1. 在 3 X 3 的空格内，用 1，2，…, 9 的 9 个数字填入 9 个空格内，使得每行数字组成的十进制数平方根为整数。试用一般图搜索搜索算法求解。

Solution：
我们通过排列方式来解决这个问题：
1，在这个图每个节点代表一个唯一的 3x3 的 2 位数组
2，如果我们能通过交换两个相邻的数组能够从一个排列到另一种排列那么这两个节点是联通的。

```cpp
int main() {
    array<int, 9> grid = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    do {
        if (check_grid(grid)) {
            cout << "找到答案:\n";
            print_grid(grid);
            break;
        }
    } while (next_permutation(grid.begin(), grid.end()));

    return 0;
}
```

1.用 arrays 库函数来存储 1 到 9
2.每次循环构造下一个排列
3.每次循环判断是否找到答案
如果找到答案终止运行
不然在判断完每一种排列后自然的结束程序

```cpp
bool has_integer_sqrt(const vector<int>& row) {
    int num = row[0] * 100 + row[1] * 10 + row[2];
    int root = std::sqrt(num);
    return root * root == num;
}
```

这个函数接受一个 vector 它的值是一个 2d array 的每行的三个数字
Num 是把每行的三个数字合并一个整数
Root 是 用库函数 sqrt 计算 num 的平方根
如果 root 的平方等于 num 说明 root 是一个完美平方数 也就是说是一个整数；

```
bool check_arr(const array<int, 9>& grid) {
    for (int i = 0; i < 9; i += 3) {
        if (!has_integer_sqrt({ grid[i], grid[i + 1], grid[i + 2] })) {
            return false;
        }
    }
    return true;
}
```

检查 2d 数组函数，接受一个 3x3 的数组
在循环检查每个行是不是能够构造一个上述要求符合的数字
如果是返回真如果有行不能构造返回假

```
void print_grid(const array<int, 9>& grid) {
    for (int i = 0; i < 9; i++) {
        cout << grid[i] << " ";
        if (i % 3 == 2) cout << "\n";
    }
}
```

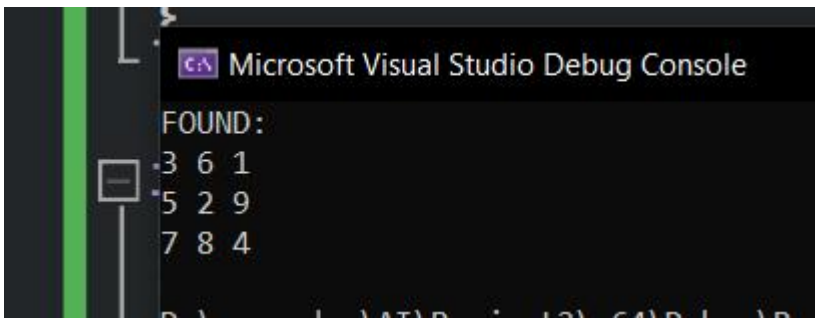
这个函数打印 2d 数组的每个元素构造一个 3x3 的矩阵；

整个代码：

```
#include <iostream>
#include <algorithm>
#include <array>
#include <cmath>
#include <iterator>
#include <vector>

using namespace std;
```

```
bool has_integer_sqrt(const vector<int>& row) {
```

```
    int num = row[0] * 100 + row[1] * 10 + row[2];
    int root = sqrt(num);
    return root * root == num;
}


void print_grid(const array<int, 9>& grid) {
    for (int i = 0; i < 9; i++) {
        cout << grid[i] << " ";
        if (i % 3 == 2) cout << "\n";
    }
}

bool check_grid(const array<int, 9>& grid) {
    for (int i = 0; i < 9; i += 3) {
        if (!has_integer_sqrt({ grid[i], grid[i + 1], grid[i + 2] })) {
            return false;
        }
    }
    return true;
}

int main() {
    array<int, 9> grid = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    do {
        if (check_grid(grid)) {
            cout << "FOUND:\n";
            print_grid(grid);
            break;
        }
    } while (next_permutation(grid.begin(), grid.end()));

    return 0;
}
```



1. 分析宽度优先搜索和深度优先搜索的优缺点，举出他们的正例和反例。

宽度优先搜索（BFS）和深度优先搜索（DFS）是图和树遍历的两种基本方法，每种方法都有其特定的应用场景、优点和缺点。

宽度优先搜索（BFS）

优点

1. 找到最短路径：BFS 能够保证在无权图中找到从起点到目标节点的最短路径。
2. 层级遍历：BFS 可以进行层级遍历，这在很多问题中很有用，比如在树的层级遍历或是在图中寻找某个节点所在的层级。

缺点

1. 空间复杂度：在最坏情况下，需要存储与图中节点数相当的信息，因此空间复杂度可以是 O(V)，其中 V 是节点数。
2. 不适用于寻找特定解：如果问题是寻找任意满足条件的解而不是最短路径，BFS 可能会浪费大量时间在遍历上。

正例

- 在社交网络中找到两人之间的最短联系路径。
- 迷宫求解中找到最短出路。

反例

- 深度较深的解决方案搜索，如解决某些逻辑问题或括号生成，使用 BFS 可能导致大量的内存使用。

深度优先搜索（DFS）

优点

1. 空间复杂度低：DFS 的空间复杂度相对较低，因为它主要由递归栈的深度决定，即 O(H)，其中 H 是图的最大深度。
2. 找到所有解：DFS 能够遍历图中的所有路径，适用于需要寻找所有可能解的问题。

缺点

1. 可能不是最短路径：DFS 不保证找到的是最短路径，尤其是在图搜索中。
2. 可能陷入无限循环：在有循环的图中，如果没有适当的检查，DFS 可能会无限递归下去。

正例

- 解决迷宫问题时寻找所有可能的出路。
- 生成所有可能的括号组合。

反例

- 需要找到最短路径的问题，如社交网络中的最短联系路径。
- 需要广泛搜索大量节点但实际解可能非常接近起点的情况，DFS 可能会浪费大量时间深入探索。

总结来说，BFS 和 DFS 各有优势和劣势，适用于不同的问题场景。选择哪一种算法取决于问题的具体需求，比如是否需要最短路径、解的深度、空间复杂度的考虑等。