CITY UNIVERSITY OF HONG KONG

Course code & title	:	CS2310 Computer Programming
Session	:	Semester B 2021/22
Time allowed	:	Two Hours

This paper has 8 pages (including this cover page).

1. This paper has **4** questions
2. Do not include another additional library
3. Download "Final.cpp" from Canvas, and re-name it using your student ID as "YourStudentID.cpp"
4. Write all your solutions in "YourStudentID.cpp"
5. Submit "YourStudentID.cpp" (.cpp file only) to Canvas

*This is a **close-book** examination.*

No materials/aids are permitted during the final exam. Candidates will be subject to disciplinary action if any unauthorized materials or aids are found on them.

Q1: [Programming Style and Basic Programming] (25%)

The formula f has two inputs: ***n*** (date type: **int**) and ***x*** (date type: **double**). The result of f is also **double** type. If ***n*** is an even number, the sign in front of ***n*** is "+"; otherwise, the sign is "−".

$$f = x^1/1 + x^2/2 - x^3/3 + x^4/4 - x^5/5 + \cdots \pm x^n/n$$

Write a program according to the following requirements:

- For the inputs of ***n*** and ***x***, we assume that they are already the **int-type** and **double-type** numbers, respectively, i.e., you do not need to check whether they are belonged to any other data types.

- For the input ***n***, check whether it is **positive**. If no, ask the user to input ***n*** again.
- Output the polynomial. The character ^ can be displayed by "^" in cout.

- Next, for the input ***x***, check whether $0.1 < x \leq 0.9$. If no, ask the user to input ***x*** again.

If both ***n*** and ***x*** are valid, use the **loop statements** to compute the value of f and output the result with only **three digits** in the **decimal** part.

Note 1: Your code should **NOT** use the function **pow(x,i)**.
Note 2: When you print the formula, there is **NO** space between any number and operator/sign

Example-1 (Input is underlined):

```
Input n: 3
x^1/1+x^2/2-x^3/3

Input x: 0.5
The result is: 0.583
```

Example-2 (Input is underlined):

```
Input n: -1
Input n: 0
Input n: 4
x^1/1+x^2/2-x^3/3+x^4/4

Input x: 0
Input x: 1
Input x: 0.9
The result is: 1.226
```

Example-3 (Input is underlined):

```
Input n: 1
x^1/1

Input x: 0.8
The result is: 0.800
```

Q2: [Program Development] (25%)

A **recursive function** is defined as follows: $f(0) = 1$, $f(1) = 2$, $f(2) = 3$, and

- $f(n) = 3 \times f(n-1) + 2 \times f(n-2) + f(n-3),\ if\ n \geq 3.$

Part (a): Write a program to conduct the following operations:

- Read a non-negative (*data type:* ***int***) integer n ($n \geq 3$) from an **input file** "input.txt" (this file contains one non-negative integer number only) and show it on the screen.
- Write a **recursive function** to calculate $f(n)$. Function's return type is **long long**.
- Output the result of $f(n)$ on the screen.

Part (b): For the result of $f(n)$, you need to further get each of its digits **sequentially** and store them in an integer (**int-type**) array using **dynamic memory allocation**. The array size should be the same as the number of digits in the result of $f(n)$. For example, if n is 16, the result of $f(n)$ is 259086967. The size of the array is nine and the digits should be stored as follows:

Digit:	2	5	9	0	8	6	9	6	7
Digit's Index:	0	1	2	3	4	5	6	7	8

Then, in this array, your program should further find the all the "**peak**" digits, i.e., the digit of index i is greater than its two neighbor digits of indices $i-1$ and $i+1$.

- No need to consider the first and last digit because they have only one neighbor.
- In the table above, peak digits are highlighted.

Print the following information on the screen:

- All the peak digit(s). If there is no peak digit in the array, output -1.
- The number of peak digits(s) being **primer numbers**. If there is no peak digit, output 0.

Note: If $\boldsymbol{a}$ is one prime number, $2 \leq \boldsymbol{a}$ and $\boldsymbol{a}$ has **NO** other positive factors between 1 and itself.

Part (c): Create a 2D dynamic array (*data type:* ***int***):

- The number of rows is n, obtained from "input.txt" in **Part (a)**, e.g., 16
- The number of columns is the number of digits of $f(n)$, obtained from **Part (b)**, e.g., 9

Then, initialize this 2D array as follows:

- The first row stores each digit of $f(n)$, from **Part (b),** sequentially, e.g., 2 5 9 0 8 6 9 6 7
- The second row stores the corresponding index of each digit, e.g., 0 1 2 3 4 5 6 7 8
- Each element in the rest row(s) equals to its row index plus column index

Print the **first two rows** and **the last one row** on the screen, and then clean up the 2D array.

Implement the **recursive function** as `f(…)` and all other parts above in `Q2()`. Note that

- **No need** to check the number's correctness from "input.txt" (downloaded from Canvas).
- **No need** to generate any output file. All the outputs are printed on the screen.

Example-1 (There is a space between two numbers for "Peak digit(s)" and the last three lines):

```
The number from the input file is: 16
The result of f(n) is: 259086967

Peak digit(s):
9 8 9
The number of prime peak(s): 0

The first two rows and the last row are:
2 5 9 0 8 6 9 6 7
0 1 2 3 4 5 6 7 8
15 16 17 18 19 20 21 22 23
```

Example-2 (There is a space between two numbers for "Peak digit(s)" and the last three lines):

```
The number from the input file is: 12
The result of f(n) is: 1496513

Peak digit(s):
9
The number of prime peak(s): 0

The first two rows and the last row are:
1 4 9 6 5 1 3
0 1 2 3 4 5 6
11 12 13 14 15 16 17
```

Example-3 (There is a space between two numbers for "Peak digit(s)" and the last three lines):

```
The number from the input file is: 21
The result of f(n) is: 162705378247

Peak digit(s):
6 7 5 8
The number of prime peak(s): 2

The first two rows and the last row are:
1 6 2 7 0 5 3 7 8 2 4 7
0 1 2 3 4 5 6 7 8 9 10 11
20 21 22 23 24 25 26 27 28 29 30 31
```

Example-4 (There is a space between two numbers for "Peak digit(s)" and the last three lines):

```
The number from the input file is: 4
The result of f(n) is: 50

Peak digit(s):
-1
The number of prime peak(s): 0

The first two rows and the last row are:
5 0
0 1
3 4
```

Q3: [Program Comprehension and Development] (25%)

Implement a class called `Candy`, which has four **private member variables**:

- `name[…]`: a `char` array, storing the candy name.
- `price`: an `int` variable, storing the candy's price.
- `month`: an `int` variable, storing the month of the candy's expire date.
- `day`: an `int` variable, storing the day of the candy's expire date.

The `Candy` class also has the following **public member functions**:

- `getName()`: returns the candy's name.
- `getPrice()`: returns the candy's price.
- `getMonth()`: returns the candy's expire month.
- `getDay()`: returns the candy's expire day.
- `set(char *n, int p, int m, int d)`: sets each member variable.

Part (a): Complete the class definition of `Candy` (Do **NOT** change the class name and do **NOT** add any extra members).

- Declare all the **member variables**.
- Candy's name is a **cstring** and contains **three** characters. Determine the array size of `name[…]`, which should be **just** large enough to store the candy name.

- Implement `getName()`, `getPrice()`, `getMonth()` and `getDay()` **inside the class directly**.
- Implement `set(…)` **outside** the class.

Part (b): In `Q3()`, we provide the information for ten candies, including their names, prices and expire dates. Declare an array to store 10 `Candy` objects, and use the information provided above to initialize each `Candy` object.

Next, implement a **non-member** function, `sortCandies(Candy cArr[])`. It sorts an array of `Candy` objects by using Bubble sort according to:

- Expire date, with both **month** and **day**, in an **ascending** order, e.g.,
  - Oct. 15 is smaller than Oct. 20, and
  - Sept. 15 is smaller than Oct. 20.
- We ignore **year** and consider **month** and **day only** in this question.
- If two candies have the **same** expire date, further sort them based on their price values in an **ascending** order.

Develop the `sortCandies(…)` function, implement the Bubble sort algorithm and print the sorted candies, as shown in the example.

Part (c): According to the sorted order of these 10 candy objects in Part (b), we obtain an array of price values $p[10]$, e.g., 88 51 69 25 73 90 48 55 19 68 in the example, denoted as:

$$p[0], p[1], p[2], \ldots, p[8], p[9],$$

Then, implement another **non-member** function, `check (Candy cArr[])`, to check whether this price array is ***special***, if it satisfies either one of two cases below:

- Case 1: $p[0] < p[1], p[1] > p[2], p[2] < p[3], p[3] > p[4], \ldots$
- Case 2: $p[0] > p[1], p[1] < p[2], p[2] > p[3], p[3] < p[4], \ldots$

In other words, If the array satisfies **one** of following two conditions, it is a **special** array.

- For Case 1, each **odd**-index element is greater than its neighbour(s). Because $p[9]$ has only one neighbor $p[8]$, we only need to consider $p[8] < p[9]$ for $p[9]$.
- For Case 2, each **even**-index element is greater than its neighbour(s). Because $p[0]$ has only one neighbor $p[1]$, we only need to consider $p[0] > p[1]$ for $p[0]$.

Implement the `check (Candy cArr[])` function to decide whether the price array is a special array or not.

Example:

```
Sorted candies (Name, Month-Day, Price):
(C03,1-1,88) (C04,3-1,51) (C01,5-1,69) (C02,7-20,25) (C09,7-
 20,73) (C05,8-15,40) (C06,9-15,48) (C08,10-15,38) (C07,10-
 15,45) (C10,10-20,39)

The price values from Part (b):
88 51 69 25 73 40 48 38 45 39
It is a special array
```

Q4: [Program Development] (25%)

Part (a): Consider an array (data type: **int**) consisting of only **0**s and **1**s. Given an index, write a program to find the **length** of the **longest** sequence with **only 1**'s that covers this given index. For example, considering an array of eight elements: 0, 1, 1, 1, 1, 0, 0, 1, if the input index is 3, the longest sequence is 4, as highlighted in the table below.

Array	0	1	1	1	1	0	0	1
Index	0	1	2	3	4	5	6	7

Note 1: Assume that the size of the array is no more than 20 and the input array size is valid.
Note 2: The index starts from 0.
Note 3: If the element itself on the input index is 0, output the length of the sequence as 0.

Example-1 (The input value is underlined):

```
Part(a):
Input the number of elements: 8
Input each element: 0 1 1 1 1 0 0 1
Input index: 3
Length of the sequence is: 4
```

Example-2 (The input value is underlined):

```
Part(a):
Input the number of elements: 7
Input each element: 0 1 0 1 0 0 1
Input index: 0
Length of the sequence is: 0
```

Part (b): Given a 2D **binary square** matrix filled with 0's and 1's, find the **largest** area with **only 1**'s that covers the input cell. We only consider cells along four directions (up, down, left and right).

	0	1	2	3	4	5	Column
0	1	0	1	1	1	1	
1	0	1	0	1	1	1	
2	0	1	1	0	1	1	
3	0	1	0	1	1	1	
4	1	1	1	0	0	1	
5	0	0	0	0	0	0	
Row							

For the example in this table, if we check the cell at the coordinate (2,1), i.e., Row index = 2 and Column index = 1, the largest area filled with 1s only are highlighted in the table above. The size of this area is 7, which is the number of cells contained in this area.

You can consider the following three steps to implement your solution. We provide a class called `Cell` used to store the *x* and *y* coordinates of a cell in the matrix to simplify the implementation.

(1) Store the coordinates of the initial input cell to the first element in a `Cell` array `cells[]`. Denote this cell in the original 2D matrix as *cur*.
(2) Check the neighbors of *cur* in the matrix for four directions (up, down, left, and right). If the value of the neighbor cell is 1 and this neighbor cell has not been checked before, then add the coordinates of this neighbor cell to `cells[]`.
(3) After finish checking all four directions for *cur*, treat the next cell after *cur* in `cells[]` as the new *cur* and repeat step (2), until all the cell coordinates stored in `cells[]` are checked and no new cell coordinates can be further added in. Then the number of cell coordinates stored in `cells[]` is the result we want.

Note 1: Assume that the size of each dimension is no more than 20 and the input array size is valid.
Note 2: The index starts from 0 for both rows and columns.
Note 3: If the value of the input cell is 0, output the area as 0.

Example-1 (The input value is underlined):

```
Part(b):
Input the array size: 4
Input each element:
1 0 1 1
1 1 0 1
0 0 1 1
1 0 0 0
Input coordinates x (row) and y (column): 0 3
The size of the area is: 5
```

Example-2 (The input value is underlined):

```
Part(b):
Input the array size: 6
Input each element:
1 0 1 0 0 1
1 1 1 0 0 1
0 0 0 0 0 1
1 0 0 0 0 0
1 1 0 0 0 1
0 0 1 1 1 1
Input coordinates x (row) and y (column): 5 5
The size of the area is: 5
```

--- END ---