目录

[统计理论 2](#_Toc84701229)

[概率 2](#_Toc84701230)

[数据分布 3](#_Toc84701231)

[N次伯努利试验(Bernoulli trials), 二项分布，有放回抽样 3](#_Toc84701232)

[超几何分布,无放回抽样 4](#_Toc84701233)

[Bootstrap 4](#_Toc84701234)

[机器学习 6](#_Toc84701235)

[k均值算法 6](#_Toc84701236)

[图像处理 6](#_Toc84701237)

[几何形状目标检测 6](#_Toc84701238)

[color的分割 (高斯混合模型GMM) 7](#_Toc84701239)

[数据结构和算法 8](#_Toc84701240)

[排序 8](#_Toc84701241)

[字符串搜索 12](#_Toc84701242)

[树和图 13](#_Toc84701243)

[数值优化 16](#_Toc84701244)

[深度学习应用 17](#_Toc84701245)

[Style transfer 17](#_Toc84701246)

[LeetCode 18](#_Toc84701247)

[递归 分而治之 19](#_Toc84701248)

[动态规划算法 19](#_Toc84701249)

[回溯算法 21](#_Toc84701250)

# 统计理论

## 概率

在抽奖问题中，无论是放回随机抽取，还是不放回随机抽取，中奖的概率都与抽奖的次序无关。如已知3张奖券中只有1张有奖，甲乙丙3名同学依次不放回地各随机抽取1张，他们中奖的概率与抽奖的次序有关吗？

解：用A,B,C分别表示甲乙丙中奖的事件，则 (即“乙中奖”等价于“甲没中奖且乙中奖”)， (即”丙中奖“等价于”甲和乙都没中奖”

P(A) = 1/3

P(B) = P() = P() P() = 2/3 \* 1/2 = 1/3

P( C ) = P() = P() P() = 2/3 \* 1/2 = 1/3

子女身高有向平均值 “回归”的倾向

高个子父亲有生高个子儿子的趋势，但一群高个子父亲的儿子们有平均身高要低于父亲们的平均身高，如高个子父亲的平均身高为185cm, 但其子女的平均身高为184cm

矮个子父亲有生矮个子儿子的趋势，但一群矮个子父亲的儿子们的平均身高要高于父亲们的平均身高，如矮个子父亲的平均身高为170cm, 但其子女的平均身高为171cm

经验回归方程：Y = AX + B + E E表示除预测因子X外的随机误差

影响儿子身高的因素：父母亲身高，生活环境 ，饮食习惯等

胸径预测树高

## 数据分布

### N次伯努利试验(Bernoulli trials), 二项分布，有放回抽样

n次独立试验，每 个试验结果只有两种状态，求发生某种状态的次数的概率分布 满足二项分布(binomial distribution)，假定某种状态的概率为p

袋子里100=黄球40+白球60, **有放回抽样**20个球，求样本中黄球个数的概率分布

N件产品中有M件次品，**无放回抽样n件**，则次品数满足几何分布

对产品抽样检验，随机抽取n件，样本中不合格品数的概率分布

n次射击，中靶次数X的概率分布

n次抛硬币，正面朝上次数X的概率分布

n次比赛，获胜几局X的概率分布

拓展一下：若甲选手获胜概率为0.6, 乙选手获取概率为0.4, 采用3局2胜制，还是5局3胜制对甲更有利？

实际上，比赛局数越多，对实力较强者越有利

解：若采用3局2胜制，甲最终获胜的概率为P(X>=2) = P(X=2) + P(X=3)

若采用5局3胜制，甲最终获胜的概率为P(X>=3) = P(X=3) + P(X=4) + P(X=5)

二项分布的均值: np，方差：np(1-p)

参加某保险人群中发生保险事故的人数

试制药品治愈某种疾病的人数

感染某种病毒的家禽数

### 超几何分布,无放回抽样

注意：无放回抽样，每次抽取不是同一个试验，而且 各次抽取的结果也不独立，故不符合n重伯努利试验

N件产品中有M件次品，随机无放回抽样n件，则次品数满足超几何分布

袋子里100=黄球40+白球60, **有放回抽样**20个球，求样本中黄球个数的概率分布

袋子里100=黄球40+白球60, **无放回抽样**20个球，求样本中黄球个数的概率分布

超二项分布的均值: np

对于不放回抽样，当抽样数n远小于总数N时，每抽取一次后，对N的影响很小，此时，超几何分布可以用二项分布来近似

## Bootstrap

应用1： 样本不同，模型结果不同，如果评价抽样差异性(sampling variability)？

理论上：population -> sample, sample, … -> (AUC1, AUC2, ….) 作出样本统计值的分布,近似实际分布，根据central limit theorem, 一般呈现normal distribution.

我们用如下两个metric来度量sampling variability

Standard error = std( m1, m2, …)

Confidence interval = [2.5%th, 97.5%th] from low to high

我们无法从population中采样出新的样本，故采用替代方案Bootstrap

将sample复制几百万遍作为假设的population, 然后从中随机采样

Basic Bootstrap Theory: Original Sample -> sample replicated a huge number of times -> Draw lots of resamples

实际中等价中有放回抽样

参考论文Samuelson, F. W. , and N. Petrick . "Comparing image detection algorithms using resampling." *IEEE International Symposium on Biomedical Imaging: Nano to Macro* IEEE, 2006

算法结果分布: AI vs 医生结果分布: phy

Diff\_center = AI – phy的中间数

原假设: AI 与医生无差异

备选假设: AI与医生有显著差异，若Diff\_center> 0, 则AI显著比医生好；若Diff\_center< 0, 则AI显著比医生差

无差异分布x-y: center around 0

实际分布（bootstrap 产生的分布）AI-phy: centered around Diff\_center

若实际分布与无差异分布靠得很近，则无差异；若离得很远，则有差异。如何度量?

Diff\_center是否落在无差异分布的接受域还是拒绝域内？

若P(x-y > Diff\_center) < alpha, 说明Diff\_center落入到拒绝域内，拒绝原假设，接受备选假设

由于无差异分布x-y不知道，我们只知道实际分布, 但由公式or分布对称性

P (AI-phy < 0) = P(x-y > Diff\_center) = 0.0535， 拒绝原假设，接受备选假设

原假设： AI – phy >= 0

备选假设 : AI – phy < 0

p-value = P(x-y < Diff\_center) =P(AI-phy>0) = 0.931, 接受原假设

应用2： bagging = Bootstrap aggregating, 有放回抽样，从而训练多个模型，然后进行投票表决

# 机器学习

## k均值算法

应用：获取成人的身高，体重，腰围，…, 设计K个型号的衣服

通过k个簇内均值刻画

1. 初始化k个均值向量 （随机选取样本），代表k个簇
2. 考察每个样本，划分到最近的k个簇 (比较样本与k个均值向量的距离)
3. 更新k个族的刻画（计算族均值向量）
4. 重复上述过程，直到算法停止

# 图像处理

## 几何形状目标检测

解析表达式的：如直线，圆，椭圆等

思想： 把图像中属于某种图形的点集（二维）映射到一个点（可以是高维）上，这个点记录了点集中点的数目，使得程序通过搜索峰值找到该点，这个点就是图形参数

图像空间几何形状 -> hough空间

直线：边缘点 -> (r, theta): 正弦线交点

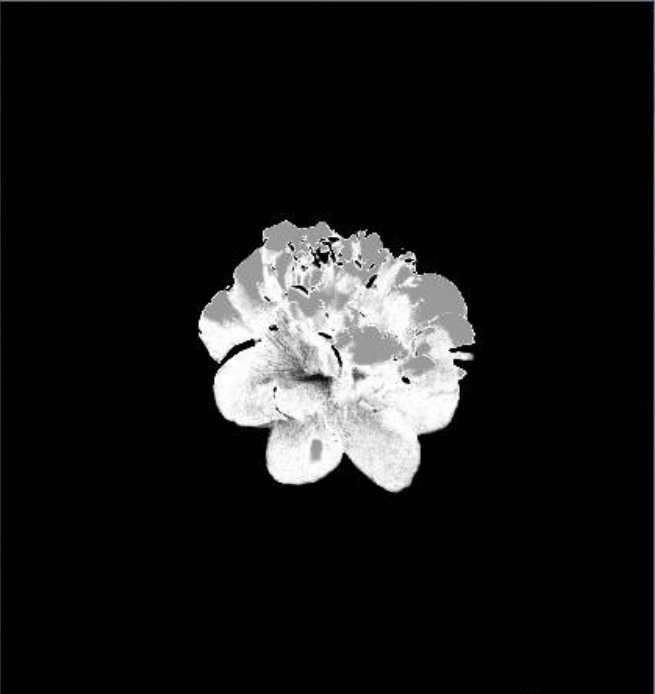
圆：边缘点 -> (x0, y0, r): 圆锥体交点

椭圆：边缘点 -> (x0, y0, a, b, theta)

求**轮廓的相似度？（轮廓匹配）**

**比较轮廓Hu矩** （通过对轮廓上所有点进行积分运算而得到的特征）

## color的分割 (高斯混合模型GMM)

算法：

1. 假定高斯分布，估算Object的mean and std (注意是每个通道)
2. 根据高斯分布，计算整张图像的PDF值即概率分布图(注意：每个像素点每个通道都有PDF值)
3. 在高斯空间，融合多通道（比如各通道乘积）, 并归一化概率分布图像
4. 分割概率分布图像得mask

img = cv.imread('flower.jpg')

roi = cv.imread('flower\_roi.jpg')

# normalize channel: R/(R+G+B), G/(R+G+B), B/(R+G+B)

roi = normalize(roi)

img = normalize(img)

pdf = np.ones(img.shape[:-1]) # pdf of RGB

for c in range(roi.shape[-1]):

calc parameter of PDF based on ROI

mean, std = np.mean(roi[:, :, c]), np.std(roi[:, :, c])

# calc PDF of each channel

p = (img[:, :, c]- mean)\*( img[:, :, c]- mean) / (2\*std\*std)

p = np.exp(-p)

p = p / (std\*np.sqrt(2\*np.pi))

pdf \*= p

cv.normalize(pdf, pdf, 0, 255, cv.NORM\_MINMAX)

# 数据结构和算法

## 排序

**快速排序**：

1. 将数组划分为left子数组 <= pivot <= right子数组
2. 对pivot左边的序列递归快速排序
3. 对pivot右边的序列递归快速排序

void q\_sort(int v[], int left, int right) {

if (left >= right) return;

int idx\_pivot = partition(v, left, right);

q\_sort(v, left, idx\_pivot - 1);

q\_sort(v, idx\_pivot + 1, right);

}

如何划分数组，使left子数组 <= pivot <= right子数组

设最左边值为pivot，

j从右到左搜索，找小于pivot的第一个值

i从左到右搜索，找大于pivot的第一个值

交换以上两个值，继续剩余搜索，直至i遇到j

此时i就是划分点，与pivot交换

int partition(int v[], int left, int right) {

int pivot = v[left];

int i = left;

int j = right;

while (i != j) {

while (v[j] >= pivot && i < j) j--;

while (v[i] <= pivot && i < j) i++;

if (i < j) swap(v[i], v[j]);

}

swap(v[left], v[i]);

return i;

}

**堆排序**

基本思想：

1. 将待排序数组构成最大堆

for (int i = n/2-1; i >= 0; --i)

max\_heap(v, i, n);

1. 将顶端数与末尾数交换，则末尾为最大值

swap(v[0], v[i]);

1. 对剩余n-1个数再建最大堆，重复2,3直至i=1

max\_heap(v, 0, i);

void heap\_sort(int v[], int n) {

// build max heap

for (int i = n/2-1; i >= 0; --i) max\_heap(v, i, n);

for (int i = n-1; i >= 1; i--) {

swap(v[0], v[i]);

max\_heap(v, 0, i);

}

}

**如何建最大堆（堆排序）？**

1. 对当前节点i, 若子节点超出边界，结束；
2. 求最大左右子节点，若大于当前节点，则互换交换；
3. 对子树建堆

void max\_heap(int v[], int curIdx, int n) {

int leftChild = 2 \* curIdx + 1;

if (leftChild >= n) return;

// max of left child and right child

if (leftChild + 1 < n && v[leftChild] < v[leftChild + 1]) ++leftChild;

// exchange parent node and max child node

if (v[curIdx] < v[leftChild]) {

swap(v[curIdx], v[leftChild]);

// update the child tree

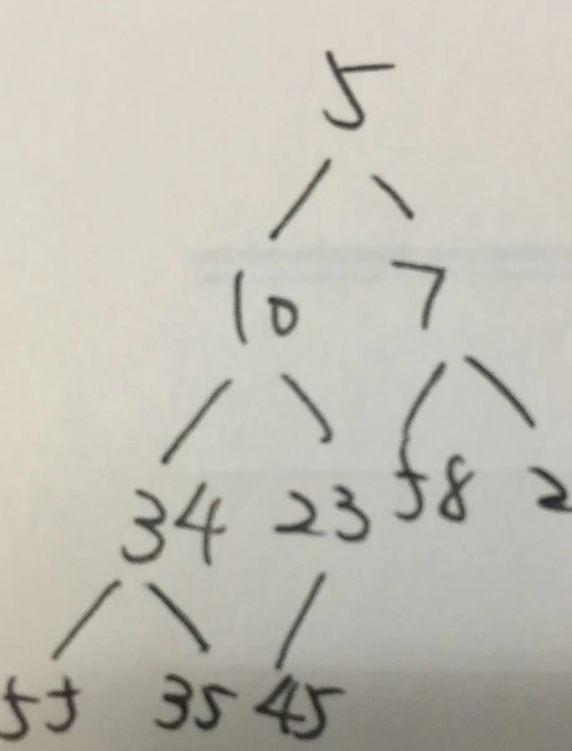
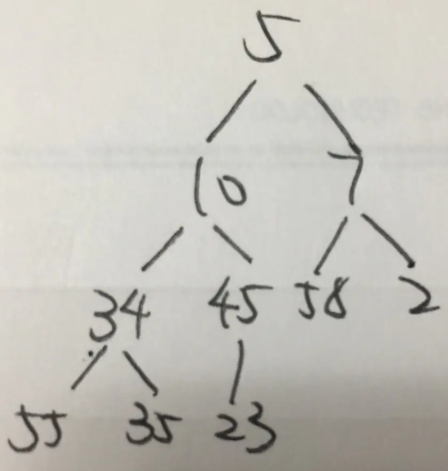
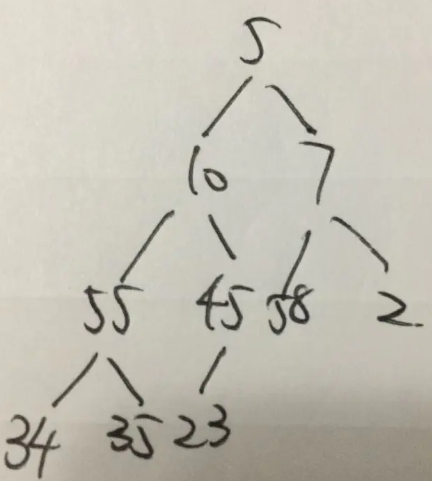
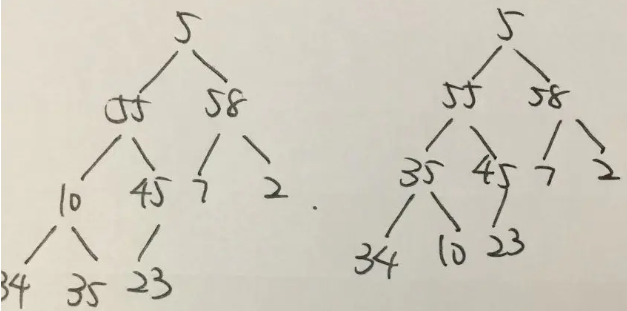
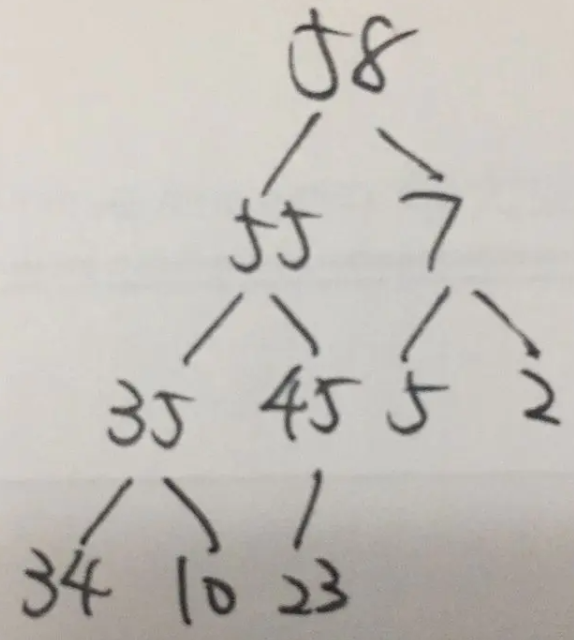
max\_heap(v, leftChild, n);

}

}

第i个序号的结点的数，那么他的父结点序号就是(i-1)/2，它的孩子结点就为2i+1与2i+2

int a[10] = {5,10,7,34,23,58,2,55,35,45}

maxheap(v, n/2-1, n) ->maxheap(v, 3, n)->maxheap(v, 2, n)-> maxheap(v, 1, n)->maxheap(v, 0,n)

**bubble sort**

对v[0 : n]依次比较相邻两个元素，若后者比前者小就交换，一轮下来最小值冒出来

对v[1 : n]重复以上过程

…

void bubble\_sort(int v[], int n) {

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

for (int j = n-1; j >= i; --j) {

if (v[j] < v[j - 1]) {

swap(v[j], v[j + 1]);

}

}

}

}

**Selection sort**

对v[0 : n]选择最小元素放入第一个 (采用交换形式)

对v[1 : n]选择最小元素放入第二个

…

void selection\_sort(int v[], int n) {

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

求v[i : n]最小值v[min\_idx];

swap(v[i], v[min\_idx]);

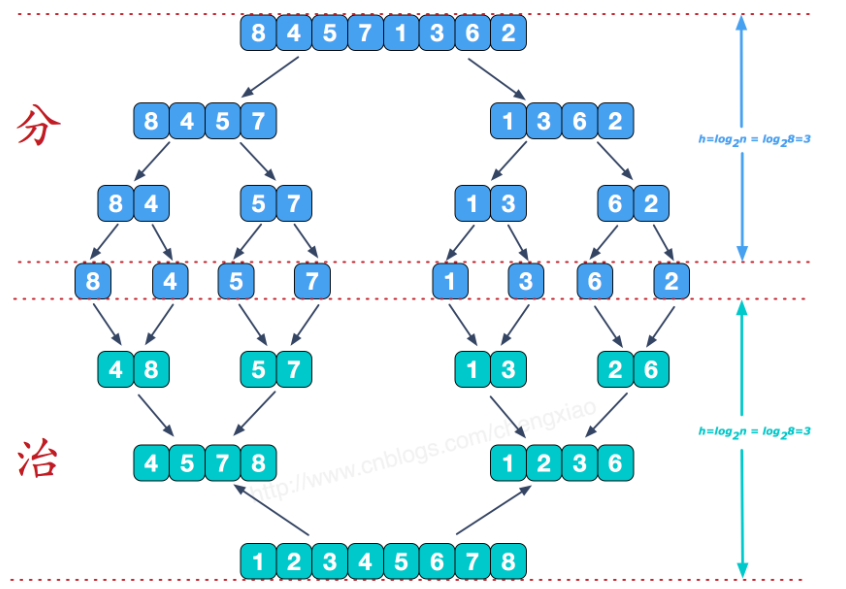
}

}

**插入排序**

**同打排一样，拿到新排插到有序的手排**

**归并排序：分而治之**



分很容易，关键是如何合并？

每次从两段中取一个数据，较小者放入temp数组

copy两段中剩余的数据于temp数组

void merge\_sort(int v[], int left, int right) {

if (left >= right) return;

int mid = left + (right - left) / 2;

merge\_sort(v, left, mid);

merge\_sort(v, mid + 1, right);

merge(v, left, mid, right);

}

void merge(int v[], int left, int mid, int right) {

int\* temp = (int\*) new int[right - left + 1];

int idx1 = left;

int idx2 = mid+1;

int idx3 = 0;

//每次从两段中取一个数据，较小者放入temp数组

while (idx1 <= mid && idx2 <= right) {

temp[idx3++] = v[idx1] < v[idx2] ? v[idx1++] : v[idx2++];

}

// copy剩余的数据

while (idx1 <= mid) temp[idx3++] = v[idx1++];

while (idx2 <= right) temp[idx3++] = v[idx2++];

for (int i = 0; i < idx3; ++i) v[left + i] = temp[i];

delete[] temp;

}

**计数排序：**

当输入的元素是整数，且序列比较集中时，用计数排序时间复杂度是O(n+k)

输入值转化为键，出现次数转化为值

void buck\_sort(int v[], int n) {

int min\_value = v[get\_min\_idx(v, n)];

int max\_value = v[get\_max\_idx(v, n)];

int bucklen = max\_value - min\_value + 1;

int\* temp = new int[bucklen];

memset(temp, 0, bucklen\*sizeof(int));

for (int i = 0; i < n; ++i) temp[v[i]-min\_value] ++;

int k = 0;

for (int i = 0; i < bucklen; ++i) {

int num = temp[i];

int value = i+min\_value;

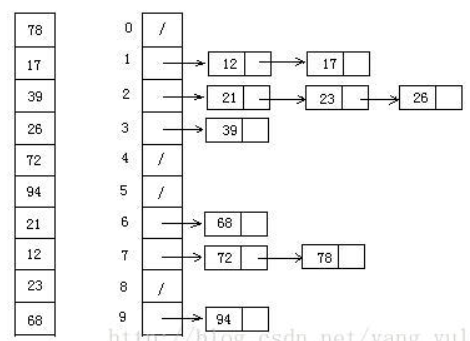
for (int j = 0; j < num; ++j) v[k++] = value;

}

}

**桶排序：**

计数排序的改进版，将数据映射到桶，各个桶有序，只要对桶分别排序。关键在于映射函数,常用哈希映射hash=e\*len/(max+1)，复杂度介于O(n)~O(nlgn)



**求top K ?**

1. 维护一个k大小的有序序列，
2. 遍历剩余数据，依次和有序序列中的最小值比较，若大则将最小值换出; 重新对k序列排序
3. 直至遍历完所有数据。

## 字符串搜索

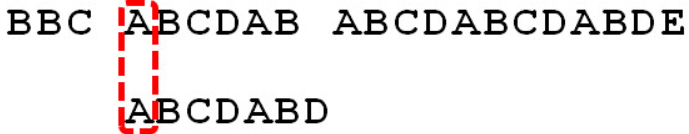
如原字符串：”BBC ABCDAB ABCDABCDABDE”

子串： ”ABCDABD”

暴力搜索：同图像处理一样，滑窗搜索

**KMP算法**：同暴力搜索一样，只是用上部分匹配的信息，部分匹配用于决定步伐

Stride = 已匹配的字符数 – 对应的部分匹配值的长度



stride = len(ABCDAB) – len(part\_match) = 6 – 2

见图b

如何计算部分匹配值？(根据子串的前缀和后缀的intersection计算出来)

“AB” 前缀：[A], 后缀为[B]，则共有元素长度为0

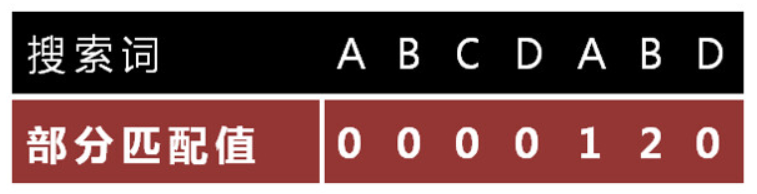
“ABC”前缀：[A, AB], 后缀为[BC, C], 则共有元素长度为0

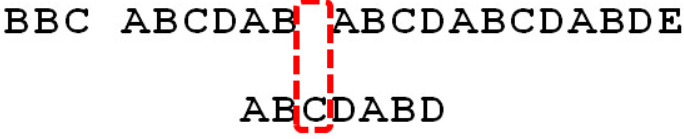
“ABCD”前缀：[A, AB, ABC], 后缀为[BCD, CD, D], 则共有元素长度为0

“ABCDA”前缀：[A, AB, ABC, ABCD], 后缀为[BCDA, CDA, DA, A], 则共有元素长度为1

…

得部分匹配表：





stride = len(AB) – len(part\_match) = 2 – 0

Repeat above process

## 树和图

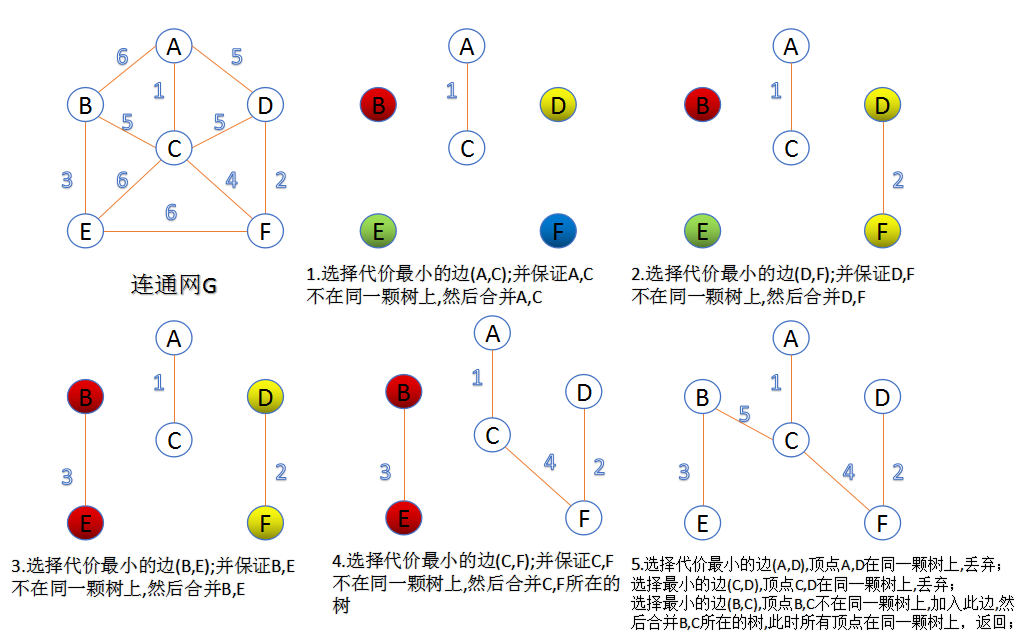
最小生成树

1. Kruskal算法 （加边法）

1. 把图中的所有边按代价从小到大排序；n个顶点看成n棵树；

2. 按权值从小到大选择边，所选边顶点应属于不同的树，加入到最小生成树

3. 重复以上,直到所有顶点都在树内

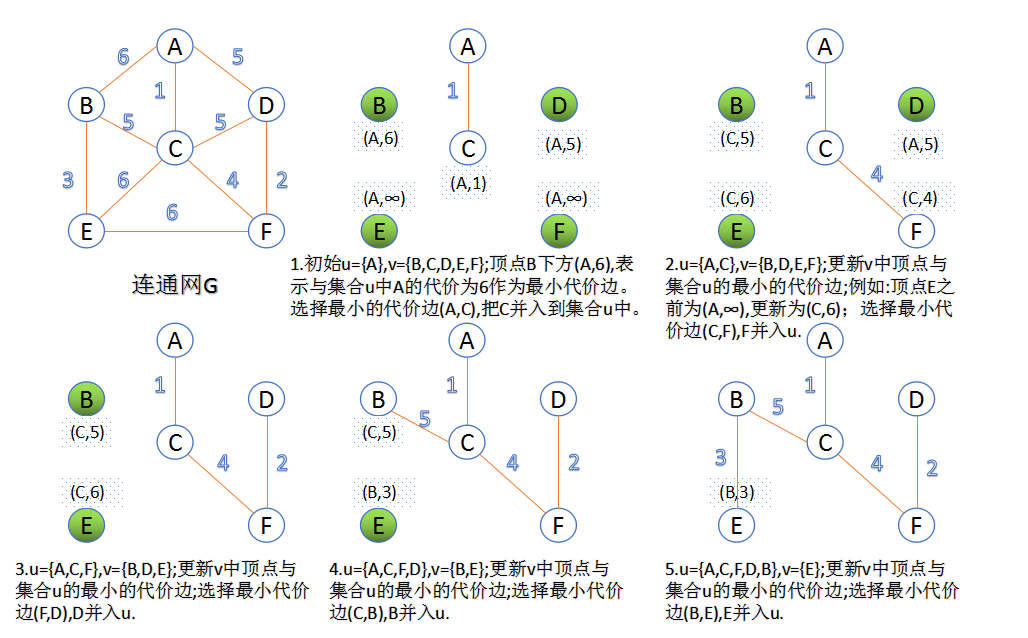


2.Prim算法 （加点法）

1）以某点开始作为初始点集合

2）求点集合相连的最小边，最小边中的新点加入点集合

3）重复以上过程，直到点集合含n个顶点



**KD-Tree**

优缺点: Kd树在维度较小时（比如20、30），算法的查找效率很高，然而当数据维度增大（例如：K≥100），查找效率会随着维度的增加而迅速下降。假设数据集的维数为D，一般来说要求数据的规模N满足N>>2的D次方，才能达到高效的搜索。

1. Kd-Tree的构建

1. 在K维数据集合中选择具有最大方差的维度k， (Note: 数据方差大说明沿该坐标轴方向上数据点分散的比较开)

然后在该维度上选择中值m为pivot对该数据集合进行划分，得到两个子集合 (Note:

确保建立的树尽量地平衡)

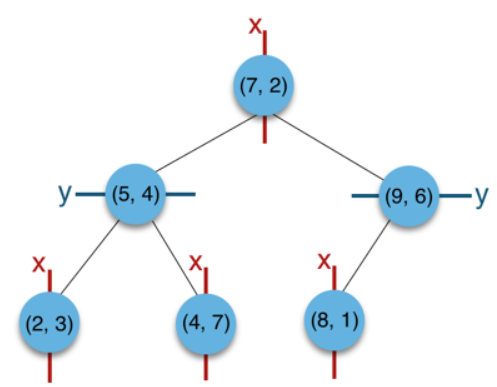
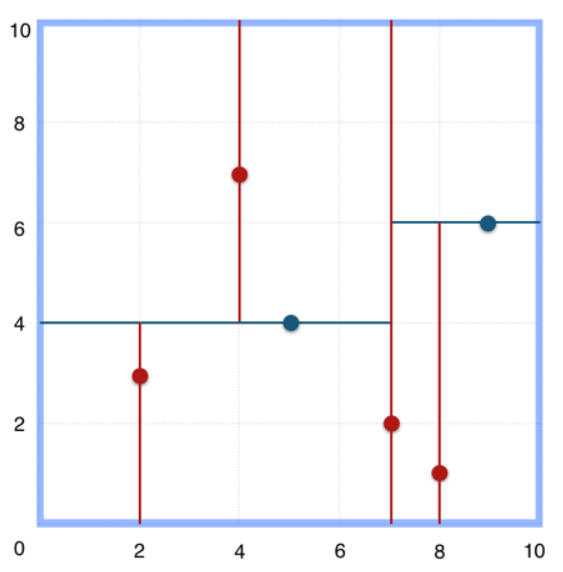
2. 对两个子集合重复（1）步骤的过程，直至所有子集合都不能再划分为止；

假定二维平面上，点的集合{(2,3)，(5,4)，(9,6)，(4,7)，(8,1)，(7,2)}

寻找根节点：计算数据在x和y轴的各自方差, 求出x轴方差大，作为切分维度x; 将数据在x轴排序，得中值(7, 2)作为根节点

划分为左子集{(2,3), (4,7), (5,4)}, 右子集{(8,1), (9,6)}

对左子集，右子集重复以上过程

1. Kd-Tree的最近邻查找
2. 将查询点Q从根结点开始，按切分维比较，获取从根结点->叶子结点的路径
3. 进行回溯操作，该操作是为了找到离Q更近的“最近邻点”

例1：查找点Q(2.1,3.1)

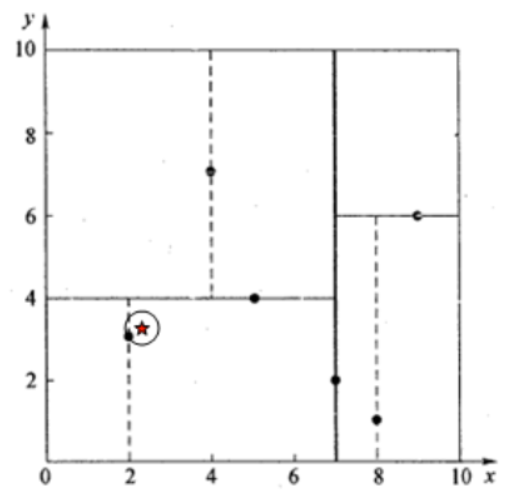
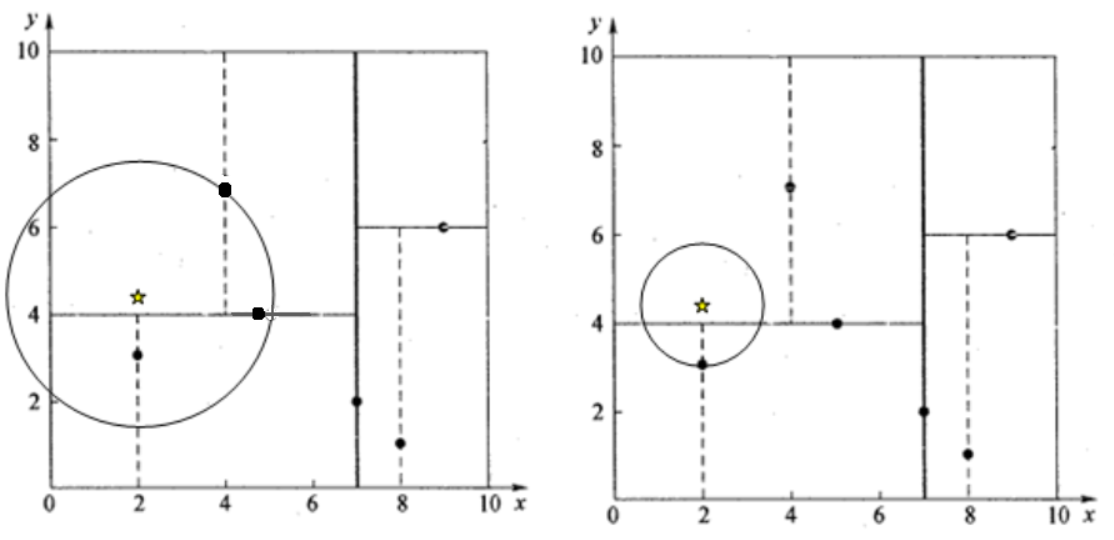
1. 获取搜索路径(7,2, x) -> (5,4, y) -> (2,3, x)
2. '回溯'操作

检查点(2,3):以(2,3)作为当前最近邻点，计算到查询点距离作为当前最近距离0.1414

检查点(5,4): 以查询点为圆心，以当前最近距离为半径画圆，如图所示,发现该圆并不和划分轴y = 4相交，即|Q[y] - 4|=|3.1 - 4|=0.9 > 0.1414，因此不进入该检查点右子空间中去搜索。

同理回溯检查点(7,2)

得最近点(2,3), 最近距离0.1414

例2：查找点Q(2,4.5)

1. **获取搜索路径**(7,2, x, 根) -> (5,4, y, 根) -> (4,7, x, 叶)
2. **'回溯'操作**

检查点(4,7,x,叶)：直接以叶节点(4,7)作为当前最近邻点，计算到查询点距离作为当前最近距离3.202

检查点(5,4,y,根): 以查询点为圆心，以当前最近距离为半径画圆，如图所示,发现该圆和划分轴y = 4相交，即|Q[y] - 4|=|4.5 - 4|=0.5 < 3.202，因此需要进入该检查点另一子空间中去搜索,

更新搜索路径为(7,2, x,根) -> (2,3, x,叶), 更新当前最近邻点(5,4), 当前最近距离3.04

检查点(2,3,x,叶): 计算查询点与叶节点距离，更新当前最近邻点(2,3),当前最近距离1.5

同理回溯剩余检查点得最近点(2,3), 最近距离1.5

Sklearn 代码：

from sklearn.neighbors import KDTree

points = np.random.random((100, 2))

tree = KDTree(points)

query = points[0]

# kNN

dists, indices = tree.query([query], k=3)

# query radius

indices = tree.query\_radius([point], r=0.2)

# 数值优化

求方程根？

牛顿法

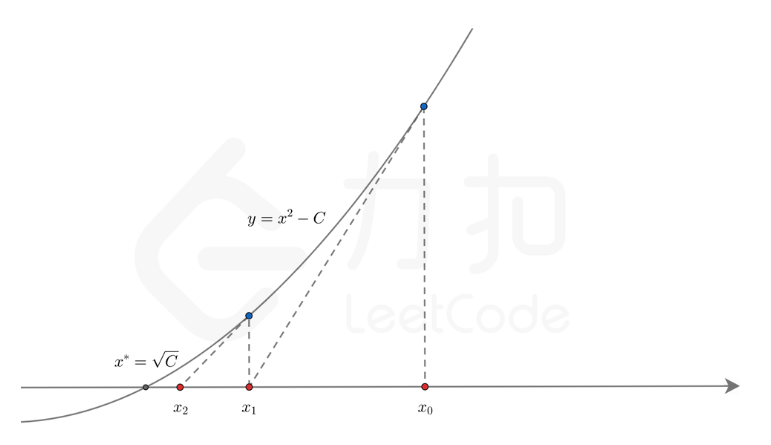
基本思想： 求 等效于用切线近似曲线, 逐步逼近

步骤：

1. 假定初始值

求平方根？ （牛顿法O(lgn)）

即 =0，牛顿法



Monte Carlo方法

求连续曲线下的面积？

随机数落在矩形

判断随机数是否位于曲线下的阴影区域内，即. 若是曲线下点数+1;

求函数极值？

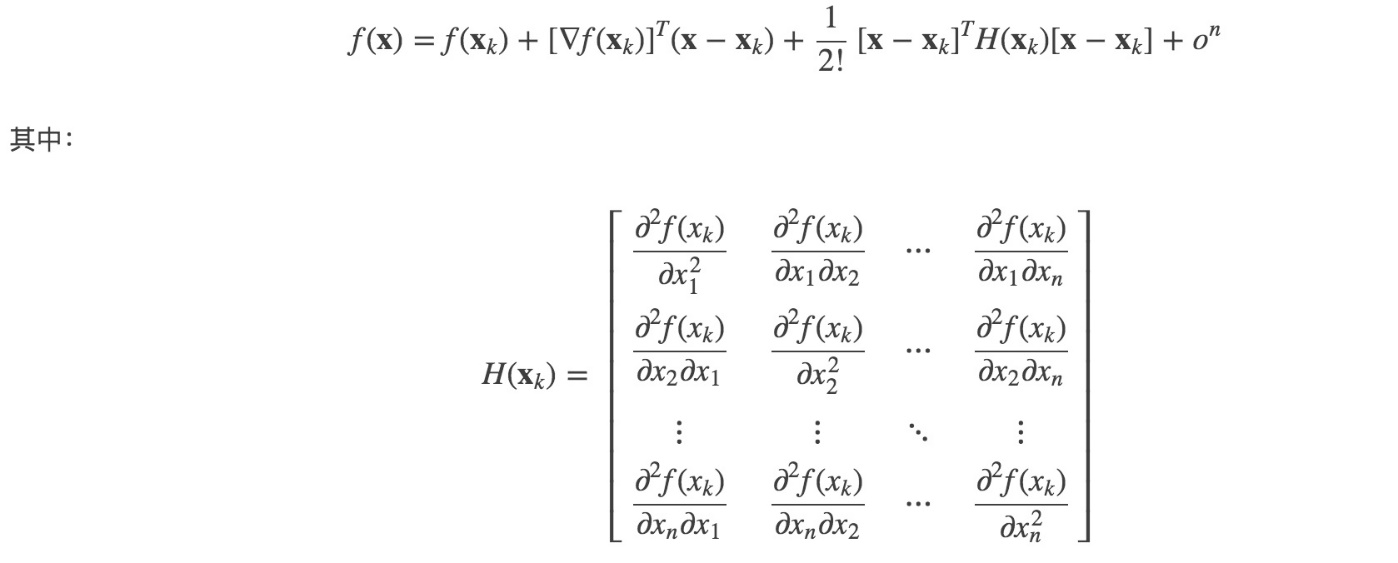
同一元函数一样，多元函数的极值只可能发生在偏导数为0或偏导数不存在的点，或函数定义域端点

求带约束条件下的函数极值

函数， 约束条件,

Lagrange乘子法：求解方程组

多元函数的Taylor展开式：



# 深度学习应用

## Style transfer

Stylized image = image – its texture + style of painting

AdaIN style transfer, Kaggle’s Painter by Numbers dataset

<https://github.com/rgeirhos/Stylized-ImageNet>

# LeetCode

双指针

**寻找链表的中点？**

让快指针一次前进两步，慢指针一次前进一步，当快指针到达链表尽头时，慢指针就处于链表的中间位置

**寻找链表的倒数第 k 个元素？**

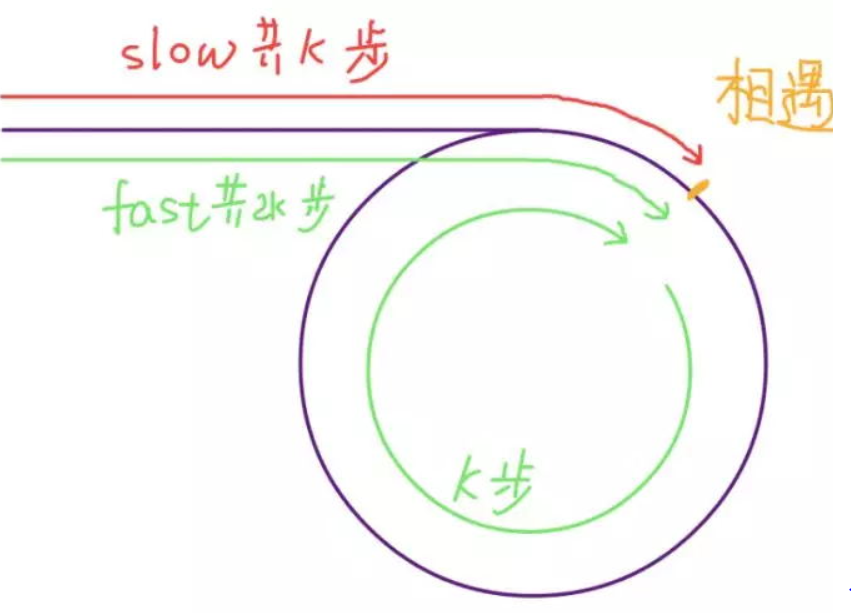
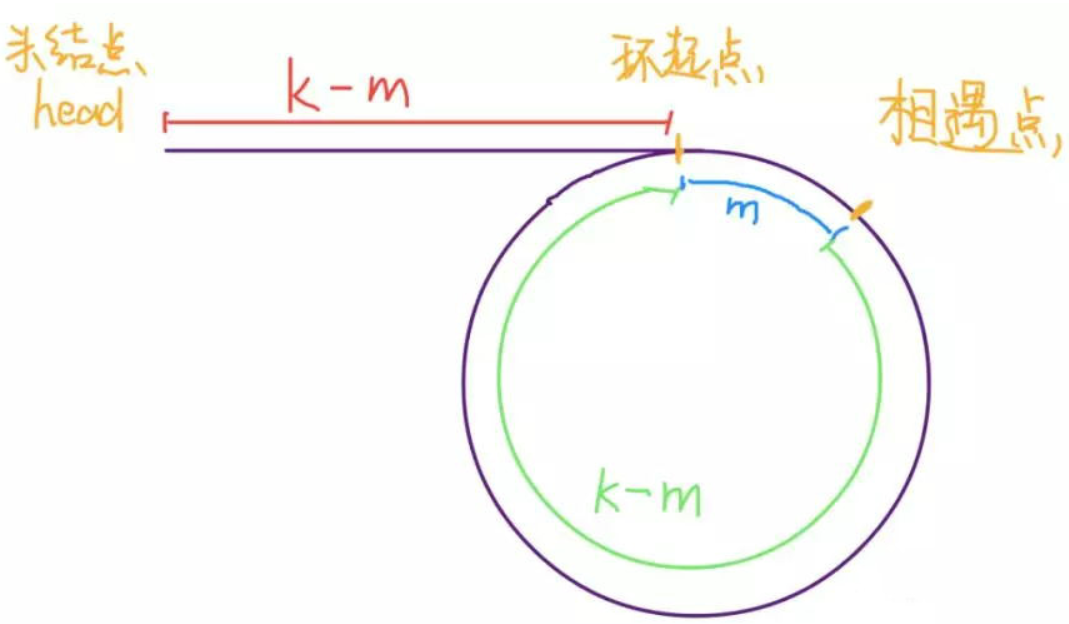
让快指针先走 k 步，然后快慢指针开始同速前进。这样当快指针走到链表末尾 null 时，慢指针所在的位置就是倒数第 k 个链表节点

**判定链表中是否含有环？**

经典解法： 用快慢指针。如果不含有环，跑得快的那个指针最终会遇到 null，说明链表不含环；如果含有环，快指针最终会超慢指针一圈，和慢指针相遇，说明链表含有环。

**已知链表中含有环，返回这个环的起始位置？**

当快慢指针相遇时，让其中任一个指针重新指向头节点，然后让它俩以相同速度前进，再次相遇时所在的节点位置就是环开始的位置

1. 假设慢指针 slow 走 k 步，快指针 fast 走2k 步，此时相遇，则环长度为k
2. 假设相遇点距环起点的距离为 m，那么环起点距头结点 head 的距离为 k – m；巧的是，如果从相遇点继续前进 k - m 步，也恰好到达环起点。
3. 所以，若一个指针从head开始，一个指针从相遇点开始，同步走，k - m 步后就会相遇，相遇之处就是环的起点。

## 递归 分而治之

**快速排序**：

将数组划分为left子数组 <= pivot <= right子数组

对pivot左边的序列递归快速排序

对pivot右边的序列递归快速排序

**归并排序：**

1. 对当前子序列, 若子序列长度为1，结束；
2. 求左子序列递归排序
3. 对右子序列递归排序
4. 合并左右子序列

void merge\_sort(int v[], int left, int right) {

if (left >= right) return;

int mid = left + (right - left) / 2;

merge\_sort(v, left, mid);

merge\_sort(v, mid + 1, right);

merge(v, left, mid, right);

}

## 动态规划算法

**数组中不重复两数之和等于目标值**

思路：一遍hash table. 在迭代中，我们将元素值和索引添加到表中，同时检查表中是否已经存在(target-当前元素), 若存在即对应解

vector<int> twoSum(vector<int>& nums, int target) {

vector<int> result;

**map<int, int> dict;**

for (int i=0; i < nums.size(); ++ i) {

int complement = target - nums[i];

**map<int, int>::iterator iter = dict.find(complement);**

if (iter != dict.end()) {

result.push\_back(iter->second);

result.push\_back(i);

break;

}

**dict[nums[i]] = i;**

}

return result;

}

**三数之和**

1. 先排序
2. 任选某位，左指针从左端开始，右指针从右端开始，若三数之和>target,则右指针向前走，若三数之和<target,则左指针向前走。

**找出一条从左上角到右下角的路径，路径上数字之和最小**

动态规划：从右下角往左上角填充二维表格，当前点 = 当前点 + min(右边点,下边点)

int minPathSum(vector<vector<int>>& grid) {

for (int i = grid.size()-1 ; i >= 0; -- i) {

for (int j = grid[0].size()-1 ; j >= 0 ; -- j) {

if (i == grid.size()-1 && j < grid[0].size()-1) grid[i][j] = grid[i][j] + grid[i][j+1];

else if (j == grid[0].size()-1 && i < grid.size()-1) grid[i][j] = grid[i][j] + grid[i+1][j];

else if (i < grid.size()-1 && j < grid[0].size()-1)

grid[i][j] = grid[i][j] + min(grid[i+1][j], grid[i][j+1]);

**}**

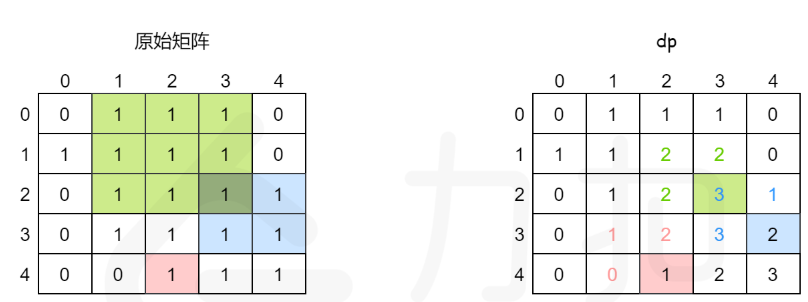
**}**

return grid[0][0];

}

**求图里目标的最大内接正方形？**

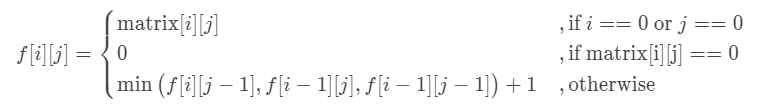
**求目标最大内接正方形的数目？**



dp[i][j] 表示以 (i, j) 为右下角的正方形的最大边长

dp[i][j] 也表示以 (i, j) 为右下角的正方形的数目为 x（即边长为 1, 2, ..., x 的正方形各一个）

则正方形数目 = sum(dp)



## 回溯算法

回溯法 ：一种通过探索**所有可能的候选解**来找出所有的解的算法。如果候选解被确认不是一个解的话（或者至少不是最后一个解），回溯算法会通过在上一步进行一些变化抛弃该解，即回溯并且再次尝试

代码框架：

result = []

def backtrack(路径, 选择列表):

if 满足结束条件:

result.add(路径)

return

for 选择 in 选择列表:

做选择

backtrack(路径, 选择列表)

撤销选择

**全排列** ： N 个数字的全排列一共有 N!个。

思路：

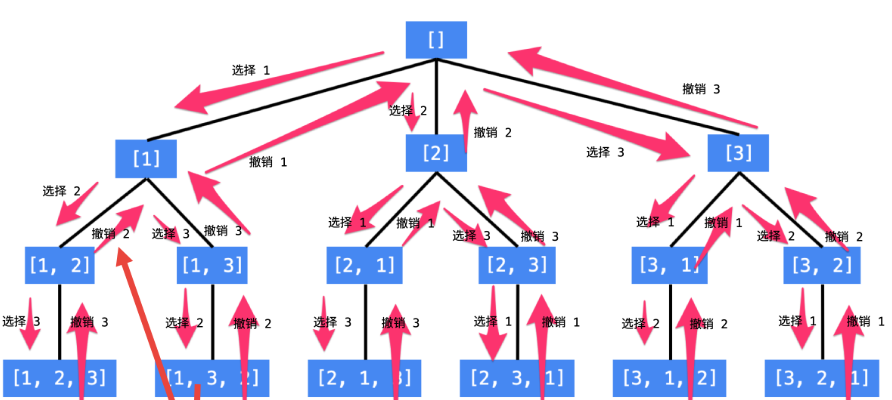
1. 顺序枚举每一位可能出现的情况，已经选择的数字在接下来要确定的数字中不能出现，得到树形结构
2. 执行一次深度优先遍历，从树的根结点到叶子结点形成的路径就是一个全排列。

以数组 [1, 2, 3] 的全排列为例：

在枚举第一位的时候，有 3 种情况。

在枚举第二位的时候，前面已经出现过的数字就不能再被选取了；

在枚举第三位的时候，前面 2 个已经选择过的数字就不能再被选取了。



深度优先遍历：

往下走一层的时候，path 变量在尾部追加，而往回走的时候，需要撤销上一次的选择， 所以path是一个栈，全局维护一份状态变量

vector<vector<int>> permute(const vector<int>& nums) {

vector<vector<int>> res;

vector<int> path;

vector<bool> isVisited(nums.size());

dfs(nums, res, 0, path, isVisited);

return res;

}

void dfs(const vector<int>& nums, vector<vector<int>>& res, int depth, vector<int>& path, vector<bool>& isVisited) {

// 叶节点，路径就是结果

if (depth == nums.size()) {

vector<int> \_path(path);

res.push\_back(\_path);

return;

}

for (int i = 0; i < nums.size(); ++i) {

if (isVisited[i]) continue;

//向下搜索，递归

**path.push\_back(nums[i]);**

isVisited[i] = true;

**dfs(nums, res, depth + 1, path, isVisited);**

// 回溯，状态重置回上一层

**path.pop\_back();**

isVisited[i] = false;

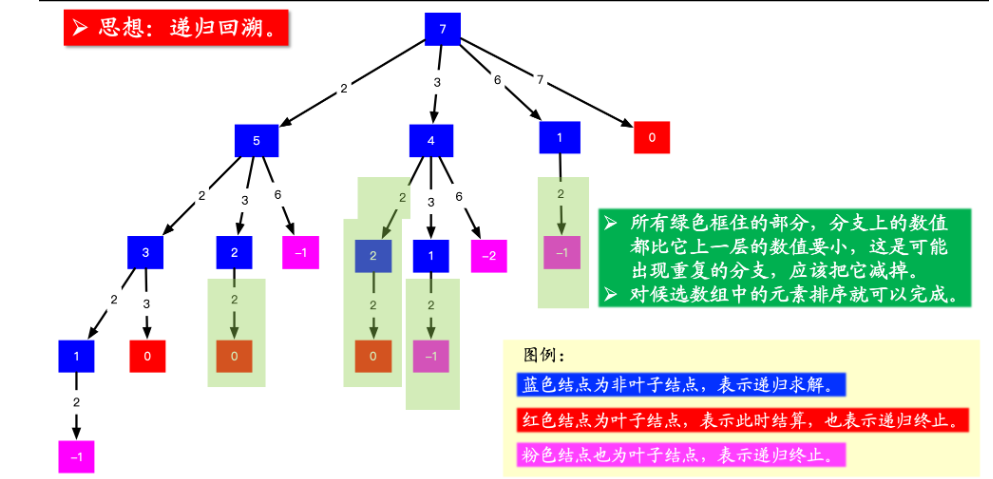
}

}

**组合**

输入: candidates = [2,3,6,7]，target = 7。

1. 候选数组里有 2 ，如果找到了 7 - 2 = 5 的所有组合，再在之前加上 2 ，就是 7 的所有组合；
2. 同理考虑 3，如果找到了 7 - 3 = 4 的所有组合，再在之前加上 3 ，就是 7 的所有组合，依次这样找下去；
3. 减到 00 或者负数的时候，到了叶子结点；
4. **从根结点到叶子结点（必须为 0）的路径**，就是题目要我们找的一个组合。



如何去重？

回溯时，若新的候选点<搜索路径尾节点，则skip

vector<vector<int>> combinationSum(const vector<int>& candidates, int target) {

vector<int> \_candidates(candidates);

sort(\_candidates.begin(), \_candidates.end()); //仅仅为了加速搜索

vector<vector<int>> res;

vector<int> path;

dfs\_combinationSum(\_candidates, res, target, path);

return res;

}

void dfs\_combinationSum(const vector<int>& candidates, vector<vector<int>>& res, int target, vector<int>& path) {

if (target == 0) {

vector<int> \_path(path);

res.push\_back(\_path);

return;

}

for (int i = 0; i < candidates.size() && **target - candidates[i] >= 0**; ++i) {

// remove repetition

**if (path.size() > 0 && candidates[i] < path[path.size() - 1]) continue;**

**path.push\_back(candidates[i]);**

**dfs\_combinationSum(candidates, res, target-candidates[i], path);**

**path.pop\_back();**

}

}

对n\*n矩阵旋转90度？（就地旋转，不占用辅助内存）

旋转90度 = 水平翻转 + 主对角线翻转