VE280 (23FA) Exercise 3

Introduction

Not a moment of sorrow for the departure of Exercise 2; the immediately arriving deadline consists of recursion, function pointers, and enumerations.

Description

Zhixian falls in love with function pointers at the first sight. He wants to combine function pointers with recursion! Here is his plan:

- Goal: general recursion for functions with the signature below:

```
int recursionSample(int, int, int);
```

- To represent whether the recursion stops at certain step, define an `enum class` named `RecursionState`, with two members `Continue` and `Stop`. (*You can learn the reasons why `enum class` instead of `enum`, and usage of `enum class` here*)
- To represent the result of each recursion step, define a `struct` named `RecursionResult`, with an array of `int` of size 3 and a `RecursionState` inside.
- In order to generate `RecursionResult`, create two functions with the signatures below:

```
RecursionResult continueRecursion(int value1, int value2, int value3);

RecursionResult stopRecursion(int returnValue);
```

- Define a new *type* named `RecursionFunction`. It represents a function pointer that takes in three `int` and returns a `RecursionResult`. Assume all integers here are *non-negative*.
- In order to do recursion with function pointers, create a function with the signature below:

```
int recurse(RecursionFunction f, int initialValue1, int initialValue2, int
initialValue3);
```

- *IMPORTANT* To ensure that the task is not hard, the recursive call will only appear *once* in the `return` statement, and the recursive steps will be `independent`. This is known as *tail recursion*. Calculating fibbonacci number (one input version) will have several recursive calls. Calculating factorial (one input version) will have dependent recursive steps. Both are not tail recursion. Also, both have multiple inputs version, which belongs to tail recursion. Search online if you are interested.

The description above may be vague, so he provides one sample below:

Sample

```
#include <iostream>
#include "triint_recursion.h"

using namespace std;
```

```
RecursionResult factorial(int n, int acc, int _){
    if(n == 0){
        return stopRecursion(acc);
    }
    return continueRecursion(n-1, acc*n, _);
}

int main(){
    int result = recurse(factorial, 5, 1, 0);
    cout << result << endl;
}
```

Output:

```
120
```

Task

- Put `struct` definition, type definition, `enum class` declaration (like this: `enum class RecursionState;`) and all function declarations in `triint_recursion.h`, and implement the rest of functionality in `triint_recursion.cpp`.

Notes:

- Since the members of `RecursionResult` are not meant to be explicitly referred by the user, what they are do not matter. You can design your own `RecursionResult` as long as all functionality is correct.
- You can create `main.cpp` to test your code.
- Remember to add `-Wall` during compilation.
- The C++ standard used on JOJ this time is `c++17`.
- Do not change the order of parameters in the function declarations.
- As mentioned in the syllabus, there will be a few hidden test cases. But don't be so worried. The test cases are not meant to be tricky.

Submission

Zip your `triint_recursion.h` and `triint_recursion.cpp`, and submit to JOJ.