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 5 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，$\| A \|_F^2 = \sum_{i,j=1} a_{ij}^2 = tr(A^TA)$，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^TA 是对称半正定矩阵，则 $\exists$ 正交矩阵 $Q, s.t.$

$$A^TA = Q^T\Lambda Q, \Lambda = diag(\lambda_1, \lambda_2, ..., \lambda_n), \lambda_i \geq 0$$

且有：

$$\| A \|_2^2 = (Ax)^T(Ax) = x^TA^TAx = x^TQ^T\Lambda Qx = (Qx)^T\Lambda(Qx)$$

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^TA 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 $\| A \|_2 = \| PA \|_2 = \| AQ \|_2 = \| PAQ \|_2$，但是会改变 $1-$ 范数;

(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 $X = \{x_1, x_2, ..., x_n\}, iid$ 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。}

$$f(X) = \frac{1}{k^n}, 0 \le x_i \le k.$$
$$lnL(k) = -nlnk, lnL(k)' = -\frac{n}{k} < 0.$$

不存在 k 的极大似然估计。

Part II
计算机算法设计与分析

首先介绍分治思想，求解问题的大概流程如下：
Q1: 从最简单的 $case$ 入手；
Q2: 复杂问题，分解为 $sub-problems$。
如何分解：
1. 看 $Input$: 输入的关键数据结构 (DS, 包括数组、树、有向无环图、图、集合)，决定是否可分；
2. 看 $Output$: 决定能否把解合起来。

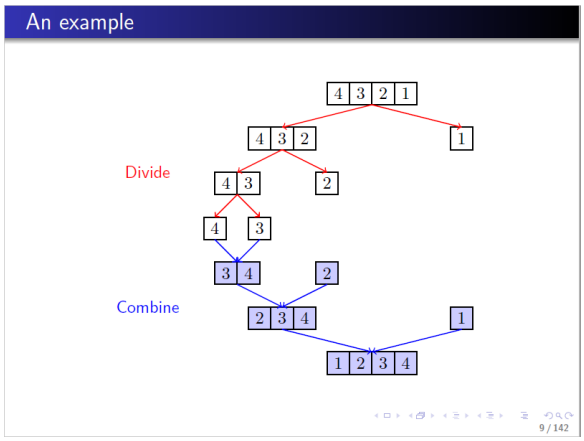
Question 1

用时间复杂度尽可能少的算法来排序一个 n 个整数的数组。
解：(1) 首先想到的是利用冒泡排序，利用两个 for 循环来排序数组，这种方法的时间复杂度是 $O(n^2)$, 代码较简单, 没有递归调用，略去；
(2) 采用 DC(divide and conquer) 思想，每次递归调用数组 $[0, n]$ 的前 $n-1$ 个元素，再回溯合并，大致过程如下图所示 (选自卜东波老师上课的 slides)。

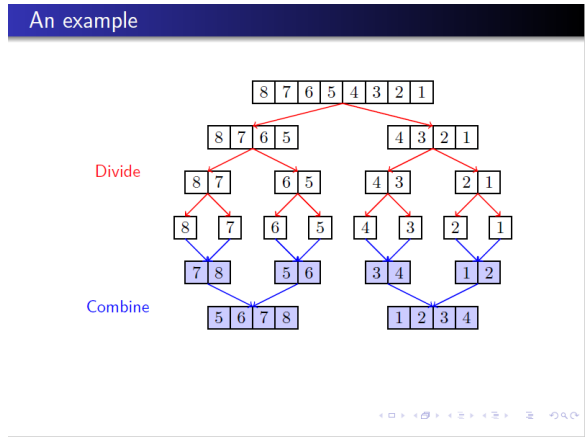
合并的时候将末尾的第 n 个元素插入前 $n-1$ 个元素当中，时间复杂度为 $O(n)$，所以有迭代式：$T(n) = T(n-1) + O(n)$，简单推导：

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

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



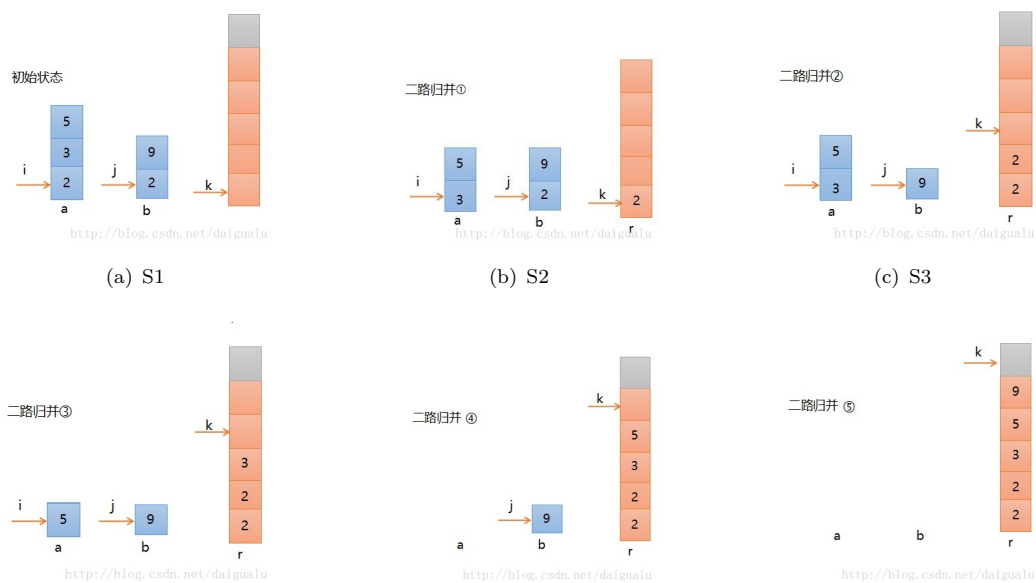
(a) 从倒数第二个元素位置一分为二



(b) 从中间的位置一分为二

图 1: 采用分治思想排序

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



(a) S1 (b) S2 (c) S3

(d) S4 (e) S5 (f) S6

图 2: 二路归并过程

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

```cpp
using namespace std;

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

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


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

  solve(a,0,n-1,b);//归并排序

  for(int i = 0;i<n;i++)
    cout<<a[i]<<' ';
  return 0;
}
```

Listing 1: 归并排序,C++

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

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

using namespace std;

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

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


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

  solve(a,0,n-1,b);//归并排序

  for(int i = 0;i<n;i++)
```

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

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；

```
  vector<vector<int>  >vec;
  vector<int>a;
  a.push_back(1);
  a.push_back(2);
  vetor<int>b;
  b.push_back(3);
  b.push_back(4);
  vec.push_back(a);
  vec.push_back(b);
```

(2) 数组规模较大，且不需要自行输入；

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

  for(int i=0;i<arry.size();i++)
    for(int j=0;j<array[0].size();j++)
      arry[i][j]=i*j;
```

(3) 数组规模较大，且需要自行输入数组元素；

```
int m,n;
cin>>m>>n;//输入时可以中间可以加空格
vector<vector<int>  >arry;
for(int i=0;i<arry.size();i++)
  for(int j=0;j<array[0].size();j++)
    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：

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

(2) 二维数组

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

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：

```
#include <iostream>
#include <stdio.h>
using namespace std;

long int merge(int a[], int left, int mid, int right,int b[])
{
  int i = mid;
  int j = right;
  long int lcount = 0;
  while (i >= left && j > mid)
  {
    if(a[i] > (long long) 3 * a[j])
    {
      lcount += j - mid;
      i--;
    }
    else
    {
      j--;
    }
  }
  i = mid;
  j = right;
  int k = 0;
  while (i >= left && j > mid)
  {
    if(a[i] > a[j])
    {
      b[k++] = a[i--];
    }
    else
    {
      b[k++] = a[j--];
    }
  }
```

```
  while (i >= left)
  {
    b[k++] = a[i--];
  }
  while (j > mid)
  {
    b[k++] = a[j--];
  }
  for (i = 0; i < k; i++)
  {
    a[right - i] = b[i];
  }
  return lcount;
}

long int solve(int a[],int left, int right,int b[])
{
  long int cnt = 0;
  if(right > left)
  {
    int mid = (right+left) / 2;
    cnt += solve(a,left, mid,b);
    cnt += solve(a,mid + 1, right,b);
    cnt += merge(a,left, mid, right,b);
  }
  return cnt;
}

long int InversePairs(int a[],int len)
{
  int *b=new int[len];
  long int count=solve(a,0,len-1,b);
  delete[] b;
  return count;
}

int main()
{
  long int n;//数组维度
  int *array;//数组
  scanf("%d", &n);
  array  = new int[n];
  for(long int i=0;i<n;i++)
    scanf("%d", &array[i]);

  long int count = InversePairs(array, n);
  printf("%d", count);
  return 0;
}
```

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)。

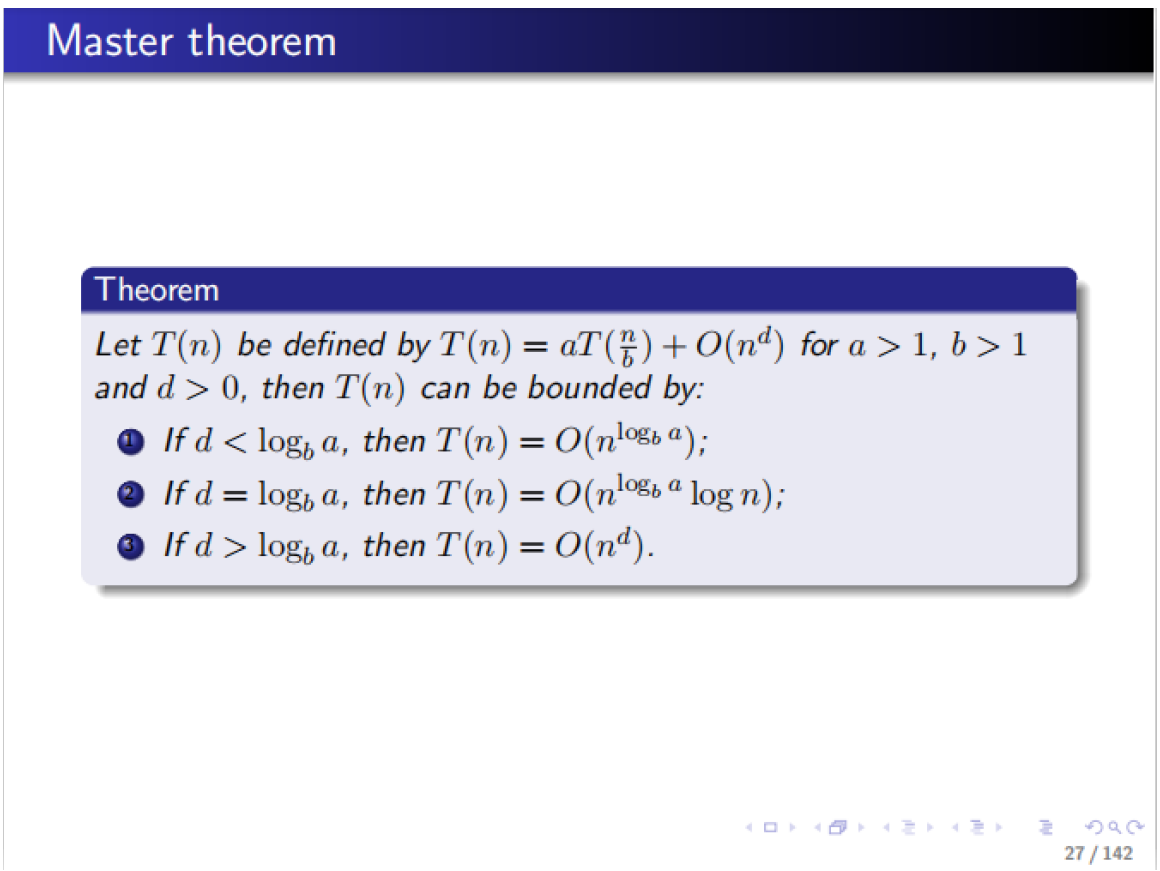


图 3: Master theorem

计算该递推式：

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

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：

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

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

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

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 $O(nlogn)$，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)

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 $A_1, A_2, A_3, ..., A_n$ 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：动态规划求解，即采用多步决策，设 $OPT(i,j)$ 表示 $A_iA_{i+1}...A_{j-1}A_j$ 的最优运算次数。假设从第 k 个位置一分为二，即 $(A_i, A_{i+1}, A_{i+2}, ..., A_k)(A_{k+1}, A_{k+2}, ..., A_j)$，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)
if i == j then
    return 0;
end if
OPT(i, j) = +∞;
for k = i to j − 1 do
    q = RECURSIVE_MATRIX_CHAIN(i, k)
        + RECURSIVE_MATRIX_CHAIN(k + 1, j)
        +p_{i−1} p_k p_j;
    if q < OPT(i, j) then
        OPT(i, j) = q;
    end if
end for
return OPT(i, j);
```

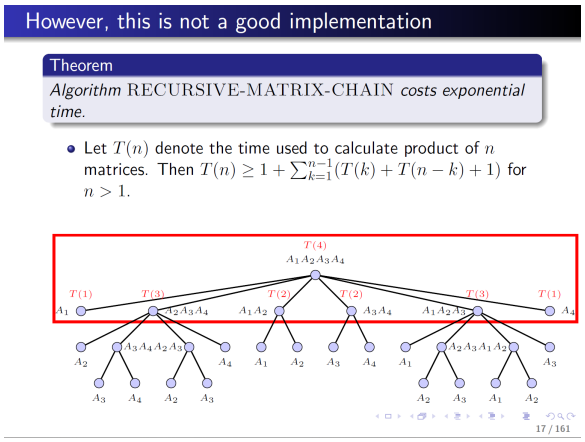
- Note: The optimal solution to the original problem can be obtained through calling RECURSIVE_MATRIX_CHAIN$(1, n)$.

15 / 161

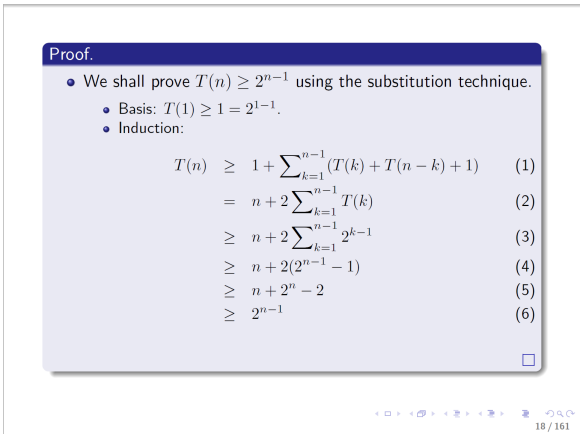
图 4: trial 1

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 $O(n^3)$。



(a) 已知的条件



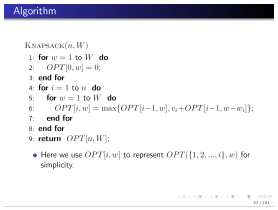
(b) 数学归纳法证明

图 5: 指数级时间复杂度证明

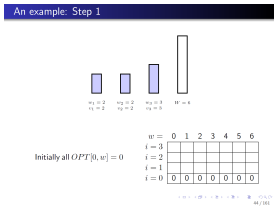
Question 14

背包问题。

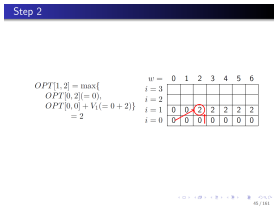
假设当包空 W，前 i 个物品的最优价值为 $OPT(i, W)$，则有递推式：$OPT(i, W) = max\{OPT(i-1, W), OPT(i-1, W-w_i) + v_i\}$，下面给出伪代码及某个背包问题的示例：



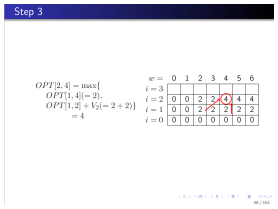
(a) 伪代码



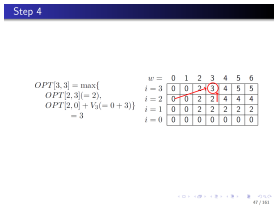
(b) Step1



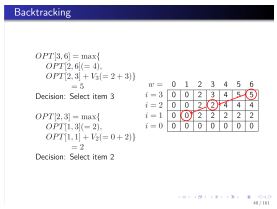
(c) Step2



(d) Step3



(e) Step4



(f) Step5

图 6: 二路归并过程

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)

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

```

```
using namespace std;

int main()
{
  int k,W;//物品的数量和包的总容量
  cin>>k>>W;//输入i和W
  int w[k] = {0};//第i个位置代表第i+1个物品的重量w(i+1)
  int v[k] = {0};//第i个位置代表第i+1个物品的价值v(i+1)
  int OPT[k+1][W+1];
  memset(OPT, 0, sizeof(OPT));//在头文件#include<string.h>中

  for(int i=0;i<k;i++)
  {
    scanf("%d",&w[i]);//输入各个物品的重量
  }
  for(int i=0;i<k;i++)
  {
    scanf("%d",&v[i]);//输入各个物品的价值
  }

  //动态规划求解，实际上在计算一个二维矩阵
  for(int i =1;i<=k;i++)
  {
    for(int j = 1;j<=W;j++)
    {
      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
        OPT[i][j] = OPT[i-1][j];
      else
        OPT[i][j] = max(OPT[i-1][j],OPT[i-1][j-w[i-1]]+v[i-1]);
    }
  }
  int O = OPT[k][W];
  for(int i = 0;i<k+1;i++)
  {
    for(int j = 0;j<W+1;j++)
    {
      cout<<OPT[i][j]<<"  ";
    }
    cout<<endl;
  }
  cout<<O<<endl;

  return 0;
}
```

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)：

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

using namespace std;
```

```
int main()
{
  int k,M,N;//任务的数量和服务器的空间和内存
  cin>>k>>M>>N;//输入i和M,N
  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)
  int n[k] = {0};//第i个位置代表第i+1个任务需求的内存(i+1)
  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)
  int OPT[k+1][M+1][N+1];//三维数组
  memset(OPT, 0, sizeof(OPT));//在头文件#include<string.h>中

  for(int i=0;i<k;i++)
  {
    scanf("%d",&m[i]);//输入各个任务的空间
  }
  for(int i=0;i<k;i++)
  {
    scanf("%d",&n[i]);//输入各个任务的内存
  }
  for(int i=0;i<k;i++)
  {
    scanf("%d",&u[i]);//输入各个任务可以服务的人数
  }

  //动态规划求解，实际上在计算一个三维矩阵
  for(int i =1;i<=k;i++)
  {
    for(int j = 1;j<=M;j++)
    {
      for(int s = 1;s<=N;s++)
      {
        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
          OPT[i][j][s] = OPT[i-1][j][s];
        else
          OPT[i][j][s] = max(OPT[i-1][j][s],OPT[i-1][j-m[i-1]][s-n[i-1]]+u[i-1]);
      }
    }
  }
  int O = OPT[k][M][N];
  cout<<O<<endl;
  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。

递归公式及伪代码如下：

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。

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
```

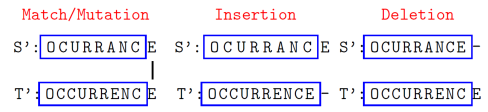
$s(T', S') = 1 - 3 + 1 + 1 + 1 + 1 - 3 - 3 + 1 + 1 + 1 = -1$

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

(a) 示例

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



(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

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:

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

(a) 递归公式

Needleman-Wunsch algorithm [1970]

NEEDLEMAN-WUNSCH(T, S)

```
for i = 0 to m do
    OPT[i, 0] = -3 * i;
end for
for j = 0 to n do
    OPT[0, j] = -3 * j;
end for
for j = 1 to n do
    for i = 1 to m do
        OPT[i, j] = max{OPT[i-1, j-1] + s(T_i, S_j), OPT[i-1, j] - 3, OPT[i, j-1] - 3};
    end for
end for
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.

(b) 伪代码

图 8: alignment 递归公式及伪代码

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: OPT("OC", "OCUR") = $\max \begin{cases} \text{OPT("O","OCUR")} & -3 & (=-11) \\ \text{OPT("O","OCU")} & -1 & (=-6) \\ \text{OPT("OC","OCU")} & -3 & (=-4) \end{cases}$

Alignment: S'= OCUR
T'= 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'= O-CURRANCE
T'= OCCURRENCE

(b) 回溯路线

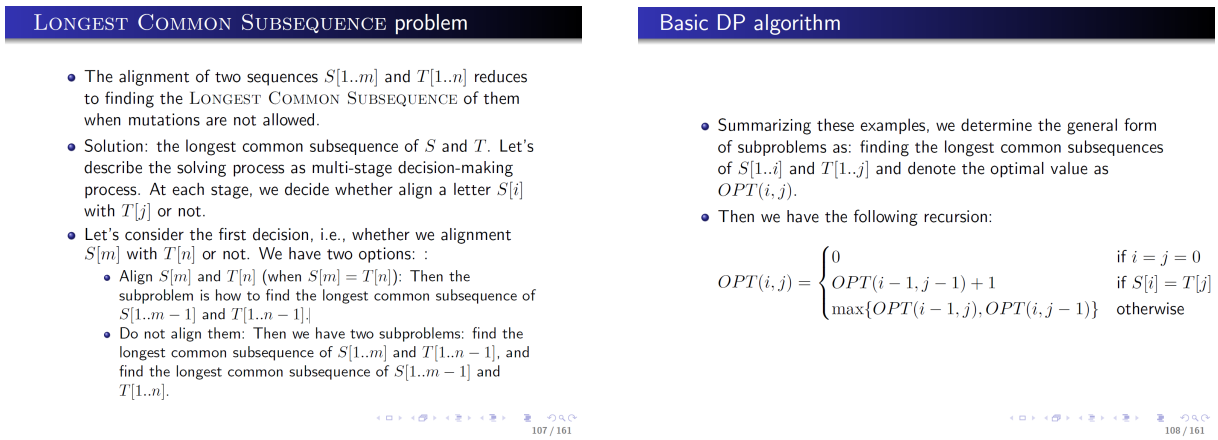
图 9: 二维矩阵及回溯路线

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：



(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；
(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 $code/LongestCommonSubsequence$)：

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

```

using namespace std;

int main()
{
  string str1,str2;
  getline(cin,str1);
  getline(cin,str2);
  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
  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(str1[i-1]==str2[j-1])
      {
        OPT[i][j] = OPT[i-1][j-1] + 1;
      }
      if(str1[i-1]!=str2[j-1])
      {
        OPT[i][j] = max(OPT[i-1][j],OPT[i][j-1]);
      }
    }
  }
  for(int i=0;i<m+1;i++)
  {
    for(int j=0;j<n+1;j++)
    {
      cout<<OPT[i][j]<<" ";
    }
    cout<<endl;
  }
  printf("%d",OPT[m][n]);

  return 0;
}
```

上面的代码实际上是在打印了这样一个二维矩阵：

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 0J 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++)
    {
```

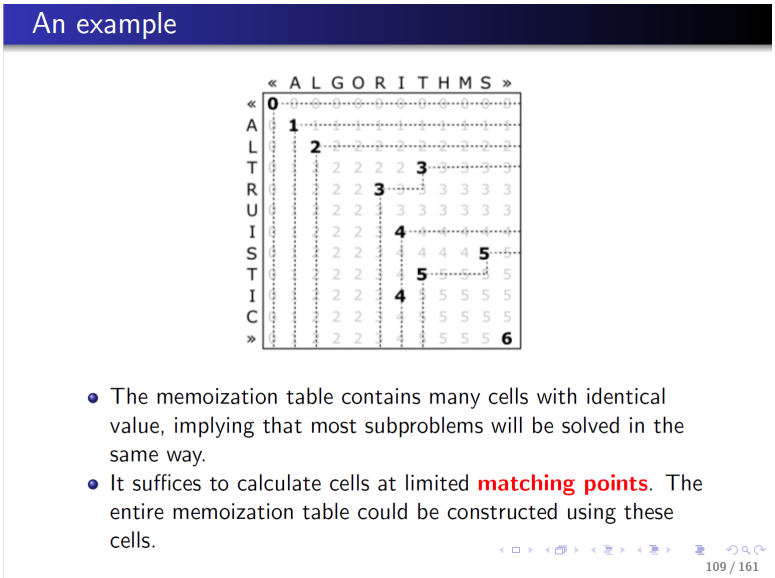


图 11: 最长公共子序列的动态决策过程

```
        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]);
        }
      }
    }
    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。观察图 7，可以看到

Question 18

第 16 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，实际上可以只计算变化的点处的值。

Question 100

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

Part III
机器学习与深度学习基础模型与算法

Question 1

手推 BP(Back Propagation) 算法。

Part IV
视频分析与处理相关知识点