# 20230912

1. math函数中arc反函数来定义pi很有用，可以精准控制pi的值
2. 字符串的操作

memset

memset(void\*s ,int ch,size\_t n);

函数解释：将s的前n个字节用ch替换并返回s。

“memset（a，0，sizeof（a））”的作用是把数组a清零，它也在string.h中定义。虽然也能用for循环完成相同的任务，但是用memset又方便又快捷。

memcpy

memcpy(void\* d,const void\*s,size\_t n)

函数解释：从s所指的地址的起始位置开始拷贝n个字节到d所指的内存地址的起始位置中。一般用法是：memcpy（b，a，sizeof（a））。

memcmp

memcmp(const void\*buf1,const void \*buf2,unsigned int count);

函数解释：比较内存区域buf1和buf2的前count个字节

当buf1<buf2时，返回值小于0

当buf1==buf2时，返回值=0

当buf1>buf2时，返回值大于0

memmove

memmove(void\* dest,const void\* src,size\_t count);

函数解释：由src所指的内存区域复制count个字节到dest所指的内存区域。

memcpy和memmove的区别：

他们的作用是一样的，唯一的区别是，当内存发生局部重叠的时候，memmove保证拷贝的结果是正确的，memcpy不保证拷贝的结果的正确。另外，源和目的的内存区域不能重叠。

# 20230914

## 头文件



## 2. using namespace std 和<iostream>

因为标准库非常的庞大，所程序员在选择的类的名称或函数名时就很有可能和标准库中的某个名字相同。所以为了避免这种情况所造成的名字冲突，就把**标准库中的一切都被放在名字空间std**中。但这又会带来了一个新问题。无数原有的C++代码都依赖于使用了多年的伪标准库中的功能，他们都是在全局空间下的。

        所以就有了<iostream.h>和<iostream>等等这样的头文件，一个是为了兼容以前的C++代码，一个是为了支持新的标准。命名空间std封装的是标准程序库的名称，**后缀为.h的头文件c++标准已经明确提出不支持了，标准程序库为了和以前的头文件区别，一般不加".h"**

## EPS

在不断减1e-6的过程中出现了误差，使得循环终止时f并不等于1，而是比1小一点。在除以4保留1位小数时成了0.2。如果不出现误差，正确答案应该是0.25四舍五入保留一位小数，即0.3。一道好的竞赛题目应避免这种情况出现(12)，但作为竞赛选手来说，有一种方法可以缓解这种情况：**加上一个EPS以后再输出。这里的EPS通常取一个比最低精度还要小几个数量级的小实数。**例如，要求保留3位小数时取EPS为1e-6。这只是个权宜之计，甚至有可能起到“反作用”（如正确答案真的是0.499999），但在实践中很好用（毕竟正确答案是0.499999的情况比0.5要少很多）。

4.从C到C＋＋

和之前的C程序比较，唯一的区别是stdio.h变成了cstdio。事实上，stdio.h仍然存在，但是C＋＋中推荐的头文件是cstdio。类似地，string.h变成了cstring，math.h变成了cmath，ctype.h变成了cctype。带.h后缀的头文件依然存在，但并不被C＋＋所推荐使用

iostream提供了输入输出流，而algorithm提供了一些常用算

法，例如代码中的min

(1)Cin>>a的含义是从标注输入中读取a，它的返回值是一个“已经读取

了a的新流”，然后从这个新流中继续读取b。如果流已经读完，while循环将退出。这种方式相比scanf的最大优势就是不再需要记忆%d、%s等占位符，同时也避开了前面提到的“longlong类型的输入输出占位符不统一”的问题。当然，C＋＋流也不是完美的，其最大缺点就是运行太慢，以至于很多竞赛题目会在题面中的显著位置注明：本题的输入量很大，请不要使用C＋＋的流输入。5.STL

STL是指C＋＋的标准模板库（Standard Template Library）。

sort(a,a+n)（a是所需要排序的数组）

lower\_bound（a,a+n,x）作用是查找“大于或者等于x的第一个位置”（在已排序数组a中寻找x）

# 20230920

1.测试STL

如何产生[0，n]之间的整数呢？很多人喜欢用rand( )%n产生区间[0，n-1]内的一个随机整数，姑且不论这样产生的整数是否仍然分布均匀，只要n大于RAND\_MAX，此法就不能得到期望的结果。由于RAND\_MAX很有可能只有32767这么小，在使用此法时应当小心。另一个方法是执行rand( )之后先除以RAND\_MAX，得到[0，1]之间的随机实数，扩大n倍后四舍五入，得到[0，n]之间的均匀整数。这样，在n很大时“精度”不好（好比把小图放大后会看到“锯齿”），但对于普通的应用，这样做已经可以满足要求了。

# 20231017

（1）cout需要#include <iostream>和using namespace std;

（2）可以用1<<maxd来表示，因为10000就是2的4次方

（3）memset(s, 0, sizeof(s));把s数组全部赋值为0，建议只赋值为0.赋值为1会出现问题，但是-1可以

（4）使用Dev-C++查看数组中的变量值而不是数组地址

1.一维数组

比如说有一个长度为10的int solution[10]

如果想要查看solution[0]的值，就在 “添加查看” 中写 \*(&solution[0])

如果想要查看整个数组的值，就可以写\*(&solution[0])@10

@后面的数字表示想要查看的长度，这里数组的长度是10所以可以写10就能看到所有的值了

ps： vector也适用

2.二维数组

其实道理和一维数组一个，二维数组本质还是一维数组嘛

比如有一个二维数组int result[15][10]

如果想看第0行的值，就写\*(&result[0][0])@10

如果想随着变量i看第i行的值，可以写\*(&result[i][0])@10

isalpha()用来判断一个字符是否为字母

isalnum用来判断一个字符是否为数字或者字母，也就是说判断一个字符是否属于a~ z||A~ Z||0~9。

isdigit() 用来检测一个字符是否是十进制数字0-9

islower()用来判断一个字符是否为小写字母，也就是是否属于a~z。

isupper()和islower 相反，用来判断一个字符是否为大写字母。



# 20231123

## BFS

#include <iostream>

#include <queue>

using namespace std;

const int MAX\_N = 100;

const int MAX\_M = 100;

const int INF = 0x3f3f3f3f;

typedef pair<int, int> P;

char maze[MAX\_N][MAX\_M + 1];

int N, M;

int sx, sy; //起点的位置

int gx, gy; //终点的位置

int d[MAX\_N][MAX\_M];//储存起点到某一点的距离

int dx[4] = { 1,0,-1,0 }, dy[4] = { 0,1,0,-1 }; //表明每次x和y方向的位移

void bfs()

{

queue<P> que;

for (int i = 0; i < N; i++)

for (int j = 0; j < M; j++)

d[i][j] = INF; //初始化所有点的距离为INF

que.push(P(sx, sy));

d[sx][sy] = 0; //从起点出发将距离设为0，并放入队列首端

while (que.size()) //题目保证有路到终点，所以不用担心死循环

{

P p = que.front();

que.pop();//弹出队首元素

int i;

for (i = 0; i < 4; i++)

{

int nx = p.first + dx[i];

int ny = p.second + dy[i];//移动后的坐标

//判断可移动且没到过

if (0 <= nx&&nx < N

&& 0 <= ny&&ny < M

&&maze[nx][ny] != '#'

&&d[nx][ny] == INF)//之前到过的话不用考虑，因为距离在队列中递增，肯定不会获得更好的解

{

que.push(P(nx, ny)); //可以移动则设定距离为之前加一，放入队列

d[nx][ny] = d[p.first][p.second] + 1;

if(nx==gx && ny==gy) break;

}

}

if(i!=4) break;

}

}

int main()

{

cin>>N>>M;

for (int i = 0; i < N; i++)

cin>>maze[i];

for (int i = 0; i < N; i++)

for (int j = 0; j < M; j++)

{

if (maze[i][j] == 'S')

{

sx = i; sy = j;

}

if (maze[i][j] == 'G')

{

gx = i; gy = j;

}

}

bfs();

cout<<d[gx][gy]<<endl;

return 0;

}

CIN:

5 5

####G

..##.

#..#.

##...

#S.#.

1. pair<int, int>

pair<int, int> 是一个C++标准库中的模板类，用于表示一个包含两个整数的对。它有两个公共成员变量：first 和 second，分别表示对中的第一个和第二个整数。

1. typedef pair<int, int> P;

是C++中的类型定义语句。它表示将 `pair<int, int>` 这个类型定义为 `P`，这样在后续的代码中就可以使用 `P` 来代替 `pair<int, int>`。这样可以提高代码的可读性，使代码更加简洁。

1. typedef pair<int, int>

P;queue<P> que;

这段代码定义了一个名为 `P` 的类型，它是一个包含两个整数的 `pair` 类型。然后，它创建了一个名为 `que` 的队列，该队列的元素类型为 `P`。这意味着 `que` 可以存储由两个整数组成的对。

C++的STL中提供了一个库函数next\_permutation。 看看下面的代

码片段， 就会明白如何使用它了。

#include<cstdio>

#include<algorithm> //包含next\_permutation

using namespace std;

int main( ) {

int n, p[10];

scanf("%d", &n);

for(int i = 0; i < n; i++) scanf("%d", &p[i]);

sort(p, p+n); //排序， 得到p的最小排列

do {

for(int i = 0; i < n; i++) printf("%d ", p[i]); //输出排列p

printf("\n");

} while(next\_permutation(p, p+n)); //求下一个排列

return 0;

}需要注意的是， 上述代码同样适用于可重集。

枚举排列的常见方法有两种： 一是递归枚举， 二是用STL中的next\_permutation。

注意：如果要完全遍历的话next\_permutation要配合sort(p, p+n);

一起使用

# 20231128

void print\_subset\_1(int n, int\* A, int cur) {

for(int i = 0; i < cur; i++) printf("%d ", A[i]); //打印当前集合

printf("\n");

int s = cur ? A[cur-1]+1 : 0; //确定当前元素的最小可能值

for(int i = s; i < n; i++) {

A[cur] = i;

print\_subset\_1(n, A, cur+1); //递归构造子集

}

}

void print\_subset\_2(int n, int\* B, int cur) {

if(cur == n) {

for(int i = 0; i < cur; i++)

if(B[i]) printf("%d ", i); //打印当前集合

printf("\n");

return;

}

B[cur] = 1; //选第cur个元素

print\_subset\_2(n, B, cur+1);

B[cur] = 0; //不选第cur个元素

print\_subset\_2(n, B, cur+1);

}

子集生成实际上是把数组的序号通过递归的方式来生成一遍

# 20240401

isalpha()用来判断一个字符是否为字母

isalnum用来判断一个字符是否为数字或者字母，也就是说判断一个字符是否属于a~ z||A~ Z||0~9。

isdigit() 用来检测一个字符是否是十进制数字0-9

islower()用来判断一个字符是否为小写字母，也就是是否属于a~z。

isupper()和islower 相反，用来判断一个字符是否为大写字母。



//#include <iostream>

//#include <queue>

//using namespace std;

//const int MAX\_N = 100;

//const int MAX\_M = 100;

//const int INF = 0x3f3f3f3f;

//

//typedef pair<int, int> P;

//char maze[MAX\_N][MAX\_M + 1];

//int N, M;

//int sx, sy; //起点的位置

//int gx, gy; //终点的位置

//

//int d[MAX\_N][MAX\_M];//储存起点到某一点的距离

//int dx[4] = { 1,0,-1,0 }, dy[4] = { 0,1,0,-1 }; //表明每次x和y方向的位移

//

//void bfs()

//{

// queue<P> que;

// for (int i = 0; i < N; i++)

// for (int j = 0; j < M; j++)

// d[i][j] = INF; //初始化所有点的距离为INF

// que.push(P(sx, sy));

// d[sx][sy] = 0; //从起点出发将距离设为0，并放入队列首端

//

// while (que.size()) //题目保证有路到终点，所以不用担心死循环

// {

// P p = que.front();

// que.pop();//弹出队首元素

// int i;

// for (i = 0; i < 4; i++)

// {

// int nx = p.first + dx[i];

// int ny = p.second + dy[i];//移动后的坐标

// //判断可移动且没到过

// if (0 <= nx&&nx < N

// && 0 <= ny&&ny < M

// &&maze[nx][ny] != '#'

// &&d[nx][ny] == INF)//之前到过的话不用考虑，因为距离在队列中递增，肯定不会获得更好的解

// {

// que.push(P(nx, ny)); //可以移动则设定距离为之前加一，放入队列

// d[nx][ny] = d[p.first][p.second] + 1;

// if(nx==gx && ny==gy) break;

//

// }

// }

// if(i!=4) break;

// }

//

//}

//

//int main()

//{

// cin>>N>>M;

// for (int i = 0; i < N; i++)

// cin>>maze[i];

// for (int i = 0; i < N; i++)

// for (int j = 0; j < M; j++)

// {

// if (maze[i][j] == 'S')

// {

// sx = i; sy = j;

// }

// if (maze[i][j] == 'G')

// {

// gx = i; gy = j;

// }

// }

// bfs();

// cout<<d[gx][gy]<<endl;

//

// return 0;

//}

1. pair<int, int>

pair<int, int> 是一个C++标准库中的模板类，用于表示一个包含两个整数的对。它有两个公共成员变量：first 和 second，分别表示对中的第一个和第二个整数。

1. typedef pair<int, int> P;

是C++中的类型定义语句。它表示将 `pair<int, int>` 这个类型定义为 `P`，这样在后续的代码中就可以使用 `P` 来代替 `pair<int, int>`。这样可以提高代码的可读性，使代码更加简洁。

1. typedef pair<int, int>

P;queue<P> que;

这段代码定义了一个名为 `P` 的类型，它是一个包含两个整数的 `pair` 类型。然后，它创建了一个名为 `que` 的队列，该队列的元素类型为 `P`。这意味着 `que` 可以存储由两个整数组成的对。

# 20240628

## Vector使用的注意事项：

### 添加元素

可以使用 push\_back 方法向 vector 中添加元素：

myVector.push\_back(7); // 将整数 7 添加到 vector 的末尾

### 访问元素

可以使用下标操作符 [] 或 at() 方法访问 vector 中的元素：

int x = myVector[0]; // 获取第一个元素

int y = myVector.at(1); // 获取第二个元素

### 获取大小

可以使用 size() 方法获取 vector 中元素的数量：

int size = myVector.size(); // 获取 vector 中的元素数量

### 迭代访问

可以使用迭代器遍历 vector 中的元素：

for (auto it = myVector.begin(); it != myVector.end(); ++it) {

std::cout << \*it << " ";

}

或者使用范围循环：

for (int element : myVector) {

std::cout << element << " ";

}

### 删除元素

可以使用 erase() 方法删除 vector 中的元素：

myVector.erase(myVector.begin() + 2); // 删除第三个元素

### 清空 Vector

可以使用 clear() 方法清空 vector 中的所有元素：

myVector.clear(); // 清空 vector

### Vector代码

merge(nums1, m, nums2, n);

vector<int> nums1 {1, 2, 3, 0, 0, 0};

vector<int> nums2 {2, 5, 6};

void merge(vector<int> &nums1, int m, vector<int> &nums2, int n) {

// for (int i = 0; i < n; i++)

// nums1[i + m] = nums2[i];

// sort(nums1.begin(), nums1.end());

**//vector进函数vector<int> &nums1**

}

int main(int argc, char \*\*argv) {

int m, n;

m = 3;

n = 3;

for (int k = 0; k < n + m; k++)

cout << nums1[k] << ' ';**//可以类似数组调用用来输出**

cout << '\n';

for (int k = 0; k < n ; k++)

cout << nums2[k] << ' ';

cout << '\n';

merge(nums1, m, nums2, n); **//vector进函数直接输入进去就行**

}

## 动态规划

动态规划的基本思想是：问题的最优解如果可以由子问题的最优解推导得到，则可以先求解子问题的最优解，在构造原问题的最优解；若子问题有较多的重复出现，则可以自底向上从最终子问题向原问题逐步求解。

比如说著名的斐波那契定理a[i]=a[i-1]+a[i-2]，就是一个特别好的理解方式，为什么直接写公式的算法（自底向上，从1开始递增）效率就是比递归（自顶向下，从n开始向下）的要好？

让我们来熟悉一下动态规划的特点：

a) 把原始问题划分成一系列子问题；

b) 求解每个子问题仅一次，并将其结果保存在一个表中，以后用到时直接存取，不重复计算，节省计算时间

c) 自底向上地计算。

d) 整体问题最优解取决于子问题的最优解（状态转移方程）（将子问题称为状态，最终状态的求解归结为其他状态的求解）

# 20241020

## 2024蓝桥杯省赛研究生组 B题 召唤数字精灵

要解决这个问题，我们需要理解表达式 \( x! - \frac{x(x+1)}{2} \) 的含义。这个表达式实际上结合了两个概念：阶乘 \( x! \) 和前 \( x \) 个自然数的和 \( \frac{x(x+1)}{2} \)，后者是等差数列求和公式。

对于 \( x! \)，当 \( x \geq 5 \) 时，\( x! \) 将包含至少一个因子 5 和至少两个因子 2，因此 \( x! \) 能被 10 整除，并且对于更大的 \( x \)，\( x! \) 将包含更多的 5 和 2 的因子，所以 \( x! \) 也能被 100 整除。

现在我们来考虑 \( \frac{x(x+1)}{2} \)。这个表达式的结果是否能被 100 整除取决于 \( x \) 和 \( x+1 \) 中是否包含足够的因子 2 和 5（总共至少三个因子 2 和至少一个因子 5）。

为了使 \( x! - \frac{x(x+1)}{2} \) 能被 100 整除，\( \frac{x(x+1)}{2} \) 必须在模 100 下与 \( x! \) 相同。因为 \( x! \) 在 \( x \geq 5 \) 后总是能被 100 整除，所以我们只需要关注 \( x < 5 \) 的情况以及 \( \frac{x(x+1)}{2} \) 是否能被 100 整除的情况。

对于 \( x = 1, 2, 3, 4 \)，我们可以手动检查：

- \( x = 1 \): \( 1! - \frac{1(1+1)}{2} = 1 - 1 = 0 \)，显然可以被 100 整除。

- \( x = 2 \): \( 2! - \frac{2(2+1)}{2} = 2 - 3 = -1 \)，不能被 100 整除。

- \( x = 3 \): \( 3! - \frac{3(3+1)}{2} = 6 - 6 = 0 \)，可以被 100 整除。

- \( x = 4 \): \( 4! - \frac{4(4+1)}{2} = 24 - 10 = 14 \)，不能被 100 整除。

对于 \( x \geq 5 \)，\( x! \) 总是能被 100 整除，所以我们需要检查 \( \frac{x(x+1)}{2} \) 是否能被 100 整除。由于 \( x \) 和 \( x+1 \) 是连续的整数，它们中的一个是偶数，另一个是奇数，这意味着它们中最多只有一个能被 5 整除，并且至少有一个能被 2 整除。因此，只有当其中一个数能被 5 整除并且另一个数至少能被 4 整除时，\( \frac{x(x+1)}{2} \) 才能被 100 整除。

总结起来，在 1 到 2024041331404202 的范围内，只有当 \( x \) 或 \( x+1 \) 同时满足上述条件时，\( x \) 才是符合条件的数。考虑到这个问题的实际规模，可能需要编写程序来遍历所有数字并进行检查，或寻找更高效的数学方法来减少计算量。由于手动计算非常不现实，您可以编写代码来解决这个问题。如果您需要帮助编写这样的程序，请告诉我。

# 20241028

## 2024蓝桥杯省赛C组 F题 商品库存管理

初始商品库存均为0，执行完全部操作后分为2种情况：

商品库存<=1 //表示被0或1个区间包含

商品库存>1 //表示被1个以上的区间包含

因为每次只能撤回一次操作，等价于求[l,r]<=1的个数，还原差分数组时计算一下区间1的个数的前缀和就解决了

#include <bits/stdc++.h>

**using** **namespace** std**;**

const int N **=** 3e5 **+** 10**;**

pair**<**int**,** int**>**pr**[**N**];**

int diff**[**N**],** sum**[**N**],** ans **=** 0**;**

int main**()** **{**

int n**,** m**;**

cin **>>** n **>>** m**;**

**for** **(**int i **=** 0**;** i **<** m**;** i**++)** **{**

int a**,** b**;**

cin **>>** a **>>** b**;**

diff**[**a**]++,** diff**[**b **+** 1**]--;**

pr**[**i**]** **=** **{**a**,** b**};**

**}**

**for** **(**int i **=** 1**;** i **<=** n**;** i**++)** **{**

diff**[**i**]** **+=** diff**[**i **-** 1**];**

sum**[**i**]** **+=** sum**[**i **-** 1**]** **+** **(**diff**[**i**]** **<=** 1**);**

**if** **(**diff**[**i**]** **==** 0**)**

ans**++;**//记录未被操作的商品

**}**

**for** **(**int i **=** 0**;** i **<** m**;** i**++)**

cout **<<** sum**[**pr**[**i**].**second**]** **-** sum**[**pr**[**i**].**first **-** 1**]** **+** ans **<<** endl**;**

**return** 0**;**

**}**

### 前缀和

### 差分

### ios::sync\_with\_stdio(0),cin.tie(0),cout.tie(0)的注意事项

ios**::**sync\_with\_stdio**(**0**),** cout**.**tie**(**0**),** cin**.**tie**(**0**);**

是在C++中关闭输入输出流的同步，以提高程序的执行效率。

具体而言，它的作用如下：

1. 提高执行效率：默认情况下，C++的输入输出流与C标准库的输入输出函数是同步的，这会造成一定的性能损失。通过使用ios::sync\_with\_stdio(0)可以关闭这种同步，从而加快输入输出的速度，提高程序的执行效率。

2. 解绑输入输出流：使用cin.tie(0)和cout.tie(0)可以取消cin与cout之间的绑定，这意味着在进行输入操作时，不需要强行刷新输出缓冲区。这样可以进一步提高程序性能。

注意事项：

1. 不适用于混合输入输出：如果你的程序在输入输出中同时使用了C++的输入输出流和C标准库的输入输出函数（如scanf和printf），则不应该使用这段代码。因为这会导致输入输出之间的不同步。

2. 不能混用输入输出函数：在使用了这段代码后，应避免使用C标准库的输入输出函数（如printf和scanf），因为这些函数与输入输出流的同步已被关闭。简单来说，关闭了同步流，就**不能用scanf和printf。**

3. 关闭了同步流，**不能用getchar()函数**。

4. 不影响程序正确性：关闭输入输出流的同步不会影响程序的正确性，它只是为了提高程序的执行效率。因此，在一些对输入输出性能要求较高的场景下，可以考虑使用这段代码。

5. 关闭了同步流（也就是使用这段代码），**不能再用cout<<endl。而应该改用cout<<'\n'**。

因为通常情况下，cout<<endl会输出一个换行符并刷新输出缓冲区，确保内容立即显示。但是，当使用了上述代码时，cout<<endl不再具有自动刷新缓冲区的功能。

这三行代码必须一起用，不能只用第一行代码，使用这三行代码时，得用C++的头文件。算法竞赛为了方便，一般使用万能头文件**bits/stdc++.h**