DFS

剪枝

- 优化搜索顺序

  大部分情况下，我们应该优先搜索分支少的方向
- 排除等效冗余
- 可行性剪枝
- 最优性剪枝
- 记忆化搜索

迭代加深

- 在搜索树中，如果某些分支的搜索顺序深度很深，而答案出现在较浅的层数中。
- 设定一个depth (搜索的深度)，当搜不到答案时，把depth++；

  加成序列

```
#include<iostream>
#include<cstring>
using namespace std;

const int N = 110;

int n,path[N];

bool dfs(int u,int depth)
{
    if(u > depth)   return false;
    if(path[u - 1] == n)    return true;

    bool st[N] = {0};//来记录当前和是否已经枚举过
    for(int i = u - 1; i >= 0; i--)//优化搜索顺序
        for(int j = i; j >= 0; j--)
        {
            int s = path[i] + path[j];//两个数之和
            if(st[s])   continue; // 去除等效冗余

            if(s > n || s <= path[u - 1])   continue;//最优性剪枝

            st[s] = true;//只为保证当前分支不搜索重复元素，不是状态的一部分，path才是状态且，path一定是不重复的，上面的if筛选过了
            path[u] = s; // path状态
            if(dfs(u + 1,depth))    return true;
            path[u] = 0;
        }

    return false;
}
```

```
int main()
{
    path[0] = 1;
    while(cin >> n, n)
    {
        int depth = 1;

        while(true)
        {
            if(dfs(1,depth))    break;
            depth ++;
        }

        for(int i = 0; i < depth; i++)
            cout << path[i] << ' ';
        puts("");
    }

    return 0;
}
```

双向DFS

- 原理与双向BFS一样，从两头开始往中间搜索，能剪一大部分枝

  但不一定得从终点和起点两个地方开始搜索，也可以从起点搜索一部分，然后再接着搜索，这样也能剪枝

  送礼物

```
#include<iostream>
#include<algorithm>
using namespace std;

const int N = 50;

long long tw,n,w[N],weight[1 << 25],cnt = 1,ans = 0;

void dfs1(int u,int s,int k)
{
    if( u == k )
    {
        weight[cnt ++] = s;
        return;
    }

    dfs1( u + 1, s , k);

    if(s + w[u] <= tw)  dfs1(u + 1,s + w[u], k);
}

void dfs2(int u,int s)
{
    if(u == n)
    {
        int t = tw - s;
```

```
        int  l = 0 , r = cnt - 1;
        while(l < r) //二分找第一个小于等于t的数
        {//也可以用upper_bound，但是upper_bound得到的是第一个大于t的数，所以获取的下标要减去 1 ，并且运行速度比较慢
            int mid = l + r + 1>> 1;
            if(weight[mid] <= t) l = mid;
            else r = mid - 1;//区间[l,r]被划分成[l,mid - 1],[mid,r]
        }

        ans = max(ans,weight[l] + s);
        return;
    }

    dfs2(u + 1, s);
    if(s + w[u] <= tw)  dfs2(u + 1,s + w[u]);
}

int main()
{
    cin >> tw >> n;

    for(int i = 0; i < n; i++)  cin >> w[i];

    sort(w , w + n);
    reverse(w , w + n);

    int k = n/2 + 2;

    dfs1(0,0,k); //搜索一部分，打表

    sort(weight , weight + cnt);
    cnt = unique(weight,weight + cnt) - weight;//去重，减去原始地址获得元素总个数

    dfs2(k ,0);

    cout << ans << endl;

    return 0;
}
```

IDA*

- **IDA* 与 A* 类似，IDA* 通过启发函数来进行剪枝，且启发函数返回的值加上当前真实值也一定小于等于(<=)总真实值**
- **IDA* 通常与迭代加深一起使用，增强剪枝效果**

  回转游戏

```
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;

int opt[8][7] =
{
```

```
    {0,2,6,11,15,20,22},
    {1,3,8,12,17,21,23},
    {10,9,8,7,6,5,4},
    {19,18,17,16,15,14,13},
    {23,21,17,12,8,3,1},
    {22,20,15,11,6,2,0},
    {13,14,15,16,17,18,19},
    {4,5,6,7,8,9,10}
};

int opposite[8] = {5,4,7,6,1,0,3,2},center[8] = {6, 7, 8, 11, 12, 15, 16,
17};

int mp[24],sum[4],num;
char path[100];

int f() // 启发函数：回字形最多的数，与 8 的差，一定小于等于真实值
{
    int s = 0;
    memset(sum,0,sizeof sum);
    for(int i = 0; i < 8; i++)  sum[mp[center[i]]]++;

    for(int i = 1; i < 4; i++)  s = max(s,sum[i]);

    return 8 - s;
}

void operate(int u)
{
    int v = mp[opt[u][0]];
    for(int i = 0; i < 6; i++)  mp[opt[u][i]] = mp[opt[u][i + 1]];
    mp[opt[u][6]] = v;
}

bool dfs(int cnt,int depth,int last) //深搜
{
    int v = f();
    if(!v)  return true;
    if(cnt + v > depth) return false;

    for(int i = 0; i < 8; i++)
    {
        if(opposite[i] == last) continue;//反向操作等于没有操作过，直接剪掉。。。
        operate(i);
        path[cnt] = char(i + 'A');
        if(dfs(cnt + 1,depth,i))    return true;
        operate(opposite[i]);
    }

    return false;
}

int main()
{
    while( cin >> mp[0] , mp[0])
    {
        for(int i = 1; i < 24; i++) cin >> mp[i];
```

```
        int depth = 0;
        while(!dfs(0,depth,-1)) depth++;

        if(!depth)  printf("No moves needed");
        else
        {
            for(int i = 0; i < depth; i++)  printf("%c",path[i]);
        }
        printf("\n%d\n",mp[6]);
    }

    return 0;
}
```

排书

```
#include<iostream>
#include<cstring>
#include<algorithm>
using namespace std;

const int N = 15;

int seq[N],n,backup[6][N];

int f()
{
    int res = 0;
    for(int i = 0; i + 1 < n; i++)
        if(seq[i + 1] != seq[i] + 1)    res ++;//计算非法后继
    //向上取整
    return (res + 2)/3; // res / 3 +1 也行，只要小于等于真实距离即可 每插入一次序列，改变三个后继
}

bool check()
{
    for(int i = 0; i + 1 < n; i++)
        if(seq[i] > seq[i + 1]) return false;

    return true;
}

bool dfs(int depth,int max)
{
    int v = f();
    if(check()) return true;
    if(depth + v > max) return false; //如果估算的步数大于最大能走，则这个分支无解

    for(int len = 1; len < n; len ++) //枚举长度
        for(int l = 0; l + len - 1 < n; l++) //枚举左端点
        {
            int r = l + len -1; //右端点
            for(int k = r + 1; k < n; k++)//枚举插入的位置
            {
                memcpy(backup[depth],seq,sizeof seq);
                int x = l; // 修改序列操作
```

```
                for(int i = r + 1; i <= k; i++,x++) seq[x] = backup[depth]
[i];
                for(int i = l; i <= r ; i++,x++)    seq[x] = backup[depth]
[i];
                if(dfs(depth + 1,max))  return true;
                memcpy(seq,backup[depth],sizeof seq);//恢复现场
            }
        }
    return false;
}

int main()
{
    int T;
    for( cin >> T; T; T--)
    {
        cin >> n;
        for(int i = 0; i < n; i++)  cin >> seq[i];

        int depth = 0;
        while(depth <= 5 && !dfs(0,depth))  depth++;//迭代加深

        if(depth >= 5)  puts("5 or more");
        else    cout << depth << endl;
    }

    return 0;
}
```