法律声明

- 本课件包括演示文稿、示例、代码、题库、视频和声音等内容，小象学院和主讲老师拥有完全知识产权的权利；只限于善意学习者在本课程使用，不得在课程范围外向任何第三方散播。任何其他人或机构不得盗版、复制、仿造其中的创意及内容，我们保留一切通过法律手段追究违反者的权利。

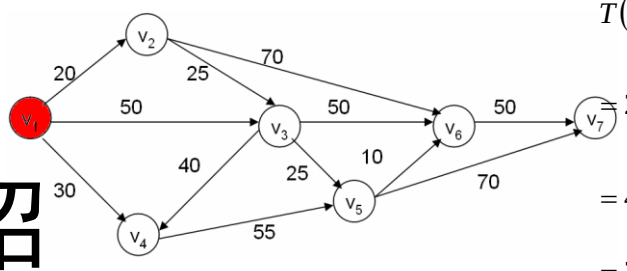
- 课程详情请咨询
  - 微信公众号：小象
  - 新浪微博：ChinaHadoop

线性表

——链表、队列、堆栈

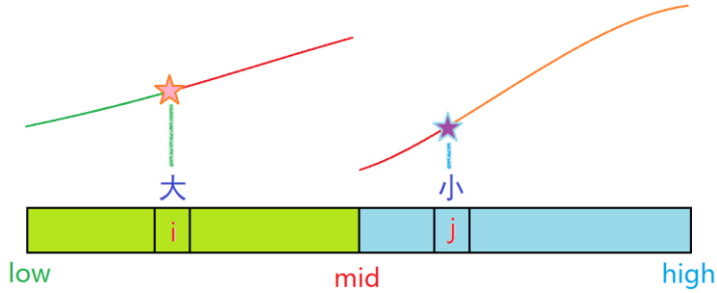
邹博

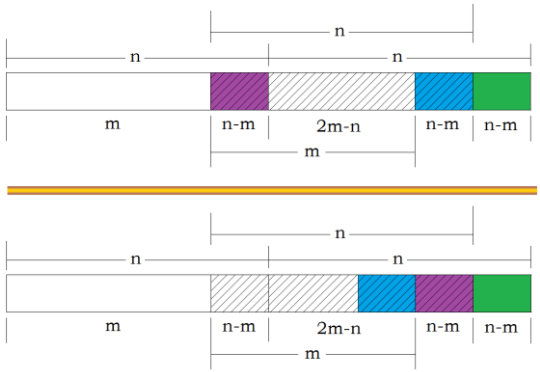
本课程介绍



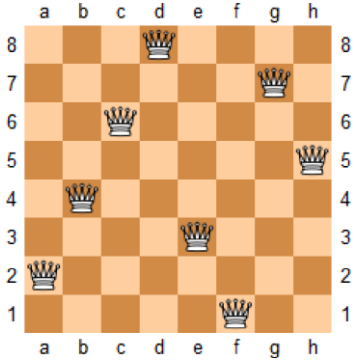
$$T(n)=2\cdot T\left(\frac{n}{2}\right)+c\cdot n$$
$$=2\cdot\left(2\cdot T\left(\frac{n}{4}\right)+c\cdot\frac{n}{2}\right)+c\cdot n=4T\left(\frac{n}{4}\right)+2c\cdot n$$
$$=4\left(2\cdot T\left(\frac{n}{8}\right)+c\cdot\frac{n}{4}\right)+2c\cdot n=8T\left(\frac{n}{8}\right)+3c\cdot n=\cdots\cdots$$
$$=2^k T(1)+kc\cdot n=an+cn\log_2 n$$

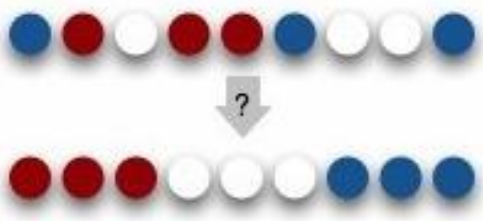
- 线性表
- 递归分治
- 字符串
- 数组
- 树
- 图
- 图实践
- 查找排序
- 动态规划
- 概率组合数论
- 海量数据处理





	细君	昭君	探春	文成
站军姿 孙武	8小时	6小时	2小时	4小时
踢正步 王翦	3小时	1小时	3小时	12小时





主要内容

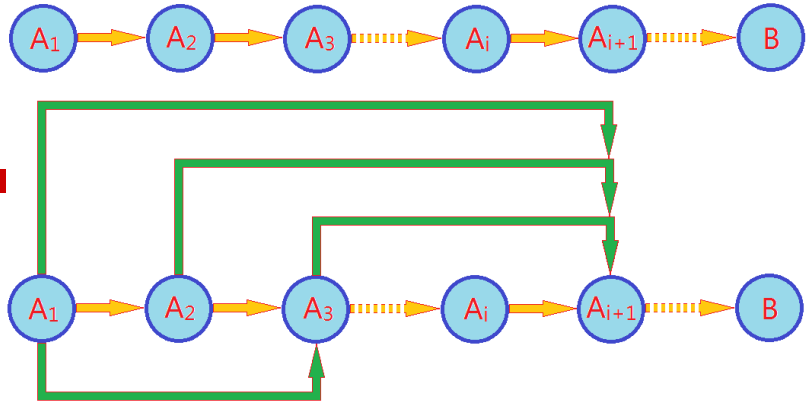
- 算法总论
  - 算法是有用的
- 链表
  - 链表相加/链表部分翻转/链表去重
  - 链表划分/链表公共结点
- 队列
  - 拓扑排序
  - 最短路径条数
- 堆栈
  - 括号是否匹配
  - 最长括号匹配
  - 计算逆波兰表达式
  - 出栈入栈问题

算法口诀

难题首选动归，
受阻贪心暴力；
考虑分治思想，
配合排序哈希。

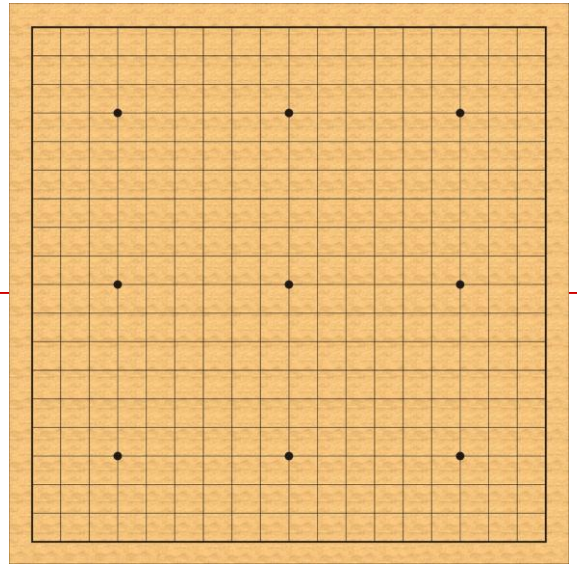
- 动态规划是解决相当数目问题的法宝；
  - 滚动数组降低空间复杂度
- 贪心法并不简单
  - Dijkstra最短路径、最小生成树Prim、Kruskal算法
- 深度优先搜索、广度优先搜索，都可以归结为暴力求解；
  - 分支限界条件加快搜索效率
- 分治法在降低问题规模问题上很有效；
  - 快速排序、归并排序——递归、广义分治法
- 排序是为了更好的查找；
  - 各种排序方法的选择
- 实在不行了，空间换时间——Hash
  - 深入理解Hash——int a[65536]/int a[256]
- 有些题目需要上述两者或者多个技术综合运用。

总论



- 算法包罗万象
  - 推理、逻辑、“机智”
  - 演绎、归纳、类比
  - 严格归纳
- 算法是脑力的游戏
- 合理运用算法，能够获得更高的效率
  - 时间复杂度优先
  - 空间复杂度优先
  - 时间复杂度和空间复杂度的折中

系统的“数数”



- 围棋棋盘由横纵19*19条线组成，这些线共组成多少个正方形？
- 思路：
  - 边长为1的正方形多少个？边长为2的呢？边长为3、4...的呢？
  - 以某点为右下角的正方形有多少个？把所有点的正方形相加。
- 系统的遍历——不漏不重：
  - 动态规划 Richard Ernest Bellman, 1953
  - 广度优先搜索Edward Forrest Moore 1959, C.Y.Lee 1961
  - 深度优先搜索John E. Hopcropft, Robert Endre Tarjan 1971-1972
  - 子集和数问题：给定N个数和某定值sum，从N个数中取若干个数，要求它们的和是sum，输出所有的取法。

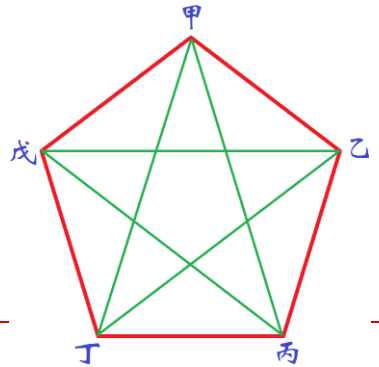
机智：“战平即可出线”

- 足球比赛，一个小组有8支球队进行单循环赛，胜者积3分，平则双方同积1分，负则不积分，规定积分最高的4支球队出线，则出线至少需要_____分，未出线最多可能有_____分。

战况分析

- 出线的4支球队中，第4名的积分是最少的，那么，它至少要多少积分才有可能出线呢？
- 8支球队中，3支强队各自战胜其他5支弱队，而5支弱队之间比赛全部战平。则5支弱队中积分全部是4分，可以采用进球数或抽签选5支弱队中的1支作为第4名出线。

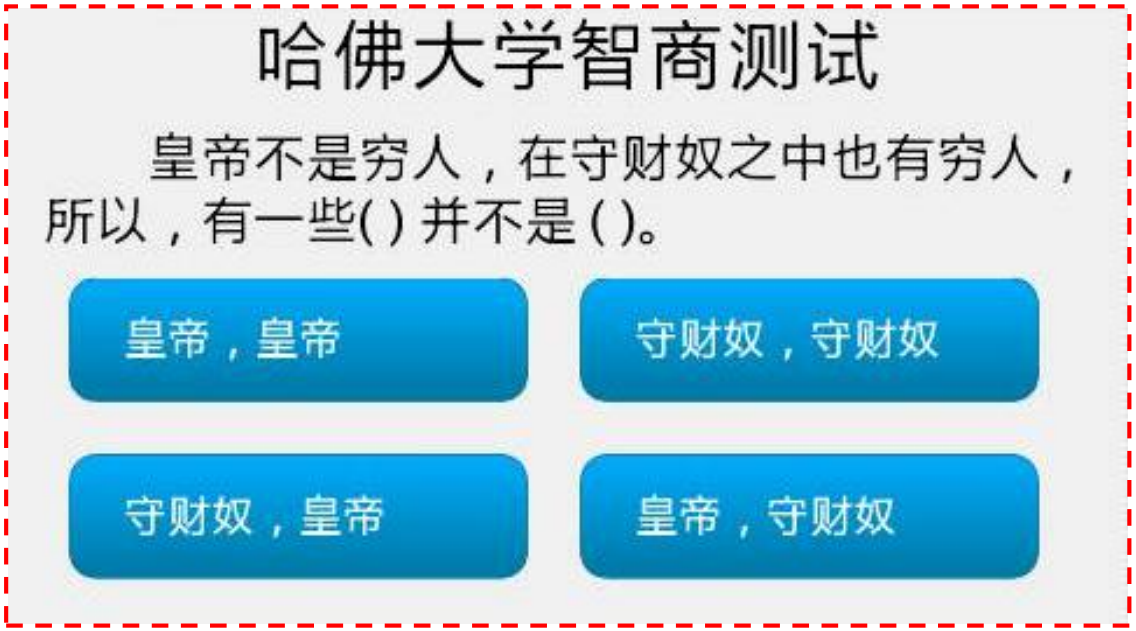
战况分析



- 未出线的4支球队中，第5名的积分最多，它最多可能多少积分却没有出线呢？
- 8支球队中，5支强队假定为甲、乙、丙、丁、戊，他们各自战胜其他3支弱队，同时，5支强队之间的战况为：甲战胜乙丙，乙战胜丙丁，丙战胜丁戊，丁战胜戊甲，戊战胜甲乙，则这5支强队同时胜5场输2场，同积15分。可以采用净胜球、进球数或抽签决定5支强队中的1支作为第5名而被淘汰。

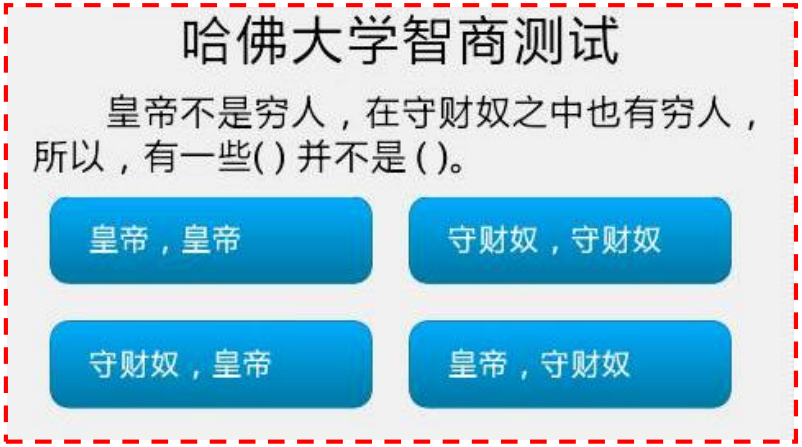
逻辑推理：完形填空

- 皇帝不是穷人，在守财奴之中也有穷人，所以，有一些_____并不是_____。

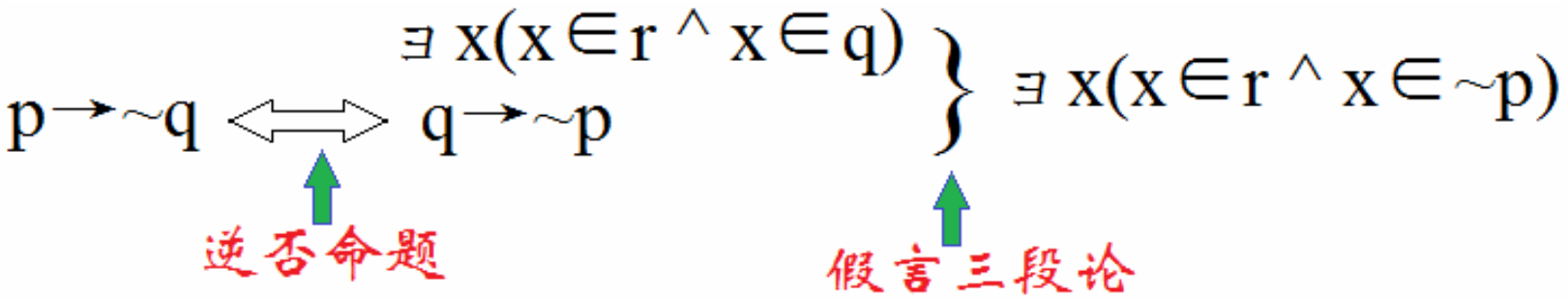


使用离散数学分析该题目

- p：这个人是皇帝
- q：这个人是穷人
- r：这个人是守财奴
- 皇帝不是穷人：$p \rightarrow \sim q$
- 在守财奴之中也有穷人：$\exists x(x \in r \wedge x \in q)$



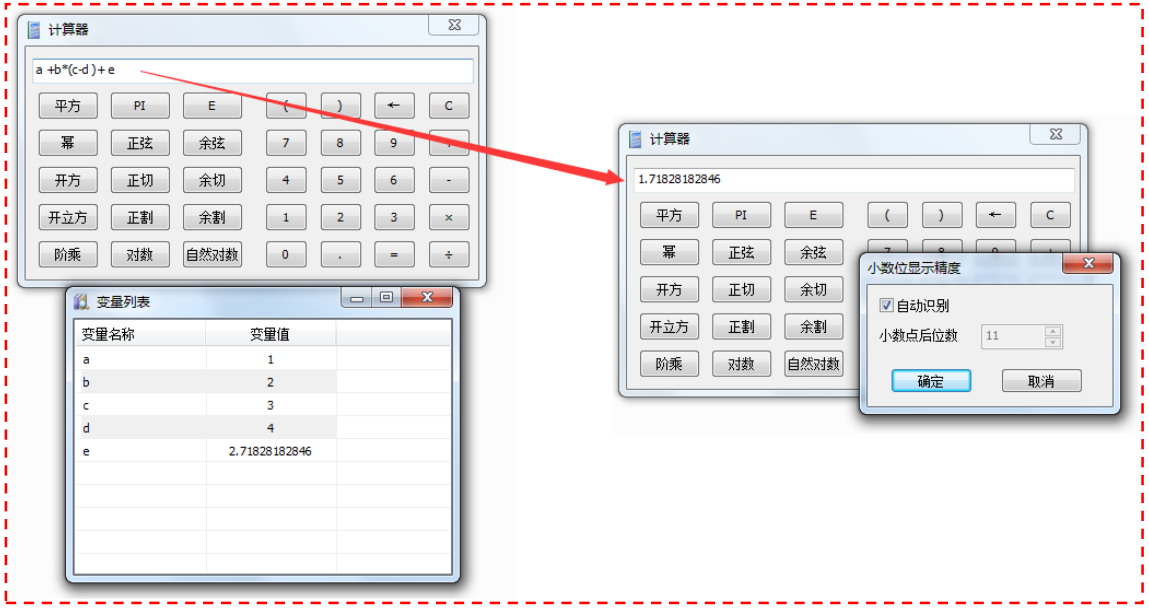
分析过程



- r：这个人是守财奴
- p：这个人是皇帝
- 有一些 守财奴 并不是 皇帝 。

系统：字符串表达式的计算

- a+b*(c-d)+e
- 朴素算法
- 逆波兰表达式
  - 栈的典型应用



链表相加

- 给定两个链表，分别表示两个非负整数。它们的数字逆序存储在链表中，且每个结点只存储一个数字，计算两个数的和，并且返回和的链表头指针。
  - 如：输入：2→4→3、5→6→4，输出：7→0→8

问题分析

- 输入：2→4→3、5→6→4，输出：7→0→8
- 因为两个数都是逆序存储，正好可以从头向后依次相加，完成“两个数的竖式计算”。
- 注意考虑两个数的位数不相同的情况。

Code

```
typedef struct tagSNode
{
    int value;
    tagSNode* pNext;

    tagSNode(int v):value(v), pNext(NULL){}
}SNode;
```

```
int _tmain(int argc, _TCHAR* argv[])
{
    SNode* pHead1 = new SNode(0);
    int i;
    for(i = 0; i < 6; i++)
    {
        SNode* p = new SNode(rand() % 10);
        p->pNext = pHead1->pNext;
        pHead1->pNext = p;
    }

    SNode* pHead2 = new SNode(0);
    for(i = 0; i < 9; i++)
    {
        SNode* p = new SNode(rand() % 10);
        p->pNext = pHead2->pNext;
        pHead2->pNext = p;
    }
    Print(pHead1);
    Print(pHead2);
    SNode* pSum = Add(pHead1, pHead2);
    Print(pSum);
    Destroy(pHead1);
    Destroy(pHead2);
    Destroy(pSum);
    return 0;
}
```

```
SNode* Add(SNode* pHead1, SNode* pHead2)
{
    SNode* pSum = new SNode(0);
    SNode* pTail = pSum;    //新结点插入到pTail的后面
    SNode* p1 = pHead1->pNext;
    SNode* p2 = pHead2->pNext;
    SNode* pCur;
    int carry = 0;  //进位
    int value;
    while(p1 && p2)
    {
        value = p1->value + p2->value + carry;
        carry = value / 10;
        value %= 10;
        pCur = new SNode(value);
        pTail->pNext = pCur; //新结点链接到pTail的后面
        pTail = pCur;

        p1 = p1->pNext; //处理下一位
        p2 = p2->pNext;
    }

    //处理较长的链
    SNode* p = p1 ? p1 : p2;
    while(p)
    {
        value = p->value + carry;
        carry = value / 10;
        value %= 10;
        pCur = new SNode(value);
        pTail->pNext = pCur;
        pTail = pCur;
        p = p->pNext;
    }

    //处理可能存在的进位
    if(carry != 0)
        pTail->pNext = new SNode(carry);
    return pSum;
}
```


pHead1：4 → 9 → 0 → 4 → 7 → 1

pHead2：1 → 7 → 1 → 5 → 5 → 4 → 2 → 8

pSum：5 → 6 → 2 → 9 → 2 → 6 → 2 → 8

进一步分析与思考

- 因为两个数字求和的范围是[0,18]，进位最大是1，从而，第i位相加不会影响到第i+2位的计算。
- 在发现一个链表为空后，直接结束for循环。最后只需要进位和较长链表的当前结点相加，较长链表的其他结点直接拷贝到最终结果即可。
  - 没有提高时间复杂度，trick而已。
- 利用上述结构实现大整数运算？
  - 如何实现乘法？
  - 如：201314+(100!) = ?

链表的部分翻转

- 给定一个链表，翻转该链表从m到n的位置。要求直接翻转而非申请新空间。
  - 如：给定1→2→3→4→5，m=2，n=4，返回1→4→3→2→5。
  - 假定给出的参数满足：1≤m≤n≤链表长度。

分析

- 空转m-1次，找到第m-1个结点，即开始翻转的第一个结点的前驱，记做head；
- 以head为起始结点遍历n-m次，将第i次时，将找到的结点插入到head的next中即可。
  - 即头插法

Code

64 → 62 → 58 → 78 → 24 → 69 → 0 → 34 → 67 → 41

64 → 62 → 58 → 34 → 0 → 69 → 24 → 78 → 67 → 41

```
typedef struct tagSNode
{
    int value;
    tagSNode* pNext;

    tagSNode(int v):value(v), pNext(NULL) {}
}SNode;
int _tmain(int argc, _TCHAR* argv[])
{
    SNode* pHead = new SNode(0);
    int i;
    for(i = 0; i < 10; i++)
    {
        SNode* p = new SNode(rand() % 100);
        p->pNext = pHead->pNext;
        pHead->pNext = p;
    }
    Print(pHead);
    Reverse(pHead, 4, 8);
    Print(pHead);
    Destroy(pHead);
    return 0;
}
```

```
void Destroy(SNode* p)
{
    SNode* next;
    while(p)
    {
        next = p->pNext;
        delete p;
        p = next;
    }
}
```

```
void Reverse(SNode* pHead, int from, int to)
{
    SNode* pCur = pHead->pNext;
    int i;
    for(i = 0; i < from-1; i++)
    {
        pHead = pCur;
        pCur = pCur->pNext;
    }

    SNode* pPre = pCur;
    pCur = pCur->pNext;
    to--;
    SNode* pNext;
    for(;i < to; i++)
    {
        pNext = pCur->pNext;
        pCur->pNext = pHead->pNext;
        pHead->pNext = pCur;
        pPre->pNext = pNext;
        pCur = pNext;
    }
}
```

排序链表中去重

- 给定排序的链表，删除重复元素，只保留重复元素第一次出现的结点。
- 如：
  - 给定：2→3→3→5→7→8→8→8→9→9→10
  - 返回：2→3→5→7→8→9→10

问题分析

- 若p->next的值和p的值相等，则将p->next->next赋值给p，删除p->next；重复上述过程，直至链表尾端。

Code

```
typedef struct tagSNode
{
    int value;
    tagSNode* pNext;

    tagSNode(int v):value(v), pNext(NULL){}
}SNode;

int _tmain(int argc, _TCHAR* argv[])
{
    SNode* pHead = new SNode(0);
    int data[] = {2,3,3,5,7,8,8,8,9,9,30};
    int size = sizeof(data) / sizeof(int);
    for(int i = size-1; i >= 0; i--)
    {
        SNode* p = new SNode(data[i]);
        p->pNext = pHead->pNext;
        pHead->pNext = p;
    }
    Print(pHead);
    DeleteDuplicateNode(pHead);
    Print(pHead);
    Destroy(pHead);
    return 0;
}
```

```
void DeleteDuplicateNode(SNode* pHead)
{
    SNode* pPre = pHead->pNext;
    SNode* pCur;
    while(pPre)
    {
        pCur = pPre->pNext;
        if(pCur && (pCur->value == pPre->value))
        {
            pPre->pNext = pCur->pNext;
            delete pCur;
        }
        else
        {
            pPre = pCur;
        }
    }
}
```

Code2

- 分析该代码的正确性

```
void DeleteDuplicateNode2(SNode* pHead)
{
    SNode* pPre = pHead;
    SNode* pCur = pPre->pNext;
    SNode* pNext;
    while(pCur)
    {
        pNext = pCur->pNext;
        while(pNext && (pCur->value == pNext->value))
        {
            pPre->pNext = pNext;
            delete pCur;
            pCur = pNext;
            pNext = pCur->pNext;
        }
        pPre = pCur;
        pCur = pNext;
    }
}
```

排序链表中去重2

- 若题目变成：若发现重复元素，则重复元素全部删除，代码应该怎么实现呢？
- 如：
  - 给定：2→3→3→5→7→8→8→8→9→9→10
  - 返回：2→5→7→10

Code

```
void DeleteDuplicateNode3(SNode* pHead)
{
    SNode* pPre = pHead;
    SNode* pCur = pPre->pNext;
    SNode* pNext;
    bool bDup;
    while(pCur)
    {
        pNext = pCur->pNext;
        bDup = false;
        while(pNext && (pCur->value == pNext->value))
        {
            pPre->pNext = pNext;
            delete pCur;
            pCur = pNext;
            pNext = pCur->pNext;
            bDup = true;
        }
        if(bDup)    //此刻的pCur与原数据重复，删之
        {
            pPre->pNext = pNext;
            delete pCur;
        }
        else        //pCur未发现重复，则pPre后移
        {
            pPre = pCur;
        }
        pCur = pNext;
    }
}
```

链表划分

- 给定一个链表和一个值x，将链表划分成两部分，使得划分后小于x的结点在前，大于等于x的结点在后。在这两部分中要保持原链表中的出现顺序。
  - 如：给定链表1→4→3→2→5→2和x = 3，返回1→2→2→4→3→5。

问题分析

- 分别申请两个指针p1和p2，小于x的添加到p1中，大于等于x的添加到p2中；最后，将p2链接到p1的末端即可。
- 时间复杂度是O(N)，空间复杂度为O(1)；该问题其实说明：快速排序对于单链表存储结构仍然适用。
  - 注：不是所有排序都方便使用链表存储，如堆排序，将不断的查找数组的n/2和n的位置，用链表做存储结构会不太方便。

Code

```
typedef struct tagSNode
{
    int value;
    tagSNode* pNext;

    tagSNode(int v):value(v), pNext(NULL){}
}SNode;

int _tmain(int argc, _TCHAR* argv[])
{
    SNode* pHead = new SNode(0);
    pHead->pNext = NULL;
    for(int i = 0; i < 10; i++)
    {
        SNode* p = new SNode(rand() % 100);
        p->pNext = pHead->pNext;
        pHead->pNext = p;
    }
    Print(pHead);
    Partition(pHead, 50);
    Print(pHead);
    Destroy(pHead);
    return 0;
}
```

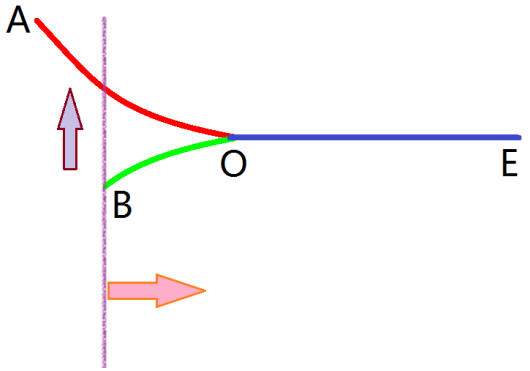
```
void Destroy(SNode* p)
{
    SNode* next;
    while(p)
    {
        next = p->pNext;
        delete p;
        p = next;
    }
}
```

```
void Partition(SNode* pHead, int pivotKey)
{
    //两个链表的头指针
    SNode* pLeftHead = new SNode(0);
    SNode* pRightHead = new SNode(0);

    //两个链表的当前最后一个元素
    SNode* left = pLeftHead;
    SNode* right = pRightHead;
    SNode* p = pHead->pNext;
    while(p)    //遍历原链表
    {
        if(p->value < pivotKey)
        {
            left->pNext = p;
            left = p;
        }
        else
        {
            right->pNext = p;
            right = p;
        }
        p = p->pNext;
    }

    //将right链接到left尾部
    left->pNext = pRightHead->pNext;
    right->pNext = NULL;

    //将整理好的链表赋值给当前链表头部
    pHead->pNext = pLeftHead->pNext;

    delete pLeftHead;
    delete pRightHead;
}
```

单链公共结点问题

- 给定两个单向链表，计算两个链表的第一个公共结点，若没有公共节点，返回空。

单链公共结点问题



- 令两链表的长度为m、n，不妨认为m≥n，由于两个链表从第一个公共结点到链表的尾结点是完全重合的。所以前面的(m-n)个结点一定没有公共结点。
- 算法：先分别遍历两个链表得到它们的长度m，n。长链表空转|m-n|次，同步遍历两链表，直到找到相同结点或到链表结束。
- 时间复杂度为O(m+n)。

Code

```
typedef struct tagSNode
{
    int value;
    tagSNode* pNext;

    tagSNode(int v):value(v), pNext(NULL) {}
}SNode;
```

```
int CalcLength(SNode* p)
{
    int nLen = 0;
    while(p)
    {
        p = p->pNext;
        nLen++;
    }
    return nLen;
}
```

```
SNode* FindFirstSameNode(SNode* pA, SNode* pB)
{
    //因为有头指针，所以指向第一个有效结点
    pA = pA->pNext;
    pB = pB->pNext;

    //计算两个链表的长度
    int nA = CalcLength(pA);
    int nB = CalcLength(pB);
    if(nA > nB)
    {
        swap(pA, pB);
        swap(nA, nB);
    }

    //空转nB-nA次
    for(int i = 0; i < nB - nA; i++)
        pB = pB->pNext;

    //齐头并进
    while(pA)
    {
        if(pA == pB)
            return pA;
        pA = pA->pNext;
        pB = pB->pNext;
    }
    return NULL;
}
```

小结

- 单链公共结点问题中，如果是链表存在环，则需要使用快慢指针的方式计算公共节点。
  - 两个指针，每次分布移动一个/两个结点
- 可以发现，纯链表的题目，往往不难，但需要需要扎实的Coding基本功，在实现过程中，要特别小心next的指向，此外，删除结点时，一定要确保该结点不再需要。
- 小心分析引用类型的指针。

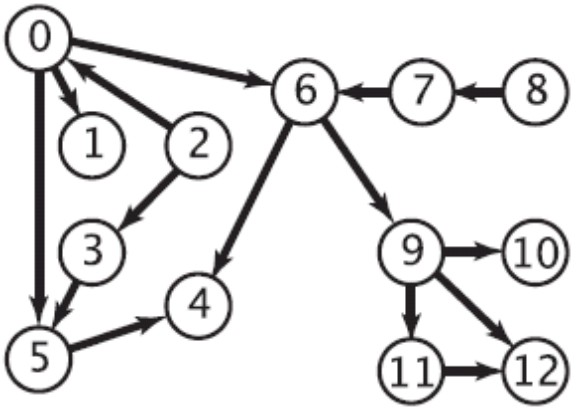
queue

- 队列是一种特殊的线性表，只允许在表的前端front进行删除操作，在表的后端rear进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。
- 队列元素服从先进先出原则
  - FIFO——First In First Out

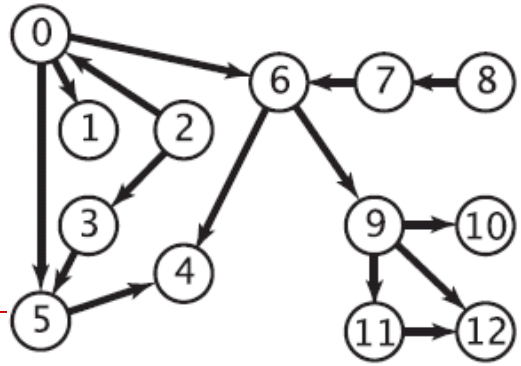
拓扑排序

- 对一个有向无环图(Directed Acyclic Graph，DAG)G进行拓扑排序，是将G中所有顶点排成线性序列，使得图中任意一对顶点u、v，若边(u,v)∈E(G)，则在线性序列中u出现在v之前。
- 一种可能的拓扑排序结果

2→8→0→3→7→1→5→6→9→4→11→10→12



拓扑排序的方法



- 从有向图中选择一个没有前驱(即入度为0)的顶点并且输出它；
- 从网中删去该顶点，并且删去从该顶点发出的全部有向边；
- 重复上述两步，直到剩余的网中不再存在没有前驱的顶点为止。

Code

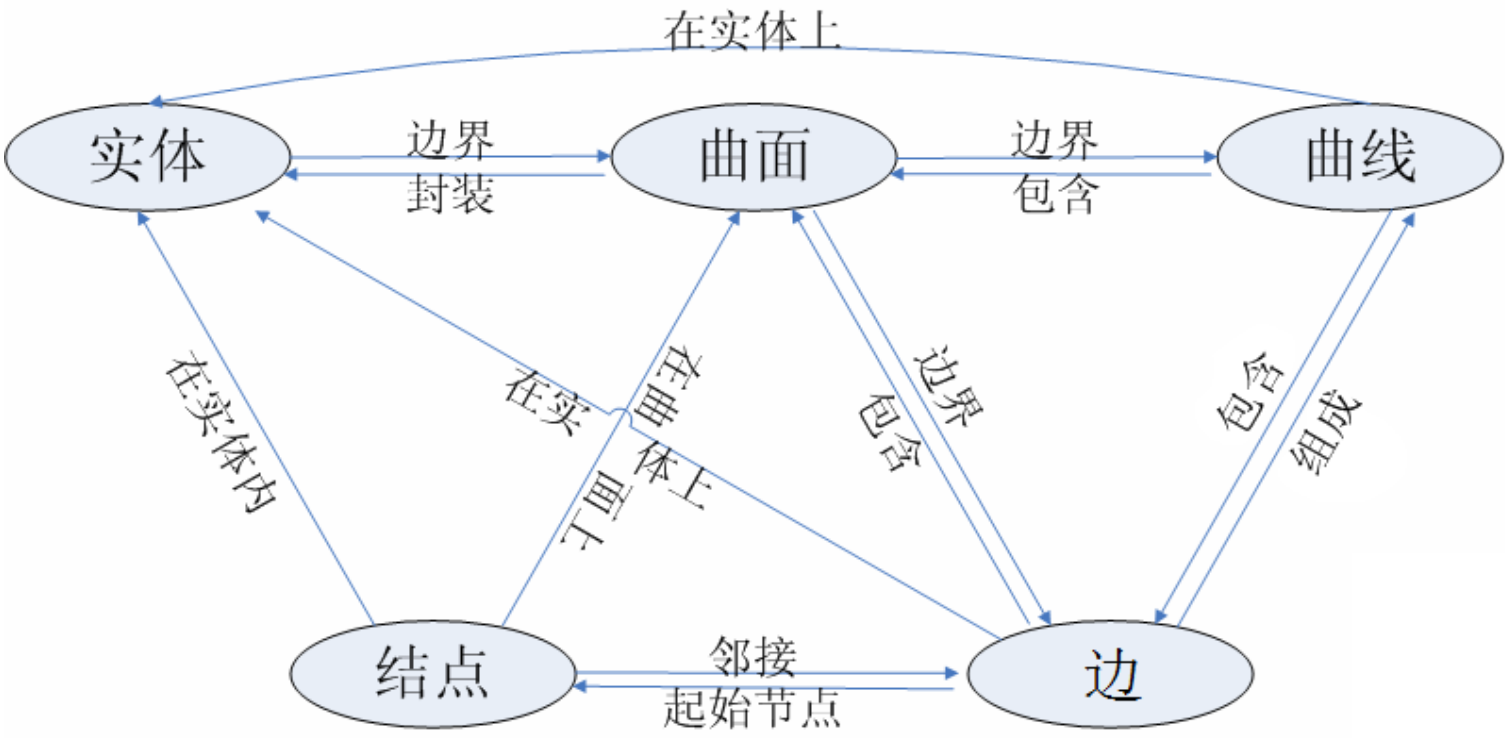
```
//结点数为n，用邻接矩阵gragh[n][n]存储边权，
//用indegree[n]存储每个结点的入度
void topologic(int* toposort)
{
    int cnt = 0;    //当前拓扑排序列表中有多少结点
    queue<int> q;   //保存入度为0的结点：还可以用栈甚至随机取
    int i;
    for(i = 0; i < n; i++)
    {
        if(indegree[i] == 0)
            q.push(i);
    }
    int cur;    //当前入度为0的结点
    while(!q.empty())
    {
        cur = q.front();
        q.pop();
        toposort[cnt++] = cur;
        for(i = 0; i < n; i++)
        {
            if(gragh[cur][i] != 0)
            {
                indegree[i]--;
                if(indegree[i] == 0)
                    q.push(i);
            }
        }
    }
}
```

拓扑排序的进一步思考

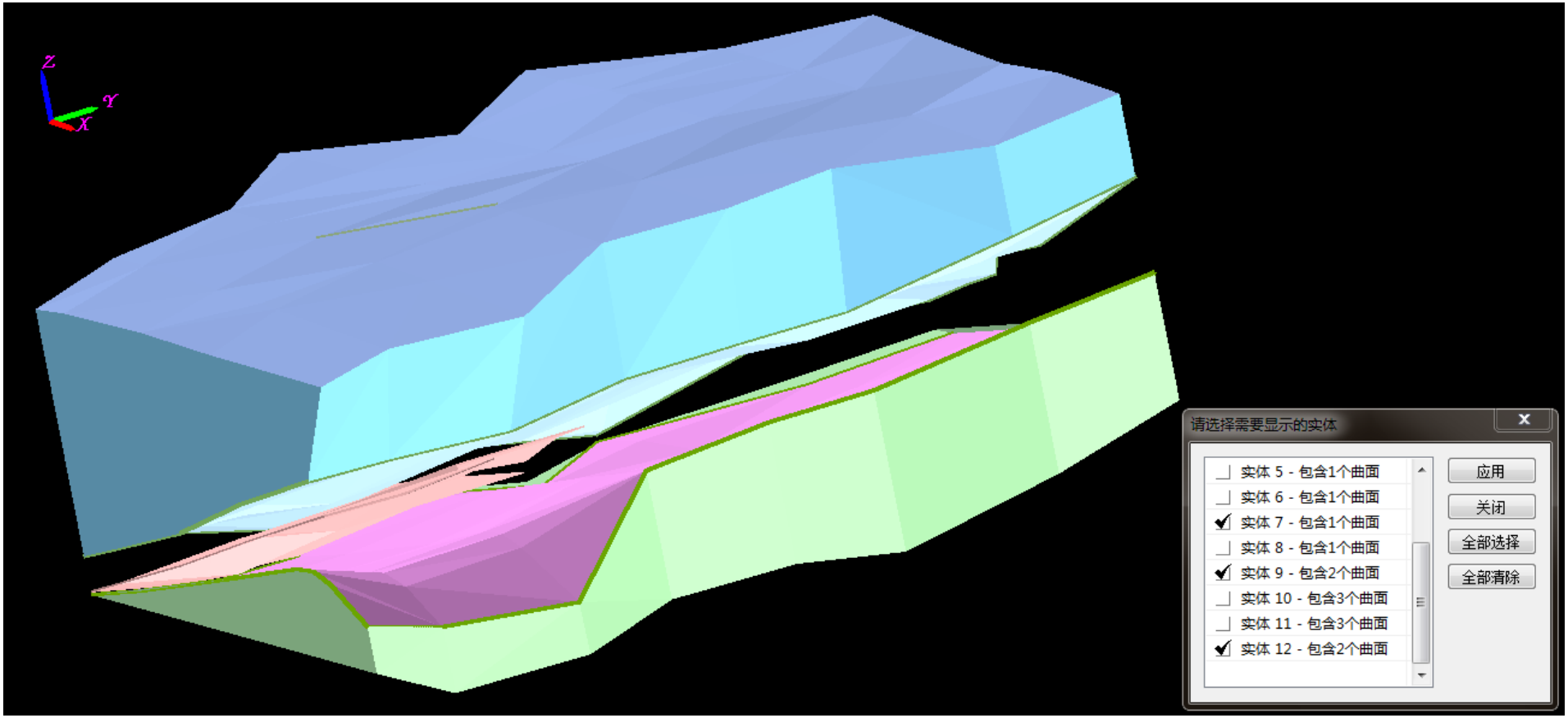
- 拓扑排序的本质是不断输出入度为0的点，该算法可用于判断图中是否存在环；
- 可以用队列(或者栈)保存入度为0的点，避免每次遍历所有点；
  - 每次更新连接点的入度即可。
- 拓扑排序其实是给定了结点的一组偏序关系。
- “拓扑”的涵义不限于此，在GIS中，它往往指点、线、面、体之间的相互邻接关系，即“橡皮泥集合”。存储这些关系，往往能够对某些算法带来好处。
  - 计算不自交的空间曲面是否能够围成三维体
    - 提示：任意三维边都邻接两个三维曲面

扩展：拓扑的几何含义

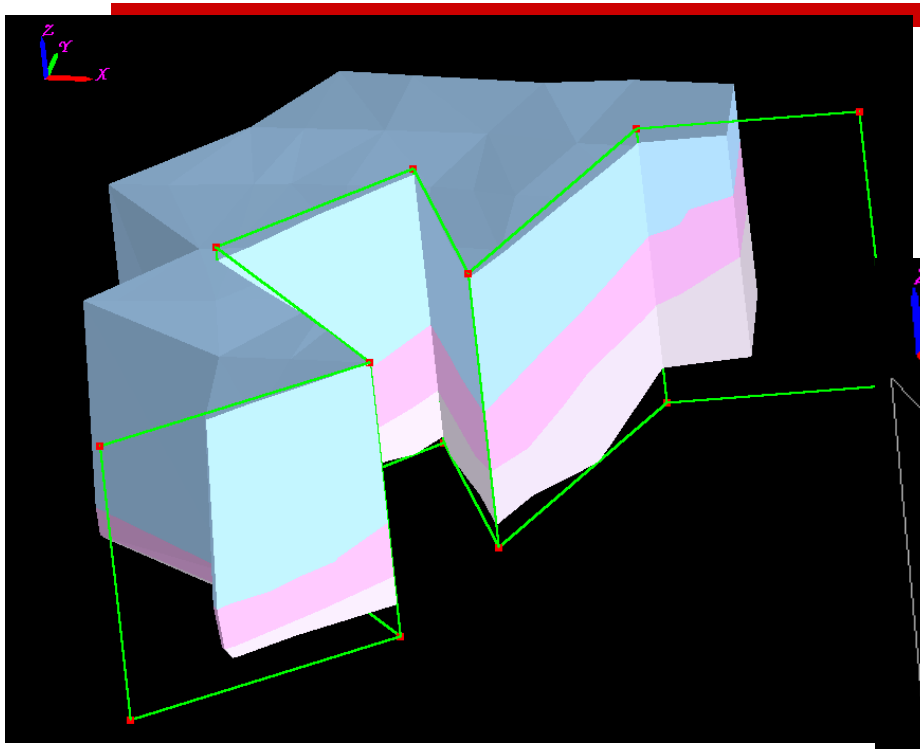
- 一种关系：如三维数据间的拓扑关系

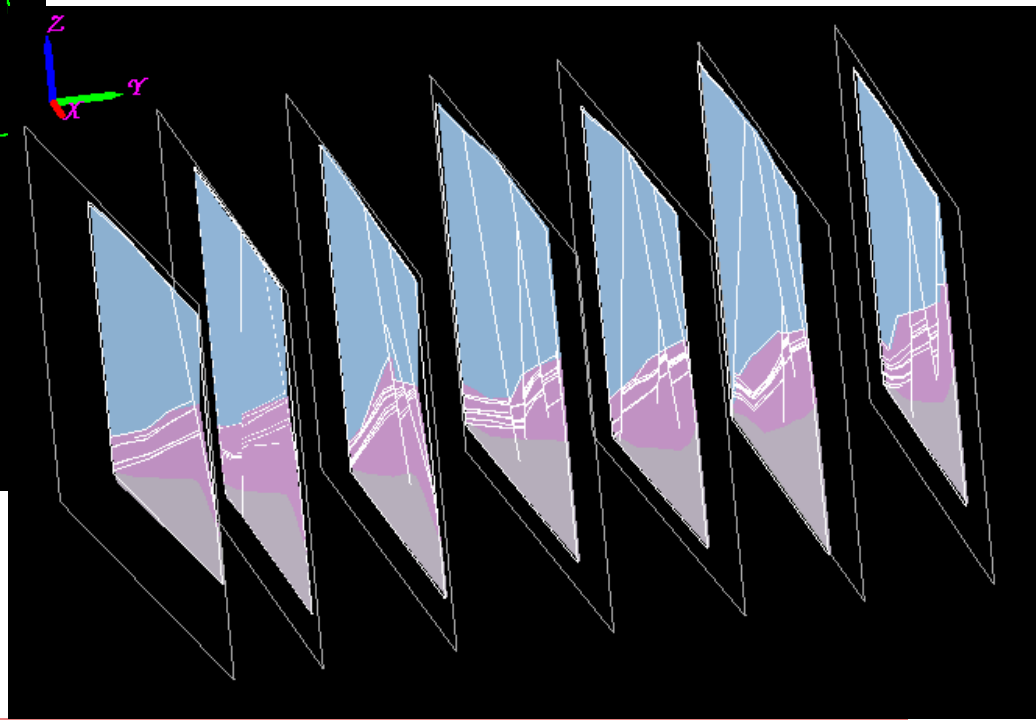


三维拓扑重建


Z
Y
X
请选择需要显示的实体
实体 5 - 包含1个曲面
实体 6 - 包含1个曲面
实体 7 - 包含1个曲面
实体 8 - 包含1个曲面
实体 9 - 包含2个曲面
实体 10 - 包含3个曲面
实体 11 - 包含3个曲面
实体 12 - 包含2个曲面
应用
关闭
全部选择
全部清除

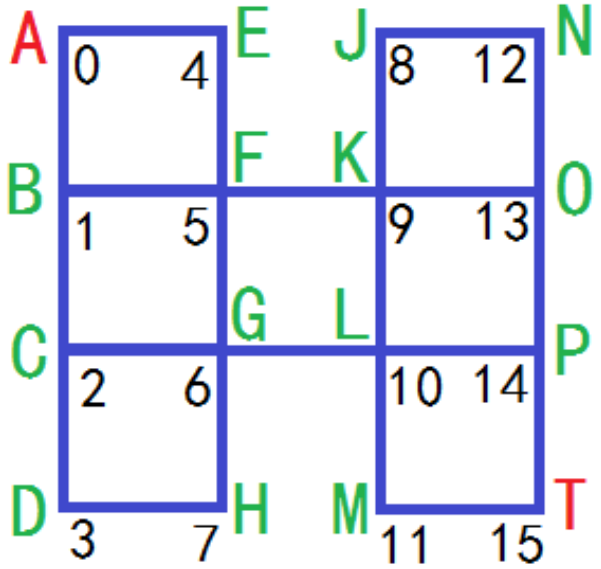
视角再次放大——面面、面线的拓扑





最短路径条数问题

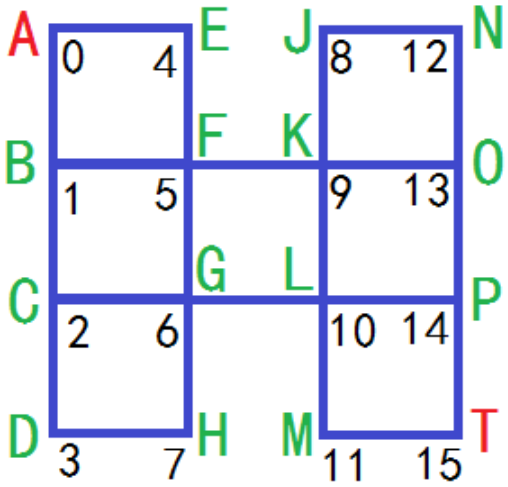
- 给定如图所示的无向连通图，假定图中所有边的权值都为1，显然，从源点A到终点T的最短路径有多条，求不同的最短路径的数目。



数据结构的选择

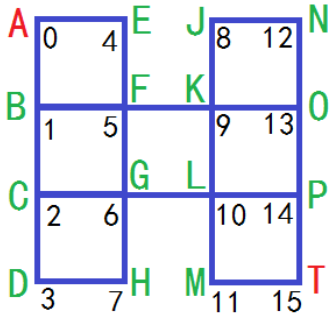
- 权值相同的最短路径问题，则单源点Dijkstra算法退化成BFS广度优先搜索，假定起点为0，终点为N：
  - 结点步数step[0…N-1]初始化为0
  - 路径数目pathNum[0…N-1]初始化为0
  - pathNum[0] = 1

算法分析



- 若从当前结点i扩展到邻接点j时：
  - 若step[j]为0，则
    - step[j]=step[i]+1，pathN[j] = pathN[i]
  - 若step[j]==step[i]+1，则
    - pathN[j] += pathN[i]
  - 可考虑扩展到结点N，则提前终止算法。

Code




```
int _tmain(int argc, _TCHAR* argv[])
{
    int G[N][N];
    memset(G, 0, sizeof(int)*N*N);
    G[0][1] = G[0][4] = 1;
    G[1][5] = G[1][0] = G[1][2] = 1;
    G[2][1] = G[2][6] = G[2][3] = 1;
    G[3][2] = G[3][7] = 1;
    G[4][0] = G[4][5] = 1;
    G[5][1] = G[5][4] = G[5][6] = G[5][9] = 1;
    G[6][2] = G[6][5] = G[6][7] = G[6][10] = 1;
    G[7][3] = G[7][6] = 1;
    G[8][9] = G[8][12] = 1;
    G[9][8] = G[9][13] = G[9][10] = 1;
    G[10][9] = G[10][14] = G[10][11] = 1;
    G[11][10] = G[11][15] = 1;
    G[12][8] = G[12][13] = 1;
    G[13][9] = G[13][12] = G[13][14] = 1;
    G[14][10] = G[14][13] = G[14][15] = 1;
    G[15][11] = G[15][14] = 1;

    cout << Calc(G) << endl;
    return 0;
}
```

```
const int N = 16;
int Calc(int G[N][N])
{
    int step[N];    //每个结点第几步可以达到
    int stepNumber[N];  //到每个结点有几种走法
    memset(step, 0, sizeof(int)*N);
    memset(stepNumber, 0, sizeof(int)*N);
    stepNumber[0] = 1;
    queue<int> q;   //当前搜索的结点
    q.push(0);
    int from,i,s;
    while(!q.empty())
    {
        from = q.front();
        q.pop();
        s = step[from] + 1;
        for(i = 1; i < N; i++) //0是起点，不遍历
        {
            if(G[from][i] == 1) //连通
            {
                //i尚未可达或发现更快的路(权值不同才可能)
                if((step[i] == 0) || (step[i] > s))
                {
                    step[i] = s;
                    stepNumber[i] = stepNumber[from];
                    q.push(i);
                }
                else if(step[i] == s) //发现相同长度的路径
                {
                    stepNumber[i] += stepNumber[from];
                }
            }
        }
    }
    return stepNumber[N-1];
}
```

括号是否匹配

- 给定字符串，仅由"()[]{}"六个字符组成。设计算法，判断该字符串是否有效。
  - 括号必须以正确的顺序配对，如：“()”、“()[]”是有效的，但“([)]”无效。

以下算法是否可行

- 对于给定的字符串str[0…N-1]，它的前缀串str[0…k]小括号的左括号减去小括号的右括号的数目，记做p[0]；
  - 同理使用p[1]、p[2]记录中括号和大括号的信息
- 思路：
- 将k从0到N-1遍历：
  - 对于0≤k≤N-2：p[0]≥0、p[1]≥0，p[2]≥0
  - 对于k=N-1，p[0]==0、p[1]==0，p[2]==0
  - 则字符串括号匹配，否则，不匹配。

算法分析

- 在考察第i位字符C与前面的括号是否匹配时：
- 如果C为左括号，开辟缓冲区记录下来，希望C能够与后面出现的同类型最近右括号匹配。
- 如果C为右括号，考察它能否与缓冲区中的左括号匹配。
  - 这个匹配过程，是检查缓冲区最后出现的同类型左括号
  - 即：后进先出——栈

括号匹配算法流程

- 从前向后扫描字符串；
- 遇到左括号x，就压栈x；
- 遇到右括号y：
  - 如果发现栈顶元素x和该括号y匹配，则栈顶元素出栈，继续判断下一个字符。
  - 如果栈顶元素x和该括号y不匹配，字符串不匹配；
  - 如果栈为空，字符串不匹配；
- 扫描完成后，如果栈恰好为空，则字符串匹配，否则，字符串不匹配。

Code

```
bool IsLeftParentheses(char c)
{
    return (c == '(') ||
      (c == '[') || (c == '{');
}

bool IsMatch(char left, char c)
{
    if(left == '(')
        return c == ')';
    if(left == '[')
        return c == ']';
    if(left == '{')
        return c == '}';
    return false;
}
```

```
bool MatchParentheses(const char* p)
{
    stack<char> s;
    char cur;
    while(*p)
    {
        cur = *p;
        if(IsLeftParentheses(cur))
            s.push(cur);
        else //if(IsRightParentheses(cur))
        {
            if(s.empty() || !IsMatch(s.top(),cur))
                return false;
            s.pop();
        }
        p++;
    }
    return s.empty();
}

int _tmain(int argc, _TCHAR* argv[])
{
    char* p = "(({})[])[()]";
    bool bMatch = MatchParentheses(p);
    if(bMatch)
        cout << p << "括号匹配。\n";
    else
        cout << p << "括号不匹配。\n";
    return 0;
}
```

最长括号匹配

- 给定字符串，仅包含左括号‘(’和右括号‘)’，它可能不是括号匹配的，设计算法，找出最长匹配的括号子串，返回该子串的长度。
- 如：
  - (()：2
  - ()()：4
  - ()(())：6
  - (()())：6

算法分析

(()：2

()()：4

()(())：6

((())：6

- 记起始匹配位置start=-1；最大匹配长度ml=0；
- 考察第i位字符c：
- 如果c为左括号，压栈；
- 如果c为右括号，它一定与栈顶左括号匹配；
  - 如果栈为空，表示没有匹配的左括号，start=i，为下一次可能的匹配做准备
  - 如果栈不空，出栈(因为和c匹配了)；
    - 如果栈为空，i-start即为当前找到的匹配长度，检查i-start是否比ml更大，使得ml得以更新；
    - 如果栈不空，则当前栈顶元素t是上次匹配的最后位置，检查i-t是否比ml更大，使得ml得以更新。
- 注：因为入栈的一定是左括号，显然没有必要将它们本身入栈，应该入栈的是该字符在字符串中的索引。

Code

- 如果c为左括号，压栈；
- 如果c为右括号，它一定与栈顶左括号匹配；
  - 如果栈为空，表示没有匹配的左括号，start=i，为下一次可能的匹配做准备
  - 如果栈不空，出栈(因为和c匹配了)；
    - 如果栈为空，i-start即为当前找到的匹配长度，检查i-start是否比ml更大，使得ml得以更新；
    - 如果栈不空，则当前栈顶元素t是上次匹配的最后位置，检查i-t是否比ml更大，使得ml得以更新。

```
int GetLongestParenthese(const char* p)
{
    int size = (int)strlen(p);
    stack<int> s;
    int answer = 0; //最终解
    int start = -1; //左括号的前一个位置
    for(int i = 0; i < size; i++)
    {
        if(p[i] == '(')
        {
            s.push(i);
        }
        else    //p[i] == ')'
        {
            if(s.empty())
                start = i;
            else
            {
                s.pop();
                if(s.empty())
                    answer = max(answer, i-start);
                else
                    answer = max(answer, i-s.top());
            }
        }
    }
    return answer;
}
```

附：进一步思考

- 经过分析算法得知，只有在右括号和左括号的发生匹配时，才有可能更新最终解；
- 做记录前缀串p[0…i-1]中左括号数目与右括号数目的差x，若x为0时，考察是否最终解得以更新即可。这个差x，其实是入栈的数目，代码中用“深度”deep表达；
- 由于可能出现左右括号不相等——尤其是左括号数目大于右括号数目，所以，再从右向前扫描一次。
- 这样完成的代码，用deep值替换了stack栈，空间复杂度由O(N)降到O(1)。

Code

```
int GetLongestParenthese2(const char* p)
{
    int size = (int)strlen(p);
    int answer = 0; //最终解
    int deep = 0;   //遇到了多少左括号
    int start = -1; //最深的(deep==0时)左括号的位置
                    //其实，为了方便计算长度，该变量是最深左括号的前一个位置
    int i;
    for(i = 0; i < size; i++)
    {
        if(p[i] == '(')
        {
            deep++;
        }
        else    //p[i] == ')'
        {
            deep--;
            if(deep == 0)
            {
                answer = max(answer, i - start);
            }
            else if(deep < 0)   //说明右括号数目大于左括号，初始化为for循环前
            {
                deep = 0;
                start = i;
            }
        }
    }

    deep = 0;       //遇到了多少右括号
    start = size;   //最深的(deep==0时)右括号的位置
                    //其实，为了方便计算长度，该变量是最深右括号的后一个位置
    for(i = size-1; i >= 0; i--)
    {
        if(p[i] == ')')
        {
            deep++;
        }
        else    //p[i] == '('
        {
            deep--;
            if(deep == 0)
            {
                answer = max(answer, start - i);
            }
            else if(deep < 0)   //说明右括号数目大于左括号，初始化为for循环前
            {
                deep = 0;
                start = i;
            }
        }
    }

    return answer;
}
```

half-part

(()：2

()()：4

()(())：6

((())：6

```
int GetLongestParenthese2(const char* p)
{
    int size = (int)strlen(p);
    int answer = 0; //最终解
    int deep = 0;   //遇到了多少左括号
    int start = -1; //最深的(deep==0时)左括号的位置
                    //其实，为了方便计算长度，该变量是最深左括号的前一个位置
    int i;
    for(i = 0; i < size; i++)
    {
        if(p[i] == '(')
        {
            deep++;
        }
        else    //p[i] == ')'
        {
            deep--;
            if(deep == 0)
            {
                answer = max(answer, i - start);
            }
            else if(deep < 0)   //说明右括号数目大于左括号，初始化为for循环前
            {
                deep = 0;
                start = i;
            }
        }
    }

    deep = 0;       //遇到了多少右括号
    start = size;   //最深的(deep==0时)右括号的位置
                    //其实，为了方便计算长度，该变量是最深右括号的后一个位置
```

p="()(())"

```
int GetLongestParenthese2(const char* p)
{
    int size = (int)strlen(p);
    int answer = 0; //最终解
    int deep = 0;   //遇到了多少左括号
    int start = -1; //最深的(deep==0时)左括号的位置
                    //其实，为了方便计算长度，该变量是最深左括号的前一个位置
    int i;
    for(i = 0; i < size; i++)
    {
        if(p[i] == '(')
        {
            deep++;
        }
        else    //p[i] == ')'
        {
            deep--;
            if(deep == 0)
            {
                answer = max(answer, i - start);
            }
            else if(deep < 0)   //说明右括号数目大于左括号，初始化为for循环前
            {
                deep = 0;
                start = i;
            }
        }
    }

    deep = 0;       //遇到了多少右括号
    start = size;   //最深的(deep==0时)右括号的位置
                    //其实，为了方便计算长度，该变量是最深右括号的后一个位置
```

循环次数	深度	起始位置	当前解
<- 第一次循环 ->			
0	1	-1	0
1	0	-1	2
2	1	-1	2
3	2	-1	2
4	1	-1	2
5	0	-1	6
6	0	6	6
<- 第二次循环 ->			
6	1	7	6
5	2	7	6
4	3	7	6
3	2	7	6
2	1	7	6
1	2	7	6
0	1	7	6

空间复杂度仅O(1)的最长括号匹配

分析括号串p="((()()))"：

循环次数	深度	起始位置	当前解
<- 第一次循环 ->			
0	1	-1	0
1	2	-1	0
2	3	-1	0
3	4	-1	0
4	3	-1	0
5	4	-1	0
6	3	-1	0
7	2	-1	0
8	1	-1	0
<- 第二次循环 ->			
8	1	9	0
7	2	9	0
6	3	9	0
5	2	9	0
4	3	9	0
3	2	9	0
2	1	9	0
1	0	9	8
0	0	0	8

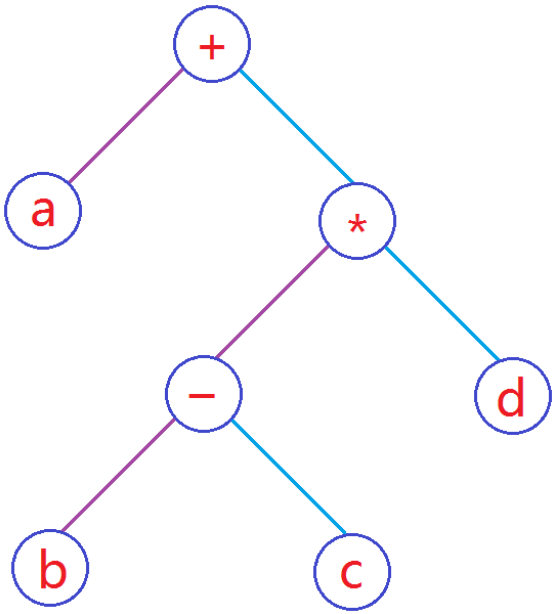
```
int GetLongestParenthese2(const char* p)
{
    int size = (int)strlen(p);
    int answer = 0; //最终解
    int deep = 0;   //遇到了多少左括号
    int start = -1; //最深的(deep==0时)左括号的位置
                    //其实，为了方便计算长度，该变量是最深左括号的前一个位置
    int i;
    for(i = 0; i < size; i++)
    {
        if(p[i] == '(')
        {
            deep++;
        }
        else    //p[i] == ')'
        {
            deep--;
            if(deep == 0)
            {
                answer = max(answer, i - start);
            }
            else if(deep < 0)   //说明右括号数目大于左括号，初始化为for循环前
            {
                deep = 0;
                start = i;
            }
        }
    }

    deep = 0;       //遇到了多少右括号
    start = size;   //最深的(deep==0时)右括号的位置
                    //其实，为了方便计算长度，该变量是最深右括号的后一个位置
    for(i = size-1; i >= 0; i--)
    {
        if(p[i] == ')')
        {
            deep++;
        }
        else    //p[i] == '('
        {
            deep--;
            if(deep == 0)
            {
                answer = max(answer, start - i);
            }
            else if(deep < 0)   //说明右括号数目大于左括号，初始化为for循环前
            {
                deep = 0;
                start = i;
            }
        }
    }

    return answer;
}
```

逆波兰表达式RPN

- Reverse Polish Notation，即后缀表达式。
- 习惯上，二元运算符总是置于与之相关的两个运算对象之间，即中缀表达方法。波兰逻辑学家J.Lukasiewicz于1929年提出了运算符都置于其运算对象之后，故称为后缀表示。
- 如：
  - 中缀表达式：a+(b-c)*d
  - 后缀表达式：abc-d*+

运算与二叉树

- 事实上，二元运算的前提下，中缀表达式可以对应一颗二叉树；逆波兰表达式即该二叉树后序遍历的结果。
- 中缀表达式：a+(b-c)*d
- 后缀表达式：abc-d*+



  - 该结论对多元运算也成立，如“非运算”等

计算逆波兰表达式

- 计算给定的逆波兰表达式的值。有效操作只有+-*/，每个操作数都是整数。
- 如：
  - "2", "1", "+", "3", "*"：9——(2+1)*3
  - "4", "13", "5", "/", "+"：6——4+(13/5)

逆波兰表达式的计算方法

- abc-d*+
- 若当前字符是操作数，则压栈
- 若当前字符是操作符，则弹出栈中的两个操作数，计算后仍然压入栈中
  - 若某次操作，栈内无法弹出两个操作数，则表达式有误。

Code

```
int _tmain(int argc, _TCHAR* argv[])
{
    const char* str[] = {"2","1","+","3","*"};
    int value = ReversePolishNotation(str, sizeof(str)/sizeof(const char*));
    cout << value << endl;
    return 0;
}
```

```
int ReversePolishNotation(const char* str[], int size)
{
    stack<int> s;
    int a,b;
    const char* token;
    for(int i = 0; i < size; i++)
    {
        token = str[i];
        if(!IsOperator(token))
        {
            s.push(atoi(token));
        }
        else
        {
            b = s.top();
            s.pop();
            a = s.top();
            s.pop();
            if(token[0] == '+')
                s.push(a+b);
            else if(token[0] == '-')
                s.push(a-b);
            else if(token[0] == '*')
                s.push(a*b);
            else// if(token[0] == '/')
                s.push(a/b);
        }
    }
    return s.top();
}
```

```
bool IsOperator(const char* token)
{
    return ((token[0] == '+')||(token[0] == '-')
        || (token[0] == '*')||(token[0] == '/'));
}
```

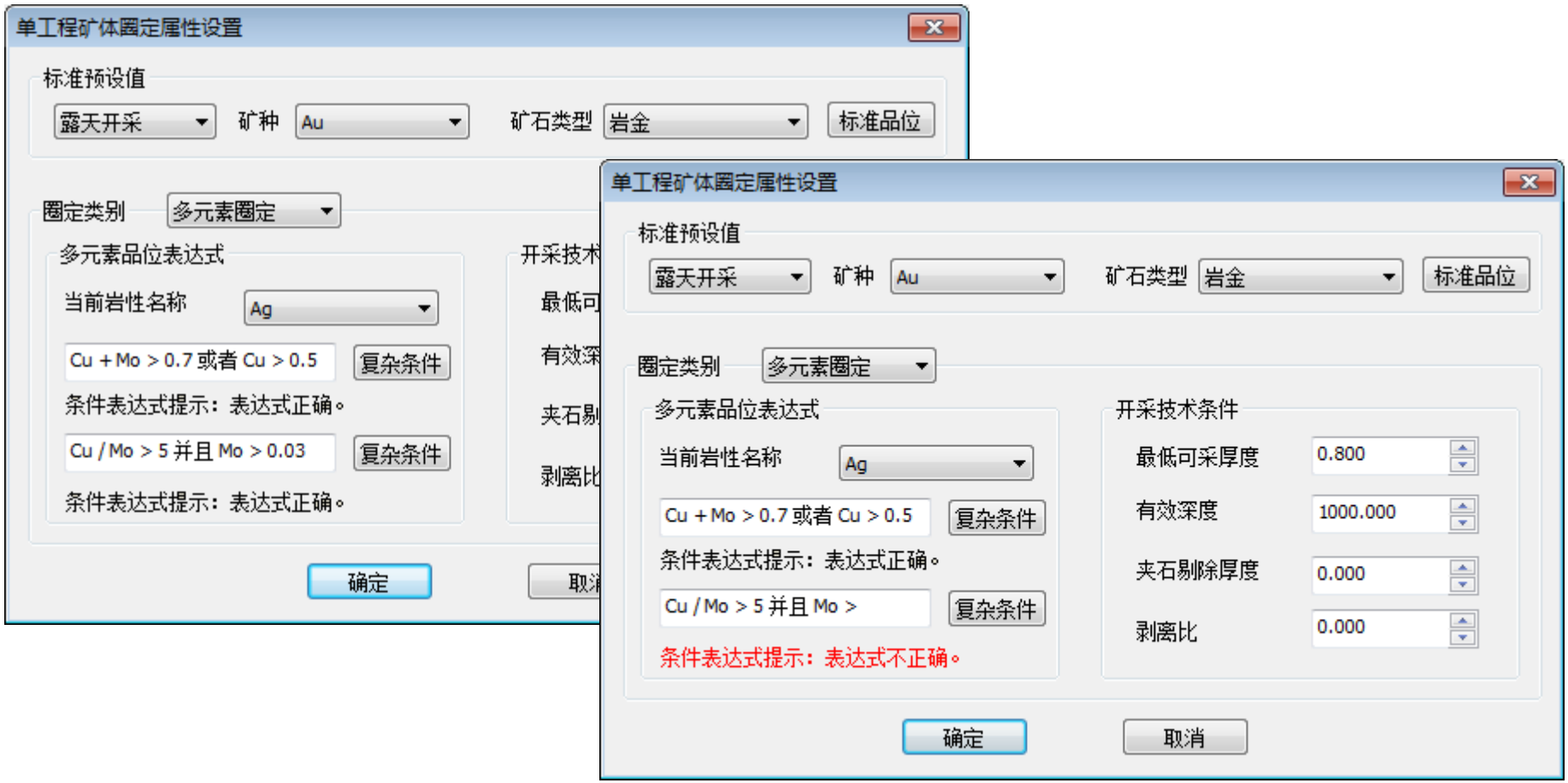
逆波兰表达式

- 计算数学表达式的最常用方法；
- 在实践中，往往给出的不是立即数，而是变量名称；若经常计算且表达式本身不变，可以事先将中缀表达式转换成逆波兰表达式存储。

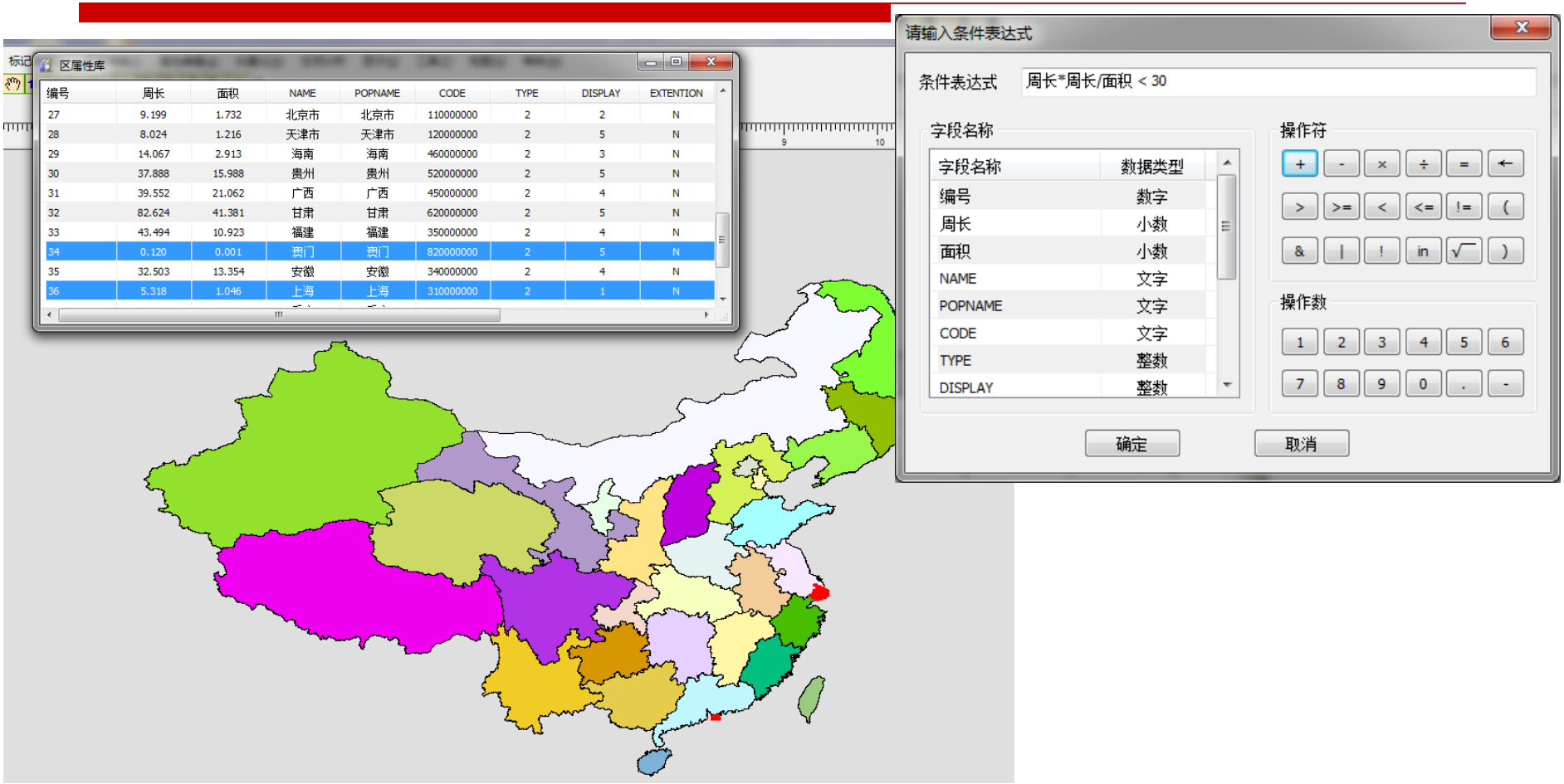
计算器

- 将中缀表达式转换成逆波兰表达式，然后正常计算。

逆波兰表达式的用途


单工程矿体圈定属性设置
标准预设值
露天开采
矿种
Au
矿石类型
岩金
标准品位
圈定类别
多元素圈定
多元素品位表达式
当前岩性名称
Ag
Cu + Mo > 0.7 或者 Cu > 0.5
复杂条件
条件表达式提示：表达式正确。
Cu / Mo > 5 并且 Mo > 0.03
复杂条件
条件表达式提示：表达式正确。
开采技术条件
最低可采厚度
0.800
有效深度
1000.000
夹石剔除厚度
0.000
剥离比
0.000
确定
取消
单工程矿体圈定属性设置
标准预设值
露天开采
矿种
Au
矿石类型
岩金
标准品位
圈定类别
多元素圈定
多元素品位表达式
当前岩性名称
Ag
Cu + Mo > 0.7 或者 Cu > 0.5
复杂条件
条件表达式提示：表达式正确。
Cu / Mo > 5 并且 Mo >
复杂条件
条件表达式提示：表达式不正确。
开采技术条件
最低可采厚度
0.800
有效深度
1000.000
夹石剔除厚度
0.000
剥离比
0.000
确定
取消

省级行政区中哪几个最接近圆形？



入栈出栈问题

- 给定无重复元素的两个等长数组，分别表述入栈序列和出栈序列，请问：这样的出栈序列是否可行。
  - 如：入栈序列为“ABCDEFG”、出栈序列为“BAEDFGC”，则可行。
  - 入栈序列“ABCD”、出栈序列“BDAC”，不可行。

问题分析

- 使用一个堆栈S来模拟压栈出栈的操作。记入栈序列为A，出栈序列为B
- 遍历B的每个元素b：
- 情形1：若b等于栈顶元素s，恰好匹配，则检查B的下一个元素，栈顶元素s出栈；
- 情形2：若b不等于栈顶元素s，则将A的当前元素入栈，目的是希望在A的剩余元素中找到b。
  - 在情形1中，若栈S为空，则认为b无法与栈内元素匹配，则调用情形2。

Code1

```
bool IsPossible(const char* strIn, const char* strOut)
{
    stack<char> s;
    while(*strOut)
    {
        if(!s.empty())
        {
            if(s.top() == *strOut)
            {
                s.pop();
                strOut++;
            }
            else
            {
                if(*strIn == 0)
                    return false;
                s.push(*strIn);
                strIn++;
            }
        }
        else
        {
            if(*strIn == 0)
                return false;
            s.push(*strIn);
            strIn++;
        }
    }
    return true;
}

int _tmain(int argc, _TCHAR* argv[])
{
    char* strIn = "ABCDEFG";
    char* strOut = "BAEDFGC";
    bool b = IsPossible(strIn, strOut);
    cout << b <<endl;
    return 0;
}
```

Code2

```
bool IsPossible(const char* strIn, const char* strOut)
{
    stack<char> s;
    while(*strOut)
    {
        if(!s.empty() && (s.top() == *strOut))
        {
            s.pop();
            strOut++;
        }
        else
        {
            if(*strIn == 0)
                return false;
            s.push(*strIn);
            strIn++;
        }
    }
    return true;
}
int _tmain(int argc, _TCHAR* argv[])
{
    char* strIn = "ABCDEFG";
    char* strOut = "BAEDFGC";
    bool b = IsPossible(strIn, strOut);
    cout << b <<endl;
    return 0;
}
```

小结

- 一般而言，栈的重要度大于队列。
- 栈的用途非常广泛，除了表达式求值，在深度优先遍历、保存现场等问题中常常出现。
- 思考：一个栈(无穷大)的进栈序列为1,2,3,..n,共多少种不同的出栈序列？

练兵场

- Linked List Cycle
  - https://leetcode.com/problems/linked-list-cycle/
- Linked List Cycle II
  - https://leetcode.com/problems/linked-list-cycle-ii/
- Largest Rectangle in Histogram
  - https://leetcode.com/problems/largest-rectangle-in-histogram/
- Valid Parentheses
  - https://leetcode.com/problems/valid-parentheses/

我们在这里

- http://wenda.ChinaHadoop.cn
  - 视频/课程/社区
- 微博
  - @ChinaHadoop
  - @邹博_机器学习
- 微信公众号
  - 小象
  - 大数据分析挖掘

感谢大家！

恳请大家批评指正！

课前甜点 – 支持向量机

——大选中，您支持谁？

——I Support Vector Machines.