目录

[RMQ 2](#_Toc525659942)

[树状数组 3](#_Toc525659943)

[Dij 4](#_Toc525659944)

[Floyd 5](#_Toc525659945)

[Java大数 7](#_Toc525659946)

[归并排序逆序对 7](#_Toc525659947)

[Prim 8](#_Toc525659948)

[Kruskal 10](#_Toc525659949)

[Trie 11](#_Toc525659950)

[KMP 13](#_Toc525659951)

[Manacher算法 15](#_Toc525659952)

[AC自动机 16](#_Toc525659953)

[后缀数组 18](#_Toc525659954)

[线段树 21](#_Toc525659955)

[滑动窗口 23](#_Toc525659956)

[生成排列 25](#_Toc525659957)

[质数筛 26](#_Toc525659958)

[快速幂取模 27](#_Toc525659959)

[欧拉函数 28](#_Toc525659960)

[乘法逆元 29](#_Toc525659961)

[阶乘质因子分解 30](#_Toc525659962)

### RMQ

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

using namespace std;

#define num 100001//数组最大长度

const int LOG=((int)log2(num))+1;//数组长度以2为底的对数

/\*

数组下标从1开始

注意老的编译器不支持log2，通过类似(1<<(j++))<=num来确定下标即可

\*/

int x[num];

int MAX[num][LOG];//注意老的编译器可能不支持数组长度这样确定，保险起见可以先算出LOG的常量长度

int MIN[num][LOG];

int len;//数组实际使用长度

void init()//初始化数组后别忘了init！

{

int f=(int)log2(len)+1,i,j;

for(i=1;i<=len;i++)

{

MAX[i][0]=x[i];

MIN[i][0]=x[i];//以i开头长度为2^0的最值当然是自己

}

for(j=1;j<=f;j++)

{

for(i=1;i+(1<<j)-1<=len;i++)//注意范围，右坐标未超出数组下标才有计算的意义

{

MAX[i][j]=max(MAX[i][j-1],MAX[i+(1<<(j-1))][j-1]);

MIN[i][j]=min(MIN[i][j-1],MIN[i+(1<<(j-1))][j-1]);//划分成长度相等的两个子区间

}

}

}

int answer(int i,int j,bool what)//求i到j最值，what为1表示求最大值，为0表示求最小值

{

int k=(int)log2(j-i+1);

if(what)

{

return max(MAX[i][k],MAX[j-(1<<k)+1][k]);

}

else

{

return min(MIN[i][k],MIN[j-(1<<k)+1][k]);

}

}

int main()

{

len=10000;

return 0;

}

### 树状数组

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

/\*

树状数组里前缀和是从1开始算的

x[i]本身没什么意义

要求原数组值使用sum(i)-sum(i-1)

\*/

using namespace std;

#define lowbit(x) (x&-x)

int MAX;//树状数组有意义部分最大下标

int x[100001];

int sum(int n)//求前缀和

{

int ans=0;

while(n>0)

{

ans+=x[n];

n-=lowbit(n);

}

return ans;

}

void add(int p,int a)//修改值，p位置，a加上的值

{

if(p<1)

{

return;

}

while(p<=MAX)

{

x[p]+=a;

p+=lowbit(p);

}

}

int main()

{

return 0;

}

### Dij

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

#include<vector>

#include<queue>

using namespace std;

/\*

本模板点编号从0开始和从1开始都行

\*/

#define MAXN (int)(1e6+1)//最大点数

#define INF 0x3f3f3f3f

struct Edge

{

int from,to,value;

Edge(int \_from,int \_to,int \_value):from(\_from),to(\_to),value(\_value){}

};

struct Qnode//为使用优先队列设置的

{

int point;//节点

int dis;//当前最短距离

Qnode(int \_point,int \_dis):point(\_point),dis(\_dis){}

bool operator < (const Qnode &right) const

{

return dis>right.dis;

//优先队列中是用<比较，将“大”的放堆顶，此处将它意义变反，实现距离小的放堆顶

}

};

struct Dijkstra

{

int n,m;//点数和边数

vector<Edge> edges[MAXN];//每个点出发的所有边，点从0开始编号

int dis[MAXN];//源点到每个节点的最短距离

bool vis[MAXN];//标记节点是否已求解

void clear()//便于重复使用

{

for(int i=0;i<=n;i++)//这样编号从0开始和从1开始都OK

{

edges[i].clear();

}

}

void Solve(int S)//求解从S出发到所有点的最短路

{

int i,j;

int from,to,dis1,dis2;

Qnode tem(0,0);

memset(vis,0,sizeof(vis));

memset(dis,0x3f,sizeof(dis));//初始各点最短路径置为无穷大

priority\_queue<Qnode> pq;

while(!pq.empty())

{

pq.pop();//保险？

}

dis[S]=0;

pq.push(Qnode(S,0));

while(!pq.empty())

{

tem=pq.top();

pq.pop();//当前距离最小的点已经得解了

if(vis[tem.point])//防止是垃圾数据

{

continue;

}

from=tem.point;//当前点

dis1=tem.dis;//当前点最短距离

vis[from]=1;//置为已求解

for(i=0;i<edges[from].size();i++)//该点出发能到达的所有点

{

to=edges[from][i].to;//下一个点

dis2=dis1+edges[from][i].value;//以当前点为中介点时的距离

if(!vis[to]&&dis2<dis[to])//点未被求解且以当前点为中介能达到更短的距离

{

//若想记录路径，在这里使用path数组记录当前最短路中当前节点的下一节点或者上一节点即可

dis[to]=dis2;

pq.push(Qnode(to,dis2));

//尽管放进去，to会重复也没关系，因为每次取出时都会检查该点是否已求解

}

}

}

}

};

int main()

{

return 0;

}

### Floyd

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

/\*

假设点编号从1开始

\*/

using namespace std;

#define MAXN (int)(1e3+1)

#define INF 0x3f3f3f3f

int x[MAXN][MAXN];

void Floyd(int n)

{

int i,j,k;

for(k=1;k<=n;k++)//选取每一个节点作为中间节点

{

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

if(x[i][k]+x[k][j]<x[i][j])

{

x[i][j]=x[i][k]+x[k][j];

}

}

}

}

}

inline void add(int i,int j,int v)

{

x[i][j]=v;

x[j][i]=v;

}

int main()

{

int i,j;

memset(x,0x3f,sizeof(x));

for(i=1;i<=3;i++)

{

x[i][i]=0;

}

add(1,2,1);

add(2,3,2);

Floyd(3);

for(i=1;i<=3;i++)

{

for(j=1;j<=3;j++)

{

printf("%d ",x[i][j]);

}

putchar('\n');

}

return 0;

}

### Java大数

package acm;

import java.math.BigInteger;

import java.util.Scanner;

public class Test

{

public static void main(String[] args)

{

Scanner input=new Scanner(System.in);

//构造

BigInteger x=BigInteger.valueOf(1000000000L);//由long型转换而来

x=new BigInteger("10000000",10);//由字符串转换而来，后一个参数为进制，省略默认为10

x=input.nextBigInteger(10);//直接读入，参数为进制，可省略

//运算等

x=x.add(x);

x=x.divide(x);

x=x.mod(x);//取余

x=x.gcd(x);//求gcd

x=x.negate();//取相反数

x.compareTo(x);//比较大小

x.isProbablePrime(100);//判断是否是质数，正确率为1-0.5^参数

x.toString(进制);//转换成任意进制表示的字符串

System.out.println(x);

input.close();

}

}

### 归并排序逆序对

#include<iostream>

#include<stdio.h>

#include<string.h>

#include<stdlib.h>

#include<algorithm>

using namespace std;

/\*

归并排序，求逆序对对数

首先应知道，设[L1,R1]，[L2,R2]其中L2>R1

[L1,R1]中数的顺序改变，并不会改变一点在[L1,R1]中

另一点在其他区间的逆序对对数，因为两点相对位置（谁左谁右并没有变）

所以归并时，每当左区间头元素大于右区间头元素，则有逆序对

对数为左区间当前剩下的元素个数

\*/

#define MAXN 100001

int x[MAXN+1];

int y[MAXN+1];//辅助数组

long long Rnum;//逆序对对数

void Merge(int L,int R,int M)

{

//将x中L到M，M+1到R两部分，合并到y中的L到R里

int i,j,k,tem;

i=L,j=M+1,k=L;

while(i<=M&&j<=R)//两部分都还能取时

{

//取更小的放前面

if(x[i]<=x[j])

{

y[k++]=x[i++];

}

else//仅当左区间头元素更大时有逆序对（注意是严格大于，不能是大于等于！）

{

Rnum+=M-i+1;//左区间剩下的都比右区间头元素大

y[k++]=x[j++];

}

}

//现在要么二者都取完，要么只剩一部分未取完

while(i<=M)

{

y[k++]=x[i++];

}

while(j<=R)

{

y[k++]=x[j++];

}

for(i=L;i<=R;i++)

{

x[i]=y[i];//复制回去

}

}

void Part(int L,int R)//拆分

{

int M;

if(R>L)//长度大于1才需要拆分

{

M=(L+R)/2;

Part(L,M);

Part(M+1,R);

Merge(L,R,M);//先拆分成2部分，分别求解后再合并

}

}

int main()

{

int i,n;

scanf("%d",&n);

for(i=1;i<=n;i++)

{

scanf("%d",&x[i]);

}

Rnum=0;

Part(1,n);

printf("%lld",Rnum);

return 0;

}

### Prim

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

/\*

先取一个节点，然后不断加边

实现时类似Dijkstra算法

用lowC存储未求解的点距离当前树的最短距离

每次寻找lowC中的最小值，对应节点视为新求解

根据这个新求解的点，更新所有其他未求解的点的lowC

\*/

using namespace std;

#define MAXN (int)(1e3+1)//最多节点数

#define INF 0x3f3f3f3f//极大值

int cost[MAXN][MAXN];//耗费矩阵（边）

int lowC[MAXN];//每个节点离当前生成的树的最短距离

bool vis[MAXN];//某个节点是否在树中

int Prim(int n)//节点数，假设从0开始编号

{

memset(vis,0,sizeof(vis));

int i,j,MIN,ans=0,p;

vis[0]=1;//先将0号节点加入

for(i=1;i<n;i++)

{

lowC[i]=cost[0][i];//因为当前只有0号节点

}

for(i=1;i<n;i++)//这个只是次数，需要寻找剩余n-1个点

{

MIN=INF;

p=-1;

for(j=1;j<n;j++)//遍历未加入节点

{

if(!vis[j]&&lowC[j]<MIN)

{

MIN=lowC[j];

p=j;

}

}

if(p==-1)

{

return -1;

}

vis[p]=1;

ans+=MIN;

for(j=1;j<n;j++)//遍历未加入节点

{

if(!vis[j]&&cost[j][p]<lowC[j])

{

lowC[j]=cost[j][p];

}

}

}

return ans;

}

int main()

{

int n=3;

for(int i=0;i<3;i++)

{

cost[i][i]=0;

}

cost[0][1]=cost[1][0]=1;

cost[1][2]=cost[2][1]=1;

cout<<Prim(3);

return 0;

}

### Kruskal

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

using namespace std;

/\*

起初所有点都孤立

给边按权值从小到大排序

只要边的两个端点分别在不同的树中（加边后不会有环）就加入

无向图，加边加一条就OK

\*/

/\*

使用步骤

1.调整MAXN与MAXE

2.每次使用前令Enum=0，然后加边

3.调用Kruskal(n)，n为节点数，则返回最小生成树权值

点编号从0开始和从1开始都OK，因为体现在了edges数组中，无需手动调

\*/

#define MAXN (int)(1e3+1)//最大点数

#define MAXE (int)(1e3+1)//最大边数

struct Edge

{

int from,to,value;

};

Edge edges[MAXE];//所有的边

int Enum;//边数，使用前先初始化为0

int points[MAXN];//所有的点编号

void add(int f,int t,int v)//加边

{

edges[Enum].from=f;

edges[Enum].to=t;

edges[Enum].value=v;

Enum++;

}

int find(int n)

{

if(points[n]!=n)

{

return points[n]=find(points[n]);

}

return n;

}

bool CMP(Edge x,Edge y)

{

return x.value<y.value;

}

int Kruskal(int n)//n为节点数，编号从0开始还是从1开始都OK

{

int i,from,to,val,ans=0,num=0,f1,f2;

for(i=0;i<=n;i++)//这样点编号从0开始还是从1开始都OK

{

points[i]=i;//并查集初始化

}

sort(edges,edges+Enum,CMP);

for(i=0;i<Enum;i++)//遍历所有的边

{

from=edges[i].from;

to=edges[i].to;

val=edges[i].value;

f1=find(from);

f2=find(to);

if(f1!=f2)//加边不会产生环

{

points[f2]=f1;

num++;//

ans+=val;//加上权值

}

}

if(num!=n-1)//图不连通，边少了

{

return -1;

}

else

{

return ans;

}

}

int main()

{

Enum=0;

add(2,1,1);

add(2,3,1);

add(4,2,1);

add(4,5,1);

cout<<Kruskal(5);

return 0;

}

### Trie

#include<iostream>

#include<cstring>

#include<memory.h>

#include<algorithm>

#include<time.h>

using namespace std;

/\*

假设字典中单词只由26个小写字母构成

每个节点的子节点最多只可能有26个

为了方便，这棵树就由数组存储，大小开字符串最大长度\*26+1

每个节点都有编号，编号即数组下标

编号为0是根节点，单词信息是存在边上的

用Trie[i][j]=k表示编号为i的节点，第j号子节点编号为k

节点编号是随机的，根据读入顺序

这个第j号就对应一个字母（将字母按字典序从0开始编号）

如j=0，表示有字母a，也就是i->j这条边字母为a

当然，若不存在某个编号的子节点，设Trie[i][x]=0即可，因为没有任何一个节点的子节点会是根节点

从根节点开始往下找即可查询单词

有时对每个单词需要有一个权值，使用val[i]记录从根节点到i节点这个单词的权值即可

对于无效的节点统一设为一个不可能的权值即可

数组会很大，所以程序中只有在用上了某个节点时才memset

垃圾数据没关系的，因为每当首次用到一个节点就会先将其子节点置为0（根节点必然用到）

插入时反正都是当为0时断定无子节点

而查找时，从根节点开始，找到的节点之前必然已经插入过，也就是清空过垃圾数据了

只要Trie中找到了，val值才视为有效，其他垃圾数据也忽略

虽说各教程都说边代表字母，实际上将点看做字母更好理解（边的远离根的那个端点）

\*/

#define MAXLEN (int)(1e5+1)//单词最大长度

#define KINDS 26//不同字母个数

int Trie[MAXLEN\*KINDS][KINDS];//编号为i的节点，第j号子节点编号为k

int val[MAXLEN\*KINDS];

int Pnum;//节点数

void insert(char \* S,int v)//插入单词S，权值为v，覆盖式插入，若不想覆盖可先判断单词是否已存在

{

int i,tem;

int id=0;//顺着寻找下去的节点编号，起初为根

for(i=0;S[i]!='\0';i++)

{

tem=S[i]-'a';//当前需插入单词的编号

if(Trie[id][tem]==0)//原来没有，需要新插入

{

memset(Trie[Pnum],0,sizeof(Trie[Pnum]));

val[Pnum]=0;//中间节点权值设为无效，这样以后寻找时可以区分是否代表单词末尾

Trie[id][tem]=Pnum++;//新增节点

}

id=Trie[id][tem];//无论是否新建节点，都应更新

}

val[id]=v;

}

int find(char \* S)//查找S，找到则返回节点编号，否则返回-1

{

int i,tem;

int id=0;//顺着寻找下去的节点编号，起初为根

for(i=0;S[i]!='\0';i++)

{

tem=S[i]-'a';//当前需查找单词的编号

if(Trie[id][tem]==0)//找不到

{

return -1;

}

id=Trie[id][tem];//放心，Trie[id][tem]一定不会是垃圾数据因为id之前插入时memset过，非零数据必然有效

}

if(val[id]!=0)

{

return id;//确保是单词节点而不是中间节点

}

else

{

return -1;

}

}

int main()

{

Pnum=1;//起初只有根节点

memset(Trie[0],0,sizeof(Trie[0]));

return 0;

}

### KMP

#include<iostream>

#include<cstring>

#include<memory.h>

#include<algorithm>

#include<cmath>

/\*

下标从0开始

此处前后缀不包含自己！

字符串匹配时，假设到了模式串i位置不匹配，则i位置以前，双方是相等的

接下来对模式串移位，在起始移到i之前时，就相当于自己在匹配自己

而自己匹配自己在每个点上结果都是一样的，可以预处理记录下来加速匹配

假设有如下匹配

ababdefg

ababc

当前在c位置不匹配，那么下一步应该直接

ababdefg

ababc

即用模式串下标2位置与该字符比较（未优化KMP版本，新匹配字符可能会与之前匹配字符相等）

Next[i]表示当i位置失配时，下一次被匹配串该位置应该与模式串哪个位置的字符比较

实际意义为模式串开头到i-1位置的子串，前缀与后缀相同时的最大长度

此模板Next[0]=1而不是-1

假设某字符串循节长度为i，则有Next[i+1]\*2==i

之后i每加1，相应Next也加1

继续可推得i结尾的字符串循环节个数为(i+1)/( (i+1)-Next[i+1] )（前提是能整除！不能整除则为1）

即串的长度除以（串的长度减去串的最长前缀后缀公共长度，也就是循环节长度）

\*/

using namespace std;

int Next[10001];//若Next[i]+1=i，实际上公共长度为i

int x[10001];//模式串

int y[1000001];//被匹配

void Cal(int lenx)//计算Next数组，lenx为x长度

{

int i,j;

Next[0]=0,Next[1]=0;

for(i=1;i<lenx;i++)

{

j=Next[i];

while(j&&x[i]!=x[j])

{

j=Next[j];

}

Next[i+1]=x[i]==x[j]?j+1:0;

}

}

int KMP(int lenx,int leny)

{

Cal(lenx);

int i,j=0;

bool flag=1;

for(i=0;i<leny;i++)

{

while(j!=0&&x[j]!=y[i])

{

j=Next[j];

}

if(x[j]==y[i])

{

j++;

}

if(j==lenx)

{

flag=0;

return i-lenx+1;//输出起始位置

}

}

if(flag)

{

return -2;

}

}

int main()

{

int lenx,leny,i,T;

scanf("%d",&T);

while(T--)

{

scanf("%d%d",&leny,&lenx);

for(i=0;i<leny;i++)

{

scanf("%d",&y[i]);

}

for(i=0;i<lenx;i++)

{

scanf("%d",&x[i]);

}

printf("%d\n",KMP(lenx,leny)+1);

}

return 0;

}

### Manacher算法

#include<iostream>

#include<cstring>

#include<memory.h>

#include<algorithm>

#include<cmath>

/\*

Manacher算法将原串进行改造成Ma，在每两个字符间插入一个'#'

这样偶数长度的回文串也有了中心字符

计算后，原串中下标i映射到Ma中2\*(i+1)

Mp[j]-1即为原串中以j为中心的最长回文串长度

如

abaaba

i: 0 1 2 3 4 5 6 7 8 9 10 11 12 13

Ma[i]: $ # a # b # a # a # b # a #

Mp[i]: 1 1 2 1 4 1 2 7 2 1 4 1 2 1

\*/

using namespace std;

const int MAXN=500010;

char Ma[MAXN\*2];

int Mp[MAXN\*2];

void Manacher(char s[],int len){

int l=0;

Ma[l++]='$';

Ma[l++]='#';

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

Ma[l++]=s[i];

Ma[l++]='#';

}

Ma[l]=0;

int mx=0,id=0;

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

Mp[i]=mx>i?min(Mp[2\*id-i],mx-i):1;

while(Ma[i+Mp[i]]==Ma[i-Mp[i]])Mp[i]++;

if(i+Mp[i]>mx){

mx=i+Mp[i];

id=i;

}

}

}

int main()

{

return 0;

}

### AC自动机

#include<iostream>

#include<cstring>

#include<memory.h>

#include<algorithm>

#include<cmath>

#include<queue>

using namespace std;

struct Trie{

int next[500010][26],fail[500010],end[500010];

int root,L;

int newnode(){

for(int i = 0;i < 26;i++)

next[L][i] = -1;

end[L++] = 0;

return L-1;

}

void init(){

L = 0;

root = newnode();

}

void insert(char buf[]){

int len = strlen(buf);

int now = root;

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

if(next[now][buf[i]-'a'] == -1)

next[now][buf[i]-'a'] = newnode();

now = next[now][buf[i]-'a'];

}

end[now]++;

}

void build(){

queue<int> Q;

fail[root] = root;

for(int i = 0;i < 26;i++)

if(next[root][i] == -1)

next[root][i] = root;

else{

fail[next[root][i]] = root;

Q.push(next[root][i]);

}

while( !Q.empty() ){

int now = Q.front();

Q.pop();

for(int i = 0;i < 26;i++)

if(next[now][i] == -1)

next[now][i] = next[fail[now]][i];

else

{

fail[next[now][i]]=next[fail[now]][i];

Q.push(next[now][i]);

}

}

}

int query(char buf[]){

int len = strlen(buf);

int now = root;

int res = 0;

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

now = next[now][buf[i]-'a'];

int temp = now;

while( temp != root ){

res += end[temp];

end[temp] = 0;

temp = fail[temp];

}

}

return res;

}

};

char buf[1000010];

Trie ac;

int main(){

int T;

int n;

scanf("%d",&T);

while( T-- ){

scanf("%d",&n);

ac.init();

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

scanf("%s",buf);

ac.insert(buf);

}

ac.build();

scanf("%s",buf);

printf("%d\n",ac.query(buf));

}

return 0;

}

### 后缀数组

#include<iostream>

#include<cstring>

#include<algorithm>

#include<cstdio>

#include<queue>

using namespace std;

/\*

height数组范围为2-len，使用时，特判输入串为1的情况

每个子串必然为原串某个后缀的前缀

相同的子串，必然在height数组对应后缀中相邻

\*/

const int MAXN=100001;//字符串最长长度

const int LOG=((int)log2(MAXN))+1;//RMQ数组二维

int t1[MAXN],t2[MAXN],c[MAXN];//求sa数组需要的中间变量，不需要赋值

int len;//字符串实际长度

//计B[i]为str以下标i开始的后缀

char str[MAXN];//原字符串

int sa[MAXN];//sa[i]表示字典序第i小的为B[i]，1<=i<=len

int Rank[MAXN];//Rank[i]表示B[i]是第Rank[i]小的，为sa的逆，0<=i<=len-1

int height[MAXN];//height[i]表示sa[i]对应的后缀与sa[i-1]对应的后缀，最长公共前缀为height[i]，2<=i<=len

int RMQ[MAXN][LOG];//height数组的RMQ

bool cmp(int \*r,int a,int b,int l){

return r[a] == r[b] && r[a+l] == r[b+l];

}

void da(char str[],int sa[],int Rank[],int height[],int n,int m){//基数排序，m需大于最大字符

n++;

int i, j, p, \*x = t1, \*y = t2;

//第一轮基数排序，如果 s 的最大值很大，可改为快速排序

for(i = 0;i < m;i++)c[i] = 0;

for(i = 0;i < n;i++)c[x[i] = str[i]]++;

for(i = 1;i < m;i++)c[i] += c[i-1];

for(i = n-1;i >= 0;i--)sa[--c[x[i]]] = i;

for(j = 1;j <= n; j <<= 1){

p = 0;

//直接利用 sa 数组排序第二关键字

for(i = n-j; i < n; i++)y[p++] = i;//后面的 j 个数第二关键字为空的最小

for(i = 0; i < n; i++)if(sa[i] >= j)y[p++] = sa[i] - j;

//这样数组 y 保存的就是按照第二关键字排序的结果

//基数排序第一关键字

for(i = 0; i < m; i++)c[i] = 0;

for(i = 0; i < n; i++)c[x[y[i]]]++;

for(i = 1; i < m;i++)c[i] += c[i-1];

for(i = n-1; i >= 0;i--)sa[--c[x[y[i]]]] = y[i];

//根据 sa 和 x 数组计算新的 x 数组

swap(x,y);

p = 1; x[sa[0]] = 0;

for(i = 1;i < n;i++)

x[sa[i]] = cmp(y,sa[i-1],sa[i],j)?p-1:p++;

if(p >= n)break;

m = p;//下次基数排序的最大值

}

int k = 0;

n--;

for(i = 0;i <= n;i++)Rank[sa[i]] = i;

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

if(k)k--;

j = sa[Rank[i]-1];

while(str[i+k] == str[j+k])k++;

height[Rank[i]] = k;

}

}

void initRMQ()

{

int f=(int)log2(len)+1,i,j;

for(i=2;i<=len;i++)

{

RMQ[i][0]=height[i];//以i开头长度为2^0的最值当然是自己

}

for(j=1;j<=f;j++)

{

for(i=2;i+(1<<j)-1<=len;i++)//注意范围，右坐标未超出数组下标才有计算的意义

{

RMQ[i][j]=min(RMQ[i][j-1],RMQ[i+(1<<(j-1))][j-1]);//划分成长度相等的两个子区间

}

}

}

int askRMQ(int i,int j)//求height数组[i,j]中最值，注意2<=i<=j<=len

{

int k=(int)log2(j-i+1);

return min(RMQ[i][k],RMQ[j-(1<<k)+1][k]);

}

int lcp(int a,int b){//求B[a]和B[b]的最长公共前缀长度，下标范围0到len-1

if(a==b)

{

return len-a;

}

a=Rank[a];b=Rank[b];

if(a>b) swap(a,b);

return askRMQ(a+1,b);

}

/\*

求长度恰为k的子串个数

维护height数组中长为k-1的区间最小值MIN（即区间lcp）

说明有长度为MIN的子串出现了至少k次

此时再向两边扩展，设两边的height值的较大值为MAX

若MAX>=MIN，说明长度小于等于MIN的子串还能再出现，也就是个数超过了k次

若MAX<MIN说明有MIN-MAX种子串，只能在当前区间出现，即出现次数恰为k

需特判k==1的情况

\*/

int getNum(int k)

{

int i,ans=0,MIN,MAX;

if(k==1)

{

for(i=1;i<=len;i++)//一个一个后缀来看

{

MIN=len-sa[i];//字典序第i小的后缀长度

MAX=max(i==1?0:height[i],i==len?0:height[i+1]);

if(MIN>MAX)

{

ans+=MIN-MAX;

}

}

return ans;

}

k--;

for(i=2;i<=len-k+1;i++)

{

MIN=askRMQ(i,i+k-1);

MAX=max(i==2?0:height[i-1],i+k-1==len?0:height[i+k]);

if(MIN>MAX)

{

ans+=MIN-MAX;

}

}

return ans;

}

int main()

{

/\*

len = strlen(str);

da(str,sa,Rank,height,len,128);

initRMQ();

\*/

int T,k,L,R,i;

scanf("%d",&T);

while(T--)

{

scanf("%d%s",&k,str);

len = strlen(str);

if(len==1)

{

if(k==1)

{

printf("1\n");

}

else

{

printf("0\n");

}

continue;

}

da(str,sa,Rank,height,len,128);

initRMQ();

printf("%d\n",getNum(k));

}

return 0;

}

### 线段树

#include<iostream>

#include<string>

#include<cstring>

#include<algorithm>

#include<cmath>

#include<cstdio>

using namespace std;

/\*

线段树处理的数据需具有区间可加性，本模板为区间求和

读完后先建树！

lazy需要memset，其他不用

\*/

typedef long long LL;

#define MAXN 100001//区间最大长度

int A[MAXN];//原数组

LL lazy[MAXN<<2];//区间修改标记

LL Seg[MAXN<<2];//一段段区间中维护的值，若为区间和可能会爆int

//更新父节点

inline void PushUp(int rt)

{

Seg[rt]=Seg[rt<<1]+Seg[rt<<1|1];

}

//下推懒惰标记

inline void PushDown(int rt,int ln,int rn)//ln表示左子树元素结点个数，rn表示右子树结点个数

{

if (lazy[rt])

{

lazy[rt<<1]+=lazy[rt];

lazy[rt<<1|1]+=lazy[rt];

Seg[rt<<1]+=lazy[rt]\*ln;

Seg[rt<<1|1]+=lazy[rt]\*rn;

lazy[rt]=0;

}

}

//建树

void Build(int L,int R,int rt)//L、R分别为区间左右端点，rt为节点号，默认为1

{

if (L==R)

{

Seg[rt]=A[L];

return;

}

int mid=(L+R)>>1;

Build(L,mid,rt<<1);

Build(mid+1,R,rt<<1|1);

PushUp(rt);

}

//单点修改

void Add(int pos,int C,int L,int R,int rt)//将位置为pos的点加上C，L、R、rt默认为1、n、1

{

if (L==R)

{

Seg[rt]+=C;//若要直接修改而不是添加，此处改成=

return;

}

int mid=(L+R)>>1;

//PushDown(rt,mid-L+1,R-mid);//只要存在区间修改就要加上这句

if (pos<=mid)

Add(pos,C,L,mid,rt<<1);

else

Add(pos,C,mid+1,R,rt<<1|1);

PushUp(rt);

}

//区间修改

void Update(int Left,int Right,int C,int L,int R,int rt)//Left到Right都加上C，L、R、rt默认为1、n、1

{

if (Left<=L&&R<=Right)

{

Seg[rt]+=C\*(R-L+1);

lazy[rt]+=C;

return;

}

int mid=(L+R)>>1;

PushDown(rt,mid-L+1,R-mid);

if (Left<=mid) Update(Left,Right,C,L,mid,rt<<1);

if (Right>mid) Update(Left,Right,C,mid+1,R,rt<<1|1);

PushUp(rt);

}

//区间查询

LL Query(int Left,int Right,int L,int R,int rt)//询问Left到Right的区间和，L、R、rt默认为1、n、1

{

if (Left<=L&&R<=Right)

return Seg[rt];

int mid=(L+R)>>1;

PushDown(rt,mid-L+1,R-mid);//只要存在区间修改就要加上这句

LL ANS=0;

if (Left<=mid) ANS+=Query(Left,Right,L,mid,rt<<1);

if (Right>mid) ANS+=Query(Left,Right,mid+1,R,rt<<1|1);

return ANS;

}

int main()

{

memset(Seg,0,sizeof(Seg));

memset(lazy,0,sizeof(lazy));

int n,i,q,L,R,T,tem;

char op;

scanf("%d%d",&n,&q);

for(i=1;i<=n;i++)

{

scanf("%d",&A[i]);

}

Build(1,n,1);

while(q--)

{

scanf(" %c",&op);

scanf("%d%d",&L,&R);

if(op=='Q')

{

printf("%lld\n",Query(L,R,1,n,1));

}

else

{

scanf("%d",&tem);

Update(L,R,tem,1,n,1);

}

}

return 0;

}

### 滑动窗口

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

#include<queue>

using namespace std;

/\*

固定长度的连续序列中的最值

这个序列是在原序列中滑动的

使用滑动窗口高效求解

窗口向右滑动时，左边元素删除元素，右边增加元素

假设是求最小值，如果窗口中有一个序列是3 2 1，会发现在1不得不被删除前

窗口滑动，最小值都不可能在3 2中取，所以应及时删除

也就是，当窗口右边要新增一个元素时，应删除前面所有比它大的元素

在将该元素防于最右端，于是窗口中元素总是递增的，最左边元素就是最小值

注意，窗口中元素个数不一定等于窗口尺寸

最左边的元素不一定是窗口框住原序列时的最左边的元素

滑动更新时判断最左边元素是否要删除，判断是否等于原序列中窗口左端第一个元素即可

\*/

int main()

{

//假设每次求窗口中最小值

int x[10]={3,4,1,8,0,1,3,7,6,4};

int len=3;//滑动窗口长度

int size=10;//原序列长度

int i,j,k,tem;

deque<int> dq;

//预处理，求得窗口初始值

dq.push\_back(x[0]);

for(i=1;i<len;i++)

{

while(!dq.empty()&&dq.back()>x[i])

{

dq.pop\_back();

}

dq.push\_back(x[i]);

}

cout<<dq.front()<<' ';

//开始更新

for(i=len;i<size;i++)

{

if(dq.front()==x[i-len])

{

dq.pop\_front();

}

while(!dq.empty()&&dq.back()>x[i])

{

dq.pop\_back();

}

dq.push\_back(x[i]);

cout<<dq.front()<<' ';

}

return 0;

}

### 生成排列

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

using namespace std;

void permutation(int x[],int i,int j)//x[i]到x[j]全排列

{

int n,k,tem;

if(i==j)

{

for(n=0;n<=j;n++)

{

printf("%d",x[n]);

}

printf("\n");

}

else

{

/\*

最简单的直接swap，使得输出不完全是按字典序

实际上，选取到某一个，把它移到首位还不够，它之前的应该右移，这才是按字典序

\*/

for(n=i;n<=j;n++)

{

tem=x[n];//优化版，每一次交换选取位与第一位，即可实现不断右移的操作

x[n]=x[i];

x[i]=tem;

permutation(x,i+1,j);

}

tem=x[i];

for(n=i;n<j;n++)//最后记得左移复原，否则递归中会影响上一次调用的结果

{

x[n]=x[n+1];

}

x[j]=tem;

}

}

int main()

{

int \*x,n,i;

scanf("%d",&n);

x=new int[n];

for(i=0;i<n;i++) x[i]=i+1;

permutation(x,0,n-1);

return 0;

}

### 质数筛

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

#include<map>

using namespace std;

/\*

原理很简单，合数总是可以看成比自己小的数的倍数

首先所有数都未标记，看做质数，然后从2开始，每当找到一个未标记的数i

认定它为质数，然后标记范围内所有它的倍数（i倍起，因为小于i的数之前筛过）

最后剩下的未标记的就是质数

寻找到sqrt(MAX)即可，因为此时i倍已经可以达到MAX

\*/

#define MAX (int)(1000)//此处控制筛选范围

bool x[MAX+1];//标记为1表示是合数，0表示质数

inline void getP()

{

long long bs,i;

long long range=(long long)(sqrt(MAX)+1);

memset(x,0,sizeof(x));//先全部标记为质数

for(i=2;i<=range;i++)

{

if(!x[i])//未标记，说明是质数

{

bs=i;//倍数一开始就设为i，因为小于i的数之前已经筛过

while(bs\*i<=MAX)

{

x[bs\*i]=1;

bs++;

}

}

}

}

int main()

{

getP();

int i;

for(i=2;i<=MAX;i++)

{

if(!x[i])

{

printf("%lld\n",i);

}

}

return 0;

}

### 快速幂取模

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

using namespace std;

//这种快速幂和每次将幂二分，递归实现的原理不同

inline long long Qpow(long long x,long long n,long long MOD)//指数看做二进制，x^(2^0) \* x^(2^1)（该位需为1）……

{

if(x==1)

{

return 1;

}

long long tem=x%MOD;//x^(2^0)

long long ans=1;

while(n)

{

if(n&1)

{

ans=(ans\*tem)%MOD;

}

n>>=1;

tem=(tem\*tem)%MOD;//x^(2^i)->x^(2^(i+1))

}

return ans;

}

int main()

{

cout<<Qpow(2,20,100);

return 0;

}

### 欧拉函数

long long Euler(long long n)//求比n小且与n互质的数的个数

{

long long ans=n;

for(int i=2;i\*i<=n;i++)

{

if(n%i==0)

{

ans-=ans/i;

while(n%i==0)

{

n/=i;

}

}

}

if(n>1)

{

ans-=ans/n;

}

return ans;

}

long long euler[MAXN+1]; //求所有小于等于MAXN的数的欧拉函数

void getEuler(){

memset(euler,0,sizeof(euler));

euler[1] = 1;

for(int i = 2;i <= MAXN ;i++)

if(!euler[i])

for(int j = i;j <= MAXN ; j += i){

if(!euler[j])

euler[j] = j;

euler[j] = euler[j]/i\*(i-1);

}

}

### 乘法逆元

#include<iostream>

#include<string>

#include<memory.h>

#include<algorithm>

#include<cmath>

#include<vector>

using namespace std;

#define LL long long//这样不仅简写long long，在不会超范围时也可方便地改为int

/\*

拓展欧几里得

求解整数x和y，使得ax+by==d，且abs(x)+abs(y)最小，其中d=gcd(a,b)

\*/

void Egcd(LL a,LL b,LL &d,LL &x,LL &y)

{

if(b==0)

{

d=a,x=1,y=0;

}

else

{

Egcd(b,a%b,d,y,x);

y-=x\*(a/b);

}

}

/\*

x与y都可能是非常大的数

(x/y)%MODE（保证能整除） != (x%MODE)/(y%MODE)

于是需要求解y%MODE的乘法逆元，转换成乘法，方便分部计算不断取余，不会溢出

\*/

LL inv(LL a,LL MODE)//计算a摸MODE下逆元

{

LL d,x,y;

Egcd(a,MODE,d,x,y);

return d==1?(x+MODE)%MODE:-1;//-1代表无逆元

}

int main()

{

LL a=12,b=3,MODE=7;//(a/b)%MODE

LL tem=inv(b,MODE);

cout<<(a\*tem)%MODE;

return 0;

}

### 阶乘质因子分解

/\*

对n!分解质因子，显然质因子都<=n

当然不可能从2-n一个个分解质因子

假设当前要求n!分解质因子后k的个数

则只有k所有小于等于n的倍数能分解出来，有n/k个

比较难解释，看例子，16!中求质因子3的个数

可能的项有3 6 9 12 15（16/3=5个）

取出5个质因子3后，变成1 2 3 4 5

最大项就为前一次的个数，此时相当于再求5!中k的个数

当某次n/k==0时结束即可

\*/

inline int solve(int n,int k)//求n!中质因子k的个数

{

int ans=0;

while(n/k!=0)

{

ans+=n/k;

n=n/k;

}

return ans;

}