

数学相关

机器学习题

半监督学习

生成模型和判别模型

LR 模型

最大熵

信息增益

梯度下降

交叉熵函数的求导

softmax 函数求导

sigmoid 函数与 softmax 函数的联系

SVM 相关的问题

**介绍 SVM**

比较熟悉的机器学习模型

为什么正则化可以防止过拟合

常见的损失函数

常见的优化算法

Batch Normalization

LSTM

LSTM 和 GRU 区别

VAE

Attention

Transformer

Dropout

随机森林

Adaboost

GBDT

XGBoost

自然语言处理

文本分类

文本相似度

阅读理解

序列标注

对话系统

概率题

富翁和乞丐的财富之旅

硬币质量问题

蓄水池采样

吃月饼

红球绿球

几何分布

乘法原理问题

编程题

完整的c++程序

二分查找

排序算法

快速排序

- 堆排序
- 单链表排序
- 二叉树
  - 二叉搜索树第k大
- 链表
  - 移除倒数第N个节点
  - 链表有环问题
  - 二叉树转链表
- DFS
- BFS
- 拓扑序
- 求幂
- 两个有序数组第k大
- 包含k个数组的至少一个元素
- 字符串
  - 回文字符串
  - 句子和词典
- 位运算
  - 位运算处理数组中的数

## 数学相关

---

[如何理解特征值和特征向量](#)

[主成分分析原理详解](#)


## 机器学习题

---

### 半监督学习

[半监督学习](#)

### 生成模型和判别模型

所谓判别模型(Discriminative Model)，也就是说模型可以直接用来判别事物的。这里所说的判别事物，最典型的就是做分类，当然还有做序列标注。既然直接可以用来分类，也就是说我们可以在已知属性的条件下，对该记录进行判断。所以，判别模型是对条件概率进行的建模，也就是  $p(y|x)$ 。这里  $x$  就是属性集合，实际上就是一个向量；而  $y$  则可能是一个值（此时对应分类问题），可能是一个向量（此时对应序列标注问题）。针对分类问题，最典型的判别模型就属 `logistic regression` 了；而序列标注问题的典型判别模型则是 `CRF` (条件随机场)。

下面说一说生成模型(Generative Model)。生成模型说的是，这个模型可以描述数据的生成过程。换句话说，已知了这个模型，我们就可以产生该模型描述的数据。而数据由两部分组成，也就是  $(x, y)$ ，前者是特征，后者则是类别（ $y$  是标量）或者序列类别（ $y$  是向量）。要描述整个数据，也就是要对  $p(x, y)$  进行建模，所以是对联合概率进行建模。有了联合概率，我们就可以模拟产生数据样本了。生成模型本身不是做分类或者序列标注的，但是可以用来解决这些问题。最典型的使用生成模型解决分类的模型要数 `naive bayes` 了，它是对  $p(x, y)$  进行的建模的。

要分类，就要计算  $p(y|x)$ ，所以使用贝叶斯定理做转换，变成  $p(x,y)/p(x)$ 。然后  $p(x)$  看成常数，实际上该模型最终还是归结到计算  $p(x,y)$  上去了。类似的，在序列标注问题中，使用的最著名的生成模型就是 HMM 了。本质上和 naive bayes 是一样的，不过  $y$  变成多维的了。

看待生成模型，思路不要局限在分类（或者是序列标注）上面。前面说到，生成模型是可以用来描述数据产生的过程的（这一点跟判别模型很不一样！）。虽然 naive bayes 和 HMM 用在标注（分类或者序列标注）上，但还有很多很多的生成模型不是这样的用途，最典型的就是著名的 LDA 了（以及一大坨的概率图模型(PGM)），它描述了文档集合数据的产生过程，可以用来做 unsupervised learning。

小结：判别模型是对条件概率建模，生成模型是对联合概率建模。判别模型一般就是直接解决分类或者序列标注问题（毕竟就是用来判定类别的嘛），而生成模型用来描述数据的产生过程，但某些判别模型通过一些变化，是可以用来解决分类（或者序列标注）问题的。

## LR 模型

[LR与SVM的异同](#)

## 最大熵

## 信息增益

熵表示随机变量不确定性的度量。

什么是信息增益 当得知特征X的信息使得信息的不确定性减少的程度。

## 梯度下降

为什么梯度反方向是局部下降最快的方向

梯度：对多元函数的各个自变量求偏导数，如函数  $\phi = 2x + 3y^2 - \sin(z)$  的梯度是  $\nabla\phi = (\frac{\partial\phi}{\partial x}, \frac{\partial\phi}{\partial y}, \frac{\partial\phi}{\partial z}) = (2, 6y, -\cos(z))$ 。梯度是一个向量，梯度的方向是函数  $\phi$  增长最快的方向，梯度的反方向是函数  $\phi$  降低最快的方向。

梯度是导数的高维形式，导数是增长的方向，那么，梯度的反方向就是增长的反方向，下降。

## 交叉熵函数的求导

$$\begin{aligned}
 \sigma(z) &= \sigma(w^T x + b) = \frac{1}{1 + e^{-(w^T x + b)}} \\
 J &= -\frac{1}{n} \sum [y \cdot \log \sigma(z) + (1-y)(1-\sigma(z))] \\
 \frac{\partial J}{\partial w_j} &= -\frac{1}{n} \sum \left[ \frac{y}{\sigma(z)} + \frac{-(1-y)}{1-\sigma(z)} \right] \cdot \sigma'(z) \cdot x_j \\
 &= -\frac{1}{n} \sum \frac{y - \sigma(z)}{\sigma(z)(1-\sigma(z))} \cdot \sigma'(z) x_j \\
 \because \sigma'(z) &= \sigma(z)(1-\sigma(z)) \\
 \therefore \frac{\partial J}{\partial w_j} &= -\frac{1}{n} \sum (y - \sigma(z)) \cdot x_j \\
 &= \frac{1}{n} \sum [\sigma(z) - y] x_j
 \end{aligned}$$

为什么交叉熵效果要好？可以看到，求导之后的形式，导数的变化与误差保持一致，所以能够使得训练的速度变快。

## softmax 函数求导

[softmax的log似然代价函数（公式求导）](#)

## sigmoid 函数与 softmax 函数的联系

[softmax](#) 回归是 sigmoid 回归的推广形式。

logistic regression:  $h(x) = \frac{1}{1+e^{-\theta^T x}}$

Softmax regression:  $h(x) = \frac{e^{\theta_1^T x}}{\sum_{j=1}^K e^{\theta_j^T x}}$

二分类时:  $h(x) = \frac{1}{e^{\theta_1^T x} + e^{\theta_2^T x}} \begin{bmatrix} e^{\theta_1^T x} \\ e^{\theta_2^T x} \end{bmatrix} = \frac{1}{1+e^{(\theta_2-\theta_1)^T x}} \begin{bmatrix} 1 \\ e^{(\theta_2-\theta_1)^T x} \end{bmatrix}$

$= \begin{bmatrix} \frac{1}{1+e^{\theta^T x}} \\ 1 - \frac{1}{1+e^{\theta^T x}} \end{bmatrix}$

## SVM 相关的问题

SVM的常见核函数及其选取？

- 线性核函数
- 多项式核函数
- 高斯核函数 (RBF)
- Sigmoid 核函数

[SVM 的核函数如何选取](#)

## 介绍 SVM

SVM 的基本模型是在特征空间上，找到最佳的分离超平面，使得训练集上正负样本间隔最大的线性分类器，引入了核方法之后可以用来解决非线性问题。通常SVM分为三种：

- 硬间隔 SVM：数据线性可分，硬间隔最大化；
- 软间隔 SVM：数据接近线性可分，软间隔最大化；
- 非线性 SVM：数据非线性可分，核方法和软间隔最大化。

**支持向量** 在线性可分的情况下，样本点中与分离超平面距离最近的样本实例称为支持向量；

## 比较熟悉的机器学习模型

## 为什么正则化可以防止过拟合

## 常见的损失函数

LR：交叉熵函数

SVM：合页损失函数



# 常见的优化算法

## 简单认识Adam优化器

### An overview of gradient descent optimization algorithms

1. SGD : 每次朝着梯度的反方向进行更新。
2. Momentum: 动能积累, 每次更新时候积累 SGD 的更新方向, 也就是每次参数更新时, 不单单考虑当前的梯度方向, 还考虑之前的梯度方向, 当然之前的梯度会有一个衰减因子。
3. Adagrad: 自动调节学习率大小, 对于频繁更新的参数, 学习率小; 对于很少更新的参数, 学习率大。怎么判断参数频繁更新呢? 可以为更新公式设置一个分母: 梯度平方累加和的平方根。这样当分母越大, 说明之前该参数越频繁更新, 反之则很少更新。存在问题: 当迭代次数多了之后, 由于是累加操作, 分母越来越大, 学习率会变得非常小, 模型更新会很慢。
4. Adadelta: 作为 Adagrad 的拓展, 是解决学习率消失的问题。在这里不会直接叠加之前所有的梯度平方, 而是引入了一个梯度衰退的因子, 使得时间久远的梯度对此刻参数更新的影响会消失。RMSProp 的思想和其类似。
5. Adam: 结合 Momentum 和 Adadelta 两种优化算法的优点。对梯度的一阶矩估计 (First Moment Estimation, 即梯度的均值) 和二阶矩估计 (Second Moment Estimation, 即梯度的未中心化的方差) 进行综合考虑, 计算出更新步长。

## Batch Normalization

机器学习领域有个很重要的假设: IID独立同分布假设, 就是假设训练数据和测试数据是满足相同分布的, 这是通过训练数据获得的模型能够在测试集获得好的效果的一个基本保障。那BatchNorm的作用是什么呢? BatchNorm就是在深度神经网络训练过程中使得每一层神经网络的输入保持相同分布的。[深入理解Batch Normalization批标准化](#)

模型比较深入之后, 输入数据经过很多层的变换之后, 分布发生了很大的变化, 理论上我们要求数据是独立同分布的, 但是比较深的层它的输入随着前面层参数的更新, 每次的输入分布都不同, 也就是所谓的内部协变量偏移, 为了缓解这个问题, BN通过在每个batch进行减均值除方差的操作, 相当于做一个归一化操作, 把每层的输入分布拉到一个0 均值1方差分布中, 缓解内部协变量偏移的问题, 又因为这个操作会损失我们前面的层学到的很多信息, 为了减小这个损失, 我们再对前面的结果做一个线性变换, 让它能恢复一部分前面的信息

### Batch Normalizing Transform [\[ edit \]](#)

In a neural network, batch normalization is achieved through a normalization step that fixes the means and variances of each layer's inputs. Ideally, the normalization would be conducted over the entire training set, but to use this step jointly with [stochastic optimization](#) methods, it is impractical to use the global information. Thus, normalization is restrained to each mini-batch in the training process.

Denote a mini-batch of the entire training set of size  $m$  as  $B$ . The mean and [variance](#) of  $B$  could thus be denoted as

$$\mu_B = \frac{1}{m} \sum_{i=1}^m x_i, \text{ and } \sigma_B^2 = \frac{1}{m} \sum_{i=1}^m (x_i - \mu_B)^2.$$

For a layer of the network with  $d$ -dimensional input,  $x = (x^{(1)}, \dots, x^{(d)})$ , each dimension of its input is then normalized separately,

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_B^{(k)2} + \epsilon}}, \text{ where } k \in [1, d] \text{ and } i \in [1, m]; \mu_B^{(k)} \text{ and } \sigma_B^{(k)2} \text{ are the per-dimension mean and variance, respectively.}$$

$\epsilon$  is added in the denominator for numerical stability and is an arbitrarily small constant. The resulting normalized activation  $\hat{x}^{(k)}$  have zero mean and unit variance, if  $\epsilon$  is not taken into account. To restore the representation power of the network, a transformation step then follows as

$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)},$$

where the parameters  $\gamma^{(k)}$  and  $\beta^{(k)}$  are subsequently learnt in the optimization process.

Formally, the transform  $BN_{\gamma^{(k)}, \beta^{(k)}} : x_{1..m}^{(k)} \rightarrow y_{1..m}^{(k)}$  is denoted as the Batch Normalizing Transform. The output of the BN transform  $y^{(k)} = BN_{\gamma^{(k)}, \beta^{(k)}}(x^{(k)})$  is then passed to other network layers, while the normalized output  $\hat{x}_i^{(k)}$  remains internal to the current layer.

## LSTM

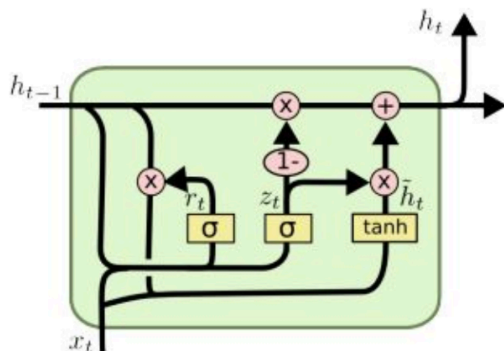
### [LSTM模型与前向反向传播算法](#)

[LSTM如何来避免梯度弥散和梯度爆炸?](#)

## LSTM 和 GRU 区别

LSTM 输入门, 遗忘门, 输出门·

GRU 更新门, 重置门



$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t])$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

$$\tilde{h}_t = \tanh(W \cdot [r_t * h_{t-1}, x_t])$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

## VAE

[AE、VAE、CVAE解读](#)

## Attention

什么是attention?

对于序列, attention能够对给不同位置赋予不同的权重, 从而能够得到模型当前最关心的部分。这个权重是通过矩阵乘法来得到的。

## Transformer

[The Transformer – Attention is all you need](#)

## Dropout

DROPOUT

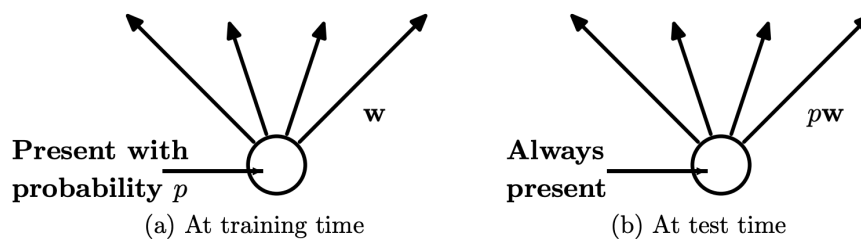


Figure 2: **Left:** A unit at training time that is present with probability  $p$  and is connected to units in the next layer with weights  $\mathbf{w}$ . **Right:** At test time, the unit is always present and the weights are multiplied by  $p$ . The output at test time is same as the expected output at training time.

## 随机森林

随机森林 (Random Forest, 简称 RF) 在以决策树为基学习器 Bagging 集成的基础上, 进一步在决策树的训练过程中引入了随机属性选择。具体来说, 传统决策树在选择划分属性时是在当前结点的属性集合 (假定有  $d$  个属性) 中选择一个最优属性; 而在 RF 中, 对基决策树的每个结点, 先从该结点的属性集合中随机选择一个包含  $k$  个属性的子集, 然后再从这个子集中选择一个最优属性进行划分。一般情况下, 推荐  $k = \log_2 d$ 。

可以看出, RF 对 Bagging 只做了小改动, Bagging 是通过自采样, 采样出  $T$  个包含  $m$  个训练样本的采样集, 对每个采样集训练一个基学习器。它通常对分类任务使用简单投票法, 对回归任务使用简单平均法。与 Bagging 中基学习器的多样性仅通过样本扰动相比, RF 的多样性还来自于属性扰动, 这使得最终集成的泛化性能可通过个体学习器之间差异度的增加而进一步提升。

( Bagging 是 bootstrap aggregating 的缩写, 有放回抽样; 无放回抽样是 pasting。)

## Adaboost

1. 提高被前一轮分类器错误分类样本的权值, 从而在新一轮的训练中收到更多的关注;
2. 增大分类误差率小的弱分类器的权值, 使其在表决中起到较大的作用;

Adaboost 的巧妙之处就在于它将这些想法自然且有效地实现在一种算法中。从偏差-方差分解的角度来看, Bagging 主要关注降低方差, Boosting 主要关注降低偏差。

## GBDT

## XGBoost

## 自然语言处理

---

### 文本分类

### 文本相似度

### 阅读理解

### 序列标注

### 对话系统

## 概率题

---

### 富翁和乞丐的财富之旅

富翁和乞丐财富差距为  $z$ , 每天富翁得到1的概率为  $p$ , 乞丐得到1的概率为  $q$ , 求将来某一天持平的概率



## 硬币质量问题

一枚硬币扔10次，8次正面朝上，求这枚硬币有问题的概率？

依旧设硬币有问题概率为 $p$ ，有问题硬币正面朝上概率为 $q$ ，一枚硬币扔10次，8次正面朝上为事件 $A$ ，则

$$P(A|\text{coin}=\text{Normal})=C_{10}^8\left(\frac{1}{2}\right)^{10}$$

$$P(A|\text{coin}=\text{Abnormal})=C_{10}^8q^8(1-q)^2$$

$$P(A,\text{coin}=\text{Abnormal})=P(A|\text{coin}=\text{Abnormal})\cdot P(\text{coin}=\text{Abnormal})=C_{10}^8q^8(1-q)^2p$$

$$P(A)=P(A|\text{coin}=\text{Normal})\cdot P(\text{coin}=\text{Normal})+P(A|\text{coin}=\text{Abnormal})\cdot P(\text{coin}=\text{Abnormal})=C_{10}^8\left(\frac{1}{2}\right)^{10}(1-p)+C_{10}^8q^8(1-q)^2p$$

$$P(\text{coin}=\text{Abnormal}|A)=P(A,\text{coin}=\text{Abnormal})/P(A)=\frac{C_{10}^8q^8(1-q)^2p}{C_{10}^8\left(\frac{1}{2}\right)^{10}(1-p)+C_{10}^8q^8(1-q)^2p}=\frac{q^8(1-q)^2p}{\left(\frac{1}{2}\right)^{10}(1-p)+q^8(1-q)^2p}$$

这个要先统计有问题硬币占总硬币数目的比例，以及这种问题硬币投掷产生的偏差才能计算，其中 $q=0.5$ 时，也就是问题硬币表现得和普通硬币一致时，答案就是 $p$ ， $q<0.5$ 时，答案小于 $p$ ， $q=0.8$ 时， $q^8(1-q)^2$ 最大，在 $p$ 不变时，答案最大

## 蓄水池采样

蓄水池抽样：从 $N$ 个元素中随机的等概率的抽取 $k$ 个元素，其中 $N$ 无法确定

先给出代码：

```
Init : a reservoir with the size: k
for i= k + 1 to N
    M = random(1, i);
    if(M < k)
        SWAP the Mth value and ith value
end for
```

上述伪代码的意思是：先选中第1到 $k$ 个元素，作为被选中的元素。然后依次对第 $k+1$ 至第 $N$ 个元素做如下操作：

每个元素都有 $k/x$ 的概率被选中，然后等概率的 $(1/k)$ 替换掉被选中的元素。其中 $x$ 是元素的序号。

算法的成立是用数学归纳法证明的：

每次都是以  $k/i$  的概率来选择

例： $k=1000$ 的话，从1001开始作选择，1001被选中的概率是 $1000/1001$ ，1002被选中的概率是 $1000/1002$ ，与我们直觉是相符的。

接下来证明：

假设当前是 $i+1$ ，按照我们的规定， $i+1$ 这个元素被选中的概率是 $k/i+1$ ，也即第  $i+1$  这个元素在蓄水池中出现的概率是 $k/i+1$

此时考虑前 $i$ 个元素，如果前 $i$ 个元素出现在蓄水池中的概率都是 $k/i+1$ 的话，说明我们的算法是没有问题的。

对这个问题可以用归纳法来证明： $k < i \leq N$

1.当 $i=k+1$ 的时候，蓄水池的容量为 $k$ ，第 $k+1$ 个元素被选择的概率明显为 $k/(k+1)$ ，此时前 $k$ 个元素出现在蓄水池的概率为  $k/(k+1)$ ，很明显结论成立。

2.假设当  $j=i$  的时候结论成立，此时以  $k/i$  的概率来选择第 $i$ 个元素，前 $i-1$ 个元素出现在蓄水池的概率都为 $k/i$ 。

证明当 $j=i+1$ 的情况：

即需要证明当以  $k/i+1$  的概率来选择第 $i+1$ 个元素的时候，此时任一前 $i$ 个元素出现在蓄水池的概率都为  $k/(i+1)$ 。

前 $i$ 个元素出现在蓄水池的概率有2部分组成，①在第 $i+1$ 次选择前得出现在蓄水池中，②得保证第 $i+1$ 次选择的时候不被替换掉

①.由2知道在第 $i+1$ 次选择前，任一前 $i$ 个元素出现在蓄水池的概率都为 $k/i$

②.考虑被替换的概率：

首先要被替换得第  $i+1$  个元素被选中(不然不用替换了)概率为  $k/i+1$ ，其次是因为随机替换的池子中 $k$ 个元素中任意一个，所以不幸被替换的概率是  $1/k$ ，故

前 $i$ 个元素(池中元素)中任一被替换的概率 =  $k/(i+1) * 1/k = 1/i+1$

则(池中元素中)没有被替换的概率为： $1 - 1/(i+1) = i/i+1$

综合① ②,通过乘法规则

得到前 $i$ 个元素出现在蓄水池的概率为  $k/i * i/(i+1) = k/i+1$

故证明成立

## 吃月饼

月神特别喜欢吃月饼，中秋节时快手发了10个月饼，已知月神一天至少吃一个月饼；请问，月神在3天内将10个月饼全部吃完的概率为？

采用插板法，10个月饼排成一行，如果在2天内吃完，就在里面插入一个板子，所以是 $C_{91}$ ,所以就应该是 $(C_{90}+C_{91}+C_{92})/(C_{90}+C_{91}+C_{92}+C_{93}+C_{94}+C_{95}+.....+C_{99})=23/256$

## 红球绿球

一个箱子中有15%的红球和85%的绿球，小明随机取出1个球，他不能看到球，但他根据手感判断该球为红色。已知小明根据手感判断颜色的正确概率为80%，那么他取到的球实际为红色的概率为？

小明取到红球，判断为红球的概率： $0.15*0.8=0.12$

小明取到绿球，判断为红球的概率： $0.85*0.2=0.17$

以上两种情况为题设所给出的前提条件，则在此基础上，小明取到的球实际为红球的概率为：

$$0.12 / (0.12 + 0.17) \approx 0.41$$

## 几何分布

**Beta**星球非常重男轻女，一个家庭如果一胎生女儿的话，会继续生下一个孩子，直到生男孩为止。已知生男孩和女孩的概率都是50%，每个家庭至少会生一个孩子，那么Beta星球平均每个家庭的孩子数量为？

这是一个几何分布，几何分布的期望等于 $E(X)=1/p$ ，设家庭有k个孩子的概率为p，则 $p = (\frac{1}{2})^k$

$$E(X) = \sum_{k=1}^{\infty} k \times (\frac{1}{2})^k = 2$$

## 乘法原理问题

7个同学围坐一圈，要选2个不相邻的作为代表，有多少种不同的选法

先选1个同学出来，7种选法；再选第2个同学出来，4种选法。又因为第一次选a第二次选b，和第二次选b第一次选a，情况一样，是重复的。所以总共有： $7 * 4 / 2 = 14$ 种选法。

## 编程题

### 完整的c++程序

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

using namespace std;

int getNums(int n) {
    vector<int> dp(n + 1, 0);
    dp[0] = 1;
    for(int i = 1; i <= n; i++) {
        for(int j = 1; j <= i; j <= 1) {
            dp[i] += dp[i - j];
        }
    }
    return dp[n];
}

int main() {
    int m;
    cin >> m;
    for(int i = 0; i < m; i++) {
        int n;
        cin >> n;
        cout << getNums(n) << endl;
    }
    return 0;
}
```

# 二分查找

[你真的会写二分吗](#)

```
while(left < right) {
    int mid = left + (right - left) >> 1;
    if(vals[mid] < key) left = mid + 1;
    else right = mid;
}
return left;
```

# 排序算法

算法	最好时间	最坏时间	平均时间	额外空间	稳定性
选择	$n^2$	$n^2$	$n^2$	1	稳定
冒泡	$n$	$n^2$	$n^2$	1	不稳定
插入	$n$	$n^2$	$n^2$	1	稳定
希尔	$n$	$n^2$	$n^{1.3}$ (不确定)	1	不稳定
归并	$n\log_2n$	$n\log_2n$	$n\log_2n$	$n$	稳定
快排	$n\log_2n$	$n^2$	$n\log_2n$	$\log_2n$ 至 $n$	不稳定
堆	$n\log_2n$	$n\log_2n$	$n\log_2n$	1	不稳定
基数	$n*k$	$n*k$	$n*k$	$n+k$	稳定

[十大排序算法](#)

[排序算法的复杂度、实现和稳定性](#)

# 快速排序

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        quickSort(nums, 0, nums.size() - 1);
    }

private:
    void quickSort(vector<int>& nums, int left, int right) {
        if(left >= right) return ;
    }
};
```

```

        int mid = partition(nums, left, right);
        quickSort(nums, left, mid - 1);
        quickSort(nums, mid + 1, right);
    }

    int partition(vector<int>& nums, int left, int right) {
        int key = nums[left];
        while(left < right) {
            while(left < right && nums[right] >= key) right--;
            nums[left] = nums[right];
            while(left < right && nums[left] < key) left++;
            nums[right] = nums[left];
        }
        nums[left] = key;
        return left;
    }
};

```

## 堆排序

```

class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        int n = nums.size();
        for(int i = n / 2; i >= 0; i--) {
            adjustHeap(nums, i, n);
        }
        for(int i = 0; i < k; i++) {
            swap(nums[0], nums[n - i - 1]);
            adjustHeap(nums, 0, n - i - 1);
        }
        return nums[n - k];
    }

private:
    void adjustHeap(vector<int>& nums, int pos, int size) {
        int left = pos * 2;
        int right = pos * 2 + 1;
        if(left < size && nums[pos] < nums[left]) {
            swap(nums[pos], nums[left]);
            adjustHeap(nums, left, size);
        }
        if(right < size && nums[pos] < nums[right]) {
            swap(nums[pos], nums[right]);
            adjustHeap(nums, right, size);
        }
    }
};

```

## 单链表排序

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        ListNode* root = merge_sort(head);
        return root;
    }

private:
    ListNode* merge_sort(ListNode* head) {
        if(!head || !head->next) {
            return head;
        }
        ListNode *slow, *fast;
        slow = head, fast = head->next;
        while(fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
        }
        ListNode *heada = head, *headb = slow->next;
        slow->next = NULL;
        return merge_list(merge_sort(heada), merge_sort(headb));
    }

    ListNode* merge_list(ListNode* heada, ListNode *headb) {
        ListNode *res = new ListNode(0);
        ListNode *p = res;
        while(heada || headb) {
            if(heada && (!headb || heada->val < headb->val)) {
                res->next = new ListNode(heada->val);
                res = res->next;
                heada = heada->next;
            }
            if(headb && (!heada || heada->val >= headb->val)) {
                res->next = new ListNode(headb->val);
                res = res->next;
                headb = headb->next;
            }
        }
        return p->next;
    }
};
```

## 二叉树

### 二叉搜索树第k大



leetcode 230 二叉搜索树第k大

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {
        stack<TreeNode*> st;
        while(root || !st.empty()) {
            while(root) {
                st.push(root);
                root = root->left;
            }
            root = st.top();
            if(--k == 0) {
                return root->val;
            }
            st.pop();
            root = root->right;
        }
        return -1;
    }
};
```

## 链表

### 移除倒数第N个节点

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode *fast = head, *slow = head;
        while(n--){
            fast = fast->next;
        }
        if(fast == NULL){
            head = head->next;
            return head;
        }
        while(fast->next != NULL){
            fast = fast->next;
            slow = slow->next;
        }
        slow->next = slow->next->next;
        return head;
    }
};
```

### 链表有环问题

## 判断一个单链表是否存在环型链接并找出环的开始节点

### 1. 判断链表是否有环

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;
        while(slow && fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast) {
                return true;
            }
        }
        return false;
    }
};
```

### 2. 判断链表是否有环，如果有环，求环的第一个节点

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;
        while(slow && fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast) {
                ListNode *p = head, *q = slow;
                while(p && q) {
                    if(p == q) {
                        return p;
                    }
                    p = p->next;
                }
            }
        }
        return NULL;
    }
};
```

```

        q = q->next;
    }
}
}
return NULL;
}
};

```

## 二叉树转链表

```

leetcode 114
class Solution {
public:
    void flatten(TreeNode* root) {
        if(!root) return ;
        flatten(root->left);
        flatten(root->right);
        TreeNode* tmp = root->right;
        root->right = root->left;
        root->left = NULL;
        while(root->right) {
            root = root->right;
        }
        root->right = tmp;
    }
};

```

## DFS

[leetcode 79](#)

```

class Solution {
public:
    bool exist(vector<vector<char>>& board, string word) {
        for(int i = 0; i < board.size(); i++) {
            for(int j = 0; j < board[0].size(); j++) {
                if(dfs(board, word, i, j))
                    return true;
            }
        }
        return false;
    }

private:
    bool dfs(vector<vector<char>>& board, string word, int x, int y) {
        if(word.empty()) return true;
        int row = board.size(), col = board[0].size();
    }
};

```

```

        if(x < 0 || y < 0 || x >= row || y >= col || board[x][y] != word[0])
return false;
        char ch = board[x][y];
        board[x][y] = '_';
        word = word.substr(1);
        bool ret = (dfs(board, word, x - 1, y)
                    || dfs(board, word, x + 1, y)
                    || dfs(board, word, x, y - 1)
                    || dfs(board, word, x, y + 1));
        board[x][y] = ch;
        return ret;
    }
};

```

## BFS

[leetcode 1091](#)

```

class Solution {
public:
    int shortestPathBinaryMatrix(vector<vector<int>>& grid) {
        int n = grid.size();
        if(grid[0][0] == 1 || grid[n - 1][n - 1] == 1) {
            return -1;
        }

        queue<vector<int>> q;
        q.push({0, 0, 1});
        while(!q.empty()) {
            vector<int> node = q.front();
            q.pop();
            for(int i = -1; i <= 1; i++) {
                for(int j = -1; j <= 1; j++) {
                    int x = node[0] + i, y = node[1] + j;
                    if(x < 0 || x >= n || y < 0 || y >= n
                       || (i == 0 && j == 0) || grid[x][y] == 1) {
                        continue;
                    }
                    if(x == n - 1 && y == n - 1) {
                        return node[2] + 1;
                    }
                    q.push({x, y, node[2] + 1});
                    grid[x][y] = 1;
                }
            }
        }
        return -1;
    }
};

```

# 拓扑序

## 拓扑排序

# 求幂

```
1. leetcode 50 pow
class Solution {
public:
    double myPow(double x, int n) {
        if(n == 0.0)
            return 1.0;
        long t = n;
        if(n < 0) {
            x = 1 / x;
            t = -t;
        }
        double res = 1.0;
        while(t) {
            if(t & 1) {
                res *= x;
            }
            t >>= 1;
            x *= x;
        }
        return res;
    }
};

2. leetcode 372 super pow
class Solution {
public:
    int superPow(int a, vector<int>& b) {
        if(b.empty()) return 1;
        int t = b.back();
        b.pop_back();
        return pow(superPow(a, b), 10) * pow(a, t) % base;
    }

private:
    const int base = 1337;
    int pow(int a, int k) {
        a %= base;
        int res = 1;
        while(k) {
            if(k & 1) res = (res * a) % base;
            a = (a * a) % base;
            k >>= 1;
        }
    }
};
```

```

    }
    return res;
}
};

```

## 两个有序数组第k大

```

leetcode 4
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        int n1 = nums1.size(), n2 = nums2.size();
        return (findKth(nums1, nums2, (n1 + n2 + 1) >> 1) +
                findKth(nums1, nums2, (n1 + n2 + 2) >> 1)) / 2.0;
    }

private:
    int findKth(vector<int> nums1, vector<int> nums2, int k) {
        int n1 = nums1.size(), n2 = nums2.size();
        if(n1 == 0) return nums2[k - 1];
        if(n2 == 0) return nums1[k - 1];
        if(k == 1) return min(nums1[0], nums2[0]);
        int mid1 = min(n1, k / 2), mid2 = min(n2, k / 2);
        if(nums1[mid1 - 1] <= nums2[mid2 - 1])
            return findKth(vector<int> (nums1.begin() + mid1, nums1.end()),
                           nums2, k - mid1);
        else
            return findKth(nums1, vector<int> (nums2.begin() + mid2,
                                                nums2.end()), k - mid2);
    }
};

```

## 包含k个数组的至少一个元素

```

leetcode 632
class Solution {
public:
    vector<int> smallestRange(vector<vector<int>>& nums) {
        int n = nums.size();
        vector<int> res{0, INT_MAX};
        vector<pair<int, int>> total;
        for(int i = 0; i < n; i++) {
            for(auto v : nums[i]) {
                total.push_back(make_pair(v, i));
            }
        }
    }
};

```



```

    }
    sort(total.begin(), total.end());
    unordered_map<int, int> mm;
    int m = total.size();
    int count = 0;
    int left = 0, right = 0;
    for(; right < m; right++) {
        mm[total[right].second]++;
        if(mm[total[right].second] == 1) count++;
        while(count == n) {
            if(total[right].first - total[left].first < res[1] - res[0]) {
                res = {total[left].first, total[right].first};
            }
            if(mm[total[left].second] == 1) count--;
            mm[total[left].second]--;
            left++;
        }
    }
    return res;
}
};

```

## 字符串

### 回文字符串

```

1. leetcode 5 最长连续回文串 DP
class Solution {
public:
    string longestPalindrome(string s) {
        if(s.empty()) return "";
        int n = s.size();
        vector<vector<int>> dp(n, vector<int> (n, 0));
        string longString = s.substr(0, 1);
        for(int i = 0; i < n; i++) dp[i][i] = 1;
        for(int i = n - 1; i >= 0; i--) {
            for(int j = i + 1; j < n; j++) {
                if(s[i] == s[j] && (dp[i + 1][j - 1] > 0 || j == i + 1)) {
                    dp[i][j] = dp[i + 1][j - 1] + 2;
                    if(j - i + 1 > longString.size()) {
                        longString = s.substr(i, j - i + 1);
                    }
                }
            }
        }
        return longString;
    }
};

```

```
};
```

## 2. leetcode 647 计算回文串数目

```
class Solution {
public:
    int countSubstrings(string s) {
        if(s.empty()) return 0;
        int n = s.size();
        vector<vector<bool>> dp(n, vector<bool> (n, false));
        for(int i = 0; i < n; i++)
            dp[i][i] = true;
        for(int i = n - 1; i >= 0; i--) {
            for(int j = i + 1; j < n; j++) {
                if(s[i] == s[j]) {
                    dp[i][j] = dp[i + 1][j - 1];
                    if(i + 1 == j)
                        dp[i][j] = true;
                }
            }
        }
        int res = 0;
        for(int i = 0; i < n; i++) {
            for(int j = i; j < n; j++) {
                if(dp[i][j])
                    res++;
            }
        }
        return res;
    }
};
```

## 3. leetcode 516 最长回文子串

```
class Solution {
public:
    int longestPalindromeSubseq(string s) {
        int n = s.size();
        vector<vector<int>> dp(n, vector<int> (n, 0));
        for(int i = 0; i < n; i++) {
            dp[i][i] = 1;
        }
        for(int i = n - 1; i >= 0; i--) {
            for(int j = i + 1; j < n; j++) {
                if(s[i] == s[j]) dp[i][j] = dp[i + 1][j - 1] + 2;
                else dp[i][j] = max(dp[i + 1][j], dp[i][j - 1]);
            }
        }
        return dp[0][n - 1];
    }
};
```

## 句子和词典

leetcode 139 140 判断一个句子是否由一个词典构成，并给出所有的构成方法

```
class Solution {
public:
    vector<string> wordBreak(string s, vector<string>& wordDict) {
        int n = s.size();
        vector<bool> dp(n + 1, 0);
        dp[0] = true;
        for(int i = 0; i < n; i++) {
            for(auto w : wordDict) {
                if(i + w.size() <= n && s.substr(i, w.size()) == w) {
                    if(dp[i] == true)
                        dp[i + w.size()] = dp[i];
                }
            }
        }
        vector<string> res;
        dfs(res, s, "", n, wordDict, dp);
        return res;
    }

private:
    void dfs(vector<string>& res, string s, string cur, int pos,
            vector<string>& wordDict, vector<bool>& dp) {
        if(pos == 0) {
            res.push_back(cur);
            return ;
        }
        if(!dp[pos]) return ;
        for(auto w : wordDict) {
            int newPos = (int)pos - (int)w.size();
            if(newPos >= 0 && s.substr(newPos, w.size()) == w) {
                string newCur = (cur == "" ? w : w + " " + cur);
                dfs(res, s, newCur, newPos, wordDict, dp);
            }
        }
    }
};
```

## 位运算

### 位运算处理数组中的数

#### 解法

leetcode 137

题目描述：给定一个包含n个整数的数组，除了一个数出现一次外所有的整数均出现三次，找出这个只出现一次的整数。

题目分析：对于除出现一次之外的所有的整数，其二进制表示中每一位1出现的次数是3的整数倍，将所有这些1清零，剩下的就是最终的数。用ones记录到当前计算的变量为止，二进制1出现“1次”（mod 3 之后的 1）的数位。用twos记录到当前计算的变量为止，二进制1出现“2次”（mod 3 之后的 2）的数位。当ones和twos中的某一位同时为1时表示二进制1出现3次，此时需要清零。即用二进制模拟三进制计算。最终ones记录的是最终结果。

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int ones = 0, twos = 0, threes = 0;
        int n = nums.size();
        for(int i = 0; i < n; i++) {
            twos |= ones & nums[i];
            ones ^= nums[i];
            threes = ~(ones & twos);
            ones &= threes;
            twos &= threes;
        }
        return ones;
    }
};
```