CISS240: Introduction to Programming
Assignment 11

Name: _____

Declare an array `x` of 10 integers; initialize all the values in `x` to 0. Continually prompt the user for an index value for integer variable `i` and an integer value `v` and set `x[i]` to `v`; the program then prints all the values in the array. This repeats until the user enters -1 for `i`.

Test 1.

```
index and new value: 0 4
array: 4 0 0 0 0 0 0 0 0 0
index and new value: 1 5
array: 4 5 0 0 0 0 0 0 0 0
index and new value: 2 6
array: 4 5 6 0 0 0 0 0 0 0
index and new value: -1
```

Test 2.

```
index and new value: 9 123
array: 0 0 0 0 0 0 0 0 0 123
index and new value: 5 456
array: 0 0 0 0 0 456 0 0 0 123
index and new value: 0 789
array: 789 0 0 0 0 456 0 0 0 123
index and new value: 9 321
array: 789 0 0 0 0 456 0 0 0 321
index and new value: -1
```

[Hint: Note that you should use two input statements for `i` and `v`, and not one. In other words, you should have this in your code

```
...
std::cin >> i;
...
std::cin >> v;
...
```

and not

```
...
std::cin >> i >> v;
...
```

Why? Because that will allow you to input `i` and then break the loop without prompting the user for `v`.

```
...
```

```
std::cin >> i;
break if i is -1
std::cin >> v;
...
```

On the other hand this code:

```
...
std::cin >> i >> v;
...
```

forces the user to enter a value for `v`.]

When given data points which do not form a smooth curve (example: stock price), one technique of smoothing the data points is to compute the **moving average**. The following is the computation of a moving average with a window size of 3, meaning to say that the averages are computed with three values.

Suppose the values are

$$a0 \qquad a1 \qquad a2 \qquad a3 \qquad a4 \qquad a5$$

Then the moving average (with the window size of 3) is

$$a0 \qquad a1 \qquad \frac{a0+a1+a2}{3} \qquad \frac{a1+a2+a3}{3} \qquad \frac{a2+a3+a4}{3} \qquad \frac{a3+a4+a5}{3}$$

For instance, if the data is

$$1.2 \qquad 2.5 \qquad 2.0 \qquad 3.4 \qquad 2.3 \qquad 2.6$$

then the moving average (with the window size 3) is

$$1.2 \qquad 2.5 \qquad 1.9 \qquad 2.6333 \qquad 2.5667 \qquad 2.7667$$

(example: $(1.2 + 2.5 + 2.0)/3$ is 1.9). Note that, for this case, the averaging process begins only at the third term; there are not enough terms to average for the first two.

Now for the programming assignment.

Write a program that prompts the user for 10 doubles and a window size, places the doubles in an array, and computes the moving average with the window size specified by the user; the window size refers to the number of terms to use in the computation of the average. Your program should output the moving averages with consecutive values separated by a space. You should display 4 decimal places in your output. The input from the user must be placed in a double array of size 10. Computations of the moving averages must be done with this array.

(There are other techniques to smooth data points.)

Test 1.

```
1 1 1 1 1 1 1 1 1 1
1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000 1.0000
```

Test 2.

```
1 2 3 4 5 6 7 8 9 10 4
1.0000 2.0000 3.0000 2.5000 3.5000 4.5000 5.5000 6.5000 7.5000 8.5000
```

(Note that the window size is 4. Therefore the fourth value in the array is 2.5000 because this is the value of $(1 + 2 + 3 + 4)/4.0$. Etc.)

You are given the following skeleton code which declares an array of 10 integers and prompts the user for 10 integers to be placed in the array. The program then prompts the user for an integer value for variable `j`.

```
// Name:
// File:

#include <iostream>

int main()
{
    const int SIZE = 10;
    int a[SIZE];

    for (int i = 0; i < SIZE; i++)
    {
        std::cin >> a[i];
    }

    int j = 0;
    std::cin >> j;

    // YOUR CODE HERE

    return 0;
}
```

Your program must read the value in the array at index `j` and then print the number of times this value appears **consecutively** in the array **starting at** index `j` going to the **right**. Read the test cases carefully before writing your program.

Test 1.

```
0 0 5 5 5 5 5 5 9 5 2
6
```

In this case the value in the array at index 2 is the value 5. Starting at index 2 and going to the right, there are six consecutive 5s.

Test 2.

```
5 5 5 5 5 5 5 5 9 5 2
6
```

The value printed is still 6 because although there are two 5s to the left of the 5 at index 3, they do not count.

Test 3.

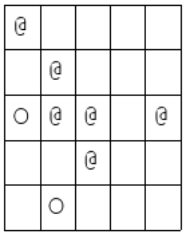
```
4 4 4 4 4 7 4 4 7 7 3
2
```

Test 4.

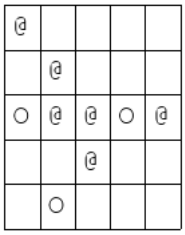
```
4 4 4 4 4 7 4 4 7 7 9
1
```

If you know the game Reversi, you will recognize that this program is part of the logic in a 1–dimensional version of Reversi. Before we talk about the program you need to write, let me first talk about Reversi.

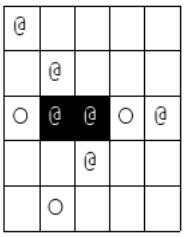
In the game of Reversi, if you put your piece on the board at an empty spot, you capture all the enemy pieces that form a straight line which are between the piece you put down and another one of your pieces. When a piece is captured, it becomes one of your pieces. Here's an example using a 5–by–5 Reversi board (the actual Reversi is larger). I use `@` for a black piece and `O` for a white piece. Here's the board:



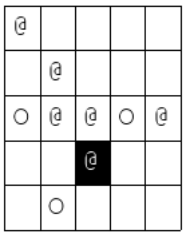
Now you put down your `O`:



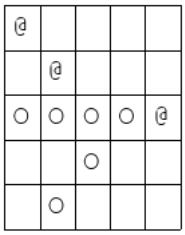
The `@` pieces which are captured becomes you pieces. For instance, the two `@` pieces shown are captured:



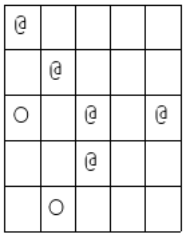
So is this one:



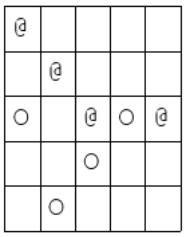
The captured pieces become yours so the board becomes:



Note that there must not be a gap between the pieces for the enemy pieces between two of your pieces to get captured. For instance, if the board looks like this:



Then putting your piece at the same spot will result in this:



The `O` piece to the left of the piece you put down does not capture the `@` on the left because there is a gap in-between. (Talk to me if you do not understand the rule of the game.)

For this question, you will write a program to perform only one step in the game and furthermore, this is a 1–dimensional version with 15 spots:

We will use an array of 15 integers to represent our 1–dimensional Reversi board. Furthermore, I will use zero (0) in the array to represent a blank space, 1 to represent a white piece and 2 to represent a black piece.

You are given the following skeleton code. It declares an array of 15 integers and then prompts the user for 15 integers to be placed into the array. The array models our 1–dimensional Reversi board.

The user (i.e., you) then enters an integer value into variable `s`. This is the index of the array where you want to put your piece which is 1 (i.e., a white piece).

You need to write code at the indicated place in the code provided below to model putting your piece on the board and capturing pieces if necessary.

The program ends by printing all the integers in the array.

Read the test cases carefully before programming.

```
// Name:
// File:

#include <iostream>

int main()
{
    int SIZE = 15;
    int a[SIZE];

    // YOUR CODE HERE

    return 0;
}
```

Test 1.

```
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
board: 000010000000000
index to put a 1: 4
board: 000010000000000
```

As you can see, at index 4, there is already a 1 (i.e., it's not a blank spot since it's not a 0.) This is therefore not a valid move.

Test 2.

```
0 0 0 0 2 0 0 0 0 0 0 0 0 0 0
board: 000020000000000
index to put a 1: 4
board: 000020000000000
```

This is similar to Test 1; it's not a valid move.

Test 3.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
board: 000000000000000
index to put a 1: 4
board: 000010000000000
```

In this case, the spot at index 4 is not taken. Therefore, your piece (i.e., 1) is placed at index 4. However, you did not capture any enemy piece.

Test 4.

```
0 0 0 0 0 2 1 0 0 0 0 0 0 0 0
board: 000002100000000
index to put a 1: 4
board: 000011100000000
```

In this case, you captured an enemy piece to the right (i.e., you captured a 2, which becomes your piece.)

Test 5.

```
0 0 0 0 0 0 0 0 2 2 2 1 0 0 0
board: 000000002221000
index to put a 1: 7
board: 000000011111000
```

In this case, you captured three enemy pieces to the right.

Test 6.

```
0 0 0 0 0 0 0 0 2 2 2 0 1 0 0 0
board: 000000022201000
index to put a 1: 6
board: 000000122201000
```

In this case, you did not capture any pieces since there is a 0 (i.e., empty space) between the 2s between the 1 you put down and the other 1 already on the board.

Test 7.

```
0 0 0 0 0 0 0 0 2 2 2 1 0 0 0
board: 000000002221000
index to put a 1: 6
board: 000000102221000
```

This is similar to Test 6.

Test 8.

```
0 0 0 0 0 1 2 2 2 2 0 0 0 0 0
board: 000001222200000
index to put a 1: 10
board: 000001111110000
```

In this case you captured four pieces to the left of the piece you put down.

Test 9.

```
0 0 0 1 0 2 2 2 2 0 0 0 0 0 0
board: 000102222000000
index to put a 1: 9
board: 000102222100000
```

In this case you did not capture any piece.

Test 10.

```
0 0 0 1 2 2 2 2 2 0 2 2 2 1 0
board: 000122222022210
index to put a 1: 9
board: 000111111111110
```

In this case you captured 5 pieces to the left and three pieces to the right of the piece you put down.

Test 11.

```
0 0 0 1 2 2 2 2 2 0 2 2 2 2 2
board: 000122222022222
index to put a 1: 9
board: 000111111122222
```

In this case you captured 5 pieces to the left.

Test 12.

```
0 0 0 0 0 0 0 0 0 0 0 0 2 2 2
board: 000000000000222
index to put a 1: 11
board: 000000000001222
```

In this case there are no captures because of obvious reasons. Make sure that this case works in the other direction as well.

Advice: Focus on capturing to the right first. Once you're done with that (and it's completely tested!), making it work for capturing to the left should be easy.

As mentioned before, an integer value (`int`) is actually limited in size. You can see this by running this program:

```
#include <iostream>

int main()
{
    int x = 1;

    for (int i = 0; i < 100; ++i)
    {
        x *= 10;
        std::cout << x << ' ';
    }

    return 0;
}
```

You can see from the output that when the integer becomes too big, it becomes negative. Specifically, on 32–bit machines, an `int` is usually from -2^{31} to $2^{31} - 1$, roughly -2 billion to 2 billion which is roughly 10 digits in length. The objective of this assignment is to simulate integers which are much larger than an `int`. We do this by using arrays. Many areas of computer science require extremely huge integers. For instance, the RSA cryptography uses integers which are several hundred digits long.

Write a program that performs addition for positive integers up to 1000 digits. Your program will prompt the user for the digits of the two integers to be added. The digits are entered separated by spaces. The input for each integer is terminated by entering -1. The (column) addition performed by the program shows the carries when they are nonzero.

[Hint: In your implementation, the digits for each integer is to be stored in an array. For instance, if the user entered

```
1 0 0 6 -1
```

(for the integer one thousand and six, i.e., 1006) and the name of the array is `x`, then

6 is to be stored in `x[0]`
0 is to be stored in `x[1]`
0 is to be stored in `x[2]`
1 is to be stored in `x[3]`

and all other `x[i]`'s are set to zero for `i` $= 4, \ldots, 999$. There should be a constant in your code for the maximum size of the array. (See your notes on arrays.) Note that in your array `x`, you have 1000 int values. But in the above example only 4 digits were used, i.e.,

`x[0], x[1], x[2], x[3]`

You should declare another `int` value, `x_len` (a length variable) and set it to 4. `x_len` represents the actual number of digits used to model the integer value of `x`. Note that the length of `x`, i.e., `x_len`, is different from the size of `x`. The size of `x` is 1000 (think of it as the capacity) but the length of `x` (think of it as the number of slots in the array that are actually used) is 4.]

(Note that with the length variable, the unused part of the array actually need not be set to 0.)

4 test cases are included. You are strongly advised to include more test cases.

Test 1.

```
1 0 0 6 -1
2 3 4 -1

    1
  1006
+  234
------
  1240
------
```

Test 2.

```
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 -1
1 -1

 11111111111111111111
  99999999999999999999
+                    1
----------------------
 100000000000000000000
----------------------
```

Test 3.

```
1 2 3 -1
9 9 9 9 9 9 9 9 9 9 9 9 9 5 9 9 9 9 9 9 -1

               111111
                  123
+ 99999999999995999999
-----------------------
  99999999999996000122
-----------------------
```

Test 4.

```
1 2 3 -1
0 -1

  123
+   0
-----
  123
-----
```

This shows you how to simulate large integers using arrays and also to provide the addition operation for your simulation of such large integers.

[Of course, for such a system to work effectively, you still need to write code to do multiplication, division, remainder, etc. All these will be dealt with in a future class.]

Write a program that prompts the user for two arrays of 1000 integers, say `x` and `y`, and prints the index in `x` where `y` occurs as a sequence for the first time. For instance, note that if

```
x: 3, 1, 6, 2, 8, 9, 6, 2, 4
y: 6, 2
```

we see that `y` occurs in `x` at index position 2 and index position 6:

```
x: 3, 1, 6, 2, 8, 9, 6, 2, 4
y: 6, 2
```

The program should print 2 in this case. On the other hand if

```
x: 3, 1, 6, 2, 8, 9, 6, 2, 4
y: 6, 8
```

the program should print -1 to indicate that `y` does not occur as a sequence within `x`. Make sure you read the test cases below carefully before diving into your code.

[When the arrays are arrays of characters, this problem is called the "substring problem"and is used extensively in areas such as bioinformatics and computational genetics. The first array is usually a DNA sequence and the second is a section of DNA that, for instance, might be a DNA sequence that indicates a dangerous mutation.]

The user enters the values for `x` and `y` by entering integers with the integer value -9999 as a sentinel to terminate data entry. Your program must allow a maximum size of 1000 for `x` and `y`.

Test 1.

```
3 1 6 2 8 9 6 2 4 -9999
6 2 -9999
2
```

Test 2.

```
3 1 6 2 8 9 6 2 4 -9999
6 8 -9999
-1
```

Test 3.

```
3 1 6 2 8 9 6 2 4 -9999
3 1 6 2 8 9 6 2 4 -9999
0
```

Test 4.

```
3 1 6 2 8 9 6 2 4 8 9 6 2 1 8 9 6 2 5 -9999
8 9 6 2 5 -9999
14
```

Test 5.

```
3 1 2 4 6 2 4 6 8 9 -9999
2 4 6 8 -9999
5
```

Test 6.

```
3 1 2 4 6 2 4 6 8 9 -9999
2 4 6 8 9 9 -9999
-1
```

Test 7.

```
1 2 3 4 5 6 7 8 -9999
1 3 5 7 -9999
-1
```

The goal here is to write a program that prompts the user for **two** sequences of numbers (think of two columns of numbers), and prints the sequence sorted (in ascending order) by the second column. Specifically, the program will continuously prompt the user for pairs of numbers (an `int` and a `double`) until an `int` of value -1 is entered. The program then sorts the data in ascending order by the **second number**, i.e., the double, and prints out the data.

For instance, if the user enters

```
1 -0.1 2 3.1 5 0.9 -1
```

the data entered basically model this table:

1	-0.1
2	3.1
5	0.9

and the program should sort the rows of this table by the second column to get

1	-0.1
5	0.9
2	3.1

Note that the pairs of data (the rows) move together as a unit.

This is of course very common in real–life applications. For instance, you can have a spreadsheet of employee data containing firstname, lastname, DOB, department, salary, etc. and you want to sort the table by salary.

Test 1.

```
1 -0.1 2 3.1 5 0.9 -1
1 -0.1
5 0.9
2 3.1
```

Test 2.

```
3 -0.7 9 12.1 21 -12.9 10 11.2 2 -3.2 -1
21 -12.9
2 -3.2
3 -0.7
10 11.2
9 12.1
```

Test 3.

```
9 12.1 0 2.0 108 -2.9 17 13.9 3 5.2 9 2.3 -1
108 -2.9
0 2
9 2.3
3 5.2
9 12.1
17 13.9
```

Recall that we already know how to check if an integer n is prime. The algorithm is basically the following:

- Test all integers d from 2 to $n-1$.
- If d divides n, then n is not prime.
- Otherwise, n is not divisible by any integer other than 1 and itself and therefore must be prime.

This method does work but in fact, some checks are redundant. For instance, if $n = 97$, there's no point in checking if 93 divides 97 : 93 is way too big to divide 97. In fact, instead of checking all d from 2 to $n-1$, it's enough to check for d from 2 to the square root of n.

Write a program that computes and stores the first $100,000$ primes in an array. Print the first 5 and the last 5 primes. Continuously get a user input `x` and an option `i`:

- If `i` is 0 (the number zero): perform a binary search for `x` in the array, and print report to the user denoting whether `x` is prime or not and notifying the user of it's position in the array.
- If `i` is 1: perform a prime factorization on `x` using your prime array.

The program ends when the user enters -1 for `x`. (You may assume all numbers will use only primes in the array. For instance, the user will not ask about a prime $>1,000,000$ or ask the program to factor a number that contains a prime $>1,000,000$)

Note: As stated above, you MUST use binary search for option 0. For option 1, you should not scan the prime array more than once (i.e., there should only be a single for–loop that runs through the prime table array once during the prime factorization.)

Test 1.

```
building prime table of 100,000 primes ...
primes: 2, 3, 5, 7, 11, ..., 1299647, 1299653, 1299673, 1299689, 1299709
2 0
2 is prime # 1
3 0
3 is prime # 2
4 0
4 is not a prime
5 0
5 is prime # 3
6 0
6 is not a prime
7 0
7 is prime # 4
631 0
631 is prime # 115
-1
```

(Note that 2 is prime # 1 and not prime # 0).

Test 2.

```
building prime table of 100,000 primes ...
primes: 2, 3, 5, 7, 11, ..., 1299647, 1299653, 1299673, 1299689, 1299709
1 1
1 = 1
2 1
2 = 2
3 1
3 = 3
4 1
4 = 2^2
5 1
5 = 5
6 1
6 = 2 * 3
7 1
7 = 7
8 1
8 = 2^3
9 1
9 = 3^2
10 1
10 = 2 * 5
11 1
11 = 11
12 1
12 = 2^2 * 3
100 1
100 = 2^2 * 5^2
-1
```

You're advised to include more test cases.

Spoilers Aheads!!! ...

Hint for option 1: Test if prime 2 divides `x`; if it does, keep dividing it by 2 to compute the largest power of 2 dividing `x`. Once you're done, move on the the NEXT prime 3. If 3 does not divide `x`, move on to the next prime 5. If 5 divides `x`, continuously divide `x` by 5 to compute the highest power of 5 dividing `x`. Etc. Therefore, the prime array is scanned only once.