Hints for Task C: pressUp, pressDown, pressLeft, pressRight methods of Game 1024

1 Examples of pressLeft, pressRight, pressUp, and pressRight

Suppose we have the following board layout, where the blank cell means 0.

Table 1: an example of Game 1024 board, where blank cells are zeros

1	1	2	2
1		2	
2	1	1	1
	1	1	1

1. After pressing left arrow, the same color cells are merged, and its value are doubled. The rightside table is what we expect.

   Note that in the third and fourth row, merge the two identical cells in the left.

Table 2: before and after pressing left arrow key

1	1	2	2
1		2	
2	1	1	1
	1	1	1

2	4		
1	2		
2	2	1	
2	1		

2. After pressing right arrow, the same color cells are merged, and its value are doubled. The rightside table is what we expect.

   Note that in the third and fourth row, merge the two identical cells in the right.

Table 3: before and after pressing right arrow key

1	1	2	2
1		2	
2	1	1	1
	1	1	1

		2	4
		1	2
	2	1	2
		1	2

3. After pressing up arrow, the same color cells are merged, and its value are doubled. The rightside table is what we expect.

   Since we press up arrow, in the second column, the top two identical cells are merged. There can be zeros, represented by blank cells, between those identical cells, just ignore those blank cells.

Table 4: before and after pressing up arrow key

1	1	2	2
1		2	
2	1	1	1
	1	1	1

2	2	4	2
2	1	2	2

4. After pressing down arrow, the same color cells are merged, and its value are doubled. The rightside table is what we expect.

   Since we press down arrow, in the second column, the bottom two identical cells are merged.

Table 5: before and after pressing down arrow key

1	1	2	2
1		2	
2	1	1	1
	1	1	1

2	1	4	2
2	2	2	2

2 Data members of Board class

Game 1024 board is represented by `Board` class with the following data members.

- `numRows`: number of rows
- `numCols`: number of columns
- `panel`: two-dimensional array of ints with `numRows` rows and `numCols` columns, representing a board for 1024 game.
- `max`: current maximum value of all cells in `panel`
- `target`: the number to be reached for a win

3 General Ideas for Pressing Arrow Keys

3.1 Work by row or column?

When pressing left / right arrow key, work in horizontal direction, that is, row by row. When pressing up / down arrow key, work in vertical direction, that is, column by column.

3.2 First Step: ignore zeros, save non-zeros to a vector of ints

Identical cells separated by zeros can still be merged. For example, when pressing left arrow on row, these two ones can be merged.

```
1 0 0 1
```

Similarly, when pressing up / down arrow key in the following column, those twos can be merged.

```
2
0
2
1
```

To ignore zeros between identical cells, save only non-zeros of a row (when pressing left/right) or non-zeros of a column (when pressing up/down).

Since the number of non-zeros in each row/column varies, use a vector of ints to save them.

The order of putting those non-zeros to a vector is important as well. Suppose we have a row {1, 0, 1, 1, 2}.

- When pressing left, work with cells in each row **from left to right**, so the vector is {1, 1, 1, 2}.
- When pressing right, work with cells in each row **from right to left**, so the vector is {2, 1, 1, 1}.

As another example, suppose we have a column as follows,

2
0
2
1

- When pressing up, work with cells in each column **from top to bottom**, so the vector is {2, 2, 1}.
- When pressing down, work with cells in each column **from bottom to top**, so the vector is {1, 2, 2}.

3.3 Merge Adjacent Identical Items

Since non-zeros elements for the vector are saved in the order consistent with the direction of arrow keys, we can merge the identical adjacent items from left to right, no matter whichever arrow key is pressed.

To reduce code redundancy, define function `merge`, called by methods of pressing arrow keys.

3.3.1 optional: why not define merge as a private method

As a sidenote, students may ask why not define `merge` as a private method of `Board` class. That approach has the following shortcomings.

(1) Adding a method, even a private one, may violate the idea of encapsulating only must-have operations on data members.

(2) Need to redefine `Board` class by adding that method to `Board.hpp`.

(3) Require that non-zeros in each row or column collected in the order consistent with the direction of arrow keys. However, some users would like to collect in different ways. For example, when pressing the right key, users can collect non-zeros from left to right, an order not consistent with right arrow key. When merging, merge from right to left.

3.3.2 Steps to merge

Key ideas in merging are listed as follows.

(1) Each adjacent pair is represented by the left index and the right index, but these indices are not independent, know one, know the other. Without loss of generality, let the left index of an adjacent pair be i, then the index of its right neighbor is $i + 1$. The value of i starts from zero.

(2) When finding an adjacent identical pair, do the following.

(a) Merge identical adjacent elements by doubling the value of the left element while setting the value of the right element to be zero. For example, given a vector with values {1, 1, 1, 2}, after merging the first two items, we get {2, 0, 1, 2}.

(b) Increase index by 2 to move to the left index of the next adjacent pair. Index is changed to 2, so we can process adjacent pair with values 1 and 2 as in {2, 0, 1, 2}.

(3) When there is no adjacent identical pair, increase index by 1. Suppose we have a vector with values {1, 2, 2, 1}, where the left index of the current adjacent pair is 0 and the right index of that pair is 1.

Since element 1 indexed at 0 and element 2 indexed at 1 are not identical, they cannot be merged. Increase current index by 1, which can be the left index of next identical adjacent pair, as shown in {1, 2, 2, 1}.

4 Hints for method pressLeft

(1) When we press left, we work on row by row. Row index, represented by row, changes from 0 to $numRows-1$. That is, $0 \leq row < numRows$.

(2) For each row, work with each cell **from left to right** since we press the left arrow key. For a given row, column index, represented by col, ranges from 0 to $numCols-1$ when working with left to right. That is, $0 \leq col < numCols$.

```
void Board::pressLeft() {
    for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols; col++) {
            //TODO: work with panel[row][col].
        }
    }
}
```

(3) For each row, collect non-zeros. Since the number of non-zeros for each row is not known, we can use `vector<int>` to store the non-zero elements in each row.

This step is needed since not some identical values separated by zeros like row with elements {1, 0, 1, 1} like the first and third elements can be merged.

```
void Board::pressLeft() {
    for (int row = 0; row < numRows; row++) {
        //Save non-zero elements of each row.
        //In the beginning of each row,
        //declare an empty vector of ints named nonZeros.
        vector<int> nonZeros;

        for (int col = 0; col < numCols; col++) {
            //TODO: work with panel[row][col].
            if (panel[row][col] != 0)
                save panel[row][col] to nonZeros;
        }
    }
}
```

After this step, when we work with the first row of Table 6.

Table 6: an example of Game 1024 board, where blank cells are zeros

1	1	2	2
1		2	
2	1	1	1
	1	1	1

After working with the first row, `nonZeros` has values {1, 1, 2, 2}.

After working with the second row, `nonZeros` has values {1, 2}.

After working with the third row, `nonZeros` has values {2, 1, 1, 1}.

After working with the fourth row, `nonZeros` has values {1, 1, 1}.

(4) Define a function (not a method of `Board` class) that merge the adjacent identical values of vector of ints.

By default, vector passes by value. In our application, we change directly on the given vector. Hence, instead of using the default setting, we pass by reference.

```
void merge(vector<int>& result) {
    //We should not use for-statement, since
    //not every element is treated the same. That is,
    //some elements are doubled, some are changed to zeros,
    //and some elements are not changed.
    //So we need to use while-statement.

    int i = 0;
    int size = ... //TODO: what is the size of vector result?
    //Since there is no deletion or insertion in the following
    //operations, the size of vector is not changed.
    //As a result, we can save the size of vector result
    //to variable size and put it before the while-loop.
    while (i < size) {
        if (i+1 is a valid index and result[i] equals to result[i+1]) {
           //i+1 must be a valid index before we use it in result[i+1]
           //Think about the following example, when i is 0 and
           //i+1 is a valid index and result[i] equals result[i+1]
           //          index 0  1  2  3
           //vector elements 2, 2, 1, 1
           //TODO: update result[i].
           //TODO: update result[i+1].
           //After the above statements, we expect to get
           //vector elements 4, 0, 1, 1
           //TODO: update i to move to the left index of the next adjacent pair.
     }
     else //either i is the last index so i+1 is not valid anymore
          //or i is not the last index so i+1 is a valid index,
          //however, result[i+1] is not same as result[i],
          //TODO: how to update i?
          //Think about the example when index is 0 and
          //vector elements are 2, 1, 1, 1
          //          index 0  1  2  3
          //vector elements 2, 1, 1, 1

    }
}
```

(5) Copy the non-zeros of vector `result` back to the row, from left to right (since we press left arrow, the direction starts from left to right).

(6) Since the number of non-zeros might not equal to the number of columns in a row, pad the rest of elements of that row by zeros.

A complete pseudocode is as follows.

```
void merge(vector<int>& result) {
    //TODO: write code here
}

void Board::pressLeft() {
    for (int row = 0; row < numRows; row++) {
        //Save non-zero elements of each row.
        //In the beginning of each row,
        //declare an empty vector of ints named nonZeros.
        vector<int> nonZeros;

        for (int col = 0; col < numCols; col++) {
            //TODO: work with panel[row][col].
            if (panel[row][col] != 0)
                save panel[row][col] to nonZeros;
        }

        //TODO: call merge function on vector nonZeros.

        //TODO: copy non-zeros of nonZeros back to the current row, from left to right.
        //Since in method pressLeft, the elements are merged from the left.

        //TODO: pad the remaining elements in the row to be zeros.
    }

    //TODO: call selectRandomCell method to put 1 to a cell with zero.
    //Since the header of is selectRandomCell is
    //void selectRandomCell(int&, int&),
    //we can declare two integer variables,
    //and pass their references to method selectRandomCell.

    //You can name these variables as row and col,
    //since the previous declared row and col are restricted
    //to nested for-loops and will not mess up with
    //the newly declared row and col at this position.
}
```

5 Pseudocode for pressRight

Difference of `pressLeft` and `pressRight` are the order of saving non-zeros and the order of padding elements: `pressLeft` starts from left to right, `pressRight` starts from right to left.

```
void Board::pressRight() {
    for (int row = 0; row < numRows; row++) {
        //Save non-zero elements of each row.
        //In the beginning of each row,
        //declare an empty vector of ints named nonZeros.
        vector<int> nonZeros;

        for (int col = numCols-1; col >= 0; col--) {
            //TODO: work with panel[row][col].
            if (panel[row][col] != 0)
               save panel[row][col] to nonZeros;
```

```
        }

        //TODO: call merge function on vector nonZeros.

        //TODO: copy non-zeros of nonZeros back to the current row, from right to left.
        //Since in method pressRight, the elements are merged from the right.

        //TODO: pad the remaining elements in the row to be zeros.
    }

    //TODO: call selectRandomCell method to put 1 to a cell with zero.
}
```

6 Pseudocode of pressUp

Method `pressUp` works from column to column. Work the cells in the same column from top to bottom. The first index of 2-dimensional array `panel` is row, the second index is column.

```
void Board::pressUp() {
    for (int col = 0; col < numCols; col++) {
        //Save non-zero elements of each column.
        //In the beginning of each column,
        //declare an empty vector of ints named nonZeros.
        vector<int> nonZeros;

        for (int row = 0; row < numRows; row++) {
            //TODO: work with panel[row][col].
            if (panel[row][col] != 0)
               save panel[row][col] to nonZeros;
        }

        //TODO: call merge function on vector nonZeros.

        //TODO: copy non-zeros of nonZeros back to the current column, from top to bottom.
        //Since in method pressUp, the elements are merged from the top.

        //TODO: pad the remaining elements in the column to be zeros.
    }

    //TODO: call selectRandomCell method to put 1 to a cell with zero.
}
```

7 Pseudocode of pressDown

Method `pressDown` works from column to column. Work the cells in the same column from **bottom** to **top**.

```
void Board::pressDown() {
    for (int col = 0; col < numCols; col++) {
        //Save non-zero elements of each column.
        //In the beginning of each column,
        //declare an empty vector of ints named nonZeros.
        vector<int> nonZeros;

```

```
        for (int row = numRows-1; row >= 0; row--) {
            //TODO: work with panel[row][col].
            if (panel[row][col] != 0)
               save panel[row][col] to nonZeros;
        }

        //TODO: call merge function on vector nonZeros.

        //TODO: copy non-zeros of nonZeros back to the current column, from bottom to top.
        //In method pressDown, elements are merged from the bottom.

        //TODO: pad the remaining elements in the column to be zeros.
    }

    //TODO: call selectRandomCell method to put 1 to a cell with zero.
}
```

8 a working sample

Here is a way that you test code in a local computer before you submit to gradescope. See code to define and test pressUp.

Code of `Board.hpp` is as follows.

```
#ifndef BOARD_H
#define BOARD_H
class Board
{
private:
    int** panel;
       //two dimensional array with numRows rows
       //and numCols columns
    int numRows;
    int numCols;
    int target; //what is the goal
    int max; //the current max in all cells of panel

public:
    Board(); //construct a 3 x 3 panel
    Board(int m); //construct a m x m panel
    Board(int m, int n); //construct a m x n panel
    void setTarget(int goal);
        //set goal of the game
    ~Board(); //destructor,
        //when no longer need the current object,
        //release dynamic memory of this object.
    void allocateMemory();
        //apply dynamic memory for panel
        //so that panel has numRows rows and
        //numCols columns
    void clear();
        //set each cell of the panel to be zero
    void print() const;
        //print the panel
    void selectRandomCell(int& row, int& col);
```

```
        //select a random cell from empty cell
    void pressUp();
    void pressDown();
    void pressLeft();
    void pressRight(); //press right key
    void start(); //start the game
    bool noAdjacentSameValue() const;
        //if there is no two adjacent cells
        //share same value, return true,
        //otherwise, return false.
};
#endif
```

Code of `Board.cpp` for Task C with implementation of `pressUp` method.

```
#include "Board.hpp"
#include <vector>
#include <iostream>
#include <iomanip> //setw
using namespace std;

void merge(vector<int>& result) {
    int i = 0;
    int size = result.size();
    //Vector result does not change size in this application,
    //so we can save result.size() to a variable size.
    while (i < size) {
        if (i+1 < size && result[i+1] == result[i]) {
          result[i] *= 2;
          result[i+1] = 0;
          i += 2;
        }
        else
        //else means the opposite of condition
        //(i+1 < size && result[i+1] == result[i]),
        //meaning i has next neighbor and
        //ith element has the same value as (i+1)th element.
        //So, in De Morgan's law,
        //(! (i+1 < size && result[i+1] == result[i]) )
        //is the same as
        //(i+1 >= size  result[i+1] != result[i])
        //which means
        //either i+1 >= size, that is, i is the last index,
        //or (i+1 < size && result[i+1] != result[i]),
        //that is, ith element has right neighbor,
        //but ith element does equal to its right neighbor.
        //In that case, we simply increase i by 1,
        //ie, move to the index of the right neighbor, if any,
        //of ith element.
        i++;
    }
}

void Board::pressUp() {
    //for each column
    //for a fixed column, run row from top to bottom,
```

```
42      //which is merge direction.
43      //(1) find number of non-zeros in that column,
44      //   from top to bottom.
45      //(2) merge the non-zeros (note that when we
46      //   get the non-zeros, we get from the direction
47      //   from top to bottom, so the non-zeros are merged
48      //   from top to bottom as well).
49      //   When merging, if the adjacent items have the same
50      //   value, then the first element is doubled,
51      //   the second element is zero,
52      //   move the next adjacent pair.
53      //   Otherwise (the adjacent items do not have same value,
54      //   move to the next adjacent pair).
55      //(3) Copy the merged value to the original column,
56      //   from top to bottom.
57      for (int col = 0; col < numCols; col++) {
58          vector<int> nonZeros; //need to be empty in each column
59          //we declare it before we work for each column
60
61          //for each column, work in each row.
62          for (int row = 0; row < numRows; row++) {
63              //the first index is row, the second index is column
64              if (panel[row][col] != 0)
65                 nonZeros.push_back(panel[row][col]);
66          }
67
68          merge(nonZeros);
69              //pass by reference, modified directly from the given parameter
70
71          //copy non-zeros of merged result from top to bottom
72          int row = 0;
73          for (int i = 0; i < nonZeros.size(); i++)
74              if (nonZeros[i] != 0) {
75                 panel[row][col] = nonZeros[i];
76                 row++;
77              }
78
79          //set the remaining elements in (col) index column to be zero.
80          while (row < numRows) {
81              panel[row][col] = 0;
82              row++;
83          }
84      }
85
86      int row = 0;
87      int col = 0;
88      selectRandomCell(row, col);
89  }
90
91  //TODO: when submit to Task C or Task D, need to use correct version.
92  void Board::selectRandomCell(int& row, int& col) {
93       //dummy function body, placeholder
94  }
95
96  //TODO: for test purpose only, need to comment out
```

```
//when submitting for grading Task C.
//Need to use correct version of Board when submitting for Task D.
Board::Board() {
    numRows = 4;
    numCols = 4;

    panel = new int*[numRows];
    for (int row = 0; row < numRows; row++)
        panel[row] = new int[numCols];

    int arr[][4] = {
        {1, 1, 2, 2},
        {1, 0, 2, 0},
        {2, 1, 1, 1},
        {0, 1, 1, 1}};

    for (int row = 0; row < numRows; row++) {
        for (int col = 0; col < numCols; col++)
            panel[row][col] = arr[row][col];
    }
}

void printSeparateLine(int numCols) {
    cout << "+";
    for (int i = 0; i < numCols; i++)
        cout << "----+";

    cout << endl;
}

//TODO: write the code
void Board::print() const {

}

Board::~Board() {
    for (int row = 0; row < numRows; row++) {
        delete[] panel[row];
        panel[row] = nullptr;
    }

    delete[] panel;
    panel = nullptr;
}
```

Code of `TestBoard.cpp` is as follows. In onlinegdb.com, function `main` must be in `main.cpp`. When test in onlinegdb, copy the following code to main.cpp.

```
#include <iostream>
#include "Board.hpp"
using namespace std;

//TODO: test pressUp, pressDown, pressLeft, pressRight methods of Board class
int main() {
    Board bd;
    bd.pressUp();
```

```
    bd.print();

    return 0;
}
```

To run the code, in command line, run

```
g++ -o run -std=c++11 TestBoard.cpp Board.cpp
./run
```

The output is as follows.

```
+----+----+----+----+
|   2|   2|   4|   2|
+----+----+----+----+
|   2|   1|   2|   2|
+----+----+----+----+
|    |    |    |    |
+----+----+----+----+
|    |    |    |    |
+----+----+----+----+
```

9 Code to submit to Task C of gradescope

Submit `Board.cpp` to Task C of gradescope.

(1) Comment definition of `Board::Board()`.

(2) Comment definition of `Board::~Board()`.

(3) Comment definition of `selectRandomCell`.

(4) Add definition of methods `pressLeft, pressRight, and pressDown`.

```
#include "Board.hpp"
#include <vector>
#include <iostream>
#include <iomanip> //setw
using namespace std;

void merge(vector<int>& result) {
    int i = 0;
    int size = result.size();
    //Vector result does not change size in this application,
    //so we can save result.size() to a variable size.
    while (i < size) {
        if (i+1 < size && result[i+1] == result[i]) {
           result[i] *= 2;
           result[i+1] = 0;
           i += 2;
        }
        else
        //else means the opposite of condition
```

```
        //(i+1 < size && result[i+1] == result[i]),
        //meaning i has next neighbor and
        //ith element has the same value as (i+1)th element.
        //So, in De Morgan's law,
        //(! (i+1 < size && result[i+1] == result[i]) )
        //is the same as
        //(i+1 >= size  result[i+1] != result[i])
        //which means
        //either i+1 >= size, that is, i is the last index,
        //or (i+1 < size && result[i+1] != result[i]),
        //that is, ith element has right neighbor,
        //but ith element does equal to its right neighbor.
        //In that case, we simply increase i by 1,
        //ie, move to the index of the right neighbor, if any,
        //of ith element.
        i++;
    }
}

void Board::pressUp() {
    //for each column
    //for a fixed column, run row from top to bottom,
    //which is merge direction.
    //(1) find number of non-zeros in that column,
    //   from top to bottom.
    //(2) merge the non-zeros (note that when we
    //   get the non-zeros, we get from the direction
    //   from top to bottom, so the non-zeros are merged
    //   from top to bottom as well).
    //   When merging, if the adjacent items have the same
    //   value, then the first element is doubled,
    //   the second element is zero,
    //   move the next adjacent pair.
    //   Otherwise (the adjacent items do not have same value,
    //   move to the next adjacent pair).
    //(3) Copy the merged value to the original column,
    //   from top to bottom.
    for (int col = 0; col < numCols; col++) {
        vector<int> nonZeros; //need to be empty in each column
        //we declare it before we work for each column

        //for each column, work in each row.
        for (int row = 0; row < numRows; row++) {
            //the first index is row, the second index is column
            if (panel[row][col] != 0)
              nonZeros.push_back(panel[row][col]);
        }

        merge(nonZeros);
            //pass by reference, modified directly from the given parameter

        //copy non-zeros of merged result from top to bottom
        int row = 0;
        for (int i = 0; i < nonZeros.size(); i++)
            if (nonZeros[i] != 0) {
```

```
75              panel[row][col] = nonZeros[i];
76              row++;
77          }
78
79      //set the remaining elements in (col) index column to be zero.
80      while (row < numRows) {
81          panel[row][col] = 0;
82          row++;
83      }
84  }
85
86  int row = 0;
87  int col = 0;
88  selectRandomCell(row, col);
89  }
90
91  //TODO: when submit to Task C, need to comment out.
92  //when submit to Task D, need to use correct version.
93  //void Board::selectRandomCell(int& row, int& col) {
94  //    //dummy function body, placeholder
95  //}
96
97  //TODO: for test purpose only, need to COMMENT OUT
98  //when submitting for grading Task C.
99  //Need to use correct version of Board when submitting for
100 //Task D.
101 //Board::Board() {
102 //    numRows = 4;
103 //    numCols = 4;
104 //
105 //    panel = new int*[numRows];
106 //    for (int row = 0; row < numRows; row++)
107 //        panel[row] = new int[numCols];
108 //
109 //    int arr[][4] = {
110 //        {1, 1, 2, 2},
111 //        {1, 0, 2, 0},
112 //        {2, 1, 1, 1},
113 //        {0, 1, 1, 1}};
114 //
115 //    for (int row = 0; row < numRows; row++) {
116 //        for (int col = 0; col < numCols; col++)
117 //            panel[row][col] = arr[row][col];
118 //    }
119 //}
120
121 void printSeparateLine(int numCols) {
122     cout << "+";
123     for (int i = 0; i < numCols; i++)
124         cout << "----+";
125
126     cout << endl;
127 }
128
129 //TODO: finish the definition
```

```
130 void Board::print() const {
131
132 }
133
134 //TODO: finish the definition
135 void Board::pressDown() {
136 }
137
138 //TODO: finish the definition
139 void Board::pressLeft() {
140 }
141
142 //TODO: finish the definition
143 void Board::pressRight() {
144 }
145
146 //TODO: comment out when submit to Task C
147 //Board::~Board() {
148 //    for (int row = 0; row < numRows; row++) {
149 //        delete[] panel[row];
150 //        panel[row] = nullptr;
151 //    }
152 //
153 //    delete[] panel;
154 //    panel = nullptr;
155 //}
```