目录

基础

枚举 折半 搜索 模拟 打表 公式 二分 尺取 构造 离散化 染色

异或 二进制位序

逆向 [Two buttons]

二分精度处理 取 eps 小于 1e 题目要求保留位数*2+1，或二分 100 次

测试顺序 变量边界 逻辑边界 乱序 极限 特殊
“大方的程序比 dirty-but-work 要好一些”

扫描 顺向 逆向 旗帜
单调枚举 [扫雷]

二元对 左-1 右正 [WF-Comma]
环的处理

区间查询
- **区间和** 树状数组 线段树
- **静态区间最值查询** 稀疏表
- **区间和是否整除模** 考察前缀和

中位数定理 [输油管道问题]

自然数列
- [Hybrid Crystal] 取数列中的元素，如果可以凑出[1…sum]区间中的任何一个数，向数列加入新数 x<=sum+1，可以凑出[1…sum+x]中的任何一个数。

斐波那契数列 斐波那契数列第 n 项

$$\begin{pmatrix} f(n) \\ f(n+1) \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n \begin{pmatrix} f(0) \\ f(1) \end{pmatrix} \quad 或 \quad \begin{bmatrix} f(n) & f(n+1) \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} f(0) & f(1) \\ 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^n$$

通项公式 $a_n = \frac{1}{\sqrt{5}}\left[(\frac{1+\sqrt{5}}{2})^n - (\frac{1-\sqrt{5}}{2})^n\right]$

常系数齐次线性递推

已知$f_x = a_0 f_{x-1} + a_1 f_{x-2} + \cdots + a_{n-1} f_{x-n}$和$f_0, f_1, \dots, f_{n-1}$，给定$t$，求$f_t$

$$构造矩阵A = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ 0 & 0 & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \\ a_{n-1} & a_{n-2} & a_{n-3} & \dots & a_0 \end{bmatrix}, B = \begin{pmatrix} f_{x-n} \\ f_{x-n+1} \\ \cdots \\ f_{x-2} \\ f_{x-1} \end{pmatrix}$$

字符串散列

简易 利用 unorder_map<string, int>为字符串编号

字符串匹配 KMP

输入模式串 p，文本串 s，在 O(N+M)内求解模式串在在文本串内的所有匹配位置的下标。注意文本串中匹配的模式串可以重叠。

```
int pre[maxN];
char s[maxN], p[maxN];  // 文本串、模板串

void prepare() {
    fill(pre, pre+maxN, -1);
    for (int i=1, j=-1; p[i]; ++i)
    {
        while (j>=0 && p[i] != p[j+1]) j = pre[j];
        if (p[i] == p[j+1]) ++j;
        pre[i] = j;
    }
}
```

```
void kmp(vector<int> &match)
{
    match.clear(); prepare();
    for (int i=0, j=-1; s[i]; ++i) {
        while (j>=0 && s[i] != p[j+1]) j = pre[j];
        if (s[i] == p[j+1]) ++j;
        if (!p[j+1]) {  // 匹配成功
            match.push_back(i-j);
            j = pre[j];
        }
    }
}
```

最长回文子串 Manacher

优化暴力匹配

```
// 1-based: scanf("%s", str+1);
int solve()
{
    int i = 0, mx = 1; str[0] = '*';
    while (str[i])
    {
        int p = i;
        while (str[i+1] == str[i]) ++i;
        int q = i;  // q之前不可能有更强的回文中心

        while (str[q-1]==str[p+1]) --q, ++p;
        mx = max(mx, q-p+1);

        ++i;
    }
    return mx;
}
```

Manacher 紧凑实现

```
// str - 字符串
// len - 储存字符串的回文半径, 空间2n-1
// n   - 字符串的长度
void manacher(char str[], int len[], int n)
{
    len[0] = 1;
    for (int i = 1, j = 0; i < (n<<1)-1; ++i)
    {
        int p = i>>1, q = i-p, r = ((j+1)>>1)+len[j]-1;
        len[i] = r<q ? 0 : min(r-q+1, len[(j<<1)-i]);
        while (p>len[i]-1 && q+len[i]<n && str[p-len[i]]==str[q+len[i]])
            ++len[i];
        if (q+len[i]-1 > r)
            j = i;
    }
}
```

经验

表达式求值

前缀表达式求值

从右至左扫描前缀表达式，遇到数字入栈，遇到操作符弹出栈顶元素运算（栈顶 op 次顶），将结果入栈。

中缀表达式转前缀

1. 初始化运算符栈 S1，中间结果栈 S2
2. 从右至左扫描中缀表达式
   a) 遇到数字时入栈 S2
   b) 遇到运算符时比较 S1 栈顶的优先级，
      i) 若 S1 为空或为')'，运算符入栈
      ii) 若运算符优先级比 S1 栈顶高或相等，运算符入栈
      iii) 弹出 S1 栈顶的运算符至 S2，回到 b)
   c) 遇到括号时，
      i) 右括号入栈
      ii) 若为左括号，弹出 S1 中的运算符，并压入 S2，直至遇到右括号为止，丢弃左右括号
3. 将 S1 中剩余的运算符弹出并压入 S2
4. 弹出 S2 元素

贪心

基本思路

贪心操作能为后续操作提供便利

假定某方案是最优解，通过贪心操作可以使比最优解更优秀的解出现，或最优解可转化为贪心解

区间贪心问题

活动安排问题

若干活动占用左闭右开的时间区间，在活动时间不重叠的情况下选择尽可能多的活动：**右端点越小的区间优先**（为后续区间让出空间）

活动安排问题 2

若干活动占用左闭右开的时间区间，同一个教室安排的活动不能重叠，在使用教室尽可能少的情况下安排所有活动：**考虑活动在时间轴上的厚度**

一元贪心

[排队接水]

二元贪心

独木舟问题

若干人乘若干独木舟，独木舟有载重限制且只能乘坐两人。安排乘坐方案，使占用的独木舟数量最少：**最轻与最终若能同乘则同乘**（极端化，最优解可转化）

任务执行顺序

若干任务，第 i 个任务计算时占用 R[i]空间，完成计算后储存结果占用 O[i]空间（R[i]>O[i]）。安排任务，使占用的总空间尽可能少 => 设有整数 N，第 i 个操作时 N 减 a[i]加 b[i]，安排操作顺序，在操作中不能出现负数的情况下 N 尽可能小：**b[i]非递增排序** 任何可行方案不优于按 b[i]非递增排序时的方案（最优解可转化）

动态规划

树塔 矩阵取数 双向矩阵取数

$$dp[step+1][x1][x2] = \max\{dp[step][x1'][x2']\} + v[\ldots]$$

最大子段和 最大子矩阵和 循环数组最大子段和（总和 – 最小子段和）

正整数分组

$$dp[i][j] = dp[i-1][\,|j-a[i]|\,] \ or\ dp[i-1][j+a[i]]$$

或转换为背包问题，背包容量 sum/2

子序列的个数

$$dp[i] = \begin{cases} dp[i-1]*2 & 若a[i]未出现 \\ dp[i-1]*2-dp[j-1] & 若a[i]最近在j位置出现 \end{cases}$$

最长公共子序列 LCS

$$dp[i][j] = \begin{cases} dp[i-1][j-1]+1 & 若x[i]=y[j] \\ max\{dp[i][j-1], dp[i-1][j]\} & 若x[i]\neq y[j] \end{cases}$$

编辑距离 $dp[i][j] = min\begin{cases} dp[i-1][j-1]+same(i,j) & dp[0][0]=0 \\ dp[i-1][j]+1 & dp[i][0]=i \\ dp[i][j-1]+1 & dp[0][j]=j \end{cases}$

最长单增子序列 LIS $\begin{cases} dp[len] = min\{tail\}, & O(nlogn) \\ dp[i] = max\{dp[j]\}+1, & O(n^2) \end{cases}$

一维

二维

石子归并 $dp[i][j] = \max_{i\le k\le j}\{dp[i][k] + dp[k][j]\} + \sum_{i\le p\le j} w[p]$

三维

[CSA-Two Rows] $dp[r][c][turn]$
Turn 选择(r,c)的玩家
Dp 从(r,c)到终点的总花费

优化

改进状态表示

四边形不等式

斜率优化

背包问题

01 背包
完全背包 与 01 背包容量遍历的顺序相反
多重背包 二进制分拆
混合背包 对属于不同背包的物品，使用对应解决背包问题的状态转移。

数据结构

分数 Fraction

Numerator 分子 Denominator 分母
构造函数接受分子 num 和分母 den 作为参数，确保符号在分子上集中，并且断言分母不为零，然后进行约分。

高精度整数

简易

堆

堆排序 O(nlogn)

```cpp
long long arr[maxN];  // 待排序数组，0-based

// 下沉操作
void perc_down(long long *arr, int i, int n)
{
    auto lson = [](int x){ return 2*x+1; };
    auto rson = [](int x){ return 2*x+2; };
    while (true)
    {
        int l = lson(i), r = rson(i);
        if (l<n) {
            int mx = (r<n && arr[r]>arr[l] ? r : l);
            if (arr[i] < arr[mx]) {
                swap(arr[i], arr[mx]);
                i = mx;
            }
            else break;
        }
        else break;
    }
}

// 堆排序
void heap_sort(long long *arr, int n)
{
    for (int i = n-1; i>=0; --i)  // 创建堆
    {
        perc_down(arr, i, n);
    }
    for (int i = n-1; i; --i)  // 输出排序结果
    {
        swap(arr[0], arr[i]);
        perc_down(arr, 0, i);
    }
}
```

稀疏表 O(nlogn)

动态查询序列在区间[L, R]上的最值。$st[i][j]$，自第 i 个元素开始连续 2^j 次方的元素的最值。

```cpp
vector<int> num;  // 维护num序列的最值
int preLog2[maxN];
int st[maxN][32];

// 建立preLog2和st
void init()
{
    int n=num.size();

    preLog2[1] = 0;
    for (int i=2; i<=n; ++i) {
        preLog2[i] = preLog2[i-1];
        if (1<<(preLog2[i]+1) == i) {
            ++preLog2[i];
        }
    }
    for (int i=n-1; i>=0; --i) {
        st[i][0] = num[i];
        for (int j=1; i+(1<<j)-1<n; ++j) {
            st[i][j] = min(st[i][j-1],
                           st[i+(1<<(j-1))][j-1]);
        }
    }
}

// 查询区间[x, y]上的最值
int query(int x, int y)
{
    int k = preLog2[y-x+1];
    return min(st[x][k], st[y-(1<<k)+1][k]);
}
```

并查集

[圆环出列] [Market] 先祖节点出列

带权并查集 维护额外信息，表示当前节点与先祖节点的关系

[食物链] 当前节点与先祖节点形成向量三角形关系

```cpp
// POJ-1182

int n;
int fa[50500], rk[50500];
// rk[x] = r(x->find(x)): 0-同类 1-捕食者 2-被捕食者

void init() {
    for (int i = 0; i <= n; ++i) fa[i] = i, rk[i] = 0;
}

// 寻找x的先祖节点，并维护rk
int find(int x)
{
    if (fa[x] == x) return x;
    int old = fa[x]; fa[x] = find(fa[x]);
    rk[x] = (rk[x] + rk[old]) % 3;
    return fa[x];
}

// 利用rk信息检查“一句话”是否合理
bool check(int a, int b, int r)
{
    if (max(a, b) > n) return false;
    if (a==b && r!=0) return false;
    if (find(a) != find(b)) return true;
    return rk[a] == (r+rk[b])%3;
}

// 合并集合，并维护rk
void merge(int a, int b, int r) {
    int ra = find(a), rb = find(b);
    if (ra == rb) return;
    fa[ra] = rb; rk[ra] = (r+rk[b]-rk[a]+3)%3;
}

int main()
```

```
{
	int k; scanf("%d%d", &n, &k);
	{
		init(); int ans = 0;
		while (k--) {
			int r, x, y; scanf("%d%d%d", &r, &x, &y);
			if (!check(x, y, r-1)) ++ans;
			else merge(x, y, r-1);
		}
		printf("%d\n", ans);
	}
}
```

[How many answers are wrong?] 当前节点与先祖节点形成数值差异关系

线段树 Segment Tree

点修改 点覆盖

根节点为 1 叶子节点 leaf 数组

```
#define lson(x) ((x)<<1)
#define rson(x) ((x)<<1|1)
int leaf[maxN];  // 记录叶子节点的索引
struct SegmentTreeNode{
	int l, r;
	LL val, vmx, vmn;
} node[maxN<<2];  // 根节点为1, 占据四倍序列长度空间

// build(1, 1, n, arr)
// 使用arr数组提供的初值, 以1为根节点, 建立覆盖区间[1, n]的线段树
void build(int rt, int l, int r, LL *arr)
{
	node[rt].l = l, node[rt].r = r;
	if (l==r) {
		leaf[l] = rt;
		node[rt].val = node[rt].vmx = node[rt].vmn = arr[l];
	}
	else {
		int mid = (l+r)/2;
		build(lson(rt), l, mid, arr);
		build(rson(rt), mid+1, r, arr);
		node[rt].val = node[lson(rt)].val + node[rson(rt)].val;
		node[rt].vmx = max(node[lson(rt)].vmx, node[rson(rt)].vmx);
		node[rt].vmn = min(node[lson(rt)].vmn, node[rson(rt)].vmn);
	}
}

void pushup(int rt) {
	while (rt>>=1) {
		node[rt].val = node[lson(rt)].val + node[rson(rt)].val;
		node[rt].vmx = max(node[lson(rt)].vmx, node[rson(rt)].vmx);
		node[rt].vmn = min(node[lson(rt)].vmn, node[rson(rt)].vmn);
	}
}

// modify(leaf[x], val)
// 将叶子x节点修改为val
void modify(int rt, LL val) {
	node[rt].val = node[rt].vmx = node[rt].vmn = val;
	pushup(rt);
}

// update(leaf[x], diff)
// 将叶子节点x值增加diff
void update(int rt, LL diff)
{
	node[rt].val += diff;
	node[rt].vmx = node[rt].vmn = node[rt].val;
	pushup(rt);
}

// query(1, l, r)
// 自根节点向下, 查询区间[l, r]和
LL query(int rt, int l, int r)
{
	if (node[rt].l==l&&node[rt].r==r)
		return node[rt].val;
	int mid = (node[rt].l+node[rt].r)/2;
	if (r<=mid)
		return query(lson(rt), l, r);
	if (mid<l)
		return query(rson(rt), l, r);
	return query(lson(rt), l, mid) + query(rson(rt), mid+1, r);
}

// query_max(1, l, r)
// 自根节点向下, 查询区间max{[l, r]}
LL query_max(int rt, int l, int r)
{
	if (node[rt].l==l&&node[rt].r==r)
		return node[rt].vmx;
	int mid = (node[rt].l+node[rt].r)/2;
	if (r<=mid)
		return query_max(lson(rt), l, r);
	if (mid<l)
		return query_max(rson(rt), l, r);
	return max(query_max(lson(rt), l, mid), query_max(rson(rt), mid+1, r));
}

// query_min(1, l, r)
// 自根节点向下, 查询区间min{[l, r]}
LL query_min(int rt, int l, int r)
{
	if (node[rt].l==l&&node[rt].r==r)
		return node[rt].vmn;
	int mid = (node[rt].l+node[rt].r)/2;
	if (r<=mid)
		return query_min(lson(rt), l, r);
	if (mid<l)
		return query_min(rson(rt), l, r);
	return min(query_min(lson(rt), l, mid), query_min(rson(rt), mid+1, r));
}
```

区间修改

根节点为 1 延迟更新 Lazy 标记

```
#define lson(x) ((x)<<1)
#define rson(x) ((x)<<1|1)
struct SegmentTreeNode{
	int l, r;
	LL val, vmx, vmn;
	LL lazy;  // 延迟标记, 表示覆盖区域内每个单独节点的增量
} node[maxN<<2];  // 根节点为1, 四倍空间初始化

// build(1, 1, n, arr)
// 使用arr数组提供的初值, 以1为根节点, 建立覆盖区间[1, n]的线段树
void build(int rt, int l, int r, LL *arr)
{
	node[rt].l = l, node[rt].r = r;
	node[rt].lazy = 0;
	if (l==r) {
		node[rt].val = node[rt].vmx = node[rt].vmn = arr[l];
	}
	else {
		int mid = (l+r)/2;
		build(lson(rt), l, mid, arr);
		build(rson(rt), mid+1, r, arr);
		node[rt].val = node[lson(rt)].val + node[rson(rt)].val;
		node[rt].vmx = max(node[lson(rt)].vmx, node[rson(rt)].vmx);
		node[rt].vmn = min(node[lson(rt)].vmn, node[rson(rt)].vmn);
	}
}

// 下传延迟标记
void pushdown(int rt)
{
	if (node[rt].lazy) {
		node[rt].val += node[rt].lazy*(node[rt].r-node[rt].l+1);
		node[rt].vmx += node[rt].lazy;
		node[rt].vmn += node[rt].lazy;
		node[lson(rt)].lazy += node[rt].lazy;
		node[rson(rt)].lazy += node[rt].lazy;
	}
	node[rt].lazy = 0;
}

// query(1, l, r)
// 自根节点向下, 查询区间[l, r]和
LL query(int rt, int l, int r)
{
	if (node[rt].l==l&&node[rt].r==r)
		return node[rt].val + node[rt].lazy*(node[rt].r-node[rt].l+1);
	pushdown(rt);
	int mid = (node[rt].l+node[rt].r)/2;
	if (r<=mid)
```

```
        return query(lson(rt), l, r);
    if (mid<l)
        return query(rson(rt), l, r);
    return query(lson(rt), l, mid) + query(rson(rt),
mid+1, r);
}

LL query_max(int rt, int l, int r)
{
    if (node[rt].l==l&&node[rt].r==r)
        return node[rt].vmx + node[rt].lazy;
    pushdown(rt);
    int mid = (node[rt].l+node[rt].r)/2;
    if (r<=mid)
        return query_max(lson(rt), l, r);
    if(mid<l)
        return query_max(rson(rt), l, r);
    return max(query_max(lson(rt), l, mid),
query_max(rson(rt), mid+1, r));
}

LL query_min(int rt, int l, int r)
{
    if (node[rt].l==l&&node[rt].r==r)
        return node[rt].vmn + node[rt].lazy;
    pushdown(rt);
    int mid = (node[rt].l+node[rt].r)/2;
    if (r<=mid)
        return query_min(lson(rt), l, r);
    if (mid<l)
        return query_min(rson(rt), l, r);
    return min(query_min(lson(rt), l, mid),
query_min(rson(rt), mid+1, r));
}

// update(1, l, r, diff)
// 自根节点向下，将[l, r]区间内的元素增加diff
void update(int rt, int l, int r, int diff)
{
    if (l<=node[rt].l&&node[rt].r<=r) {
        node[rt].lazy += diff;
        return;
    }
    int mid = (node[rt].l+node[rt].r)/2;
    if (r<=mid) {
        update(lson(rt), l, r, diff);
    }
    else if (mid<l) {
        update(rson(rt), l, r, diff);
    }
    else {
        update(lson(rt), l, r, diff);
        update(rson(rt), l, r, diff);
    }
    node[rt].val = query(lson(rt), node[rt].l, mid) +
query(rson(rt), mid+1, node[rt].r);
    node[rt].vmx = max(query_max(lson(rt), node[rt].l,
mid), query_max(rson(rt), mid+1, node[rt].r));
    node[rt].vmn = min(query_min(lson(rt), node[rt].l,
mid), query_min(rson(rt), mid+1, node[rt].r));
}
```

区间覆盖

```
#define lson(x) ((x)<<1)
#define rson(x) ((x)<<1|1)
struct SegmentTreeNode{
    int l, r;
    LL val, vmx, vmn;
    LL lazy;  // 延迟标记，表示管辖区域内每个单独节点的增量
    LL force;   // 覆盖标记，表示管辖区域内每个节点被修改后
的值，注意修改为0的情况
} node[maxN<<2];  // 根节点为1，四倍空间初始化
bool covered[maxN<<2];  // 节点管辖区域是否被覆盖

// build(1, 1, n, arr)
// 使用arr数组提供的初值，以1为根节点，建立覆盖区间[1, n]的线
段树
void build(int rt, int l, int r, LL *arr)
{
    node[rt].l = l, node[rt].r = r;
    node[rt].lazy = node[rt].force = 0;
    if (l==r) {
        node[rt].val = node[rt].vmx = node[rt].vmn =
arr[l];
    }
    else {
        int mid = (l+r)/2;
        build(lson(rt), l, mid, arr);
        build(rson(rt), mid+1, r, arr);
        node[rt].val = node[lson(rt)].val +
node[rson(rt)].val;
        node[rt].vmx = max(node[lson(rt)].vmx,
node[rson(rt)].vmx);
        node[rt].vmn = min(node[lson(rt)].vmn,
node[rson(rt)].vmn);
    }
}

// 下传延迟标记或覆盖标记
void pushdown(int rt)
{
    if (covered[rt]) {
        node[rt].val = node[rt].force*(node[rt].r-
node[rt].l+1);
        node[rt].vmx = node[rt].vmn = node[rt].force;
        covered[lson(rt)] = covered[rson(rt)] = true;
        node[lson(rt)].lazy  = node[rson(rt)].lazy  =
0;
        node[lson(rt)].force = node[rson(rt)].force =
node[rt].force;
        covered[rt] = false;
    }
    if (node[rt].lazy) {
        node[rt].val += node[rt].lazy*(node[rt].r-
node[rt].l+1);
        node[rt].vmx += node[rt].lazy;
        node[rt].vmn += node[rt].lazy;
        node[lson(rt)].lazy += node[rt].lazy;
        node[rson(rt)].lazy += node[rt].lazy;
        node[rt].lazy = 0;
    }
}

// query(1, l, r)
// 自根节点向下，查询区间[l, r]和
LL query(int rt, int l, int r)
{
    if (node[rt].l==l&&node[rt].r==r) {
        if (covered[rt]) return
(node[rt].force+node[rt].lazy)*(node[rt].r-node[rt].l+1);
        return node[rt].val +
node[rt].lazy*(node[rt].r-node[rt].l+1);
    }
    pushdown(rt);
    int mid = (node[rt].l+node[rt].r)/2;
    if (r<=mid)
        return query(lson(rt), l, r);
    if (mid<l)
        return query(rson(rt), l, r);
    return query(lson(rt), l, mid) + query(rson(rt),
mid+1, r);
}

LL query_max(int rt, int l, int r)
{
    if (node[rt].l==l&&node[rt].r==r) {
        if (covered[rt]) return node[rt].force +
node[rt].lazy;
        return node[rt].vmx + node[rt].lazy;
    }
    pushdown(rt);
    int mid = (node[rt].l+node[rt].r)/2;
    if (r<=mid)
        return query_max(lson(rt), l, r);
    if(mid<l)
        return query_max(rson(rt), l, r);
    return max(query_max(lson(rt), l, mid),
query_max(rson(rt), mid+1, r));
}

LL query_min(int rt, int l, int r)
{
    if (node[rt].l==l&&node[rt].r==r) {
        if (covered[rt]) return node[rt].force +
node[rt].lazy;
        return node[rt].vmn + node[rt].lazy;
    }
    pushdown(rt);
    int mid = (node[rt].l+node[rt].r)/2;
    if (r<=mid)
        return query_min(lson(rt), l, r);
    if (mid<l)
        return query_min(rson(rt), l, r);
    return min(query_min(lson(rt), l, mid),
query_min(rson(rt), mid+1, r));
}

// update(1, l, r, diff)
// 自根节点向下，将[l, r]区间内的元素增加diff
void update(int rt, int l, int r, int diff)
{
    if (l<=node[rt].l&&node[rt].r<=r) {
        node[rt].lazy += diff;
        return;
    }
    pushdown(rt);
    int mid = (node[rt].l+node[rt].r)/2;
```

```
    if (r<=mid) {
        update(lson(rt), l, r, diff);
    }
    else if (mid<l) {
        update(rson(rt), l, r, diff);
    }
    else {
        update(lson(rt), l, r, diff);
        update(rson(rt), l, r, diff);
    }
    node[rt].val = query(lson(rt), node[rt].l, mid) +
query(rson(rt), mid+1, node[rt].r);
    node[rt].vmx = max(query_max(lson(rt), node[rt].l,
mid), query_max(rson(rt), mid+1, node[rt].r));
    node[rt].vmn = min(query_min(lson(rt), node[rt].l,
mid), query_min(rson(rt), mid+1, node[rt].r));
}

// modify(1, l, r, val)
// 自根节点向下，将[l, r]区间中的元素修改为val
void modify(int rt, int l, int r, int val)
{
    if (l<=node[rt].l&&node[rt].r<=r) {
        node[rt].lazy = 0; node[rt].force = val;
        covered[rt] = true;
        return;
    }
    pushdown(rt);
    int mid = (node[rt].l+node[rt].r)/2;
    if (r<=mid) {
        modify(lson(rt), l, r, val);
    }
    else if (mid<l) {
        modify(rson(rt), l, r, val);
    }
    else {
        modify(lson(rt), l, r, val);
        modify(rson(rt), l, r, val);
    }
    node[rt].val = query(lson(rt), node[rt].l, mid) +
query(rson(rt), mid+1, node[rt].r);
    node[rt].vmx = max(query_max(lson(rt), node[rt].l,
mid), query_max(rson(rt), mid+1, node[rt].r));
    node[rt].vmn = min(query_min(lson(rt), node[rt].l,
mid), query_min(rson(rt), mid+1, node[rt].r));
}
```

扫描线方法

https://blog.csdn.net/lwt36/article/details/48908031

树状数组 Binary Indexed Tree

区间求和单点更新

```
int cell[maxN];  // 有效索引从1开始

#define lowbit(x) (x)&(-(x))
void add(int x, int v) {
    for (int i = x; i < maxN; i += lowbit(i))
        cell[i] += v;
}

int get(int x) {
    int sum = 0;
    for (int i = x; i; i -= lowbit(i))
        sum += cell[i];
    return sum;
}
```

区间求和区间更新

记$\Delta(x)$为区间[x, maxN)上元素的共同增量，则前缀和

$$\mathrm{sum}(x)=\sum_{i=1}^{x}(x-i+1)\cdot\Delta(i)=(x+1)\cdot\sum_{i=1}^{x}\Delta(i)-\sum_{i=1}^{x}i\cdot\Delta(i)$$

两个求和记号所在的部分可以使用两个树状数组进行维护，序列的初始值可以保存在维护$\sum_{i=1}^{x}i\cdot\Delta(i)$的树状数组中

```
// 1-based; 序列的初始值可以保存在Δ(i)*i树状数组中
LL Di[maxN], Dii[maxN];  // Δ(i), Δ(i)*i

void add(LL *bit, int x, int val) {
    for (int i=x; i<maxN; i+=lowbit(i)) {
        bit[i] += val;
    }
}

LL sum(LL *bit, int x) {
    LL rslt = 0;
    for (int i=x; i; i-=lowbit(i)) {
        rslt += bit[x];
    }
    return rslt;
}

// 将[a, b]区间中的元素增加val
void add(int a, int b, LL val) {
    add(Di, a, val);
    add(Di, b+1, -val);
    add(Dii, a, -a*val);
    add(Dii, b+1, (b+1)*val);
}

// 求前缀和[1, x]
LL sum(int x) {
    return sum(Di, x)*(x+1) + sum(Dii, x);
}

// 区间和[a, b]
LL get(int a, int b) {
    return sum(b) - sum(a-1);
}
```

二维树状数组

在一维的基础上增加一维，查询 sum[x][y]实际上是把查询分摊到负责前 x 行的桶，对负责前 x 行的桶求解本行的 sum[y]后合并，递归过程。

```
// cell[x][y]增加v
void add(int x, int y, int v)
{
    for (int i = x; i<maxN; i+=lowbit(i))
        for (int j = y; j<maxN; j+=lowbit(j))
            cell[i][j] += v;
}

// (1, 1)至(x, y)中所有单元格的权值和
int get(int x, int y)
{
    int rslt = 0;
    for (int i = x; i; i-=lowbit(i))
        for (int j = y; j; j-=lowbit(j))
            rslt += cell[i][j];
    return rslt;
}
```

字典树 Trie

简易 利用 map<char, int>不难实现

左偏树 Leftist Tree

编号为 0 的节点表示空节点

```
struct LeftTree
{
    const static int MXN = 100100;
    int tot = 0;
    int l[MXN], r[MXN], v[MXN], d[MXN];

    // 初始化值为x的元素
    int init(int x)
    {
        tot++;
        v[tot] = x;
        l[tot] = r[tot] = d[tot] = 0;
        return tot;
    }

    // 合并堆顶编号为x, y的堆
    int merge(int x, int y)
    {
        if (!x) return y;
        if (!y) return x;
        if (v[x] < v[y])
            swap(x, y);
        r[x] = merge(r[x], y);
        if (d[l[x]] < d[r[x]])
            swap(l[x], r[x]);
        d[x] = d[r[x]] + 1;
        return x;
    }

    // 向堆顶编号为x的堆中插入值为v的元素
    int insert(int x, int v)
    {
        return merge(x, init(v));
```

```
    }

    // 取编号为x的堆的堆顶元素
    int top(int x)
    {
        return v[x];
    }

    // 弹出编号为x的堆的堆顶元素，返回新堆顶的编号
    int pop(int x)
    {
        return merge(l[x], r[x]);
    }
};
```

哈夫曼树

以频率为节点权值维护节点队列。合并队列中权值最小的两个节点，将合并的新节点放入队列中，重复步骤，直至队列中只存在一个节点。

图论

匹配

二分图最大匹配匈牙利算法

```
vector<int> g[maxN];  // g[x]：与x点相连的右侧的点
int from[maxN], tot;  // from[x]: 与右侧点x匹配的左侧的点，
tot：最大匹配数
bool use[maxN];

bool match(int x)  // x：待匹配的一个左侧的点
{
    for (unsigned i=0; i<g[x].size(); ++i) if
(!use[g[x][i]])
    {
        use[g[x][i]] = true;
        if (from[g[x][i]] == -1 ||
match(from[g[x][i]])) {
            from[g[x][i]] = x; return true;
        }
    }
    return false;
}

int hungary(int n)  // n：左侧点的个数
{
    memset(from, -1, sizeof(from)); tot = 0;
    for (int i=1; i<=n; ++i) {
        memset(use, 0, sizeof(use));
        if (match(i)) ++tot;
    }
    return tot;
}
```

单源非负最短路 Dijkstra

升级 堆优化

SPFA

队列非空时，队头出列；松弛队头的边，已松弛且不在队列中的顶点入队。入队超过 n 次则途图中存在负环。

最小生成树理论基础

环定理 对于连通图中的环 C，若环 C 中的一条边 e 的权值大于该环中任意一个其他边的权值，那么该边不是最小生成树中的边。

切分定理 给定任意任意一个切分，横切边中权值最小的边属于最小生成树

最小权值边定理 如果图具有最小权值的边只有一条，那么该边在图的任意一个最小生成树中。

最小生成树顶点优先 Prim

类似于 Dijkstra，但维护的距离是顶点到已松弛顶点的集合的距离。

最小生成树边优先 Kruskal

维护顶点的集合 $S=V_0$，$T=(V-S)$。边升序遍历，对于每一条边 (s, t)，若 $s\in S$，$t\in T$，则将边加入树中，并将 t 并入 s；T 中没有顶点时，算法结束，所得树为最小生成树。

网络流

最大流 Dinic

计算几何

海伦公式 $A = \sqrt{p\sqrt{p-a}\sqrt{p-b}\sqrt{p-c}}$ $p = \frac{a+b+c}{2}$

基础工具

符号判定（误差修正）

向量与点

点积 叉积 欧几里得距离 模长 旋转

```
const double eps = 1e-8;
const double PI  = acos(-1);

int sgn(double x)  // 返回x的符号
{
    if (abs(x)<eps) return 0;
    return x>0 ? 1 : -1;
}

inline double sqr(double x) { return x*x; }

struct Point
{
    double x, y;

    Point(double x_ = 0, double y_ = 0) : x(x_), y(y_)
{};
    void read() {
        scanf("%lf%lf", &x, &y);
    }
    double norm() {
        return sqrt(sqr(x)+sqr(y));
    }

    friend Point operator + (const Point &a, const Point
&b) {
        return Point(a.x+b.x, a.y+b.y);
    }
    friend Point operator - (const Point &a, const Point
&b) {
        return Point(a.x-b.x, a.y-b.y);
    }
    friend Point operator * (const Point &a, const
double &b) {
        return Point(a.x*b, a.y*b);
    }
    friend Point operator / (const Point &a, const
double &b) {
        return Point(a.x/b, a.y/b);
    }
    friend bool operator == (const Point &a, const Point
&b) {
        return sgn(a.x-b.x)==0 && sgn(a.y-b.y)==0;
    }
};

// 叉积
double det(const Point &a, const Point &b) {
    return a.x*b.y-a.y*b.x;
}

// 点积
double dot(const Point &a, const Point &b) {
    return a.x*b.x+a.y*b.y;
```

```cpp
}

// 两点间距离
double dist(const Point &a, const Point &b) {
     return (a-b).norm();
}

// 将向量绕原点逆时针旋转弧度A（由旋转前后模长不变推导）
// 绕任意点Q旋转，只需在答案中追加Q的横纵坐标
Point rotate(const Point &p, double A) {
     double tx=p.x, ty=p.y;
     return Point(tx*cos(A)-ty*sin(A),
tx*sin(A)+ty*cos(A));
}
```

最近点对 O(nLogn)

考虑分治，合并步骤注意。

```cpp
Point p[maxN];  // 点集
int s[maxN];    // 临时变量

bool cmpx(int i, int j) {
     return sgn(p[i].x-p[j].x)<0;
}

bool cmpy(int i, int j) {
     return sgn(p[i].y-p[j].y)<0;
}

// 最近点对分治步骤
double min_dist(Point p[], int s[], int l, int r)
{
     double ans = 1e100;
     if (r-l<20) {
          for (int q=l; q<r; ++q) for (int w=q+1; w<r;
++w) {
               ans = min(ans, (p[s[q]]-p[s[w]]).norm());
          }
          return ans;
     }

     int tl, tr, m=(l+r)/2;
     ans = min(min_dist(p, s, l, m), min_dist(p, s, m,
r));
     for (tl=l; p[s[tl]].x<p[s[m]].x-ans; ++tl);
     for (tr=r-1; p[s[tr]].x>p[s[m]].x+ans; --tr);
     sort(s+tl, s+tr, cmpy);
     for (int q=tl; q<tr; ++q) {
          for (int w=q+1; w<min(tr, q+6); ++w) {
               ans = min(ans, (p[s[q]]-p[s[w]]).norm());
          }
     }
     sort(s+tl, s+tr, cmpx);
     return ans;
}

// 最近点对 - 求解n个点中最近两点的距离
// p - 点集, n - 点数
// s - 临时变量, 算法执行后得到x坐标非降序排列的点编号
double min_dist(Point p[], int s[], int n)
{
     iota(s, s+n, 0);
     sort(s, s+n, cmpx);  // 以x坐标非降序编号点
     return min_dist(p, s, 0, n);
}
```

数论

二项式定理 $(x+a)^n = \sum_{k=0}^{n}\binom{n}{k}x^k a^{n-k}$

组合数 $C_n^k = \binom{n}{k} = \frac{n!}{(n-m)!\cdot m!}$

$$C(n,m) = C(n-1,m) + C(n-1,m-1)$$

错排公式 $D_n = (n-1)(D_{n-1} + D_{n-2})$

费马小定理

若 p 为质数，$a^p \equiv a\ (mod\ p)$

若 a 不是 p 的倍数，$a^{p-1} \equiv 1\ (mod\ p)$

引理，$a^p \equiv 1\ (mod\ p) \rightarrow a \equiv \pm 1\ (mod\ p)$

威尔逊定理 $(p-1)! \equiv -1 (mod\ p)$

自然数 N 因子个数 $f(n)$ 考虑分解质因数

乘法逆元 $ax \equiv 1(mod\ b)$

当且仅当$gcd(a,b) = 1$时乘法逆元存在，可用于计算模意义下的除法。

利用费马小定理计算 $a \times a^{p-2} \equiv 1(mod\ p)$ p 为质数

利用拓展欧几里得计算 $ax \equiv 1(mod\ b)$, $gcd(a,b) = 1$

```cpp
int cal_inv(int a, int p)
{
     int g, x, y; exgcd(a, p, g, x, y);
     return (x%p+p)%p;
}
```

利用递推公式计算

设$p = k \times i + r, r < i, 1 < r < p$，有$k \times i + r \equiv 0(\mathrm{mod\ p})$，两边同时乘以$r^{-1} + i^{-1}$，移项整理得$i^{-1} \equiv -\left\lfloor\frac{p}{i}\right\rfloor \times (p\ mod\ i)^{-1}$

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

期望

等价期望形式

$$E(x) = 1 * P(x = 1) + 2 * P(x = 2) + 3 * P(x = 3) + \cdots$$

$$E(x) = P(x > 0) + P(x > 1) + P(x > 2) + \cdots$$

经验 [Expected-LCP] Trie+期望

博弈

Nim 博弈

n 堆石子，两名玩家先后选取一堆石子取任意多个，将所有石子取完为胜。

解法：异或所有石堆石子的数量，值为 0 则先手败。
证明：记能转换为终结态或 P 态的状态为 N 态，终结态或能转换为 N 态的状态为 P 态。异或操作满足：1)所有的终结态都被判断为 N 态 2)判断为 P 态的状态可以通过合法操作移动到 N 态（二进制特点和异或特点）3)判断为 P 态的状态无法直接转换为另一个 P 态（异或特点）

Bash 博弈

两名玩家先后从一堆石子中取[1, k]个石子，取走最后一个为胜利。

猜想

Bertrand 猜想 对于任意$n > 3$，存在$n < p < 2n$，其中p为质数

质数间隔 在 1e13 范围内，相邻素数的最大间隔为 777

欧拉函数φ(n)

φ(n) 小于或等于 n 的正整数中与 n 互质的数的个数。

1） $\varphi(1) = 1$

2） 若 n 是素数 p 的 k 次幂，$\varphi(n) = p^k - p^{k-1} = (p-1)p^{k-1}$

3） 若 m,n 互质，$\varphi(mn) = \varphi(m) \cdot \varphi(n)$

递推式：令 p 为 N 的最小质因数，若$p^2|N$, $\varphi(N) = \varphi\left(\frac{N}{p}\right) \times p$；否则，$\varphi(N) = \varphi\left(\frac{N}{p}\right) \times (p-1)$

```cpp
int euler(int n)  // 欧拉函数
```

```
{
    int ans = n;
    for (int i=2; i*i<=n; ++i)
    {
        if (n%i==0)
        {
            ans=ans*(i-1)/i;
            while (n%i==0) n/=i;
        }
    }
    if (n>1) ans=ans*(n-1)/n;
    return ans;
}
```

欧拉降幂 $a^b (mod\ c) = a^{b\ mod\ \varphi(c)+\varphi(c)}\ (mod\ c), b \geq \varphi(c)$

Miller-Rabin 素性测试 O(logN)

需要快速幂 mod_pow(a, b, mod)

```
// 独立Miller-Robin测试，返回n是否为质数
bool test(LL n, LL a, LL d)
{
    if (n==2) return true;
    if (n==a) return true;
    if ((n&1)==0) return false;
    while (!(d&1)) d >>= 1;
    LL t = mod_pow(a, d, n);
    while ((d!=n-1) && (t!=1) && t!=(n-1))
    {
        t = t * t % n;
        d <<= 1;
    }
    return (t==n-1 || (d&1));
}

// 判定n是否为素数
bool isprime(LL n)
{
    if (n<2) return false;
    int a[] = {2, 3, 61};
// 更大范围的素数需要更广的测试集
    for (int i=0; i < 3; ++i) if (!test(n, a[i], n-1))
return false;
    return true;
}
```

拓展欧几里得 $ax + by = gcd(a, b)$

```
void exgcd(LL a, LL b, LL& g, LL &x, LL &y)
{
    if (!b) g=a, x=1, y=0;
    else exgcd(b, a%b, g, y, x), y-=a/b*x;
}
```

$$x = x_0 + \frac{b}{\gcd(a,b)} \cdot t, \qquad y = y_0 - \frac{a}{\gcd(a,b)} \cdot t$$

单变元模线性方程组 $ax \equiv b (mod\ n)$

相当于求解$ax + ny = b$，当且仅当$gcd(a, n)\,|n$时有解，且有$gcd(a, n)$个解。通解：

$$x_i = \left[x_0 + i \cdot \left(\frac{n}{gcd(a,n)}\right)\right] (mod\ n), \qquad i = 0, 1, 2, \ldots, gcd(a, n) - 1$$

```
vector<LL> line_mod_equation(LL a, LL b, LL n)
{
    LL x, y, g; exgcd(a, n, g, x, y);

    vector<LL> ans;
    if (b%g == 0) {
        x %= n; x += n; x %= n;
        ans.push_back(x*(b/g)%(n/g));
        for (LL i=1; i<g; ++i)
            ans.push_back((ans[0]+i*(n/g))%n);
    }
    return ans;
}
```

语言及黑科技

Java

```
// BigInteger and BigDecimal
import java.math.*;
import java.util.Scanner;

add multiply subtract divide
```

C++

```
set_intersection()
set_union()
set_difference()
```

字符串格式工具

```
string stoi stol stoll stod to string
*char atoi atol atof
```

正则表达式 Regit

IO 优化

```
template<typename T = int>
inline T read() {
    T val = 0, sign = 1; char ch;
    for (ch = getchar(); ch < '0' || ch > '9'; ch =
getchar())
        if (ch == '-') sign = -1;
    for (; ch >= '0' && ch <= '9'; ch = getchar())
        val = val * 10 + ch - '0';
    return sign * val;
}
```

时空优化

展开循环：牺牲程序的尺寸加快程序的执行速度

```
#pragma GCC optimize("unroll-loops")
```