算法笔记总结

C/C++快速入门

1. 数的范围

标识符	实际范围	实用范围	备注
`int`	-2^{31} ~ $2^{31}-1$ -2147483648 ~ 2147483647	-2×10^9 ~ 2×10^9	绝对值在 10^9 范围内的数都可以定义为 `int` 型
`long long`	-2^{63} ~ $2^{63}-1$	-9×10^{18} ~ 9×10^{18}	`long long` 赋大于 $2^{31}-1$ 的初值，需要在初值后面加上 `LL`
`float`		6 ~ 7 位	尽量不用，四位数乘四位数都不准
`double`		15 ~ 16 位	

```
long long bignum = 123456789012345LL;
```

无穷大的数	十六进制的形式	十进制的形式
$2^{30}-1$ （避免相加超过`int`的情况）	`0x3fffffff`	`1000000000`

2. 常用 `math` 函数

函数名	作用	返回值	备注
`fabs(double x)`	对 `double` 型变量 `x` 取绝对值	返回 `double` 型变量	
`floor(double x)`	对 `double` 型变量 `x` 向下取整	返回 `double` 型变量	
`ceil(double x)`	对 `double` 型变量 `x` 向上取整	返回 `double` 型变量	
`pow(double r,double p)`	对 `double` 型变量 `r` 和 `p` 计算 r^p	返回 `double` 型变量	
`sqrt(double x)`	对 `double` 型变量 `x` 计算算数平方根	返回 `double` 型变量	
`log(double x)`	对 `double` 型变量 `x` 取以自然对数为底的对数	返回 `double` 型变量	不能对任意底数求对数
`round(double x)`	对 `double` 型变量 `x` 进行四舍五入	返回 `double` 型变量	需进行取整才能用 `(int)x`

3. sscanf 与 sprintf 函数

写法	实际	含义
scanf("%d",&a)	scanf(screen,"%d",&a)	把 screen 的内容以 %d 的形式传输至 a 中
printf("%d",a)	printf(screen,"%d",a)	把 a 的内容以 %d 的形式传输至 screen 中
sscanf(str,"%d",&a)		把 str 的内容以 %d 的形式传输至 a 中
sprintf(str,"%d",a)		把 a 的内容以 %d 的形式传输至 str 中

sscanf 示例:

```
int n;
char str[5] = "123";
sscanf(str,"%d",&n); // 将str以%d的形式写到n中
printf("%d\n",n);
// 123

int n;
double db;
char str2[100];

char str[100] = "2048:3.14,hello";
sscanf(str,"%d:%lf,%s",&n,&db,str2);
printf("%d:%lf,%s",n,db,str2);
// 2048:3.140000,hello
```

sprintf 示例:

```
int n = 233;
char str[5];
sprintf(str,"%d",n); // 将n以%d的形式写到str中
printf("%s\n",str);
return 0;
// 233

int n = 2048;
double db = 3.14;
char str2[100] = "hello";

char str[100];
sprintf(str,"%d:%lf,%s",n,db,str2);
printf("%s",str);
// 2048:3.140000,hello
```

4. 浮点数的比较

浮点数直接比较会出现误差，要引入一个小变量

```
const double eps = 1e-8;
```

5. 圆周率π

```
const double pi = acos(-1.0);
```

---

6. 多点测试

没有任何提示，一直输入到结束

```
int a,b;
while(scanf("%d %d",&a,&b) != EOF){
    printf("%d %d\n",a,b);
}
int a,b;
while(cin >> a >> b){
    printf("%d %d\n",a,b);
}
```

有提示里面加判断条件即可

有总数直接运行固定次数次即可

入门模拟数学题

1. 进制转换

将 P 进制的数转换为 Q 进制需要2步：

1. 将 P 进制数 x 转换为十进制数 y
2. 将十进制数 y 转换为Q进制数 z

```
int main(void){
    int x; // 需要转换的数
    int P; // 原进制
    int Q; // 目标进制
    scanf("%d %d %d",&x,&P,&Q);
    int y = 0; // 十进制过渡数
    int product = 1;
    while(x != 0){
        y = y + (x % 10) * product;
        x /= 10;
        product *= P;
    }
    printf("y = %d ",y);

    int z[40]; // 存放转换的数
    int num = 0;
    do{
        z[num++] = y % Q;
```

```
        y /= Q;
    }while(y != 0);

    for(int i=num-1;i>=0;i--){
        printf("%d",z[i]);
    }
    return 0;
}
```

---

2. 字符串 $hash$

将一个字符串映射为一个整数

```
int hashFunc(char s[],int len){
    int id = 0;
    for(int i=0;i<len;i++){
        if(s[i] >= 'A' && s[i] <= 'Z'){
            id = id * 52 + (s[i] - 'A');
        }
        else if(s[i] >= 'a' && s[i] <= 'z'){
            id = id * 52 + (s[i] - 'a');
        }
    }
    return id;
}
```

---

3. 二分查找

查找数组为严格递增序列，二分区间为 $[left, right]$，传入的初值为 $[0, N-1]$

```
int binarySearch(int left,int right,int x){
    int mid;
    while(left <= right){
        mid = left + (right-left) / 2;
        if(a[mid] == x){
            return mid;
        }
        else if(a[mid] > x){
            right = mid - 1;
        }
        else{
            left = mid + 1;
        }
    }
    return -1;
}
```

求序列中第一个大于等于 x 的元素的位置

如果不存在这个数 x，返回它应该插入的位置

```
int lower_bound(int left,int right,int x){
    int mid;
    while(left < right){
        mid = left + (right - left) / 2;
        if(a[mid] >= x){
            right = mid;
        }
        else{
            left = mid + 1;
        }
    }
    return left;
}
```

求序列中第一个大于 x 的元素的位置

如果不存在这个数 x，返回它应该插入的位置

```
int upper_bound(int left,int right,int x){
    int mid;
    while(left < right){
        mid = left + (right - left) / 2;
        if(a[mid] > x){ // 只有这里有区别
            right = mid;
        }
        else{
            left = mid + 1;
        }
    }
    return left;
}
```

---

4. 快速幂

给定三个正整数 a、b、m，求 a^b

```
typedef long long LL;
LL binaryPow(LL a,LL b,LL m){
    if(b == 0){
        return 1;
    }
    if(b % 2 == 1){
        return a * binaryPow(a,b-1,m) % m;
    }
    else{
        LL mul = binaryPow(a,b/2,m);
        return mul * mul % m;
    }
}
```

---

5. 最大公约数

```
int gcd(int a,int b){
    if(b == 0){
        return a;
    }
    return gcd(b, a % b);
}
```

---

6. 分数运算

```
// 分数的表示
struct Fraction{
    int up;
    int down;
};

// 分数的化简
Fraction reduction(Fraction result){
    // 如果分母小于0，则将分子和分母都变成相反数，只允许分子小于0
    if(result.down < 0){
        result.up = -result.up;
        result.down = -result.down;
    }
    // 如果分子为0，令分母等于1
    if(result.up == 0){
        result.down = 1;
    }
    // 如果分子不为0，求最大公约数进行约分
    else{
        int d = gcd(abs(result.up),abs(result.down));
        result.down /= d;
        result.up /= d;
    }
    return result;
}

// 分数的加法
Fraction add(Fraction a,Fraction b){
    Fraction result;
    result.down = a.down * b.down;
    result.up = a.up * b.down + b.up * a.down;
    return reduction(result);
}

// 分数的减法
Fraction minu(Fraction a,Fraction b){
    Fraction result;
    result.down = a.down * b.down;
    result.up = a.up * b.down - b.up * a.down;
    return reduction(result);
}

// 分数的乘法
```

```
Fraction multi(Fraction a,Fraction b){
    Fraction result;
    result.down = a.down * b.down;
    result.up = a.up * b.up;
    return reduction(result);
}

// 分数的除法
Fraction multi(Fraction a,Fraction b){
    Fraction result;
    result.down = a.down * b.up;
    result.up = a.up * b.down;
    return reduction(result);
}

// 分数的输出
void showResult(Fraction r){
    // 首先进行约分
    r = reduction(r);
    // 分母为1则仅输出分子
    if(r.down == 1){
        printf("%d",r.up);
    }
    // 假分数
    else if(abs(r.down) < abs(r.up)){
        printf("%d %d/%d",r.up/r.down,abs(r.up) % r.down,r.down);
    }
    // 真分数
    else{
        printf("%d/%d",r.up/r.down,r.up,r.down);
    }
}
```

---

7. 大整数运算

```
// 大整数的表示
struct bign{
    int d[1000]; // 反向存储
    int len; // 方便随时获取大整数的长度
    bign(){
        memset(d,0,sizeof(d));
        len = 0;
    }
};

// 将整数转化为大整数
bign change(char str[]){
    bign a;
    a.len = strlen(str);
    for(int i=0;i<a.len;i++){
        a.d[i] = str[a.len - i - 1] - '0';
    }
    return a;
}
```

```
// 比较两个bign变量的大小
int compare(bign a,bign b){
    if(a.len > b.len){
        return 1;
    }
    else if(a.len < b.len){
        return -1;
    }
    else{
        for(int i=a.len - 1;i>=0;i--){
            if(a.d[i] > b.d[i]){
                return 1;
            }
            else if(a.d[i] < b.d[i]){
                return -1;
            }
        }
        return 0;
    }
}

// 高精度加法
bign add(bign a,bign b){
    bign c;
    int carry = 0; // 进位
    for(int i=0;i<a.len || i<b.len;i++){
        int temp = a.d[i] + b.d[i] + carry;
        c.d[c.len++] = temp % 10;
        carry = temp / 10;
    }
    if(carry != 0){
        c.d[c.len++] = carry;
    }
    return c;
}

// 高精度减法
bign sub(bign a,bign b){
    bign c;
    for(int i=0;i<a.len || i<b.len;i++){
        if(a.d[i] < b.d[i]){
            a.d[i+1]--;
            a.d[i] += 10;
        }
        c.d[c.len++] = a.d[i] - b.d[i];
    }
    while(c.len-1 >= 1 && c.d[c.len-1] == 0){
        c.len--;
    }
    return c;
}

// 高精度与低精度的乘法
bign multi(bign a,int b){
    bign c;
    int carry = 0;
    for(int i=0;i<a.len;i++){
        int temp = a.d[i] * b + carry;
```

```
        c.d[c.len++] = temp % 10;
        carry = temp / 10;
    }
    while(carry != 0){
        c.d[c.len++] = carry % 10;
        carry /= 10;
    }
    return c;
}

// 高精度与低精度的除法
bign divide(bign a,int b,int &r){
    bign c;
    c.len = a.len;
    for(int i=a.len - 1;i >= 0;i--){
        r = r * 10 + a.d[i];
        if(r < b){
            c.d[i] = 0;
        }
        else{
            c.d[i] = r / b;
            r = r % b;
        }
    }
    while(c.len - 1 >= 1 && c.d[c.len - 1] == 0){
        c.len--;
    }
    return c;
}
```

8. 求质数，质因子分解

Given any positive integer N, you are supposed to find all of its prime factors

Sample Input:

```
97532468
```

Sample Output:

```
97532468=2^2*11*17*101*1291
```

Solution:

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

const int maxn = 100010;

int prime[100010];
int pnum = 0;
```

```
// 普通判断质数
bool isPrime(int n){
    if(n == 1){
        return false;
    }
    for(int i=2;i<=(int)sqrt(1.0*n);i++){
        if(n % i == 0){
            return false;
        }
    }
    return true;
}
void Find_Prime(){
    for(int i=1;i<maxn;i++){
        if(isPrime(i) == true){
            prime[pnum++] = i;
        }
    }
}

// 埃氏筛法求质数表
void Find_Prime(){
    bool judge[100010];
    memset(judge,false,sizeof(judge));
    for(int i=2;i<=100010;i++){
        if(judge[i] == false){
            prime[pnum++] = i;
            for(int j=i+i;j<=100010;j += i){
                judge[j] = true;
            }
        }
    }
}

// 存储质数的结构
struct factor{
    int x; // 分解出的质数
    int cnt; // 分解出的质数次数
}Factor[10];

int main(void){
    Find_Prime();
    int n,num = 0;
    scanf("%d",&n);
    if(n == 1){
        printf("1=1");
    }
    else{
        printf("%d=",n);
        int b = (int)sqrt(1.0*n);
        for(int i=0;i<pnum && prime[i] <= b;i++){
            if(n % prime[i] == 0){
                Factor[num].x = prime[i];
                Factor[num].cnt = 0;
                while(n % prime[i] == 0){
                    Factor[num].cnt++;
                    n /= prime[i];
                }
```

```
                num++;
            }
            if(n == 1){
                break;
            }
        }
        if(n != 1){
            Factor[num].x = n;
            Factor[num++].cnt = 1;
        }

        for(int i=0;i<num;i++){
            if(i != 0){
                printf("*");
            }
            if(Factor[i].cnt == 1){
                printf("%d",Factor[i].x);
            }
            else{
                printf("%d^%d",Factor[i].x,Factor[i].cnt);
            }
        }
    }
    return 0;
}
```

链表

```
// 链表的结构
struct node{
    int data;
    node* next;
};

// 创建链表
node* create(int Array[],int size){
    node *p,*pre,*head;
    head = new node;
    head->next = NULL;
    pre = head;
    for(int i=0;i<size;i++){
        p = new node;
        p->data = Array[i];
        p->next = NULL;
        pre->next = p;
        pre = p;
    }
    return head;
}
// 打印链表
void Print(node* head){
    node* p = head->next;
    while(p != NULL){
        printf("%d ",p->data);
        p = p->next;
```

```
    }
}
// 找到链表中的节点
int findnode(node* head,int num){
    int index = 0;
    node* p = head->next;
    while(p != NULL){
        index++;
        if(p->data == num){
            return index;
        }
        p = p->next;
    }
    return -1;
}
// 将数据插入链表
void insert(node* head,int pos,int x){
    node* p = head;
    for(int i=0;i<pos-1;i++){
        p = p->next;
    }
    node* q = new node;
    q->data = x;
    q->next = p->next;
    p->next = q;
}
// 删除链表中的结点
void del(node* head,int pos){
    node* p = head->next;
    node* pre = head;
    for(int i=0;i<pos-1;i++){
        p = p->next;
        pre = pre->next;
    }
    pre->next = p->next;
    delete(p);
}
```

树

1. 二叉树

```
// 二叉树的存储结构
struct node{
    int data;
    node* lchild;
    node* rchild;
};

// 二叉树新建一个结点
node* newNode(int v){
    node* Node = new node;
    Node->data = v;
    Node->lchild = Node->rchild = NULL;
    return Node;
```

```
}

// 根据值在二叉树中搜索并替换值
void search(node* root,int x,int newdata){
    if(root == NULL){
        return;
    }
    if(root->data == x){
        root->data = newdata;
    }
    search(root->lchild,x,newdata);
    search(root->rchild,x,newdata);
}

// 向二叉树中插入结点
void insert(node* &root,int x){
    if(root == NULL){
        root = newNode(x);
        return;
    }
    if(1){
        insert(root->lchild,x);
    }
    else{
        insert(root->rchild,x);
    }
}

// 根据数组创建二叉树
node* Create(int data[],int n){
    node* root = NULL;
    for(int i=0;i<n;i++){
        insert(root,data[i]);
    }
    return root;
}

// 二叉树的前序遍历
void preorder(node* root){
    if(root == NULL){
        return;
    }
    printf("%d",root->data);
    preorder(root->lchild);
    preorder(root->rchild);
}

// 二叉树的中序遍历
void inorder(node* root){
    if(root == NULL){
        return;
    }
    inorder(root->lchild);
    printf("%d",root->data);
    inorder(root->rchild);
}

// 二叉树的后序遍历
```

```
void postorder(node* root){
    if(root == NULL){
        return;
    }
    postorder(root->lchild);
    postorder(root->rchild);
    printf("%d",root->data);
}

// 二叉树的层序遍历
void layerorder(node* root){
    queue<node*> q;
    q.push(root);
    while(!q.empty()){
        node* now = q.front();
        q.pop();
        printf("%d ",now->data);
        if(now->lchild != NULL){
            q.push(now->lchild);
        }
        if(now->rchild != NULL){
            q.push(now->rchild);
        }
    }
}

// 根据后序遍历和中序遍历重建二叉树
Node* Createtree(int postL,int postR,int inL,int inR){
    if(postL > postR){
        return NULL;
    }
    Node* root = new Node;
    root->data = post[postR];
    int k;
    for(k = inL;k<=inR;k++){
        if(in[k] == root->data){
            break;
        }
    }
    int num = k - inL;
    root->lchild = Createtree(postL,postL + num - 1,inL,k-1);
    root->rchild = Createtree(postL + num,postR-1,k+1,inR);
    return root;
}
```

---

2. 树

```
// 树的存储结构
struct node{
    int data;
    int layer;
    vector<int> child;
}Node[100000];
int index = 0;
```

```
// 定义新结点
int newNode(int v){
    Node[index].data= v;
    Node[index].child.clear();
    return index++;
}

// 树的先根遍历
void preOrder(int root){
    printf("%d ",Node[root].data);
    for(int i=0;i<Node[root].child.size();i++){
        preOrder(Node[root].child[i]);
    }
}

// 树的层序遍历
void layerOrder(int root){
    queue<int> q;
    q.push(root);
    Node[root].layer = 0;
    while(!q.empty()){
        int now = q.front();
        printf("%d ",Node[now].data);
        q.pop();
        for(int i=0;i<Node[now].child.size();i++){
            int child = Node[now].child[i];
            Node[child].layer = Node[now].layer + 1;
            q.push(child);
        }
    }
}
```

---

3. 二叉查找树

```
// 存储结构
struct node{
    int data;
    node* lchild;
    node* rchild;
};

// 搜索结点
void search(node* root,int x){
    if(root == NULL){
        printf("Failed!");
    }
    if(root->data== x){
        printf("OK");
    }
    else if(root->data < x){
        search(root->rchild,x);
    }
    else{
        search(root->lchild,x);
    }
```

```
    return;
}

// 插入结点
void insert(node* &root,int x){
    if(root == NULL){
        root = new node;
        root->lchild = NULL;
        root->rchild = NULL;
        root->data = x;
        return;
    }
    if(root->data == x){
        return;
    }
    else if(x < root->data){
        insert(root->lchild,x);
    }
    else{
        insert(root->rchild,x);
    }
}


node* findmax(node* root){
    while(root->rchild != NULL){
        root = root->rchild;
    }
    return root;
}


node* findmin(node* root){
    while(root->lchild != NULL){
        root = root->lchild;
    }
    return root;
}


void deletenode(node* &root,int x){
    if(root == NULL){
        return;
    }
    if(root->data == x){
        if(root->lchild == NULL && root->rchild == NULL){
            root = NULL;
        }
        else if(root->lchild != NULL){
            node* pre = findmax(root->lchild);
            root->data = pre->data;
            deletenode(root->lchild,pre->data);
        }
        else{
            node* pre = findmin(root->rchild);
            root->data = pre->data;
            deletenode(root->rchild,pre->data);
        }
```

```
    }
    else if(x < root->data){
        deletenode(root->lchild,x);
    }
    else{
        deletenode(root->rchild,x);
    }
}
```

---

4. 并查集

```
int father[10];

int findfather(int x){
    int a = x;
    while(x != father[x]){
        x = father[x];
    }
    while(a != father[a]){
        int z = a;
        a = father[a];
        father[z] = x;
    }
    return x;
}
void Union(int a,int b){
    int faA = findfather(a);
    int faB = findfather(b);
    if(faA != faB){
        father[faA] = faB;
    }
}
int main(void){
    int N;
    for(int i=1;i<=N;i++){
        father[i] = i;
    }
}
```

图

1. 最短路径

```
#include <bits/stdc++.h>
using namespace std;
const int maxv = 1000;
const int inf = 1000000000;

int n,G[maxv][maxv];
int d[maxv];
bool vis[maxv];
```

```
void Dijkstra(int s){
    memset(vis,false,sizeof(vis));
    fill(d,d+maxv,inf);
    for(int i=0;i<n;i++){
        int u = -1,MIN = inf;
        for(int j=0;j<n;j++){
            if(vis[j] == false && d[j] < MIN){
                u = j;
                MIN = d[j];
            }
        }
        if(u == -1){
            return;
        }
        vis[u] = true;
        for(int v=0;v<n;v++){
            if(vis[v] == false && G[u][v] != inf && d[u] + G[u][v] < d[v]){
                d[v] = d[u] + G[u][v];
            }
        }
    }
}


struct Node{
    int v,dis;
};

vector<Node> Adj[maxv];
void Dijkstra2(int s){
    memset(vis,false,sizeof(vis));
    fill(d,d+maxv,inf);
    for(int i=0;i<n;i++){
        int u=-1,MIN=inf;
        for(int j=0;j<n;j++){
            if(vis[j] == false && d[j] < MIN){
                u = j;
                MIN = d[j];
            }
        }
        if(u == -1){
            return;
        }
        vis[u] = true;
        for(int j=0;j<Adj[u].size();j++){
            int v = Adj[u][j].v;
            if(vis[v] == false && d[u] + Adj[u][j].dis < d[v]){
                d[v] = d[u] + Adj[u][j].dis;
            }
        }
    }
}

int pre[maxv];
void DFS(int s,int v){
    if(v == s){
        printf("%d\n",s);
```

```
            return;
        }
        DFS(s,pre[v]);
        printf("%d\n",v);
    }
```

线段树

1. P3372 【模板】线段树 1

题目描述

已知一个数列，需要进行下面两种操作：

1. 将某区间每一个数加上 k。
2. 求出某区间每一个数的和。

输入格式

第一行包含两个整数 n,m，分别表示该数列数字的个数和操作的总个数。

第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。

接下来 m 行每行包含 3 或 4 个整数，表示一个操作，具体如下：

1. $1\ x\ y\ k$：将区间 $[x,y]$ 内每个数加上 k。
2. $2\ x\ y$：输出区间 $[x,y]$ 内每个数的和。

输出格式

输出包含若干行整数，即为所有操作 2 的结果。

输入

```
5 5
1 5 4 2 3
2 2 4
1 2 3 2
2 3 4
1 1 5 1
2 1 4
```

输出

```
11
8
20
```

解答

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

```
const int maxn = 100010;

// 线段树结构
struct SegmentTree{
    int l,r; // 区间的左右端点
    long long sum,add; // sum存储求和结果，add懒标记
}tree[4*maxn]; // 开四倍大小

int w[maxn]; // 原始数据

int n,m; // 题目给的总数和询问次数

// 自底向上更新信息
void update(int node){
    // 结点的总和 = 左孩子的总和 + 右孩子的总和
    tree[node].sum = tree[node * 2].sum + tree[node * 2 + 1].sum;
}

// 懒标记
void lazy_tag(int node){
    // 左孩子结点
    int leftnode = node * 2;
    // 右孩子结点
    int rightnode = node * 2 + 1;
    // 如果懒标记不为0
    if(tree[node].add){
        // 将懒标记传递给左孩子
        tree[leftnode].add += tree[node].add;
        // 左孩子的总和 = 左孩子的总和 + 传递下来的懒标记 * 左孩子的区间 （每一个都要加）
        tree[leftnode].sum += (long long)(tree[leftnode].r - tree[leftnode].l + 1) *
tree[node].add;
        // 将懒标记传递给右孩子
        tree[rightnode].add += tree[node].add;
        // 右孩子的总和 = 右孩子的总和 + 传递下来的懒标记 * 右孩子的区间 （每一个都要加）
        tree[rightnode].sum += (long long)(tree[rightnode].r - tree[rightnode].l + 1) *
tree[node].add;
        // 懒标记置为0
        tree[node].add = 0;
    }
    return;
}

// 建立线段树
void buildtree(int node,int start,int end){
    // 结点信息的初始化
    tree[node].l = start;
    tree[node].r = end;
    tree[node].add = 0;
    // start == end 说明找到了叶子结点
    if(start == end){
        // 叶子结点的和是它本身
        tree[node].sum = w[start];
        return;
    }
    else{
        int mid = (start + end) / 2;
        // 递归建立左子树
        buildtree(node * 2,start,mid);
```

```
        // 递归建立右子树
        buildtree(node * 2 + 1,mid+1,end);
        // 更新信息
        update(node);
    }
}

//单点查询
int ask_point(int node,int x){
    // 找到了叶子结点，返回叶子结点的值
    if(tree[node].l == tree[node].r){
        return tree[node].sum;
    }
    // 将懒标记下传
    lazy_tag(node);
    int mid=(tree[node].l + tree[node].r) / 2;
    // 向左孩子方向继续寻找
    if(x <= mid){
        ask_point(node * 2,x);
    }
    // 向右孩子方向继续寻找
    else{
        ask_point(node*2 + 1,x);
    }
}

// 单点修改
void change_point(int node,int x,int y){
    // 找到了叶子结点，更改掉叶子结点的值
    if(tree[node].l == tree[node].r){
        tree[x].sum = y;
        return;
    }
    // 将懒标记下传
    lazy_tag(node);
    int mid = (tree[node].l + tree[node].r) / 2;
    if(x <= mid){
        change_point(node * 2,x,y);
    }
    else{
        change_point(node * 2 + 1,x,y);
    }
    // 自底向上更新信息
    update(node);
}

// 区间查询
long long ask_interval(int node,int start,int end){
    // 完全被包含直接返回总和
    if(tree[node].l >= start && tree[node].r <= end){
        return tree[node].sum;
    }
    // 不被包含
    else if(tree[node].r < start || tree[node].l > end){
        return 0;
    }
    else{
        // 更新懒标记
```

```
        lazy_tag(node);
        int mid = (tree[node].l + tree[node].r) / 2;
        long long res = 0;
        if(start <= mid){
            res += ask_interval(node * 2,start,end);
        }
        if(end > mid){
            res += ask_interval(node * 2 + 1,start,end);
        }
        return res;
    }
}

// 区间更新
void change_interval(int node,int start,int end,int val){
    // 完全被包含
    if(tree[node].l >= start && tree[node].r <= end){
        // 更改懒标记
        tree[node].add += val;
        // 更新结点的总和 = 结点的区间 * 更新的值
        tree[node].sum += (long long)(tree[node].r - tree[node].l + 1) * val;
    }
    // 不被包含
    else if(tree[node].r < start || tree[node].l > end){
        return;
    }
    // 不完全被包含
    else{
        lazy_tag(node); // 向下操作，先更新懒标记
        int mid = (tree[node].l + tree[node].r) / 2;
        // 操作左半边区间
        if(start <= mid){
            change_interval(node * 2,start,end,val);
        }
        // 操作右半边区间
        if(end > mid){
            change_interval(node * 2 + 1,start,end,val);
        }
        // 更新信息
        update(node);
    }
}


int main(void){
    scanf("%d %d",&n,&m);
    for(int i=1;i<=n;i++){
        scanf("%d",&w[i]);
    }
    buildtree(1,1,n);
    while(m--){
        int kind,x,y,k;
        scanf("%d",&kind);
        if(kind == 1){
            scanf("%d %d %d",&x,&y,&k);
            change_interval(1,x,y,k);
        }
        else{
```

```
            scanf("%d %d",&x,&y);
            printf("%lld\n",ask_interval(1,x,y));
        }
    }
    return 0;
}
```

2. P3373 【模板】线段树 2

题目描述

已知一个数列，需要进行下面两种操作：

1. 将某区间每一个数乘上 x
2. 将某区间每一个数加上 x
3. 求出某区间每一个数的和

输入格式

第一行包含三个整数 n,m,p，分别表示该数列数字的个数、操作的总个数和模数。

第二行包含 n 个用空格分隔的整数，其中第 i 个数字表示数列第 i 项的初始值。

接下来 m 行每行包含若干个整数，表示一个操作，具体如下：

1. $1\ x\ y\ k$：将区间 $[x,y]$ 内每个数乘上 k。
2. $2\ x\ y\ k$：将区间 $[x,y]$ 内每个数加上 k。
3. $3\ x\ y$：输出区间 $[x,y]$ 内每个数的和对 p 取模所得的结果。

输出格式

输出包含若干行整数，即为所有操作 3 的结果。

输入

```
5 5 38
1 5 4 2 3
2 1 4 1
3 2 5
1 2 4 2
2 3 5 5
3 1 4
```

输出

```
17
2
```

解答

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

```
const int maxn = 100010;

struct SegmentTree{
    int l,r;
    long long v,add,mul; // v为结果，add存储和的懒标记，mul存储积的懒标记
}tree[4*maxn];

long long w[maxn];
int n,m,p; // p是余数

// 自底向上更新信息
void update(int node){
    tree[node].v = (tree[node * 2].v + tree[node * 2 + 1].v) % p;
}

// 懒标记
void lazy_tag(int node){
    int leftnode = node * 2;
    int rightnode = node * 2 + 1;

    int mid = (tree[node].l + tree[node].r) / 2;

    // 更新结果
    tree[leftnode].v = (tree[leftnode].v * tree[node].mul + tree[node].add * (mid -
tree[node].l + 1)) % p;
    tree[rightnode].v = (tree[rightnode].v * tree[node].mul + tree[node].add *
(tree[node].r - mid)) % p;

    // 更新积的懒标记
    tree[leftnode].mul = (tree[leftnode].mul * tree[node].mul) % p;
    tree[rightnode].mul = (tree[rightnode].mul * tree[node].mul) % p;

    // 更新和的懒标记
    tree[leftnode].add = (tree[leftnode].add * tree[node].mul + tree[node].add) % p;
    tree[rightnode].add = (tree[rightnode].add * tree[node].mul + tree[node].add) % p;

    // 懒标记初始化
    tree[node].mul = 1;
    tree[node].add = 0;

    return;
}

// 建立线段树
void buildtree(int node,int start,int end){
    tree[node].l = start;
    tree[node].r = end;
    tree[node].add = 0;
    tree[node].mul = 1;
    if(start == end){
        tree[node].v = w[start];
        tree[node].v %= p;
    }
    else{
        int mid = (start + end) / 2;
        // 递归建立左子树
        buildtree(node * 2,start,mid);
        // 递归建立右子树
```

```
        buildtree(node * 2 + 1,mid+1,end);
        update(node);
    }
}

// 区间查询
long long ask_interval(int node,int start,int end){
    if(tree[node].l >= start && tree[node].r <= end){
        return tree[node].v % p;
    }
    else if(tree[node].r < start || tree[node].l > end){
        return 0;
    }
    else{
        lazy_tag(node);
        int mid = (tree[node].l + tree[node].r) / 2;
        long long res = 0;
        if(start <= mid){
            res += ask_interval(node * 2,start,end);
        }
        if(end > mid){
            res += ask_interval(node * 2 + 1,start,end);
        }
        return res % p;
    }
}

// 区间乘法修改
void change_interval_mul(int node,int start,int end,long long val){
    // 完全被包含
    if(tree[node].l >= start && tree[node].r <= end){
        tree[node].v = (tree[node].v * val) % p;
        tree[node].mul = (tree[node].mul * val) % p;
        tree[node].add = (tree[node].add * val) % p;
        return;
    }
    // 不被包含
    else if(tree[node].r < start || tree[node].l > end){
        return;
    }
    // 不完全被包含
    else{
        lazy_tag(node); // 向下操作，先更新懒标记
        int mid = (tree[node].l + tree[node].r) / 2;
        // 操作左半边区间
        if(start <= mid){
            change_interval_mul(node * 2,start,end,val);
        }
        // 操作右半边区间
        if(end > mid){
            change_interval_mul(node * 2 + 1,start,end,val);
        }
        update(node);
    }
}

// 区间加法修改
void change_interval_add(int node,int start,int end,long long val){
```

```
    // 完全被包含
    if(tree[node].l >= start && tree[node].r <= end){
        tree[node].v = (tree[node].v + (tree[node].r - tree[node].l + 1) * val) % p;
        tree[node].add = (tree[node].add + val) % p;
        return;
    }
    // 不被包含
    else if(tree[node].r < start || tree[node].l > end){
        return;
    }
    // 不完全被包含
    else{
        lazy_tag(node); // 向下操作，先更新懒标记
        int mid = (tree[node].l + tree[node].r) / 2;
        // 操作左半边区间
        if(start <= mid){
            change_interval_add(node * 2,start,end,val);
        }
        // 操作右半边区间
        if(end > mid){
            change_interval_add(node * 2 + 1,start,end,val);
        }
        update(node);
    }
}

int main(void){
    scanf("%d %d %d",&n,&m,&p);
    for(int i=1;i<=n;i++){
        scanf("%lld",&w[i]);
    }
    buildtree(1,1,n);
    while(m--){
        int kind,x,y;
        long long k;
        scanf("%d",&kind);
        if(kind == 1){
            scanf("%d %d %lld",&x,&y,&k);
            change_interval_mul(1,x,y,k);
        }
        else if(kind == 2){
            scanf("%d %d %lld",&x,&y,&k);
            change_interval_add(1,x,y,k);
        }
        else{
            scanf("%d %d",&x,&y);
            printf("%lld\n",ask_interval(1,x,y));
        }
    }
    return 0;
}
```