

# 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)

2020 年 9 月 23 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:

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

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  $\|A\|_2 = \|PA\|_2 = \|AQ\|_2 = \|PAQ\|_2$ , 即矩阵的  $2-$  范数和  $F-$  范数是正交不变量, 但  $1-$  范数不是; (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, 多提一个矩阵的  $2,1$ -范数,  $\|W\| = \sum_{i=1}^m \sqrt{\sum_{j=1}^n w_{ij}^2}$ , 先计算每一行的  $2-$  范数, 变为向量后再计算向量的  $1-$  范数。

## 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。

## Question 3

### 矩阵分解

下面两种分解经常用于线性方程 (未知量等于方程个数) 的求解:

1. 设矩阵  $A \in R^{n*n}$ , 若  $A$  能分解为一个下三角矩阵  $L$  和一个上三角矩阵  $U$  的乘积, 即  $A = LU$ , 则这种分解成为矩阵  $A$  的三角分解。当  $L$  为单位下三角矩阵 (主对角元素全为 1) 时称为 *Dollittle* 分解; 当  $U$  为单位上三角矩阵 (主对角元素全为 1) 时称为 *Crout* 分解

2. 设矩阵  $A \in R^{n*n}$  为对称正定阵, 则存在一个非奇异下三角矩阵  $L$ , 使得  $A = LL^T$ , 当限定  $L$  的对角元素为正时, 这种分解是唯一的。

设矩阵  $A \in R^{n*n}$  为对称正定阵, 则存在惟一的分解,  $A = LDL^T$ , 其中  $L$  是单位下三角矩阵,  $D$  为对角矩阵, 且  $D$  的对角元素都是正数。

下面的内容关于矩阵特征值和特征向量的求解:

1. 乘幂法: 计算矩阵的最大特征值及对应的特征向量, 可以接着用降价法求矩阵的次大特征值和相应的特征向量;

2. 反幂法: 计算非奇异矩阵按模最小特征值及其特征向量。

3. *Givens* 变换: 对于任意  $x = (x_1, x_2, \dots, x_n) \in R^n$ , 当  $x_i$  不等于零时, 可以经过一次平面旋转变换将其化为零, 并同时将第  $j$  个分量变为  $Gx_j = (x_i^2 + x_j^2)^{\frac{1}{2}}$ , 而其他分量保持不变。

4. *Jacobi* 方法是一种求实对称矩阵的全部特征值和相应特征向量的方法 (如果矩阵阶数不高可以使用)。

5. *Householder* 变换 (反射变换)

设  $w \in R^n$ , 且  $\|w\|_2 = 1$ , 则矩阵  $H = I - 2ww^T$  称为 *Householder* 矩阵或反射矩阵, 这里的  $I$  为  $n$  阶单位矩阵。 $\forall x = (x_1, x_2, \dots, x_n)^T \in R^n$ , 当其后边  $n - r + 1$  个分量不全为零时, 可以经过一次反射变换将其后  $n - r$  的分量化为 0, 且第  $r$  个分量变为:  $+(-)(\sum_{j=r}^n |x_j|^2)^{\frac{1}{2}}$ , 而其余分量保持不变。

*Givens* 变换是把向量中的一个元素变成 0, *Householder* 变换是把向量中多个元素变为 0

6. 矩阵的  $QR$  分解 (用来求特征值)

设  $A \in R^{n*n}$ , 则存在正交矩阵  $Q$  和上三角矩阵  $R$ , 使得  $A = QR$ , 并且在  $A$  是非奇异矩阵,  $R$  的对角元均大于零的条件下, 分解是唯一的。

一般的计算过程为: 先利用 *Householder* 变换将原矩阵化为上 *Hessenberg* (矩阵的下次对角线下方元素均为零), 对于  $n$  维 *Hessenberg* 矩阵, 通常用  $n - 1$  个 *Givens* 变换阵将它化为上三角阵, 然后即可使用  $QR$  分解求特征值, 求得特征值之后, 再利用反幂求相应的特征向量。

下面再补个一般的矩阵分解 (*schur* 分解):

1.  $A$  为实矩阵, 则  $A = U^T S U$ , 即  $A$  正交相似于  $S$ ,  $S$  对角线部分都是矩阵块, 矩阵块下方都是零 (对角线为块的上三角阵);

2.  $A$  为复矩阵,  $A = U^T S U$ ,  $A$ 酉相似于  $S$ ,  $S$  为上三角阵。

## Question 4

### 多元泰勒展开和视频光流之间的关系

一个视频的光流是衡量视频相邻帧之间的变化量，假设视频的规模是  $T * W * H$ ，则计算出来的光流也是  $T$  帧，且尺寸是  $W * H$ ，但只有两通道，分别存储每个像素在  $x$  方向和  $y$  方向的变化量，具体原理如下（先给出多元泰勒公式）：

$$f(x + \delta x) = f(x) + f'(x)\delta x + \frac{f''(x)}{2!}(\delta x)^2 + o((\delta x)^2)$$
$$F(X + \delta X) = F(X) + \nabla F \cdot \delta X + \frac{1}{2}\delta X^T H \delta X$$

再考虑视频光流的计算，视频光流的计算基于如下两个假设：(1) 连续的两帧图像之间，目标像素灰度值不变；(2) 相邻的像素之间有相似的运动。假设某一帧图像在  $t$  时刻  $(x, y)$  处的像素值为  $I(x, y, t)$ ， $d_t$  时间后，该像素移动到  $(x + d_x, y + d_y)$ ，此时对应的像素值为  $I(x + d_x, y + d_y, t + d_t)$ ，根据假设有：

$$I(x, y, t) = I(x + d_x, y + d_y, t + d_t)$$

由于相邻帧的时间间隔非常小，则位移也是无穷小量，可以用多元泰勒公式进行展开：

$$I(x + d_x, y + d_y, t + d_t) = I(x, y, t) + \nabla I^T \delta X + \epsilon$$
$$= I(x, y, t) + \frac{\partial I}{\partial x} \delta x + \frac{\partial I}{\partial y} \delta y + \frac{\partial I}{\partial t} \delta t + \epsilon$$

两边同时除以  $\delta t$ ：

$$\frac{\partial I}{\partial x} \frac{\delta x}{\delta t} + \frac{\partial I}{\partial y} \frac{\delta y}{\delta t} + \frac{\partial I}{\partial t} \frac{\delta t}{\delta t} + \frac{\epsilon}{\delta t} = 0$$

两边同时取极限：

$$f_x u + f_y v + f_t = 0$$

其中  $\frac{\partial I}{\partial x} = f_x$ ,  $\frac{\partial I}{\partial y} = f_y$ ,  $\frac{\partial I}{\partial t} = f_t$ ,  $\frac{\delta x}{\delta t} = u$ ,  $\frac{\delta y}{\delta t} = v$ , 整体称为光流方程,  $u, v$  是需要求解的量 (Lucas-Kanada 算法)。

## 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）。

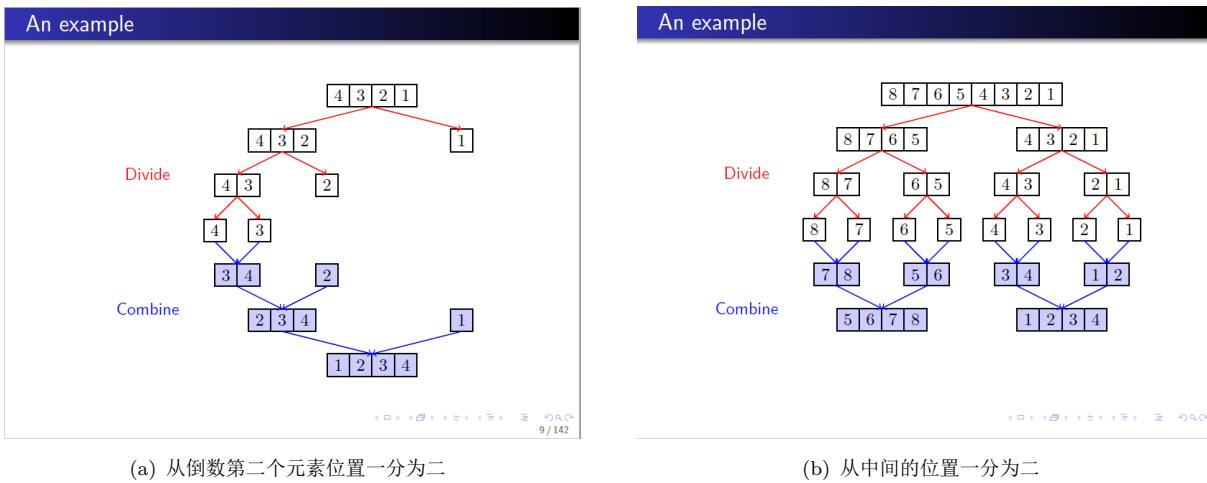


图 1：采用分治思想排序

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。

```

1 #include <iostream>
3 #include <stdio.h>
5 using namespace std;

```

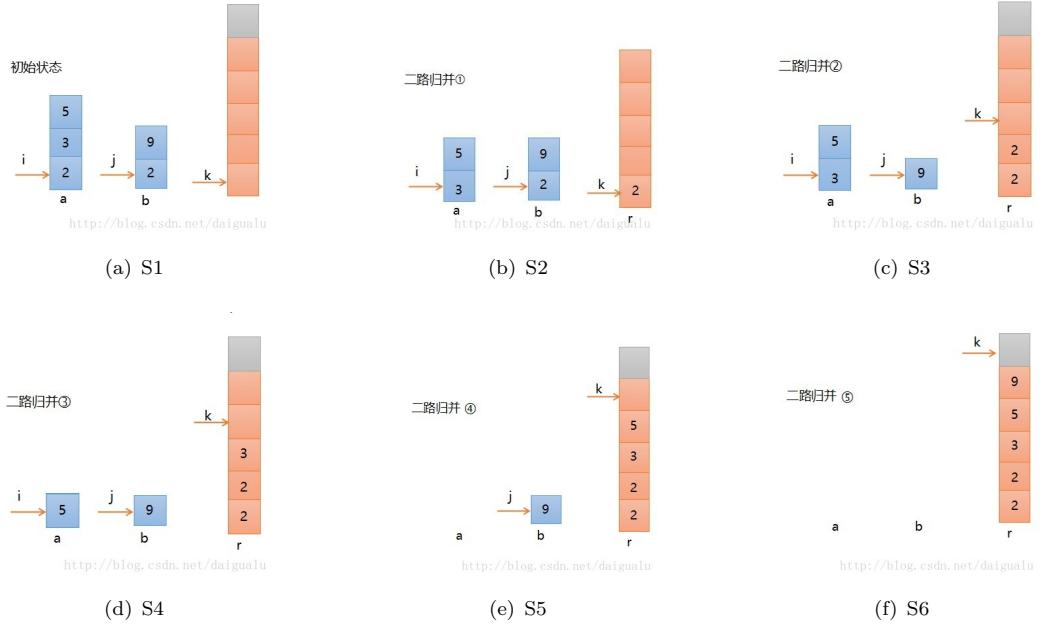


图 2: 二路归并过程

```

7 void merge(int a[], int left, int mid, int right, int b[])
{
9     int i = mid;
11    int j = right;
13    int k = 0;
14    while (i >= left && j >= mid+1)
15    {
16        if(a[i] > a[j])
17        {
18            b[k++] = a[i--];
19        }
20        else
21        {
22            b[k++] = a[j--];
23        }
24    }
25    while (i >= left)
26    {
27        b[k++] = a[i--];
28    }
29    while (j >= mid+1)
30    {
31        b[k++] = a[j--];
32    }
33    for (i = 0; i < k; i++)
34    {
35        a[right - i] = b[i];
36    }
37}
38 void solve(int a[], int left, int right, int b[])
39 {
40     if(right > left)
41     {
42         int mid = (right+left) / 2;
43         merge(a, left, mid, right, b);
44         solve(a, left, mid, b);
45         solve(a, mid+1, right, b);
46     }
47 }

```

```

43     solve(a, left , mid,b);
44     solve(a,mid + 1, right ,b);
45     merge(a, left , mid, right ,b);
46 }
47
48 int main()
49 {
50     long int n;//数组维度
51     scanf("%d", &n);
52     int *a = new int[n];
53     int *b = new int[n];
54     for(long int i=0;i<n; i++)
55     {
56         scanf("%d", &a[i]); //scanf的速度要比cin的速度快
57     }
58     solve(a,0,n-1,b); //归并排序
59
60     for(int i = 0; i<n; i++)
61         cout<<a[i]<<' ';
62     return 0;
63 }
64 }
```

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？)

```

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

```

36 long int solve(int a[],int left , int right,int b[])
{
38     if(right > left)
{
40         int mid = (right+left) / 2;
        solve(a, left , mid,b);
42         solve(a,mid + 1, right ,b);
        merge(a, left , mid, right ,b);
44     }
}
46
48 int main()
{
49     long int n;// 数组维度
50     scanf("%d" , &n);
51     int *a = new int[n];
52     int *b = new int[n];
53     for(long int i=0;i<n; i++)
{
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 }
62
63 }
```

## 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：

```

2     vector<vector<int> >vec;
3     vector<int>a;
4     a.push_back(1);
5     a.push_back(2);
6     vector<int>b;
7     b.push_back(3);
8     b.push_back(4);
9     vec.push_back(a);
10    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++)
```

```
1   for (int j=0;j<array[0].size();j++)
7     array[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<array[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 arrayszie;//数组规模
2 scanf("%d",&arrayszie);//输入数组规模，scanf比cin快很多
3 int *arry=new int[arrayszie];//数组名是指针
4 for(int count=0;count<arrayszie;count++)
5   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;
5 long int merge(int a[], int left, int mid, int right,int b[])
{
7   int i = mid;
```

```

9   int j = right;
10  long int lcount = 0;
11  while ( i >= left && j > mid)
12  {
13      if(a[ i ] > (long long) 3 * a[ j ])
14      {
15          lcount += j - mid;
16          i--;
17      }
18      else
19      {
20          j--;
21      }
22  }
23  i = mid;
24  j = right;
25  int k = 0;
26  while ( i >= left && j > mid)
27  {
28      if(a[ i ] > a[ j ])
29      {
30          b[ k++ ] = a[ i-- ];
31      }
32      else
33      {
34          b[ k++ ] = a[ j-- ];
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
64 long int InversePairs(int a[],int len)
65 {
66     int *b=new int [len];
67     long int count=solve(a,0 ,len-1,b);
68     delete [] b;
69     return count;
70 }
71 int main()

```

```

73 {
74     long int n;//数组维度
75     int *array;//数组
76     scanf("%d", &n);
77     array = new int[n];
78     for (long int i=0;i<n; i++)
79         scanf("%d", &array[i]);
80
81     long int count = InversePairs(array, n);
82     printf("%d", count);
83     return 0;
84 }
```

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：**与归并排序按照数组的下标来分治不同，快速排序按照数组的值来分治，即选取 pivot 来排序一个数组。可以和后面的牛客网剑指 offer “最小的 K 个数”一题联系起来。首先设定一个分界值 pivot，通过该分界值将数组分成左右两部分。将大于或等于分界值的数据集中到数组右边，小于分界值的数据集中到数组的左边。然后，左边和右边的数据可以独立排序。对于左侧的数据，又可以取一个分界值，将该部分数据分成左右两部分，同样在左边放置较小值，右边放置较大值。右侧的数据也可以做类似处理。重复上述过程，可以看出，这是一个递归定义。通过递归将左侧部分排好序后，再递归排好右侧部分的顺序。当左、右两个部分各数据排序完成后，整个数组的排序也就完成了。

技术路线为：

(1) 设置两个变量 *i*,*j*, 排序开始的时候: *i* = 0, *j* = *N* - 1; (2) 以第一个数组元素作为关键数据，赋值给 *key*, 即 *key* = *A*[0]; (3) 从 *j* 开始向前搜索，即由后开始向前搜索 (*j* --), 找到第一个小于 *key* 的值 *A*[*j*]，将 *A*[*j*] 和 *A*[*i*] 的值交换; (4) 从 *i* 开始向后搜索，即由前开始向后搜索 (*i* ++), 找到第一个大于 *key* 的 *A*[*i*]，将 *A*[*i*] 和 *A*[*j*] 的值交换; (5) 重复第 (3)(4) 步，直到 *i* = *j*,((3)(4) 步中，没找到符合条件的值，即 (4) 中 *A*[*j*] 不小于 *key*, (4) 中 *A*[*i*] 不大于 *key* 的时候改变 *j*,*i* 的值，使得 *j* = *j* - 1, *i* = *i* + 1，直至找到为止。找到符合条件的值，进行交换的时候 *i*,*j* 指针位置不变。另外，*i* == *j* 这一过程一定正好是 *i* + 或 *j* - 完成的时候，此时令循环结束)。

代码为：

```

1 #include<iostream>
2
3 using namespace std;
4
5 void QuickSort(int a[], int start, int end)
6 {
7     if (start >= end)
8     {
9         return;
10    }
```

```

11 int i = start;
12 int j = end;
13 int pivot = a[start];
14
15 while(i < j)
16 {
17     while(a[j] > pivot && j > i)
18     {
19         j--;
20     }
21     int temp1 = a[i];
22     a[i] = a[j];
23     a[j] = temp1;
24
25     while(a[i] < pivot && i < j)
26     {
27         i++;
28     }
29     int temp2 = a[i];
30     a[i] = a[j];
31     a[j] = temp2;
32
33     QuickSort(a, start, i - 1);
34     QuickSort(a, j + 1, end); // i 和 j 相等
35 }
36 }
37
38 int main()
39 {
40     int a[5] = {5, 4, 3, 2, 1};
41     QuickSort(a, 0, 4);
42     for(int i = 0; i < 5; i++)
43     {
44         cout << a[i] << endl;
45     }
46     system("pause");
47     return 0;
48 }
```

## 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，一个是指数组的下标分治，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： $T(n) = aT(\frac{n}{b}) + O(d^n)$ 。

结论见下图 (卜东波老师的课上 slides)。

## Master theorem

### Theorem

Let  $T(n)$  be defined by  $T(n) = aT(\frac{n}{b}) + O(n^d)$  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，时间复杂度为  $O(n)$ 。

快速排序使用了分治法的策略。它的基本思想是，选择一个基准数（一般称之为枢纽元），通过一趟排序将要排序的数据分割成独立的两部分：在枢纽元左边的所有元素都不比它大，右边所有元素都比它大，此时枢纽元就处在它应该在的正确位置上了。在本问题中，假设有  $N$  个数存储在数组  $a$  中。我们从  $a$  中随机找出一个元素作为枢纽元，把数组分为两部分。其中左边元素都不比枢纽元大，右边元素都不比枢纽元小。此时枢纽元所在的位置记为  $mid$ 。如果右半边（包括  $a[mid]$ ）的长度恰好为  $k$ ，说明  $a[mid]$  就是需要的第  $k$  大元素，直接返回  $a[mid]$ 。如果右半边（包括  $a[mid]$ ）的长度大于  $k$ ，说明要寻找的第  $k$  大元素就在右半边，往右半边寻找。如果右半边（包括  $a[mid]$ ）的长度小于  $k$ ，说明要寻找的第  $k$  大元素就在左半边，往左半边寻找。（下面代码没有调试过，可能会报错）

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

int divide(int a[], int start, int end)
{
    if (start >= end)
    {
        return;
    }
    int i = start;
    int j = end;
    int pivot = a[start];

    while (i < j)
    {
        while (a[j] > pivot && j > i)
        {
            j--;
        }
        int temp1 = a[i];
        a[i] = a[j];
        a[j] = temp1;

        while (a[i] < pivot && i < j)
        {
            i++;
        }
        int temp2 = a[i];
        a[i] = a[j];
        a[j] = temp2;
    }
    return i; // 此时 i=j
}

int findKMax(int a[], int low, int high, int k)
{
    int mid = divide(a, low, high); // 包括 a[mid] 的右半边长度
    int length_of_right = high - mid + 1;
    if (length_of_right == k)
        return a[mid];
    else if (length_of_right > k)
    {
        // 右半边长度比 k 长，说明第 k 大的元素还在右半边，因此在右半边找
        return findKMax(a, mid + 1, high, k);
    }
}
```

```

46     }
47     else
48     {
49         return findKMax(a, low, mid - 1, k - length_of_right);
50     }
51 }
52 int main()
53 {
54     int A[] = { 1,2,2,2,3,3,3 };
55     int n=7,k = 3;
56     int result= findMaxK(A, 0,n-1, k);
57     cout << "第" << k << "大的数字为" << result << endl;
58     system("pause");
59     return 0;
60 }
```

## Question 7

leetcode 第 33 题，搜索旋转排序数组。问题描述：假设按照升序排序的数组在预先未知的某个点上进行了旋转。例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2]。搜索一个给定的目标值，如果数组中存在这个目标值，则返回它的索引，否则返回-1。可以假设数组中不存在重复的元素，算法时间复杂度必须是  $O(\log n)$  级别。

```

1 public:
2     int search(vector<int>& nums, int target) {
3 //用二分法，先判断左右两边哪一边是有序的，再判断是否在有序的列表之内
4         if (nums.size() <= 0)
5             return -1;
6         int left = 0;
7         int right = nums.size() - 1;
8         int mid;
9         while(left < right){
10             mid = (right - left)/2 + left;
11             if(nums[mid] == target)
12                 return mid;
13
14             // 如果中间的值大于最左边的值，说明左边有序
15             if(nums[mid] > nums[left])
16             {
17                 if (nums[left] <= target&&target <= nums[mid])
18                     right = mid;
19                 else
20                     // 这里 +1，因为上面是 <= 符号
21                     left = mid + 1;
22             }
23             // 否则右边有序
24             else
25             {
26                 // 注意：这里必须是 mid+1，因为根据我们的比较方式，mid属于左边的序列
27                 if (nums[mid+1] <= target&&target <= nums[right])
28                     left = mid + 1;
29                 else
30                     right = mid;
31             }
32         }
33         if(nums[left] == target)
34             return left;
35     }
```

```
37     }  
38 };
```

c

## Question 8

leetcode 第 153 题，寻找旋转排序数组中的最小值。问题描述：假设按照升序排序的数组在预先未知的某个点上进行了旋转。(例如，数组 [0,1,2,4,5,6,7] 可能变为 [4,5,6,7,0,1,2])。找出其中最小的元素。可以假设数组中不存在重复元素。

```
1 class Solution {  
2 public:  
3     int findMin(vector<int>& nums) {  
4         //用二分法，先判断左右两边哪一边是无序的，再判断是否在有序的列表之内  
5         int min = nums[0];  
6         if (nums.size() <= 0)  
7             return -1;  
8         int left = 0;  
9         int right = nums.size() - 1;  
10        int mid;  
11        while(left < right)  
12        {  
13            if (right == left+1)  
14            {  
15                if (nums[right]<min)  
16                    min = nums[right];  
17            }  
18            mid = (right - left)/2 + left;  
19            if (nums[mid]<min)  
20                min = nums[mid];  
21  
22            // 如果中间的值大于最左边的值，说明左边有序  
23            if (nums[mid] > nums[left])  
24            {  
25                left = mid;// 此处与33题不一样，此处一定是mid，不能+1  
26            }  
27            // 否则右边有序  
28            else  
29            {  
30                right = mid;  
31            }  
32        }  
33        return min;  
34    }  
};
```

与上一题不一样，这道题是要找旋转数组的最小值，最小值是要找无序的部分(最小值一定在无序的部分)。当只剩两个元素的时候，右边的数会比较小。或者写成下面形式：

```
1 class Solution {  
2 public int findMin(int[] nums) {  
3     int l = 0, r = nums.length - 1;  
4     while (l < r) {  
5         int mid = l + (r - l) / 2;  
6         if (nums[mid] > nums[r])
```

```
7         l = mid + 1;
8     else
9         r = mid;
10    }
11   return nums[1];
12 }
13 }
```

## 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：

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

4 using namespace std;

6 // 定义二叉树结构体
struct TreeNode{
8     int data;
10    TreeNode *lchild;
10    TreeNode *rchild;
11};

12 /*
14 // 先序创建二叉树
TreeNode* CreateBiTree(TreeNode *T)
16 {
17     int ch;
18     cin >> ch;
19     if (ch == -1)
20     {
21         T = NULL;
22         return T;
23     }
24     else
25     {
26         T = new TreeNode;
27         T->val = ch;
28         cout << "input" << ch << "'s left son node:";
29         CreateBiTree(T->left);
30         cout << "input" << ch << "'s right son node:";
31         CreateBiTree(T->right);
32     }
33 }
```

```

32     }
33     return T;
34 }
35 */
36 //上面这段创建树的代码创建好树再返回，跟下面的相比少了指针
37
38 //先序创建二叉树
39 void CreateBiTree(TreeNode **T)
40 {
41     int ch;
42     cin >> ch;
43     if (ch == -1)
44     {
45         *T = NULL;
46         return;
47     }
48     else
49     {
50         *T = new TreeNode;
51         (*T)->data = ch;
52         cout << "input" << ch << "'s left son node:" ;
53         CreateBiTree(&((*T)->lchild));
54         cout << "input" << ch << "'s right son node:" ;
55         CreateBiTree(&((*T)->rchild));
56     }
57     return;
58 }
59
60 //先序遍历
61 void PreOrderBiTree(TreeNode *T)
62 {
63     if (T == NULL)
64     {
65         return;
66     }
67     else
68     {
69         cout << T->data << " ";
70         PreOrderBiTree(T->lchild);
71         PreOrderBiTree(T->rchild);
72     }
73 }
74
75 //中序遍历
76 void MiddleOrderBiTree(TreeNode *T)
77 {
78     if (T == NULL)
79     {
80         return;
81     }
82     else
83     {
84         MiddleOrderBiTree(T->lchild);
85         cout << T->data << " ";
86         MiddleOrderBiTree(T->rchild);
87     }
88 }
89
90 //后序遍历
91 void PostOrderBiTree(TreeNode *T)
92 {
93     if (T == NULL)
94     {
95         return;
96     }

```

```

98     else
99     {
100        PostOrderBiTree(T->lchild);
101        PostOrderBiTree(T->rchild);
102        cout << T->data << " ";
103    }
104
105 //树的深度
106 int TreeDeep(TreeNode *T)
107 {
108    int deep = 0;
109    if (T != NULL)
110    {
111        int leftdeep = TreeDeep(T->lchild);
112        int rightdeep = TreeDeep(T->rchild);
113        deep = leftdeep >= rightdeep?leftdeep+1:rightdeep+1;
114    }
115    return deep;
116}
117
118 //叶子节点个数
119 int LeafCount(TreeNode *T)
120 {
121    static int count;
122    if (T != NULL)
123    {
124        if (T->lchild == NULL && T->rchild == NULL)
125        {
126            count++;
127        }
128        LeafCount(T->lchild);
129        LeafCount(T->rchild);
130    }
131    return count;
132}
133
134 int main()
135 {
136    cout << "输入第一个节点的值,-1表示没有儿子节点:" << endl;
137    TreeNode* T;
138    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
139    cout << T->data << endl;
140    system("pause");
141    return 0;
142}

```

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需要返回的，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>

4 using namespace std;
5
6 void f1(int point)
7 {
8     point = 1;
9 }
10
11 void f2(int * point)
12 {

```

```

14     *point = 1;
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。

```

1 int maxDistance = 0;
2 int maxDistance(TreeNode *T)
3 {
4     int distl = 0;
5     int distr = 0;
6     if (T == NULL)
7     {
8         return 0;
9     }
10    if (T->lchild != NULL)
11        distl = maxDistance(T->lchild) + 1;
12    if (T->rchild != NULL)
13        distr = maxDistance(T->rchild) + 1;
14    if (distl + distr > maxDistance)
15        maxDistance = distl + distr;
16    return max(distl, distr);
17 }
```

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：

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

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:   +  $p_{i-1}p_kp_j;$ 
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。

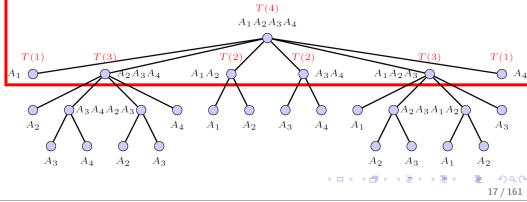
假设当包空  $W$ ，前  $i$  个物品的最优价值为  $OPT(i, W)$ ，则有递推式： $OPT(i, W) = \max\{OPT(i - 1, W), OPT(i - 1, W - w_i) + v_i\}$ ，下面给出伪代码及某个背包问题的示例：

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: Input:  $p, W$ 
2:  $OPT[0, w] = 0$ ;
3: end for
4: for  $i = 1$  to  $n$  do
5:   for  $w = 0$  to  $W$  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

$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$
$v_1 = 2$	$v_2 = 3$	$v_3 = 4$	$v_4 = 5$	$v_5 = 6$
$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$
$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$

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

(b) Step1

### Step 2

$OPT[1, 2] = \max\{$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[1, 1](= 2)$ ,	$i = 2$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[1, 2] + V_2 (= 2+2)$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$\vdots$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$

(c) Step2

**Step 3**

$w = 0$	$1$	$2$	$3$	$4$	$5$	$6$
$i = 3$	$0$	$0$	$2$	$3$	$4$	$5$
$i = 2$	$0$	$0$	$2$	$3$	$4$	$5$
$i = 1$	$0$	$0$	$2$	$2$	$2$	$2$
$i = 0$	$0$	$0$	$0$	$0$	$0$	$0$

(d) Step3

### Step 4

$OPT[2, 3] = \max\{$	$i = 3$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[2, 2](= 4)$ ,	$i = 2$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[2, 3] + V_3 (= 2+3)$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$\vdots$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$

(e) Step4

### Backtracking

$OPT[3, 6] = \max\{$	$i = 3$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[2, 6](= 4)$ ,	$i = 2$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[3, 6] + V_3 (= 2+3)$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$\vdots$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$

Decision: Select item 3

$OPT[2, 5] = \max\{$	$i = 3$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[2, 4](= 4)$ ,	$i = 2$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[2, 5] + V_3 (= 2+3)$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$\vdots$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$

Decision: Select item 2

$OPT[1, 5] = \max\{$	$i = 3$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[1, 4](= 4)$ ,	$i = 2$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$OPT[1, 5] + V_2 (= 2+2)$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$\vdots$	$i = 1$	$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$

(f) Step5

图 6: 伪代码及递归公式示例

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>
2 #include <string.h>
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>
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>>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(n(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;
```

```

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

34 //动态规划求解，实际上在计算一个三维矩阵
35 for(int i =1;i<=k; i++)
36 {
37     for(int j = 1;j<=M; j++)
38     {
39         for(int s = 1;s<=N; s++)
40         {
41             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
42                 OPT[i][j][s] = OPT[i-1][j][s];
43             else
44                 OPT[i][j][s] = max(OPT[i-1][j][s],OPT[i-1][j-m[i-1]][s-n[i-1]]+u[i-1]);
45         }
46     }
47 }
48 int O = OPT[k][M][N];

50 cout<<O<<endl;
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
```

```

13 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)
14 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)
15 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*[M+1];//二维数组
17 int **newOPT = new int*[M+1];//二维数组
18 for(int i=0;i<k; i++)
19 {
20     scanf("%d",&m[i]); //输入各个任务的空间
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 *ocurrance* 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': OCCURE-NCE

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

- 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 '|'. 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 '|'. Then it suffices to align  $T[1..m-1]$  and  $S[1..n]$ .

Match/Mutation	Insertion	Deletion
S': OCURRANCE	S': OCURRANC E	S': OCURRANCE -
T': OCCURRENCE	T': OCCURRENCE -	T': OCCURRENCE E

(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:

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

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。

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

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	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

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 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

## 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：最长子序列问题实际上是序列匹配的特例，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  $T[1..n]$ .

### 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  $\text{OPT}(i, j)$ .
- Then we have the following recursion:

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

(a) 递推过程分析

(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：

1. 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);
2. 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):

```
#include <iostream>
2 #include <string>
# include <string.h>
4 #include <algorithm>
# include <stdio.h>
6
using namespace std;
8
int main()
{
    string str1,str2;
12    getline(cin,str1);
    getline(cin,str2);
14    int m = str1.length(); // 第一个字符串的长度
    int n = str2.length(); // 第二个字符串的长度
16    int OPT[m+1][n+1]; // 0 矩阵
    memset(OPT, 0, sizeof(OPT)); // 在头文件 #include<string.h> 中
18    // 计算二维数组，规模为 mn

20    for (int i=1;i<m+1;i++)
    {
22        for (int j=1;j<n+1;j++)
        {
24            if(str1[i-1]==str2[j-1])
            {
26                OPT[i][j] = OPT[i-1][j-1] + 1;
            }
28            if(str1[i-1]!=str2[j-1])
            {
30                OPT[i][j] = max(OPT[i-1][j],OPT[i][j-1]);
            }
32        }
    }
34    for (int i=0;i<m+1;i++)
    {
36        for (int j=0;j<n+1;j++)
        {
38            cout<<OPT[i][j]<<" ";
        }
40        cout<<endl;
    }
42    printf("%d",OPT[m][n]);

44    return 0;
}
```

```

1 #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         cout<<solution[i]<<endl;
60     delete []m;
61     delete []n;
62 }
```

```

62     delete [] solution;
63     return 0;
64 }
```

上面的代码实际上是在打印了这样一个二维矩阵：

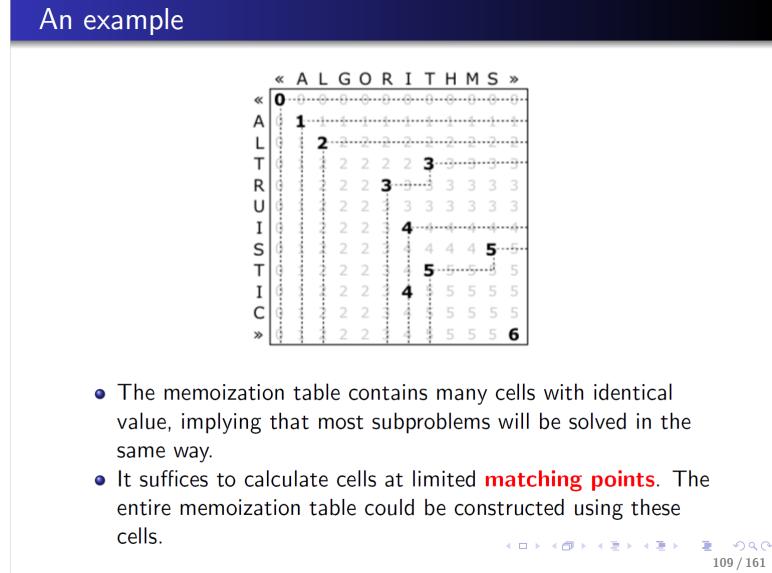


图 11：最长公共子序列的动态决策过程

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）：

```

class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int m = text1.length(); // 第一个字符串的长度
        int n = text2.length(); // 第二个字符串的长度
        int OPT[m+1][n+1]; // 0 矩阵
        memset(OPT, 0, sizeof(OPT)); // 在头文件 #include<string.h> 中
        // 计算二维数组，规模为 mn
        for (int i=1;i<m+1;i++)
        {
            for (int j=1;j<n+1;j++)
            {
                if (text1[i-1]==text2[j-1])
                {
                    OPT[i][j] = OPT[i-1][j-1] + 1;
                }
                if (text1[i-1]!=text2[j-1])
                {
                    OPT[i][j] = max(OPT[i-1][j], OPT[i][j-1]);
                }
            }
        }
    }
}
```

```

24     return OPT[m][n];
} ;
}

```

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

```

```

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 };

```

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：

```

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

```

```

17     int vv2 = robp(v2);
18     return max(vv1, vv2);
19 }
20 int robp(vector<int>& num) {
21     int k = num.size();
22     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 }
37 };

```

## 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：

```

1 #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 {

```

```

18     int ch;
19     cin >> ch;
20     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 }

35 //用一个数组分别记录偷根节点和不偷根节点时的最大值
36 class Solution {
37 public:
38     int rob(TreeNode* root) {
39         int * res = doRob(root);
40         return max(res[0],res[1]);
41     }
42 }

43 int * doRob(TreeNode * root)
44 {
45     int * res = new int [2];
46     res[0] = 0;
47     res[1] = 0;
48     if (root == NULL)
49     {
50         return res;
51     }
52     int* left = doRob(root->lchild);
53     int * right = doRob(root->rchild);
54     //不偷根节点， 最大值为两个子树的最大值之和
55     res[0] = max(left[0],left[1])+max(right[0],right[1]);
56     //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
57     res[1] = left[0] + right[0] + root->data;
58     return res;
59 }
60 };

61 int main()
62 {
63     cout << "输入第一个节点的值,-1表示没有儿子节点:" << endl;
64     TreeNode* T;
65     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
66     Solution s;
67     int opt = s.rob(T);
68     cout << opt << endl;
69     system("pause");
70     return 0;
71 }
72 }
```

伪代码:

```

1 rob(TreeNode *p)
2     if p==NULL
3         return 0;
```

```

4     else
5         d(p) = max(d(l)+d(r),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，考虑无负圈和无负边两种情况。

1. 若有向无环图中没有负圈时，分为无圈和有圈（无负圈），具体看下面两种情况。对于有向无环图来说，

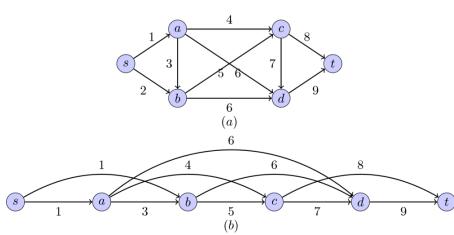
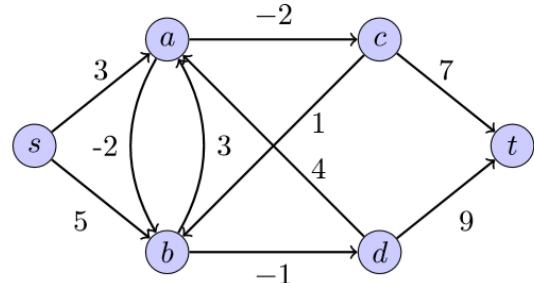


图 3.23: 有向无环图及结点的拓扑排序示例

(a) 无圈



(b) 有圈 (无负圈)

图 12: 有向无环图

可以将所有结点排成一个结点序列，使得每个结点都在其前驱结点之后，即边的方向都是从左向右，这个操作称为线性化 (Linearization)，也称为拓扑排序 (Topological sorting)。这样做的好处是，DAG 相当于是一个数组， $U$  在图中可以到达  $V$ ，则在数组中节点  $U$  在节点  $V$  前面。计算某个节点  $V$  时， $V$  的所有前驱节点都已经计算完毕。

(1). 上面无圈的图中计算起始节点  $s$  到某个节点  $v$  的最短路径，设为  $d(v)$ 。假设求解节点  $s$  到节点  $c$  的最短距离，则由动态规划， $d(c) = \min\{d(a) + r_{ac}, d(b) + r_{bc}\}$ ，依次递归求解最短距离。

(2). 对于有圈的情况，会变得稍微复杂点。比如依次求解  $d(a), d(b), d(a) = \min\{3, 3+d(b), 4+d(d)\}$ ， $d(b) = \min\{5, -2+d(a), 1+d(c)\}$ ，这样递归求解的时候  $d(a)$  和  $d(b)$  会产生依赖关系。原因是子问题定义的太粗了，需要添加限制条件来细化子问题，使得解没有循环依赖性，加个解的属性做限制。

现在限制路径上的节点不能超过  $k$  个（共有  $k$  个节点），则路径中一定没有圈。设  $OPT(v, k)$  表示从初始节点  $s$  至多经过  $k$  步到达节点  $v$  的最短路径，则有动态规划递推式： $OPT(v, k) = \min\{OPT(v, k-1), \min_{(v,w) \in E} \{OPT(w, k-1) + d(v, w)\}\}$

对递推式的理解：实际上至多  $k$  步应该包含至多  $k$  步，至多  $k-1$  步，至多  $k-2$  步等等，但是由于在求解至多  $k-1$  步时，即  $OPT(v, k-1)$  时，按照给定的递推式  $OPT(v, k-1) = \min\{OPT(v, k-2), \min_{(v,w) \in E} \{OPT(w, k-2) + d(v, w)\}\}$ ，已经包含了  $OPT(v, k-2)$ ，所以只需分为递推式的两种情况即可。

2. 若有向无环图中不仅没有负圈，而且没有负边的情况请参考贪心算法部分。

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

<p>Viterbi's decoding algorithm: recursion</p> <ul style="list-style-type: none"> <li>• First we rewrite <math>\max_X P(X, Y)</math> as:</li> <math display="block">\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}</math> <li>• Let denote <math>v_i(k)</math> as</li> <math display="block">\max_{x_{i-1}} \dots \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}</math> <p>Then we have:</p> <math display="block">\max_X P(X, Y) = \max_k v_n(k)</math> <li>• We can also observe the following recursion:</li> <math display="block">v_i(k) = e_k(y_i) \max_l (a_{lk} v_{i-1}(l))</math> </ul>	<p>Viterbi's decoding algorithm</p> <pre> VITERBIDECODING( Y, a, e ) 1: Initialize <math>v_1(k) = a_{0k} e_k(y_1)</math> for all state <math>k</math>; 2: for <math>i = 2</math> to <math>n</math> do 3:   for each state <math>k</math> do 4:     <math>v_i(k) = e_k(y_i) \max_l (a_{lk} v_{i-1}(l))</math>; 5:     <math>ptr_i(k) = argmax_l (a_{lk} v_{i-1}(l))</math>; 6:   end for 7: end for 8: <math>P(X^*, Y) = \max_k (v_n(k))</math>; 9: <math>x_n^* = argmax_k (v_n(k))</math>; 10: for <math>i = n - 1</math> to 1 do 11:   <math>x_i^* = ptr_{i-1}(x_{i+1}^*)</math>; 12: end for 13: return <math>X</math>; </pre>
Dongbo Bu C57110082 Algorithm Design and Analysis	Dongbo Bu C57110082 Algorithm Design and Analysis
(a) 推导过程	(b) Viterbi 算法

图 13: Viterbi 算法及其推导

介绍完动态规划算法，下面的题将面向贪心算法

## Question 23

区间调度问题，输入: $n$  个活动  $A = \{A_1, A_2, \dots, A_n\}$ ，每个活动都要占用资源。活动  $A_i$  使用资源区间为  $[S_i, F_i]$ 。活动  $A_i$  产生收益  $W_i$ 。求最优的区间调度，使得收益最大。

解：这道题经常出现在 google 的面试题当中，分两种情况讨论。当收益  $W_i$  不尽相同时采用动态规划，都相同时采用贪心算法。为了简单起见，所有的活动  $A_1, A_2, \dots, A_n$  都已经按照结束时间顺序排列。

(1) 当收益  $W_i$  不尽相同时，设  $OPT(i)$  表示任务  $A_1, A_2, \dots, A_i$  的最优调度， $pre(i)$  表示任务  $A_i$  开始之前就结束的所有任务，则有动态规划递推式： $OPT(i) = \max\{OPT(i-1), OPT(pre(i)) + W_i\}$

(2) 当收益  $W_i$  都相同时，每次选择的规则是依次选结束最早的活动 (或者最晚开始的活动，原理一样)，就可以在给定的资源线上尽可能开展最多的活动，即最高的收益。贪心算法每次的选择都是最优的结果。

总结：贪心算法与动态规划算法有不同的地方。(1) 动态规划算法需要回溯，贪心算法不需要;(2) 动态规划算法需要递归地算全局最优，贪心算法是每一步取局部最优，不需要递归，最终结果还是全局最优;(3) 每一个贪心算法背后，都有一个较笨拙的动态规划算法。

## Question 24

最短路径问题。对于有向无环图，当没有负圈时可以采用动态规划求解最短路径；当没有负边时，可以采用动态规划或者贪心算法。

无负圈情况在动态规划部分已经总结，这里介绍无负边的求解方法。无负边首先也可以使用动态规划求解，递推公式与无负圈相同， $OPT(v, k) = \min\{OPT(v, k-1), \min_{(u,v) \in E} OPT(u, k-1) + d(u, v)\}$ 。从伪代码中可以看出，初始状态要将第一列赋值无穷大，第一行赋值 0，然后根据递推公式一列一列

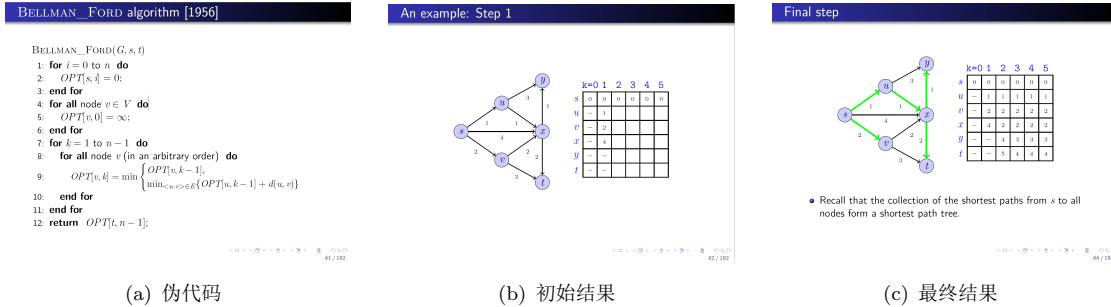


图 14: 伪代码及实例运算过程

地计算。

现在使用贪心算法（戴斯彻算法，Dijkstra 算法）求解无负边的情况，引入集合  $S$ 。每次计算时，在集合  $s$  中的节点不用计算，不在  $s$  中的只计算  $s$  往外一步的最短节点，将其加入  $s$  中。直接给出伪代码。

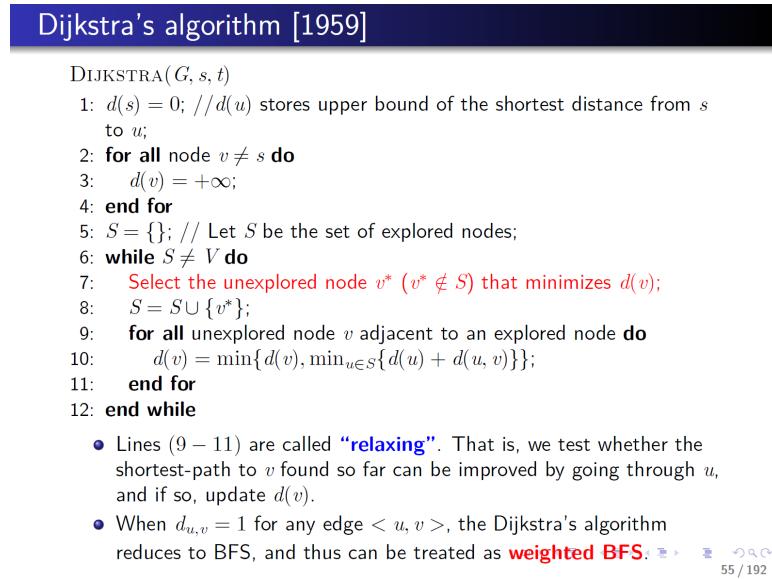


图 15: Dijkstra 算法

什么时候使用贪心算法求解有最优解？（目标函数是线性的，并且限制条件是 *matroid* 的，翻译为拟阵。）

## Question 25

例题：给定一组向量  $\{v_1, v_2, \dots, v_n\}$ ，每个向量都有相应的权重  $\{w_1, w_2, \dots, w_n\}$ ，求这组向量中的极大无关组使得权重最大。首先考虑动态规划算法： $OPT(1, 2, \dots, i) = \max\{OPT(1, 2, \dots, i-1), OPT(i^C) + w_i\}$ ， $OPT(i^C)$  表示与  $i$  线性无关的向量集合。

如果  $v$  不是零向量，且权重最大，则最优解中一定含有  $v$ 。使用贪心算法求解：将向量按照权重下降排列，依次选择。如果某个向量与之前的向量线性相关，则跳过，即可获得最优解。

(上述这个问题还有个近似的形式：考虑一个图，每条边都有权重，选择一些边使得权重最大且图中没有圈。)

## Question X

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

1. 二维数组查找，在一个二维数组中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。

```
1 //二维数组查找
2 //在一个二维数组中（每个一维数组的长度相同），每一行都按照从左到右递增的顺序排序，每一列都按照从上到下
3 //递增的顺序排序。
4 //请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数。
5 #include <iostream>
6 #include <vector>
7
8 using namespace std;
9
10 class Solution {
11 public:
12     bool Find(int target, vector<vector<int>> array) {
13         int rowCount = array.size();
14         int colCount = array[0].size();
15         int i, j;
16         for (i = rowCount - 1, j = 0; i >= 0 && j < colCount;)
17         {
18             if (target == array[i][j])
19                 return true;
20             if (target < array[i][j])
21             {
22                 i--;
23                 continue;
24             }
25             if (target > array[i][j])
26             {
27                 j++;
28                 continue;
29             }
30         }
31         return false;
32     }
33 }
34
35 int main()
36 {
37     vector<int> a;
38     vector<int> b;
39     vector<vector<int>>c;
40     a.push_back(2);
```

```

41 a.push_back(3);
42 a.push_back(4);
43 a.push_back(5);
44 b.push_back(4);
45 b.push_back(5);
46 b.push_back(6);
47 b.push_back(7);
48 c.push_back(a);
49 c.push_back(b);
50 Solution s;
51 cout<<s.Find(9,c);
52 system("pause");
53 return 0;
}

```

2. 替换空格, 请实现一个函数, 将一个字符串中的每个空格替换成 “%20”。例如, 当字符串为 We Are Happy. 则经过替换之后的字符串为 We%20Are%20Happy。

```

class Solution {
public:
    void replaceSpace(char *str, int length) {
        int num = 0; // 空格总数
        int nlength = length;
        int nindex = 0;
        for (int i = 0; i < length; i++) {
            if (str[i] == ' ')
                num++;
        }
        nlength += 2 * num; // 替换之后的字符总长度
        nindex = nlength;
        for (int i = length; i >= 0; i--) {
            if (i == nindex)
                break;
            if (str[i] != ' ')
            {
                str[nindex] = str[i];
                nindex--;
            }
            else
            {
                str[nindex] = '0';
                nindex--;
                str[nindex] = '2';
                nindex--;
                str[nindex] = '%';
                nindex--;
            }
        }
    }
};

```

```

1 //字符串替换
2 //请实现一个函数, 将一个字符串中的每个空格替换成“%20”。
3 //例如, 当字符串为We Are Happy. 则经过替换之后的字符串为We%20Are%20Happy。
4
5 #include <iostream>
6 #include <string>
7

```

```

1 using namespace std;
2
3 class Solution
4 {
5 public:
6     char * replaceSpace(string str)
7     {
8         int len = str.length();
9         //先计算有多少个空格
10        int count = 0;
11        for(int i=0;i<len;i++)
12        {
13            if(str[i]==' ')
14            {
15                count++;
16            }
17        }
18        char *new_str = new char[len+2*count];//新的字符串
19        int N = 0;//记录当前有几个空格
20        for(int i=0;i<len;i++)
21        {
22            if(str[i]!=' ')
23            {
24                new_str[i+2*N] = str[i];
25            }
26            else
27            {
28                new_str[i+2*N] = '%';
29                new_str[i+2*N+1] = '2';
30                new_str[i+2*N+2] = '0';
31                N++;
32            }
33        }
34        return new_str;
35    }
36 }
37
38 int main()
39 {
40     Solution s;
41     string str= "ab cd ds";
42     char* new_str = s.replaceSpace(str);
43     string t(new_str);
44     cout<<t;
45     system("pause");
46 }

```

此代码在网站中无法通过，但仍然是正确的，因为更改了类中的一些参数形式。

3. 从尾到头打印链表。输入一个链表，按链表从尾到头的顺序返回一个 ArrayList。

```

1 //从尾到头打印链表
2 //输入一个链表，按链表从尾到头的顺序返回一个ArrayList。
3 #include <iostream>
4 #include <vector>
5
6 using namespace std;
7
8 //链表结构体
9 struct ListNode{
10     int val;
11     ListNode* next;
12 };

```

```

11     ListNode*next;
12 };
13
14 class Solution {
15 public:
16     vector<int> printListFromTailToHead(ListNode* head) {
17         vector<int> A;
18         while(head!=NULL)
19         {
20             A.push_back(head->val);
21             head = head->next;
22         }
23         int len = A.size();
24         vector<int>B(len);
25         for(int i=0;i<len;i++)
26         {
27             B[i] = A[len-i-1];
28         }
29         return B;
30     }
31 };
32
33 ListNode *Create()
34 {
35     ListNode *head = new ListNode;//构造头结点
36     head->val = -1;
37     head->next = NULL;//指向NULL
38
39     ListNode *Cur = head; //构造当前结点，用于记录当前链表构造的位置，初始位置为head
40
41     int data; //插入链表的数据
42     while(1)
43     {
44         cout << "请输入当前节点的数值: " << endl;
45         cin >> data;
46         if(data == -1) //插入-1时结束链表构造
47         {
48             break;
49         }
50
51         ListNode *New = new ListNode; //构造新结点，用于循环插入链表
52         New->val = data; //新结点数据
53         New->next = NULL; //新节点指向NULL
54         Cur->next = New; //当前结点指向新构造的结点
55         Cur = New; //当前结点顺移至新结点处，记录链表插入位置
56     }
57     return head; //返回头结点
58 }
59
60
61
62 int main()
63 {
64     ListNode*list = Create();
65     while(list!=NULL)
66     {
67         cout<<list->val<<" ";
68         list = list->next;
69     }
70     system("pause");
71     return 0;
72 }
73 }
```

4. 重建二叉树。输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入先序遍历序列 1,2,4,7,3,5,6,8 和中序遍历序列 4,7,2,1,5,3,8,6，则重建二叉树并返回。

```
1 //重建二叉树
2 //输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含
3 //重复的数字。
4 //例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回。
5 #include <iostream>
6 #include <vector>
7
8 using namespace std;
9
10 struct TreeNode
11 {
12     int val;
13     TreeNode* left;
14     TreeNode* right;
15 };
16
17 //后序输出二叉树
18 void PostOrderBiTree(TreeNode*T)
19 {
20     if (T==NULL)
21     {
22         return;
23     }
24     else
25     {
26         PostOrderBiTree(T->left);
27         PostOrderBiTree(T->right);
28         cout<<T->val<<" ";
29     }
30 }
31
32 class Solution
33 {
34 public:
35     TreeNode* reConstructBinaryTree(vector<int> pre, vector<int> vin) {
36         int len = vin.size();
37         if (len==0)
38         {
39             return NULL;
40         }
41         TreeNode *head = new TreeNode;
42         head->val = pre[0];
43         vector<int> left_pre, left_vin, right_pre, right_vin;//分别存储左右子树的先序和中序
44
45         //在中序里面寻找根节点的位置
46         int gen = 0;
47         for(int i=0;i<len;i++)
48         {
49             if (vin[i]==pre[0])
50             {
51                 gen = i;
52                 break;
53             }
54         }
55
56         //左子树的先序和中序
57         for(int i=0;i<gen;i++)
58         {
```

```

59     left_vin.push_back(vin[i]);
60     left_pre.push_back(pre[i+1]);
61 }
62
63 //右子树的先序和中序
64 for(int i=gen+1;i<len;i++)
65 {
66     right_vin.push_back(vin[i]);
67     right_pre.push_back(pre[i]);
68 }
69
70 head->left = reConstructBinaryTree(left_pre, left_vin);
71 head->right = reConstructBinaryTree(right_pre, right_vin);
72 return head;
73 }
74 };
75
76 int main()
77 {
78     Solution s;
79     vector<int> pre, vin;
80     pre.push_back(1);
81     pre.push_back(2);
82     pre.push_back(3);
83     vin.push_back(2);
84     vin.push_back(1);
85     vin.push_back(3);
86     TreeNode*T = s.reConstructBinaryTree(pre, vin);
87     PostOrderBiTree(T); //后序输出，验证构建的正确性
88
89     system("pause");
90     return 0;
91 }

```

递归操作。首先在先序中找出根节点，然后在中序中找出根节点的位置序号，该位置序号前面的是该根节点的左子树，后面的是根节点的左子树；然后将左右子树的先序中序遍历结果存起来，依次进行左右子树的递归。

5. 用两个栈实现队列。用两个栈来实现一个队列，完成队列的 Push 和 Pop 操作。队列中的元素为 int 类型。

```

1 //用两个栈实现队列
2 //用两个栈来实现一个队列，完成队列的Push和Pop操作。队列中的元素为int类型。
3 #include <iostream>
4 #include <stack>
5
6 using namespace std;
7
8 class Solution
9 {
10 public:
11     void push(int node) {
12         stack1.push(node);
13     }
14
15     int pop() {
16         if(stack2.empty())
17         {
18             while(!stack1.empty())
19             {
20                 stack2.push(stack1.top());
21                 stack1.pop();
22             }
23             return stack2.top();
24         }
25         else
26         {
27             stack2.pop();
28             return stack2.top();
29         }
30     }
31
32     bool empty() {
33         return stack1.empty() && stack2.empty();
34     }
35
36     int top() {
37         if(stack2.empty())
38         {
39             while(!stack1.empty())
40             {
41                 stack2.push(stack1.top());
42                 stack1.pop();
43             }
44             return stack2.top();
45         }
46         else
47         {
48             return stack2.top();
49         }
50     }
51
52     int size() {
53         return stack1.size() + stack2.size();
54     }
55
56     void print() {
57         cout << "Stack1: ";
58         while(!stack1.empty())
59         {
60             cout << stack1.top() << " ";
61             stack1.pop();
62         }
63         cout << endl;
64         cout << "Stack2: ";
65         while(!stack2.empty())
66         {
67             cout << stack2.top() << " ";
68             stack2.pop();
69         }
70         cout << endl;
71     }
72
73     void clear() {
74         stack1.clear();
75         stack2.clear();
76     }
77 }

```

```

20     {
21         int p1 = stack1.top();
22         stack1.pop();
23         stack2.push(p1);
24     }
25     int p2 = stack2.top();
26     stack2.pop();
27     return p2;
28 }

30 private:
31     stack<int> stack1;
32     stack<int> stack2;
33 };
34

36 int main()
37 {
38     Solution S;
39     S.push(3);
40     S.push(4);
41     cout<<S.pop()<<" "<<S.pop();
42     system("pause");
43     return 0;
44 }

```

这里用  $C++$  来写，牛客网上都是用  $java$  写的。思路是相同的，进队列时直接进  $stack1$ ，出队列时需要把  $stack1$  中的所有元素压到  $stack2$  中，然后直接  $pop$   $stack2$  中的元素，当  $stack2$  中的元素为空时，才能接着将  $stack1$  中的元素压到  $stack2$  中，否则要一直把  $stack2$   $pop$  空为止。不同的地方在于  $C++$  中栈的  $pop$  没有返回值，只是把第一个元素删除，所以要想达到取出栈顶的元素的效果，应该先用  $top()$  取出，再用  $pop$  删除栈顶元素。

6. 把一个数组最开始的若干个元素搬到数组的末尾，我们称之为数组的旋转。输入一个非递减排序的数组的一个旋转，输出旋转数组的最小元素。例如数组 [3,4,5,1,2] 为 [1,2,3,4,5] 的一个旋转，该数组的最小值为 1。NOTE: 给出的所有元素都大于 0，若数组大小为 0，请返回 0。

采用二分法解答这个问题， $mid = low + (high - low)/2$ ，需要考虑三种情况：(1) $array[mid] > array[high]$ : 出现这种情况的 array 类似 [3,4,5,6,0,1,2]，此时最小数字一定在  $mid$  的右边， $low = mid + 1$ ；

(2) $array[mid] < array[high]$ : 出现这种情况的 array 类似 [2,2,3,4,5,6,6]，此时最小数字一定就是  $array[mid]$  或者在  $mid$  的左边，因为右边必然都是递增的， $high = mid$ ；

(3) $array[mid] == array[high]$ :

出现这种情况的 array 类似 [1,0,1,1,1] 或者 [1,1,1,0,1]，此时最小数字不好判断在  $mid$  左边，还是右边，这时只好一个一个试， $high = high - 1$ 。

```

class Solution {
public:
    int minNumberInRotateArray(vector<int> rotateArray) {
        int length = rotateArray.size();
        if (length==0)
            return 0;
        int left = 0;
        int right = length - 1;

        while(left<right)
        {
            int mid = (left+right)/2;

```

```

14         if (rotateArray[mid] < rotateArray[right])
15             right = mid; // 不能是 mid-1
16         else if (rotateArray[mid] > rotateArray[right])
17             left = mid + 1;
18         else
19             right--; // 此时 rotateArray[mid]==rotateArray[right], 无法判断最小值在哪边, 只能缩小范围
20     }
21     return rotateArray[left];
22 }

```

7. 斐波那契数列。要求输入一个整数  $n$ , 请你输出斐波那契数列的第  $n$  项 (从 0 开始, 第 0 项为 0)。 $n \leq 39$ 。

```

// 递归
2 class Solution1{
public:
4     int Fibonacci(int n)
{
6         if(n<2)
7         {
8             return n;
9         }
10        else
11        {
12            return Fibonacci(n-1)+Fibonacci(n-2);
13        }
14    };
16
// 数组
18 class Solution2{
public:
20     int Fibonacci(int n)
{
22         int *F = new int[40];
F[0] = 0;
F[1] = 1;
24         for(int i = 2;i<=n;i++)
{
26             F[i] = F[i-1]+F[i-2];
28         }
30         return F[n];
31     };
32
// 三个变量代替数组
34 class Solution3{
public:
36     int Fibonacci(int n)
{
38         int pre1 = 0;
39         int pre2 = 1;
40         int cur = 0; // 存储当前计算结果
41         for(int i=2;i<=n;i++)
{
42             cur = pre1+pre2;
43             pre1 = pre2;
44             pre2 = cur;
45         }
46         if(n<2)
47         {
48

```

```

50         return n;
51     }
52     return cur;
53 };

```

8. 跳台阶。一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法（先后次序不同算不同的结果）。

跟上面一题思路一样：

```

1 class Solution{
2 public:
3     int jumpFloor(int n)
4     {
5         int pre1 = 1;
6         int pre2 = 2;
7         int cur = 0;//当前计算结果
8         for(int i=3;i<=n;i++)
9         {
10             cur = pre1+pre2;
11             pre1 = pre2;
12             pre2 = cur;
13         }
14         if(n<3)
15         {
16             return n;
17         }
18         return cur;
19     }
20 };

```

9. 变态跳台阶。一只青蛙一次可以跳上 1 级台阶，也可以跳上 2 级……它也可以跳上 n 级。求该青蛙跳上一个 n 级的台阶总共有多少种跳法。

```

1 class Solution{
2 public:
3     int jumpFloorII(int number)
4     {
5         if(number<2)
6         {
7             return number;
8         }
9         else
10        {
11             int pre = 0;
12             int cur = 1;
13             for(int i=2;i<=number;i++)
14             {
15                 pre = cur;
16                 cur = 2*pre;
17             }
18             return cur;
19         }
20     }
21 };

```

10. 矩形覆盖。我们可以用  $2 \times 1$  的小矩形横着或者竖着去覆盖更大的矩形。请问用  $n$  个  $2 \times 1$  的小矩形无重叠地覆盖一个  $2 \times n$  的大矩形，总共有多少种方法？  
(本质还是斐波那契数列和第八题一样的代码)

11. 二进制中的 1 的个数。输入一个整数，输出该数二进制表示中 1 的个数。其中负数用补码表示。

```
1 class Solution {
2     public:
3         int NumberOf1(int n) {
4             int count = 0;
5             while(n!=0)
6             {
7                 n = n&(n-1);
8                 count++;
9             }
10            return count;
11        }
12    };
13};
```

这道题要明白一个结论：一个二进制数  $n$  减 1 后与原二进制数进行  $\&$  运算（即  $n \& (n - 1)$ ）会消去最右边的 1，实际上挺好想的，能做多少次与运算。就有多少个 1。 $C++$  中负数默认转的就是补码形式，所以不需要 *if* 判断。

12. 数值的整数次方。给定一个 *double* 类型的浮点数 *base* 和 *int* 类型的整数 *exponent*。求 *base* 的 *exponent* 次方。保证 *base* 和 *exponent* 不同时为 0。

```
1 class Solution{
2     public:
3         double Power(double base,int exponent)
4         {
5             if(base==0.0)
6             {
7                 return 0.0;
8             }
9             else if(exponent==0)
10            {
11                return 1.0;
12            }
13            else
14            {
15                double result;
16                int e = exponent>0?exponent:-exponent;
17                for(int i=1;i<=e;i++)
18                {
19                    result = result*base;
20                }
21                return exponent>0?result:1/result;
22            }
23        }
24    };
25};
```

13. 调整数组顺序使奇数位于偶数前面。输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有的奇数位于数组的前半部分，所有的偶数位于数组的后半部分，并保证奇数和奇数，偶

数和偶数之间的相对位置不变。

```
1 #include <iostream>
2 #include <vector>
3
4 using namespace std;
5
6 class Solution {
7 public:
8     void reOrderArray(vector<int> &array) {
9         int len = array.size();
10        for (int i=0;i<len;i++)
11        {
12            for (int j=0;j<len-1;j++)
13            {
14                if (array[j]&2==0&&array[j+1]&2==1)
15                {
16                    int temp = array[j];
17                    array[j] = array[j+1];
18                    array[j+1] = temp;
19                }
20            }
21        }
22    };
23
24    int main()
25    {
26        Solution s;
27        vector<int> array;
28        array.push_back(1);
29        array.push_back(2);
30        array.push_back(3);
31        array.push_back(4);
32        array.push_back(5);
33        array.push_back(6);
34        s.reOrderArray(array);
35        for (int i=0;i<array.size();i++)
36        {
37            cout<<array[i]<<" ";
38        }
39        cout<<endl;
40        system("pause");
41        return 0;
42    }
}
```

使用双循环冒泡方法，遇到偶数与奇数相邻的情况就换两个数的位置。

14. 链表中倒数第 k 个结点。输入一个链表，输出该链表中倒数第 k 个结点。

```
1 #include <iostream>
2 using namespace std;
3
4 struct ListNode{
5     int val;
6     ListNode*next;
7 };
8
9 //先将链表反转
10 ListNode* reverse(ListNode* head)
```

```

11 {
12     ListNode*pre = NULL;
13     ListNode*cur = NULL;
14     while(head!=NULL)
15     {
16         cur = head->next;
17         head->next = pre;
18         pre = head;
19         head = cur;
20     }
21     return pre;
22 }
23
24 class Solution {
25 public:
26     int FindKthToTail(ListNode* pListHead, unsigned int k) {
27         pListHead = reverse(pListHead); //先将链表反转
28         int i = 1;
29         while(i<k)
30         {
31             i++;
32             pListHead = pListHead->next;
33         }
34         return pListHead->val;
35     }
36 };
37
38 ListNode *Create()
39 {
40     ListNode *head = new ListNode;//构造头结点
41     head->val = -1;
42     head->next = NULL;//指向NULL
43
44     ListNode *Cur = head; //构造当前结点，用于记录当前链表构造的位置，初始位置为head
45
46     int data; //插入链表的数据
47     while(1)
48     {
49         cout << "请输入当前节点的数值：" << endl;
50         cin >> data;
51         if(data == -1) //插入-1时结束链表构造
52         {
53             break;
54         }
55
56         ListNode *New = new ListNode; //构造新结点，用于循环插入链表
57         New->val = data; //新结点数据
58         New->next = NULL; //新结点指向NULL
59         Cur->next = New; //当前结点指向新构造的结点
60         Cur = New; //当前结点顺移至新结点处，记录链表插入位置
61     }
62     return head; //返回头结点
63 }
64
65 int main()
66 {
67     Solution s;
68     ListNode*head = Create();
69     int data = s.FindKthToTail(head,4);
70     cout<<data;
71     system("pause");
72     return 0;
73 }

```

上述方法先将列表反转，输出第  $k$  个节点的值。也可以设置两个快慢指针，起始位置相差  $k$  个节点，然后快慢一起走，当快指针到达链表末尾时，慢指针到达倒数第  $k$  个节点。

或者采用双指针，让第一个指针先走  $k$  步，然后第二个指针再走，当第一个指针走到末尾的时候，第二个指针就在倒数第  $k$  个节点处，此外要注意  $k$  和链表长度的比较，代码如下：

```
1 struct ListNode {
2     int val;
3     struct ListNode *next;
4     ListNode(int x) :
5         val(x), next(NULL) {
6     }
7 };
8
9 class Solution {
10 public:
11     ListNode* FindKthToTail(ListNode* pListHead, int k) {
12         ListNode* pre = NULL;
13         ListNode* p = NULL;
14         pre = pListHead;
15         p = pListHead;
16         int length = k; // 记录k
17         int count = 0; // 记录节点个数，后续需要和k比较
18         while(p!=NULL)
19         {
20             p = p->next;
21             count++;
22             if(k<=0)
23                 pre = pre->next;
24             k--;
25         }
26         if(count<length)
27             return NULL;
28         return pre;
29     }
30 };
```

#### 4. 反转链表。输入一个链表，反转链表后，输出新链表的表头。

```
//反转链表
//输入一个链表，反转链表后，输出新链表的表头。
4 #include <iostream>
6
8 struct ListNode{
9     int val;
10    ListNode*next;
11 };
12
14 class Solution {
15 public:
16     ListNode* ReverseList(ListNode* pHead) {
17         ListNode *p = NULL;
18         ListNode *pre = NULL;
19
20         while(pHead!=NULL)
21         {
22             p = pHead->next;
```

```

24         pHead->next = pre;
25         pre = pHead;
26         pHead = p;
27     }
28 }
29
30 ListNode *Create()
31 {
32     ListNode *head = new ListNode;//构造头结点
33     head->val = -1;
34     head->next = NULL;//指向NULL
35
36     ListNode *Cur = head; //构造当前结点，用于记录当前链表构造的位置，初始位置为head
37
38     int data; //插入链表的数据
39     while(1)
40     {
41         cout << "请输入当前节点的数值：" << endl;
42         cin >> data;
43         if(data == -1) //插入-1时结束链表构造
44         {
45             break;
46         }
47
48         ListNode *New = new ListNode; //构造新结点，用于循环插入链表
49         New->val = data; //新结点数据
50         New->next = NULL; //新节点指向NULL
51         Cur->next = New; //当前结点指向新构造的结点
52         Cur = New; //当前结点顺移至新结点处，记录链表插入位置
53     }
54     return head; //返回头结点
55 }
56
57 int main()
58 {
59     Solution s;
60     ListNode*head = Create();
61     ListNode*phead = s.ReverseList(head);
62     while(phead!=NULL)
63     {
64         cout<<phead->val<<" ";
65         phead = phead->next;
66     }
67     system("pause");
68     return 0;
69 }
70 }
```

关键在于类中的链表操作，定义两个链表指针  $p, pre$ ,  $pre$  表示  $p$  的前一个节点，假设待反转的链表当前节点为  $head$ ，首先用  $p$  记录  $head$  的下一个节点，然后将  $head$  与  $p$  断开，令  $head$  指向  $pre$ ，再将  $head$  赋值给  $pre$ ，最后将  $p$  赋值给  $head$ ，进行下一次操作，当  $head$  到最后一个节点时，操作之后  $pre$  就是最后一个节点，而且前面的所有节点都已经反转地指向后驱节点。

16. 合并两个排序的链表。输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

```

2 struct ListNode {
3     int val;
4     struct ListNode *next;
5 }
```

```

4     ListNode( int x ) : val(x) , next(NULL) {}
5 };
6 class Solution {
7 public:
8     ListNode* Merge(ListNode* pHead1, ListNode* pHead2)
9     {
10         ListNode* head = new ListNode(-1);
11         ListNode* cur = head; // 记录当前节点的位置
12
13         while(pHead1!=NULL&&pHead2!=NULL)
14         {
15             if(pHead1->val<=pHead2->val)
16             {
17                 ListNode* New = new ListNode(-1);
18                 New->val = pHead1->val;
19                 cur->next = New;
20                 cur = New;
21                 pHead1 = pHead1->next;
22             }
23             else
24             {
25                 ListNode* New = new ListNode(-1);
26                 New->val = pHead2->val;
27                 cur->next = New;
28                 cur = New;
29                 pHead2 = pHead2->next;
30             }
31         }
32         if(pHead1!=NULL)
33             cur->next = pHead1;
34         if(pHead2!=NULL)
35             cur->next = pHead2;
36         return head->next;
37     }
38 };

```

17. 树的子结构。输入两棵二叉树 A, B, 判断 B 是不是 A 的子结构。(ps: 我们约定空树不是任意一个树的子结构)

```

//输入两棵二叉树A, B, 判断B是不是A的子结构或者子树
1 //子树的意思是包含了一个结点, 就得包含这个结点下的所有节点, 一棵大小为n的二叉树有n个子树, 就是分别以每
2 //个结点为根的子树;
3 //子结构的意思是包含了一个结点, 可以只取左子树或者右子树, 或者都不取。
4 #include <iostream>

5 using namespace std;

7 struct TreeNode{
8     int val;
9     TreeNode* lchild;
10    TreeNode* rchild;
11};

13
14 bool doesTree1HaveTree2(TreeNode* node1,TreeNode* node2)
15 {
16     //如果Tree2已经遍历完了都能对应的上, 返回true
17     if(node2==NULL)
18     {
19         return true;
20     }
21     //如果Tree2还没有遍历完, Tree1却遍历完了。返回false
22 }

```

```

24     if (node1==NULL)
25     {
26         return false;
27     }
28 //如果其中有一个点没有对应上，返回false
29     if (node1->val!=node2->val)
30     {
31         return false;
32     }
33 //如果根节点对应的上，那么就分别去子节点里面匹配
34     return doesTree1HaveTree2(node1->lchild ,node2->lchild )&&doesTree1HaveTree2(node1->rchild ,node2->
35         rchild );
36 }
37
38 //判断是否为子结构
39 bool HasSubTree(TreeNode* root1 ,TreeNode* root2)
40 {
41     bool result = false;
42 //当Tree1和Tree2都不为零的时候，才进行比较。否则直接返回false
43     if (root2 != NULL && root1 != NULL)
44     {
45         //如果找到了对应Tree2的根节点的点
46         if(root1->val == root2->val)
47         {
48             //以这个根节点为为起点判断是否包含Tree2
49             result = doesTree1HaveTree2(root1 ,root2 );
50         }
51         //如果找不到，那么就再去root的左儿子当作起点，去判断时候包含Tree2
52         if (!result)
53         {
54             result = HasSubTree(root1->lchild ,root2 );
55         }
56         //如果还找不到，那么就再去root的右儿子当作起点，去判断时候包含Tree2
57         if (!result)
58         {
59             result = HasSubTree(root1->rchild ,root2 );
60         }
61     }
62     return result;
63 }
64 //判断是否为子树
65
66
67
68 //先序创建二叉树
69 void CreateBiTree(TreeNode **T)
70 {
71     int ch;
72     cin >> ch;
73     if (ch == -1)
74     {
75         *T = NULL;
76         return ;
77     }
78     else
79     {
80         *T = new TreeNode;
81         (*T)->val = ch;
82         cout << "input" << ch << "'s left son node:" ;
83         CreateBiTree(&((*T)->lchild));
84         cout << "input" << ch << "'s right son node:" ;
85         CreateBiTree(&((*T)->rchild));
86     }

```

```

87     }
88     return;
89 }
90
91 int main(){
92     TreeNode*A;
93     TreeNode*B;
94     CreateBiTree(&A);
95     CreateBiTree(&B);
96     cout<<HasSubTree(A,B)<<endl;;
97     system("pause");
98     return 0;
99 }
```

只写了如何判断子结构，还有判断子树待考究。

18. 二叉树的镜像。操作给定的二叉树，将其变换为源二叉树的镜像。

```

1 struct TreeNode{
2     int val;
3     TreeNode*left;
4     TreeNode*right;
5 };
6
7 class Solution {
8 public:
9     void Mirror(TreeNode *pRoot) {
10         if(pRoot==NULL)
11         {
12             return;
13         }
14         TreeNode*temp = pRoot->left;
15         pRoot->left = pRoot->right;
16         pRoot->right = temp;
17         Mirror(pRoot->left);
18         Mirror(pRoot->right);
19     }
20 };
```

19. 顺时针打印矩阵。输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字，例如，如果输入如下 4x4 矩阵：1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16，则依次打印出数字 1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10。

```

class Solution {
1 public:
2     vector<int> printMatrix(vector<vector<int> > matrix) {
3         vector<int> ssz;
4         if(matrix.size()==0||matrix[0].size()==0)
5         {
6             return ssz;
7         }
8         int up = 0;
9         int down = matrix.size()-1;
10        int left = 0;
11        int right = matrix[0].size()-1;
12
13        while(true)
14        {
15            for(int col=left;col<=right;col++)
16            {
```

```

18     ssz.push_back(matrix[up][col]);
19 }
20 up++;
21 if(up>down)
22 {
23     break;
24 }

25 for(int row=up;row<=down;row++)
26 {
27     ssz.push_back(matrix[row][right]);
28 }
29 right--;
30 if(left>right)
31 {
32     break;
33 }

34 for(int col=right;col>=left;col--)
35 {
36     ssz.push_back(matrix[down][col]);
37 }
38 down--;
39 if(up>down)
40 {
41     break;
42 }

43 for(int row=down;row>=up;row--)
44 {
45     ssz.push_back(matrix[row][left]);
46 }
47 left++;
48 if(left>right)
49 {
50     break;
51 }
52 }
53 return ssz;
54 }
55 };
56 };
57 };

```

20. 定义栈的数据结构，请在该类型中实现一个能够得到栈中所含最小元素的 min 函数（时间复杂度应为  $O(1)$ ）。思路：用一个栈 s 保存数据，用另外一个栈 smin 保存依次入栈最小的数，即 smin 的栈顶元素是当前栈的最小元素。

比如 s 中依次入栈 5, 4, 3, 8, 10, 11, 12, 1

则 smin 依次入栈 5, 4, 3, no,no, no, no, 1

其中 no 代表此次不入栈，每次入栈的时候，如果入栈的元素比 smin 中的栈顶元素小或等于则入栈，否则不入栈，每次出栈的时候，判断 s 中出栈元素和 smin 栈顶元素是否相同，相同则 smin 出栈。

```

class Solution {
public:
    void push(int value) {
        s.push(value);
        if(s_min.empty())
            s_min.push(value);
        else if(value<=s_min.top())
            s_min.push(value);
    }
    void pop() {
        if(s.top()==s_min.top())

```

```

12     s_min.pop();
13     s.pop();
14 }
15 int top() {
16     return s.top();
17 }
18 int min() {
19     return s_min.top();
20 }

22 private:
23     stack<int> s;
24     stack<int> s_min;
25 };

```

21. 输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否可能为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列 1,2,3,4,5 是某栈的压入顺序，序列 4,5,3,2,1 是该压栈序列对应的一个弹出序列，但 4,3,5,1,2 就不可能是该压栈序列的弹出序列。(注意：这两个序列的长度是相等的)

```

1 class Solution {
2 public:
3     bool IsPopOrder(vector<int> pushV, vector<int> popV) {
4         int length = pushV.size();
5         stack<int> s;
6         int j=0;
7         for(int i=0;i<length;i++)
8         {
9             s.push(pushV[i]);
10            while((!s.empty())&&(s.top()==popV[j]))
11            /*
12             要是没有!s.empty() 或者两个条件顺序调换，则当popV[j]已经指到最后一个元素的时候，再进行j++,
13             popv[j]就没有指定的元素了，这样就会溢出，所以需要!s.empty()提前结束while，不让运行后面的
14             的popV[i]。
15             */
16            {
17                s.pop();
18                j++;
19            }
20        }
21        if(s.empty())
22            return true;
23        return false;
24    }
25 };

```

22. 从上往下打印二叉树。从上往下打印出二叉树的每个节点，同层节点从左至右打印。  
基本思路，采用 BFS 算法，将已经遍历过的节点加入到队列当中，最后输出队列即可。

```

1 struct TreeNode{
2     int val;
3     TreeNode* left;
4     TreeNode* right;
5 };
6
7 class Solution{
8 public:
9     vector<int> PrintFromTopToBottom(TreeNode* root)

```

```

11 {
12     vector<int> v; // 要返回的数组
13     queue<TreeNode*> q; // 队列
14     q.push(root);
15     while (!q.empty())
16     {
17         TreeNode* T = q.front(); // 队列中的树取出来
18         q.pop(); // 删除最前面的树
19         if (T==NULL)
20         {
21             continue;
22         }
23         v.push_back(T->val);
24         q.push(T->left);
25         q.push(T->right);
26     }
27     return v;
28 }

```

23. 二叉搜索树的后序遍历序列，输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历的结果。如果是则输出 Yes, 否则输出 No。假设输入的数组的任意两个数字都互不相同。

基本思路：后续遍历的最后的输出值一定是根节点，根据根节点来划分左右子树的后序遍历结果。第一个大于根节点的值作为划分的界，然后判断后面的数是不是都大于根节点。如果满足，则继续递归调用左右子树的后序遍历结果。

```

bool isBST(vector<int> sequence ,int start ,int end)
{
    if (start>=end)
    {
        return true;
    }
    int val = sequence[end]; // 根节点
    int split; // 划分的索引，第一个 for 循环为了找到划分的位置
    for (int i=start;i<end;i++)
    {
        if (sequence[i]>val)
        {
            split = i;
            break;
        }
    }
    // 判断后面是否还有小于根节点的数，如果有，返回 false
    for (int i=split;i<end;i++)
    {
        if (sequence[i]<val)
        {
            return false;
        }
    }
    return isBST(sequence,start,split-1)&&isBST(sequence,split,end-1);
}

class Solution{
public:
    bool VerifySequenceOfBST(vector<int> sequence)
    {
        int len = sequence.size();
        if (len==NULL)
        {

```

```

36         return false;
37     }
38 }
39 };

```

28. 数组中出现次数超过一半的数字。数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为 9 的数组 1,2,3,2,2,2,5,4,2。由于数字 2 在数组中出现了 5 次，超过数组长度的一半，因此输出 2，如果不存在则输出 0。

```

1 class Solution {
2 public:
3     int MoreThanHalfNum_Solution(vector<int> numbers) {
4         int len = numbers.size();
5         if (len==0)
6         {
7             return 0;
8         }
9         int prevalue = numbers[0];
10        int count = 1;
11        for (int i=1;i<len ;i++)
12        {
13            if (numbers[ i]==prevalue)
14            {
15                count++;
16            }
17            else
18            {
19                count--;
20                if (count==0)
21                {
22                    prevalue = numbers[ i];
23                    count = 1;
24                }
25            }
26        }
27        //判断prevalue出现的次数是否真的大于数组规模的一半
28        count = 0;
29        for (int i=0;i<len ;i++)
30        {
31            if (numbers[ i]==prevalue)
32            {
33                count++;
34            }
35        }
36        if (count>len /2)
37        {
38            return prevalue;
39        }
40        return 0;
41    }
42 };
43 };

```

28. 数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为 9 的数组 1,2,3,2,2,2,5,4,2。由于数字 2 在数组中出现了 5 次，超过数组长度的一半，因此输出 2。如果不存在则输出 0。

解法一：利用 vector 的 sort()

首先将 vector 中的数字排序，因为要求输出重复次数大于数组大小一般的数，排序结束后，相同的数字都在相邻位置，直接判断数组当前位置的数字与数组位置 + 数组一半长度位置的数字是否相等，相等则输出该数字，代码如下：

```
1 class Solution {
2     public:
3         int MoreThanHalfNum_Solution(vector<int> numbers) {
4             int middle = numbers.size() / 2;
5             sort(numbers.begin(), numbers.end());
6             for (int i=0; i+middle<numbers.size(); i++) {
7                 if (numbers[i]==numbers[i+middle])
8                     return numbers[i];
9             }
10            return 0;
11        }
12    };
13 }
```

## 解法二：空间换时间

使用 c++ 中的 hash 数据结构 map，保存 vector 中每个数字在数组中出现的次数，再找到次数超过一半的，代码如下（这里只是写了一个如何使用 map 的实例，将类中返回的  $number_{map}$  换成就是这题的另一种解法）：

```
1 #include<iostream>
2 #include<vector>
3 #include<map>/hash表的头文件
4 #include<iterator>/迭代器的头文件
5
6 using namespace std;
7
8 class Solution {
9     public:
10         map<int, int> MoreThanHalfNum_Solution(vector<int> numbers) {
11             map<int, int> number_map;
12             // 构造hash表
13             for (int i=0; i<numbers.size(); i++) {
14                 number_map[numbers[i]] += 1;
15             }
16             int middle_size = numbers.size() / 2;
17             int num = 0;
18             for (map<int, int>::iterator it=number_map.begin(); it!=number_map.end(); it++) {
19                 if (middle_size<(it->second))
20                     num = it->first;
21             }
22             return number_map;
23         }
24     };
25
26
27
28 int main()
29 {
30     int len = 10;
31     vector<int> v(len);
32     for (int i=0; i<len; i++) {
33         v[i] = i;
34     }
35     Solution s;
36     map<int, int> number_map;
37     number_map = s.MoreThanHalfNum_Solution(v);
```

```

41     for (map<int , int >::iterator it=number_map.begin() ; it!=number_map.end() ; it++)
42     {
43         cout<<it->first <<" : "<<it->second<<endl ;
44     }
45 }
```

28. 给定正整数  $n$ , 找到若干个完全平方数 (比如 1, 4, 9, 16, ...) 使得它们的和等于  $n$ 。你需要让组成和的完全平方数的个数最少。

解法一：动态规划

```

2 class Solution {
3 public:
4     int numSquares(int n)
5     {
6         vector<int> dp;
7         for (int i=0;i<=n; i++)
8         {
9             dp.pushback(10); // 先赋值较大的数
10        }
11        dp[0] = 0;
12        for (int i=1;i<=n; i++)
13        {
14            int j=1;
15            while(i-j*j>=0)
16            {
17                dp[i]=min(dp[i],dp[i-j*j]+1);
18                j++;
19            }
20        }
21    }
22 }
```

解法二：数学理论支持

(1)Lagrange 四平方定理：任何一个正整数都可以表示成不超过四个整数的平方之和。

(2) 满足四数平方和定理的数  $n$ (这里要满足由四个数构成，小于四个不行)，必定满足  $n = 4^a(8b + 7)n$

```

2 bool is_sqrt(long long n)
3 {
4     int m = sqrt(n);
5     if (m*m == n)
6         return true;
7     else
8         return false;
9 }
10 int solve(long long n)
11 {
12     if (is_sqrt(n))
13         return 1;
14     while (n % 4 == 0)
15         n /= 4;
16     if (n % 8 == 7)
17         return 4;
18     for (int i = 0; i*i < n; i++)
19     {
20         if (is_sqrt(n - i*i))
21             return 2;
```

```

22     }
23     return 3;
24 }
```

29. 最小的 K 个数，输入 n 个整数，找出其中最小的 K 个数。例如输入 4,5,1,6,2,7,3,8 这 8 个数字，则最小的 4 个数字是 1,2,3,4。

```

1 class Solution {
2 public:
3     vector<int> GetLeastNumbers_Solution(vector<int> input, int k) {
4         vector<int> v;
5         if(k>input.size())
6         {
7             return v;
8         }
9         sort(input.begin(),input.end());
10        for(int i=0;i<k; i++)
11        {
12            v.push_back(input[i]);
13        }
14        return v;
15    }
16};
```

具体的方放有两种：1. 先排好序再取前  $k$  个元素，快速排序或者归并排序需要  $O(n \log n)$  的复杂度，再乘以常数  $k$  还是  $O(n \log n)$  的复杂度；1. 循环  $k$  次，每次取第  $i, i = 1, \dots, k$  小的元素（具体方法见该部分 question6），该方法时间复杂度为  $O(n)$ 。这里采用了第一种方法。

30. 连续子数组的最大和，6,-3,-2,7,-15,1,2,2, 连续子向量的最大和为 8(从第 0 个开始, 到第 3 个为止)。给一个数组，返回它的最大连续子序列的和 (子向量的长度至少是 1)。

```

class Solution {
public:
    int FindGreatestSumOfSubArray(vector<int> array) {
        int len = array.size();
        vector<int> dp(len);
        int Max = array[0];
        dp[0] = array[0];
        for(int i=1; i<len; i++)
        {
            int max = dp[i-1]+array[i];
            if(max>array[i])
            {
                dp[i]=max;
            }
            else
            {
                dp[i]=array[i];
            }
            if(dp[i]>Max)
            {
                Max = dp[i];
            }
        }
        return Max;
    }
};
```

26 };

这里的  $dp[i]$  并不表示从 0 到  $i$  的最优连续子序列的和，而是  $dp[i] = \max\{dp[i-1] + array[i], array[i]\}$ ，表示以元素  $array[i]$  结尾的最大连续子数组和，最终整个数组里的最大连续子数组应该是  $dp$  中最大的那个数。

37. 数字在排序数组中出现的次数，统计一个数字在排序数组中出现的次数。

第一反应想到的是全部遍历一遍，但由于是排序数组，所以相邻两个不一样的时候就可以跳出循环，时间复杂度为  $O(1)$ .

```
1 class Solution {
2 public:
3     int GetNumberOfK(vector<int> data ,int k) {
4         int count=0;
5         int len = data.size();
6         for(int i=0;i<len ;i++)
7         {
8             if(data [ i]==k)
9             {
10                 count++;
11             }
12             if(data [ i]==k&&data [ i+1]!=k)
13             {
14                 break;
15             }
16         }
17         return count;
18     }
19 }
```

38. 二叉树的深度。输入一棵二叉树，求该树的深度。从根结点到叶结点依次经过的结点（含根、叶结点）形成树的一条路径，最长路径的长度为树的深度。

```
1 struct TreeNode{
2     int val;
3     TreeNode*left ;
4     TreeNode*right ;
5 };
6
7 class Solution{
8 public:
9     int TreeDepth(TreeNode*pRoot)
10    {
11        if(pRoot==NULL)
12        {
13            return 0;
14        }
15        int dleft = TreeDepth(pRoot->left );
16        int dright = TreeDepth(pRoot->right );
17        return max(dleft ,dright )+1;
18    }
19 }
```

39. 平衡二叉树。输入一棵二叉树，判断该二叉树是否是平衡二叉树。平衡二叉树的左右子树也是平衡二叉树，那么所谓平衡就是左右子树的高度差不超过 1。

```
1 struct TreeNode{
2     int val;
3     TreeNode* left;
4     TreeNode* right;
5 };
6
7 int TreeDepth(TreeNode*root)
8 {
9     if (root==NULL)
10    {
11        return 0;
12    }
13    int dleft = TreeDepth(root->left);
14    if (dleft== -1)
15    {
16        return -1;
17    }
18    int dright = TreeDepth(root->right);
19    if (dright== -1)
20    {
21        return -1;
22    }
23    return abs(dright-dleft)>1?-1:max(dright , dleft )+1;
24 }
25
26 class Solution {
27 public:
28     bool IsBalanced_Solution(TreeNode* pRoot) {
29         if (TreeDepth(pRoot)== -1)
30         {
31             return false;
32         }
33         return true;
34     }
35 }
```

40. 一个整型数组里除了两个数字之外，其他的数字都出现了两次。请写程序找出这两个只出现一次的数字。

```
1 class Solution {
2 public:
3     void FindNumsAppearOnce(vector<int> data ,int* num1,int *num2) {
4         map<int , int> vmap;
5         for(int i=0;i<data .size () ;i++)
6         {
7             vmap [data [i]]++;
8         }
9
10        int count = 0;
11        for(map<int , int >::iterator it=vmap .begin () ;it!=vmap .end () ;it++)
12        {
13            if (it->second == 1)
14            {
15                count++;
16                if (count==1)
17                    *num1 = it->first;//这里写num1 = &(it->first)通过不了，原因？？
18                if (count==2)
19
```

```

19             *num2 = it->first;
20         }
21     }
22 };

```

剑指 offer 上都是采用异或运算，这里给出一个使用 hash 数据结构来求解的方法，首先遍历数组，直接记录每个数字和  $q_i$  出现的次数；然后遍历 map，找出出现次数为 1 的数字。

42. 输入一个递增排序的数组和一个数字 S，在数组中查找两个数，使得他们的和正好是 S，如果有对数字的和等于 S，输出两个数的乘积最小的。对应每个测试案例，输出两个数，小的先输出。

由均值不等式可知：当两个正整数越接近时，对应的乘积越大，所以乘积最小的两个数应该是递增数组中最边缘的两个数，所以有以下两种思路：

解法一：双循环搜索，时间复杂度为  $O(n^2)$ ；

解法二：使用双指针，时间复杂度为  $O(n)$ ；

```

1 /*
2  class Solution {
3 public:
4     vector<int> FindNumbersWithSum(vector<int> array ,int sum) {
5         int length = array.size();
6         vector<int> v;
7         for(int i=0;i<length;i++)
8             for(int j=i+1;j<length;j++)
9                 if(array[i]+array[j]==sum)
10                {
11                    v.push_back(array[i]);
12                    v.push_back(array[j]);
13                    return v;
14                }
15        return v;
16    }
17 };
18 */
19
20 class Solution {
21 public:
22     vector<int> FindNumbersWithSum(vector<int> array ,int sum) {
23         int length = array.size();
24         vector<int> v;
25         int start = 0;
26         int end = length - 1;
27
28         while(start<end)
29         {
30             if(array[start]+array[end]==sum)
31             {
32                 v.push_back(array[start]);
33                 v.push_back(array[end]);
34                 return v;
35             }
36             else if(array[start]+array[end]<sum)
37                 start++;
38             else
39                 end--;
40         }
41         return v;
42     }
43 };

```

---

43. 汇编语言中有一种移位指令叫做循环左移（ROL），现在有个简单的任务，就是用字符串模拟这个指令的运算结果。对于一个给定的字符序列 S，请你把其循环左移 K 位后的序列输出。例如，字符序列 S=“ abcXYZdef ”，要求输出循环左移 3 位后的结果，即 “XYZdefabc”。

```
1 #include<iostream>
2 #include<string>
3
4 using namespace std;
5
6 class Solution {
7 public:
8     string LeftRotateString(string str, int n) {
9         int len = str.length();
10        char* str1 = new char[len];//string无法指定长度，先定义char*，再转为string
11        if(len==0)
12            return "";
13        else
14        {
15            for(int i=0;i<len;i++)
16            {
17                if(i<n)
18                    str1[len-(n-i)] = str[i];
19                else
20                {
21                    str1[i-n] = str[i];
22                }
23            }
24        }
25        string str_return = str1; //char*转string
26        return str_return;
27    }
28};
29
30
31 int main()
32 {
33     Solution S;
34     string s = "12345";
35     cout<<S.LeftRotateString(s, 2);
36     return 0;
37 }
```

44. 牛客最近来了一个新员工 Fish，每天早晨总是会拿着一本英文杂志，写些句子在本子上。同事 Cat 对 Fish 写的内容颇感兴趣，有一天他向 Fish 借来翻看，但却读不懂它的意思。例如，“student. a am I”。后来才意识到，这家伙原来把句子单词的顺序翻转了，正确的句子应该是 “I am a student.”。Cat 对一一的翻转这些单词顺序可不在行，你能帮助他么？

采用的思路是先遍历字符串，遇到空格记录单词的起始位置，依次将单词进栈；然后根据栈是否为空进行 while 循环，将每个单词依次放置到字符串中，代码如下：

```
1 #include<iostream>
2 #include<string>
3 #include<stack>
4
5 using namespace std;
```

```

6 class Solution {
8 public:
9     string ReverseSentence(string str) {
10         int len = str.length();
11         stack<string> stk;
12         int start = 0;//记录当前单词开始位置
13         int end = 0;//记录当前单词结束位置
14
15         for(int i=0;i<len;i++)
16         {
17             if(str[i]== ' ')
18             {
19                 end = i-1;
20                 string word = str.substr(start, end-start+1);
21                 stk.push(word);
22                 start = i+1;
23             }
24         }
25         stk.push(str.substr(start, len-start));//最后一个空格到结尾的单词入栈
26
27         //栈中单词放置到char*中
28         char* str_return = new char[len];
29         int position = 0;
30         while(!stk.empty())
31         {
32             string str_tmp = stk.top();
33             stk.pop();
34             int tmp_len = str_tmp.length();
35             //cout<<tmp_len<<endl;
36             for(int i=0;i<tmp_len;i++)
37             {
38                 str_return[i+position] = str_tmp[i];
39             }
40             if(tmp_len+position<len)
41                 str_return[tmp_len+position] = ' ';//防止最后一个单词的后面还有空格
42             position += (tmp_len+1);
43         }
44         string str_return1 = str_return;//char*转string
45         return str_return1;
46     }
47 };
48
49 int main()
50 {
51     Solution s;
52     string str = "student. a am I";
53     string str_return1 = s.ReverseSentence(str);
54     cout<<str_return1.size()<<endl;
55     cout<<str_return1<<endl;
56     return 0;
57 }
```

48. 写一个函数，求两个整数之和，要求在函数体内不得使用 +、-、\*、/四则运算符号。

```

1 /*
2 首先看十进制是如何做的： 5+7=12，三步走：
3 第一步：相加各位的值，不算进位，得到2；
4 第二步：计算进位值，得到10。如果这一步的进位值为0，那么第一步得到的值就是最终结果；
5 第三步：重复上述两步，只是相加的值变成上述两步的得到的结果2和10，得到12；

6 同样我们可以用三步走的方式计算二进制值相加： 5-101， 7-111
```

```

9 第一步：相加各位的值，不算进位，得到010，二进制每位相加就相当于各位做异或操作，101^111;
10 第二步：计算进位值，得到1010，相当于各位做与操作得到101，再向左移一位得到1010，(101&111)<<1;
11 第三步重复上述两步， 各位相加 010^1010=1000，进位值为100=(010&1010)<<1;
12 继续重复上述两步：1000^100 = 1100，进位值为0，跳出循环，1100为最终结果。
13 */

15
16 class Solution {
17 public:
18     int Add(int num1, int num2)
19     {
20         while(num2!=0) //当进位为0时，不带进位的结果就是相加的结果
21         {
22             int tmp = num1^num2; //计算不带进位的结果，相当于异或运算
23             num2 = (num1&num2)<<1; //计算进位结果，相当于与运算再左移一位
24             num1 = tmp;
25         }
26         return num1;//返回不带进位的结果
27     }
28 };

```

50. 在一个长度为 n 的数组里的所有数字都在 0 到 n-1 的范围内。数组中某些数字是重复的，但不知道有几个数字是重复的。也不知道每个数字重复几次。请找出数组中任意一个重复的数字。例如，如果输入长度为 7 的数组 2,3,1,0,2,5,3，那么对应的输出是第一个重复的数字 2。

```

class Solution {
1 public:
// Parameters:
2 //      numbers:      an array of integers
3 //      length:        the length of array numbers
4 //      duplication:  (Output) the duplicated number in the array number
5 // Return value:      true if the input is valid , and there are some duplications in the array number
6 //                      otherwise false
7     bool duplicate(int numbers[], int length, int* duplication) {
8         int* hash = new int[length];
9         for(int i=0;i<length;i++)
10            hash[i] = 0;
11         for(int i=0;i<length;i++)
12            hash[numbers[i]]++;
13         for(int i=0;i<length;i++)
14         {
15             if(hash[i]>1)
16             {
17                 duplication[0] = i;
18                 return true;
19             }
20         }
21     }
22     return false;
23 }
24 };

```

51. 给定一个数组 A[0,1,...,n-1]，请构建一个数组 B[0,1,...,n-1]，其中 B 中的元素  $B[i] = A[0]*A[1]*...*A[i-1]*A[i+1]*...*A[n-1]$ 。不能使用除法。（注意：规定  $B[0] = A[1] * A[2] * ... * A[n-1]$ ， $B[n-1] = A[0] * A[1] * ... * A[n-2]$ ；）对于 A 长度为 1 的情况，B 无意义，故而无法构建，因此该情况不会存在。

```

1 #include<iostream>
2 #include<vector>

```

```
3     using namespace std;
5
6 class Solution {
7 public:
8     vector<int> multiply(const vector<int>& A) {
9         int length = A.size();
10        vector<int> B;
11        for(int i=0;i<length;i++) {
12            B.push_back(1); //每一位的初始值都为1
13            for(int j=0;j<length;j++) {
14                if(j==i)
15                    continue;
16                else
17                {
18                    B[i] *= A[j];
19                }
20            }
21        }
22        return B;
23    }
24};
25
26
27 int main()
28 {
29     vector<int> A;
30     A.push_back(1);
31     A.push_back(2);
32     A.push_back(3);
33     A.push_back(4);
34     Solution s;
35     vector<int> B = s.multiply(A);
36     cout<<B.size()<<endl;
37     cout<<B[0]<<" "<<B[1]<<" "<<B[2]<<" "<<B[3]<<endl;
38     return 0;
39 }
```

## Question Y

leetcode 中的题

1. 给定一个整数数组 `nums` 和一个目标值 `target`, 请你在该数组中找出和为目标值的那两个整数, 并返回他们的数组下标。可以假设每种输入只会对应一个答案。但是, 数组中同一个元素不能使用两遍。

解法一：双循环暴力搜索

```
#include<iostream>
2 #include<vector>

4 using namespace std;
6
7 class Solution{
8 public:
9     vector<int> twoSum(vector<int>& nums, int target) {
10         vector<int> v;
11         int m = nums.size();
12         for(int i = 0; i < m; i++)
13         {
```

```

14     for( int j = i + 1; j<m; j++)
15     {
16         if( nums[ i]+nums[ j]==target )
17         {
18             v.push_back(i);
19             v.push_back(j);
20             return v;
21         }
22     }
23 }
24 return v;
25 }
26 };
27
28 int main()
29 {
30     Solution s;
31     vector<int> nums;
32     vector<int> v;
33     nums.push_back(2);
34     nums.push_back(7);
35     nums.push_back(11);
36     nums.push_back(15);
37     v = s.twoSum(nums, 9);
38     cout<<v[0]<< ' '<<v[1]<<endl;
39     return 0;
40 }
```

解法二：借助 c++ 中的 hash 数据结构 map，在遍历数组的过程中，判断目标值 target 和当前数组值的差值是否在 map 当中；如果在则直接返回两个索引序号，如果不在则记录数组值和对应的索引，分别存储为 key-value 到 map 当中，代码如下：

```

1 #include<iostream>
2 #include<vector>
3 #include<map>
4 #include<iterator>
5
6 using namespace std;
7
8 class Solution{
9 public:
10     vector<int> twoSum(vector<int>& nums, int target){
11         map<int , int> vmap;
12         vector<int> v;
13         int length = nums.size();
14         for(int i = 0; i<length; i++)
15         {
16             int diff = target - nums[i];
17             //if(vmap.find( diff)!=vmap.end())//判断map里面是否有某个key, find函数如果找到就返回对应的迭代器, 找不到就返回末尾的迭代器
18             if(vmap.count( diff)==1)//判断map里面是否有某个key, 由于map中的key不允许重复, 因此如果key存在就返回1, 不存在就返回0
19             {
20                 cout<<nums[ i]<<endl;
21                 v.push_back(vmap[ diff]);
22                 v.push_back(i);
23                 return v;
24             }
25             vmap[ nums[ i]] = i;
26         }
27     return v;
28 }
```

```

29     }
30 };
31
32 int main()
33 {
34     Solution s;
35     vector<int> nums;
36     vector<int> v;
37     nums.push_back(2);
38     nums.push_back(7);
39     nums.push_back(11);
40     nums.push_back(15);
41     v = s.twoSum(nums, 9);
42     cout << v[0] << ' ' << v[1] << endl;
43     /*
44     for (map<int, int>::iterator it=vmap.begin(); it!=vmap.end(); it++)
45         cout << it->first << ":" << it->second << endl;
46     */
47     return 0;
48 }
49

```

## 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

由于贝叶斯决策比较理论，也比较好理解，这里不过多介绍。

## Question 1

**概率密度估计** 概率密度估计主要分为参数概率密度估计和非参数概率密度估计。参数概率密度估计包括极大似然估计和贝叶斯估计，极大似然估计一般都是假设样本服从某个分布，然后估计参数的准确数值；贝叶斯估计直接估计样本服从的分布。

**非参数概率密度估计方法基本思路：**为了估计样本  $x$  处的概率密度函数  $p(x|w_i)$ ，构造一系列包含点  $x$  的区域  $R_1, R_2, \dots, R_n, \dots$ ，记  $v_n$  为  $R_n$  的体积， $k_n$  为落在  $R_n$  中的样本个数， $p_n(x)$  表示对  $p(x|w_i)$  的第  $n$  次估计， $p_n(x) = \frac{k_n/n}{v_n}$ 。

由上式，有两种构造序列的方法来估计概率密度函数：

(1) parzen 窗估计：固定局部区域体积  $v$ ,  $k$  变化。 $V_n = \frac{1}{\sqrt{v_n}}$ ，实际上是每一次估计时体积不变。体积会逐渐收敛，要求  $k_n$  和  $k_n/n$  能够保证  $p_n(x)$  能够收敛到  $p(x)$ ；

(2) k 近邻估计：固定局部样本数  $k$ ,  $V$  变化。 $K_n = \sqrt{n}$ ，实际上是每一次估计时局部样本数目不变。区

域会逐渐生长。

计算：

(1)parzen:

$R_n$  是一个  $d$  维的超立方体，宽度为  $h_n$ ,  $v_n = h_n^d$ ( $x$  是中心点). 如果  $x_i$  落在  $v_n$  中,  $\phi(\frac{x-x_i}{h_n}) = 1$ , 否则为 0, 则  $k_n = \sum_{i=1}^n \phi(\frac{x-x_i}{h_n})$ . 推出  $p_n(x) = \frac{1}{v_n} \sum_{i=1}^n \phi(\frac{x-x_i}{h_n})$ .

(2)k 近邻估计：

$k_i$ : 区域内第  $i$  类的样本数;

$k$ : 区域内所有的样本数;

$n_i$ : 所有样本中第  $i$  类的样本数;

$n$ : 所有样本数。

k 近邻后验概率密度估计：(计算方式有两种)

$$1. p_n(x, w_i) = \frac{k_i/n}{v}$$

$$p_n(w_i|x) = \frac{p_n(x, w_i)}{p_n(x)} = \frac{p_n(x, w_i)}{\sum_{j=1}^c p_n(x, w_j)} = \frac{\frac{k_i/n}{v}}{\sum_{j=1}^c \frac{k_j/n}{v}} = \frac{k_i}{k}$$

$$2. p_n(x|w_i) = \frac{k_i/n_i}{v}, p(w_i) = \frac{n_i}{n}, p(x) = \frac{k/n}{v}$$

$$p_n(w_i|x) = \frac{p_n(x|w_i)p(w_i)}{p(x)} = \frac{k_i}{k}$$

## Question 2

数据聚类数据聚类方法分为均值聚类，分级聚类和谱聚类三种。(这里将重点介绍一些数学方面的推导过程，而不是像市场课程只介绍技术路线。)

### 一. 均值聚类

技术路线很简单，开始随机初始化  $k$  个聚类中心，然后选取离聚类中心最近的  $k$  个样本，求它们的均值作为新的聚类中心，这样一直迭代直至收敛。下面将介绍为什么采取均值作为新的聚类中心。

从总体样本的概率密度函数出发： $p(x|\theta) = \sum_{j=1}^c p(x|w_j, \theta_j)P(w_j)$ ,  $P(w_j)$  是各个类别的先验概率，称为混合比例。这里的主要任务是要估计  $\theta$ ，一旦  $\theta$  得到估计，可以将上述混合密度分解为多个已知的密度成分，并且可以采用最大化后验概率来确定样本的类别，考虑最大似然估计。

### 8.4 最大似然估计

• 目标：估计一个  $\hat{\theta}$  使  $p(D|\theta)$  最大。

– 考虑对数似然 (log-likelihood):

$$\begin{aligned} f_{lh}(\theta) &= \ln(p(D|\theta)) \\ &= \sum_{k=1}^n \ln(p(x_k|\theta)) \\ &= \sum_{k=1}^n \ln \left( \sum_{j=1}^c p(x_k|\omega_j, \theta_j) P(\omega_j) \right) \end{aligned}$$

$$f_{lh}(\theta) \text{ 对参数 } \theta \text{ 的梯度(假定参数独立): } p(x_k|\theta) = \sum_{j=1}^c p(x_k|\omega_j, \theta_j) P(\omega_j)$$

$$\begin{aligned} \nabla_{\theta} f_{lh}(\theta) &= \sum_{k=1}^n \frac{1}{p(x_k|\theta)} \nabla_{\theta} p(x_k|\theta) \\ &= \sum_{k=1}^n \frac{1}{p(x_k|\theta)} \nabla_{\theta} (p(x_k|\omega_i, \theta_i) P(\omega_i)) \\ &= \sum_{k=1}^n \frac{P(\omega_i)}{p(x_k|\theta)} \nabla_{\theta} (p(x_k|\omega_i, \theta_i)) \\ &= \sum_{k=1}^n \frac{P(\omega_i)}{p(x_k|\theta)} p(x_k|\omega_i, \theta_i) \nabla_{\theta} \ln(p(x_k|\omega_i, \theta_i)) \\ &= \sum_{k=1}^n \frac{P(\omega_i, x_k|\theta)}{p(x_k|\theta)} \nabla_{\theta} \ln(p(x_k|\omega_i, \theta_i)) \\ &= \sum_{k=1}^n P(\omega_i | x_k, \theta) \nabla_{\theta} \ln(p(x_k|\omega_i, \theta_i)) \end{aligned}$$



第23页

(a) 似然函数



第24页

(b) 似然函数的梯度

图 16: 似然函数及梯度

这里推导过程中的  $P(w_i|x_k, \theta)$  不是概率密度，而是将  $x_k$  代入  $p(w_i|x, \theta)$  函数中的取值。令上述  $c$

个方程等于 0(可以用牛顿法直接求解  $\theta$ ), 以及添加先验概率条件, 可以得到两个条件。//

## 8.4 最大似然估计

- 令梯度等于零, 可得如下  $c$  个方程:

$$\sum_{k=1}^n P(\omega_i | \mathbf{x}_k, \hat{\boldsymbol{\theta}}) \nabla_{\boldsymbol{\theta}_i} \ln(p(\mathbf{x}_k | \omega_i, \hat{\boldsymbol{\theta}}_i)) = 0, \quad i=1,2,\dots,c$$

求解上述方程可得待估计的  $\hat{\boldsymbol{\theta}}$ 。

- 进一步:** 当未知量中包含先验概率  $P(\omega_i)$  (即混合比例) 时, 应限制如下两个条件:

$$P(\omega_i) \geq 0, \quad i=1,2,\dots,c, \quad \text{且} \quad \sum_{i=1}^c P(\omega_i) = 1.$$

## 8.4 最大似然估计

- 实际上, 如果似然函数可微, 且  $P(\omega_i) \neq 0$ , 那么  $P(\omega_i)$  和  $\hat{\boldsymbol{\theta}}_i$  必然同时满足以下条件:

条件1:  $\hat{P}(\omega_i) = \frac{1}{n} \sum_{k=1}^n \hat{P}(\omega_i | \mathbf{x}_k, \hat{\boldsymbol{\theta}})$

条件2:  $\sum_{k=1}^n \hat{P}(\omega_i | \mathbf{x}_k, \hat{\boldsymbol{\theta}}) \nabla_{\boldsymbol{\theta}_i} \ln(p(\mathbf{x}_k | \omega_i, \hat{\boldsymbol{\theta}}_i)) = 0, \quad i=1,2,\dots,c$

其中,  $\hat{P}(\omega_i | \mathbf{x}_k, \hat{\boldsymbol{\theta}}) = \frac{p(\mathbf{x}_k | \omega_i, \hat{\boldsymbol{\theta}}_i) \hat{P}(\omega_i)}{\sum_{j=1}^c p(\mathbf{x}_k | \omega_j, \hat{\boldsymbol{\theta}}_j) \hat{P}(\omega_j)}$

全概率公式

(a)  $c$  个方程及添加的约束条件

(b) 得到的两个条件

图 17: 要求解的方程以及求解的结果

推导过程如下, 采用拉格朗日乘子法求解:

### • 关于条件1的证明

- 首先, 考虑所有变量时对数似然函数可以写成:

$$f_n(\boldsymbol{\theta}, \boldsymbol{\alpha}) = \sum_{k=1}^n \ln(p(\mathbf{x}_k | \boldsymbol{\theta})) = \sum_{k=1}^n \ln \left( \sum_{i=1}^c p(\mathbf{x}_k | \omega_i, \boldsymbol{\theta}_i) P(\omega_i) \right)$$

其中引入新记号:  $\boldsymbol{\alpha} = [\alpha_1, \alpha_2, \dots, \alpha_c]^T = [P(\omega_1), P(\omega_2), \dots, P(\omega_c)]^T$

- 然后, 拉格朗日函数可以写成:

$$L(\boldsymbol{\theta}, \boldsymbol{\alpha}) = f_n(\boldsymbol{\theta}, \boldsymbol{\alpha}) + \lambda \left( \sum_{i=1}^c \alpha_i - 1 \right)$$

拉格朗日乘子

### • 关于条件1的证明(续)

- 求目标函数关于变量的偏导数, 并令其等于 0:

$$\frac{\partial L(\boldsymbol{\theta}, \boldsymbol{\alpha})}{\partial \boldsymbol{\theta}_i} = \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \boldsymbol{\theta}_i)}{p(\mathbf{x}_k | \boldsymbol{\theta})} + \lambda = 0, \quad i=1,2,\dots,c$$

- 在方程的两边乘以  $\alpha_i$ , 并将  $c$  个方程相加, 可得

$$\begin{aligned} & \sum_{i=1}^c \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \boldsymbol{\theta}_i) \alpha_i}{p(\mathbf{x}_k | \boldsymbol{\theta})} + \lambda \sum_{i=1}^c \alpha_i = 0 \\ & \Rightarrow \lambda = - \sum_{i=1}^c \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \boldsymbol{\theta}_i) \alpha_i}{p(\mathbf{x}_k | \boldsymbol{\theta})} = - \sum_{i=1}^c \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \boldsymbol{\theta}_i) \alpha_i}{p(\mathbf{x}_k | \boldsymbol{\theta})} \\ & = - \sum_{i=1}^c \frac{p(\mathbf{x}_i | \boldsymbol{\theta})}{p(\mathbf{x}_i | \boldsymbol{\theta})} = -\eta \end{aligned}$$

### • 关于条件1的证明(续)

- 最后由如下公式

$$\frac{\partial L(\boldsymbol{\theta}, \boldsymbol{\alpha})}{\partial \alpha_i} = \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \boldsymbol{\theta}_i)}{p(\mathbf{x}_k | \boldsymbol{\theta})} + \lambda = 0, \quad i=1,2,\dots,c$$

- 可得

$$\sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \boldsymbol{\theta}_i) \alpha_i}{p(\mathbf{x}_k | \boldsymbol{\theta})} = n \alpha_i, \quad i=1,2,\dots,c$$

$$\Rightarrow \alpha_i = \frac{1}{n} \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \boldsymbol{\theta}_i) \alpha_i}{p(\mathbf{x}_k | \boldsymbol{\theta})} = \frac{1}{n} \sum_{k=1}^n \frac{p(\mathbf{x}_k | \omega_i, \boldsymbol{\theta}_i) p(\omega_i)}{p(\mathbf{x}_k | \boldsymbol{\theta})}$$

因此, 条件1得证。

(a) S1

(b) S2

(c) S3

图 18: 使用拉格朗日乘子法证明条件一的过程

现在做四个假设:

- 样本服从的先验概率密度为高斯分布, 即:  $\ln p(x|w_i, \mu_i) = -\ln((2\pi)^{\frac{d}{2}} |\Sigma|^{\frac{1}{2}}) - \frac{1}{2}(x - \mu_i)^T \Sigma^{-1}(x - \mu_i)$ ;
- 各个类别出现的先验概率相等;
- 每个样本以概率 1 属于一个类;
- 协方差矩阵为一个很小的数  $\epsilon$  乘以单位阵。

在这四个假设之下, 来求解参数  $\mu_i = \frac{1}{n_i} \sum_{x_k \in w_i} x_k, i=1,2,\dots,c$ , 得到 K-均值聚类算法。

## 二. 分级聚类

分级聚类将树的每一个叶子节点当成一个样本, 归到同一个根节点的视为一类。主要分类方式分为两种, 一种是先考虑样本之间的距离, 合并了一些之后, 需要考虑类与类之间的距离(自底向上); 第二种是从所有样本中逐渐细分样本(自顶向下)。K-均值聚类需要事先知道聚类的个数, 分级聚类貌似可以避免, 但是从哪层 level 划分也是需要探究的, 聚类个数仍然是个问题。

## 三. 谱聚类

联系均值聚类方法, 均值聚类假设样本分布服从高斯分布, 默认均值是聚类中心。如果不同类别的样本

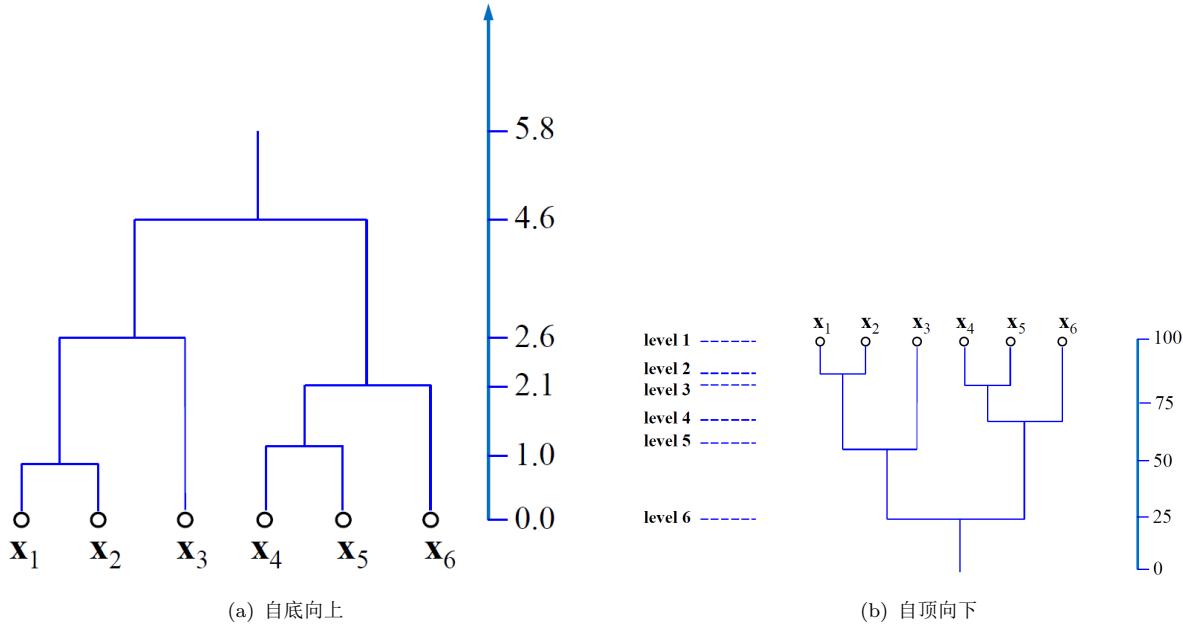


图 19: 两种分级聚类的方式

分布在多个同心圆上，均值分类将失效。因为同心圆的均值都在同一点，这样所有类别都被分成同一类。

定理：设  $G$  为一个具有非负连接权重的无向图，由  $G$  导出的拉普斯矩阵  $L$  的零特征值重数等于图  $G$  的连通子图的个数  $k$ 。这个定理将聚类与图论联系起来。证明过程略

使用图论来求做谱聚类，首先要将样本点转换成一张图，构造图的方式有全连接或局部连接。局部连接中需要用到  $k$  近邻 ( $\epsilon$  邻域) 来构造图，即对每个数据点  $x_i$ ，首先在所有样本中找出不包含  $x_i$  的  $k$  个最邻近的样本点，然后  $x_i$  与每个邻近样本点均有一条边相连，从而完成图构造。图构造好之后要考虑图的  $Laplace$  矩阵 (后续的操作都在围绕着图的  $Laplace$  矩阵操作)。

图的  $Laplace$  矩阵目前有三种主流的构造方式：

1. 直接使用图的  $Laplace$  矩阵  $L$ ;
2.  $L_{sym} = D^{-\frac{1}{2}} L D^{-\frac{1}{2}} = I - D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$ ,  $D$  为度矩阵,  $W$  为邻接矩阵;
3.  $L_{rw} = D^{-1} L = I - D^{-1} W$ .

$Laplace$  矩阵构造好之后，需要求解  $Laplace$  矩阵的最小的  $k$  个特征值对应的特征向量，将这  $k$  个特征向量依次排列组合在一起，得到  $n * k$  的矩阵，则每一行的  $1 * k$  向量将作为该样本点的新特征值，然后再进行均值聚类。所以谱聚类的过程实际上是一个特征选择 + 均值聚类的过程。分别对应下面三个算法过程。

### Question 3

**Boosting** Boosting 是集成学习中的常用技术手段，通过改变训练样本的权重，学习多个分类器，并将这些分类器进行线性组合，以达到非线性分类器的效果。Adaboost 是 Boosting 方法中最具代表性的方法，Adaboost 算法在深度学习流行以前是生物特征识别 (特别是人脸识别) 的主要方法。

Adaboost 核心思想：在分类问题中，它通过改变训练样本的权重，学习多个分类器，并将这些分类器进行组合，提高分类性能。将弱分类器的线性组合作为强分类器，给每个样本多设置一个权重，计算所得的误差函数也是各个样本的误差乘以权重得到，而不是均值误差。当某次分类，一些样本被分错时，

Un-normalized (classical) Spectral Clustering—Algorithm 1	Normalized Spectral Clustering—Algorithm 2 (Shi 算法)	Normalized Spectral Clustering—Algorithm 3 (Ng 算法)
<pre> 1 input: similarity matrix <math>\mathbf{W}</math>, number <math>k</math> of clusters 2 compute the un-normalized Laplacian matrix <math>\mathbf{L} = \mathbf{D} - \mathbf{W}</math> 3 compute the first <math>k</math> eigenvectors <math>\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k</math> of the <math>\mathbf{L}</math> 4 let <math>\mathbf{U} \in \mathbb{R}^{n \times k}</math> be the matrix containing the vectors <math>\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k</math>, namely, <math>\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k] \in \mathbb{R}^{n \times k}</math> 5 for <math>i = 1, 2, \dots, n</math>, let <math>\mathbf{y}_i \in \mathbb{R}^k</math> be the vector corresponding to the <math>i</math>-th row of <math>\mathbf{U}</math>. 6 cluster the points <math>\{\mathbf{y}_i\}_{i=1,\dots,n}</math> in <math>\mathbb{R}^k</math> with k-means algorithm into clusters <math>A_1, A_2, \dots, A_k</math> 7 output <math>A_1, A_2, \dots, A_k</math>.</pre>	<pre> 1 input: similarity matrix <math>\mathbf{W}</math>, number <math>k</math> of clusters 2 compute the unnormalized Laplacian matrix <math>\mathbf{L} = \mathbf{D} - \mathbf{W}</math> 3 compute the first <math>k</math> generalized eigenvectors <math>\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k</math> of the generalized eigen-problem <math>\mathbf{L}\mathbf{u} = \lambda \mathbf{D}\mathbf{u}</math> 4 Let <math>\mathbf{U} \in \mathbb{R}^{n \times k}</math> be the matrix containing the vectors <math>\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k</math>, namely, <math>\mathbf{U} = [\mathbf{u}_1, \mathbf{u}_2, \dots, \mathbf{u}_k] \in \mathbb{R}^{n \times k}</math> 5 for <math>i = 1, 2, \dots, n</math>, let <math>\mathbf{y}_i \in \mathbb{R}^k</math> be the vector corresponding to the <math>i</math>-th row of <math>\mathbf{U}</math>. 6 cluster the points <math>\{\mathbf{y}_i\}_{i=1,\dots,n}</math> in <math>\mathbb{R}^k</math> with k-means algorithm into clusters <math>A_1, A_2, \dots, A_k</math> 7 output <math>A_1, A_2, \dots, A_k</math>.</pre>	<small>On spectral clustering: analysis and an algorithm. NIPS 2002.</small>

(a) algorithm1

(b) algorithm2

(c) algorithm3

图 20: 三种构造 Laplace 矩阵方式分别对应的算法过程

则分错的样本权重需要提高。在 Adaboost 当中，有两个权重，一个是样本的权重，一个是分类器的权重。

### 1. 在每轮训练中，如何改变训练数据的权值或分布？

提高那些被前一轮弱分类器分错的样本的权重，降低已经被正确分类的样本的权重。错分的样本将在下一轮弱分类器中得到更多关注。于是分类问题被一系列弱分类器“分而治之”。

### 2. 如何将一系列的弱分类器组合成一个强分类器？

关于弱分类器的组合，Adaboost 的做法是：采用加权（多数）表决的方法。具体地，加大分类错误率较小的弱分类器的权重，使其在表决中起更大的作用。

具体算法流程如下：

## 8.8 Adaboost

### • Adaboost 算法步骤

- 输入训练数据集：  
 $T = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_n, y_n)\}$
- 输入弱学习算法
- (1) 初始化训练数据的权值分布  
 $D_1 = \{w_{11}, w_{12}, \dots, w_{1n}\}, w_{1j} = 1/n, i = 1, \dots, n$
- (2) 对  $m = 1, 2, \dots, M$
- (2a) 使用具有权值分布  $D_m$  的训练数据，学习基本分类器  
 $G_m(\mathbf{x}) : X \rightarrow \{-1, +1\}$

中国科学院大学

(a) Step1

### Adaboost 算法步骤(续)

- (2b) 计算  $G_m(\mathbf{x})$  在训练数据集上的分类错误率(加权):  
 $c_m = P(G_m(\mathbf{x}_i) \neq y_i) = \sum_{i=1}^n w_{im} I(G_m(\mathbf{x}_i) \neq y_i)$
- (2c) 计算  $G_m(\mathbf{x})$  的贡献系数:  
 $\alpha_m = \frac{1}{2} \ln \frac{1-c_m}{c_m}$

$\alpha_m$  表示  $G_m(\mathbf{x})$  在最终分类器中的重要性。当  $c_m \leq 0.5$  时， $\alpha_m \geq 0$ 。同时， $\alpha_m$  将随着  $c_m$  的减小而增大。  
所以，分类误差率越小的基本分类器在最终分类器中的作用越大。

中国科学院大学

(b) Step2

### Adaboost 算法步骤(续)

- (2d) 更新训练数据集的权重分布:  
 $D_{m+1} = \{w_{m+1,1}, w_{m+1,2}, \dots, w_{m+1,n}\}$
- 具体计算如下:  
 $w_{m+1,i} = \frac{w_{mi}}{Z_m} \times \begin{cases} \exp(-\alpha_m), & \text{if } G_m(\mathbf{x}_i) = y_i \\ \exp(\alpha_m), & \text{if } G_m(\mathbf{x}_i) \neq y_i \end{cases}$ 

若正确分类，减少权重；否则，增加权重

 $= \frac{w_{mi}}{Z_m} \times \exp(-\alpha_m y_i G_m(\mathbf{x}_i))$ 

其中， $Z_m$  是规范化因子，它使  $D_{m+1}$  成为一个概率分布；  
 $Z_m = \sum_{i=1}^n w_{mi} \exp(-\alpha_m y_i G_m(\mathbf{x}_i))$

中国科学院大学

(c) Step3

### Adaboost 算法步骤(续)

- (3) 构建基本分类器的线性组合:

$$f(\mathbf{x}) = \sum_{m=1}^M \alpha_m G_m(\mathbf{x})$$

对于两类分类问题，得到最终的分类器：

$$G(\mathbf{x}) = \text{sign}(f(\mathbf{x})) = \text{sign}\left(\sum_{m=1}^M \alpha_m G_m(\mathbf{x})\right)$$

中国科学院大学

(d) Step4

图 21: Adaboost 算法流程

下面给出一个例子：

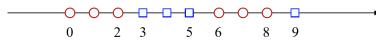
## 8.8 Adaboost

$\text{sign}(x \cdot v)$

- 例子：给定如表所示训练数据。假设弱分类器由“如果  $x < v$ ，则  $x$  属于第一类；如果  $x > v$ ，则  $x$  属于第二类”产生，其阈值使该分类器在训练数据集上分类误差率最低。试用Adaboost算法学习一个强分类器。

序号	1	2	3	4	5	6	7	8	9	10
x	0	1	2	3	4	5	6	7	8	9
y	1	1	1	-1	-1	-1	1	1	1	-1

上表中：共10个一维空间的样本，x表示样本，y表示标签。



李航著：统计学习方法(第8章),清华大学出版社,2012

图 22: 例题

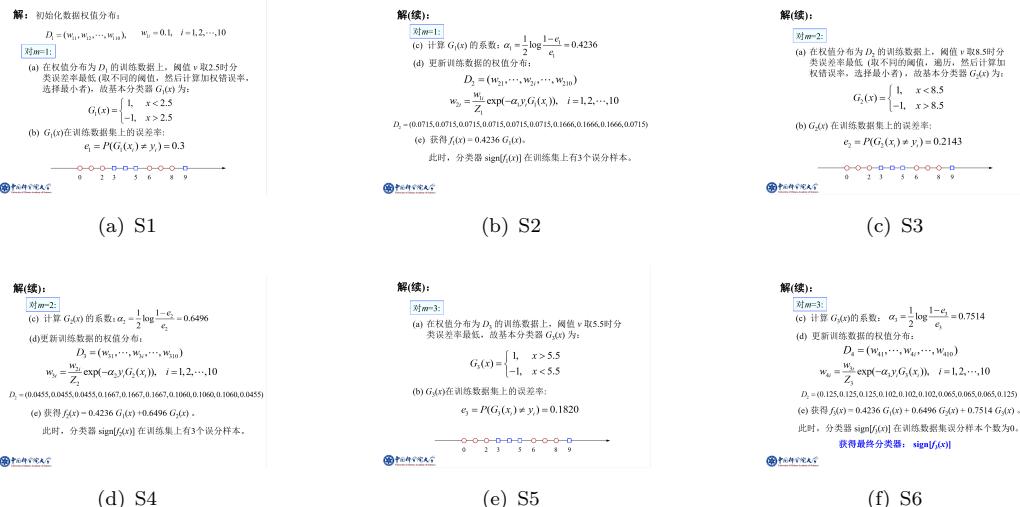


图 23: 解答过程

## Question 4

### 支持向量机

支持向量机是 1995 年-2005 年十年期间非常流行的方法，经典的 BP 算法于 1980 年提出，然后受限于计算能力，当时只能训练 3-5 层的网络，这是 SVM 和 Boosting 出现，因为有理论支撑。

1. 首先介绍 VC 维的概念：对于一个模型，它可以以 100% 的精度二分类  $n$  个样本，最大的  $n$  称为它的 VC 维。VC 维实际上刻画了一个模型的复杂程度，一个分类  $n$  维样本的线性分类器参数有  $n+1$  个，其 VC 维也是  $n+1$ 。但是参数的个数和 VC 维并不等同，如  $f(x) = \text{Asin}(wx)$  参数只有两个，但是它可以分开任意个样本，所以 VC 维是无穷大；

2. 在二分类线性分类器中，对于一条直线，会出现 Large Margin，即样本点离这条直线的最小距离。Margin 越大，则该分类器的 VC 维就越小，泛化能力就越好；

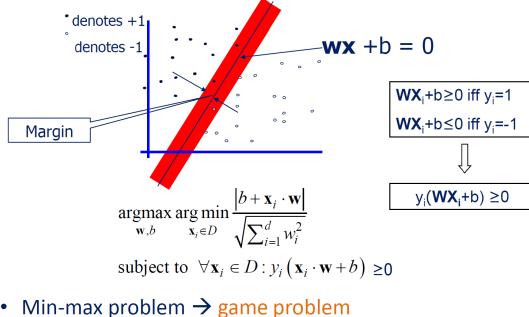
3. SVM 分为 Hard-Margin SVM(两类完全可分) 和 Soft-Margin SVM(两类不完全可分)。

(1) Hard-Margin SVM:

样本完全可分，只需最大化样本离分界面的最小距离即可，目标函数及限制条件可以写为：

可以化简为：

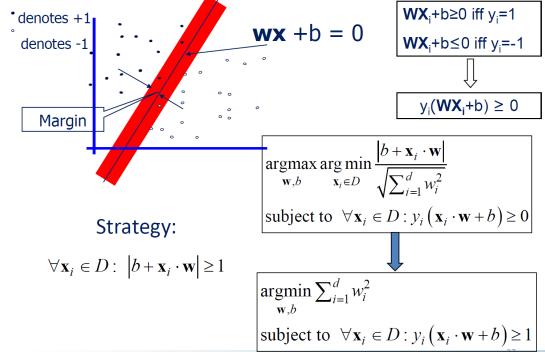
### Estimate Margin (Method 1)



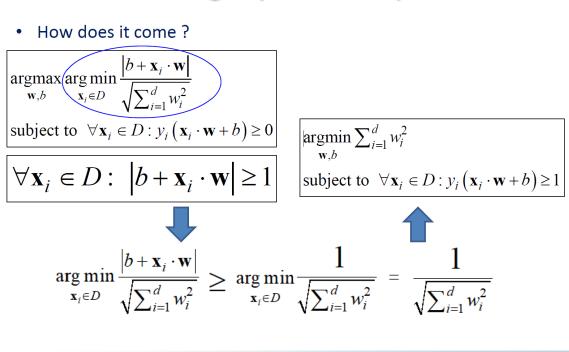
26

图 24: Hard-Margin SVM 目标函数及限制条件

### Estimate Margin (Method 1)



### Estimate Margin (Method 1)



(a) S1

(b) S2

图 25: 化简

都是二次规划问题(目标函数二次,限制条件一次),求解方法已经非常成熟。

## (2) Soft-Margin SVM:

实际情况中样本并不是完全可分的,这时引入松弛变量,即允许分错,但是又不允许太多。目标函数即限制条件可以写为:

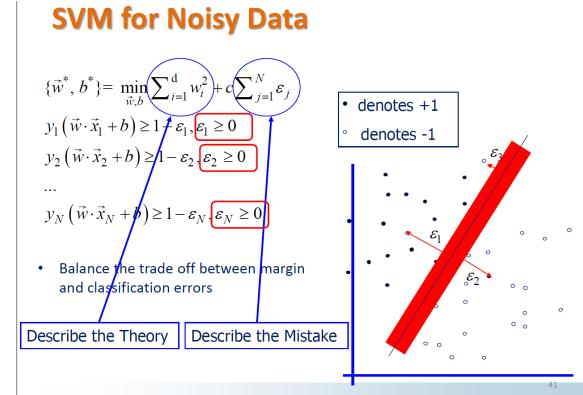


图 26: Soft-Margin SVM 目标函数及限制条件

进一步分析,引入的松弛变量实际上就是一个  $hinge(x) = \max\{1 - x, 0\}$ 。 $\epsilon \geq 1 - y(f(x)) = 1 - yf(x)$ :

- (1) 若  $yf(x) \geq 1$ , 即  $1 - yf(x) \leq 0$ , 则分类正确, 此时  $\epsilon = 0$ ;
  - (2) 若  $yf(x) \leq 1$ , 即  $1 - yf(x) \geq 0$ , 则分类错误或者样本点在 Margin 里面, 此时  $\epsilon > 0$ , 且  $\epsilon$  越大, 表示分错, 代价应当越大;
- 综上应有:  $\epsilon = hinge(yf(x))$ , 函数图像如下:

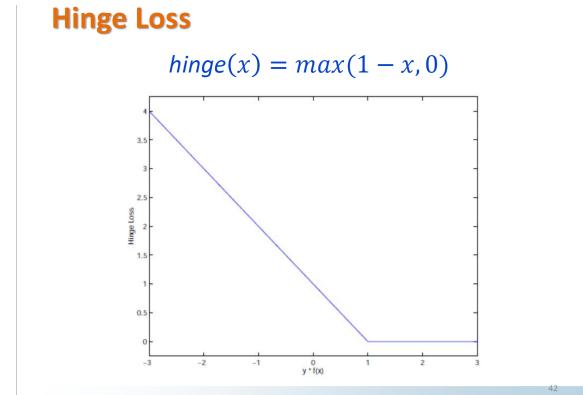


图 27: hinge(x)

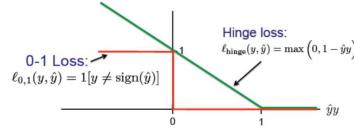
将目标函数改良为:

上述改良过程主要有两个要点,一是通过引入松弛变量,给损失函数添加了  $\text{relu}(hinge(x))$  函数;二是使用正则项。

4. 不管是 Hard-Margin SVM 还是 Soft-Margin SVM 他们的对偶问题(后续详细介绍对偶问题)都是一样的:

每个样本点都有一个参数  $\alpha$ :

## Hinge Loss



Hinge loss upper bounds 0/1 loss!

→ It is the tightest convex upper bound on the 0/1 loss

• The SVM is a Tikhonov regularization problem, using the hinge loss:

$$\underset{f \in \mathcal{H}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^n (1 - y_i f(x_i))_+ + \lambda \|f\|_{\mathcal{H}}^2.$$

-43-

图 28: 改良的目标函数

## Dual Problem

- We can transform the problem to its dual

$$\begin{aligned} \max. W(\alpha) &= \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{subject to } \alpha_i &\geq 0, \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned}$$

Dot product of X

$\alpha$ 's → New variables  
(Lagrangian multipliers)

- This is a convex quadratic programming (QP) problem
  - Global maximum of  $\alpha_i$  can always be found
  - well established tools for solving this optimization problem

- Note:  $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$

-49-

(a) Hard-Margin SVM

## Dual Problem for Soft-Margin SVM

- The dual of the problem is

$$\begin{aligned} \max. W(\alpha) &= \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ \text{subject to } C &\geq \alpha_i \geq 0, \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned}$$

- $\mathbf{w}$  is also recovered as  $\mathbf{w} = \sum_{j=1}^s \alpha_j y_j \mathbf{x}_{t_j}$
- The only difference with the linear separable case is that there is an upper bound  $C$  on  $\alpha_i$
- Once again, a QP solver can be used to find  $\alpha_i$  efficiently!!!

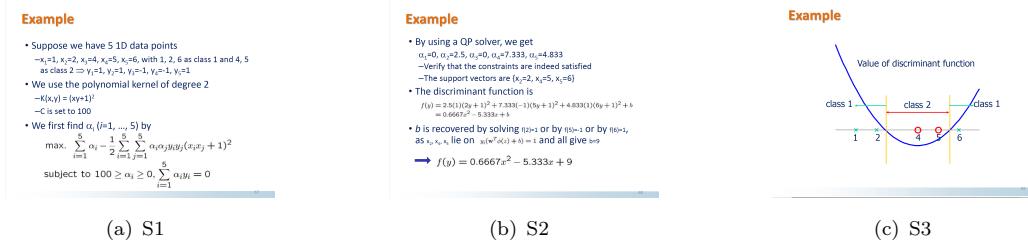
-53-

(b) Soft-Margin SVM

图 29: 对偶问题

- (1)  $\alpha = 0$ , 对应样本点在边界之外, 分类正确;
- (2)  $0 < \alpha < C$ , 对应样本点在边界上, 且为支撑向量;
- (3)  $\alpha = C$ , 对应样本点在边界内, 且为支撑向量;

从上述过程可以看出, 支持向量机实际上只能解决线性问题, 但是通过核函数的引进, 非线性问题通过升维可以变为线性问题。支持向量机通过将传统的求解模型 (有参数  $W, b$ ) 转换到其对偶形式 (只有参数  $\alpha$ , 每一个样本都有一个参数  $\alpha_i$ ), 这样避免了维数灾难的问题, 因为在对偶形式中  $\alpha$  只与样本个数有关, 并且解是全局的。最后给出一个实例:



(a) S1

(b) S2

(c) S3

图 30: 实例

支持向量机中的核方法实际上是在将数据升维, 一系列样本在低维空间中不能用线性分类器分开, 需要通过核方法做升维 (对偶问题中只出现了样本的核函数形式), 升维到高维空间中就可以用线性分类器来做分类

关于如何加速训练 SVM, 有两点改进:

- (1) 只有支持向量会影响决策边界: 训练时采用组块策略, 不是一次选择所有样本。而是每次添加一些新样本, 确定支撑向量, 再添加新样本, 再确定新样本, 一直到支撑向量不变为止, 即收敛;
- (2) 当达到最优结果时, 此时 KKT 条件是满足的: 每次添加完新数据之后, 用序列最小化优化方法 (SMO) 来计算  $\alpha$ , 可以每次选择计算两个  $\alpha$ 。很多文献中有关于如何选择两个乘子的方法, 如果每次随机选择  $\alpha$ , 会导致收敛速度缓慢。

上述介绍了支持向量机分类两类问题, 对于多分类问题, 有 *oneVSall*, *oneVSome*, *DAGSVM*, *halfVShalf*, *LatticeSVM* 等, 或者基于二分类的理论直接建立多分类的理论。

关于深度学习与支持向量机的区别:

- (1) 支持向量机中的特征升维, 是直接学习核函数  $K(x, y) = \phi(x)\phi(y)$ , 而深度学习是直接学习函数  $\phi(x)$  和  $\phi(y)$ ;
- (2) SVM 的目标函数是凸函数, 在学习的时候可以直接到达全局最优解; 深度学习不是凸函数, 需要一些迭代策略 (如带冲量项的梯度下降);
- (3) 深度学习的解一般不是最优解, 但时常有效的原因: SVM 实际上是对问题的一种近似, 但是可以精确求解; 深度学习虽然不是问题的最优解, 但是对问题的直接求解, 会跳过支持向量机由于模型近似带来的固有误差。

## Question 5

对偶问题。对偶问题的实质是要给目标函数估计一个界: 最小化一个目标函数, 对偶问题是估计该目标函数的最大下界; 最大化一个目标函数, 对偶问题是估计该目标函数的最小上界。

例如，原问题是  $\min g(x)$ ，则对偶问题可以写成： $\max f(y)$ ，其中：对于任意一个  $x$ ,  $g(x)$  都是  $f(y)$  的上界；对于任何一个  $y$ ,  $f(y)$  都是  $g(x)$  的下界。

### 一、线性规划的对偶问题：

对于一个无约束的目标函数，采用求驻点（即求导）方法即可求得最优解，如果添加约束条件，可以使用 lagrange 方法将问题转换为一个无约束的问题，再进行求导。具体转换方式如下：

#### Standard form of constrained optimization problems

- Consider the following constrained optimization problem (might be non-convex).

$$\begin{aligned} \min \quad & f_0(x) \\ \text{s.t.} \quad & f_i(x) \leq 0 \quad i=1, \dots, m \\ & h_i(x) = 0 \quad i=1, \dots, p \end{aligned}$$

- Here the variables  $x \in \mathbb{R}^n$  and we use  $\mathcal{D} = \bigcap_{i=0}^m \text{dom } f_i \cap \bigcap_{i=1}^p \text{dom } h_i$  to represent the domain of definition. We use  $p^*$  to represent the optimal value of the problem.

(a) 带约束的原问题

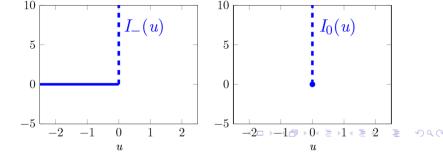
#### An equivalent unconstrained optimization problem

- We can transform this **constrained optimization** problem into an equivalent **unconstrained optimization** problem:

$$\min f_0(x) + \sum_{i=1}^m L_i(f_i(x)) + \sum_{i=1}^p I_0(h_i(x))$$

where  $x \in \mathcal{D}$ ,  $L_i(u)$  and  $I_0(u)$  are indicator functions for non-positive reals and the set  $\{0\}$ , respectively:

$$L_i(u) = \begin{cases} 0 & u \leq 0 \\ \infty & u > 0 \end{cases} \quad I_0(u) = \begin{cases} 0 & u = 0 \\ \infty & u \neq 0 \end{cases}$$



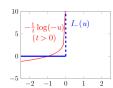
(b) 无约束的优化问题

图 31: 约束问题转换为无约束问题

转换后的无约束目标函数是不可导的，无法直接求解。结合对偶问题的描述，求解一个目标函数的最小值，实际上可以用该目标函数下界的最大值来估计，利用不同的下界估计函数，对应着不同的方法。如下图，用对数障碍函数做下界，对应着内点法；用平方函数做下界，对应着罚函数法；用线性函数做下界，对应着 lagrange 乘子法。

关于线性函数估计下界，即 lagrange 乘子法，lagrange 乘子可以当作是乘子系数，也是对偶问题的变

#### Approximating $L_i(u)$ using a differentiable function (1)



An approximation to  $L_i(u)$  is the **logarithmic barrier function**:

$$L_i(u) = -\frac{1}{t} \log(-u) \quad (t > 0)$$

The difference between  $L_i(u)$  and  $L_i(u)$  decreases as  $t$  increases. This approximation was used in the interior point method.

(a) 对数障碍函数估计下界

#### Approximating $L_i(u)$ using a differentiable function (2)



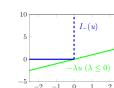
Another approximation to  $L_i(u)$  is a **penalty function**:

$$L_i(u) = \phi(u) = \begin{cases} \frac{1}{2}(t-1)u^2 & u \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

The penalty function "penalizes" any  $u$  which is greater than zero. It is a "hands-off" method for converting constrained problems into unconstrained problems, to which an initial feasible solution is easy to obtain. However, it is not always effective because the minimum of the objective function is undefined out of the feasible set or the unconstrained problem becomes ill-conditioned as  $t$  increases.

(b) 平方项函数估计下界

#### Approximating $L_i(u)$ using a differentiable function (3)



Another approximation to  $L_i(u)$  is a simple **linear function**:

$$L_i(u) = \phi(u) = \begin{cases} -\lambda u & (\lambda \leq 0) \\ 0 & (\lambda > 0) \end{cases}$$

Despite the considerable difference between  $L_i(u)$  and  $L_i(u)$ ,

$L_i(u)$  still provides lower bound information of  $L_i(u)$ .

$L_i(u)$  is called the **lagrange multiplier**.

$L_i(u)$  is called the **lagrange function**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L_i(u)$  is called the **lagrange dual problem**.

$L_i(u)$  is called the **lagrange dual value**.

$L_i(u)$  is called the **lagrange dual variable**.

$L_i(u)$  is called the **lagrange dual function**.

$L$

$$\begin{aligned}
& \min x_1 + 2x_2 \\
\text{s.t.} & 3x_1 + 4x_2 \geq 5 \\
& 6x_1 + 7x_2 \geq 8 \\
& x_1, x_2 \geq 0
\end{aligned}$$

### 1. 对偶问题推导

(1) 写出 lagrange 函数

lagrange 函数为:  $L(x_1, x_2, \lambda_1, \lambda_2) = x_1 + 2x_2 - \lambda_1(3x_1 + 4x_2 - 5) - \lambda_2(6x_1 + 7x_2 - 8) = 5\lambda_1 + 8\lambda_2 + (1 - 3\lambda_1 - 6\lambda_2)x_1 + (2 - 4\lambda_1 - 7\lambda_2)x_2$ .

当  $\lambda_1 \geq 0, \lambda_2 \geq 0$ , 并且  $x_1, x_2$  是可行的, 则  $L(x_1, x_2, \lambda_1, \lambda_2)$  是原函数的一个下界。

(2) 求解 lagrange 函数的下界

lagrange 对偶函数

$$g(\lambda_1, \lambda_2) = \inf_{x_1, x_2 \in D} L(x_1, x_2, \lambda_1, \lambda_2) = \begin{cases} 5\lambda_1 + 8\lambda_2 & 3\lambda_1 + 6\lambda_2 - 1 \leq 1, 4\lambda_1 + 7\lambda_2 - 2 \leq 2 \\ -\infty & \text{otherwise} \end{cases} \quad (1)$$

(3) 写出对偶问题

$$\begin{aligned}
& \max 5\lambda_1 + 8\lambda_2 \\
\text{s.t.} & 3\lambda_1 + 6\lambda_2 \leq 1 \\
& 4\lambda_1 + 7\lambda_2 \leq 2 \\
& \lambda_1, \lambda_2 \geq 0
\end{aligned}$$

下面解释线性规划对偶问题中的记忆规则:

Q1: 为何原约束  $3x_1 + 4x_2 \geq 5$  时, 对偶问题中  $y_1 \geq 0$ ?

A1: 在计算 lagrange 函数时, 需要找到原目标函数的一个下界, 只有当  $\lambda_1$  和约束条件同号时才有  $f(x_1, x_2) \geq L(x_1, x_2, \lambda_1, \lambda_2)$ ;

Q2: 为何原约束  $x_1 \geq 0$  时, 对偶问题的约束  $3\lambda_1 + 6\lambda_2 \leq 1$ ?

A2: lagrange 函数  $L(x_1, x_2, \lambda_1, \lambda_2)$  中  $x_1$  的系数一定得是正数, 否则  $g(\lambda_1, \lambda_2) = -\infty$ ; 等式条件的约束也可类似推导, 不过约束条件会有一点变化。

2.lagrange 函数的最优解 (KKT 条件) 与对偶问题的最优值 (slater 条件), 只介绍, 不证明

弱对偶问题: 原问题的最小值与对偶问题的最大值 (所求的一个下界) 之间存在 *dualitygap*

强对偶问题: 原问题的最小值  $p^*$  与对偶问题的最大值 (所求的一个下界)  $d^*$  之间不存在 *dualitygap*, 即  $p^* = d^*$

slater 条件:

(1) 目标函数  $f(x)$  是凸函数;

(2) 存在一个  $x, s.t. f_i(x) < 0, i = 1, \dots, m$

当满足 slater 条件时, 该规划问题一定是强对偶问题 (线性规划或者二次规划均可), 由于线性规划一定满足 slater 条件, 所以线性规划是强对偶问题。

KKT 条件:

KKT 条件是关于 lagrange 函数最优解的条件。包括两条

(1) lagrange 条件:  $\nabla L(x^*, y^*, \lambda) = 0$ ;

(2) 互补松弛性:  $\lambda f_1(x^*, y^*) = 0$ .

二、二次规划的对偶问题 (结合支持向量机):

### Lagrange Multipliers

$$\begin{aligned} & \text{Minimize } f(x) \\ & \text{subject to } \begin{cases} a(x) \geq 0 \\ b(x) \leq 0 \\ c(x) = 0 \end{cases} \quad L(x, \alpha) = f(x) - \alpha_1 a(x) - \alpha_2 b(x) - \alpha_3 c(x) \\ & \begin{cases} \alpha_1 \geq 0 \\ \alpha_2 \leq 0 \\ \alpha_3 \text{ is unconstrained} \end{cases} \\ & \text{We can recover the primal problem by maximizing the Lagrangian with respect to the Lagrange multipliers} \\ & \max_{\alpha} L(x, \alpha) = \begin{cases} f(x), & \text{if } \begin{cases} a(x) \geq 0 \\ b(x) \leq 0 \\ c(x) = 0 \end{cases} \\ +\infty, & \text{otherwise} \end{cases} \\ & \text{So, the Primal problem can be changed into Dual problem} \\ & \min_x \max_{\alpha} L(x, \alpha) = \max_{\alpha} \min_x L(x, \alpha) \end{aligned}$$

### Karush-Kuhn-Tucker conditions

- For a local minimum

$$\begin{aligned} & \text{Stationarity } \nabla f(x^*) - \alpha_1 \nabla a(x^*) - \alpha_2 \nabla b(x^*) - \alpha_3 \nabla c(x^*) = 0 \\ & \text{Primal feasibility } \begin{cases} a(x^*) \geq 0 \\ b(x^*) \leq 0 \\ c(x^*) = 0 \end{cases} \\ & \text{Dual feasibility } \begin{cases} \alpha_1 \geq 0 \\ \alpha_2 \leq 0 \\ \alpha_3 \text{ is unconstrained} \end{cases} \\ & \text{Complementary slackness } \begin{cases} \alpha_1 a(x^*) = 0 \\ \alpha_2 b(x^*) = 0 \\ \alpha_3 c(x^*) = 0 \end{cases} \end{aligned}$$

(a) 二次规划的 lagrange 函数

(b) KKT 条件

图 33: 二次规划的 lagrange 函数及 KKT 条件

Hard-Margin SVM 对偶问题:

### Lagrange Transformation

$$\begin{aligned} & \text{Minimize } \frac{1}{2} \|\mathbf{w}\|^2 \\ & \text{subject to } 1 - y_i(\mathbf{w}^T \mathbf{x}_i + b) \leq 0 \quad \text{for } i = 1, \dots, n \end{aligned}$$

- The Lagrangian is

$$\mathcal{L} = \frac{1}{2} \mathbf{w}^T \mathbf{w} + \sum_{i=1}^n \alpha_i (1 - y_i(\mathbf{w}^T \mathbf{x}_i + b))$$

- Setting the gradient of  $\mathcal{L}$  w.r.t.  $\mathbf{w}$  and  $b$  to zero, we have

$$\mathbf{w} + \sum_{i=1}^n \alpha_i (-y_i) \mathbf{x}_i = \mathbf{0} \Rightarrow \mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$$

$$\sum_{i=1}^n \alpha_i y_i = 0 \quad \alpha_i \geq 0$$

### Dual Problem

- We can transform the problem to its dual

$$\begin{aligned} & \max_{\alpha} W(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \\ & \text{subject to } \alpha_i \geq 0, \sum_{i=1}^n \alpha_i y_i = 0 \end{aligned}$$

Dot product of X

$\alpha$ 's → New variables (Lagrangian multipliers)

- This is a convex quadratic programming (QP) problem
  - Global maximum of  $\alpha_i$  can always be found
  - well established tools for solving this optimization problem

- Note:  $\mathbf{w} = \sum_{i=1}^n \alpha_i y_i \mathbf{x}_i$

(a) 1

(b) 2

图 34: Hard-Margin SVM 对偶问题

Soft-Margin SVM 对偶问题:

### Question 6

决策树基本流程为: 训练集建立决策树 (ID3, C4.5), 验证集用来剪枝, 测试集测试精度。  
决策树有如下几个特点:

## Soft-Margin Case

- Lagrangian Problem

$$L = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i - \sum_{i=1}^n \alpha_i (y_i(\mathbf{w}^T \mathbf{x}_i + b) - 1 + \xi_i) - \sum_{i=1}^n \gamma_i \xi_i$$

$\alpha_i \geq 0 \quad \gamma_i \geq 0$

$$\frac{\partial L}{\partial \mathbf{w}} = 0 \Rightarrow \mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$$

$$\frac{\partial L}{\partial b} = 0 \Rightarrow \sum_i \alpha_i y_i = 0$$

$$\frac{\partial L}{\partial \xi_i} = 0 \Rightarrow \alpha_i + \gamma_i = C \Rightarrow 0 \leq \alpha_i \leq C$$

## Dual Problem for Soft-Margin SVM

- The dual of the problem is

$$\max. \quad W(\alpha) = \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1, j=1}^n \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j$$

$$\text{subject to } C \geq \alpha_i \geq 0 \quad \sum_{i=1}^n \alpha_i y_i = 0$$

- $\mathbf{w}$  is also recovered as  $\mathbf{w} = \sum_{j=1}^s \alpha_j y_j \mathbf{x}_j$

- The only difference with the linear separable case is that there is an upper bound  $C$  on  $\alpha_i$

- Once again, a QP solver can be used to find  $\alpha_i$  efficiently!!!

52

(a) 1

53

(b) 2

图 35: Soft-Margin SVM 对偶问题

- (1) 决策树的分类过程是可解释的，深度学习不可解释；
- (2) 对离散变量（离散特征，比如高收入、中收入和低收入特征）有很好的解决机制，深度学习和支持向量机主要是解决连续的特征（比如收入是 7000, 8000, 10000 等）；
- (3) 决策树容易过拟合，对噪声很敏感，泛化能力差。

### 一、建立决策树

建立决策树的过程最重要的是确定特征的选择顺序，引入信息论中的信息增益 (ID3) 和信息增益率 (C4.5) 来确定特征的选择顺序。

#### 1.ID3:

特征  $A$  对训练数据集  $D$  的信息增益  $g(D, A)$ ，定义为集合的经验熵  $H(D)$  与特征  $A$  给定条件下  $D$  的经验条件熵  $H(D|A)$  之差，即  $g(D|A) = H(D) - H(D|A)$ ，信息增益准则的特征选择方法是：对训练数据集（或者子集） $D$ ，计算其每个特征的信息增益，并比较他们的大小，选择信息增益最大的特征。

缺点：信息增益偏向于有大量值的特征（如学号、姓名，每个值都是一类），当特征是连续变量时，ID3 每次都会选择这些连续变量，不好处理，需要引入可以处理连续变量的方法；

#### 2.C4.5:

在信息增益的基础上除以一项分裂系数，得到信息增益率。特征值越连续，分裂系数越大，对于连续变量有惩罚的作用。

#### 3.CART(分类与回归树):

- (1)CART 规定决策树必须是二叉树，可以舍弃信息增益率的概念；
- (2) 将熵换成基尼系数；

### 二、剪枝

剪枝就是减少规则（用验证集中的数据来验证各个规则，看是否可靠）。剪枝是减少“弱枝”，弱枝值得是在验证集上误分类率高的数枝。

特点：剪枝会增加训练数据集上的错误分类率，但精简的树会提高测试集上的预测能力。缺点是在剪枝的时候不知道剪到什么程度合适，需要引入新的决策模型（随机森林）

### 三、随机森林

随机森林是集成学习的一种方法，基于 Bagging。决策树的一个缺点就是容易过拟合，随机森林是由一群弱的决策树构成的一片森林。

为了解决过拟合的问题，随机森林在建立分类器的过程中有两次随机性（样本随机和特征随机）：

- (1) 从样本集中用 bagging 采样选出  $n$  个样本（样本随机），建立 CART；
  - (2) 在树的每个节点上，从所有的特征中随机选择  $k$  个特征（特征随机），选择出一个最佳的分割属性作为节点（RI：从特征的子集当中选择最优；RC：将特征的子集线性组合形成新的特征）；
  - (3) 重复以上两步  $m$  次，构建  $m$  颗 CART（不剪枝，建立的 CART 树由于随机样本和随机特征，具有很弱的分类能力）；
  - (4) 这  $m$  颗树形成 Random Forest；
  - (5) 建立好随机森林后，输入一个样本，每个决策树都要输出一个决策分类，对分类结果做投票得到最终的分类结果。
- 注意：
- (1) 在样本随机时，采用的是 0.632 自助法，每次只选择 63% 的样本数作为样本子集；
  - (2) bagging 是采用重采样样本的方法，boosting 是采用重加权样本或分类器的方法。

## Question 7

### 特征提取、特征降维以及特征选择

“模式识别”在早期的含义实际上特指特征工程，即如何提取、降维以及选择数据中的特征，然后再使用经典的机器学习方法进行分类或者回归。现在这两个过程被深度学习方法统一起来。

深度学习实际上是一种自动提取特征的方法，卷积神经网络中的卷积-池化层会把物体（比如要识别的猫，狗）弄到图像的中间位置，相当于图像的配准过程，所以深度学习也叫表示学习。深度学习降低了计算机视觉各个子方向的门槛，转方向的时候（比如从文字识别到虹膜识别）不需要该领域特别多的基础知识，只要会深度学习框架就能训练。

深度学习提取特征虽然强大，但也有缺点：

- (1) 小样本学习：样本个数很少，不适合使用深度学习训练，此时可以使用手工设计特征；
- (2) 连续性学习：在给定的样本上和类别上可以训练的精度很高，当增加一些类别和样本时，深度学习需要重新训练来学习特征，手工设计特征不需要。

#### 一、特征提取

经典的特征提取方法有四种：

1.SIFT( Scale Invariant Feature Transform)，尺度不变性（实际上具有平移、尺度以及旋转不变性）

(1)DOG 首先使用不同带宽的高斯函数进行滤波，相邻带宽的滤波结果作差，得到一系列的 DOG；

(2) 取极值点（极大或者极小）；

(3) 计算极值点的梯度方向（点 + 方向）；

(4) 计算极值点周围的点的方向，每个像素点变成一个 128 维的向量，但是极值点个数不一样。

#### 2.Haar 特征

3.HOG 特征

4.LBP 特征 (Local Binary Pattern)

#### 二、特征降维（维数消减）

特征降维与特征选择不一样（虽然都可以起到维数消减的作用），特征降维是将  $n$  维的样本降到  $d$  ( $d < n$ ) 维空间，再降维过程中  $n$  维特征都用到了，而特征选择是直接选择其中的  $d$  维特征，其他维度的特征直接舍弃。

维数消减方法分为线性降维和非线性降维。

## 1. 线性降维

### PCA(Principal Components Analysis, 主成分分析)

主成分分析是一种无监督降维方法，实际上是在寻找方差最大的维度作为主成分，推导过程以及算法流程如下图 (第一幅图第四行应该是  $u_1, \dots, u_k$ ):

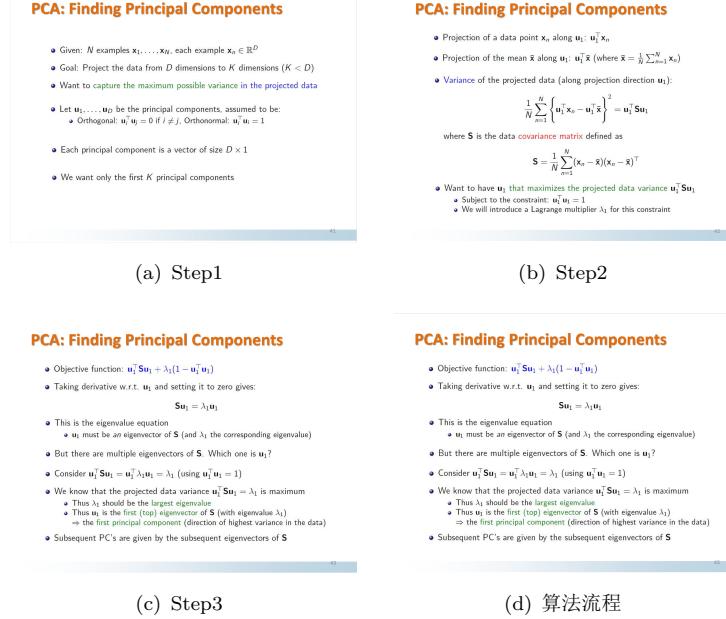


图 36: 主成分分析推导过程以及算法流程

主成分分析可以用来压缩数据，压缩过程为  $\hat{\mathbf{x}} = \sum_{i=1}^K (x_i^\top \mathbf{u}_i) \mathbf{u}_i = \sum_{i=1}^K z_i \mathbf{u}_i$ ，著名的应用有 Eigen-faces。

当样本数  $N$  很小，而特征维数  $D$  很大时 ( $N < D$ )，传统 PCA 不再适用，因为要求的特征值和特征向量的矩阵规模 ( $D * D$ ) 很大，此时可以求解较小规模矩阵的特征向量和特征值 (对应关系如下图):

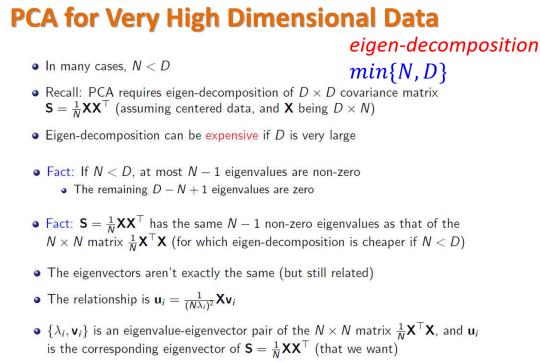


图 37: 样本数小于特征维数时的 PCA

PCA 还有一种另一种优化形式如下：  
观察其形式，PCA 与 AutoEncoder 非常类似。实际上：

### Optimality Property of PCA

Main theoretical result:

The matrix  $G$  consisting of the first  $p$  eigenvectors of the covariance matrix  $S$  solves the following min problem:

$$\min_{G \in \mathbb{R}^{d \times p}} \|X - G(G^T X)\|_F^2 \text{ subject to } G^T G = I_p$$

$$\|X - \bar{X}\|_F^2 \quad \text{reconstruction error}$$

*PCA projection minimizes the reconstruction error among all linear projections of size  $p$ .*

50

图 38: PCA 的另一种优化形式

- (1) RBM 和 AutoEncoder 是非线性的 PCA，都可以做到维数消减；
- (2) RBM 和 AutoEncoder 可以连接很多层进行训练，PCA 不可以，PCA 是线性的，多次嵌套还是相当于一次 PCA。

CCA(Canonical Correlation Analysis，典型相关性分析)  
两组变量的相关性，经济学当中使用较多，不适合解决分类问题。

### LDA(Linear Discriminant Analysis，线性判别分析)

LDA 是一种有监督的降维方法，主要解决分类问题。从最有利于分类的角度来降维：

- (1) 先假设每个类别都服从高斯分布，再进行白化(转换为标准高斯分布)，得到每个类别的均值；
- (2) 对  $c_1, c_2, \dots, c_k$   $k$  个类别均值  $\mu_1, \mu_2, \dots, \mu_k$  进行 PCA。

缺点：

- (1) 每一个类是单模态高斯分布 | 多模态 LDA；
- (2) 每一个类的协方差矩阵都相同 | 异方差 LDA；
- (3) 降维维数不能超过  $C-1$ 。

ICA(Independent Component Analysis，独立成分分析)  
ICA 用于去除冗余，和 PCA 不同，ICA 追求的是输出的变量相互独立，而非仅不相关，因此，ICA 需要利用数据分布的高阶统计信息而非仅二阶信息。

## 2. 非线性降维

### Kernel PCA

在 PCA 的基础上引入核方法，PCA 是在给定的样本下算出样本的协方差矩阵，然后计算该矩阵的较大特征值对应的特征向量作为主成分，便于解决线性数据的降维问题。

Kernel PCA 主要解决非线性降维问题，对于给定的非线性数据，首先引入映射  $\phi(x)$  变为线性关系，在空间  $\phi$  中进行 PCA 降维，得到主成分  $v_i$ 。

对于给定的样本，先映射到  $\phi$  空间进行投影计算各主成分  $v_i$  的系数，由于  $\phi$  空间比较复杂，主成分  $v_i$  不容易计算。所以引入核方法，可以用核矩阵的主成分  $a_i$  来代替计算  $v_i$ 。

第四步里面要计算降维后的样本的各个分量  $z_i = \phi^T v_i$ ，先将样本变换到高维空间，再和相应维度的主成分做内积，但是  $\phi$  和  $v_i$  都不知道，所以可以使用核矩阵的特征向量  $a_i$  和以及已知的核函数  $k(x_i, x_j)$

代替求解。推导过程如下：

### Kernel PCA

- Given  $N$  observations  $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$ ,  $\forall \mathbf{x}_i \in \mathbb{R}^D$ , define the  $D \times D$  covariance matrix (assuming centered data  $\sum_i \mathbf{x}_i = 0$ )

$$\mathbf{S} = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_i \mathbf{x}_i^\top$$

- Linear PCA:** Compute eigenvectors  $\mathbf{u}$ , satisfying:  $\mathbf{S}\mathbf{u} = \lambda_i \mathbf{u}$ ,  $\forall i = 1, \dots, D$
- Consider a nonlinear transformation  $\phi(\mathbf{x})$  from an  $M$  dimensional space
- $M \times M$  covariance matrix in this space (assume centered data  $\sum_i \phi(\mathbf{x}_i) = 0$ )
- $\mathbf{C} = \frac{1}{N} \sum_{i=1}^N \phi(\mathbf{x}_i) \phi(\mathbf{x}_i)^\top$
- Kernel PCA:** Compute eigenvectors  $\mathbf{v}$ , satisfying:  $\mathbf{C}\mathbf{v} = \lambda_i \mathbf{v}$ ,  $\forall i = 1, \dots, M$
- Ideally, we would like to do this without having to compute the  $\phi(\mathbf{x}_i)$ 's

### Kernel PCA

- Kernel PCA:** Compute eigenvectors  $\mathbf{v}$ , satisfying:  $\mathbf{C}\mathbf{v} = \lambda_i \mathbf{v}$
- Plugging in the expression for  $\mathbf{C}$ , we have the eigenvector equation:

$$\frac{1}{N} \sum_{i=1}^N \phi(\mathbf{x}_i) \phi(\mathbf{x}_i)^\top \mathbf{v} = \lambda_i \mathbf{v}$$

- Using the above, we can write  $\mathbf{v}$ , as  $\mathbf{v}_i = \sum_{m=1}^M a_{im} \phi(\mathbf{x}_m)$

- Plugging this back in the eigenvector equation:

$$\frac{1}{N} \sum_{i=1}^N \phi(\mathbf{x}_i) \phi(\mathbf{x}_i)^\top \sum_{m=1}^M a_{im} \phi(\mathbf{x}_m) = \lambda_i \sum_{m=1}^M a_{im} \phi(\mathbf{x}_m)$$

- Pre-multiplying both sides by  $\phi(\mathbf{x}_j)^\top$  and re-arranging

$$\frac{1}{N} \sum_{i=1}^N \phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j) \sum_{m=1}^M a_{im} \phi(\mathbf{x}_m)^\top \phi(\mathbf{x}_m) = \lambda_i \sum_{m=1}^M a_{im} \phi(\mathbf{x}_m)^\top \phi(\mathbf{x}_j)$$

(a) Step1

(b) Step2

### Kernel PCA

- Using  $\phi(\mathbf{x}_i)^\top \phi(\mathbf{x}_j) = k(\mathbf{x}_i, \mathbf{x}_j)$ , the eigenvector equation becomes:

$$\frac{1}{N} \sum_{i=1}^N \sum_{j=1}^N k(\mathbf{x}_i, \mathbf{x}_j) \sum_{m=1}^M a_{im} a_{jm} k(\mathbf{x}_m, \mathbf{x}_i) = \lambda_i \sum_{m=1}^M a_{im} a_{im} k(\mathbf{x}_i, \mathbf{x}_i)$$

- Define  $\mathbf{K}$  as the  $N \times N$  kernel matrix with  $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$
- $\mathbf{K}$  is the similarity of two examples  $\mathbf{x}_i$  and  $\mathbf{x}_j$  in the  $\phi$  space
- $\phi$  is implicitly defined by kernel function  $k$  (which can be, e.g., RBF kernel)

- Define  $\mathbf{a}$  as the  $N \times 1$  vector with elements  $a_{im}$

- Using  $\mathbf{K}$  and  $\mathbf{a}$ , the eigenvector equation becomes:

$$\mathbf{K}\mathbf{a} = \lambda_i \mathbf{a} \Rightarrow \boxed{\mathbf{K}\mathbf{a} = \lambda_i \mathbf{N}\mathbf{a}}$$

- This corresponds to the original Kernel PCA eigenvalue problem  $\mathbf{C}\mathbf{v} = \lambda_i \mathbf{v}$ .

- For a projection to  $K < D$  dimensions, top  $K$  eigenvectors of  $\mathbf{K}$  are used

$$\boxed{\mathbf{v}_i = \sum_{m=1}^M a_{im} \phi(\mathbf{x}_m)}$$

(c) Step3

### Kernel PCA: The Projection

- Suppose  $\{\mathbf{a}_1, \dots, \mathbf{a}_K\}$  are the top  $K$  eigenvectors of kernel matrix  $\mathbf{K}$

- The  $K$ -dimensional KPCA projection  $\mathbf{z} = [z_1, \dots, z_K]$  of a point  $\mathbf{x}$ :

$$z_i = \phi(\mathbf{x})^\top \mathbf{a}_i$$

- Recall the definition of  $\mathbf{v}_i$ :

$$\boxed{\mathbf{v}_i = \sum_{m=1}^N a_{im} \phi(\mathbf{x}_m)}$$

- Thus

$$\boxed{z_i = \phi(\mathbf{x})^\top \mathbf{v}_i = \sum_{m=1}^N a_{im} \phi(\mathbf{x}_m)^\top \phi(\mathbf{x})}$$

(d) 算法流程

图 39: Kernel 主成分分析推导过程以及算法流程

### 流形学习 (Manifold Learning)

#### (1)LLE(Locally Linear Embedding , 局部线性嵌入)

- 在高维空间中, 用样本的邻居来重构该样本, 即求解一个 QP 问题;
- 求解得系数之后, 利用该系数来拟合低维表示。

优点: 能保持近邻关系, 即保持流形。

算法与效果如下:

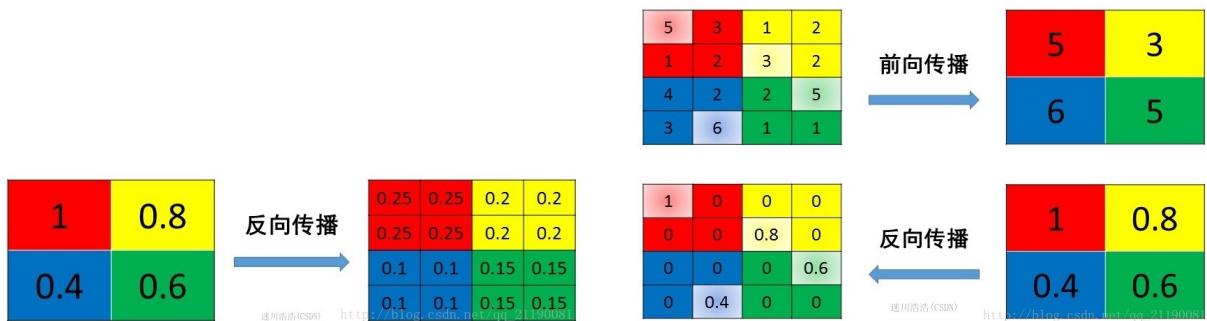


图 40: 算法流程以及效果图

#### (2)ISOMAP(Isometric Feature Mapping , 等距特征映射)

计算高维空间中两个点的测地距离, 降维的时候保持距离不变。

## Question 1

**手推 BP(Back Propagation) 算法。**

先设出以下变量 (只考虑一个样本的三层全连接神经网络):

- (1) 训练数据输入输出对:  $\{x = (x_1, x_2, \dots, x_i, \dots, x_d), t = (t_1, t_2, \dots, t_j, \dots, t_c)\}$ , 样本已经包括偏移量;
- (2) 输出层节点的加权输入及输出:  $\{net = (net_1, net_2, \dots, net_j, \dots, net_c), z = (z_1, z_2, \dots, z_j, \dots, z_c)\}$ ;
- (3) 隐藏层节点的加权输入及输出:  $\{neth = (neth_1, neth_2, \dots, neth_h, \dots, neth_{n_H}), y = (y_1, y_2, \dots, z_h, \dots, y_{n_H})\}$ ;
- (4) 输入层节点  $i$  到隐藏层节点  $h$  的权重:  $W_{ih}$ ;
- (5) 隐藏层节点  $h$  到输出层节点  $j$  的权重:  $W_{hj}$ ;
- (6) 损失函数使用  $MSE, J(W) = \frac{1}{2} \sum_{j=1}^c (t_j - z_j)^2$ .

则:

- (1) 隐藏层节点  $h$  的输入加权和:  $neth_h = \sum_{i=1}^d W_{ih}x_i$ ;
- (2) 经过激励函数, 隐藏层节点  $h$  的输出:  $y_h = f_1(neth_h) = f_1(\sum_{i=1}^d W_{ih}x_i), f_1(x) = \frac{1}{1+e^{-x}}$ ;
- (3) 输出层节点  $i$  的输入加权和:  $net_j = \sum_{h=1}^{n_H} W_{hj}y_h = \sum_{h=1}^{n_H} W_{hj}f_1(\sum_{i=1}^d W_{ih}x_i)$ ;
- (4) 经过激励, 输出层节点  $j$  的输出:  $z_j = f(net_j) = f_2(\sum_{h=1}^{n_H} W_{hj}y_h) = f_2(\sum_{h=1}^{n_H} W_{hj}f_1(\sum_{i=1}^d W_{ih}x_i)), f_2(x_i) = \frac{e^{x_i}}{\sum_{i=1}^n e^{x_i}}$ .

1. 先确定隐藏层节点  $h$  到输出层节点  $j$  的连接权重调节量:

$$\begin{aligned}\Delta W_{hj} &= -\eta \frac{\partial J}{\partial W_{hj}} \\ &= -\eta \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial W_{hj}} \\ &= -\eta \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial net_j} \frac{\partial net_j}{\partial W_{hj}} \\ &= \eta(t_j - z_j)f'_2(net_j)y_h \\ &= \eta\delta_j y_h\end{aligned}$$

其中  $\delta_j = -\frac{\partial J}{\partial net_j} = f'_2(net_j)(t_j - z_j) = (t_j - z_j) \frac{e^{net_j} (\sum_{k \neq j} e^{net_k})}{\sum_{j=1}^c e^{net_j}} = (t_j - z_j)f_2(net_j)(1 - f_2(net_j))$

2. 再确定输入层节点  $i$  到隐藏层节点  $h$  的连接权重调节量:

$$\begin{aligned}
\Delta W_{ih} &= -\eta \frac{\partial J}{\partial W_{ih}} \\
&= -\eta \sum_{j=1}^c \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial W_{ih}} \\
&= -\eta \sum_{j=1}^c \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial net_j} \frac{\partial net_j}{\partial W_{ih}} \\
&= -\eta \sum_{j=1}^c \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial net_j} \frac{\partial net_j}{\partial y_h} \frac{\partial y_h}{\partial W_{ih}} \\
&= -\eta \sum_{j=1}^c \frac{\partial J}{\partial z_j} \frac{\partial z_j}{\partial net_j} \frac{\partial net_j}{\partial y_h} \frac{\partial y_h}{\partial neth_h} \frac{\partial neth_h}{\partial W_{ih}} \\
&= \eta \sum_{j=1}^c (t_j - z_j) f'_2(net_j) W_{hj} f'_1(neth_h) x_i \\
&= \eta \delta_h x_i
\end{aligned}$$

其中  $\delta_h = -\frac{\partial J}{\partial neth_h} = \sum_{j=1}^c (t_j - z_j) f'_2(net_j) W_{hj} f'_1(neth_h) = f'_1(neth_h) \sum_{j=1}^c W_{hj} \delta_j$

总结：相邻层之间连接节点  $a$  到  $b$  的权重调节量由两部分决定，一是边起始对应的节点的输出（激励之后），二是边指向对应的节点收集到的误差（即损失函数对该点加权和的导数的相反数，此导数是经过激励函数的导数放缩过的）。前一层所收集到的误差等于后一层收集到的误差的加权求和再缩放一个前一层激励函数的导数。

## Question 2

### 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， $E_{new}(w) = E(w) + kw^T w$ ，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；

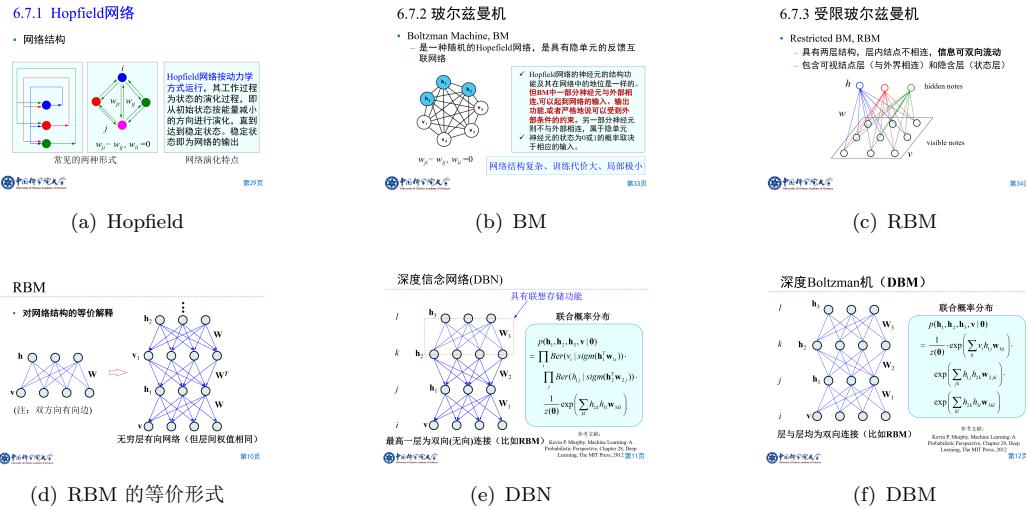


图 41: Hopfield, BM, RBM, DBN, DBM

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，当池化模板是  $2 * 2$  时，就是把 1 个像素的梯度传递给 4 个像素，但是需要保证传递的 loss(梯度) 总和不变。根据这条原则，mean pooling 和 max pooling 的反向传播也是不同的 ([https://blog.csdn.net/Jason\\_yyz/article/details/80003271](https://blog.csdn.net/Jason_yyz/article/details/80003271))。

1. 平均池化：mean pooling 的前向传播就是把一个模板中的值求取平均来做 pooling，那么反向传播的过程也就是把某个元素的梯度等分为 n 份分配给前一层，这样就保证池化前后的梯度之和保持不变；
2. 最大池化：max pooling 也要满足梯度之和不变的原则，max pooling 的前向传播是把模板中最大的值传递给后一层，而其他像素的值直接被舍弃掉。那么反向传播也就是把梯度直接传给前一层某一个像素，而其他像素不接受梯度，也就是为 0。所以 max pooling 操作和 mean pooling 操作不同点在于需要记录下池化操作时到底哪个像素的值是最大，也就是 max id，这个变量就是记录最大值所在位置的，因为在反向传播中要用到。

两种池化方式反向传播的过程见下图：

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。

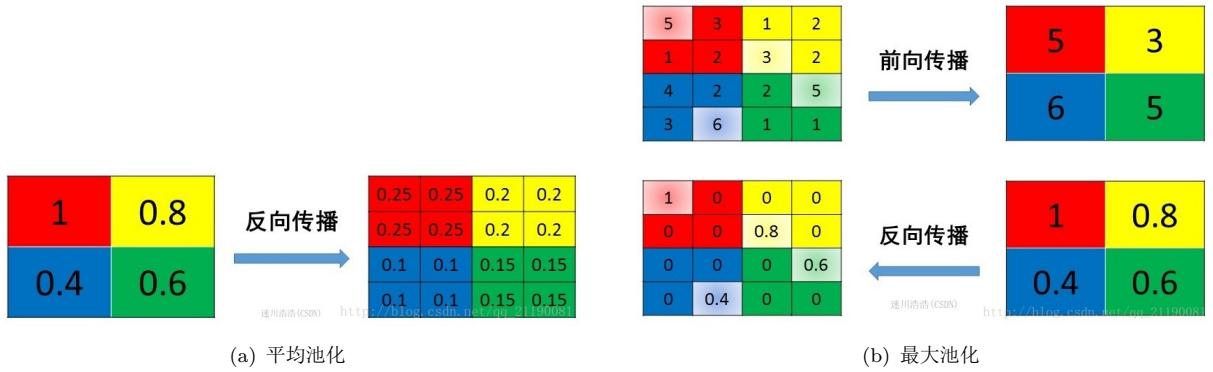


图 42: 池化层反向传播

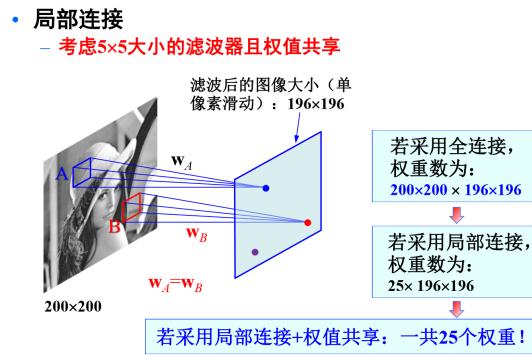


图 43: 参数情况

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。

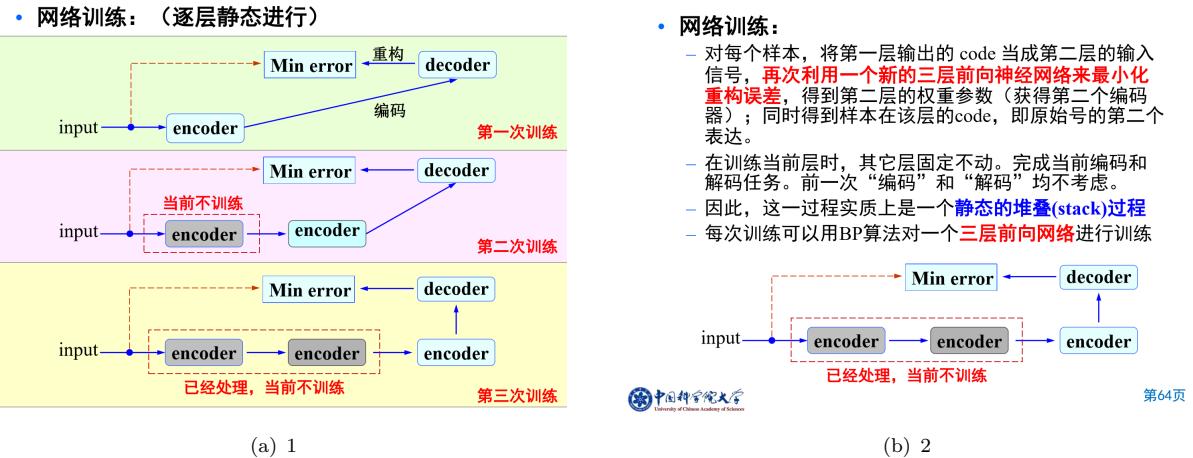


图 44: AutoEncoder 训练过程

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。

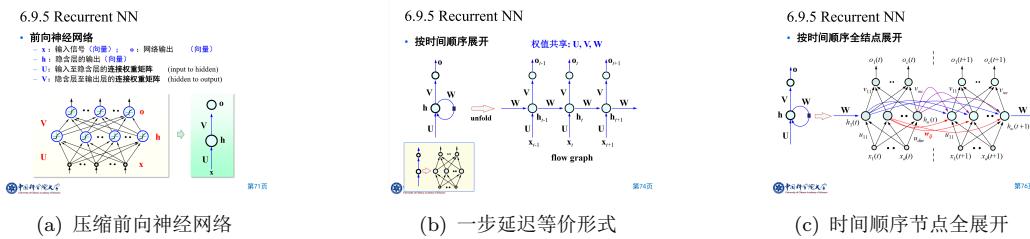


图 45: RNN

*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  $x_t$ ，如何得到这一时刻的输出（输出到下一时刻，不是输出网络） $h_t$ ？

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  $U, W, U_f, W_f, U_{in}, W_{in}, U_o, W_o$ 。

### 6.9.6 LSTM

#### • 结构描述——采用矩阵形式

遗忘门、输入门、输出门:

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

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

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

候选记忆 (新贡献部分):

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

Cell产生的新记忆:

$$\mathbf{s}_t = \mathbf{f}_t \otimes \mathbf{s}_{t-1} + \mathbf{i}_t \otimes \mathbf{c}_t$$

Cell的输出:

$$\mathbf{h}_t = \mathbf{o}_t \otimes \tanh(\mathbf{s}_t)$$

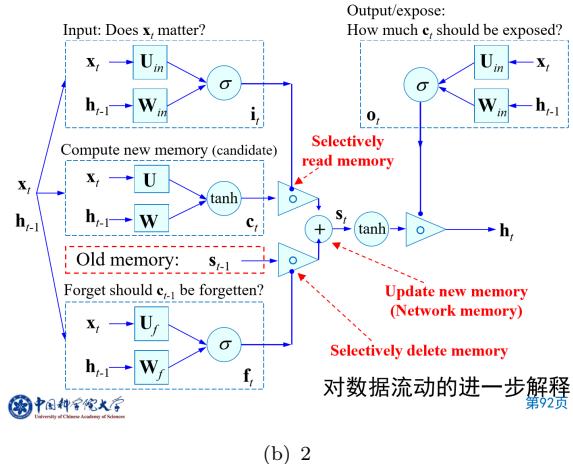
网络的输出:

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



第91页

(a) 1



(b) 2

图 46: LSTM

## Part IV

# 面试经历

### 一、陌陌计算机视觉实习生岗位:

技术一面问题:

1. tensorflow 有什么特点? 手写 tensorflow 代码, 搭建卷积神经网络。
2. 写出神经网络中的 Batch Normalization 的公式, 在 tensorflow 中有哪些方式? (哪些参数)。
3. 神经网络的 BP 算法, 误差传到池化层的时候该怎么传? (均值池化和最大池化)
4. 介绍 ResNet 和 GoogleNet 的核心思想?
5. 图像边缘检测有哪些传统方法? 如果用深度学习工具来分割图像该怎么设计机制? (比如 Unet 网络)
6. Unet 网络模型先下采样, 再上采样, 在下采样的过程是准确的, 但是上采样的过程会使得图像变模糊, Unet 中有什么机制可以改善这一效果? (同级校对)
7. 数据结构: 二叉树的中序遍历 (递归与非递归), 输出数组滑窗中的最大值。如 [3,5,-1,3,2,-4,5], 窗口大小为 3, 输出 [5,5,3,3,5], 写完  $O(n)$  复杂度的之后要求优化 (使用栈等数据结构)
8. 在 SLIC 超像素分割算法 (KMeans 聚类算法) 中有什么方法可以自适应地调整 k 值?
9. 介绍图像的双线性插值。

答案参考 (后续整理):

1.tensorflow 是静态图机制, Pytorch 是动态图机制。

静态图是指在图构建完成后, 在模型运行时无法进行修改。这里的“图”即为模型的意思, 一般一个图就是一个模型。这个图建好之后, 运行前需要 freeze, 然后进行一些优化, 例如图融合等, 最后启动 session 根据之前构建的静态图进行计算, 计算过程无法对静态图进行更改。

动态图和静态图对应, 在模型运行过程中可以对图进行修改。熟悉 PyTorch 的朋友应该了解, 因为 PyTorch 采用的就是动态图机制。不过一般情况, 模型运行过程中也不需要对其进行修改。

TensorFlow 静态图带来的一个弊端就是难 Debug。因为静态图在 freeze 后运行前会进行图优化, 一些 operation 会被融合, 所以有的 operation 会消失, 无法在计算阶段提供断点和单步调试功能 (图构建阶段能单步调试, 但是只显示 tensor, 用处不大)。当然, 断点和单步调试功能也会影响程序运行的效率。而 PyTorch 采用了动态图, 自然有 Debug 方面的优势, 提供了单步调试的功能, 可以在计算过程查看所有 tensor 的值, 非常直观方便。<https://blog.csdn.net/guanxs/article/details/90523905>

2.Batch Normalization 方法为了使各层有适当的广度，强制性地调整激活值的分布，有以下优点：  
(1) 可以使学习快速进行 (可以增大学习率); 如果每层的 scale 不一致，实际上每层需要的学习率是不一样的，同一层不同维度的 scale 往往也需要不同大小的学习率，通常需要使用最小的那个学习率才能保证损失函数有效下降，Batch Normalization 将每层、每维的 scale 保持一致，那么我们就可以直接使用较高的学习率进行优化。(<https://blog.csdn.net/u014365862/article/details/77159778>)

- (2) 不那么依赖初始值;
- (3) 抑制过拟合 (降低 Dropout);

Batch Norm 是使得数据分布的均值为 0，方差为 1 的正规化。写成公式如下：

$$\begin{aligned}\mu_B &= \frac{1}{m} \sum_{i=1}^m x_i \\ \sigma_B^2 &= \sum_{i=1}^m \frac{1}{m} (x_i - \mu_B)^2 \\ \hat{x}_i &= \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}\end{aligned}$$

正规化之后进行缩放和平移，写成公式如下：

$$y_i = \gamma \hat{x}_i + \beta$$

一开始  $\gamma = 1$ ,  $\beta = 0$ , 然后通过学习训练调整合适的值。

在 tensorflow 中：`tf.nn.batch_normalization(x, mean, var, beta, gamma, eps)`，参数的意义跟上面公式一样。

3. 第三部分第四题讨论过这个问题。

4.resNet 称为残差卷积神经网络。

resNet 有的一个核心思想就是跳跃连接，这样做有两个非常重要的好处：(1)BP 算法中梯度从后往前传，每多传一层就会放缩一次误差，跳跃连接可以缓解梯度消失的情况；(2)这种跳跃连接使得目标函数变成一个关于参数的凸函数，传统的神经网络关于参数是非凸的，这样会很难求解全局最小解。

5.6. 介绍 Unet 网络模型：

7. 二叉树的中序遍历 (递归和非递归)：

递归形式中，先序和后序只需更改核心三行的顺序即可；

非递归形式：递归本质上是在调用栈

```
1 #include<iostream>
2 #include<vector>
3 #include<stack>
4
5 using namespace std;
6
7 struct TreeNode{
8     int val;
9 }
```

```

9  TreeNode*left ;
10 TreeNode*right ;
11};

13 //递归遍历
14 vector<int> midOrder(TreeNode *T)
15{
16    vector<int> v;
17    if (T==NULL)
18    {
19        return ;
20    }
21    midOrder(T->left);
22    v.push_back(T->val); //此处改变顺序就得到先序和后序
23    midOrder(T->right);
24    return v;
25}

27 //非递归遍历，中序
28 vector<int> Midorder(TreeNode *T)
29{
30    vector<int> v;
31    stack<TreeNode*> s;
32    while(T||!s.empty())
33    {
34        while(T!=NULL)//如果当前的T不为空，则一直要到最左边
35        {
36            s.push(T);
37            T = T->left;
38        }
39        if (!s.empty())
40        {
41            T = s.top();
42            s.pop();
43            v.push_back(T->val);
44            T = T->right;
45        }
46    }
47    return v;
48}

49 //非递归遍历，先序
50 vector<int> Midorder(TreeNode *T)
51{
52    vector<int> v;
53    stack<TreeNode*> s;
54    while(T||!s.empty())
55    {
56        while(T!=NULL)//先输出当前的节点值，如果当前的T不为空，则一直要到最左边
57        {
58            v.push_back(T->val);
59            s.push(T);
60            T = T->left;
61        }
62        if (!s.empty())
63        {
64            T = s.top();
65            s.pop();
66            T = T->right;
67        }
68    }
69    return v;
70}

73 //非递归遍历，后序，只需将先序遍历的左右交换，同时将打印部分改为压入另一个栈，最后打印这个栈中的元素即

```

```

    得到后序
vector<int> Midorder(TreeNode *T)
75 {
    vector<int> v;
77     stack<TreeNode*> s;
    stack<TreeNode*> ss;
79     while(T || !s.empty())
{
81         while(T!=NULL)//先输出当前的节点值，如果当前的T不为空，则一直要到最左边
82         {
83             ss.push(T);//按照根-右-左进栈
84             s.push(T);
85             T = T->right;
86         }
87         if (!s.empty())
88         {
89             T = s.top();
90             s.pop();
91             T = T->left;
92         }
93     }
94     while (!ss.empty())
95     {
96         T = ss.top();
97         ss.pop();
98         v.push_back(T->val);
99     }
100 }
101 }
```

8. 在网上看到一种方法“手腕法”， $SSE = \sum_{i=1}^k \sum_{p \in C_i} d(p, p_i)$ ，其中， $C_i$  是第  $i$  个簇， $p$  是  $C_i$  中的样本点， $m_i$  是  $C_i$  的质心 ( $C_i$  中所有样本的均值)， $SSE$  是所有样本的聚类误差，代表了聚类效果的好坏。手肘法的核心思想是：随着聚类数  $k$  的增大，样本划分会更加精细，每个簇的聚合程度会逐渐提高，那么误差平方和  $SSE$  自然会逐渐变小。并且，当  $k$  小于真实聚类数时，由于  $k$  的增大会大幅增加每个簇的聚合程度，故  $SSE$  的下降幅度会很大，而当  $k$  到达真实聚类数时，再增加  $k$  所得到的聚合程度回报会迅速变小，所以  $SSE$  的下降幅度会骤减，然后随着  $k$  值的继续增大而趋于平缓，也就是说  $SSE$  和  $k$  的关系图是一个手肘的形状，而这个肘部对应的  $k$  值就是数据的真实聚类数。当然，这也是该方法被称为手肘法的原因。<https://blog.csdn.net/xyisv/article/details/82430107>

#### 9. 图像的双线性插值：

双线性插值本质上是在两个方向分别进行单线性插值 [https://blog.csdn.net/qq\\_37577735/article/details/80041586](https://blog.csdn.net/qq_37577735/article/details/80041586)。已知数据  $(x_1, y_1)$  与  $(x_2, y_2)$ ，要计算区间  $[x_1, x_2]$  内某一位置  $x$  在直线上的  $y$  值，则有：

$$\begin{aligned}
\frac{y - y_1}{x - x_1} &= \frac{y_2 - y}{x_2 - x} \\
(x_2 - x)(y - y_1) &= (y_2 - y)(x - x_1) \\
(x_2 - x)y - (x_2 - x)y_1 &= (x - x_1)y_2 - (x - x_1)y \\
(x_2 - x_1)y &= (x_2 - x)y_1 + (x - x_1)y_2 \\
y &= \frac{x_2 - x}{x_2 - x_1}y_1 + \frac{x - x_1}{x_2 - x_1}y_2
\end{aligned}$$

上面的公式仔细看就是用  $x$  和  $x_1, x_2$  的距离作为一个权重，用于  $y_1$  和  $y_2$  的加权，双线性本质上是在图像的两个方向上做线性插值，核心思想是在两个方向上分别进行一次线性插值。假设想得到未知函数  $f$  在点  $P = (x, y)$  的值，已知最近的四个点的坐标为  $Q_{11}(x_1, y_1), Q_{12}(x_1, y_2), Q_{21}(x_2, y_1), Q_{22}(x_2, y_2)$ ，

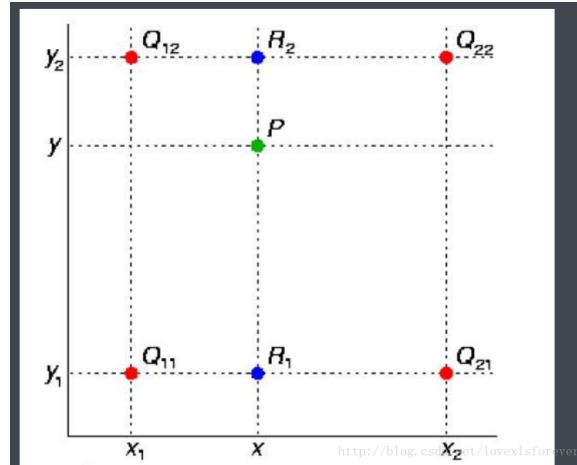


图 47: 双线性插值示意图

如下图: 首先进行  $x$  方向的插值, 得到:

$$f(R_1) = \frac{x_2 - x}{x_2 - x_1} f(Q_{11}) + \frac{x - x_1}{x_2 - x_1} f(Q_{21})$$

$$f(R_2) = \frac{x_2 - x}{x_2 - x_1} f(Q_{12}) + \frac{x - x_1}{x_2 - x_1} f(Q_{22})$$

然后在  $y$  方向进行线性插值, 得到:

$$f(P) = \frac{y_2 - y}{y_2 - y_1} f(R_1) + \frac{y - y_1}{y_2 - y_1} f(R_2)$$

由于图像的双线性插值只会用相邻的 4 个点, 因此上述公式的分母都是 1, 下面给出一个计算的代码:

```

1 import cv2
2 import numpy as np
3 import time
4 import os
5
6 def resize(src, new_size):
7     dst_w, dst_h = new_size # 目标图像宽高
8     src_h, src_w = src.shape[:2] # 源图像宽高
9     if src_h == dst_h and src_w == dst_w:
10         return src.copy()
11     scale_x = float(src_w) / dst_w # x缩放比例
12     scale_y = float(src_h) / dst_h # y缩放比例
13
14     # 遍历目标图像, 插值
15     dst = np.zeros((dst_h, dst_w, 3), dtype=np.uint8)
16     for n in range(3): # 对channel循环
17         for dst_y in range(dst_h): # 对height循环
18             for dst_x in range(dst_w): # 对width循环
19                 # 目标在源上的坐标
20                 src_x = (dst_x + 0.5) * scale_x - 0.5
21                 src_y = (dst_y + 0.5) * scale_y - 0.5
22                 # 计算在源图上四个近邻点的位置
23                 src_x_0 = int(np.floor(src_x))
24                 src_y_0 = int(np.floor(src_y))
25                 src_x_1 = min(src_x_0 + 1, src_w - 1)
26                 src_y_1 = min(src_y_0 + 1, src_h - 1)

```

```

27     # 双线性插值
29         value0 = (src_x_1 - src_x) * src[src_y_0, src_x_0, n] + (src_x - src_x_0) * src[src_y_0
30             , src_x_1, n]
31         value1 = (src_x_1 - src_x) * src[src_y_1, src_x_0, n] + (src_x - src_x_0) * src[src_y_1
32             , src_x_1, n]
33         dst[dst_y, dst_x, n] = int((src_y_1 - src_y) * value0 + (src_y - src_y_0) * value1)
34     return dst
35
36 if __name__ == '__main__':
37     img_in = cv2.imread('./test/3.jpg')
38     img_out = cv2.resize(img_in, (600,600))
39     cv2.imwrite('./.jpg', img_out)
40
41     time_start = time.time()
42     for img in os.listdir('./test'):
43         img_in = cv2.imread(os.path.join('./test',img))
44         img_out = cv2.resize(img_in, (600,600))
45         cv2.imwrite('./test_preprocess_chazhi', img_out)
46
47     time_end = time.time()
48     time_c = time_end - time_start
49     print('time_c:{} s'.format(time_c))

```

代码中最重要的是目标图像和原图像坐标的对齐，这一步是为了插值出来的目标图像和原图像有个几何中心的对齐，使得插值的结果更加均匀地分布在原图像上。假设原图尺寸为  $(m_1, n_1)$ ，插值图像尺寸为  $(m_2, n_2)$ ，则  $x, y$  方向上的放缩因子分别为  $scale_x = \frac{m_1}{m_2}$ ,  $scale_y = \frac{n_1}{n_2}$ 。如果以原点  $(0, 0)$  对齐：要插值的目标图像上的像素坐标为  $(dst_x, dst_y)$ ，对应到原图上像素坐标为  $(src_x, src_y) = (dst_x * scale_x, dst_y * scale_y)$ ，以这个坐标周围四个点来进行插值会导致插值使用的像素偏向于原图像的右下方或左上方，而不是均匀分布整个图像。如  $(3, 3)$  的图像插值为  $(5, 5)$ ， $x, y$  方向上的比例因子都是  $\frac{3}{5}$ ，在插值中心元素  $(2, 2)$  时，对应的原图像上是坐标  $(2 * \frac{3}{5}, 2 * \frac{3}{5}) = (1.667, 1.667)$ ，使用这个坐标周边四个像素值来插值会导致插值的图像像素集中在原图像的右下方；同理，如果插值的图像比原图小的话，会导致像素集中在左上方。所以这里采用了一个对齐公式：

$$src_x = dst_x * scale_x + \frac{1}{2}(scale_x - 1)$$

$$src_y = dst_y * scale_y + \frac{1}{2}(scale_y - 1)$$

用这个对齐公式去计算在原图上的像素位置，然后使用这个像素周围的四个像素的位置来插值新的图像。此外再注意插值公式的分母都为 1，即可理解上面这段代码。

### 技术二面问题：

1. 介绍线性子空间； 2. 介绍共轭梯度算法，联系梯度下降和牛顿法；
3. 卷积神经网络中一般都有哪些层？简要介绍 BN 层；
4. 介绍极大似然估计方法流程；
5. 操作系统（或计算机组成原理）中内存的 LRU 算法；
6. 学过哪些深度学习模型和机器学习算法；
7. 手写代码，将一个数组逆时针旋转 90°，要求不能开内存；（通过中间变量来赋值）