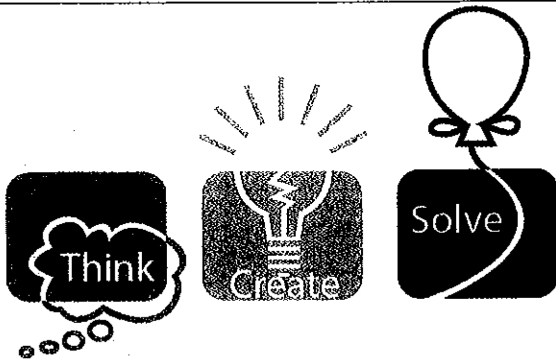
University of Central Florida



2012 (Fall) "Practice" Local Programming Contest

Problems

Problem#	Filename	Problem Name
1	wallst	Wall Street Monopoly
2	stringage	How Old Are You Mr. String?
3	power	Clean Up the Powers that Be
4	pagealloc	The Clock Algorithm
5	pantry	Pete's Pantry
6	rigid	Metallic Equipment Rigid
7	lifeform	Lifeform Detector
8	order	Ordered Numbers

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

For example, if you are solving Wall Street Monopoly:

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

UCF "Practice" Local Contest — Aug 25, 2012

Wall Street Monopoly

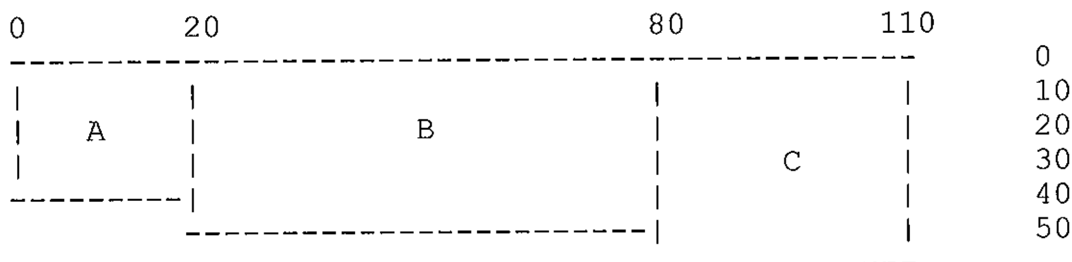
filename: `wallst`

Arup enjoys hanging out in downtown Orlando and realizes that it would be much more fun if he were making money instead of spending money. He gets the brilliant idea of trying to buy separate pieces of adjacent property. Ultimately, he wants to fuse all of these adjacent spaces into one so that he can own one mega-club that absolutely everyone wants to go to. Unfortunately, he finds out that the city charges him money every single time he "fuses" adjacent lots. It turns out that the way in which they charge him depends on the order in which he fuses his lots together. In particular, if lots A, B and C are contiguous but separate initially, the cost of fusing A and B, followed by fusing the lot AB with the lot C, could be different than the cost of fusing B and C, followed by fusing the lot A with the lot BC (note that fusing A and C first is not an option since A and C are not adjacent). Given the method in which Arup is charged for fusing lots (which is based on the dimensions of each lot), as well as the dimensions of each of his lots, you are to determine his minimum cost in fusing the lots.

The Problem:

We will refer to a lot's size as *length* x *depth*, i.e., the first dimension represents the length of the lot on the street and the second dimension represents the depth of the lot.

Given two lots, the cost of fusing those two lots is 100 dollars times the product of the minimum (smaller) dimensions of each lot (in feet). Thus, if one lot is 20' x 30' and its adjacent lot is '60 x 40', then the cost of fusing the two lots is \$100x20x40 = \$80,000 (20 is the smaller dimension of the first lot and 40 is the smaller dimension of the second lot). Also, the size of the newly fused lot will be the sum of the length of each lot by the maximum (larger) of the depth of both lots. In this case, the fused lot would be 80' x 40' (80 feet long and 40 feet deep; 80 is the sum of the lengths and 40 is the larger of the depths). Three adjacent lots with the following dimensions: A - 20'x30', B - 60'x40', and C - 30'x50', are shown below:



Given the dimensions (lengths and depths) of several adjacent lots, determine the minimum cost of fusing the separate lots into one.

The Input:

The first line of the input consists of a single positive integer, *n*, representing the number of test cases in the input file (i.e., the number of data sets to be processed). The following *n* lines contain the input cases, one per line. For each input test case, the first integer, k $(0 < k < 20)$, represents the number of lots to be fused. This is followed by *k* pairs of positive integers less than 101 representing the dimensions (length and depth) of each of the lots in feet, in the order in which they are located.

The Output:

For each input case, output a single line with the following format:

```
The minimum cost for lot #m is $X.
```

where m $(1 \leq m \leq n)$ is the input case number (starting with 1) and *X* is the integer dollar amount corresponding to the minimum cost to fuse all of the separate lots for that particular input case.

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

Sample Input:

```
2
3 20 30 60 40 30 50
2 10 90 30 40
```

Sample Output:

```
The minimum cost for lot #1 is $200000.

The minimum cost for lot #2 is $30000.
```

UCF "Practice" Local Contest — Aug 25, 2012

How Old Are You Mr. String?

filename: `stringage`

Given two strings, each consisting of only the lower-case letters, you are to compare their age. $String_1$ is considered older than $string_2$ if $string_1$ has more occurrences of the letter z than $string_2$ does. If both strings have the same number of z's, $string_1$ is older if it has more y's. If they have the same number of y's, then number of x's determine the older string, etc. If the two strings have the same number of z's, the same number of y's, …, the same number of a's, then the two strings are considered to be the same age.

The Input:

The first input line contains a positive integer, *n*, indicating the number of data sets (how many pairs of strings are to be compared). This is followed by *2n* input lines, each data set consisting of two lines (strings). Assume that each string is at least one and at most 70 letters, starts in column one and contains no other characters.

The Output:

For each data set, print the heading "`Data set #`*i*`:` " where *i* is the number for the set (starting with 1). Then, print one of the following three messages:

```
First string is older
First string is younger
The two strings are the same age
```

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

(Sample Input/Output on the next page)

Sample Input:

```
3
yzzz
abcxyz
ay
xy
aliorooji
oroojiali
```

Sample Output:

```
Data set #1: First string is older

Data set #2: First string is younger

Data set #3: The two strings are the same age
```

UCF "Practice" Local Contest — Aug 25, 2012

Clean Up the Powers that Be

filename: `power`

Dr. Orooji, a number theoretician on the side, has written an amazing program that prime factorizes numbers. He is so ecstatic that his program works, that he has forgotten to properly format (clean up) the output of his program. Luckily, Dr. O has hired you as a student assistant, to do his grunt work, formatting the output of his ingenius program. Your goal is to take the output of Dr. O's program, and nicely format it for everyone else.

The Problem:

Dr. O's output consists of several cases (data sets) which are the input to your program. Each input case (to your program) contains several pairs of numbers, each pair being a base and an exponent. Although each of the bases are prime numbers already, Dr. O hasn't bothered to list them in numerical order. Furthermore, sometimes he's listed the same base several times – thus a 2^3 might be listed as well as a 2^5. Clearly these should be coalesced into one term, 2^8. Finally, Dr. O's output does not put exponents in a "superscript" position. Your job will be to fix all three of these issues with each input case given.

The Input:

Input starts with a positive integer, *n*, on the first line by itself, indicating the total number of test cases in the input file (i.e., the number of data sets to be processed). The next *n* lines contain the test cases, one per line. Each test case will contain several positive integers separated by spaces on a single line. The first positive integer on each line, k $(0 < k < 20)$, will represent the number of terms in the prime factorization for that case. This will be followed by *k* pairs of positive integers, all separated by spaces. The first integer of each pair will be a positive prime number less than 10,000 (representing a base). The second integer of each pair will be a positive integer less than 100,000, representing the exponent to which the corresponding prime number is raised in the prime factorization given.

The Output:

For each input case, first output the following header line:

`Prime Factorization #`*`m`*`:`

where m $(1 \leq m \leq n)$ represents the input case number (starting with 1).

The rest of the output for each case will be on the following two lines. In particular, each base will be on the second line and each exponent will be on the first line. The bases will be in ascending numerical order. The corresponding exponents will start in the following column (after its base ends) on the first row. Each following base will start in the next column after the previous exponent ends.

For example, the number $2^5 3^{13} 101^{17}$ would be represented as follows:

```
 5 13    17
2 3   101
```

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

Sample Input:

```
3
3 2 5 101 17 3 13
4 5 2 5 19 3 5 11 1
5 2 10 3 11 2 12 3 13 2 14
```

Sample Output:

```
Prime Factorization #1:
 5 13    17
2 3   101

Prime Factorization #2:
 5 21   1
3 5   11

Prime Factorization #3:
 36 24
2   3
```

UCF "Practice" Local Contest — Aug 25, 2012

The Clock Algorithm

filename: `pagealloc`

In operating systems design, there is a technique called *virtual memory*, which enables the computer to run a program whose required memory (or logical memory) is larger than the available physical memory of the computer. The memory (for computer and program) are divided into pages, each consisting of a block of fixed size. When a program requests to access a page that is not yet loaded into memory, a *page fault* occurs. *Thus, a page fault will occur the first time each page is accessed.* In addition, if the program needs to access more pages than the available memory can provide, one of the pages that were loaded previously will have to be swapped out (written onto disk) in order to provide free memory space for the requested page. The algorithm that decides which page to swap out is called the page replacement strategy. One of the widely-known replacement algorithm is called the least-recently-used (LRU) strategy.

The Problem:

In this problem, we'll explore a variation of LRU known as the clock algorithm. This algorithm works as follows. Initially, all n page cells in memory are free; you can think of them as an array with indices $1 \ldots n$. Each element of the array contains the amount of memory for one page, and a "*flag*". The clock algorithm also keeps a "*hand pointer*", which initially points to cell 1.

When the program needs to access a page, first it checks to see if the page is currently loaded in memory; if so, it simply accesses that page (no page fault) and sets the page's flag to *new*.

If the page it needs is not currently loaded in memory, then it loads the page in the next available free cell (cell with the lowest index) if there is one available and sets the page's flag to *new*. If there is no free cell, then it checks the cell pointed by *hand pointer*. If the page pointed by hand is marked as *old*, then it will be swapped out for the new request (i.e., the page is loaded and marked as *new*) and the pointer advances one position. If the page pointed by hand is marked as *new* (i.e., the page is not *old*), the algorithm then marks that cell/page as *old* and the pointer advances one position. Advancing the pointer simply means the pointer will point to the next memory cell (with 1 higher index), and wraps back to the first cell if advancing from the nth cell, similar to the hands of a clock. Eventually, some page pointed by the hand will be seen as *old* and is swapped out, giving a free cell for the requested page.

Note that when a page is referenced or loaded, its flag is set to *new*.

This strategy gives each cell a second chance, being set to *old* before getting swapped out. If a page is referenced while the cell/page is *old*, it will be marked as *new* again, indicating that it is recently used. Thus a page will only be swapped out if the hand pointer sees it twice (first time marks it as *old*, and second time swaps it out) without the page ever being used during that period.

Your task is to implement the clock algorithm.

The Input:

There will be multiple test cases. The first line of each test case contains two integers, n ($1 \leq n \leq 50$), the number of page cells in available memory, and r ($1 \leq r \leq 50$), the number of page requests made by the program. The next line contains r integers r_i ($1 \leq r_i \leq 50$), separated by spaces. These are pages that are accessed by the program (the pages are listed in the order of being accessed by the program).

The last test case will be followed by a line which contains "`0 0`", i.e., end-of-data is indicated by $n = 0$ and $r = 0$.

The Output:

At the beginning of each test case, output "`Program` *p*", where p is the test case number (starting from 1). For each request, output: "`Page` r_i `loaded into cell` *c*." if the request r_i is not in memory, and is loaded into cell c. If the request r_i is already in memory, output: "`Access page` r_i `in cell` *c*." where c is the cell that page r_i is located in memory. At the end of each test case, output: "`There are a total of` *k* `page faults.`" where k is the total number of page faults (loads) experienced during the execution of the program.

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

Notes:

- The size of each page is not important, as the input is already given in units of pages.
- The logical memory requirement for a program is not specified, but you may assume that all programs have a size of 50 pages (as bounded by page request, r_i).

Sample Input:

```
3 12
3 2 1 5 3 2 4 3 2 1 5 4
5 16
1 3 5 7 9 9 7 5 1 8 3 5 2 3 12 18
8 1
1
0 0
```

(Sample Output on the next page)

Sample Output:

```
Program 1
Page 3 loaded into cell 1.
Page 2 loaded into cell 2.
Page 1 loaded into cell 3.
Page 5 loaded into cell 1.
Page 3 loaded into cell 2.
Page 2 loaded into cell 3.
Page 4 loaded into cell 1.
Access page 3 in cell 2.
Access page 2 in cell 3.
Page 1 loaded into cell 2.
Page 5 loaded into cell 3.
Access page 4 in cell 1.
There are a total of 9 page faults.

Program 2
Page 1 loaded into cell 1.
Page 3 loaded into cell 2.
Page 5 loaded into cell 3.
Page 7 loaded into cell 4.
Page 9 loaded into cell 5.
Access page 9 in cell 5.
Access page 7 in cell 4.
Access page 5 in cell 3.
Access page 1 in cell 1.
Page 8 loaded into cell 1.
Access page 3 in cell 2.
Access page 5 in cell 3.
Page 2 loaded into cell 4.
Access page 3 in cell 2.
Page 12 loaded into cell 5.
Page 18 loaded into cell 3.
There are a total of 9 page faults.

Program 3
Page 1 loaded into cell 1.
There are a total of 1 page faults.
```

UCF "Practice" Local Contest — Aug 25, 2012

Pete's Pantry

filename: `pantry`

Pete isn't feeling too well today. In an effort to try and get better, he wants to heat up a can of chicken soup. However, his pantry is a mess, and as soon as he grabs the can he wants, the stacks of cans all collapse and roll onto the kitchen floor. "If I only had a way to stack these cans to keep this from happening," Pete thought.

The Problem:

Your job is to write a program to help Pete stack cans in his pantry. Cans come in all shapes and sizes, but for simplicity, we'll assume that all cans are the same height, and only their width (diameter) will vary. The stacking must follow one simple rule:

No can shall be stacked atop a narrower can, or a can of equal width.

Pete doesn't follow written directions very well, so he wants your program to show him how to stack his cans. For example, a can with the label "BEANS" should look like this:

```
#######
#     #
#     #
#BEANS#
#     #
#     #
#     #
#######
```

The width of each can is determined by the can's label. The label can have multiple lines, so the width of the can is simply the width of the longest line of the can's label, plus two (for # delimeters) . All cans are 8 units tall, counting the edges, so no can label will have more than six lines. Pete also wants a particular order to his cans, so your program should process the input cans in the order given. Try fitting a given can in the leftmost stack first, and work to the right if needed. If there is no existing stack that will hold a given can, start a new stack just to the right of the last existing stack (the first can should be placed in the leftmost spot on the shelf).

The Input:

Pete will have multiple sets of cans for you to stack. The first line of input will contain a positive integer, n, indicating the number of sets of cans (i.e., the number of data sets to be processed). Each set of cans will begin with a positive integer c ($1 \le c \le 100$), on a line by itself. On each of the next c lines will be a can label. Can labels consist of upper- and lower-case letters, digits, and spaces. The pound sign ('#') may also appear in a can label and designates the start of a new line (i.e., the can label must be put on multiple lines in output). There will be no leading or

trailing spaces on any line of the can label (in input) and the input lines will not exceed column 60. Assume that there is at least one letter (or digit) before and after each '#'.

The Output:

For each set of cans print "`Can Stack` #*d*:", where *d* is the number of the set (starting with 1). Then, print two header lines to show column numbers. Then, print the stacks of cans using the format described above. Can labels must be centered both vertically and horizontally on the can (if an exact centering isn't possible, favor the left half and/or top half of the can). Each stacked can should be centered above the can below it (again, favor the left side if an exact centering is impossible). There should be exactly one space between the bottommost cans in each stack. Assume that no final arrangement will be more than 60 characters wide, and no stack will be taller than 10 cans.

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

Sample Input:

```
2
2
Coaches8#Ali88
Geeks of#UCFdorms#Soup
6
Pork and#Beans
Cream of#Mushroom#Soup
Baby#Carrots
Snow#Peas
Grape#Jelly
Aunt Helens#Down Home#Country#Goodness
```

(Sample Output on the next page)

Sample Output:

```
Can Stack #1:
         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890
########## ##########
#        # #        #
#        # #Geeks of#
#Coaches8# #UCFdorms#
# Ali88  # #  Soup  #
#        # #        #
#        # #        #
########## ##########

Can Stack #2:
         1         2         3         4         5         6
123456789012345678901234567890123456789012345678901234567890
 ######
 #    #
 #    #
 #Snow#
 #Peas#
 #    #
 #    #
 ######
#########   #######
#       #   #     #
#       #   #     #
# Baby  #   #Grape#
#Carrots#   #Jelly#
#       #   #     #
#       #   #     #
#########   #######
########## ########## #############
#        # #        # #           #
#        # #Cream of# #Aunt Helens#
#Pork and# #Mushroom# # Down Home #
# Beans  # #  Soup  # #  Country  #
#        # #        # # Goodness  #
#        # #        # #           #
########## ########## #############
```

UCF "Practice" Local Contest — Aug 25, 2012

Metallic Equipment Rigid

filename: `rigid`

Rigid Reptile is trying to covertly infiltrate the compound of his nemesis Pistol Wildcat. The compound is under heavy surveillance by large numbers of video cameras and genetically enhanced soldiers. Luckily for Reptile, the soldiers' enhancements grant them eyesight, hearing, and intelligence that is far inferior to that of an ordinary person. Between his cardboard box and tricks he picked up from Decoy Calamari, he'll have little trouble evading the soldiers. The security cameras, though, are a different story.

The Problem:

The cameras in Pistol Wildcat's compound have a circular detection radius which varies from camera to camera. Any unauthorized person (such as Rigid Reptile) who gets closer ($\leq$) to a camera than the specified radius will immediately trigger an alarm. Ordinarily, the genetically enhanced soldiers think nothing of a person-sized cardboard box walking around, but when an alarm goes off, they are apt to shoot anything and everything. Naturally, Reptile would like to avoid this. You must write a program that, given Rigid Reptile's path as a series of two-dimensional line segments and the positions and detection radii of the cameras, outputs which cameras (if any) Reptile gets too close to.

The Input:

Input begins with a single positive integer, n, on a line by itself, indicating the number of compounds facing Rigid Reptile (i.e., the number of data sets to be processed). Following that line are n compound descriptions. The description of a compound begins with a line containing two integers c and p representing the number of cameras and the number of points along Reptile's path, respectively, where $1 \leq c \leq 50$ and $2 \leq p \leq 50$. The next c lines contain three integers indicating the x coordinate, y coordinate, and detection radius, respectively, of a camera (consider the first camera to be numbered 1, the second camera to be numbered 2, etc.). The following p lines each contain two integers representing the x and y coordinates of successive points along Reptile's path (assume all points are distinct).

The Output:

Rigid Reptile moves in a straight line between successive points on his path. If at any time his Euclidean distance to a camera is less than or equal to its detection radius (or within 0.01), he has triggered that camera's alarm. For each compound (data set), print a line indicating which cameras Reptile has triggered. This line should be of the form:

`Compound #`*`i`*`:` *`message`*

where i is the compound number (starting at 1). If Reptile triggered no cameras, *message* should be "`Rigid Reptile was undetected`". If Reptile triggered one or more cameras,

message should be of the form "`Reptile triggered these cameras: list`" where *list* is a list of the cameras triggered by reptile, sorted in ascending numerical order (leave exactly one space between numbers on this list).

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

Sample Input:

```
3
3 3
5 5 3
2 8 2
7 3 1
4 5
2 7
6 3
1 4
0 0 5
10 0
10 10
0 10
0 20
2 2
0 0 5
20 0 5
13 1
27 1
```

Sample Output:

```
Compound #1: Reptile triggered these cameras: 1 2 3

Compound #2: Rigid Reptile was undetected

Compound #3: Reptile triggered these cameras: 2
```

UCF "Practice" Local Contest — Aug 25, 2012

Lifeform Detector

Filename: `lifeform`

Government scientists at Area 51 are developing a new program to detect alien lifeforms. In particular, they are interested in finding evidence of additional visits by aliens in the 1950's. From their first visit, the scientists have the make-up of the aliens' DNA (which is radically different from human DNA). The complete grammar for the DNA is as follows (lower-case letters represent terminal symbols and ε is the empty string):

```
<S> → a<T>b<S> | c<S> | ε

<T> → a<T>b<S> | c<S>
```

The Problem:

Given possible alien DNA patterns, determine if they match the alien DNA description given by the grammar.

The Input:

The first line of the input will consist of a positive integer *n*, representing the number of DNA patterns (i.e., the number of data sets to be processed). Each of the next *n* lines will contain a string of lower-case letters (at least one letter and at most 50 letters) that represent the pattern to check against the alien DNA grammar. Note that the input will be at least one letter even though the grammar allows for empty (null) string, i.e., empty string will not be in the input. Assume that input will not contain any character other than lower-case letters.

The Output:

For each DNA pattern, output the header "`Pattern `*`i`*`: `" where *i* is the number of the pattern in the input (starting with 1). Then, print "`More aliens!`" if the pattern matches the alien DNA description or "`Still Looking.`" if the pattern does not match. Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

(Sample Input/Output on the next page)

Sample Input:

```
2
aacbcbcc
aa
```

Sample Output:

```
Pattern 1: More aliens!

Pattern 2: Still Looking.
```

UCF "Practice" Local Contest — August 25, 2012

Ordered Numbers (Filename: order)

Given three integer numbers, your program is to print them from the smallest value to the largest value.

The Input:

The first input line is a positive integer n, indicating the number of data sets. Each of the following n input lines contains three integers, representing a data set.

The Output:

For each data set, first print a heading. Then, print the three numbers in the original order and from the smallest to the largest. Leave a blank line after the output for each data set. Follow the format illustrated in Sample Output.

Sample Input:

```
2
10 7 5
4 30 20
```

Sample Output:

```
Data set #1:
   Original order: 10 7 5
   Smallest to largest: 5 7 10

Data set #2:
   Original order: 4 30 20
   Smallest to largest: 4 20 30
```