University Interscholastic League

Computer Science Competition

2000 Regional Programming Set

I. General Notes

1. Do the problems in any order you like.

2. Problems have different point values, depending upon their difficulty.

3. Several problems have a reference to the columns of an input line. In these cases "column" refers to the character position on the input line. That is, "column 1" refers to the first character position on the line, and "columns 1-3" refers to the first three positions on the line.

4. Output should contain exactly what is specified by the problem. There should be no extra blank lines, spaces, or debugging output.

5. Input files will contain exactly what is specified. Specifically, there will be no blank lines or extra spaces in the input files.

II. Point Values and Names of Problems

Number	Name	Point Value
Problem 1	Automatic Homework	8
Problem 2	A Messy Proposition	3
Problem 3	Basically It	3
Problem 4	Double Talk	7
Problem 5	Hoops Anyone?	6
Problem 6	Pseudo-Random Number Checker	3
Problem 7	Typing Ahead	6
Problem 8	Wagon Train	5
Total		**41**

Problem 1

Automatic Homework

8 Points

Program Name: homework.cpp **Input File: homework.dat**

You are given a homework assignment to find the two U.S. state capitals that are closest by road mileage. To help in this task, your teacher gives you an atlas. To complete your assignment, you must tally the mileage along the roads between each pair of points. You quickly realize that exhaustively testing the distances between all capitals will be enormously difficult, requiring you to check 1225 (= 49 + 48 + 47 + ... + 3 + 2 + 1) paths. Therefore, you need to find another method for getting the correct answer.

You realize you can eliminate candidate distances between any two capitals if the Euclidean distance between them is greater than the best road distance you have found so far. Clearly, Hawaii and Alaska are not needed for this exercise. You also decide that some of the state capitals in the West are too remote to be viable candidates. Therefore, the input may not include all 48 continental state capitals.

The atlas has a 30-inch wide by 20-inch tall map of the "Lower 48" states. The map is drawn from a view of the spherical globe centered over Kansas, which reduces the distortion caused by projection of a 3-dimensional globe onto a 2-dimensional piece of paper. Since you are only attempting to find candidate pairs, you may ignore any remaining distortion. Given the (X, Y) coordinates in inches, and that one inch on the map is equal to 150 miles, write a program that prints the sorted top five candidates for closest capital pairs with the distance between each pair. Later, to complete your assignment, you will only need to compute the road distances between those candidates.

Input

Input to your program will consist of between 5 and 48 lines, each of which will contain a single state capital. The format for the lines of input is as follows:

STATE CITY X.X Y.Y

Where STATE is the name of a state; CITY is the capital of STATE; X.X is the X coordinate to 0.1 inches of accuracy, and Y.Y is the Y coordinate to 0.1 inches of accuracy. STATE and CITY will contain the underscore character ("_") in the place of a blank space, so there is no whitespace within STATE or CITY. You know that ($0.0 \le X \le 30.0$) and ($0.0 \le Y \le 20.0$). Although your program does not know exactly which corner of the map is the origin, it is the same for all measurements.

The input will include exactly one space between STATE and CITY, between CITY and X.X, and between X.X and Y.Y. There will be no leading, trailing, or extraneous embedded spaces on any input line nor will there be any leading zeroes in any input line unless needed as a digit before the decimal point. No state capital is listed twice.

Output

Your program should print the top five candidate pairs of state capitols (one per line of output) sorted from closest together to farthest apart. The format for each output line is as follows:

```
CITY1, STATE1 to CITY2, STATE2 distance is M.M
```

Where STATE1 and CITY1 are the STATE and CITY of the first capital under consideration and STATE2 and CITY2 are the second capital under consideration. The first capital printed is the one that appears earlier in the input file. M.M is the distance in miles (left justified with no leading zeroes) between the two capitals to an accuracy of 0.1 miles.

CONTINUED NEXT PAGE

PROBLEM 1 CONTINUED

Example: Input File

```
Louisiana Baton_Rouge 12.3 10.7
Mississippi Jackson 12.6 9.9
Texas Austin 9.7 10.8
Florida Tallahassee 15.2 10.7
Alabama Montgomery 14.3 10.1
South_Carolina Columbia 16.2 9.0
Georgia Atlanta 14.8 9.4
Tennessee Nashville 13.7 8.3
North_Carolina Raleigh 16.8 8.1
```

Output to screen

```
Baton_Rouge, Louisiana to Jackson, Mississippi distance is 128.2
Montgomery, Alabama to Atlanta, Georgia distance is 129.0
Columbia, South_Carolina to Raleigh, North_Carolina distance is 162.2
Tallahassee, Florida to Montgomery, Alabama distance is 162.2
Tallahassee, Florida to Atlanta, Georgia distance is 204.0
```

Problem 2

A Messy Proposition

3 Points

Program Name: messy.cpp **Input File: messy.dat**

A refuse collection company has to provide bids to local neighborhoods for the cost of collecting trash each week. Each neighborhood provides a list of streets and an estimated amount (in weight) of trash generated by the residents of the street. Your program should use these rules for routes when calculating bids:

1. A refuse truck holds a maximum of 10,000 pounds of trash.
2. A truck will start on the first street of the neighborhood and collect all of the trash on that street. The truck then collects trash on subsequent streets in the same order as they are listed in the input file.
3. If a truck cannot hold all of the trash on a new street, the truck will go to the landfill and empty its load before collecting any trash from the new street.
4. The truck will end the route with a trip to the landfill to empty the truck.
5. Regardless of the amount of trash to collect, it takes a truck exactly eight minutes to pick up all of the refuse on a single street.
6. It takes 30 minutes for the truck to go to the landfill, unload, and return to the street. On its last trip, it takes 30 minutes for the truck to go to the landfill, unload, and return to the collection station.
7. A truck and its crew costs $120/hour.
8. Each trip to the landfill costs $57, regardless of the amount of trash to be dumped.

Your program will read the data for a given neighborhood and determine the cost to collect and dispose of the trash on the street.

Input

Input to your program consists of a series of integers representing the weight of trash on a street. The truck will collect the trash in the order that the weights are listed. There is one integer weight per line and all weights are between 1 and 10000 pounds (inclusive). There will be at least one street in the input file. There is no limit to the number of streets.

Output

Output of your program should contain a single line formatted as "`Bid $`*X*" where *X* is the number of dollars (left justified with no leading zeroes) that the company should bid for the job based on the eight rules above. Case counts, your program should contain the dollar sign, and you should **not** print the number of cents.

Example: Input File

```
1765
2808
952
4206
3102
3902
1292
3985
8324
1928
4426
397
3277
```

Output to screen

```
Bid $910
```

Problem 3	**Basically It**	3 Points

Program Name: basic.cpp **Input File: basic.dat**

The year is 1970 and a new language is needed for computer programming. Your company has decided to create a new programming language called SIMPLE (Symbolic Instructions for Math and other Programming Logic Execution). Your company is implementing this language in multiple releases. Your job is to write the first release as described below. (Fortunately, someone has sent the compiler you are using for this contest through a time machine. Unfortunately, in 1978 a rival language beats your company's marketing strategy and BASIC becomes famous and SIMPLE goes the way of CPM.)

The first release of SIMPLE implements only two instructions as shown in Table 1.

Instruction	Format	Explanation	Examples
LET <var> = <constant>	LET in columns 1-3. Blanks in columns 4, 6, and 8. <var> in column 5. = in column 7. <constant> in column 9.	The value <constant> is assigned to <var >.	LET A = 5 LET B = 9
LET <var> = <var1> + <var2>	LET in columns 1-3. Blanks in columns 4, 6, 8, 10, and 12. <var> in column 5. <var1> in column 9. <var2> in column 13.	The values of <var1> and <var2> are added and the result is assigned to <var>.	LET C = A + B
PRINT <var>	PRINT in columns 1-5. Blank in column 6. <var> in column 7.	The value stored in <var> is printed (without a sign character) on a line by itself starting in column 1.	PRINT C

Table 1 : Instructions and Formats for SIMPLE Release 1

A <var> is a single upper-case alphabetic character (A-Z). A <constant> is a single digit integer (0-9) and is never preceded by a sign character. If you work the examples above, you will find that the PRINT C instruction will print 14 as the result.

Input

Your program will read a series of instructions from the input file. Every instruction will be syntactically correct, on a line by itself, and will be formatted as in Table 1. There will be no extraneous spaces at the end of any statement and no blank lines in the input. PRINT statements will not involve uninitialized variables. Your program will read to the end of the file, parsing and processing the SIMPLE instructions.

Output

For each line of input, your program is to process the instruction on the line as directed by Table 1. Your program will not produce any output for LET instructions. For each PRINT instruction, your program should print the value of the target variable. Note that the value of the variable of a PRINT statement may be multiple digits in length. All output produced by a PRINT instruction should start in column 1 of the output line and contain no leading zeroes.

CONTINUED NEXT PAGE

PROBLEM 3 CONTINUED

Example: Input file

```
LET A = 5
LET B = 9
LET M = A + A
PRINT A
PRINT M
LET D = M + B
PRINT D
```

Output to screen

```
5
10
19
```

Problem 4	Double Talk	7 Points

Program Name: talk.cpp **Input File: talk.dat**

Politicians are notoriously tricky. They have a habit of making statements that they can later claim you misunderstood. For example, if you heard a politician claim, "We never haven't considered a person's age when interviewing them for a job," what would you think they had said?

The multiple negative is a common tool of double-talking politicians. You are to write a program that will examine a series of quotes for multiple negatives. If it finds two or more negatives, it will report the quote as potential double-talk. Table 2 contains a list of the negatives your program should recognize.

Negative	Rules for Recognizing the Negative	Example Your Program Should Recognize as Negative	Example Your Program Should Not Recognize as Negative
not	"not" will appear as a separate word delineated by blanks before and after.	i have not been to tokyo	i have been to nottingham
never	"never" will appear as a separate word delineated by blanks or the start of the sentence	i will never go to libya	nevertheless, i have flown over the mediterranean
n't	"n't" will appear only as a suffix to another word of 1 or more alphabetic characters and is delineated by a trailing space.	i haven't been to tokyo	i will visit the ancient byzantine town of mahn'to

Table 2 : Recognizing Negatives in Sentences

Input

Input to your program will contain a series of quotes (one per line) from politicians. Quotes will not exceed 80 characters in length and are in lower case. Words are delineated by exactly one space, the beginning of a line, or the end of a line. The only punctuation the quotes may contain is the apostrophe. All quotes will conform to the "Rules for Recognizing the Negative" from Table 2.

Output

Output from your program will consist of the quotes with two or more negatives, one per line, as they appear in the input file.

Example: Input File

```
i have not been to tokyo
never have i not visited tripoli when touring africa
i haven't been to nottingham
we never haven't considered a person's age when interviewing them for a job
```

Output to screen

```
never have i not visited tripoli when touring africa
we never haven't considered a person's age when interviewing them for a job
```

Problem 5

Hoops Anyone?

6 Points

Program Name: hoops.cpp **Input File: hoops.dat**

For this problem, you are to write a program that will collect statistics for a basketball game. Table 3 lists all the events for which your program will collect statistics.

Event Acronym	Event Description	Parameter Explanation	Example Input Line
START	Start of the game.	No parameters necessary.	START
END	End of the game.	No parameters necessary.	END
QRTR	End of the current quarter. Quarters are numbered 1 - 4. There are exactly 3 QRTR events in a game as the 4th quarter ends with the END event.	No parameters necessary.	QRTR
FG	The player denoted by the parameter scores a field goal that is worth 2 points.	The single 2-digit integer parameter is the jersey number of the player who scored the field goal.	FG 31
FT	The player denoted by the parameter scores a free throw that is worth 1 point.	The single 2-digit integer parameter is the jersey number of the player who scored the free throw.	FT 23
3P	The player denoted by the parameter scores a field goal that is worth 3 points.	The single 2-digit integer parameter is the jersey number of the player who scored the three-point goal.	3P 12

Table 3 : Basketball Event Descriptions

Some things you should know:

1. The game consists of four 10-minute quarters. You are not tracking the playing time of each player, only the scoring statistics.
2. Players have unique 2-digit jersey numbers where the only valid digits are 0, 1, 2, 3, 4, and 5.
3. All events reported are in chronological order.
4. Parameters for the FG, FT, and 3P events are separated from the event acronym by exactly one space.
5. No player will score more than 99 points in a quarter or more than 999 points in a game. In fact, the team will not score more than 99 points in a quarter or more than 999 points in a game.

Your program is to produce a report in the following format where only those items in bold are printed.

COLUMNS ->> / Rows	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
Title Row	**P**	**L**	**A**	**Y**	**E**	**R**		**1**	**Q**		**2**	**Q**		**3**	**Q**		**4**	**Q**		**T**	**O**	**T**	**A**	**L**
Player Row #1			#	#				#	#		#	#		#	#		#	#			#	#	#	
Player Row #2			#	#				#	#		#	#		#	#		#	#			#	#	#	
Player Row #3			#	#				#	#		#	#		#	#		#	#			#	#	#	
…			#	#				#	#		#	#		#	#		#	#			#	#	#	
Player #N			#	#				#	#		#	#		#	#		#	#			#	#	#	
Team Totals Row	**T**	**E**	**A**	**M**				#	#		#	#		#	#		#	#			#	#	#	

Your program will print the bold information in the table above, starting with the title row and ending with the team totals row. Print the information in the title row exactly as it appears above. Then each player's statistics are printed on a separate line. For each player appearing in the input file, print his or her two-digit jersey number in columns 3-4. Print the number of points that the player scored in the 1st quarter in columns 8-9. Print the number of points that the player scored in the 2nd quarter in columns 11-12. Use columns 14-15 and 17-18 similarly for the 3rd and 4th quarter points. The total number of points scored by the player during the game should be printed in columns 21-23. In the last row, the numbers represent the number of points the team scored in each quarter, and the total number of points the team scored in the game. All point totals are right justified and do not use leading zeros.

CONTINUED NEXT PAGE

PROBLEM 5 CONTINUED

Input

Your program will read a series of events that describe a basketball game. Each line of input contains exactly one event. The first line of input will contain the START event and the last line of input will contain the END event. The file contains exactly 3 QRTR events. The file will also contain an unspecified number of FG, FT, and 3P events.

Output

Your program will read all events from the input file. When your program has collected all data from the input file, it should print the report to the screen. The format of the report is described above. The players should be sorted by jersey number in ascending order. Only print information for those jersey numbers that appear in the input file.

Example: Input File

```
START
FG 45
FG 00
FG 34
FT 34
FG 21
3P 45
QRTR
FT 45
FG 01
FG 45
3P 00
3P 45
FT 15
FG 45
3P 45
QRTR
3P 45
FG 21
FG 34
FT 00
FT 00
FG 12
FG 00
FG 45
QRTR
FT 12
FG 34
FG 00
FT 15
FT 15
FG 12
3P 12
FT 21
3P 12
END
```

Output to screen

```
PLAYER 1Q 2Q 3Q 4Q TOTAL
  00    2  3  4  2   11
  01    0  2  0  0    2
  12    0  0  2  9   11
  15    0  1  0  2    3
  21    2  0  2  1    5
  34    3  0  2  2    7
  45    5 11  5  0   21
TEAM   12 17 15 16   60
```

Problem 6 | # Pseudo-Random Number Checker | **3 Points**

Program Name: pseudo.cpp **Input File: pseudo.dat**

Computer applications often need random numbers as part of simulations and other processing. Since computers are unable to generate truly random numbers, we must use algorithms that generate pseudo-random numbers. The simplest such algorithm uses the following equation repeatedly:

$$result = (seed + increment) \% base$$

where *result* is the number that is generated,
seed is the result from the previous application of the equation ($0 \le seed < (base-1)$),
increment is a constant value such that ($1 \le increment < base$),
base is a constant value such that (*base* = (max random number desired + 1)), and
% is the modulus operator.

This equation is applied repeatedly using the current value of *result* as the value of *seed* in the next iteration. For example, the starting values of *base*=5, *increment*=3, and *seed*=0, generate the repeating sequence of pseudo-random numbers 3, 1, 4, 2, 0, 3, 1, 4, 2, 0, ….

A complete uniform distribution is one that generates all numbers between 0 and *base*-1 (inclusive) before repeating the initial seed. The example above is a complete uniform distribution because it generates all numbers between 0 and 4 inclusive before arriving at the initial seed. Using *base*=9, *increment*=3, and *seed*=4 generates the sequence 7, 1, 4, 7, 1, 4, …, which is not a complete uniform distribution because it will never generate the values 0, 2, 3, 5, 6, or 8. Note that the initial seed has no bearing on whether the distribution is a complete uniform distribution.

Input

For this problem, you program is to read a sequence of base/increment pairs and determine whether each pair will generate a complete uniform distribution. Input to your program consists of multiple lines of input with each line containing exactly one base integer ($1 < base \le 10000$) and one increment integer ($1 \le increment < base$). Each line will contain the base integer starting in column 1 followed by a single blank followed by the increment integer. There are no other spaces or extraneous values on the line.

Output

For each line of input, your program should print one of the following two lines:

```
base=b, increment=i, is a complete uniform distribution
base=b, increment=i, is not a complete uniform distribution
```

where *b* is the base value and *i* is the increment value supplied in the input. Your program prints the first message if the base/increment pair will generate all numbers between 0 and *base*-1 (inclusive) exactly once before repeating any of the numbers. Otherwise, your program should print the second message.

Example: Input File

```
5 3
9 3
45 27
100 47
```

Output to screen

```
base=5, increment=3, is a complete uniform distribution
base=9, increment=3, is not a complete uniform distribution
base=45, increment=27, is not a complete uniform distribution
base=100, increment=47, is a complete uniform distribution
```

Problem 7

Typing Ahead

6 Points

Program Name: typing.cpp **Input File: typing.dat**

In this problem, you will be writing the piece of a word processor that "looks ahead" for a typist and suggests a word. Typically, the look-ahead function would receive a prefix input from the word processor expecting a returned word. However, your program will use flat text input files.

This look-ahead function selects the shortest word beginning with the given prefix from a dictionary. If multiple words qualify as the shortest, the look-ahead function will choose the first one in the dictionary. For example, given the following dictionary segment and the prefix "reg", your program would select "register" because it is the shortest of the words starting with "reg".

Dictionary Words	Possible Match (Y/N)	Word Length
…	N	Not Applicable
red	N	Not Applicable
register	Y	8
regular	Y	7
regulate	Y	8
regulation	Y	10
reheat	N	Not Applicable
…	N	Not Applicable

Input

There are two components to the input file: the dictionary and the prefixes. The dictionary component begins on Line 1. Line 1 of the input file gives the number of dictionary entries as a single integer ($1 \le D \le 100$) beginning in column 1. Lines 2 through *D+1* each contain a single dictionary entry. Entries consist of lower case alphabetic characters and are between 1 and 20 characters long. The entries are sorted alphabetically. The prefixes begin on the line after the last dictionary entry, *D+2*. Each of these lines contains a single prefix. Prefixes consist of lower case alphabetic characters and are between 1 and 20 characters long.

Output

For each prefix in the input file, your program will print a response, one per line, to the screen. If no word in the dictionary matches the prefix, your program should print the prefix. Otherwise, the program should print the best look-ahead word from the dictionary.

Example: Input File

```
11
acropolis
read
red
regal
regalia
register
regular
regulate
regulation
reheat
zebra
regu
regali
ra
```

Output to screen

```
regular
regalia
ra
```

Problem 8 | # Wagon Train | **5 Points**

Program Name: wagon.cpp **Input File: wagon.dat**

During the 1800s, there were countless wagon trains that traveled across the United States. The horses required to pull the wagons could travel 18-22 miles per day depending on the terrain. This is why, along east-west highways, so many towns are spaced 20 miles apart. These towns are a result of people settling where the wagons camped.

As a member of the Ranching Heritage Appreciation Society (RHAS), you and a group of 100 other people love to take spring-break wagon train adventures along America's Highways. The challenge is that you do not always travel from east to west. Thus, the towns are sometimes improperly spaced for wagon travel.

A travel strategy is a rule of the form, "Camp overnight at the first town you arrive at after travelling a minimum of *M* miles." You are to write a program that will determine what, if any, travel strategy will allow your wagon train to complete the journey within the five days of spring break. The journey is limited to five days because the weekends before and after the journey are required for assembly and disassembly of the wagon. A journey can be completed with a travel strategy if it follows the rules of travel:

1. The wagon train begins at the start point.
2. By the end of the fifth day, the wagon train must have reached the end point.
3. The wagon train may reach the end point at any time during any travel day.
4. The wagon train cannot travel more than 22 miles in any single day.
5. When the wagon train arrives in a town, it makes a decision based on the travel strategy. If the wagon train has traveled at least *M* miles, it will camp for the night. If the wagon train has traveled less than *M* miles, it will continue to the next town. Note that if *M* is improperly chosen, this may result in the wagon train travelling more than 22 miles.
6. Towns are considered to be a single point.
7. Distances between towns are rounded to the nearest mile.

Consider the example journey and sample strategies in Table 4, given the above rules.

Town	**Journey Segment Distance**	**Sample Strategy with *M*=8**	**Sample Strategy with *M*=14**	**Sample Strategy with *M*=18**
Pratt (start)	N/A	Start	Start	Start
Cullison	10	Day 1 = 10 (camp)	Day 1 = 10 (<14)	Day 1 = 10 (<18)
Wellsford	8	Day 2 = 8 (camp)	Day 1 = 18 (camp)	Day 1 = 18 (camp)
Haviland	4	Day 3 = 4 (<8)	Day 2 = 4 (<14)	Day 2 = 4 (<18)
Brenham	6	Day 3 = 10 (camp)	Day 2 = 10 (<14)	Day 2 = 10 (<18)
Greensburg	5	Day 4 = 5 (<8)	Day 2 = 15 (camp)	Day 2 = 15 (<18)
Mullinville	8	Day 4 = 13 (camp)	Day 3 = 8 (<14)	Day 2 = 23 (error)
Bucklin	10	Day 5 = 10 (camp)	Day 3 = 18 (camp)	N/A
Ford	10	Journey unfinished	Day 4 = 10 (<14)	N/A
Willroads Gardens	12	Journey unfinished	Day 4 = 22 (camp)	N/A
Dodge City (end)	5	Journey unfinished	Day 5 = 5 (finish)	N/A

Table 4 : Sample Journey from Pratt, Kansas to Dodge City, Kansas

In Table 4, you can see that the strategy (*M*=8) does not work because the wagon does not reach the end point by the end of the fifth day. The strategy (*M*=18) does not work because the wagon train will travel more than 22 miles on the second day and result in overstressed horses. The strategy (*M*=14) is successful because it allows the journey to be completed by the end of the fifth day without traveling more than 22 miles on any given day.

Your job is to write a program that, given the list of journey segment distances, will determine if a strategy ($1 \le M \le 22$) exists. Note that the town names are unimportant and that no journey segment distance will be greater than 22 miles.

CONTINUED NEXT PAGE

PROBLEM 8 CONTINUED

Input

Input to your program is a sequence of journey definitions. A journey definition uses exactly one line of input and consists only of integers. The first integer N ($1 \leq N \leq 25$) represents the number of journey segments. The next N integers are the journey segment distances. Your program should read to the end of the file.

Output

For each journey in the input file, your program should indicate if the journey is OK. If there exists at least one travel strategy that will allow the journey to be completed while following the rules of travel, your program should print "`Journey is OK`" on a line by itself to the screen. If there are no strategies that will allow the journey to be completed, your program should print "`Journey is not OK`" on a line by itself to the screen.

Example: Input File

```
10 10 8 4 6 5 8 10 10 12 5
5 20 20 20 20 20
5 22 22 22 22 22
10 3 7 5 3 12 12 12 19 12 5
```

Output to screen

```
Journey is OK
Journey is OK
Journey is OK
Journey is not OK
```