

AI 岗位基础面试问题

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

2019 年 11 月 17 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 $\|A\|_2 = \lambda$ (λ 是 A 的最大特征值)。

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

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

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

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

且有:

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

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

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

当 A 是对称正定阵时, 特征值均大于 0。 $A^T A$ 可以视为 $f(A)g(A)$, 其特征值的最大值为 λ_1^2 , λ_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 ∞ 范数, 计算结果可以用‘一列无穷行’记忆。

Question 2

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

解: {求解极大似然估计, 应该先写出极大似然函数 $\ln(L(\theta))$, 再对参数 θ 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 *Cout* 分解

2. 设矩阵 $A \in R^{n*n}$ 为对称正定阵, 则存在一个使得非奇异下三角矩阵 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。

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。

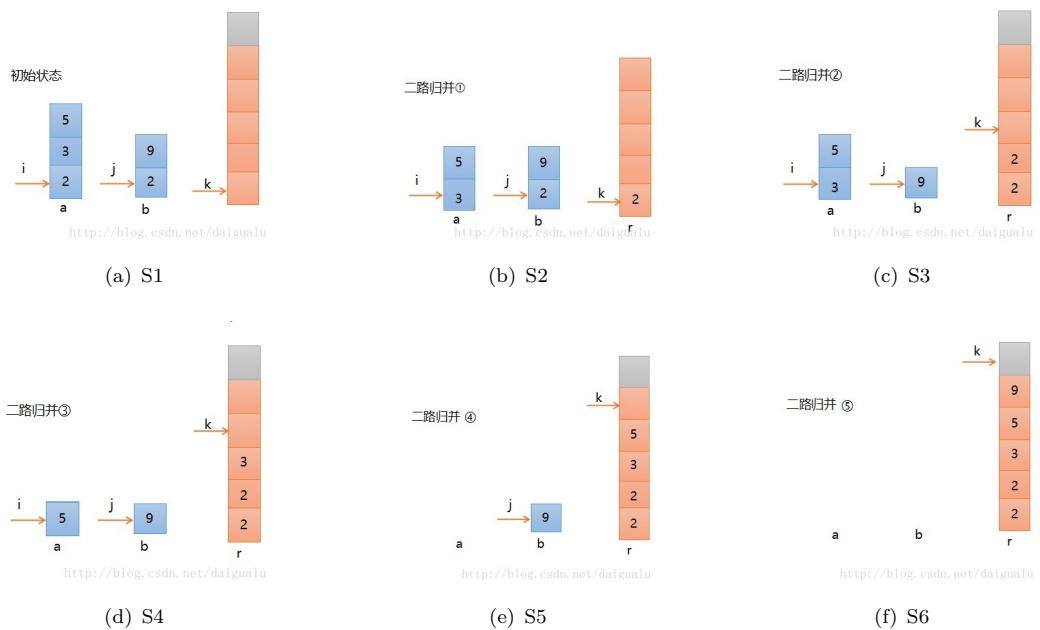


图 2: 二路归并过程

```

1 #include <iostream>
3 #include <stdio.h>
5 using namespace std;
7 long int merge(int a[], int left, int mid, int right,int b[])
{
9    int i = mid;
11   int j = right;
13   int k = 0;
15   while (i >= left && j >= mid+1)
17   {
18       if(a[i] > a[j])
19       {
20           b[k++] = a[i--];
21       }
22       else
23       {
24           b[k++] = a[j--];
25       }
26   }
27 }
```

```

23     }
24     while ( i >= left )
25     {
26         b [k++] = a [i--];
27     }
28     while ( j >= mid+1 )
29     {
30         b [k++] = a [j--];
31     }
32     for ( i = 0; i < k; i++ )
33     {
34         a [right - i] = b [i];
35     }
36 }

37 long int solve( int a[], int left , int right , int b[])
38 {
39     if( right > left )
40     {
41         int mid = (right+left) / 2;
42         solve(a, left , mid,b);
43         solve(a, mid + 1, right ,b);
44         merge(a, left , mid, right ,b);
45     }
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
59     solve(a,0,n-1,b); //归并排序
60
61     for( int i = 0;i<n; i++)
62         cout<<a [i]<<'\n';
63     return 0;
64 }
65 }
```

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 using namespace std;

4 long int merge( int a[], int left , int mid, int right , int b[])
5 {
6     int i = left ;
7     int j = mid+1;
8     int k = 0;
9     while ( i <= mid && j <= right )
10    {
11        if( a [i] > a [j])
12        {
13            b [k] = a [j];
14            j++;
15            k++;
16        }
17        else
18        {
19            b [k] = a [i];
20            i++;
21            k++;
22        }
23    }
24    for( int i = left ; i < k; i++)
25        a [i] = b [i];
26 }
```

```

16     b[k++] = a[i++];
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 }

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

46

47 int main()
48 {
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     {
55         scanf("%d", &a[i]);//scanf的速度要比cin的速度快
56     }
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 }
```

Question 2

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

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

(1) 数组规模较小时使用；

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

(2) 数组规模较大，且不需要自行输入；

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

(3) 数组规模较大，且需要自行输入数组元素；

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

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

(1) 一维数组：

```
1 int arraysize; //数组规模
2 scanf("%d",&arraysize); //输入数组规模，scanf比cin快很多
3 int *arry=new int[arraysize]; //数组名是指针
4 for(int count=0;count<arraysize;count++)
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;
4
5 long int merge(int a[], int left, int mid, int right, int b[])
6 {
7     int i = mid;
8     int j = right;
9     long int lcount = 0;
10    while (i >= left && j > mid)
11    {
12        if (a[i] > (long long) 3 * a[j])
13        {
14            lcount += j - mid;
15            i--;
16        }
17        else
18        {
19            j--;
20        }
21    }
22    i = mid;
23    j = right;
24    int k = 0;
25    while (i >= left && j > mid)
26    {
27        if (a[i] > a[j])
28        {
29            b[k++] = a[i--];
30        }
31        else
32        {
33            b[k++] = a[j--];
34        }
35    }
36    while (i >= left)
37    {
38        b[k++] = a[i--];
39    }
40    while (j > mid)
41    {
42        b[k++] = a[j--];
43    }
44    for (i = 0; i < k; i++)
45    {
46        a[right - i] = b[i];
47    }
48    return lcount;
49 }
50
51 long int solve(int a[], int left, int right, int b[])
52 {
53     long int cnt = 0;
54     if (right > left)
55     {
```

```

57     int mid = (right+left) / 2;
58     cnt += solve(a, left , mid,b);
59     cnt += solve(a,mid + 1, right ,b);
60     cnt += merge(a, left , mid, right ,b);
61 }
62 return cnt;
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()
72 {
73     long int n;//数组维度
74     int *array;//数组
75     scanf("%d", &n);
76     array = new int [n];
77     for(long int i=0;i<n; i++)
78         scanf("%d", &array [i]);
79
80     long int count = InversePairs(array , n);
81     printf("%d", count);
82     return 0;
83 }

```

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

Question 4

快速排序算法

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

Question 5

计算分治问题时间复杂度的总结，主定理 (Master theorem))

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

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

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。

Question 7

leetcode 第 33 题，搜索旋转排序数组。问题描述：

Question 7

leetcode 第 153 题，寻找旋转排序数组中的最小值。问题描述：

Question 9

leetcode 第题，寻找一个数组的众数。问题描述：

Question 10

leetcode 第 200 题，计算岛屿的个数。问题描述：

这道题实际上是计算连通域的个数，可以用 DFS 和 BFS 求解。

Question 11

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

先介绍递归方法：

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

```

21     {
22         T = NULL;
23         return;
24     }
25     else
26     {
27         T = new TreeNode;
28         T->data = ch;//
29         cout << "input" << ch << "'s left son node:";
30         CreateBiTree(T->lchild);//
31         cout << "input" << ch << "'s right son node:";
32         CreateBiTree(T->rchild);//
33     }
34     return;
35 }
36 /*/
37 //上面这段创建树的代码创建的树无法使用,跟下面的相比少了指针
38
39 //先序创建二叉树
40 void CreateBiTree(TreeNode **T)
41 {
42     int ch;
43     cin >> ch;
44     if (ch == -1)
45     {
46         *T = NULL;
47         return;
48     }
49     else
50     {
51         *T = new TreeNode;
52         (*T)->data = ch;
53         cout << "input" << ch << "'s left son node:";
54         CreateBiTree(&((*T)->lchild));
55         cout << "input" << ch << "'s right son node:";
56         CreateBiTree(&((*T)->rchild));
57     }
58     return;
59 }
60
61 //先序遍历
62 void PreOrderBiTree(TreeNode *T)
63 {
64     if (T == NULL)
65     {
66         return;
67     }
68     else
69     {
70         cout << T->data << " ";
71         PreOrderBiTree(T->lchild);
72         PreOrderBiTree(T->rchild);
73     }
74 }
75
76 //中序遍历
77 void MiddleOrderBiTree(TreeNode *T)
78 {
79     if (T == NULL)
80     {
81         return;
82     }
83     else
84     {
85         MiddleOrderBiTree(T->lchild);
86     }
87 }
88
89 //后序遍历
90 void PostOrderBiTree(TreeNode *T)
91 {
92     if (T == NULL)
93     {
94         return;
95     }
96     else
97     {
98         PostOrderBiTree(T->lchild);
99         PostOrderBiTree(T->rchild);
100        cout << T->data << " ";
101    }
102 }
103
104 //层序遍历
105 void LevelOrderBiTree(TreeNode *T)
106 {
107     if (T == NULL)
108     {
109         return;
110     }
111     else
112     {
113         queue<TreeNode*> q;
114         q.push(T);
115         while (!q.empty())
116         {
117             TreeNode *t = q.front();
118             cout << t->data << " ";
119             q.pop();
120             if (t->lchild != NULL)
121                 q.push(t->lchild);
122             if (t->rchild != NULL)
123                 q.push(t->rchild);
124         }
125     }
126 }
127
128 //求二叉树的深度
129 int DepthBiTree(TreeNode *T)
130 {
131     if (T == NULL)
132     {
133         return 0;
134     }
135     else
136     {
137         int ldepth = DepthBiTree(T->lchild);
138         int rdepth = DepthBiTree(T->rchild);
139         if (ldepth > rdepth)
140             return ldepth + 1;
141         else
142             return rdepth + 1;
143     }
144 }
145
146 //求二叉树的节点数
147 int NodeCountBiTree(TreeNode *T)
148 {
149     if (T == NULL)
150     {
151         return 0;
152     }
153     else
154     {
155         int lcount = NodeCountBiTree(T->lchild);
156         int rcount = NodeCountBiTree(T->rchild);
157         return lcount + rcount + 1;
158     }
159 }
160
161 //求二叉树的叶子数
162 int LeafCountBiTree(TreeNode *T)
163 {
164     if (T == NULL)
165     {
166         return 0;
167     }
168     else
169     {
170         int lcount = LeafCountBiTree(T->lchild);
171         int rcount = LeafCountBiTree(T->rchild);
172         if (T->lchild == NULL && T->rchild == NULL)
173             return 1;
174         else
175             return lcount + rcount;
176     }
177 }
178
179 //求二叉树的分支数
180 int BranchCountBiTree(TreeNode *T)
181 {
182     if (T == NULL)
183     {
184         return 0;
185     }
186     else
187     {
188         int lcount = BranchCountBiTree(T->lchild);
189         int rcount = BranchCountBiTree(T->rchild);
190         return lcount + rcount + 1;
191     }
192 }
193
194 //求二叉树的高度
195 int HeightBiTree(TreeNode *T)
196 {
197     if (T == NULL)
198     {
199         return 0;
200     }
201     else
202     {
203         int lheight = HeightBiTree(T->lchild);
204         int rheight = HeightBiTree(T->rchild);
205         if (lheight > rheight)
206             return lheight + 1;
207         else
208             return rheight + 1;
209     }
210 }
211
212 //求二叉树的广度
213 int WidthBiTree(TreeNode *T)
214 {
215     if (T == NULL)
216     {
217         return 0;
218     }
219     else
220     {
221         queue<TreeNode*> q;
222         q.push(T);
223         int width = 1;
224         while (!q.empty())
225         {
226             int count = q.size();
227             for (int i = 0; i < count; i++)
228             {
229                 TreeNode *t = q.front();
230                 cout << t->data << " ";
231                 q.pop();
232                 if (t->lchild != NULL)
233                     q.push(t->lchild);
234                 if (t->rchild != NULL)
235                     q.push(t->rchild);
236             }
237             width++;
238         }
239     }
240 }
241
242 //求二叉树的满度
243 int FullWidthBiTree(TreeNode *T)
244 {
245     if (T == NULL)
246     {
247         return 0;
248     }
249     else
250     {
251         queue<TreeNode*> q;
252         q.push(T);
253         int fullwidth = 1;
254         while (!q.empty())
255         {
256             int count = q.size();
257             for (int i = 0; i < count; i++)
258             {
259                 TreeNode *t = q.front();
260                 cout << t->data << " ";
261                 q.pop();
262                 if (t->lchild != NULL)
263                     q.push(t->lchild);
264                 if (t->rchild != NULL)
265                     q.push(t->rchild);
266             }
267             fullwidth++;
268         }
269     }
270 }
271
272 //求二叉树的平衡度
273 int BalanceBiTree(TreeNode *T)
274 {
275     if (T == NULL)
276     {
277         return 0;
278     }
279     else
280     {
281         int lheight = HeightBiTree(T->lchild);
282         int rheight = HeightBiTree(T->rchild);
283         if (lheight - rheight >= 2 || rheight - lheight >= 2)
284             return 1;
285         else
286             return 0;
287     }
288 }
289
290 //求二叉树的平衡因子
291 int BalanceFactorBiTree(TreeNode *T)
292 {
293     if (T == NULL)
294     {
295         return 0;
296     }
297     else
298     {
299         int lheight = HeightBiTree(T->lchild);
300         int rheight = HeightBiTree(T->rchild);
301         if (lheight - rheight >= 1 || rheight - lheight >= 1)
302             return 1;
303         else
304             return 0;
305     }
306 }
307
308 //求二叉树的平衡性
309 int BalanceBiTree(TreeNode *T)
310 {
311     if (T == NULL)
312     {
313         return 0;
314     }
315     else
316     {
317         int lbalance = BalanceBiTree(T->lchild);
318         int rbalance = BalanceBiTree(T->rchild);
319         if (lbalance == 0 && rbalance == 0)
320             return 0;
321         else
322             return 1;
323     }
324 }
325
326 //求二叉树的平衡因子
327 int BalanceFactorBiTree(TreeNode *T)
328 {
329     if (T == NULL)
330     {
331         return 0;
332     }
333     else
334     {
335         int lbalance = BalanceFactorBiTree(T->lchild);
336         int rbalance = BalanceFactorBiTree(T->rchild);
337         if (lbalance == 0 && rbalance == 0)
338             return 0;
339         else
340             return 1;
341     }
342 }
343
344 //求二叉树的平衡性
345 int BalanceBiTree(TreeNode *T)
346 {
347     if (T == NULL)
348     {
349         return 0;
350     }
351     else
352     {
353         int lbalance = BalanceFactorBiTree(T->lchild);
354         int rbalance = BalanceFactorBiTree(T->rchild);
355         if (lbalance == 1 && rbalance == 1)
356             return 1;
357         else
358             return 0;
359     }
360 }
361
362 //求二叉树的平衡因子
363 int BalanceFactorBiTree(TreeNode *T)
364 {
365     if (T == NULL)
366     {
367         return 0;
368     }
369     else
370     {
371         int lbalance = BalanceFactorBiTree(T->lchild);
372         int rbalance = BalanceFactorBiTree(T->rchild);
373         if (lbalance == 1 && rbalance == 1)
374             return 1;
375         else
376             return 0;
377     }
378 }
379
380 //求二叉树的平衡性
381 int BalanceBiTree(TreeNode *T)
382 {
383     if (T == NULL)
384     {
385         return 0;
386     }
387     else
388     {
389         int lbalance = BalanceFactorBiTree(T->lchild);
390         int rbalance = BalanceFactorBiTree(T->rchild);
391         if (lbalance == 1 && rbalance == 1)
392             return 1;
393         else
394             return 0;
395     }
396 }
397
398 //求二叉树的平衡因子
399 int BalanceFactorBiTree(TreeNode *T)
400 {
401     if (T == NULL)
402     {
403         return 0;
404     }
405     else
406     {
407         int lbalance = BalanceFactorBiTree(T->lchild);
408         int rbalance = BalanceFactorBiTree(T->rchild);
409         if (lbalance == 1 && rbalance == 1)
410             return 1;
411         else
412             return 0;
413     }
414 }
415
416 //求二叉树的平衡性
417 int BalanceBiTree(TreeNode *T)
418 {
419     if (T == NULL)
420     {
421         return 0;
422     }
423     else
424     {
425         int lbalance = BalanceFactorBiTree(T->lchild);
426         int rbalance = BalanceFactorBiTree(T->rchild);
427         if (lbalance == 1 && rbalance == 1)
428             return 1;
429         else
430             return 0;
431     }
432 }
433
434 //求二叉树的平衡因子
435 int BalanceFactorBiTree(TreeNode *T)
436 {
437     if (T == NULL)
438     {
439         return 0;
440     }
441     else
442     {
443         int lbalance = BalanceFactorBiTree(T->lchild);
444         int rbalance = BalanceFactorBiTree(T->rchild);
445         if (lbalance == 1 && rbalance == 1)
446             return 1;
447         else
448             return 0;
449     }
450 }
451
452 //求二叉树的平衡性
453 int BalanceBiTree(TreeNode *T)
454 {
455     if (T == NULL)
456     {
457         return 0;
458     }
459     else
460     {
461         int lbalance = BalanceFactorBiTree(T->lchild);
462         int rbalance = BalanceFactorBiTree(T->rchild);
463         if (lbalance == 1 && rbalance == 1)
464             return 1;
465         else
466             return 0;
467     }
468 }
469
470 //求二叉树的平衡因子
471 int BalanceFactorBiTree(TreeNode *T)
472 {
473     if (T == NULL)
474     {
475         return 0;
476     }
477     else
478     {
479         int lbalance = BalanceFactorBiTree(T->lchild);
480         int rbalance = BalanceFactorBiTree(T->rchild);
481         if (lbalance == 1 && rbalance == 1)
482             return 1;
483         else
484             return 0;
485     }
486 }
487
488 //求二叉树的平衡性
489 int BalanceBiTree(TreeNode *T)
490 {
491     if (T == NULL)
492     {
493         return 0;
494     }
495     else
496     {
497         int lbalance = BalanceFactorBiTree(T->lchild);
498         int rbalance = BalanceFactorBiTree(T->rchild);
499         if (lbalance == 1 && rbalance == 1)
500             return 1;
501         else
502             return 0;
503     }
504 }
505
506 //求二叉树的平衡因子
507 int BalanceFactorBiTree(TreeNode *T)
508 {
509     if (T == NULL)
510     {
511         return 0;
512     }
513     else
514     {
515         int lbalance = BalanceFactorBiTree(T->lchild);
516         int rbalance = BalanceFactorBiTree(T->rchild);
517         if (lbalance == 1 && rbalance == 1)
518             return 1;
519         else
520             return 0;
521     }
522 }
523
524 //求二叉树的平衡性
525 int BalanceBiTree(TreeNode *T)
526 {
527     if (T == NULL)
528     {
529         return 0;
530     }
531     else
532     {
533         int lbalance = BalanceFactorBiTree(T->lchild);
534         int rbalance = BalanceFactorBiTree(T->rchild);
535         if (lbalance == 1 && rbalance == 1)
536             return 1;
537         else
538             return 0;
539     }
540 }
541
542 //求二叉树的平衡因子
543 int BalanceFactorBiTree(TreeNode *T)
544 {
545     if (T == NULL)
546     {
547         return 0;
548     }
549     else
550     {
551         int lbalance = BalanceFactorBiTree(T->lchild);
552         int rbalance = BalanceFactorBiTree(T->rchild);
553         if (lbalance == 1 && rbalance == 1)
554             return 1;
555         else
556             return 0;
557     }
558 }
559
560 //求二叉树的平衡性
561 int BalanceBiTree(TreeNode *T)
562 {
563     if (T == NULL)
564     {
565         return 0;
566     }
567     else
568     {
569         int lbalance = BalanceFactorBiTree(T->lchild);
570         int rbalance = BalanceFactorBiTree(T->rchild);
571         if (lbalance == 1 && rbalance == 1)
572             return 1;
573         else
574             return 0;
575     }
576 }
577
578 //求二叉树的平衡因子
579 int BalanceFactorBiTree(TreeNode *T)
580 {
581     if (T == NULL)
582     {
583         return 0;
584     }
585     else
586     {
587         int lbalance = BalanceFactorBiTree(T->lchild);
588         int rbalance = BalanceFactorBiTree(T->rchild);
589         if (lbalance == 1 && rbalance == 1)
590             return 1;
591         else
592             return 0;
593     }
594 }
595
596 //求二叉树的平衡性
597 int BalanceBiTree(TreeNode *T)
598 {
599     if (T == NULL)
600     {
601         return 0;
602     }
603     else
604     {
605         int lbalance = BalanceFactorBiTree(T->lchild);
606         int rbalance = BalanceFactorBiTree(T->rchild);
607         if (lbalance == 1 && rbalance == 1)
608             return 1;
609         else
610             return 0;
611     }
612 }
613
614 //求二叉树的平衡因子
615 int BalanceFactorBiTree(TreeNode *T)
616 {
617     if (T == NULL)
618     {
619         return 0;
620     }
621     else
622     {
623         int lbalance = BalanceFactorBiTree(T->lchild);
624         int rbalance = BalanceFactorBiTree(T->rchild);
625         if (lbalance == 1 && rbalance == 1)
626             return 1;
627         else
628             return 0;
629     }
630 }
631
632 //求二叉树的平衡性
633 int BalanceBiTree(TreeNode *T)
634 {
635     if (T == NULL)
636     {
637         return 0;
638     }
639     else
640     {
641         int lbalance = BalanceFactorBiTree(T->lchild);
642         int rbalance = BalanceFactorBiTree(T->rchild);
643         if (lbalance == 1 && rbalance == 1)
644             return 1;
645         else
646             return 0;
647     }
648 }
649
650 //求二叉树的平衡因子
651 int BalanceFactorBiTree(TreeNode *T)
652 {
653     if (T == NULL)
654     {
655         return 0;
656     }
657     else
658     {
659         int lbalance = BalanceFactorBiTree(T->lchild);
660         int rbalance = BalanceFactorBiTree(T->rchild);
661         if (lbalance == 1 && rbalance == 1)
662             return 1;
663         else
664             return 0;
665     }
666 }
667
668 //求二叉树的平衡性
669 int BalanceBiTree(TreeNode *T)
670 {
671     if (T == NULL)
672     {
673         return 0;
674     }
675     else
676     {
677         int lbalance = BalanceFactorBiTree(T->lchild);
678         int rbalance = BalanceFactorBiTree(T->rchild);
679         if (lbalance == 1 && rbalance == 1)
680             return 1;
681         else
682             return 0;
683     }
684 }
685
686 //求二叉树的平衡因子
687 int BalanceFactorBiTree(TreeNode *T)
688 {
689     if (T == NULL)
690     {
691         return 0;
692     }
693     else
694     {
695         int lbalance = BalanceFactorBiTree(T->lchild);
696         int rbalance = BalanceFactorBiTree(T->rchild);
697         if (lbalance == 1 && rbalance == 1)
698             return 1;
699         else
700             return 0;
701     }
702 }
703
704 //求二叉树的平衡性
705 int BalanceBiTree(TreeNode *T)
706 {
707     if (T == NULL)
708     {
709         return 0;
710     }
711     else
712     {
713         int lbalance = BalanceFactorBiTree(T->lchild);
714         int rbalance = BalanceFactorBiTree(T->rchild);
715         if (lbalance == 1 && rbalance == 1)
716             return 1;
717         else
718             return 0;
719     }
720 }
721
722 //求二叉树的平衡因子
723 int BalanceFactorBiTree(TreeNode *T)
724 {
725     if (T == NULL)
726     {
727         return 0;
728     }
729     else
730     {
731         int lbalance = BalanceFactorBiTree(T->lchild);
732         int rbalance = BalanceFactorBiTree(T->rchild);
733         if (lbalance == 1 && rbalance == 1)
734             return 1;
735         else
736             return 0;
737     }
738 }
739
740 //求二叉树的平衡性
741 int BalanceBiTree(TreeNode *T)
742 {
743     if (T == NULL)
744     {
745         return 0;
746     }
747     else
748     {
749         int lbalance = BalanceFactorBiTree(T->lchild);
750         int rbalance = BalanceFactorBiTree(T->rchild);
751         if (lbalance == 1 && rbalance == 1)
752             return 1;
753         else
754             return 0;
755     }
756 }
757
758 //求二叉树的平衡因子
759 int BalanceFactorBiTree(TreeNode *T)
760 {
761     if (T == NULL)
762     {
763         return 0;
764     }
765     else
766     {
767         int lbalance = BalanceFactorBiTree(T->lchild);
768         int rbalance = BalanceFactorBiTree(T->rchild);
769         if (lbalance == 1 && rbalance == 1)
770             return 1;
771         else
772             return 0;
773     }
774 }
775
776 //求二叉树的平衡性
777 int BalanceBiTree(TreeNode *T)
778 {
779     if (T == NULL)
780     {
781         return 0;
782     }
783     else
784     {
785         int lbalance = BalanceFactorBiTree(T->lchild);
786         int rbalance = BalanceFactorBiTree(T->rchild);
787         if (lbalance == 1 && rbalance == 1)
788             return 1;
789         else
790             return 0;
791     }
792 }
793
794 //求二叉树的平衡因子
795 int BalanceFactorBiTree(TreeNode *T)
796 {
797     if (T == NULL)
798     {
799         return 0;
800     }
801     else
802     {
803         int lbalance = BalanceFactorBiTree(T->lchild);
804         int rbalance = BalanceFactorBiTree(T->rchild);
805         if (lbalance == 1 && rbalance == 1)
806             return 1;
807         else
808             return 0;
809     }
810 }
811
812 //求二叉树的平衡性
813 int BalanceBiTree(TreeNode *T)
814 {
815     if (T == NULL)
816     {
817         return 0;
818     }
819     else
820     {
821         int lbalance = BalanceFactorBiTree(T->lchild);
822         int rbalance = BalanceFactorBiTree(T->rchild);
823         if (lbalance == 1 && rbalance == 1)
824             return 1;
825         else
826             return 0;
827     }
828 }
829
830 //求二叉树的平衡因子
831 int BalanceFactorBiTree(TreeNode *T)
832 {
833     if (T == NULL)
834     {
835         return 0;
836     }
837     else
838     {
839         int lbalance = BalanceFactorBiTree(T->lchild);
840         int rbalance = BalanceFactorBiTree(T->rchild);
841         if (lbalance == 1 && rbalance == 1)
842             return 1;
843         else
844             return 0;
845     }
846 }
847
848 //求二叉树的平衡性
849 int BalanceBiTree(TreeNode *T)
850 {
851     if (T == NULL)
852     {
853         return 0;
854     }
855     else
856     {
857         int lbalance = BalanceFactorBiTree(T->lchild);
858         int rbalance = BalanceFactorBiTree(T->rchild);
859         if (lbalance == 1 && rbalance == 1)
860             return 1;
861         else
862             return 0;
863     }
864 }
865
866 //求二叉树的平衡因子
867 int BalanceFactorBiTree(TreeNode *T)
868 {
869     if (T == NULL)
870     {
871         return 0;
872     }
873     else
874     {
875         int lbalance = BalanceFactorBiTree(T->lchild);
876         int rbalance = BalanceFactorBiTree(T->rchild);
877         if (lbalance == 1 && rbalance == 1)
878             return 1;
879         else
880             return 0;
881     }
882 }
883
884 //求二叉树的平衡性
885 int BalanceBiTree(TreeNode *T)
886 {
887     if (T == NULL)
888     {
889         return 0;
890     }
891     else
892     {
893         int lbalance = BalanceFactorBiTree(T->lchild);
894         int rbalance = BalanceFactorBiTree(T->rchild);
895         if (lbalance == 1 && rbalance == 1)
896             return 1;
897         else
898             return 0;
899     }
900 }
901
902 //求二叉树的平衡因子
903 int BalanceFactorBiTree(TreeNode *T)
904 {
905     if (T == NULL)
906     {
907         return 0;
908     }
909     else
910     {
911         int lbalance = BalanceFactorBiTree(T->lchild);
912         int rbalance = BalanceFactorBiTree(T->rchild);
913         if (lbalance == 1 && rbalance == 1)
914             return 1;
915         else
916             return 0;
917     }
918 }
919
920 //求二叉树的平衡性
921 int BalanceBiTree(TreeNode *T)
922 {
923     if (T == NULL)
924     {
925         return 0;
926     }
927     else
928     {
929         int lbalance = BalanceFactorBiTree(T->lchild);
930         int rbalance = BalanceFactorBiTree(T->rchild);
931         if (lbalance == 1 && rbalance == 1)
932             return 1;
933         else
934             return 0;
935     }
936 }
937
938 //求二叉树的平衡因子
939 int BalanceFactorBiTree(TreeNode *T)
940 {
941     if (T == NULL)
942     {
943         return 0;
944     }
945     else
946     {
947         int lbalance = BalanceFactorBiTree(T->lchild);
948         int rbalance = BalanceFactorBiTree(T->rchild);
949         if (lbalance == 1 && rbalance == 1)
950             return 1;
951         else
952             return 0;
953     }
954 }
955
956 //求二叉树的平衡性
957 int BalanceBiTree(TreeNode *T)
958 {
959     if (T == NULL)
960     {
961         return 0;
962     }
963     else
964     {
965         int lbalance = BalanceFactorBiTree(T->lchild);
966         int rbalance = BalanceFactorBiTree(T->rchild);
967         if (lbalance == 1 && rbalance == 1)
968             return 1;
969         else
970             return 0;
971     }
972 }
973
974 //求二叉树的平衡因子
975 int BalanceFactorBiTree(TreeNode *T)
976 {
977     if (T == NULL)
978     {
979         return 0;
980     }
981     else
982     {
983         int lbalance = BalanceFactorBiTree(T->lchild);
984         int rbalance = BalanceFactorBiTree(T->rchild);
985         if (lbalance == 1 && rbalance == 1)
986             return 1;
987         else
988             return 0;
989     }
990 }
991
992 //求二叉树的平衡
```

```

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

```

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

```

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

Question 12

寻找二叉树上最远的两个节点的距离。问题描述：

介绍完分治思想部分的题，接下来的题将会面向动态规划。流程大致如下：

Q1: 从最简单的 case 入手；

Q2: 对大的 case 分解。

如何分解？这实际上是一个多步决策的过程：

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

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

Question 13

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

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

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

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

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

```

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

```

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

图 4: trial 1

此算法的时间复杂度为指数级的，证明如图 5。

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

Question 14

背包问题。

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

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

```

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

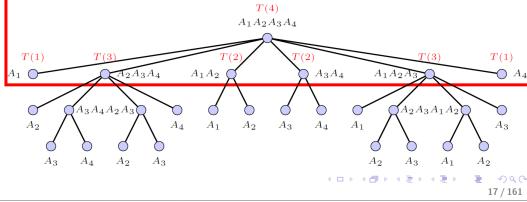
```

However, this is not a good implementation

Theorem

Algorithm RECURSIVE-MATRIX-CHAIN costs exponential time.

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



(a) 已知的条件

Proof.

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

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

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

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

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

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

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

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

□

(b) 数学归纳法证明

图 5: 指数级时间复杂度证明

Algorithm

```

1: 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

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

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

(b) Step1

Step 2

$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$i = 1$	$i = 2$	$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$

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

(c) Step2

Step 3

$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$	$i = 8$	$i = 9$

$OPT[2, 4] = \max\{OPT[1, 4] (= 2), OPT[1, 3] + V_2 (= 2 + 3)\} = 5$

(d) Step3

Step 4

$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$	$i = 8$	$i = 9$

$OPT[3, 3] = \max\{OPT[2, 3] (= 2), OPT[2, 2] + V_3 (= 2 + 3)\} = 5$

(e) Step4

Backtracking

$w = 0$	$w = 1$	$w = 2$	$w = 3$	$w = 4$	$w = 5$	$w = 6$
$i = 3$	$i = 4$	$i = 5$	$i = 6$	$i = 7$	$i = 8$	$i = 9$

Decision: Select item 3

$OPT[3, 6] = \max\{OPT[2, 6] (= 4), OPT[2, 5] + V_3 (= 2 + 3)\} = 5$

$OPT[2, 6] = \max\{OPT[1, 6] (= 2), OPT[1, 5] + V_2 (= 0 + 2)\} = 2$

Decision: Select item 2

(f) Step5

图 6: 伪代码及递归公式示例

```

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

```

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

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

```

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

```

```

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

```

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

```

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

```

```

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

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

48 cout<<O<<endl;
49 return 0;
50 }
```

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

```

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

```

21     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
36 //动态规划求解，实际上在计算一个三维矩阵
37     for (int i = 1; i <= k; i++)
38     {
39         for (int j = 1; j <= M; j++)
40         {
41             for (int s = 1; s <= N; s++)
42             {
43                 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
44                     newOPT[j][s] = OPT[j][s];
45                 else
46                     newOPT[j][s] = max(OPT[j][s], OPT[j - m[i - 1]][s - n[i - 1]] + u[i - 1]);
47             }
48         }
49     }
50 //新的矩阵代替旧的矩阵
51     for (int j = 1; j <= M; j++)
52     {
53         for (int s = 1; s <= N; s++)
54         {
55             OPT[j][s] = newOPT[j][s];
56         }
57     }
58
59     int O = newOPT[M][N];
60     delete []m;
61     delete []n;
62     delete []u;
63     for (int i = 0; i < M + 1; i++)
64     {
65         delete []OPT[i];
66         delete []newOPT[i];
67     }
68     cout << O << endl;
69     system("pause");
70     return 0;
71 }
```

Question 15

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

程。

Alignment is useful cont'd

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

① Alignment 1:

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

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

② Alignment 2:

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

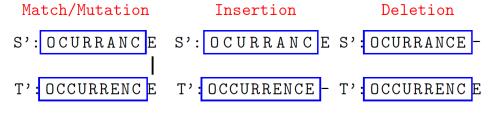
$$s(T', S') = 1 - 3 + 1 + 1 + 1 + 1 - 3 - 3 + 1 + 1 + 1 = -1$$

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

The general form of sub-problems and recursions II

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

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



72 / 161

(b) 递归公式推导过程

(a) 示例

图 7: 打分函数示例及递归公式推导过程

递归公式及伪代码如下：

The general form of sub-problems and recursions III

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

- We can prove the following optimal substructure property:

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

Needleman-Wunsch algorithm [1970]

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

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

76 / 161

77 / 161

(a) 递归公式

(b) 伪代码

图 8: alignment 递归公式及伪代码

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

Question 16

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

General cases

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

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

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

(a) 二维矩阵

Find the optimal alignment via backtracking

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

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

(b) 回溯路线

图 9: 二维矩阵及回溯路线

Question 17

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

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

LONGEST COMMON SUBSEQUENCE problem

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

107 / 161

Basic DP algorithm

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

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

108 / 161

(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：

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

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

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

```

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

```
#include <iostream>
```

```

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

上面的代码实际上是在打印了这样一个二维矩阵：

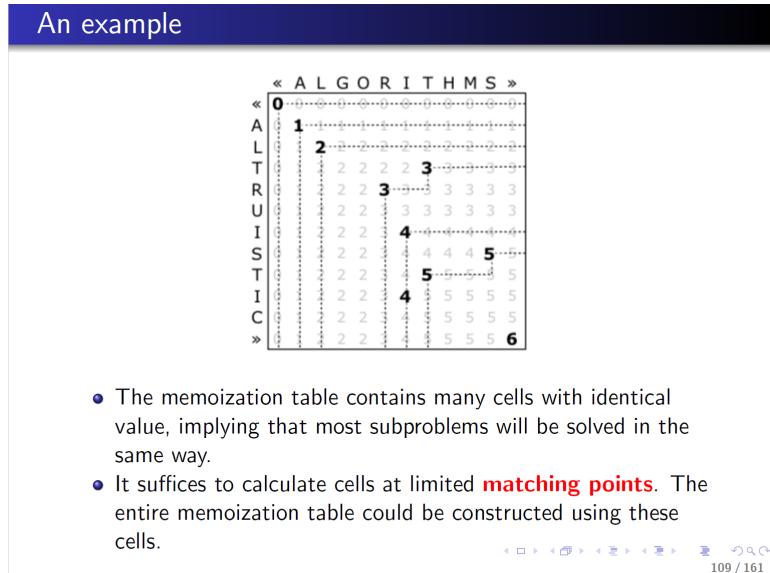


图 11：最长公共子序列的动态决策过程

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

```

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

```

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

Question 18

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

Question 19

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

房子并列：

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

实际代码：

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

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

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

伪代码：(只写后一种方法的伪代码)

```

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

```

房子围成一圈：

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

2. 代码及伪代码:

```

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

```

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

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

```
2 #include <iostream>
3 #include <stdio.h>
4 #include <algorithm>
5
6 using namespace std;
7
8 //定义二叉树结构体
9 struct TreeNode{
10     int data;
11     TreeNode *lchild;
12     TreeNode *rchild;
13 };
14
15 //先序创建二叉树
16 void CreateBiTree(TreeNode **T)
17 {
18     int ch;
19     cin >> ch;
20     if (ch == -1)
21     {
22         *T = NULL;
23         return ;
24     }
25 }
```

```

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

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

```

伪代码:

```

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

```

Question 21

有向无环图中的单元最短路径，考虑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，

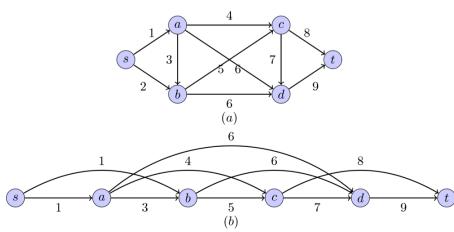
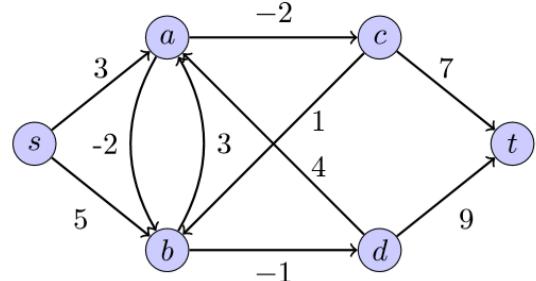


图 3.23: 有向无环图及结点的拓扑排序示例

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

Viterbi's decoding algorithm: recursion

- First we rewrite $\max_X P(X, Y)$ as:
$$\max_{x_n} \max_{x_{n-1}} \dots \max_{x_1} e_{x_n}(y_n) a_{x_{n-1} x_n} e_{x_{n-1}}(y_{n-1}) \dots a_{x_1 x_2} e_{x_1}(y_1) a_{0 x_1}$$
- Let denote $v_i(k)$ as
$$\max_{x_{i-1}} \max_{x_1} e_k(y_i) a_{x_{i-1} k} e_{x_{i-1}}(y_{i-1}) \dots a_{x_1 x_2} e_{x_1}(y_1) a_{0 x_1}$$

Then we have:

$$\max_X P(X, Y) = \max_k v_n(k)$$
- We can also observe the following recursion:
$$v_i(k) = e_k(y_i) \max_l (a_{lk} v_{i-1}(l))$$

Viterbi's decoding algorithm

```

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

(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

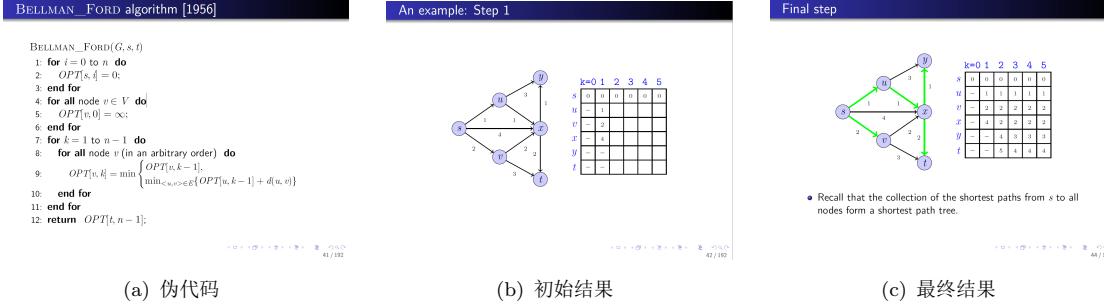


图 14: 伪代码及实例运算过程

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。

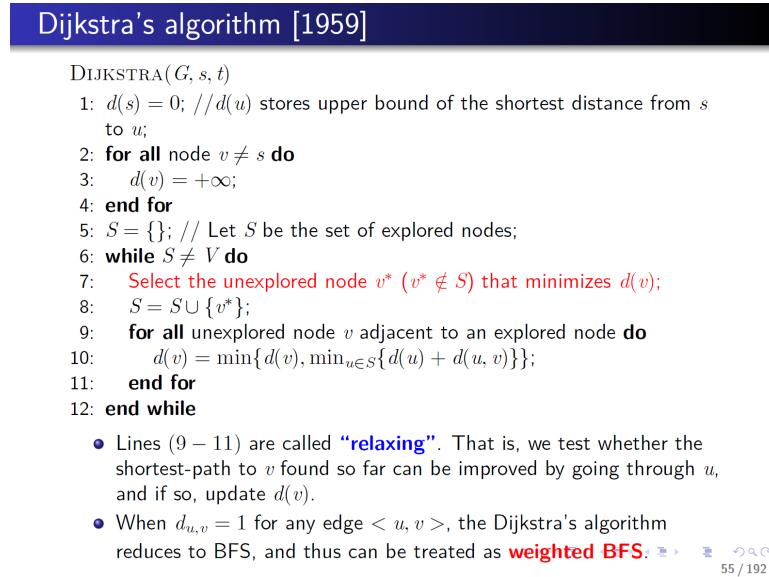


图 15: Dijksta 算法

Question 100

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

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
// 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。
3 // 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。
#include <iostream>
5 #include <vector>

```

```

7 using namespace std;
9
9 class Solution {
public:
11    bool Find(int target ,vector<vector<int>> array) {
12        int rowCount = array.size();
13        int colCount = array[0].size();
14        int i,j;
15        for(i=rowCount-1,j=0;i>=0&&j<colCount;)
16        {
17            if(target == array[i][j])
18                return true;
19            if(target < array[i][j])
20            {
21                i--;
22                continue;
23            }
24            if(target > array[i][j])
25            {
26                j++;
27                continue;
28            }
29        }
30        return false;
31    }
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;
54 }
```

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。

```

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
8 using namespace std;
9
10 class Solution
```

```

12 {
13     public:
14         char * replaceSpace( string str )
15     {
16         int len = str.length();
17         //先计算有多少个空格
18         int count = 0;
19         for( int i=0;i<len ;i++)
20         {
21             if( str[ i]== ' ')
22             {
23                 count++;
24             }
25         }
26         char *new_str = new char[ len+2*count]; //新的字符串
27         int N = 0;//记录当前有几个空格
28         for( int i=0;i<len ;i++)
29         {
30             if( str[ i]!= ' ')
31             {
32                 new_str[ i+2*N] = str[ i];
33             }
34             else
35             {
36                 new_str[ i+2*N] = '%';
37                 new_str[ i+2*N+1] = '2';
38                 new_str[ i+2*N+2] = '0';
39                 N++;
40             }
41         }
42         return new_str;
43     }
44 }
45 int main()
46 {
47     Solution s;
48     string str= "ab cd ds";
49     char* new_str = s.replaceSpace(str);
50     string t(new_str);
51     cout<<t;
52     system("pause");
53 }
```

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 };
13
```

```

15 class Solution {
16 public:
17     vector<int> printListFromTailToHead(ListNode* head) {
18         vector<int> A;
19         while(head!=NULL)
20         {
21             A.push_back(head->val);
22             head = head->next;
23         }
24         int len = A.size();
25         vector<int>B(len);
26         for(int i=0;i<len;i++)
27         {
28             B[i] = A[len-i-1];
29         }
30         return B;
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. 反转链表。输入一个链表，反转链表后，输出新链表的表头。

```

1 //反转链表
2 //输入一个链表，反转链表后，输出新链表的表头。
3
4 #include <iostream>
5
6 using namespace std;
7
8 struct ListNode{
9     int val;
10    ListNode*next;
11 };
12
13 class Solution {
14 public:
15     ListNode* ReverseList(ListNode* pHead) {
16         ListNode *p = NULL;
17         ListNode *pre = NULL;
18
19         while(pHead!=NULL)
20         {
21             p = pHead->next;
22             pHead->next = pre;
23             pre = pHead;
24             pHead = p;
25         }
26         return pre;
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; //新结点指向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     {

```

```

65     cout<<phead->val<<" ";
66     phead = phead->next;
67 }
68 system("pause");
69 return 0;
}

```

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前驱节点。

5. 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。

Part III

机器学习与深度学习基础模型与算法

机器学习模型主要分为两大类：生成式模型和判别式模型。

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

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

机器学习部分

Question 1

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

Question 2

数据聚类
深度学习部分

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: $\{x = (x_1, x_2, \dots, x_d), t = (t_1, t_2, \dots, t_c)\}$, 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: $\{neth = (neth_1, neth_2, \dots, neth_h, \dots, neth_{n_H}), z = (y_1, y_2, \dots, y_h, \dots, y_{n_H})\}$;
- (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；

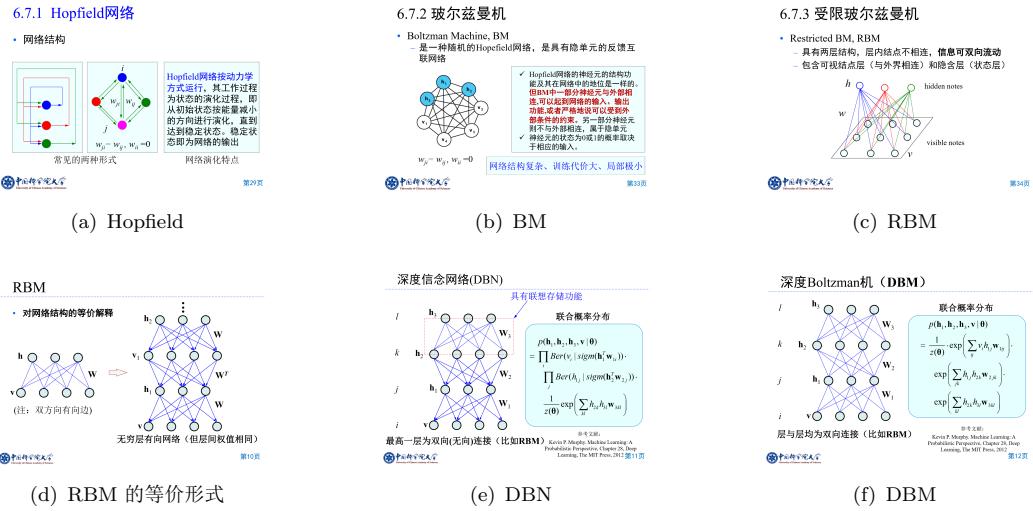


图 16: Hopfield, BM, RBM, DBN, DBM

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)。

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。

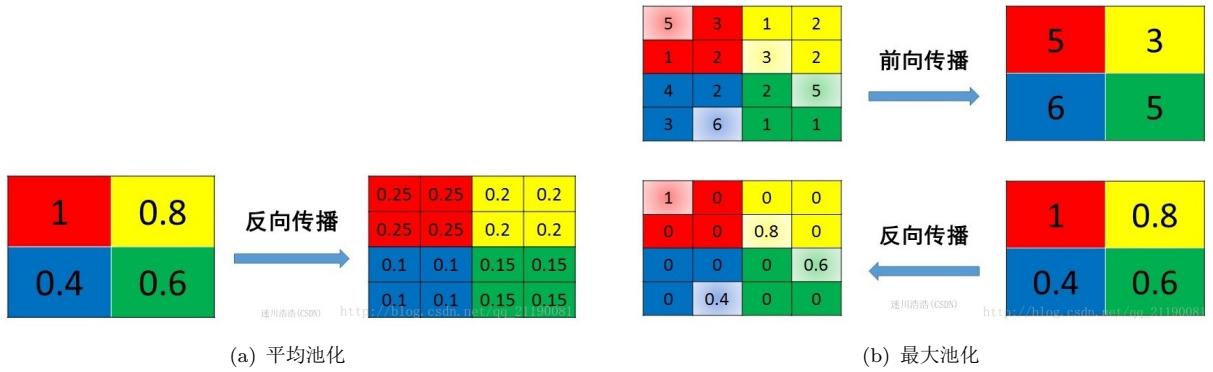


图 17: 池化层反向传播

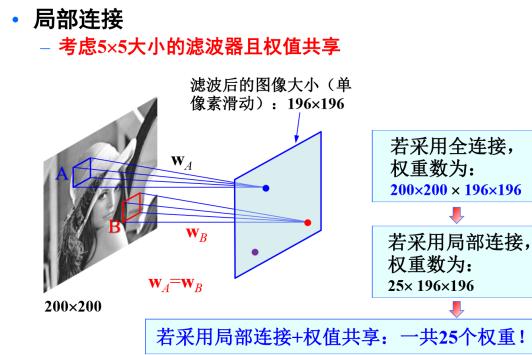


图 18: 参数情况

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。

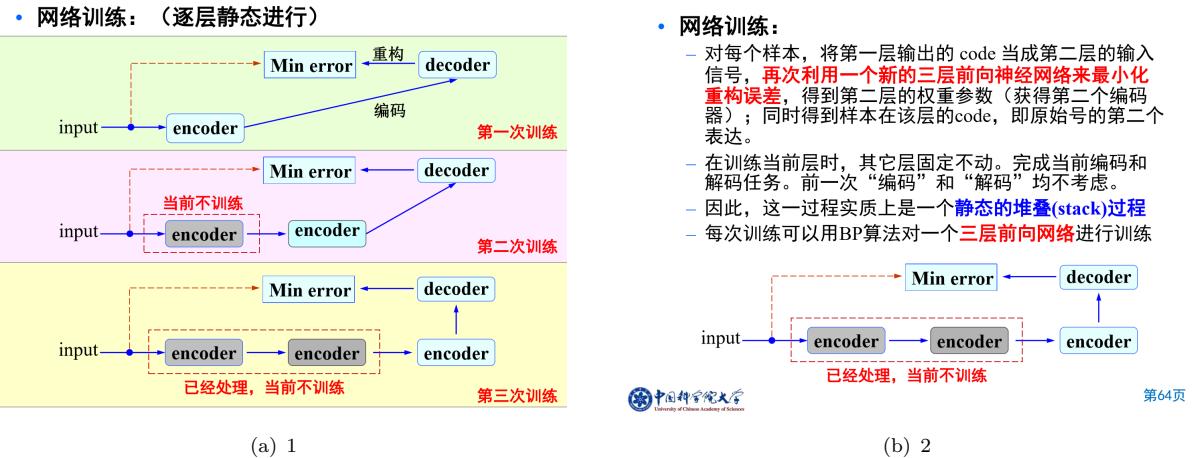


图 19: AutoEncoder 训练过程

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。

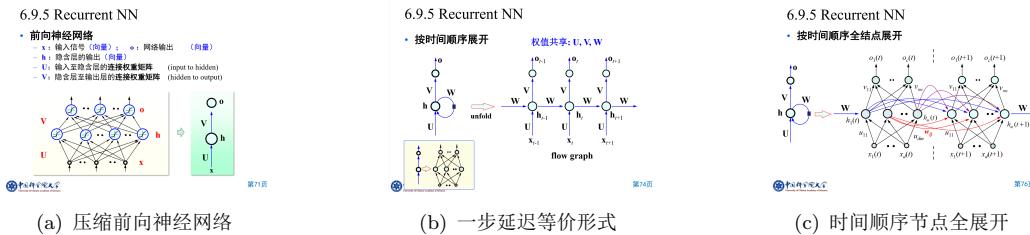


图 20: RNN

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:

$$\begin{aligned} \mathbf{f}_t &= \text{sigmod}(\mathbf{b}_f + \mathbf{U}_f \mathbf{x}_t + \mathbf{W}_f \mathbf{h}_{t-1}) \\ \mathbf{i}_t &= \text{sigmod}(\mathbf{b}_i + \mathbf{U}_i \mathbf{x}_t + \mathbf{W}_i \mathbf{h}_{t-1}) \\ \mathbf{o}_t &= \text{sigmod}(\mathbf{b}_o + \mathbf{U}_o \mathbf{x}_t + \mathbf{W}_o \mathbf{h}_{t-1}) \end{aligned}$$

候选记忆 (新贡献部分) :

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

Cell产生的新记忆:

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

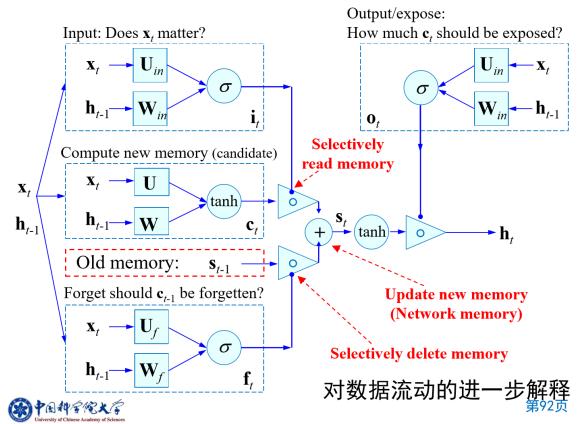
Cell的输出:

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

网络的输出:

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

(a) 1



(b) 2

图 21: LSTM