

# OJ log

## 上机之前

考试范围：每次作业（除并发）+上机练习

## 经验之谈

1. debug怎么都找不出来：不要乱，列出不同的情况，挨个试
2. 倒着遍历：见采购日志那道题，因为对于同maid的物品，它的最终卖出价格随买入时间**单调不增**，最终盈利取决于**它之后**的价格中的最大值，因此倒着扫描会很妙。
3. 数据类型要看清啊：原料采购bug竟然是因为数据类型错了！！！float类型的变量，传参的时候传成了int
4. 像铺木地板一样，不会有很复杂的算法，但是逻辑判断复杂，有很多分支，要将情况分类清楚，分成几大类分别处理
5. 血的教训，处理输入后检验内容总是空白，后来发现是把double类型试图存到int变量里
6. 运行异常多去判断一下是u不是nullptr之类的
7. 血的教训，看完题先拉到最后检查是否有题目提示

## 1031oj

1. 写一点交一次，如果有语法错误可以及时发现&定位
2. class定义：{}；最后有分号
3. 根据报错内容找bug，不要小看报错信息的提醒

## 未雨绸缪

### 一.库

vector

```
| #include <vector>
```

max(a, b) min(a, b)

```
| #include<algorithm>
```

```
stringstream ss;  
while(ss >> str){  
    #include<sstream>
```

## Deque

```
#include <deque>  
#include <iomanip>  
Cout << fixed << setprecision(1) << adouble;  
#include<cctype>  
c = c.toupper();  
c = c.tolower();  
<limits>  
numeric_limits<int>::max()  
numeric_limits<int>::min()
```

## 二.报错

运行异常，请检查错误：  
数组、vector等越界访问

## 三.读入操作

```
cin >> insts;
```

不读入最后的换行符，换行符依然在缓冲区中

```
getline(cin, insts);
```

不读入最后的换行符，但换行符被消耗，缓冲区没有换行符

```
char c;  
cin.get(c);
```

读入一个字符

```
cin.ignore();
```

忽略换行符

## 为了避免出错

每次读入一整行再做分解

## 四.输出

`std::round(num)` 会将数字四舍五入到最接近的整数

`floor(num)` 向下取整，抛弃小数部分

### 1.1 “最多”保留一位小数

4 -> 4

4.1 -> 4.1

4.12 -> 4.1

```
1 #include <cmath>
2
3 double validDouble(double d){
4     if(round(d) == d){ //floor(d) == d
5         return d;
6     }else{
7         return round(d * 10) / 10;
8     }
9 }
```

### 1.2 保留一位小数 且 直接输出

4 -> 4.0

4.1 -> 4.1

4.12 -> 4.1

```
1 #include <iomanip>
2
3 cout << fixed << setprecision(1) << value << endl;
```

## 五.排序

排序的函数、自定义排序方法

### 1.1 库函数

[C++ 排序函数 sort\(\),qsort\(\)的用法\\_c++sort函数的用法-CSDN博客](#)

```
| partial_sort(first, first+k, last);
```

将[first, last)中，最小的k给元素排序，放在[first, first+k)中，[first+k, last)中的顺序不能保证。

### 1.2 自定义排序方法

```
1 #include <algorithm>
2
3 sort(vec.begin(), vec.end(), cmp);
4
5 bool cmp(int a, int b){ //返回true, 如果a应该排在b的前面
6     return a>b; //因此cmp构建vec递减
7 }
```

### 1.3 lambda表达式

```
1     sort(newest.begin(), newest.end(), [](const RevNode* a, const RevNode* b){
2         if(a->time1 == b->time1){
3             return a->time2 > b->time2; //此为降序排序
4         }
5         return a->time1 > b->time1;
6     });
```

### 1.4 不需要排序，找到元素最大最小

```
1 #include <algorithm>
2
3 int min = *min_element(cards.begin(), cards.end());
4 int max = *max_element(cards.begin(), cards.end());
```

hw1

# 神奇的电子表

## 1.4 tm

后来马哥说这算作弊

`tm` 是 C/C++ 标准库中定义的一种结构体，用于表示时间和日期。它定义在 `<ctime>` 头文件（C++ 中）或 `<time.h>` 头文件（C 语言中）中。

常用函数：

- `time_t mktime(struct tm* timeptr)`：将 `tm` 结构转换为 `time_t` 类型（即自1970年1月1日以来的秒数）。
- `struct tm* localtime(const time_t* timer)`：将 `time_t` 转换为本地时间的 `tm` 结构。
- `double difftime(time_t end, time_t start)`：计算两个 `time_t` 时间之间的差值（秒数）。

## 1.5 制表法

```
1 int monthStrToNum(const string &month){
2     static map<string, int> monthMap = {
3         {"January", 1}, {"February", 2}, {"March", 3},
4         {"April", 4}, {"May", 5}, {"June", 6},
5         {"July", 7}, {"August", 8}, {"September", 9},
6         {"October", 10}, {"November", 11}, {"December", 12}
7     };
8     return monthMap[month];
9 }
```

## 1.6 Bug：计算机的2038问题

可能是因为这个，用例过了5/10

## 1.7 选择基准时间

从2024年9月22日17: 32到2025年1月1日12: 22 肯定没有人会想算

但是从1970年1月1日0点开始就好多了

思路如下：

1970.1.1.0.0.0  
baseDT

$$\begin{array}{c} \xrightarrow[\Delta_1]{eB} \xrightarrow[\Delta_3]{EDT} \\ \xrightarrow[\Delta_2]{BJT} \xrightarrow[\Delta_4]{BJDT} \end{array}$$

$$\frac{EDT - eB}{59} = \frac{BJDT - eB}{60}$$

$$60(\Delta_3 - \Delta_1) = 59(\Delta_4 - \Delta_2)$$

$$\Delta_4 = \frac{60}{59}(\Delta_3 - \Delta_1) + \Delta_2$$

## 1.8 GPT写的真好啊。。。

```

1 /**
2  根据和基准时间的秒数差，计算现在的时间
3  */
4 DateTime secToDT(long long diffInSec) {
5     DateTime dt;
6
7     // 第一步：计算秒数对应的年份
8     dt.y = baseDT.y;
9     while (true) {
10         // 判断当前年份是否是闰年
11         bool isLeap = (dt.y % 4 == 0 && dt.y % 100 != 0) || (dt.y % 400 == 0);
12         // 计算当前年份的总秒数
13         int daysInYear = isLeap ? 366 : 365;
14         long long secondsInYear = daysInYear * SECONDS_IN_A_DAY;
15
16         // 如果剩余秒数大于当前年份的秒数，说明时间还在后续年份
17         if (diffInSec >= secondsInYear) {
18             diffInSec -= secondsInYear;
19             dt.y++; // 前进到下一年
20         } else {
21             break; // 找到对应的年份
22         }
23     }
24
25     // 第二步：计算月份和日期

```

```

26     dt.mon = 1;
27     while (true) {
28         // 获取当前月份的天数
29         int daysInMonth = monthDayNum(dt.mon);
30         // 闰年二月特殊处理
31         if (dt.mon == 2 && ((dt.y % 4 == 0 && dt.y % 100 != 0) || (dt.y % 400
== 0))) {
32             daysInMonth = 29;
33         }
34         long long secondsInMonth = daysInMonth * SECONDS_IN_A_DAY;
35
36         // 如果剩余秒数大于当前月份的秒数, 说明时间还在后续月份
37         if (diffInSec >= secondsInMonth) {
38             diffInSec -= secondsInMonth;
39             dt.mon++; // 前进到下一个月
40         } else {
41             break; // 找到对应的月份
42         }
43     }
44
45     // 计算天数
46     dt.day = 1 + diffInSec / SECONDS_IN_A_DAY; // 天数从1开始
47     diffInSec %= SECONDS_IN_A_DAY;
48
49     // 计算小时、分钟和秒
50     dt.h = diffInSec / SECONDS_IN_AN_HOUR;
51     diffInSec %= SECONDS_IN_AN_HOUR;
52
53     dt.min = diffInSec / SECONDS_IN_A_MINUTE;
54     diffInSec %= SECONDS_IN_A_MINUTE;
55
56     dt.sec = diffInSec;
57
58     return dt;
59 }

```

```

1  /**
2  根据现在的时间, 计算和基准时间相差的秒数
3  */
4  long long diffBaseDTSec(DateTime dt) {
5  //我自己写的, , , 不如gpt——不堪入目
6      // int diffSec = dt.sec - baseDT.sec;
7      // int diffMin = dt.min - baseDT.min;
8      // int diffH = dt.h - baseDT.h;
9      // long long diffDayWithoutLeap = 365 * dt.y + dt.day;

```

```

10 // for(int i = 1; i < dt.mon; i++){
11 //     diffDayWithoutLeap += monthDayNum(i);
12 // }
13 // int l4 = ((dt.y - 1) - 1968) / 4;
14 // int l100 = ((dt.y - 1) - 1900) / 100;
15 // int l400 = ((dt.y - 1) - 1600) / 400;
16
17 // int add1 = 0;
18 // //最后一年单独计算
19 // if ((dt.y % 4 == 0 && dt.y % 100 != 0) || dt.y % 400 == 0) { //最后一年是
闰年
20 //     if (dt.mon > 2) {
21 //         add1 = 1;
22 //     }
23 // }
24
25 // long long diffInDay = diffDayWithoutLeap + l4 - l100 + l400 + add1;
26 // long long diffInSec = diffSec + 60 * diffMin + 60 * 60 * diffH + 60 *
60 * 24 * diffInDay;
27
28 // return diffInSec;
29 // 1. 计算年份的总天数 (包括闰年)
30 long long totalDays = 0;
31
32 for (int year = baseDT.y; year < dt.y; year++) {
33     // 判断是否是闰年
34     if ((year % 4 == 0 && year % 100 != 0) || (year % 400 == 0)) {
35         totalDays += 366; // 闰年
36     } else {
37         totalDays += 365; // 平年
38     }
39 }
40
41 // 2. 计算当前年份内的天数
42 // 判断当前年份是否是闰年
43 bool isLeap = (dt.y % 4 == 0 && dt.y % 100 != 0) || (dt.y % 400 == 0);
44
45 // 累加从1月到当前月份的天数 (不包括当前天)
46 for (int month = 1; month < dt.mon; month++) {
47     if (month == 2 && isLeap) {
48         totalDays += 29; // 闰年二月
49     } else {
50         totalDays += monthDayNum(month);
51     }
52 }
53
54 // 3. 累加当前天数 (包括当天)

```



```

55     totalDays += (dt.day - 1); // 当前天需要减去1, 因为从0开始
56
57     // 4. 计算总秒数
58     long long totalSeconds = totalDays * SECONDS_IN_A_DAY; // 天数转秒数
59     totalSeconds += dt.h * SECONDS_IN_AN_HOUR; // 小时转秒数
60     totalSeconds += dt.min * SECONDS_IN_A_MINUTE; // 分钟转秒数
61     totalSeconds += dt.sec; // 剩余秒数
62
63     return totalSeconds;
64 }

```

## 方程的解

### 1.1 二分法的细节

```

1 double bisection(double min, double max, double epsilon){
2     double r;
3     while(max - min >= epsilon){
4         r = (max+min)/2;
5         if(f(r) == 0){
6             break;
7         }else if(f(r) * f(min) < 0 || (f(r) * f(min) == 0 && f(r) * f(max) >
8             0)){ //[min, r]
9             max = r;
10        }else{ //[r, max]
11            min = r;
12        }
13    }
14    return r;
15 }

```

第七行 $f(r) * f(min) == 0 \ \&\& \ f(r) * f(max) > 0$ 去掉, 会导致当端点恰好是零点时出错, 因为题目要求的是开区间  $(0, 10)$ , 所以要用这种方法来避免。

### 1.2 二分法的收敛条件和精度要求

上面代码段第二行: `double epsilon = 1e-7;`

### 1.3 浮点数输出的精度

`setprecision` 是 C++ 中 `<iomanip>` 头文件里的一个函数, 用于设置浮点数输出的精度。具体来说, 它可以控制输出流中浮点数的小数位数。

```
1 #include <iomanip>
2
3 cout << fixed << setprecision(6) << value << endl; // 输出 3.141593
```

## 字母频率统计

复习：21: 00

### 1.1 容器中的元素排序

`sort` 是 C++ 标准库中的一个函数，用于对容器中的元素进行排序。它的定义在 `<algorithm>` 头文件中。

```
1 sort(begin_iterator, end_iterator);
```

**begin\_iterator**: 指向要排序的范围的起始位置。

**end\_iterator**: 指向要排序的范围的结束位置（不包括该位置）。

### 1.2 数组清空

`chars.clear();`

### 1.3 读入不正确的时候：考虑时候有换行符等导致读入的不是期待的

`cin.get();` 能消耗一个换行符

### 1.4 操作顺序

```
1 void query(int *count){
2     int max = 0;
3     vector<char> chars;
4     for(int i = 0; i < 26; i++){
5         if(count[i] == max && max > 0){ //一样多
6             chars.push_back('a'+i);
7         }else if (count[i] > max){
8             chars.clear();
9             chars.push_back('a'+i);
10            max = count[i];
11        }
12    }
13
14    sort(chars.begin(), chars.end());
15    for(auto c : chars){
```

```

16         cout << c;
17     }
18     cout << endl;
19 }
20
21 void add(string word, int* count){
22     for(auto c : word){
23         int index = c - 'a';
24         if(index >= 26) continue;
25         count[index]++;
26     }
27 }

```

题目中要求输入两种操作，要考虑到任何一种操作作为第一个操作的情况。

如果第一个操作是query，一开始max=-1的话，会输出26个字母，导致了错误。

## 最短子数组

复习：16: 51-17:25+20min = 50min

错误原因：一开始在纸上列关系式时，移到右边的符号忘记变

经验：找例子试，多输出

## 解题思路

1. **前缀和数组**：首先我们可以计算一个前缀和数组 `prefix_sum`，其中 `prefix_sum[i]` 表示从数组的开始到位置 `i` 的元素和。这将使得计算某个子数组和的时间复杂度降低到  $O(1)O(1)O(1)$ ，因为子数组的和可以通过：

$$\text{subarray sum} = \text{prefix\_sum}[r] - \text{prefix\_sum}[l - 1]$$

2. **滑动窗口法**：这是一个常见的用于解决最短子数组问题的技术。我们可以使用滑动窗口法，维护一个左右边界 `lll` 和 `rrr` 来代表当前窗口，逐渐向右移动右边界 `rrr`，并在每次满足条件时更新最优解，同时移动左边界 `lll`。
3. **二分查找优化**：二分查找的思想可以应用在前缀和数组中。对于每个位置 `rrr`，我们可以通过二分查找在 `prefix_sum` 中找到最小的 `lll`，使得：

$$\text{prefix\_sum}[r] - \text{prefix\_sum}[l - 1] \geq \text{target}$$

通过这种方式，利用二分查找来快速找到满足条件的左边界 `lll`。

## 1.1 oj系统array维度只能先转换为const类型

## 1.2 宝藏二分查找函数！！

`lower_bound` 是 C++ 标准库中的函数，定义在头文件 `<algorithm>` 中。

类似的是 `upper_bound`，此外还有 `binary_search()`。

简单概括：

- `lower_bound`(起始地址，结束地址，要查找的数值)：返回被查序列中第一个大于等于查找值的指针。
- `upper_bound`(起始地址，结束地址，要查找的数值)：返回被查序列中第一个大于查找值的指针；
- `binary_search`(起始地址，结束地址，要查找的数值) 返回的是是否存在这么一个数，是一个 `bool` 值。

下面以 `lower_bound` 为例详细讲解：

### 功能

`lower_bound` 函数用于在有序数组或容器中查找不小于某个值的第一个位置。其返回的是一个指向该元素的迭代器（或指针）。具体来说，对于一个有序区间 `[first, last)` 和一个目标值 `val`，`lower_bound` 返回第一个满足 `= val` 条件的位置。

### 语法

```
std::lower_bound(first, last, value);
```

- `first` 和 `last` 是迭代器，表示查找的区间 `[first, last)`；
- `value` 是要查找的值。

### 返回值

`lower_bound` 返回指向第一个大于或等于 `value` 的位置的迭代器。如果不存在满足条件的元素，则返回 `last`。【这个要注意！】

例子：

```
1 int l = lower_bound(arr.begin(), arr.end(), required_sum) - arr.begin();
```

## 1.3 得到迭代器之后如何知道其下标

```
1 // 计算下标
2 int index = std::distance(prefix.begin(), it);
```

## 1.4 c和c++风格的数组！！

在 C++ 中，数组可以分为两种类型：**C 风格数组**和**C++ 标准库提供的容器类数组**。这两种数组在使用上有明显的区别。

```
1 //c
2 int prefix_sum[size];
3 int minl = upper_bound(prefix_sum, prefix_sum + size, target) - prefix_sum;
4 //c++
5 vector<int> prefix_sum(size);
6 int minl = upper_bound(prefix_sum.begin(), prefix_sum.end(), target) -
    prefix_sum.begin();
```

## 1.5 Bug

用这个例子

```
1 6
2 1 3 5 7 9 11
3 10
```

会怀疑upper\_bound函数是否可用——经检查可用

bug原因：题目要求找的是最小长度，不是最小下标和（次优先）

## 密钥格式化

### 1.1 画图理解题意

### 1.2 考虑各种边界条件

### 1.3 按分隔符分割字符串

```
1 int main(){
2     string input, snippet;
3     cin >> input;
4     vector<string> vec;
5     //将字符串转换成流
6     stringstream ss(input);
7     //while成功：读取，直到遇到-
8     while(getline(ss, snippet, '-')){
9         vec.push_back(snippet);
```

```

10     }
11
12     for(auto it = vec.begin(); it != vec.end(); it++){
13         cout << *it << endl; // * (迭代器)
14     }
15 }

```

## hw2

### 原料采购

不会在任何情况下出现桶装单价上升时，总购买量反而增加的情况，问题中的总购买量是严格单调递减的。图片中是证明：

涨价后多出的用于买散装货的资金为  $m \Delta x$

原来用于买桶装，能买：  $\frac{m}{x}$       现在：  $\frac{m}{x+\Delta x-k}$

$$\begin{aligned}
 \text{左} - \text{右} &= \frac{m}{x} - \frac{mk}{(x+\Delta x-k)(x+\Delta x)} \\
 &= \frac{m[(x+\Delta x)^2 - k(x+\Delta x) - kx]}{x(x+\Delta x-k)(x+\Delta x)} \\
 &= \frac{m(x^2 + 2x\Delta x + \Delta x^2 - k\Delta x)}{x(x+\Delta x-k)(x+\Delta x)} > 0
 \end{aligned}$$

∴ 所以 总购买量一定减少

资金分配确实会影响每个采购员购买的桶数和散装原料的量，但这种影响并不会导致总购买量的上升。（要把所有员工的资金作为整体考虑）

即使剩余资金在增加，最终散装原料的贡献不足以弥补桶装原料减少带来的影响。因此，随着桶装价格上升，总购买的原料量一定是严格减少的。

选择 **二分查找**

bug竟然是因为数据类型错了！！！！

Bug: 猜测是精度问题，不知道提示是什么意思

后来是发现，float类型的变量，传参的时候传成了int

### BF解释器

## 1. 读入操作

```
cin >> insts;
```

不读入最后的换行符，换行符依然在缓冲区中

```
getline(cin, insts);
```

不读入最后的换行符，但换行符被消耗，缓冲区没有换行符

```
char c;
```

```
cin.get(c);
```

读入一个字符

## 2. cin.get()

两种用法

1) cin.get(变量名)

接收空格、回车等字符。剩下的数据会保留在输入流（缓冲区）中，下一次继续读取。且只读一个字符（char）。空格、回车也会当做一个字符被存储。

2) cin.get(变量名, 个数)

接收空格，遇到回车结束，且不会过滤掉回车符，回车符一直保留在输入流（缓存区），就会导致后面无法接收字符（例如上面的第2张图片）。因为数组末尾要加‘\0’，所有**实际接收的字符数量少一位**。

## 3. []一一对应，用到栈

```
1         if(ins == '['){
2             if(arr[dp] == 0){
3                 //jump to ']'
4                 int countBracks = 1;
5                 while(countBracks > 0 && ++i < insts.size()){
6                     if(insts[i] == ']') countBracks--;
7                     if(insts[i] == '[') countBracks++;
8                 }
9             }else{
10                loopstack.push(i);
11            }
12        }else if(ins == ' '){
13            if(arr[dp] != 0){
14                //jump to '['
15                /*
16                不是pop
```

```

17             不用+1, 因为在下一个for循环会自动加1
18             */
19             i = loopstack.top();
20         }else{
21             loopstack.pop();
22         }
23     }

```

## 外卖骑手

很有意思的一道题！

要结合贪心算法和一个决定先行条件的数据结构

## 0. 复习-二维数组作为形参

错误的：

```

1 void dfs(int sum, int count, int &dist[][], int addr) //不对

```

依然有错误

```

1 void dfs(int sum, int count, int dist[][N], int addr) {
2     // 代码
3 }
4 void dfs(int sum, int count, int** dist, int addr) {
5     // 代码
6 }

```

用vector代替

```

1 void dfs(int sum, int count, std::vector<std::vector<int>>& dist, int addr) {
2     // 代码
3 }

```

## 1.1 算法

最小生成树-管道联通问题



## 1.2 思路提示

1. 运用《数据结构与算法》中的知识，思考如何计算任意两个地点之间的最短距离
2. 可以使用深度优先搜索（递归、回溯）算法，尝试所有可能的送外卖顺序，求解最短时间。

## 1.3 迪杰斯特拉（dijkstra）算法（后来改用弗洛伊德

单源最短路径问题的求解方法

（1）计算两两地点之间的最短距离

计算哪些点之间的最短距离？——后来发现不需要考虑，因为该算法计算指定开始点，计算出该开始点到所有节点的最短距离。

- rest[i]到cust[j]
- rest[i]到rest[j]
- cust[i]到cust[j]
- cust[i]到rest[j]  $i \neq j$

（2）尝试可能的取餐、送餐顺序

用递归+回溯

rest、cust一共2orderNum个的排序

要满足：

- -> cust[i]: status[i] = 1
- -> rest[i]: status[i] = 0

## 1.4 floyd算法

- 不固定起始点
- 三个循环
- 易记不易错

```
1 void floyd(){
2     for(int i = 0; i < nodeNum; i++){
3         for(int j = 0; j < nodeNum; j++){
4             if(edge[i][j] == -1){
5                 spath[i][j] = 1e9; //转换成无穷大，后面判断逻辑就简单很多
6             }else{
7                 spath[i][j] = edge[i][j];
8             }
9         }
10    }
```

```

10     }
11     for(int k = 0; k < nodeNum; k++){
12         for(int i = 0; i < nodeNum; i++){
13             for(int j = 0; j < nodeNum; j++){
14                 spath[i][j] = min(spath[i][j], spath[i][k] + spath[k][j]);
15             }
16         }
17     }
18 }

```

## 1.5 用vector代替array，解决非常量带来的困扰

```

1  int orderNum;
2  int nodeNum;
3  vector<vector<int>> edge; // 使用vector代替二维数组
4  vector<vector<int>> orders; // 使用vector代替二维数组
5  int main(){
6      // 1.处理输入
7      cin >> orderNum;
8      cin >> nodeNum;
9
10     // 初始化二维vector
11     edge.resize(nodeNum, vector<int>(nodeNum));
12     orders.resize(orderNum, vector<int>(3));
13 }

```

## 1.6 Bug

- 当前顺序不一定能实现，比如过程中能走的点的距离都是-1——好像没问题，跳过就行
- distance递归函数的参数——好像没问题
- 真正原因：题目的node是从1开始计数的！！！！

## 1.7 超时\*2 错误答案\*1

解决超时：

- (1) 不一定要计算每两个之间的距离
- (2) 在递归中，提前结束一些【已知不可能是最优解的情况】
- (3) diakstra算法有问题: 1.不可达一般表示成无穷大，但本题是-1，导致的问题

## 1.8 错误答案\*1

原因：把迪杰斯特拉算法换成弗洛伊德之后，bug消失了

# 采购日志

## 1.1 数据结构：嵌套vector

因为一件商品的购买次数未知，而且价格不一样。本来想用链表，后来想起来c++的伟大变长vector

## 1.2 答案错误\*2 超时\*3

## 1.3 换个数据结构

只记录某种商品购买的总个数和总盈利，以及该商品目前出现过的最高价格

## 1.4 倒着遍历，

(1) 需要先存储日志

```
1 /*
2  for (int i = 0; i < lognum; ++i) {
3      getline(cin, line);
4      logs.push_back(line);
5  }
6  */
7  for (int i = 0; i < lognum; ++i) {
8      getline(cin, logs[i]);
9  }
```

(2) 逐行处理时，处理每行每行命令内容的效率问题

```
1 //stringstream效率低下
2  for(int l = lognum-1; l >= 0; l--){
3      // 使用 stringstream 来解析每行的输入
4      stringstream ss(logs[l]);
5      ss >> ins >> id >> money;
6  }
```

```
1  for(int l = lognum-1; l >= 0; l--){
2      const string &log = logs[l];
3
4      if(log[0] == 'b') {
5          int id = 0, money = 0, idx = 4;
6          // Parse the id and money efficiently
7          while (log[idx] != ' ') {
```

```

8         id = id * 10 + (log[idx] - '0');
9         ++idx;
10    }
11    ++idx;
12    while (idx < log.size()) {
13        money = money * 10 + (log[idx] - '0');
14        ++idx;
15    }
16
17    if (max_price[id] > money) {
18        sum += (max_price[id] - money);
19    }
20    max_price[id] = max(max_price[id], money);
21 }else{
22     //对于命令是“update”进行相似处理
23 }
24 //cout << id << "-" << max_price[id]<< endl;
25 }

```

## 1.5 爆int

long long

## 消除星星

好玩的一道题！难度在**递归**

提示中有三种实现计算“消除块的方法”，我第一反应是用dfs，于是就这么干了

### （1）深度优先搜索

DFS 是通过递归的方式，从一个起点开始，不断向四个方向（上下左右）扩展，找到所有相邻并且颜色相同的星星，直到无法继续扩展为止。之后可以通过回溯的方式标记消除的星星。

### （2）广度优先搜索

BFS 则是通过队列的方式从点击的星星开始进行广度扩展，每次从队列中取出一个星星，访问其相邻的星星，并将相同颜色的星星加入队列中继续扩展。

### （3）并查集

Union-find并查集用于将相连的星星分为一个集合。通过并查集，我们可以在初始化时找到所有相邻的同色星星，并将其合并成一个集合。在点击某个星星时，整个集合都会被消除。

## 123总结

- **DFS** 和 **BFS** 都是通过遍历的方式处理相连的星星，DFS适合**递归**实现，BFS适合**迭代**处理。

- **并查集** 则是在**初始化时**将所有相邻同色的星星进行合并，通过高效的查找操作快速找到并消除同一组星星。

这三种方法都有各自的优缺点，DFS 和 BFS 都是  $O(nm)$  复杂度，适合**小规模问题**；并查集对于**大量查询消除块**的问题较为高效。

## 校园祭打卡

莫名想到compilers课上学到的变量的活跃区间，确实是形式上是类似的，但是“不动点算法”不能应用到这道题上。

搜到了leetcode上一道题【ps.还是要刷力扣啊！】452

### 1.1 vector的sort函数

【其实，本题不定义比较函数也可以】

原因1.start相同时，不必要按照end降序排序

原因2.即使vector是嵌套的，编译器会自己找到排序规则

```
1 #include <iostream>
2 #include <vector>
3 #include <algorithm> // std::sort
4
5 int main() {
6     // 初始化一个vector<vector<int>>的数据结构
7     std::vector<std::vector<int>> vec = {
8         {3, 4},
9         {2, 8},
10        {3, 1},
11        {1, 5},
12        {2, 3}
13    };
14
15    // 使用std::sort排序，按照第一个元素为主，第二个元素为次排序
16    std::sort(vec.begin(), vec.end(), [](const std::vector<int>& a, const
std::vector<int>& b) {
17        if (a[0] == b[0]) {
18            return a[1] < b[1]; // 当第一个元素相同时，按照第二个元素升序排序
19        }
20        return a[0] < b[0]; // 按照第一个元素升序排序
21    });
22
23    // 输出排序后的结果
24    for (const auto& v : vec) {
25        std::cout << "[" << v[0] << ", " << v[1] << "]" << std::endl;
26    }
```

```
27
28     return 0;
29 }
```

`std::sort(vec.begin(), vec.end(), ...)` 是用于对整个 `vec` 进行排序的语句。

使用 Lambda 表达式 `[](const std::vector<int>& a, const std::vector<int>& b)` 来定义排序规则：

- 如果 `a[0] == b[0]`，表示它们的第一个元素相同，此时按第二个元素 `a[1]` 和 `b[1]` 的大小进行排序。
- 如果 `a[0] != b[0]`，则按照第一个元素进行排序。

或者，

```
1 private:
2     static bool cmp(const vector<int>& a, const vector<int>& b) {
3         return a[0] < b[0];
4     }
5 public:
6     int findMinArrowShots(vector<vector<int>>& points) {
7         if (points.size() == 0) return 0;
8         sort(points.begin(), points.end(), cmp);
9     }
```

- 比较函数返回bool
- 比较函数参数是引用

## 1.2 换了一种贪心思路

如果将停留我们考虑所有区间里面，右端点最小的那个区间  $[l, r]$ ，然后考虑第一次访问的位置  $pos$ 。我们有贪心策略：

1.  $pos$  首先定为  $l$ .
2. 若走到  $l$  我们还没走满 27 的倍数，则继续走，直到最后一步走满 27, 取  $pos = align(pos, pre\_pos)$ .
3. 若  $pos$  超过  $r$ , 由于是第一次访问，这显然是不可接受的。  $pos = min(pos, r)$ .

如果只有这一个区间，第一次访问定在这显然是最优的。但其实在  $n$  个区间里这么走也同样最优。我们尝试着证明：

1. 首先  $pos$  肯定不能超过  $r$ .
2. 注意到后面所有区间的右端点都必定大于等于  $r$ , 设左端点为  $l'$ , 则有两种情况。
  1. 一种是  $l' \leq pos$ , 即  $pos$  落在该区间里面，我们可以顺便停留。
  2. 另一种是  $pos < l'$  的情况，该区间无法在  $pos$  停留。但我们此时除了  $pos$ ，有更好的停留点吗？没有。因为  $pos$  要么是  $r$ ，要么是走满了 27 步必须停下来。

因此我们将所有区间按照右端点升序排序，挨个遍历。对于未访问过的区间，我们计算  $pos$  并将为访问区间中所有与其有交的区间访问，并将访问标记为 1。复杂度  $O(n^2)$ 。

把数组按右边界排序

贪心1.每次尝试走到最近的一个右边界

贪心2.每次停留尽量打卡最多的摊位

## hw3

### 助教寻宝

#### 1.1 主要考点：函数指针

一开始想把不同策略做成函数重载，函数指针需要匹配参数类型最接近的策略函数。实际上这是不可行的，“函数指针自行匹配最接近的函数”尽量不要使用；后来gpt告诉的没有这样，策略使用不同函数名，但是相同的参数列表，函数指针依次指向不同策略。

#### 1.2 函数指针有点像策略模式！

类持有不同的策略类型

### 矩阵的幂

#### 1.1 自定义结构体

```
1 typedef struct{
```

```

2     int rownum;
3     int colnum;
4     int **data;
5 }Matrix;

```

## 1.2 free(): double free detected in tcache 2

**定义拷贝构造函数：**你需要为 `Matrix` 类定义一个深拷贝构造函数，以确保每次复制一个矩阵时，都会为 `data` 分配新的内存，而不是简单地复制指针。

## 1.3 ans = ans \* a;

ans是Matrix类型

**定义赋值运算符：**你还需要定义赋值运算符，以便在 `ans = ans * a` 这样的赋值操作中正确地管理内存。

## 1.4 申请动态内存+（必须）释放

```

1 Matrix(int r, int c){
2     rownum = r;
3     colnum = c;
4     data = new long long*[r];
5     for(int i = 0; i < r; ++i){
6         data[i] = new long long[c];
7     }
8
9     for(int i = 0; i < r; ++i){
10        for(int j = 0; j < c; ++j){
11            if(i == j){
12                data[i][j] = 1;
13            }else{
14                data[i][j] = 0;
15            }
16        }
17    }
18 }
19
20 ~Matrix(){
21     for(int i = 0; i < rownum; ++i){
22         delete[] data[i];
23     }
24     delete[] data;
25 }
26

```



## 1.5 重载运算符+ = \*还不一样

- 是否返回引用：为什么？问问gpt

```
1      Matrix operator*(const Matrix& other){
2          int m = rownum, n = colnum, l = other.colnum;
3          Matrix newMatrix(m, l);
4
5          for(int i = 0; i < m; ++i){
6              for(int j = 0; j < l; ++j){
7                  newMatrix.data[i][j] = 0;
8                  for(int k = 0; k < n; ++k){
9                      newMatrix.data[i][j] = (newMatrix.data[i][j] + ((data[i]
10 [k] % M) * (other.data[k][j] % M)) % M) % M;
11              }
12          }
13          return newMatrix;
14      }
15
16      Matrix& operator=(const Matrix& other){
17          if(this == &other) return *this;
18
19          for(int i = 0; i < rownum; ++i){
20              delete[] data[i];
21          }
22          delete[] data;
23
24          rownum = other.rownum;
25          colnum = other.colnum;
26          data = new long long*[rownum];
27          for(int i = 0; i < rownum; ++i){
28              data[i] = new long long[colnum];
29              for(int j = 0; j < colnum; ++j){
30                  data[i][j] = other.data[i][j];
31              }
32          }
33
34          return *this;
35      }
```

## 1.6 充分取余+longlong 避免越界

一开始答案错误，把int改成long long，就AC了

```
1 newMatrix.data[i][j] = (newMatrix.data[i][j] + ((data[i][k] % M) *
  (other.data[k][j] % M)) % M) % M;
```

## 1.7 取余技巧

- $(a + b) \% p = (a \% p + b \% p) \% p$ , 其中%表示取余数
  - $(a \times b) \% p = (a \% p \times b \% p) \% p$ , 其中%表示取余数

## 可编程计算器

### 1.1 对象的生命周期

```
1 //这是临时的对象，当函数返回时，这些局部变量已经被销毁，
2 //导致 treeStack 中存储的是指向无效内存的指针。
3 TreeNode treeNode(token);
4 treeStack.push(&treeNode);
5 //使用指针类型来创建节点并推送到栈中，而不是使用栈对象（局部变量）
6 TreeNode* treeNode = new TreeNode(token); // 使用 new 创建动态分配的节点
  treeStack.push(node);
7 treeStack.push(&treeNode);
```

### 1.2 lambda表达式

两个参考：

[https://www.bilibili.com/video/BV1fG41157Ew/?spm\\_id\\_from=333.337.search-card.all.click&vd\\_source=bdc50daafcab565c2482122d29bc9943](https://www.bilibili.com/video/BV1fG41157Ew/?spm_id_from=333.337.search-card.all.click&vd_source=bdc50daafcab565c2482122d29bc9943)

<https://oi-wiki.org/lang/lambda/>

```
1 functionTable[op[0]] = [this, l, r, expression](double a, double b){
2     registerConstant(l, a);
3     registerConstant(r, b);
4     return calculate(expression);
5 };
```

### 1.3 functionTable

```
1 typedef std::function<double(double, double)> BINARY_OP;
```

```

2
3 std::map<char, BINARY_OP> functionTable;    // 函数表
4
5 //初始化
6 functionTable['+'] = [](double a, double b ){return a + b;};
7 functionTable['-'] = [](double a, double b ){return a - b;};
8 functionTable['*'] = [](double a, double b ){return a * b;};
9 functionTable['/'] = [](double a, double b ){return (b != 0)? (a / b) : 0 ;};

```

## 1.4 stack的基本操作

```

1 //1、声明
2 stack<string> strStack;
3 //2、添加
4 strStack.push(str);
5 //3、删除
6 strStack.pop();    //注意：返回void
7 //4、查看栈顶元素
8 strStack.top();    //注意：只是查看，并未删除
9 //5、是否为空
10 strStack.empty()

```

## 1.5 map的基本操作

```

1 //1、声明
2 map<int, string> m;
3 //2、声明时初始化
4 map<int, string> ID_Name = {
5     { 2015, "Jim" },
6     { 2016, "Tom" },
7     { 2017, "Bob" } };
8 //3、添加
9 map[1] = "hello";
10 //4、清空
11 map.clear();
12 //5、查看一个key是否存在
13 map.count(i) != 0;
14 map.find(i) != map.end();
15
16 //注意map[]和map.find()的区别
17 std::map<int, std::string> m;
18 m[1] = "Hello";    // 键 1 不存在，会插入一个新元素
19 m[2] = "World";    // 键 2 不存在，会插入一个新元素

```

```

20 std::cout << m[1]; // 输出 "Hello"
21 std::cout << m[3]; // 输出空字符串, 因为插入了键 3 和默认构造的值 ""
22
23 std::map<int, std::string> m;
24 m[1] = "Hello"; // 插入键值对 1 -> "Hello"
25
26 auto it = m.find(1); // 查找键 1
27 if (it != m.end()) {
28     std::cout << it->second; // 输出 "Hello"
29 }
30
31 it = m.find(2); // 查找键 2 (不存在)
32 if (it == m.end()) {
33     std::cout << "Key 2 not found"; // 输出 "Key 2 not found"
34 }
35

```

## 1.6 string类型转换

```

1 stod(str1); //转换为double
2 stoi(str2); //转换为int

```

【注意】不符合格式时，比如str1是“c1”，无法转换为double，会报错参数无效

## 图书馆管理系统

### 1.1 运行异常：访问冲突

```

1 double calculate_fee(int days) const {
2     // todo: add codes here
3     return borrow_strategy->calculate(days);
4 }

```

本地报错信息为

Process finished with exit code -1073741819 (0xC0000005)

**访问冲突**（Access Violation）。此错误通常是由于程序尝试访问无效的内存地址或执行了非法操作导致的。

可能原因：

空指针或悬空指针（Dangling Pointer）访问

数组越界

非法访问内存

递归调用过深（栈溢出）

指针类型错误（类型转换问题）

静态成员或全局变量的初始化问题

修改后：

```
1     double calculate_fee(int days) const {
2         // todo: add codes here
3         if(borrow_strategy){
4             return borrow_strategy->calculate(days);
5         }
6         return 0.0;
7     }
```

## 1.2 高级for循环可能遍历到end()

```
1     for(auto book : books){
2         fee += book.second->calculate_fee(days);
3     }
```

# 角色管理

## 1.1 悬空指针

```
1 Warrior temp = *new Warrior(id, "Warrior");
2 roles[id] = &temp;
```

临时对象 `temp` 会在函数结束时被销毁，而你将其指针存储在 `roles` 中，这会导致 **悬空指针** 问题，随后调用 `print` 或其他方法可能导致未定义行为。

解决方法：

```
1 roles[id] = new Warrior(id, "Warrior");
```

## 1.2 虚函数和纯虚函数

```
virtual void upgrade();  
virtual double getPower();
```

### 问题解析


#### 1. 虚函数没有定义:

- 你声明了 `Role` 类的虚函数 `upgrade` 和 `getPower`，但没有在类中提供它们的定义。
- 在 C++ 中，如果一个类声明了虚函数但没有提供定义，链接器会报出 `undefined reference` 错误。
- 这是因为虚函数需要一个具体的实现，哪怕这个实现只是一个空函数。

#### 2. 纯虚函数和抽象类:

- 如果 `Role` 类是设计为抽象基类（仅用于继承），你可以将这两个虚函数声明为纯虚函数：

cpp

 复制代码

```
virtual void upgrade() = 0;  
virtual double getPower() = 0;
```

- 这样，`Role` 类不能直接实例化，但其派生类（如 `Warrior`，`Mage`，`Rogue`）必须提供这些函数的实现。

## hw4

### 文本编辑器

#### 1.1 双端队列deque

```
1 deque<char> q;  
2 int cursor;  
3 q.insert(cursor, c); //x  
4 q.insert(q.begin()+cursor, c); //√
```

## 1. 两端插入操作:

- `push_back(elem);` —— 在容器尾部添加一个数据
- `push_front(elem);` —— 在容器头部插入一个数据
- `pop_back();` —— 删除容器最后一个数据
- `pop_front();` —— 删除容器第一个数据

## 2. 指定位置操作:

- `insert(pos, elem);` —— 在pos位置插入一个 `elem` 元素的拷贝，返回新数据的位置。
- `insert(pos, n, elem);` —— 在pos位置插入n个elem数据，无返回值。
- `insert(pos, beg, end);` —— 在pos位置插入[beg, end)区间的数据，无返回值。
- `clear();` —— 清空容器的所有数据
- `erase(beg, end);` —— 删除[beg, end)区间的数据，返回下一个数据的位置。
- `erase(pos);` —— 删除pos位置的数据，返回下一个数据的位置。

# 沙威玛商店评价

用时2小时53分，考试难度可能和这个差不多。这次在输入处理上浪费了太多时间。

## 1.1 输入

注意点

1. `cin.ignore()`忽略换行符
2. 每行一起读（我一开始幻想先`cin>>inst`，再`getline()`，出错概率太大了）
3. `ss`一起输出到各个变量，不要到各个小函数里单独处理了（其实也不麻烦，总代码量没有变多，只是看起来丑一点）

```
1 int main(){
2     int n;
3     cin >> n;
4     cin.ignore();
5
6     for(int nn = 0; nn < n; nn++){
7         string inst;
8         string line;
```

```

9      getline(cin, line);
10     stringstream ss(line);
11     ss >> inst;
12     // cout <<"inst:"<< inst << endl;
13
14     if(inst == "insert"){
15         string time1, time2;
16         int id;
17         double foodr, servicer, environmentr;
18         ss >> id >> foodr >> servicer >> environmentr >> time1 >> time2;
19         insert(id, foodr, servicer, environmentr, time1, time2);
20     }else ...
21     ...
22 }
23 }

```

## 1.2 输出“最多”一位小数

在未雨绸缪里写

## 1.3 列表初始化

好像不管是什么类型都是全部{}

```

1 map<string, double> rating = {"food", 0}, {"service", 0}, {"environment", 0}};
2 string dimensions[3] = {"food", "service", "environment"};

```

## 1.4 看map是否有这个键值 / 找链表结尾

```

1 RevNode* p = customerList[id];
2 if(p == nullptr){//ver = 1
3     customerList[id] = new RevNode(id, foodr, servicer, environmentr,
4     time1, time2, ver);
5 }else{
6     while(p->next != nullptr){
7         ver++;
8         p = p->next;
9     }
10    //此时p是链表的结尾
11    p->next = new RevNode(id, foodr, servicer, environmentr, time1, time2,
12    ver);
13 }

```



# 并发进程

## 1.1 参考：

互斥锁mutex <https://www.cnblogs.com/chentiao/p/18468699>

原子变量atomic<> <https://developer.aliyun.com/article/1605238>

条件变量conditon\_variable <https://zhuanlan.zhihu.com/p/608499318>

RAII思想 <https://blog.csdn.net/wsqianguz/article/details/134098624>

# 写时复制树

## 0. 参考和讲解

非递归 [https://blog.csdn.net/qg\\_58887972/article/details/135708757?spm=1001.2014.3001.5502](https://blog.csdn.net/qg_58887972/article/details/135708757?spm=1001.2014.3001.5502)

递归 <https://blog.csdn.net/Aft3rGl0w/article/details/135704274>

环境配置clang12 clion wsl <https://zhuanlan.zhihu.com/p/592802373>

无源码但高手参考

<https://zhuanlan.zhihu.com/p/622663820>

最后用例全通过是参考：<https://blog.csdn.net/Aft3rGl0w/article/details/135704274>?

ops\_request\_misc=%257B%2522request%255Fid%2522%253A%2522fc6098c2554ec0ccc327664eaae1202a%2522%252C%2522scm%2522%253A%252220140713.130102334.pc%255Fall.%2522%257D&request\_id=fc6098c2554ec0ccc327664eaae1202a&biz\_id=0&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~first\_rank\_ecpm\_v1~rank\_v31\_ecpm-1-135704274-null-null.142

## 1.1 智能指针

cur\_node是 std::shared\_ptr<const TrieNode> 类型，得到指针需要 cur\_node.get()

| auto value\_node = dynamic\_cast<const TrieNodeWithValue<T> \*>(cur\_node.get());

因为T可能是不可复制类型，所以请总是使用`std::move`，当创建T的`shared\_ptr`时

| std::shared\_ptr<T> value\_ptr = std::make\_shared<T>(std::move(value));

## 1.2 注意map用法

1) map[]和map.find()的区别

2)

```
1 //children_是map
2 auto cur_node = root_;
```

```

3     for(auto ch : key){
4         auto it = cur_node->children_.find(ch);
5         if(it == cur_node->children_.end()){
6             return nullptr;
7         }
8         cur_node = it->second; //注意指针是pair类型
9     }

```

3) 我感觉不常用，因为失败的时候会直接抛出异常

C++ map at()函数用于通过给定的键值访问map中的元素。如果map中不存在所访问的键，则抛出 out\_of\_range异常。

## 1.3 终于能在wsl+clion运行起来了

### 困难1-clang环境配置

上面的环境参考链接

困难2-fmt/core.h找不到，一开始以为是c++版本支持的问题，后来gpt才知道和cmakelist文件有关

```

1 find_package(fmt REQUIRED)
2 target_link_libraries(trie fmt::fmt)
3
4 find_package(Threads REQUIRED)
5 target_link_libraries(trie pthread)

```

## 1017课堂练习-不重叠时间段

9:53-10:29 38min

### 0. 和校园祭打卡非常像

校园祭打卡按照结束位置排序，这一题按照开始位置排序

运用贪心算法，局部最优是什么我不知道

因为有重叠了n个时间段的某个时间，就**必定**要去除n-1个时间段。局部最优是留下结束时间最靠前的时间段，减少对n之后的时间段的影响

### 1. getline只接受string作为参数

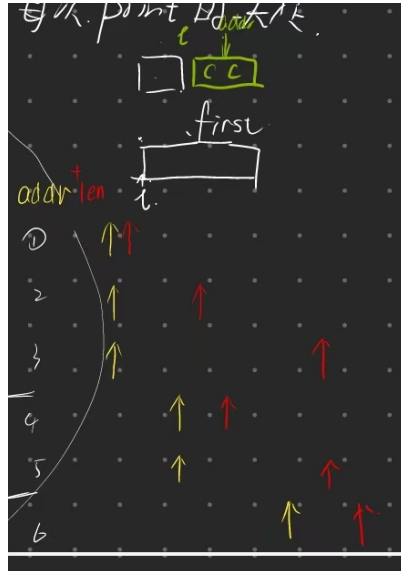
# 1024课堂练习

## 铺木地板

没有考察算法，以为是很复杂的规划

遇到del时更新标记，每次打印时都遍历分析就可以了

复杂之处是分支判断



# 1121课堂练习

## 1.1 继承关系中的构造函数

派生类的构造函数必须显式调用基类的构造函数来初始化基类的成员。如果基类没有默认构造函数（即无参数的构造函数），派生类的构造函数需要通过**初始化列表**调用基类的构造函数。

下面每个例子都值得细品：

```
1 class Role{
2     public:
3     int hp;
4     int atk;
5     Role(int h, int a){
6         hp = h;
7         atk = a;
8     }
9 };
10 class Veg : public Role{
11     public:
12     Veg(int h, int a, string veg_type) : Role(h, a) {}
13 };
14 class Pea : public Veg{
```

```

15     public:
16     Pea(int h, int a) : Veg(h, a, "pea") {}
17 };
18
19 class Nut : public Veg{
20     public:
21     Nut(int h) : Veg(h, 0, "nut"){ }
22 };
23 class Enemy : public Role{
24     public:
25     int speed;
26     int addr;
27
28     Enemy(int h, int a, int s, int ad) : Role(h, a), speed(s), addr(ad){ }
29 };

```

## 1.2 手动（堆）自动（栈）管理内存

背景问题：

```

1 vector<vector<Veg*>> vega;
2 system.vega[x][y] = new Pea(hp, atk);
3 vega.resize(line, vector<Veg*>(col, nullptr));

```

会提示“请避免手动分配对象”

原因：

`new` 会分配动态内存，但你需要显式使用 `delete` 释放内存。如果忘记释放，就会导致内存泄漏。如果 `system.vega[x][y]` 已经指向其他动态对象，而直接赋值新对象指针，旧对象的内存可能未被释放，造成资源泄露。

解决：

### （1）使用智能指针管理对象

`system.vega[x][y]` 是一个**指针**，可以将其定义为智能指针（如 `std::unique_ptr`）

```

1 vector<vector<unique_ptr<Veg>>> vega;
2 vega.resize(rows, vector<unique_ptr<Veg>>(col));
3 vega[x][y] = make_unique<Pea>(hp, atk); // 自动管理对象内存

```

优点：

- `std::unique_ptr` 会在超出作用域时自动释放内存。
- 防止内存泄漏，不需要显式 `delete`。

## (2) 使用栈对象

如果 `vegs[x][y]` **不需要是指针**，可以直接在栈上存储对象，避免动态内存分配

```
1 vector<vector<optional<Pea>>> vegs; // 使用 std::optional 表示可能为空
2 vegs.resize(rows, vector<optional<Pea>>(col));
3 vegs[x][y] = Pea(hp, atk); // 在栈上构造对象
```

优点：

- 无需显式管理内存。
- 栈上对象的生命周期与作用域绑定，简单高效

## (3) 使用 `std::shared_ptr` 管理共享资源

如果多个地方需要共享同一对象，可以使用 `std::shared_ptr`

```
1 vector<vector<shared_ptr<Veg>>> vegs;
2 vegs.resize(rows, vector<shared_ptr<Veg>>(cols));
3 vegs[x][y] = make_shared<Pea>(hp, atk); // 共享对象
```

注意：

- `std::shared_ptr` 适用于共享所有权的场景，但稍微增加了内存和性能开销。
- 如果只有一个地方持有对象，推荐使用 `std::unique_ptr`。

## 1.3 二维vector初始化

```
vegs.resize(line, vector<unique_ptr<Veg>>(col, nullptr));
```

报错：In template: static\_assert failed due to requirement

'is\_constructible<std::unique\_ptr<Veg, std::default\_delete<Veg>>, const std::unique\_ptr<Veg, std::default\_delete<Veg>> &::value' "result type must be constructible from value type of input range"

报错的原因是 `std::unique_ptr` 不支持复制操作

`std::vector` 的 `resize` 方法会尝试用默认值初始化元素，但 `std::unique_ptr` 无法被复制，因此会引发编译错误。

解决：

```
1 vogs.resize(line); // 初始化行数
2 for (int i = 0; i < line; ++i) {
3     vogs[i].resize(col); // 初始化每行的列数
4 }
```

## 1205课堂练习

当答案错误却不知道是什么相关的用例出了问题时，投机取巧：让某个用例相关代码运行异常，当重新提交，错误数量减少、异常数量增加时，就知道刚刚改动代码确实存在答案错误

## 1212课堂练习

### 1.1 一开始错误原因是：没有用string类重载的<> <=等运算符，而是手动的挨个字母判断

还需要复习一下string类的比较运算符规则

### 1.2 大小写转换

```
1 #include <cctype>
2     for (char& c : str) { //注意！是引用！！
3         c = std::toupper(c); // 将每个字符转换为大写
4         c = std::tolower(c); //大-》小
5     }
```

## 1219课堂练习-卡片游戏

### 1. 二分查找遇到的溢出

因为这次的二分查找分的不是索引，而是card上的数字，而数字有达到INT\_MAX的情况，所以第一行可以保证正确性，而第二行会因为max+min超过了INT\_MAX而溢出

当然，把min max mid 都改成long类型也能通过用例

```
1     int mid = min + (max-min) / 2;
2     int mid = (max+min) / 2;
```

### 2. 这是一个难的二分查找

其中的单调关系和二分查找判断函数我不是很懂

函数的目标是判断是否可以从给定的卡片中选择 `k` 张卡片，并且这 `k` 张卡片之间不相邻，并且卡片的数字都小于等于给定的最大值 `score`。

```
1 bool findkcards(int score){
2     int lastindex = -2;
3     int count = 0;
4     for(int i = 0; i < n; i++){
5         if(cards[i] <= score && (i-lastindex) > 1){
6             count ++;
7             lastindex = i;
8             if(count >= k){
9                 return true;
10            }
11        }
12    }
13    return false;
14 }
```

函数为什么正确？

这是一个有效的贪心策略，确保我们在每次选择时都尽量选择较小的数字（不超过 `x`），从而尽量保持满足选择 `k` 张不相邻的卡片的条件。

## 其它

`vector` 初始化方法：

### 1. 默认构造

`std::vector` 可以通过默认构造函数创建一个空的向量。

```
std::vector<int> vec; // 创建一个空的vector, size = 0
```

### 2. 指定大小初始化

可以通过构造函数指定一个初始大小，`vector` 会创建一个包含指定数量元素的容器，并使用默认构造函数初始化每个元素。

```
std::vector<int> vec(5); // 创建一个包含 5 个元素的 vector, 默认值为 0
```

### 3. 指定大小和初始值

除了指定大小，还可以指定每个元素的初始值。这样 `vector` 会创建指定数量的元素，并使用指定的值进行初始化。

```
std::vector<int> vec(5, 10); // 创建一个包含 5 个元素的 vector, 每个元素值为
```

#### 4. 使用迭代器初始化

`vector` 也可以通过一个范围（例如，另一个容器的迭代器范围）来初始化。这种方式是通过复制其他容器的数据来创建 `vector`。

```
std::vector<int> vec1 = {1, 2, 3, 4, 5};  
std::vector<int> vec2(vec1.begin(), vec1.end()); // 使用 vec1 的迭代器范围初始化 vec2
```

#### 5. 使用列表初始化（C++11 及以上）

使用列表初始化，可以通过花括号 `{}` 初始化一个 `vector`，并指定初始值。对于 C++11 及以上版本，可以通过这种方式简洁地初始化 `vector`。

```
std::vector<int> vec = {1, 2, 3, 4, 5}; // 使用列表初始化，vec 初始化为 {1, 2, 3, 4, 5}
```

#### 6. 拷贝构造初始化

可以通过另一个 `vector` 对象来初始化一个新的 `vector`，这种方式会进行元素的拷贝。

```
std::vector<int> vec1 = {1, 2, 3, 4, 5};  
std::vector<int> vec2 = vec1; // 使用 vec1 来初始化 vec2，拷贝所有元素
```

#### 7. 移动构造初始化（C++11 及以上）

如果你希望在初始化时避免不必要的拷贝操作，可以使用移动构造函数。这样可以把已有 `vector` 的内容转移到新 `vector` 中。

```
std::vector<int> vec1 = {1, 2, 3, 4, 5};  
std::vector<int> vec2 = std::move(vec1); // vec2 通过移动构造初始化，vec1 的内容被移到 vec2，vec1 变为空
```

#### 8. 使用 `std::initializer_list` 初始化

通过 `std::initializer_list`，你可以使用初始化列表初始化 `vector`。这是一种非常简洁且常用的方式，尤其在 C++11 中得到了广泛的应用。

```
std::vector<int> vec = {1, 2, 3, 4, 5}; // 使用初始化列表进行初始化
```

#### 9. 使用 `assign` 方法初始化

`std::vector` 提供了 `assign` 方法，允许用指定值或指定范围的元素来重新初始化 `vector`。

- 使用指定值：

- ```
std::vector<int> vec;  
vec.assign(5, 10); // 将 vec 初始化为包含 5 个值为 10 的元素
```

- 使用指定范围：

- ```
std::vector<int> vec1 = {1, 2, 3, 4, 5};  
std::vector<int> vec2;  
vec2.assign(vec1.begin(), vec1.end()); // 使用 vec1 的范围初始化 vec2
```



## 总结

- 默认构造: `std::vector<int> vec;`
- 指定大小初始化: `std::vector<int> vec(5);`
- 指定大小和初始值: `std::vector<int> vec(5, 10);`
- 使用迭代器初始化: `std::vector<int> vec(vec1.begin(), vec1.end());`
- 使用列表初始化 (C++11 及以上) : `std::vector<int> vec = {1, 2, 3};`
- 拷贝构造: `std::vector<int> vec2 = vec1;`
- 移动构造: `std::vector<int> vec2 = std::move(vec1);`
- 使用 `assign` 方法: `vec.assign(5, 10);` 或 `vec.assign(vec1.begin(), vec1.end());`