优化输入输出

```
ios::sync_with_stdio(false);
cin.tie(0);
cout.tie(0);
```

序列容器

vector - 动态数组

成员函数	功能描述	示例代码	时间复杂度
push_back()	尾部添加元素	vec.push_back(10);	O(1) 均摊
pop_back()	删除尾部元素	vec.pop_back();	O(1)
insert()	指定位置插入元素	vec.insert(vec.begin()+1, 5);	O(n)
erase()	删除指定位置元素	vec.erase(vec.begin());	O(n)
operator[]	随机访问元素	int x = vec[2];	O(1)
at()	带边界检查的访问	int x = vec.at(2);	O(1)
front()	访问首元素	int x = vec.front();	O(1)
back()	访问尾元素	int x = vec.back();	O(1)
size()	获取元素数量	int n = vec.size();	O(1)
resize()	调整容器大小	vec.resize(20);	O(n)
reserve()	预分配内存	vec.reserve(100);	O(n)
clear()	清空容器	vec.clear();	O(n)
empty()	检查是否为空	if (vec.empty()) {...}	O(1)

deque - 双端队列

成员函数	功能描述	示例代码
push_front()	头部添加元素	dq.push_front(10);
push_back()	尾部添加元素	dq.push_back(20);
pop_front()	删除头部元素	dq.pop_front();
pop_back()	删除尾部元素	dq.pop_back();
operator[]	随机访问	int x = dq[1];

list - 双向链表

成员函数	功能描述	示例代码
`push_front()`	头部添加元素	`lst.push_front(5);`
`pop_front()`	删除头部元素	`lst.pop_front();`
`splice()`	转移元素到新位置	`lst.splice(it, other_list);`
`remove()`	删除所有匹配值	`lst.remove(42);`
`unique()`	删除连续重复元素	`lst.unique();`
`merge()`	合并有序链表	`lst.merge(sorted_list);`
`sort()`	链表排序	`lst.sort();`

关联容器

set/multiset - 有序集合

成员函数	功能描述	示例代码	时间复杂度
`insert()`	插入元素	`s.insert(42);`	O(log n)
`erase()`	删除元素	`s.erase(42);`	O(log n)
`find()`	查找元素	`auto it = s.find(42);`	O(log n)
`count()`	统计元素数量	`int n = s.count(42);`	O(log n)
`lower_bound()`	返回首个不小于值的迭代器	`auto it = s.lower_bound(30);`	O(log n)
`upper_bound()`	返回首个大于值的迭代器	`auto it = s.upper_bound(50);`	O(log n)
`equal_range()`	返回匹配区间	`auto p = s.equal_range(42);`	O(log n)

map/multimap - 键值对容器

成员函数	功能描述	示例代码
`operator[]`	访问/插入元素	`m["key"] = value;`
`insert()`	插入键值对	`m.insert({"key", value});`
`emplace()`	原地构造元素	`m.emplace("key", value);`
`at()`	带异常检查的访问	`int x = m.at("key");`
`erase()`	删除键	`m.erase("key");`

unordered_set/map - 哈希容器

成员函数	功能描述	示例代码	时间复杂度
`insert()`	插入元素	`us.insert(42);`	O(1) 平均
`erase()`	删除元素	`us.erase(42);`	O(1) 平均
`find()`	查找元素	`auto it = um.find("key");`	O(1) 平均
`bucket_count()`	获取桶数量	`int n = um.bucket_count();`	O(1)
`load_factor()`	获取负载因子	`float lf = um.load_factor();`	O(1)
`rehash()`	调整哈希表大小	`um.rehash(100);`	O(n)

容器适配器

stack - 栈

成员函数	功能描述	示例代码
`push()`	压入元素	`stk.push(10);`
`pop()`	弹出元素	`stk.pop();`
`top()`	访问栈顶	`int x = stk.top();`
`empty()`	检查空栈	`if (stk.empty()) {...}`

queue - 队列

成员函数	功能描述	示例代码
`push()`	入队	`q.push(20);`
`pop()`	出队	`q.pop();`
`front()`	访问队首	`int x = q.front();`
`back()`	访问队尾	`int x = q.back();`

priority_queue - 优先队列

成员函数	功能描述	示例代码
`push()`	添加元素	`pq.push(30);`
`pop()`	删除堆顶	`pq.pop();`
`top()`	访问堆顶	`int x = pq.top();`

C++ STL 高级用法模板

下面是一个更全面的 C++ STL 使用模板，包含优先队列、排序算法和 vector 初始化的详细用法：

```cpp
#include <iostream>
#include <vector>
#include <queue>
#include <algorithm>
#include <functional>
#include <set>
#include <unordered_set>

using namespace std;

// 自定义比较函数用于排序
bool customCompare(int a, int b) {
    return a > b; // 降序排列
}

// 自定义结构体
struct Person {
    string name;
    int age;

    // 重载<运算符用于优先队列比较
    bool operator<(const Person& other) const {
        return age < other.age; // 年龄大的优先级高
    }

    // 重载==运算符用于unordered_set
    bool operator==(const Person& other) const {
        return name == other.name && age == other.age;
    }
};

// 为自定义结构体提供哈希函数
namespace std {
    template<>
    struct hash<Person> {
        size_t operator()(const Person& p) const {
            return hash<string>()(p.name) ^ hash<int>()(p.age);
        }
    };
}

int main() {
    /******************** 1. 优先队列用法 ********************/
    cout << "=== 优先队列用法 ===" << endl;

    // 默认最大堆
    priority_queue<int> maxHeap;
    maxHeap.push(3);
    maxHeap.push(1);
    maxHeap.push(4);
    cout << "最大堆顶部: " << maxHeap.top() << endl; // 输出4
```

```cpp
// 最小堆 - 使用greater<int>
priority_queue<int, vector<int>, greater<int>> minHeap;
minHeap.push(3);
minHeap.push(1);
minHeap.push(4);
cout << "最小堆顶部: " << minHeap.top() << endl; // 输出1

// 自定义比较函数的最小堆
auto cmp = [](int left, int right) { return left > right; };
priority_queue<int, vector<int>, decltype(cmp)> customMinHeap(cmp);
customMinHeap.push(3);
customMinHeap.push(1);
customMinHeap.push(4);
cout << "自定义最小堆顶部: " << customMinHeap.top() << endl; // 输出1

// 使用自定义结构体的优先队列
priority_queue<Person> personQueue;
personQueue.push({"Alice", 25});
personQueue.push({"Bob", 30});
personQueue.push({"Charlie", 20});
cout << "年龄最大的人: " << personQueue.top().name << endl; // 输出Bob

/******************** 2. Vector 初始化方法 ********************/
cout << "\n=== Vector 初始化方法 ===" << endl;

// 1. 默认初始化
vector<int> v1;
cout << "v1大小: " << v1.size() << endl;

// 2. 指定大小和初始值
vector<int> v2(5, 10); // 5个元素，每个都是10
cout << "v2: ";
for (int x : v2) cout << x << " ";
cout << endl;

// 3. 指定大小，使用默认初始值(0)
vector<int> v3(5); // 5个元素，都是0
cout << "v3: ";
for (int x : v3) cout << x << " ";
cout << endl;

// 4. 使用初始化列表
vector<int> v4 = {1, 2, 3, 4, 5};
cout << "v4: ";
for (int x : v4) cout << x << " ";
cout << endl;

// 5. 从数组初始化
int arr[] = {6, 7, 8, 9, 10};
vector<int> v5(arr, arr + sizeof(arr)/sizeof(arr[0]));
cout << "v5: ";
for (int x : v5) cout << x << " ";
cout << endl;
```

```
    // 6. 从另一个vector初始化
    vector<int> v6(v4.begin(), v4.end());
    cout << "v6: ";
    for (int x : v6) cout << x << " ";
    cout << endl;

    // 7. 使用fill初始化
    vector<int> v7(5);
    fill(v7.begin(), v7.end(), 42);
    cout << "v7: ";
    for (int x : v7) cout << x << " ";
    cout << endl;

    /******************** 3. 排序算法用法 ********************/
    cout << "\n=== 排序算法用法 ===" << endl;

    // 1. vector排序
    vector<int> nums = {5, 2, 8, 1, 9};
    sort(nums.begin(), nums.end()); // 默认升序
    cout << "升序排序: ";
    for (int x : nums) cout << x << " ";
    cout << endl;

    // 2. 降序排序
    sort(nums.begin(), nums.end(), greater<int>());
    cout << "降序排序: ";
    for (int x : nums) cout << x << " ";
    cout << endl;

    // 3. 使用自定义比较函数
    sort(nums.begin(), nums.end(), customCompare);
    cout << "自定义比较函数排序: ";
    for (int x : nums) cout << x << " ";
    cout << endl;

    // 4. 使用Lambda表达式排序
    sort(nums.begin(), nums.end(), [](int a, int b) {
        return a < b; // 升序
    });
    cout << "Lambda升序排序: ";
    for (int x : nums) cout << x << " ";
    cout << endl;

    // 5. 部分排序
    vector<int> bigNums = {9, 8, 7, 6, 5, 4, 3, 2, 1};
    partial_sort(bigNums.begin(), bigNums.begin() + 3, bigNums.end());
    cout << "部分排序(前3个最小): ";
    for (int x : bigNums) cout << x << " ";
    cout << endl;

    // 6. 对自定义结构体排序
    vector<Person> people = {{"Alice", 25}, {"Bob", 30}, {"Charlie", 20}};
    sort(people.begin(), people.end(), [](const Person& a, const Person& b)
```

```
{
        return a.age < b.age; // 按年龄升序
    });
    cout << "按年龄排序: ";
    for (const auto& p : people) cout << p.name << "(" << p.age << ") ";
    cout << endl;

    /******************** 4. Set 和 Map 的高级用法 ********************/
    cout << "\n=== Set 和 Map 的高级用法 ===" << endl;

    // 1. 自定义比较函数的set
    auto setCmp = [](int a, int b) { return a > b; };
    set<int, decltype(setCmp)> customSet(setCmp);
    customSet.insert(3);
    customSet.insert(1);
    customSet.insert(4);
    cout << "自定义set(降序): ";
    for (int x : customSet) cout << x << " ";
    cout << endl;

    // 2. 存储自定义对象的set
    set<Person> personSet; // 需要重载<运算符
    personSet.insert({"Alice", 25});
    personSet.insert({"Bob", 30});
    personSet.insert({"Charlie", 20});
    cout << "Person set: ";
    for (const auto& p : personSet) cout << p.name << " ";
    cout << endl;

    // 3. 自定义哈希函数的unordered_set
    unordered_set<Person> personHashSet; // 需要提供哈希函数和==运算符
    personHashSet.insert({"Alice", 25});
    personHashSet.insert({"Bob", 30});
    personHashSet.insert({"Charlie", 20});
    cout << "Person哈希set大小: " << personHashSet.size() << endl;

    /******************** 5. 算法函数示例 ********************/
    cout << "\n=== 算法函数示例 ===" << endl;

    vector<int> data = {1, 2, 3, 4, 5, 6, 7, 8, 9};

    // 1. 查找算法
    auto it = find(data.begin(), data.end(), 5);
    if (it != data.end()) {
        cout << "找到5，位置: " << distance(data.begin(), it) << endl;
    }

    // 2. 计数算法
    int count = count_if(data.begin(), data.end(), [](int x) {
        return x % 2 == 0; // 偶数
    });
    cout << "偶数个数: " << count << endl;

    // 3. 变换算法
```

```
    vector<int> doubled;
    transform(data.begin(), data.end(), back_inserter(doubled),
              [](int x) { return x * 2; });
    cout << "加倍后的数据: ";
    for (int x : doubled) cout << x << " ";
    cout << endl;

    // 4. 删除算法
    vector<int> numsWithDups = {1, 2, 2, 3, 4, 4, 5};
    auto last = unique(numsWithDups.begin(), numsWithDups.end());
    numsWithDups.erase(last, numsWithDups.end());
    cout << "去重后的数据: ";
    for (int x : numsWithDups) cout << x << " ";
    cout << endl;

    return 0;
}
```

1. 优先队列用法总结

类型	声明方式	说明
最大堆	`priority_queue<int>`	默认，顶部元素最大
最小堆	`priority_queue<int, vector<int>, greater<int>>`	使用greater，顶部元素最小
自定义比较	`priority_queue<int, vector<int>, decltype(cmp)>`	使用自定义比较函数
自定义结构体	`priority_queue<Person>`	需要重载<运算符

2. Vector 初始化方法

方法	示例	说明
默认初始化	`vector<int> v1;`	空vector
指定大小和初值	`vector<int> v2(5, 10);`	5个元素，每个都是10
指定大小	`vector<int> v3(5);`	5个元素，默认值0
初始化列表	`vector<int> v4 = {1, 2, 3};`	使用花括号初始化
从数组初始化	`vector<int> v5(arr, arr+size);`	使用数组指针范围
从另一个vector	`vector<int> v6(v4.begin(), v4.end());`	使用迭代器范围
使用fill	`fill(v.begin(), v.end(), value);`	填充特定值

3. 排序算法总结

排序类型	示例	说明
默认升序	`sort(vec.begin(), vec.end());`	使用<运算符
降序排序	`sort(vec.begin(), vec.end(), greater<int>());`	使用greater
自定义函数	`sort(vec.begin(), vec.end(), customCompare);`	使用自定义比较函数
Lambda表达式	`sort(vec.begin(), vec.end(), [](a,b){return a<b;});`	使用Lambda
部分排序	`partial_sort(start, middle, end);`	部分排序
自定义对象	`sort(people.begin(), people.end(), comp);`	需要提供比较方法

快读

```
// 快读模板
template <typename T>
void read(T &x) {
    x = 0;
    T f = 1;
    char ch = getchar();
    while (!isdigit(ch)) {
        if (ch == '-') f = -1;
        ch = getchar();
    }
    while (isdigit(ch)) {
        x = x * 10 + ch - '0';
        ch = getchar();
    }
    x *= f;
}
```

并查集

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

#include <unordered_map>
#include <vector>
using namespace std;

template<typename T>
class UnionFind {
private:
    unordered_map<T, T> parent;  // 父节点映射
```

```
    unordered_map<T, int> rank;    // 秩映射
    int count = 0;                 // 连通分量计数

public:
    // 添加新元素
    void add(T x) {
        if (parent.count(x)) return;
        parent[x] = x;
        rank[x] = 0;
        count++;
    }

    // 带路径压缩的查找
    T find(T x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }

    // 按秩合并
    void unite(T x, T y) {
        T rx = find(x), ry = find(y);
        if (rx == ry) return;

        if (rank[rx] < rank[ry]) {
            parent[rx] = ry;
        } else if (rank[rx] > rank[ry]) {
            parent[ry] = rx;
        } else {
            parent[ry] = rx;
            rank[rx]++;
        }
        count--;
    }

    // 检查连通性
    bool connected(T x, T y) {
        return find(x) == find(y);
    }

    // 连通分量数量
    int componentCount() const {
        return count;
    }
};

void solve(){
    int n;
    cin>>n;
    UnionFind<int> a;
    vector<pair<int,int>> b;
    int i,j,e;
    while(n--){
```

```
            cin>>i>>j>>e;
            a.add(i);
            a.add(j);
            if(e==1){
                a.unite(i,j);
            }else{
                b.push_back({i,j});
            }
        }
        for(auto c:b){
            if(a.connected(c.first,c.second)){
                cout<<"NO"<<endl;
                return;
            }
        }
        cout<<"YES"<<endl;
        return;
    }

    int main(){
        int t;
        cin>>t;
        while(t--){
            solve();
        }
        return 0;
    }
```

kmp

```
#include<bits/stdc++.h>

using namespace std;

vector<int> pattern_next(const string& pattern){
    int m = pattern.size();
    vector<int> next(m, 0);
    for (int i = 1, j = 0; i < m; i++) {
        while (j > 0 && pattern[i] != pattern[j])
            j = next[j - 1];
        if (pattern[i] == pattern[j])
            j++;
        next[i] = j;
    }
    return next;
}

vector<int> kmp(const string& text, const string& pattern, vector<int>
next) {
    vector<int> ans;
```

```
    int n = text.size(), m = pattern.size();
    //if (m == 0) return 0;
    if (n < m) return ans;

    for (int i = 0, j = 0; i < n; i++) {
        while (j > 0 && text[i] != pattern[j])
            j = next[j - 1];
        if (text[i] == pattern[j])
            j++;
        if (j == m)
            ans.push_back(i - m + 1);
    }
    return ans;
}

int main(){
    string text,pattern;
    cin>>text>>pattern;
    vector<int> next=pattern_next(pattern);
    vector<int> ans=kmp(text,pattern,next);
    for(auto i:ans)cout<<i+1<<endl;
    for(auto i:next)cout<<i<<" ";
    cout<<endl;
    return 0;
}
```

trie

```
#include<bits/stdc++.h>

using namespace std;

int getidx(char ch) {
    if (ch >= 'a' && ch <= 'z') return ch - 'a';
    else if (ch >= 'A' && ch <= 'Z') return ch - 'A' + 26;
    else return ch - '0' + 52;
}

struct trie {
    int nex[3000001][62], cnt;
    int pass[3000001];
    int endp[3000001];

    void insert(string s) {
        int p = 0; // 从根节点0开始
        pass[p]++;
        for (int i = 0; i < s.length(); i++) {
            int c = getidx(s[i]);
            if (!nex[p][c]) {
                nex[p][c] = ++cnt;
```

```
                // 初始化新节点
                for (int j = 0; j < 62; j++) {
                    nex[cnt][j] = 0;
                }
                pass[cnt] = 0;
                endp[cnt] = 0;
            }
            p = nex[p][c];
            pass[p]++;
        }
        endp[p]++;
    }

    int search_whole_s(string s) {
        int p = 0;
        for (int i = 0; i < s.length(); i++) {
            int c = getidx(s[i]);
            if (!nex[p][c]) return 0;
            p = nex[p][c];
        }
        return endp[p];
    }

    int search_prefix_s(string s) {
        int p = 0;
        for (int i = 0; i < s.length(); i++) {
            int c = getidx(s[i]);
            if (!nex[p][c]) return 0;
            p = nex[p][c];
        }
        return pass[p];
    }

    void clear() {
        // 重置根节点0
        for (int j = 0; j < 62; j++) {
            nex[0][j] = 0;
        }
        pass[0] = 0;
        endp[0] = 0;
        cnt = 0; // 重置节点计数器
    }
};

trie a;

void solve() {
    a.clear();
    int n, p;
    cin >> n >> p;
    string s;
    for (int i = 0; i < n; i++) {
        cin >> s;
        a.insert(s);
```

```
    }
    for (int i = 0; i < p; i++) {
        cin >> s;
        cout << a.search_prefix_s(s) << endl;
    }
}

int main() {
    int t;
    cin >> t;
    while (t--) {
        solve();
    }
    return 0;
}
```

ac自动机

```
#include <cstdio>
#include <cstring>
#include <queue>
using namespace std;

constexpr int N = 2e5 + 6;
constexpr int LEN = 2e6 + 6;
constexpr int SIZE = 2e5 + 6;

int n;

namespace AC {
struct Node {
  int son[26];  // 子结点
  int ans;      // 匹配计数
  int fail;     // fail 指针
  int du;       // 入度
  int idx;

  void init() {  // 结点初始化
    memset(son, 0, sizeof(son));
    ans = fail = idx = 0;
  }
} tr[SIZE];

int tot;  // 结点总数
int ans[N], pidx;

void init() {
  tot = pidx = 0;
  tr[0].init();
}
```

```
void insert(char s[], int &idx) {
  int u = 0;
  for (int i = 1; s[i]; i++) {
    int &son = tr[u].son[s[i] - 'a'];  // 下一个子结点的引用
    if (!son) son = ++tot, tr[son].init();  // 如果没有则插入新结点，并初始化
    u = son;                                // 从下一个结点继续
  }
  // 由于有可能出现相同的模式串，需要将相同的映射到同一个编号
  if (!tr[u].idx) tr[u].idx = ++pidx;  // 第一次出现，新增编号
  idx = tr[u].idx;  // 这个模式串的编号对应这个结点的编号
}

void build() {
  queue<int> q;
  for (int i = 0; i < 26; i++)
    if (tr[0].son[i]) q.push(tr[0].son[i]);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    for (int i = 0; i < 26; i++) {
      if (tr[u].son[i]) {                                    // 存在对应子结点
        tr[tr[u].son[i]].fail = tr[tr[u].fail].son[i];  // 只用跳一次 fail 指
针
        tr[tr[tr[u].fail].son[i]].du++;                  // 入度计数
        q.push(tr[u].son[i]);                            // 并加入队列
      } else
        tr[u].son[i] =
            tr[tr[u].fail]
                .son[i];  // 将不存在的字典树的状态链接到了失配指针的对应状态
    }
  }
}

void query(char t[]) {
  int u = 0;
  for (int i = 1; t[i]; i++) {
    u = tr[u].son[t[i] - 'a'];  // 转移
    tr[u].ans++;
  }
}

void topu() {
  queue<int> q;
  for (int i = 0; i <= tot; i++)
    if (tr[i].du == 0) q.push(i);
  while (!q.empty()) {
    int u = q.front();
    q.pop();
    ans[tr[u].idx] = tr[u].ans;
    int v = tr[u].fail;
    tr[v].ans += tr[u].ans;
    if (!--tr[v].du) q.push(v);
  }
```

```
}
}  // namespace AC

char s[LEN];
int idx[N];

int main() {
  AC::init();
  scanf("%d", &n);
  for (int i = 1; i <= n; i++) {
    scanf("%s", s + 1);
    AC::insert(s, idx[i]);
    AC::ans[i] = 0;
  }
  AC::build();
  scanf("%s", s + 1);
  AC::query(s);
  AC::topu();
  for (int i = 1; i <= n; i++) {
    printf("%d\n", AC::ans[idx[i]]);
  }
  return 0;
}
```

线段树

```
#include <vector>
#include <iostream>
using namespace std;

class SegmentTree {
private:
    int n;  // 原始数组的大小
    vector<long long> tree;  // 线段树数组（存储区间和）
    vector<long long> lazy;  // 懒惰标记数组（存储区间待加的值）

    // 构建线段树
    void build(const vector<long long>& arr, int node, int start, int end)
{
        if (start == end) {
            tree[node] = arr[start];
            return;
        }
        int mid = (start + end) / 2;
        int left_node = 2 * node + 1;
        int right_node = 2 * node + 2;
        build(arr, left_node, start, mid);
        build(arr, right_node, mid + 1, end);
        tree[node] = tree[left_node] + tree[right_node];
    }
```

```
    // 下推懒惰标记
    void push_down(int node, int start, int end) {
        if (lazy[node] != 0) {
            // 更新当前节点值
            int mid = (start + end) / 2;
            int left_node = 2 * node + 1;
            int right_node = 2 * node + 2;

            // 更新左右子节点的值和懒惰标记
            tree[left_node] += lazy[node] * (mid - start + 1);
            tree[right_node] += lazy[node] * (end - mid);
            lazy[left_node] += lazy[node];
            lazy[right_node] += lazy[node];

            // 清除当前节点的懒惰标记
            lazy[node] = 0;
        }
    }

    // 区间更新：在区间 [l, r] 上加上值 val
    void update_range(int node, int start, int end, int l, int r, long long
val) {
        // 当前区间完全包含在更新区间内
        if (l <= start && end <= r) {
            tree[node] += val * (end - start + 1);
            lazy[node] += val;
            return;
        }

        push_down(node, start, end);  // 下推懒惰标记
        int mid = (start + end) / 2;
        int left_node = 2 * node + 1;
        int right_node = 2 * node + 2;

        if (l <= mid) update_range(left_node, start, mid, l, r, val);
        if (r > mid) update_range(right_node, mid + 1, end, l, r, val);

        tree[node] = tree[left_node] + tree[right_node];
    }

    // 区间查询：求区间 [l, r] 的和
    long long query_range(int node, int start, int end, int l, int r) {
        if (r < start || end < l) return 0;  // 区间无交集
        if (l <= start && end <= r) return tree[node];  // 当前区间完全包含在
查询区间内

        push_down(node, start, end);  // 下推懒惰标记
        int mid = (start + end) / 2;
        int left_node = 2 * node + 1;
        int right_node = 2 * node + 2;

        long long left_sum = query_range(left_node, start, mid, l, r);
        long long right_sum = query_range(right_node, mid + 1, end, l, r);
```

```
        return left_sum + right_sum;
    }

public:
    SegmentTree(const vector<long long>& arr) {
        n = arr.size();
        tree.resize(4 * n, 0);  // 分配4倍空间
        lazy.resize(4 * n, 0);
        build(arr, 0, 0, n - 1);
    }

    // 区间更新接口：给区间 [l, r] 的每个元素加上 val
    void update(int l, int r, long long val) {
        update_range(0, 0, n - 1, l, r, val);
    }

    // 区间查询接口：返回区间 [l, r] 的元素和
    long long query(int l, int r) {
        return query_range(0, 0, n - 1, l, r);
    }
};

// 使用示例
int main() {
    int m,n,q,x,y,i;
    cin>>n>>m;
    vector<long long> arr;
    long long a,k;
    for(i=0;i<n;i++){
        cin>>a;
        arr.push_back(a);
    }
    SegmentTree st(arr);

    for(i=0;i<m;i++){
        cin>>q>>x>>y;
        if(q==1){
            cin>>k;
            st.update(x-1,y-1,k);
        }else{
            cout<<st.query(x-1,y-1)<<endl;
        }
    }

    return 0;
}
```

```
#include <iostream>
#include <vector>
using namespace std;
```

```
class SegmentTree {
private:
    int n, mod;
    vector<long long> tree;
    vector<long long> add;
    vector<long long> mul;

    void push_down(int node, int start, int end) {
        int mid = (start + end) / 2;
        int left_node = node * 2 + 1;
        int right_node = node * 2 + 2;
        int left_len = mid - start + 1;
        int right_len = end - mid;

        // 更新左子树
        if (left_node < tree.size()) {
            // 先应用乘法标记，再应用加法标记
            tree[left_node] = (tree[left_node] * mul[node] % mod +
add[node] * left_len % mod) % mod;
            // 更新左子树的懒惰标记
            mul[left_node] = (mul[left_node] * mul[node]) % mod;
            add[left_node] = (add[left_node] * mul[node] % mod + add[node])
% mod;
        }

        // 更新右子树
        if (right_node < tree.size()) {
            tree[right_node] = (tree[right_node] * mul[node] % mod +
add[node] * right_len % mod) % mod;
            mul[right_node] = (mul[right_node] * mul[node]) % mod;
            add[right_node] = (add[right_node] * mul[node] % mod +
add[node]) % mod;
        }

        // 重置当前节点标记
        mul[node] = 1;
        add[node] = 0;
    }

    void build(const vector<long long>& arr, int node, int start, int end)
{
        if (start == end) {
            tree[node] = arr[start] % mod;
            return;
        }
        int mid = (start + end) / 2;
        int left_node = node * 2 + 1;
        int right_node = node * 2 + 2;
        build(arr, left_node, start, mid);
        build(arr, right_node, mid + 1, end);
        tree[node] = (tree[left_node] + tree[right_node]) % mod;
    }

    void update_mul(int node, int start, int end, int l, int r, long long
```

```
k) {
        if (r < start || end < l) return;
        if (l <= start && end <= r) {
            // 更新当前节点值和标记
            tree[node] = (tree[node] * k) % mod;
            mul[node] = (mul[node] * k) % mod;
            add[node] = (add[node] * k) % mod;
            return;
        }
        push_down(node, start, end);
        int mid = (start + end) / 2;
        int left_node = node * 2 + 1;
        int right_node = node * 2 + 2;
        update_mul(left_node, start, mid, l, r, k);
        update_mul(right_node, mid + 1, end, l, r, k);
        tree[node] = (tree[left_node] + tree[right_node]) % mod;
    }

    void update_add(int node, int start, int end, int l, int r, long long
k) {
        if (r < start || end < l) return;
        if (l <= start && end <= r) {
            // 更新当前节点值和标记
            int len = end - start + 1;
            tree[node] = (tree[node] + k * len) % mod;
            add[node] = (add[node] + k) % mod;
            return;
        }
        push_down(node, start, end);
        int mid = (start + end) / 2;
        int left_node = node * 2 + 1;
        int right_node = node * 2 + 2;
        update_add(left_node, start, mid, l, r, k);
        update_add(right_node, mid + 1, end, l, r, k);
        tree[node] = (tree[left_node] + tree[right_node]) % mod;
    }

    long long query(int node, int start, int end, int l, int r) {
        if (r < start || end < l) return 0;
        if (l <= start && end <= r) return tree[node];

        push_down(node, start, end);
        int mid = (start + end) / 2;
        int left_node = node * 2 + 1;
        int right_node = node * 2 + 2;
        long long left_sum = query(left_node, start, mid, l, r);
        long long right_sum = query(right_node, mid + 1, end, l, r);
        return (left_sum + right_sum) % mod;
    }

public:
    SegmentTree(const vector<long long>& arr, int mod) {
        this->mod = mod;
        n = arr.size();
```

```
        tree.resize(4 * n, 0);
        add.resize(4 * n, 0);
        mul.resize(4 * n, 1);
        build(arr, 0, 0, n - 1);
    }

    void multiply(int l, int r, long long k) {
        update_mul(0, 0, n - 1, l, r, k);
    }

    void addVal(int l, int r, long long k) {
        update_add(0, 0, n - 1, l, r, k);
    }

    long long sum(int l, int r) {
        return query(0, 0, n - 1, l, r);
    }
};

int main() {
    int n, q, mod;
    scanf("%d %d %d", &n, &q, &mod);
    vector<long long> arr(n);
    for (int i = 0; i < n; i++) {
        scanf("%lld", &arr[i]);
    }

    SegmentTree st(arr, mod);

    while (q--) {
        int op, x, y;
        long long k;
        scanf("%d", &op);
        if (op == 1) {
            scanf("%d %d %lld", &x, &y, &k);
            st.multiply(x - 1, y - 1, k % mod);
        } else if (op == 2) {
            scanf("%d %d %lld", &x, &y, &k);
            st.addVal(x - 1, y - 1, k % mod);
        } else {
            scanf("%d %d", &x, &y);
            printf("%lld\n", st.sum(x - 1, y - 1));
        }
    }
    return 0;
}
```

哈希

字符串哈希

```
using ull = unsigned long long;
ull base = 131, mod1 = 212370440130137957, mod2 = 1e9 + 7;

ull get_hash1(std::string s) {
    ull ans = 0,len=s.size();
    for (int i = 0; i < len; i++) ans = (ans * base + (ull)s[i]) % mod1;
    return ans;
}

ull get_hash2(std::string s) {
    ull ans = 0,len=s.size(); ...(mod2)
    return ans;
}

bool cmp(const std::string s, const std::string t) {
    return get_hash1(s) != get_hash1(t) || get_hash2(s) != get_hash2(t);
}
```

树哈希

```
//siz为子树大小,h为子树哈希值,g为x为根时的树哈希
void dfs1(int x, int f) {
    siz[x] = 1;
    h[x] = 1;
    for(auto y : edge[x]) {
        if(y == f) continue;
        dfs1(y, x);
        h[x] = (h[x] + 1ll * h[y] * p[siz[y]] % Mod) % Mod;
        siz[x] += siz[y];
    }
    return;
}

void dfs2(int x, int f, int V) {
    g[x] = (h[x] + 1ll * V * p[n - siz[x]] % Mod) % Mod;
    for(auto y:edge[x]) {
        if(y == f) {
            continue;
        }
        dfs2(y, x, (g[x] - 1ll * h[y] * p[siz[y]] % Mod + Mod) % Mod);
    }
    return;
}

//另一种方法

constexpr int N = 60, M = 998244353;
const ull mask = std::mt19937_64(time(nullptr))();

ull shift(ull x) {
```

```
    x ^= mask;
    x ^= x << 13;
    x ^= x >> 7;
    x ^= x << 17;
    x ^= mask;
    return x;
}

std::vector<int> edge[N];
ull sub[N], root[N];
std::map<ull, int> trees;

void getSub(int x) {
    sub[x] = 1;
    for (int i : edge[x]) {
        getSub(i);
        sub[x] += shift(sub[i]);
    }
}

void getRoot(int x) {
    for (int i : edge[x]) {
        root[i] = sub[i] + shift(root[x] - shift(sub[i]));
        getRoot(i);
    }
}
```

可持续化线段树

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

const int MAX_NODE = 40000000;  // 节点池大小
const int MAXM = 1000000 + 5;   // 最大操作数

struct Node {
    int lc, rc;       // 左右子节点索引
    long long val;    // 存储的值
};

Node tree[MAX_NODE];
int rt[MAXM];         // 每个版本的根节点索引
int tot;              // 节点计数器

// 构建线段树
int build(int l, int r, const vector<long long>& arr) {
    int p = ++tot;
    if (l == r) {
        tree[p].val = arr[l];
        return p;
```

```
        }
        int mid = (l + r) >> 1;
        tree[p].lc = build(l, mid, arr);
        tree[p].rc = build(mid + 1, r, arr);
        return p;
    }

    // 更新操作：修改位置pos的值为val
    int update(int now, int l, int r, int pos, long long val) {
        int p = ++tot;
        tree[p] = tree[now];  // 复制原节点
        if (l == r) {
            tree[p].val = val;
            return p;
        }
        int mid = (l + r) >> 1;
        if (pos <= mid) {
            tree[p].lc = update(tree[now].lc, l, mid, pos, val);
        } else {
            tree[p].rc = update(tree[now].rc, mid + 1, r, pos, val);
        }
        return p;
    }

    // 查询操作：获取位置pos的值
    long long query(int now, int l, int r, int pos) {
        if (l == r) {
            return tree[now].val;
        }
        int mid = (l + r) >> 1;
        if (pos <= mid) {
            return query(tree[now].lc, l, mid, pos);
        } else {
            return query(tree[now].rc, mid + 1, r, pos);
        }
    }

    int main() {
        ios::sync_with_stdio(false);
        cin.tie(nullptr);
        int n, m;
        cin >> n >> m;
        vector<long long> arr(n + 1);
        for (int i = 1; i <= n; i++) {
            cin >> arr[i];
        }

        tot = 0;
        rt[0] = build(1, n, arr);  // 构建初始版本

        for (int i = 1; i <= m; i++) {
            int op, v, p;
            long long c;
            cin >> v >> op;
```

```
        if (op == 1) {
            cin >> p >> c;
            rt[i] = update(rt[v], 1, n, p, c);
        } else {
            cin >> p;
            long long ans = query(rt[v], 1, n, p);
            cout << ans << '\n';
            rt[i] = rt[v];  // 新版本是访问版本的复制
        }
    }
    return 0;
}
```

可持续化并查集

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

// 线段树节点结构，用于维护可持久化并查集
struct SegmentTree {
  int lc, rc;   // 左右子节点索引
  int val;      // 存储并查集中节点的父节点
  int rnk;      // 存储并查集中节点的秩（树高度）
};

constexpr int MAXN = 100000 + 5;   // 最大元素数量
constexpr int MAXM = 200000 + 5;   // 最大操作数量

SegmentTree t[MAXN * 2 + MAXM * 40]; // 线段树节点池（可持久化需要大量节点）
int rt[MAXM]; // 存储每个操作后的线段树根节点索引（版本控制）
int n, m, tot; // tot: 线段树节点计数器

// 构建初始线段树
int build(int l, int r) {
  int p = ++tot; // 创建新节点
  if (l == r) {  // 叶子节点：初始父节点为自身，秩为1
    t[p].val = l;
    t[p].rnk = 1;
    return p;
  }
  int mid = (l + r) / 2;
  t[p].lc = build(l, mid);     // 递归构建左子树
  t[p].rc = build(mid + 1, r); // 递归构建右子树
  return p;
}

// 查询节点秩（树高度）
int getRnk(int p, int l, int r, int pos) {
  if (l == r) {
    return t[p].rnk;
```

```
    }
    int mid = (l + r) / 2;
    if (pos <= mid) {
      return getRnk(t[p].lc, l, mid, pos);
    } else {
      return getRnk(t[p].rc, mid + 1, r, pos);
    }
  }

  // 修改节点秩（可持久化）
  int modifyRnk(int now, int l, int r, int pos, int val) {
    int p = ++tot;      // 新建节点
    t[p] = t[now];      // 复制原节点信息
    if (l == r) {
      t[p].rnk = max(t[p].rnk, val); // 更新秩（取最大值保证正确性）
      return p;
    }
    int mid = (l + r) / 2;
    if (pos <= mid) {
      t[p].lc = modifyRnk(t[now].lc, l, mid, pos, val); // 递归修改左子树
    } else {
      t[p].rc = modifyRnk(t[now].rc, mid + 1, r, pos, val); // 递归修改右子树
    }
    return p;
  }

  // 查询节点父节点
  int query(int p, int l, int r, int pos) {
    if (l == r) {
      return t[p].val;
    }
    int mid = (l + r) / 2;
    if (pos <= mid) {
      return query(t[p].lc, l, mid, pos);
    } else {
      return query(t[p].rc, mid + 1, r, pos);
    }
  }

  // 查找节点的根节点（无路径压缩）
  int findRoot(int p, int pos) {
    int f = query(p, 1, n, pos); // 查询当前父节点
    if (pos == f) { // 找到根节点
      return pos;
    }
    return findRoot(p, f); // 递归向上查找
  }

  // 修改节点父节点（可持久化）
  int modify(int now, int l, int r, int pos, int fa) {
    int p = ++tot;      // 新建节点
    t[p] = t[now];      // 复制原节点
    if (l == r) {
      t[p].val = fa;    // 更新父节点
```

```
    return p;
  }
  int mid = (l + r) / 2;
  if (pos <= mid) {
    t[p].lc = modify(t[now].lc, l, mid, pos, fa); // 递归修改左子树
  } else {
    t[p].rc = modify(t[now].rc, mid + 1, r, pos, fa); // 递归修改右子树
  }
  return p;
}

int main() {
  cin.tie(nullptr)->sync_with_stdio(false);
  cin >> n >> m;
  rt[0] = build(1, n); // 初始化版本0：每个元素独立成树

  for (int i = 1; i <= m; i++) {
    int op, a, b;
    cin >> op;

    if (op == 1) { // 合并操作
      cin >> a >> b;
      int fa = findRoot(rt[i - 1], a); // 查找a的根
      int fb = findRoot(rt[i - 1], b); // 查找b的根

      if (fa != fb) {
        // 按秩合并：保持fa是秩较小的树
        if (getRnk(rt[i - 1], 1, n, fa) > getRnk(rt[i - 1], 1, n, fb)) {
          swap(fa, fb);
        }
        // 第一步：将fa的父节点设为fb
        int tmp = modify(rt[i - 1], 1, n, fa, fb);
        // 第二步：更新fb的秩（若两树高度相同则+1）
        rt[i] = modifyRnk(tmp, 1, n, fb, getRnk(rt[i - 1], 1, n, fa) + 1);
      } else {
        // 已在同一集合，版本不变
        rt[i] = rt[i - 1];
      }
    } else if (op == 2) { // 版本回退
      cin >> a;
      rt[i] = rt[a]; // 直接使用历史版本
    } else { // 查询连通性
      cin >> a >> b;
      rt[i] = rt[i - 1]; // 不修改版本
      // 输出是否连通（1/0）
      cout << (findRoot(rt[i], a) == findRoot(rt[i], b)) << '\n';
    }
  }
  return 0;
}
```

欧拉筛

P3383 【模板】线性筛素数 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

每个数都是被自己最小的质数筛掉的。 例如 $45=5*9=3*15$ ，我们希望45只被3筛掉，i循环到9的时候，9最大也就筛到27，随后i%p=0 break了；i到15的时候，`vis[3*15]` 筛掉了45，随后break；这样子 ，所有数都是被自己的最小素数筛掉的。

```
#include<bits/stdc++.h>

using namespace std;
bool vis[100000000];
vector<int> P;
int main(){
    std::ios::sync_with_stdio(0);
    int n,q;
    scanf("%d %d",&n,&q);
    vis[1]=true;
    for(int i=2;i<n;++i){
        if(!vis[i]) P.push_back(i);
        for(auto p:P){
            if(p*i>=n) break;
            vis[p*i]=true;
            if(i%p==0) break;
        }
    }
    int x;
    for(int i=0;i<q;++i){
        scanf("%d",&x);
        printf("%d\n",P[x-1]);
    }
    return 0;
}
```

快速幂

```
ll p = 100003;

long long int quik_power(ll base, ll power) {
    base %= p;
    long long int result = 1;
    while (power > 0) {
        if (power & 1) {
            result = (result * base) % p;   // 分离出当前项并累乘后保存
        }
        power >>= 1;                          // 指数折半
        base = (base * base) % p;           // 底数变其平方
    }
    return result;                          // 返回最终结果
}
```

图论

tarjan

割边

洛谷P1656

```
#include <bits/stdc++.h>
using namespace std;
int maps[151][151];//邻接矩阵，简单易懂
struct Edge {
    int x,y;
} E[5001];//这是存答案的，用邻接表存，应该不用解释
int dfn[151],low[151],n,m,id,cnt,f[151];/*这些数组的含义：
dfn:
{
下标：点编号
内存的值：深度优先搜索时第几个遍历
}
low:
{
下标：点编号
内存的值：这个点能通过它的子孙到达的dfn值最小的点的dfn
}
f:
{
下标：点标号
内存的值：它遍历的上一个点
}
变量的含义：
n:结点个数
m:边个数
id:用于dfn标记
cnt:用于邻接表存图
*/
bool cmp(struct Edge a,struct Edge b) {
    if(a.x==b.x)return a.y<b.y;
    return a.x<b.x;
}//因题目要求，边要排序，要做这道题的人应该都知道cmp
void addEdge(int x,int y) {
    E[++cnt].x=x;
    E[cnt].y=y;
}//addedge函数，存入邻接表
void tarjan(int x) {
    int c=0,y;
    dfn[x]=low[x]=++id;
    for(register int i=1; i<=n; i++) {
        if(!maps[x][i])continue;//首先要有边
        y=i;//处理对象
        if(dfn[y]&&y!=f[x])low[x]=min(low[x],dfn[y]);//如果是它爸爸，割边就没有
```

```
用了，好好理解
        if(!dfn[y]) {//如果找到祖先还有什么用呢
            f[y]=x;//不是祖先就认爸爸
            tarjan(y);//dfs过程
            low[x]=min(low[x],low[y]);//回溯时带着爸爸更新low
            if(low[y]>dfn[x])addEdge(x,y);//是割边，就加入吧
        }
    }
}//tarjan部分，证明在下面
int main() {
    int x,y;
    cin>>n>>m;
    for(register int i=1; i<=m; i++) {
        cin>>x>>y;
        maps[x][y]=maps[y][x]=1;//存边
    }
    for(register int i=1; i<=n; i++) {
        if(!dfn[i])tarjan(i);//tarjan
    }
    sort(E+1,E+cnt2+1,cmp);//sort大法好
    for(register int i=1; i<=cnt; i++) {
        cout<<min(E[i].x,E[i].y)<<' '<<max(E[i].x,E[i].y)<<endl;//输出
    }
    return 0;//程序结束了，证明开始了
}
```

割点

P3388 【模板】割点（割顶） - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)

```
#include<bits/stdc++.h>
#define pb push_back
using namespace std;
const int N=2e4+1e3;

inline int read(){
    int res=0,f=1;char c=getchar();
    for(;!isdigit(c);c=getchar()) if(c=='-') f=-1;
    for(;isdigit(c);c=getchar()) res=(res<<3)+(res<<1)+(c^48);
    return res*f;
}

int n,m,ans;
vector<int> g[N];
bool cut[N];

int dfn[N],low[N],dfncnt;
int st[N],top;
bool inst[N];
void tarjan(int u,int anc){
    dfn[u]=low[u]=++dfncnt;
```

```
    int child=0;
    for(auto v:g[u]){
        if(!dfn[v]){
            tarjan(v,anc),low[u]=min(low[u],low[v]);
            if(low[v]>=dfn[u]&&u!=anc) cut[u]=true;//如果儿子的low值大于等于
dfn，则代表其为儿子与其他非子树点相连的唯一途径
            if(u==anc) child++;
        }
        else low[u]=min(low[u],dfn[v]);
    }
    if(child>=2&&u==anc) cut[u]=true;//对根的处理
}

int main(){
    n=read(),m=read();
    for(int i=1,u,v;i<=m;i++){
        u=read(),v=read();
        g[u].pb(v);g[v].pb(u);
    }
    for(int i=1;i<=n;i++)
        if(!dfn[i]) tarjan(i,i);
    for(int i=1;i<=n;i++)
        if(cut[i]) ans++;
    printf("%d\n",ans);
    for(int i=1;i<=n;i++)
        if(cut[i]) printf("%d ",i);
    return 0;
}
```

Dijkstra

```
#include<bits/stdc++.h>

using namespace std;

const long long INF = numeric_limits<long long>::max();

int main() {
    int N, M;
    cin >> N >> M;

    vector<long long> A(N + 1);
    for (int i = 1; i <= N; i++) {
        cin >> A[i];
    }

    vector<vector<pair<int, long long>>> graph(N + 1);

    for (int j = 0; j < M; j++) {
        int u, v;
        long long b;
```

```
        cin >> u >> v >> b;
        graph[u].emplace_back(v, b);
        graph[v].emplace_back(u, b);
    }

    vector<long long> dist(N + 1, INF);
    dist[1] = A[1];
    priority_queue<pair<long long, int>, vector<pair<long long, int>>,
greater<pair<long long, int>>> pq;
    pq.emplace(A[1], 1);

    while (!pq.empty()) {
        auto [current_weight, u] = pq.top();
        pq.pop();

        if (current_weight > dist[u]) {
            continue;
        }

        for (auto &[v, b] : graph[u]) {
            long long new_dist = current_weight + b + A[v];
            if (new_dist < dist[v]) {
                dist[v] = new_dist;
                pq.emplace(new_dist, v);
            }
        }
    }

    for (int i = 2; i <= N; i++) {
        cout << dist[i] << " ";
    }
    cout << endl;

    return 0;
}
```

好的，这里为你整理了一份C++中数论相关知识的常用公式、算法模板和代码实现。这份指南侧重于算法竞赛和面试中高频出现的内容，代码力求简洁高效。

目录

---

1. 基础模运算

公式：

- $(a + b) \mod m = (a \mod m + b \mod m) \mod m$
- $(a - b) \mod m = (a \mod m - b \mod m + m) \mod m$ (防止负数)
- $(a * b) \mod m = (a \mod m * b \mod m) \mod m$
- $(a / b) \mod m \ne (a \mod m / b \mod m) \mod m$ (需要用逆元)

代码： 加减乘在C++中直接使用 `%` 并处理负数即可。

```
// 假设 mod 是一个已定义的常量，例如 const int mod = 1e9+7;

// 加法
int add(int a, int b) {
    return (a + b) % mod;
}

// 减法，防止出现负数
int sub(int a, int b) {
    return (a - b + mod) % mod;
}

// 乘法
int mul(int a, int b) {
    return (1LL * a * b) % mod; // 1LL 防止溢出
}
```

---

2. 快速幂

公式： $base^{exp} \mod mod$，通过将指数二进制分解来加速计算。

代码：

```
// 迭代法（推荐）
long long fast_pow(long long base, long long exp, long long mod) {
    long long result = 1;
    base %= mod; // 先取模，防止base过大
    while (exp) {
        if (exp & 1) { // 如果当前二进制位为1
            result = (result * base) % mod;
        }
        base = (base * base) % mod; // base平方
        exp >>= 1; // exp右移一位
    }
    return result;
}

// 递归法（易于理解）
long long fast_pow_recursive(long long base, long long exp, long long mod)
{
    if (exp == 0) return 1;
    long long half = fast_pow_recursive(base, exp / 2, mod);
    if (exp % 2 == 0) {
        return (half * half) % mod;
    } else {
        return (((half * half) % mod) * base) % mod;
    }
}
```

---

3. 最大公约数 (GCD) 和最小公倍数 (LCM)

公式：

- $gcd(a, b) = gcd(b, a % b)$ (欧几里得算法)
- $lcm(a, b) = |a * b| / gcd(a, b)$

代码：

```
// 递归版
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a % b);
}

// 迭代版
int gcd_iterative(int a, int b) {
    while (b != 0) {
        int temp = a % b;
        a = b;
        b = temp;
    }
    return a;
}
```

```
// 最小公倍数
// 注意：先除后乘，防止溢出
int lcm(int a, int b) {
    return a / gcd(a, b) * b;
}
```

---

4. 扩展欧几里得算法 (EXGCD)

公式： 求解 $ax + by = gcd(a, b)$ 的一组整数解 (x, y)。常用于求解线性同余方程和逆元。

代码：

```
// 函数返回 gcd(a, b)，并通过引用返回解 x, y
int exgcd(int a, int b, int &x, int &y) {
    if (b == 0) {
        x = 1;
        y = 0;
        return a;
    }
    int g = exgcd(b, a % b, y, x); // 递归，注意这里交换了x, y的位置
    y -= a / b * x;
    return g;
}
```

---

5. 素数（质数）

试除法判定质数

代码：

```
bool is_prime(int n) {
    if (n < 2) return false;
    // 优化：只枚举到 sqrt(n)
    for (int i = 2; i <= n / i; i++) { // 使用 i <= n/i 防止i*i溢出
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

埃氏筛法

用于快速筛选出 [2, n] 范围内的所有素数。

代码：

```cpp
const int MAXN = 1000000;
bool is_prime[MAXN + 1];
vector<int> primes;

void eratosthenes_sieve(int n) {
    fill(is_prime, is_prime + n + 1, true);
    is_prime[0] = is_prime[1] = false;
    // 优化：从 i*i 开始标记，且只筛到 sqrt(n)
    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
            // 注意：从 i*i 开始，因为 2i, 3i, ... (i-1)i 已经被标记过了
            if ((long long)i * i > n) continue; // 防止i*i溢出
            for (long long j = (long long)i * i; j <= n; j += i) {
                is_prime[j] = false;
            }
        }
    }
}
```

线性筛法（欧拉筛）

埃氏筛会将合重复标记（如 12 会被 2 和 3 都标记）。线性筛保证每个合数只被其最小质因子标记，时间复杂度为 $O(n)$。

代码：

```cpp
const int MAXN = 1000000;
bool is_prime[MAXN + 1];
vector<int> primes;

void euler_sieve(int n) {
    fill(is_prime, is_prime + n + 1, true);
    is_prime[0] = is_prime[1] = false;
    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
        }
        // 用已得到的素数 primes[j] 去筛
        for (int j = 0; j < primes.size() && i * primes[j] <= n; j++) {
            is_prime[i * primes[j]] = false;
            // 关键：保证每个合数只被最小质因子筛掉
            if (i % primes[j] == 0) {
                break;
            }
        }
    }
}
```

6. 欧拉函数

$\phi(n)$：小于等于 n 的正整数中与 n 互质的数的个数。

求单个欧拉函数

公式： $n \prod_{p|n}(1 - \frac{1}{p})$，其中 p 是 n 的质因子。

代码：

```
int euler_phi(int n) {
    int ans = n;
    for (int i = 2; i <= n / i; i++) {
        if (n % i == 0) {
            ans = ans / i * (i - 1); // 先除后乘，防止溢出
            while (n % i == 0) n /= i;
        }
    }
    if (n > 1) ans = ans / n * (n - 1);
    return ans;
}
```

线性筛求欧拉函数

在欧拉筛的过程中，同步计算 `[1, n]` 所有数的欧拉函数。

代码：

```
const int MAXN = 1000000;
int phi[MAXN + 1];
bool is_prime[MAXN + 1];
vector<int> primes;

void get_eulers(int n) {
    fill(is_prime, is_prime + n + 1, true);
    is_prime[0] = is_prime[1] = false;
    phi[1] = 1; // 注意 phi[1] = 1
    for (int i = 2; i <= n; i++) {
        if (is_prime[i]) {
            primes.push_back(i);
            phi[i] = i - 1; // 质数的欧拉函数为 i-1
        }
        for (int j = 0; j < primes.size() && i * primes[j] <= n; j++) {
            is_prime[i * primes[j]] = false;
            int p = primes[j];
            if (i % p == 0) {
                // i 包含质因子 p
                phi[i * p] = phi[i] * p; // 公式 1
                break;
            } else {
```

```
                // i 和 p 互质
                phi[i * p] = phi[i] * phi[p]; // 积性函数性质，公式 2
            }
        }
    }
}
```

---

7. 乘法逆元

解决模意义下的除法问题：$(a / b) \mod m = (a * inv(b)) \mod m$。

扩展欧几里得求逆元

求解 $bx \equiv 1 \pmod m$，*即等价于求解 $b*x + m*y = 1$*。要求 $gcd(b, m) = 1$。

代码：

```
int inv_exgcd(int a, int mod) {
    int x, y;
    int g = exgcd(a, mod, x, y);
    if (g != 1) return -1; // 无解
    return (x % mod + mod) % mod; // 调整为非负
}
```

费马小定理求逆元

前提：mod 为素数。 根据费马小定理：如果 mod 是质数，且 b 与 mod 互质，则 $b^{mod-2} \equiv inv(b) \pmod{mod}$。

代码：

```
int inv_fermat(int a, int mod) {
    return fast_pow(a, mod - 2, mod);
}
```

线性求逆元

求 1 到 n 所有数关于 mod 的逆元。

递推公式： $inv[i] = (mod - mod / i) * inv[mod % i] % mod$

代码：

```
const int MAXN = 1000000;
int inv[MAXN + 1];
```

```
void get_invs(int n, int mod) {
    inv[1] = 1;
    for (int i = 2; i <= n; i++) {
        inv[i] = (long long)(mod - mod / i) * inv[mod % i] % mod;
    }
}
```

---

8. 中国剩余定理 (CRT)

求解同余方程组： $\begin{cases} x \equiv a_1 \pmod{m_1} \\ x \equiv a_2 \pmod{m_2} \\ \vdots \\ x \equiv a_n \pmod{m_n} \end{cases}$ ，其中 $m_1, m_2, ..., m_n$ **两两互质**。

代码：

```
// 假设 m[] 两两互质
long long crt(const vector<long long> &a, const vector<long long> &m) {
    long long M = 1;
    for (auto mod : m) M *= mod;

    long long x = 0;
    for (int i = 0; i < a.size(); i++) {
        long long M_i = M / m[i];
        // 求 M_i 在模 m[i] 意义下的逆元
        long long inv_M_i = inv_exgcd(M_i, m[i]); // 使用ExGCD求逆元
        // 也可以用费马小定理，但需要m[i]是质数
        // long long inv_M_i = fast_pow(M_i, m[i]-2, m[i]);

        x = (x + a[i] * M_i % M * inv_M_i % M) % M;
    }
    return (x % M + M) % M;
}
```

---

9. 组合数学

递推预处理组合数

公式： $C_n^m = C_{n-1}^{m} + C_{n-1}^{m-1}$ (杨辉三角)

代码：

```
const int MAXN = 2000; // n 的最大值
int C[MAXN + 1][MAXN + 1];
const int mod = 1e9 + 7;

void init_comb() {
    for (int i = 0; i <= MAXN; i++) {
```

```
            C[i][0] = C[i][i] = 1;
            for (int j = 1; j < i; j++) {
                C[i][j] = (C[i - 1][j] + C[i - 1][j - 1]) % mod;
            }
        }
    }
    // 查询：C[n][m]
```

逆元预处理组合数

当 n 很大（如1e5）而 mod 为素数时，使用逆元公式计算：$C_n^m = \frac{n!}{m!(n-m)!} \mod mod$。

代码：

```
const int MAXN = 100000;
long long fact[MAXN + 1], inv_fact[MAXN + 1]; // 阶乘 和 阶乘的逆元
const int mod = 1e9 + 7;

void init_comb_adv() {
    fact[0] = inv_fact[0] = 1;
    // 预处理阶乘
    for (int i = 1; i <= MAXN; i++) {
        fact[i] = fact[i - 1] * i % mod;
    }
    // 预处理阶乘的逆元 （利用费马小定理）
    inv_fact[MAXN] = fast_pow(fact[MAXN], mod - 2, mod);
    // 逆推，利用 inv_fact[i] = inv_fact[i+1] * (i+1) % mod
    for (int i = MAXN - 1; i >= 1; i--) {
        inv_fact[i] = inv_fact[i + 1] * (i + 1) % mod;
    }
}

long long comb(int n, int m) {
    if (m < 0 || m > n) return 0;
    return fact[n] * inv_fact[m] % mod * inv_fact[n - m] % mod;
}
```

卢卡斯定理

用于求 $C_n^m \mod p$，其中 p 是**素数**，且 n 和 m 可能非常大（远大于 p）。

公式： $C_n^m \equiv C_{n \mod p}^{m \mod p} \cdot C_{n/p}^{m/p} \pmod{p}$

代码：

```
// 需要先实现 comb(n, m, p) 函数，用于求 0<=n,m<p 时的组合数C(n, m) mod p
// 可以用递推法 init_comb(p-1, p) 预处理一个小的组合数表，或者直接用公式+逆元计算（因为n, m < p）
```

```
long long lucas(long long n, long long m, long long p) {
    if (m == 0) return 1;
    // 递归，C(n%p, m%p) * lucas(n/p, m/p, p) % p
    return (comb(n % p, m % p, p) * lucas(n / p, m / p, p)) % p;
}

// 一个简单的、用于Lucas定理内部的小范围Comb函数
long long comb_small(long long n, long long m, long long p) {
    if (m < 0 || m > n) return 0;
    // 这里使用逆元公式计算，因为n, m < p
    long long a = 1, b = 1;
    for (int i = 1; i <= m; i++) {
        a = a * (n - i + 1) % p;
        b = b * i % p;
    }
    return a * fast_pow(b, p - 2, p) % p;
}
// 在 lucas 函数中调用 comb_small
```

莫队

数列分块

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <unordered_map>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int T;
    cin >> T;
    while (T--) {
        int n, m;
        cin >> n >> m;
        vector<int> records(m);
        for (int i = 0; i < m; i++) {
            cin >> records[i];
        }

        unordered_map<int, int> count_map;
        unordered_map<int, int> first_time_map;

        for (int i = 0; i < m; i++) {
            int p = records[i];
            count_map[p]++;
            if (first_time_map.find(p) == first_time_map.end()) {
                first_time_map[p] = i;
```

```
            }
        }

        vector<int> problems;
        for (const auto& kv : count_map) {
            problems.push_back(kv.first);
        }
        int k = problems.size();

        vector<pair<int, int>> list1;
        for (int p : problems) {
            list1.push_back(make_pair(count_map[p], p));
        }
        sort(list1.begin(), list1.end(), [](const pair<int, int>& a, const
pair<int, int>& b) {
            if (a.first != b.first) {
                return a.first > b.first;
            }
            return a.second < b.second;
        });

        vector<pair<int, int>> list2;
        for (int p : problems) {
            list2.push_back(make_pair(first_time_map[p], p));
        }
        sort(list2.begin(), list2.end(), [](const pair<int, int>& a, const
pair<int, int>& b) {
            return a.first < b.first;
        });

        int diff = 0;
        for (int i = 0; i < k; i++) {
            if (list1[i].second != list2[i].second) {
                diff++;
            }
        }
        cout << diff << '\n';
    }
    return 0;
}
```

数列找不同

```
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const int N = 1e5+5;
struct node{
    int l,r,id,index;
};
int cntEle[N];//确保最值包含
```

```
vector<int> arr;
vector<int> ans;
int len;
int sum;//统计重复出现元素的种类个数
void move(int pos,int sign){//考虑左右端点 左移右移 带来的统计变化
    if(sign==1){//新加元素
        if(cntEle[arr[pos]]==1) sum++;//重复出现
        cntEle[arr[pos]]++;
    }else{//退出元素
        cntEle[arr[pos]]--;
        if(cntEle[arr[pos]]==1) sum--;//变为单独出现
    }
}
bool cmp(const node& x,const node& y){//左端点在同一奇数块的区间，右端点按升序排列，反之降序。
    return (x.index^y.index) ? x.index<y.index : (x.index& 1 ) ? x.r<y.r:x.r>y.r;
}
int main(){
    ios::sync_with_stdio(false);
    cin.tie(nullptr);cout.tie(nullptr);
    int n,q;
    cin>>n>>q;
    len=(int) sqrt(n);
    arr.resize(n);
    for(int i=0;i<n;i++)
        cin>>arr[i];
    // cin>>q;
    vector<node> query(q);
    for(int i=0;i<q;i++){
        cin>>query[i].l>>query[i].r;
        query[i].l--;query[i].r--;
        query[i].id=i;
        query[i].index=query[i].l/len;
    }
    sort(query.begin(),query.end(),cmp);
    ans.clear();ans.resize(q,0);
    for(int i=0,l=0,r=-1;i<q;i++){//调整区间
        while (l > query[i].l) move(--l, 1);
        while (r < query[i].r) move(++r, 1);
        while (l < query[i].l) move(l++, -1);
        while (r > query[i].r) move(r--, -1);
        ans[query[i].id]=sum;
    }
    for(int i=0;i<q;i++){
        cout<<(ans[i]==0?"Yes\n":"No\n");
    }
    return 0;
}
```