搜索专题

湖南师大附中 许力

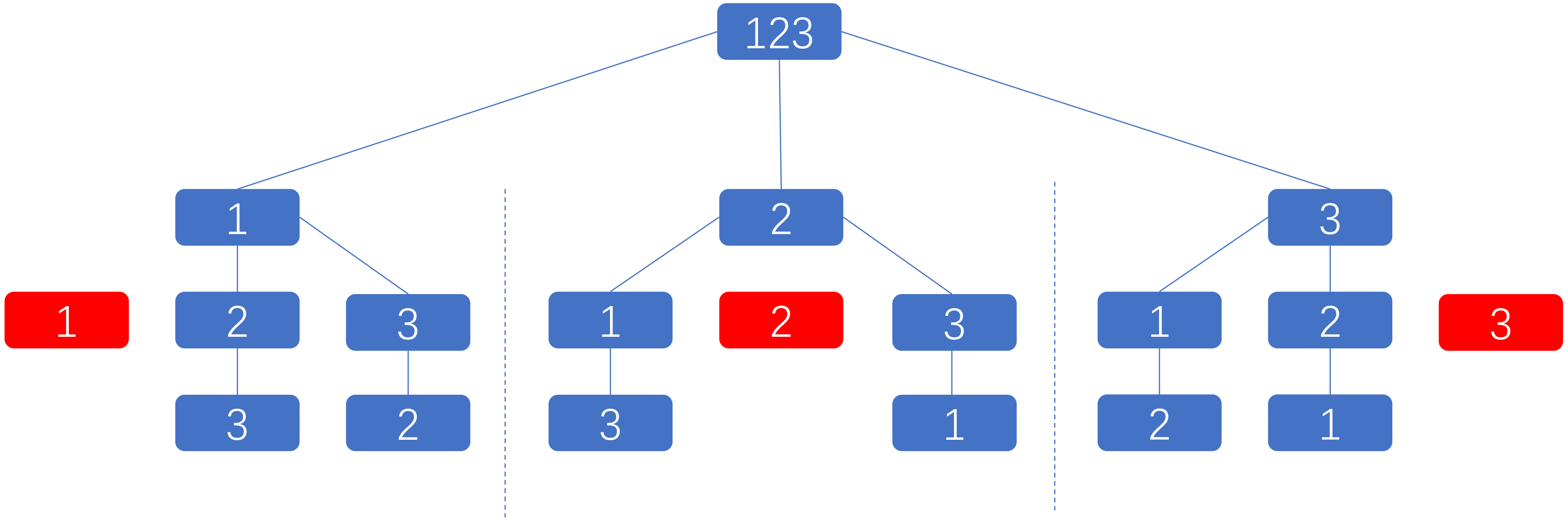
目录

搜索

- 什么是搜索?
  – 利用计算机的高速运算性能，根据题意穷举解集的部分或全部

搜索

- 比如全排列问题，类似于在一棵搜索树中，不断往叶节点拓展，如无法拓展则返回上一层，直到访问完所有可能的节点为止



全排列问题的递归写法

- 在递归中：
  - 每递归调用一次，相当于在栈顶压入新的元素
  - 而每次递归调用结束返回上一层，相当于释放当前栈顶的元素

- 利用递归的这一特性，我们可以：

1. 逐个尝试选取一个数字
2. 生成一种方案后把数字回收，返回上一个位置再尝试新的方案
3. 用flag数组标记哪些数字已经放下去了

尝试摆放

```
for (int i = 1; i <= n; i ++)
    if (!flag[i]){  //如果数字 i还没有被试过
        a[j] = i;  //将数字 i放在 j号位置
        flag[i] = 1; //打标记 i 已被试过
        dfs(j + 1)  // 尝试 j的下一个位置
        flag[i] = 0;  //将刚才尝试的数字 i收回以备下一次尝试
    }
```

输出方案

```
void dfs(int j)  //尝试摆放第 j 个位置
{
    if (j > n){   //全部 n 个位置已摆放完毕
        for (int i = 1; i <= n; i ++)
            printf("%d ", a[i]);
        printf("\n");
    }
    else 尝试摆放
    return;
}
```

全排列问题的递归写法

```
void dfs(int j) // j代表尝试摆放第 j个位置
{
    if (j > n)   //此时全部 n个位置已摆放完毕
        输出方案;
    else for (int i = 1; i <= n; i ++)
            if (flag[i] == 0)   //如果数字 i还未放入
            {
                a[j] = i;     //将数字 i放入第 j个位置
                flag[i] = 1;   //标记数字 i已放入
                dfs(j + 1);   //递归尝试下一个位置 j+1
                flag[i] = 0;   //将刚才尝试的数字收回
            }
    return;
}

dfs(1);  // main()函数中调用
```

深搜的基本模型

```
void dfs(int step)
{
    判断边界
        符合条件则输出;
    else for (int i = 1; i <= n; i ++)
            if ( i 点未被标记)
            {
                尝试;
                标记已访问;
                继续下一步 dfs(step + 1);
                释放标记;
            }
    return;
}
```

素数环

- 从1~n（n≤20）组成一个数字序列，要求相邻的两个数的和是一个素数，这样的一个序列称为“素数环”，给定n，要求输出所有的素数环

sample1	sample2
Input 6 Output 1 4 3 2 5 6 1 6 5 2 3 4	Input 8 Output 1 2 3 8 5 6 7 4 1 2 5 8 3 4 7 6 1 4 7 6 5 8 3 2 1 6 7 4 3 8 5 2

素数环

- 实际上还是一个全排列问题，不过多了限制条件

```
void dfs(int j)
{
    if (j > n) printf(a[]);
    else for (int i = 1; i <= n; ++ i)
            if (flag[i] == 0 && prime[i + j])
            //预处理prime数组
            {
                尝试每一种可能
                继续下一步 dfs(j + 1)
            }
}
```

深搜的常见处理技巧

- 自然数拆分：将整数 n 拆解为若干个小于 n 的自然数之和，输出所有拆分方案

  – 直接深搜无法保证方案中的数不下降，比如无法保证在搜出 1+2+2 后不再搜出 2+1+2

  – 常用的技巧是在搜索的时候，还增加一个变量，用于记录上次搜出的方案中的最后一个数

深搜的常见处理技巧

```
void dfs(int step, int x, int sum)
{
    if (sum == n) 输出方案;
    else for (int i = x; i <= n - sum; i ++)
            {
                // x为上次搜出的最后一个数
                // sum为当前已搜出的数之和
            }
}
```

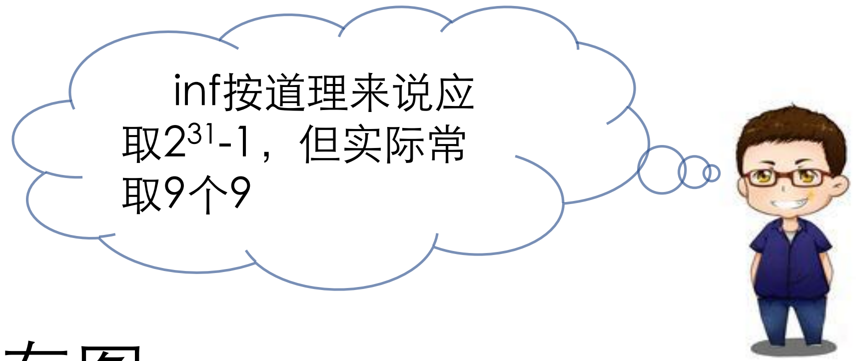
图上的DFS模型

```
void dfs(int u)
{
    flag[u] = 1;
    输出 u;
    for (int v = 1; v <= n; v ++)
    {
        if (e[u][v] && !flag[v])
            dfs(v);
        //逐个尝试与 u相连的点
        //如果 u有边连向 v，且 v未被访问
        //从 v出发继续遍历
     }
}
```

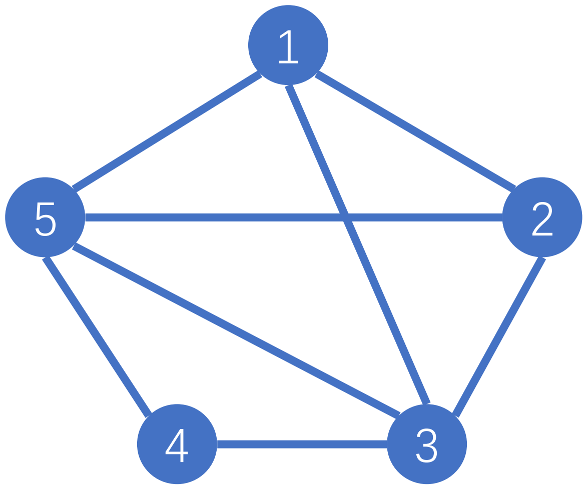
插讲一条：读图和存图

- 邻接矩阵
- 邻接表（链式前向星）

邻接矩阵存图



- 要遍历图，首先需要解决一个问题：存图
- 邻接矩阵存图，空间复杂度O(n^2)



	1	2	3	4	5
1	0	inf	inf	inf	inf
2	inf	0	inf	inf	inf
3	inf	inf	0	inf	inf
4	inf	inf	inf	0	inf
5	inf	inf	inf	inf	0

	1	2	3	4	5
1	0	1	1	inf	1
2	1	0	1	inf	1
3	1	1	0	1	1
4	inf	inf	1	0	1
5	1	1	1	1	0

邻接矩阵存图

- 邻接矩阵初始化

```
scanf("%d%d", &n, &m);    // n个顶点, m条边
for (int i = 1; i <= n; i ++)
    for (int j = 1; j <= n; j ++)
        if (i == j) e[i][j] = 0;    //设顶点到自己为 0
            else e[i][j] = INF;     //初次之外, 初始没有边
```

邻接矩阵存图

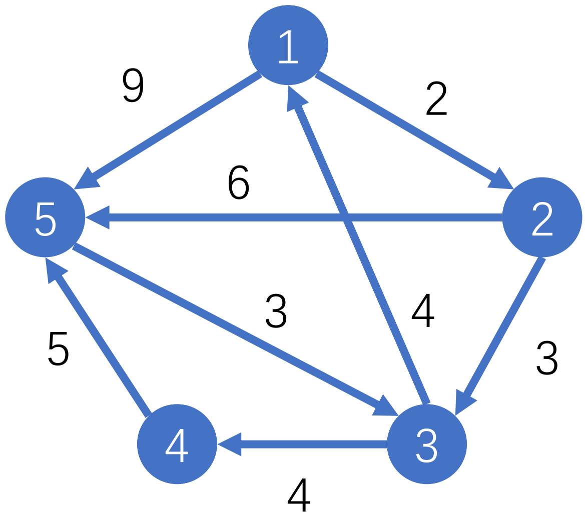
- 读入图

```
for (int i = 1; i <= m; i ++)
{
    scanf("%d%d", &u, &v);
    e[u][v] = e[v][u] = 1;  //无向图对称
 }

for (int i = 1; i <= m; i ++)
{
    scanf("%d%d%d", &u, &v, &w);
    e[u][v] = w;  //有向带权图
 }
```

邻接矩阵存图

- 如果是带权图、有向图，则如下：

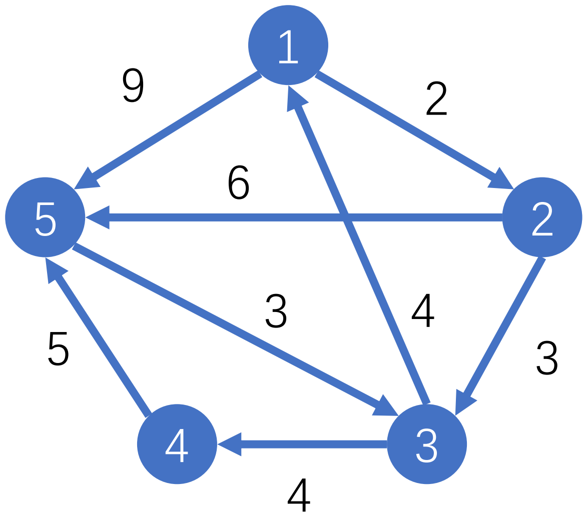


	1	2	3	4	5
1	0	inf	inf	inf	inf
2	inf	0	inf	inf	inf
3	inf	inf	0	inf	inf
4	inf	inf	inf	0	inf
5	inf	inf	inf	inf	0

	1	2	3	4	5
1	0	2	inf	inf	9
2	inf	0	3	inf	6
3	4	inf	0	4	inf
4	inf	inf	inf	0	5
5	inf	inf	3	inf	0

邻接矩阵存图

- 邻接矩阵存图，一般只适用于顶点数不超过10,000的图，而且无法存储重边



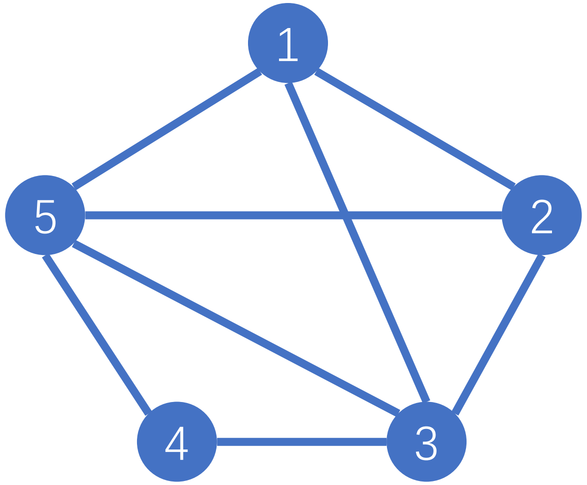
	1	2	3	4	5
1	0	inf	inf	inf	inf
2	inf	0	inf	inf	inf
3	inf	inf	0	inf	inf
4	inf	inf	inf	0	inf
5	inf	inf	inf	inf	0

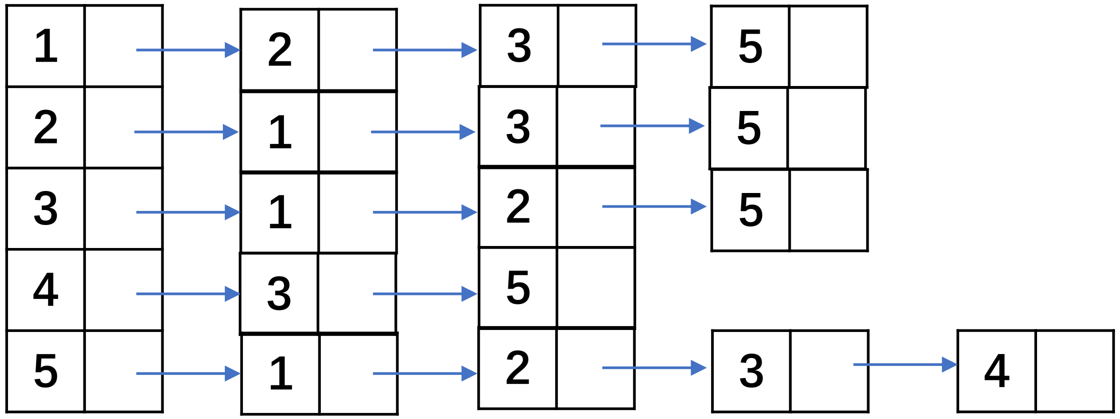
➡

	1	2	3	4	5
1	0	2	inf	inf	9
2	inf	0	3	inf	6
3	4	inf	0	4	inf
4	inf	inf	inf	0	5
5	inf	inf	3	inf	0

链式前向星

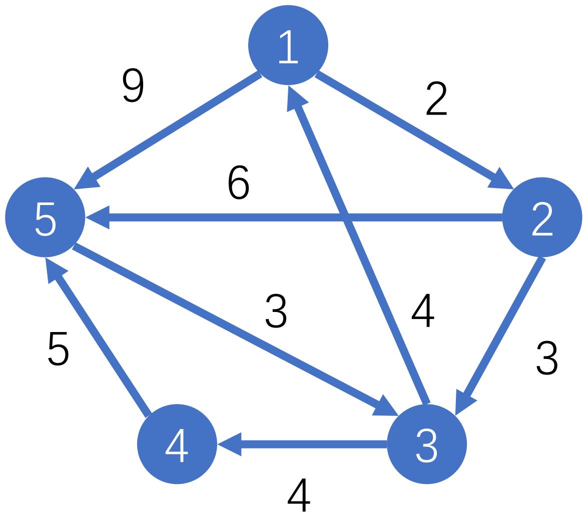
- 链式前向星的存储原理基于邻接表，空间复杂度O(n+m)
- 邻接表如下：
- 和邻接矩阵把每条边都存储下来不同，邻接表存图只记录每个顶点所连向的边

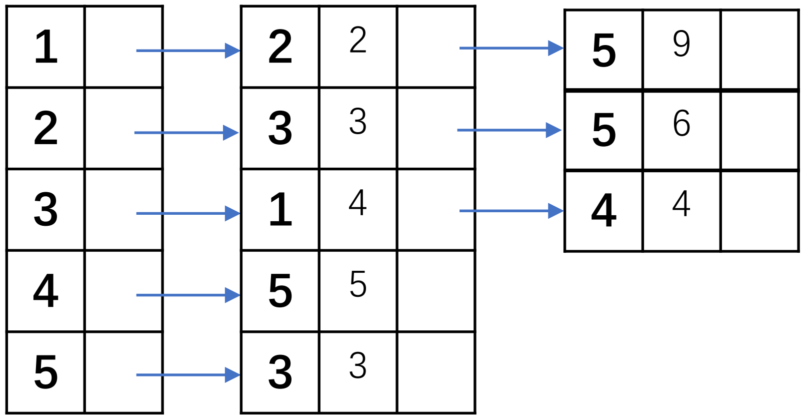




邻接表存图

- 有向图、带权图则如下：





邻接表的数组模拟实现

- 但是这样有一个很严重的问题，我们不知道某个顶点u，与之相连的点有多少个

```
struct edge
{
    int u, v, w;    //起点、终点、边权
    int *next;    //链表的下一个元素
} e[N];
```

邻接表的vector实现

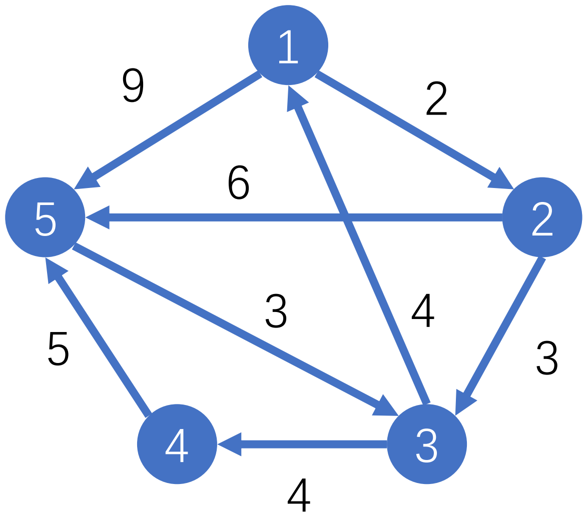
- 用 vector 数组 vec[ ] 来记录与顶点 u 相连的点

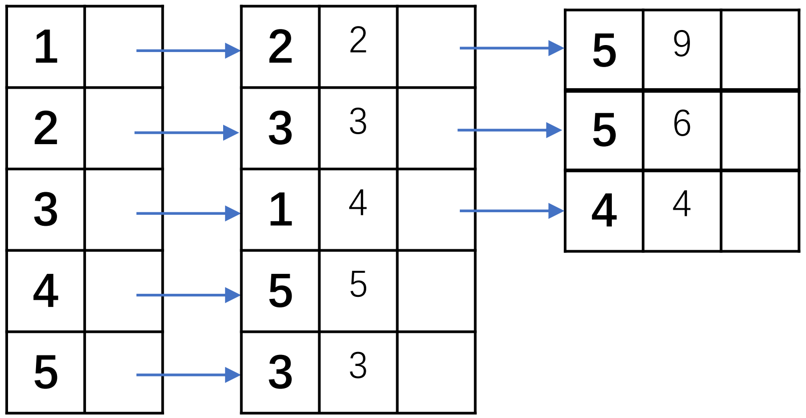
```
vector<int> v;

for (int i = 1; i <= m; i ++)
{
    scanf("%d%d", &u, &v);
    vec[u].push_back(v);
}
```

邻接表存图

- 邻接表存图，可以存储点数100,000的图，而且处理重边也没问题
- 在邻接表的原理基础上存边集数组后做优化，就得到最优的存图方式：“链式前向星”





链式前向星

```
struct edge
{
    int next, to;  //next记录下条边的编号，to记录当前边的终点
} h[N * 2];

void add(int u, int v)
{
    //head记录链首
    h[++tot].next = head[u];    //将新边的 next指向当前链首
    head[u] = tot;   //更新链首为新边 tot
    h[tot].to = v;  //新边终点 v
}

for (int i = head[j]; i != 0; i = h[i].next)
//用前向星枚举每一条以 j为起点的边
{

}
```

图上的典型搜索问题

- 黑色为障碍点， 求起点 (1,1) 到终点 (p,q) 的最短路径

- DFS需要记录当前点坐标 (x,y) 及走过的步数


```
void dfs(int x, int y, int step)
{

}
```

图上的典型搜索问题

- 四个搜索的方向，可以用数组 next[4][2] 来控制

```
int next[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
```

- 分别对应向右、向下、向左、向上

图上的典型搜索问题

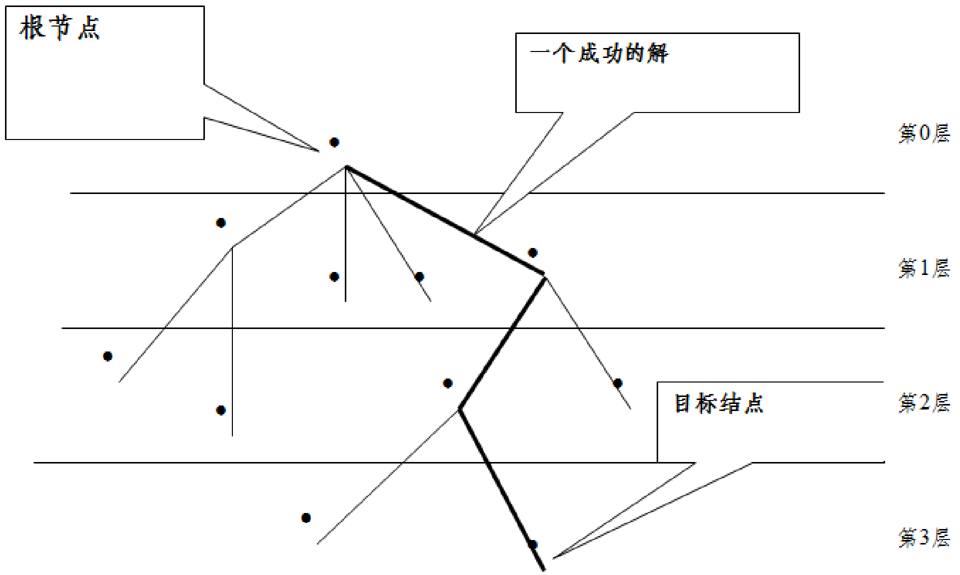
```
void dfs(int x, int y, int step)
{
    int next[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
    int tx, ty, k;   //tx、ty是移动之后的当前点，k是方向
    if (达到目标) 维护最小值
    else for (k = 0; k <= 3; k ++)  //枚举 4 个方向
    {
        tx = x + next[k][0];
        ty = y + next[k][1];    //计算移动之后的点坐标
        if (tx < 1 || tx > n || ty < 1 || ty > m) continue;//越界
        if (a[tx][ty] == 0 && flag[tx][ty] == 0)
        //如果移动之后的点不是障碍，且未访问过
        {
            flag[tx][ty] = 1;
            dfs(tx, ty, step + 1);
            flag[tx][ty] = 0;
        }
    }
}
```

深搜

- 适用于求解一条从初始节点至目标节点的可能路径的问题。若要求得最优解，必须记下达到目标的路径和相应的路程值，并维护最值
- 深度优先搜索并不是以最快的方式搜索到解，搜索效率取决于目标节点在搜索树中的位置

剪枝优化

- 所谓剪枝，就是尽早发现并结束没可能出解的状态分枝，从而提高效率



剪枝

- 可行性剪枝
  - 当搜索到一个状态时，如果可以判断这个状态之后的状态都不合法，则直接退出当前状态。一般用于求方案数的题目中
- 最优性剪枝
  - 当搜索到一个状态时，如果可以判断这个状态之后的状态都不会比当前的最优状态更优，则直接退出当前状态。一般用于求最优解的题目中

小木棍

- 给出n（n≤65）根小木棍的长度len（len≤50），已知这n根小木棍原本是由若干根长度相同的长木棍截断而来，求长木棍的最小可能长度

sample
Input 9 5 2 1 5 2 1 5 2 1 Output 6

小木棍

- 样例数据看不出什么，再看一组数据：

46、45、36、36、36、24、19、16、14、13（已排好序）

46+36+13 = 45+36+14 = 36+24+19+16 = 95

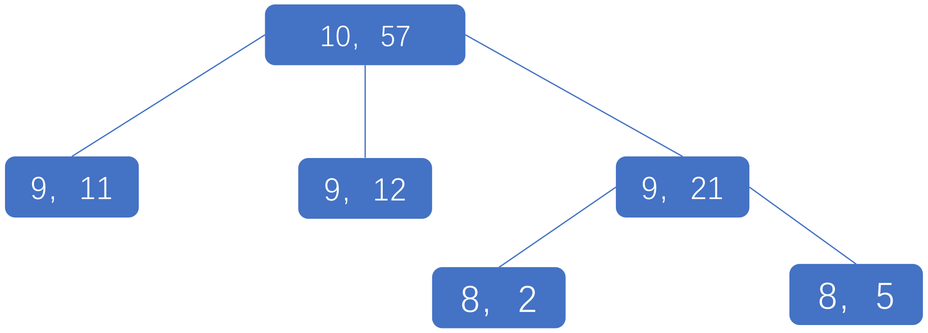
- 是否从最大的小木棒长度开始，dfs到小木棒长度总和的一半结束？
- 基本思路是对每个假设的长木棍长度，dfs尝试能否拼成功

小木棍

- DFS可以考虑记录两个状态：

1. 当前尝试后还剩的小木棍数量
2. 当前尝试后还剩的长木棍（假设的）长度

- 以数据46、45、36、36、36、24、19、16、14、13为例：
- 假设当前要尝试拼接的长木棍长度为57，那么dfs(10,57)，接下来的状态：



小木棍

- 剪枝1：这个假设的长木棍长度，应该是小木棍长度总和的因子（285：95、57）
- 剪枝2：在同一位置尝试下一步时，所有长度相同的小木棍只尝试一次
- 剪枝3：如果拼接第 i 根长木棍失败需要更换小木棍，不会更换第一根，原因是如果换了第一根就成功了，那么第一根必然会出现在另一种成功的拼法里

小木棍

- 剪枝4：只有当前还剩的长木棍长度不小于要尝试的小木棍长度，才要DFS，而且如果刚好相等则直接返回上一层
- 剪枝5：每次拼长木棍时，只要当前尝试的不是第一根小木棍，就不要从头开始尝试，而是应该从刚才所试小木棍的下一条开始。因此，类似于数字拆分，应该增加一个变量用于记录当前已尝试的最后一根小木棍的长度

- 为什么按从大到小的顺序DFS可以大大减少递归层数？

生日蛋糕

- 制作一个体积为Nπ的M层生日蛋糕，每层都是一个圆柱体。设从下往上数第 i(1 ≤ i≤ M)层蛋糕是半径为 R_i 、高度为 H_i 的圆柱。当$i<M$时，要求$R_i>R_{i+1}$且$H_i>H_{i+1}$
- 由于要在蛋糕上抹奶油，为尽可能节约经费，对给出的N和M，找出蛋糕的制作方案（适当的R_i和H_i的值），使蛋糕侧面积S最小

分析

- 直接DFS：
  - 考虑对于每一个状态需要记录的变量：当前已做完i层，半径为r_i高度为h_i，V表示剩余的体积，S表示当前的表面积和
  - 然后转移枚举下一层的半径和高

分析

- 可行性剪枝
  - 如果当前的侧面积+剩余部分侧面积的下限≥当前最优解，应该返回

- 最优性剪枝
  - 剩余的体积太少，剩余部分最小化也会超过Nπ
  - 要最小化剩余部分体积，剩余每一层的半径和高都要最小化。那么最后一层的半径和高都是1，倒数第二层都是2，第m-k+1层都是k。所以$V_{min}=1^3+2^3+\cdots+(m-k+1)^3$

分析

- 最优性剪枝

- 剩余的体积太多，剩余部分最大化也达不到Nπ

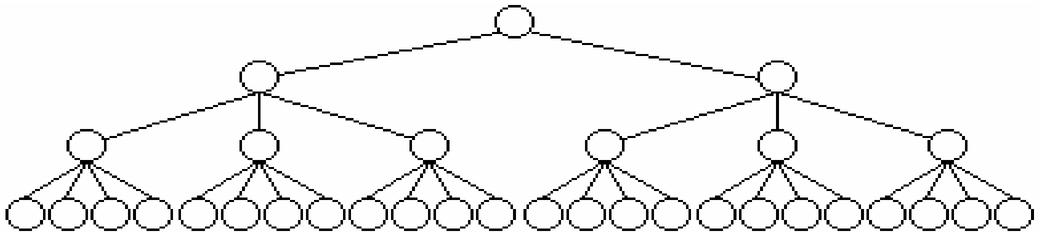
- 怎样最大化剩余部分的体积?
- 类似的方法，每一层的半径和高都要最大化
- 即第i+k层半径R_i-k，高H_i-k

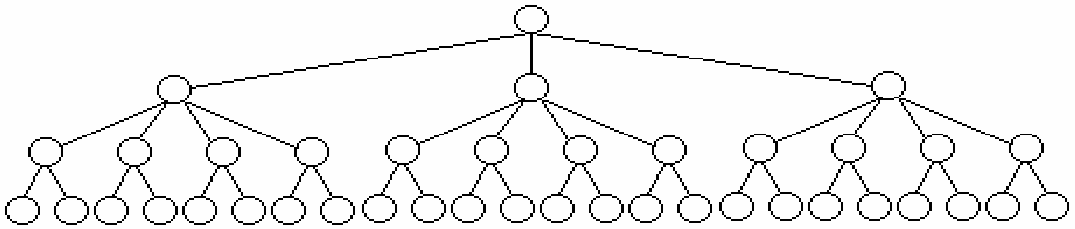
参考代码

```
void dfs(int x, int r, int h, int S, int V)
//还剩下 x层, 之前一层半径 r, 高度 h, 目前已经累计面积 S, 剩余体积 V
{
    if (vmin[x] > V) return;
    //剩余体积太小, 最小化也会超过
    if (x * (r - 1) * (r - 1) * (h - 1) < V) return;
    //剩余体积太大, 最大化也达不到
    if (x == 0) { ans = min(ans, S); return; }
    //如果堆完了, 就更新答案
    for(int i = x; i < r; ++ i)        //枚举当前一层的半径
    {
        if (S + 2 * V / i > ans) continue;        //可行性剪枝
        for(int j = x; j < h; ++ j)        //枚举当前一层的高度
            dfs(x-1, i, j, S+2*i*j, V-i*i*j);
    }
}
```

搜索顺序的选择

- DFS在搜索顺序是任意的。在其它条件相当的前提下，让可取值少的节点优先
- 比如下左图，从第1层剪去1棵子树，则从所有应当考虑的3元组中一次消去12个3元组；对于下右图，虽然同样从第1层剪去1棵子树，却只从应当考虑的3元组中消去8个3元组。前者的效果比后者更好





木板间隔问题

- 有2n个木块，每个木块标上1到n的自然数中的一个，每个数字会出现在两个木块上。把这些木块排成一排，要求是：标号为i的两个木块之间要间隔i个其它木块。比如说n=3的情况，下面就是一个可行的排列：

  3，1，2，1，3，2

- 对给定的n，求出一个可行排列

分析

- 从大到小优于从小到大
- 如果是从小到大的顺序，每确定一块木板的值，对后续搜索几乎不能起到约束作用
- 但如果从大到小，根据条件“标号为i的两个木块之间要间隔i个其它木块”，就可以去掉很多不合理的取值

搜索顺序确定的原则

1. 取值范围小的先搜索

2. 一个搜索元素确定以后对其它搜索元素取值范围的影响称为制约力。制约力较大的搜索元素先搜索

质数方阵

- 在一个5*5的方阵中填入数字，使得沿行，沿列及两个对角线的5个数字都能构成一个5位质数
- 所有质数的各位数字之和必须等于一个常数。这个常数和方阵左上角中的数字预先给出。若存在多个解，须全部得出

1	1	3	5	1
1	4	0	3	3
3	0	3	2	3
5	3	2	0	1
1	3	3	1	3

分析

1. 最后一行和最后一列：可以填的数字只有：1，3，7，9

2. 两条对角线：与方阵中的所有五位质数有关

3. 其他行列：特殊性取决于行列中已经确定的格子个数

1	1	3	5	1
1	4	0	3	3
3	0	3	2	3
5	3	2	0	1
1	3	3	1	3

分析

- 根据元素的取值范围和制约力确定搜索顺序

1. 最后一行和最后一列是取值范围最小的搜索元素，而且它们对其他所有的元素都有一定的制约力，因此要放在搜索序列的最前面

2. 两条对角线同样影响到其他所有的搜索元素，制约力在剩下的格子中是最大的，因此也应当优先搜索

3. 剩下的行列依据它们取值范围的大小确定搜索顺序

DFS的缺陷

- 由于要存储所有已被扩展节点，所以需要的存储空间往往比较大
- 明显的深度过大会爆栈，这时需要迭代加深

迭代加深

- 什么是迭代加深？
  - DFS基于递归，随着层数增加，复杂度呈爆炸性增长
  - 如果题目明确要求解所在的层次越低越好，则我们可以按照从小到大枚举递归的层次限定搜索的深度

迭代加深

- 一般会设定一个初始深度maxd（通常是1）
- DFS进行搜索，限制层数为maxd，如果找到答案就结束，否则maxd++，继续DFS
- 如果搜索某一层时没有找到新的状态，说明搜索已经结束，原问题无解

埃及分数

- 对于一个分数 a/b，可以拆分成多个单位分数（即分子为1）的和
- 比如：2/3 = 1/2 + 1/6，但不允许 2/3 = 1/3 + 1/3，因为分母出现了重复数字

- 埃及分数是这样一种拆分方式：首先是加数越少越好；其次如果加数相同，则最小的分数越大越好；如果最小的一样大，则次小的越大越好，依次类推。比如：
  - 19/45 = 1/3 + 1/12 + 1/180
  - 19/45 = 1/3 + 1/15 + 1/45
  - 19/45 = 1/3 + 1/18 + 1/30
  - 19/45 = 1/4 + 1/6 + 1/180
  - 19/45 = 1/5 + 1/6 + 1/18
- 最后一种最好，因为1/18最大。现在给出a和b（a,b≤1000）,求最好的埃及分数表示

分析

- DFS的递归层数无上限，不加限制的话可能会无限递归下去
- 但是可以明确更优的解出现在更浅的层数上

分析

- 从小到大递归层数限定maxd，只考虑拆分个数不多于maxd的解

- 搜索到的分数都要大于(a/b-x)/(maxd-i)，否则返回
- 其中x为目前已经累积的和，maxd-i为递归剩余的层数

分析

- 比如19/45这个分数，假设当前已经得到

19/45 = 1/5 + 1/100 + ……

- 那么下一个能搜索到的最大分数为1/101，而这意味着(19/45-1/5-1/100)/101，需要23个1/101才可以凑齐19/45，还需要扩展21层才可能搜出所有的解

- 所以如果此时的maxd<23，就可以直接返回

分析

- 此外，还可以加上其他的小剪枝，比如最后一个分数可以直接通过做减法得到（降维的思想），这也都是常用的技巧

最大团问题

- 一个无向图 G=(V,E)，V 是点集，E 是边集。取 V 的一个子集 U，若对于 U 中任意两个点 u 和 v，有边 (u,v)∈E，那么称 U 是 G 的一个完全子图。 U 是一个团当且仅当 U 不被包含在一个更大的完全子图中。
- G的最大团指的是节点数最多的一个团

Sample Input	Sample Output
4 5 1 2 1 3 1 4 2 3 2 4	3 //存在1-2-3/1-2-4两个团

分析

- 先看看裸的DFS会怎么做

1. 从一个点u开始，放入集合S，遍历与之相连的所有节点放入集合S1
2. 从S1中选择一个点v1，这个点肯定是团中的点，遍历与之相连的所有节点放入集合S2
3. 依次类推，当某个集合S为空，则结束当前的DFS，找到一个完全子图，更新最大团，返回上层DFS，直到找完所有完全子图

分析

- 这样做的复杂度当然是不能接受的
- 如果我们在DFS的过程中，确保集合S中的点是按编号有序的，并且只考虑比当前点 i 编号大的点所能构成的完全子图的话，虽然在效率上并没有什么变化，但是给剪枝提供了方便

分析

- 设 $f(i)$表示只考虑编号 $\geqslant i$ 的点所能构成的最大团的点数

- 基本思路:
  - 从 n 到 1 依次搜索，假设搜索到 i 号点，可以强制选择 i 号点作为团中的点，然后从小到大依次枚举 i 号点所连向的编号大于 i 的点，选为团中的点
  - 然后不断进行这样的操作，注意要保证当前选的点要和之前选的点都有连边

分析

- 最优性剪枝 1：
  - 注意 f(i) ≤ f(i+1) + 1，因为只添加一个点，不可能把最大团的点数加2。所以如果把 f(i) 更新为了 f(i+1) + 1，那么就可以直接退出

分析

- 最优性剪枝2:
  - 如果当前深度是 depth，最后一个枚举的点是 u，设 D(u) 为编号大于 u 的且与u 有连边的点的个数，那么如果 depth + D(u) ⩽ f(i)，那么这之后也肯定不可能会有更优的解，直接退出

分析

- 最优性剪枝3：
  - 假设当前深度是 depth，最后一个枚举的点是 u，如果 depth + f(u+1) ⩽ f(i)，那么这之后肯定不可能会有更优的解，直接退出

参考代码

```
bool dfs(int sum, int depth)
{
    if (depth > ans)  //最优性剪枝1
    {
        ans = depth;
        return 1;
    }
    for (int i = 1; i <= sum; ++ i)
    {
        if (depth + sum - i + 1 <= ans)   //最优性剪枝2
            return 0;
        int u = set[depth][i];
        if (depth + f[u] <= ans)    //最优性剪枝3
            return 0;
        int num = 0;
        for (int j = i + 1; j <= ans; ++ j)
            if (map[u][set[depth][j]])
                set[depth + 1][++ sum] = set[depth][j];
        if (dfs(num, depth + 1)) return 1;
    }
    return 0;
}
```

棋盘（普及P3难度）

有一个m × m的棋盘，棋盘上每一个格子可能是红色、黄色或没有任何颜色的。你现在要从棋盘的最左上角走到棋盘的最右下角。

任何一个时刻，你所站在的位置必须是有颜色的（不能是无色的），你只能向上、下、左、右四个方向前进。当你从一个格子走向另一个格子时，如果两个格子的颜色相同，那你不需要花费金币；如果不同，则你需要花费 1 个金币。

另外，你可以花费 2 个金币施展魔法让下一个无色格子暂时变为你指定的颜色。但这个魔法不能连续使用，而且这个魔法的持续时间很短，也就是说，如果你使用了这个魔法，走到了这个暂时有颜色的格子上，你就不能继续使用魔法；只有当你离开这个位置，走到一个本来就有颜色的格子上的时候，你才能继续使用这个魔法，而当你离开了这个位置（施展魔法使得变为有颜色的格子）时，这个格子恢复为无色。

现在你要从棋盘的最左上角，走到棋盘的最右下角，求花费的最少金币是多少？

$1 \leqslant m \leqslant 100$， $1 \leqslant n \leqslant 1{,}000$

分析

Sample Input	Sample Output	Sample Input	Sample Output
5 7 1 1 0 1 2 0 2 2 1 3 3 1 3 4 0 4 4 1 5 5 0	8	5 5 1 1 0 1 2 0 2 2 1 3 3 1 5 5 0	-1

分析

- 要花费最小，变色时应变为与上一个位置同色
- 三种不同的颜色（红色/黄色/无色），可以用数组chess[ ][ ]存为0/1/2
- 相比普通的棋盘类搜索，多了一个魔法值需要考虑，可以在DFS的时候多记录一个magic，表示是否可以使用魔法

参考代码

```
void dfs(int x, int y, int sum, bool magic)
{



 }

for (int i = 1; i <= n; i ++)
{
    scanf("%d%d%d", &x, &y, &color);
    chess[x][y] = color + 1;    //0代表无色，1代表红色，2代表黄色
}
```

分析

- 直接这样DFS，会被卡50分
- 所以我们有必要记录每走到一个格子所累积的金币花费，一旦再次走到这个格子发现花费更多，就剪枝
- 这种在搜索的时候添加一个同步的数组记录搜索过程的做法，称为记忆化

参考代码

```
if (sum >= f[x][y]) return;
//记忆化剪枝，f[x][y]记录走到(x,y)的最小花费
f[x][y] = sum;
```

参考代码

```
for (int i = 0; i < 4; i ++)  // 4个方向
{
    int tx = x + next[i][0];
    int ty = y + next[i][1];  //走到的下一步位置
    if (tx < 1 || tx > m || ty < 1 || ty > m) continue;   //越界
    if (chess[tx][ty] == 0)
    {
        if (!magic) continue;
        chess[tx][ty] = chess[x][y];  //把走到的下一个格子变为当前格子颜色
        dfs(tx, ty, sum + 2, false);  //不能再用魔法
        chess[tx][ty] = 0;  //释放标记回溯
    }
    else if (chess[tx][ty] != chess[x][y]) dfs(tx, ty, sum + 1, true);
    else if (chess[tx][ty] == chess[x][y]) dfs(tx, ty, sum, true);
}
```

奶酪（提高P1难度）

现有一块大奶酪，它的高度为 h，它的长度和宽度我们可以认为是无限大的，奶酪中间有许多***半径相同***的球形空洞。我们可以在这块奶酪中建立空间坐标系，在坐标系中，奶酪的下表面为$z = 0$，奶酪的上表面为$z = h$。

现在，奶酪的下表面有一只小老鼠 Jerry，它知道奶酪中所有空洞的球心所在的坐标。如果两个空洞相切或是相交，则 Jerry 可以从其中一个空洞跑到另一个空洞，特别地，如果一个空洞与下表面相切或是相交，Jerry 则可以从奶酪下表面跑进空洞；如果一个空洞与上表面相切或是相交，Jerry 则可以从空洞跑到奶酪上表面。

位于奶酪下表面的 Jerry 想知道，在***不破坏奶酪***的情况下，能否利用已有的空洞跑到奶酪的上表面去?

空间内两点$P_1(x_1, y_1, z_1)$、$P_2(x_2, y_2, z_2)$的距离公式如下：

$$\mathrm{dist}(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2 + (z_1 - z_2)^2}$$

$1 \leqslant n \leqslant 1{,}000$

分析

- 两圆相切或相交：圆心距离 $\leqslant 2 * r$
- 首先要找到可以从下表面进入的圆：某圆心高度 $z - r \leqslant 0$
- 然后DFS搜索
- 一旦发现一个点的圆心高度$z + r \geqslant h$，搜索即可结束
- 只有当答案为“no”的时候才需要遍历整个搜索树，所以实际效率比想象的要高
- 浮点数用double，h，r 要开long long

参考代码

```
for (int i = 1; i <= n; i ++)
    scanf("%lld%lld%lld", &x[i], &y[i], &z[i]);

void dfs(int i)
{
    if (z[i] + r >= h) {fnd = 1; return;}
    flag[i] = 1;
    for (int j = 1; j <= n; j ++)
        if (!flag[j] && dis(i, j) <= 2 * r) dfs(j);
 }
```

参考代码

```
for (int i = 1; i <= n; i ++)
{
    if (!flag[i] && z[i] - r <= 0) dfs(i);
    if (fnd) {printf("Yes\n"); break;}
}
if (!fnd) {printf("No\n"); fnd = 0;}
```

参考代码

- 也可以开结构体，不过 dfs 就需要额外记录节点的编号

```
struct node
{
    long long x, y, z;
} p[N];

void dfs(node i, int num)
{
    if (i.z + r >= h) {fnd = 1; return;}
    flag[num] = 1;
    for (int j = 1; j <= n; j ++)
        if (!flag[j] && dis(i, p[j]) <= 2 * r) dfs(p[j], j);
}
```

信息传递（提高P2难度）

有 n 个同学（编号为 1 到 n）正在玩一个信息传递的游戏。在游戏里每人都有一个固定的信息传递对象，其中，编号为 i 的同学的信息传递对象是编号为T_i的同学。

游戏开始时，每人都只知道自己的生日。之后每一轮中，所有人会同时将自己当前所知的生日信息告诉各自的信息传递对象（注意：可能有人可以从若干人那里获取信息，但是每人只会把信息告诉一个人，即自己的信息传递对象）。当有人从别人口中得知自己的生日时，游戏结束。请问该游戏一共可以进行几轮？

$n \leq 200000$

分析

- 把每个同学看成一个节点，把一次信息传递看成是连边，一个人要想从别人口中得知自己的生日，那必然是构成一个环
- 所以这道题实际就是求所有这些环中，最小环的长度
- 可是也有可能不构成环的节点和连边存在，因为有多对一的关系，那就一层一层去掉那些入度为0的点，直到只剩下环为止

分析

- 那么我们读入的时候，就需要记录点的入度
- 然后删除那些入度为0的点，并且在删除该点后，入度—

- 注意这个删除有可能不止一个点，所以要打标记

参考代码

```
for (int i = 1; i <= n; i ++)
    scanf("%d", &a[i]), de[a[i]] ++;  //读入时记录节点的入度
while (1)  //删掉入度为 0的点
{
    int flag = 1;    //打删除标记
    for (int i = 1; i <= n; i ++)
        if (de[i] == 0 && !vis[i]){   //找到入度为 0的点
            flag = 0;
            vis[i] = 1;  //标记已访问
            de[a[i]] --;  //与之相连的点入度减 1
        }
    if (flag) break;
 }
```

分析

- 访问到之前已访问过的点，就 return。因为每个点只有一个出度

参考代码

```
void dfs(int x, int y)
{
    if (x == y) return;   //访问到之前已访问过的点
    if (!vis[y]) vis[y] = 1;
    if (!vis[x]){
        vis[x] = 1;
        len ++;    // 环长度计数
        dfs(a[x], y);     //访问下一个相连的节点
    }
}
```

推荐题单I

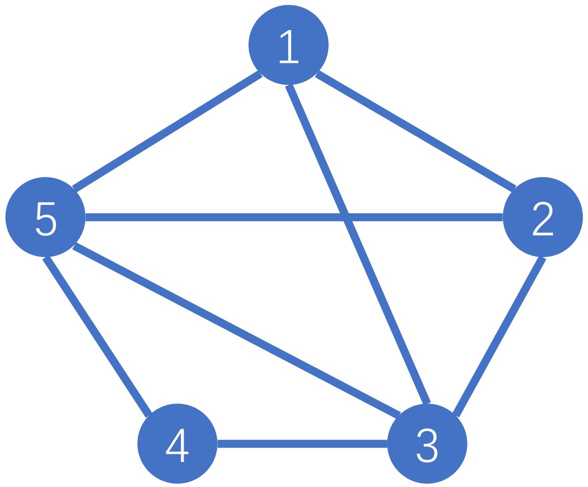
- 1706 全排列问题
- 1036 选数
- 1019 单词接龙
- 1219 八皇后
- 2402 自然数的拆分问题
- 1605 迷宫
- 1101 单词方阵
- 1120 小木棍
- 1731 生日蛋糕
- 1691 有重复元素的排列问题
- 1025 数的划分
- 1141 01迷宫
- 1092 虫食算
- 1236 算24点

广搜

- 广搜体现了明显的层次性，是很多图论算法的原型

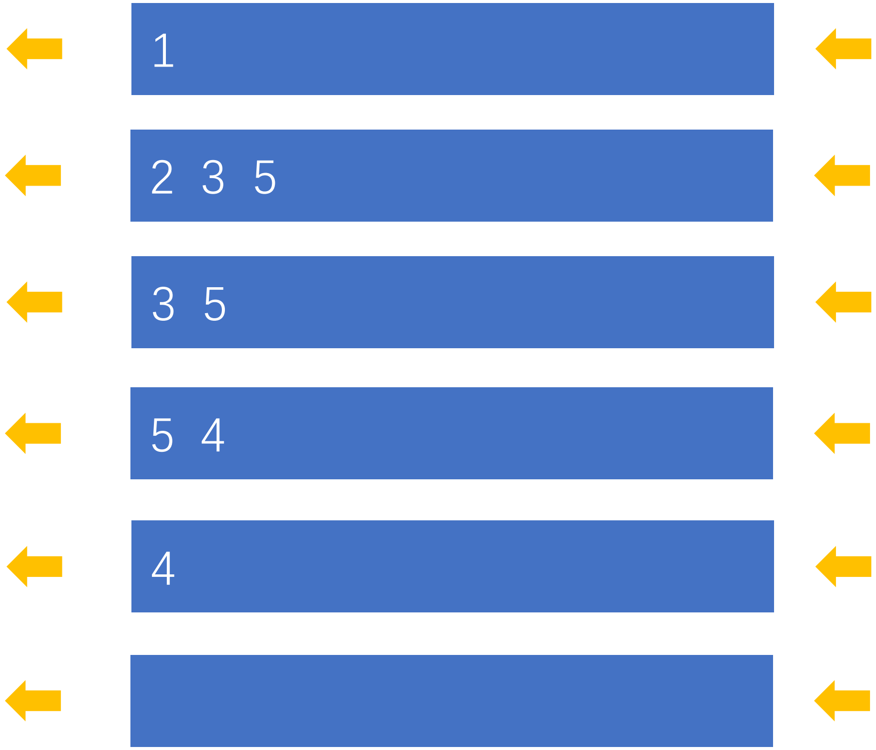
广搜

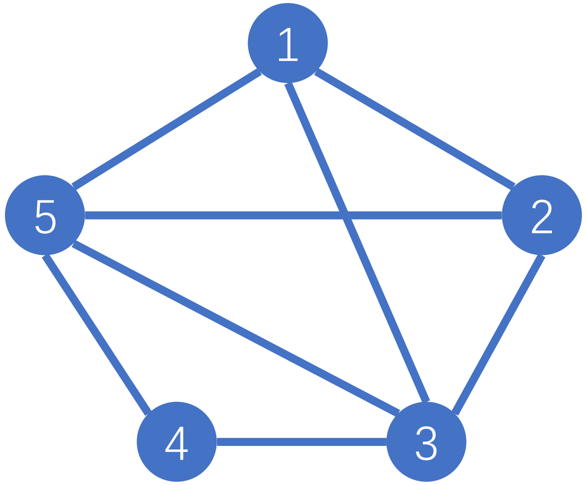
- 广搜不存在回溯的问题，所以不需要递归。但是需要记录之前访问的先后顺序
- 比如1号顶点被访问，然后访问与之相连的2、3、5号顶点，这个访问顺序就需要被记住，因为接下来需要访问所有与2号顶点相连的点（没有，因为1、3、5都已被标记访问），接下来是3号顶点（4号），5号顶点（没有）



队列

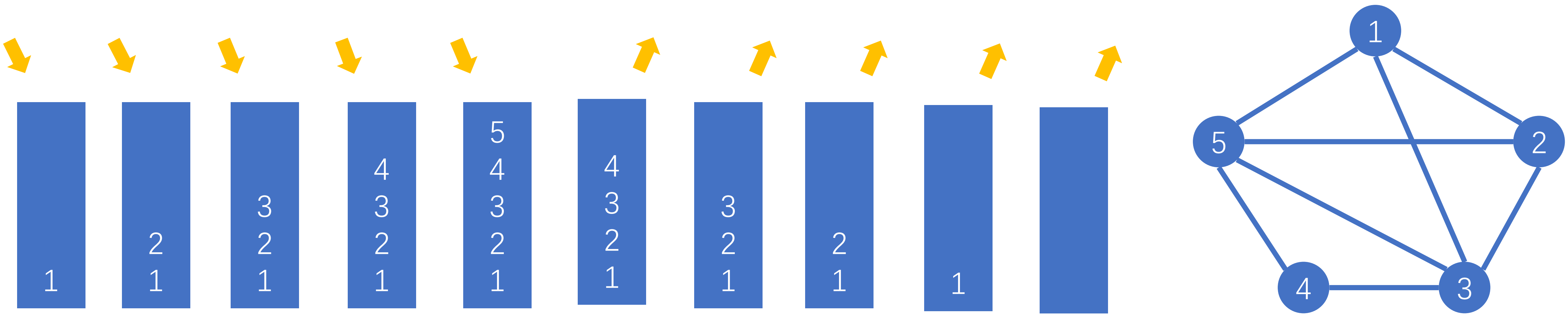
- 记录之前访问的顺序，我们用“队列”来完成





栈

- 而在DFS中，不需要记录访问顺序，但是需要逐层深入/回溯，这一过程是用“栈”来完成的



队列

- “栈”无需自己写，递归过程中会自动开系统栈
- “队列”需要自己写，一般有两种方式：

1. 用数组模拟实现队列
2. 用STL中自带的queue（常数比priority_queue大很多，慎用）

数组模拟队列

- 定义 `int que[1000], L = 1, R = 0;`
- 队列不空 `while (L <= R)`
- 新元素入队 `que[++R] = i;`
- 取队首元素并出队 `int cur = que[L++];`

queue

- 定义 queue<int> q;
- 队列不空 while (!q.empty())
- 新元素入队 q.push(m)
- 取队首元素 q.front();
- 队首元素出队 q.pop();

BFS的一般模型

```
void bfs()
{
    while (队列不空)
    {
        取队首元素;
        if (满足条件)
            输出方案;
        else 该点符合条件的兄弟节点入队
    }
 }
```

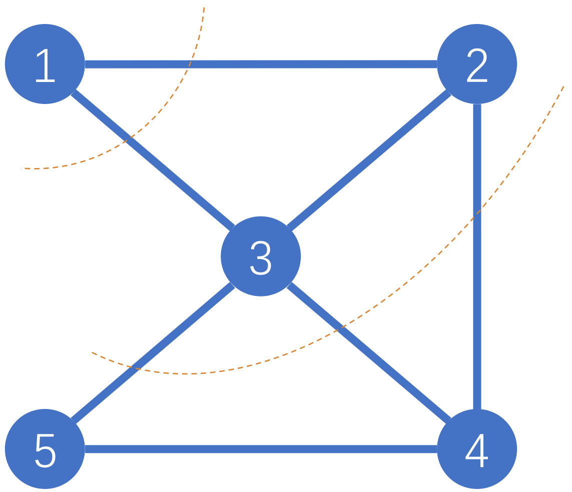
图的BFS遍历

```c
void bfs(int step)
{
    int L = 1, R = 0;
    que[++R] = step, flag[step] = 1;
    while (L <= R)
    {
        int cur = que[L++];
        printf("%d ", cur);
        for (int i = 1; i <= n; i ++)
            if (e[cur][i] == 1 && flag[i] == 0)
                que[++R] = i, flag[i] = 1;
    }
 }
```

图的BFS遍历

- 从图中找出从1号点到5号点的最少换乘方案

1. BFS逐层拓展
2. 直到目标点进入队列



- 为什么目标点进入队列，就一定是最短的？
- 边权为1的最短路，可以直接用BFS做

BFS中的状态存储

- 在 BFS 中，通常需要对搜索到的状态进行存储和标记，所以我们需要有一定的状态存储的技巧

图的状态存储

- 关于图的状态:
  - 一般记录当前位置，或者加上步数

棋盘类的状态存储

- 关于棋盘的状态：
  - 如果棋盘上面的棋子只有 “有”或者“没有” 的区别，那么我们可以把棋盘的状态转成二进制数然后进行存储
  - 拓展一点，如果棋子还有颜色，那就可以转成与颜色数相关的进制数来存储

排列类的状态存储

- 关于排列的状态：
  - 有些问题的状态是数的排列，虽然我们可以把排列直观地转成 n 进制数来存储，但是空间利用率较低，所以我们可以用康托展开来有效地存储关于排列的状态

康托展开

- 康托展开表示的是当前排列在n个不同元素的全排列中的名次。比如213在这3个数所有排列中排第3

- 康托展开的公式

$$X = a_1*(n-1)! + a_2*(n-2)! + ... + a_{n-1}*1! + a_n*0!$$

a_i表示i~n中有几个元素比它小

康托展开

- 比如3241，想要知道它在1/2/3/4所有的排列中编号是多少

- 可知a_1=2（1/2比3小），a_2=1（1比2小），a_3=1（1比4小），a_4=0
- 代入计算得 X = 15

- 一个排列唯一对应着一个 X，一个 X 也唯一对应着一个排列，所以我们可以用康托展开来存储排列的状态

参考代码

```
LL cantor(char str[])
{
    int len = strlen(str);
    LL ans = 0;
    for (int i = 0; i < len; i ++)
    {
        int tmp = 0;
        for (int j = i + 1; j < len; j ++)
            if (str[j] < str[i]) tmp ++;
        ans += tmp * f[len - i - 1];   //f[]为阶乘
    }
    return ans;
}
```

康托逆展开

- 知道 X 又怎么求排列?
- 首先 a_1 = X / (n-1)! ， a_2 = (X - a_1*(n-1)!) / (n-2)! ， 依此类推可求出所有的 a_i
- 然后可以求出 A_1，知道 A_1 后就可以求出 A_2，依此类推可求出所有 A_i

康托逆展开

- 比如我们借助前页的公式已经求出：$a_1 = 2, a_2 = 1, a_3 = 1, a_4 = 0$

- 首先可以得到 $A_1=3$，然后 $A_2=2$
- 现在考虑 A_3，A_3 应该是比 1 大，并且没有在 A_1 和 A_2 中出现过的最小的数，就是 4 ，最后 $A_4=1$
- 所以我们可以求得原排列为：3 2 4 1

四子连棋

- 在一个4*4的棋盘上摆放了14颗棋子，其中有7颗白色棋子，7颗黑色棋子，有两个空白地带，任何一颗黑白棋子都可以向上下左右四个方向移动到相邻的空格，这叫行棋一步，黑白双方交替走棋，任意一方可以先走，如果某个时刻使得任意一种颜色的棋子形成四子一线（包括斜线），这样的状态为目标棋局，给出初始状态，求最少移动步数

●	○	●	
○	●	○	●
●	○	●	○
○	●	○	

分析

- 我们可以用 0/1/2 来分别表示没有棋子，有白色棋子，以及有黑色棋子，那么状态数就有 3^16 种，同时我们还要记录当前行棋的玩家是谁，所以总状态数还要乘以 2，总共状态总数就是 2×3^16 = 86093442 种

八数码

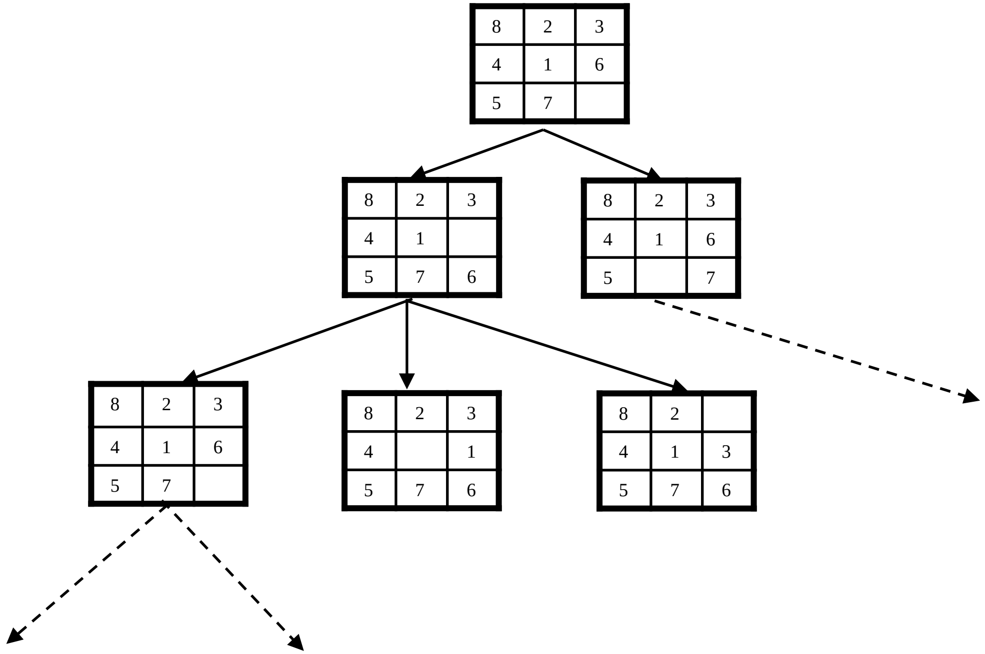
- 有一个3*3的棋盘，其中有0~8共9个数字，0表示空格，其他相邻的数字可以和0交换位置。求由初始状态到达目标状态所需的最少步数

8	2	3
1	4	6
5	7	

1	2	3
4	5	6
7	8	

分析

- 搜索状态



分析

- 结点是一个3*3的格子，需要存储状态，可以将每种状态对应成一个9位数字，比如初始状态可以表示为823146570

分析

- 在这样的表示法下，原有的横向纵向移动也改为：

1. 空格向上移动：空格的位置减3，即交换X和X-3的字符
2. 空格向左移动：空格的位置减1，即交换X和X-1的字符
3. 空格向右移动：空格的位置加1，即交换X和X+1的字符
4. 空格向下移动：空格的位置加3，即交换X和X+3的字符

- 上述四条规则可归纳为一条：交换X和X+(2*k-5)的字符，其中k（1~4）为方向

判重

- 在刚才的例题中，9位数字可以表示，如果超出呢?
- 超出则以二进制表示，不过数字很大以后记得取模
- 判重需要一个标志位序列，即每个状态对应标志位序列中的一位

判重

- 状态的编码方式，决定了标志位序列的空间复杂度

- 比如刚才的例题，既可以直接用一个9进制数来表示状态，也可以把一个状态看成一个排列，而以这个排列在全部排列中的位置作为状态编号
- 当然无论哪种，都需要写对应的转换函数

哈希判重

- 比如有如右棋盘:

1	0	1	0
0	1	1	1
0	0	1	0
1	1	0	0

- 可以表示为（忽略空格）：1010 0111 0010 1100
- 第 i 行 j 列单元格的转换公式为：$4*(i-1)+j$
- 这个二进制数串可以和大小为 $2^{16}-1$ 的布尔数组一一对应起来
- 于是我们可以在 O(1) 的时间里进行状态检查

哈希判重

- 这个过程即为哈希判重
- 那个大小为2^{16}-1的布尔数组即为哈希表/散列表
- 对应关系（这里就是直接对应）称为哈希函数/散列函数

哈希判重

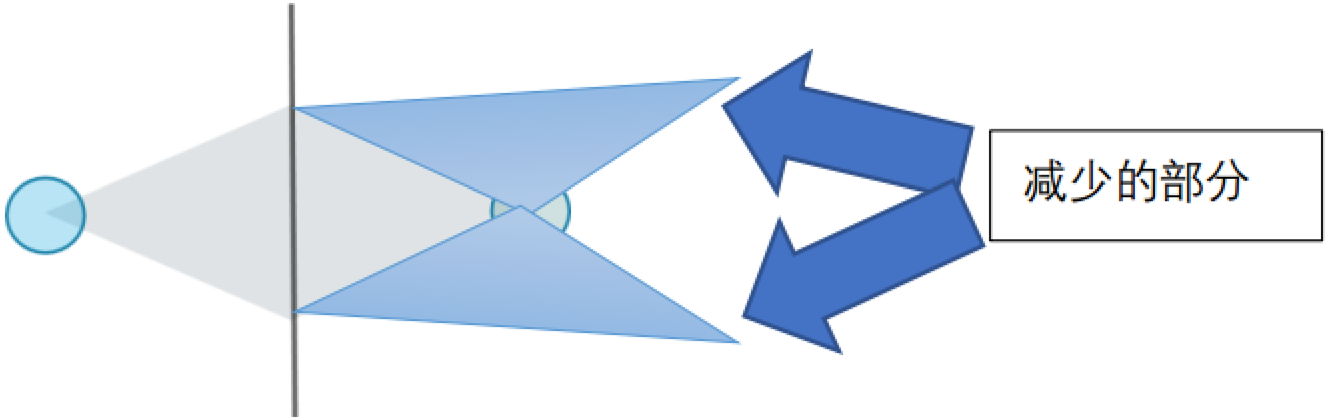
- 哈希判重对于BFS的意义：
  - 在BFS当中，扩张出来的新的节点总是有很大的重复，必须对这些节点进行判重操作
  - 最简单的判重方法是相当耗时的，它需要将新的节点与已经生成的节点逐一进行比较
  - 在搜索的初期阶段，这一步的耗时还不明显，但随着节点数目增大，判重的耗时就大大增加到几乎无法承受的地步，甚至是时间复杂度的主要开销来源

双向BFS

- 顾名思义，从起点状态和终点状态同时BFS，然后借助于判重
- 判重数组的值可以多设一组：被逆序访问过。那么一旦某种状态顺序逆序都被访问过，显然就是连接答案的状态

双向BFS

- 最差情况优化50%



- 双向BFS适用的题目：扩展节点较多，目标节点深度较大

双向BFS

- 双向BFS的进一步优化

1. 每次 while 循环时只扩展正反两个方向中节点数目较少的一个，可以使两边的发展速度保持一定的平衡，从而减少总扩展节点的个数，加快搜索速度

2. 用堆代替队列来维护状态，也是BFS中的常见优化

方程的解数

- 已知一个n元高次方程：

$$k_1x_1^{p1}+k_2x_2^{p2}+\cdots\cdots+k_nx_n^{pn} = 0$$

- 其中：$x_1, x_2, \cdots, x_n$是未知数，$k_1,k_2,\cdots,k_n$是系数，p1,p2,…pn是指数。且方程中的所有数均为整数
- 假设未知数$1 \leqslant x_i \leqslant M$，求这个方程的整数解的个数

$1 \leqslant n \leqslant 6$；$1 \leqslant M \leqslant 150$

$|k_1M^{p1}|+|k_2M^{p2}|+\cdots\cdots+|k_nM^{pn}|< 2^{31}$

方程的整数解的个数小于2^{31}

分析

- 暴力的复杂度O(nm)
- 我们转化一下那个方程（假设n=6）

$$k_1x_1^{p1}+k_2x_2^{p2}+k_3x_3^{p3} = -(k_4x_4^{p4}+k_5x_5^{p5}+k_6x_6^{p6})$$

- 只要将左边的取值和右边的取值分别求出来，再判断一下相等的对数即可，时间复杂度$O(m^3)$

对比

- BFS自带效果是会优先搜索“距离最近”的解

- 而有些题中，起点到终点的距离是固定的，比如全排列问题，这时候DFS更直观而BFS没有优势

对比

- BFS是同一层节点一起进行
- DFS同一时间只进行当前一种方案，所以状态数很多时，BFS需要对所有节点都开检查数组

两种搜索方式的比较

	深度优先搜索	广度优先搜索
搜索方式	下一个搜出的节点，是当前节点的儿子节点	下一个搜索的节点，是当前节点的兄弟节点
所用数据结构	（系统）栈	队列
常见优化	剪枝、迭代加深	Hash判重、双向广搜、A*搜索
适用范围	只需要求解一种可行方案	求最优解的题目

推荐题单II

- 1379 八数码
- 1162 填涂颜色
- 1032 字串变换
- 1126 机器人搬重物
- 1443 马的遍历
- 1213 时钟
- 1434 滑雪
- 1225 黑白棋游戏
- 1278 单词游戏