Part I

Templates

本次 OJ 所有题目将使用 C++17 标准编译

预计完成时间：2–3 小时

Problem A. Simple std::max

Descriptions

首先，请回忆类模板和函数模板的用法及使用场景，并做一个简单的练习：

编写一个函数模板 Max(a, b)，使得对于任何定义了 operator < 的类的两个实例，都能返回其中较大的一个。

Submitting

在 implement.h 中进行实现，压缩后提交即可。

Problem B. Constants as template parameters

Descriptions

一般来说，比较常见的是在尖括号中使用 T 来表示模板参数，如 template <typename T>；但常数（浮点除外）也可以作为模板参数：

```
template<typename T, int length>
class StaticVector {
public:
   T data[length];
};
```

如上述代码，length 作为编译时常量，可用于确定数组长度，这样就进一步增强了模板的泛化能力。

作为练习，请设计一个模板类 ClassWithFixedSize<int size>，使得：

sizeof (ClassWithFixedSize<size>) 恰好等于 size。

Submitting

在 implement.h 中进行实现，压缩后提交即可。

Problem C. Template specialization (I)

Descriptions

模板一般被用来将相同的逻辑应用到不同的类型上。那么能否针对某些类型提供差异性的展开呢?

```
template<typename T>
bool is_float(T _){
   return false;
}

template<>
bool is_float(float _){
   return true;
}
```

例如，上述模板函数 is_float 用于判断传入的参数类型是否为 float。如果传入的参数类型不是 float，则编译时会展开第一个模板，从而返回 false；反之会展开第二个模板，并返回 true。

作为练习，请编写一个模板函数 Equal<T> (T a, T b)，用于判断传入的两个参数是否相等。由于浮点数类型可能包含精度问题，在比较 float、double、long double 类型是否相等时，如果两数差的绝对值小于 1e-3，就认为两数相等，你的答案中需要考虑到这一点。

Submitting

在 implement.h 中进行实现，压缩后提交即可。

Tips

std::vector<bool> 是一个知名的模板特化实例。首先，它不是一个 STL 容器；其次，它存储的不是 bool，最好避免使用这个容器。

Problem D. Template specialization (II)

Descriptions

请参考上一题中关于模板特化的知识，编写泛型函数 is_same_type(T1, T2)，使得两个参数的类型相同时返回 true，不同时返回 false。

Submitting

在 implement.h 中进行实现，压缩后提交即可。

Tips

注：C++ STL 中其实已经实现了这个函数（std::is_same），而且只用了两行。

Problem E. Curiously recurring template pattern (CRTP)

Descriptions

继承是面向对象编程中常用的手段，可以使子类具有父类的功能。但遗憾的是，父类中的函数难以访问子类的成员。一种打破限制的方式是使用虚函数，但会带来时间和空间开销。

CRTP (又称静态多态) 是另一种解决此问题的思路。它不使用虚函数，而是使用模板将子类信息传递到父类。观察以下代码，class A 继承了 GetSizeProvider，并通过模板参数传递了自己的类型。若执行 A::get_size()，可以得到正确答案 123，说明 A 成功获得了父类的功能，且父类的函数成功访问到了子类的信息。

```
template<typename T>
class GetSizeProvider {
public:
  static unsigned get_size() {
    return sizeof(T);
  }
};

class A : public GetSizeProvider<A> {
  char arr[123];
};
```

在一周前的上机测试中，助教介绍了访问者模式，并使用了虚函数进行实现。本题对应的代码文件中提供了其简化版本，请使用 CRTP 补全该实现。

Submitting

在 implement.h 中进行实现，压缩后提交即可。

Thinking

在学习继承时，似乎规定过类不能继承自己，否则会构成循环继承。那么 class A : public GetSizeProvider<A> 是否违反了这一规定？为什么？

Exploration

LLVM 中的参考实现：

https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/IR/InstVisitor.h#L78

Problem F. Mixin

Descriptions

仔细观察上一题中的继承用法。在一般情况下（回忆课上的例子：图形-多边形-三角形），继承是为了在父类基础上添加新的功能。而在 Problem F 中，类 A 继承 GetSizeProvider 仅仅是为了从中获得 get_size() 接口的实现，这种方式被称为 Mixin（混合）。事实上，Mixins 反转了常规的继承方向，它允许在引入新的数据成员以及函数接口时，不必复制相关接口代码。

作为练习，在 main.cpp 中提供了一个简单的 class Book，它是一个支持了 Mixin 的变参类模板，可接受任意数量的类作为模板参数，并把这些类全部继承：

```
template<typename... Mixins>
class Book : public Mixins ... {
public:
    std::string name;
    ......
}
```

你的任务是补全 implement.h 中的 withLabel、withColor 和 withPrice 三个类，使得 main() 中的 book 具有正确实现的接口。

Submitting

在 implement.h 中进行实现，压缩后提交即可。

Problem G. Template meta programming

Descriptions

在编译期间，C++ 模板的某些特性可以和实例化过程相结合，从而产生一种 C++自己内部的原始递归的“编程语言”。这种技术被称为元编程（Meta Programming），属于函数式编程的一种，可用来在编译期计算某些复杂函数的值。

观察以下代码，factorial 模板类可用于计算阶乘。举例来说，若需要计算 10!的结果，只需读取 factorial<10>::value 的值，而 factorial<10>在模板展开时依赖于 factorial<9>的结果，依此类推……为了避免无限递归，我们将 factorial<0>的值指定为 1。你可以尝试更改 main() 中的模板参数，看看是否能计算出正确的值（小心溢出）。

```
template<int n>
struct factorial {
   static_assert(n >= 0, "n must >=0");
   static constexpr int value = factorial<n - 1>::value * n;
};

template<>
struct factorial<0> {
   static constexpr int value = 1;
};

int main() {
   cout << factorial<10>::value;
}
```

作为练习，请编写类模板 fibnacci<int n>，使得 fibnacci<n>::value 的值为斐波那契数列的第 n 项。假设数列的第 0 项为 0，第 1 项为 1。

Submitting

在 implement.h 中进行实现，压缩后提交即可。

Exploration

本题演示了简单的值元编程（Value Metaprogramming），在编译期进行数学计算；另一种元编程是类型元编程（Type Metaprogramming），用于获取或改变类型的性质（关键词：类型萃取）。这两种技术经常在 C++ STL 的实现中被混合使用。

Part II

Standard Template Library and its applications

Problem H. How to use stdin/stdout

Descriptions

由于接下来的题目将使用标准输入输出，这里先练习如何从标准输入读入，并输出到标准输出。对于 C++语言，从 stdin 读入可使用 cin、scanf 或 fread 等；输出到 stdout 可使用 cout、printf 或 fwrite 等。

作为练习，请先读入一个正整数 n（n<100），接下来读入 n 个整数；随后输出这 n 个整数的和。

Input

一个正整数 n，随后是 n 个整数

Output

一个整数，即 n 个整数的和

Input Example

5
1 2 3 4 5

Output Example

15

Submitting

将代码压缩后提交即可，压缩包内至少应含有 main.cpp。

Problem I. Two priority_queues

Descriptions

给定一个长度 n 的集合 S，并进行 q 次操作，操作共有两种：

(1) 向集合中添加一个数

(2) 询问集合的“绝对中位数”

注：绝对中位数，即集合中元素升序排序后，下标为$\lceil \frac{\text{len}}{2} - 1 \rceil$的数，其中 len 为集合的大小。例如排序后为[1,2,3]时，绝对中位数为 2；排序后为[1,2,3,4,5,6]时，绝对中位数为 3.

Input

第一行包括两个整数 n, q 表示序列的长度和操作次数$(1 \le n, q \le 1e5)$
第二行包括 n 个整数，表示集合 S
接下来 q 行每行表示一个操作：
对于 (1) 操作，输入格式为“1 x”，x 为加入的数字
对于 (2) 操作，输入格式为“2”，表示查询绝对中位数

Output

对于每个 (2) 操作，输出一个整数，即序列的绝对中位数。

Input Example

```
5 6
4 2 1 3 5
2
1 6
1 1
2
1 1
2
```

Output Example

```
3
3
2
```

Submitting

将代码压缩后提交即可，压缩包内至少应含有 main.cpp。

Hint

q 次排序或使用 nth_element()不是本题的最优做法，本题可以在 $O(q \log n)$复杂度内完成。

Problem J. Sparse vector

Descriptions

在科学计算中，向量（Vector）是表示信息的常见方式。对于一个 n 维整数向量 $v \in Z^n$，如果 v 仅在少量维度上的取值不为 0，则称其为稀疏向量。在机器学习的数据预处理阶段，许多向量都是稀疏的，如果可以适当地压缩他们，则可以节省大量空间。

本题要求实现两个稀疏向量 u 和 v 的内积乘法（保证两个向量的维数相同）。

Input

第一行包含用空格分隔的三个正整数 n, a, b，其中 n 表示两个向量的维数，a, b 分别表示两个向量所含非零值的个数 ($0 < n \leq 1e9, 0 < a \leq n, 0 < b \leq n$)。

接下来 a 行为向量 u 的稀疏表示，即每行包含用空格分隔的两个数 index 和 value，表示在 index 维度上有不为零的值 value。

接下来 b 行为向量 v 的稀疏表示，含义同上。

Output

输出到标准输出，只需输出一个整数，表示向量 u, v 的内积 $u \cdot v$。

Input Example

```
10 3 4
4 5
7 -3
10 1
1 10
4 20
5 30
7 40
```

Output Example

```
-20
```

Submitting

将代码压缩后提交即可，压缩包内至少应含有 main.cpp。

Problem K. std::tuple

Descriptions

众所周知，C++ 中的函数只能返回一个类型。当有些函数需要向外部返回多个变量时，一般采用在参数中传递指针/引用，或是单独声明一个返回值结构体的方式。但这样做可能使得函数结构变得更复杂。

std::tuple<...>是 std::pair 的扩展版， 可以将多个不同类型的值打包成一个。例如在 get_book_info() 函数中，计划向外部提供 book 的 name、color、price 等信息，这时只需使用 return {name, color, price}; 即可返回一个 std::tuple<string, Color, int>对象，其包含所有的信息。

而在外部，只需使用 auto [name, color, price] = get_book_info(); 即可将三个对象解包。

作为练习，implement.h 中的 div_int 函数用于返回整数除法的商和余数，请补全空缺的代码片段，使程序正常运行。

Submitting

在 implement.h 中进行实现，压缩后提交即可。

Problem L. std::optional

Descriptions

在使用 C++进行编程时，经常会遇到以下情况：某函数在正常情况下应返回一个对象，但如果出现错误，则无法返回。针对这种情况，我们一般有以下几种处理方式：

1. 返回指针（或使用 auto_ptr），这样就可以使用 nullptr 来表示出错；
2. 返回“-1”，但如果返回对象的所有空间被充分用于表示信息，可能找不到类似的表示；
3. 使用异常，在出错时 throw Exception，在外部使用 try-catch 进行捕获，但会影响性能；
4. 将原来要返回的对象放到参数中作为引用传递，并将一个独立的返回值作为成功与否的状态标志返回。

以上几种方式都不可避免地增加了代码复杂性，或降低了程序性能。而 C++ 17 新引入的 std::optional 可以进一步简化这个问题：

```
optional<int> div_int(int dividend, int divisor) {
    if (divisor != 0) {
        return {dividend / divisor};
    }
    return {}; // no return value
}

int main() {
    auto res = div_int(20, 1);
    if (res) {
        cout << "Quotient: " << res.value();
    } else {
        cout << "Can't divide.";
    }
}
```

如以上代码所示，std::optional<int>可用来承载一个 int 对象，但也可能为空。通过判断 has_value() 的值或是直接转换为 bool 后的值，可以得知 optional 是否包含了一个可用的对象，具体被包含对象的值可以通过 value()来获取。

作为练习，请实现一个 read_file(const string& filename)函数，用于将指定文件中的数据全部读出；考虑到文件可能不存在，请在出错时返回空的 optional 对象。

Submitting

在 implement.h 中进行实现，压缩后提交即可。

Discussion

https://zhuanlan.zhihu.com/p/251306766