of walls.

Output: For each game, you should first output a line identifying the game number, beginning at 1, and either the word `complete` or `incomplete`, designating whether or not the player successfully completed that game. Following that should be a representation of the final board configuration.

Example input:	**Example output:**
8 9	Game 1: incomplete
#########	#########
#...#...#	#...#...#
#..bb.b.#	#..bb...#
#...#w#.#	#...#.#.#
#...#b#.#	#...#.#.#
#...++++#	#...+W+B#
#...#..##	#...#b.##
#########	#########
ULRURDDDUULLDDD	Game 2: complete
6 7	#######
#######	#..####
#..####	#.B.B.#
#.+.+.#	#.w.#.#
#.bb#w#	##....#
##....#	#######
#######	
DLLUDLULUURDRDDLUDRR	
0 0	

Problem H: Crash and Go(relians)

Source file: `crash.{c, cpp, java}`
Input file: `crash.in`

The Gorelians are a warlike race that travel the universe conquering new worlds as a form of recreation. Generally, their space battles are fairly one-sided, but occasionally even the Gorelians get the worst of an encounter. During one such losing battle, the Gorelians' space ship became so damaged that the Gorelians had to evacuate to the planet below. Because of the chaos (and because escape pods are not very accurate) the Gorelians were scattered across a large area of the planet (yet a small enough area that we can model the relevant planetary surface as planar, not spherical). Your job is to track their efforts to regroup. Fortunately, each escape pod was equipped with a locator that can tell the Gorelian his current coordinates on the planet, as well as with a radio that can be used to communicate with other Gorelians. Unfortunately, the range on the radios is fairly limited according to how much power one has.

When a Gorelian lands on the alien planet, the first thing he does is check the radio to see if he can communicate with any other Gorelians. If he can, then he arranges a meeting point with them, and then they converge on that point. Once together, they are able to combine the power sources from their radios, which gives them a larger radio range. They then repeat the process—see who they can reach, arrange a meeting point, combine their radios—until they finally cannot contact any more Gorelians.

Gorelian technology allows two-way communication as long as *at least one of them* has a radio with enough range to cover the distance between them. For example, suppose Alice has a radio with a range of 40 km, and Bob has a range of 30 km, but they are 45 km apart (Figure 1). Since neither has a radio with enough range to reach the other, they cannot talk. However, suppose they were only 35 km apart (Figure 2). Bob's radio still does not have enough range to reach Alice, but that does not matter—they can still talk because Alice's radio has enough range to reach Bob.

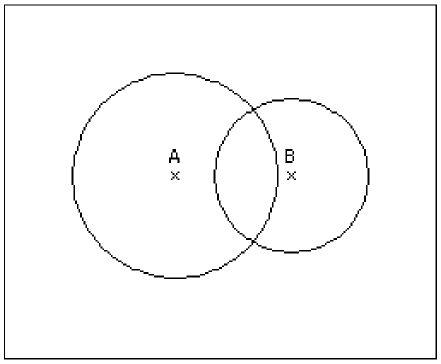


*Figure 1: Alice and Bob can **not** talk*

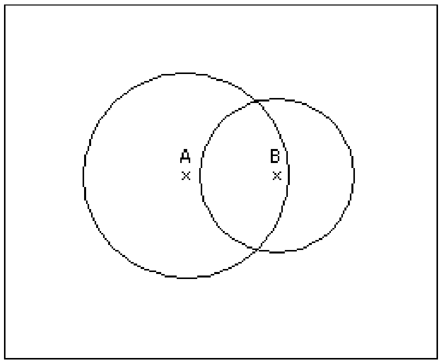


Figure 2: Alice and Bob can talk

If a Gorelian successfully contacts other Gorelians, they will meet at the point that is the average of all their locations. In the case of Alice and Bob, this would simply be the midpoint of A and B (Figure 3). Note that the Gorelians turn off their radios while traveling; they will not attempt to communicate with anyone else until they have all gathered at the meeting point. Once the Gorelians meet, they combine their radios to make a new radio with a larger range. In particular, the *area* covered by the new radio is equal to the sum of the *areas* covered by the old radio. In our example, Alice had a range of 40 km, so her radio covered an area of 1600π km. Bob's radio covered an area of 900π km. So when they combine their radios they can cover 2500π km—meaning they have a range of 50 km. At this point they will try again to contact other Gorelians.

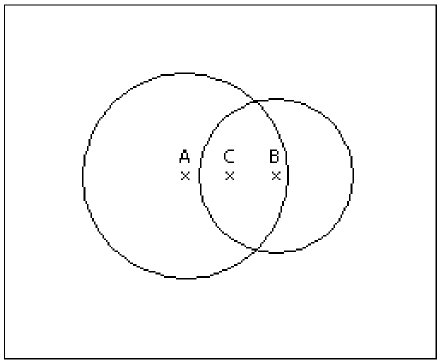


Figure 3: Alice and Bob agree to meet at the midpoint

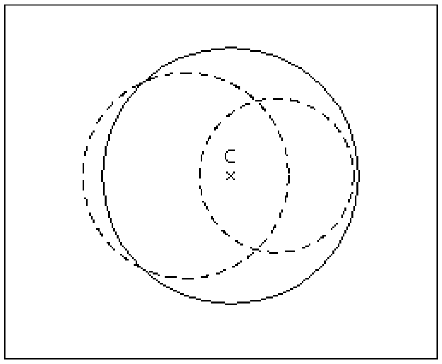


Figure 4: Alice and Bob combine their radios

This process continues until no more Gorelians can be contacted. As an example, suppose the following Gorelians have all landed and all have a radio range of 30 km: Alice (100, 100), Bob (130, 80), Cathy (80, 60), and Dave (120, 150). At this point, none of the Gorelians can contact anyone else (Figure 5). Now Eddy lands at position (90, 80) (Figure 6). Eddy can contact Alice and Cathy, so they arrange to meet at (90, 80), which is the average of their locations. Combining their radios gives them a range of √2700 ≈ 51.96 km (Figure 7).

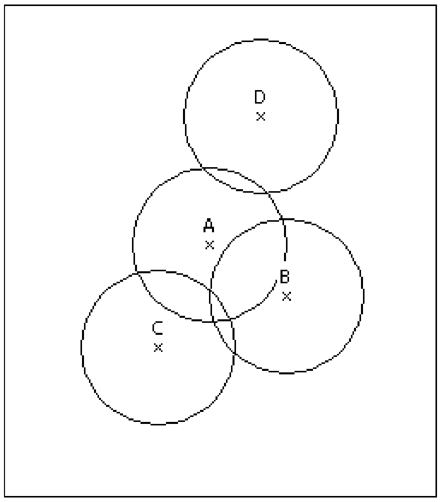


Figure 5: Nobody can talk

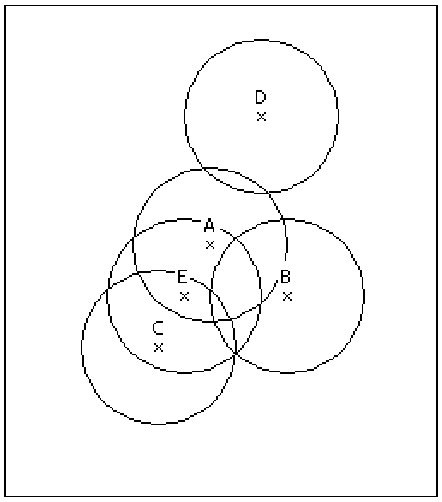


Figure 6: Eddy joins the group

Now they check again with their new improved range and find that they can reach Bob. So they meet Bob at (110, 80) and combine their radios to get a new radio with a range of 60 (Figure 8). Unfortunately, this is not far enough to be able to reach Dave, so Dave remains isolated.

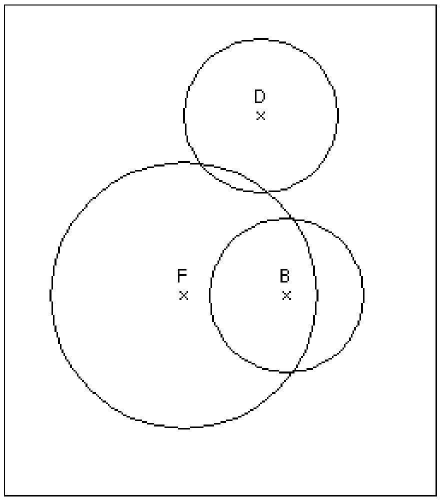


Figure 7: Alice, Cathy, and Eddy team up

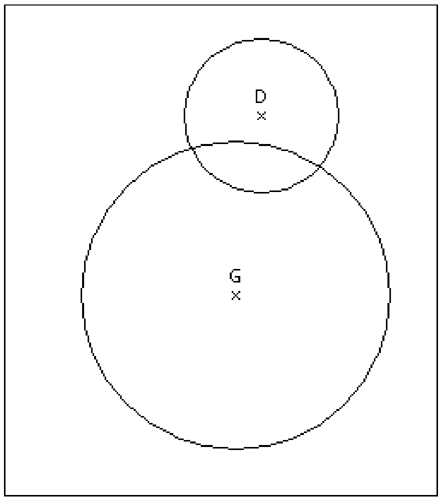


Figure 8: Bob joins the group

Input: The input will consist of one or more data sets. Each data set will begin with an integer *N* representing the number of Gorelians for this dataset ($1 \le N \le 100$). A value of $N = 0$ will signify the end of the input.

Next will come *N* lines each containing three integers *X*, *Y*, and *R* representing the x- and y-coordinate where the Gorelian lands and the range of the radio ($0 \le X \le 1000$, $0 \le Y \le 1000$, and $1 \le R \le 1000$). Note that only the Gorelians' *initial* coordinates/range will be integral; after merging with other Gorelians they may no longer be integral. *You should use double-precision arithmetic for all computations.*

The Gorelians land in the order in which they appear in the input file. When a Gorelian lands, he merges with any Gorelians he can contact, and the process keeps repeating until no further merges can be made. The next Gorelian does not land until all previous merges have been completed.

Output: The output will be one line per data set, reporting the number of independent groups of Gorelians that remain at the end of the process.

Example input:	**Example output:**

5
100 100 30
130 80 30
80 60 30
120 150 30
90 80 30
6
100 100 50
145 125 10
60 140 15
160 145 20
130 135 25
80 80 30
0
	2
	3

Problem D: The Bridges of San Mochti

Source file: `bridges.{c, cpp, java}`
Input file: `bridges.in`

You work at a military training facility in the jungles of San Motchi. One of the training exercises is to cross a series of rope bridges set high in the trees. Every bridge has a maximum capacity, which is the number of people that the bridge can support without breaking. The goal is to cross the bridges as quickly as possible, subject to the following tactical requirements:

One unit at a time!
If two or more people can cross a bridge at the same time (because they do not exceed the capacity), they do so as a unit; they walk as close together as possible, and they all take a step at the same time. It is never acceptable to have two different units on the same bridge at the same time, even if they don't exceed the capacity. Having multiple units on a bridge is not tactically sound, and multiple units can cause oscillations in the rope that slow everyone down. This rule applies even if a unit contains only a single person.

Keep moving!
When a bridge is free, as many people as possible begin to cross it as a unit. Note that this strategy doesn't always lead to an optimal overall crossing time (it may be faster for a group to wait for people behind them to catch up so that more people can cross at once). But it is not tactically sound for a group to wait, because the people they're waiting for might not make it, and then they've not only wasted time but endangered themselves as well.

Periodically the bridges are reconfigured to give the trainees a different challenge. Given a bridge configuration, your job is to calculate the minimum amount of time it would take a group of people to cross all the bridges subject to these requirements.

For example, suppose you have nine people who must cross two bridges: the first has capacity 3 and takes 10 seconds to cross; the second has capacity 4 and takes 60 seconds to cross. The initial state can be represented as (9 0 0), meaning that 9 people are waiting to cross the first bridge, no one is waiting to cross the second bridge, and no one has crossed the last bridge. At 10 seconds the state is (6 3 0). At 20 seconds the state is (3 3 /3:50/ 0), where /3:50/ means that a unit of three people is crossing the second bridge and has 50 seconds left. At 30 seconds the state is (0 6 /3:40/ 0); at 70 seconds it's (0 6 3); at 130 seconds it's (0 2 7); and at 190 seconds it's (0 0 9). Thus the total minimum time is 190 seconds.

Input: The input consists of one or more bridge configurations, followed by a line containing two zeros that signals the end of the input. Each bridge configuration begins with a line containing a negative integer –B and a positive integer P, where B is the number of bridges and P is the total number of people that must cross the bridges. Both B and P will be at most 20. (The reason for putting –B in the input file is to make the first line of a configuration stand out from the remaining lines.) Following are B lines, one for each bridge, listed in order from the first bridge that must be crossed to the last. Each bridge is defined by two positive integers C and T, where C is the capacity of the bridge (the maximum number of people the bridge can hold), and T is the time it takes to cross the bridge (in seconds). C will be at most 5, and T will be at most 100. Only one unit, of size at most C, can cross a bridge at a time; the time required is always T, regardless of the size of the unit (since they all move as one). The end of one bridge is always close to the beginning of the next, so the travel time between bridges is zero.

Output: For each bridge configuration, output one line containing the minimum amount of time it will take (in seconds) for all of the people to cross all of the bridges while meeting both tactical requirements.

Example input:	Example output:

```
-1 2
5 17
-1 8
3 25
-2 9
3 10
4 60
-3 10
2 10
3 30
2 15
-4 8
1 8
4 30
2 10
1 12
0 0
```
	```
	17
	75
	190
	145
	162
	```

Last modified on October 22, 2008 at 10:24 PM.