

# AI 岗位基础面试问题

作者：孙峥  
专业：计算机技术  
邮箱：sunzheng2019@ia.ac.cn  
学校：中国科学院大学 (中国科学院)  
学院：人工智能学院 (自动化研究所)

2019 年 11 月 11 日

# Part I

## 基础数学问题

### Question 1

定义矩阵的范数:  $\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}$ ,  $A$  是对称正定阵, 证明  $\|A\|_2 = \lambda$  ( $\lambda$  是  $A$  的最大特征值)。

证: {先说明一些相关的知识点: 矩阵范数定义的时候, 有非负性, 绝对齐性, 三角不等式, 还比向量范数多一个相容性。然后引入矩阵的  $F$  范数,  $\|A\|_F^2 = \sum_{i,j=1} a_{ij}^2 = \text{tr}(A^T A)$ , 可以验证矩阵的  $F$  范数是矩阵范数。再引入矩阵的  $p$  范数,  $\|A\|_p = \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p} = \max_{\|x\|_p=1} \|Ax\|_p$ , 容易证明这样定义的也是矩阵范数。由于是向量的  $p$  范数导出的矩阵的  $p$  范数, 所以此矩阵范数又称为算子范数 (《泛函分析》中有定义)。}

上述说明的矩阵范数有以下两个重要性质: (1) 矩阵的  $F$  范数和  $2-$  范数都与向量的  $2-$  范数相容; (2) 所定义的算子范数, 即  $p-$  范数都与向量的  $p-$  范数相容; (3) 任一矩阵范数, 一定存在与之相容的向量范数。下面开始证明这道题, 网上可以查找到的证明过程都非常复杂, 需要  $A \geq B, A \leq B$ , 然后导出  $A = B$  的过程, 此处提供一种相对简单的方法, 是我在本科时候的《数值分析》课上由林丹老师讲授。}

假设  $A$  是一般矩阵,  $A^T A$  是对称半正定矩阵, 则  $\exists$  正交矩阵  $Q, s.t.$

$$A^T A = Q^T \Lambda Q, \Lambda = \text{diag}(\lambda_1, \lambda_2, \dots, \lambda_n), \lambda_i \geq 0$$

且有:

$$\|A\|_2^2 = (Ax)^T (Ax) = x^T A^T A x = x^T Q^T \Lambda Q x = (Qx)^T \Lambda (Qx)$$

由于  $Q$  正交, 且  $\|x\|_2 = 1$ , 有  $\|Qx\|_2 = 1$ , 则:

$$\begin{aligned} \|A\|_2^2 &= \max_{\|x\|_2=1} \|Ax\|_2^2 \\ &= \max_{\|x\|_2=1} (Qx)^T \Lambda (Qx) \\ &= \max_{\|y\|_2=1} (y)^T \Lambda (y) \\ &= \max_{\|y\|_2=1} \sum_{i=1}^n y_i^2 \lambda_i \\ &= \lambda_1 \end{aligned}$$

当  $A$  是对称正定阵时, 特征值均大于 0。 $A^T A$  可以视为  $f(A)g(A)$ , 其特征值的最大值为  $\lambda_1^2$ ,  $\lambda_1$  是  $A$  特征值的最大值, 证毕。

- (1) 证明过程中用到了正交矩阵不改变向量或矩阵的  $2-$  范数的性质。假设  $P, Q$  均为正交矩阵, 则  $\|A\|_2 = \|PA\|_2 = \|AQ\|_2 = \|PAQ\|_2$ , 但是会改变  $1-$  范数;
- (2) 除了矩阵的  $2-$  范数, 还有  $1-$  范数和  $\infty$  范数, 计算结果可以用‘一列无穷行’记忆。

## Question 2

设  $X = \{x_1, x_2, \dots, x_n\}$ , iid 服从  $U(0, k)$  的均匀分布, 求  $k$  的极大似然估计。

解: {求解极大似然估计, 应该先写出极大似然函数  $\ln(L(\theta))$ , 再对参数  $\theta$  求导即可, 必要时需要验证二阶导。}

$$f(X) = \frac{1}{k^n}, 0 \leq x_i \leq k.$$
$$\ln L(k) = -n \ln k, \ln L(k)' = -\frac{n}{k} < 0.$$

不存在  $k$  的极大似然估计。

## Part II

# 计算机算法设计与分析

首先介绍分治思想, 求解问题的大概流程如下:

Q1: 从最简单的 *case* 入手;

Q2: 复杂问题, 分解为 *sub-problems*。

如何分解:

1. 看 *Input*: 输入的关键数据结构 (DS, 包括数组、树、有向无环图、图、集合), 决定是否可分;
2. 看 *Output*: 决定能否把解合起来。

## Question 1

用时间复杂度尽可能少的算法来排序一个  $n$  个整数的数组。

解: (1) 首先想到的是利用冒泡排序, 利用两个 for 循环来排序数组, 这种方法的时间复杂度是  $O(n^2)$ , 代码较简单, 没有递归调用, 略去;

(2) 采用 DC(divide and conquer) 思想, 每次递归调用数组  $[0, n]$  的前  $n - 1$  个元素, 再回溯合并, 大致过程如下图所示 (选自卜东波老师上课的 slides)。

合并的时候将末尾的第  $n$  个元素插入前  $n - 1$  个元素当中, 时间复杂度为  $O(n)$ , 所以有迭代式:  
 $T(n) = T(n - 1) + O(n)$ , 简单推导:

$$\begin{aligned} T(n) &\leq T(n - 1) + cn \\ &\leq T(n - 2) + c(n - 1) + cn \\ &\leq \dots \\ &\leq c(1 + 2 + 3 + \dots + n) \\ &= O(n^2) \end{aligned}$$

代码相对简单, 略去;

(3) 和 (2) 中方法的分治一样, 按照下标来分治, 此时分治从该数组的中心位置一分为二, 分别对两个子问题排序, 分别排好序之后再回溯合并, 大致过程如图所示 (选自卜东波老师上课的 slides), 这实际上就是归并排序 (二路归并)。归并的过程可以简单描述为: 先准备一个数组, 数组容量是两个子问题的

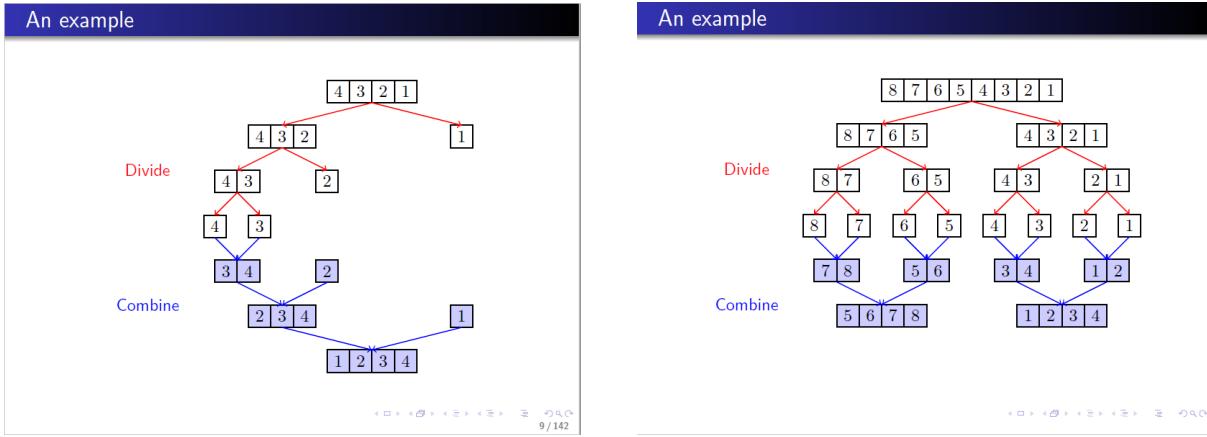


图 1：采用分治思想排序

规模之和，比较  $a[i]$  和  $b[j]$  的大小，若  $a[i] \leq b[j]$ ，则将第一个有序表中的元素  $a[i]$  复制到  $r[k]$  中，并令  $i$  和  $k$  分别加上 1；否则将第二个有序表中的元素  $b[j]$  复制到  $r[k]$  中，并令  $j$  和  $k$  分别加上 1；如此循环下去，直到其中一个有序表取完；然后再将另一个有序表中剩余的元素复制到  $r$  中从下标  $k$  到最后的单元，大致过程如下（参考 <https://blog.csdn.net/daigualu/article/details/78399168>）。介绍完方法，下面给出实际的可运行代码（C++，在文件夹 code/MergeOrder 中），利用分治和归并排序的思想来排序某一数组，其中的数组规模和元素是自行输入，更加灵活。

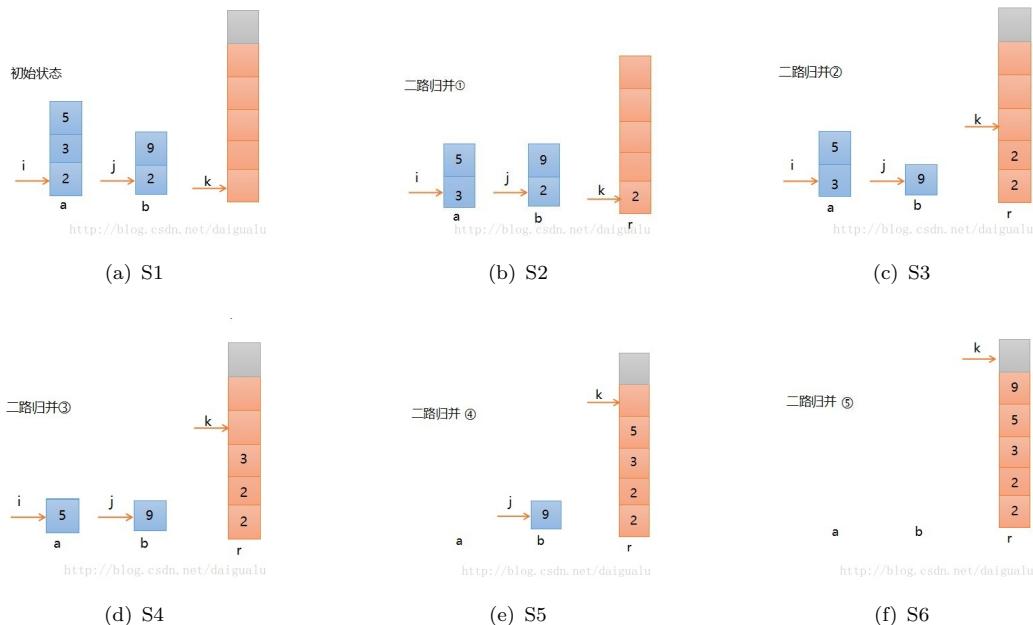


图 2：二路归并过程

```

1 #include <iostream>
2 #include <stdio.h>
3

```

```

5 using namespace std;
7 long int merge(int a[], int left, int mid, int right,int b[])
{
9     int i = mid;
10    int j = right;
11    int k = 0;
12    while (i >= left && j >= mid+1)
13    {
14        if(a[i] > a[j])
15        {
16            b[k++] = a[i--];
17        }
18        else
19        {
20            b[k++] = a[j--];
21        }
22    }
23    while (i >= left)
24    {
25        b[k++] = a[i--];
26    }
27    while (j >= mid+1)
28    {
29        b[k++] = a[j--];
30    }
31    for (i = 0; i < k; i++)
32    {
33        a[right - i] = b[i];
34    }
35 }

36 long int solve(int a[],int left , int right,int b[])
{
37     if(right > left)
38     {
39         int mid = (right+left) / 2;
40         solve(a,left , mid,b);
41         solve(a,mid + 1, right ,b);
42         merge(a,left , mid, right ,b);
43     }
44 }
45 }

46 int main()
47 {
48     long int n;//数组维度
49     scanf("%d", &n);
50     int *a = new int[n];
51     int *b = new int[n];
52     for(long int i=0;i<n;i++)
53     {
54         scanf("%d", &a[i]); //scanf的速度要比cin的速度快
55     }
56     solve(a,0 ,n-1,b); //归并排序
57
58     for(int i = 0;i<n;i++)
59         cout<<a[i]<<' ';
60     return 0;
61 }

```

Listing 1: 归并排序,C++

上述的代码过程中，两个子问题的归并实际上是从后向前的归并，下面给出从前向后的归并过程，二者本质一样。(但是不知道为什么下面这个代码无法完成排序？)

```
#include <iostream>
#include <stdio.h>

using namespace std;

long int merge(int a[], int left, int mid, int right, int b[])
{
    int i = left;
    int j = mid+1;
    int k = 0;
    while (i <= mid && j <= right)
    {
        if (a[i] > a[j])
        {
            b[k++] = a[i++];
        }
        else
        {
            b[k++] = a[j++];
        }
    }
    while (i <= mid)
    {
        b[k++] = a[i++];
    }
    while (j <= right)
    {
        b[k++] = a[j++];
    }
    for (i = 0; i < k; i++)
    {
        a[right - i] = b[i];
    }
}

long int solve(int a[], int left, int right, int b[])
{
    if (right > left)
    {
        int mid = (right+left) / 2;
        solve(a, left, mid, b);
        solve(a, mid + 1, right, b);
        merge(a, left, mid, right, b);
    }
}

int main()
{
    long int n;//数组维度
    scanf("%d", &n);
    int *a = new int[n];
    int *b = new int[n];
    for (long int i=0;i<n;i++)
    {
        scanf("%d", &a[i]); //scanf的速度要比cin的速度快
    }

    solve(a, 0, n-1, b); //归并排序
    for (int i = 0; i < n; i++)
    {
```

```
62     cout<<a[i]<< ' ';
63     return 0;
64 }
```

## Question 2

C++ 中输入二维 (多维) 数组的方法。(这不是个具体的问题，只是为了面试要求手写代码的时候可参考)。

1. 使用 C++ 中的 *vector* 数据结构，*vector* 是一个动态数组结构，可以在其中添加或删除元素。在头文件中声明 `#include <vector>`，定义一维数组 `vector < int > a;`，定义二维数组 `vector < vector < int > > a;`，注意最后两个尖括号之间应该有个空格，使用方法如下：

(1) 数组规模较小时使用；

```
1 vector<vector<int> >vec;
2 vector<int>a;
3 a.push_back(1);
4 a.push_back(2);
5 vector<int>b;
6 b.push_back(3);
7 b.push_back(4);
8 vec.push_back(a);
9 vec.push_back(b);
```

(2) 数组规模较大，且不需要自行输入；

```
1 vector<vector<int> >arry(6); // 先确定数组的行数
2 for(int i=0;i<arry.size();i++)
3     arry[i].resize(8); // 确定每行的列数
4
5 for(int i=0;i<arry.size();i++)
6     for(int j=0;j<arry[0].size();j++)
7         arry[i][j]=i*j;
```

(3) 数组规模较大，且需要自行输入数组元素；

```
1 int m,n;
2 cin>>m>>n; // 输入时可以中间可以加空格
3 vector<vector<int> >arry;
4 for(int i=0;i<arry.size();i++)
5     for(int j=0;j<arry[0].size();j++)
6         cin>>arry[i][j]; // 输入时每行之间可以回车
```

2. 利用指针生成二维数组，数组名是实际上是一个指针，使用指针来分配指针，使用方法如下：

(1) 一维数组：

```
1 int arraysize; // 数组规模
2 scanf("%d",&arraysize); // 输入数组规模，scanf比cin快很多
3 int *arry=new int [arraysize]; // 数组名是指针
4 for(int count=0;count<arraysize;count++)
    scanf("%d",&arry[count]);
```

## (2) 二维数组

```
1 int row,col;
2 scanf("%d %d",&row,&col);
3 int **arry=new int*[row];//指向指针的指针，申请row个指向int*的指针
4 for(int i=0;i<row;i++)
5 {
6     arry[i]=new int [col];//arry每个元素都是指针
7     for(int j=0;j<col;j++)
8         scanf("%d",&arry[i][j]);
9 }
```

以上的代码在输入元素时，用的都是 `scanf` 函数，需要声明头文件 `#include <stdio.h>`。使用 `scanf` 函数要比 `cin` 快很多，在很多 OJ 题当中，当自己的算法时间不通过时，可以通过更换输入函数来使代码通过（个人经验）。

## Question 3

利用问题 1 和问题 2 中的方法，来解决数组逆序数计算问题（包括数组显著逆序数计算问题）。

解：这是第 1 题第 (3) 种方法归并排序的引用。计算（显著）逆序数：可以在归并排序一个数组时，进行（显著）逆序数的计算，显著逆序数就是  $a_i > k * a_j, i < j$ ，网上找到的逆序数计算的代码和显著逆序数的可能不一样。此处把逆序数的计算也当成显著逆序数的一种来统一计算。代码如下：

```
1 #include <iostream>
2 #include <stdio.h>
3 using namespace std;
4
5 long int merge(int a[], int left, int mid, int right,int b[])
6 {
7     int i = mid;
8     int j = right;
9     long int lcount = 0;
10    while (i >= left && j > mid)
11    {
12        if(a[i] > (long long) 3 * a[j])
13        {
14            lcount += j - mid;
15            i--;
16        }
17        else
18        {
19            j--;
20        }
21    }
22    i = mid;
23    j = right;
24    int k = 0;
25    while (i >= left && j > mid)
26    {
27        if(a[i] > a[j])
28        {
29            b[k++] = a[i--];
30        }
31        else
32        {
33            b[k++] = a[j--];
34        }
35    }
36 }
```

```

37     while ( i >= left )
38     {
39         b [k++] = a [i--];
40     }
41     while ( j > mid )
42     {
43         b [k++] = a [j--];
44     }
45     for ( i = 0; i < k; i++ )
46     {
47         a [right - i] = b [i];
48     }
49     return lcount;
50 }

51 long int solve(int a[],int left , int right,int b[])
52 {
53     long int cnt = 0;
54     if(right > left)
55     {
56         int mid = (right+left) / 2;
57         cnt += solve(a, left , mid,b);
58         cnt += solve(a, mid + 1, right ,b);
59         cnt += merge(a, left , mid, right ,b);
60     }
61     return cnt;
62 }

63 long int InversePairs(int a[],int len)
64 {
65     int *b=new int[len];
66     long int count=solve(a,0,len-1,b);
67     delete [] b;
68     return count;
69 }

70 int main()
71 {
72     long int n;//数组维度
73     int *array;//数组
74     scanf("%d", &n);
75     array = new int[n];
76     for(long int i=0;i<n;i++)
77         scanf("%d", &array[i]);

78     long int count = InversePairs(array , n);
79     printf("%d", count);
80     return 0;
81 }

```

代码跟归并排序的过程差不多，不同的是在 *merge* 函数中，进行归并排序之前会计算两个子数组之间的显著逆序数个数（两个子数组内的显著逆序数由于递归调用已经计算完毕），就是 *merge* 函数中的第一个 *while* 循环，计算过后要将 *i,j* 设置成原来的值，再进行排序。此处应注意的是，网上关于逆序数（不是显著逆序数）的计算是边排序边计算逆序数，二者是一起的，原因就是顺序规则跟计算逆序数的规则是一致的。所以要计算逆序数，除了把上述代码中 *merge* 函数中第一个 *while* 循环里的 3 改成 1 之外，还可以在排序的同时计算逆序数，由于计算逆序数的代码网上可以很容易找到，此处略去。

## Question 4

### 快速排序算法

解：与第 1 题按照数组的下标来分治不同，这道题按照数组的值来分治，即选取 *pivot* 来排序一个数组。

## Question 5

### 计算分治问题时间复杂度的总结，主定理 (Master theorem)

解：第 1 题第 (3) 种方法和第 4 题分别介绍了排序一个数组的方法，一个是按照数组的下标分治，一个是按照数组的值来分治。但是没有分析两种方法的时间复杂度，下面给出一般的分治问题的复杂度分析方法。

假设某个问题的规模为  $n$ ，分成  $a$  个子问题，每个子问题的规模为  $\frac{n}{b}$ （这里的  $a, b$  不一定相等，因为子问题往往有重叠的部分，所以  $a \geq b$ ），有递推式： $T(n) = aT\left(\frac{n}{b}\right) + O(d^n)$ 。

结论见下图（卜东波老师的课上 slides）。

Theorem

Let  $T(n)$  be defined by  $T(n) = aT\left(\frac{n}{b}\right) + O(n^d)$  for  $a > 1$ ,  $b > 1$  and  $d > 0$ , then  $T(n)$  can be bounded by:

- ① If  $d < \log_b a$ , then  $T(n) = O(n^{\log_b a})$ ;
- ② If  $d = \log_b a$ , then  $T(n) = O(n^{\log_b a} \log n)$ ;
- ③ If  $d > \log_b a$ , then  $T(n) = O(n^d)$ .

图 3: Master theorem

计算该递推式：

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + O(d^n) \\
&\leq aT\left(\frac{n}{b}\right) + cn^d \\
&\leq a[aT\left(\frac{n}{b^2}\right) + c\left(\frac{n}{b}\right)^n] + cn^d \\
&\leq \dots \\
&\leq cn^d + ac\left(\frac{n}{b}\right)^d + a^2c\left(\frac{n}{b^2}\right)^d + \dots + a^{\log_b n - 1}c\left(\frac{n}{b^{\log_b n - 1}}\right)^d + a\log_b n \\
&\leq cn^d[1 + \frac{a}{b^d} + (\frac{a}{b^d})^2 + \dots + (\frac{a}{b^d})^{\log_b n - 1}] + a^{\log_b n}
\end{aligned}$$

对上式中的等比项分类讨论：

(1)  $a < b^d$ , 即  $d > \log_b a$ , 以指数项的第一项计算, 则:

$$\begin{aligned}
T(n) &\leq cn^d + a^{\log_b n} \\
&= cn^d + n^{\log_b a} \\
&= O(n^d)
\end{aligned}$$

(2)  $a = b^d$ , 即  $d = \log_b a$ , 所有的指数项都要计算, 则:

$$\begin{aligned}
T(n) &\leq cn^d \log_b n + a^{\log_b n} \\
&= cn^{\log_b a} \log_b n + a^{\log_b n} \\
&= O(cn^{\log_b a} \log_b n) \\
&= O(n^{\log_b a} \log n)
\end{aligned}$$

(3)  $a > b^d$ , 即  $d < \log_b a$ , 以指数项的最后一项计算, 则:

$$\begin{aligned}
T(n) &\leq cn^d \left(\frac{a}{b^d}\right)^{\log_b n - 1} + a^{\log_b n} \\
&= \frac{cn^d}{\frac{a}{b^d}} n^{\log_b \frac{a}{b^d}} + n^{\log_b a} \\
&= \frac{c}{a} (nb)^d n^{\log_b \frac{a}{b^d}} + n^{\log_b a} \\
&= \frac{cb^d}{a} n^d n^{\log_b a - \log_b b^d} + n^{\log_b a} \\
&= \frac{cb^d}{a} n^d \frac{n^{\log_b a}}{n^d} + n^{\log_b a} \\
&= \frac{cb^d}{a} n^{\log_b a} + n^{\log_b a} \\
&= O(n^{\log_b a})
\end{aligned}$$

命题证毕。

## Question 6

用时间复杂度尽可能少的算法找出一个数组中第  $k$  个小的元素

解：首先想到的是对数组进行排序，即可以找出数组中第  $k$  小的元素。上述已经介绍  $n$  个元素的数组最快的排序算法（归并排序和快速排序）时间复杂度为  $O(n \log n)$ ，此处介绍更快的解决此问题的算法。

## Question 7

leetcode 第 33 题，搜索旋转排序数组。问题描述：

## Question 7

leetcode 第 153 题，寻找旋转排序数组中的最小值。问题描述：

## Question 9

leetcode 第题，寻找一个数组的众数。问题描述：

## Question 10

leetcode 第 200 题，计算岛屿的个数。问题描述：

这道题实际上是计算连通域的个数，可以用 DFS 和 BFS 求解。

## Question 11

二叉树的构建（递归与非递归方法），先序中序后序遍历（递归与非递归方法），叶子节点计数，深度计算等，面试时手写代码用得上

先介绍递归方法：

```
1 #include <iostream>
2 #include <stdio.h>
3
4 using namespace std;
5
6 // 定义二叉树结构体
7 struct TreeNode{
8     int data;
9     TreeNode *lchild;
10    TreeNode *rchild;
11 };
12
13 /*
14 // 先序创建二叉树
15 void CreateBiTree(TreeNode *T)
16 {
17     int ch;
18     cin >> ch;
19     if (ch == -1)
```

```

21     {
22         T = NULL;
23         return;
24     }
25     else
26     {
27         T = new TreeNode;
28         T->data = ch;//
29         cout << "input" << ch << "'s left son node:";
30         CreateBiTree(T->lchild);//
31         cout << "input" << ch << "'s right son node:";
32         CreateBiTree(T->rchild);//
33     }
34     return;
35 }
36 /*/
37 //上面这段创建树的代码创建的树无法使用,跟下面的相比少了指针
38
39 //先序创建二叉树
40 void CreateBiTree(TreeNode **T)
41 {
42     int ch;
43     cin >> ch;
44     if (ch == -1)
45     {
46         *T = NULL;
47         return;
48     }
49     else
50     {
51         *T = new TreeNode;
52         (*T)->data = ch;
53         cout << "input" << ch << "'s left son node:";
54         CreateBiTree(&((*T)->lchild));
55         cout << "input" << ch << "'s right son node:";
56         CreateBiTree(&((*T)->rchild));
57     }
58     return;
59 }
60
61 //先序遍历
62 void PreOrderBiTree(TreeNode *T)
63 {
64     if (T == NULL)
65     {
66         return;
67     }
68     else
69     {
70         cout << T->data << " ";
71         PreOrderBiTree(T->lchild);
72         PreOrderBiTree(T->rchild);
73     }
74 }
75
76 //中序遍历
77 void MiddleOrderBiTree(TreeNode *T)
78 {
79     if (T == NULL)
80     {
81         return;
82     }
83     else
84     {
85         MiddleOrderBiTree(T->lchild);
86     }
87 }
88
89 //后序遍历
90 void PostOrderBiTree(TreeNode *T)
91 {
92     if (T == NULL)
93     {
94         return;
95     }
96     else
97     {
98         PostOrderBiTree(T->lchild);
99         PostOrderBiTree(T->rchild);
100        cout << T->data << " ";
101    }
102 }
103
104 //层序遍历
105 void LevelOrderBiTree(TreeNode *T)
106 {
107     if (T == NULL)
108     {
109         return;
110     }
111     else
112     {
113         cout << T->data << " ";
114         LevelOrderBiTree(T->lchild);
115         LevelOrderBiTree(T->rchild);
116     }
117 }
118
119 //广度优先遍历
120 void BreadthFirstSearchBiTree(TreeNode *T)
121 {
122     if (T == NULL)
123     {
124         return;
125     }
126     else
127     {
128         cout << T->data << " ";
129         BreadthFirstSearchBiTree(T->lchild);
130         BreadthFirstSearchBiTree(T->rchild);
131     }
132 }
133
134 //深度优先遍历
135 void DepthFirstSearchBiTree(TreeNode *T)
136 {
137     if (T == NULL)
138     {
139         return;
140     }
141     else
142     {
143         cout << T->data << " ";
144         DepthFirstSearchBiTree(T->lchild);
145         DepthFirstSearchBiTree(T->rchild);
146     }
147 }
148
149 //求二叉树的深度
150 int DepthBiTree(TreeNode *T)
151 {
152     if (T == NULL)
153     {
154         return 0;
155     }
156     else
157     {
158         int lDepth = DepthBiTree(T->lchild);
159         int rDepth = DepthBiTree(T->rchild);
160         if (lDepth > rDepth)
161             return lDepth + 1;
162         else
163             return rDepth + 1;
164     }
165 }
166
167 //求二叉树的节点数
168 int NodeCountBiTree(TreeNode *T)
169 {
170     if (T == NULL)
171     {
172         return 0;
173     }
174     else
175     {
176         int lCount = NodeCountBiTree(T->lchild);
177         int rCount = NodeCountBiTree(T->rchild);
178         return lCount + rCount + 1;
179     }
180 }
181
182 //求二叉树的叶子数
183 int LeafCountBiTree(TreeNode *T)
184 {
185     if (T == NULL)
186     {
187         return 0;
188     }
189     else
190     {
191         int lCount = LeafCountBiTree(T->lchild);
192         int rCount = LeafCountBiTree(T->rchild);
193         if (T->lchild == NULL && T->rchild == NULL)
194             return 1;
195         else
196             return lCount + rCount;
197     }
198 }
199
200 //求二叉树的分支数
201 int BranchCountBiTree(TreeNode *T)
202 {
203     if (T == NULL)
204     {
205         return 0;
206     }
207     else
208     {
209         int lBranch = BranchCountBiTree(T->lchild);
210         int rBranch = BranchCountBiTree(T->rchild);
211         return lBranch + rBranch + 1;
212     }
213 }
214
215 //求二叉树的高度
216 int HeightBiTree(TreeNode *T)
217 {
218     if (T == NULL)
219     {
220         return 0;
221     }
222     else
223     {
224         int lHeight = HeightBiTree(T->lchild);
225         int rHeight = HeightBiTree(T->rchild);
226         if (lHeight > rHeight)
227             return lHeight + 1;
228         else
229             return rHeight + 1;
230     }
231 }
232
233 //求二叉树的宽度
234 int WidthBiTree(TreeNode *T)
235 {
236     if (T == NULL)
237     {
238         return 0;
239     }
240     else
241     {
242         int lWidth = WidthBiTree(T->lchild);
243         int rWidth = WidthBiTree(T->rchild);
244         if (lWidth > rWidth)
245             return lWidth + 1;
246         else
247             return rWidth + 1;
248     }
249 }
250
251 //求二叉树的平衡因子
252 int BalanceFactorBiTree(TreeNode *T)
253 {
254     if (T == NULL)
255     {
256         return 0;
257     }
258     else
259     {
260         int lDepth = DepthBiTree(T->lchild);
261         int rDepth = DepthBiTree(T->rchild);
262         if (lDepth > rDepth)
263             return lDepth - rDepth;
264         else
265             return rDepth - lDepth;
266     }
267 }
268
269 //求二叉树的平衡性
270 int IsBalancedBiTree(TreeNode *T)
271 {
272     if (T == NULL)
273     {
274         return 1;
275     }
276     else
277     {
278         int lBalance = BalanceFactorBiTree(T->lchild);
279         int rBalance = BalanceFactorBiTree(T->rchild);
280         if (lBalance >= -1 && rBalance <= 1)
281             return 1;
282         else
283             return 0;
284     }
285 }
286
287 //求二叉树的满度
288 int FullDegreeBiTree(TreeNode *T)
289 {
290     if (T == NULL)
291     {
292         return 0;
293     }
294     else
295     {
296         int lFull = FullDegreeBiTree(T->lchild);
297         int rFull = FullDegreeBiTree(T->rchild);
298         if (lFull > rFull)
299             return lFull + 1;
300         else
301             return rFull + 1;
302     }
303 }
304
305 //求二叉树的空度
306 int EmptyDegreeBiTree(TreeNode *T)
307 {
308     if (T == NULL)
309     {
310         return 0;
311     }
312     else
313     {
314         int lEmpty = EmptyDegreeBiTree(T->lchild);
315         int rEmpty = EmptyDegreeBiTree(T->rchild);
316         if (lEmpty > rEmpty)
317             return lEmpty + 1;
318         else
319             return rEmpty + 1;
320     }
321 }
322
323 //求二叉树的完全度
324 int PerfectDegreeBiTree(TreeNode *T)
325 {
326     if (T == NULL)
327     {
328         return 0;
329     }
330     else
331     {
332         int lPerfect = PerfectDegreeBiTree(T->lchild);
333         int rPerfect = PerfectDegreeBiTree(T->rchild);
334         if (lPerfect > rPerfect)
335             return lPerfect + 1;
336         else
337             return rPerfect + 1;
338     }
339 }
340
341 //求二叉树的左满度
342 int LeftFullDegreeBiTree(TreeNode *T)
343 {
344     if (T == NULL)
345     {
346         return 0;
347     }
348     else
349     {
350         int lLeft = LeftFullDegreeBiTree(T->lchild);
351         int rLeft = LeftFullDegreeBiTree(T->rchild);
352         if (lLeft > rLeft)
353             return lLeft + 1;
354         else
355             return rLeft + 1;
356     }
357 }
358
359 //求二叉树的右满度
360 int RightFullDegreeBiTree(TreeNode *T)
361 {
362     if (T == NULL)
363     {
364         return 0;
365     }
366     else
367     {
368         int lRight = RightFullDegreeBiTree(T->lchild);
369         int rRight = RightFullDegreeBiTree(T->rchild);
370         if (lRight > rRight)
371             return lRight + 1;
372         else
373             return rRight + 1;
374     }
375 }
376
377 //求二叉树的左空度
378 int LeftEmptyDegreeBiTree(TreeNode *T)
379 {
380     if (T == NULL)
381     {
382         return 0;
383     }
384     else
385     {
386         int lLeft = LeftEmptyDegreeBiTree(T->lchild);
387         int rLeft = LeftEmptyDegreeBiTree(T->rchild);
388         if (lLeft > rLeft)
389             return lLeft + 1;
390         else
391             return rLeft + 1;
392     }
393 }
394
395 //求二叉树的右空度
396 int RightEmptyDegreeBiTree(TreeNode *T)
397 {
398     if (T == NULL)
399     {
400         return 0;
401     }
402     else
403     {
404         int lRight = RightEmptyDegreeBiTree(T->lchild);
405         int rRight = RightEmptyDegreeBiTree(T->rchild);
406         if (lRight > rRight)
407             return lRight + 1;
408         else
409             return rRight + 1;
410     }
411 }
412
413 //求二叉树的左完全度
414 int LeftPerfectDegreeBiTree(TreeNode *T)
415 {
416     if (T == NULL)
417     {
418         return 0;
419     }
420     else
421     {
422         int lLeft = LeftPerfectDegreeBiTree(T->lchild);
423         int rLeft = LeftPerfectDegreeBiTree(T->rchild);
424         if (lLeft > rLeft)
425             return lLeft + 1;
426         else
427             return rLeft + 1;
428     }
429 }
430
431 //求二叉树的右完全度
432 int RightPerfectDegreeBiTree(TreeNode *T)
433 {
434     if (T == NULL)
435     {
436         return 0;
437     }
438     else
439     {
440         int lRight = RightPerfectDegreeBiTree(T->lchild);
441         int rRight = RightPerfectDegreeBiTree(T->rchild);
442         if (lRight > rRight)
443             return lRight + 1;
444         else
445             return rRight + 1;
446     }
447 }
448
449 //求二叉树的左平衡度
450 int LeftBalanceDegreeBiTree(TreeNode *T)
451 {
452     if (T == NULL)
453     {
454         return 0;
455     }
456     else
457     {
458         int lLeft = LeftBalanceDegreeBiTree(T->lchild);
459         int rLeft = LeftBalanceDegreeBiTree(T->rchild);
460         if (lLeft > rLeft)
461             return lLeft + 1;
462         else
463             return rLeft + 1;
464     }
465 }
466
467 //求二叉树的右平衡度
468 int RightBalanceDegreeBiTree(TreeNode *T)
469 {
470     if (T == NULL)
471     {
472         return 0;
473     }
474     else
475     {
476         int lRight = RightBalanceDegreeBiTree(T->lchild);
477         int rRight = RightBalanceDegreeBiTree(T->rchild);
478         if (lRight > rRight)
479             return lRight + 1;
480         else
481             return rRight + 1;
482     }
483 }
484
485 //求二叉树的左平衡因子
486 int LeftBalanceFactorBiTree(TreeNode *T)
487 {
488     if (T == NULL)
489     {
490         return 0;
491     }
492     else
493     {
494         int lLeft = LeftBalanceFactorBiTree(T->lchild);
495         int rLeft = LeftBalanceFactorBiTree(T->rchild);
496         if (lLeft > rLeft)
497             return lLeft - rLeft;
498         else
499             return rLeft - lLeft;
500     }
501 }
502
503 //求二叉树的右平衡因子
504 int RightBalanceFactorBiTree(TreeNode *T)
505 {
506     if (T == NULL)
507     {
508         return 0;
509     }
510     else
511     {
512         int lRight = RightBalanceFactorBiTree(T->lchild);
513         int rRight = RightBalanceFactorBiTree(T->rchild);
514         if (lRight > rRight)
515             return lRight - rRight;
516         else
517             return rRight - lRight;
518     }
519 }
520
521 //求二叉树的左平衡性
522 int IsLeftBalancedBiTree(TreeNode *T)
523 {
524     if (T == NULL)
525     {
526         return 1;
527     }
528     else
529     {
530         int lBalance = IsLeftBalancedBiTree(T->lchild);
531         int rBalance = IsLeftBalancedBiTree(T->rchild);
532         if (lBalance >= -1 && rBalance <= 1)
533             return 1;
534         else
535             return 0;
536     }
537 }
538
539 //求二叉树的右平衡性
540 int IsRightBalancedBiTree(TreeNode *T)
541 {
542     if (T == NULL)
543     {
544         return 1;
545     }
546     else
547     {
548         int lBalance = IsRightBalancedBiTree(T->lchild);
549         int rBalance = IsRightBalancedBiTree(T->rchild);
550         if (lBalance >= -1 && rBalance <= 1)
551             return 1;
552         else
553             return 0;
554     }
555 }
556
557 //求二叉树的左平衡因子
558 int LeftBalanceFactorBiTree(TreeNode *T)
559 {
560     if (T == NULL)
561     {
562         return 0;
563     }
564     else
565     {
566         int lLeft = LeftBalanceFactorBiTree(T->lchild);
567         int rLeft = LeftBalanceFactorBiTree(T->rchild);
568         if (lLeft > rLeft)
569             return lLeft - rLeft;
570         else
571             return rLeft - lLeft;
572     }
573 }
574
575 //求二叉树的右平衡因子
576 int RightBalanceFactorBiTree(TreeNode *T)
577 {
578     if (T == NULL)
579     {
580         return 0;
581     }
582     else
583     {
584         int lRight = RightBalanceFactorBiTree(T->lchild);
585         int rRight = RightBalanceFactorBiTree(T->rchild);
586         if (lRight > rRight)
587             return lRight - rRight;
588         else
589             return rRight - lRight;
590     }
591 }
592
593 //求二叉树的左平衡性
594 int IsLeftBalancedBiTree(TreeNode *T)
595 {
596     if (T == NULL)
597     {
598         return 1;
599     }
600     else
601     {
602         int lBalance = IsLeftBalancedBiTree(T->lchild);
603         int rBalance = IsLeftBalancedBiTree(T->rchild);
604         if (lBalance >= -1 && rBalance <= 1)
605             return 1;
606         else
607             return 0;
608     }
609 }
610
611 //求二叉树的右平衡性
612 int IsRightBalancedBiTree(TreeNode *T)
613 {
614     if (T == NULL)
615     {
616         return 1;
617     }
618     else
619     {
620         int lBalance = IsRightBalancedBiTree(T->lchild);
621         int rBalance = IsRightBalancedBiTree(T->rchild);
622         if (lBalance >= -1 && rBalance <= 1)
623             return 1;
624         else
625             return 0;
626     }
627 }
628
629 //求二叉树的左平衡因子
630 int LeftBalanceFactorBiTree(TreeNode *T)
631 {
632     if (T == NULL)
633     {
634         return 0;
635     }
636     else
637     {
638         int lLeft = LeftBalanceFactorBiTree(T->lchild);
639         int rLeft = LeftBalanceFactorBiTree(T->rchild);
640         if (lLeft > rLeft)
641             return lLeft - rLeft;
642         else
643             return rLeft - lLeft;
644     }
645 }
646
647 //求二叉树的右平衡因子
648 int RightBalanceFactorBiTree(TreeNode *T)
649 {
650     if (T == NULL)
651     {
652         return 0;
653     }
654     else
655     {
656         int lRight = RightBalanceFactorBiTree(T->lchild);
657         int rRight = RightBalanceFactorBiTree(T->rchild);
658         if (lRight > rRight)
659             return lRight - rRight;
660         else
661             return rRight - lRight;
662     }
663 }
664
665 //求二叉树的左平衡性
666 int IsLeftBalancedBiTree(TreeNode *T)
667 {
668     if (T == NULL)
669     {
670         return 1;
671     }
672     else
673     {
674         int lBalance = IsLeftBalancedBiTree(T->lchild);
675         int rBalance = IsLeftBalancedBiTree(T->rchild);
676         if (lBalance >= -1 && rBalance <= 1)
677             return 1;
678         else
679             return 0;
680     }
681 }
682
683 //求二叉树的右平衡性
684 int IsRightBalancedBiTree(TreeNode *T)
685 {
686     if (T == NULL)
687     {
688         return 1;
689     }
690     else
691     {
692         int lBalance = IsRightBalancedBiTree(T->lchild);
693         int rBalance = IsRightBalancedBiTree(T->rchild);
694         if (lBalance >= -1 && rBalance <= 1)
695             return 1;
696         else
697             return 0;
698     }
699 }
700
701 //求二叉树的左平衡因子
702 int LeftBalanceFactorBiTree(TreeNode *T)
703 {
704     if (T == NULL)
705     {
706         return 0;
707     }
708     else
709     {
710         int lLeft = LeftBalanceFactorBiTree(T->lchild);
711         int rLeft = LeftBalanceFactorBiTree(T->rchild);
712         if (lLeft > rLeft)
713             return lLeft - rLeft;
714         else
715             return rLeft - lLeft;
716     }
717 }
718
719 //求二叉树的右平衡因子
720 int RightBalanceFactorBiTree(TreeNode *T)
721 {
722     if (T == NULL)
723     {
724         return 0;
725     }
726     else
727     {
728         int lRight = RightBalanceFactorBiTree(T->lchild);
729         int rRight = RightBalanceFactorBiTree(T->rchild);
730         if (lRight > rRight)
731             return lRight - rRight;
732         else
733             return rRight - lRight;
734     }
735 }
736
737 //求二叉树的左平衡性
738 int IsLeftBalancedBiTree(TreeNode *T)
739 {
740     if (T == NULL)
741     {
742         return 1;
743     }
744     else
745     {
746         int lBalance = IsLeftBalancedBiTree(T->lchild);
747         int rBalance = IsLeftBalancedBiTree(T->rchild);
748         if (lBalance >= -1 && rBalance <= 1)
749             return 1;
750         else
751             return 0;
752     }
753 }
754
755 //求二叉树的右平衡性
756 int IsRightBalancedBiTree(TreeNode *T)
757 {
758     if (T == NULL)
759     {
760         return 1;
761     }
762     else
763     {
764         int lBalance = IsRightBalancedBiTree(T->lchild);
765         int rBalance = IsRightBalancedBiTree(T->rchild);
766         if (lBalance >= -1 && rBalance <= 1)
767             return 1;
768         else
769             return 0;
770     }
771 }
772
773 //求二叉树的左平衡因子
774 int LeftBalanceFactorBiTree(TreeNode *T)
775 {
776     if (T == NULL)
777     {
778         return 0;
779     }
780     else
781     {
782         int lLeft = LeftBalanceFactorBiTree(T->lchild);
783         int rLeft = LeftBalanceFactorBiTree(T->rchild);
784         if (lLeft > rLeft)
785             return lLeft - rLeft;
786         else
787             return rLeft - lLeft;
788     }
789 }
790
791 //求二叉树的右平衡因子
792 int RightBalanceFactorBiTree(TreeNode *T)
793 {
794     if (T == NULL)
795     {
796         return 0;
797     }
798     else
799     {
800         int lRight = RightBalanceFactorBiTree(T->lchild);
801         int rRight = RightBalanceFactorBiTree(T->rchild);
802         if (lRight > rRight)
803             return lRight - rRight;
804         else
805             return rRight - lRight;
806     }
807 }
808
809 //求二叉树的左平衡性
810 int IsLeftBalancedBiTree(TreeNode *T)
811 {
812     if (T == NULL)
813     {
814         return 1;
815     }
816     else
817     {
818         int lBalance = IsLeftBalancedBiTree(T->lchild);
819         int rBalance = IsLeftBalancedBiTree(T->rchild);
820         if (lBalance >= -1 && rBalance <= 1)
821             return 1;
822         else
823             return 0;
824     }
825 }
826
827 //求二叉树的右平衡性
828 int IsRightBalancedBiTree(TreeNode *T)
829 {
830     if (T == NULL)
831     {
832         return 1;
833     }
834     else
835     {
836         int lBalance = IsRightBalancedBiTree(T->lchild);
837         int rBalance = IsRightBalancedBiTree(T->rchild);
838         if (lBalance >= -1 && rBalance <= 1)
839             return 1;
840         else
841             return 0;
842     }
843 }
844
845 //求二叉树的左平衡因子
846 int LeftBalanceFactorBiTree(TreeNode *T)
847 {
848     if (T == NULL)
849     {
850         return 0;
851     }
852     else
853     {
854         int lLeft = LeftBalanceFactorBiTree(T->lchild);
855         int rLeft = LeftBalanceFactorBiTree(T->rchild);
856         if (lLeft > rLeft)
857             return lLeft - rLeft;
858         else
859             return rLeft - lLeft;
860     }
861 }
862
863 //求二叉树的右平衡因子
864 int RightBalanceFactorBiTree(TreeNode *T)
865 {
866     if (T == NULL)
867     {
868         return 0;
869     }
870     else
871     {
872         int lRight = RightBalanceFactorBiTree(T->lchild);
873         int rRight = RightBalanceFactorBiTree(T->rchild);
874         if (lRight > rRight)
875             return lRight - rRight;
876         else
877             return rRight - lRight;
878     }
879 }
880
881 //求二叉树的左平衡性
882 int IsLeftBalancedBiTree(TreeNode *T)
883 {
884     if (T == NULL)
885     {
886         return 1;
887     }
888     else
889     {
890         int lBalance = IsLeftBalancedBiTree(T->lchild);
891         int rBalance = IsLeftBalancedBiTree(T->rchild);
892         if (lBalance >= -1 && rBalance <= 1)
893             return 1;
894         else
895             return 0;
896     }
897 }
898
899 //求二叉树的右平衡性
900 int IsRightBalancedBiTree(TreeNode *T)
901 {
902     if (T == NULL)
903     {
904         return 1;
905     }
906     else
907     {
908         int lBalance = IsRightBalancedBiTree(T->lchild);
909         int rBalance = IsRightBalancedBiTree(T->rchild);
910         if (lBalance >= -1 && rBalance <= 1)
911             return 1;
912         else
913             return 0;
914     }
915 }
916
917 //求二叉树的左平衡因子
918 int LeftBalanceFactorBiTree(TreeNode *T)
919 {
920     if (T == NULL)
921     {
922         return 0;
923     }
924     else
925     {
926         int lLeft = LeftBalanceFactorBiTree(T->lchild);
927         int rLeft = LeftBalanceFactorBiTree(T->rchild);
928         if (lLeft > rLeft)
929             return lLeft - rLeft;
930         else
931             return rLeft - lLeft;
932     }
933 }
934
935 //求二叉树的右平衡因子
936 int RightBalanceFactorBiTree(TreeNode *T)
937 {
938     if (T == NULL)
939     {
940         return 0;
941     }
942     else
943     {
944         int lRight = RightBalanceFactorBiTree(T->lchild);
945         int rRight = RightBalanceFactorBiTree(T->rchild);
946         if (lRight > rRight)
947             return lRight - rRight;
948         else
949             return rRight - lRight;
950     }
951 }
952
953 //求二叉树的左平衡性
954 int IsLeftBalancedBiTree(TreeNode *T)
955 {
956     if (T == NULL)
957     {
958         return 1;
959     }
960     else
961     {
962         int lBalance = IsLeftBalancedBiTree(T->lchild);
963         int rBalance = IsLeftBalancedBiTree(T->rchild);
964         if (lBalance >= -1 && rBalance <= 1)
965             return 1;
966         else
967             return 0;
968     }
969 }
970
971 //求二叉树的右平衡性
972 int
```

```

85     cout << T->data << " ";
86     MiddleOrderBiTree(T->rchild);
87 }
88
89 //后序遍历
90 void PostOrderBiTree(TreeNode *T)
91 {
92     if (T == NULL)
93     {
94         return;
95     }
96     else
97     {
98         PostOrderBiTree(T->lchild);
99         PostOrderBiTree(T->rchild);
100        cout << T->data << " ";
101    }
102}
103
104//树的深度
105 int TreeDeep(TreeNode *T)
106{
107     int deep = 0;
108     if (T != NULL)
109     {
110         int leftdeep = TreeDeep(T->lchild);
111         int rightdeep = TreeDeep(T->rchild);
112         deep = leftdeep >= rightdeep?leftdeep+1:rightdeep+1;
113     }
114     return deep;
115}
116
117//叶子节点个数
118 int LeafCount(TreeNode *T)
119{
120     static int count;
121     if (T != NULL)
122     {
123         if (T->lchild == NULL && T->rchild == NULL)
124         {
125             count++;
126         }
127         LeafCount(T->lchild);
128         LeafCount(T->rchild);
129     }
130     return count;
131}
132
133 int main()
134{
135     cout << "输入第一个节点的值,-1表示没有儿子节点:" << endl;
136     TreeNode* T;
137     CreateBiTree(&T); //加个引用创建之后，T会变化；如果使用上面的，调用函数之后T仍然是空的
138     cout << T->data << endl;
139     system("pause");
140     return 0;
141}

```

上面给出了两种构建二叉树的写法，第一种构建的二叉树是无法使用的，要注意指针和引用，原理与下面这段代码相同。

```

1 #include <iostream>
2 #include <stdio.h>
3
4 using namespace std;
5
6 void f1(int point)
7 {
8     point = 1;
9 }
10
11 void f2(int * point)
12 {
13     *point = 1;
14 }
15
16 int main()
17 {
18     int a = 0;
19     f1(a);
20     cout<<a<<endl;
21     f2(&a);
22     cout<<a<<endl;
23     system("pause");
24     return 0;
25 }
```

## Question 12

寻找二叉树上最远的两个节点的距离。问题描述：

介绍完分治思想部分的题，接下来的题将会面向动态规划。流程大致如下：

Q1: 从最简单的 case 入手；

Q2: 对大的 case 分解。

如何分解？这实际上是一个多步决策的过程：

1. 解能否逐步构造出来 (类似于分治思想中的数据结构和解能否可分)；
2. 目标函数能够分解 (与分治思想不同，分治没有目标函数)。

动态规划快的原因是：问题定义可能是指数级多的 case 找最优，DP 可以去除冗余，这一点在具体的问题中会体现。

## Question 13

矩阵乘法。问题描述： $n$  个矩阵  $A_1, A_2, A_3, \dots, A_n$  相乘，矩阵  $A_i$  的规模为  $p_{i-1} * p_i$ ，确定最优的运算顺序（结合律），使得整体运算次数最少（只看乘法，不看加法）。

解： $n$  个矩阵  $A_1, A_2, A_3, \dots, A_n$  相乘，有卡特兰数种运算情况 ( $G(n) = G(1)G(n-1) + G(2)G(n-2) + \dots + G(n-1)G(1)$ )。动态规划求解，即采用多步决策，设  $OPT(i, j)$  表示  $A_i A_{i+1} \dots A_{j-1} A_j$  的最优运算次数。假设从第  $k$  个位置一分为二，即  $(A_i, A_{i+1}, A_{i+2}, \dots, A_k)(A_{k+1}, A_{k+2}, \dots, A_j)$ ，则有递推式：

$$OPT(i, j) = OPT(i, k) + OPT(k+1, j) + p_{i-1} p_k p_j$$

如何选择中间的位置是一个枚举过程，对于每一个位置都要递归地去调用分成的两部分，然后每一部分再进行枚举过程，以此类推，伪代码如下：

## Trial 1: Explore the recursion in the top-down manner

```
RECURSIVE_MATRIX_CHAIN( $i, j$ )
1: if  $i == j$  then
2:   return 0;
3: end if
4:  $OPT(i, j) = +\infty;$ 
5: for  $k = i$  to  $j - 1$  do
6:    $q = RECURSIVE\_MATRIX\_CHAIN(i, k)$ 
7:   +  $RECURSIVE\_MATRIX\_CHAIN(k + 1, j)$ 
8:   +  $p_{i-1} p_k p_j$ ;
9:   if  $q < OPT(i, j)$  then
10:     $OPT(i, j) = q$ ;
11:   end if
12: end for
13: return  $OPT(i, j)$ ;
```

- Note: The optimal solution to the original problem can be obtained through calling  $RECURSIVE\_MATRIX\_CHAIN(1, n)$ .

图 4: trial 1

此算法的时间复杂度为指数级的，证明如图 5。

指数级时间复杂度很大，实践中并不实用。观察上述递归过程，实际上有很多“冗余”，很多子问题  $OPT(i, j)$  在重复计算。 $i, j$  各有  $n$  种情况，共有  $O(n^2)$  个子问题。每个子问题的最优值可以先存储起来，以空间换时间。如果粗略估计，每个子问题又有  $n$  种划分，故时间复杂度为  $O(n^3)$ 。

## Question 14

背包问题。

假设当包空  $W$ ，前  $i$  个物品的最优价值为  $OPT(i, W)$ ，则有递推式： $OPT(i, W) = \max\{OPT(i - 1, W), OPT(i - 1, W - w_i) + v_i\}$ ，下面给出伪代码及某个背包问题的示例：

可以看出求解背包问题的过程实际上就是按照递推公式求解一个二维矩阵的问题，下面给出实际运行的代码。(code/Knapsack)

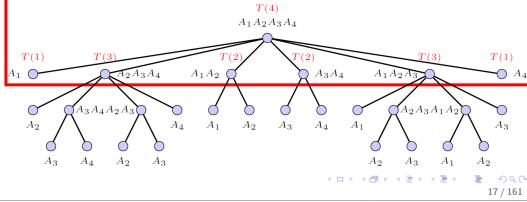
```
1 #include <iostream>
# include <string.h>
```

However, this is not a good implementation

### Theorem

Algorithm RECURSIVE-MATRIX-CHAIN costs exponential time.

- Let  $T(n)$  denote the time used to calculate product of  $n$  matrices. Then  $T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1)$  for  $n > 1$ .



(a) 已知的条件

### Proof.

- We shall prove  $T(n) \geq 2^{n-1}$  using the substitution technique.

- Basis:  $T(1) \geq 1 = 2^{1-1}$ .
- Induction:

$$T(n) \geq 1 + \sum_{k=1}^{n-1} (T(k) + T(n-k) + 1) \quad (1)$$

$$= n + 2 \sum_{k=1}^{n-1} T(k) \quad (2)$$

$$\geq n + 2 \sum_{k=1}^{n-1} 2^{k-1} \quad (3)$$

$$\geq n + 2(2^{n-1} - 1) \quad (4)$$

$$\geq n + 2^n - 2 \quad (5)$$

$$\geq 2^{n-1} \quad (6)$$

□

(b) 数学归纳法证明

图 5: 指数级时间复杂度证明

**Algorithm**

```

1: Initialize  $OPT[0, w] = 0$ ;
2:  $i = 1$  to  $n$  do
3:   end for
4:   for  $j = 1$  to  $w$  do
5:     for  $k = 1$  to  $j$  do
6:        $OPT[i, w] = \max\{OPT[i-1, w], v_i + OPT[i-1, w-w_i]\}$ ;
7:     end for
8:   end for
9: else  $OPT[n, W]$ 
• Here we use  $OPT[i, w]$  to represent  $OPT[\{1, 2, \dots, i\}, w]$  for simplicity.

```

(a) 伪代码

### An example: Step 1

$w = 3$	$v_1 = 2$	$v_2 = 3$	$v_3 = 4$	$v_4 = 5$	$w = 4$
$i = 3$					
$i = 2$					
$i = 1$					

Initially all  $OPT[i, w] = 0$

(b) Step1

### Step 2

$OPT[1, 2] = \max\{$	$i = 3$	$w = 0$
$OPT[1, 2](= 0)$ ,	$i = 2$	$w = 1$
$OPT[1, 2] + V_1 (= 2 + 2)$	$i = 1$	$w = 2$
$i = 0$	$\boxed{0}$	$0$

(c) Step2

**Step 3**

$w = 0$	$1$	$2$	$3$	$4$	$5$	$6$
$i = 3$	$0$	$0$	$2$	$4$	$5$	$6$
$i = 2$	$0$	$0$	$2$	$4$	$4$	$4$
$i = 1$	$0$	$0$	$2$	$2$	$2$	$2$
$i = 0$	$\boxed{0}$	$0$	$0$	$0$	$0$	$0$

(d) Step3

### Step 4

$OPT[2, 3] = \max\{$	$i = 3$	$w = 0$
$OPT[2, 3](= 2)$ ,	$i = 2$	$w = 1$
$OPT[2, 3] + V_2 (= 2 + 3)$	$i = 1$	$w = 2$

(e) Step4

### Backtracking

$OPT[3, 6] = \max\{$	$i = 3$	$w = 0$
$OPT[2, 6](= 4)$ ,	$i = 2$	$w = 1$
$OPT[2, 6] + V_3 (= 4 + 3)$	$i = 1$	$w = 2$
$i = 0$	$\boxed{0}$	$0$

(f) Step5

图 6: 伪代码及递归公式示例

```

3 #include <stdio.h>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10    int k,W;// 物品的数量和包的总容量
11    cin>>k>>W;// 输入 i 和 W
12    int w[k] = {0};// 第 i 个位置代表第 i+1 个物品的重量 w(i+1)
13    int v[k] = {0};// 第 i 个位置代表第 i+1 个物品的价值 v(i+1)
14    int OPT[k+1][W+1];
15    memset(OPT, 0, sizeof(OPT)); // 在头文件 #include<string.h> 中
16
17    for(int i=0;i<k; i++)
18    {
19        scanf("%d",&w[i]); // 输入各个物品的重量
20    }
21    for(int i=0;i<k; i++)
22    {
23        scanf("%d",&v[i]); // 输入各个物品的价值
24    }
25
26    // 动态规划求解，实际上在计算一个二维矩阵
27    for(int i = 1; i <= k; i++)
28    {
29        for(int j = 1; j <= W; j++)
30        {
31            if(j-w[i-1]<0)// 如果没有这一条件，下面 else 中的语句很容易超出索引
32                OPT[i][j] = OPT[i-1][j];
33            else
34                OPT[i][j] = max(OPT[i-1][j], OPT[i-1][j-w[i-1]]+v[i-1]);
35        }
36    }
37    int O = OPT[k][W];
38    for(int i = 0; i < k+1; i++)
39    {
40        for(int j = 0; j < W+1; j++)
41        {
42            cout<<OPT[i][j]<<" ";
43        }
44        cout<<endl;
45    }
46    cout<<O<<endl;
47
48    return 0;
49 }

```

求解完背包问题，顺便解决一个类似的问题，问题描述：一台服务器，空间  $M$ ，内存  $N$ 。现在有若干个任务，每个任务需求  $X_i$  的空间和  $Y_i$  的内存，并且能服务  $U_i$  的人数，在服务器上如何分配任务可以使得同时服务的人数最多。

这道题跟背包问题一样，不过涉及到两个限制条件，所以需要计算一个三维矩阵，假设  $OPT(i, M, N)$  表示将空间  $M$ ， $N$  分配给前  $i$  个任务能够同时服务的最多人数。可以给出递归公式： $OPT(i, M, N) = \max\{OPT(i - 1, M, N), OPT(i - 1, M - m_i, N - n_i) + u_i\}$ ，下面给出代码（在文件 code/Maximum Number of Users）：

```

1 #include <iostream>
2 #include <string.h>
3 #include <stdio.h>
4 #include <algorithm>

```

```

6 using namespace std;
8 int main()
{
10    int k,M,N;//任务的数量和服务器的空间和内存
11    cin>>k>>M>>N;//输入 i 和M,N
12    int m[k] = {0};//第 i 个位置代表第 i+1 个任务需求的空间m(i+1)
13    int n[k] = {0};//第 i 个位置代表第 i+1 个任务需求的内存(i+1)
14    int u[k] = {0};//第 i 个位置代表第 i+1 个任务可以服务的人数u(i+1)
15    int OPT[k+1][M+1][N+1];//三维数组
16    memset(OPT, 0, sizeof(OPT)); //在头文件#include<string.h>中
17
18    for (int i=0;i<k; i++)
19    {
20        scanf("%d",&m[i]); //输入各个任务的空间
21    }
22    for (int i=0;i<k; i++)
23    {
24        scanf("%d",&n[i]); //输入各个任务的内存
25    }
26    for (int i=0;i<k; i++)
27    {
28        scanf("%d",&u[i]); //输入各个任务可以服务的人数
29    }
30
31    //动态规划求解，实际上在计算一个三维矩阵
32    for (int i = 1; i <= k; i++)
33    {
34        for (int j = 1; j <= M; j++)
35        {
36            for (int s = 1; s <= N; s++)
37            {
38                if ((j-m[i-1]<0) || (s-n[i-1]<0)) //如果没有这一条件，下面else中的语句很容易超出索引
39                    OPT[i][j][s] = OPT[i-1][j][s];
40                else
41                    OPT[i][j][s] = max(OPT[i-1][j][s], OPT[i-1][j-m[i-1]][s-n[i-1]]+u[i-1]);
42            }
43        }
44    }
45    int O = OPT[k][M][N];
46    cout<<O<<endl;
47    return 0;
48 }

```

上面的代码写的并不好，建议遇到不知道数组规模多大的时候采用动态数组的定义，即用指针来动态定义数组，可以改写成以下形式：

```

1 #include <iostream>
2 #include <string.h>
3 #include <stdio.h>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10    int k,M,N;//任务的数量和服务器的空间和内存
11    cin>>M>>N>>k;//输入 k 和M,N
12    int *m = new int[k];//第 i 个位置代表第 i+1 个任务需求的空间m(i+1)
13    int *n = new int[k];//第 i 个位置代表第 i+1 个任务需求的内存(i+1)
14    int *u = new int[k];//第 i 个位置代表第 i+1 个任务可以服务的人数u(i+1)

```

```

16 int ***OPT = new int**[k+1];//三维数组
17 for(int i=0;i<k; i++)
18 {
19     scanf("%d",&m[i]);//输入各个任务的空间
20     scanf("%d",&n[i]);//输入各个任务的内存
21     scanf("%d",&u[i]);//输入各个任务的服务的人数
22 }
23 for (int i = 0; i < k + 1; i++)
24 {
25     OPT[i] = new int*[M + 1];
26     for (int j = 0; j < M + 1; j++)
27     {
28         OPT[i][j] = new int[N + 1];
29         for (int s = 0; s < N + 1; s++)
30             OPT[i][j][s] = 0;
31     }
32 }

33 //动态规划求解，实际上在计算一个三维矩阵
34 for(int i =1;i<=k; i++)
35 {
36     for(int j = 1;j<=M; j++)
37     {
38         for(int s = 1;s<=N; s++)
39         {
40             if((j-m[i-1]<0) || (s-n[i-1]<0))//如果没有这一条件，下面else中的语句很容易超出索引
41                 OPT[i][j][s] = OPT[i-1][j][s];
42             else
43                 OPT[i][j][s] = max(OPT[i-1][j][s],OPT[i-1][j-m[i-1]][s-n[i-1]]+u[i-1]);
44         }
45     }
46 }
47 int O = OPT[k][M][N];
48
49 cout<<O<<endl;
50
51 return 0;
52 }
```

上述代码的问题实际开了个三维动态数组，而实际递归计算的时候，只用到了两层的二维数组，所以在计算的时候可以进一步简化，只用两个动态的二维数组就可以代替三维动态数组。代码如下：

```

1 #include <iostream>
2 #include <string.h>
3 #include <stdio.h>
4 #include <algorithm>
5
6 using namespace std;
7
8 int main()
9 {
10     int k,M,N;//任务的数量和服务器的空间和内存
11     cin>>M>>N>>k;//输入k和M,N
12     int *m = new int [k];//第i个位置代表第i+1个任务需求的空间m(i+1)
13     int *n = new int [k];//第i个位置代表第i+1个任务需求的内存(n(i+1))
14     int *u = new int [k];//第i个位置代表第i+1个任务可以服务的人数u(i+1)
15     int **OPT = new int *[M+1];//二维数组
16     int **newOPT = new int *[M+1];//二维数组
17     for(int i=0;i<k; i++)
18     {
19         scanf("%d",&m[i]);//输入各个任务的空间
20     }
21 }
```

```

21     scanf("%d",&n[i]); //输入各个任务的内存
22     scanf("%d",&u[i]); //输入各个任务的服务的人数
23 }
24
25 for (int i = 0; i < M + 1; i++)
26 {
27     OPT[i] = new int[N + 1];
28     newOPT[i] = new int[N + 1];
29     for (int j = 0; j < N + 1; j++)
30     {
31         OPT[i][j] = 0;
32         newOPT[i][j] = 0;
33     }
34 }
35 //动态规划求解，实际上在计算一个三维矩阵
36 for (int i = 1; i <= k; i++)
37 {
38     for (int j = 1; j <= M; j++)
39     {
40         for (int s = 1; s <= N; s++)
41         {
42             if ((j - m[i - 1] < 0) || (s - n[i - 1] < 0)) //如果没有这一条件，下面else中的语句很容易超出索引
43                 newOPT[j][s] = OPT[j][s];
44             else
45                 newOPT[j][s] = max(OPT[j][s], OPT[j - m[i - 1]][s - n[i - 1]] + u[i - 1]);
46         }
47     }
48     //新的矩阵代替旧的矩阵
49     for (int j = 1; j <= M; j++)
50     {
51         for (int s = 1; s <= N; s++)
52         {
53             OPT[j][s] = newOPT[j][s];
54         }
55     }
56 }
57
58 int O = newOPT[M][N];
59 delete []m;
60 delete []n;
61 delete []u;
62 for (int i = 0; i < M + 1; i++)
63 {
64     delete []OPT[i];
65     delete []newOPT[i];
66 }
67 cout << O << endl;
68 system("pause");
69 return 0;
}

```

## Question 15

序列匹配。问题描述：获得两个序列如 *occurrancce* 和 *occurrence* 的最优匹配（用打分函数衡量）。打分函数的设置非常重要，假设打分函数设置为：1. 两字母相同，加 1 分；2. 两字母不同，减 3 分；3. 插入或者删除，减 3 分（如果打分函数设置不合理，那么下面描述的动态规划算法，将会毫无意义。所以打分函数的设置也是一个知识点，这个 pdf 中就不叙述了）。下图给出一个示例和递归公式的推导过

程。

### Alignment is useful cont'd

- Application 2: In addition, we can also determine the most likely operations changing "OCCURRENCE" into "OCURRANCE".

① Alignment 1:

```
S': O-CURRANCE
| | | | | |
T': OCCURRENCE
```

$$s(T', S') = 1 - 3 + 1 + 1 + 1 + 1 - 1 + 1 + 1 + 1 = 4$$

② Alignment 2:

```
S': O-CURR-ANCE
| | | | | |
T': OCCURRE-NCE
```

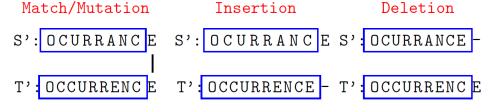
$$s(T', S') = 1 - 3 + 1 + 1 + 1 + 1 - 3 - 3 + 1 + 1 + 1 = -1$$

- Thus, the first alignment might describes the real generating process of "OCURRANCE" from "OCCURRENCE".

### The general form of sub-problems and recursions II

- Let's consider the first decision made for  $S_m$  in the optimal solution. There are three cases:

- $S_n$  comes from  $T_m$ : represented as aligning  $S_n$  with  $T_m$ . Then it suffices to align  $T[1..m-1]$  with  $S[1..n-1]$ .
- $S_n$  is an INSERTION: represented as aligning  $S_n$  with a space ' $\cdot$ '. Then it suffices to align  $T[1..m]$  and  $S[1..n-1]$ .
- $S_n$  comes from  $T[1..m-1]$ : represented as aligning  $T_m$  with a space ' $\cdot$ '. Then it suffices to align  $T[1..m-1]$  and  $S[1..n]$ .



(a) 示例

(b) 递归公式推导过程

图 7: 打分函数示例及递归公式推导过程

递归公式及伪代码如下：

### The general form of sub-problems and recursions III

- Summarizing these three examples of sub-problems, we can design the general form of sub-problems as: aligning a **prefix** of  $T$  (denoted as  $T[1..i]$ ) and **prefix** of  $S$  (denoted as  $S[1..j]$ ). Denote the optimal solution value as  $OPT(i, j)$ .

- We can prove the following optimal substructure property:

$$OPT(i, j) = \max \begin{cases} s(T_i, S_j) + OPT(i-1, j-1) \\ s(\cdot, S_j) + OPT(i, j-1) \\ s(T_i, \cdot) + OPT(i-1, j) \end{cases}$$

### Needleman-Wunsch algorithm [1970]

```
NEEDLEMAN-WUNSCH( $T, S$ )
1: for  $i = 0$  to  $m$  do
2:    $OPT[i, 0] = -3 * i$ ;
3: end for
4: for  $j = 0$  to  $n$  do
5:    $OPT[0, j] = -3 * j$ ;
6: end for
7: for  $j = 1$  to  $n$  do
8:   for  $i = 1$  to  $m$  do
9:      $OPT[i, j] = \max\{OPT[i-1, j-1] + s(T_i, S_j), OPT[i-1, j] - 3, OPT[i, j-1] - 3\}$ ;
10:  end for
11: end for
12: return  $OPT[m, n]$ ;
```

Note: the first column is introduced to describe the alignment of prefixes  $T[1..i]$  with an empty sequence  $\epsilon$ , so does the first row.

(a) 递归公式

(b) 伪代码

图 8: alignment 递归公式及伪代码

实际上递归公式就是在算一个二维矩阵，规模为  $(m+1) * (n+1)$ ，注意刚开始的时候会有一个空字符表示某个字母跟空字符匹配。在矩阵的最右下角，即  $(OPT(m+1, n+1))$  会得到最优匹配的得分。但是如何获得最优得分所对应的匹配，这里需要做一下回溯的过程。下面给出二维矩阵及回溯路线，路线中  $-2$  直接向上到  $1$ ，说明  $-2$  横向对应的  $c$  匹配了空字符。

## Question 16

接着上一题，如果要匹配的两个字符串是两篇文章，那么计算时间和存储空间（需要存储一个二维矩阵）会变得非常巨大，这一题介绍高级动态规划，来优化上一题的算法

## General cases

S: ''	O	C	U	R	R	A	N	C	E	
T: ''	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3	1	-2	5	8	-11	-14	-17	-20	-23
C	-6	-2	2	-1	4	-7	-10	-13	-16	-19
C	-9	-5	-1	1	-2	-5	-8	-11	-12	-15
U	-12	-8	-4	0	0	-3	-6	-9	-12	-13
R	-15	-11	-7	-3	1	1	-2	-5	-8	-11
R	-18	-14	-10	-6	-2	2	0	-3	-6	-9
E	-21	-17	-13	-9	-5	-1	1	-1	-4	-5
N	-24	-20	-16	-12	-8	-4	-2	2	-1	-4
C	-27	-23	-19	-15	-11	-7	-5	-1	3	0
E	-30	-26	-22	-18	-14	-10	-8	-4	0	4

Score:  $\text{OPT}(\text{OC}^*, \text{OCUR}^*) = \max \left\{ \begin{array}{l} \text{OPT}(\text{OC}^*, \text{OCUR}^*) = -3 \quad (=11) \\ \text{OPT}(\text{OC}^*, \text{OCUR}^*) = -1 \quad (=6) \\ \text{OPT}(\text{OC}^*, \text{OCUR}^*) = -3 \quad (=4) \end{array} \right.$

Alignment:  $S^* = \text{OCUR}$   
 $T^* = \text{OC}^*$

(a) 二维矩阵

## Find the optimal alignment via backtracking

S: ''	O	C	U	R	R	A	N	C	E	
T: ''	0	-3	-6	-9	-12	-15	-18	-21	-24	-27
O	-3	1	-2	-5	-8	-11	-14	-17	-20	-23
C	-6	2	2	-1	-4	-7	-10	-13	-16	-19
C	-9	-5	-1	1	-2	-5	-8	-11	-12	-15
U	-12	-8	-4	0	0	-3	-6	-9	-12	-13
R	-15	-11	-7	-3	0	1	-2	-5	-8	-11
R	-18	-14	-10	-6	-2	2	0	-3	-6	-9
E	-21	-17	-13	-9	-5	-1	1	-1	-4	-5
N	-24	-20	-16	-12	-8	-4	-2	2	-1	-4
C	-27	-23	-19	-15	-11	-7	-5	-1	3	0
E	-30	-26	-22	-18	-14	-10	-8	-4	0	4

Optimal Alignment:  $S^* = \text{O-CURRANCE}$   
 $T^* = \text{OCCURRENCE}$

(b) 回溯路线

图 9: 二维矩阵及回溯路线

## Question 17

leetcode 第 1143 题，最长子序列。问题描述：此题中的最长子序列不是最长子字符串，比如  $text1 = "abcde"$ ,  $text2 = "ace"$ ，最长公共子序列是 "ace"，长度为 3。

解：最长子序列问题是序列匹配的特例，把序列匹配 (alignment) 中的打分函数设置如下：1. 两字母相同，加 1 分；2. 两字母不同，不加分；3. 插入或者删除，不加分，这样设置的打分函数的最终结果就是两个序列最长公共子序列的长度。递推公式的推导过程见下图：

### LONGEST COMMON SUBSEQUENCE problem

- The alignment of two sequences  $S[1..m]$  and  $T[1..n]$  reduces to finding the LONGEST COMMON SUBSEQUENCE of them when mutations are not allowed.
- Solution: the longest common subsequence of  $S$  and  $T$ . Let's describe the solving process as multi-stage decision-making process. At each stage, we decide whether align a letter  $S[i]$  with  $T[j]$  or not.
- Let's consider the first decision, i.e., whether we align  $S[m]$  with  $T[n]$  or not. We have two options:
  - Align  $S[m]$  and  $T[n]$  (when  $S[m] = T[n]$ ): Then the subproblem is how to find the longest common subsequence of  $S[1..m-1]$  and  $T[1..n-1]$ .
  - Do not align them: Then we have two subproblems: find the longest common subsequence of  $S[1..m]$  and  $T[1..n-1]$ , and find the longest common subsequence of  $S[1..m-1]$  and  $T[1..n]$ .

107 / 161

(a) 递推过程分析

### Basic DP algorithm

- Summarizing these examples, we determine the general form of subproblems as: finding the longest common subsequences of  $S[1..i]$  and  $T[1..j]$  and denote the optimal value as  $OPT(i, j)$ .
- Then we have the following recursion:

$$OPT(i, j) = \begin{cases} 0 & \text{if } i = j = 0 \\ OPT(i-1, j-1) + 1 & \text{if } S[i] = T[j] \\ \max\{OPT(i-1, j), OPT(i, j-1)\} & \text{otherwise} \end{cases}$$

108 / 161

(b) 递推公式

图 10: 递推公式及其推导过程

下面准备编写这题的程序，先说明一些关于数据结构字符串 (string) 的相关知识点：

- 计算  $string str$  的长度，使用  $str.length()$  函数 (还有其他方法，比如  $str.size()$ )，可以自行百度；
- 要求自行输入  $string str$ :

- (1) 使用 `getline(cin, str)` 函数, 这个函数包括在头文件 `#include <string>` 中, 或者直接 `cin >> str1 >> str2;` 这样输入的两个字符串之间可以用字符隔开;
- (2) 使用 `scanf("%s", str1)`, 或者 `scanf("%s", &str1);`
- (3) 定义一个字符数组 `char a[20]`, 然后 `scanf("%s", a)`, 或者 `scanf("%s", &a)`
3. 在字符串 `string str` 的指定位置前面插入字符串, 使用 `str.insert(4, "hello");`
4. 在字符串 `string str` 的指定的开始位置 (第一个参数) 替换掉指定长度 (第二个参数) 的字符串, 使用 `str.replace(3, 4, "may")`。

下面给出实际可以运行的代码 (在文件 `code/LongestCommonSubsequence`) 和改进的代码 (数组采用动态定义):

```

1 #include <iostream>
2 #include <string>
3 #include <string.h>
4 #include <algorithm>
5 #include <stdio.h>
6
7 using namespace std;
8
9 int main()
10 {
11     string str1,str2;
12     getline(cin,str1);
13     getline(cin,str2);
14     int m = str1.length(); // 第一个字符串的长度
15     int n = str2.length(); // 第二个字符串的长度
16     int OPT[m+1][n+1]; // 0 矩阵
17     memset(OPT, 0, sizeof(OPT)); // 在头文件 #include<string.h> 中
18     // 计算二维数组, 规模为 mn
19
20     for (int i=1;i<m+1;i++)
21     {
22         for (int j=1;j<n+1;j++)
23         {
24             if (str1[i-1]==str2[j-1])
25             {
26                 OPT[i][j] = OPT[i-1][j-1] + 1;
27             }
28             if (str1[i-1]!=str2[j-1])
29             {
30                 OPT[i][j] = max(OPT[i-1][j],OPT[i][j-1]);
31             }
32         }
33     }
34     for (int i=0;i<m+1;i++)
35     {
36         for (int j=0;j<n+1;j++)
37         {
38             cout<<OPT[i][j]<<" ";
39         }
40         cout<<endl;
41     }
42     printf("%d",OPT[m][n]);
43
44     return 0;
45 }
```

```
#include <iostream>
```

```

2 #include <string>
3 #include <string.h>
4 #include <algorithm>
5 #include <stdio.h>
6
7 using namespace std;
8
9
10 int func(string str1, string str2)
11 {
12     int m = str1.length();
13     int n = str2.length();
14     int **OPT=new int*[m+1];//0矩阵
15     for (int i = 0; i < m + 1; i++)
16     {
17         OPT[i] = new int[n + 1];
18         for (int j = 0; j < n + 1; j++)
19             OPT[i][j] = 0;
20     }
21
22 //计算二维数组，规模为mn
23
24     for (int i=1;i<m+1;i++)
25     {
26         for (int j=1;j<n+1;j++)
27         {
28             if(str1[i-1]==str2[j-1])
29             {
30                 OPT[i][j] = OPT[i-1][j-1] + 1;
31             }
32             if(str1[i-1]!=str2[j-1])
33             {
34                 OPT[i][j] = max(OPT[i-1][j],OPT[i][j-1]);
35             }
36         }
37     }
38     return OPT[m][n];
39 }
40
41 int main()
42 {
43     int k;//一组测试有k对序列
44     scanf("%d", &k);
45     int *m = new int[k];
46     int *n = new int[k];
47     int *solution=new int[k];
48
49     for (int i=0;i<k;i++)
50     {
51         cout<<"输入规模："<<endl;
52         scanf("%d%d",&m[i],&n[i]);
53         string str1,str2;
54         cout<<"输入字符串："<<endl;
55         cin >> str1 >> str2;//连续输入两个字符串，中间可以有空格
56         solution[i] = func(str1,str2);
57     }
58     for (int i=0;i<k;i++)
59     {
60         cout<<solution[i]<<endl;
61     }
62     delete [] m;
63     delete [] n;
64     delete [] solution;
65     return 0;
66 }
```

上面的代码实际上是在打印了这样一个二维矩阵：

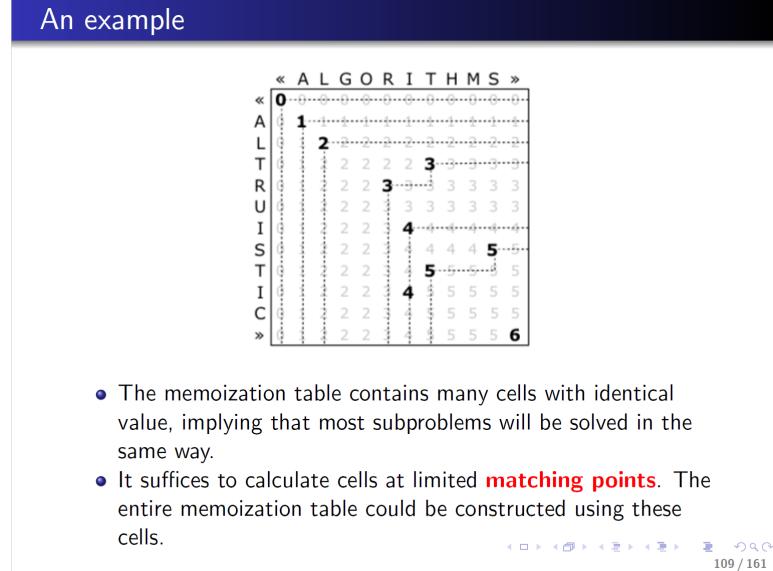


图 11：最长公共子序列的动态决策过程

图中的最后一行和最后一列实际上可以省略，所以如果字符串的规模分别为  $m, n$ ，则二维矩阵  $OPT$  的规模必须为  $m + 1, n + 1$ ，然后利用递推公式来计算二维数组中的各个数值，最后  $OPT[m][n]$  表示两个字符串的最长公共子序列的长度。下面再给出 leetcode 上的类函数的写法（面试时手写代码，都要写成这种形式。上面那种要输入完整信息的是比较旧的 OJ 测试使用的）：

```

1 class Solution {
2 public:
3     int longestCommonSubsequence(string text1, string text2) {
4         int m = text1.length(); // 第一个字符串的长度
5         int n = text2.length(); // 第二个字符串的长度
6         int OPT[m+1][n+1]; // 0 矩阵
7         memset(OPT, 0, sizeof(OPT)); // 在头文件 #include<string.h> 中
8         // 计算二维数组，规模为 mn
9         for (int i=1;i<m+1;i++)
10        {
11            for (int j=1;j<n+1;j++)
12            {
13                if (text1[i-1]==text2[j-1])
14                {
15                    OPT[i][j] = OPT[i-1][j-1] + 1;
16                }
17                if (text1[i-1]!=text2[j-1])
18                {
19                    OPT[i][j] = max(OPT[i-1][j], OPT[i][j-1]);
20                }
21            }
22        }
23        return OPT[m][n];
24    };
}

```

上述代码对应的代码在 *leetcode* 中可以通过，但是有的 *OJ* 测试系统中，对算法的时间和空间复杂度要求非常高，这种算法不一定能通过，还可以进一步优化。实际上在计算的时候，可以像之前背包系列问题那样，每次运算只涉及到两列或两层，不用开二维或三维数组。

## Question 18

上一题求最长公共子序列长度中的方法需要计算的量有点多，浪费了太多的时间和空间，实际上可以只计算变化的点处的值。

## Question 19

**leetcode 第 198 题。**问题描述：一个窃贼偷窃一排房子，每个房子里有一定价值的物品，但是不同连续偷连续两个房子，会触发警报，求窃贼可以偷的最高价值。如果房子改成一圈又如何？

房子并列：

1. 递推式：房子排成一排，设  $OPT(n)$  表示偷前  $n$  个房子的最大价值， $v_n$  表示第  $n$  个房子的价值，则有递推式： $OPT(n) = \max\{OPT(n-1), OPT(n-2) + v_n\}$ 。在编程的时候可以开很大的内存来存储  $OPT$ ，或者只用两个变量来计算。

实际代码：

```
1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         int k = nums.size();
5         int *OPT = new int[k+1];
6         for(int i = 0; i < k+1; i++)
7         {
8             OPT[i] = 0;
9         }
10        OPT[1] = nums[0];
11        // 动态规划求解
12        for(int i = 2; i < k+1; i++)
13        {
14            OPT[i] = max(OPT[i-1], nums[i-1]+OPT[i-2]);
15        }
16
17        return OPT[k];
18    }
19 }
```

```
1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         int k = nums.size();
5         int curmax = 0;
6         int premax = 0;
7         // 动态规划求解
8         for(int i = 0; i < k; i++)
9         {
10            int temp = curmax;
11            curmax = max(premax+nums[i], curmax);
12            premax = temp;
13        }
14
15        return curmax;
16    }
17 }
```

```
13 }  
14  
15     return curmax;  
16 }  
17 };
```

伪代码：(只写后一种方法的伪代码)

```
1 rob(nums)
2     k = nums.size();
3     premax = 0;
4     curman = 0;
5     for i=0 to k do
6         temp = curmax;
7         curmax = max(premax+nums[i], curmax);
8         premax = temp;
9     end for
10    return curmax;
```

房子围成一圈：

1. 递推式：将房子围成一圈的问题改成房子排两排的问题。因为第 0 个房子和第  $n$  个房子靠近，所以这两个房子不能同时偷，以此将原问题分成两部分，房子围成一圈时偷窃的所有方法的最大价值 =  $\max\{\text{第 } 0 \text{ 个房子不偷时的所有方法的最大价值}, \text{第 } n \text{ 个房子不偷时的最大价值}\}$ ，然后两个子问题就可以用房子单排的方法来解答。

## 2. 代码及伪代码:

```

1    robs(nums)
2        k = nums.size();
3        nums1 = nums[1, 2, 3, ..., k-1];
4        nums2 = nums[2, 3, 4, ..., k];
5        v1 = rob(nums1);
6        v2 = rob(nums2);
7        return max(v1, v2)

```

```
1 class Solution {
2 public:
3     int rob(vector<int>& nums) {
4         int k = nums.size();
5         if (k == 0)
6         {
7             return 0;
8         }
9         if (k==1)
10        {
11            return nums[0];
12        }
13        vector<int> v1(nums.begin(),nums.begin() + k-1); // 取vector中一部分元素
14        vector<int> v2(nums.begin() + 1,nums.begin() + k);
15        int vv1 = robp(v1);
16        int vv2 = robp(v2);
17        return max(vv1,vv2);
18    }
19    int robp(vector<int>& num) {
20        int k = num.size();
21        if(k==0)
```

```

23     {
24         return 0;
25     }
26     int curmax = 0;
27     int premax = 0;
28     //动态规划求解
29     for(int i = 0; i < k; i++)
30     {
31         int temp = curmax;
32         curmax = max(premax+num[i], curmax);
33         premax = temp;
34     }
35     return curmax;
36 };

```

## Question 20

leetcode 第 334 题。问题描述：在上次打劫完一条街道之后和一圈房屋后，小偷又发现了一个新的可行窃的地区。这个地区只有一个入口，我们称之为“根”。除了“根”之外，每栋房子有且只有一个“父”房子与之相连。一番侦察之后，聪明的小偷意识到“这个地方的所有房屋的排列类似于一棵二叉树”。如果两个直接相连的房子在同一天晚上被打劫，房屋将自动报警。计算在不触动警报的情况下，小偷一晚能够盗取的最高金额。

### 1. 递推式

设  $d(p)$  表示根节点为  $p$  的二叉树最优偷盗的金额， $v(d)$  表示节点  $d$  的价值。分析一个子二叉树  $\{p, l, r, ll, lr, rl, rr\}$ ，容易得到递推式： $d(p) = \max\{d(l) + d(r), d(ll) + d(lr) + d(rl) + d(rr) + v(d), d(l) + d(rl) + d(rr), d(r) + d(ll) + d(lr)\}$ ，简单分析  $d(r) \geq d(rl) + d(rr)$ ,  $d(l) \geq d(ll) + d(lr)$ ，所以递推式只涉及两项  $d(p) = \max\{d(l) + d(r), d(ll) + d(lr) + d(rl) + d(rr) + v(d)\}$ 。

### 2. 代码及伪代码

实际可运行代码：

```

#include <iostream>
2 #include <stdio.h>
3 #include <algorithm>
4
5 using namespace std;
6
7 //定义二叉树结构体
8 struct TreeNode{
9     int data;
10    TreeNode *lchild;
11    TreeNode *rchild;
12};
13
14 //先序创建二叉树
15 void CreateBiTree(TreeNode **T)
16 {
17     int ch;
18     cin >> ch;
19     if (ch == -1)
20     {
21         *T = NULL;
22         return;
23     }
24     *T = new TreeNode();
25     (*T)->data = ch;
26     CreateBiTree(&(*T)->lchild);
27     CreateBiTree(&(*T)->rchild);
28 }
29
30 //输出二叉树
31 void OutputBiTree(TreeNode *T)
32 {
33     if (T != NULL)
34     {
35         cout << T->data << " ";
36         OutputBiTree(T->lchild);
37         OutputBiTree(T->rchild);
38     }
39 }
40
41 //计算二叉树的深度
42 int DepthBiTree(TreeNode *T)
43 {
44     if (T == NULL)
45     {
46         return 0;
47     }
48     else
49     {
50         int ldepth = DepthBiTree(T->lchild);
51         int rdepth = DepthBiTree(T->rchild);
52         if (ldepth > rdepth)
53         {
54             return ldepth + 1;
55         }
56         else
57         {
58             return rdepth + 1;
59         }
60     }
61 }
62
63 //计算二叉树的节点数
64 int NodeNumBiTree(TreeNode *T)
65 {
66     if (T == NULL)
67     {
68         return 0;
69     }
70     else
71     {
72         int lnode = NodeNumBiTree(T->lchild);
73         int rnode = NodeNumBiTree(T->rchild);
74         return lnode + rnode + 1;
75     }
76 }
77
78 //计算二叉树的叶子数
79 int LeafNodeNumBiTree(TreeNode *T)
80 {
81     if (T == NULL)
82     {
83         return 0;
84     }
85     else
86     {
87         if (T->lchild == NULL && T->rchild == NULL)
88         {
89             return 1;
90         }
91         else
92         {
93             int lleaf = LeafNodeNumBiTree(T->lchild);
94             int rleaf = LeafNodeNumBiTree(T->rchild);
95             return lleaf + rleaf;
96         }
97     }
98 }
99
100 //计算二叉树的非叶节点数
101 int NonLeafNodeNumBiTree(TreeNode *T)
102 {
103     if (T == NULL)
104     {
105         return 0;
106     }
107     else
108     {
109         if (T->lchild == NULL && T->rchild == NULL)
110         {
111             return 0;
112         }
113         else
114         {
115             int lnonleaf = NonLeafNodeNumBiTree(T->lchild);
116             int rnonleaf = NonLeafNodeNumBiTree(T->rchild);
117             return lnonleaf + rnonleaf + 1;
118         }
119     }
120 }
121
122 //计算二叉树的满度数
123 int FullDegreeBiTree(TreeNode *T)
124 {
125     if (T == NULL)
126     {
127         return 0;
128     }
129     else
130     {
131         if (T->lchild == NULL && T->rchild == NULL)
132         {
133             return 1;
134         }
135         else
136         {
137             int lfull = FullDegreeBiTree(T->lchild);
138             int rfull = FullDegreeBiTree(T->rchild);
139             return lfull + rfull;
140         }
141     }
142 }
143
144 //计算二叉树的平衡度数
145 int BalanceDegreeBiTree(TreeNode *T)
146 {
147     if (T == NULL)
148     {
149         return 0;
150     }
151     else
152     {
153         if (T->lchild == NULL && T->rchild == NULL)
154         {
155             return 1;
156         }
157         else
158         {
159             int lbalance = BalanceDegreeBiTree(T->lchild);
160             int rbalance = BalanceDegreeBiTree(T->rchild);
161             return lbalance + rbalance;
162         }
163     }
164 }
165
166 //计算二叉树的深度
167 int DepthBiTree(TreeNode *T)
168 {
169     if (T == NULL)
170     {
171         return 0;
172     }
173     else
174     {
175         int ldepth = DepthBiTree(T->lchild);
176         int rdepth = DepthBiTree(T->rchild);
177         if (ldepth > rdepth)
178         {
179             return ldepth + 1;
180         }
181         else
182         {
183             return rdepth + 1;
184         }
185     }
186 }
187
188 //计算二叉树的节点数
189 int NodeNumBiTree(TreeNode *T)
190 {
191     if (T == NULL)
192     {
193         return 0;
194     }
195     else
196     {
197         int lnode = NodeNumBiTree(T->lchild);
198         int rnode = NodeNumBiTree(T->rchild);
199         return lnode + rnode + 1;
200     }
201 }
202
203 //计算二叉树的叶子数
204 int LeafNodeNumBiTree(TreeNode *T)
205 {
206     if (T == NULL)
207     {
208         return 0;
209     }
210     else
211     {
212         if (T->lchild == NULL && T->rchild == NULL)
213         {
214             return 1;
215         }
216         else
217         {
218             int lleaf = LeafNodeNumBiTree(T->lchild);
219             int rleaf = LeafNodeNumBiTree(T->rchild);
220             return lleaf + rleaf;
221         }
222     }
223 }
224
225 //计算二叉树的非叶节点数
226 int NonLeafNodeNumBiTree(TreeNode *T)
227 {
228     if (T == NULL)
229     {
230         return 0;
231     }
232     else
233     {
234         if (T->lchild == NULL && T->rchild == NULL)
235         {
236             return 0;
237         }
238         else
239         {
240             int lnonleaf = NonLeafNodeNumBiTree(T->lchild);
241             int rnonleaf = NonLeafNodeNumBiTree(T->rchild);
242             return lnonleaf + rnonleaf + 1;
243         }
244     }
245 }
246
247 //计算二叉树的满度数
248 int FullDegreeBiTree(TreeNode *T)
249 {
250     if (T == NULL)
251     {
252         return 0;
253     }
254     else
255     {
256         if (T->lchild == NULL && T->rchild == NULL)
257         {
258             return 1;
259         }
260         else
261         {
262             int lfull = FullDegreeBiTree(T->lchild);
263             int rfull = FullDegreeBiTree(T->rchild);
264             return lfull + rfull;
265         }
266     }
267 }
268
269 //计算二叉树的平衡度数
270 int BalanceDegreeBiTree(TreeNode *T)
271 {
272     if (T == NULL)
273     {
274         return 0;
275     }
276     else
277     {
278         if (T->lchild == NULL && T->rchild == NULL)
279         {
280             return 1;
281         }
282         else
283         {
284             int lbalance = BalanceDegreeBiTree(T->lchild);
285             int rbalance = BalanceDegreeBiTree(T->rchild);
286             return lbalance + rbalance;
287         }
288     }
289 }
290
291 //计算二叉树的深度
292 int DepthBiTree(TreeNode *T)
293 {
294     if (T == NULL)
295     {
296         return 0;
297     }
298     else
299     {
300         int ldepth = DepthBiTree(T->lchild);
301         int rdepth = DepthBiTree(T->rchild);
302         if (ldepth > rdepth)
303         {
304             return ldepth + 1;
305         }
306         else
307         {
308             return rdepth + 1;
309         }
310     }
311 }
312
313 //计算二叉树的节点数
314 int NodeNumBiTree(TreeNode *T)
315 {
316     if (T == NULL)
317     {
318         return 0;
319     }
320     else
321     {
322         int lnode = NodeNumBiTree(T->lchild);
323         int rnode = NodeNumBiTree(T->rchild);
324         return lnode + rnode + 1;
325     }
326 }
327
328 //计算二叉树的叶子数
329 int LeafNodeNumBiTree(TreeNode *T)
330 {
331     if (T == NULL)
332     {
333         return 0;
334     }
335     else
336     {
337         if (T->lchild == NULL && T->rchild == NULL)
338         {
339             return 1;
340         }
341         else
342         {
343             int lleaf = LeafNodeNumBiTree(T->lchild);
344             int rleaf = LeafNodeNumBiTree(T->rchild);
345             return lleaf + rleaf;
346         }
347     }
348 }
349
350 //计算二叉树的非叶节点数
351 int NonLeafNodeNumBiTree(TreeNode *T)
352 {
353     if (T == NULL)
354     {
355         return 0;
356     }
357     else
358     {
359         if (T->lchild == NULL && T->rchild == NULL)
360         {
361             return 0;
362         }
363         else
364         {
365             int lnonleaf = NonLeafNodeNumBiTree(T->lchild);
366             int rnonleaf = NonLeafNodeNumBiTree(T->rchild);
367             return lnonleaf + rnonleaf + 1;
368         }
369     }
370 }
371
372 //计算二叉树的满度数
373 int FullDegreeBiTree(TreeNode *T)
374 {
375     if (T == NULL)
376     {
377         return 0;
378     }
379     else
380     {
381         if (T->lchild == NULL && T->rchild == NULL)
382         {
383             return 1;
384         }
385         else
386         {
387             int lfull = FullDegreeBiTree(T->lchild);
388             int rfull = FullDegreeBiTree(T->rchild);
389             return lfull + rfull;
390         }
391     }
392 }
393
394 //计算二叉树的平衡度数
395 int BalanceDegreeBiTree(TreeNode *T)
396 {
397     if (T == NULL)
398     {
399         return 0;
400     }
401     else
402     {
403         if (T->lchild == NULL && T->rchild == NULL)
404         {
405             return 1;
406         }
407         else
408         {
409             int lbalance = BalanceDegreeBiTree(T->lchild);
410             int rbalance = BalanceDegreeBiTree(T->rchild);
411             return lbalance + rbalance;
412         }
413     }
414 }
415
416 //计算二叉树的深度
417 int DepthBiTree(TreeNode *T)
418 {
419     if (T == NULL)
420     {
421         return 0;
422     }
423     else
424     {
425         int ldepth = DepthBiTree(T->lchild);
426         int rdepth = DepthBiTree(T->rchild);
427         if (ldepth > rdepth)
428         {
429             return ldepth + 1;
430         }
431         else
432         {
433             return rdepth + 1;
434         }
435     }
436 }
437
438 //计算二叉树的节点数
439 int NodeNumBiTree(TreeNode *T)
440 {
441     if (T == NULL)
442     {
443         return 0;
444     }
445     else
446     {
447         int lnode = NodeNumBiTree(T->lchild);
448         int rnode = NodeNumBiTree(T->rchild);
449         return lnode + rnode + 1;
450     }
451 }
452
453 //计算二叉树的叶子数
454 int LeafNodeNumBiTree(TreeNode *T)
455 {
456     if (T == NULL)
457     {
458         return 0;
459     }
460     else
461     {
462         if (T->lchild == NULL && T->rchild == NULL)
463         {
464             return 1;
465         }
466         else
467         {
468             int lleaf = LeafNodeNumBiTree(T->lchild);
469             int rleaf = LeafNodeNumBiTree(T->rchild);
470             return lleaf + rleaf;
471         }
472     }
473 }
474
475 //计算二叉树的非叶节点数
476 int NonLeafNodeNumBiTree(TreeNode *T)
477 {
478     if (T == NULL)
479     {
480         return 0;
481     }
482     else
483     {
484         if (T->lchild == NULL && T->rchild == NULL)
485         {
486             return 0;
487         }
488         else
489         {
490             int lnonleaf = NonLeafNodeNumBiTree(T->lchild);
491             int rnonleaf = NonLeafNodeNumBiTree(T->rchild);
492             return lnonleaf + rnonleaf + 1;
493         }
494     }
495 }
496
497 //计算二叉树的满度数
498 int FullDegreeBiTree(TreeNode *T)
499 {
500     if (T == NULL)
501     {
502         return 0;
503     }
504     else
505     {
506         if (T->lchild == NULL && T->rchild == NULL)
507         {
508             return 1;
509         }
510         else
511         {
512             int lfull = FullDegreeBiTree(T->lchild);
513             int rfull = FullDegreeBiTree(T->rchild);
514             return lfull + rfull;
515         }
516     }
517 }
518
519 //计算二叉树的平衡度数
520 int BalanceDegreeBiTree(TreeNode *T)
521 {
522     if (T == NULL)
523     {
524         return 0;
525     }
526     else
527     {
528         if (T->lchild == NULL && T->rchild == NULL)
529         {
530             return 1;
531         }
532         else
533         {
534             int lbalance = BalanceDegreeBiTree(T->lchild);
535             int rbalance = BalanceDegreeBiTree(T->rchild);
536             return lbalance + rbalance;
537         }
538     }
539 }
540
541 //计算二叉树的深度
542 int DepthBiTree(TreeNode *T)
543 {
544     if (T == NULL)
545     {
546         return 0;
547     }
548     else
549     {
550         int ldepth = DepthBiTree(T->lchild);
551         int rdepth = DepthBiTree(T->rchild);
552         if (ldepth > rdepth)
553         {
554             return ldepth + 1;
555         }
556         else
557         {
558             return rdepth + 1;
559         }
560     }
561 }
562
563 //计算二叉树的节点数
564 int NodeNumBiTree(TreeNode *T)
565 {
566     if (T == NULL)
567     {
568         return 0;
569     }
570     else
571     {
572         int lnode = NodeNumBiTree(T->lchild);
573         int rnode = NodeNumBiTree(T->rchild);
574         return lnode + rnode + 1;
575     }
576 }
577
578 //计算二叉树的叶子数
579 int LeafNodeNumBiTree(TreeNode *T)
580 {
581     if (T == NULL)
582     {
583         return 0;
584     }
585     else
586     {
587         if (T->lchild == NULL && T->rchild == NULL)
588         {
589             return 1;
590         }
591         else
592         {
593             int lleaf = LeafNodeNumBiTree(T->lchild);
594             int rleaf = LeafNodeNumBiTree(T->rchild);
595             return lleaf + rleaf;
596         }
597     }
598 }
599
600 //计算二叉树的非叶节点数
601 int NonLeafNodeNumBiTree(TreeNode *T)
602 {
603     if (T == NULL)
604     {
605         return 0;
606     }
607     else
608     {
609         if (T->lchild == NULL && T->rchild == NULL)
610         {
611             return 0;
612         }
613         else
614         {
615             int lnonleaf = NonLeafNodeNumBiTree(T->lchild);
616             int rnonleaf = NonLeafNodeNumBiTree(T->rchild);
617             return lnonleaf + rnonleaf + 1;
618         }
619     }
620 }
621
622 //计算二叉树的满度数
623 int FullDegreeBiTree(TreeNode *T)
624 {
625     if (T == NULL)
626     {
627         return 0;
628     }
629     else
630     {
631         if (T->lchild == NULL && T->rchild == NULL)
632         {
633             return 1;
634         }
635         else
636         {
637             int lfull = FullDegreeBiTree(T->lchild);
638             int rfull = FullDegreeBiTree(T->rchild);
639             return lfull + rfull;
640         }
641     }
642 }
643
644 //计算二叉树的平衡度数
645 int BalanceDegreeBiTree(TreeNode *T)
646 {
647     if (T == NULL)
648     {
649         return 0;
650     }
651     else
652     {
653         if (T->lchild == NULL && T->rchild == NULL)
654         {
655             return 1;
656         }
657         else
658         {
659             int lbalance = BalanceDegreeBiTree(T->lchild);
660             int rbalance = BalanceDegreeBiTree(T->rchild);
661             return lbalance + rbalance;
662         }
663     }
664 }
665
666 //计算二叉树的深度
667 int DepthBiTree(TreeNode *T)
668 {
669     if (T == NULL)
670     {
671         return 0;
672     }
673     else
674     {
675         int ldepth = DepthBiTree(T->lchild);
676         int rdepth = DepthBiTree(T->rchild);
677         if (ldepth > rdepth)
678         {
679             return ldepth + 1;
680         }
681         else
682         {
683             return rdepth + 1;
684         }
685     }
686 }
687
688 //计算二叉树的节点数
689 int NodeNumBiTree(TreeNode *T)
690 {
691     if (T == NULL)
692     {
693         return 0;
694     }
695     else
696     {
697         int lnode = NodeNumBiTree(T->lchild);
698         int rnode = NodeNumBiTree(T->rchild);
699         return lnode + rnode + 1;
700     }
701 }
702
703 //计算二叉树的叶子数
704 int LeafNodeNumBiTree(TreeNode *T)
705 {
706     if (T == NULL)
707     {
708         return 0;
709     }
710     else
711     {
712         if (T->lchild == NULL && T->rchild == NULL)
713         {
714             return 1;
715         }
716         else
717         {
718             int lleaf = LeafNodeNumBiTree(T->lchild);
719             int rleaf = LeafNodeNumBiTree(T->rchild);
720             return lleaf + rleaf;
721         }
722     }
723 }
724
725 //计算二叉树的非叶节点数
726 int NonLeafNodeNumBiTree(TreeNode *T)
727 {
728     if (T == NULL)
729     {
730         return 0;
731     }
732     else
733     {
734         if (T->lchild == NULL && T->rchild == NULL)
735         {
736             return 0;
737         }
738         else
739         {
740             int lnonleaf = NonLeafNodeNumBiTree(T->lchild);
741             int rnonleaf = NonLeafNodeNumBiTree(T->rchild);
742             return lnonleaf + rnonleaf + 1;
743         }
744     }
745 }
746
747 //计算二叉树的满度数
748 int FullDegreeBiTree(TreeNode *T)
749 {
750     if (T == NULL)
751     {
752         return 0;
753     }
754     else
755     {
756         if (T->lchild == NULL && T->rchild == NULL)
757         {
758             return 1;
759         }
760         else
761         {
762             int lfull = FullDegreeBiTree(T->lchild);
763             int rfull = FullDegreeBiTree(T->rchild);
764             return lfull + rfull;
765         }
766     }
767 }
768
769 //计算二叉树的平衡度数
770 int BalanceDegreeBiTree(TreeNode *T)
771 {
772     if (T == NULL)
773     {
774         return 0;
775     }
776     else
777     {
778         if (T->lchild == NULL && T->rchild == NULL)
779         {
780             return 1;
781         }
782         else
783         {
784             int lbalance = BalanceDegreeBiTree(T->lchild);
785             int rbalance = BalanceDegreeBiTree(T->rchild);
786             return lbalance + rbalance;
787         }
788     }
789 }
790
791 //计算二叉树的深度
792 int DepthBiTree(TreeNode *T)
793 {
794     if (T == NULL)
795     {
796         return 0;
797     }
798     else
799     {
800         int ldepth = DepthBiTree(T->lchild);
801         int rdepth = DepthBiTree(T->rchild);
802         if (ldepth > rdepth)
803         {
804             return ldepth + 1;
805         }
806         else
807         {
808             return rdepth + 1;
809         }
810     }
811 }
812
813 //计算二叉树的节点数
814 int NodeNumBiTree(TreeNode *T)
815 {
816     if (T == NULL)
817     {
818         return 0;
819     }
820     else
821     {
822         int lnode = NodeNumBiTree(T->lchild);
823         int rnode = NodeNumBiTree(T->rchild);
824         return lnode + rnode + 1;
825     }
826 }
827
828 //计算二叉树的叶子数
829 int LeafNodeNumBiTree(TreeNode *T)
830 {
831     if (T == NULL)
832     {
833         return 0;
834     }
835     else
836     {
837         if (T->lchild == NULL && T->rchild == NULL)
838         {
839             return 1;
840         }
841         else
842         {
843             int lleaf = LeafNodeNumBiTree(T->lchild);
844             int rleaf = LeafNodeNumBiTree(T->rchild);
845             return lleaf + rleaf;
846         }
847     }
848 }
849
850 //计算二叉树的非叶节点数
851 int NonLeafNodeNumBiTree(TreeNode *T)
852 {
853     if (T == NULL)
854     {
855         return 0;
856     }
857     else
858     {
859         if (T->lchild == NULL && T->rchild == NULL)
860         {
861             return 0;
862         }
863         else
864         {
865             int lnonleaf = NonLeafNodeNumBiTree(T->lchild);
866             int rnonleaf = NonLeafNodeNumBiTree(T->rchild);
867             return lnonleaf + rnonleaf + 1;
868         }
869     }
870 }
871
872 //计算二叉树的满度数
873 int FullDegreeBiTree(TreeNode *T)
874 {
875     if (T == NULL)
876     {
877         return 0;
878     }
879     else
880     {
881         if (T->lchild == NULL && T->rchild == NULL)
882         {
883             return 1;
884         }
885         else
886         {
887             int lfull = FullDegreeBiTree(T->lchild);
888             int rfull = FullDegreeBiTree(T->rchild);
889             return lfull + rfull;
890         }
891     }
892 }
893
894 //计算二叉树的平衡度数
895 int BalanceDegreeBiTree(TreeNode *T)
896 {
897     if (T == NULL)
898     {
899         return 0;
900     }
901     else
902     {
903         if (T->lchild == NULL && T->rchild == NULL)
904         {
905             return 1;
906         }
907         else
908         {
909             int lbalance = BalanceDegreeBiTree(T->lchild);
910             int rbalance = BalanceDegreeBiTree(T->rchild);
911             return lbalance + rbalance;
912         }
913     }
914 }
915
916 //计算二叉树的深度
917 int DepthBiTree(TreeNode *T)
918 {
919     if (T == NULL)
920     {
921         return 0;
922     }
923     else
924     {
925         int ldepth = DepthBiTree(T->lchild);
926         int rdepth = DepthBiTree(T->rchild);
927         if (ldepth > rdepth)
928         {
929             return ldepth + 1;
930         }
931         else
932         {
933             return rdepth + 1;
934         }
935     }
936 }
937
938 //计算二叉树的节点数
939 int NodeNumBiTree(TreeNode *T)
940 {
941     if (T == NULL)
942     {
943         return 0;
944     }
945     else
946     {
947         int lnode = NodeNumBiTree(T->lchild);
948         int rnode = NodeNumBiTree(T->rchild);
949         return lnode + rnode + 1;
950     }
951 }
952
953 //计算二叉树的叶子数
954 int LeafNodeNumBiTree(TreeNode *T)
955 {
956     if (T == NULL)
957     {
958         return 0;
959     }
960     else
961     {
962         if (T->lchild == NULL && T->rchild == NULL)
963         {
964             return 1;
965         }
966         else
967         {
968             int lleaf = LeafNodeNumBiTree(T->lchild);
969             int rleaf = LeafNodeNumBiTree(T->rchild);
970             return lleaf + rleaf;
971         }
972     }
973 }
974
975 //计算二叉树的非叶节点数
976 int NonLeafNodeNumBiTree(TreeNode *T)
977 {
978     if (T == NULL)
979     {
980         return 0;
981     }
982     else
983     {
984         if (T->lchild == NULL && T->rchild == NULL)
985         {
986             return 0;
987         }
988         else
989         {
990             int lnonleaf = NonLeafNodeNumBiTree(T->lchild);
991             int rnonleaf = NonLeafNodeNumBiTree(T->rchild);
992             return lnonleaf + rnonleaf + 1;
993         }
994     }
995 }
996
997 //计算二叉树的满度数
998 int FullDegreeBiTree(TreeNode *T)
999 {
1000     if (T == NULL)
1001     {
1002         return 0;
1003     }
1004     else
1005     {
1006         if (T->lchild == NULL && T->rchild == NULL)
1007         {
1008             return 1;
1009         }
1010         else
1011         {
1012             int lfull = FullDegreeBiTree(T->lchild);
1013             int rfull = FullDegreeBiTree(T->rchild);
1014             return lfull + rfull;
1015         }
1016     }
1017 }
1018
1019 //计算二叉树的平衡度数
1020 int BalanceDegreeBiTree(TreeNode *T)
1021 {
1022     if (T == NULL)
1023     {
1024         return 0;
1025     }
1026     else
1027     {
1028         if (T->lchild == NULL && T->rchild == NULL)
1029         {
1030             return 1;
1031         }
1032         else
1033         {
1034             int lbalance = BalanceDegreeBiTree(T->lchild);
1035             int rbalance = BalanceDegreeBiTree(T->rchild);
1036             return lbalance + rbalance;
1037         }
1038     }
1039 }
1040
1041 //计算二叉树的深度
1042 int DepthBiTree(TreeNode *T)
1043 {
1044     if (T == NULL)
1045     {
1046         return 0;
1047     }
1048     else
1049     {
1050         int ldepth = DepthBiTree(T->lchild);
1051         int rdepth = DepthBiTree(T->rchild);
1052         if (ldepth > rdepth)
1053         {
1054             return ldepth + 1;
1055         }
1056         else
1057         {
1058             return rdepth + 1;
1059         }
1060     }
1061 }
1062
1063 //计算二叉树的节点数
1064 int NodeNumBiTree(TreeNode *T)
1065 {
1066     if (T == NULL)
1067     {
1068         return 0;
1069     }
1070     else
1071     {
1072         int lnode = NodeNumBiTree(T->lchild);
1073         int rnode = NodeNumBiTree(T->rchild);
1074         return lnode + rnode + 1;
1075     }
1076 }
1077
1078 //计算二叉树的叶子数
1079 int LeafNodeNumBiTree(TreeNode *T)
1080 {
1081     if (T == NULL)
1082     {
1083         return 0;
1084     }
1085     else
1086     {
1087         if (T->lchild == NULL && T->rchild == NULL)
1088         {
1089             return 1;
1090         }
1091         else
1092         {
1093             int lleaf = LeafNodeNumBiTree(T->lchild);
1094             int rleaf = LeafNodeNumBiTree(T->rchild);
1095             return lleaf + rleaf;
1096         }
1097     }
1098 }
1099
1100 //计算二叉树的非叶节点数
1101 int NonLeafNodeNumBiTree(TreeNode *T)
1102 {
1103     if (T == NULL)
1104     {
1105         return 0;
1106     }
1107     else
1108     {
1109         if (T->lchild == NULL && T->rchild == NULL)
1110         {
1111             return 0;
1112         }
1113         else
1114         {
1115             int lnonleaf = NonLeafNodeNumBiTree(T->lchild);
1116             int rnonleaf = NonLeafNodeNumBiTree(T->rchild);
1117             return lnonleaf + rnonleaf + 1;
1118         }
1119     }
1120 }
1121
1122 //计算二叉树的满度数
1123 int FullDegreeBiTree(TreeNode *T)
1124 {
1125     if (T == NULL)
1126     {
1127         return 0;
1128     }
1129     else
1130     {
1131         if (T->lchild == NULL && T->rchild == NULL)
1132         {
1133             return 1;
1134         }
1135         else
1136         {
1137             int lfull = FullDegreeBiTree(T->lchild);
1138             int rfull = FullDegreeBiTree(T->rchild);
1139             return lfull + rfull;
1140         }
1141     }
1142 }
1143
1144 //计算二叉树的平衡度数
1145 int BalanceDegreeBiTree(TreeNode *T)
1146 {
1147     if (T == NULL)
1148     {
1149         return 0;
1150     }
1151     else
1152     {
1153         if (T->lchild == NULL && T->rchild == NULL)
1154         {
1155             return 1;
1156         }
1157         else
1158         {
1159             int lbalance = BalanceDegreeBiTree(T->lchild);
1160             int rbalance = BalanceDegreeBiTree(T->rchild);
1161             return lbalance + rbalance;
1162         }
1163     }
1164 }
1165
1166 //计算二叉树的深度
1167 int DepthBiTree(TreeNode *T)
1168 {
1169     if (T == NULL)
1170     {
1171         return 0;
1172     }
1173     else
1174     {
1175         int ldepth = DepthBiTree(T->lchild);
1176         int rdepth = DepthBiTree(T->rchild);
1177         if (ldepth > rdepth)
1178         {
1179             return ldepth + 1;
1180         }
1181         else
1182         {
1183             return rdepth + 1;
1184         }
1185     }
1186 }
1187
1188 //计算二叉树的节点数
1189 int NodeNumBiTree(TreeNode *T)
1190 {
1191     if (T == NULL)
1192     {
1193         return 0;
1194     }
1195     else
1196     {
1197         int lnode = NodeNumBiTree(T->lchild);
1198         int rnode = NodeNumBiTree(T->rchild);
1199         return lnode + rnode + 1;
1200     }
1201 }
1202
1203 //计算二叉树的叶子数
1204 int LeafNodeNumBiTree(TreeNode *T)
1205 {
1206     if (T == NULL)
1207     {
1208         return 0;
1209     }
1210     else
1211     {
1212         if (T->lchild == NULL && T->rchild == NULL)
1213         {
1214             return 1;
1215         }
1216         else
1217         {
1218             int lleaf = LeafNodeNumBiTree(T->lchild);
1219             int rleaf = LeafNodeNumBiTree(T->rchild);
1220             return lleaf + rleaf;
1221         }
1222     }
1223 }
1224
1225 //计算二叉树的非叶节点数
1226 int NonLeafNode
```

```

24     }
25     else
26     {
27         *T = new TreeNode;
28         (*T)->data = ch;
29         cout << "input" << ch << "'s left son node:" ;
30         CreateBiTree(&((*T)->lchild));
31         cout << "input" << ch << "'s right son node:" ;
32         CreateBiTree((&(*T)->rchild));
33     }
34     return ;
35 }

36 //用一个数组分别记录偷根节点和不偷根节点时的最大值
37 class Solution {
38 public:
39     int rob(TreeNode* root) {
40         int * res = doRob(root);
41         return max(res[0],res[1]);
42     }
43
44     int * doRob(TreeNode * root)
45     {
46         int * res = new int [2];
47         res[0] = 0;
48         res[1] = 0;
49         if(root == NULL)
50             return res;
51         int* left = doRob(root->lchild);
52         int * right = doRob(root->rchild);
53         //不偷根节点，最大值为两个子树的最大值之和
54         res[0] = max(left[0],left[1])+max(right[0],right[1]);
55         //偷根节点，最大值为两个子树不包含根节点的最大值加上根节点的值
56         res[1] = left[0] + right[0] + root->data;
57         return res;
58     }
59 };
60
61
62 int main()
63 {
64     cout << "输入第一个节点的值,-1表示没有儿子节点:" << endl;
65     TreeNode* T;
66     CreateBiTree(&T); //加个引用创建之后，T会变化；如果使用上面的，调用函数之后T仍然是空的
67     Solution s;
68     int opt = s.rob(T);
69     cout << opt << endl;
70     system("pause");
71     return 0;
72 }

```

伪代码:

```

rob(TreeNode *p)
2     if p==NULL
3         return 0;
4     else
5         d(p) = max(d(l)+d(r),d(l1)+d(lr)+d(r1)+d(rr)+v(d))
6     end if

```

## Question 21

有向无环图中的单元最短路径，考虑有环和无环两种情况。

## Question 22

上一题中提到子问题定义太粗容易造成递归循环，这一题再给出隐马模型中将子问题定义的细的例子，并简要介绍维特比 (Viterbi) 算法。

隐马尔可夫模型有真实状态值和观测状态值， $a_{kl}$  表示真实状态从  $k$  到  $l$ ,  $a_k$  表示初始真实状态为  $k$ ,  $e_k(b)$  表示真实状态为  $k$  的情况下，观测状态为  $b$ 。解码问题是已知观测序列，寻找最可能的真实状态序列。

1. 定义子问题如下： $v_i = \max_{x_1, x_2, x_3, \dots, x_i} p(x_1, x_2, x_3, \dots, x_i, y_1, y_2, y_3, \dots, y_i)$ ，发现并无递归规律可循，因为子问题定义的范围很大；
2. 定义子问题如下： $v_i(k) = \max_{x_1, x_2, x_3, \dots, x_{i-1}} p(x_1, x_2, x_3, \dots, x_i = k, y_1, y_2, y_3, \dots, y_i)$ ，则可以使用动态规划算法 (Viterbi 算法) 求解，递推公式为： $v_i(k) = e_k(y_i) \max_l (a_{lk} v_{i-1}(l))$ 。

Viterbi's decoding algorithm: recursion

- First we rewrite  $\max_X P(X, Y)$  as:

$$\max_{x_n} \max_{x_{n-1}} \dots \max_{x_1} e_{x_n}(y_n) a_{x_{n-1} x_n} e_{x_{n-1}}(y_{n-1}) \dots a_{x_1 x_2} e_{x_1}(y_1) a_{0 x_1}$$

- Let denote  $v_i(k)$  as

$$\max_{x_{i-1}} \max_{x_1} e_k(y_i) a_{x_{i-1} k} e_{x_{i-1}}(y_{i-1}) \dots a_{x_1 x_2} e_{x_1}(y_1) a_{0 x_1}$$

Then we have:

$$\max_X P(X, Y) = \max_k v_n(k)$$

- We can also observe the following recursion:

$$v_i(k) = e_k(y_i) \max_l (a_{lk} v_{i-1}(l))$$

Viterbi's decoding algorithm

```
VITERBIDECODING(  $Y, a, e$  )
1: Initialize  $v_1(k) = a_{0k}e_k(y_1)$  for all state  $k$ ;
2: for  $i = 2$  to  $n$  do
3:   for each state  $k$  do
4:      $v_i(k) = e_k(y_i) \max_l (a_{lk} v_{i-1}(l))$ ;
5:      $ptr_i(k) = argmax_l (a_{lk} v_{i-1}(l))$ ;
6:   end for
7: end for
8:  $P(X^*, Y) = max_k(v_n(k))$ ;
9:  $x_n^* = argmax_k(v_n(k))$ ;
10: for  $i = n - 1$  to 1 do
11:    $x_i^* = ptr_{i-1}(x_{i+1}^*)$ ;
12: end for
13: return  $X$ ;
```

Dongbo Bu CS711008Z Algorithm Design and Analysis

Dongbo Bu CS711008Z Algorithm Design and Analysis

(a) 推导过程

(b) Viterbi 算法

图 12: Viterbi 算法及其推导

(关于隐马尔可夫的具体内容可以看第三部分机器学习中关于隐马尔可夫模型的介绍)

介绍完动态规划算法，下面的题将面向贪心算法

## Question 23

课程表调度

## Question 100

牛客网剑指 offer 中的题，和 leetcode 一样，面试手写代码基本上都是直接写类中的函数

## Part III

# 机器学习与深度学习基础模型与算法

机器学习模型主要分为两大类：生成式模型和判别式模型。

生成式模型是研究某一类的样本，研究这一类样本的特性利用极大似然方法： $\max_{\theta} p(D|M(\theta))$ ，即在什么参数的情况下，能够产生某一类的样本。比如贝叶斯公式  $p(w_i|x) = \frac{p(x|w_i)p(w_i)}{p(x)}$  中的  $p(x|w_i)$  可以根据极大似然估计得到参数值，进而得到分布。

判别式模型是研究所有的样本，利用极大后验方法： $\max_{\theta} p(M(\theta)|D)$ ，即在给定的所有样本的情况下，参数为多少的概率最大。比如深度学习中的神经网络，根据所有的样本来更新参数。

机器学习部分

## Question 1

隐马尔可夫模型隐马尔可夫模型主要分为三大问题：估值问题，解码问题，训练问题。

深度学习部分

## Question 6

手推 BP(Back Propagation) 算法。

## Question 7

网络训练的常见问题

1. 初始化网络的权重：可正可负，通常从一个均匀分布中随机选择初始值， $-w_0 < w < w_0$ ；
2. 正则化技术：为了防止网络出现 *overfitting* 的一种有效方法是采用一些正则化技术，如利用矩阵的 2-范数修正能量函数， $E_{new}(w) = E(w) + kw^Tw$ ，无论是哪种正则化方式，基本的思想都是希望通过限制权重的大小，使得模型不能任意拟合训练数据中的随机噪声；
3. 学习率太小，则收敛太慢，太大则不稳定；
4. 网络“训不动”：网络训不动分为梯度消失和网络麻痹。从上一题 BP 算法的推导来看，误差反向传播的过程中，误差会乘以激励函数的导数，是慢慢缩小的，会导致梯度消失。当这个导数趋于 0 的时候，误差会趋于 0，导致网络麻痹。

## Question 3

简单介绍 Hopfield 网络，BM, RBM, DBN, DBM。

在工业界研究 CNN 的时候，科学界主要在研究下面这些内容。

1. Hopfield 网络中各个节点地位等同，全连接起来都是输入输出节点，从初始状态开始运行到最终的平衡状态；

2. BM(Boltzmann Machine) 与 Hopfield 网络不同的是，BM 对节点功能做了区分，其中的一部分神经元是输入输出，受外界条件影响，另一部分视为隐藏节点，是一个深层网络；

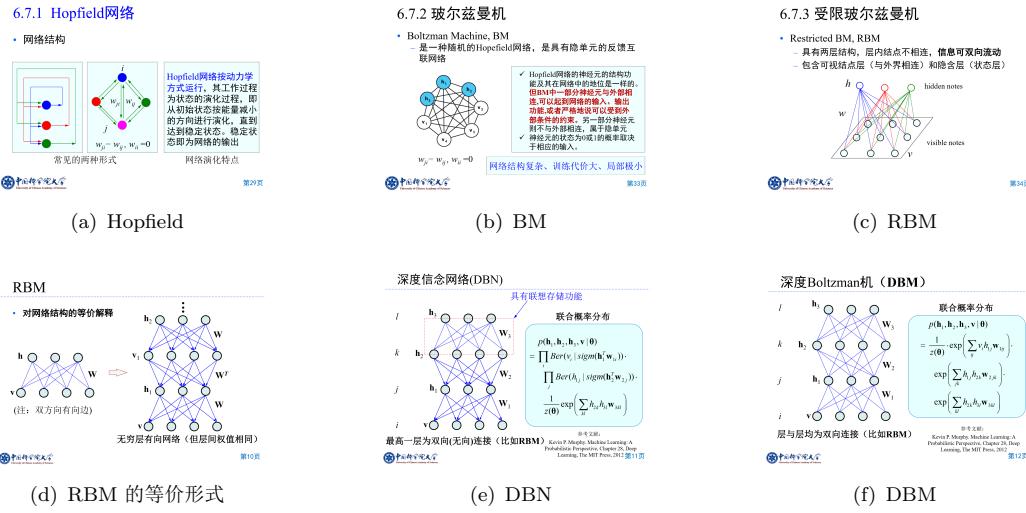


图 13: Hopfield, BM, RBM, DBN, DBM

3. *RBM (Restricted Boltzmann Machine)* 中，可视节点与隐藏节点相连（全连接），层内不连接。训练的时候需要用到隐藏节点与可视节点的联合分布（较复杂！），*RBM* 是一种很重要的特征表示方法，跟后续的自动编码器功能类似，是反馈神经网络的典型代表。

*RBM* 有一个等价的形式，如图中 (d) 所示，因为 *RBM* 是双向流动的，所以相当于一个无穷层有向网络，但层间权值相等。

4. *DBN (deep belief network)* 是玻尔兹曼机（双向）+一般的前向网络（单向）。

## Question 4

简单介绍 *CNN* 网络，*AutoEncoder*, *RNN*, *LSTM*。

1. *CNN* 实际上是前向神经网络的特例，是少量神经元的线性加权求和再激励，所以其反向传播过程与全连接前向神经网络类似。池化操作是在每个通道上做池化，添加了池化层的网络，其反向传播过程与前向神经网络不一样，待考究！

卷积操作可以大幅度减少参数，主要通过两个途径：(1) 局部连接：原因有二，一是视觉生理学相关研究普遍认为，人对外界的认知是从局部到全局的。视觉皮层的神经元就是局部接受信息的，即只响应某些特定区域的刺激；二是图像空间相关性，对图像而言，局部邻域内的像素联系较紧密，距离较远的像素相关性则较弱；(2) 权值共享。

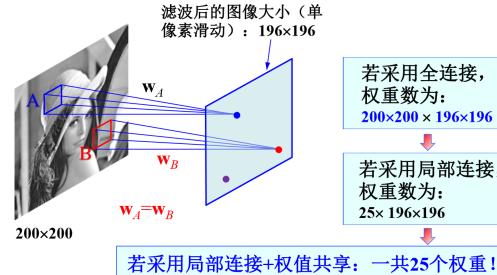
2. *AutoEncoder* 是一种尽可能重构输入信号的神经网络，让整个网络的输出与输入相等。在此网络中，隐含层则可以理解为用于记录数据的特征，像主成分分析中获得的主成分那样，因此这是一种典型的表示学习方法。训练过程如下图，训练完成之后，可以得到一个初始值（权重的初始化），以此训练每个 encoder，每个 encoder 之后都是原样本的一种特征表示。

3. *RNN*: 将一般的前向神经网络做个压缩，并延迟一步时间，就得到 *RNN* 的基本结构，按时间顺序全结点展开更容易看懂连接的方式。

*LSTM* 的数据流动如下：

考虑 *RNN*, *LSTM* 的不同之处：由前一时刻的输出（输出到下一时刻，不是输出网络） $h_{t-1}$ ，和这一时刻的输入  $x_{t-1}$ ，如何得到这一时刻的输出（输出到下一时刻，不是输出网络） $h_t$ ？

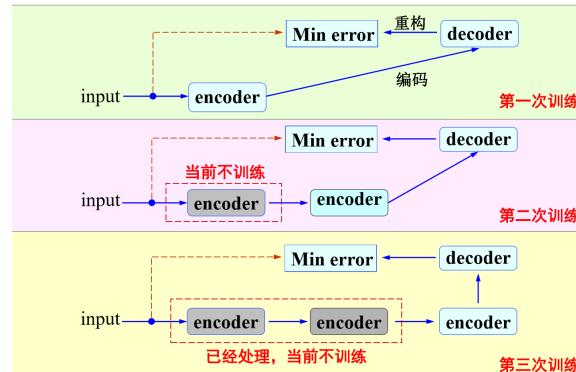
• 局部连接  
– 考虑 $5 \times 5$ 大小的滤波器且权值共享



第40页

图 14: 参数情况

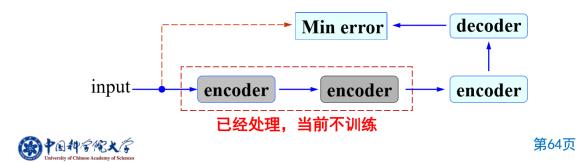
• 网络训练: (逐层静态进行)



(a) 1

• 网络训练:

- 对每个样本, 将第一层输出的 code 当成第二层的输入信号, **再次利用一个新的三层前向神经网络来最小化重构误差**, 得到第二层的权重参数 (获得第二个编码器); 同时得到样本在该层的code, 即原始号的第二个表达。
- 在训练当前层时, 其它层固定不动。完成当前编码和解码任务。前一次“编码”和“解码”均不考虑。
- 因此, 这一过程实质上是一个**静态的堆叠(stack)**过程
- 每次训练可以用BP算法对一个**三层前向网络**进行训练



(b) 2

图 15: AutoEncoder 训练过程

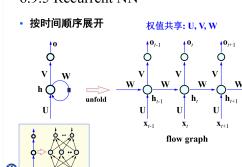
6.9.5 Recurrent NN

- 前向神经网络
  - $x$ : 网络输入 (向量)
  - $h$ : 隐含层输出 (向量)
  - $U$ : 输入至隐含层的连接权重矩阵 (input to hidden)
  - $V$ : 隐含层至输出层的连接权重矩阵 (hidden to output)



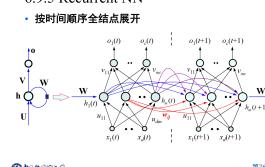
(a) 压缩前向神经网络

6.9.5 Recurrent NN



(b) 一步延迟等价形式

6.9.5 Recurrent NN



(c) 时间顺序节点全展开

图 16: RNN

## 6.9.6 LSTM

### • 结构描述——采用矩阵形式

遗忘门、输入门、输出门:

$$f_t = \text{sigmod}(\mathbf{b}_f + \mathbf{U}_f \mathbf{x}_t + \mathbf{W}_f \mathbf{h}_{t-1})$$

$$i_t = \text{sigmod}(\mathbf{b}_m + \mathbf{U}_{in} \mathbf{x}_t + \mathbf{W}_{in} \mathbf{h}_{t-1})$$

$$o_t = \text{sigmod}(\mathbf{b}_o + \mathbf{U}_o \mathbf{x}_t + \mathbf{W}_o \mathbf{h}_{t-1})$$

候选记忆（新贡献部分）：

$$c_t = \tanh(\mathbf{b} + \mathbf{U} \mathbf{x}_t + \mathbf{W} \mathbf{h}_{t-1})$$

Cell产生的新记忆:

$$s_t = f_t \otimes s_{t-1} + i_t \otimes c_t$$

Cell的输出:

$$h_t = o_t \otimes \tanh(s_t)$$

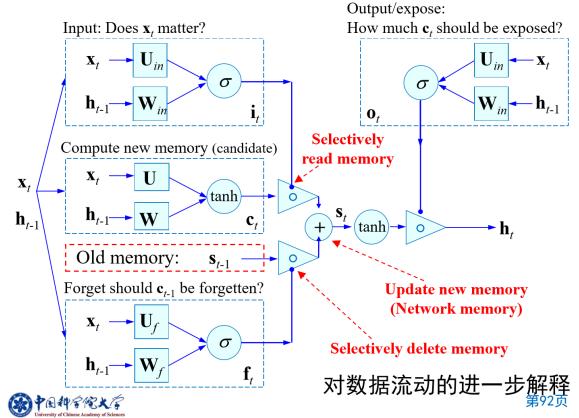
网络的输出:

$$z_t = \text{softmax}(\mathbf{V} \mathbf{h}_t + \mathbf{c})$$



第91页

(a) 1



(b) 2

图 17: LSTM

1.RNN: 直接计算  $h_t = \tanh(b + Ux_t + Wh_{t-1})$ , 只需训练  $U, W$ ;

2.LSTM: 先计算  $c_t = \tanh(b + Ux_t + Wh_{t-1})$  作为候选记忆,  $s_{t-1}$  是上一时刻的记忆;

计算三个权重  $(0|1)f_t, i_t, o_t$ , 表示多大程度上, 公式如图;

计算这一时刻的记忆  $s_t$ , 公式如图;

最后计算这一时刻的输出  $h_t$ , 公式如图。需要训练  $U, W, U_f, W_f, U_{in}, W_{in}, U_o, W_o$ 。

## Part IV

## 视频分析与处理相关知识点