

## 异或运算

$a \oplus b$  相当于  $a = a \oplus b$ , 将十进制数字转化为二进制进行运算, 相同为0, 相异为1, 0和任何数异或运算都是原来的那个数。

可以用来判断数组中哪个数字只出现过一次 (通过将所有数与0进行异或运算)

## 快慢指针

- 单链表中可任意用来寻找“**中点**”，快指针 (fast) 每一步走两个结点，慢指针 (slow) 每一步走一个结点。当快指针到达链表末尾时，慢指针应该指向链表最中间的结点。如果是**单数**恰好为中间，如果是**双数**则是中间的**第二个节点**
- 在查找链表倒数第k个节点时，可以通过先让fast先走k步，再与slow同时运动，让两个指针**保持一个距离**，当fast到达结尾时，那么slow的位置就是倒数第k的位置。
- 对于**环形链表**，把快慢指针想象成一个追及问题，当两个指针重合时，就代表链表中有**环**

## 动态规划

例题一：[最大子数组和](#)

### 方法一：动态规划



#### 思路和算法

假设  $nums$  数组的长度是  $n$ , 下标从 0 到  $n - 1$ 。

我们用  $f(i)$  代表以第  $i$  个数结尾的「连续子数组的最大和」，那么很显然我们要求的答案就是：

$$\max_{0 \leq i \leq n-1} \{f(i)\}$$

因此我们只需要求出每个位置的  $f(i)$ ，然后返回  $f$  数组中的最大值即可。那么我们如何求  $f(i)$  呢？我们可以考虑  $nums[i]$  单独成为一段还是加入  $f(i-1)$  对应的那一段，这取决于  $nums[i]$  和  $f(i-1) + nums[i]$  的大小，我们希望获得一个比较大的，于是可以写出这样的动态规划转移方程：

$$f(i) = \max\{f(i-1) + nums[i], nums[i]\}$$

例题二：[买股票的最佳时机](#)

动态转移方程： $dp[i] = \min\{dp[i-1], prices[i]\}$ ,  $dp[i]$  这个数组存着  $i$  位置前最小的价格

例题三：[分割数组以求得最大和](#)

**无后效性**: 每一个子问题只求解一次，以后求解问题的过程不会修改以前求解的子问题的结果

## 背包问题（动态规划）

0/1背包: (物品只能选一次)

```
f[i][j] = max(f[i][j], f[i-1][j-v[i]]+w[i]); //二维
```

```
//滚动数组优化，当前行f[i]的状态只与上一行有关，所以可以用一维数组优化  
//如果是从小到大，前一行的状态会被新一行的状态覆盖掉，这样使用前面已经求出来的状态就会出错  
for(int j = m; j >= v[i]; j--) //从大到小的重量  
    f[j] = max(f[j], f[j - v[i]] + w[i]); //一维
```

完全背包:(物品可以选无数次)

```
f[i][j]=max(f[i][j],f[i][j-v[i]]+w[i]) //二维转移方程
```

```
//从小到大可以满足一件物品不止选一次，物品不选则是前一行的状态，选该物品，则是同一行前面的状态再加上当前的价值  
for(int j = v[i] ; j<=m ; j++) //注意了，这里的j是从小到大枚举，和01背包不一样  
{  
    f[j] = max(f[j], f[j-v[i]]+w[i]); //一维  
}
```

分组背包(同组的物品只能选一个):

和01背包类似，但是最外层的循环是组的数量，对每个组中的物品都讨论选或不选的价值，取最大值

需要三层循环：第一层i循环组的数量，第二层j循环背包的容量，第三层k逐个判断每组物品的价值取最大值

```
for(int i=1;i<=Max;i++){  
    for(int j=m;j>=0;j--){  
        for(int k=1;k<=s[i];k++){  
            if(j>=w[index[i][k]]){//index存的是第i组第k个数在原始数组中的下标，从而  
            //获得该物品的大小和价值  
                dp[j]=max(dp[j],dp[j-w[index[i][k]]]+v[index[i][k]]);  
            }  
        }  
    }  
}
```

## 获得环形链表相交的节点

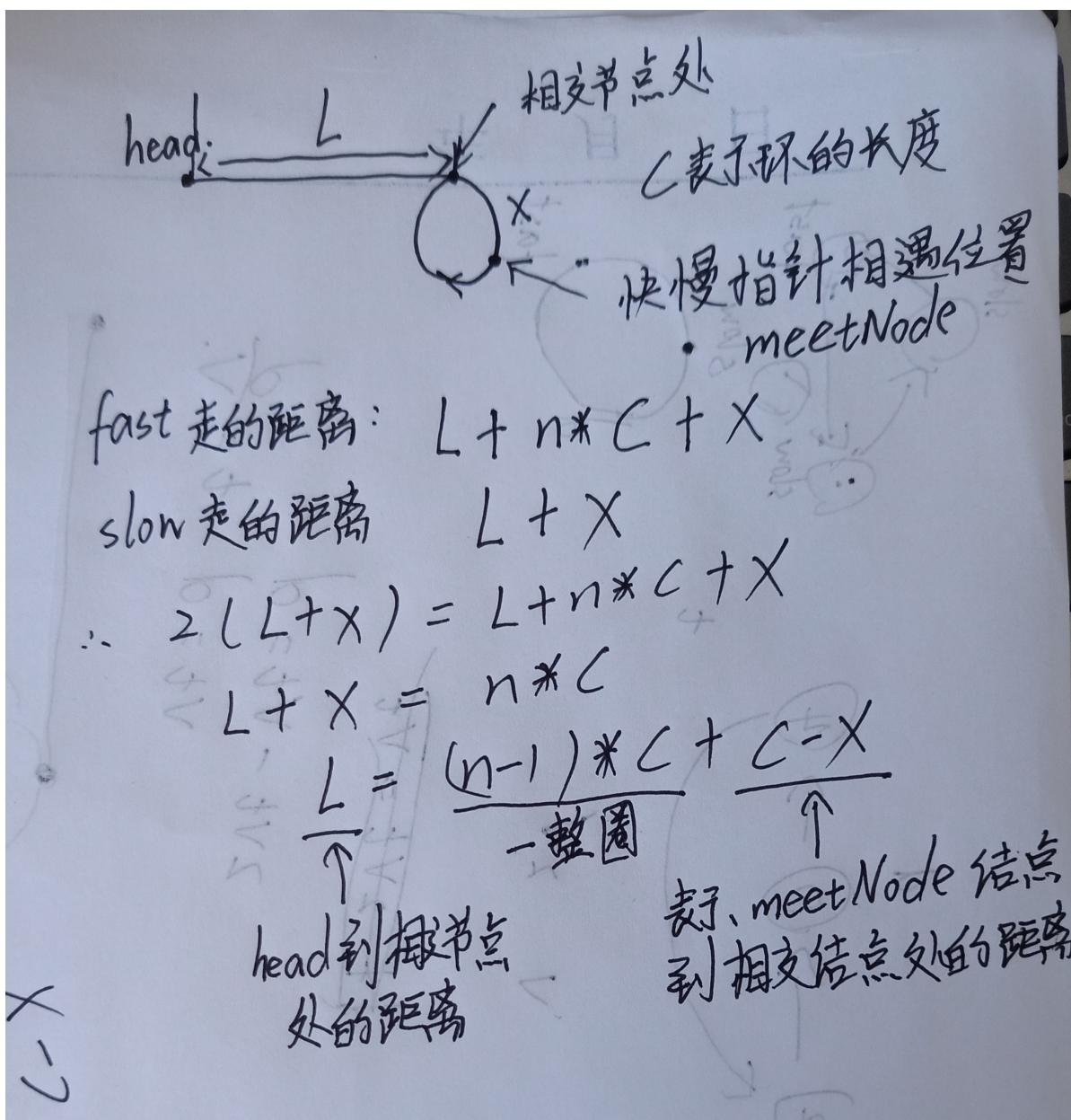
- 利用快慢指针求解：获得fast和slow相遇的节点meetNode
- 利用一个结论：两个节点，一个从头结点head出发，一个从meetNode节点出发，两个节点速度相同，最后会在环形链表的相交节点处（入环的第一个节点）相遇。

```

ListNode* fast=head,*slow=head;//定义快慢指针
while(fast&&fast->next){
    fast=fast->next->next;
    slow=slow->next;
    if(slow==fast){
        ListNode* meetNode=fast;//获得快慢指针相遇的节点
        while(meetNode!=head){ //利用结论获得相交的节点处（入环的第一个节点）
            meetNode=meetNode->next;
            head=head->next;
        }
        return meetNode;
    }
}

```

以下是手写证明：



## 复制带随机指针的链表

[复制带随机指针的链表 leetcode](#)

1. 先将新拷贝的结点链接在原来结点的**后边**
2. 通过原链表来改变拷贝链表的随机指针 (**最关键的一步**)

```
copyNode->random=cur->random->next;//cur是原链表的结点, copyNode是拷贝的结点
```

3. 将拷贝链表和原链表还原

## 串的模式匹配

1. BF算法(暴力算法)

2. [KMP算法](#)

求next的算法(关键代码):

```
int* GetNext(string t) {
    int len = t.size(); //获得模式串的长度
    int* next = new int[len]; //开辟一段数组
    int index = 0; //表示数组下标,每一次循环要求的是next[index+1]中的值
    int j = -1; //表示next数组中的值
    next[0] = -1; //next[0]赋初值
    while (index < len) {
        if (j == -1 || t[index] == t[j]) {
            j++;
            index++;
            next[index] = j;
        }
        else {
            j = next[j];
        }
    }
    return next;
}
```

```
//KMP算法测试
int main() {
    string sstring;
    string t;
    getline(cin, sstring); //原串
    getline(cin, t); //模块串
    int i = 0, j = 0, slen, tlen, *next;
    slen = sstring.size(); //获得原串的长度
    tlen = t.size(); //模块串的长度
    next = GetNext(t); //获得next数组
    while (i < slen && j < tlen) {
        while ((sstring[i] == t[j]) && (j < tlen)) {
            i++;
            j++;
        }
        if (j == tlen) {
            cout << "已经找到字串, 位置为" << i - j << endl;
            break;
        }
    }
    else {
        if (next[j] == -1) { //表示已经到达模块串的第一位, 需要将原串的指针移动到下一位
            j = 0;
        }
    }
}
```

```

        i++;
    }
    else {
        j = next[j];
    }
}

}
return 0;
}

```

## Huffman算法(哈夫曼算法)

通过生成哈夫曼树(最优二叉树)来进行哈夫曼编码

1. 将有权值的叶子结点按照从小到大的顺序排列
2. 取两个最小权值得结点作为新结点得左右孩子，小的为左孩子，大的为右孩子
3. 将新结点加入有序排列，继续重复步骤二

Huffman编码的**平均编码长度**计算：

先通过哈夫曼算法构造出最优二叉树后，判断每一个字符的编码长度，最后将**编码长度**乘以每个字符出现的概率求和

## 高精度加法

1. 因为输入的数大于long long了，所以就用string先存着；
2. 将string里存的数逆序存入数字数组，这样模拟手工从右往左计算过程。
3. 循环（长的那个数组有多少个数，就循环多少次），两数相加，如果数>10，那就保留各位，十位加到下一个数中。
4. 因为数逆序存入所以要逆序输出。

```

string s1,s2;
int a[250],b[250],c[500];

int main()
{
    cin>>s1>>s2;

    for(int i=0;i<s1.size();i++) //将s1字符串逆序存入数组a,将s2字符串逆序存入数组b
    {
        a[s1.size()-i-1]=s1[i]-'0';
    }

    for(int i=0;i<s2.size();i++)
    {
        b[s2.size()-i-1]=s2[i]-'0';
    }
}

```

```

int len=s1.size();
if(s2.size()>len)
{
    len=s2.size();
}

for(int i=0;i<len;i++)
{
    c[i]=a[i]+b[i];
}

for(int i=0;i<len;i++) //对进位进行处理
{
    if(c[i]>=10) c[i+1]=c[i+1]+c[i]/10;
    c[i]=c[i]%10;
}

if(c[len]!=0) len++; //如果最高位有进位，那么c[len]还会有值

for(int i=len-1;i>=0;i--)
{
    cout<<c[i];
}
cout<<endl;
return 0;
}

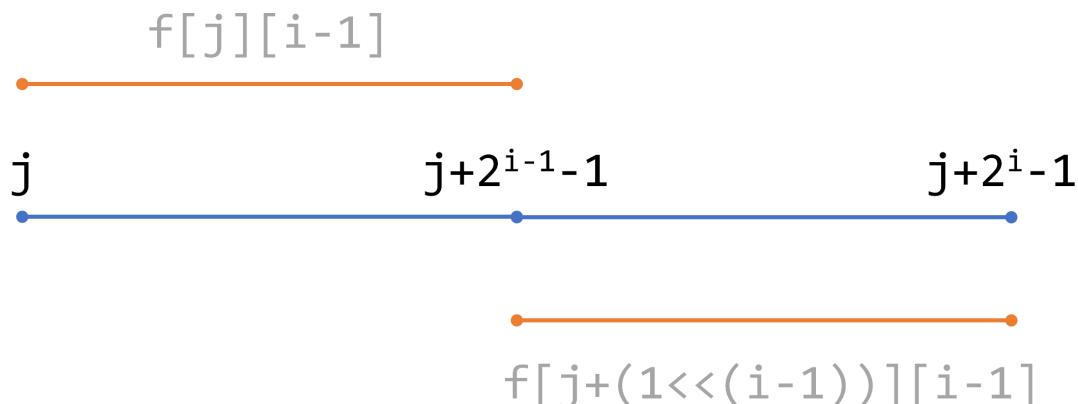
```

## st表(Sparse Table, 稀疏表)

st表是一种数据结构，主要用于解决RMQ（区间最大值或最小值）例如：给你一个数列，求解在一个范围内的数值的最大值或最小值。

主要利用了**倍增的思想**和**动态规划的思想**。

1. 动态规划的**预处理**（以2为倍数增加长度）



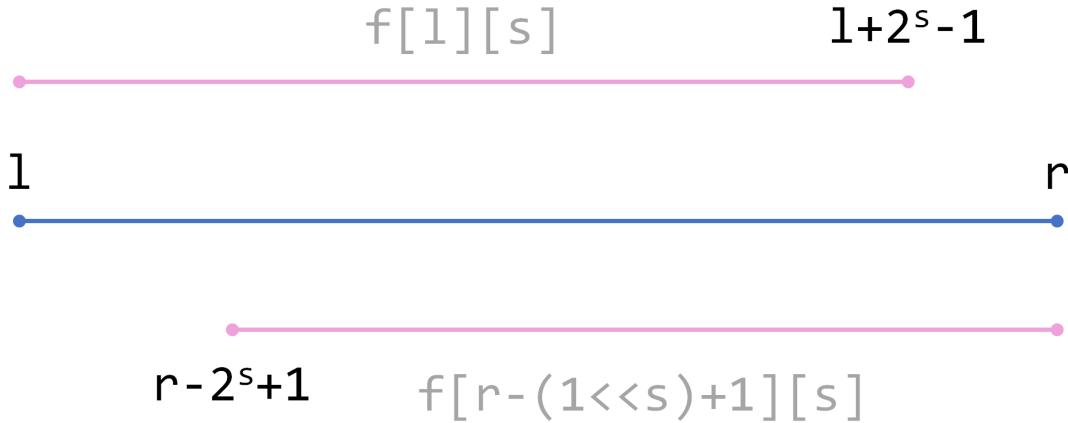
```

//由上图得，将要求得一个区间分为两个区间
//f[j][i]存的是从 j 到 j+2^i-1 范围内的 最大值，中间包含2^i个数

for(int i=1;i<=log2(n);i++){ // <<是移位运算符，1<<i相当于2^i
    for(int j=1;j+(1<<i)-1<=n;j++){
        f[j][i]=max(f[j][i-1],f[j+(1<<(i-1))][i-1]);
    }
}

```

## 2. 进行区间查询



找到一个值s，使得 $l+2^s-1$ ，尽可能接近r， $r-2^s+1$ 尽可能接近l，两个区间的长度都是 $2^s$   
最后比较两个区间的最大值，取较大的那个

```
max(f[1][s], f[r-(1<<s)+1][s]);
```

## 归并排序 稳定排序

时间复杂度O(nlogn) 空间复杂度O(n)

就是给定两个有序数组，将两个数组合并在一起升序。

定义一个更大的数组，给定两个指针分别指向两个数组，每次取较小值放入新数组。

```

//1. 分离函数
void mergesort(int x,int y)           //分离，x 和 y 分别代表要分离数列的开头和结尾
{
    if (x>=y) return;                //如果开头 ≥ 结尾，那么就说明数列分完了，分的只有一个数了，就
    返回
    int mid=(x+y)/2;               //将中间数求出来，用中间数把数列分成两段
    mergesort(x,mid);              //左右两端继续分离
    mergesort(mid+1,y);
    merge(x,mid,y);               //分离玩之后就合并，升序排序，从最小段开始
}

```

//2. 合并算法

```

void merge(int low,int mid,int high) //将两段的数据合并成一段，每一段数据都已经升序排序
{
}

```

```

int i=low,j=mid+1,k=low;
//i, j 分别标记第一和第二个数列的当前位置, k 是标记当前要放到整体的哪一个位置
while (i<=mid && j<=high) //如果两个数列的数都没放完, 循环
{
    if (a[i]<a[j])
        b[k++]=a[i++]; //a[n] (原始数组)和b[n] (临时存数据的数组)为全局函数
    else
        b[k++]=a[j++]; //将a[i] 和 a[j] 中小的那个放入 b[k], 然后将相应的标记变量
增加
    } // b[k++]=a[i++] 和 b[k++]=a[j++] 是先赋值, 再增加
    while (i<=mid)
        b[k++]=a[i++];
    while (j<=high)
        b[k++]=a[j++]; //当有一个数列放完了, 就将另一个数列剩下的数按顺序放好
    for (int i=low;i<=high;i++)
        a[i]=b[i]; //将 b 数组里的东西放入 a 数组, 因为 b 数组还可能要继
续使用
}

```

## 快速排序 不稳定排序

最差时间复杂度 $O(n^2)$ 和冒泡排序一样, 平均时间复杂度 $O(n \log 2n)$

递归算法:

```

void sort(int* a,int l,int r){
    if(l>r) return;
    int i=l,j=r;
    int std=a[l]; //最左端作为标准值 因为标准值是最左端的数, 所以要先从右边开始找
    while(i!=j){
        while(a[j]>=std&&i<j){ //从右往左 找到比标准小的数
            j--;
        }

        while(a[i]<=std&&i<j){ //从左往右 找到比标准大的数
            i++;
        }
        if(j>i){ //交换找到的两个值
            int t=a[i];
            a[i]=a[j];
            a[j]=t;
        }
    }
    //退出循环表示i==j, 将标准值换到i=j的地方, 继续递归运行
    a[l]=a[i];
    a[i]=std;
    sort(a,l,i-1);
    sort(a,i+1,r);
}

```

## 快速选择查找 (基于快速排序)

可以用来查找第k小(大)的数, 在快速排序每一轮确定基准值位置的时候判断是否是要选择的数

平均时间复杂度为 $O(n)$ , 最坏时间复杂度为 $O(n^2)$

```

//快速选择排序算法
int quickChoose(int left,int right,int k){
    if(left>right) return -1;
    int i=left,j=right;
    int std=num[left];
    while(i!=j){
        while(i<j&&num[j]>=std) j--;
        while(i<j&&num[i]<=std) i++;
        if(i<j){
            int t=num[i];
            num[i]=num[j];
            num[j]=t;
        }
    }
    num[left]=num[j];
    num[j]=std;
    if(k-1>j)quickChoose(j+1,right,k); //表示要查找的数在基准值位置的右边
    if(k-1<j)quickChoose(left,j-1,k); //在基准值的左边
    if(k-1==j) return num[k-1]; //返回第k小(大)的值
}

```

## 桶排序

不基于比较的排序算法

通过统计值域内每个数据的个数，然后根据个数排序

```

int count[1002];//存放数的个数 这里数的值域是[0,1000]
void bucketSort(int n){
    //统计每个数据的个数
    for(int i=0;i<n;i++){
        count[num[i]]++;
    }
    int cnt=0;
    //值域为[0,1000]
    for(int i=0;i<=1000;i++){
        while(count[i]!=0){
            num[cnt++]=i;//将数填回原数组
            count[i]--;
        }
    }
}

```

## 堆排序(升序) 不稳定排序

时间复杂度  $O(n \log n)$

空间复杂度  $O(1)$

```

//对某个根节点调整为大根堆
//从上往下进行调整
void adjust_down(int *a,int n,int i){ //i是需要调整的根节点的下标
    int father=i;
    int child=i*2+1;

```

```

while(child<n){
    //比较孩子结点的大小，选出较大的那个
    if((child+1)<=n-1&&a[child]<=a[child+1]) ++child;
    //交换父节点和孩子结点，并顺着孩子结点向下继续调整
    if(a[father]<a[child]){
        int temp;
        temp=a[child];
        a[child]=a[father];
        a[father]=temp;
        father=child;
        child=father*2+1;
    }else{
        //一旦不能继续调整就退出循环
        break;
    }
}

void heap_sort(int *a,int n){
    //建立大根堆
    //((n-1)/2)从后往前，第一个有孩子的结点
    for(int i=(n-1)/2;i>=0;i--){
        adjust_down(a,n,i);
    }
    //摘取大顶，与最后一个结点交换
    for(int i=0;i<n-1;i++){
        int temp=a[0];
        a[0]=a[n-i-1];
        a[n-i-1]=temp;
        adjust_down(a,n-i-1,0);
    }
}

```

## 并查集

一种树形的数据结构

主要构成 pre[]数组、find()、join()

作用：求图的连通分支数

```

//pre数组初始化
void Init(int n){
    //每个结点的祖先都是自己
    for(int i=0;i<n;i++){
        pre[i]=i;
    }
}

```

```

//pre[x]中存放的是x结点的父节点
//find()函数找到某个结点的根，结点的祖先
int find(int x) //查找某个结点的父节点
{
    while(pre[x] != x)
        x = pre[x];
    return x;
}

```

**find()函数优化：路径压缩。**就是将所有结点的父节点都改为根结点，这样子查找某个结点的父节点只需要向上查找一次

```

int find(int x) //查找结点 x的根结点
{
    if(pre[x] == x) return x; //递归出口：x的上级为 x本身，即 x为根结点
    return pre[x] = find(pre[x]); //此代码相当于先找到根结点 rootx，然后pre[x]=rootx
}

```

```

//将两个集合合并
void join(int x,int y)
{
    //找到各自的父节点
    int fx=find(x);
    int fy=find(y);
    //将fy和fx其中任意一个作为根
    if(fy!=fx)
        pre[fy]=fx;
}

```

## Kruskal算法--最小生成树

```

//并查集中的查找父节点函数
int find(int x){
    if(pre[x] == x) return x;
    return pre[x]=find(pre[x]); //路径压缩
}
struct edge{
    int u;
    int v;
    int w;
};
edge a[M]; //定义边的数组
//n是结点的个数，m是边的个数
int kruskal(int n,int m){
    int cnt=0; //统计加入生成树的边的条数
    for(int i=1;i<=m;i++){
        //找到边的两个端点的父节点，判断是否会造成环
        int father1=find(a[i].u); //find()函数，找到该节点对应的父节点(和哪些点连通) 具体见并查集
        int father2=find(a[i].v);
        //如果两个父节点相同，表示加入这条边会在图中形成环
        if(father1==father2){
            continue;
        }
        else{
            pre[father2]=father1;
            cnt++;
        }
    }
    if(cnt<n-1) return -1;
    return 0;
}

```

```

}else{
    //表示可以加入图
    pre[father1]=father2;//将两个结点的父节点相连(并查集中的合并操作)
    ans+=a[i].w;//ans统计最小生成树的权值和
    cnt++;
    if(cnt==n-1)break;
}
}

//输出最小生成树的长度和
if(cnt==n-1){
    cout<<ans;
}else{
    cout<<"orz";//表示不是连通图，不能构成生成树
}

}

```

```

//测试函数
#include<iostream>
#include<algorithm>
using namespace std;
const int N=5100;
const int M=200050;
int pre[N];//并查集
int ans; //统计最小生成树的长度和
bool compareEdge(edge a1,edge a2){//伪函数
    return a1.w<a2.w;//升序
}
int main(){
    int n,m;
    cin>>n>>m;
    for(int i=1;i<=m;i++){
        cin>>e[i].u>>e[i].v>>e[i].w;
    }
    sort(e+1,e+m+1,compareEdge);//排序
    //初始化pre数组
    for(int i=1;i<=n;i++){
        pre[i]=i;
    }
    kruskal(n,m);
    return 0;
}

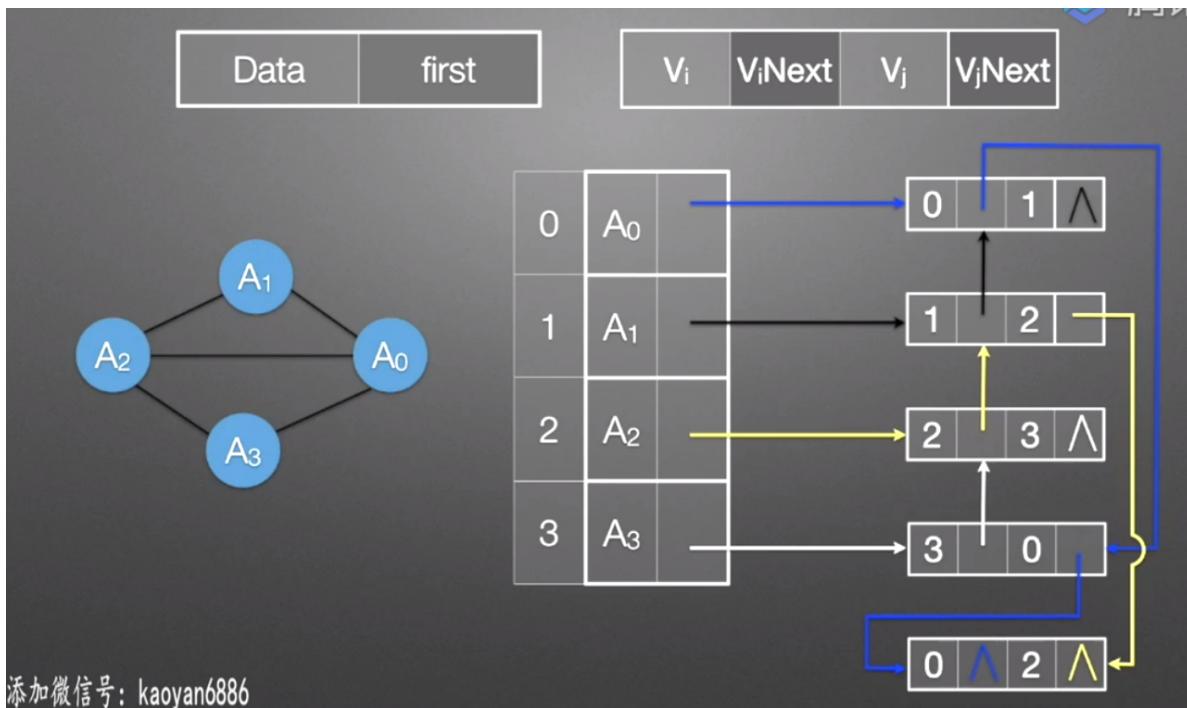
```

## 红黑树

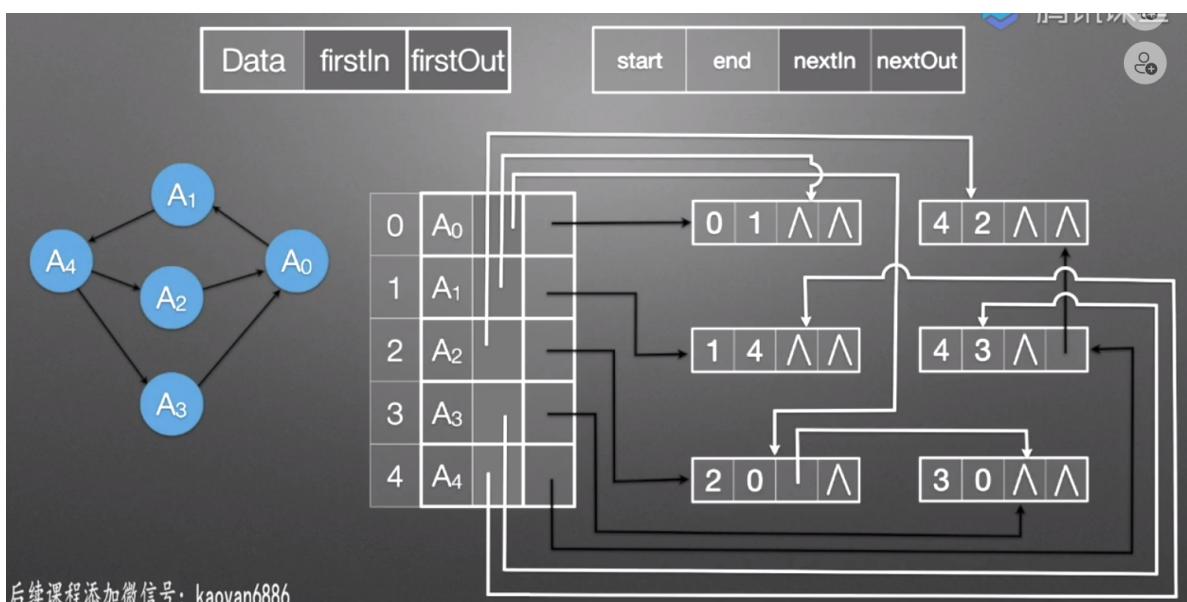
特征：

- 1.根节点是黑色的
- 2.红色结点和黑色结点交替
- 3.任意节点到叶子结点的路径包含相同数目的黑色结点 (红黑树中的叶子结点是指最外部的空结点)

## 多重邻接表(存储无向图)



## 十字链表(存储有向图)



## AOV网(解决拓扑排序问题)---活动赋予顶点

方式：每次选出入度为0的结点

用邻接矩阵 时间复杂度  $O(n^2)$

用邻接表 时间复杂度  $O(n+e)$

## 哈希表平均查找长度

成功时：ASL=(所有元素查找成功的比较次数)/元素的个数

哈希函数为  $H(key)=key \bmod p$  ( $p$ 一般来说是一个质数，除留余数法)

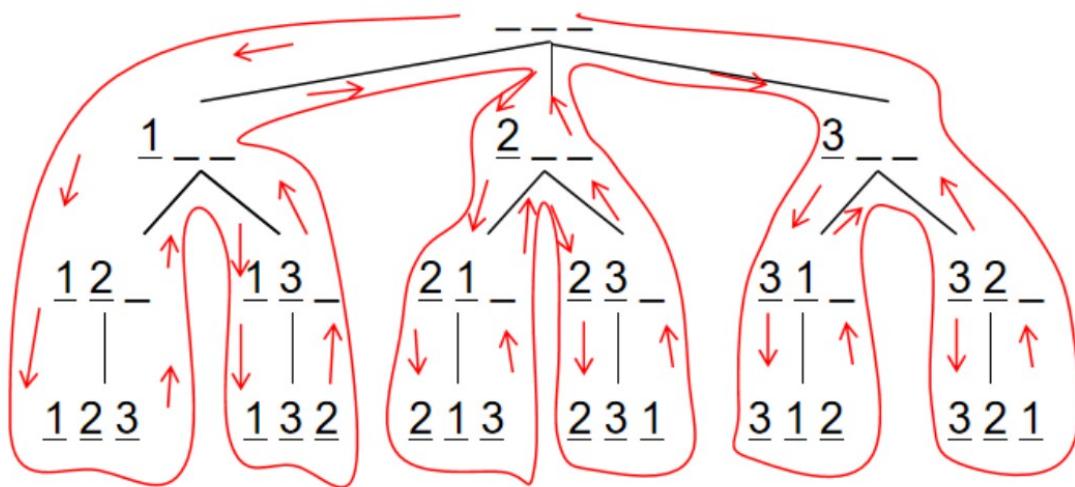
不成功时：ASL=(0到p-1向后查找直到遇到空的比较次数)/p (不是除以数组本身大小)

# 全排列问题

深度优先搜索思想DFS：

```
int arrange[N]; //存全排列的数组
int exist[N]; //判断是否存在
void createArrange(int k, int n){ //k是数组的位置，n是数的个数
    if(k>n){
        showArrange(n); //如果k>n表示所有数组位置都已经赋值，一次全排列情况结束，打印数组
        return;
    }else{
        for(int i=1; i<=n; i++){
            if(!exist[i]){
                arrange[k]=i; //如果有没被填入数组的数，就填入，然后标记已经选过
                exist[i]=1;
                createArrange(k+1, n); //进入递归
                exist[i]=0; //当递归退出表示一次全排列结束，会返回上一层递归的位置，执行下面
                的语句，将当前的数标为
                arrange[k]=0; //还没有选过
            }
        }
    }
}
```

递归情况如图：(以n=3为例)



# 遗传算法

遗传算法解决最优化问题的搜索算法

## 1. 基因编码

二进制编码 010010011011011110111110

浮点数编码 1.2 -3.3 -2.0 -5.4 -2.7 -4.3

## 2. 建立表现型到基因型的映射关系，就是将基因映射到一个区间范围内(有点像解码的过程)

## 3. 建立适应性函数，衡量物种是否能够存活的标准

4. 选择函数，物种繁殖的概率，一般适应度占比越大，被选择的概率就越大。使用轮盘赌

5. 遗传变异

交叉： **二进制编码**随机交换同一位置的的编码，产生新的个体

**浮点数编码**就产生介于父代和母代基因编码值之间的数

基因突变：二进制编码将某些位的基因改变

浮点是编码就是增加或减少一个小随机数

```
#解决函数求极值问题
# 遗传算法
import numpy as np

DNA_SIZE=8      #编码的位数
POP_SIZE=200    #种族的个数
CROSSVER_RATE=0.9      #交叉的概率
MUTATION_RATE=0.01    #变异的概率
N_GENERATIONS=5000    #迭代的次数
X_BOUND=[-3,3]    # x的取值范围
Y_BOUND=[-3,3]    # y的取值范围

def F(x,y):
    return (x+y)**2+np.sin(y)

# 得到最大适应度
def get_fitness(pop):
    x,y=translateDNA(pop)
    pred=F(x,y)#获得函数值
    return (pred-np.min(pred))+1e-3    #加上个较小值，防止适应度小于零
#将每条二进制编码的基因投影到定义域中
def translateDNA(pop):
    """
    解码
    :param pop: 种群矩阵，一行表示一个二进制编码的个体（可能解），行数为种群中个体数目
    :return: 返回的x,y 是一个行 为种群大小 列为 1 的矩阵 每一个值代表[-3,3]上x,y的可能取值
    (十进制数)
    """
    x_pop=pop[:,1::2]    # pop中的奇数列表示x    对每一个染色体，从奇数列位x，从1开始
    y_pop=pop[:,0::2]    # pop中的偶数列表示y
    #dot函数，获得两个数的乘积    arrange()生成范围内的数
    #-1表示逆序遍历列表
    x=x_pop.dot(2**np.arange(DNA_SIZE)[::-1])/float(2**DNA_SIZE-1)*(X_BOUND[1]-X_BOUND[0])+X_BOUND[0]
    y=y_pop.dot(2**np.arange(DNA_SIZE)[::-1])/float(2**DNA_SIZE-1)*(Y_BOUND[1]-Y_BOUND[0])+Y_BOUND[0]
    return x,y

# population_matrix = np.random.randint(2, size=(POP_SIZE, DNA_SIZE * 2))
# print(len(translateDNA(population_matrix)[0]))

# 交叉、变异
def crossover_and_mutation(pop,CROSSVER_RATE=0.8):
    #定义新的种族
    new_pop=[]
    #对每一挑染色体都进行交叉或变异
    for father in pop:    # 遍历种群中的每一个个体，将该个体作为父亲
        child=father      # 孩子先得到父亲的全部基因（代表一个个体的一个二进制0, 1串）
```

```

        if np.random.rand() <CROSSVER_RATE: #产生子代时不是必然发生交叉，而是以一定的概率发生交叉
            mother=pop[np.random.randint(POP_SIZE)] # 在种群中选择另一个个体作为母亲
            cross_points=np.random.randint(low=0,high=DNA_SIZE*2) #随机产生交叉的点 0-15
            child[cross_points:]=mother[cross_points:] #母亲交叉点往后全部给孩子
            mutation(child)
            new_pop.append(child)
        return new_pop

def mutation(child,MUTATION_RATE=0.1):
    if np.random.rand()<MUTATION_RATE:
        mutate_points=np.random.randint(0,DNA_SIZE*2) # 随机产生一个实数，代表要变异基因的位置
        child[mutate_points]=child[mutate_points]^1 # 和1异或 将变异点位置的二进制反转

def select(pop,fitness): # 自然选择，优胜劣汰
    #在两百条基因中中随机选出200条, replace=true表示可以有重复, p是每个元素采样的概率, 根据适应度所占总适应度得到比例
    #200个中随机一个染色体
    idx=np.random.choice(np.arange(POP_SIZE),size=POP_SIZE,replace=True,p=(fitness)/fitness.sum())
    return pop[idx]

def print_info(pop):
    fitness=get_fitness(pop)
    max_fitness_index=np.argmax(fitness)#返回列表中最大值得索引值
    # print('此时种群',pop)
    # print('max_fitness:',fitness[max_fitness_index])
    x,y=translateDNA(pop)
    # print('最优基因型: ',pop[max_fitness_index])
    # print('(x,y):',x[max_fitness_index],y[max_fitness_index])
    print('max_fitness:%s, 函数最大值:%s'%
(fitness[max_fitness_index],F(x[max_fitness_index],y[max_fitness_index])))

if __name__=='__main__':
    #获得200条长度为16的二进制编码, x,y各占8位
    pop=np.random.randint(2,size=(POP_SIZE,DNA_SIZE*2))
    for i in range(N_GENERATIONS):
        #x,y=translateDNA(pop)
        pop=np.array(crossover_and_mutation(pop)) # 交叉变异获得新的种群
        fitness=get_fitness(pop) # 得到适应度
        pop=select(pop,fitness) # 优胜劣汰
        if(i%100==0):
            print('第%s次迭代:'%i)
            print_info(pop)
    print_info(pop)

```

## 数学建模算法

梯度下降算法

决策树模型

梯度提升决策树算法GBDT

随机森林

集成学习思想

正则项函数

非线性多目标规划模型

强监督学习模型

Boosting算法

多项式拟合

加法模型

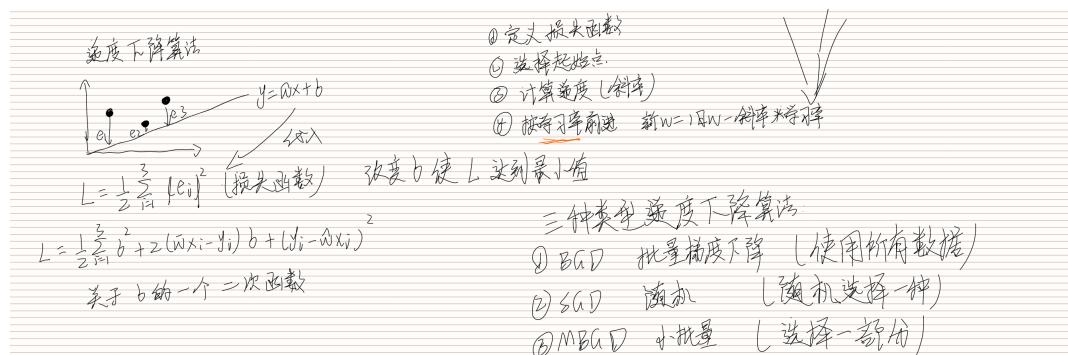
## 回归算法

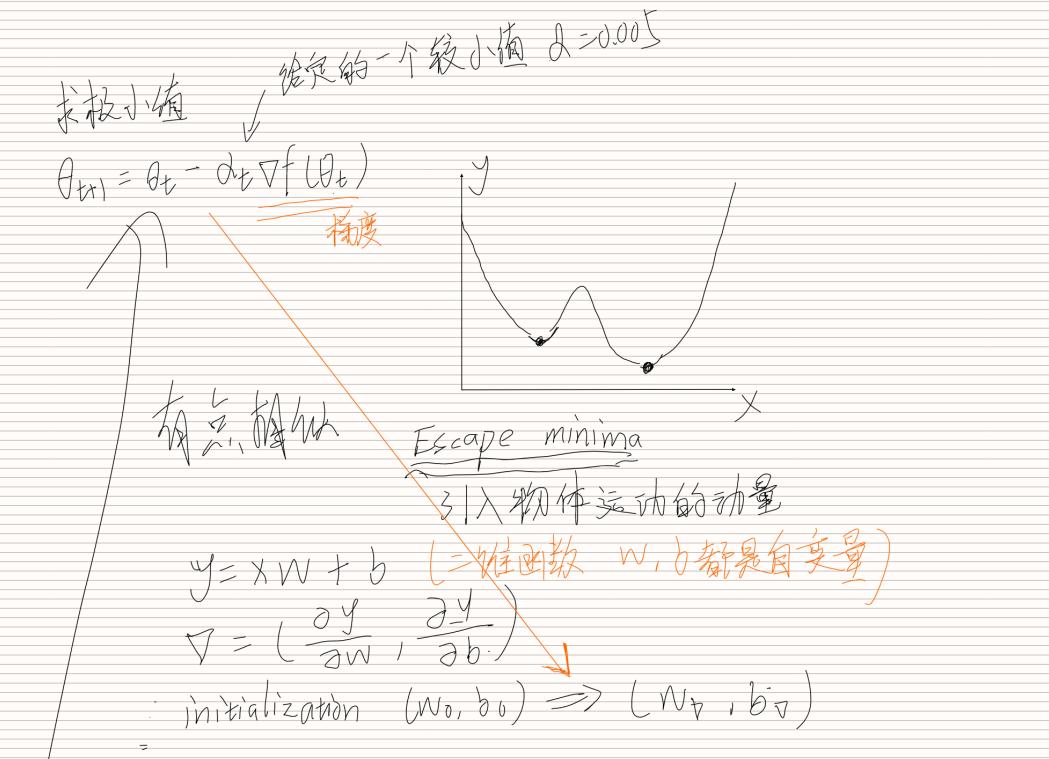
### 1. 线性回归

#### ◦ 利用梯度下降法求解

$$y = wx + b \quad (w, b \text{ 都是需要求解的变量})$$

1. 定义损失函数
2. 选择起始点 (假设令  $w=0, b=0$ )
3. 计算损失函数的梯度 (分别计算关于  $w$  和  $b$  的偏导数)
4. 按照学习率前进 (按照学习率重新获得新的  $w$  和  $b$  的值)
5. 求解当前  $y$  的均方误差和上一次的均方误差, 如果足够接近则表示已经获得相对正确的值





```

import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
# 线性回归利用梯度下降求解
path = 'D:\\学习\\数学建模\\LinearRegressionTest_data.txt'
data = pd.read_csv(path, header=None)
plt.scatter(data[:, 0], data[:, 1], marker='+')
data = np.array(data)
m = data.shape[0] # 获得数组的行数 shape[1] 可以获得数组的列数
theta = np.array([0, 0]) # 初始化两个 theta 的值
data = np.hstack(([np.ones([m, 1]), data])) # 将参数的元组水平方向叠加，相当于在所有数据前都加上了 1
y = data[:, 2] # 获得 y
data = data[:, :2] # 获得前面两列的数据

# 损失函数
def cost_function(data, theta, y):
    cost = np.sum((data.dot(theta) - y) ** 2) # 将 data 和 theta 两个矩阵相乘
    return cost / (2 * m)

# 求得梯度
def gradient(data, theta, y):
    grad = np.empty(len(theta)) # 创建未初始化的数组
    grad[0] = np.sum(data.dot(theta) - y)
    for i in range(1, len(theta)):
        grad[i] = (data.dot(theta) - y).dot(data[:, i])
    return grad

# 梯度下降法
def gradient_descent(data, theta, y, eta):
    while True:
        last_theta = theta
        grad = gradient(data, theta, y)
        theta = theta - eta * grad
        print(theta)
    
```

```

    if abs(cost_function(data, last_theta, y) - cost_function(data,
theta, y)) < 1e-15:
        break
    return theta

res = gradient_descent(data, theta, y, 0.00001)
x = np.arange(3, 25)
Y = res[0] + res[1] * x
plt.plot(x, Y, color='r')
plt.show()

```

### ◦ 最小二乘法

本质上就是直接利用导数求极值的方式直接获得最大值或最小值。与梯度下降法的区别就是，梯度下降法通过每次选择梯度最大的方向前进，不断改变w和b的值，直到达到相对最优解（可能会陷入局部最优解）

## 2. 曲线回归

### 3. 决策树回归

与分类树用基尼指数最小原则不同，对回归树用平方误差最小化准则

### 4. 随机森林回归

### 5. 梯度提升树回归

### 6. 保序回归

### 7. L1/2稀疏迭代回归

## 预测与评价方法

### 移动平均法(Moving Average)

用于对**平稳序列**进行预测

- **SMA(简单移动平均)**

**固定跨越期限内求平均值作为预测值**

- **WMA(加权移动平均)**

**加权移动平均给固定跨越期限内的每个变量值以不同的权重**

- **EMA(指数移动平均)**

## 预测与评价

指数移动平均

$$TEMA \rightarrow V_t = \begin{cases} 0 & t=0 \\ \beta V_{t-1} + (1-\beta) \theta_t & t \geq 1 \end{cases}$$

$\beta = 0.9$

当前项真实值  
 $\theta$ 为某一时刻真实值  
 $V$ 为某一时刻指数移动平均值

$$\textcircled{1} \quad \therefore V_{100} = 0.1 \theta_{100} + 0.9 V_{99}$$

$$\textcircled{2} \quad V_{100} = 0.1 \theta_{100} + 0.9 \times (0.1 \theta_{99} + 0.9 V_{98})$$

$$= 0.1 \theta_{100} + 0.9 \times 0.1 \theta_{99} + 0.9^2 V_{98}$$

$$\textcircled{3} \quad V_{100} = 0.1 \theta_{100} + 0.9 \times 0.1 \theta_{99} + 0.9^2 (0.1 \theta_{98} + 0.9 V_{97})$$

$$= 0.1 \theta_{100} + 0.9 \times 0.1 \theta_{99} + 0.9^2 \times 0.1 \theta_{98} + 0.9^3 V_{97}$$

同理可得

$$\textcircled{4} \quad V_{100} = 0.1 \theta_{100} + 0.9 \times 0.1 \theta_{99} + 0.9^2 \times 0.1 \theta_{98} + 0.9^3 \times 0.1 \theta_{97} + 0.9^4 V_{96}$$

$$V_{100} = 0.1 (\theta_{100} + 0.9 \theta_{99} + 0.9^2 \theta_{98} \dots 0.9^9 \theta_1) + 0.9^{100} \cancel{V_{96}}$$

本质上是加权平均

$$\lim_{\varepsilon \rightarrow 0} (1-\varepsilon)^{\frac{1}{\varepsilon}} = \frac{1}{\varepsilon} \quad \frac{1}{\varepsilon} \approx 0.36B$$

当权重  $< \frac{1}{\varepsilon}$  时，忽略当前项及之后的加权值

我们并不关心权重因子，而是指数移动平滑的步长 (权重因子  $B$  本质上即 权重系数  $\beta^T$  ( $T$  对应周期大小) 控制有效项的数目)

$$\therefore \beta = 1 - \varepsilon$$

$$\therefore \lim_{\beta \rightarrow 1} \beta^{\frac{1}{1-\beta}} = \frac{1}{\varepsilon} \Rightarrow T \approx \frac{1}{1-\beta}$$

$$\text{假设 } \beta = 0.9 \quad (0.9)^{10} \approx \frac{1}{\varepsilon} \quad \therefore T = 10$$

$$\text{假设 } \beta = 0.98 \quad (0.98)^{50} \approx \frac{1}{\varepsilon} \quad \therefore T = 50$$

直接代入  $V_t = \beta V_{t-1} + (1-\beta) \theta_t$  会与真实值有较大偏差

1. 引入偏差修正公式

$$V_t = \frac{V_t}{1-\beta^t}$$

随着权重因子  $\beta$  的增大 ( $\beta$  增大，表示先前的指数移动平均占比增大)，指数移动平均曲线逐渐变得更加平滑，但同时指数移动平均值的实时性 (当前值所占的比重减小，对平均值的影响减弱) 也随之变弱。

...

指数平均和引入偏差修正公式后的指数平均预测对比

...

```
import matplotlib.pyplot as plt
import numpy as np
```

```
def main():
```

```
    beta = 0.9      # 权重系数
    num_samples = 100  # 样本数量
```

```

# step 1 generate random seed
np.random.seed(0)
raw_tmp= np.random.randint(32, 38, size=num_samples)
x_index = np.arange(num_samples)
# raw_tmp = [35, 34, 37, 36, 35, 38, 37, 37, 39, 38, 37] # temperature
print(raw_tmp)

# step 2 calculate ema result and do not use correction
v_ema = []
v_pre = 0
for i, t in enumerate(raw_tmp):
    v_t = beta * v_pre + (1-beta) * t
    v_ema.append(v_t)
    v_pre = v_t #记录前一个指数移动平均数
print("v_me:", v_ema)

# step 3 correct the ema results
v_ema_corr = []
for i, t in enumerate(v_ema):
    v_ema_corr.append(t/(1-np.power(beta, i+1)))
print("v_ema_corr", v_ema_corr)

# step 4 plot ema and correction ema reslut
plt.plot(x_index, raw_tmp, label='raw_tmp') # Plot some data on the
(implicit) axes.
plt.plot(x_index, v_ema, label='v_ema') # etc.
plt.plot(x_index, v_ema_corr, label='v_ema_corr')
plt.xlabel('time')
plt.ylabel('T')
plt.title("exponential moving average")
plt.legend()
plt.savefig('./ema.png')
plt.show()

if __name__ == "__main__":
    main()

```

## 熵权法

算法步骤：

### 1. 数据归一化

对于正向指标：

$$x_{ij} = 0.998 \frac{x_{ij} - \min\{x_{1j}, x_{2j}, \dots, x_{nj}\}}{\max\{x_{1j}, x_{2j}, \dots, x_{nj}\} - \min\{x_{1j}, x_{2j}, \dots, x_{nj}\}} + 0.002$$

对于负向指标：

$$x_{ij} = 0.998 \frac{\max\{x_{1j}, x_{2j}, \dots, x_{nj}\} - x_{ij}}{\max\{x_{1j}, x_{2j}, \dots, x_{nj}\} - \min\{x_{1j}, x_{2j}, \dots, x_{nj}\}} + 0.002$$

其中系数0.998和0.002的目的是为了使  $x_{ij}$  的值大于0，防止在后续计算  $\ln x_{ij}$  时出现  $\ln 0$  的情况。这里的0.998可以更改成任意更接近于1的数，如：0.999, 0.997等。

2. 计算第j项指标下第i个方案的指标值比重

$$P_{ij} = \frac{x_{ij}}{\sum_{i=1}^n x_{ij}}, \text{ 且 } \sum P_{ij} = 1$$

3. 计算第j项指标的信息熵

$$e_j = -k \sum_{i=1}^n P_{ij} \ln P_{ij}, \text{ 其中 } k = \frac{1}{\ln n}$$

4. 计算各指标的权重

$$\omega_j = \frac{1-e_j}{\sum_{j=1}^m (1-e_j)}$$

5. 计算评价对象的综合评价值

$$S_i = \sum_{j=1}^m w_j P_{ij}.$$

## 模糊综合评价法

因素集 (评价指标集)  $U = \{u_1, u_2, \dots, u_n\}$

评语集 (评价的结果集)  $V = \{v_1, v_2, \dots, v_m\}$

权重集 (指标的权重)  $A = \{a_1, a_2, \dots, a_n\}$

评判结果集  $B = [b_1, b_2, \dots, b_n]$

### 1. 一级综合模糊评价

主要用于考核涉及指标较少

评价步骤:

1. 确定因素集
2. 确定评语集 (比如好、较好、中等、较差等评价)
3. 确定各因素的权重 (确定权重的方式可以是熵权法、层次分析法)
4. 确定模糊综合判断矩阵

$$R = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1m} \\ r_{21} & r_{22} & \cdots & r_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ r_{n1} & r_{n2} & \cdots & r_{nm} \end{pmatrix} = \begin{pmatrix} R_1 \\ R_2 \\ \vdots \\ R_n \end{pmatrix}$$

该矩阵是通过定性分析得出的，每个人对ui指标按照评语集V进行评价，并统计出每个评语所占的比重，获得R<sub>i</sub>

### 5. 模糊综合评判

综合评判的结果为  $B = A \cdot R$

$b_i$ 的含义是要评价的对象对于评语 i 的**隶属度**，选取隶属程度最大的就是评价对象的评价结果

### 2. 多层次模糊评价

主要用于考核涉及的指标复杂，具有多层次（某个指标可以分为更细的评价指标）

评价步骤与一级综合模糊评价类似，先对一级指标中的二级指标进行评判，得到的B就是更高层次评价的R（得到的评判结果作为高层次评价的判断矩阵），然后对一级指标重复上述步骤进行评价。

例如如下情况

$$\text{因素集 } U = \left\{ \begin{array}{l} \text{学习成绩 } U_1(0.4) \left\{ \begin{array}{l} \text{专业课成绩 } u_1^{(1)}(0.6) \\ \text{非专业课成绩 } u_2^{(1)}(0.4) \end{array} \right. \\ \text{竞赛成绩 } U_2(0.3) \left\{ \begin{array}{l} \text{国家级竞赛成绩 } u_1^{(2)}(0.5) \\ \text{省级竞赛成绩 } u_2^{(2)}(0.3) \\ \text{校级竞赛成绩 } u_3^{(2)}(0.2) \end{array} \right. \\ \text{个人荣誉 } U_3(0.2) \left\{ \begin{array}{l} \text{国家级奖项 } u_1^{(3)}(0.5) \\ \text{省级奖项 } u_2^{(3)}(0.3) \\ \text{校级奖项 } u_3^{(3)}(0.2) \end{array} \right. \\ \text{志愿服务 } U_4(0.1) \cdots \text{志愿服务时长 } u_1^{(4)}(1) \end{array} \right.$$

## 仿真方法

### 1. Monte Carlo Method (蒙特卡洛法)

蒙特卡罗方法又称统计模拟法、**随机抽样技术**，是一种随机模拟方法，多用于求解复杂的多维积分问题

每次输入都随机选择输入值。由于每个输入很多时候本身就是一个**估计区间**（可能是均匀分布，也可能是正态分布），因此计算机模型会随机选取每个输入的该区间内的任意值，通过大量成千上万甚至百万次的模拟次数，最终得出一个累计概率分布图

- 取舍采样 (rejection sampling)
- 马尔科夫链MC采样

### 2. 元胞自动机

