康托展开和逆康托展开

定义

康托展开（Cantor expansion）是一个**全排列**到一个自然数的**双射**，常用于**构建 hash 表**时的**空间压缩**.

设有 n 个数 $(1,2,3,4,\cdots,n)$，可以组成不同 $n!$ 种的排列组合，其康托展开**唯一且最大约为** $n!$.

康托展开表示的就是当前排列在 n 个不同元素的全排列中的名次。

时间复杂度为 $O(n^2)$

适用范围

搜索，动态规划中常常用一个数字来表示一种状态，大大降低空间复杂度

公式

$$X = a_1 \times (n-1)! + a_2 \times (n-2)! + \cdots + a_n \times 0!$$

其中 **X 代表当前排列在全排列中的排名**，a_i **代表当前数是数列中未出现的数中第几小的（从 0 开始计数）**

例如 4, 2, 3, 1，初始化 X=0

- 4是当前数列中未出现的数中第3小的, $X += 3 \times (4-1)!$
- 2是当前数列中未出现的数中第1小的, $X += 1 \times (4-2)!$
- 3是当前数列中未出现的数中第1小的, $X += 1 \times (4-3)!$,由于2已出现过
- 1是当前数列中未出现的数中第0小的, $X += 0 \times (4-4)!$

可求出 4, 2, 3, 1 所对应的唯一在全排列中的名次（此处 X 表示排名，排名是从 1 开始的，需将结果+1）：$X = 3 \times 3! + 1 \times 2! + 1 \times 1! + 0 \times 0! + 1 = 18 + 2 + 1 + 0 + 1 = 22$

注：每一次用到的是**当前有多少个小于它的数还没有出现**

代码实现

先预处理阶乘（能用循环写就不用递归写）

```
int fac[10];
void init()//初始化
{
    fac[0]=1;
    //递推求阶乘
    for(int i=1;i<=9;i++)
        fac[i]=fac[i-1]*i;
}
//或者直接打表
int fac[10]={1,1,2,6,24,120,720,5040,40320,362880};
```

康托 cantor 核心代码

```
int cantor(int n,int a[])//n 表示个数,a[]表示该排列顺序
{
    int res=0;
    for(int i=0;i<=n-1;i++)
    {
        int cnt=0;
        //内循环作用:找到 a[i]是当前数列中未出现的数中的第几小
        for(int j=i+1;j<=n-1;j++)
            if(a[j]<a[i])
                cnt++;
        res+=cnt*(fac[n-1-i]);
    }
    return res+1;
    //如果输出的是排名就要+1,普通的话就直接返回 res 即可
}
```

逆康托展开

因为排列的排名和排列是一一对应的，所以康托展开满足双射关系，是可逆的。可以通过类似上面的过程倒推回来。

首先把排名 X 减去 1，变为以 0 开始的排名

例如求 1,2,3,4 的全排列序列中，排名第 22 的序列是什么

22-1=21，21 代表着有 21 个排列比这个排列小

第一个数 a_1

$\left\lfloor \frac{21}{(4-1)!} \right\rfloor = 3$ 比 a_1 小且没有出现过的数有 3 个，$a_1 = 4$

$X = X \bmod [3 \times (4-1)!] = 21 \bmod 18 = 3$（求余过程可直接对阶乘求余）

第二个数 a_2

$\left\lfloor \frac{3}{(4-2)!} \right\rfloor = 1$ 比 a_2 小且没有出现过的数有 1 个，$a_2 = 2$

$X = X \bmod [1 \times (4-2)!] = 3 \bmod 2 = 1$

第三个数 a_3

$\left\lfloor \frac{1}{(4-3)!} \right\rfloor = 1$ 比 a_3 小且没有出现过的数有 1 个，$a_3 = 3$

$X = X \bmod [1 \times (4-3)!] = 1 \bmod 1 = 0$

第四个数 a_4

$\left\lfloor \frac{0}{(4-4)!} \right\rfloor = 0$ 比 a_4 小且没有出现过的数有 0 个，$a_4 = 1$

最终得到的数列为 4,2,3,1

逆康托代码

```
void incantor(int n,int x)
{
    x--;
    memset(vis,0,sizeof(vis));
    for(int i=0;i<=n-1;i++)
    {
        int cnt=x/fac[n-1-i];//比 ai 小且没有出现过的数的个数
        x%=fac[n-i-1];//更新 x
        for(int j=1;j<=n;j++)//找到对于的 ai,从 1 开始寻找
        {
            if(vis[j]==1)//被标记，就跳过
                continue;
            if(cnt==0)//cnt==0 代表当前数就是 ai
            {
                vis[j]=1;//标记
                res[i]=j;
                break;
            }
            cnt--;
        }
    }
}
```

前缀和和差分

前缀和和差分的关系：互为逆运算

设前缀和数组为 sum[ ]，原数组为 a[ ]，
则 sum[ ]是 a[ ]的前缀和数组，a[ ]是 sum[ ]的差分数组。

前缀和——前缀和是指序列的前 n 项和，可理解为数列的前 n 项和

问题引入：（一维前缀和）

输入一个长度为 n 的整数序列。接下来再输入 m 个询问，每个询问输入一对区间 l，r。对于每个询问，输出原序列中从第 l 个数到第 r 个数的和。

暴力的思想：跑 m 次，每次从区间左端点加到右端点，遍历区间求和，此时的时间复杂度为 $O(n\times m)$，如果 n 和 m 的数据量多起来就有可能 TLE。

如果使用前缀和可将时间复杂度降到 $O(n+m)$，大大提高运算效率。

具体做法：

1. 预处理
//注意有的题目需要开 long long （大致计算一下 n 个数×a 数组的最大范围会不会爆 int）

假设原数组为 a[ ]（该数组下标从 1 开始），前缀和一维数组为 sum[ ]，sum[i]表示 a 数组中前 i 个数的和。可写成 $sum[i]=sum[i-1]+a[i]$

```
const int MAXN=100010;
int a[MAXN],sum[MAXN];

//假设 a 数组已经输入(a 数组下标从 1 开始)
for(int i=1;i<=n;i++)
    sum[i]=sum[i-1]+a[i];
```

2. 查询

对于每次区间[l，r]的和查询，只需要计算 $sum[r]-sum[l-1]$，时间复杂度为 $O(1)$

原理：
$$\begin{aligned}&sum[r]=a[1]+a[2]+\cdots+a[l-1]+a[l]+a[l+1]+\cdots+a[r];\\&sum[l-1]=a[1]+a[2]+\cdots+a[l-1];\\&sum[r]-sum[l-1]=a[l]+a[l+1]+\cdots a[r].\end{aligned}$$

```
scanf("%d %d",&l,&r);
printf("%d\n",sum[r]-sum[l-1]);
```

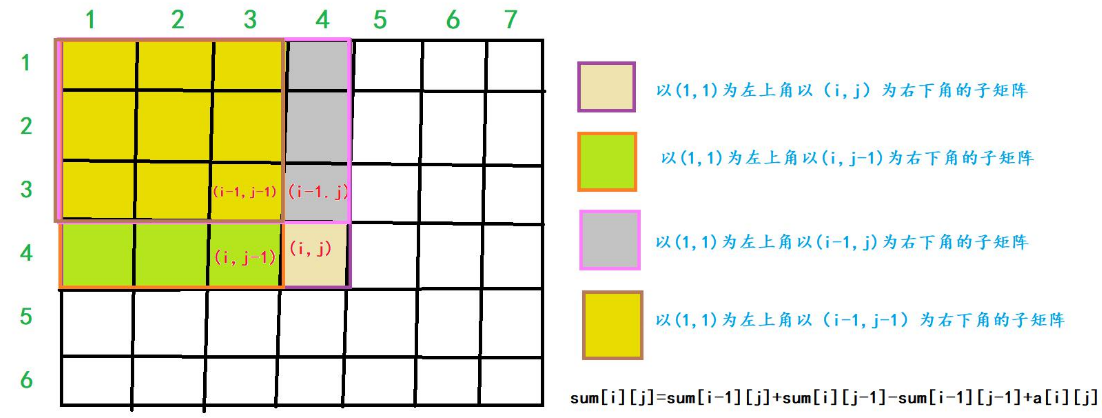
扩展：二维前缀和

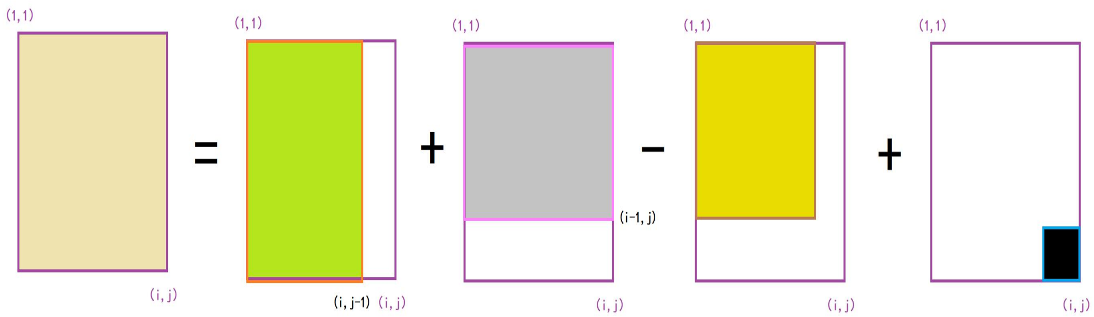
输入一个 n 行 m 列的整数矩阵，再输入 q 个询问，每个询问包含四个整数 x_1，y_1，x_2，y_2，表示一个子矩阵的左上角坐标和右下角坐标。对于每个询问输出**子矩阵中所有数的和**。

与一维前缀和类比，设前缀和二维数组为 sum[][]，其中 sum[i][j]表示的是以(1,1)为左上角到以(i,j)为右下角的子矩阵中的元素和。

1. 如何构造出 sum[][]数组？

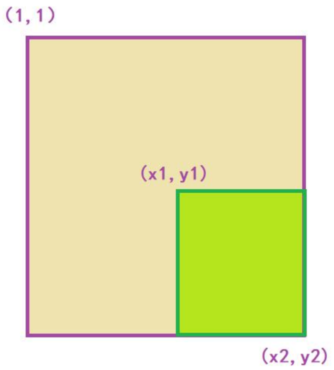
$$sum[i][j] = sum[i][j-1] + sum[i-1][j] - sum[i-1][j-1] + a[i][j]$$

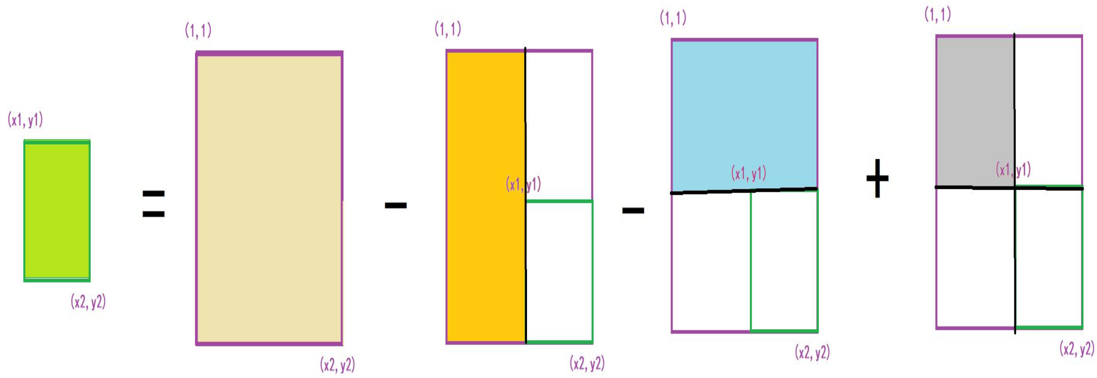




2. 如何查询到(x_1,y_1)为左上角(x_2,y_2)为右下角的子矩阵和？

$$res = sum[x_2][y_2] - sum[x_1-1][y_2] - sum[x_2][y_1-1] + sum[x_1-1][y_1-1]$$





差分——差分是前缀和的逆运算

差分数组：（先考虑一维差分）

首先给定一个原数组 a： $a[1], a[2], \cdots, a[n]$，

然后构造一个数组 d: $d[1], d[2], \cdots, d[n]$，使得 $a[i] = d[1] + d[2] + \cdots + d[i]$

a 数组是 d 数组的前缀和数组，反过来 d 数组就是 a 数组的差分数组。
每一个 a[i]都是 d 数组从 1 到 i 的区间和。

如何构造 d 数组?

$d[0] = 0$

$d[1] = a[1] - a[0]$

$d[2] = a[2] - a[1]$

$d[3] = a[3] - a[2]$

……

$d[i] = a[i] - a[i-1]$

如果有 d 数组，通过前缀和就可以在 $O(n)$ 的复杂度还原回 a 数组。

问题引入：（一维差分）

输入一个长度为 n 的整数序列。接下来再输入 m 次修改，每次修改需要将区间[l,r]的每一个数进行加（或减）一个数 c，最后输出原序列修改后的情况。

暴力的思想：for 循环 l 到 r 区间，时间复杂度 $O(n)$，如果我们需要对原数组执行 m 次这样的操作，时间复杂度就会变成 $O(n \times m)$。

这是就可以考虑采用**差分**实现。

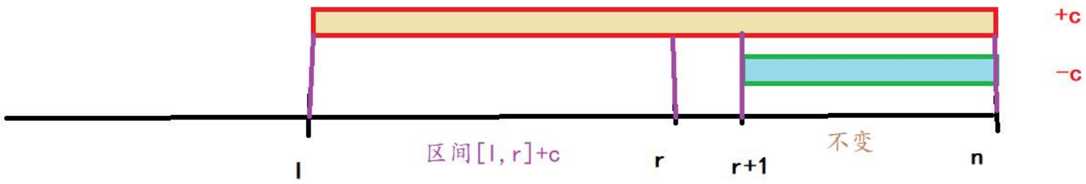
a 数组是 d 数组的前缀和数组，比如对 **d 数组的 d[i]的修改，会影响到 a 数组中从 a[i]及往后的每一个数**。

首先我们让差分数组 d 的 d[l]+c，通过前缀和运算，a 数组变为

$$a[1],a[2],\cdots,a[l]+c,a[l+1]+c,\cdots,a[n]+c$$

然后再对 d[r+1]-c,通过前缀和运算,a 数组变为

$$a[1],a[2],\cdots,a[l]+c,a[l+1]+c,\cdots,a[r]+c,a[r+1],\cdots a[n]$$



即给 a 数组中的[l, r]区间中的每一个数都加上 c,只需对差分数组 d 做 **d[l] + = c, d[r+1] - = c**。时间复杂度为 O(1)，大大提高了效率。

```
const int MAXN=100010;
int a[MAXN],d[MAXN];
//假设 a 数组已经输入(a 数组下标从 1 开始)
//1.构造差分数组
for(int i=1;i<=n;i++)
    d[i]=a[i]-a[i-1];
//2.修改操作
scanf("%d %d %d",&l,&r,&c);
d[l]+=c;
d[r+1]-=c;
//3.通过前缀和还原原数组
for(int i=1;i<=n;i++)
    a[i]=a[i-1]+d[i];
```

扩展：二维差分

如果扩展到二维，我们需要让二维数组被选中的子矩阵中的每个元素的值加上 c,是否也可以达到 O(1)的时间复杂度。答案是可以的，考虑**二维差分**。

假设 a[][]数组是 d[][]数组的前缀和数组，那么 d[][]数组就是 a[][]的差分数组。

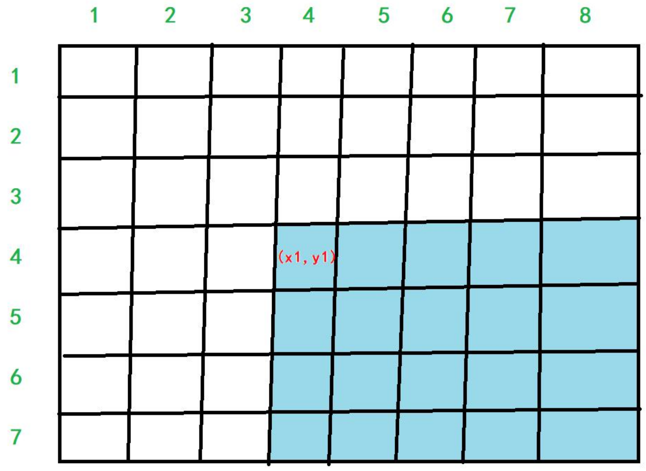
原数组:a[i][j]　构造差分数组:b[i][j]使得数组 a 中 a[i][j]是以(1,1)为左上角(i,j)为右下角的子矩阵的和。

如何构造差分数组 d?

其实关于差分数组，我们并不用考虑其构造方法，可使用**差分操作在对原数组进行修改**的过程中，实际上就可以构造出差分数组。

如何对子矩阵中的均加（或减）上一个数?

a 数组是 d 数组的前缀和数组,比如对 **d 数组的 d[i][j]的修改,会影响到 a 数组中从 a[i][j] 及往后的每一个数**。



类比一维差分，执行以下操作使得选中的子矩阵的每个元素的值都加（或减）上 c

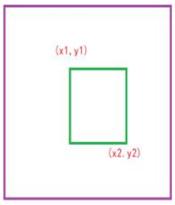
$d[x_1][y_1]+=c;$

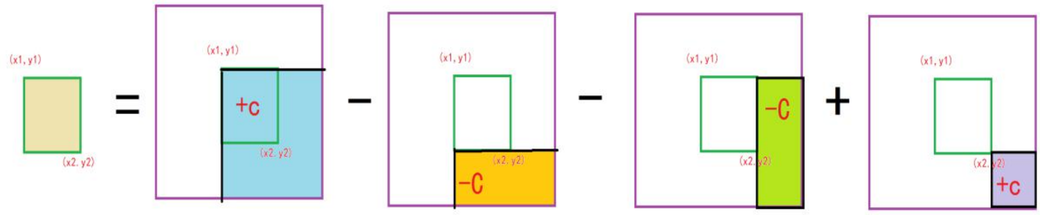
$d[x_1][y_2+1]-=c;$　　效果等价于：

$d[x_2+1][y_1]-=c;$

$d[x_2+1][y_2+1]+=c;$

```
    for(int i=x1;i<=x2;i++)
        for(int j=y1;j<=y2;j++)
            a[i][j]+=c;
```





将上述操作封装成一个插入函数:

```
void insert(int x1,int y1,int x2,int y2,int c)
{
    d[x1][y1]+=c;
    d[x2+1][y1]-=c;
    d[x1][y2+1]-=c;
    d[x2+1][y2+1]+=c;
}
```

我们可以先假想 a 数组为空，那么 d 数组一开始也为空，但是实际上 a 数组并不为空，因此我们每次让以(i,j)为左上角到以(i,j)为右下角面积内元素(其实就是一个小方格的面积)去插入 c=a[i][j]，等价于原数组 a 中(i,j) 到(i,j)范围内加上了 a[i][j] ,因此执行 $n \times m$ 次插入操作，就成功构造了差分 d 数组。
亦根据二维前缀和数组求法也可逆推出差分数组的公式

$$\text{前缀和公式}: sum[i][j] = sum[i][j-1] + sum[i-1][j] - sum[i-1][j-1] + a[i][j]$$

故二维差分数组构造的公式**如下：**

$$d[i][j] = a[i][j] - a[i][j-1] - a[i-1][j] + a[i-1][j-1];$$

```
    for(int i=1;i<=n;i++)
        for(int j=1;j<=m;j++)
            insert(i,j,i,j,a[i][j]);
            //d[i][j]=a[i][j]-a[i-1][j]-a[i][j-1]+a[i-1][j-1];
```

同理原数组 a 可借助二维前缀和公式还原（此处就不写了）。

EOF 的使用

1. scanf 的返回值等于成功输入的个数（成功读取的数据的个数）

scanf 函数返回成功读入的数据项数，读入数据时遇到了“文件结束（end of file）”返回 EOF

```
/******************************************/
scanf("%d",&a);//正常取值返回 1
/******************************************/
scanf("%d,%d",&a,&b);//正常取值返回 2
//如果输入 “1,2” ，scanf 函数将返回 2
//如果输入 “1 2” ，scanf 函数将返回 1，非正常取值
//如果输入 “,1 2”，scanf 函数将返回 0，非正常取值
```

2. 文件结束标志用 EOF（end of file）表示，多数编译器规定它的值为-1（EOF 实际上是在 #include<stdio.h>的库函数的定义的特殊值（符号常量，通常用#define 指令把 EOF 定义为-1））——在文件的最后的位置并未存储-1 作为结束标志，读文件时遇到文件结束标志便不再向后移动。

```
#define EOF -1
```

输入数据有多组，可以用如下两种方式实现结束

```
/*********************************/
while(scanf("%d",&a)!=EOF)
/*********************************/
while(~scanf("%d",&a))
//用按位取反符'~'来简化 EOF=-1
/*********************************/
```

取反符~的原理：

如果 scanf 函数的返回值为 4，原码二进制表示为 0000 0000 0000 0100，再取反得，~4 的补码二进制表示为 1111 1111 1111 1011，它的反码二进制表示为 1000 0000 0000 0100，它的原码二进制为 1000 0000 0000 0101（十进制为-5），只有当 while 里的表达式的值是 0 才结束循环，而此时 while 里的表达式不为 0，不结束循环

如果 scanf 函数的返回值是 EOF（即-1），补码二进制表示为 1111 1111 1111 1111，再取反得，~(-1)的补码二进制表示为 0000 0000 0000 0000（十进制为 0），此时 while 里的表达式的值是 0，结束循环