解题模版

```
#include<bits/stdc++.h>
#define endl '\n'
#define debug(x) cout<<#x<<'='<<x<<endl;
#define fi first
#define se second
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
const int maxn=10000010;
typedef pair<ll,ll> pii;
const int INF=0x3f3f3f3f;
#define int long long
#define Push push_back
#define Pop pop_back
#define mp make_pair
#define pointf(x) cout<<fixed<<setprecision(x)<<endl
int n,m,k;
int read(){
    int x=0,f=0;
    char ch=getchar();
    while(ch<'0'||ch>'9')f|=(ch=='-'),ch=getchar();
    while(ch>='0'&&ch<='9')x=(x<<1)+(x<<3)+(ch^48),ch=getchar();
    return f?-x:x;
}
void solve(){
}
signed main(){
    ios::sync_with_stdio(0);
    cin.tie(0);
    cout.tie(0);
    int T=1;
    cin>>T;
    while(T--){
        solve();
    }
    return 0;
}
```

```
#include <iostream>
#include <cstring>
#include <cstdio>
#include <algorithm>
#include <vector>
#include <fstream>
#include <cmath>
#include <queue>
```

1.数

1.1.数组的特殊输入

1.1.1.输入字符串(无空格输入数字)并储存于数组中

```
void func1_1_1(){
    string input_string;
    vector<int> array;
    cin >> input_string;
    for (int i = 0; i < input_string.size(); i++) {
        int num = input_string[i] - '0';
        if (num >= 0 && num <= 9) {
        array.push_back(num);
        }
    }
}
```

- input_string用来储存输入的字符串
- array用来储存输入的数字
- 需要头文件

1.1.2.输入字符串并转化为数字

```
void func1_1_2(){
    string str1;
    cin>>str1;
    stringstream geek(str1);
    int x=0;
    geek>>x;
    cout<<x;
}
```

- str1用来储存输入的字符串
- x用来储存字符串转化出的数字
- 需要头文件

1.1.3注意

使用cin，cout的速度比scanf,printf慢不少，可能导致TLE

1.2.用数组储存特大整数和其运算(模板)

```
struct BigIntTiny {
    int sign;
    std::vector<int> v;

    BigIntTiny() : sign(1) {}
    BigIntTiny(const std::string &s) {
        *this = s;
    }
    BigIntTiny(int v) {
        char buf[21];
        sprintf(buf, "%d", v);
        *this = buf;
```

```
    }
    void zip(int unzip) {
        if (unzip == 0) {
            for (int i = 0; i < (int)v.size(); i++)
                v[i] = get_pos(i * 4) + get_pos(i * 4 + 1) * 10 + get_pos(i * 4 +
2) * 100 + get_pos(i * 4 + 3) * 1000;
        } else
            for (int i = (v.resize(v.size() * 4), (int)v.size() - 1), a; i >= 0;
i--)
                a = (i % 4 >= 2) ? v[i / 4] / 100 : v[i / 4] % 100, v[i] = (i &
1) ? a / 10 : a % 10;
        setsign(1, 1);
    }
    int get_pos(unsigned pos) const {
        return pos >= v.size() ? 0 : v[pos];
    }
    BigIntTiny &setsign(int newsign, int rev) {
        for (int i = (int)v.size() - 1; i > 0 && v[i] == 0; i--)
            v.erase(v.begin() + i);
        sign = (v.size() == 0 || (v.size() == 1 && v[0] == 0)) ? 1 : (rev ?
newsign * sign : newsign);
        return *this;
    }
    std::string to_str() const {
        BigIntTiny b = *this;
        std::string s;
        for (int i = (b.zip(1), 0); i < (int)b.v.size(); ++i)
            s += char(*(b.v.rbegin() + i) + '0');
        return (sign < 0 ? "-" : "") + (s.empty() ? std::string("0") : s);
    }
    bool absless(const BigIntTiny &b) const {
        if (v.size() != b.v.size()) return v.size() < b.v.size();
        for (int i = (int)v.size() - 1; i >= 0; i--)
            if (v[i] != b.v[i]) return v[i] < b.v[i];
        return false;
    }
    BigIntTiny operator-() const {
        BigIntTiny c = *this;
        c.sign = (v.size() > 1 || v[0]) ? -c.sign : 1;
        return c;
    }
    BigIntTiny &operator=(const std::string &s) {
        if (s[0] == '-')
            *this = s.substr(1);
        else {
            for (int i = (v.clear(), 0); i < (int)s.size(); ++i)
                v.push_back(*(s.rbegin() + i) - '0');
            zip(0);
        }
        return setsign(s[0] == '-' ? -1 : 1, sign = 1);
    }
    bool operator<(const BigIntTiny &b) const {
        return sign != b.sign ? sign < b.sign : (sign == 1 ? absless(b) :
b.absless(*this));
    }
    bool operator==(const BigIntTiny &b) const {
```

```
        return v == b.v && sign == b.sign;
    }
    BigIntTiny &operator+=(const BigIntTiny &b) {
        if (sign != b.sign) return *this = (*this) - -b;
        v.resize(std::max(v.size(), b.v.size()) + 1);
        for (int i = 0, carry = 0; i < (int)b.v.size() || carry; i++) {
            carry += v[i] + b.get_pos(i);
            v[i] = carry % 10000, carry /= 10000;
        }
        return setsign(sign, 0);
    }
    BigIntTiny operator+(const BigIntTiny &b) const {
        BigIntTiny c = *this;
        return c += b;
    }
    void add_mul(const BigIntTiny &b, int mul) {
        v.resize(std::max(v.size(), b.v.size()) + 2);
        for (int i = 0, carry = 0; i < (int)b.v.size() || carry; i++) {
            carry += v[i] + b.get_pos(i) * mul;
            v[i] = carry % 10000, carry /= 10000;
        }
    }
    BigIntTiny operator-(const BigIntTiny &b) const {
        if (b.v.empty() || b.v.size() == 1 && b.v[0] == 0) return *this;
        if (sign != b.sign) return (*this) + -b;
        if (absless(b)) return -(b - *this);
        BigIntTiny c;
        for (int i = 0, borrow = 0; i < (int)v.size(); i++) {
            borrow += v[i] - b.get_pos(i);
            c.v.push_back(borrow);
            c.v.back() -= 10000 * (borrow >>= 31);
        }
        return c.setsign(sign, 0);
    }
    BigIntTiny operator*(const BigIntTiny &b) const {
        if (b < *this) return b **this;
        BigIntTiny c, d = b;
        for (int i = 0; i < (int)v.size(); i++, d.v.insert(d.v.begin(), 0))
            c.add_mul(d, v[i]);
        return c.setsign(sign * b.sign, 0);
    }
    BigIntTiny operator/(const BigIntTiny &b) const {
        BigIntTiny c, d;
        BigIntTiny e=b;
        e.sign=1;

        d.v.resize(v.size());
        double db = 1.0 / (b.v.back() + (b.get_pos((unsigned)b.v.size() - 2) /
1e4) +
                           (b.get_pos((unsigned)b.v.size() - 3) + 1) / 1e8);
        for (int i = (int)v.size() - 1; i >= 0; i--) {
            c.v.insert(c.v.begin(), v[i]);
            int m = (int)((c.get_pos((int)e.v.size()) * 10000 +
c.get_pos((int)e.v.size() - 1)) * db);
            c = c - e * m, c.setsign(c.sign, 0), d.v[i] += m;
            while (!(c < e))
```

```
                c = c - e, d.v[i] += 1;
        }
        return d.setsign(sign * b.sign, 0);
    }
    BigIntTiny operator%(const BigIntTiny &b) const {
        return *this - *this / b * b;
    }
    bool operator>(const BigIntTiny &b) const {
        return b < *this;
    }
    bool operator<=(const BigIntTiny &b) const {
        return !(b < *this);
    }
    bool operator>=(const BigIntTiny &b) const {
        return !(*this < b);
    }
    bool operator!=(const BigIntTiny &b) const {
        return !(*this == b);
    }
};
```

1.3数组的初始化

```
memset(a,0,sizeof(a));
```

2.字符

2.1字符串的输入

2.1.1 cin

对于char a[10]

```
#include<iostream>
using namespace std;
int main(){
    char a[10];
    cin>>a;
    cout<<a;
}
```

- 不能对char*a使用cin

对于string str

```
#include<iostream>
using namespace std;
int main(){
    string str;
    cin>>str;
    cout<<str;
}
```

2.1.2 cin.get()

```
#include<iostream>
#include<string>
using namespace std;
int main()
{
    char a[20];
    cin.get(a, 20);
    cout << a;
    return 0;
}
```

- 这个方法只能正针对于是字符数组，不能使用string来输入
- 输入完后，需要cin.get()吸收回车键

2.1.3 cin.getline()

```
int main()
{
    char a[20];
    cin.getline(a, 20,'\n');
    cout << a;
    return 0;
}
```

- 这个方法只能正针对于是字符数组，不能使用string来输入
- 三个参数，第一个是存储的数组地址，第二个是最大存储量，第三个是结束字符

2.1.3 getline()

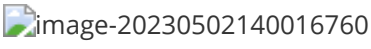
```
#include<iostream>
#include<string>
using namespace std;
int main()
{
    string s;
    getline(cin,s);
    cout << s;
    return 0;
}
```

- 弥补了之前cin.getline()和cin.get（）的不能读取string的一个小的弊端
- 要包括string头文件
- 不知道能不能用字符数组

2.1.4 gets()

```
int main()
{
    char s[10];
    gets(s);
    cout << s;
    return 0;
}
```

- 这个函数有些奇葩的是，这个函数必须要包含头文件string，但是这个函数却不能直接对变量string来进行赋值。
- 这个函数由于不安全，在VS2015及以后的IDE中就不存在这个函数，而是用gets_s()函数来代替。



2.2 字符串处理

2.2.1 KMP算法（字符数组快速查找）

```
//
void kmp_pre(char x[],int m,int next[]) {
    int i,j;
    j=next[0]=-1;
    i=0;
    while(i<m) {
        while(-1!=j&&x[i]!=x[j])
            j=next[j];
        next[++i]=++j;
    }
}


void preKMP(char x[],int m,int kmpNext[]){
    int i,j;
    j=kmpNext[0]=-1;
    i=0;
    while(i<m){
        while(-1!=j&&x[i]!=x[j])
        j=kmpNext[j];
        if(x[++i]==x[++j])
        kmpNext[i]=kmpNext[j];
        else
        kmpNext[i]=j;
    }
}
```

2.2.2字符串类查找

```
ss.find();
ss.rfind();
ss.find_first_of();
ss.find_last_of();
ss.find_first_not_of();
ss.find_last_not_of();
```

一些常用操作

```
        char *s1=ss+2;
        string s=s1;
        size_t it=s.find(' ');
        string temp = (it != string::npos) ? s.substr(0, it):s;
        s = (it != string::npos) ? s.substr(it + 1):"";
```

2.3字符串流（stringstream）

```
#include <iostream>
#include <string>
#include <sstream>
using namespace std;

int main()
{
    stringstream ss;
    int first = 0, second = 0;
    ss << "456";  // 插入字符串
    ss >> first;  //转换成int
    cout << first << endl;
    ss.clear();  //在进行多次转换前， 必须清除ss
    ss << true;
    ss >> second;
    cout << second << endl;
    return 0;
}
```

2.4字符串类

```
string str="hello";
str.insert(1,"aaa");//在下标为1处插入aaa
str.c_str()//返回C语言数组
transform(str.begin(),str.end(),str.begin(),::tolower);//大写转小写
transform(str.begin(),str.end(),str.begin(),::toupper);//小写转大写
reverse(s.begin(),s.end());//字符串反转
```

2.5字符串的hash

2.5.1自然溢出

对于自然溢出方法，我们定义 Base ，而MOD对于自然溢出方法，就是 unsigned long long 整数的自然溢出（相当于MOD 是2^{64}-1）

```
#define ull unsigned long long

ull Base;
ull hash[MAXN], p[MAXN];

hash[0] = 0;
p[0] = 1;
```

定义了上面的两个数组，首先 ***hash[i]***表示 ***[0,i]***字串的hash 值。而 ***p[i] ***表示 $Base^i$ ，也就是底的 ***i***次方。

那么对应的 Hash 公式为：

$$hash[i] = hash[i-1] * Base + idx(s[i])$$

只有一位不同的子串可以在O(1)时间内转换

例题：

https://leetcode-cn.com/problems/distinct-echo-substrings/

代码如下

```
#define ull unsigned long long   // 自然溢出用 unsigned long long
const int MAXN = 2e4 + 50;
class Solution {
public:
    unordered_set<ull> H;
    ull base = 29;
    ull hash[MAXN], p[MAXN];

    int distinctEchoSubstrings(string text) {
        int n = text.size();
        hash[0] = 0, p[0] = 1;
        for(int i = 0;i < n;i++)
            hash[i+1] = hash[i]*base + (text[i] - 'a' + 1);

        for(int i = 1;i < n;i++)
            p[i] = p[i-1]*base;

        for(int len = 2; len <= n; len += 2)
        {
            for(int i = 0;i + len -1 < n;i++)
            {
                int x1 = i, y1 = i + len/2 - 1;
                int x2 = i + len/2, y2 = i + len - 1;
                ull left = hash[y1 + 1] - hash[x1] * p[y1 + 1 - x1];
                ull right = hash[y2 + 1] - hash[x2] * p[y2 + 1 - x2];
                if(left == right) H.insert(left);
            }
        }
```

```
        return H.size();
    }
};
```

3.算法数学函数

```
#include<algorithm>//头文件
#include<cmath>
```

3.1 sort函数

快速排序函数

```
sort(/*起始指针*/, /*结束后一个指针*/, /*排序依据(初始化为升序)*/);
```

3.2 对数函数

首先要知道exp（）函数

exp（n）值为e^n次方；

另外log函数包括两种函数 一种以e为低的log（）函数

另一种为以10为底的log 10（）函数；

具体用法见下面

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    double a=9,b=10;
    cout<<log(a)<<endl;
    cout<<log(exp(a))<<endl;
    cout<<log10(b)<<endl;
    return 0;
}
```

另外如果自定义以m为底，求log n的值

需要double a=log(n)/log(m);

举例如下：

```
#include<iostream>
#include<cmath>
using namespace std;
int main()
{
    double a=2,b=2;//以2为底的对数函数
    for(b=2;b<=16;b=b+2)
    {
            cout<<"b="<<b<<"时，以2为底的对数函数="<<log(b)/log(a)<<endl;
    }
    return 0;
}
```

3.3其他函数

```
min(a,b);
max(a,b);
int n=nth_element(arr.begin(),arr.begin()+n,arr.end());
//找出数组排序后在n位置的数，时间复杂度O(n);
int newLenth=unique(arr.begin(),arr.end())-arr.begin();
//将不重复的元素提前，返回新数组最后一个元素的迭代器，newLenth为新数组长度
tgamma();//阶乘函数
sqrt();//开方函数
lower_bound(data.begin(), data.end(), i);//查找首个不小于给定值的元素的函数
upper_bound(data.begin(), data.end(), i);//查找首个大于给定值的元素的函数
```

4.集合

4.1定义

```
set<int> st;//集合
multiset<int> mst;//多重集
```

4.2基础操作

```
st.insert(1);
st.find(1);//返回对应元素的迭代器
st.erase(1);

mst.insert(1);
mst.insert(1);
mst.count(a);//重复元素的个数
for(set<int>::iterator i=all.begin();i!=all.end();i++)
        cout<<*i<<endl; //注意指针运算符
```

5.映射

```
#include<map>
```

5.1构造函数

```
map<int,string> Mymap;
map<int,string> Mymap2 (Mymap.begin(),Mymap.end());
map<int,string> Mymap3 (Mymap2);
```

5.2插入数据

```
Mymap.insert(map<int,string>::value_type(11,"oneone"));
Mymap.insert(pair<int ,string>(111,"oneoneone"));
```

5.3查找数据

```
map<int ,string>::iterator itr;
itr = Mymap.find(1);
if(itr!=Mymap.end())//如找不到，返回Mymap.end()
{
    cout<<"找到key为1的项:"<<endl;
    cout<< itr->first <<" "<<itr->second<<endl;//可以打印该项的key和value
}
else
    cout<<"找不到key为1的项！"<<'\n';
```

5.4删除元素

```
itr = Mymap.find(1);
    if(itr!=Mymap.end())
    {
        Mymap.erase(itr);//通过迭代器对象删除
        Mymap.erase(1);//也可以通过主键删除
    }

Mymap.erase(Mymap.begin(),Mymap.end());//相当于Mymap.clear(),可以清除整个map的内容
```

5.5其他基本操作

```
Mymap().swap(Mymap2);
Mymap().size();
Mymap().empty();
Mymap().begin();
Mymap().end();
Mymap().count();
```

6.链表

6.1链表的实现

```
int cnt;
struct Edge{
    int to;//这条边的终点
    int next;//这条边的下一条边的编号
    int val;//边的权值
}e[maxn];
int head[maxn];//head[i]表示i节点的第一条边的编号
void add(int u,int v,int w){
    e[++cnt].to=v;
    e[cnt].val=w;
    e[cnt].next=head[u];
    head[u]=cnt;//链表头部插入新元素
}
```

7.队列

7.1优先队列

```
//升序队列
priority_queue <int,vector<int>,greater<int> > q;//第三个变量可以是自定义的只有一个比较函数的结构体
//降序队列
priority_queue <int,vector<int>,less<int> > q;
```

7.2双端队列

```
//元素访问
q.front(); //返回队首元素
q.back();// 返回队尾元素
//修改
q.push_back();// 在队尾插入元素
q.pop_back();// 弹出队尾元素
q.push_front();// 在队首插入元素
q.pop_front();// 弹出队首元素
q.insert();// 在指定位置前插入元素（传入迭代器和元素）
q.erase();// 删除指定位置的元素（传入迭代器）
容量
q.empty();// 队列是否为空
q.size();// 返回队列中元素的数量
```

此外，`deque` 还提供了一些运算符。其中较为常用的有：

- 使用赋值运算符 `=` 为 `deque` 赋值，类似 `queue`。
- 使用 `[]` 访问元素，类似 `vector`。

7.3循环队列

使用数组模拟队列会导致一个问题：随着时间的推移，整个队列会向数组的尾部移动，一旦到达数组的最末端，即使数组的前端还有空闲位置，再进行入队操作也会导致溢出（这种数组里实际有空闲位置而发生了上溢的现象被称为「假溢出」）。

解决假溢出的办法是采用循环的方式来组织存放队列元素的数组，即将数组下标为 0 的位置看做是最后一个位置的后继。（数组下标为 `x` 的元素，它的后继为 `(x + 1) % SIZE`）。这样就形成了循环队列。

8.哈希表

```
std::unordered_map<std::string, std::string> umap;
```

成员方法	功能	用法
begin()	返回指向容器中第一个键值对的正向迭代器。	$\text{auto it} = \text{mymap.begin()}$
end()	返回指向容器中最后一个键值对之后位置的正向迭代器。	$\text{it!} = \text{mymap.end()}$
cbegin()	和 begin() 功能相同，只不过在其基础上增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。	$\text{auto it} = \text{mymap.cbegin()}$
cend()	和 end() 功能相同，只不过在其基础上，增加了 const 属性，即该方法返回的迭代器不能用于修改容器内存储的键值对。	$\text{it!} = \text{mymap.cend()}$
empty()	若容器为空，则返回 true；否则 false。	mymap.empty()
size()	返回当前容器中存有键值对的个数。	mymap.size()
max_size()	返回容器所能容纳键值对的最大个数，不同的操作系统，其返回值亦不相同。	mymap.max_size()
operator[key]	该模板类中重载了 [] 运算符，其功能是可以向访问数组中元素那样，只要给定某个键值对的键 key，就可以获取该键对应的值。注意，如果当前容器中没有以 key 为键的键值对，则其会使用该键向当前容器中插入一个新键值对。	$\text{string name} = \text{mymap[key]};$ $\text{mymap[key2]} = \text{name};$

at(key)	返回容器中存储的键 key 对应的值，如果 key 不存在，则会抛出 out_of_range 异常。	mymap.at(key) = value;
find(key)	查找以 key 为键的键值对，如果找到，则返回一个指向该键值对的正向迭代器；反之，则返回一个指向容器中最后一个键值对之后位置的迭代器（如 end() 方法返回的迭代器）。	mymap.find(key);
count(key)	在容器中查找以 key 键的键值对的个数。	mymap.count(key)
equal_range(key)	返回一个 pair 对象，其包含 2 个迭代器，用于表明当前容器中键为 key 的键值对所在的范围。	mymap.equal_range(key);
emplace()	向容器中添加新键值对，效率比 insert() 方法高。	mymap.emplace(key, value);
emplace_hint()	向容器中添加新键值对，效率比 insert() 方法高。	
insert()	向容器中添加新键值对。	mymap.insert(pair < string, double > (key, value)); mymap.insert({{key, value}, {key, value}});
erase()	删除指定键值对。	mymap.erase(key);
clear()	清空容器，即删除容器中存储的所有键值对。	mymap.clear();

swap()	交换 2 个 unordered_map 容器存储的键值对，前提是必须保证这 2 个容器的类型完全相等。	mymap1.swap(mymap2);
bucket_count()	返回当前容器底层存储键值对时，使用桶（一个线性链表代表一个桶）的数量。	unsigned n = mymap.bucket_count();
max_bucket_count()	返回当前系统中，unordered_map 容器底层最多可以使用多少桶。	mymap.max_bucket_count()
bucket_size(n)	返回第 n 个桶中存储键值对的数量。	unsigned nbuckets = mymap.bucket_count();
bucket(key)	返回以 key 为键的键值对所在桶的编号。	mymap.bucket(key)
load_factor()	返回 unordered_map 容器中当前的负载因子。负载因子，指的是的当前容器中存储键值对的数量（size()）和使用桶数（bucket_count()）的比值，即 load_factor() = size() \ bucket_count()。	mymap.load_factor()
max_load_factor()	返回或者设置当前 unordered_map 容器的负载因子。	mymap.max_load_factor()
rehash(n)	将当前容器底层使用桶的数量设置为 n。	mymap.rehash(20);
reserve()	将存储桶的数量（也就是 bucket_count() 方法的返回值）设置为至少容纳count个元（不超过最大负载因子）所需的数量，并重新整理容器。	mymap.reserve(6);
hash_function()	返回当前容器使用的哈希函数对象。	string map :: hasherfn = mymap.hash_function();

9.数学

9.1数论

9.1.1素数

9.1.1.1素数筛选（判断小于maxn数是否为素数）

```
const int MAXN=1000010;
bool notprime[MAXN];
void init(){
    memset(notprime,false,sizeof(notprime));
    notprime[0]=true;
    notprime[1]=true;
    for(int i=2;i<MAXN;i++){
        if(!notprime[i]){
            if(i*i>MAXN)
                continue;
            for(int j=i*i;j<MAXN;j+=i)
                notprime[j]=true;
        }
    }
}
```

9.1.1.2找出所有小于MAXN的素数（prime[0]标记个数）

```
const int MAXN=10000;
int prime[MAXN+1];
void solve() {
    memset(prime,0,sizeof(prime));
    for(int i=2; i<MAXN; i++) {
        if(!prime[i])prime[++prime[0]]=i;
        for(int j=1; j<=prime[0]&&prime[j]<=MAXN/i; j++) {
            prime[prime[j]*i]=1;
            if(i%prime[j]==0) break;
        }
    }
}
```

9.1.1.3素性测试

定义

素性测试（Primality test）是一类在 **不对给定数字进行素数分解**（prime factorization）的情况下，测试其是否为素数的算法。

素性测试有两种：

1. 确定性测试：绝对确定一个数是否为素数。常见示例包括 Lucas–Lehmer 测试和椭圆曲线素性证明。
2. 概率性测试：通常比确定性测试快很多，但有可能（尽管概率很小）错误地将 合数 识别为质数（尽管反之则不会）。因此，通过概率素性测试的数字被称为 **可能素数**，直到它们的素数可以被确定性地证明。而通过测试但实际上是合数的数字则被称为 **伪素数**。有许多特定类型的伪素数，最常

见的是费马伪素数，它们是满足费马小定理的合数。概率性测试的常见示例包括 Miller–Rabin 测试。

Fermat 素性测试

Fermat 素性检验 是最简单的概率性素性检验。

我们可以根据 费马小定理 得出一种检验素数的思路：

基本思想是不断地选取在 $[2, n-1]$ 中的基 a，并检验是否每次都有 $a^{n-1} \equiv 1 \pmod{n}$。

实现

```
bool millerRabin(int n) {
  if (n < 3) return n == 2;
  // test_time 为测试次数,建议设为不小于 8
  // 的整数以保证正确率,但也不宜过大,否则会影响效率
  for (int i = 1; i <= test_time; ++i) {
    int a = rand() % (n - 2) + 2;
    if (quickPow(a, n - 1, n) != 1) return 0;
  }
  return 1;
}
```

9.1.2 卢卡斯定理（Lucas定理）

Lucas 定理用于求解大组合数取模的问题，其中模数必须为素数。正常的组合数运算可以通过递推公式求解（详见 排列组合），但当问题规模很大，而模数是一个不大的质数的时候，就不能简单地通过递推求解来得到答案，需要用到 Lucas 定理。

Lucas 定理内容如下：对于质数 p，有

$$\binom{n}{m} \bmod p = \binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor} \cdot \binom{n \bmod p}{m \bmod p} \bmod p$$

观察上述表达式，可知 $n \bmod p$ 和 $m \bmod p$ 一定是小于 p 的数，可以直接求解，$\binom{\lfloor n/p \rfloor}{\lfloor m/p \rfloor}$ 可以继续用 Lucas 定理求解。这也就要求 p 的范围不能够太大，一般在 10^5 左右。边界条件：当 $m = 0$ 的时候，返回 1。当$n < m$时，$\binom{n}{m} = 0$

时间复杂度为 $O(f(p) + g(n) \log n)$，其中 $f(n)$ 为预处理组合数的复杂度，$g(n)$ 为单次求组合数的复杂度。

```
long long Lucas(long long n, long long m, long long p) {
  if (m == 0) return 1;
  return (C(n % p, m % p, p)/*组合数求解*/ * Lucas(n / p, m / p, p)) % p;
}
```

证明

特殊的，当p==2时，

$$\binom{n}{m} \bmod 2 = \binom{\lfloor n/2 \rfloor}{\lfloor m/2 \rfloor} \cdot \binom{n \bmod 2}{m \bmod 2} \bmod 2$$

只有出现$n \bmod 2 == 0 \&\& m \bmod 2 == 1$原式可以被2整除，即如果$n \& m == m$则 $\binom{n}{m}$只可能为奇数

9.2矩阵

9.2.1矩阵的定义

```
struct mat {
  LL a[sz][sz];

  mat() { memset(a, 0, sizeof a); }
  mat operator*(const mat& T) const {
    mat res;
    int r;
    for (int i = 0; i < sz; ++i)
      for (int k = 0; k < sz; ++k) {
        r = a[i][k];
        for (int j = 0; j < sz; ++j)
          res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= MOD;
      }
    return res;
  }

  mat operator^(LL x) const {
    mat res, bas;
    for (int i = 0; i < sz; ++i) res.a[i][i] = 1;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) bas.a[i][j] = a[i][j] % MOD;
    while (x) {
      if (x & 1) res = res * bas;
      bas = bas * bas;
      x >>= 1;
    }
    return res;
  }
};
```

9.2.2矩阵的乘法

```
/ 以下文的参考代码为例
mat operator*(const mat& T) const {
  mat res;
  for (int i = 0; i < sz; ++i)
    for (int j = 0; j < sz; ++j)
      for (int k = 0; k < sz; ++k) {
        res.a[i][j] += mul(a[i][k], T.a[k][j]);
        res.a[i][j] %= MOD;
      }
  return res;
}

// 不如
```

```
mat operator*(const mat& T) const {
  mat res;
  int r;
  for (int i = 0; i < sz; ++i)
    for (int k = 0; k < sz; ++k) {
      r = a[i][k];
      for (int j = 0; j < sz; ++j)
        res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= MOD;
    }
  return res;
}
```

9.2.3矩阵的快速幂

```
 mat operator^(LL x) const {
    mat res, bas;
    for (int i = 0; i < sz; ++i) res.a[i][i] = 1;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) bas.a[i][j] = a[i][j] % MOD;
    while (x) {
      if (x & 1) res = res * bas;
      bas = bas * bas;
      x >>= 1;
    }
    return res;
  }
```

在比赛中，由于线性递推式可以表示成矩阵乘法的形式，也通常用矩阵快速幂来求线性递推数列的某一项。

总代码

```
struct mat {
  LL a[sz][sz];

  mat() { memset(a, 0, sizeof a); }

  mat operator-(const mat& T) const {
    mat res;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) {
        res.a[i][j] = (a[i][j] - T.a[i][j]) % MOD;
      }
    return res;
  }

  mat operator+(const mat& T) const {
    mat res;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) {
        res.a[i][j] = (a[i][j] + T.a[i][j]) % MOD;
      }
    return res;
  }
```

```
  mat operator*(const mat& T) const {
    mat res;
    int r;
    for (int i = 0; i < sz; ++i)
      for (int k = 0; k < sz; ++k) {
        r = a[i][k];
        for (int j = 0; j < sz; ++j)
          res.a[i][j] += T.a[k][j] * r, res.a[i][j] %= MOD;
      }
    return res;
  }

  mat operator^(LL x) const {
    mat res, bas;
    for (int i = 0; i < sz; ++i) res.a[i][i] = 1;
    for (int i = 0; i < sz; ++i)
      for (int j = 0; j < sz; ++j) bas.a[i][j] = a[i][j] % MOD;
    while (x) {
      if (x & 1) res = res * bas;
      bas = bas * bas;
      x >>= 1;
    }
    return res;
  }
};
```

9.3 位运算

9.3.1左移运算和右移运算

以二进制0110举例：0110左移一位就是在右边添加1位0，得出01100，由于固定位数，所以左边超出的1位被舍弃，结果就是1100；0110左移两位就是在右边添加2位0，得出011000，由于固定位数，所以左边超出的2位被舍弃，结果就是1000，以此类推。右移也是同理，0110右移1位就是0011，右移2位就是0001。
在C++中左移的运算符号是`<<`，右移的运算符号是`>>`。`0110`左移2位的代码就是`0110 << 2`，`0110`右移2位的代码就是`0110 >> 2`。

9.3.2与运算

在C++中，与运算的运算符号是&。

1 & 1的结果是1；1 & 0或者0 & 1的结果是0，0 & 0的结果也是0。

而二进制运算0110 & 0100的结果是0100。可以看出与运算就是二进制各个位上对应的数进行与运算，都是1的时候结果是1，有一个0或者都是0的时候结果是0。

9.3.3或运算

在C++中，或运算的运算符号是|。

1 | 1的结果是1；1 | 0或者0 | 1的结果是1，0 | 0的结果也是0。

而二进制运算0110 | 0100的结果是0110。可以看出或运算就是二进制各个位上对应的数进行或运算，都是1或者有一个是1的时候结果是1，都是0的时候结果是0。

9.3.4异或运算

在C++中，异或运算的运算符号是^。

1 ^ 1的结果是0；1 ^ 0或者0 ^ 1的结果是1，0 ^ 0的结果也是0。也就是说，异或相同的数结果是0，异或不同的数结果是1，而二进制运算0110 ^ 0100的结果是0010。

9.3.5取反运算

在C++中，取反运算的运算符号是。取反运算就是二进制各个位上的数，0变为1，1变为0。例如二进制运算0110的结果是1001。

注意：取反运算时，需要注意它的数据类型，不同的数据类型的位数都不相同，而上面也说过左边多出来的0可以省略，所以就算是同一个数，如0b1101，二进制位数是4位时结果是0b0010，二进制位数是8位时结果是0b11110010，它们的值是不一样的。所以，取反必须要注意它的数据类型哟。

9.3.6 汉明权重

就是一个数中非0数字的位数，在位运算中，就是数的二进制形式中1的个数

求一个数的汉明权重可以循环求解：我们不断地去掉这个数在二进制下的最后一位（即右移 位），维护一个答案变量，在除的过程中根据最低位是否为 1 更新答案

```
// 求 x 的汉明权重
int popcount(int x) {
    int cnt = 0;
    while (x) {
        cnt += x & 1;
        x >>= 1;
    }
    return cnt;
}
```

求一个数的汉明权重还可以使用 `lowbit` 操作：我们将这个数不断地减去它的 `lowbit`，直到这个数变为 0。

```
int popcount(int x) {
    int cnt = 0;
    while (x) {
        cnt++;
        x -= x & -x;//x&-x二进制就是x的二进制形式的最低位的1
    }
    return cnt;
}
```

构造汉明权重递增的排列

在 状压 DP中，按照 popcount 递增的顺序枚举有时可以避免重复枚举状态。这是构造汉明权重递增的排列的一大作用。

下面我们来具体探究如何在 $O(n)$ 时间内构造汉明权重递增的排列。

我们知道，一个汉明权重为 n 的最小的整数为 2^n-1。只要可以在常数时间构造出一个整数汉明权重相等的后继，我们就可以通过枚举汉明权重，从 2^n-1 开始不断寻找下一个数的方式，在 $O(n)$ 时间内构造出 $0\sim n$ 的符合要求的排列。

而找出一个数 x 汉明权重相等的后继有这样的思路，以 $(10110)_2$ 为例：

- 把 $(10110)_2$ 最右边的 1 向左移动，如果不能移动，移动它左边的 1，以此类推，得到 $(11010)_2$ 。
- 把得到的 $(11010)_2$ 最后移动的 1 原先的位置一直到最低位的所有 1 都移到最右边。这里最后移动的 1 原来在第三位，所以最后三位 010 要变成 001，得到 $(11001)_2$。

这个过程可以用位运算优化：

```
int t = x + (x & -x);
x = t | ((((t&-t)/(x&-x))>>1)-1);
```

- 第一个步骤中，我们把数 加上它的 `lowbit`，在二进制表示下，就相当于把 x 最右边的连续一段 1 换成它左边的一个 1。如刚才提到的二进制数 $(10110)_2$，它在加上它的 `lowbit` 后是 $(11000)_2$。这其实得到了我们答案的前半部分。
- 我们接下来要把答案后面的1 补齐，t 的 `lowbit` 是x 最右边连续一段 1 最左边的 1 移动后的位置，而 x 的 `lowbit` 则是 x 最右边连续一段 1 最左边的位置。还是以 $(10110)_2$ 为例，$t = (11000)_2$，$\text{lowbit}(t) = (01000)_2$，$\text{lowbit}(x) = (00010)_2$。
- 接下来的除法操作是这种位运算中最难理解的部分，但也是最关键的部分。我们设**原数**最右边连续一段 1最高位的 1 在第 r 位上（位数从 0 开始），最低位的 1 在第l 位，t 的 `lowbit` 等于 `1 << (r+1)` ，x 的 `lowbit` 等于 `1 << l`， `(((t&-t)/(x&-x))>>1)` 得到的，就是 `(1<<(r+1))/(1<<l)/2 = (1<<r)/(1<<l) = 1<<(r-l)` ，在二进制表示下就是1后面跟上$r-l$ 个零，零的个数正好等于连续1 的个数减去 1 。举我们刚才的数为例，$\frac{\text{lowbit(t)}/2}{\text{lowbit(x)}} = \frac{(00100)_2}{(00010)_2} = (00010)_2$ 。把这个数减去 1 得到的就是我们要补全的低位，或上原来的数就可以得到答案。

所以枚举 $0 \sim n$ 按汉明权重递增的排列的完整代码为：

```
    for (int i = 0; (1<<i)-1 <= n; i++) {
        for (int x =(1<<i)-1, t; x <= n; t = x+(x&-x), x = x ? (t|((((t&-t)/(x&-
x))>>1)-1)) :(n+1)) {
            cout<<x<<' ';
        }
    }
```

其中要注意 0 的特判，因为 0 没有相同汉明权重的后继

9.4 多项式与生成函数

代数基本定理

定义

任何复系数一元n 次多项式（n 至少为 1方程在复数域上至少有一根。

由此推出，n 次复系数多项式方程在复数域内有且只有 n 个根，重根按重数计算。

有时这个定理也表述为：

任何一个非零的一元 n 次复系数多项式，都正好有 n 个复数根。

代数基本定理的证明，一般会用到复变函数或者近世代数，因此往往作为一个熟知结论直接应用。

根据代数基本定理，一个复系数多项式 $f(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0$ 一定可以唯一地分解为：

$$f(x)=a_n(x-x_1)^{k_1}(x-x_2)^{k_2}\dots(x-x_t)^{k_t}$$

其中各个根均为复数，。

虚根成对定理

代数基本定理的研究对象是复系数多项式。当对实系数多项式进行研究时，虽然也能分解出复数根，却需要将研究范围扩大，不太方便。

虚根：非实数根。

定理：实系数多项式的根的共轭复数也是该多项式的根。

证明：直接在代数基本定理的等式两端取共轭即证毕。

如果根本身是实数，则取共轭仍为它本身，不受影响。

如果根是虚根，则虚根的共轭复数也是原多项式的根。那么，两个虚根就可以配对。

定理：实数系数方程的共轭虚根一定成对出现，并且共轭虚根的重数相等。

证明：假设一个根为$a+b\mathrm{i}$，则另一个根为$a-b\mathrm{i}$。这意味着在分解式中存在两项：

$$(x-a-b\mathrm{i})(x-a+b\mathrm{i})=x^2-2ax+a^2+b^2$$

可以看到两项乘在一起，各项系数会全部变为实数。这个等式右端的二次实系数多项式整除原始的多项式。

于是，在代数基本定理的等式中，两遍同时除以这个二次三项式，得到的仍旧是实系数多项式的等式。对新等式重复操作，随着次数的下降，若干次后即不存在虚根。

因此，每对共轭虚根的重数相等。证毕。

以下是虚根成对定理的推论：

- 实系数奇次多项式至少有一个实根，并且总共有奇数个实根。
- 实系数偶次多项式可能没有实根，总共有偶数个实根。

称上述二次三项式 $x^2-2ax+a^2+b^2=x^2+px+q$ 为二次实系数不可约因式。不可约是指它在实数范围内不可约。

定理：实系数多项式一定是一次或者二次实系数不可约因式的积。

证明：

只要实系数多项式有一个实根 c，就有一个实系数因式 $x-c$ 和它对应；有一对虚根 $a\pm b\mathrm{i}$，就有一个实系数因式 $x^2-2ax+a^2+b^2$ 和它对应。

因此，只要在原始的代数基本定理分解式中，利用虚根成对定理进行配对，即证毕。

根据虚根成对定理，一个实系数多项式 一定可以唯一地分解为：

$$f(x)=a_n(x-x_1)^{k_1}(x-x_2)^{k_2}\dots(x-x_t)^{k_t}(x^2+p_1x+q_1)^{l_1}(x^2+p_2x+q_2)^{l_2}\dots(x^2+p_sx+q_s)^{l_s}$$

其中各项系数均为实数,$k_1+k_2+\ldots+k_t+2(l_1+l_2+\ldots+l_s)=n$。

林士谔算法

简介

怎样对实系数多项式进行代数基本定理的分解？如果将数域扩充至复数会很复杂。

如果只在实数范围内进行分解，只能保证，当次数大于 2 的时候，一定存在实系数二次三项式因式。

这是因为，如果该多项式有虚根，直接凑出一对共轭虚根即可。如果该多项式只有实根，任取两个实根对应的一次因式乘在一起，也能得到实系数二次三项式因式。

找到二次三项式因式之后，再从二次式中解实根或复根就极为容易。于是便有逐次 **找出一个二次因子** 来求得方程的复根的计算方法，这种方法避免了复数运算。

在 1940 年 8 月、1943 年 8 月和 1947 年 7 月，林士谔先后在 MIT 出版的《数学物理》杂志上接连正式发表了 3 篇关于解算高阶方程式复根方法的论文[1]，每次均有改进。

这个方法今天还在现代计算机中进行快速运算，计算机程序包（如 MATLAB）中的多项式求根程序依据的原理也是这个算法。

过程

要想找到一个二次三项式因子，就要将多项式分解为：

$$f(x)=(x^2+p_1x+q_1)g(x)$$

由于无法一下子找到二次三项式因子，按照迭代求解的思路，对于初始值有：

$$f(x)=(x^2+px+q)g(x)+rx+s$$

会产生一个一次式作为余项。只要余项足够小，即可近似地找到待求因子。

我们希望最终解是初始值加一个偏移修正：

$$\begin{aligned}p_1&=p+dp\\q_1&=q+dq\end{aligned}$$

余式中的两个数(r,s)由除式的给定系数(p,q)决定。有偏导数关系：

$$\begin{aligned}dr&=\frac{\partial r}{\partial p}dp+\frac{\partial r}{\partial q}dq\\ds&=\frac{\partial s}{\partial p}dp+\frac{\partial s}{\partial q}dq\end{aligned}$$

在初始的等式中，被除式$f(x)$是给定的，商式$g(x)$和余式$rx+s$随着除式x^2+px+q的变化而变化。因此有偏导数关系

$$\begin{aligned}0&=xg(x)+\frac{\partial g(x)}{\partial p}(x^2+px+q)+\frac{\partial r}{\partial p}x+\frac{\partial s}{\partial p}\\0&=g(x)+\frac{\partial g(x)}{\partial q}(x^2+px+q)+\frac{\partial r}{\partial q}x+\frac{\partial s}{\partial q}\end{aligned}$$

注意到，偏导数只是一个数值，与变元x无关。因此有整除关系

$$\begin{aligned}xg(x)&=-\frac{\partial g(x)}{\partial p}(x^2+px+q)-\frac{\partial r}{\partial p}x-\frac{\partial s}{\partial p}\\g(x)&=-\frac{\partial g(x)}{\partial q}(x^2+px+q)-\frac{\partial r}{\partial q}x-\frac{\partial s}{\partial q}\end{aligned}$$

这里的结论是，待求的偏导数，恰好是对商式继续做除法的余式。多项式对给定二次三项式的除法，直接计算即可。这里就求得了四个偏导数。

我们希望 s 和 r 加上偏移 ds 与 dr 得到 0，即 ds 与 dr 是 s 和 r 的相反数。因此要解方程：

$$-\frac{\partial r}{\partial p}dp-\frac{\partial r}{\partial q}dq=r$$
$$-\frac{\partial s}{\partial p}dp-\frac{\partial s}{\partial q}dq=s$$

从上述方程组中解得 p 和 q 相应的偏移 dp 和 dq，直接用二阶行列式求解即可。

```
const double eps=1e-7;
// a 是原始的多项式，n 是多项式次数，p 是待求的一次项，q 是待求的常数项
double b[10], c[10];
void Shie(double a[], int n, double *p, double *q) {
  // 数组 b 是多项式 a 除以当前迭代二次三项式的商
  memset(b, 0, sizeof(b));
  // 数组 c 是多项式 b 乘以 x 平方再除以当前迭代二次三项式的商
  memset(c, 0, sizeof(c));
  *p = 0;
  *q = 0;
  double dp = 1;
  double dq = 1;
  while (dp > eps || dp < -eps || dq > eps || dq < -eps)  // eps 自行设定
  {
    double p0 = *p;
    double q0 = *q;
    b[n - 2] = a[n];
    c[n - 2] = b[n - 2];
    b[n - 3] = a[n - 1] - p0 * b[n - 2];
    c[n - 3] = b[n - 3] - p0 * b[n - 2];
    int j;
    for (j = n - 4; j >= 0; j--) {
      b[j] = a[j + 2] - p0 * b[j + 1] - q0 * b[j + 2];
      c[j] = b[j] - p0 * c[j + 1] - q0 * c[j + 2];
    }
    double r = a[1] - p0 * b[0] - q0 * b[1];
    double s = a[0] - q0 * b[0];
    double rp = c[1];
    double sp = b[0] - q0 * c[2];
    double rq = c[0];
    double sq = -q0 * c[1];
    dp = (rp * s - r * sp) / (rp * sq - rq * sp);
    dq = (r * sq - rq * s) / (rp * sq - rq * sp);
    *p += dp;
    *q += dq;
  }
}
```

快速傅里叶变换

前置知识：复数。

本文将介绍一种算法，它支持在 的时间内计算两个 次多项式的乘法，比朴素的 算法更高效。由于两个整数的乘法也可以被当作多项式乘法，因此这个算法也可以用来加速大整数的乘法计算。

引入

我们现在引入两个多项式A 和B：

$$A = 5x^2 + 3x + 7$$
$$B = 7x^2 + 2x + 1$$

两个多项式相乘的积$C = A \times B$，我们可以在 $O(n^2)$ 的时间复杂度中解得（这里n 为 A 或者 B 多项式的次数）：

$$\begin{aligned} C &= A \times B \\ &= 35x^4 + 31x^3 + 60x^2 + 17x + 7 \end{aligned}$$

很明显，多项式C 的系数c_i 满足 $c_i = \sum_{j=0}^{i} a_j b_{i-j}$。而对于这种朴素算法而言，计算每一项的时间复杂度都为$O(n)$，一共有$O(n)$ 项，那么时间复杂度为 $O(n^2)$。

能否加速使得它的时间复杂度降低呢？如果使用快速傅里叶变换的话，那么我们可以使得其复杂度降低到 $O(n \log n)$。

傅里叶变换

傅里叶变换（Fourier Transform）是一种分析信号的方法，它可分析信号的成分，也可用这些成分合成信号。许多波形可作为信号的成分，傅里叶变换用正弦波作为信号的成分。

设 $f(t)$ 是关于时间 t 的函数，则傅里叶变换可以检测频率 ω 的周期在 $f(t)$ 出现的程度：

$$F(\omega) = \mathbb{F}[f(t)] = \int_{-\infty}^{\infty} f(t) \mathrm{e}^{-\mathrm{i}\omega t} dt$$

它的逆变换是

$$f(t) = \mathbb{F}^{-1}[F(\omega)] = \frac{1}{2\pi} \int_{-\infty}^{\infty} F(\omega) \mathrm{e}^{\mathrm{i}\omega t} d\omega$$

逆变换的形式与正变换非常类似，分母 2π 恰好是指数函数的周期。

傅里叶变换相当于将时域的函数与周期为 2π 的复指数函数进行连续的内积。逆变换仍旧为一个内积。

傅里叶变换有相应的卷积定理，可以将时域的卷积转化为频域的乘积，也可以将频域的卷积转化为时域的乘积。

离散傅里叶变换

离散傅里叶变换（Discrete Fourier transform，DFT）是傅里叶变换在时域和频域上都呈离散的形式，将信号的时域采样变换为其 DTFT（discrete-time Fourier transform）的频域采样。

傅里叶变换是积分形式的连续的函数内积，离散傅里叶变换是求和形式的内积。

设 $\{x_n\}_{n=0}^{N-1}$ 是某一满足有限性条件的序列，它的离散傅里叶变换（DFT）为：

$$X_k = \sum_{n=0}^{N-1} x_n \mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}kn}$$

其中e 是自然对数的底数，i是虚数单位。通常以符号 $\mathcal{F}$ 表示这一变换，即

$$\hat{x} = \mathcal{F}x$$

类似于积分形式，它的 **逆离散傅里叶变换**（IDFT）为：

$$x_n = \frac{1}{N}\sum_{k=0}^{N-1} X_k \mathrm{e}^{\mathrm{i}\frac{2\pi}{N}kn}$$

可以记为：

$$x = \mathcal{F}^{-1}\hat{x}$$

实际上，DFT 和 IDFT 变换式中和式前面的归一化系数并不重要。在上面的定义中，DFT 和 IDFT 前的系数分别为 1 和 $\frac{1}{N}$。有时我们会将这两个系数都改 $\frac{1}{\sqrt{N}}$。

离散傅里叶变换仍旧是时域到频域的变换。由于求和形式的特殊性，可以有其他的解释方法。

如果把序列 x_n 看作多项式 $f(x)$ 的 x^n 项系数，则计算得到的 X_k 恰好是多项式 $f(x)$ 代入单位根 $\mathrm{e}^{\frac{-2\pi\mathrm{i}k}{N}}$ 的点值 $f(\mathrm{e}^{\frac{-2\pi\mathrm{i}k}{N}})$。

这便构成了卷积定理的另一种解释办法，即对多项式进行特殊的求值操作。离散傅里叶变换恰好是多项式在单位根处进行求值。

例如计算：

$$\binom{n}{3} + \binom{n}{7} + \binom{n}{11} + \binom{n}{15} + \dots$$

定义函数 为：

$$f(x) = (1+x)^n = \binom{n}{0}x^0 + \binom{n}{1}x^1 + \binom{n}{2}x^2 + \binom{n}{3}x^3 + \dots$$

然后可以发现，代入四次单位根 $f(\mathrm{i})$ 得到这样的序列：

$$f(\mathrm{i}) = (1+\mathrm{i})^n = \binom{n}{0} + \binom{n}{1}\mathrm{i} - \binom{n}{2} - \binom{n}{3}\mathrm{i} + \dots$$

于是下面的求和恰好可以把其余各项消掉：

$$f(1) + \mathrm{i}f(\mathrm{i}) - f(-1) - \mathrm{i}f(-\mathrm{i}) = 4\binom{n}{3} + 4\binom{n}{7} + 4\binom{n}{11} + 4\binom{n}{15} + \dots$$

因此这道数学题的答案为：

$$\binom{n}{3} + \binom{n}{7} + \binom{n}{11} + \binom{n}{15} + \dots = \frac{2^n + \mathrm{i}(1+\mathrm{i})^n - \mathrm{i}(1-\mathrm{i})^n}{4}$$

这道数学题在单位根处求值，恰好构成离散傅里叶变换。

矩阵公式

由于离散傅立叶变换是一个 **线性** 算子，所以它可以用矩阵乘法来描述。在矩阵表示法中，离散傅立叶变换表示如下：

$$\begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \alpha & \alpha^2 & \cdots & \alpha^{N-1} \\ 1 & \alpha^2 & \alpha^4 & \cdots & \alpha^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \alpha^{N-1} & \alpha^{2(N-1)} & \cdots & \alpha^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{N-1} \end{bmatrix}$$

其中 $\alpha = \mathrm{e}^{-\mathrm{i}\frac{2\pi}{N}}$。

快速傅里叶变换

FFT 是一种高效实现 DFT 的算法，称为快速傅立叶变换（Fast Fourier Transform，FFT）。它对傅里叶变换的理论并没有新的发现，但是对于在计算机系统或者说数字系统中应用离散傅立叶变换，可以说是进了一大步。快速数论变换（NTT）是快速傅里叶变换（FFT）在数论基础上的实现。

在 1965 年，Cooley 和 Tukey 发表了快速傅里叶变换算法。事实上 FFT 早在这之前就被发现过了，但是在当时现代计算机并未问世，人们没有意识到 FFT 的重要性。一些调查者认为 FFT 是由 Runge 和 König 在 1924 年发现的。但事实上高斯早在 1805 年就发明了这个算法，但一直没有发表。

分治法实现

FFT 算法的基本思想是分治。就 DFT 来说，它分治地来求当 $x = \omega_n^k$ 的时候 $f(x)$ 的值。基 - 2 FFT 的分治思想体现在将多项式分为奇次项和偶次项处理。

举个例子，对于一共 8 项的多项式：

$$f(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4 + a_5x^5 + a_6x^6 + a_7x^7$$

按照次数的奇偶来分成两组，然后右边提出来一个 x：

$$\begin{aligned} f(x) &= (a_0 + a_2x^2 + a_4x^4 + a_6x^6) + (a_1x + a_3x^3 + a_5x^5 + a_7x^7) \\ &= (a_0 + a_2x^2 + a_4x^4 + a_6x^6) + x(a_1 + a_3x^2 + a_5x^4 + a_7x^6) \end{aligned}$$

分别用奇偶次次项数建立新的函数：

$$\begin{aligned} G(x) &= a_0 + a_2x + a_4x^2 + a_6x^3 \\ H(x) &= a_1 + a_3x + a_5x^2 + a_7x^3 \end{aligned}$$

那么原来的 $f(x)$ 用新函数表示为：

$$f(x) = G\left(x^2\right) + x \times H\left(x^2\right)$$

利用偶数次单位根的性质 $\omega_n^i = -\omega_n^{i+n/2}$，和 $G\left(x^2\right)]$和 $H\left(x^2\right)$ 是偶函数，我们知道在复平面上 ω_n^i 和 $\omega_n^{i+n/2}$ 的 $G(x^2)$ 的 $H(x^2)$ 对应的值相同。得到：

$$\begin{aligned} f(\omega_n^k) &= G((\omega_n^k)^2) + \omega_n^k \times H((\omega_n^k)^2) \\ &= G(\omega_n^{2k}) + \omega_n^k \times H(\omega_n^{2k}) \\ &= G(\omega_{n/2}^k) + \omega_n^k \times H(\omega_{n/2}^k) \end{aligned}$$

和：

$$\begin{aligned} f(\omega_n^{k+n/2}) &= G(\omega_n^{2k+n}) + \omega_n^{k+n/2} \times H(\omega_n^{2k+n}) \\ &= G(\omega_n^{2k}) - \omega_n^k \times H(\omega_n^{2k}) \\ &= G(\omega_{n/2}^k) - \omega_n^k \times H(\omega_{n/2}^k) \end{aligned}$$

因此我们求出了 $G(\omega_{n/2}^k)$ 和 $H(\omega_{n/2}^k)$ 后，就可以同时求出 $f(\omega_n^k)$ 和 $f(\omega_n^{k+n/2})$。于是对 G 和 H 分别递归 DFT 即可。

考虑到分治 DFT 能处理的多项式长度只能是 $2^m(m \in \mathbf{N}^*)$，否则在分治的时候左右不一样长，右边就取不到系数了。所以要在第一次 DFT 之前就把序列向上补成长度为 $2^m(m \in \mathbf{N}^*)$(高次系数补 0) 、最高项次数为 2^m-1 的多项式。

在代入值的时候，因为要代入 n 个不同值，所以我们代入 $\omega_n^0, \omega_n^1, \omega_n^2, \cdots, \omega_n^{n-1}(n=2^m(m \in \mathbf{N}^*))$ 一共 2^m 个不同值。

代码实现方面，STL 提供了复数的模板，当然也可以手动实现。两者区别在于，使用 STL 的 `complex` 可以调用 `exp` 函数求出 ω_n。但事实上使用欧拉公式得到的虚数来求 ω_n 也是等价的。

以上就是 FFT 算法中 DFT 的介绍，它将一个多项式从系数表示法变成了点值表示法。

值的注意的是，因为是单位复根，所以说我们需要令 n 项式的高位补为零，使得 $n=2^k, k \in \mathbf{N}^*$。

```
#include <cmath>
#include <complex>

using Comp = std::complex<double>;  // STL complex

constexpr Comp I(0, 1);  // i
constexpr int MAX_N = 1 << 20;
#define M_PI 3.14159265358979323846
Comp tmp[MAX_N];

// rev=1,DFT; rev=-1,IDFT
void DFT(Comp* f, int n, int rev) {
  if (n == 1) return;
  for (int i = 0; i < n; ++i) tmp[i] = f[i];
  // 偶数放左边，奇数放右边
  for (int i = 0; i < n; ++i) {
    if (i & 1)
      f[n / 2 + i / 2] = tmp[i];
    else
      f[i / 2] = tmp[i];
  }
  Comp *g = f, *h = f + n / 2;
  // 递归 DFT
  DFT(g, n / 2, rev), DFT(h, n / 2, rev);
  // cur 是当前单位复根，对于 k = 0 而言，它对应的单位复根 omega^0_n = 1。
  // step 是两个单位复根的差，即满足 omega^k_n = step*omega^{k-1}*n，
  // 定义等价于 exp(I*(2*M_PI/n*rev))
  Comp cur(1, 0), step(cos(2 * M_PI / n), sin(2 * M_PI * rev / n));
  for (int k = 0; k < n / 2;
       ++k) {  // F(omega^k_n) = G(omega^k*{n/2}) + omega^k*n\*H(omega^k*{n/2})
    tmp[k] = g[k] + cur * h[k];
    // F(omega^{k+n/2}*n) = G(omega^k*{n/2}) - omega^k_n*H(omega^k\_{n/2})
    tmp[k + n / 2] = g[k] - cur * h[k];
    cur *= step;
  }
  for (int i = 0; i < n; ++i) f[i] = tmp[i];
}
```

剩下的有机会再看吧，先把模版贴上

```
// 位逆序置换实现（O(n)）
// 同样需要保证 len 是 2 的幂
// 记 rev[i] 为 i 翻转后的值
void change(Complex y[], int len) {
  for (int i = 0; i < len; ++i) {
    rev[i] = rev[i >> 1] >> 1;
    if (i & 1) {  // 如果最后一位是 1，则翻转成 len/2
      rev[i] |= len >> 1;
    }
  }
  for (int i = 0; i < len; ++i) {
    if (i < rev[i]) {  // 保证每对数只翻转一次
      swap(y[i], y[rev[i]]);
    }
  }
  return;
}
```

```
/*
 * 非递归版 FFT（对应方法一）
 * 做 FFT
 * len 必须是 2^k 形式
 * on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  // 位逆序置换
  change(y, len);
  // 模拟合并过程，一开始，从长度为一合并到长度为二，一直合并到长度为 len。
  for (int h = 2; h <= len; h <<= 1) {
    // wn：当前单位复根的间隔：w^1_h
    Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));
    // 合并，共 len / h 次。
    for (int j = 0; j < len; j += h) {
      // 计算当前单位复根，一开始是 1 = w^0_n，之后是以 wn 为间隔递增： w^1_n
      // ...
      Complex w(1, 0);
      for (int k = j; k < j + h / 2; k++) {
        // 左侧部分和右侧是子问题的解
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        // 这就是把两部分分治的结果加起来
        y[k] = u + t;
        y[k + h / 2] = u - t;
        // 后半个 「step」 中的ω一定和 「前半个」 中的成相反数
        // 「红圈」上的点转一整圈「转回来」，转半圈正好转成相反数
        // 一个数相反数的平方与这个数自身的平方相等
        w = w * wn;
      }
    }
  }
  // 如果是 IDFT，它的逆矩阵的每一个元素不只是原元素取倒数，还要除以长度 len。
  if (on == -1) {
    for (int i = 0; i < len; i++) {
```

```
      y[i].x /= len;
    }
  }
}
```

```
/*
 * 非递归版 FFT（对应方法二）
 * 做 FFT
 * len 必须是 2^k 形式
 * on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  change(y, len);
  for (int h = 2; h <= len; h <<= 1) {              // 模拟合并过程
    Complex wn(cos(2 * PI / h), sin(2 * PI / h));  // 计算当前单位复根
    for (int j = 0; j < len; j += h) {
      Complex w(1, 0);  // 计算当前单位复根
      for (int k = j; k < j + h / 2; k++) {
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        y[k] = u + t;  // 这就是把两部分分治的结果加起来
        y[k + h / 2] = u - t;
        // 后半个 「step」 中的ω一定和 「前半个」 中的成相反数
        // 「红圈」上的点转一整圈「转回来」，转半圈正好转成相反数
        // 一个数相反数的平方与这个数自身的平方相等
        w = w * wn;
      }
    }
  }
  if (on == -1) {
    reverse(y + 1, y + len);
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
    }
  }
}
```

模版（大数乘法）

```
//FFT 模板（HDU 1402 - A * B Problem Plus）
#include <cmath>
#include <cstring>
#include <iostream>

const double PI = acos(-1.0);

struct Complex {
  double x, y;

  Complex(double _x = 0.0, double _y = 0.0) {
    x = _x;
    y = _y;
  }
```

```
  Complex operator-(const Complex &b) const {
    return Complex(x - b.x, y - b.y);
  }

  Complex operator+(const Complex &b) const {
    return Complex(x + b.x, y + b.y);
  }

  Complex operator*(const Complex &b) const {
    return Complex(x * b.x - y * b.y, x * b.y + y * b.x);
  }
};

/*
 * 进行 FFT 和 IFFT 前的反置变换
 * 位置 i 和 i 的二进制反转后的位置互换
 *len 必须为 2 的幂
 */
void change(Complex y[], int len) {
  int i, j, k;

  for (int i = 1, j = len / 2; i < len - 1; i++) {
    if (i < j) std::swap(y[i], y[j]);

    // 交换互为小标反转的元素，i<j 保证交换一次
    // i 做正常的 + 1，j 做反转类型的 + 1，始终保持 i 和 j 是反转的
    k = len / 2;

    while (j >= k) {
      j = j - k;
      k = k / 2;
    }

    if (j < k) j += k;
  }
}

/*
 * 做 FFT
 *len 必须是 2^k 形式
 *on == 1 时是 DFT，on == -1 时是 IDFT
 */
void fft(Complex y[], int len, int on) {
  change(y, len);

  for (int h = 2; h <= len; h <<= 1) {
    Complex wn(cos(2 * PI / h), sin(on * 2 * PI / h));

    for (int j = 0; j < len; j += h) {
      Complex w(1, 0);

      for (int k = j; k < j + h / 2; k++) {
        Complex u = y[k];
        Complex t = w * y[k + h / 2];
        y[k] = u + t;
```

```
        y[k + h / 2] = u - t;
        w = w * wn;
      }
    }
  }

  if (on == -1) {
    for (int i = 0; i < len; i++) {
      y[i].x /= len;
    }
  }
}

constexpr int MAXN = 200020;
Complex x1[MAXN], x2[MAXN];
char str1[MAXN / 2], str2[MAXN / 2];
int sum[MAXN];
using std::cin;
using std::cout;

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  while (cin >> str1 >> str2) {
    int len1 = strlen(str1);
    int len2 = strlen(str2);
    int len = 1;

    while (len < len1 * 2 || len < len2 * 2) len <<= 1;

    for (int i = 0; i < len1; i++) x1[i] = Complex(str1[len1 - 1 - i] - '0', 0);

    for (int i = len1; i < len; i++) x1[i] = Complex(0, 0);

    for (int i = 0; i < len2; i++) x2[i] = Complex(str2[len2 - 1 - i] - '0', 0);

    for (int i = len2; i < len; i++) x2[i] = Complex(0, 0);

    fft(x1, len, 1);
    fft(x2, len, 1);

    for (int i = 0; i < len; i++) x1[i] = x1[i] * x2[i];

    fft(x1, len, -1);

    for (int i = 0; i < len; i++) sum[i] = int(x1[i].x + 0.5);

    for (int i = 0; i < len; i++) {
      sum[i + 1] += sum[i] / 10;
      sum[i] %= 10;
    }

    len = len1 + len2 - 1;

    while (sum[len] == 0 && len > 0) len--;

    for (int i = len; i >= 0; i--) cout << char(sum[i] + '0');
```

```
    cout << '\n';
  }

  return 0;
}
```

多项式初等函数

```
constexpr int MAXN = 262144;
constexpr int mod = 998244353;

using i64 = long long;
using poly_t = int[MAXN];
using poly = int *const;

void derivative(const poly &h, const int n, poly &f) {
  for (int i = 1; i != n; ++i) f[i - 1] = (i64)h[i] * i % mod;
  f[n - 1] = 0;
}

void integrate(const poly &h, const int n, poly &f) {
  for (int i = n - 1; i; --i) f[i] = (i64)h[i - 1] * inv[i] % mod;
  f[0] = 0; /* C */
}

void polyln(const poly &h, const int n, poly &f) {
  /* f = ln h = ∫ h' / h dx */
  assert(h[0] == 1);
  static poly_t ln_t;
  const int t = n << 1;

  derivative(h, n, ln_t);
  std::fill(ln_t + n, ln_t + t, 0);
  polyinv(h, n, f);

  DFT(ln_t, t);
  DFT(f, t);
  for (int i = 0; i != t; ++i) ln_t[i] = (i64)ln_t[i] * f[i] % mod;
  IDFT(ln_t, t);

  integrate(ln_t, n, f);
}

void polyexp(const poly &h, const int n, poly &f) {
  /* f = exp(h) = f_0 (1 - ln f_0 + h) */
  assert(h[0] == 0);
  static poly_t exp_t;
  std::fill(f, f + n + n, 0);
  f[0] = 1;
  for (int t = 2; t <= n; t <<= 1) {
    const int t2 = t << 1;
```

```
    polyln(f, t, exp_t);
    exp_t[0] = sub(pls(h[0], 1), exp_t[0]);
    for (int i = 1; i != t; ++i) exp_t[i] = sub(h[i], exp_t[i]);
    std::fill(exp_t + t, exp_t + t2, 0);

    DFT(f, t2);
    DFT(exp_t, t2);
    for (int i = 0; i != t2; ++i) f[i] = (i64)f[i] * exp_t[i] % mod;
    IDFT(f, t2);

    std::fill(f + t, f + t2, 0);
  }
}
```

9.5 组合数学

贝尔数

贝尔数B_n 以埃里克·坦普尔·贝尔命名，是组合数学中的一组整数数列，开首是（OEIS A000110）：

$B_0 = 1, B_1 = 1, B_2 = 1, B_3 = 1, B_4 = 1, B_5 = 1, \ldots$

B_n 是基数为 n 的集合的划分方法的数目。集合 S 的一个划分是定义为 S 的两两不相交的非空子集的族，它们的并是 S。例如 B_3 因为 3 个元素的集合 $\{a, b, c\}$ 有 5 种不同的划分方法：

$$\{\{a\}, \{b\}, \{c\}\}$$
$$\{\{a\}, \{b, c\}\}$$
$$\{\{b\}, \{a, c\}\}$$
$$\{\{c\}, \{a\ , b\}\}$$
$$\{\{a, b, c\}\}$$

B_0 是 1 因为空集正好有 1 种划分方法。

递归公式

贝尔数适合递推公式

$$B_{n+1} = \sum_{k=0}^{n} \binom{n}{k} B_k$$

证明：

B_{n+1} 是含有 $n+1$ 个元素集合的划分个数，设 B_n 的集合为 $\{b_1, b_2, b_3, \ldots, b_n\}$，!$B_{n+1}$ 的集合为 $\{b_1, b_2, b_3, \ldots, b_n, b_{n+1}\}$，那么可以认为 B_{n+1} 是有 B_n 增添了一个 b_{n+1} 而产生的，考虑元素 b_{n+1} 。

- 假如它被单独分到一类，那么还剩下 n 个元素，这种情况下划分数为 $\binom{n}{n} B_n$;
- 假如它和某 1 个元素分到一类，那么还剩下 $n-1$ 个元素，这种情况下划分数为 $\binom{n}{n-1} B_{n-1}$;
- 假如它和某 2 个元素分到一类，那么还剩下 $n-2$ 个元素，这种情况下划分数为 $\binom{n}{n-2} B_{n-2}$;

- ……

以此类推就得到了上面的公式。

每个贝尔数都是相应的 第二类斯特林数 的和。 因为第二类斯特林数是把基数为 n 的集合划分为正好 k 个非空集的方法数目。

$$B_n = \sum_{k=0}^{n} \left\{ n \atop k \right\}$$

贝尔三角形

用以下方法构造一个三角矩阵（形式类似杨辉三角形）：

- $a_{0,0} = 1$;
- 对于 $n \geq 1$，第 n 行首项等于上一行的末项，即 $a_{n,0} = a_{n-1,n-1}$;
- 对于 $m, n \geq 1$，第n 行第 m 项等于它左边和左上角两个数之和，即 $a_{n,m} = a_{n,m-1} + a_{n-1,m-1}$。

部分结果如下：

$$
\begin{array}{lllllll}
1 & & & & & & \\
1 & 2 & & & & & \\
2 & 3 & 5 & & & & \\
5 & 7 & 10 & 15 & & & \\
15 & 20 & 27 & 37 & 52 & & \\
52 & 67 & 87 & 114 & 151 & 203 & \\
203 & 255 & 322 & 409 & 523 & 674 & 877
\end{array}
$$

每行的首项是贝尔数。可以利用这个三角形来递推求出贝尔数。

```
constexpr int MAXN = 2000 + 5;
int bell[MAXN][MAXN];

void f(int n) {
  bell[0][0] = 1;
  for (int i = 1; i <= n; i++) {
    bell[i][0] = bell[i - 1][i - 1];
    for (int j = 1; j <= i; j++)
      bell[i][j] = bell[i - 1][j - 1] + bell[i][j - 1];
  }
}
```

指数生成函数

考虑贝尔数的指数生成函数及其导函数：

$$
\begin{aligned}
\hat{B}(x) &= \sum_{n=0}^{+\infty} \frac{B_n}{n!} x^n \\
&= 1 + \sum_{n=0}^{+\infty} \frac{B_{n+1}}{(n+1)!} x^{n+1} \\
\hat{B}'(x) &= \sum_{n=0}^{+\infty} \frac{B_{n+1}}{n!} x^n
\end{aligned}
$$

根据贝尔数的递推公式可以得到：

$\frac{B_{n+1}}{n!}=\sum_{k=0}^{n}\frac{1}{(n-k)!}\frac{B_k}{k!}$

这是一个卷积的式子，因此有：

$$\hat{B}'(x)=\mathrm{e}^x\hat{B}(x)$$

这是一个微分方程，解得：

$$\hat{B}(x)=\exp\left(\mathrm{e}^x+C\right)$$

最后当 $x=0$ 时，带入后解得 $C=-1$，得到贝尔数指数生成函数的封闭形式：

$$\hat{B}(x)=\exp\left(\mathrm{e}^x-1\right)$$

预处理出 e^x-1 的前 n 项后做一次 多项式 exp 即可得出贝尔数前 n 项，时间复杂度瓶颈在多项式 exp，可做到 $O(n\log n)$ 的时间复杂度。

9.6多个数的最小公约数（公倍数）

可以发现，当我们求出两个数的 $\gcd$ 时，求最小公倍数是 $O(1)$ 的复杂度。那么对于多个数，我们其实没有必要求一个共同的最大公约数再去处理，最直接的方法就是，当我们算出两个数的 gcd，或许在求多个数的 gcd 时候，我们将它放入序列对后面的数继续求解，那么，我们转换一下，直接将最小公倍数放入序列即可。

10.算法基础

10.1枚举

寻找数组中相反数的对数

```
bool met[MAXN * 2];
memset(met, 0, sizeof(met));
for (int i = 0; i < n; ++i) {
    if (met[MAXN - a[i]]) ++ans;
    met[MAXN + a[i]] = true;
}
```

10.2排序

10.2.1相关STL

```
qsort(arr,n,sizeof(int),compare);//qsort 函数有四个参数：数组名、元素个数、元素大小、比较规则。
//比较函数的一种示例写法为：
int compare(const void *p1, const void *p2)  // int 类型数组的比较函数
{
  int *a = (int *)p1;
  int *b = (int *)p2;
  if (*a > *b)
    return 1;  // 返回正数表示 a 大于 b
  else if (*a < *b)
    return -1;  // 返回负数表示 a 小于 b
  else
    return 0;  // 返回 0 表示 a 与 b 等价
}
```

```
// a[0] .. a[n - 1] 为需要排序的数列
// 对 a 原地排序，将其按从小到大的顺序排列
std::sort(a, a + n);
// cmp 为自定义的比较函数
std::sort(a, a + n, cmp);

std::nth_element(first, nth, last);
std::nth_element(first, nth, last, cmp);
/*它重排 [first, last) 中的元素，使得 nth 所指向的元素被更改为 [first, last) 排好序后该位置会出现的元素。这个新的 nth 元素前的所有元素小于或等于新的 nth 元素后的所有元素。*/
std::stable_sort(first, last);
std::stable_sort(first, last, cmp);
/*稳定排序，保证相等元素排序后的相对位置与原序列相同。
时间复杂度为 O(nlog (n)^2)，当额外内存可用时，复杂度为 O(nlog n)。*/
// mid = first + k
std::partial_sort(first, mid, last);
std::partial_sort(first, mid, last, cmp);
/*将序列中前 k 元素按 cmp 给定的顺序进行原地排序，后面的元素不保证顺序。未指定 cmp 函数时，默认按从小到大的顺序排序。*/
```

10.2.2自定义比较

参见：运算符重载

```
struct data {
  int a, b;

  bool operator<(const data rhs) const {
    return (a == rhs.a) ? (b < rhs.b) : (a < rhs.a);
  }
} da[1009];

bool cmp(const data u1, const data u2) {
  return (u1.a == u2.a) ? (u1.b > u2.b) : (u1.a > u2.a);
}

// ...
std::sort(da + 1, da + 1 + 10);  // 使用结构体中定义的 < 运算符，从小到大排序
std::sort(da + 1, da + 1 + 10, cmp);  // 使用 cmp 函数进行比较，从大到小排序
```

严格弱序

进行排序的运算符必须满足严格弱序，否则会出现不可预料的情况（如运行时错误、无法正确排序）。

严格弱序的要求：

1. $x \not< x$（非自反性）
2. 若 $x < y$，则 $y \not< x$非对称性)
3. 若 $x < y, y < z$，则$x < z$（传递性)
4. 若$x \not< y, y \not< x, y \not< z, z \not< y$，则$x \not< z, z \not< x$ （不可比性的传递性)

常见的错误做法：

- 使用 `<=` 来定义排序中的小于运算符。
- 在调用排序运算符时，读取外部数值可能会改变的数组（常见于最短路算法）。
- 将多个数的最大最小值进行比较的结果作为排序运算符（如皇后游戏/加工生产调度 中的经典错误）。

10.3前缀和&差分

10.3.1前缀和

定义

前缀和可以简单理解为「数列的前n项的和」，是一种重要的预处理方式，能大大降低查询的时间复杂度。

C++ 标准库中实现了前缀和函数 `std::partial_sum`，定义于头文件 `<numeric>` 中。

```
partial_sum(v.begin(),v.end(),ostream_iterator<int>(cout, " "));
partial_sum(v.begin(),v.end(),v1.begin(),op);
```

二维/多维前缀和

基于容斥原理来计算高维前缀和的方法，其优点在于形式较为简单，无需特别记忆，但当维数升高时，其复杂度较高。这里介绍一种基于 DP 计算高维前缀和的方法。该方法即通常语境中所称的

在我的理解中这是求一个边长为2的高维正形的前缀和，从0开始。

每一项的前缀和所有等于其二进制子集的和，***在某俩个数有二进制关系的前提下，进行异或等操作后可变成子集或超集关系时可使用此方法。***

树上前缀和

设 表示结点 到根节点的权值总和。
然后：

- 若是点权，x, y 路径上的和为 $sum_x + sum_y - sum_{lca} - sum_{fa_{lca}}$。
- 若是边权， 路径上的和为 。

  LCA 的求法参见 最近公共祖先。

10.3.2差分

解释

差分是一种和前缀和相对的策略，可以当做是求和的逆运算。

这种策略的定义是令$bi = \begin{cases} a_i - a_{i-1} & i \in [2,n] \\ a_1 & i = 1 \end{cases}$

性质

- a_i的值是 b_i 的前缀和，即 $a_n = \sum_{i=1}^{n} b_i$
- 计算 a_i 的前缀和$sum = \sum_{i=1}^{n} a_i$=$\sum_{i=1}^{n} \sum_{j=1}^{i} b_j = \sum_{j}^{n} (n-j+1)b_j$

C++ 标准库中实现了差分函数 `std::adjacent_difference`，定义于头文件 `<numeric>` 中。

```
adjacent_difference(v.begin(), v.end(), v.begin());
```

10.4二分

10.4.1二分法

定义

二分查找（英语：binary search），也称折半搜索（英语：half-interval search）、对数搜索（英语：logarithmic search），是用来在一个有序数组中查找某一元素的算法。

```
int binary_search(int start, int end, int key) {
  int ret = -1;  // 未搜索到数据返回-1下标
  int mid;
  while (start <= end) {
    mid = start + ((end - start) >> 1);  // 直接平均可能会溢出，所以用这个算法
    if (arr[mid] < key)
      start = mid + 1;
    else if (arr[mid] > key)
      end = mid - 1;
    else {  // 最后检测相等是因为多数搜索情况不是大于就是小于
      ret = mid;
      break;
    }
  }
  return ret;  // 单一出口
}
```

右边不对左边对但是肯不够，所以右边-1左边不-1

```
        while(r<=n) {
            int ll=r, rr=n;
            while (ll < rr)
            {
                int mid = (ll + rr + 1) >> 1;
                if (get(i, mid) == g)
```

```
                ll = mid;
            else
                rr = mid - 1;
        }
        sum = (sum + i * g * (r + rr) * (rr - r + 1) / 2) % mod;
        r = rr + 1;
        if (r <= n)
            g = __gcd(g, arr[r]);
    }
```

10.4.2快速幂

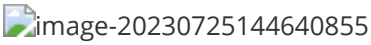
```
int qpow(int a, int n)
{
    if (n == 0)
        return 1;
    else if (n % 2 == 1)
        return qpow(a, n - 1) * a;
    else
    {
        int temp = qpow(a, n / 2);
        return temp * temp;
    }
}
```

10.4.3三分法

引入

如果需要求出单峰函数的极值点，通常使用二分法衍生出的三分法求单峰函数的极值点。

三分法与二分法的基本思想类似，但每次操作需在当前区间 $[l,r]$(下图中除去虚线范围内的部分）内任取两点 $l_{mid},r_{mid}(l_{mid}<r_{mid})$(下图中的两蓝点）。如下图，如果 $f(l_{mid})<f(r_{mid})$，则在 $[r_{mid},r]$(下图中的红色部分）中函数必然单调递增，最小值所在点（下图中的绿点)必然不在这一区间内，可舍去这一区间。反之亦然。



三分法每次操作会舍去两侧区间中的其中一个。为减少三分法的操作次数，应使两侧区间尽可能大。因此，每一次操作时的 l_{mid} 和 r_{mid} 分别取 $mid-\varepsilon$ 和 $mid+\varepsilon$ 是一个不错的选择

```
while (r - l > eps) {
  mid = (lmid + rmid) / 2;
  lmid = mid - eps;
  rmid = mid + eps;
  if (f(lmid) < f(rmid))
    r = mid;
  else
    l = mid;
}
```

10.5 递归与分治

分治法实现数组排序

```
void merge(int arr[],int left,int mid,int right) {
    int arr2[right-left+1];
    int i=left,j=mid+1;
    int k=0;
    while(i <= mid&&j <= right) {
        if(arr[i]<=arr[j])
            arr2[k++]=arr[i++];
        else
            arr2[k++]=arr[j++];
    }
    for(; i<=mid; i++) {
        arr2[k++]=arr[i];
    }
    for(; j<=right; j++) {
        arr2[k++]=arr[j];
    }
    for(i=left,k=0; k<right-left+1; k++,i++)
        arr[i]=arr2[k];
    return;
}
void merge_sort(int arr[],int left,int right) {
    if (left==right) return;
    int mid=left+(right-left)/2;
    merge_sort(arr,left,mid);
    merge_sort(arr,mid+1,right);
    merge(arr,left,mid,right);
}
```

分治法实现二分查找

```
void merge_findMinMax(int arr[], int left, int right, int &minVal, int &maxVal){
    if(left>right)
        return;
    int mid=left+(right-left)/2;
    minVal=min(arr[mid],minVal);
    maxVal=max(arr[mid],maxVal);
    merge_findMinMax(arr,left,mid-1,minVal,maxVal);
    merge_findMinMax(arr,mid+1,right,minVal,maxVal);
}
```

11.图论

11.1树上问题

树的直径

两次DFS

```
const int N = 10000 + 10;

int n, c, d[N];
```

```
vector<int> E[N];

void dfs(int u, int fa) {
  for (int v : E[u]) {
    if (v == fa) continue;
    d[v] = d[u] + 1;
    if (d[v] > d[c]) c = v;
    dfs(v, u);
  }
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    E[u].push_back(v), E[v].push_back(u);
  }
  dfs(1, 0);
  d[c] = 0, dfs(c, 0);
  printf("%d\n", d[c]);
  return 0;
}
```

树形DP

```
const int N = 10000 + 10;

int n, d = 0;
int d1[N], d2[N];
vector<int> E[N];

void dfs(int u, int fa) {
  d1[u] = d2[u] = 0;
  for (int v : E[u]) {
    if (v == fa) continue;
    dfs(v, u);
    int t = d1[v] + 1;
    if (t > d1[u])
      d2[u] = d1[u], d1[u] = t;
    else if (t > d2[u])
      d2[u] = t;
  }
  d = max(d, d1[u] + d2[u]);
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i < n; i++) {
    int u, v;
    scanf("%d %d", &u, &v);
    E[u].push_back(v), E[v].push_back(u);
  }
  dfs(1, 0);
  printf("%d\n", d);
```

```
    return 0;
}
```

最近公共祖先

定义

最近公共祖先简称 LCA（Lowest Common Ancestor）。两个节点的最近公共祖先，就是这两个点的公共祖先里面，离根最远的那个。 为了方便，我们记某点集 $S = \{v_1, v_2, \ldots, v_n\}$ 的最近公共祖先为 $\mathrm{LCA}(v_1, v_2, \ldots, v_n)$ 或 $\mathrm{LCA}(S)$。

性质

1. $\mathrm{LCA}(\{u\}) = u$;
2. u 是 v 的祖先，当且仅当 $\mathrm{LCA}(u, v) = u$;
3. 如果 u 不为 v 的祖先并且 v 不为 u 的祖先，那么 u, v分别处于 $\mathrm{LCA}(u, v)$ 的两棵不同子树中;
4. 前序遍历中，$\mathrm{LCA}(S)$ 出现在所有 S 中元素之前，后序遍历中 $\mathrm{LCA}(S)$ 则出现在所有 S 中元素之后;
5. 两点集并的最近公共祖先为两点集分别的最近公共祖先的最近公共祖先，即 $\mathrm{LCA}(A \cup B) = \mathrm{LCA}(\mathrm{LCA}(A), \mathrm{LCA}(B))$;
6. 两点的最近公共祖先必定处在树上两点间的最短路上;
7. $d(u, v) = h(u) + h(v) - 2h(\mathrm{LCA}(u, v))$，其中 d 是树上两点间的距离，h 代表某点到树根的距离。

朴素算法

过程

可以每次找深度比较大的那个点，让它向上跳。显然在树上，这两个点最后一定会相遇，相遇的位置就是想要求的 LCA。 或者先向上调整深度较大的点，令他们深度相同，然后再共同向上跳转，最后也一定会相遇。

性质

朴素算法预处理时需要 dfs 整棵树，时间复杂度为 $O(n)$，单次查询时间复杂度为 $\Theta(n)$。但由于随机树高为 $O(\log n)$，所以朴素算法在随机树上的单次查询时间复杂度为 $O(\log n)$。

倍增算法

过程

倍增算法是最经典的 LCA 求法，他是朴素算法的改进算法。通过预处理 $\mathrm{fa}_{x,i}$ 数组，游标可以快速移动，大幅减少了游标跳转次数。$\mathrm{fa}_{x,i}$ 表示点 x 的第 2^i 个祖先。$\mathrm{fa}_{x,i}$ 数组可以通过 dfs 预处理出来。

现在我们看看如何优化这些跳转： 在调整游标的第一阶段中，我们要将 u, v 两点跳转到同一深度。我们可以计算出 u, v 两点的深度之差，设其为 y。通过将 y 进行二进制拆分，我们将 y 次游标跳转优化为「y 的二进制表示所含 `1` 的个数」次游标跳转。 在第二阶段中，我们从最大的 i 开始循环尝试，一直尝试到 0(包括 !0) ，如果 $\mathrm{fa}_{u,i} \neq \mathrm{fa}_{v,i}$，则 $u \leftarrow \mathrm{fa}_{u,i}, v \leftarrow \mathrm{fa}_{v,i}$，那么最后的 LCA 为 $\mathrm{fa}_{u,0}$。

性质

倍增算法的预处理时间复杂度为 $O(n \log n)$，单次查询时间复杂度为 $O(\log n)$。 另外倍增算法可以通过交换 `fa` 数组的两维使较小维放在前面。这样可以减少 cache miss 次数，提高程序效率。

动态树模版

```
#include<bits/stdc++.h>
#define R register int
#define I inline void
#define G if(++ip==ie)if(fread(ip=buf,1,SZ,stdin))
#define lc c[x][0]
#define rc c[x][1]
using namespace std;
const int SZ=1<<19,N=3e5+9;
char buf[SZ],*ie=buf+SZ,*ip=ie-1;
inline int in(){
    G;while(*ip<'-')G;
    R x=*ip&15;G;
    while(*ip>'-'){x*=10;x+=*ip&15;G;}
    return x;
}
int f[N],c[N][2],v[N],s[N],st[N];
bool r[N];
inline bool nroot(R x){//判断节点是否为一个Splay的根（与普通Splay的区别1）
    return c[f[x]][0]==x||c[f[x]][1]==x;
}//原理很简单，如果连的是轻边，他的父亲的儿子里没有它
I pushup(R x){//上传信息
    s[x]=s[lc]+s[rc]+v[x];//路径值操作
}
I pushr(R x){R t=lc;lc=rc;rc=t;r[x]^=1;}//翻转操作
I pushdown(R x){//判断并释放懒标记
    if(r[x]){
        if(lc)pushr(lc);
        if(rc)pushr(rc);
        r[x]=0;
    }
}
I rotate(R x){//一次旋转
    R y=f[x],z=f[y],k=c[y][1]==x,w=c[x][!k];
    if(nroot(y))c[z][c[z][1]==y]=x;c[x][!k]=y;c[y][k]=w;//额外注意if(nroot(y))语句，此处不判断会引起致命错误（与普通Splay的区别2）
    if(w)f[w]=y;f[y]=x;f[x]=z;
    pushup(y);
}
I splay(R x){//只传了一个参数，因为所有操作的目标都是该Splay的根（与普通Splay的区别3）
    R y=x,z=0;
    st[++z]=y;//st为栈，暂存当前点到根的整条路径，pushdown时一定要从上往下放标记（与普通Splay的区别4）
    while(nroot(y))st[++z]=y=f[y];
    while(z)pushdown(st[z--]);
    while(nroot(x)){
        y=f[x];z=f[y];
        if(nroot(y))
            rotate((c[y][0]==x)^(c[z][0]==y)?x:y);
        rotate(x);
    }
    pushup(x);
}
/*当然了，其实利用函数堆栈也很方便，代替上面的手工栈，就像这样
```

```
I pushall(R x){
    if(nroot(x))pushall(f[x]);
    pushdown(x);
}*/
I access(R x){//访问
    for(R y=0;x;x=f[y=x])
        splay(x),rc=y,pushup(x);
}
I makeroot(R x){//换根
    access(x);splay(x);
    pushr(x);
}
int findroot(R x){//找根（在真实的树中的）
    access(x);splay(x);
    while(lc)pushdown(x),x=lc;
    splay(x);
    return x;
}
I split(R x,R y){//提取路径
    makeroot(x);
    access(y);splay(y);
}
I link(R x,R y){//连边
    makeroot(x);
    if(findroot(y)!=x)f[x]=y;
}
I cut(R x,R y){//断边
    makeroot(x);
    if(findroot(y)==x&&f[y]==x&&!c[y][0]){
        f[y]=c[x][1]=0;
        pushup(x);
    }
}
int main()
{
    R n=in(),m=in();
    for(R i=1;i<=n;++i)v[i]=in();
    while(m--){
        R type=in(),x=in(),y=in();
        switch(type){
        case 0:split(x,y);printf("%d\n",s[y]);break;
        case 1:link(x,y);break;
        case 2:cut(x,y);break;
        case 3:splay(x);v[x]=y;//先把x转上去再改，不然会影响Splay信息的正确性
        }
    }
    return 0;
}
```

```
#include <iostream>
#define ls(x) T[x][0]
#define rs(x) T[x][1]
#define ms(x) T[x][2]
using namespace std;
```

```cpp
constexpr int MAXN = 300005;

int T[MAXN][3], s[MAXN][2], tot, v[MAXN], r[MAXN], top, st[MAXN], f[MAXN];

int new_node() {
  if (top) {
    top--;
    return st[top + 1];
  }
  return ++tot;
}

bool isroot(int x) { return rs(f[x]) != x && ls(f[x]) != x; }

bool direction(int x) { return rs(f[x]) == x; }

void pushup(int x, int ty) {
  if (ty) {
    s[x][1] = s[ls(x)][1] ^ s[rs(x)][1] ^ s[ms(x)][1];
    return;
  }
  //链路
  s[x][0] = s[ls(x)][0] ^ v[x] ^ s[rs(x)][0];
  s[x][1] = s[ls(x)][1] ^ s[ms(x)][1] ^ s[rs(x)][1] ^ v[x];
}

void pushrev(int x) {
  if (!x) return;
  r[x] ^= 1;
  swap(ls(x), rs(x));
}

void pushdown(int x, int ty) {
  if (ty) return;
  if (r[x]) {
    pushrev(ls(x));
    pushrev(rs(x));
    r[x] = 0;
  }
}

void pushall(int x, int ty) {
  if (!isroot(x)) pushall(f[x], ty);
  pushdown(x, ty);
}

void setfather(int x, int fa, int ty) {
  if (x) f[x] = fa;
  T[fa][ty] = x;
}

void rotate(int x, int ty) {
  int y = f[x], z = f[y], d = direction(x), w = T[x][d ^ 1];
  if (z) T[z][ms(z) == y ? 2 : direction(y)] = x;
  T[x][d ^ 1] = y;
  T[y][d] = w;
```

```
  if (w) f[w] = y;
  f[y] = x;
  f[x] = z;
  pushup(y, ty);
  pushup(x, ty);
}

void splay(int x, int ty, int gl = 0) {
  pushall(x, ty);
  for (int y; y = f[x], (!isroot(x)) && y != gl; rotate(x, ty)) {
    if (f[y] != gl && (!isroot(y))) rotate(direction(x) ^ direction(y) ? x : y,
ty);
  }
}

void clear(int x) {
  ls(x) = ms(x) = rs(x) = s[x][0] = s[x][1] = r[x] = v[x] = 0;
  st[++top] = x;
}

void Delete(int x) {
  setfather(ms(x), f[x], 1);
  if (ls(x)) {
    int p = ls(x);
    pushdown(p, 1);
    while (rs(p)) p = rs(p), pushdown(p, 1);
    splay(p, 1, x);
    setfather(rs(x), p, 1);
    setfather(p, f[x], 2);
    pushup(p, 1);
    pushup(f[x], 0);
  } else
    setfather(rs(x), f[x], 2);
  clear(x);
}

void splice(int x) {
  splay(x, 1);
  int y = f[x];
  splay(y, 0);
  pushdown(x, 1);
  if (rs(y)) {
    swap(f[ms(x)], f[rs(y)]);
    swap(ms(x), rs(y));
    pushup(x, 1);
  } else
    Delete(x);
  pushup(rs(y), 0);
  pushup(y, 0);
}

void access(int x) {
  splay(x, 0);
  int ys = x;
  if (rs(x)) {
    int y = new_node();
```

```
    setfather(ms(x), y, 0);
    setfather(rs(x), y, 2);
    rs(x) = 0;
    setfather(y, x, 2);
    pushup(y, 1);
    pushup(x, 0);
  }
  while (f[x]) {
    splice(f[x]);
    x = f[x];
  }
  splay(ys, 0);
}

int find_root(int x) {
  access(x);
  pushdown(x, 0);
  while (ls(x)) x = ls(x), pushdown(x, 0);
  splay(x, 0);
  return x;
}

void make_root(int x) {
  access(x);
  pushrev(x);
}

void expose(int x, int y) {
  make_root(x);
  access(y);
}

void Link(int x, int y) {
  if (find_root(x) == find_root(y)) return;
  access(x);
  make_root(y);
  setfather(y, x, 1);
  pushup(x, 0);
  pushup(y, 0);
}

void cut(int x, int y) {
  expose(x, y);
  if (ls(y) != x || rs(x)) return;
  f[x] = ls(y) = 0;
  pushup(y, 0);
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  int n, m;
  cin >> n >> m;
  tot = n;
  for (int i = 1; i <= n; i++) {
    cin >> v[i];
    pushup(i, 0);
```

```
  }
  for (int i = 1; i <= m; i++) {
    int op, u, v;
    cin >> op >> u >> v;
    if (op == 0) {
      expose(u, v);
      cout << s[v][0] << '\n';
    }
    if (op == 1) link(u, v);
    if (op == 2) cut(u, v);
    if (op == 3) {
      access(u);
      v[u] = v;
      pushup(u, 0);
    }
  }
  return 0;
}
```

```
void pushup(int x, int op) {
  if (op == 0) {
    // 是 Compress Node
    len[x] = len[ls(x)] + len[rs(x)];
    diam[x] = maxs[ls(x)][1] + maxs[rs(x)][0];
    diam[x] =
        max(diam[x], max(maxs[ls(x)][1], maxs[rs(x)][0]) + maxs[ms(x)][0]);
    diam[x] = max(diam[x], max(max(diam[ls(x)], diam[rs(x)]), diam[ms(x)]));
    maxs[x][0] =
        max(maxs[ls(x)][0], len[ls(x)] + max(maxs[ms(x)][0], maxs[rs(x)][0]));
    maxs[x][1] =
        max(maxs[rs(x)][1], len[rs(x)] + max(maxs[ms(x)][0], maxs[ls(x)][1]));
  } else {
    // 是 Rake Node
    diam[x] = maxs[ls(x)][0] + maxs[rs(x)][0];
    diam[x] =
        max(diam[x], maxs[ms(x)][0] + max(maxs[ls(x)][0], maxs[rs(x)][0]));
    diam[x] = max(max(diam[x], diam[ms(x)]), max(diam[ls(x)], diam[rs(x)]));
    maxs[x][0] = max(maxs[ms(x)][0], max(maxs[ls(x)][0], maxs[rs(x)][0]));
  }
  return;
}
```

12数据结构

12.1哈希表

哈希表又称散列表，一种以「key-value」形式存储数据的数据结构。所谓以「key-value」形式存储数据，是指任意的键值 key 都唯一对应到内存中的某个位置。只需要输入查找的键值，就可以快速地找到其对应的 value。可以把哈希表理解为一种高级的数组，这种数组的下标可以是很大的整数，浮点数，字符串甚至结构体。

拉链法

拉链法也称开散列法（open hashing）。

拉链法是在每个存放数据的地方开一个链表，如果有多个键值索引到同一个地方，只用把他们都放到那个位置的链表里就行了。查询的时候需要把对应位置的链表整个扫一遍，对其中的每个数据比较其键值与查询的键值是否一致。如果索引的范围是 $1 \ldots M$，哈希表的大小为 N，那么一次插入/查询需要进行期望 $O(\frac{N}{M})$ 次比较。

```
const int SIZE = 1000000;
const int M = 999997;

struct HashTable {
  struct Node {
    int next, value, key;
  } data[SIZE];

  int head[M], size;

  int f(int key) { return (key % M + M) % M; }

  int get(int key) {
    for (int p = head[f(key)]; p; p = data[p].next)
      if (data[p].key == key) return data[p].value;
    return -1;
  }

  int modify(int key, int value) {
    for (int p = head[f(key)]; p; p = data[p].next)
      if (data[p].key == key) return data[p].value = value;
  }

  int add(int key, int value) {
    if (get(key) != -1) return -1;
    data[++size] = (Node){head[f(key)], value, key};
    head[f(key)] = size;
    return value;
  }
};
```

这里再提供一个封装过的模板，可以像 map 一样用，并且较短

```
struct hash_map {  // 哈希表模板
  struct data {
    long long u;
    int v, nex;
  };  // 前向星结构
  data e[SZ << 1];  // SZ 是 const int 表示大小
  int h[SZ], cnt;
  int hash(long long u) { return (u % SZ + SZ) % SZ; }
  // 这里使用 (u % SZ + SZ) % SZ 而非 u % SZ 的原因是
  // C++ 中的 % 运算无法将负数转为正数
  int& operator[](long long u) {
    int hu = hash(u);  // 获取头指针
```

```
    for (int i = h[hu]; i; i = e[i].nex)
      if (e[i].u == u) return e[i].v;
    return e[++cnt] = (data){u, -1, h[hu]}, h[hu] = cnt, e[cnt].v;
  }
  hash_map() {
    cnt = 0;
    memset(h, 0, sizeof(h));
  }
};
```

闭散列法

闭散列方法把所有记录直接存储在散列表中，如果发生冲突则根据某种方式继续进行探查。

比如线性探查法：如果在 `d` 处发生冲突，就依次检查 `d + 1`，`d + 2`.....

```
const int N = 360007;  // N 是最大可以存储的元素数量

class Hash {
 private:
  int keys[N];
  int values[N];

 public:
  Hash() { memset(values, 0, sizeof(values)); }

  int& operator[](int n) {
    // 返回一个指向对应 Hash[Key] 的引用
    // 修改成不为 0 的值 0 时候视为空
    int idx = (n % N + N) % N, cnt = 1;
    while (keys[idx] != n && values[idx] != 0) {
      idx = (idx + cnt * cnt) % N;
      cnt += 1;
    }
    keys[idx] = n;
    return values[idx];
  }
};
```

12.2并查集

引入

并查集是一种用于管理元素所属集合的数据结构，实现为一个森林，其中每棵树表示一个集合，树中的节点表示对应集合中的元素。

顾名思义，并查集支持两种操作：

- 合并（Union）：合并两个元素所属集合（合并对应的树）
- 查询（Find）：查询某个元素所属集合（查询对应的树的根节点），这可以用于判断两个元素是否属于同一集合

并查集在经过修改后可以支持单个元素的删除、移动；使用动态开点线段树还可以实现可持久化并查集。

初始化

初始时，每个元素都位于一个单独的集合，表示为一棵只有根节点的树。方便起见，我们将根节点的父亲设为自己。

```
struct dsu {
  vector<size_t> pa;

  explicit dsu(size_t size) : pa(size) { iota(pa.begin(), pa.end(), 0); }
};
```

查询

我们需要沿着树向上移动，直至找到根节点。

```
size_t dsu::find(size_t x) { return pa[x] == x ? x : find(pa[x]); }
```

路径压缩

查询过程中经过的每个元素都属于该集合，我们可以将其直接连到根节点以加快后续查询。

```
size_t dsu::find(size_t x) { return pa[x] == x ? x : pa[x] = find(pa[x]); }
```

合并

要合并两棵树，我们只需要将一棵树的根节点连到另一棵树的根节点。

```
void dsu::unite(size_t x, size_t y) { pa[find(x)] = find(y); }
```

启发式合并

合并时，选择哪棵树的根节点作为新树的根节点会影响未来操作的复杂度。我们可以将节点较少或深度较小的树连到另一棵，以免发生退化。

```
struct dsu {
  vector<size_t> pa, size;

  explicit dsu(size_t size_) : pa(size_), size(size_, 1) {
    iota(pa.begin(), pa.end(), 0);
  }
  void unite(size_t x, size_t y) {
    x = find(x), y = find(y);
    if (x == y) return;
    if (size[x] < size[y]) swap(x, y);
    pa[y] = x;
    size[x] += size[y];
  }
};
```

删除

要删除一个叶子节点，我们可以将其父亲设为自己。为了保证要删除的元素都是叶子，我们可以预先为每个节点制作副本，并将其副本作为父亲。

```
struct dsu {
  vector<size_t> pa, size;

  explicit dsu(size_t size_) : pa(size_ * 2), size(size_ * 2, 1) {
    iota(pa.begin(), pa.begin() + size_, size_);
    iota(pa.begin() + size_, pa.end(), size_);
  }
  void erase(size_t x) {
    --size[find(x)];
    pa[x] = x;
  }
};
```

移动

与删除类似，通过以副本作为父亲，保证要移动的元素都是叶子。

```
void dsu::move(size_t x, size_t y) {
  auto fx = find(x), fy = find(y);
  if (fx == fy) return;
  pa[x] = fy;
  --size[fx], ++size[fy];
}
```

完整代码

```
struct dsu {
    vector<size_t> pa, size;
    explicit dsu(size_t size_) : pa(size_ * 2), size(size_ * 2, 1) {
        iota(pa.begin(), pa.begin() + size_, size_);
        iota(pa.begin() + size_, pa.end(), size_);
    }
    size_t find(size_t x) {
        return pa[x] == x ? x : pa[x] = find(pa[x]);
    }
    void unite(size_t x, size_t y) {
        x = find(x), y = find(y);
        if (x == y) return;
        if (size[x] < size[y]) swap(x, y);
        pa[y] = x;
        size[x] += size[y];
    }
    void erase(size_t x) {
        --size[find(x)];
        pa[x] = x;
    }
    void move(size_t x, size_t y) {
        auto fx = find(x), fy = find(y);
        if (fx == fy) return;
        pa[x] = fy;
        --size[fx], ++size[fy];
    }
};
```

ST表

```cpp
#include <bits/stdc++.h>
using namespace std;

template <typename T>
class SparseTable {
  using VT = vector<T>;
  using VVT = vector<VT>;
  using func_type = function<T(const T &, const T &)>;

  VVT ST;

  static T default_func(const T &t1, const T &t2) { return max(t1, t2); }

  func_type op;

 public:
  SparseTable(const vector<T> &v, func_type _func = default_func) {
    op = _func;
    int len = v.size(), l1 = ceil(log2(len)) + 1;
    ST.assign(len, VT(l1, 0));
    for (int i = 0; i < len; ++i) {
      ST[i][0] = v[i];
    }
    for (int j = 1; j < l1; ++j) {
      int pj = (1 << (j - 1));
      for (int i = 0; i + pj < len; ++i) {
        ST[i][j] = op(ST[i][j - 1], ST[i + (1 << (j - 1))][j - 1]);
      }
    }
  }

  T query(int l, int r) {
    int lt = r - l + 1;
    int q = floor(log2(lt));
    return op(ST[l][q], ST[r - (1 << q) + 1][q]);
  }
};
```

红黑树

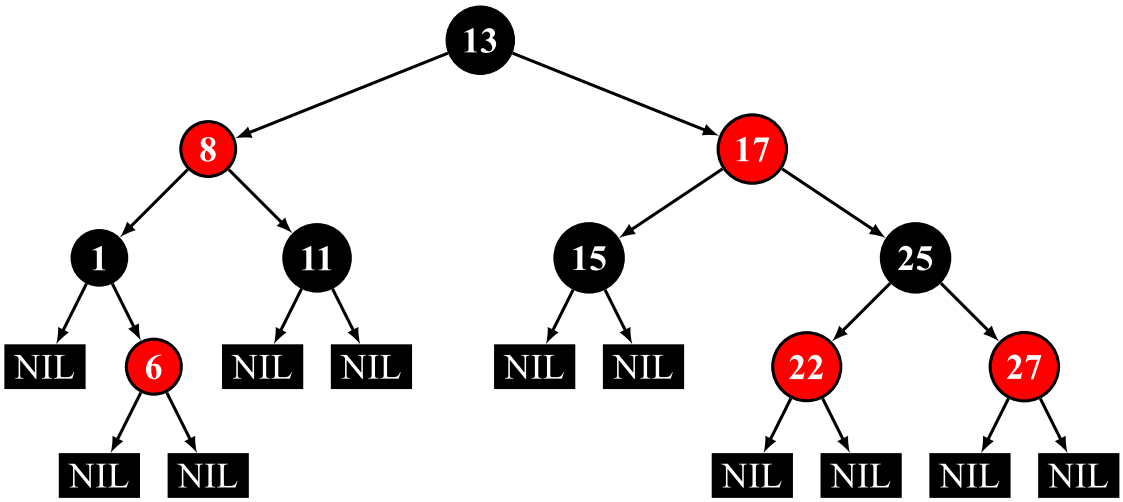
红黑树是一种自平衡的二叉搜索树。每个节点额外存储了一个 color 字段 ("RED" or "BLACK")，用于确保树在插入和删除时保持平衡。

性质

一棵合法的红黑树必须遵循以下四条性质：

1. 节点为红色或黑色
2. NIL 节点（空叶子节点）为黑色
3. 红色节点的子节点为黑色
4. 从根节点到 NIL 节点的每条路径上的黑色节点数量相同

下图为一棵合法的红黑树：



注：部分资料中还加入了第五条性质，即根节点必须为黑色，这条性质要求完成插入操作后若根节点为红色则将其染黑，但由于将根节点染黑的操作也可以延迟至删除操作时进行，因此，该条性质并非必须满足。（在本文给出的代码实现中就没有选择满足该性质）。为严谨起见，这里同时引用 维基百科原文 进行说明：

> Some authors, e.g. Cormen & al.,1claim "the root is black" as fifth requirement; but not Mehlhorn & Sanders2or Sedgewick & Wayne.3Since the root can always be changed from red to black, this rule has little effect on analysis. This article also omits it, because it slightly disturbs the recursive algorithms and proofs.

结构

红黑树类的定义

```
template <typename Key, typename Value, typename Compare = std::less<Key>> class RBTreeMap {  // 排序函数
    Compare compare = Compare();   // 节点结构体
    struct Node {    ...  };   // 根节点指针  Node* root = nullptr;  // 记录红黑树中当前的节点个数
    size_t count = 0;
}
```

节点维护的信息

Identifier	Type	Description
`left`	`Node*`	左子节点指针
`right`	`Node*`	右子节点指针
`parent`	`Node*`	父节点指针
`color`	`enum { BLACK, RED }`	颜色枚举
`key`	`Key`	节点键值，具有唯一性和可排序性

Identifier	Type	Description
`value`	`Value`	节点内储存的值

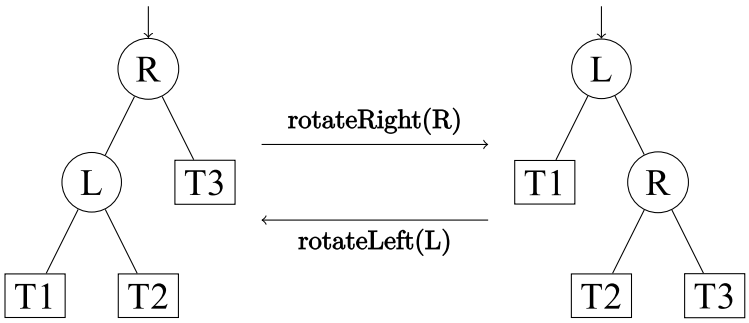
注：由于本文提供的代码示例中使用 `std::share_ptr` 进行内存管理，对此不熟悉的读者可以将下文中所有的 `NodePtr` 和 `ConstNodePtr` 理解为裸指针 `Node*`。但在实现删除操作时若使用 `Node*` 作为节点引用需注意应手动释放内存以避免内存泄漏，该操作在使用 `std::shared_ptr` 作为节点引用的示例代码中并未体现。

过程

注：由于红黑树是由 B 树衍生而来（发明时的最初的名字 symmetric binary B-tree 足以证明这点），并非直接由平衡二叉树外加限制条件推导而来，插入操作的后续维护和删除操作的后续维护中部分对操作的解释作用仅是帮助理解，并不能将其作为该操作的原理推导和证明。

旋转操作

旋转操作是多数平衡树能够维持平衡的关键，它能在不改变一棵合法 BST 中序遍历结果的情况下改变局部节点的深度。



如上图，从左图到右图的过程被称为右旋，右旋操作会使得 子树上结点的深度均减 1，使 子树上结点的深度均加 1，而 子树上节点的深度则不变。从右图到左图的过程被称为左旋，左旋是右旋的镜像操作。

这里给出红黑树中节点的左旋操作的示例代码：

```
void rotateLeft(ConstNodePtr node) {
  // clang-format off
  //     |                        |
  //     N                        S
  //    / \     l-rotate(N)      / \
  //   L   S    ==========>     N   R
  //      / \                  / \
  //     M   R                L   M
  assert(node != nullptr && node->right != nullptr);
  // clang-format on
  NodePtr parent = node->parent;
  Direction direction = node->direction();

  NodePtr successor = node->right;
  node->right = successor->left;
  successor->left = node;
```

```
  // 以下的操作用于维护各个节点的`parent`指针
  // `Direction`的定义以及`maintainRelationship`
  // 的实现请参照文章末尾的完整示例代码
  maintainRelationship(node);
  maintainRelationship(successor);

  switch (direction) {
    case Direction::ROOT:
      this->root = successor;
      break;
    case Direction::LEFT:
      parent->left = successor;
      break;
    case Direction::RIGHT:
      parent->right = successor;
      break;
  }

  successor->parent = parent;
}
```

注：代码中的 `successor` 并非平衡树中的后继节点，而是表示取代原本节点的新节点，由于在图示中 `replacement` 的简称 `R` 会与右子节点的简称 `R` 冲突，因此此处使用 `successor` 避免歧义。

插入操作

红黑树的插入操作与普通的 BST 类似，对于红黑树来说，新插入的节点初始为红色，完成插入后需根据插入节点及相关节点的状态进行修正以满足上文提到的四条性质。

插入后的平衡维护

Case 1

该树原先为空，插入第一个节点后不需要进行修正。

Case 2

当前的节点的父节点为黑色且为根节点，这时性质已经满足，不需要进行修正。

Case 3

当前节点 N 的父节点 P 是为根节点且为红色，将其染为黑色即可，此时性质也已满足，不需要进一步修正。

```
// clang-format off
// Case 3: Parent is root and is RED
//   Paint parent to BLACK.
//    <P>         [P]
//     |   ====>   |
//    <N>         <N>
//   p.s.
//    `<X>` is a RED node;
//    `[X]` is a BLACK node (or NIL);
//    `{X}` is either a RED node or a BLACK node;
// clang-format on
assert(node->parent->isRed());
node->parent->color = Node::BLACK;
```

```
return;
```

Case 4

当前节点 N 的父节点 P 和叔节点 U 均为红色，此时 P 包含了一个红色子节点，违反了红黑树的性质，需要进行重新染色。由于在当前节点 N 之前该树是一棵合法的红黑树，根据性质 3 可以确定 N 的祖父节点 G 一定是黑色，这时只要后续操作可以保证以 G 为根节点的子树在不违反性质 4 的情况下再递归维护祖父节点 G 以保证性质 3 即可。

因此，这种情况的维护需要：

1. 将 P，U 节点染黑，将 G 节点染红（可以保证每条路径上黑色节点个数不发生改变）。
2. 递归维护 G 节点（因为不确定 G 的父节点的状态，递归维护可以确保性质 3 成立）。

```
// clang-format off
// Case 4: Both parent and uncle are RED
//   Paint parent and uncle to BLACK;
//   Paint grandparent to RED.
//        [G]             <G>
//        / \             / \
//      <P> <U>  ====>  [P] [U]
//      /               /
//    <N>             <N>
// clang-format on
assert(node->parent->isRed());
node->parent->color = Node::BLACK;
node->uncle()->color = Node::BLACK;
node->grandParent()->color = Node::RED;
maintainAfterInsert(node->grandParent());
return;
```

Case 5

当前节点 N 与父节点 P 的方向相反（即 N 节点为右子节点且父节点为左子节点，或 N 节点为左子节点且父节点为右子节点。类似 AVL 树中 LR 和 RL 的情况）。根据性质 4，若 N 为新插入节点，U 则为 NIL 黑色节点，否则为普通黑色节点。

该种情况无法直接进行维护，需要通过旋转操作将子树结构调整为 Case 6 的初始状态并进入 Case 6 进行后续维护。

```
// clang-format off
// Case 5: Current node is the opposite direction as parent
//   Step 1. If node is a LEFT child, perform l-rotate to parent;
//           If node is a RIGHT child, perform r-rotate to parent.
//   Step 2. Goto Case 6.
//      [G]                 [G]
//      / \    rotate(P)    / \
//    <P> [U]  ========>  <N> [U]
//      \                   /
//      <N>               <P>
// clang-format on
```

```
// Step 1: Rotation
NodePtr parent = node->parent;
if (node->direction() == Direction::LEFT) {
  rotateRight(node->parent);
} else /* node->direction() == Direction::RIGHT */ {
  rotateLeft(node->parent);
}
node = parent;
// Step 2: vvv
```

Case 6

当前节点 N 与父节点 P 的方向相同（即 N 节点为右子节点且父节点为右子节点，或 N 节点为左子节点且父节点为右子节点。类似 AVL 树中 LL 和 RR 的情况）。根据性质 4，若 N 为新插入节点，U 则为 NIL 黑色节点，否则为普通黑色节点。

在这种情况下，若想在不改变结构的情况下使得子树满足性质 3，则需将 G 染成红色，将 P 染成黑色。但若这样维护的话则性质 4 被打破，且无法保证在 G 节点的父节点上性质 3 是否成立。而选择通过旋转改变子树结构后再进行重新染色即可同时满足性质 3 和 4。

因此，这种情况的维护需要：

1. 若 N 为左子节点则右旋祖父节点 G，否则左旋祖父节点 G.（该操作使得旋转过后 P - N 这条路径上的黑色节点个数比 P - G - U 这条路径上少 1，暂时打破性质 4）。
2. 重新染色，将 P 染黑，将 G 染红，同时满足了性质 3 和 4。

```
// clang-format off
// Case 6: Current node is the same direction as parent
//   Step 1. If node is a LEFT child, perform r-rotate to grandparent;
//           If node is a RIGHT child, perform l-rotate to grandparent.
//   Step 2. Paint parent (before rotate) to BLACK;
//           Paint grandparent (before rotate) to RED.
//        [G]                 <P>               [P]
//        / \    rotate(G)    / \    repaint    / \
//      <P> [U]  ========>  <N> [G]  =======>  <N> <G>
//      /                        \                \
//    <N>                        [U]              [U]
// clang-format on
assert(node->grandParent() != nullptr);

// Step 1
if (node->parent->direction() == Direction::LEFT) {
  rotateRight(node->grandParent());
} else {
  rotateLeft(node->grandParent());
}

// Step 2
node->parent->color = Node::BLACK;
node->sibling()->color = Node::RED;

return;
```

删除操作

红黑树的删除操作情况繁多，较为复杂。这部分内容主要通过代码示例来进行讲解。大多数红黑树的实现选择将节点的删除以及删除之后的维护写在同一个函数或逻辑块中（例如 Wikipedia 给出的 代码示例，linux 内核中的 rbtree 以及 GNU libstdc++ 中的 `std::_Rb_tree` 都使用了类似的写法）。笔者则认为这种实现方式并不利于对算法本身的理解，因此，本文给出的示例代码参考了 OpenJDK 中 `TreeMap` 的实现，将删除操作本身与删除后的平衡维护操作解耦成两个独立的函数，并对这两部分的逻辑单独进行分析。

Case 0

若待删除节点为根节点的话，直接删除即可，这里不将其算作删除操作的 3 种基本情况中。

Case 1

若待删除节点 N 既有左子节点又有右子节点，则需找到它的前驱或后继节点进行替换（仅替换数据，不改变节点颜色和内部引用关系），则后续操作中只需要将后继节点删除即可。这部分操作与普通 BST 完全相同，在此不再过多赘述。

注：这里选择的前驱或后继节点保证不会是一个既有非 NIL 左子节点又有非 NIL 右子节点的节点。这里拿后继节点进行简单说明：若该节点包含非空左子节点，则该节点并非是 N 节点右子树上键值最小的节点，与后继节点的性质矛盾，因此后继节点的左子节点必须为 NIL。

```
// clang-format off
// Case 1: If the node is strictly internal
//   Step 1. Find the successor S with the smallest key
//           and its parent P on the right subtree.
//   Step 2. Swap the data (key and value) of S and N,
//           S is the node that will be deleted in place of N.
//   Step 3. N = S, goto Case 2, 3
//     |                    |
//     N                    S
//    / \                  / \
//   L  ..   swap(N, S)   L  ..
//       |   ========>       |
//       P                   P
//      / \                 / \
//     S  ..                N  ..
// clang-format on

// Step 1
NodePtr successor = node->right;
NodePtr parent = node;
while (successor->left != nullptr) {
  parent = successor;
  successor = parent->left;
}
// Step 2
swapNode(node, successor);
maintainRelationship(parent);
// Step 3: vvv
```

Case 2

待删除节点为叶子节点，若该节点为红色，直接删除即可，删除后仍能保证红黑树的 4 条性质。若为黑色，删除后性质 4 被打破，需要重新进行维护。

注：由于维护操作不会改变待删除节点的任何结构和数据，因此此处的代码示例中为了实现方便起见选择先进行维护，再解引用相关节点。

```
// clang-format off
// Case 2: Current node is a leaf
//   Step 1. Unlink and remove it.
//   Step 2. If N is BLACK, maintain N;
//           If N is RED, do nothing.
// clang-format on
// The maintain operation won't change the node itself,
//  so we can perform maintain operation before unlink the node.
if (node->isBlack()) {
  maintainAfterRemove(node);
}
if (node->direction() == Direction::LEFT) {
  node->parent->left = nullptr;
} else /* node->direction() == Direction::RIGHT */ {
  node->parent->right = nullptr;
}
```

Case 3

待删除节点 N 有且仅有一个非 NIL 子节点，则子节点 S 一定为红色。因为如果子节点 S 为黑色，则 S 的黑深度和待删除结点的黑深度不同，违反性质 4。由于子节点 S 为红色，则待删除节点 N 为黑色，直接使用子节点 S 替代 N 并将其染黑后即可满足性质 4。

```
// Case 3: Current node has a single left or right child
//   Step 1. Replace N with its child
//   Step 2. Paint N to BLACK
NodePtr parent = node->parent;
NodePtr replacement = (node->left != nullptr ? node->left : node->right);

switch (node->direction()) {
  case Direction::ROOT:
    this->root = replacement;
    break;
  case Direction::LEFT:
    parent->left = replacement;
    break;
  case Direction::RIGHT:
    parent->right = replacement;
    break;
}

if (!node->isRoot()) {
  replacement->parent = parent;
}

node->color = Node::BLACK;
```

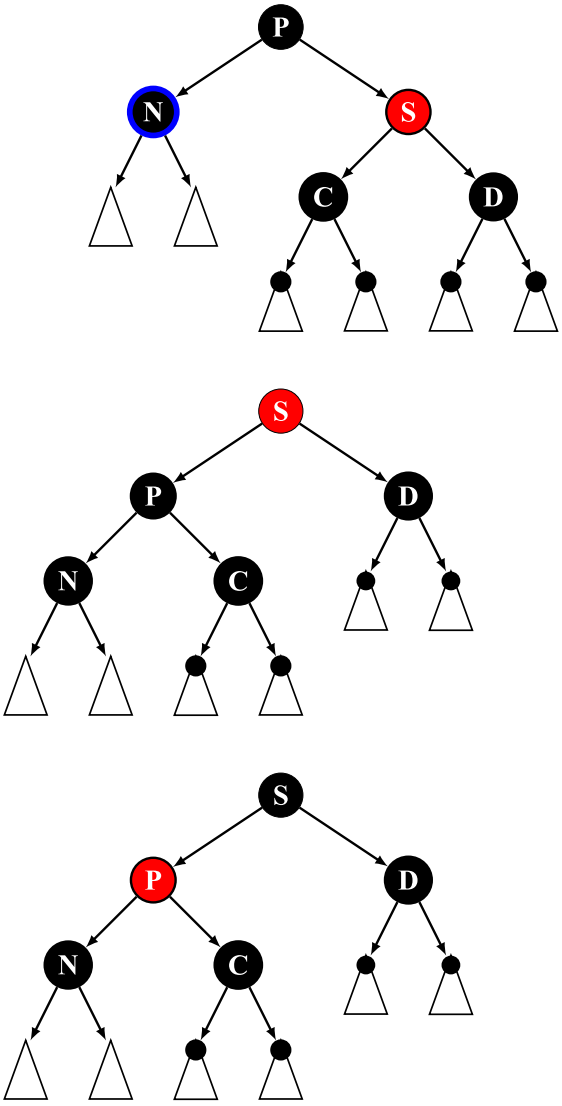
删除后的平衡维护

Case 1

兄弟节点 (sibling node) S 为红色，则父节点 P 和侄节点 (nephew node) C 和 D 必为黑色（否则违反性质 3）。与插入后维护操作的 Case 5 类似，这种情况下无法通过直接的旋转或染色操作使其满足所有性质，因此通过前置操作优先保证部分结构满足性质，再进行后续维护即可。

这种情况的维护需要：

1. 若待删除节点 N 为左子节点，左旋 P; 若为右子节点，右旋 P。
2. 将 S 染黑，P 染红（保证 S 节点的父节点满足性质 4）。
3. 此时只需根据结构，在以 P 节点为根的子树中，继续对节点 N 进行维护即可（无需再考虑旋转染色后的 S 和 D 节点）。



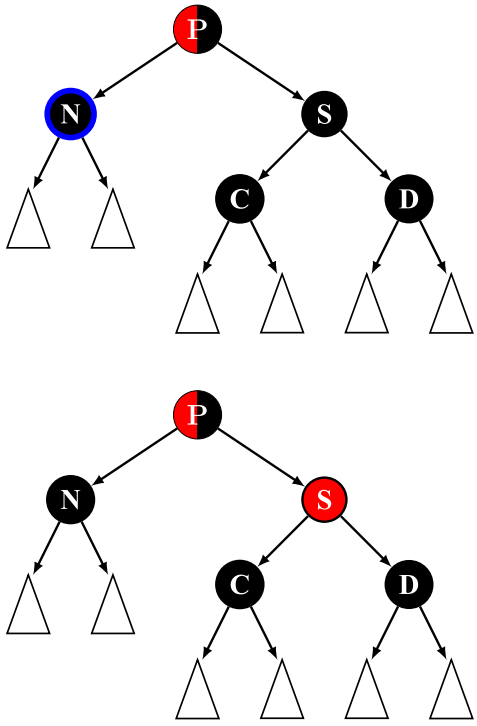

```
// clang-format off

// Case 1: Sibling is RED, parent and nephews must be BLACK
//   Step 1. If N is a left child, left rotate P;
//           If N is a right child, right rotate P.
//   Step 2. Paint S to BLACK, P to RED
//   Step 3. Goto Case 2, 3, 4, 5
//      [P]                   <S>               [S]
//      / \    l-rotate(P)    / \    repaint    / \
```

```
//    [N] <S>  ==========>  [P] [D]  ======>  <P> [D]
//        / \               / \               / \
//      [C] [D]           [N] [C]           [N] [C]
// clang-format on
ConstNodePtr parent = node->parent;
assert(parent != nullptr && parent->isBlack());
assert(sibling->left != nullptr && sibling->left->isBlack());
assert(sibling->right != nullptr && sibling->right->isBlack());
// Step 1
rotateSameDirection(node->parent, direction);
// Step 2
sibling->color = Node::BLACK;
parent->color = Node::RED;
// Update sibling after rotation
sibling = node->sibling();
// Step 3: vvv
```

Case 2

兄弟节点 S 和侄节点 C, D 均为黑色，父节点 P 为红色。此时只需将 S 染红，将 P 染黑即可满足性质 3 和 4。

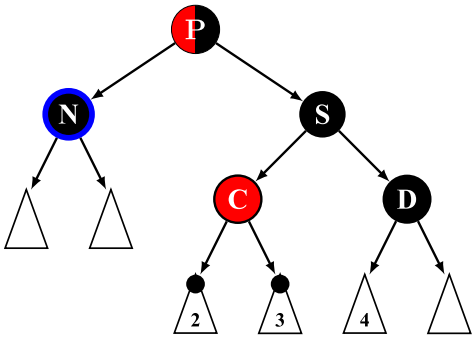


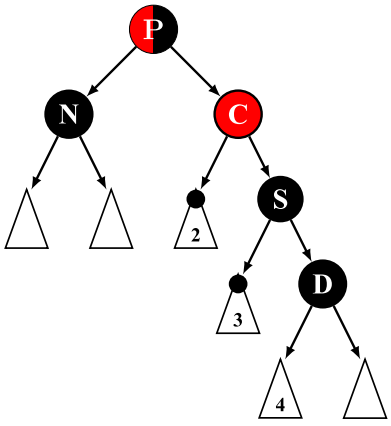
```
// clang-format off
// Case 2: Sibling and nephews are BLACK, parent is RED
//   Swap the color of P and S
//      <P>             [P]
//      / \             / \
//    [N] [S]  ====>  [N] <S>
//        / \             / \
//      [C] [D]         [C] [D]
// clang-format on
sibling->color = Node::RED;
node->parent->color = Node::BLACK;
return;
```

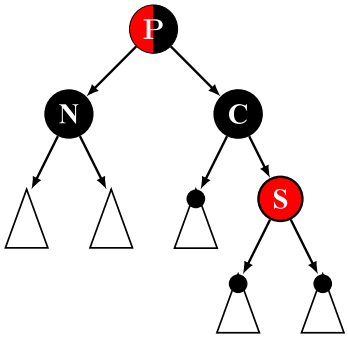
Case 3

兄弟节点 S，父节点 P 以及侄节点 C, D 均为黑色。

此时也无法通过一步操作同时满足性质 3 和 4，因此选择将 S 染红，优先满足局部性质 4 的成立，再递归维护 P 节点根据上部结构进行后续维护。







```
// clang-format off
// Case 3: Sibling, parent and nephews are all black
//   Step 1. Paint S to RED
//   Step 2. Recursively maintain P
//      [P]             [P]
//      / \             / \
//    [N] [S]  ====>  [N] <S>
//        / \             / \
//      [C] [D]         [C] [D]
// clang-format on
sibling->color = Node::RED;
maintainAfterRemove(node->parent);
return;
```

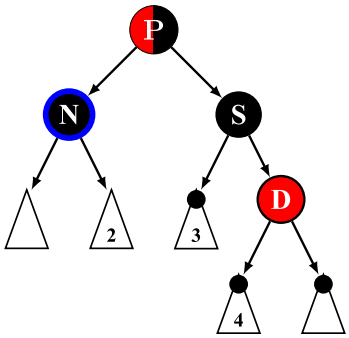
Case 4

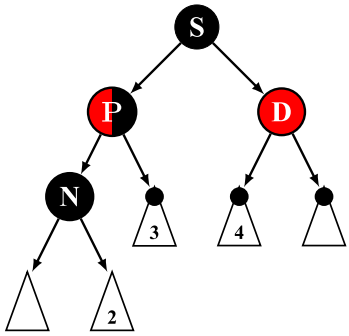
兄弟节点是黑色，且与 N 同向的侄节点 C（由于没有固定中文翻译，下文还是统一将其称作 close nephew）为红色，与 N 反向的侄节点 D（同理，下文称作 distant nephew）为黑色，父节点既可为红色又可为黑色。

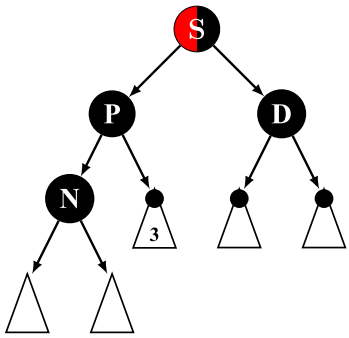
此时同样无法通过一步操作使其满足性质，因此优先选择将其转变为 Case 5 的状态利用后续 Case 5 的维护过程进行修正。

该过程分为三步：

1. 若 N 为左子节点，右旋 P，否则左旋 P。
2. 将节点 S 染红，将节点 C 染黑。
3. 此时已满足 Case 5 的条件，进入 Case 5 完成后续维护。








```
// clang-format off
// Case 4: Sibling is BLACK, close nephew is RED,
//         distant nephew is BLACK
//   Step 1. If N is a left child, right rotate P;
//           If N is a right child, left rotate P.
//   Step 2. Swap the color of close nephew and sibling
//   Step 3. Goto case 5
//                             {P}                {P}
//      {P}                    / \                / \
//      / \    r-rotate(S)  [N] <C>   repaint  [N] [C]
//    [N] [S]  ==========>        \   ======>        \
//        / \                    [S]                <S>
//      <C> [D]                    \                  \
//                                 [D]                [D]
// clang-format on

// Step 1
rotateOppositeDirection(sibling, direction);
// Step 2
closeNephew->color = Node::BLACK;
sibling->color = Node::RED;
// Update sibling and nephews after rotation
sibling = node->sibling();
```

```
closeNephew = direction == Direction::LEFT ? sibling->left : sibling->right;
distantNephew = direction == Direction::LEFT ? sibling->right : sibling->left;
// Step 3: vvv
```

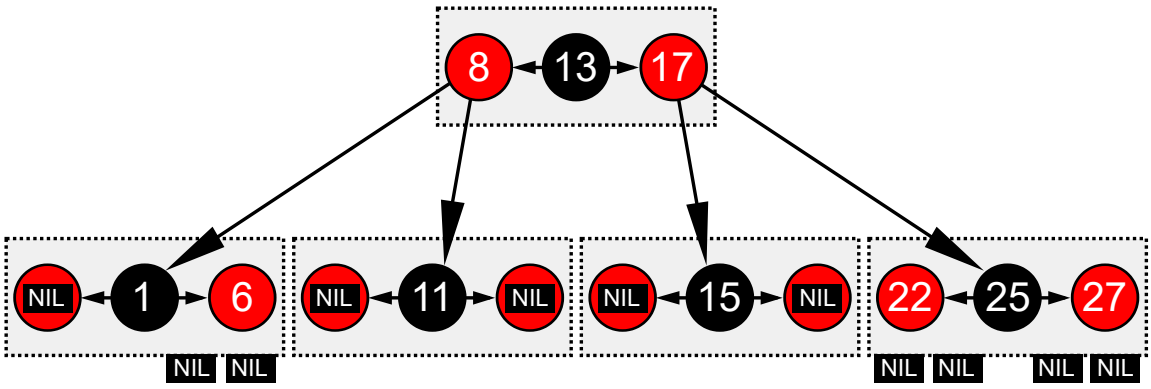
Case 5

兄弟节点是黑色，且 close nephew 节点 C 为黑色，distant nephew 节点 D 为红色，父节点既可为红色又可为黑色。此时性质 4 无法满足，通过旋转操作使得黑色节点 S 变为该子树的根节点再进行染色即可满足性质 4。具体步骤如下：

1. 若 N 为左子节点，左旋 P，反之右旋 P。
2. 交换父节点 P 和兄弟节点 S 的颜色，此时性质 3 可能被打破。
3. 将 distant nephew 节点 D 染黑，同时保证了性质 3 和 4。

```
// clang-format off
// Case 5: Sibling is BLACK, close nephew is BLACK,
//         distant nephew is RED
//   Step 1. If N is a left child, left rotate P;
//           If N is a right child, right rotate P.
//   Step 2. Swap the color of parent and sibling.
//   Step 3. Paint distant nephew D to BLACK.
//      {P}                   [S]               {S}
//      / \    l-rotate(P)    / \    repaint    / \
//    [N] [S]  ==========>  {P} <D>  =======>  [P] [D]
//        / \               / \               / \
//      [C] <D>           [N] [C]           [N] [C]
// clang-format on
assert(closeNephew == nullptr || closeNephew->isBlack());
assert(distantNephew->isRed());
// Step 1
rotateSameDirection(node->parent, direction);
// Step 2
sibling->color = node->parent->color;
node->parent->color = Node::BLACK;
// Step 3
distantNephew->color = Node::BLACK;
return;
```

红黑树与 4 阶 B 树（2-3-4 树）的关系



红黑树是由德国计算机科学家 Rudolf Bayer 在 1972 年从 B 树上改进过来的，红黑树在当时被称作 "symmetric binary B-tree"，因此与 B 树有众多相似之处。比如红黑树与 4 阶 B 树每个簇（对于红黑树来说一个簇是一个非 NIL 黑色节点和它的两个子节点，对 B 树来说一个簇就是一个节点）的最大容量为 3 且最小填充量均为 。因此我们甚至可以说红黑树与 4 阶 B 树（2-3-4 树）在结构上是等价的。

对这方面内容感兴趣的可以观看 从 2-3-4 树的角度学习理解红黑树（视频） 进行学习。

虽然二者在结构上是等价的，但这并不意味这二者可以互相取代或者在所有情况下都可以互换使用。最显然的例子就是数据库的索引，由于 B 树不存在旋转操作，因此其所有节点的存储位置都是可以被确定的，这种结构对于不区分堆栈的磁盘来说显然比红黑树动态分配节点存储空间要更加合适。另外一点就是由于 B 树/B+ 树内储存的数据都是连续的，对于有着大量连续查询需求的数据库来说更加友好。而对于小数据量随机插入/查询的需求，由于 B 树的每个节点都存储了若干条记录，因此发生 cache miss 时就需要将整个节点的所有数据读入缓存中，在这些情况下 BST（红黑树，AVL，Splay 等）则反而会优与 B 树/B+ 树。对这方面内容感兴趣的读者可以去阅读一下 为什么 rust 中的 Map 使用的是 B 树而不是像其他主流语言一样使用红黑树。

红黑树在实际工程项目中的使用

由于红黑树是目前主流工业界综合效率最高的内存型平衡树，其在实际的工程项目中有着广泛的使用，这里列举几个实际的使用案例并给出相应的源码链接，以便读者进行对比学习。

Linux

源码：

- `linux/lib/rbtree.c`

Linux 中的红黑树所有操作均使用循环迭代进行实现，保证效率的同时又增加了大量的注释来保证代码可读性，十分建议读者阅读学习。Linux 内核中的红黑树使用非常广泛，这里仅列举几个经典案例。

CFS 非实时任务调度

Linux 的稳定内核版本在 2.6.24 之后，使用了新的调度程序 CFS，所有非实时可运行进程都以虚拟运行时间为键值用一棵红黑树进行维护，以完成更公平高效地调度所有任务。CFS 弃用 active/expired 数组和动态计算优先级，不再跟踪任务的睡眠时间和区别是否交互任务，而是在调度中采用基于时间计算键值的红黑树来选取下一个任务，根据所有任务占用 CPU 时间的状态来确定调度任务优先级。

epoll

epoll 全称 event poll，是 Linux 内核实现 IO 多路复用 (IO multiplexing) 的一个实现，是原先 poll/select 的改进版。Linux 中 epoll 的实现选择使用红黑树来储存文件描述符。

Nginx

源码：

- `nginx/src/core/ngx_rbtree.h`
- `nginx/src/core/ngx_rbtree.c`

nginx 中的用户态定时器是通过红黑树实现的。在 nginx 中，所有 timer 节点都由一棵红黑树进行维护，在 worker 进程的每一次循环中都会调用 `ngx_process_events_and_timers` 函数，在该函数中就会调用处理定时器的函数 `ngx_event_expire_timers`，每次该函数都不断的从红黑树中取出时间值最小的，查看他们是否已经超时，然后执行他们的函数，直到取出的节点的时间没有超时为止。

关于 nginx 中红黑树的源码分析公开资源很多，读者可以自行查找学习。

STL

源码：

- GNU libstdc++
  - `libstdc++-v3/include/bits/stl_tree.h`
  - `libstdc++-v3/src/c++98/tree.cc`
- LLVM libcxx
  - `libcxx/include/__tree`
- Microsoft STL
  - `stl/inc/xtree`

大多数 STL 中的 `std::map` 和 `std::set` 的内部数据结构就是一棵红黑树（例如上面提到的这些）。不过值得注意的是，这些红黑树（包括可能有读者用过的 `std::_Rb_tree`）都不是 C++ 标准，虽然部分竞赛（例如 NOIP）并未明令禁止这类数据结构，但还是应当注意这类标准库中的非标准实现不应该在工程项目中直接使用。

由于 STL 的特殊性，其中大多数实现的代码可读性都不高，因此并不建议读者使用 STL 学习红黑树。

OpenJDK

源码：

- `java.util.TreeMap`
- `java.util.TreeSet`
- `java.util.HashMap`

JDK 中的 `TreeMap` 和 `TreeSet` 都是使用红黑树作为底层数据结构的。同时在 JDK 1.8 之后 `HashMap` 内部哈希表中每个表项的链表长度超过 8 时也会自动转变为红黑树以提升查找效率。

笔者认为，JDK 中的红黑树实现是主流红黑树实现中可读性最高的，本文提供的参考代码很大程度上借鉴了 JDK 中 `TreeMap` 的实现，因此也建议读者阅读学习 JDK 中 `TreeMap` 的实现。

参考代码

下面的代码是用红黑树实现的 `Map`，即有序不可重映射：

```
/**
 * @file RBTreeMap.hpp
 * @brief An RBTree-based map implementation
 * @details The map is sorted according to the natural ordering of its
 *  keys or by a {@code Compare} function provided; This implementation
 *  provides guaranteed log(n) time cost for the contains, get, insert
 *  and remove operations.
 * @author [r.ivance](https://github.com/RIvance)
 */

#ifndef RBTREE_MAP_HPP
#define RBTREE_MAP_HPP

#include <cassert>
#include <cstddef>
#include <cstdint>
#include <functional>
#include <memory>
```

```cpp
#include <stack>
#include <utility>
#include <vector>

/**
 * An RBTree-based map implementation
 * https://en.wikipedia.org/wiki/Red-black_tree
 *
 * A red-black tree (RBTree) is a kind of self-balancing binary search tree.
 * Each node stores an extra field representing "color" (RED or BLACK), used
 * to ensure that the tree remains balanced during insertions and deletions.
 *
 * In addition to the requirements imposed on a binary search tree the following
 * must be satisfied by a red-black tree:
 *
 *  1. Every node is either RED or BLACK.
 *  2. All NIL nodes (`nullptr` in this implementation) are considered BLACK.
 *  3. A RED node does not have a RED child.
 *  4. Every path from a given node to any of its descendant NIL nodes goes
 * through the same number of BLACK nodes.
 *
 * @tparam Key the type of keys maintained by this map
 * @tparam Value the type of mapped values
 * @tparam Compare the compare function
 */
template <typename Key, typename Value, typename Compare = std::less<Key> >
class RBTreeMap {
 private:
  using USize = size_t;

  Compare compare = Compare();

 public:
  struct Entry {
    Key key;
    Value value;

    bool operator==(const Entry &rhs) const noexcept {
      return this->key == rhs.key && this->value == rhs.value;
    }

    bool operator!=(const Entry &rhs) const noexcept {
      return this->key != rhs.key || this->value != rhs.value;
    }
  };

 private:
  struct Node {
    using Ptr = std::shared_ptr<Node>;
    using Provider = const std::function<Ptr(void)> &;
    using Consumer = const std::function<void(const Ptr &)> &;

    enum { RED, BLACK } color = RED;

    enum Direction { LEFT = -1, ROOT = 0, RIGHT = 1 };
```

```
    Key key;
    Value value{};

    Ptr parent = nullptr;
    Ptr left = nullptr;
    Ptr right = nullptr;

    explicit Node(Key k) : key(std::move(k)) {}

    explicit Node(Key k, Value v) : key(std::move(k)), value(std::move(v)) {}

    ~Node() = default;

    inline bool isLeaf() const noexcept {
      return this->left == nullptr && this->right == nullptr;
    }

    inline bool isRoot() const noexcept { return this->parent == nullptr; }

    inline bool isRed() const noexcept { return this->color == RED; }

    inline bool isBlack() const noexcept { return this->color == BLACK; }

    inline Direction direction() const noexcept {
      if (this->parent != nullptr) {
        if (this == this->parent->left.get()) {
          return Direction::LEFT;
        } else {
          return Direction::RIGHT;
        }
      } else {
        return Direction::ROOT;
      }
    }

    inline Ptr &sibling() const noexcept {
      assert(!this->isRoot());
      if (this->direction() == LEFT) {
        return this->parent->right;
      } else {
        return this->parent->left;
      }
    }

    inline bool hasSibling() const noexcept {
      return !this->isRoot() && this->sibling() != nullptr;
    }

    inline Ptr &uncle() const noexcept {
      assert(this->parent != nullptr);
      return parent->sibling();
    }

    inline bool hasUncle() const noexcept {
      return !this->isRoot() && this->parent->hasSibling();
    }
```

```
    inline Ptr &grandParent() const noexcept {
      assert(this->parent != nullptr);
      return this->parent->parent;
    }

    inline bool hasGrandParent() const noexcept {
      return !this->isRoot() && this->parent->parent != nullptr;
    }

    inline void release() noexcept {
      // avoid memory leak caused by circular reference
      this->parent = nullptr;
      if (this->left != nullptr) {
        this->left->release();
      }
      if (this->right != nullptr) {
        this->right->release();
      }
    }

    inline Entry entry() const { return Entry{key, value}; }

    static Ptr from(const Key &k) { return std::make_shared<Node>(Node(k)); }

    static Ptr from(const Key &k, const Value &v) {
      return std::make_shared<Node>(Node(k, v));
    }
  };

  using NodePtr = typename Node::Ptr;
  using ConstNodePtr = const NodePtr &;
  using Direction = typename Node::Direction;
  using NodeProvider = typename Node::Provider;
  using NodeConsumer = typename Node::Consumer;

  NodePtr root = nullptr;
  USize count = 0;

  using K = const Key &;
  using V = const Value &;

 public:
  using EntryList = std::vector<Entry>;
  using KeyValueConsumer = const std::function<void(K, V)> &;
  using MutKeyValueConsumer = const std::function<void(K, Value &)> &;
  using KeyValueFilter = const std::function<bool(K, V)> &;

  class NoSuchMappingException : protected std::exception {
   private:
    const char *message;

   public:
    explicit NoSuchMappingException(const char *msg) : message(msg) {}

    const char *what() const noexcept override { return message; }
```

```
};

RBTreeMap() noexcept = default;

~RBTreeMap() noexcept {
  // Unlinking circular references to avoid memory leak
  this->clear();
}

/**
 * Returns the number of entries in this map.
 * @return size_t
 */
inline USize size() const noexcept { return this->count; }

/**
 * Returns true if this collection contains no elements.
 * @return bool
 */
inline bool empty() const noexcept { return this->count == 0; }

/**
 * Removes all of the elements from this map.
 */
void clear() noexcept {
  // Unlinking circular references to avoid memory leak
  if (this->root != nullptr) {
    this->root->release();
    this->root = nullptr;
  }
  this->count = 0;
}

/**
 * Returns the value to which the specified key is mapped; If this map
 * contains no mapping for the key, a {@code NoSuchMappingException} will
 * be thrown.
 * @param key
 * @return RBTreeMap<Key, Value>::Value
 * @throws NoSuchMappingException
 */
Value get(K key) const {
  if (this->root == nullptr) {
    throw NoSuchMappingException("Invalid key");
  } else {
    NodePtr node = this->getNode(this->root, key);
    if (node != nullptr) {
      return node->value;
    } else {
      throw NoSuchMappingException("Invalid key");
    }
  }
}

/**
 * Returns the value to which the specified key is mapped; If this map
```

```
   * contains no mapping for the key, a new mapping with a default value
   * will be inserted.
   * @param key
   * @return RBTreeMap<Key, Value>::Value &
   */
  Value &getOrDefault(K key) {
    if (this->root == nullptr) {
      this->root = Node::from(key);
      this->root->color = Node::BLACK;
      this->count += 1;
      return this->root->value;
    } else {
      return this
          ->getNodeOrProvide(this->root, key,
                             [&key]() { return Node::from(key); })
          ->value;
    }
  }

  /**
   * Returns true if this map contains a mapping for the specified key.
   * @param key
   * @return bool
   */
  bool contains(K key) const {
    return this->getNode(this->root, key) != nullptr;
  }

  /**
   * Associates the specified value with the specified key in this map.
   * @param key
   * @param value
   */
  void insert(K key, V value) {
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
      this->root->color = Node::BLACK;
      this->count += 1;
    } else {
      this->insert(this->root, key, value);
    }
  }

  /**
   * If the specified key is not already associated with a value, associates
   * it with the given value and returns true, else returns false.
   * @param key
   * @param value
   * @return bool
   */
  bool insertIfAbsent(K key, V value) {
    USize sizeBeforeInsertion = this->size();
    if (this->root == nullptr) {
      this->root = Node::from(key, value);
      this->root->color = Node::BLACK;
      this->count += 1;
```

```
  } else {
    this->insert(this->root, key, value, false);
  }
  return this->size() > sizeBeforeInsertion;
}

/**
 * If the specified key is not already associated with a value, associates
 * it with the given value and returns the value, else returns the associated
 * value.
 * @param key
 * @param value
 * @return RBTreeMap<Key, Value>::Value &
 */
Value &getOrInsert(K key, V value) {
  if (this->root == nullptr) {
    this->root = Node::from(key, value);
    this->root->color = Node::BLACK;
    this->count += 1;
    return root->value;
  } else {
    NodePtr node = getNodeOrProvide(this->root, key,
                                    [&]() { return Node::from(key, value); });
    return node->value;
  }
}

Value operator[](K key) const { return this->get(key); }

Value &operator[](K key) { return this->getOrDefault(key); }

/**
 * Removes the mapping for a key from this map if it is present;
 * Returns true if the mapping is present else returns false
 * @param key the key of the mapping
 * @return bool
 */
bool remove(K key) {
  if (this->root == nullptr) {
    return false;
  } else {
    return this->remove(this->root, key, [](ConstNodePtr) {});
  }
}

/**
 * Removes the mapping for a key from this map if it is present and returns
 * the value which is mapped to the key; If this map contains no mapping for
 * the key, a {@code NoSuchMappingException} will be thrown.
 * @param key
 * @return RBTreeMap<Key, Value>::Value
 * @throws NoSuchMappingException
 */
Value getAndRemove(K key) {
  Value result;
  NodeConsumer action = [&](ConstNodePtr node) { result = node->value; };
```

```
  if (root == nullptr) {
    throw NoSuchMappingException("Invalid key");
  } else {
    if (remove(this->root, key, action)) {
      return result;
    } else {
      throw NoSuchMappingException("Invalid key");
    }
  }
}

/**
 * Gets the entry corresponding to the specified key; if no such entry
 * exists, returns the entry for the least key greater than the specified
 * key; if no such entry exists (i.e., the greatest key in the Tree is less
 * than the specified key), a {@code NoSuchMappingException} will be thrown.
 * @param key
 * @return RBTreeMap<Key, Value>::Entry
 * @throws NoSuchMappingException
 */
Entry getCeilingEntry(K key) const {
  if (this->root == nullptr) {
    throw NoSuchMappingException("No ceiling entry in this map");
  }

  NodePtr node = this->root;

  while (node != nullptr) {
    if (key == node->key) {
      return node->entry();
    }

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left != nullptr) {
        node = node->left;
      } else {
        return node->entry();
      }
    } else {
      /* key > node->key */
      if (node->right != nullptr) {
        node = node->right;
      } else {
        while (node->direction() == Direction::RIGHT) {
          if (node != nullptr) {
            node = node->parent;
          } else {
            throw NoSuchMappingException(
                "No ceiling entry exists in this map");
          }
        }
        if (node->parent == nullptr) {
          throw NoSuchMappingException("No ceiling entry exists in this map");
        }
```

```
        return node->parent->entry();
      }
    }
  }

  throw NoSuchMappingException("No ceiling entry in this map");
}

/**
 * Gets the entry corresponding to the specified key; if no such entry exists,
 * returns the entry for the greatest key less than the specified key;
 * if no such entry exists, a {@code NoSuchMappingException} will be thrown.
 * @param key
 * @return RBTreeMap<Key, Value>::Entry
 * @throws NoSuchMappingException
 */
Entry getFloorEntry(K key) const {
  if (this->root == nullptr) {
    throw NoSuchMappingException("No floor entry exists in this map");
  }

  NodePtr node = this->root;

  while (node != nullptr) {
    if (key == node->key) {
      return node->entry();
    }

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left != nullptr) {
        node = node->left;
      } else {
        while (node->direction() == Direction::LEFT) {
          if (node != nullptr) {
            node = node->parent;
          } else {
            throw NoSuchMappingException("No floor entry exists in this map");
          }
        }
        if (node->parent == nullptr) {
          throw NoSuchMappingException("No floor entry exists in this map");
        }
        return node->parent->entry();
      }
    } else {
      /* key > node->key */
      if (node->right != nullptr) {
        node = node->right;
      } else {
        return node->entry();
      }
    }
  }

  throw NoSuchMappingException("No floor entry exists in this map");
```

```
  }

  /**
   * Gets the entry for the least key greater than the specified
   * key; if no such entry exists, returns the entry for the least
   * key greater than the specified key; if no such entry exists,
   * a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return RBTreeMap<Key, Value>::Entry
   * @throws NoSuchMappingException
   */
  Entry getHigherEntry(K key) {
    if (this->root == nullptr) {
      throw NoSuchMappingException("No higher entry exists in this map");
    }

    NodePtr node = this->root;

    while (node != nullptr) {
      if (compare(key, node->key)) {
        /* key < node->key */
        if (node->left != nullptr) {
          node = node->left;
        } else {
          return node->entry();
        }
      } else {
        /* key >= node->key */
        if (node->right != nullptr) {
          node = node->right;
        } else {
          while (node->direction() == Direction::RIGHT) {
            if (node != nullptr) {
              node = node->parent;
            } else {
              throw NoSuchMappingException(
                  "No higher entry exists in this map");
            }
          }
          if (node->parent == nullptr) {
            throw NoSuchMappingException("No higher entry exists in this map");
          }
          return node->parent->entry();
        }
      }
    }

    throw NoSuchMappingException("No higher entry exists in this map");
  }

  /**
   * Returns the entry for the greatest key less than the specified key; if
   * no such entry exists (i.e., the least key in the Tree is greater than
   * the specified key), a {@code NoSuchMappingException} will be thrown.
   * @param key
   * @return RBTreeMap<Key, Value>::Entry
```

```
     * @throws NoSuchMappingException
     */
    Entry getLowerEntry(K key) const {
      if (this->root == nullptr) {
        throw NoSuchMappingException("No lower entry exists in this map");
      }

      NodePtr node = this->root;

      while (node != nullptr) {
        if (compare(key, node->key) || key == node->key) {
          /* key <= node->key */
          if (node->left != nullptr) {
            node = node->left;
          } else {
            while (node->direction() == Direction::LEFT) {
              if (node != nullptr) {
                node = node->parent;
              } else {
                throw NoSuchMappingException("No lower entry exists in this map");
              }
            }
            if (node->parent == nullptr) {
              throw NoSuchMappingException("No lower entry exists in this map");
            }
            return node->parent->entry();
          }
        } else {
          /* key > node->key */
          if (node->right != nullptr) {
            node = node->right;
          } else {
            return node->entry();
          }
        }
      }

      throw NoSuchMappingException("No lower entry exists in this map");
    }

    /**
     * Remove all entries that satisfy the filter condition.
     * @param filter
     */
    void removeAll(KeyValueFilter filter) {
      std::vector<Key> keys;
      this->inorderTraversal([&](ConstNodePtr node) {
        if (filter(node->key, node->value)) {
          keys.push_back(node->key);
        }
      });
      for (const Key &key : keys) {
        this->remove(key);
      }
    }
```

```
  /**
   * Performs the given action for each key and value entry in this map.
   * The value is immutable for the action.
   * @param action
   */
  void forEach(KeyValueConsumer action) const {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value); });
  }

  /**
   * Performs the given action for each key and value entry in this map.
   * The value is mutable for the action.
   * @param action
   */
  void forEachMut(MutKeyValueConsumer action) {
    this->inorderTraversal(
        [&](ConstNodePtr node) { action(node->key, node->value); });
  }

  /**
   * Returns a list containing all of the entries in this map.
   * @return RBTreeMap<Key, Value>::EntryList
   */
  EntryList toEntryList() const {
    EntryList entryList;
    this->inorderTraversal(
        [&](ConstNodePtr node) { entryList.push_back(node->entry()); });
    return entryList;
  }

 private:
  static void maintainRelationship(ConstNodePtr node) {
    if (node->left != nullptr) {
      node->left->parent = node;
    }
    if (node->right != nullptr) {
      node->right->parent = node;
    }
  }

  static void swapNode(NodePtr &lhs, NodePtr &rhs) {
    std::swap(lhs->key, rhs->key);
    std::swap(lhs->value, rhs->value);
    std::swap(lhs, rhs);
  }

  void rotateLeft(ConstNodePtr node) {
    // clang-format off
    //     |                       |
    //     N                       S
    //    / \     l-rotate(N)     / \
    //   L   S    ==========>    N   R
    //      / \                 / \
    //     M   R               L   M
    assert(node != nullptr && node->right != nullptr);
```

```
    // clang-format on
    NodePtr parent = node->parent;
    Direction direction = node->direction();

    NodePtr successor = node->right;
    node->right = successor->left;
    successor->left = node;

    maintainRelationship(node);
    maintainRelationship(successor);

    switch (direction) {
      case Direction::ROOT:
        this->root = successor;
        break;
      case Direction::LEFT:
        parent->left = successor;
        break;
      case Direction::RIGHT:
        parent->right = successor;
        break;
    }

    successor->parent = parent;
  }

  void rotateRight(ConstNodePtr node) {
    // clang-format off
    //       |                   |
    //       N                   S
    //      / \   r-rotate(N)    / \
    //     S   R  ==========>   L   N
    //    / \                     / \
    //   L   M                   M   R
    assert(node != nullptr && node->left != nullptr);
    // clang-format on

    NodePtr parent = node->parent;
    Direction direction = node->direction();

    NodePtr successor = node->left;
    node->left = successor->right;
    successor->right = node;

    maintainRelationship(node);
    maintainRelationship(successor);

    switch (direction) {
      case Direction::ROOT:
        this->root = successor;
        break;
      case Direction::LEFT:
        parent->left = successor;
        break;
      case Direction::RIGHT:
        parent->right = successor;
```

```
      break;
    }

    successor->parent = parent;
  }

  inline void rotateSameDirection(ConstNodePtr node, Direction direction) {
    assert(direction != Direction::ROOT);
    if (direction == Direction::LEFT) {
      rotateLeft(node);
    } else {
      rotateRight(node);
    }
  }

  inline void rotateOppositeDirection(ConstNodePtr node, Direction direction) {
    assert(direction != Direction::ROOT);
    if (direction == Direction::LEFT) {
      rotateRight(node);
    } else {
      rotateLeft(node);
    }
  }

  void maintainAfterInsert(NodePtr node) {
    assert(node != nullptr);

    if (node->isRoot()) {
      // Case 1: Current node is root (RED)
      //  No need to fix.
      assert(node->isRed());
      return;
    }

    if (node->parent->isBlack()) {
      // Case 2: Parent is BLACK
      //  No need to fix.
      return;
    }

    if (node->parent->isRoot()) {
      // clang-format off
      // Case 3: Parent is root and is RED
      //   Paint parent to BLACK.
      //    <P>         [P]
      //     |   ====>   |
      //    <N>         <N>
      //   p.s.
      //    `<X>` is a RED node;
      //    `[X]` is a BLACK node (or NIL);
      //    `{X}` is either a RED node or a BLACK node;
      // clang-format on
      assert(node->parent->isRed());
      node->parent->color = Node::BLACK;
      return;
    }
```

```
  if (node->hasUncle() && node->uncle()->isRed()) {
    // clang-format off
    // Case 4: Both parent and uncle are RED
    //   Paint parent and uncle to BLACK;
    //   Paint grandparent to RED.
    //        [G]             <G>
    //        / \             / \
    //      <P> <U>  ====>  [P] [U]
    //      /               /
    //    <N>             <N>
    // clang-format on
    assert(node->parent->isRed());
    node->parent->color = Node::BLACK;
    node->uncle()->color = Node::BLACK;
    node->grandParent()->color = Node::RED;
    maintainAfterInsert(node->grandParent());
    return;
  }

  if (!node->hasUncle() || node->uncle()->isBlack()) {
    // Case 5 & 6: Parent is RED and Uncle is BLACK
    //   p.s. NIL nodes are also considered BLACK
    assert(!node->isRoot());

    if (node->direction() != node->parent->direction()) {
      // clang-format off
      // Case 5: Current node is the opposite direction as parent
      //   Step 1. If node is a LEFT child, perform l-rotate to parent;
      //           If node is a RIGHT child, perform r-rotate to parent.
      //   Step 2. Goto Case 6.
      //      [G]                 [G]
      //      / \    rotate(P)    / \
      //    <P> [U]  ========>  <N> [U]
      //      \                 /
      //      <N>             <P>
      // clang-format on

      // Step 1: Rotation
      NodePtr parent = node->parent;
      if (node->direction() == Direction::LEFT) {
        rotateRight(node->parent);
      } else /* node->direction() == Direction::RIGHT */ {
        rotateLeft(node->parent);
      }
      node = parent;
      // Step 2: vvv
    }

    // clang-format off
    // Case 6: Current node is the same direction as parent
    //   Step 1. If node is a LEFT child, perform r-rotate to grandparent;
    //           If node is a RIGHT child, perform l-rotate to grandparent.
    //   Step 2. Paint parent (before rotate) to BLACK;
    //           Paint grandparent (before rotate) to RED.
    //        [G]                 <P>               [P]
```

```
      //        / \    rotate(G)    / \    repaint    / \
      //      <P> [U]  ========>  <N> [G]  ======>  <N> <G>
      //      /                         \                \
      //    <N>                         [U]              [U]
      // clang-format on

      assert(node->grandParent() != nullptr);

      // Step 1
      if (node->parent->direction() == Direction::LEFT) {
        rotateRight(node->grandParent());
      } else {
        rotateLeft(node->grandParent());
      }

      // Step 2
      node->parent->color = Node::BLACK;
      node->sibling()->color = Node::RED;

      return;
    }
  }

  NodePtr getNodeOrProvide(NodePtr &node, K key, NodeProvider provide) {
    assert(node != nullptr);

    if (key == node->key) {
      return node;
    }

    assert(key != node->key);

    NodePtr result;

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left == nullptr) {
        result = node->left = provide();
        node->left->parent = node;
        maintainAfterInsert(node->left);
        this->count += 1;
      } else {
        result = getNodeOrProvide(node->left, key, provide);
      }
    } else {
      /* key > node->key */
      if (node->right == nullptr) {
        result = node->right = provide();
        node->right->parent = node;
        maintainAfterInsert(node->right);
        this->count += 1;
      } else {
        result = getNodeOrProvide(node->right, key, provide);
      }
    }
```

```
    return result;
  }

  NodePtr getNode(ConstNodePtr node, K key) const {
    assert(node != nullptr);

    if (key == node->key) {
      return node;
    }

    if (compare(key, node->key)) {
      /* key < node->key */
      return node->left == nullptr ? nullptr : getNode(node->left, key);
    } else {
      /* key > node->key */
      return node->right == nullptr ? nullptr : getNode(node->right, key);
    }
  }

  void insert(NodePtr &node, K key, V value, bool replace = true) {
    assert(node != nullptr);

    if (key == node->key) {
      if (replace) {
        node->value = value;
      }
      return;
    }

    assert(key != node->key);

    if (compare(key, node->key)) {
      /* key < node->key */
      if (node->left == nullptr) {
        node->left = Node::from(key, value);
        node->left->parent = node;
        maintainAfterInsert(node->left);
        this->count += 1;
      } else {
        insert(node->left, key, value, replace);
      }
    } else {
      /* key > node->key */
      if (node->right == nullptr) {
        node->right = Node::from(key, value);
        node->right->parent = node;
        maintainAfterInsert(node->right);
        this->count += 1;
      } else {
        insert(node->right, key, value, replace);
      }
    }
  }

  void maintainAfterRemove(ConstNodePtr node) {
    if (node->isRoot()) {
```

```
    return;
  }

  assert(node->isBlack() && node->hasSibling());

  Direction direction = node->direction();

  NodePtr sibling = node->sibling();
  if (sibling->isRed()) {
    // clang-format off
    // Case 1: Sibling is RED, parent and nephews must be BLACK
    //   Step 1. If N is a left child, left rotate P;
    //           If N is a right child, right rotate P.
    //   Step 2. Paint S to BLACK, P to RED
    //   Step 3. Goto Case 2, 3, 4, 5
    //      [P]                   <S>               [S]
    //      / \    l-rotate(P)    / \    repaint    / \
    //    [N] <S>  ==========>  [P] [D]  ======>  <P> [D]
    //        / \               / \               / \
    //      [C] [D]           [N] [C]           [N] [C]
    // clang-format on
    ConstNodePtr parent = node->parent;
    assert(parent != nullptr && parent->isBlack());
    assert(sibling->left != nullptr && sibling->left->isBlack());
    assert(sibling->right != nullptr && sibling->right->isBlack());
    // Step 1
    rotateSameDirection(node->parent, direction);
    // Step 2
    sibling->color = Node::BLACK;
    parent->color = Node::RED;
    // Update sibling after rotation
    sibling = node->sibling();
    // Step 3: vvv
  }

  NodePtr closeNephew =
      direction == Direction::LEFT ? sibling->left : sibling->right;
  NodePtr distantNephew =
      direction == Direction::LEFT ? sibling->right : sibling->left;

  bool closeNephewIsBlack = closeNephew == nullptr || closeNephew->isBlack();
  bool distantNephewIsBlack =
      distantNephew == nullptr || distantNephew->isBlack();

  assert(sibling->isBlack());

  if (closeNephewIsBlack && distantNephewIsBlack) {
    if (node->parent->isRed()) {
      // clang-format off
      // Case 2: Sibling and nephews are BLACK, parent is RED
      //   Swap the color of P and S
      //      <P>             [P]
      //      / \             / \
      //    [N] [S]  ====>  [N] <S>
      //        / \             / \
      //      [C] [D]         [C] [D]
```

```
        // clang-format on
        sibling->color = Node::RED;
        node->parent->color = Node::BLACK;
        return;
      } else {
        // clang-format off
        // Case 3: Sibling, parent and nephews are all black
        //   Step 1. Paint S to RED
        //   Step 2. Recursively maintain P
        //      [P]             [P]
        //      / \             / \
        //    [N] [S]  ====>  [N] <S>
        //        / \             / \
        //      [C] [D]         [C] [D]
        // clang-format on
        sibling->color = Node::RED;
        maintainAfterRemove(node->parent);
        return;
      }
    } else {
      if (closeNephew != nullptr && closeNephew->isRed()) {
        // clang-format off
        // Case 4: Sibling is BLACK, close nephew is RED,
        //         distant nephew is BLACK
        //   Step 1. If N is a left child, right rotate P;
        //           If N is a right child, left rotate P.
        //   Step 2. Swap the color of close nephew and sibling
        //   Step 3. Goto case 5
        //                              {P}               {P}
        //      {P}                     / \               / \
        //      / \    r-rotate(S)  [N] <C>   repaint  [N] [C]
        //    [N] [S]  ==========>       \   ======>       \
        //        / \                   [S]               <S>
        //      <C> [D]                   \                 \
        //                                [D]               [D]
        // clang-format on

        // Step 1
        rotateOppositeDirection(sibling, direction);
        // Step 2
        closeNephew->color = Node::BLACK;
        sibling->color = Node::RED;
        // Update sibling and nephews after rotation
        sibling = node->sibling();
        closeNephew =
            direction == Direction::LEFT ? sibling->left : sibling->right;
        distantNephew =
            direction == Direction::LEFT ? sibling->right : sibling->left;
        // Step 3: vvv
      }

      // clang-format off
      // Case 5: Sibling is BLACK, close nephew is BLACK,
      //         distant nephew is RED
      //      {P}                   [S]
      //      / \    l-rotate(P)    / \
```

```
      //    [N] [S]  ==========>  {P} <D>
      //        / \                / \
      //      [C] <D>            [N] [C]
      // clang-format on
      assert(closeNephew == nullptr || closeNephew->isBlack());
      assert(distantNephew->isRed());
      // Step 1
      rotateSameDirection(node->parent, direction);
      // Step 2
      sibling->color = node->parent->color;
      node->parent->color = Node::BLACK;
      if (distantNephew != nullptr) {
        distantNephew->color = Node::BLACK;
      }
      return;
    }
  }

  bool remove(NodePtr node, K key, NodeConsumer action) {
    assert(node != nullptr);

    if (key != node->key) {
      if (compare(key, node->key)) {
        /* key < node->key */
        NodePtr &left = node->left;
        if (left != nullptr && remove(left, key, action)) {
          maintainRelationship(node);
          return true;
        } else {
          return false;
        }
      } else {
        /* key > node->key */
        NodePtr &right = node->right;
        if (right != nullptr && remove(right, key, action)) {
          maintainRelationship(node);
          return true;
        } else {
          return false;
        }
      }
    }

    assert(key == node->key);
    action(node);

    if (this->size() == 1) {
      // Current node is the only node of the tree
      this->clear();
      return true;
    }

    if (node->left != nullptr && node->right != nullptr) {
      // clang-format off
      // Case 1: If the node is strictly internal
      //   Step 1. Find the successor S with the smallest key
```

```
      //           and its parent P on the right subtree.
      //   Step 2. Swap the data (key and value) of S and N,
      //           S is the node that will be deleted in place of N.
      //   Step 3. N = S, goto Case 2, 3
      //     |                    |
      //     N                    S
      //    / \                  / \
      //   L  ..   swap(N, S)   L  ..
      //       |   ========>       |
      //       P                   P
      //      / \                 / \
      //     S  ..                N  ..
      // clang-format on

      // Step 1
      NodePtr successor = node->right;
      NodePtr parent = node;
      while (successor->left != nullptr) {
        parent = successor;
        successor = parent->left;
      }
      // Step 2
      swapNode(node, successor);
      maintainRelationship(parent);
      // Step 3: vvv
    }

    if (node->isLeaf()) {
      // Current node must not be the root
      assert(node->parent != nullptr);

      // Case 2: Current node is a leaf
      //   Step 1. Unlink and remove it.
      //   Step 2. If N is BLACK, maintain N;
      //           If N is RED, do nothing.

      // The maintain operation won't change the node itself,
      //  so we can perform maintain operation before unlink the node.
      if (node->isBlack()) {
        maintainAfterRemove(node);
      }
      if (node->direction() == Direction::LEFT) {
        node->parent->left = nullptr;
      } else /* node->direction() == Direction::RIGHT */ {
        node->parent->right = nullptr;
      }
    } else /* !node->isLeaf() */ {
      assert(node->left == nullptr || node->right == nullptr);
      // Case 3: Current node has a single left or right child
      //   Step 1. Replace N with its child
      //   Step 2. If N is BLACK, maintain N
      NodePtr parent = node->parent;
      NodePtr replacement = (node->left != nullptr ? node->left : node->right);
      switch (node->direction()) {
        case Direction::ROOT:
          this->root = replacement;
```

```
            break;
          case Direction::LEFT:
            parent->left = replacement;
            break;
          case Direction::RIGHT:
            parent->right = replacement;
            break;
        }

        if (!node->isRoot()) {
          replacement->parent = parent;
        }

        if (node->isBlack()) {
          if (replacement->isRed()) {
            replacement->color = Node::BLACK;
          } else {
            maintainAfterRemove(replacement);
          }
        }
      }

      this->count -= 1;
      return true;
    }

    void inorderTraversal(NodeConsumer action) const {
      if (this->root == nullptr) {
        return;
      }

      std::stack<NodePtr> stack;
      NodePtr node = this->root;

      while (node != nullptr || !stack.empty()) {
        while (node != nullptr) {
          stack.push(node);
          node = node->left;
        }
        if (!stack.empty()) {
          node = stack.top();
          stack.pop();
          action(node);
          node = node->right;
        }
      }
    }
};
#endif  // RBTREE_MAP_HPP
```

13 动态规划

13.1 动态规划基础

最长公共子序列

子序列的定义可以参考 子序列。一个简要的例子：字符串 `abcde` 与字符串 `acde` 的公共子序列有 `a`、`c`、`d`、`e`、`ac`、`ad`、`ae`、`cd`、`ce`、`de`、`ade`、`ace`、`cde`、`acde`，最长公共子序列的长度是 4。

设 $f(i,j)$ 表示只考虑 A 的前 i 个元素，B 的前 j 个元素时的最长公共子序列的长度，求这时的最长公共子序列的长度就是 **子问题**。$f(i,j)$ 就是我们所说的 **状态**，则 $f(n,m)$ 是最终要达到的状态，即为所求结果。

对于每个 $f(i,j)$，存在三种决策：如果 $A_i = B_j$，则可以将它接到公共子序列的末尾；另外两种决策分别是跳过 A_i 或者 B_j。状态转移方程如下：

$$f(i,j) = \begin{cases} f(i-1,j-1)+1 & A_i = B_j \\ \max(f(i-1,j), f(i,j-1)) & A_i \neq B_j \end{cases}$$

可参考 SourceForge 的 LCS 交互网页 来更好地理解 LCS 的实现过程。

该做法的时间复杂度为 $O(nm)$。

最长不下降子序列

算法一

设 $f(i)$ 表示以 A_i 为结尾的最长不下降子序列的长度，则所求为 $\max_{1\le i\le n} f(i)$。

计算 $f(i)$ 时，尝试将 A_i 接到其他的最长不下降子序列后面，以更新答案。于是可以写出这样的状态转移方程：$f(i) = \max_{1\le j<i, A_j\le A_i}(f(j)+1)$。

容易发现该算法的时间复杂度为 $O(n^2)$。

```
int a[MAXN], d[MAXN];
int dp() {
  d[1] = 1;
  int ans = 1;
  for (int i = 2; i <= n; i++) {
    d[i] = 1;
    for (int j = 1; j < i; j++)
      if (a[j] <= a[i]) {
        d[i] = max(d[i], d[j] + 1);
        ans = max(ans, d[i]);
      }
  }
  return ans;
}
```

算法二($n \log n$)

当 n 的范围扩大到 $n \le 10^5$ 时，第一种做法就不够快了，下面给出了一个 $O(n \log n)$ 的做法。

首先，定义 $a_1 \dots a_n$ 为原始序列，d 为当前的不下降子序列，len 为子序列的长度，那么 d_{len} 就是长度为 len 的不下降子序列末尾元素。

初始化：$d_1 = a_1, len = 1$。

现在我们已知最长的不下降子序列长度为 1，那么我们让 i 从 2 到 n 循环，依次求出前 i 个元素的最长不下降子序列的长度，循环的时候我们只需要维护好 d 这个数组还有 len 就可以了。**关键在于如何维护。**

考虑进来一个元素 a_i：

1. 元素大于等于 d_{len}，直接将该元素插入到 d 序列的末尾。
2. 元素小于 d_{len}，找到 **第一个** 大于它的元素，用 a_i 替换它。

```
for (int i = 0; i < n; ++i) scanf("%d", a + i);
memset(dp, 0x1f, sizeof dp);
mx = dp[0];
for (int i = 0; i < n; ++i) {
  *std::upper_bound(dp, dp + n, a[i]) = a[i];
}
ans = 0;
while (dp[ans] != mx) ++ans;
```

13.2 记忆化搜素

定义

记忆化搜索是一种通过记录已经遍历过的状态的信息，从而避免对同一状态重复遍历的搜索实现方式。

因为记忆化搜索确保了每个状态只访问一次，它也是一种常见的动态规划实现方式。

如果我们每查询完一个状态后将该状态的信息存储下来，再次需要访问这个状态就可以直接使用之前计算得到的信息，从而避免重复计算。这充分利用了动态规划中很多问题具有大量重叠子问题的特点，属于用空间换时间的「记忆化」思想。

与递推的联系与区别

在求解动态规划的问题时，记忆化搜索与递推的代码，在形式上是高度类似的。这是由于它们使用了相同的状态表示方式和类似的状态转移。也正因为如此，一般来说两种实现的时间复杂度是一样的。

下面给出的是递推实现的代码（为了方便对比，没有添加滚动数组优化），通过对比可以发现二者在形式上的类似性。

在求解动态规划的问题时，记忆化搜索和递推，都确保了同一状态至多只被求解一次。而它们实现这一点的方式则略有不同：递推通过设置明确的访问顺序来避免重复访问，记忆化搜索虽然没有明确规定访问顺序，但通过给已经访问过的状态打标记的方式，也达到了同样的目的。

与递推相比，记忆化搜索因为不用明确规定访问顺序，在实现难度上有时低于递推，且能比较方便地处理边界情况，这是记忆化搜索的一大优势。但与此同时，记忆化搜索难以使用滚动数组等优化，且由于存在递归，运行效率会低于递推。因此应该视题目选择更适合的实现方式。

如何写记忆化搜索

方法一

1. 把这道题的 dp 状态和方程写出来
2. 根据它们写出 dfs 函数
3. 添加记忆化数组

举例：

$dp_i = \max\{dp_j + 1\} \quad (1 \le j < i \wedge a_j < a_i)$（最长上升子序列）

方法二

1. 写出这道题的暴搜程序（最好是 dfs）
2. 将这个 dfs 改成「无需外部变量」的 dfs
3. 添加记忆化数组

举例

13.3 背包DP

0-1背包

解释

例题中已知条件有第 i 个物品的重量 w_i，价值 v_i，以及背包的总容量 W。

设 DP 状态 $f_{i,j}$ 为在只能放前 i 个物品的情况下，容量为 j 的背包所能达到的最大总价值。

考虑转移。假设当前已经处理好了前 $i-1$ 个物品的所有状态，那么对于第 i 个物品，当其不放入背包时，背包的剩余容量不变，背包中物品的总价值也不变，故这种情况的最大价值为 $f_{i-1,j}$；当其放入背包时，背包的剩余容量会减小 w_i，背包中物品的总价值会增大 v_i，故这种情况的最大价值为 $f_{i-1,j-w_i} + v_i$。

由此可以得出状态转移方程：

$$f_{i,j} = \max(f_{i-1,j}, f_{i-1,j-w_i} + v_i)$$

这里如果直接采用二维数组对状态进行记录，会出现 MLE。可以考虑改用滚动数组的形式来优化。

由于对 f_i 有影响的只有 f_{i-1}，可以去掉第一维，直接用 f_i 来表示处理到当前物品时背包容量为 i 的最大价值，得出以下方程：

$$f_j = \max(f_j, f_{j-w_i} + v_i)$$

务必牢记并理解这个转移方程，因为大部分背包问题的转移方程都是在此基础上推导出来的。

```
for (int i = 1; i <= n; i++)
  for (int l = W; l >= w[i]; l--)
    f[l] = max(f[l], f[l - w[i]] + v[i]);
```

完全背包

解释

完全背包模型与 0-1 背包类似，与 0-1 背包的区别仅在于一个物品可以选取无限次，而非仅能选取一次。

我们可以借鉴 0-1 背包的思路，进行状态定义：设 $f_{i,j}$ 为只能选前 i 个物品时，容量为 j 的背包可以达到的最大价值。

需要注意的是，虽然定义与 0-1 背包类似，但是其状态转移方程与 0-1 背包并不相同。

过程

可以考虑一个朴素的做法：对于第 件物品，枚举其选了多少个来转移。这样做的时间复杂度是 的。

状态转移方程如下：

$f_{i,j}=\max_{k=0}^{+\infty}(f_{i-1,j-k\times w_i}+v_i\times k)$

考虑做一个简单的优化。可以发现，对于 ，只要通过 转移就可以了。因此状态转移方程为：

$f_{i,j}=\max(f_{i-1,j},f_{i,j-w_i}+v_i)$

理由是当我们这样转移时，$f_{i,j-w_i}$ 已经由 $f_{i,j-2\times w_i}$ 更新过，那么 $f_{i,j-w_i}$ 就是充分考虑了第 i 件物品所选次数后得到的最优结果。换言之，我们通过局部最优子结构的性质重复使用了之前的枚举过程，优化了枚举的复杂度。

与 0-1 背包相同，我们可以将第一维去掉来优化空间复杂度。如果理解了 0-1 背包的优化方式，就不难明白压缩后的循环是正向的（也就是上文中提到的错误优化）。

```
#include <iostream>
using namespace std;
const int maxn = 1e4 + 5;
const int maxW = 1e7 + 5;
int n, W, w[maxn], v[maxn];
long long f[maxW];

int main() {
  cin >> W >> n;
  for (int i = 1; i <= n; i++) cin >> w[i] >> v[i];
  for (int i = 1; i <= n; i++)
    for (int l = w[i]; l <= W; l++)
      if (f[l - w[i]] + v[i] > f[l]) f[l] = f[l - w[i]] + v[i];  // 核心状态方程
  cout << f[W];
  return 0;
}
```

多重背包

多重背包也是 0-1 背包的一个变式。与 0-1 背包的区别在于每种物品有k_i 个，而非一个。

一个很朴素的想法就是：把「每种物品选 k_i 次」等价转换为「有k_i 个相同的物品，每个物品选一次」。这样就转换成了一个 0-1 背包模型，套用上文所述的方法就可已解决。状态转移方程如下：

$f_{i,j}=\max_{k=0}^{k_i}(f_{i-1,j-k\times w_i}+v_i\times k)$

时间复杂度 $O(W\sum_{i=1}^{n}k_i)$。

二进制分组优化

考虑优化。我们仍考虑把多重背包转化成 0-1 背包模型来求解。

解释

显然，复杂度中的 $O(nW)$ 部分无法再优化了，我们只能从 $O(\sum k_i)$处入手。为了表述方便，我们用 $A_{i,j}$ 代表第 i 种物品拆分出的第 j 个物品。

在朴素的做法中，$\forall j\le k_i$，$A_{i,j}$ 均表示相同物品。那么我们效率低的原因主要在于我们进行了大量重复性的工作。举例来说，我们考虑了「同时选$A_{i,1},A_{i,2}$」与「同时选 $A_{i,2},A_{i,3}$」这两个完全等效的情况。这样的重复性工作我们进行了许多次。那么优化拆分方式就成为了解决问题的突破口。

过程

我们可以通过「二进制分组」的方式使拆分方式更加优美。

具体地说就是令 $A_{i,j}\ (j \in [0, \lfloor \log_2(k_i+1) \rfloor - 1])$ 分别表示由 2^j 个单个物品「捆绑」而成的大物品。特殊地，若 k_i+1 不是 2的整数次幂，则需要在最后添加一个由 $k_i - 2^{\lfloor \log_2(k_i+1) \rfloor - 1}$ 个单个物品「捆绑」而成的大物品用于补足。

举几个例子：

- $6 = 1 + 2 + 3$
- 8=1+2+4+1$
- $18 = 1 + 2 + 4 + 8 + 3$
- $31 = 1 + 2 + 4 + 8 + 16$

显然，通过上述拆分方式，可以表示任意 $\leq k_i$ 个物品的等效选择方式。将每种物品按照上述方式拆分后，使用 0-1 背包的方法解决即可。

时间复杂度$O(W \sum_{i=1}^{n} \log_2 k_i)$

实现

```
index = 0;
for (int i = 1; i <= m; i++) {
  int c = 1, p, h, k;
  cin >> p >> h >> k;
  while (k > c) {
    k -= c;
    list[++index].w = c * p;
    list[index].v = c * h;
    c *= 2;
  }
  list[++index].w = p * k;
  list[index].v = h * k;
}
```

混合背包

混合背包就是将前面三种的背包问题混合起来，有的只能取一次，有的能取无限次，有的只能取 次。

这种题目看起来很吓人，可是只要领悟了前面几种背包的中心思想，并将其合并在一起就可以了。下面给出伪代码：

```
for (循环物品种类) {
  if (是 0 - 1 背包)
    套用 0 - 1 背包代码;
  else if (是完全背包)
    套用完全背包代码;
  else if (是多重背包)
    套用多重背包代码;
}
```

二维费用背包

```
for (int k = 1; k <= n; k++)
  for (int i = m; i >= mi; i--)    // 对经费进行一层枚举
    for (int j = t; j >= ti; j--)  // 对时间进行一层枚举
      dp[i][j] = max(dp[i][j], dp[i - mi][j - ti] + 1);
```

分组背包

```
for (int k = 1; k <= ts; k++)            // 循环每一组
  for (int i = m; i >= 0; i--) // 循环背包容量
    for (int j = 1; j <= cnt[k]; j++)   // 循环该组的每一个物品
      if (i >= w[t[k][j]])  // 背包容量充足
        dp[i] = max(dp[i], dp[i - w[t[k][j]]] + c[t[k][j]]);  // 像0-1背包一样状态转移
```

杂项

小优化

根据贪心原理，当费用相同时，只需保留价值最高的；当价值一定时，只需保留费用最低的；当有两件物品 i, j 且 i 的价值大于 j 的价值并且 i 的费用小于 j 的费用时，只需保留 i。

背包问题变种

实现

输出方案其实就是记录下来背包中的某一个状态是怎么推出来的。我们可以用$g_{i,v}$ 表示第 i 件物品占用空间为 v 的时候是否选择了此物品。然后在转移时记录是选用了哪一种策略（选或不选）。输出时的伪代码：

```
int v = V;  // 记录当前的存储空间
// 因为最后一件物品存储的是最终状态，所以从最后一件物品进行循环
for (从最后一件循环至第一件) {
  if (g[i][v]) {
    选了第 i 项物品;
    v -= 第 i 项物品的重量;
  } else {
    未选第 i 项物品;
  }
}
```

求方案数

对于给定的一个背包容量、物品费用、其他关系等的问题，求装到一定容量的方案总数。

这种问题就是把求最大值换成求和即可。

例如 0-1 背包问题的转移方程就变成了：

$$dp_i = \sum(dp_i, dp_{i-c_i})$$

初始条件：$dp_0 = 1$

因为当容量为 0 时也有一个方案，即什么都不装。

求最优方案总数

要求最优方案总数，我们要对 0-1 背包里的 dp 数组的定义稍作修改，DP 状态 为在只能放前 个物品的情况下，容量为 j 的背包「正好装满」所能达到的最大总价值。

这样修改之后，每一种 DP 状态都可以用一个 $g_{i,j}$ 来表示方案数。

$f_{i,j}$ 表示只考虑前 i 个物品时背包体积「正好」是 j 时的最大价值。

$g_{i,j}$表示只考虑前 i 个物品时背包体积「正好」是 j 时的方案数。

转移方程：

如果 $f_{i,j}=f_{i-1,j}$ 且 $f_{i,j}\neq f_{i-1,j-v}+w$ 说明我们此时不选择把物品放入背包更优，方案数由 转移过来，

如果 $f_{i,j}\neq f_{i-1,j}$ 且 $f_{i,j}=f_{i-1,j-v}+w$ 说明我们此时选择把物品放入背包更优，方案数由 转移过来，

如果 $f_{i,j}=f_{i-1,j}$ 且 $f_{i,j}=f_{i-1,j-v}+w$ 说明放入或不放入都能取得最优解，方案数由$g_{i-1,j}$ 和 $g_{i-1,j-v}$ 转移过来。

初始条件：

```
memset(f, 0x3f3f, sizeof(f));  // 避免没有装满而进行了转移
f[0] = 0;
g[0] = 1;  // 什么都不装是一种方案
```

因为背包体积最大值有可能装不满，所以最优解不一定是 f_m。

最后我们通过找到最优解的价值，把 g_j数组里取到最优解的所有方案数相加即可。

```
for (int i = 0; i < N; i++) {
  for (int j = V; j >= v[i]; j--) {
    int tmp = std::max(dp[j], dp[j - v[i]] + w[i]);
    int c = 0;
    if (tmp == dp[j]) c += cnt[j];                        // 如果从dp[j]转移
    if (tmp == dp[j - v[i]] + w[i]) c += cnt[j - v[i]];  // 如果从dp[j-v[i]]转移
    dp[j] = tmp;
    cnt[j] = c;
  }
}
int max = 0;  // 寻找最优解
for (int i = 0; i <= V; i++) {
  max = std::max(max, dp[i]);
}
int res = 0;
for (int i = 0; i <= V; i++) {
  if (dp[i] == max) {
    res += cnt[i];  // 求和最优解方案数
  }
}
```

背包的第 k 优解

普通的 0-1 背包是要求最优解，在普通的背包 DP 方法上稍作改动，增加一维用于记录当前状态下的前 k 优解，即可得到求 0-1 背包第 k 优解的算法。 具体来讲：$dp_{i,j,k}$ 记录了前 i 个物品中，选择的物品总体积为 j 时，能够得到的第 k 大的价值和。这个状态可以理解为将普通 0-1 背包只用记录一个数据的 $dp_{i,j}$ 扩展为记录一个有序的优解序列。转移时，普通背包最优解的求法是 $dp_{i,j}=\max(dp_{i-1,j},dp_{i-1,j-v_i}+w_i)$，现在我们则是要合并 $dp_{i-1,j}$，$dp_{i-1,j-v_i}+w_i$ 这两个大小为 k 的递减序列，并保留合并后前 k 大的价值记在 $dp_{i,j}$ 里，这一步利用双指针法，复杂度是 $O(k)$ 的，整体时间复杂度为 $O(nmk)$。空间上，此方法与普通背包一样可以压缩掉第一维，复杂度是 $O(mk)$ 的。

13.4 区间DP

定义

区间类动态规划是线性动态规划的扩展，它在分阶段地划分问题时，与阶段中元素出现的顺序和由前一阶段的哪些元素合并而来有很大的关系。

令状态$f(i,j)$表示将下标位置i到j的所有元素合并能获得的价值的最大值，那么$f(i,j)=\max\{f(i,k)+f(k+1,j)+cost\}$，$cost$为将这两组元素合并起来的代价。

性质

区间 DP 有以下特点：

合并：即将两个或多个部分进行整合，当然也可以反过来；

特征：能将问题分解为能两两合并的形式；

求解：对整个问题设最优值，枚举合并点，将问题分解为左右两个部分，最后合并两个部分的最优值得到原问题的最优值。

13.5 DAG上的DP

定义

DAG 即 有向无环图，一些实际问题中的二元关系都可使用 DAG 来建模，从而将这些问题转化为 DAG 上的最长（短）路问题。

13.6 树形DP

树形 DP，即在树上进行的 DP。由于树固有的递归性质，树形 DP 一般都是递归进行的。

基础

以下面这道题为例，介绍一下树形 DP 的一般过程。

P1352 没有上司的舞会

我们设$f(i,0/1)$代表以i为根的子树的最优解（第二维的值为 0 代表i不参加舞会的情况，1 代表i参加舞会的情况）。

对于每个状态，都存在两种决策（其中下面的x都是i的儿子）：

- 上司不参加舞会时，下属可以参加，也可以不参加，此时有$f(i,0)=\sum\max\{f(x,1),f(x,0)\}$；
- 上司参加舞会时，下属都不会参加，此时有$f(i,1)=\sum f(x,0)+a_i$。

我们可以通过 DFS，在返回上一层时更新当前结点的最优解。

```
#include <algorithm>
#include <cstdio>
using namespace std;

struct edge {
  int v, next;
} e[6005];

int head[6005], n, cnt, f[6005][2], ans, is_h[6005], vis[6005];
```

```
void addedge(int u, int v) {  // 建图
  e[++cnt].v = v;
  e[cnt].next = head[u];
  head[u] = cnt;
}

void calc(int k) {
  vis[k] = 1;
  for (int i = head[k]; i; i = e[i].next) {  // 枚举该结点的每个子结点
    if (vis[e[i].v]) continue;
    calc(e[i].v);
    f[k][1] += f[e[i].v][0];
    f[k][0] += max(f[e[i].v][0], f[e[i].v][1]);  // 转移方程
  }
  return;
}

int main() {
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) scanf("%d", &f[i][1]);
  for (int i = 1; i < n; i++) {
    int l, k;
    scanf("%d%d", &l, &k);
    is_h[l] = 1;
    addedge(k, l);
  }
  for (int i = 1; i <= n; i++)
    if (!is_h[i]) {  // 从根结点开始DFS
      calc(i);
      printf("%d", max(f[i][1], f[i][0]));
      return 0;
    }
}
```

习题

- HDU 2196 Computer

树上背包

树上的背包问题，简单来说就是背包问题与树形 DP 的结合。

洛谷 P2014 CTSC1997 选课

每门课最多只有一门先修课的特点，与有根树中一个点最多只有一个父亲结点的特点类似。

因此可以想到根据这一性质建树，从而所有课程组成了一个森林的结构。为了方便起见，我们可以新增一门 0 学分的课程（设这个课程的编号为 0），作为所有无先修课课程的先修课，这样我们就将森林变成了一棵以 0 号课程为根的树。

我们设 $f(u,i,j)$ 表示以 u 号点为根的子树中，已经遍历了 u 号点的前 i 棵子树，选了 j 门课程的最大学分。

转移的过程结合了树形 DP 和 背包DP 的特点，我们枚举 u 点的每个子结点v，同时枚举以 v 为根的子树选了几门课程，将子树的结果合并到 u 上。

记点 x 的儿子个数为 s_x，以 x 为根的子树大小为 siz_x，可以写出下面的状态转移方程：

$$f(u,i,j) = \max_{v,k\le j,k\le siz\_v} f(u,i-1,j-k) + f(v,s_v,k)$$

注意上面状态转移方程中的几个限制条件，这些限制条件确保了一些无意义的状态不会被访问到。

f 的第二维可以很轻松地用滚动数组的方式省略掉，注意这时需要倒序枚举 j 的值。

可以证明，该做法的时间复杂度为 $O(nm)$。

```
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;
int f[305][305], s[305], n, m;
vector<int> e[305];

int dfs(int u) {
  int p = 1;
  f[u][1] = s[u];
  for (auto v : e[u]) {
    int siz = dfs(v);
    // 注意下面两重循环的上界和下界
    // 只考虑已经合并过的子树，以及选的课程数超过 m+1 的状态没有意义
    for (int i = min(p, m + 1); i; i--)
      for (int j = 1; j <= siz && i + j <= m + 1; j++)
        f[u][i + j] = max(f[u][i + j], f[u][i] + f[v][j]);  // 转移方程
    p += siz;
  }
  return p;
}

int main() {
  scanf("%d%d", &n, &m);
  for (int i = 1; i <= n; i++) {
    int k;
    scanf("%d%d", &k, &s[i]);
    e[k].push_back(i);
  }
  dfs(0);
  printf("%d", f[0][m + 1]);
  return 0;
}
```

13.7 状压DP

状压 DP 是动态规划的一种,将问题的状态用一个二进制数表示，转换成十进制数储存，并可以适当利用位运算处理。

13.8 数位DP

数位是指把一个数字按照个、十、百、千等等一位一位地拆开，关注它每一位上的数字。如果拆的是十进制数，那么每一位数字都是 0~9，其他进制可类比十进制。

数位 DP：用来解决一类特定问题，这种问题比较好辨认，一般具有这几个特征：

1. 要求统计满足一定条件的数的数量（即，最终目的为计数）；
2. 这些条件经过转化后可以使用「数位」的思想去理解和判断；
3. 输入会提供一个数字区间（有时也只提供上界）来作为统计的限制；
4. 上界很大（比如 10^{18}），暴力枚举验证会超时。

数位 DP 的基本原理：

考虑人类计数的方式，最朴素的计数就是从小到大开始依次加一。但我们发现对于位数比较多的数，这样的过程中有许多重复的部分。例如，从 7000 数到 7999、从 8000 数到 8999、和从 9000 数到 9999 的过程非常相似，它们都是后三位从 000 变到 999，不一样的地方只有千位这一位，所以我们可以把这些过程归并起来，将这些过程中产生的计数答案也都存在一个通用的数组里。此数组根据题目具体要求设置状态，用递推或 DP 的方式进行状态转移。

数位 DP 中通常会利用常规计数问题技巧，比如把一个区间内的答案拆成两部分相减（即 $ans_{[l,r]} = ans_{[0,r]} - ans_{[0,l-1]}$）

那么有了通用答案数组，接下来就是统计答案。统计答案可以选择记忆化搜索，也可以选择循环迭代递推。为了不重不漏地统计所有不超过上限的答案，要从高到低枚举每一位，再考虑每一位都可以填哪些数字，最后利用通用答案数组统计答案。

13.附加 DP优化

四边形不等式优化

区间类（2D1D）动态规划中的应用

在区间类动态规划（如石子合并问题）中，我们经常遇到以下形式的 2D1D 状态转移方程：

$$f_{l,r} = \min_{k=l}^{r-1}\{f_{l,k} + f_{k+1,r}\} + w(l,r) \qquad (1 \le l < r \le n)$$

直接简单实现状态转移，总时间复杂度将会达到 $O(n^3)$，但当函数 $w(l,r)$ 满足一些特殊的性质时，我们可以利用决策的单调性进行优化。

- **区间包含单调性**：如果对于任意 $l \le l' \le r' \le r$，均有 $w(l',r') \le w(l,r)$ 成立，则称函数 w 对于区间包含关系具有单调性。
- **四边形不等式**：如果对于任意 $l_1 \le l_2 \le r_1 \le r_2$，均有 $w(l_1,r_1) + w(l_2,r_2) \le w(l_1,r_2) + w(l_2,r_1)$ 成立，则称函数 w 满足四边形不等式（简记为「交叉小于包含」）。若等号永远成立，则称函数 w 满足 **四边形恒等式**。

引理 1：若 $w(l,r)$ 满足区间包含单调性和四边形不等式，则状态 $f_{l,r}$ 满足四边形不等式。

杂项

随机化

本文将对 OI/ICPC 中的随机化相关技巧做一个简单的分类，并对每个分类予以介绍。本文也将介绍一些在 OI/ICPC 中很少使用，但与 OI/ICPC 在风格等方面较为贴近的方法，这些内容前将用 `(*)` 标注。

这一分类并不代表广泛共识，也必定不能囊括所有可能性，因此仅供参考。

记号和约定：

- 表示事件 发生的概率。
- 表示随机变量 的期望。
- 赋值号 表示引入新的量，例如 表示引入值为 的量 。

用随机集合覆盖目标元素

庞大的解空间中有一个（或多个）解是我们想要的。我们可以尝试进行多次撒网，只要有一次能够网住目标解就能成功。

附加公式

Stirling公式（阶乘近似公式）

$n! \approx \sqrt{(2\pi n)}(\frac{n}{e})^n$

LRU缓存

```
struct DLinkedNode {
    int key, value;
    DLinkedNode* prev;
    DLinkedNode* next;
    DLinkedNode(): key(0), value(0), prev(nullptr), next(nullptr) {}
    DLinkedNode(int _key, int _value): key(_key), value(_value), prev(nullptr),
next(nullptr) {}
};
class LRUCache {
private:
    unordered_map<int, DLinkedNode*> cache;
    DLinkedNode* head;
    DLinkedNode* tail;
    int size;
    int capacity;
public:
    LRUCache(int capacity1) {
        capacity=capacity1;
        size = 0;
        head = new DLinkedNode;
        tail = new DLinkedNode;
        head->next = tail;
        tail->prev = head;
        cache.clear();
    }
    void removeNode(DLinkedNode* node) {
        node->prev->next = node->next;
        node->next->prev = node->prev;
        cache.erase(node->key);
    }
    void addToHead(DLinkedNode* node) {
        node->prev = head;
        node->next = head->next;
        head->next->prev = node;
        head->next = node;
        cache[node->key]=node;
    }
    void movetohead(DLinkedNode*node) {
        removeNode(node);
        addToHead(node);
    }
    DLinkedNode* removeTail() {
```

```
        DLinkedNode* node = tail->prev;
        removeNode(node);
        return node;
    }
    int get(int key) {
        if(!cache.count(key)) {
            return -1;
        }
        DLinkedNode*node=cache[key];
        movetohead(node);
        return node->value;
    }

    void put(int key, int value) {
        if(!cache.count(key)) {
            DLinkedNode*node =new DLinkedNode(key,value);
            addToHead(node);
            size++;
            if(size>capacity){
                size--;
                removeTail();
            }
        }
        else{
            DLinkedNode*node=cache[key];
            node->value=value;
            movetohead(node);
        }
    }
};
```