DKC^3 2021 – Long Programming Problems

Important Reminders:

- Time remaining will be announced at 30 min., 5 min. and 1 min. left in the session.

- Assume leading zeros are used in all decimal notations unless the problem says otherwise. (ex: 0.167)

- Example input and output files are available in the **C:\DKC3\Long Programming** folder for each problem. These may be used to test your programs.

- Each program must read from its respective input file in the **C:\DKC3\Long Programming** folder and generate an output file in that same folder. The names of these files must match what is specified for each problem in this document. These output files will be retrieved electronically and scored by the judges at the end of the session.

- Please store your source code in the **C:\DKC3** folder. Do **NOT** store any code in the **C:\DKC3\Long Programming** subfolder. Anything stored there will be overwritten.

- At the end of the session, judges will overwrite the example input files with the official input files for each problem. Each file will contain 10 test cases. One member of each team will be asked by a competition organizer to run all completed programs. Programs may be recompiled at this point prior to being run, but no changes can be made to source code.

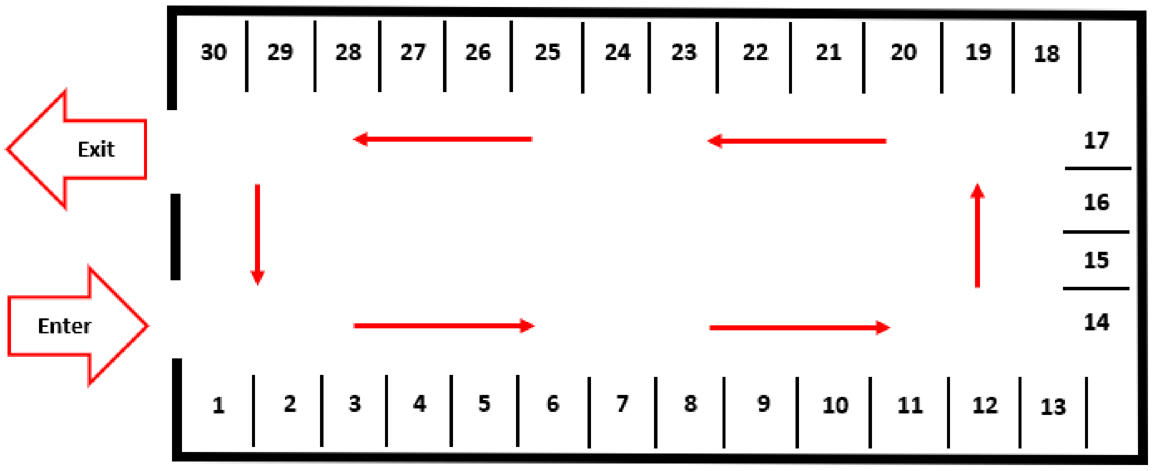
- Each program must complete execution within **two** minutes.

- After running your programs, be sure to close out of all IDEs.

DKC^3 2021 – Long Programming Problems

1. Parking Lot (75 points)

Parking is always an issue at Digi-Key University. Because of limited parking lots, students often are forced to wait for a parking space to free up. One of the small parking lots is arranged in a rectangular shape, with 30 spaces numbered 1, 2, 3..... 29, 30. Traffic flow is one way in a counter- clockwise direction. A diagram of the parking lot is below:



Note that the first position encountered upon entering is 1 and the last is 30. Cars may exit or continue to drive in a counter-clockwise direction. The following assumptions apply to this problem:

- At the start, class is in session and the lot is full (all 30 spaces are occupied by parked cars).
- In addition to the 30 cars already parked in the lot, ***k*** cars are in the lot waiting for positions to become available. (1 <= ***k*** <= 30)
- Each waiting car is positioned behind one of the occupied spaces. When a position empties, the space is filled either by the car waiting at that position or, if no car is waiting at that position, by the closest car, bearing in mind that the traffic flow is one way.
- When a waiting car advances ***n*** positions to a free spot, all other cars advance ***n*** positions. Since the lot is circular, advancing 4 positions from position 28 means advancing to position 2. If the position at which a car is currently waiting opens up, they advance 0 spaces and take that spot.
- None of the waiting cars exit until they have found a parking spot.

Input/Output Information on the next page.

Input
Each test case consists of two lines. The first line consists of a list of integers separated by spaces representing initial positions of waiting cars in the parking lot, in no particular order. The second line consists of integers in the same format representing parking spaces vacated. Positions are vacated in the order in which their numbers appear in the second line of the test case. An asterisk (*) on a line by itself separates each test case. There are ten test cases.

Output
For each test case, output as many lines as there are cars waiting to park. Each line should consist of the format "***x*** parked in ***y***" where ***x*** is the original position of the waiting car and ***y*** is the empty space in which that car parks, based on the description and assumptions stated above. The output lines must appear in the order that the parking spots open up, given in the second line of the input. If the input ends and there are still cars in the lot waiting to park, output "***x*** did not park" in the order that ***x*** appeared in the first line of the input. Print an asterisk (*) on a line by itself to separate the sections of output.

Input File: C:\DKC3\Long Programming\ParkingIn.txt
Output File: C:\DKC3\Long Programming\ParkingOut.txt

Examples:

Input:

```
6 25 17 15 1 13
1 3 30 21
*
1 7 8
7 8 9
*
```

Output:

```
1 parked in 1
25 parked in 3
17 parked in 30
6 parked in 21
15 did not park
13 did not park
*
7 parked in 7
8 parked in 8
1 parked in 9
*
```

DKC^3 2021 – Long Programming Problems

2. Joe's Socks (75 points)

Joe often goes out shopping with his friends. They typically buy candy bars and oddly colored socks. Joe keeps a journal of what they buy but tends to leave out details in his short hand.

For example:

{ Brown, Green }	***The two friends bought Brown and Green socks*
Joe = Brown socks	***Joe bought Brown socks*
Brown socks <> Crunch bar	***The person who bought Brown socks did not buy a Crunch bar*

Given one of Joe's journal entries, determine what color socks he bought. If there is not enough information to tell, just write all the possible colors in alphabetical order.

Input
Each test case is one of Joe's journal entries. It starts with a list of all socks for a shopping trip surrounded by curly braces, padded with a space on each side. The following lines are Joe's notes about the shopping trip. *Remember*: There is only one list of socks per journal entry, so a new list is the start of a new test case.

Output
Alphabetical, coma-separated list of the socks which might be Joe's based on his notes. *Note*: Sock colors are case sensitive, but case differences should not impact sort order. If two colors are identical aside from casing, either order will be accepted.

Input File: C:\DKC3\Long Programming\JoesSocksIn.txt
Output File: C:\DKC3\Long Programming\JoesSocksOut.txt

Examples:

Input:

```
{ Orange, Red, Yellow }
Ruby = Orange socks
Joe = Crunch bar
Kit Kat bar <> Red socks
{ Pink, blue, Cyan }
Bob = Cyan socks
Joe = Snickers bar
```

Output:

```
Yellow
blue, Pink
```

DKC^3 2021 – Long Programming Problems

3. Pokémon Go Go (75 points)

Project Catch Them All, Inc., wants to create a new product, called Pokémon Go Go. Users can purchase this application to help them play Pokémon Go. The software accesses the poké stop locations near the current location as well as a list of Pokémon that can be found at each stop. The application then computes the shortest route one can follow to catch all the unique Pokémon, and return to the starting point.

The program assumes that the user is in a city where travel is restricted to moving only in the north–south and east–west directions. The program also assumes that all poké stops are on the intersection of two roads.

For example, consider a case where the application finds five nearby poké stops. Each stop's location is indicated by two integers, (r,c), where r is the number of blocks north (positive) or south (negative) of your starting position and c is the number of blocks west (negative) or east (positive) of your starting position. Consider if the locations of the five poké stops are (5, 9), (20, 20), (1, 1), (1, 8) and (2, 8) while the names of the Pokémon found at these stops are Charmander, Torchic, Torchic, Eevee, and Dratini, respectively. It is clear that one does not have to visit both the second and third stops, since the same Pokémon can be caught at either of them. The best route is to visit the first, fifth, fourth, and third stops (in that order) for a total distance of 28 blocks, since:

- The distance from (0, 0) to (5, 9) is 14.
- The distance from (5, 9) to (2, 8) is 4.
- The distance from (2, 8) to (1, 8) is 1.
- The distance from (1, 8) to (1, 1) is 7.
- The distance from (1, 1) to (0, 0) is 2.

Input/Output information on the next page.

Input
The input holds 10 cases, separated by a period on one line. Each case begins with a single integer n, $0 < n \le 20$, which indicates the number of poké stops to consider. Each of the next n lines specifies the location of a poké stop and the name of a Pokémon that can be found there. The location is specified by two integers r and c separated by a single space, $-100 \le r, c \le 100$. The integers r and c indicate that the stop is r blocks north (positive) or south (negative) and c blocks east (positive) or west (negative) of the starting point. The location is followed by a single space and followed by the string p indicating the name of the Pokémon that can be caught there. Names have between 1 and 25 letters (using only a–z and A–Z). The number of unique Pokémon is always less than or equal to 15. Multiple pokémon can reside at a single poké stop and are listed on separate lines.

Output
For each test case, output the shortest distance, in blocks, required to catch all the unique Pokémon and return to the starting point.

Input File: C:\DKC3\Long Programming\PokemonIn.txt
Output File: C:\DKC3\Long Programming\PokemonOut.txt

Examples:

Input:

```
5
5 9 Charmander
20 20 Torchic
1 1 Torchic
1 8 Eevee
2 8 Dratini
.
1
1 1 Umbreon
.
```

Output:

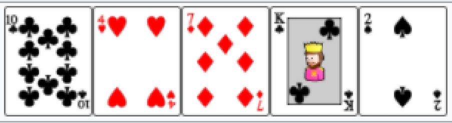
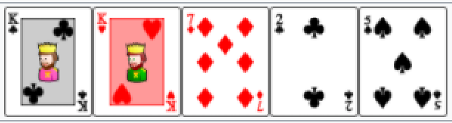
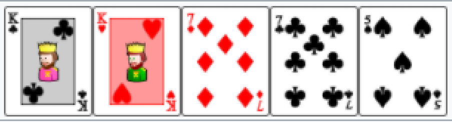
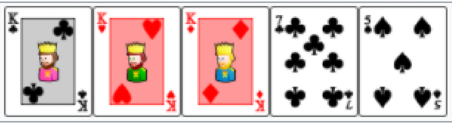
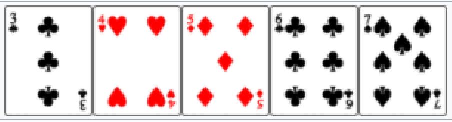
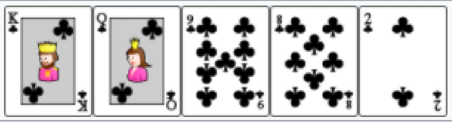
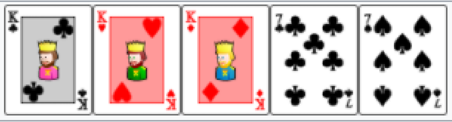
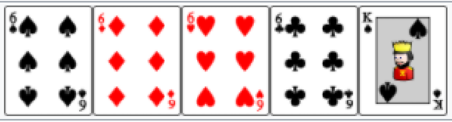
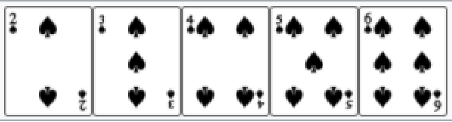
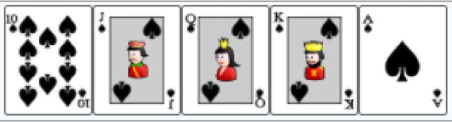
```
28
4
```

DKC^3 2021 – Long Programming Problems

4. Poker Probability (75 points)

Texas Hold 'Em is one of the most popular variants of the card game of poker. Two cards, known as hole cards, are dealt face down to each player, and then five community cards are dealt face up in three stages. The stages consist of a series of three cards ("the flop"), later an additional single card ("the turn" or "fourth street"), and a final card ("the river" or "fifth street"). Each player seeks the best five card poker hand from any combination of the seven cards; the five community cards and their two hole cards. Given the hands of five players and the flop, write a program to determine which hand has the highest chance to win and the probability of that hand winning?

The five cards progressively get better from top to bottom in the diagram below:

Name	Description	Example
Highcard	Simple value of the card. Lowest: 2 – Highest: Ace (King in example)	
Pair	Two cards with the same value	
Two pairs	Two times two cards with the same value	
Three of a kind	Three cards with the same value	
Straight	Sequence of 5 cards in increasing value (Ace can precede 2 and follow up King)	
Flush	5 cards of the same suit	
Full house	Combination of three of a kind and a pair	
Four of a kind	Four cards of the same value	
Straight flush	Straight of the same suit	
Royal flush	Straight flush from Ten to Ace	

Input

The input will consist of 10 test cases, with every two lines being a unique test case.

- The first line of each test case will consist of the five players' hole cards.
  - Each player is dealt two cards.
    - These cards will be separated by a space.
  - Each set of cards will be separated by a comma.
  - Each card will consist of two characters.
    - The first character will be the value.
      - [1-9]
      - T – 10
      - J – Jack
      - Q – Queen
      - K – King
      - A – Ace
    - The second character will be the suit.
      - S – Spades
      - C – Clubs
      - H – Hearts
      - D – Diamonds
- The second line of each test case will consist of the flop.
  - The format of the cards will follow the same rules as the first input line.
  - Each card will be separated by a comma.

Output

There should be one line of output for each test case. The output should be the set of two cards with the highest probability of winning, followed by the probability rounded to the nearest hundredth. Separate the cards and the probability by a comma. If there is a tie, output both sets of cards separated by a comma, followed by the probability. Maintain the original order if outputting more than one set of cards.

Input File: C:\DKC3\Long Programming\PokerProbabilityIn.txt
Output File: C:\DKC3\Long Programming\PokerProbabilityOut.txt

Examples:

Input:

```
5D 5H,7C AS,2S QC,7D JH,JS 8H
2C,3D,AC
5D 5H,7C AS,2S QC,7S AH,JS 8H
2C,3D,AC
```

Output:

```
7C AS,50.34
7C AS,7S AH,49.93
```