

基础知识 2

#计算机1024转换问题

```
#include <iostream>
#include <algorithm>
#include <cmath>
typedef long long ll;
using namespace std;
int main()
{
    // 请在此输入您的代码
    ll T; cin >> T;
    while(T -- )
    {
        ll x, y;
        string op;
        cin >> x >> op >> y;

        if(op == "B")
        {
            cout << x / y << endl;
        }
        else if(op == "KB")
        {
            cout << (x * 1024) / y << endl;
        }
        else
        {
            cout << (x * 1024 * 1024) / y << endl;
        }

    }
    return 0;
}
```

在计算机科学中，通常使用二进制的倍数来表示存储容量和数据传输速度。常见的二进制倍数有：

- 1 字节 (Byte) = 8 比特 (bit)
- 1 千字节 (KB) = 1024 字节
- 1 兆字节 (MB) = 1024 KB
- 1 吉字节 (GB) = 1024 MB
- 1 太字节 (TB) = 1024 GB

这种以 1024 为基数的二进制倍数是计算机领域的标准，因此在处理存储容量时经常使用 1024 的幂次方。在你的代码中，根据输入的操作类型 "B"、"KB"、"MB"，选择不同的换算系数。例如：

- 如果操作类型是 "B"，直接输出 `x / y`，表示字节数。
- 如果操作类型是 "KB"，输出 `(x * 1024) / y`，将字节数转换为千字节 (KB)。
- 如果操作类型是 "MB"，输出 `(x * 1024 * 1024) / y`，将字节数转换为兆字节 (MB)。

这种以 1024 为基数的二进制倍数换算在计算机存储和传输领域是常见的。

输出格式

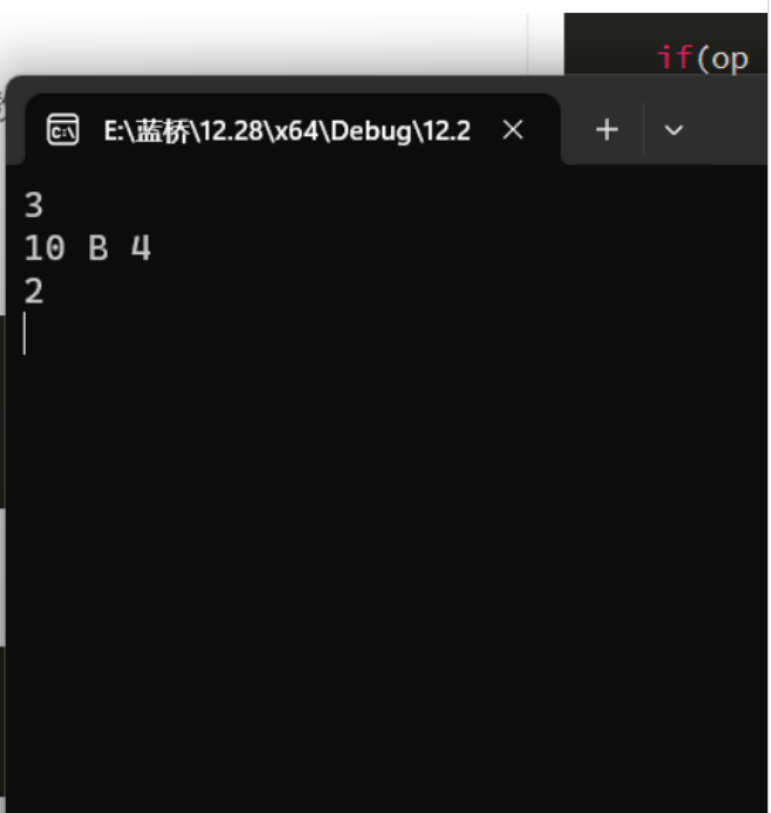
对于每组测试数据，输出一行一个整数，表示 `p` 类型数组的最大长度。

样例输入

```
3
10 B 4
1 KB 5
2 MB 6
```

样例输出

```
2
204
349525
```



```
if (op == 'B') {
    p = x / y;
} else if (op == 'KB') {
    p = (x * 1024) / y;
} else if (op == 'MB') {
    p = (x * 1024 * 1024) / y;
}
```

运行结果这样也可以 只要结果对就行

#逻辑运算符

1. `++x`: 先将 `x` 的值加 1, 然后取得增加后的值。
2. `&&` (逻辑与): 如果左侧表达式为真, 才会执行右侧表达式, 否则不执行。
3. `y--`: 先取得 `y` 的当前值, 然后将 `y` 的值减 1。
4. `||` (逻辑或): 如果左侧表达式为假, 才会执行右侧表达式, 否则不执行。
5. `++z`: 先将 `z` 的值加 1, 然后取得增加后的值。

根据上述解释, 这个表达式的执行顺序是从左到右, 而且涉及逻辑与和逻辑或, 具体的执行过程如下:

- 如果 `++x` 的结果为真 (非零), 则执行 `y--`, 并且不执行 `++z`。
- 如果 `++x` 的结果为假 (0), 则执行 `++z`, 并且不执行 `y--`。

这是因为逻辑与 (`&&`) 的优先级高于逻辑或 (`||`), 而且在 C++ 中逻辑运算符具有短路性质。

请注意, 前置递增/递减运算符会直接影响变量的值, 而后置递增/递减运算符会返回原始值, 然后再进行递增/递减操作。

注意

1. Pascal 使用 `integer` 会爆掉哦!
2. 有负数哦!
3. C/C++ 的 main 函数必须是 `int` 类型, 而且 C 最后要 `return 0`。这不仅对洛谷其他题目有效, 也是 NOIP/CSP/NOI 比赛的要求!

好吧, 同志们, 我们就从这一题开始, 向着大牛的路进发。

任何一个伟大的思想, 都有一个微不足道的开始。

为了计算从点 $(3, 6)$ 到点 $(7, 9)$ 的最短路径数量，我们可以使用之前提到的组合数的公式：

$$C(m+n, m) = \binom{m+n}{m} = \frac{(m+n)!}{m! \cdot n!}$$

在这里， m 表示横向的步数， n 表示纵向的步数。对于这个问题，从 $(3, 6)$ 到 $(7, 9)$ ，横向的步数 $m = 7 - 3 = 4$ ，纵向的步数 $n = 9 - 6 = 3$ 。

因此，最短路径数量为：

$$C(4+3, 4) = C(7, 4) = \frac{7!}{4! \cdot 3!} = \frac{5040}{24 \cdot 6} = 35$$

所以，从点 $(3, 6)$ 到点 $(7, 9)$ 的最短路径数量为 35 种。这表示有 35 种不同的方式可以从起点到达目标点，每种方式都是最短路径。

"DP" 是 "Dynamic Programming" 的缩写，中文翻译为"动态规划"。动态规划是一种解决多阶段决策问题的数学方法，通常用于优化问题。在计算机科学中，动态规划常用于解决需要在多个阶段作出决策的问题，通过记录子问题的解，避免重复计算，提高算法效率。

动态规划的基本思想是将一个复杂问题分解成若干个重叠子问题，先求解子问题，然后将子问题的解记录下来，避免重复计算。这样，通过递归或迭代，可以最终求解原问题。

在编程中，动态规划常常用数组或表格来存储子问题的解，从而在计算过程中避免重复的计算。

下面简要介绍一些动态规划的特点和思路：

1. **最优子结构 (Optimal Substructure)**：问题的最优解可以通过子问题的最优解来构造。
2. **重叠子问题 (Overlapping Subproblems)**：问题可以分解成子问题，而子问题之间可能有重复的部分，这时可以通过记录已解决的子问题的结果来避免重复计算。
3. **状态转移方程 (State Transition Equation)**：定义子问题的解和原问题之间的关系，形成状态转移方程。
4. **自底向上或自顶向下的计算**：动态规划可以采用自底向上的迭代方式或自顶向下的递归方式进行计算。

1. 时间复杂度 (Time Complexity)：

- 时间复杂度描述的是算法运行时间与输入数据规模之间的增长关系。它表示算法的运行时间随着输入规模的增加而增加的趋势。
- 时间复杂度用大O符号表示，例如 $O(n)$ 、 $O(n^2)$ 、 $O(\log n)$ 等。其中， $O(n)$ 表示线性时间复杂度， $O(n^2)$ 表示平方时间复杂度， $O(\log n)$ 表示对数时间复杂度等。
- 时间复杂度描述的是运行时间与输入规模之间的关系，而不是具体的运行时间。

2. 空间复杂度 (Space Complexity)：

- 空间复杂度描述的是算法在运行过程中需要的内存空间与输入规模之间的增长关系。它表示算法的内存使用随着输入规模的增加而增加的趋势。
- 空间复杂度同样用大O符号表示，例如 $O(1)$ 、 $O(n)$ 、 $O(n^2)$ 等。其中， $O(1)$ 表示常数空间复杂度， $O(n)$ 表示线性空间复杂度， $O(n^2)$ 表示平方空间复杂度等。
- 空间复杂度描述的是内存使用与输入规模之间的关系，而不是具体的内存使用量。

在进行算法分析时，我们通常关注最坏情况下的时间复杂度和空间复杂度。最坏情况下的时间复杂度表示算法在最不利情况下的运行时间，而空间复杂度则表示算法在最不利情况下的内存使用。

总体来说，好的算法应该具有较低的时间复杂度和空间复杂度，以便在处理大规模数据时能够高效运行，并节省计算资源。

在冒泡排序中，我们对学生的平均成绩进行比较和交换，以便按照平均成绩从高到低进行排序。平均成绩是与学生信息——对应的，因为每个学生都有自己的平均成绩。

考虑以下情况：如果我们只交换了学生信息而没有交换平均成绩，那么学生信息数组 `arr` 中的学

生与平均成绩数组 `average` 中的成绩就不再匹配。这就会导致排序的结果不正确，因为输出的学生信息与平均成绩不匹配。

为了保持正确的一一对应关系，我们在进行学生信息的交换时，同时也要交换对应的平均成绩。这样，最终输出的学生信息和平均成绩就是正确匹配的。这确保了排序结果的正确性。

简而言之，平均成绩和学生信息是一一对应的，如果我们改变了学生信息的顺序，也必须相应地改变平均成绩的顺序，以保持它们之间的关联性

```
#include <iostream>
#include <string>

struct card
{
    int num;
    std::string name;
    double score_c, score_m, score_e;
};

int main()
{
    card arr[5];
    for (int i = 0; i < 5; i++)
    {
        std::cin >> arr[i].num >> arr[i].name >> arr[i].score_c >> arr[i].score_m
        >> arr[i].score_e;
    }

    double average[5];
    for (int j = 0; j < 5; j++)
    {
        average[j] = (arr[j].score_c + arr[j].score_m + arr[j].score_e) / 3;
        std::cout << average[j] << " ";
    }

    // 冒泡排序
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < 4 - i; j++)
        {
            if (average[j] < average[j + 1])
            {
                // 交换两个学生的信息
                card temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}
```

```

        // 交换平均成绩
        double temp_avg = average[j];
        average[j] = average[j + 1];
        average[j + 1] = temp_avg;
    }
}

// 输出排序后的结果
for (int i = 0; i < 5; i++)
{
    std::cout << arr[i].num << "\t" << arr[i].name << "\t" << arr[i].score_c <<
"\t" << arr[i].score_m << "\t" << arr[i].score_e << std::endl;
}

return 0;
}

```

`while(y--<0)`

在这个循环中，先判断在输出是减减

如果给定三角形的顶点坐标为 (x_1, y_1) , (x_2, y_2) , 和 (x_3, y_3) , 那么可以使用以下公式计算三角形的面积:

$$\text{Area} = \frac{1}{2} |x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)|$$

这是三角形面积的一般公式, 称为行列式法。其中, $|a|$ 表示取绝对值。

通过这个公式, 你可以计算给定三角形的面积。如果面积为正, 则表示这是一个非退化三角形。

具体的 C++ 代码示例如下:

cpp

Copy code

```
#include <iostream>
#include <cmath>

double calculateTriangleArea(double x1, double y1, double x2, double y2, double x3, double y3) {
    return 0.5 * std::abs(x1 * (y2 - y3) + x2 * (y3 - y1) + x3 * (y1 - y2));
}

int main() {
    double x1, y1, x2, y2, x3, y3;

    // 输入三角形的顶点坐标
```


在C++中, `int argc` 和 `char *argv[]` 是用于从命令行接收参数的标准形式。这两个参数用于获取命令行中传递给程序的参数数量和参数值。

- `argc` (argument count) 表示命令行参数的数量, 包括程序名称。即, `argc` 是传递给程序的命令行参数的个数, 至少为1 (程序名称本身算一个参数)。
- `argv` (argument vector) 是一个指针数组, 存储了命令行参数的值。`argv[0]` 存储程序的名称, 而 `argv[1]`、`argv[2]` 等存储传递给程序的其他参数。这些参数都是以字符串形式传递的。

在你的代码中, `int argc` 和 `char *argv[]` 是 `main` 函数的参数, 但在代码中并没有使用它们。通常, 当你不需要从命令行接收参数时, 可以简单地写成 `int main()`, 而不用关心 `argc` 和 `argv`。

如果你希望从命令行传递参数, 并且使用了这两个参数, 你可以在命令行中运行程序时输入额外的参数, 例如:

```
bash
```

[Copy code](#)

```
./your_program arg1 arg2 arg3
```

其中 `arg1`、`arg2`、`arg3` 等就是命令行参数, 它们会被存储在 `argv` 数组中。

#调试输入

```
1,2,3,4,5,6,7,8
```

```
3
```

输出数据

```
data is error [1,2,3,4,5,6,7,8]
```

```
The square brackets at both ends of the input data are abnormal.
```

和相似点

✓ Accepted

输入数据

```
[1,2,3,4,5,6,7,8]
3
```

输出数据

```
1,2,3,
```

期望答案

```
1,2,3,
```

#尾后迭代器

在C++的标准库中，尾后迭代器（`end iterator`）是一个迭代器，它指向容器中最后一个元素的下一个位置，表示遍历容器的末尾。在大多数情况下，尾后迭代器的使用是为了标识一个范围的结束，而不是指向有效的元素。

考虑一个示例：如果有一个数组 `arr`，其元素为 `[1, 2, 3, 4, 5]`，那么 `arr.end()` 就是指向数组最后一个元素的下一个位置，即指向数组的末尾之后的位置。

在C++的标准容器中，比如 `std::vector`、`std::list`、`std::map` 等，使用 `end()` 函数返回一个尾后迭代器。这是一个方便的方式来表示容器中元素的结束位置。例如：

```
#include <iostream>
#include <vector>
```

```
int main() {
    std::vector<int> vec = {1, 2, 3, 4, 5};

    // 使用迭代器遍历容器
    for (auto it = vec.begin(); it != vec.end(); ++it) {
        std::cout << *it << " ";
    }

    // 或者使用范围-based for 循环
    for (const auto& element : vec) {
        std::cout << element << " ";
    }

    return 0;
}
```

在这个例子中，`vec.end()` 指向 `vec` 中最后一个元素的下一个位置。在循环中，我们检查迭代器是否等于 `vec.end()`，如果等于，表示已经到达容器的末尾，循环结束。
相当于寻找最后一个元素的下一个元素 就是空白

```
/** * @param stu_info: A dictionary * @param stu_id: An intager * @return: A string
*/ string getStuName(map<int, string> &stu_info, int stu_id) {
if(stu_info.find(stu_id) == stu_info.end())
{ return "no this man"; }else{ return stu_info[stu_id]; } }
```

在这个代码中在这段代码中，`map<int, string> &stu_info` 是一个参数，表示一个引用类型的 `std::map` 对象。这个 `std::map` 是一个关联容器，它将整数（`int`）映射到字符串（`string`）。`stu_info` 是一个引用，意味着它并不是对实际 `std::map` 对象的拷贝，而是直接引用了传递给函数的原始对象
`&stu_info`：`&` 符号表示引用，将 `stu_info` 声明为一个对 `std::map<int, string>` 类型的引用。这意味着函数中对 `stu_info` 的任何修改都会影响到调用该函数时传递给它的原始 `std::map` 对象。

在C++的`std::map`中，`end()`是一个成员函数，返回指向容器中最后一个元素之后位置的迭代器。这个位置是一个尾后迭代器，指示没有定义的元素。

在这里，`stu_info.end()`用于表示`std::map`中的一个迭代器，指向尾后位置。在条件语句`if(stu_info.find(stu_id) == stu_info.end())`中，它与`find`函数一起使用，以检查给定的`stu_id`是否在`stu_info`中。

具体来说，`stu_info.find(stu_id)`函数尝试在`stu_info`中查找键为`stu_id`的元素。如果找到了，它返回一个指向该元素的迭代器；如果没有找到，它返回`stu_info.end()`，即尾后迭代器。因此，上述条件语句检查给定的`stu_id`是否在`stu_info`中，如果找到了，条件成立，否则条件不成立。

```
/** * @param stu_info: A dictionary * @param stu_id: An intager * @return: A string */ string getStuName(map<int, string> &stu_info, int stu_id) { auto itr = stu_info.find(stu_id); if(itr== stu_info.end()){ return "no this man"; } else{ return itr->second; } }
```

``auto itr`` 是一种 C++11 引入的自动类型推导语法。在这里，``auto`` 关键字被用于声明一个变量，而变量的类型将由编译器自动推导，无需显式指定。

具体地说，``itr`` 是一个迭代器，通过 ``auto`` 关键字告诉编译器根据赋值语句右侧的类型进行推导。在这里，``find`` 函数返回的类型是 ``std::map<int, string>::iterator``，即 ``stu_info`` 的迭代器类型。

使用 ``auto`` 可以简化代码，特别是在处理模板类型、迭代器等场景下，能够更灵活地适应不同类型。在这段代码中，``auto itr = stu_info.find(stu_id);`` 的效果就相当于显式写出 ``std::map<int, string>::iterator itr = stu_info.find(stu_id);``。

总的来说，``auto`` 的使用使得代码更加简洁，减少了繁琐的类型声明，同时提高了代码的可读性。

在这段代码中相当于 先将要寻找的那个代码找到
然后再在整个内容中进行寻找
因为本身find就是一个迭代器中所特有的
所以使用时运用了迭代器

``auto p : prices`` 是 C++11 引入的范围-based for 循环语法，用于遍历容器中的元素。

在这里，``prices`` 是一个容器（例如 ``std::vector<int>``），``auto`` 关键字用于自动推导容器中元素的类型，而 ``p`` 则是遍历过程中每个元素的临时变量，它的类型将根据容器元素的类型进行自动推导。

具体而言，这个循环的含义是：对于容器 ``prices`` 中的每个元素，将其依次赋值给变量 ``p``，然后执行循环体中的代码。在这段代码中，``p`` 就是 ``prices`` 中的每个元素的值。

下面是一个简单的例子，演示了如何使用范围-based for 循环遍历一个 ``std::vector<int>`` 容器

代码：

```
#include <iostream>
#include <vector>

int main() {
    std::vector<int> prices = {10, 20, 30, 40, 50};

    // 使用范围-based for 循环遍历容器
```

```

    for (auto p : prices) {
        std::cout << p << " ";
    }

    return 0;
}

```

上述代码会输出：`10 20 30 40 50`。这里，`auto p` 的类型会被推导为 `int`，因为 `prices` 是一个存储整数的容器

#int大最大值和最小值的表达

`INT_MAX` 是 C++ 中 `<climits>` 头文件中定义的一个宏，表示 `int` 类型的最大值。它是一个系统定义的常量，代表了 `int` 类型可以表示的最大整数值。在大多数系统中，`INT_MAX` 的值是 2147483647。

在这段代码中，作者使用 `INT_MAX` 初始化了两个变量 `fi` 和 `se`。这两个变量的目的是用来追踪数组 `prices` 中的最小值和次小值。通过将它们初始化为 `INT_MAX`，确保了第一个遍历到的元素会成为初始的最小值。

值得注意的是，在实际使用中，你可能会看到 `INT_MAX` 被用于表示整数的最大值，而 `INT_MIN` 表示整数的最小值。这两者都是系统定义的常量，可以在 `<climits>` 头文件中找到。

#寻找两个最小的数

```

class Solution { public: int buyChoco(vector<int>& prices, int money)
{ int fi = INT_MAX, se = INT_MAX; for (auto p : prices)
{ if (p < fi) { se = fi; fi = p; }
  else if (p < se) { se = p; } }
  return money < fi + se ? money : money - fi - se; }
};

```

在这个代码中相当于不用输出一啥退出循环

利用容器

就可以找到两个最小值

1. ``for (auto p : prices) { ... }``：使用范围-based for 循环遍历数组 ``prices`` 中的每个元素。

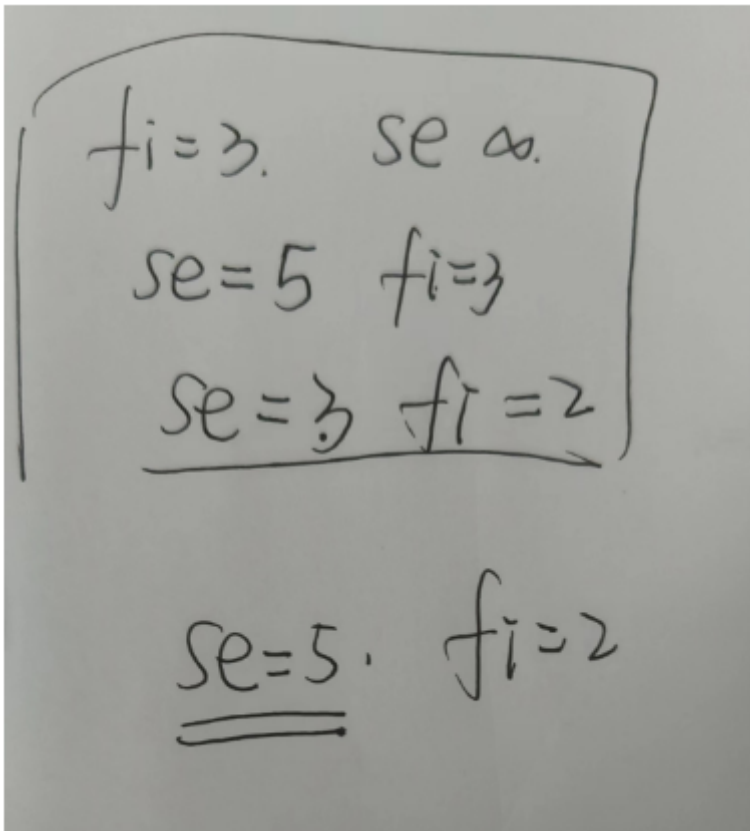
2. ``if (p < fi) { se = fi; fi = p; }``：如果当前元素 ``p`` 小于当前的最小值 ``fi``，则将 ``fi`` 的值赋给 ``se``，然后将 ``p`` 赋给 ``fi``，更新最小值。

3. ``else if (p < se) { se = p; }``：如果当前元素 ``p`` 大于等于 ``fi`` 但小于当前的次小值 ``se``，则将 ``p`` 赋给 ``se``，更新次小值。

4. ``return money < fi + se ? money : money - fi - se;``：最后，计算最小值和次小值的和 ``fi + se``。如果这个和小于等于给定的 ``money``，则返回这个和；否则，返回 ``money - fi - se``。

最后大三目运算 简直了 太牛逼了 想不到啊

下面这个图片是为了推到`se=fi`;
如果没有这行代码会发生的事



#迭代器的使用之一

`vector<int>& prices`` 是一个函数参数的声明，用于传递一个整数向量（``vector<int>``）的引用给函数。这意味着在函数内部对这个引用进行的任何修改都会影响到调用函数时传递的原始向量。

具体来说：

- ``vector<int>``：表示一个整数向量，是 C++ 标准库中的容器，可以存储一系列整数。
- `&`：表示引用。通过引用传递参数，可以避免函数对原始向量进行拷贝，从而提高效率。引用使得函数能够直接访问和修改原始向量的内容。
- ``prices``：是向量的名字（变量名），这里表示被传递的整数向量。

#迭代器遍历

```
/**  
 * @param stu_info: 一个字典  
 * @param stu_id: 一个整数
```

```

* @return: 一个字符串
*/
string getStuName(map<int, string> &stu_info, int stu_id)
{
    // 使用迭代器遍历map
    for (map<int, string>::iterator it = stu_info.begin(); it != stu_info.end();
it++)
    {
        // 检查当前迭代器的键是否与提供的学生ID匹配
        if (stu_id == it->first)
        {
            // 如果找到匹配, 则返回相应的姓名
            return it->second;
        }
    }

    // 如果未找到匹配项, 则返回字符串"no this man"
    return "no this man";
}

```

****初始化迭代器: ****

```
map<int, string>::iterator it = stu_info.begin();
```

这行代码初始化一个名为`it`的迭代器, 并将其指向`stu_info`的第一个元素。在这里, `stu_info.begin()`返回指向映射的第一个元素的迭代器。

****循环条件: ****

```
it != stu_info.end();
```

****迭代器递增****

```
it++
```

#队列

```

/**
* @param q: 一个队列
* @return: 一个整数
*/
int getQueue3(queue<int> &q) {
    // 如果队列中的元素不足三个, 返回 -1
    if(q.size() < 3) {
        return -1;
    } else {
        // 弹出队列的前两个元素
        q.pop();
        q.pop();
        // 返回队列的第三个元素的值
        return q.front();
    }
}

```

```
}
```

在 C++ 的标准库中，`std::queue` 类型也有 `size` 方法，该方法返回队列中当前的元素个数。因此，理论上，你可以在任何时候使用 `size` 方法来获取队列中的元素个数。

然而，需要注意的是，在多线程环境下，如果同时有其他线程在修改队列的大小，那么由于没有使用额外的同步机制，`size` 方法的返回值可能不是实时的，可能存在并发问题。在多线程环境下，你可能需要考虑采取适当的同步机制来确保操作的正确性。

在单线程环境下，使用 `size` 是安全的，你可以随时使用它来获取队列的当前大小。在这个特定的 `getQueue3` 函数中，作者使用 `q.size() < 3` 来检查队列中是否至少有三个元素，以避免访问队列中不存在的元素

在这个上下文中，`q.size()` 返回的是队列 `q` 的元素个数。因此，`q.size() < 3` 表示队列中的元素个数是否小于 3。

这样的判断是为了确保队列中至少有三个元素。如果队列中的元素个数小于 3，那么在尝试访问第三个元素之前，就已经不足够了。为了避免在队列元素个数不足时引发错误，函数使用 `if(q.size() < 3)` 进行条件判断。

如果队列中的元素个数小于 3，则函数返回 -1，表示无法获取第三个元素。如果队列中的元素个数不小于 3，则函数执行 `q.pop()` 两次，即移除队列的前两个元素，然后返回第三个元素的值（通过 `q.front()` 获取）。

这样的判断和处理是为了确保在队列元素足够多的情况下才尝试访问第三个元素，避免出现越界或错误的情况

`q.pop()` 是用于从队列中移除队首元素的操作。在这个特定的函数中，`q.pop()` 被调用了两次，目的是移除队列的前两个元素，因为函数的目标是返回队列的第三个元素。

这是因为队列是一个先进先出（FIFO）的数据结构，而题目要求返回的是第三个元素。所以，为了得到第三个元素，我们需要弹出前两个元素。具体过程如下：

1. 第一个 `q.pop()` 移除队列的头部元素；
2. 第二个 `q.pop()` 移除队列的新头部元素，此时队列中的元素就剩下原始队列的第三个元素及之后的元素；
3. `q.front()` 返回队列当前的头部元素，即原始队列的第三个元素。

这样的设计是基于队列的性质，确保函数正确地返回了队列的第三个元素。

第二种

```
/** * @param q: A queue * @return: An intager */ int getQueue3(queue<int> &q) {  
    if(q.size()<3){ return -1; }  
    else{ for(int i = 1;i<3;++i)
```



```
{ q.pop(); }  
return q.front(); } }
```

在 C++ 中，`return {i, j};` 是一种使用花括号初始化列表返回多个值的方式。这是 C++11 引入的一项特性，被称为初始化列表或者统一初始化语法。

这种方式可以用于返回包含多个值的标准库容器，如 `std::pair`、`std::tuple`，或者简单地返回一个数组。示例：

```
#include <iostream>  
#include <tuple>  
  
std::tuple<int, int> getTwoNumbers() {  
    int i = 10;  
    int j = 20;  
    return {i, j};  
}  
  
int main() {  
    auto result = getTwoNumbers();  
    std::cout << "First number: " << std::get<0>(result) << std::endl;  
    std::cout << "Second number: " << std::get<1>(result) << std::endl;  
  
    return 0;  
}
```

在这个例子中，`getTwoNumbers` 函数返回了一个 `std::tuple` 包含两个整数。然后在 `main` 函数中，使用 `std::get<0>` 和 `std::get<1>` 来获取 `tuple` 中的两个值。

如果你的编译器支持 C++17，你还可以使用结构化绑定（Structured Binding）：

```
#include <iostream>  
#include <tuple>  
  
std::tuple<int, int> getTwoNumbers() {  
    int i = 10;  
    int j = 20;  
    return {i, j};  
}  
  
int main() {  
    auto [first, second] = getTwoNumbers();
```

```
std::cout << "First number: " << first << std::endl;
std::cout << "Second number: " << second << std::endl;

return 0;
}
```

#leetcode1

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int n = nums.size();
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                if (nums[i] + nums[j] == target) {
                    return {i, j};
                }
            }
        }
        return {};
    }
};
```

- **twoSum 函数签名:** `vector<int> twoSum(vector<int>& nums, int target)`
 - **返回类型:** `vector<int>`，表示返回一个包含两个整数的向量。
 - **参数:**
 - `nums`：类型为 `vector<int>&` 的引用，表示输入的整数数组。
 - `target`：类型为 `int`，表示目标和。
- **函数主体:**
 - `int n = nums.size();`：获取数组 `nums` 的大小，即元素个数。
 - `for (int i = 0; i < n; ++i)`：外层循环，遍历数组 `nums` 的每个元素，索引从 `0` 到 `n-1`。
 - `for (int j = i + 1; j < n; ++j)`：内层循环，在外层循环的基础上，从当前元素的下一个元素开始遍历，索引从 `i+1` 到 `n-1`。
 - `if (nums[i] + nums[j] == target)`：如果当前两个元素的和等于目标值 `target`，说明找到了满足条件的一对元素。
 - `return {i, j};`：直接返回一个包含满足条件的元素索引的向量。
 - 循环结束后，如果没有找到满足条件的一对元素，则执行 `return {};`；返回一个空向量，表示未找到解。

这个函数的目的是找到数组中两个元素，使它们的和等于给定的目标值 `target`。如果找到了这样的一对元素，函数返回它们的索引；否则，返回一个空向量。这个解法的时间复杂度为 $O(n^2)$ ，因为使用了嵌套的循环来遍历所有可能的元素组合

`++X&&Y--||++Z`

1. `++X`：这是对变量 `x` 的前增量操作。它在使用值之前增加 `x` 的值。
2. `Y--`：这是对变量 `y` 的后减量操作。它减小 `y` 的值，但在表达式中使用的是当前值，然后 `y` 在此之后递减。
3. `||`：这是逻辑或运算符。如果至少一个操作数为 `true`，它就返回 `true`。

现在，结合这些操作：

- `++X`：增加 `x`。
- `Y--`：使用当前 `y` 的值，然后递减它。
- `++X && Y--`：如果 `++X` 和 `Y--` 都为真，这部分将为真。逻辑与 (`&&`) 要求两个操作数都为真才为真。
- `++X && Y-- || ++Z`：最后，逻辑或 (`||`) 将在 `++X && Y--` 为真或 `++Z` 为真时返回 `true`。如果左侧为真，则不会执行右侧，因此 `++Z` 不会被执行。

需要注意的是整个表达式的最终值将是 `true` 或 `false`，基于布尔条件和运算。此外，增量和减量的副作用将修改 `x`、`y` 和 `z` 的值。

#动态数组


ChatGPT

`vector<int>` 是 C++ 中用于表示整数类型动态数组的一种数据类型。它是 C++ 标准库 (Standard Template Library, STL) 中的一部分，提供了一种灵活的、动态大小的数组结构。让我们对这个类型进行一些解读：

- `vector`：这是 C++ 标准库中的一个模板类，表示动态数组。`vector` 可以容纳多个元素，而且它的大小可以动态调整。
- `<int>`：这是 `vector` 的模板参数，指定了数组中元素的类型。在这里，`<int>` 表示这个 `vector` 只能容纳整数类型的元素。

在你的代码中，你经常会看到类似这样的声明：

cpp

 Copy code

```
vector<int> result;
```

这行代码声明了一个名为 `result` 的变量，其类型是 `vector<int>`。这意味着 `result` 是一个整数类型的动态数组。你可以在程序运行时向 `result` 中添加元素，也可以根据需要动态调整数组的大小。在你的例子中，`result` 可能会被用来存储两数之和的索引，例如 `{i, j}`。

#STL类与对象例题1

```
#include <vector>
#include <iostream>

using namespace std;

class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int n = nums.size();
        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                if (nums[i] + nums[j] == target) {
                    return {i, j};
                }
            }
        }
        return {};
    }
};

int main() {
    // 在这里可以调用 Solution 类中的函数进行测试
```

```
// 示例：
Solution solution;
vector<int> nums = {2, 7, 11, 15};
int target = 9;
vector<int> result = solution.twoSum(nums, target);


// 输出结果
cout << "Result: ";
for (int num : result) {
    cout << num << " ";
}
cout << endl;

return 0;
}
```

#for循环遍历容器

这是 C++ 中的范围 (range-based) for 循环的语法，用于方便地遍历容器（如数组、`vector` 等）。让我们解读这行代码：

cpp

 Copy code

```
for (int num : result) {
    // 循环体，针对 result 中的每个元素执行
}
```

- `int num`：这部分声明了一个整数变量 `num`，用于存储 `result` 中每次迭代的元素的值。
- `::`：表示范围 (range-based) for 循环的语法。
- `result`：这是被遍历的容器，即 `vector<int>` 类型的变量。
- **循环体**：在花括号 `{}` 内的代码是循环体，会针对 `result` 中的每个元素执行一次。

在你的例子中，可能会使用这个循环来遍历 `result` 中的元素，例如：

在你的例子中，可能会使用这个循环来遍历 `result` 中的元素，例如：

```
cpp                                                                    Copy code

for (int num : result) {
    // 这里可以对每个元素进行操作，例如输出到控制台
    cout << num << " ";
}
```

这样的循环结构允许你更简洁地遍历容器，而不必使用传统的索引方式。在每次循环迭代中，`num` 将依次取得 `result` 中的每个元素的值。

#反向迭代器

这行代码是在比较字符串 `s` 是否与其反转后的字符串相等，用于检查整数是否为回文数。

```
cpp                                                                    Copy code

return s == std::string(s.rbegin(), s.rend());
```


让我们逐步解释：

1. `s.rbegin()`：返回一个指向 `s` 字符串的最后一个字符的反向迭代器（逆向迭代器）。
2. `s.rend()`：返回一个指向字符串 `s` 开头之前的位置的逆向迭代器。
3. `std::string(s.rbegin(), s.rend())`：使用反向迭代器构造了一个新的字符串，这个字符串是原字符串 `s` 的反转。
例如，如果 `s` 是 "121"，那么 `std::string(s.rbegin(), s.rend())` 构造的新字符串就是 "121" 的反转，即 "121"。
4. `s == std::string(s.rbegin(), s.rend())`：检查原字符串 `s` 是否与其反转后的字符串相等。
如果 `s` 是回文数（例如 "121"），则这个条件为真，函数返回 `true`，表示整数是回文数；否则，返回 `false`。

是的, `std::to_string(x)` 是一个类型转换函数, 将整数 `x` 转换为其字符串表示形式。这个函数是 C++ 标准库的一部分, 定义在 `<string>` 头文件中。

在你的 `isPalindrome` 函数的上下文中:

cpp

 Copy code

```
std::string s = std::to_string(x);
```

这行代码将整数 `x` 转换为一个字符串 (`std::string`)。这个操作使得你可以将数值视为字符序列, 这在执行字符串操作时非常有用, 比如在回文检查中比较字符。

例如, 如果 `x` 是 `121`, 经过这个转换后, `s` 将是字符串 `"121"`。然后, 可以轻松地反转这个字符串并进行比较, 以检查是否是回文。

#string反向迭代器例题1

```
#include <iostream>
#include <string>

class Solution {
public:
    bool isPalindrome(int x) {
        // 将整数转换为字符串
        std::string s = std::to_string(x);

        // 检查字符串是否与其反转后的字符串相同
        return s == std::string(s.rbegin(), s.rend());
    }
};

int main() {
    // 创建 Solution 对象
    Solution solution;

    // 测试一些案例
    int num1 = 121;
    int num2 = -121;
    int num3 = 10;

    std::cout << num1 << " is palindrome: " << std::boolalpha <<
solution.isPalindrome(num1) << std::endl;
    std::cout << num2 << " is palindrome: " << std::boolalpha <<
```

```

solution.isPalindrome(num2) << std::endl;
    std::cout << num3 << " is palindrome: " << std::boolalpha <<
solution.isPalindrome(num3) << std::endl;

    return 0;
}

```

#三种初始化方式

```

1 // initialization of variables
2
3 #include <iostream>
4 using namespace std;
5
6 int main ()
7 {
8     int a=5;           // initial value: 5
9     int b(3);          // initial value: 3
10    int c{2};           // initial value: 2
11    int result;         // initial value undetermined
12
13    a = a + b;
14    result = a - c;
15    cout << result;
16
17    return 0;
18 }

```

6

#类型推导

• 类型推导: auto 和 decltype

初始化新变量时，编译器可以由初始值设定项自动确定变量的类型。为此，将其用作变量的类型说明符就足够了：auto

```

1 int foo = 0;
2 auto bar = foo; // the same as: int bar = foo;

```

这里，被声明为具有类型;因此，type of 是用于初始化它的值的类型：在本例中，它使用的类型，即。

未初始化的变量也可以使用带有说明符的类型推导：barauto barfoo intdecltype

```

1 int foo = 0;
2 decltype(foo) bar; // the same as: int bar;

```

此处，声明为具有与 相同的类型。

并且是最近添加到该语言中的强大功能。但是，它们引入的类型推导功能旨在用于无法通过其他方式获取类型或使用它可以提高代码可读性的情况。上面的两个例子可能不是这些用例。事实上，它们可能降低了可读性，因为在阅读代码时，必须搜索 的类型才能真正知道 的类型。

barfooauto decltypefoo bar

一些程序员还使用一个技巧在多行中包含长字符串文字：在 C++ 中，行尾的反斜杠（\）被视为行继续字符，它将该行和下一行合并为一行。因此，以下代码：

 12 	 x = "string expressed in \ two lines"
------------	--

相当于：

 1 	 x = "string expressed in two lines"
-----------	---

#基于循环容器for

语法

属性(可选) **for** (初始化语句(可选)范围变量声明 : 范围表达式) 循环语句

属性 - 任意数量的属性

初始化语句 - (C++20 起) 以下之一：

- 表达式语句 (可以是空语句“;”)
- 简单声明，典型地为带初始化的变量声明，但它可以声明任意多个变量，或者是结构化绑定声明

- 别名声明 (C++23 起)

注意，所有 初始化语句 必然以分号 ; 结尾，因此它经常被非正式地描述为后随分号的表达式或声明。

范围变量声明 - 一个具名变量的声明，它的类型是由 范围表达式 所表示的序列的元素的类型，或该类型的引用。通常用 **auto 说明符**进行自动类型推导。

范围表达式 - 可以表示一个合适的序列（数组，或定义了 **begin** 和 **end** 成员函数或自由函数的对象，见下文）的任意表达式，或一个花括号初始化器列表。

循环语句 - 任意语句，通常是一条复合语句，它是循环体

范围变量声明 可以是结构化绑定声明：

```
for (auto&& [first,second] : mymap)
{
    // 使用 first 和 second
}
```

(C++17 起)

解释

上述语法产生的代码等价于下列代码（__range、__begin 和 __end 仅用于阐释）：

```
{
    auto && __range = 范围表达式 ;
    for (auto __begin = 首表达式, __end = 尾表达式 ; __begin != __end; ++__begin)
    {
        范围变量声明 = *__begin;
        循环语句
    }
}
```

(C++17 前)

```

{
    auto && __range = 范围表达式 ;
    auto __begin = 首表达式 ;
    auto __end = 尾表达式 ;
    for ( ; __begin != __end; ++__begin)
    {
        范围变量声明 = *__begin;
        循环语句
    }
}

{
    初始化语句
    auto && __range = 范围表达式 ;
    auto __begin = 首表达式 ;
    auto __end = 尾表达式 ;
    for ( ; __begin != __end; ++__begin)
    {
        范围变量声明 = *__begin;
        循环语句
    }
}

```

(C++17 起)
(C++20 前)

(C++20 起)

总结就是（啥类型：访问啥）

注解

如果初始化器（范围表达式）是花括号初始化器列表，那么 __range 被推导为 std::initializer_list<>&&。

在泛型代码中，使用推导的转发引用，如 `for (auto&& var : sequence)`，是安全且受推荐的做法。

如果范围类型拥有名为 begin 的成员和名为 end 的成员，那么使用成员解释方案。其中无视成员是类型、数据成员、函数还是枚举项及其可访问性。从而像 `class meow { enum { begin = 1, end = 2 }; /* 类的剩余部分 */ }` 的类不能用于基于范围的 for 循环，即使有命名空间作用域的 begin/end 函数存在。

虽然通常在 循环语句 中使用声明于 范围变量声明 的变量，但并不要求这么做。

从 C++17 开始，首表达式 和 尾表达式 的类型不必相同，而且实际上 尾表达式 的类型不必是迭代器：它只需要能与一个迭代器比较是否不等。这允许以一个谓词（例如“迭代器指向空字符”）对范围进行分界。
(C++17 起)

当基于范围的 for 循环被用于一个具有写时复制语义的（非 const）对象时，它可能会通过（隐式）调用非 const 的 begin() 成员函数触发深层复制。

如果想要避免这种行为（比如循环实际上不会修改这个对象），可以使用 std::as_const：

```

struct cow_string { /* ... */ }; // 写时复制的字符串
cow_string str = /* ... */;

// for(auto x : str) { /* ... */ } // 可能会导致深层复制

for(auto x : std::as_const(str)) { /* ... */ }

```

(C++17 起)

```

#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {0, 1, 2, 3, 4, 5};
}

```

```

for (const int& i : v) // 以 const 引用访问
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << i << ' ';
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';

for (auto i : v) // 以值访问, i 的类型是 int
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << i << ' ';
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';

for (auto&& i : v) // 以转发引用访问, i 的类型是 int&
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << i << ' ';
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';

const auto& cv = v;

for (auto&& i : cv) // 以转发引用访问, i 的类型是 const int&
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << i << ' ';
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';

for (int n : {0, 1, 2, 3, 4, 5}) // 初始化器可以是花括号初始化器列表
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << n << ' ';
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';

int a[] = {0, 1, 2, 3, 4, 5};
for (int n : a) // 初始化器可以是数组
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << n << ' ';
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';

for ([[maybe_unused]] int n : a)
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << 1 << ' '; // 不必使
用循环变量
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';

for (auto n = v.size(); auto i : v) // 初始化语句 (C++20)
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << --n + i << ' ';
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';

for (typedef decltype(v)::value_type elem_t; elem_t i : v)
// typedef 声明作为初始化语句 (C++20)
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << i << ' ';
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';

for (using elem_t = decltype(v)::value_type; elem_t i : v)
// 别名声明作为初始化语句, 同上 (C++23)
    [std::cout](http://zh.cppreference.com/w/cpp/io/cout) << i << ' ';
[std::cout](http://zh.cppreference.com/w/cpp/io/cout) << '\n';
}

```

```
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> v = {0, 1, 2, 3, 4, 5};

    for (const int& i : v) // 以 const 引用访问
        std::cout << i << ' ';
    std::cout << '\n';

    for (auto i : v) // 以值访问, i 的类型是 int
        std::cout << i << ' ';
    std::cout << '\n';

    for (auto& i : v) // 以转发引用访问, i 的类型是 int&
        std::cout << i << ' ';
    std::cout << '\n';

    const auto& cv = v;

    for (auto& i : cv) // 以转发引用访问, i 的类型是 const int&
        std::cout << i << ' ';
    std::cout << '\n';

    for (int n : {0, 1, 2, 3, 4, 5}) // 初始化器可以是花括号初始化器列表
        std::cout << n << ' ';
    std::cout << '\n';

    int a[] = {0, 1, 2, 3, 4, 5};
    for (int n : a) // 初始化器可以是数组
        std::cout << n << ' ';
    std::cout << '\n';

    for ([[maybe_unused]] int n : a)
        std::cout << 1 << ' '; // 不必使用循环变量
    std::cout << '\n';

    for (auto n = v.size(); auto i : v) // 初始化语句 (C++20)
        std::cout << --n + i << ' ';
    std::cout << '\n';
}
```

```
for (typedef decltype(v)::value_type elem_t; elem_t i : v)
// typedef 声明作为初始化语句 (C++20)
    std::cout << i << ' ';
std::cout << '\n';

for (using elem_t = decltype(v)::value_type; elem_t i : v)
// 别名声明作为初始化语句，同上 (C++23)
    std::cout << i << ' ';
std::cout << '\n';
}
```