2022 CSP-J1 试题 带解析

一、 单项选择题

（共 15 题，每题 2 分，共计 30 分；每题有且仅有一个正确选项）

1 以下哪种功能没有涉及C++语言的**面向对象**特性支持：（）。
A.C++中调用 `printf()` 函数
B.C++中调用用户定义的类成员函数
C.C++中构造一个 `class` 或 `struct`
D.C++中构造来源于同一基类的多个派生类
A， `printf()` 函数是C函数。

2 有 `6` 个元素，按照 `6 5 4 3 2 1` 的顺序进入栈 `S` ，请问下列哪个出栈序列是非法的（）。
A. `543612`
B. `453126`
C. `346521`
D. `234156`
C，如果数值a出栈，那么在a前入栈的元素要么已出栈，要么顺序地排列在栈中。当4出栈时，4前入栈的6,5一定都在栈中，情况为：栈底-6-5。所以接下来不可能是6出栈，只能是5出栈。

3 运行以下代码片段的行为是（）。

```
int x = 101;
int y = 201;
int *p = &x;
int *q = &y;
p = q;
```

A.将 `x` 的值赋为 `201`
B.将 `y` 的值赋为 `101`
C.将 `q` 指向 `x` 的地址
D.将 `p` 指向 `y` 的地址
D，p是指向x的指针，也就是x的地址。q是指向y的指针，也就是y的地址。把q赋值给p，也就是让p从指向x的指针变为指向y的指针。

4 链表和数组的区别包括（）。
A.数组不能排序，链表可以
B.链表比数组能存储更多的信息
C.数组大小固定，链表大小可动态调整
D.以上均正确
C

5 对假设栈 `S` 和队列 `Q` 的初始状态为空。存在e1~e6六个互不相同的数据，每个数据按照进栈 `S` 、出栈 `S` 、进队列 `Q` 、出队列 `Q` 的顺序操作，不同数据间的操作可能会交错。已知栈S中依次有数据 `e1 e2 e3 e4 e5 e6` 进栈，队列Q依次有数据 `e2 e4 e3 e6 e5 e1` 出队列。则栈S的容量至少是（）个数据。
A.2
B.3
C.4
D.6
B，队列Q出队列就是栈S出栈，画个图模拟一下出入栈过程即可解决问题。

6 对表达式 `a+(b-c)*d` 的前缀表达式为（），其中 `+ - *` 是运算符。
A. `*+a-bcd`
B. `+a*-bcd`
C. `abc-d*+`
D. `abc-+d`
B
中缀转前缀，
先运算 `b-c` ，变为 `-bc` 。然后是 `X*d` （X为b-c），变为 `*Xd` 。把X的前缀表达式代入，为 `*-bcd` 。最后是 `a+X` （X为 `(b-c)*d` ），变为 `+aX` ，把X的前缀表达式代入，为 `+a*-bcd` 。
转后缀把运算符放到后面就可以了。

7 假设字母表{ `a, b, c, d, e` }在字符串出现的频率分别为10%，15%，30%，16%，29%。若使用哈夫曼编码方式对字母进行不定长的二进制编码，字母d的编码长度为（）位。
A.1
B.2
C.2或3
D.3
B
构建哈夫曼树的方法为：每次选取两个权值最小的结点，加上双亲结点构成一

棵树。
初始一共有5个结点，每个结点的权值分别为：`a:10,b:15,c:30,d:16,e:29`
选择权值最小的两个结点a和b，设结点f是a、b的双亲，权值25。
选择权值最小的两个结点f和d，设结点g是f、d的双亲，权值41。
选择权值最小的两个结点c和e，设结点h是c、e的双亲，权值59。
选择权值最小的两个结点g和h，设结点i是g、h的双亲，权值100。
构成的树如下图所示。
在哈夫曼树中，从根结点开始，每向下走一层编码多1位。根据构造出来的哈夫曼树可知，d的编码是两位。

8 一棵有`n`个结点的完全二叉树用数组进行存储与表示，已知根结点存储在数组的第1个位置。若存储在数组第9个位置的结点存在兄弟结点和两个子结点，则它的兄弟结点和右子结点的位置分别是（）。
A.8 18
B.10 18
C.8 19
D.10 19
C
二叉树的顺序存储结构中，第i结点的左孩子的下标为`2*i`，右孩子的下标为`2*i+1`，双亲的下标为`i/2`。
因此第9位置结点一定是第4结点的右孩子（左孩子下标都是偶数，右孩子下标都是奇数）。该结点的兄弟结点是4的左孩子，在数组中的下标应该比9少1，为8。而9的右孩子的下标为$9*2+1=19$。

9 考虑由N个顶点构成的有向连通图，采用邻接矩阵的数据结构表示时，该矩阵中至少存在（）个非零元素。
A. `N-1`
B. `N`
C. `N+1`
D. `N2`
A。
有向连通图，分为强连通图、单向连通图、弱连通图。若把有向边都当做无向边，如果这个无向图是连通图，那么这个图是弱连通图。
`n`个顶点的无向图，最少有`n-1`条边。那么这个有向图中最少有`n-1`条边，就可以构成弱连通图。有向图中每条边在邻接矩阵中就是一个元素，占一个位置，因此至少存在`n-1`个非零元素。

10 以下对数据结构的表述不恰当的一项为（）。
A.图的深度优先遍历算法常使用的数据结构为栈。
B.栈的访问原则为后进先出，队列的访问原则是先进先出。
C.队列常常被用于广度优先搜索算法。
D.栈与队列存在本质不同，无法用栈实现队列。
D
选项A：图的深度优先遍历算法经常用递归来完成，而递归实际是利用了C++中的函数递归调用栈，本质上是使用了栈的结构。实际上，如果直接使用栈，也可以完成图的深度优先遍历。
选项B、C都是正确的表述。
选项D中，栈与队列本质上都是功能受限的线性表，本质是相同的。用栈实现队列，虽然平时不会这样做，这样做也没什么意义，但还是可以实现的。举例如下：
设栈 `s1` 与 `s2` 来实现一个队列的功能。
入队：元素入栈 `s1`
出队：如果栈 `s2` 不为空，那么栈 `s2` 出栈。如果 `s2` 为空，那么把 `s1` 中的所有元素出栈并入栈到 `s2` ，而后 `s2` 出栈。

11 以下哪组操作能完成在双向循环链表结点 `p` 之后插入结点 `s` 的效果（其中，`next` 域为结点的直接后继， `prev` 域为结点的直接前驱)：（）。
A. `p->next->prev=s; s->prev=p; p->next=s; s->next=p->next;`
B. `p->next->prev=s; p->next=s; s->prev=p; s->next=p->next;`
C. `s->prev=p; s->next=p->next; p->next=s; p->next->prev=s;`
D. `s->next=p->next; p->next->prev=s; s->prev=p; p->next=s;`
D。
观察选项可知，四个选项为这四个语句的不同排列：

```
s->prev=p;
s->next=p->next;
p->next=s;
p->next->prev=s;
```

这里发生变化，又可能给其它量赋值的就是 `p->next` 。
使 `p->next` 发生变化的语句为： `p->next=s;`
而 `s->next=p->next;` 与 `p->next->prev=s;` 中用到的都应该是变化前的 `p->next` ，指向的是原来 `p` 的下一个结点。
所以 `p->next=s;` 应该放在最后，选 D。

12 以下排序算法的常见实现中，哪个选项的说法是错误的：（）。
A.冒泡排序算法是稳定的

B.简单选择排序是稳定的
C.简单插入排序是稳定的
D.归并排序算法是稳定的
B
考察排序的稳定性。选择排序是不稳定的，冒泡、插入、归并都是稳定的。

13 八进制数32.1对应的十进制数是（）。
A.24.125
B.24.250
C.26.125
D.26.250
C
进制转换问题，八进制转十进制的方法为：按位权展开。
整数部分： $3*8^1+2*8^0=26$
小数部分： $1*8^{-1}=0.125$
因此八进制 32.1 为十进制 26.125 。

14 一个字符串中任意个连续的字符组成的子序列称为该字符串的子串，则字符串 `abcab` 有（）个内容互不相同的子串。
A. `12`
B. `13`
C. `14`
D. `15`
B
abcab 的不相同子串有：
长为 0 的子串：空串
长为 1 的子串： `a,b,c`
长为 2 的子串： `ab,bc,ca`
长为 3 的子串： `abc,bca,cab`
长为 4 的子串： `abca,bcab`
长为 5 的子串： `abcab`
共有 13 个。

15 以下对递归方法的描述中，**正确**的是（）
A.递归是允许使用多组参数调用函数的编程技术
B.递归是通过调用自身来求解问题的编程技术
C.递归是面向对象和数据而不是功能和逻辑的编程语言模型
D.递归是将用某种高级语言转换为机器代码的编程技术

B
选项 A：“多组参数调用函数”是没有意义的表述，任意一个带参函数，都可以用多组参数来调用。
选项 B：论述正确。
选项 C：**面向对象编程**（或者说“类”）是面向对象和数据而不是功能和逻辑的编程语言模型。
选项 D：**编译**是将用某种高级语言转换为机器代码的编程技术。

二、阅读程序

（程序输入不超过数组或字符串定义的范围；判断题正确填✓，错误填✗；除特殊说明外，判断题 1.5 分，选择题 3 分，共计 40 分）

1

```cpp
#include <iostream>

using namespace std;

int main()
{
    unsigned short x, y;  //7
    cin >> x >> y;
    x = (x | x << 2) & 0x33;
    x = (x | x << 1) & 0x55;
    y = (y | y << 2) & 0x33;
    y = (y | y << 1) & 0x55;
    unsigned short z = x | y << 1; //13
    cout << z << endl;
    return 0;
}
```

假设输入的 x 、 y 均是不超过 15 的自然数，完成下面的判断题和选择题。

解析

考点：位运算
本题无实际意义，执行一系列位运算就可以。注意运算符的优先级左移运算符 `<<` 的优先级比按位或 `|` 的优先级更高。

16.删去第 7 行与第 13 行的 `unsigned` ，程序行为不变。（）
✓，声明变量时是否加 `unsigned` 只影响对该机器数数值的认定，而不影响机器数本身。既然是 `short` 类型，这就是一个16位的机器数。所有位运算都是针对机器数的，程序行为不变。

17.将第 7 行与第 13 行的 `short` 均改为 `char` ，程序行为不变。（）
×， `short` 是16位机器数， `char` 是8位。同样的位运算操作对16位与8位机器数操作的过程不一定是一样的。（例如对机器数10000000左移1位操作，如果是16位机器数，结果为100000000，如果是八位机器数，结果为00000000）。

18.程序总是输出一个整数 0 。（）
×
不是，否则无法解答后面的三道题。

19.当输入为 2 2 时，输出为 10 。（）
×
x和y经过的操作相同，最后都得到 `0100` ， `z = x | y << 1` 得到 `0000 1100`

20.当输入为 2 2 时，输出为 59 。（）
×

21.当输入为 13 8 时，输出为（）。
A. 0
B. 209
C. 197
D. 226
B
x结果： `0010 0000`
y结果： `0100 0000`
z结果： `1101 0001`
$2^7 + 2^6 + 2^4 + 2^0 = 128 + 64 + 16 + 1 = 209$

2

```
#include <algorithm>
#include <iostream>
#include <limits>

using namespace std;
```

```cpp
const int MAXN = 105;
const int MAXK = 105;

int h[MAXN][MAXK];

int f(int n, int m)
{
    if (m == 1) return n;
    if (n == 0) return 0;

    int ret = numeric_limits<int>::max();
    for (int i=1; i<=n; i++)
        ret = min(ret, max(f(n-i, m), f(i-1, m-1)) + 1); //
19
    return ret;
}

int g(int n, int m)
{
    for (int i=1; i<=n; i++)
        h[i][1] = i;
    for (int j=1; j<=m; j++)
        h[0][j] = 0;

    for (int i=1; i<=n; i++) {
        for (int j=2; j<=m; j++) {
            h[i][j] = numeric_limits<int>::max();
            for (int k = 1; k<=i; k++)
                h[i][j] = min(
                    h[i][j],
                    max(h[i-k][j], h[k-1][j-1]) + 1);
        }
    }

    return h[n][m];
}

int main()
{
    int n, m;
    cin >> n >> m;
```

```
    cout << f(n, m) << endl << g(n, m) << endl;
    return 0;
}
```

假设输入的 n 、 m 均是不超过100的正整数，完成下面的判断题和单选题：

解析

考点：
动态规划，递归、递推， C++ numeric_limits

- numeric_limits<数据类型>::max() 返回该数据类型可以表示的最大值
- numeric_limits<数据类型>::min() 返回该数据类型可以表示的最小值
- numeric_limits<数据类型>::epsilon() 返回该数据类型可以区分的两个数值的最小差值（即如果两个数值的差值小于该值，计算机认为这两个数值相等）

观察代码可知
f函数使用递归算法：

- 递归关系： f(n,m) 的值为： i 从 1 循环到 n ， max(f(n-i, m), f(i-1, m-1))+1 的最小值
- 递归出口： m 是 1 时 f(n,m) 值为 n ， n 是 0 时 f(n,m) 值为 0 。

g函数使用了递推算法：

- 初始状态： j 是1时 h[i][j] 值为 i ， i 是 0 时 h[i][j] 值为 0 。
- 递推关系： h[i][j] 的值为： k 从 1 循环到 i ， max(h[i-k][j], h[k-1][j-1])+1 的最小值

对比二者，如果把 f 函数中的 n 替换为 i ， m 替换为 j ， i 替换为 k 。递归出口对应递推初始状态，递归关系对应递推关系。递推和递归本就是可以相互转化的两种方法。可以看出二者是解决相同问题的不同方法。

22、当输入为 7 3 时， 第 19 行中用来取最小值的 min 函数执行了449次。（ ）
×
调用一次 f(n,m) 程序执行次数为 n 次。

23、输出的两行整数总是相同的。（）
✓，通过观察可知：`f`函数使用递归，`g`函数使用递推解决了相同的问题，因此结果是相同的。

24 当`m`为`1`时，输出的第一行总为`n`。（）
✓，递归函数`f`中，如果`m`为`1`，会返回`n`。递推函数`g`中，当`j`为`1`时，`h[i][j]`为`i`。那么返回值`h[n][m]`一定为`n`。

25 算法`g(n, m)`最为准确的时间复杂度分析结果为（）。
A. `O(n^(3/2)m)`
B. `O(nm)`
C. `O(n²m)`
D. `O(nm²)`
C
两个for循环，时间复杂度分别是$O(m)$和$O(n)$

```
    for (int i=1; i<=n; i++)
        h[i][1] = i;
    for (int j=1; j<=m; j++)
        h[0][j] = 0;
```

最后一个三层循环：

```
for (int i=1; i<=n; i++) {
        for (int j=2; j<=m; j++) {
            for (int k = 1; k<=i; k++)
```

j循环的循环次数是m，剩下i循环和k循环有联系，观察：i是1时循环1次，i是2时循环2次，.....，i是n时循环n次，那么在j固定时，i从1循环到n，最内层循环体执行次数为$1+2+...+n=(1+n)*n/2$。
循环嵌套，乘起来就是$m(1+n)*n/2$，去掉常数得到时间复杂度是$O(n^2m)$

26 当输入为`20 2`时，输出的第一行为（）。
A. `4`
B. `5`
C. `6`
D. `20`

C

`h[20][2]`的值为6

27 当输入为`100 100`时，输出的第一行为（）。

A. `6`

B. `7`

C. `8`

D. `9`

B

列举出每个数字第一次出现的位置

`h[1][100]=1`
`h[2][100]=2`
`h[4][100]=3`
`h[8][100]=4`
`h[16][100]=5`
`h[32][100]=6`
`h[64][100]=7`
`h[128][100]=8`

因此`h[100][100]`的值为`7`。

3

```
#include <iostream>

using namespace std;

int n, k;

int solve1()  //返回使得x²≤n的最大整数x
{
    int l = 0, r = n;
    while (l<=r) {
        int mid = (l+r)/2;
        if (mid * mid <= n) l = mid+1;
        else r = mid -1;
    }
    return l - 1;  //l-1是满足(l-1)²≤n的最大数
}

double solve2(double x)
```

```
{
    if (x == 0) return x;
    for (int i=0; i<k; i++)
        x = (x + n/x) / 2;  //假设x数列有极限A，随着k增加，xi会越来越接近√n
    return x;
}

int main()
{
    cin >> n >> k;
    double ans = solve2(solve1());
    cout << ans << ' ' << (ans * ans == n) << endl;
    return 0;
}
```

假设 `int` 为 `32` 位有符号整数类型， 输入的 `n` 是不超过 `47000` 的自然数、 `k` 是不超过 `int` 表示范围的自然数，完成下面的判断题和单选题：

解析

考点：二分法

`solve1` 函数使用了二分算法，满足 `mid*mid<=n` 时， `l=mid+1` 取右半边，否则 `r = mid -1` 取左半边，最后返回的是 `l-1` ，由于 `l` 都是 `mid+1` ，所以最后取到的是满足条件的 `mid` 。
可以看出，这个二分算法求的是：满足 `mid*mid<=n` 条件的， `mid` 的最大值。实际就是求$\lfloor\sqrt{n}\rfloor$。

再看 `solve2` 函数，传入的参数 `x` 是：$\lfloor\sqrt{n}\rfloor$，而当$\sqrt{n}$不是整数时，n/x是个略大于$\sqrt{n}$的数字，因为 `x` 是向下取整$\sqrt{n}$，所以比实际的$\sqrt{n}$小，因此n/x一定比n/x大。
$\lfloor\sqrt{n}\rfloor$与n/x二者取平均数后的值赋值给 `x` ， `x` 变得更接近$\sqrt{n}$。循环重复这一过程，会使 `x` 不断逼近$\sqrt{n}$，最终达到一种逼近$\sqrt{n}$的效果。

28 该算法最准确的时间复杂度分析结果为$O(\log n + k)$。（）
✓，因为solve1函数使用二分法，时间复杂度是$\log n$，其返回值传给solve2函数，循环次数是k次，两个函数的时间复杂度加起来$O(\log n + k)$。

29 当输入为 `9801 1` 时，输出的第一个数为 `99` 。（）
✓， `solve1` 求出$\lfloor\sqrt{n}\rfloor$，$\lfloor\sqrt{9801}\rfloor = 99$。 `solve2` 中 `x=99` ，k为1， `for` 循环执

行一次，`x=(x+n/x)/2=(99+9801/99)/2=99`
输出`ans`为`99`。

30 对于任意输入的`n`，随着所输入`k`的增大，输出的第二个数会变成`1`。（）
×，随着输入k的增大，ans会接近或等于$\sqrt{n}$，但ans是double类型的，只能保存有限的数位，也就是说只能表示有理数。如果$\sqrt{n}$是无理数，那么ans不可能与$\sqrt{n}$相等，因此ans*ans也不会等于n，输出的第二个表达式的值会一直为假，输出0。

31 该程序有存在缺陷。当输入的`n`过大时，第`12`行的乘法有可能溢出，因此应当将`mid`强制转换为`64`位整数再计算。（）
×
题目限定了`n`最大是47000，第一次求出`mid=n/2`为23500，`mid*mid`为$23500^2=552250000$，int类型可以表示该数字，并不会溢出。而后`r=mid-1`，下一次mid会变得更小，`mid*mid`也会更小，因此不会出现溢出的情况。
（如果说n过大会超过题目限定的47000，那极端一点，当n特别大时，转为64位的long long类型计算仍然可能溢出。）

32 当输入为`2 1`时，输出的第一个数最接近（）。
A.`1`
B.`1.414`
C.`1.5`
D.`2`
C，solve1得到$\lfloor\sqrt{2}\rfloor=1$，传入solve2循环1次，返回ans是1.5

33 当输入为`3 10`时，输出的第一个数最接近（）。
A.`1.7`
B.`1.732`
C.`1.75`
D.`2`
B，solve1得到$\lfloor\sqrt{3}\rfloor=1$，传入solve2循环10次，模拟到第3次即可得到1.732

34 当输入为`256 11`时，输出的第一个数（）。
A.等于`16`
B.接近但小于`16`
C.接近但大于`16`
D.前三种情况都有可能
A，$\lfloor\sqrt{256}\rfloor=16$，256是完全平方数，就是说$\sqrt{256}=16$，不管solve2中循环几次，得到的结果都是16。

三、完善程序

（单选题，每小题 3 分，共计 30 分）

1（枚举因数）

从小到大打印正整数 n的所有正因数。
试补全枚举程序。

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int n;
    cin >> n;
    vector<int> fac;
    fac.reserve((int)ceil(sqrt(n)));
    int i;
    for (i=1; i*i<n; ++i) {
        if (①) {
            fac.push_back(i);
        }
    }

    for (int k=0; k<fac.size(); ++k) {
        cout << ② << "";
    }
    if (③) {
        cout << ④ << "";
    }
    for (int k=fac.size()-1; k>=0; --k) {
        cout << ⑤ << "";
    }
}
```


解析

考点：
因数；
STL vector

- `vec.reserve(容量)`，手动设置 `vector` 的容量，预先为 `vector` 对象分配空间，以免在运行过程中触发扩容。
  第 8 行声明了一个 vector 类对象，名字为 fac。factor 是因数的意思，所以 fac 这个 vector 保存的是因数。
  第 9 行，ceil 是向上取整（返回值类型是 double），sqrt 是开根号。第 9 行做的事情是把 fac 的容量设为$\lceil n \rceil$
  for循环这里就是把 n 的所有 `i^2<n` 的因数保存在 `fac` 之中。 因此条件是判断因数的，数字 i 若是 n 的因数，那么 n 能整除 i，n 除以 i 的余数为 0。
  35 ①处应填（）
  A. `n % i == 0`
  B. `n % i == 1`
  C. `n % (i-1) == 0`
  D. `n % (i-1) == 1`
  A

36 ②处应填（）
A. `n / fac[k]`
B. `fac[k]`
C. `fac[k]-1`
D. `n / (fac[k]-1)`
B
从小到大打印刚刚找到的因数。

接着是看是否有$i^2 = n$的情况，如果有则输出 `i`。
第 37 题选 C `i*i==n`，第 38 题选 D `i`。
37 ③处应填（）
A. `(i-1) * (i-1) == n`
B. `(i-1) * i == n`
C. `i * i == n`
D. `i * (i-1) == n`
C

38 ④处应填（）
A. `n-i`
B. `n-i+1`
C. `i-1`
D. `i`
D

最后打印$i^2 > n$的因数 `i`
假设 fac 中的元素是 `a, b, c, d`，
那么接下来应该输出 `n/d, n/c, n/b, n/a`。
fac 中的元素为：`fac[0], fac[1], …, fac[fax.size()-1]`，接下来应该输出的是：`n/fac[fax.size()-1], n/fac[fax.size()-2], ..., n/fac[0]`。
39 ⑤处应填（）
A. `n / fac[k]`
B. `fac[k]`
C. `fac[k]-1`
D. `n / (fac[k]-1)`
A

2（洪水填充）

现有用字符标记像素颜色的8×8图像。颜色填充的操作描述如下：给定起始像素的位置和待填充的颜色，将起始像素和所有可达的像素（可达的定义：经过一次或多次的向上、下、左、右四个方向移动所能到达且终点和路径上所有像素的颜色都与起始像素颜色相同），替换为给定的颜色。

试补全程序。

```
#include<bits/stdc++.h>
using namespace std;

const int ROWS = 8;
const int COLS = 8;

struct Point {
    int r, c;
    Point(int r, int c): r(r), c(c) {}
};

bool is_valid(char image[ROWS][COLS], Point pt,
              int prev_color, int new_color) {
    int r = pt.r;
    int c = pt.c;
    return (0 <= r && r < ROWS && 0 <= c && c < COLS &&
            ___①___ && image[r][c] != new_color);
}
```

```
void flood_fill(char image[ROWS][COLS], Point cur, int
new_color) {
    queue<Point> queue;
    queue.push(cur);

    int prev_color = image[cur.r][cur.c];
    ___②___;

    while (!queue.empty()) {
        Point pt = queue.front ();
        queue.pop ();

        Point points[4] = {___③___, Point(pt.r - 1, pt.c),
                           Point(pt.r, pt.c + 1),
Point(pt.r, pt.c - 1)};
        for (auto p : points) {
            if (is_valid(image, p, prev_color, new_color)) {
                ___④___;
                ___⑤___;
            }
        }
    }
}

int main() {
    char image[ROWS][COLS] = {
    {'g', 'g', 'g', 'g', 'g', 'g', 'g', 'g'},
    {'g', 'g', 'g', 'g', 'g', 'g', 'r', 'r'},
    {'g', 'r', 'r', 'g', 'g', 'r', 'g', 'g'},
    {'g', 'b', 'b', 'b', 'b', 'r', 'g', 'r'},
    {'g', 'g', 'g', 'b', 'b', 'r', 'g', 'r'},
    {'g', 'g', 'g', 'b', 'b', 'b', 'b', 'r'},
    {'g', 'g', 'g', 'g', 'g', 'b', 'g', 'g'},
    {'g', 'g', 'g', 'g', 'g', 'b', 'b', 'g'}
    };

    Point cur(4, 4);
    char new_color = 'y';

    flood_fill(image, cur, new_color);
```

```
    for (int r = 0; r < ROWS; r++) {
        for (int c = 0; c < COLS; c++) {
            cout << image[r][c] << '';
        }
        cout << endl;
    }
//输出:
// g g g g g g g g
// g g g g g g r r
// g r r g g r g g
// g y y y y r g r
// g g g y y r g r
// g g g y y y y r
// g g g g g y g g
// g g g g g y y g

    return 0;
}
```

解析

考点:

1 bfs，连通块问题

2 迭代式for循环

遍历一个可迭代对象中的所有元素。可迭代对象可以为：数组或 vector、list 等 stl 容器。

`for(元素类型 变量名 : 可迭代对象)`

元素类型为可迭代对象中元素的类型。可以写 auto 来自动设置元素类型。

取到的变量是值，改变该变量不会改变可迭代对象中元素的值。

`for(元素类型 &变量名 : 可迭代对象)`

取到的变量是引用，改变该变量会改变可迭代对象中元素的值。

3 auto变量类型

使用 auto 关键字可以让编译器自动确定声明的变量的类型。

例：`auto p = 3.2`（auto 在这里相当于 double）

例：`vector<int> vec; auto it = vec.begin(); iterator`

函数名是：洪水填充，看函数内的结构就能看出用了广搜算法。

先设队列，而后初始结点入队。

`cur` 是起始位置，`prev_color` 就是起始位置的颜色。

该题要做的是将包含起始位置的连通块的颜色从原颜色（`prev_color`）变为新颜色(`new_color`)。
初始位置已经入队了，初始位置应该改变颜色。
看第②空（第 41 题），可知这里要写的是使初始位置改变颜色，初始位置是（`cur.r, cur.c`），所以应该写：`image[cur.r][cur.c] = new_color`
41 ②处应填（）
A. `image[cur.r+1][cur.c] = new_color`
B. `image[cur.r][cur.c] = new_color`
C. `image[cur.r][cur.c+1] = new_color`
D. `image[cur.r][cur.c] = prev_color`
B

接着开始循环，只要队列不空，每次循环出队一个位置 `pt`。接下来确定 `pt` 位置上下左右的四个位置，保存在 `points` 数组中。
`Point(pt.r - 1, pt.c)` 是 pt 上方位置
`Point(pt.r, pt.c + 1)` 是 pt 右侧位置
`Point(pt.r, pt.c - 1)` 是 pt 左侧位置
还缺下方位置，应该是 `Point(pt.r+1, pt.c)`。
42 ③处应填（）
A. `Point(pt.r, pt.c)`
B. `Point(pt.r, pt.c+1)`
C. `Point(pt.r+1, pt.c)`
D. `Point(pt.r+1, pt.c+1)`
C

在循环内，先判断 `pt` 位置是否合法，判断函数为 `is_valid`
传入的 `image` 为图像二维数组，`pt` 为被判断是否合法的位置，`prev_color` 为起始位置颜色，`new_color` 为要改变为的新颜色。
这里 r，c 为 pt 的行号和列号，首先 `0 <= r && r < ROWS && 0 <= c && c < COLS` 判断 pt 位置需要在地图内，`image[r][c] != new_color` 表示新位置的颜色应该不与新颜色相同（如果与新颜色相同，说明这里已经变色了），还差一点需要填空。由于要把与起始位置颜色相同的在同一连通块内的位置都变色，因此这里需要**与起始位置颜色相同**，才是合法的，才需要进行变色。所以①这里应该填 `image[r][c] == prev_color`
40 ①处应填（）
A. `image[r][c] == prev_color`
B. `image[r][c] != prev_color`

C. `image[r][c] == new_color`
D. `image[r][c] != new_color`
A

回到这里，如果该位置合法，那么按照广搜模板，这里应该访问 `p` 位置，然后 `pt` 位置入队。
这里所谓的访问，应该是将 p 位置设为新颜色: `image[p.r][p.c] = new_color`，
而后把新位置 p 入队: `que.push(p)`
43 ④处应填（）
A. `prev_color = image[p.r][p.c]`
B. `new_color = image[p.r][p.c]`
C. `image[p.r][p.c] = prev_color`
D. `image[p.r][p.c] = new_color`
D

44 ⑤处应填（）
A. `queue.push(p)`
B. `queue.push(pt)`
C. `queue.push(cur)`
D. `queue.push(Point(ROWS, COLS))`
A