程序设计思维与实践

Thinking and Practice in Programming

搜索 | 内容负责：郑得贤

1 广 度 优 先 搜 索

Breadth First Search

广度优先搜索

- BFS
  - 在 DS 课上已经学过了 BFS 的概念，所以主要关注：
    - ACM 上利用了 STL<queue/priority_queue> 一般的解题代码框架
    - CSP 中 BFS 题的考察
    - 隐式图问题
    - BFS 的优化

搜索就是扩散

迷宫问题

- BFS 引例 —— 迷宫问题：
  - 一个n*m的迷宫；

    有的格子里有障碍物，不能走；

    有的格子是空地，可以走；
  - 给定一个迷宫，求从左上角走到右下角最少需要走多少步(数据保证一定能走到)。只能在水平方向或垂直方向走，不能斜着走。

样例输入

01000

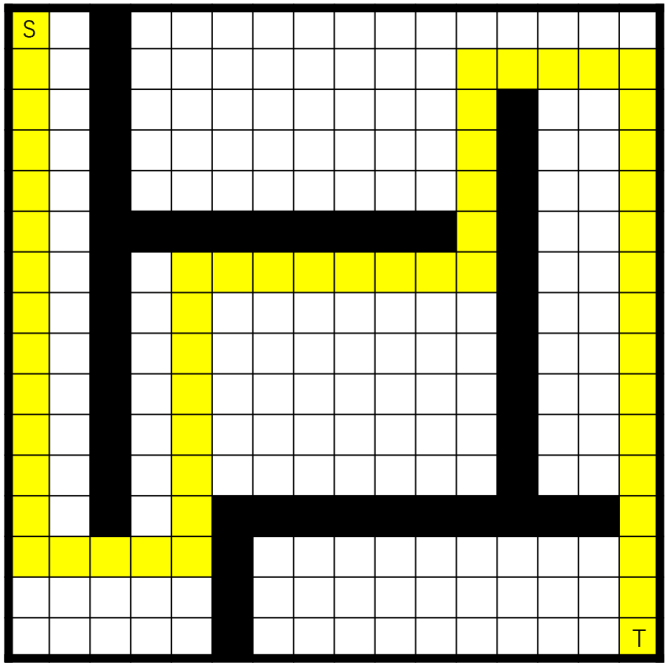
01010

01010

00010

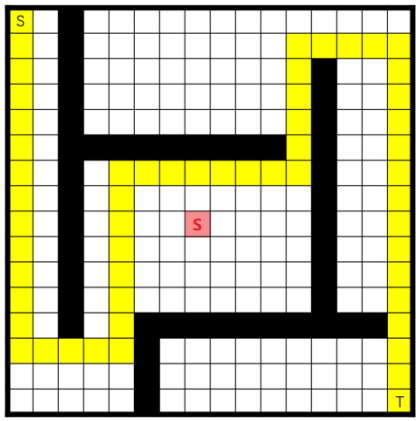
样例输出

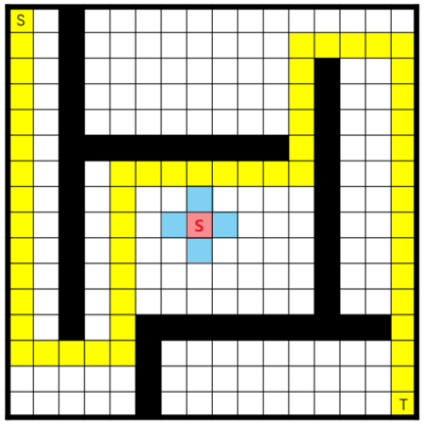
13

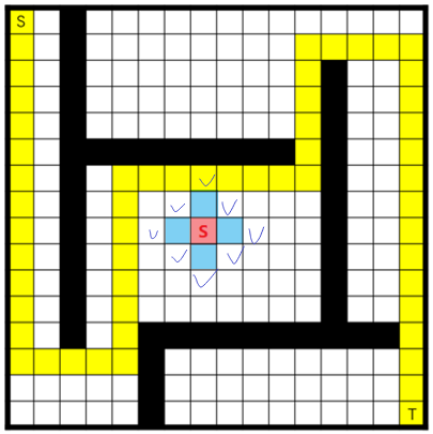


迷宫问题

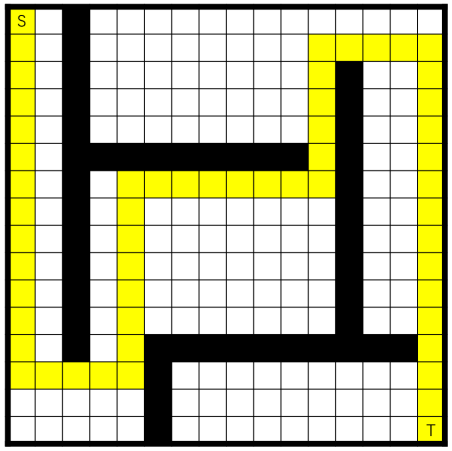
- BFS 引例 —— 迷宫问题：
  - 类似树的按层遍历。
  - 1. 访问初始点(sx,sy)，并将其标记为已访问过。
  - 2. 访问(sx,sy)的所有未被访问过可到达的邻接点，并均标记为已访问过，将这些点加入到队列中。
  - 3. 再按照队列中的次序，访问每一个顶点的所有未被访问过的邻接点，并均标记为已访问过，加入到队列中，依此类推。
  - 4. 直到到达终点或者图中所有和初始点Vi有路径相通的顶点都被访问过为止。







迷宫问题



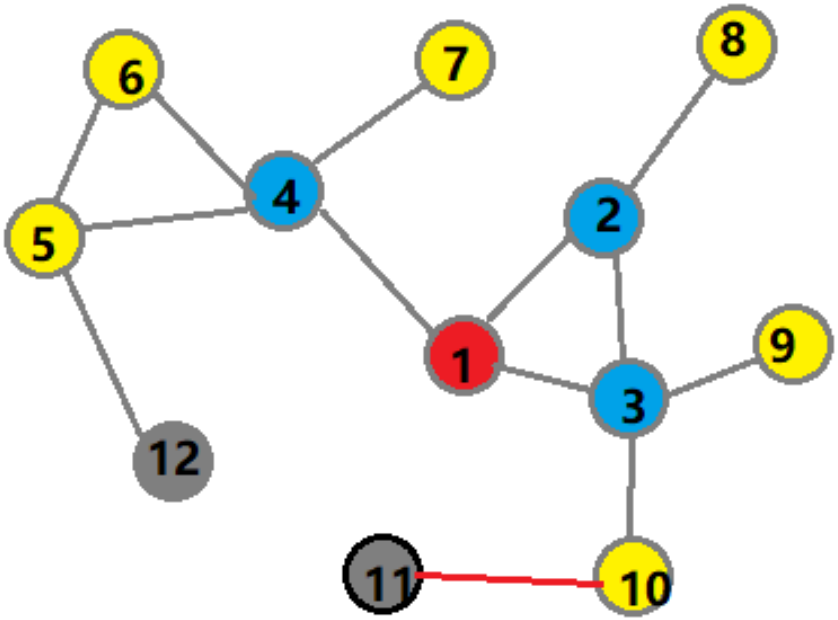
- BFS 引例 —— 迷宫问题：
  - 代码

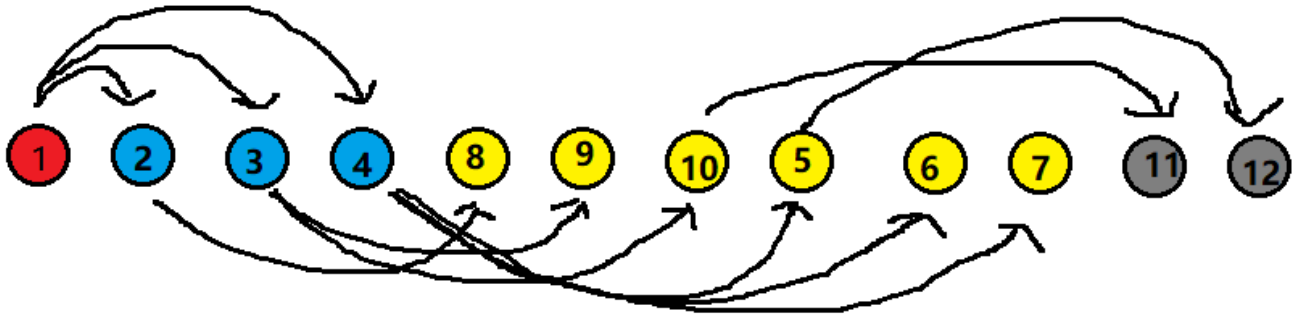
```
int vis[100][100], dis[100][100], n, m;
int sx, sy, tx, ty;
int dx[] = {0, 0, -1, 1};
int dy[] = {1, -1, 0, 0};

int bfs()  {
    queue<Point> q;
    q.push(Point(sx, sy));
    vis[sx][sy] = 1; dis[sx][sy] = 0;
    while (!q.empty())  {
        Point Now = q.front();
        q.pop();
        for (int k = 0;k < 4; k++)  {
            int x = Now.first + dx[k];
            int y = Now.second + dy[k];
            if (x < 1 || x > n || y < 1 || y > n) continue;
            if (vis[x][y]) continue;
            dis[x][y] = dis[Now.first][Now.second] + 1;
            vis[x][y] = 1;
            q.push(Point(x, y));
        }
    }
    return dis[tx][ty];
}
```

迷宫问题

- BFS 过程：



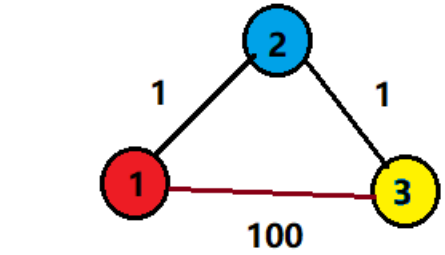


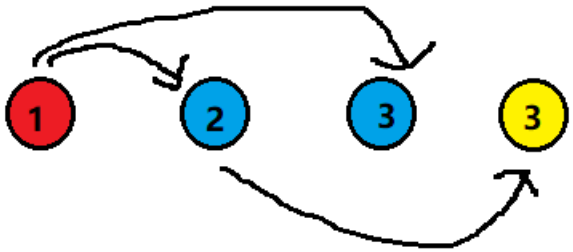
BFS解题框架

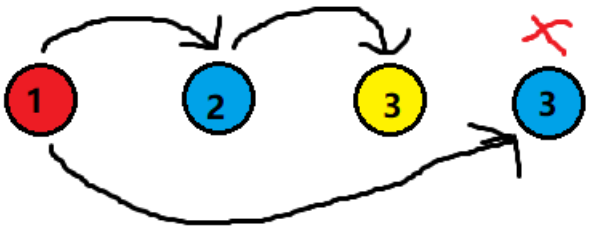
- BFS 解决一般问题的框架 总结：
  - vis 数组，用来记录某个状态下的最优结果，以免反复到达，一般是一维或二维，有可能使用高维数组（当状态比较复杂，比如增加了时间维度）
  - queue 队列，用于记录“层层”拓展的中间节点
  - dis 可有可无，上例中使用了这个来记录到某个点的最短距离，但是可以和 vis 数组进行合并
  - While(!Q.empty() 循环
    - 取点
    - 拓展周边节点
    - 特判合法性，加入队列
    - 到达目标节点跳出循环（可加可不加）

BFS解题框架

- 若相邻节点的距离不是1，而是更一般的情况







BFS解题框架

- 若相邻节点的距离不是1，而是更一般的情况

```
#define Point pair<int, int>
#define edge pair<Point, int>

using namespace std;

int vis[100][100], dis[100][100], n, m;
int sx, sy, tx, ty;
vector<edge> E[100][100];

struct Node  {
    int dis;
    Point p;
    Node(int x, int y, int Dis)  {
        p = Point(x, y), dis = Dis;
    }
    friend bool operator <(const Node a, const Node b)  {
        return a.dis < b.dis;
    }
};
```

BFS解题框架

- 若相邻节点的距离不是1，而是更一般的情况

```
int bfs()  {
    priority_queue<Node> q;
    q.push(Node(sx, sy, 0));
    vis[sx][sy] = 1; dis[sx][sy] = 0;
    while (!q.empty())  {
        Point Now = (q.top()).p;
        q.pop();
        if (vis[Now.first][Now.second]) continue;
        vis[Now.first][Now.second] = 1;
        for (edge e : E[Now.first][Now.second])  {
            Point Nxt = e.first;
            dis[Nxt.first][Nxt.second] = dis[Now.first][Now.second] + e.second;
            q.push(Node(Nxt.first, Nxt.second, -dis[Nxt.first][Nxt.second]));
        }
    }
    return dis[tx][ty];
}
```

最优配餐CSP201409-4

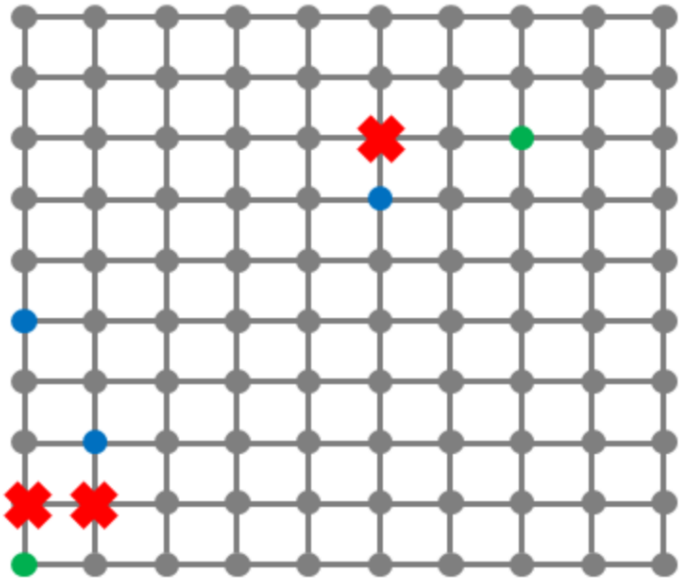
- CSP 中 BFS 的考察

问题描述

栋栋最近开了一家餐饮连锁店，提供外卖服务。随着连锁店越来越多，怎么合理的给客户送餐成为了一个急需解决的问题。

栋栋的连锁店所在的区域可以看成是一个n×n的方格图（如下图所示），方格的格点上的位置上可能包含栋栋的分店（绿色标注）或者客户（蓝色标注），有一些格点是不能经过的（红色标注）。

方格图中的线表示可以行走的道路，相邻两个格点的距离为1。栋栋要送餐必须走可以行走的道路，而且不能经过红色标注的点。



送餐的主要成本体现在路上所花的时间，每一份餐每走一个单位的距离需要花费1块钱。每个客户的需求都可以由栋栋的任意分店配送，每个分店没有配送总量的限制。

现在你得到了栋栋的客户的需求，请问在最优的送餐方式下，送这些餐需要花费多大的成本。

最优配餐

- 解题思路
  - 若只有一个起点的话，那么成本 = 每个点的距离 * 每个点的客户需求量
  - 方案1：分别对每个起点点进行一次BFS，距离 = 所有点为起点的距离的最小值
  - 方案2：直接进行一次BFS，把起始的所有点都一次性添加进队列中，dis初值赋为0（其余点的dis值为inf），最终得到的dis值就是到该点的最小距离。
    - 更好的理解方式：加一个超级源点 -> 到各起始点的代价是 0

最优配餐

```
void bfs()  {
    queue<pii> q;
    for (int i = 1;i <= n; i++)
        for (int j = 1;j <= n; j++)
            if (!dis[i][j]) q.push(pii(i, j));
    while (!q.empty())  {
        pii Now = q.front();
        q.pop();
        for (int i = 0;i < 4; i++)  {
            int x = Now.first + dx[i];
            int y = Now.second + dy[i];
            if (dis[x][y] == MAX)  {
                dis[x][y] = dis[Now.first][Now.second] + 1;
                Ans += dis[x][y] * 1ll * ned[x][y];
                q.push(pii(x, y));
            }
        }
    }
}
```

游戏CSP201604-4

- CSP 中 BFS 的考察

问题描述

小明在玩一个电脑游戏，游戏在一个$n \times m$的方格图上进行，小明控制的角色开始的时候站在第一行第一列，目标是前往第n行第m列。

方格图上有一些方格是始终安全的，有一些在一段时间是危险的，如果小明控制的角色到达一个方格的时候方格是危险的，则小明输掉了游戏，如果小明的角色到达了第n行第m列，则小明过关。第一行第一列和第n行第m列永远都是安全的。

每个单位时间，小明的角色必须向上下左右四个方向相邻的方格中的一个移动一格。

经过很多次尝试，小明掌握了方格图的安全和危险的规律：每一个方格出现危险的时间一定是连续的。并且，小明还掌握了每个方格在哪段时间是危险的。

现在，小明想知道，自己最快经过几个时间单位可以达到第n行第m列过关。

输入格式

输入的第一行包含三个整数n，m，t，用一个空格分隔，表示方格图的行数n、列数m，以及方格图中有危险的方格数量。

接下来t行，每行4个整数r，c，a，b，表示第r行第c列的方格在第a个时刻到第b个时刻之间是危险的，包括a和b。游戏开始时的时刻为0。输入数据保证r和c不同时为1，而且当r为n时c不为m。一个方格只有一段时间是危险的（或者说不会出现两行拥有相同的r和c）。

输出格式

输出一个整数，表示小明最快经过几个时间单位可以过关。输入数据保证小明一定可以过关。

样例输入

```
3 3 3
2 1 1 1
1 3 2 10
2 2 2 10
```

样例输出

```
6
```

前30%的评测用例满足：$0 < n, m \leqslant 10$，$0 \leqslant t < 99$。

所有评测用例满足：$0 < n, m \leqslant 100$，$0 \leqslant t < 9999$，$1 \leqslant r \leqslant n$，$1 \leqslant c \leqslant m$，$0 \leqslant a \leqslant b \leqslant 100$。

游戏

- 解题思路
  - 该题是 BFS 的另一种变形，这里障碍物并不是每时每刻都存在的，而是只在一个连续的区间存在。
  - 首先记录的时候就应该多记录一维时间，然后才能判断是否能够扩展到其它位置
  - 注意这里不要直接使用 vis[][] 数组判断这一个点是否经过，因为一个可以会重复经过，要使用 vis[][][] 多记录一维时间
  - 本题目样例是较好的，它可以提示我们不能只是记录这个点是否访问过。

游戏

```
int bfs()  {
    queue<Node> q;
    q.push(Node(1, 1, 0));
    Vis[1][1][0] = 1;
    while (!q.empty())  {
        Node Now = q.front();
        q.pop();
        for (int k = 0;k < 4; k++)  {
            int x = Now.x + dx[k];
            int y = Now.y + dy[k];
            if (x < 1 || x > n || y < 1 || y > m) continue;
            int t = Now.t + 1;
            if (Beg[x][y] && t >= Beg[x][y] && t <= End[x][y]) continue;
            if (Vis[x][y][t]) continue;
            if (t >= 300) continue;
            if (x == n && y == m) return t;
            Vis[x][y][t] = 1;
            q.push(Node(x, y, t));
        }
    }
    return 0;
}
```

隐式图问题

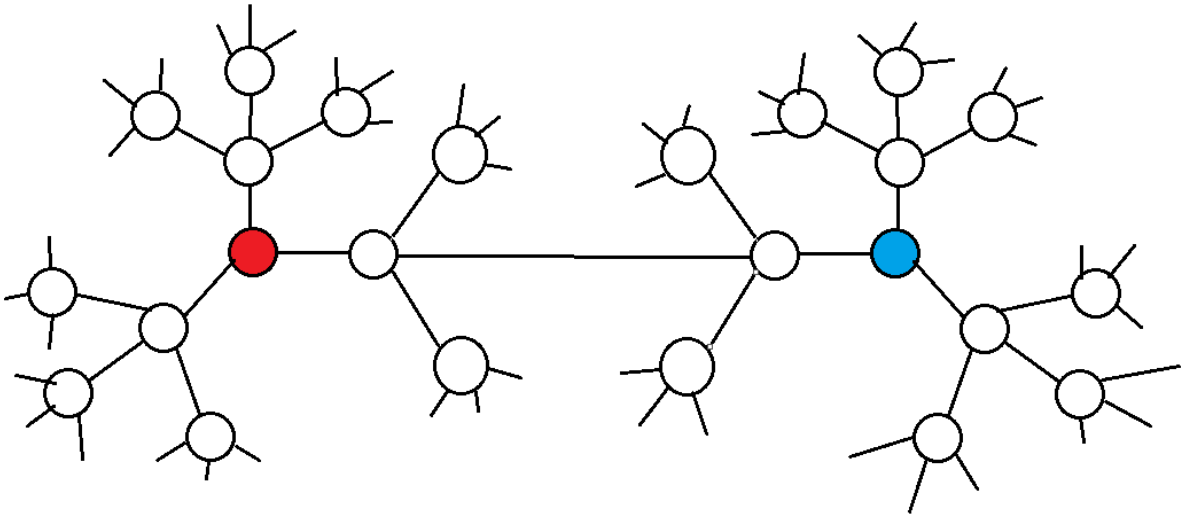
- 隐式图问题
  - 隐式图是仅给出初始结点、目标结点以及生成子结点的约束条件（题意隐含给出），要求按扩展规则应用于扩展结点的过程，找出其他结点，使得隐式图的足够大的一部分编程显式，直到包含目标结点为止。
  - 倒水问题
    - 给你两个容器，容量分别为A， B ，问是否能够经过有限的步骤倒水，得到容量为 C 的水

隐式图问题

- 隐式图问题
  - 倒水问题
    - 给你两个容器（容量分别为A， B）、一台饮水机，问是否能够经过有限的步骤倒水，得到容量为 C 的水
    - 对于每一种状态，都有以下几种转移方式：
      - A倒入B
      - B倒入A
      - A倒空/满
      - B倒空/满
    - 可以转移到不同的状态。
    - 对于每一种状态，我们需要记录是否已经被访问过了（BFS的过程）
    - 可以直接使用高维数组记录（本题），有时需要用 map/set 或哈希算法来记录状态

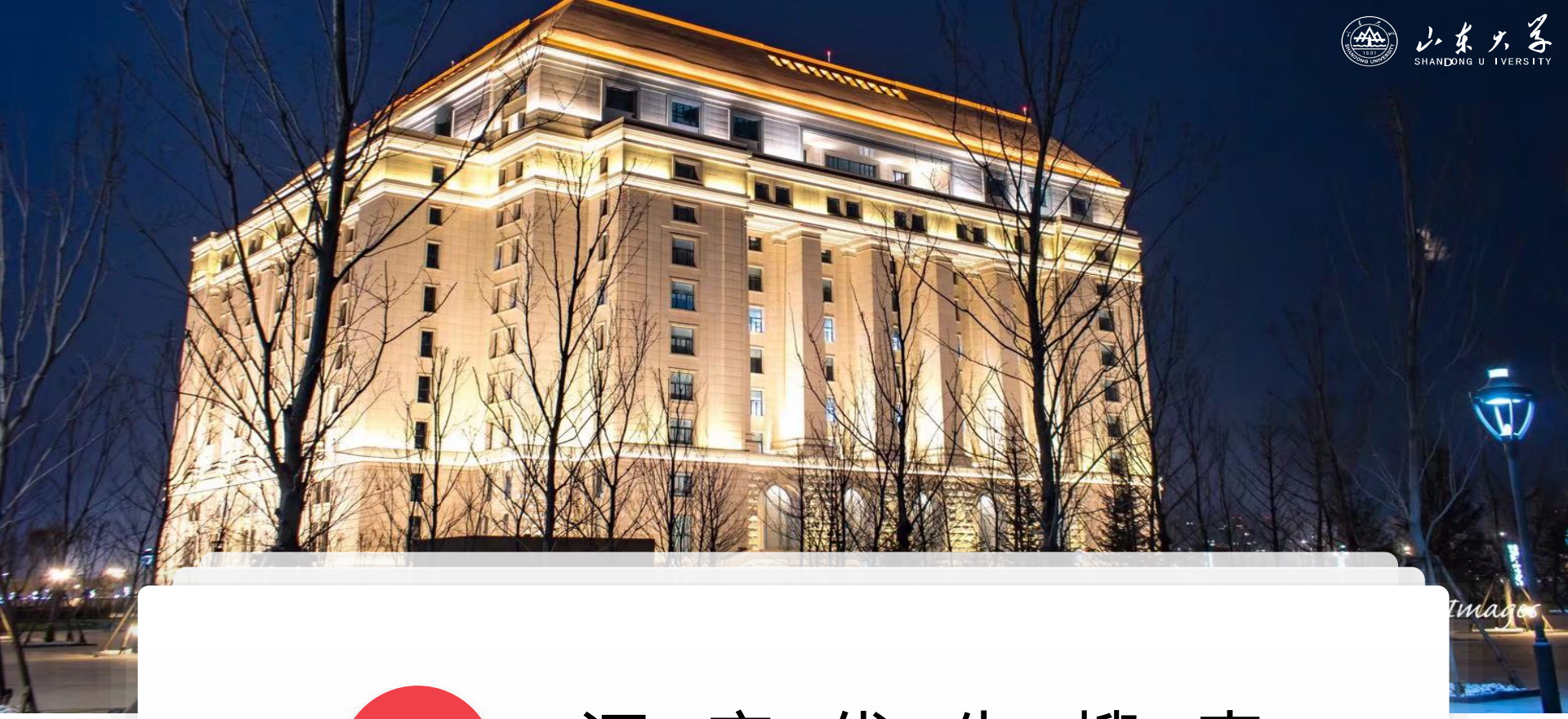
双向BFS

- BFS 优化 —— 双向广度优先搜索 （了解）
  - 在广度优先搜索的基础上进行优化，采用双向搜索的方式，即从起始节点向目标节点方向搜索，同时从目标节点向起始节点方向搜索。
  - 特点：
    - 1.双向搜索只能用于广度优先搜索中。
    - 2.双向搜索扩展的节点数量要比单向少的多。



双向BFS

- BFS 优化 —— 双向广度优先搜索 （了解）
  - 用同一个队列来保存正向和逆向扩展的结点。开始时，将起点坐标和终点坐标同时入队列。这样，第1个出队的坐标是起点，正向搜索扩展队列；第2个出队的坐标是终点，逆向搜索扩展队列。两个方向的扩展依次交替进行。
  - 简单修改vis[][]数组元素的置值方法即可。初始时，vis数组的全部元素值为0，由正向扩展来的结点的vis对应元素值置为1，由逆向扩展来的结点的vis对应元素值置为2。
  - 设当前结点为cur，由cur可以扩展出新结点next。若vis[][]==0，则next结点未访问过，将next结点入队并进行相应设置；若vis[][]!=0且cur和next的vis值不相同，表明正向反向相遇，搜索成功。



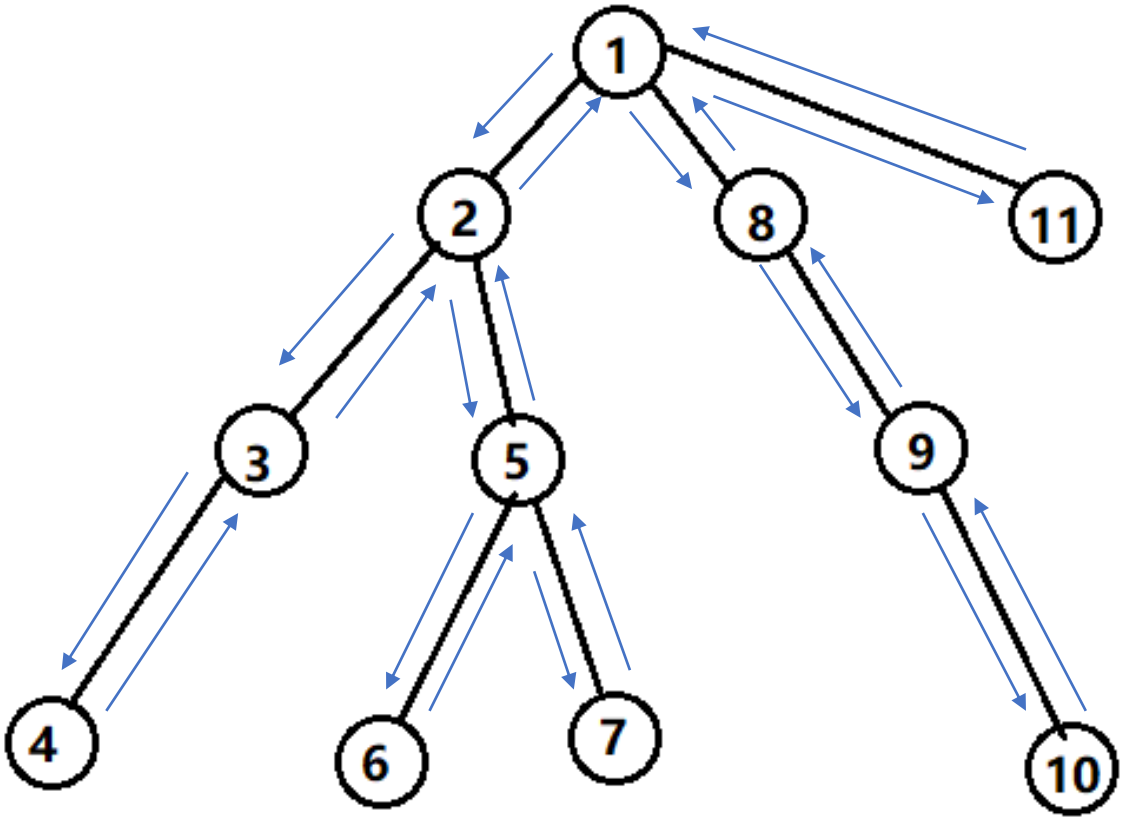
2 深 度 优 先 搜 索

Depth First Search

DFS

- 深度优先搜索（Depth-First-Search）
- 在「数据结构与算法」这门课程中已经接触到，源于图的遍历算法
- 而现在我们考虑的是搜索算法
- 为了求得问题的解，先选择某一种可能情况向前（子结点）探索，在探索过程中，一旦发现原来的选择不符合要求，就回溯至父亲结点重新选择另一结点，继续向前探索，如此反复进行，直至求得最优解。
- 深度优先搜索可以采用栈或递归的方式实现

DFS



DFS

- 一般代码框架

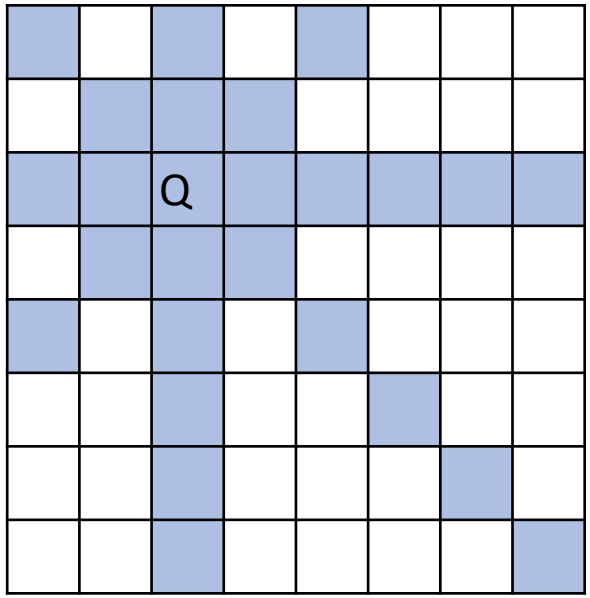
```
void DFS(type n) {  //n用于描述当前状态
    if (符合条件)  {
        //统计答案
        return ;
    }
    if (可以剪枝)  {
        return ;
    }
    for (i:1~p)  { //遍历该阶段的所有决策
        选择可行决策  //可行性剪枝（可以没有）
        标记该点以访问//记忆化（可以没有）
        DFS(n + 1);
        还原进行决策造成的改变
    }
}
```

DFS

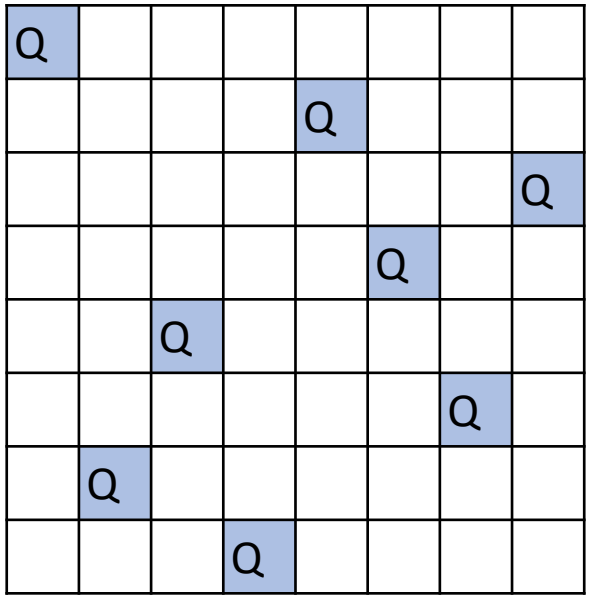
- 剪枝方法
- 最优化剪枝：在搜索过程中，如果当前阶段的代价已经超过我们已知的最小代价，那么此时继续搜索下去就失去了意义。
- 可行性剪枝：如果当前状态已经超出题目给出的约束范围，或者遇到一种不可能到达结果的情况，则停止搜索。
- 记忆化：在搜索过程中记录搜索状态的结果，当重复遍历一个状态时，可以直接返回状态的结果

八皇后问题

- 八皇后问题
- 在棋盘上放置8个皇后，使得它们互不攻击，此时每个皇后的攻击范围为同行同列和同对角线，要求找出所有的解。



皇后的攻击范围



一个可行解

八皇后问题

- 最简单的思路
- 从64个格子中选一个子集，使得子集中恰有8个格子，且任意两个选出的格子不在同一行、同一列或同一个对角线上。
- 这正是子集枚举问题
- 但64个格子的子集有2^{64} =18446744073709551616
- 太大了!

八皇后问题

- 第二个思路
- 从64个格子中选8个格子，这是组合生成问题。
- C(64, 8) = 4426165368，比第一种优秀
- 但还是不够优秀

八皇后问题

- 想法：每一行每一列只能有一个皇后
- —— 记录每一列的皇后放在了哪一行，问题将转换为全排列问题
- 1-8 的全排列只有 8!= 40320个，随后再判某个排列合法
- 更优秀了！

```
int vis[N], a[N];

void dfs(int t)  {
    if (t > 8)  {
        if (满足对角线条件) 输出解
        return ;
    }
    for (int i = 1;i <= 8; i++)
        if (!vis[i]) {
            a[t] = i;
            vis[i] = 1;
            dfs(t + 1);
            vis[i] = 0;
        }
}
```

123
132
213
231
312
321

八皇后问题

- 枚举量其实可以更少，在求全排列的过程中有很多排列是不合法的
- 可以在枚举的过程中检查这个位置是否可以放置棋子（可行性剪枝）
- a[i] 表示第 i 列的棋子在第 a[i] 行
- 全排列可以保证行列不冲突，通过观察可发现

  同一主对角线（左上到右下）上的点(x,y)满足x-y相同

  同一副对角线（左下到右上）上的点(x,y)满足x+y相同

```
int vis[N + 5], a[N + 5];
int left[N * 2 + 5], rigth[N * 2 + 5];

void dfs(int t)  {
    if (t > 8)  {
        输出解
        return ;
    }
    for (int i = 1;i <= 8; i++)
        if (!vis[i] && !left[i - t + N] && !rigth[i + t]) {
            a[t] = i;
            vis[i] = 1;
            left[i - t + N] = 1;
            rigth[i + t] = 1;
            dfs(t + 1);
            vis[i] = 0;
            left[i - t + N] = 0;
            rigth[i + t] = 0;
        }
}
```

选数问题

- 给定n个正整数，要求选出K个数，使得选出来的K个数的和为sum，求合法方案数
- K<=n<=16

- 例如 {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}, n = 10, K = 3, sum = 10

- 2 + 3 + 5 = 10
- 1 + 4 + 5 = 10
- 1 + 3 + 6 = 10
- 1 + 2 + 7 = 10

- 所以ans为4

选数问题

- 一个典型的子集枚举问题，可以枚举所有子集判断是否合法
- 复杂度 $O(K \cdot 2^n)$

选数问题

- 但是这里面有很多 显然 不可能合法的情况
- 是否可以直接跳过
- 哪些情况？
  - 选的数的个数超过了 K
  - 选的数的和超过了 sum

```
void dfs(int sum, int kk, int x)  {
    if (sum== s && kk == k)  {
        cnt++;
        return;
    }
    if (x > n || sum > s || kk > k)  return ;
    dfs(sum, kk, x + 1);
    dfs(sum + a[x], kk + 1, x + 1);
}
```

再卖菜CSP201809-4

问题描述

在一条街上有n个卖菜的商店，按1至n的顺序排成一排，这些商店都卖一种蔬菜。

第一天，每个商店都自己定了一个正整数的价格。店主们希望自己的菜价和其他商店的一致，第二天，每一家商店都会根据他自己和相邻商店的价格调整自己的价格。具体的，每家商店都会将第二天的菜价设置为自己和相邻商店第一天菜价的平均值（用去尾法取整）。

注意，编号为1的商店只有一个相邻的商店2，编号为n的商店只有一个相邻的商店n−1，其他编号为i的商店有两个相邻的商店i−1和i+1。

给定第二天各个商店的菜价，可能存在不同的符合要求的第一天的菜价，请找到符合要求的第一天菜价中字典序最小的一种。

字典序大小的定义：对于两个不同的价格序列(a_1, a_2, ..., a_n)和(b_1, b_2, b_3, ..., b_n)，若存在i (i>=1), 使得a_i<b_i，且对于所有j<i，a_j=b_j，则认为第一个序列的字典序小于第二个序列。

输入格式

输入的第一行包含一个整数n，表示商店的数量。

第二行包含n个正整数，依次表示每个商店第二天的菜价。

输出格式

输出一行，包含n个正整数，依次表示每个商店第一天的菜价。

样例输入

```
8
2 2 1 3 4 9 10 13
```

样例输出

```
2 2 2 1 6 5 16 10
```

数据规模和约定

对于30%的评测用例，2<=n<=5，第二天每个商店的菜价为不超过10的正整数；

对于60%的评测用例，2<=n<=20，第二天每个商店的菜价为不超过100的正整数；

对于所有评测用例，2<=n<=300，第二天每个商店的菜价为不超过100的正整数。

请注意，以上都是给的第二天菜价的范围，第一天菜价可能会超过此范围。

再卖菜

- 先考虑一下这个题如何使用 DFS 进行搜索
- 然后再考虑如何优化以及能够优化到什么程度

再卖菜

再卖菜	02-01 22:27	551B	C++	运行超时	20	运行超时	3.093MB

- 最简单的想法就是暴力枚举每个商店的菜价，n 重循环最后判断是否合法，n 重循环可以使用递归来实现。
- 复杂度$O(n \cdot 300^n)$，样例都跑不动
- 但是依旧可以得分！！！
  所以考场上一定不要放弃部分分

```
void dfs(int x)  {
    if (flag) return ;
    if (x > n)  {
        if ((d1[1] + d1[2]) / 2 != d2[1]) return ;
        if ((d1[n] + d1[n - 1]) / 2 != d2[n]) return ;
        for (int i = 2;i < n; i++)
            if ((d1[i - 1] + d1[i] + d1[i + 1]) / 3 != d2[i]) return ;
        flag = 1;
        for (int i = 1;i <= n; i++)
            cout << d1[i] << " ";
        return ;
    }
    for (int i = 1;i <= 300; i++)  {
        d1[x] = i;
        dfs(x + 1);
    }
}
```

再卖菜

再卖菜	02-01 22:50	1.019KB	C++	运行超时	80	运行超时	3.031MB

- 显然需要剪枝，可行性剪枝
- 在枚举每一个商店的菜价时需要判断是否满足题目要求
  - 如果 (d1[x-2] + d1[x-1] + d1[x]) / 3 < d2[x – 1] 说明 x 的菜价低了，达不到x-1的平均数
  - 如果 (d1[x-2] + d1[x-1] + d1[x]) / 3 == d2[x – 1] 说明 x 的菜价合适，符合题目要求
  - 否则就太大了，再搜索下去没有意义，直接回溯
- 这样已经可以拿到80啦

```
void dfs(int x)  {
    if (flag) return ;
    if (x > n)  {
        if ((d1[n] + d1[n - 1]) / 2 != d2[n]) return ;
        flag = 1;
        for (int i = 1;i <= n; i++)
            cout << d1[i] << " ";
        return ;
    }
    for (int i = 1;i <= 300; i++)  {
        d1[x] = i;
        if (x == 1) dfs(x + 1);
        if (x == 2) {
            if ((d1[1] + d1[2]) / 2 < d2[1]) continue;
            if ((d1[1] + d1[2]) / 2 == d2[1]) dfs(x + 1);
            if ((d1[1] + d1[2]) / 2 > d2[1]) return ;
        }
        if (x >= 3)  {
            if ((d1[x - 2] + d1[x - 1] + d1[x]) / 3 < d2[x - 1]) continue;
            if ((d1[x - 2] + d1[x - 1] + d1[x]) / 3 == d2[x - 1]) dfs(x + 1);
            if ((d1[x - 2] + d1[x - 1] + d1[x]) / 3 >  d2[x - 1])return ;
        }
```

再卖菜

- 考虑如何进一步优化
- 在我们剪枝的过程中，如果发现 (d1[x-2]+d1[x-1]+d1[x])/3 < d2[x-1] 则 continue
- 这个操作说明菜价太低，而往往在一个范围内这个不等式一直是成立的，我们没有必要一直 continue，而只要把等号成立的范围求出来即可。
- 根据题目要求，(d1[x-2]+d1[x-1]+d1[x])/3=d2[x-1]，而d1[x-2]、d1[x-1] 与 d2[x-1]都是已知的（想想为什么）
- 由于是下取整，那么余数可能是 0，1 或 2
- 由此可以计算出 d1[x]=3*d2[x-1]-d1[x-2]-d1[x-1]+0/1/2

再卖菜

再卖菜	02-01 23:06	1.463KB	C++	运行超时	80	运行超时	3.039MB

还是80分，别急
我们还能进一步
优化！！

```
void dfs(int x)  {
    if (flag) return ;
    if (x > n)  {
        if ((d1[n] + d1[n - 1]) / 2 != d2[n]) return ;
        flag = 1;
        for (int i = 1;i <= n; i++)
            cout << d1[i] << " ";
        return ;
    }
    if (x == 1)  {
        for (int i = 1;i <= 200; i++)  {
            d1[1] = i;
            dfs(x + 1);
        }
    }
    if (x == 2)  {
        for (int i = 0;i < 2; i++)  {
            d1[2] = 2 * d2[1] - d1[1] + i;
            if (d1[2] > 0) dfs(x + 1);
        }
    }
    if (x >= 3)  {
        for (int i = 0;i < 3; i++)  {
            d1[x] = 3 * d2[x - 1] - d1[x - 2] - d1[x - 1] + i;
            if (d1[x] > 0) dfs(x + 1);
        }
    }
}
```

再卖菜

- 我们来继续优化，问题出在哪里了呢？
- 首先来看一看 80 的 DFS 函数中，主要用到了哪些信息
  - x/d1[x-2]/d1[x-1] -> 搜索的“状态”
- 在同一个状态下 DFS 的行为永远是一样的（为什么？）
- 也就是说 如果两次 DFS 都到达了同一个状态，那么后来的那次 DFS是重复的，是多余的，我们知道这个状态不可能得到题目要求的解，也就没有必要从这个状态继续搜索下去了。

再卖菜

- 那么怎样可以记录哪些状态已经访问了，哪些状态没有被访问过
- 之前说过，DFS 的状态可以被 3 个元素所标记
- 那么可以使用一个三维的布尔型数值 vis 来记录
- 如果 vis[s][y][z] = 1说明状态 { x=s, d1[x-2] = y, d1[x-1] = z} 已经被访问过了
- 反之说明没有被访问过
- 如果被访问过就直接回溯，不进行往后的搜索

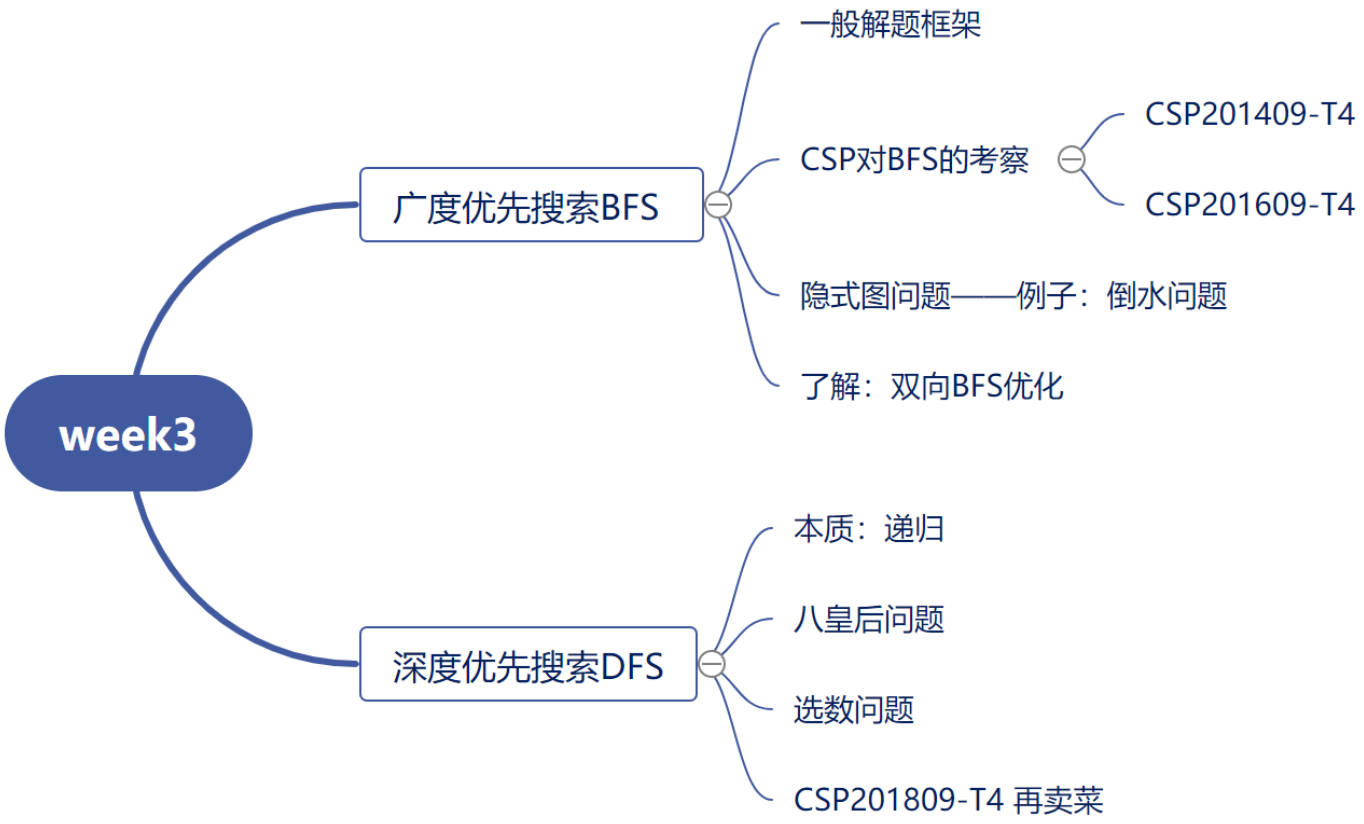
再卖菜

再卖菜	02-01 23:29	1.998KB	C++	正确	100	0ms	3.976MB

记忆化搜索，
是一种重要的剪枝策略

```
void dfs(int x, int s1, int s2)  {
    if (vis[x][s1][s2]) return ;
    vis[x][s1][s2] = 1;
    if (flag) return ;
    if (x > n)  {
        if ((d1[n] + d1[n - 1]) / 2 != d2[n]) return ;
        flag = 1;
        for (int i = 1;i <= n; i++)
            cout << d1[i] << " ";
        return ;
    }
    if (x == 1)  {
        for (int i = 1;i <= 200; i++)  {
            d1[1] = i;
            dfs(x + 1, s2, d1[x]);
        }
    }
    if (x == 2)  {
        for (int i = 0;i < 2; i++)  {
            d1[2] = 2 * d2[1] - d1[1] + i;
            if (d1[2] > 0) dfs(x + 1, s1, d1[x]);
        }
    }
    if (x >= 3)  {
        for (int i = 0;i < 3; i++)  {
            d1[x] = 3 * d2[x - 1] - d1[x - 2] - d1[x - 1] + i;
            if (d1[x] > 0) dfs(x + 1, s2, d1[x]);
        }
    }
}
```

总结


week3
广度优先搜索BFS
一般解题框架
CSP对BFS的考察
CSP201409-T4
CSP201609-T4
隐式图问题——例子：倒水问题
了解：双向BFS优化
深度优先搜索DFS
本质：递归
八皇后问题
选数问题
CSP201809-T4 再卖菜

感谢收听

Thank You For Your Listening

本次课件引用了《挑战程序设计竞赛》、cuichen的PPT