Sixteenth Annual
University of Central Florida

ACM · UPE

High School Programming Tournament

Problems

Problem Name	Filename
Secretly Lost	SECRET
Once More Around The Bends	DIVING
Parawhaling!	WHALE
Hawaiian Fire Drill!	SEATING
Birdman of Waikiki	BIRDMAN
On With The Countdown	PLAYLIST
Conflicting Conferences	SCHEDULE
Spider-Man's Diamond Head Dilemma	SPIDEY
Illegal Hawaiian Virus Collection	VIRUS
Meta-Game	GAME

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

For example, if you are solving Meta-Game:
Call your program file: game.pas, game.c, game.cpp or game.java
Call your input file: game.in

Secretly Lost

Filename: SECRET

Your friend has visited the Dole Pineapple Plantation in Hawaii where they have the world's largest maze. Unfortunately, your friend has become desperately lost within the maze and has sent a letter by carrier pigeon to you for help. Being incredibly embarrassed at getting lost, the letter was sent using a simple form of steganography, a cryptographic technique that hides a short, secret message in a long, innocuous transmission. You need to rescue your friend so you must decode the message quickly!

The Problem:

Given the letter as a series of sentences, decode the message from the letter. The letter is an ordinary paragraph of text, and the secret message is the n^{th} word of each sentence. If a sentence has fewer than *n* words, that sentence corresponds to a full stop in the secret message. Words are delimited by spaces or end of line.

For example, the letter might be:

> I write software help files here. It's fun! If you could come and visit some time, that would be great. You can help at proofreading. Say hello to Dawn for me. Later.

The hidden message for *n*=4 would be:

> help STOP come at Dawn STOP

The Input:

The first line of the input will contain a single positive integer, *m*, representing the number of messages to decode. Each message will start with a line that has a non-negative integer, *s*, that represents the number of sentences in that message; followed by a positive integer, *n*, that represents the steganographic index. On each of the next *s* lines will be a sentence of that message. Each sentence will have at most 255 characters.

The Output:

For each message, output the decoded message on a single line. A decoded message will have at most 255 characters.

Sample Input:

```
2
6 4
I write software help files here.
It's fun!
If you could come and visit some time, that would be great.
You can help at proofreading.
Say hello to Dawn for me.
Later.
4 2
We welcome you to UCF!
Come to the awards dinner!
Go UCF!
Yeah!
```

Sample Output:

```
help STOP come at Dawn STOP
welcome to UCF! STOP
```

Once More Around the Bends

Filename: `DIVING`

Dan and his new bride Della are spending their honeymoon in Hawaii. Della is an experienced diver, but Dan is just learning. He wants to impress his wife with his diving prowess, but first he must finish his classes and get his SCUBA certification. The final exam is a deep water dive, where he must descend to a certain depth, spend as much time as possible there, and then safely ascend to the surface, leaving no air in his tank when he is done. As you may know, if a diver ascends too quickly, he becomes afflicted with a condition known as the bends, where nitrogen gas accumulates at his joints. To avoid this, Dan must spend a certain amount of time decompressing at a lesser depth on his way up. Dan is lousy at math, so he needs your help.

The Problem:

Given a target depth that Dan must dive, implement a diving alarm that tells him when to start his ascent and decompression. For each 10 feet below 30 feet that Dan dives, he must spend one minute in decompression (this is simplified from real world diving tables, but it should work for this problem). Also, it takes Dan one minute to safely ascend or descend 10 feet. Dan's SCUBA school has supplied him with a 60-minute air tank. Given these constraints, compute how long Dan can stay at his target depth.

The Input:

Input will begin with a single, positive integer, *n*, indicating the number of dives that Dan must complete. On the next *n* lines will be the target depths of each dive, one positive integer per line. All depths will be multiples of ten.

The Output:

Print the amount of time (in minutes) that Dan can spend at his target depth and still safely make it back to the surface. If Dan can't make it to the target depth, then he can spend no time there. Follow the format of the Sample Output, leaving one blank line after the output for each dive.

Sample Input:

```
3
90
20
1000
```

Sample Output:

```
36 minute(s) at 90 feet

56 minute(s) at 20 feet

0 minute(s) at 1000 feet
```

Parawhaling!

Filename: `WHALE`

Ali loves to go parasailing. This year, he won a radio contest that sent him to Hawaii, where he is spending every day parasailing off the beaches of Waikiki. One day, he notices an advertisement for whale watching cruises. Ali gets the bright idea to use his parasailing skills to go whale watching more efficiently than anyone can by sitting on a dumb boat. His plan is to let the cruise go out in the morning and then ask one of the passengers where they saw whales. He can then visit all of those places while parasailing and get a much better view than the paying customers! Ali calls this new "sport" parawhaling.

The Problem:

Given a set of coordinates where the whale watchers saw whale activity, plan a route for Ali to follow on his parasailing trip. Being an extreme sports fanatic, Ali is impatient and doesn't care about the most efficient route, he just wants to go to the next closest whale watching point from his current location. Since he can see for long distances while parasailing, he only wants to get within three miles of the next point before moving on. Once Ali has visited a point, he doesn't visit the same point again on the same day.

The Input:

There will be multiple data sets. Input will begin with a single positive integer, *n*, on the first line, indicating how many days Ali will go parawhaling. Each day's data set will begin with a single positive integer, *m*, specifying the number of points where the whale watching patrons saw whales ($m < 20$). Each of the next *m* lines will contain a pair of integers specifying the *x*, *y* coordinates of a point of whale activity ($-100 < x,y < 100$). Coordinates are specified in miles. Ali always begins the day at the Waikiki Marina at coordinates 0, 0.

The Output:

For each day, print out a flight plan consisting of the whale watching points where Ali should fly in the order he should fly to them. The first point is the marina at 0,0. The second point should be the whale watching point closest to the marina. The third point should be the whale watching point closest to Ali's position while observing the second point, and so forth. Remember that Ali doesn't want to go all the way to each whale watching point, he only wants to get within three miles of them, but if the route to the next point takes him closer to the point he is currently visiting, he doesn't mind getting closer to the current point. There will only be one correct route each day, so once Ali has arrived at any point, he'll never have to decide between two or more points the same distance away. Print a header for each day in the format of "`Day #x:`" where `x` represents the number of the data set, starting with 1. Following this header, print each point along the route on its own line in the correct order. Print each coordinate of each point right-justified in a three-column field with a comma between the two coordinates. Leave one blank line after the output for each day. Refer to the Sample Output for examples.

Sample Input:

```
2
4
0 10
0 20
10 10
10 20
5
-20 10
-10 10
0 10
11 10
20 10
```

Sample Output:

```
Day #1:
  0,  0
  0, 10
 10, 10
 10, 20
  0, 20

Day #2:
  0,  0
  0, 10
-10, 10
-20, 10
 11, 10
 20, 10
```

Hawaiian Fire Drill!

Filename: `SEATING`

Ryan climbs back into the van and sits in the only seat left. "Hey, this isn't where I sat on the way here," he thinks. Then he realizes that everyone in the van is sitting in a different seat, and wonders how many different arrangements of people there can be.

The Problem:

The rental agreement says that only certain people are allowed to drive the van. Your job is to find the number of different seating arrangements there are, given this restriction.

The Input:

Input begins with an integer n, ($0 \le n \le 500$), the number of data sets in the file, on a line by itself. Each set consists of two positive integers on a line, separated by spaces. The first integer is the total number of people on the trip, and the second is how many are allowed to drive. There are exactly as many seats in the van as there are people (Spooky, huh?). There will never be more than 12 people on the trip.

The Output:

For each set, output the following phrase on its own line:

`Data set #`*`x`*`:` *`y`* `combination(s) possible.`

x is the number of the current data set, starting with 1, and *y* is the number of seating combinations that are possible and legal.

Sample Input:

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

Sample Output:

```
Data set #1: 12 combination(s) possible.
Data set #2: 48 combination(s) possible.
Data set #3: 2 combination(s) possible.
```

Birdman of Waikiki

Filename: `BIRDMAN`

On Waikiki Beach in Hawaii, there is a fellow known as the "Birdman of Waikiki." He collects money for various charities by taking photographs of you posing with his birds and then asking for donations. A popular time for this is at sunset (Hawaii has some spectacular sunsets). Unfortunately, the Birdman of Waikiki makes the amateur mistake of framing the shot with you in the foreground blocking the sunset in the background. He needs a program to help him.

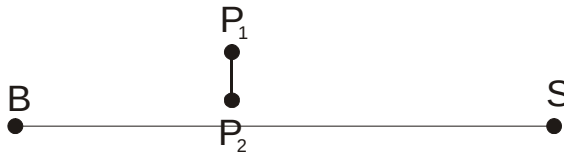
The Problem:

Given the location of the sunset, the location of the Birdman of Waikiki, and a description of you, determine whether the Birdman of Waikiki is showing the sunset in the picture or not. The sunset and the Birdman of Waikiki will each be represented as 2-D points. A line segment will represent you (you're awfully thin, after all) and will be specified by two 2-D points. The Birdman of Waikiki is properly showing the sunset if you do not obstruct the path from him to the sunset in any way (in other words, no point along the line segment describing you inclusive of the end points blocks the segment from the Birdman of Waikiki to the sun).

The Input:

The first line will contain a single integer, *n*, representing the number of photographs to check. For each *n*, there will be a line with four integer pairs. The first integer pair will be a point, *B*, and is the (x,y) location of the Birdman of Waikiki. The second integer pair will be a point, *S*, and is the (x,y) location of the sunset. The third and fourth integer pairs, points P_1 and P_2, will be the (x,y) locations of the ends of the segment that describe you.

Birdman of Waikiki did *not* show sunset properly.



Birdman of Waikiki did show sunset properly.

The Output:

For each photograph, output on a new line either "`Good picture, Birdman!`" if the sunset can be seen or "`Move to the left or right!`" if it cannot.

Sample Input:

```
2
5 0 5 100 3 2 7 2
1 2 5 1 5 2 5 4
```

Sample Output:

```
Move to the left or right!
Good picture, Birdman!
```

On With The Countdown

Filename: `PLAYLIST`

John, Jim, Mike, Scott, Tom, Bob, Jane, Al, Sue, and Mary are driving around together in Hawaii and decide to play some music. The problem is that they all like different types of music, and can never agree on what to play.

The Problem:

Your job is, given a list of songs and the number of people who like them, output the playlist with the most-voted-for songs, in descending order.

The Input:

The first line in the file will contain a non-negative integer representing the number of test cases in the file. Each test case is formatted as follows:

The first line of each test case will contain only the target playlist length, *n*. The second will contain the number of available songs, *m*. The following *m* lines will begin with the name of a song. This will be followed by exactly one space, and then a positive integer indicating the number of people voting for that song. Your program should output the *n* songs with the most votes, in descending order by number of votes.

n and *m* will satisfy $0 < n \leq m < 100$. Each song will be represented by one word (sequence of case-sensitive alphanumeric characters plus underscore, with length $1 \leq l \leq 200$). The songs in the list will be unique, and there will be no ties for votes within the top *n* songs.

The Output:

Your output should consist, for the *i*th input case starting at 1, of a line containing "`Case #i:`", followed by the *n* songs on the playlist in order of descending votes, each on a line by itself.

Sample Input:

```
2
2
4
Nirvana_Lithium 3
System_of_a_Down_Needles 4
Bach_5th_Symphony 2
Sheryl_Crow_Abilene 1
1
1
Papa_Roach 60
```

Sample Output:

```
Case #1:
System_of_a_Down_Needles
Nirvana_Lithium

Case #2:
Papa_Roach
```

Conflicting Conferences

Filename: SCHEDULE

A large Hawaiian hotel rents out its conference room for events. It needs to avoid renting the room out to more than one group at a time. However, it wants to the room to be used for as long as possible each day. To prevent confusion, the hotel operates in twenty-four hour time (ie, 10:00 PM is written as 22:00).

The Problem:

Your job is to write a program that checks whether a certain room has overlapping events, and if so, to list which times more than one event is scheduled. The rooms are rented out by the hour, so all times are specified as start hour to end hour. Events may be scheduled back-to-back (for example, an event may start at 11:00 and end at 12:00, while the next event starts at 12:00 and ends at 13:00). The conference room is only available for rent from 8:00 to 22:00, since the hotel does cleaning and other maintenance at night.

The Input:

The first line contains a positive integer, denoting the number of rooms. The information for each room is formatted as follows:

The first line of information for each room contains a string that represents the name of the room and the date of the schedule, and should be printed as a header in the output. This string will be at least 1 and less than 100 characters. The line following that contains the number of events scheduled, n ($1 \leq n \leq 100$). The following n lines each represent one event. Each line will contain two integers, s and e, ($8 \leq s < e \leq 22$), representing the start and end times of that event.

The Output:

For each room, print two lines of information. The first line should be the header specified in the input. The second line will be either

```
SCHEDULE OK!
```

if there are no conflicts, or

```
CONFLICTS AT <START1>-<END1> … <STARTN>-<ENDN>
```

if there are. Each <START> and <END> specifier should be exactly one hour apart. If multiple hours overlap, list each individual hour. For example, if during the hours from 12:00 to 17:00, there are overlapping events, output:

```
CONFLICTS AT 12-13 13-14 14-15 15-16 16-17
```

Leave a blank line after each room.

Sample Input:

```
3
Coral Ballroom 5/3/2002
3
8 12
11 17
16 20
Kamehameha Conference Room  3 May 2002
1
8 21
Aloha Lounge May 3, 2002
2
11 15
15 22
```

Sample Output:

```
Coral Ballroom 5/3/2002
CONFLICTS AT 11-12 16-17

Kamehameha Conference Room  3 May 2002
SCHEDULE OK!

Aloha Lounge May 3, 2002
SCHEDULE OK!
```

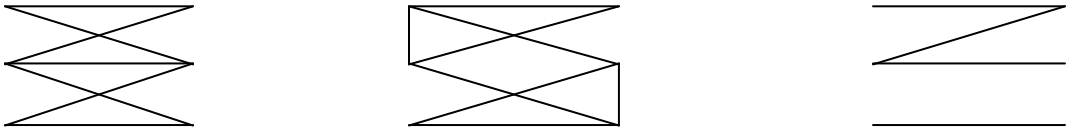
Spider-Man's Diamond Head Dilemma

Filename: `SPIDEY`

Spider-Man has received an ultimatum from the fiendish Green Goblin, stating that he will soon destroy the world, unless he is given the ridiculous ransom of one hundred billion dollars (in tens and twenties). Through some clever detective work, Spider-Man discovers that the Goblin is planning on dropping a huge bomb right over the famous Diamond Head volcanic crater near Honolulu, Hawaii. The bomb will not only destroy Hawaii and everyone living and visiting there, but it will cause a tectonic disturbance of such magnitude that the entire Pacific Ocean floor will erupt with fiery lava! Spidey has a plan to catch the big bomb by placing a web over the top of Diamond Head, but he needs your help to plan where to attach the web to the crater walls.

The Problem:

Given a set of points indicating where the web is attached to the crater walls, and the actual web connections between them, determine whether or not the web will safely catch and hold the bomb. The points are numbered 0 through *n*. Even numbers are attachments on the west wall of the crater, and odd numbers are on the east wall. For the web to be able to safely hold the bomb, there must be at least one web fiber connection from each point on the east wall to some point on the west wall, and vice versa. Furthermore, no point on the east wall can be directly connected to another point on the east wall, and no point on the west wall can be directly connected to another point on the west wall. Spider-Man has told you that doing so compromises the structural integrity of the web (hey, he's the expert, right?). Finally, once the web is finished, it must be all one piece. That is, Spider-Man must be able to walk from any attachment point to any other by walking across web fibers. See the illustrations below. In the illustrations, the left side of each figure represents the west wall, and the right side is the east wall.



The web on the left will safely catch the bomb. The middle web will not, because it has direct connections between two points on the same side. The web on the right will also not catch the bomb, because it is not one piece.

Note that web connections are not directed, so a connection from point 0 to point 3 is the same as a connection from point 3 to point 0.

The Input:

There will be multiple data sets. Input will begin with a single, positive integer, *m*, on a line by itself, representing the number of web configurations Spidey needs tested. This will be followed by *m* web descriptions. Each web begins with a line containing two integers, *v* and *e*, ($0<v<16$; $0<e<120$). *v* represents the number of attachment points, and *e* represents the number of web connections between the points. This line will be followed by *e* lines of 2 integers each, *a* and *b* ($0 \leq a,b<v$). Each pair of integers represents a web connection from point *a* to point *b*.

The Output:

For each web, print "`Way to go, Spider-Man!`" if the web will hold. If the web will not hold, print "`It's the end of the world!`" Leave one blank line after the output for each web.

Sample Input:

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

Sample Output:

```
Way to go, Spider-Man!

It's the end of the world!
```

Illegal Hawaiian Virus Collection

Filename: `VIRUS`

Victor likes to collect viruses (yes, this is a strange hobby). He has all sorts of harmless and harmful ones that he keeps with him at all times (sealed up very securely of course). He is very attached to his virus collection and he does not like to travel without it. So when he made plans to go to Honolulu, Hawaii, his virus collection was sure to go with him. Unfortunately, Hawaii (unlike other states) requires you to declare any viruses you may have in your possession. Because Hawaii prohibits certain viruses from entering the state (they don't want their fruit crops destroyed by a rogue virus!), Victor is sure to leave any prohibited viruses at home so that they are not seized and destroyed. Fortunately for Victor, the Hawaiian government has provided an informative website to help Victor determine if his viruses are okay to enter Hawaii or not.

The Problem:

Your job is to help Victor determine which viruses he can take with him to Hawaii. The Hawaiian government has provided on their website the exact sequences of DNA that are in the viruses that are prohibited. So if the Hawaiian government said that the following sequence was prohibited:

```
ATTCCGTA
```

Then the following virus would be rejected because it contains that sequence:

```
ATTATTAGGATTACATTCCGTAACCGTTTTAG
```

However, the sequences may not have to be consecutive. They may have a fixed number of "wildcards" (represented by the ASCII star character "*") meaning that any DNA letter in a DNA sequence will replace that wild card (but only one DNA letter per wildcard). For example, the following sequence:

```
ATA**CGGC*A
```

Would reject the following virus DNA:

```
AGGAAATTACCCCATAGGCGGCTA
```

The Input:

You will read in a file that contains both the list of bad virus DNA sequences and then the DNA sequences of Victor's collection.

The first line of the file will contain a single positive integer that represents the number of data sets in the file (unfortunately for Victor, the Hawaiian government updates their website several times a day with new and/or different virus DNA so your program must be able to handle multiple, independent data sets).

The first line of each data set will contain a single positive integer, n, ($1 \leq n \leq 30$) indicating how many bad DNA subsequences were found on the Hawaii government's website. The following n lines will contain strings of DNA characters that will be at least 1 character long and at most 20 characters in length.

Following the bad DNA sequences will be a line containing a single positive integer, m, ($1 \leq m \leq 100$) which will indicate how many viruses are in Victor's collection. The following m lines will contain the DNA for his viruses. Each line will contain a string of DNA characters that will be at least 1 character long and at most 200 characters in length.

Notes to remember:

- The DNA alphabet only consists of the letters: A, C, G, and T.
- All letters will be uppercase.
- A "*" in the bad DNA sequences indicates a wildcard which must be matched by any one DNA letter in Victor's virus DNA. There will be no "*"s in the DNA sequences of Victor's viruses.
- Bad virus DNA in one data set is **NOT** bad virus DNA in another data set.

The Output:

At the beginning of each data set, print the line:

```
Data set #k:
```

where *k* is the data set number starting with 1.

For each virus that you check for bad DNA sequences, print the following output if the virus is acceptable in Hawaii:

```
Virus #d: Cool! Victor can take it with him!
```

Or print the following output if the virus is not acceptable in Hawaii (i.e. it matched one of the bad DNA sequences):

```
Virus #d: Nuts. This virus is illegal in Hawaii!
```

Where *d* should be the virus number, starting with 1, for each data set. Other than replacing *d* and *k* with the numbers as specified, the lines should be printed exactly as shown above (watch your uppercase and lowercase letters as well as punctuation!). The output for each virus should be on its own line (see the Sample Output below for an example).

Put a blank line in between data sets as a separator.

Sample Input:

```
2
3
ATTGG***A
**GTTGACCCCCC
GAGAGAGAGA
5
GATTGATTGAGAGAGAGATCATC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
GGGGGGGAAGTTGACCCCCC
AATTGGCATACCCCTTT
GGGGAAAATTTTAAAACCCCGGGGGAAAA
2
ATGG
ATGCATGCATGC
2
GCCGCCGCCTGCCGCCGCCT
ATATATGGATATATGGATATATGG
```

Sample Output:

```
Data set #1:
Virus #1: Nuts. This virus is illegal in Hawaii!
Virus #2: Cool! Victor can take it with him!
Virus #3: Nuts. This virus is illegal in Hawaii!
Virus #4: Nuts. This virus is illegal in Hawaii!
Virus #5: Cool! Victor can take it with him!

Data set #2:
Virus #1: Cool! Victor can take it with him!
Virus #2: Nuts. This virus is illegal in Hawaii!
```

Meta-Game

Filename: GAME

The UCF programming team didn't think there would be enough to do in Hawaii, so they brought along a bunch of board games. However, being a programming team, instead of actually *playing* the board games, they used them to develop an abstract boardgame model and generate a few simulated boards and games based on this model. What you are to do is simulate a game played on a simplified version of this model.

The Problem:

It works like this: the game board is a cyclic board separated into 'squares' where one can place any number of game pieces, each square is marked by one of a number of 'modifiers' that control what happens a player leaves that square. Each turn, a player rolls a standard six-sided die, and based on the number on the die and the 'modifier' of the square his or her piece is on, s/he moves forward or backward a certain number of squares. Your program will process a representation of a game board, followed by the die roll sequence for several players, and output the ending square for each player. (All players start on square zero.)

The 'modifiers' are as follows:

>	NORMAL. The player moves forward the number rolled.
<	REVERSE. The player moves backward the number rolled.
)	SLOW: The player moves forward half the number rolled, rounded down.
(	SLOW REVERSE: The player moves backward half the number rolled, rounded down.
!	FAST: The player moves forward twice the number rolled.
1-9	BONUS: The player moves forward that many extra spaces, in addition to his or her die roll.
_	QUAGMIRE: The player moves forward one space on an even roll, and remains in place on an odd roll. (This is the underscore character ('_'), not a hyphen ('-')).
|	WALL: The player moves backward one space regardless of the die rolled; the roll still occurs and this still takes exactly one turn.

Here is a sample board, with the positions listed:

```
>|!(_
01234
```

Here is the move sequence for the first sample input case below: The first player rolls a 1 and hits the wall at position 1. Then he rolls a two, but has to ignore it and move back one because of the wall. Then he rolls *another* one, ending him up at the wall again. The second player rolls a 2, hopping over the wall and landing on the "fast" modifier. Then he rolls a 6...this brings him all the way around the board twice, then forward two spaces, landing him in the quagmire. His next two rolls are both odd, so he is stuck there at the end. The final player rolls a 3, putting him over the "slow reverse" square. Then he rolls a one; since one-half rounded down is zero, he stays on that square. Then he rolls a 5, which moves him back 2 spaces onto the 'wall' square. His 6 is ignored and he moves back to position 0, and then rolls a 5, which moves him around the board, leaving him at zero again.

The Input:

The first line of the input will be the number of data sets. Immediately following are the input cases, each specified as follows:

The first line in each set will contain 2 integers separated by exactly one space, indicating the size of the board, *b*, in squares and the number of players, *p*. *b* will be in the range $0 < b < 200$, and *p* will be in the range $0 \leq p < 20$. The next line contains the board, which will be a string *b* characters long, each of which will be one of the modifiers above. After that will be *p* lines, each representing a player. Each such line will contain the number of turns that player takes, followed by a space, followed by a single string consisting of that number of die rolls in order. The number of turns will be between 1 and 200, inclusive.

The Output:

The output should, for the *i*th data set, begin with a single line consisting of "`Board #i:`". The next *p* lines should be of the format "`Player #j: Ended at position l`"; *l* should be the position where player number *j* ends up after all listed rolls. Output a blank line after the output for each data set.

Sample Input:

```
2
5 3
>|!(_
3 121
4 2613
5 31565
2 3
><
3 111
6 222666
4 6143
```

Sample Output:

```
Board #1:
Player #1: Ended at position 1
Player #2: Ended at position 4
Player #3: Ended at position 0

Board #2:
Player #1: Ended at position 1
Player #2: Ended at position 0
Player #3: Ended at position 0
```