初级问题

基础穷举

- 递归函数

1. 优化：计算一次以后，用数列将结果存储起来，可以优化之后的计算

```
int memo[MAX_N + 1];

int fib(int n) {
  if (n <= 1) return n;
  if (memo[n] != 0) return memo[n];
  return memo[n] = fib(n - 1) + fib(n - 2);
}
```

- 栈

1. 举例

```
#include <stack>
#include <cstdio>

using namespace std;

int main() {
  stack<int> s;             // 声明存储int类型数据的栈
  s.push(1);                // {} → {1}
  s.push(2);                // {1} → {1,2}
  s.push(3);                // {1,2} → {1,2,3}
  printf("%d\n", s.top()); // 3
  s.pop();                  // 从栈顶移除 {1,2,3}→{1,2}
  printf("%d\n", s.top()); // 2
  s.pop();                  // {1,2} → {1}
  printf("%d\n", s.top()); // 1
  s.pop();                  // {1} → {}
  return 0;
}
```

- 队列

1. 举例

```
#include <queue>
#include <cstdio>

using namespace std;

int main() {
  queue<int> que;               // 声明存储int类型数据的队列
  que.push(1);                  // {} → {1}
  que.push(2);                  // {1} → {1,2}
  que.push(3);                  // {1,2} → {1,2,3}
  printf("%d\n", que.front()); // 1
  que.pop();                    // 从队尾移除 {1,2,3}→{2,3}
  printf("%d\n", que.front()); // 2
  que.pop();                    // {2,3} → {3}
  printf("%d\n", que.front()); // 3
  que.pop();                    // {3} → {}
  return 0;
}
```

- 深度优先搜索

1. 设计：是否遍历了所有可能并对每种情况进行了判断，是否考虑了flase的情况，是否能进一步优化
2. 深搜的递归函数编写和状态管理都更简单，需要的内存空间与最大的递归深度成正比，与状态数相比深度并不会太大，相对来说更省内存?
3. 深搜的处理顺序：判断当前状态是否得到所求解，递归不加下一个状态的情况，递归加入下一个状态的情况，若没有得到所求解的情况返回false
4. 深搜适合处理：有没有正确解，正确解的个数

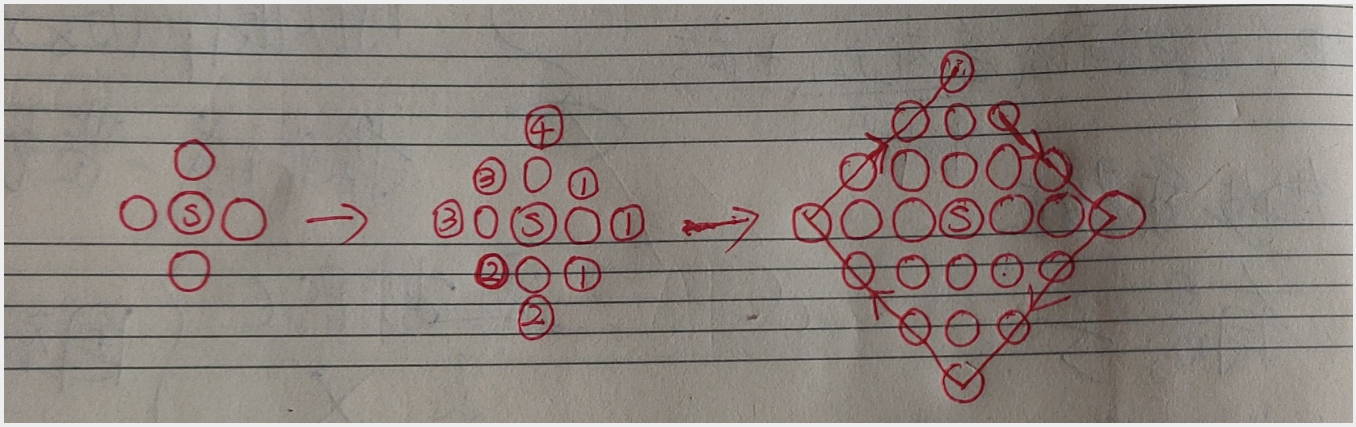
-
  - 从一组数中能否选出特定的数使其运算结果为指定的数
-
  - 八连通积水问题

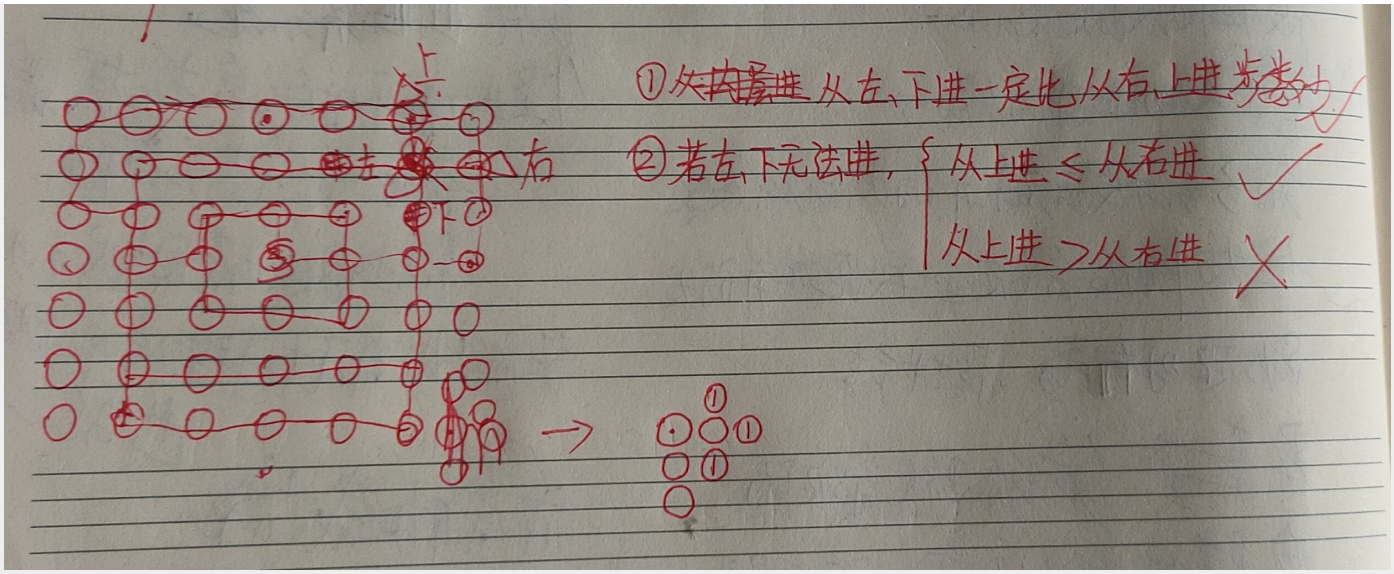
- 宽度优先搜索

1. 深搜隐式利用了栈，宽搜利用了队列，会把状态逐个加入队列，通常需要与状态数成正比的内存空间，相对而言更耗内存
2. 宽搜按照距开始状态由近及远的顺序进行搜索，很容易用来求解最短路径、最少操作等问题
3. 当状态更加复杂时，可能需要封装成一个类来表示状态
4. 宽度优先搜索中，只要将已经访问过的状态用标记管理起来 ，就可以很好地做到由近及远的搜索。 这个问题中由于要求最短距离，不妨用d[N][M]数组把最短距离保存起来。初始时用充分大的常 数INF来初始化它，这样尚未到达的位置就是INF, 也就同时起到了标记的作用。虽然到达终点时就会停止搜索，可如果继续下去直到队列为空的话，就可以计算出到各个位置的 最短距离。此外，如果搜索到最后，d依然为INF的话，便可得知这个位置就是无法从起点到达的 位置。在今后的程序中，使用像INF这样充分大的常数的情况还很多。不把INF当作例外，而是直接参与普通运算的情况也很常见。这种情况下，如果INF过大就可能带来溢出的危险。
5. IDDFS迭代加深深度优先搜索与宽度优先搜索的状态转移顺序类似，但更加节约内存。最开始将深度优先搜索的递归次数限制在1次，在找到解之前不断增加递归深度?
6. 宽搜的处理顺序：从起点开始循环，每次都把取出队列头部的点进行检测，让后将其相邻的能走通的点依次放入队列的尾部，并将起点到该相邻点的距离+1

- ◦ 走出迷宫的最小步数

1. 这道题的前提是已经知道重点的坐标但不知道能否走通，如果不知道重点的坐标只需加入每一步都对被扫描的点进行检测看是不是终点
2. 思路：INF初始化，将起点加入队列，从起点开始循环进行宽搜，通过循环结束时终点到起点的距离值(到终点步数选最小的记录)来判断能否达到和能达到的最小步数
3. 宽搜的处理顺序：从起点开始循环，每次都把取出队列头部的点进行检测，让后将其相邻的能走通的点依次放入队列的尾部，并将起点到该相邻点的距离+1
4. INF的作用：充当了某个位置是否被检测过的标识符，充当了迷宫是否能走通的标识符（若走不通，最后返回终点到起点的距离值仍为INF）
5. typedef pair<int,int> P;? pair模板只能定义两个变量等价于只有两个变量的结构体，可以直接用pair.first和pair.second分别直接调用第一个和第二个变量，充当某一时刻被扫描的位置
6. char maze[MAX_N][MAX_M+1];?
7. d[MAX_N][MAX_M];//表示到各个位置的最短举例的数组
8. int dx[4]={1,0,-1,0},dy[4] = {0,1,0,-1};//二重数组配合for循环来表示二维向量，与八连通的处理办法不同
9. queue<. P > que;创建一个名字为que的队列，这个队列的每个元素都是P这种模板的结构体
10. 书中寻找最短步数的漏洞，终点步数赋值前进行比较





- 特殊状态的枚举
  不用深搜简短地实现寻找可行解：next_permutation函数或位运算?
- 剪枝
  当当前状态无论这样转移都不会存在解时不再搜索而是直接跳过
- 栈内存和堆内存
  栈内存区：主调函数拥有的局部变量等信息，程序启动后不能再扩大，因此函数递归深度有上限但仍可以在C和C++中进行上万次递归，java中可以指定栈的大小
  堆内存区：用new或malloc进行分配的内存区，也是全局变量保存的地方，当必须申请巨大的数组的时候，放在堆内存区可以减少栈溢出的危险，申请内存时一般申请稍微大一点