French Azerbaijani University
Arash Habibi
Second year
2020-2021

Data Structure and Algorithms 1

Qwirkle project

1 Introduction

1.1 The Qwirkle project overview

The aim of this project is to design a program enabling one or several players to play *Qwirkle*. The real *Qwirkle* game is described in this other page. And you can play *Qwirkle on line* on this page. But in this project, we change certain rules [1].

The project can be carried out at different levels. At the lowest level, the program simply displays one colored tile. At the most difficult level, the computer does not let you make illegal arrangements, counts the points of each player and can even play against a human player. If you can reach stage 4, you can be very proud. Read the general conditions (section 3) before beginning to carry out the project.

However, even at the highest level, there will be no graphical user interface. The board and the tiles will be represented by ASCII characters and printed from top to bottom on the terminal.

1.2 Contents

The game is composed of 32 tiles marked with letters *A, B, C* and *D* with colors : red, green, blue and yellow (figure 1). Each player has a deck of 6 tiles in front of him. The remaining tiles are put in a *draw bag*. In our version, the game is also equipped with a *board* with 11 *rows* and 11 *columns*, which makes 121 *cells*. In future versions, these numbers and sizes (32, 11 and 11) are likely to be modified.

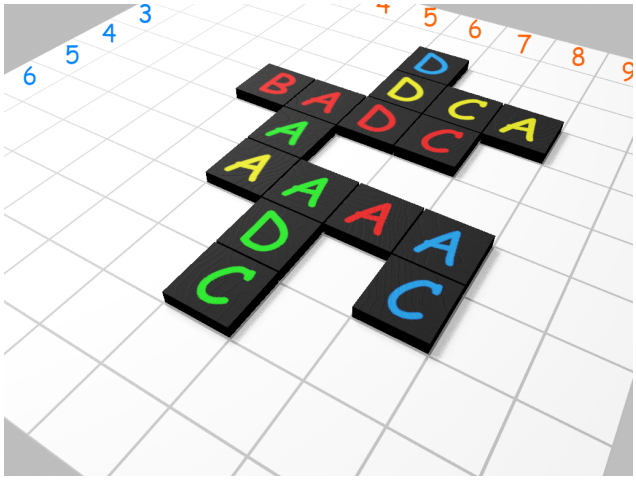

FIGURE 1: *Qwirkle* : board and tiles.

1.3 Neighbors and sequences

The global aim in *Qwirkle* is to make tile *sequences*.

neighbor : Two tiles are *neighbors* if and only if they have one adjoining edge. Two tiles with only adjoining corners are not neighbors ;

sequence : A *tile sequence* is a set of several *neighboring* tiles on the same line (same row or same column) and which have the same letter or the same color. But a sequence cannot contain any identical tiles (same color AND same letter).

complete : A *complete tile sequence* is a sequence of four tiles in which all four colors or all four letters appear.

Figures 2, 3, 4 and 5 represent sequences. As shown in figure 4, one tile can belong to two sequences (row and column). Figure 5 represents three sequences, two of which are complete (the BDCA sequence and the C C C C sequence).

1. More specifically, the tiles are not marked by colored patterns, but colored letters ; the board is limited to 11 rows and columns, and the players put one tile at a time (except in the last level)

FIGURE 2: An *A* sequence

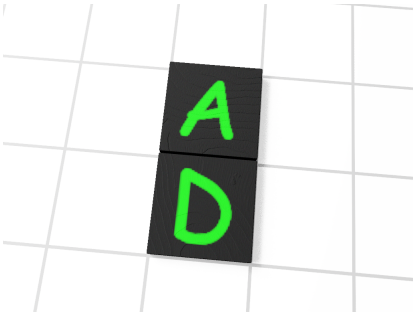

FIGURE 3: A green sequence

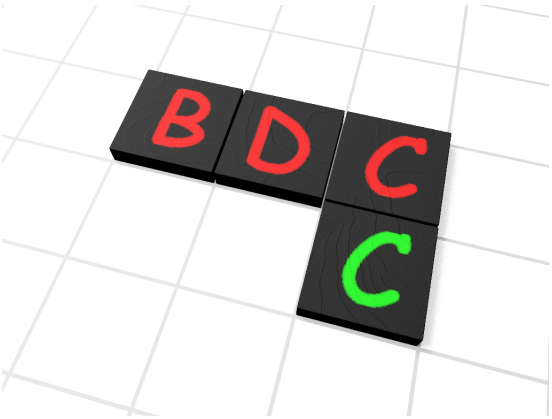

FIGURE 4: The C tile belongs to two sequences.

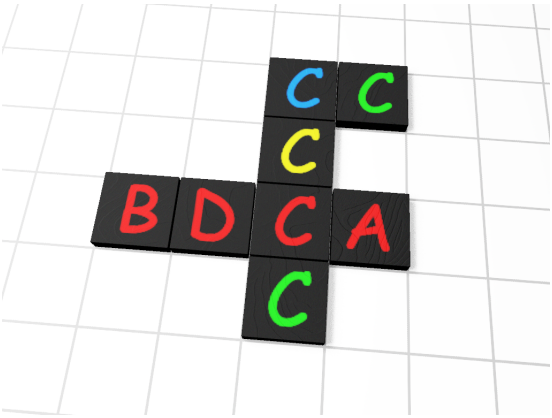

FIGURE 5: Three sequences

As stated in the beginning of this setion :

> *"A tile sequence is a set of several neighboring tiles on the same line (same row or same column) and which have the same letter or the same color. But a sequence cannot contain any identical tiles (same color AND same letter)."*

— *"Several"* implies that an isolated tile is not a sequence (figure 6).

— A sequence cannot contain any identical tiles (same color and same letter). This is the reason why figure 7 is not a sequence and also why, more generally, no sequence can have more than 4 tiles.

— Figure 8 looks very much like figure 4. We could think that it also represents two sequences. The difference between both figures is that in figure 8, the C tile is on the same line as the BDC tiles. And the above rule requires that all neighboring tiles on the same line have the same color or the same letter. This is not true in figure 8. So it does not form a sequence (not even one).

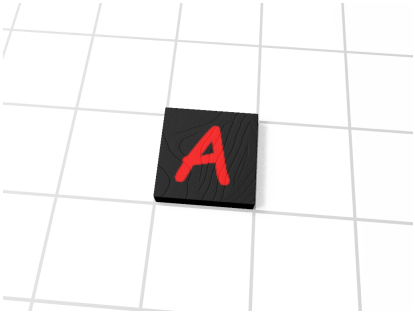

FIGURE 6: This is NOT a sequence.

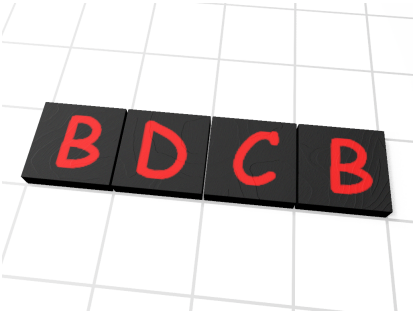

FIGURE 7: This is NOT a sequence

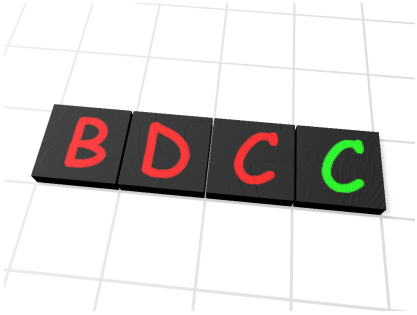

FIGURE 8: This is NOT a sequence

1.4 Initialization and rules

1.4.1 Initialization

draw bag At the beginning of the game, the 32 tiles are put in a *draw bag*.

deck Each player picks 6 tiles at random from the draw bag. These 6 tiles form the player's *deck*.

board One more tile is picked from the bag and put in the middle of the *board*.

1.4.2 Rules

Each player, in turn :

1. chooses one tile from her [2] deck and puts it on the board, such that the entire board contains only valid sequences ;
2. if she does not have such a tile in her deck, then she skips her turn.
3. if she has put a tile on the board, then she picks one tile from the draw bag and adds it to her deck, so that it contains, again, 6 tiles.

For example the board in figure 1 has only valid sequences. Here are a few implications of the above rules applied to this situation.

— If a player wants to place an A tile, she can only place it on cell (2,8) (row 2, column 8), on cell (3,4), on cell (5,4) and nowhere else.

— If the player has an A tile, she can place it nowhere. So she has to choose another tile or, if that is not possible, she has to skip her turn.

— No tile can be placed on the (5,6) cell (row 5 column 6), because that would make all the tiles on column 6 neighbors and the whole column (7 tiles) cannot be a sequence.

— No tile can be placed on the (5,7) cell (row 5 column 7) because the C C tiles cannot be in the same sequence as the A tile.

If you do not understand these statements or do not agree with them, please let me know. Either I am wrong or I did not explain the rules clearly.

1.4.3 Game end

Tiles are only taken from the draw bag and never put back. This means that, at some point, the draw bag will be empty. The players continue to play as long as all of them have at least one tile in their deck. The game ends :

— either when at least one player has an empty deck ;

— or when all players are blocked and cannot put any of their tiles on the board.

The winner is the player with the greatest number of points.

1.4.4 Counting the points

Each time a player adds a tile on the board, it brings her as many points as the number of tiles in all of the sequences to which the added tile belongs. For each sequence that has become complete, the players earns 4 extra points.

For example :

— in figure 3, adding a green C at the right of the A will make a sequence of 2 tiles (AC) and therefore will bring 2 points. But if the player had added it on top of the A tile, it would have made a sequence of three tiles (CAD) and would have brought 3 points.

— in figure 2, adding a blue A tile at the right of the A will make a sequence of 4 tiles (A A A A) and therefore will bring 4 points. But since this addition makes the sequence complete, it will bring 4 more points, so 8 points in all.

— in figure 5, adding a D tile at the left of the C tile (on top of the D) makes two sequences : one horizontal DC sequence (2 points) and one vertical D D sequence (2 points), so 4 points in all.

---

2. This game can be played by any person regardless of his or her gender. So when I write about *the player*, I should say *"He or She"*, which may make the text less readable. Some people save time and effort and replace the *"He or She"* by *"He"*. In the rest of this document, I will replace it by *"She"*, for a change. I hope you don't mind. Gentlemen, that may make us feel a bit strange, but I think we will survive !

2 Project implementation

2.1 General procedure

Section 2 describes the different levels. For each level, you will be given a set of specifications to meet. For the lower levels, you will be narrowly guided. In the higher levels, you will have more and more freedom to meet those specifications in your way. Here are the general guidelines that will be developed in our lectures. But I suggest that you do not wait for those lectures and feel free to see how these steps make sense for you and the way you think they should be followed. As you may guess, the first steps may require less time than the program/test steps. However they are very important because they determine your strategic programming choices and the capabilities of your program. Nevertheless, you may notice, in the course of these steps that you may have made the wrong choices. But it is never too late to come back to previous steps and make the correct choices.

1. algorithm : Write the *algorithm*, that is, the steps that you have to take to solve the problem. An algorithm does not depend on any programming language. You should be able to write it in plain English (or Russian or Azeri).

2. object names : In the algorithm, locate all the *nouns*. These nouns are potentially the name of the *objects* or *data structures* that you will have to create.

3. attributes : Determine the *attributes* these objects are associated with. It can be only a number, or it can be a whole set of numbers, character strings, booleans etc. The objects that are associated with more than one basic attribute (number, word, boolean) can be considered as real *data structures*. Each data structure will probably be associated with a header file and a source file.

4. relations : In this section, it may appear that some objects or data structures are parts of other data structures.

5. operations : In the algorithm, make the list of all the *operations* that should be carried out on each data structure.

6. data type : The set of operations determines what kind of *abstract data type* (tuple, stack, queue, list) and *concrete data type* (struct, array, simply-doubly-circular linked list) should be used.

7. header file : The set of operations and the data type enable you to write a header file for each data structure, with the definition of the structure and one or more functions for each operation.

8. source file : Write, compile and test the functions listed in the header file.

2.2 Stage 1 : Trainee level : testing the tools

Writing the *Qwirkle* program requires first being able to ask for user input and to print colored tiles on the terminal. This is what the program should do at this level :

Specifications :

1. Ask the user for her name ;
2. Ask the user *"Dear <username>, enter a capital letter and a color : "* and wait for her input.
3. If the color is anything else than *red, green, blue, yellow, purple, orange, white* then print an error message and list the available colors for the user.
4. Otherwise, print that letter with the specified color, on a black background.
5. If the user enters only a capital letter (without any color), print the letter with the default color on the default background.
6. If the user enters anything else, terminate the program politely, otherwise go back to step 2.

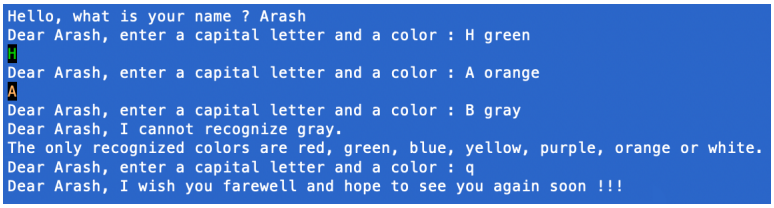

FIGURE 9: Demo

1. algorithm : In such a simple program, the algorithm is nothing more than the specifications.

2. object names : The list of all nouns are : *user, name, letter, color, background (color), program.*

3. attributes : Let us list the attributes that characterize each noun :

- user : name (character string)
- name : character string
- letter : character
- color : character string
- background color : character string
- program : none

4. relations : All of these nouns are not necessarily bound to make a separate data structure. Let us examine their relations :

- *name* characterizes *user*. Since *name* is the only attribute of *user*, they can both be represented by a character string called *user*.
- *color* and *background color* or of the same nature. They can be represented by the same data structure. You can decide whether you wish to represent it by the user-readable form (character string) or the machine-readable form (color index) or both.
- *program* : there is no attribute representing the program, so we do not need to represent it by any data structure.

In conclusion, we will use a character string for the user's name, a character for the letter, and a color index for the color (so we will have to write a function to translate color names to color indices and/or conversely).

5. operations : Let us read through the algorithm again and determine what operations need to be performed on each object.

user name : We need to (1.) ask for the user name and (2.) print the user name

letter : We need to (1.) print a letter with the default color and default background color and (2.) recognize whether it is a capital letter or not.

color : (1.) For a given color name, we need to produce the color index and (2.) we need to print a letter with a specified color index and with a black background color.

6. data type : Almost all of the objects of this program can be represented by basic data types (numbers, character strings).

Only colors may possibly be represented by a couple (name,index). However, the `printf` function, which enables to print text in the terminal, enables to write in color, but the colors are represented as color indices (integers between 0 and 255). And all of these colors do not have names. So maybe the best way would be to represent colors only by the color index and enable the conversion between the color indices and a more human readable format.

7. Header file You may write a header file with all these specifications, and function prototypes for the operations. Then you can compare it with the `Color.h` header file on *Moodle*.

8. Source file Same thing for the source file `Color.c` on *Moodle*. These files also contain a function called `C_printAllColors` which prints numbers between 0 and 255 on the terminal with the corresponding colors. This enables you to find the color index corresponding to a specific color.

The only thing that still must be done, is to create a `main.c` source file which uses the functions of `Color.c` to meet the specification of the program. The main difficulty at this level is to read a user input that may consist in one word or several words. Make sure you know the use of functions `fgets` and `sscanf`. If you find other solutions for doing the same thing, feel free to try.

In the rest of this project and in the practical exam, several rules must be respected :

— Your program should be modular. Each data structure and the connected function declarations should be defined a separate header file (with a `.h` extension) and these functions should be defined in the corresponding source file (with a `.c` extension).

— Only header files can be included by `#include` directives. Never include source files.

— Header files contain only data structure definitions and function prototypes, never function definitions or function calls.

— You should make separate compilation with a makefile. This means that, if you have already compiled your program and that you change a detail in one source file, you should be able to recompile the program without recompiling the unchanged source files.

2.3 Stage 2 : Technician level : the *draw bag* and the *deck*

In the previous level, the header file and the source files were supplied. In this level they will not be supplied. In this level the aim is to create a *bag* of tiles and a *deck*. Both represent a set of tiles. The bag initially contains all of the tiles. The tiles from the deck are tiles picked at random in the bag.

```
Hello, what is your name ? Raphael
Dear Raphael, how many samples of each tile would you like to have ? 1
Perfect ! Your bag contains 16 tiles now.
Dear Raphael, how many tiles do you wish to pick from the bag ? 5
C D B C B
There are 11 tiles left in the bag.
Dear Raphael, how many tiles do you wish to pick from the bag ? 10
C A C D B B A A D A
There are 1 tiles left in the bag.
Dear Raphael, how many tiles do you wish to pick from the bag ? 3
D
There are 0 tiles left in the bag.
Farewell dear Raphael ! Come back soon !
```

FIGURE 10: One interaction example

Specification :

1. Ask the user for her name.
2. Ask the user for the number n of identical tiles.
3. Create a bag (a collection) containing n A tiles, n A tiles, n A tiles and so on for all letters and all colors. This should produce $16 \times n$ tiles in all.
4. Create an initially empty deck (another collection of tiles).
5. Ask the user how many tiles she needs to pick from this bag. Let us call t the number of needed tiles.
6. If there are at least t tiles left in the bag, remove them from the bag, otherwise remove all of the tiles from the bag.
7. Put the removed tiles in the deck.
8. Print the contents of the deck.
9. Print the number of remaining tiles in the bag.
10. If the bag is empty, terminate politely, otherwise go to step 5.

An example of such an interaction can be seen in figure 10. Once again, let us follow the 8 steps together.

1. algorithm : Once again, the global algorithms is supplied by the specifications.

2. object names : Apart from the previously mentioned nouns, we have : *tile, bag, deck, collection* ;

3. attributes : Let us list the attributes that characterize each noun :

- tile : a letter and a color ;
- collection : a set of tiles ;
- bag : a set of tiles ;
- deck : a set of tiles ;

4. relations : *Collection*, *bag* and *deck* represent the same notion. A tile is an element of the bag or the deck. In conclusion, we need a tile data structure and another data structure which could contain several tiles.

5. operations : Let us read through the algorithm again and determine what operations need to be performed on each object.

tile : We need to (1.) create a tile (with a specific letter and a specific color) and (2.) print the tile on the terminal.

collection : We need to :

- create an empty collection ;
- add tiles to the collection ;
- determine whether the collection is empty ;
- determine the number of elements in the collection ;
- access a tile for a given index ;
- access a tile at random in the collection ;
- remove a specific tile from the collection ;
- remove all the tiles of a collection ;
- remove a tile at random from a collection and add it to another collection ;
- print the contents of a collection.

6. data type : What data type for the tile and for the collection ?

tile : The *tile* contains always the same number of elements (letter and color). Therefore the abstract data type is a *tuple*. In C, homogeneous tuples (with elements of the same type) can be created using structs, arrays or linked lists. But in our case, tiles are not homogeneous (letters and colors). So the only way to implement them is a `struct`.

bag : The *bag* and the *deck* do not always contain the same number of tiles, and the number of tiles may even change during the program. So this cannot be a tuple. Is it a stack ? a queue ? a list ? The key criterion is to determine from where the elements exit the structure :

- If the exiting elements are the newest elements, then we have a stack ;
- If the exiting elements are the oldest elements, then we have a queue ;
- If the exiting elements can be any element, then we have a list.

In here we choose the exiting elements at random, so we have a list. But what concrete data structure should we use ? Abstract lists can be implemented by arrays or by linked lists. This requires that we calculate the complexity of the operations of searching an element and removing an element. We will see that later in the lecture. For the time being, I ask you to implement these collections by a linked list of tiles.

7. Header file Make two header files `Tile.h` and `TileList.h`. In these header files, define the data structure and write the function prototypes corresponding to the operations listed above. To those operations, I always add one operation : print on the terminal the raw data of the structure (I call them `T_show` and `TL_show`). When you debug a program involving only numbers, you probably make an intensive use of `printf` to see why your program does not behave the way you thought it shoud behave. When you debug a program with structures, you need the same thing : some sort of `printf` for whole structures. This is the purpose of these two functions.

8. Source file Write the declared functions in `Tile.c` and `TileList.c`.

What you still have to do, is to write a `main.c` file that meets the specifications above.

In the rest of this project, and in the practical exam, your program should stay modular, which means that your functions must not be too long. If one of your functions appears long, feel free to divide it in several smaller functions. Moreover, if some operations need to be carried out often, feel free to define new functions (not listed in the operations mentioned above).

2.4 Stage 3 : Engineer level : the board

In this level, you will be able to put tiles from the deck to a board.

Specification :

1. Ask the user for her name ;
2. Ask the user for the number n of identical tiles and form a bag with $16n$ tiles. Make a deck and put 6 tiles from the bag to the deck.
3. Ask the user for the number of rows and columns on the board ;
4. Show the empty board and the deck (figure 11) ;
5. Ask the user to describe the tile you wish to place : tile index row, column ;
6. If the user enters more or less than the 3 specified arguments, or if those arguments cannot be interpreted as a tile index, a row or a column, then ignore the request and go back to step 5 ;
7. If the designated cell (row, column) does not belong to the board, or if it is already occupied by another cell, print an error message explaining why it is impossible to satisfy the user's request, and go back to step 5 ;
8. otherwise, remove the tile from the deck, put it on the board and take another tile at random from the bag to put in on the deck. Show the board and the deck and print the number of tiles remaining in the bag (figure 12). The tiles displayed on the board must represent not only the tile specified in the current iteration, but also all the tiles that the user has placed in the previous iterations (figure 13).
9. If the bag is empty, the game continues but the size of the deck decreases at each turn. If the deck is empty, terminate, otherwise go to step 5.

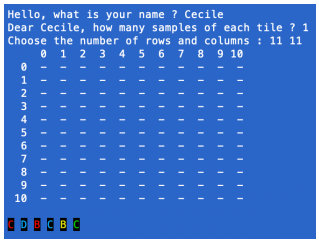

FIGURE 11: An empty board with 11 rows and 11 columns

FIGURE 12: A board with a B tile on (3,5).

FIGURE 13: A B tile on (3,5) (as previously) and a A in (4,6).

Once again, let us follow the above steps together.

1. algorithm : Once again, the global algorithms is supplied by the specifications, except when it comes to display the board with all the tiles that we have placed. The detail of this algorithms depends on the choice of our data structure.

2. object names : Apart from the previously mentioned nouns, we have : *board, row, column* and *cell* ;

3. attributes : Let us list the attributes that characterize each noun :

- board : number of rows, number of columns, set of tiles on the board ;
- row : an index (between 0 and nb_rows-1) ;
- column : an index (between 0 and nb_columns-1) ;
- cell : a row index, a column index, possibly a tile (?) ;

4. relations : It appears clearly that a board should be characterized by a number of rows and columns and a set of tiles.

5. operations : Let us read through the algorithm again and determine what operations need to be performed on the board.

- Create a new board with a specified number or rows and columns ;
- Determine whether a cell (defined by a row number and a column number) belongs to the board or not ;
- Indicate whether the cell (row, column) has a tile or not ;
- Access the tile on cell (row, column) if any ;

— Add a tile on cell (row, column).

— Print the contents of the board (as on the above figures).

6. data type : This is a real data structure issue. How can we put the tiles on the board ? There are at least two solutions. Both work. But the choice of one or the other will have a great influence on your program's performances.

1. Each tile is characterized not only by a letter and a color, but also by a row and a column. The board is a linked list of tiles. It is useless to represent cells.
2. the tiles keep the same structure as before. The board is a 2D array of $nb\_rows \times nb\_columns$ cells. Each cell is a pointer to the tile located on that cell. If there is no tile on that cell, the pointer is NULL. The position of the tile is determined by the row index and column index of the cell.

It is up to you to determine which one is the most efficient in terms of memory and in terms of computation time (and explain why). If we have to make a tradeoff, computation time, in our case, is more important than memory. I recommend you to determine how you would carry out the different operations (5) with both data structures. If you find that one of them does not work, I must have not been very clear because both do work. But which one is the most efficient ?

7. and 8. Header and source file : Implement this data structure in `Board.h` and `Board.c` and write a `main.c` which meets the specifications.

2.5 Stage 4 : Expert level : watch out where you put your tiles !

In the previous stage, you can already start playing alone or with another player. But the program does not help you check whether the tile placement is valid or not, and it does not either help you count your points. This is the aim of the present level. But the game can only be played by one player.

The interaction with the user is the same as in the previous level but :

1. The initial board should contain at least one tile taken at random in the bag and placed in the middle of the board.
2. If the player tries to put a tile on a board at a place where it has no neighbors, the program emits an error message explaining that each tile must have at least one neighbor. Then it asks the player to choose another tile or another position.
3. The program should check whether the place suggested by the player for his tile satisfies the rules stated in section 1.4.2. Hint if the suggested position is (r, c) then we must simply examine row r on one hand and column c on the other hand. In both lines, the added tile must form a sequence with all of its neighbors, and its neighbor's neighbors and so on.
4. The number of points earned by the player for each tile is calculated according to the principles mentioned in section 1.4.4.
5. Print the total number of points gained by the user and the number of tiles remaining in the bag.
6. We will no more ask the user to decide the size of the board or the number of identical tiles. We will take 11 rows and 11 columns for the board. The number of identical tiles will be 1 during the programming stage and will be 2 for games. So they will have to be easily modified, but not by the user. Only by the programmer.

From this stage on, it is up to you to read the specifications, to follow the eight steps and decide whether it is necessary or not to create new data structures, new header file and new source files. The following three stages are independant and can be added in any order.

2.6 Stage 5 : Guru level : several players

In this section, each player must have a specific name, deck and number of points. The program should ask how many users will be playing. It should ask the names, fill the decks of each player and ask them, in turn, what tile and what position they choose. If a player types 's', she skips her turn. The program should compute the number of points for each user and, at the end (section 1.4.3), the program should congratulate the winner.

Moreover, at this point, your `main.c` file may grow a bit too large, which is not a good programming habit. One suggestion is to create a `Qwirkle` object characterized by the board, the bag, and a list of players. The `main.c` file should look as simple as :

```
#include "Qwirkle.h"
int main()
{
  Qwirke q = Q_new(...);
  Q_displayAll(q);

  while(!Q_gameIsOver(q))
    for(int i=0; i<q.nb_players; i++)
    {
      Q_oneTurn(q,i);
      Q_displayAll(q);
    }

  Q_congratulate(q);
  return 0;
}
```

2.7 Stage 6 : Wizard level : the computer is a player

In this level, it should be possible for 0, 1, several or all players to be human or computer.

2.8 Stage 7 : Mutant genious mad scientist level : several tiles at each turn

In this level, at each turn, each player can put several tiles on the board. At least one of the tiles must be neighbor to a tile that was already on the board. All the tiles that the player puts on the board must belong to the same sequence and, as stated in 1.4.2, when the player has finished putting these tiles, the board must only contain valid sequences. You may freely decide how the user can specify several tiles and several positions.

The number of points is calculated in the same manner as before. It is the total number of tiles in all the sequences to which the laid tiles belong. For example, if the initial board is the one represented on figure 1, and if the player has tiles C, A, B in his deck, then she can put them on the board respectively on cells (2,7), (2,8) and (2,9). This makes three sequences : a vertical C, C, C sequence (3 points), a vertical A, A sequence (2 points) and a horizontal D, C, A, B complete sequence (8 points), which makes 13 points.

---

3 General conditions

This project will not be graded, therefore there is no deadline. You may work on this project alone or with other students or with the help of any person you wish to work with. However, I strongly recommend that you do not copy/paste any code but understand the proposed code and write your own version of the project. Otherwise, it will be very difficult (almost impossible) for you to do the practical exam, which will be closely related to this project and which will, hopefully, take place in the University in limited time.

The practical exam, will be graded. It will require that you have reached at least the engineer level. One small part of the exam can be done even if you have not done the project at all. Another part will consist of modifying your project. Thus if you are able to modify features introduced at the highest level, your grade will be better than if you only modify features of the engineer level.

On the other hand, it may be much more difficult to make changes to a complex program than to a simple program. This is the reason why I strongly recommend that, once you have finished a level, that you saved the program somewhere and never changed it. Then make a copy of this program and continue to higher levels with this copy. In this way you have a clean version of each level ready to be used. If you have any questions, feel free to ask me or your other teachers. My email address is `ahabibi@unistra.fr`.