# ACM板子

## 代码与编译

#编译选项
编译时加入以下命令：

```
-DONLINE_JUDGE -fno-tree-ch -O0 -Wall -std=c++11
```

#文件头

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define Inf 0x3f3f3f3f
#define INF 0x3f3f3f3f3f3f3f3f
// #define int long long
template <typename T>
inline T& read(T& a) {  // 快读
    a = 0;
    bool f = false;
    char c = getchar();
    while (!isdigit(c)) {
        f |= c == '-';
        c = getchar();
    }
    while (isdigit(c)) {
        a = (a << 1) + (a << 3) + (c ^ 48);
        c = getchar();
    }
    if (f) a = -a;
    return a;
}
```

## 基础算法

#二分答案
以复杂度乘$O(\log n)$的代价搜索答案并验证

#三分法
求纯凹函数的最小值或纯凸函数的最大值

#CDQ分治

```cpp
int f(int l, int r) {  // 区间对问题
    if (l == r) return 0;  // 最小问题
    int m = l + r >> 1;
    int re = max(f(l, m), f(m + 1, r));  // 递归
    unordered_map<int, int> mp;
    for (int i = m, s = 0; i >= l; --i) {  // 建桶
        mp[s += a[i]] = m + 1 - i;
    }
    for (int i = m + 1, s = 0; i <= r; ++i) {  // 用桶
        re = max(re, mp[s -= a[i]] + i - m);
    }
    return re;
}
```

也可以是基于归并排序的CDQ分治

---

## 数据结构

#栈 #队列 #链表
#哈希表
#堆
#单调栈
#单调队列

#树状数组
这里的功能是维护前缀最大值:

```cpp
int bt[N];

inline void add(int p, int k){
    for (; p < N; p += p & -p)
        bt[p] = max(bt[p], k);
}

inline int qry(int p){
    int re = 0;
    for (; p; p -= p & -p)
        re = max(re, bt[p]);
    return re;
}
```

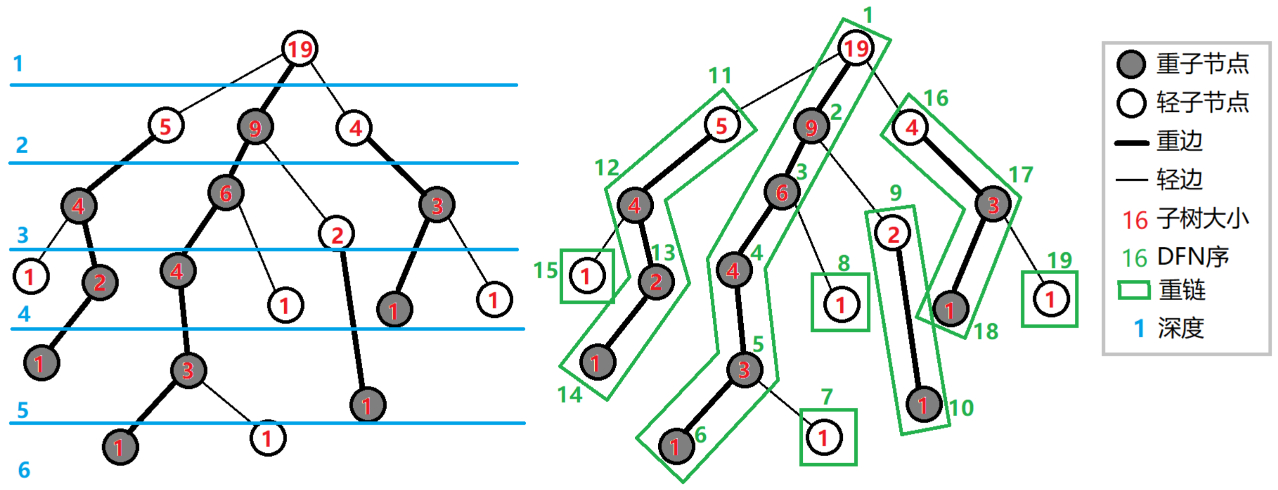
#并查集 （简单版本）
一定要初始化 `fa[i] = i`，否则合并会出问题！！！

```
// 寻找祖先
int find(int x) {
    return fa[x] == x ? x : fa[x] = find(fa[x]);
}
// （非启发式）合并
void hb(int x, int y) {
    fa[find(x)] = find(y);
}
```

#线段树

```
int n;
ll a[N];
ll tr[N << 2], lz[N << 2]; // 注意开大数组！！！！

ll init(int l = 1, int r = n, int p = 1) {
    if (l == r) return tr[p] = a[l];
    int m = l + r >> 1, ps = p << 1;
    return tr[p] = init(l, m, ps) + init(m + 1, r, ps | 1);
}

void pd(int l, int r, int p) {
    if (l == r || lz[p] == 0) return;
    int m = l + r >> 1, ps = p << 1;
    lz[ps] += lz[p];
    lz[ps | 1] += lz[p];
    tr[ps] += lz[p] * (m - l + 1);
    tr[ps | 1] += lz[p] * (r - m);
    lz[p] = 0;
}

void add(int s, int t, int k, int l = 1, int r = n, int p = 1) {
    if (s == l && t == r) {
        tr[p] += k * (t - s + 1);
        lz[p] += k;
        return;
    }
    pd(l, r, p);
    int m = l + r >> 1, ps = p << 1;
    if (s ≤ m) add(s, min(t, m), k, l, m, ps);
    if (t > m) add(max(s, m + 1), t, k, m + 1, r, ps | 1);
    tr[p] = tr[ps] + tr[ps | 1];
}

ll qry(int s, int t, int l = 1, int r = n, int p = 1) {
    if (s == l && t == r) return tr[p];
    pd(l, r, p);
    int m = l + r >> 1, ps = p << 1;
    ll re = 0;
    if (s ≤ m) re += qry(s, min(t, m), l, m, ps);
    if (t > m) re += qry(max(s, m + 1), t, m + 1, r, ps | 1);
    return re;
}
```

#链式前向星 #树链剖分 #重链剖分 #LCA #树链剖分线段树



已知一棵包含 NN 个结点的树（连通且无环），每个节点上包含一个数值，需要支持以下操作：

- `1 x y z`，表示将树从 `x` 到 `y` 结点最短路径上所有节点的值都加上 `z`
- `2 x y`，表示求树从 `x` 到 `y` 结点最短路径上所有节点的值之和
- `3 x z`，表示将以 `x` 为根节点的子树内所有节点值都加上 `z`
- `4 x`， 表示求以 `x` 为根节点的子树内所有节点值之和

```
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
#define Inf 0x3f3f3f3f
#define INF 0x3f3f3f3f3f3f3f3f
// #define int long long
template <typename T>
inline T& read(T& a) {  // 快读
    a = 0;
```

```cpp
    bool f = false;
    char c = getchar();
    while (!isdigit(c)) {
        f |= c == '-';
        c = getchar();
    }
    while (isdigit(c)) {
        a = (a << 1) + (a << 3) + (c ^ 48);
        c = getchar();
    }
    if (f) a = -a;
    return a;
}

const int N = 100005;
int n, m, r;
ll P;
ll w[N];
int hed[N], to[N << 1], nxt[N << 1], p = 1;

inline void add(int u, int v) {
    nxt[++p] = hed[u];
    hed[u] = p;
    to[p] = v;
}

int fa[N], dep[N], siz[N], hs[N];
void dfs1(int o = r) {
    siz[o] = 1;
    for (int i = hed[o]; i; i = nxt[i]) {
        int t = to[i];
        if (t == fa[o]) continue;
        fa[t] = o;
        dep[t] = dep[o] + 1;
        dfs1(t);
        siz[o] += siz[t];
        if (siz[hs[o]] < siz[t]) hs[o] = t;
    }
}

int dfn[N], rnk[N], pos = 0, top[N];
void dfs2(int o = r) {
    rnk[dfn[o] = ++pos] = o;
    if (hs[o]) {
        top[hs[o]] = top[o];
        dfs2(hs[o]);
        for (int i = hed[o]; i; i = nxt[i]) {
            int t = to[i];
            if (t == fa[o] || t == hs[o]) continue;
            top[t] = t;
            dfs2(t);
        }
    }
}

inline int lca(int x, int y) {
    while (top[x] != top[y]) {
        if (dep[top[x]] > dep[top[y]])
            x = fa[top[x]];
        else
            y = fa[top[y]];
    }
    if (dep[x] < dep[y])
        return x;
    else
        return y;
}

// 以结点的dfn排序建立线段树，可以通过rnk反求结点编号
ll sum[N << 2], laz[N << 2];
ll build(int l = 1, int r = n, int p = 1) {
    if (l == r) return sum[p] = w[rnk[l]];
    int m = l + r >> 1, ps = p << 1;
    return sum[p] = build(l, m, ps) + build(m + 1, r, ps | 1);
}

inline void pd(int l, int r, int p) {
    if (laz[p] == 0 || l == r) return;
    int m = l + r >> 1, ps = p << 1;
    sum[ps] += laz[p] * (m - l + 1);
    laz[ps] += laz[p];
    sum[ps | 1] += laz[p] * (r - m);
    laz[ps | 1] += laz[p];
    laz[p] = 0;
}

void chg(int s, int t, ll k, int l = 1, int r = n, int p = 1) {
    if (s > t) swap(s, t);  // 重要！！！
    if (s == l && t == r) {
        sum[p] += k * (r - l + 1);
        laz[p] += k;
        return;
    }
    pd(l, r, p);  // 重要！！！
    int m = l + r >> 1, ps = p << 1;
    if (s <= m) chg(s, min(t, m), k, l, m, ps);
    if (t > m) chg(max(s, m + 1), t, k, m + 1, r, ps | 1);
    sum[p] = sum[ps] + sum[ps | 1];
```

```cpp
}

ll qry(int s, int t, int l = 1, int r = n, int p = 1) {
    if (s > t) swap(s, t);  // 重要！！！
    if (s == l && t == r) return sum[p];
    pd(l, r, p);  // 重要！！！
    int m = l + r >> 1, ps = p << 1;
    ll re = 0;
    if (s <= m) re += qry(s, min(t, m), l, m, ps);
    if (t > m) re += qry(max(s, m + 1), t, m + 1, r, ps | 1);
    return re;
}

inline void init() {
    dep[r] = 1;
    dfs1();
    top[r] = r;
    dfs2();
    build();
}

inline void case1() {
    int x, y, z;
    read(x), read(y), read(z);
    int L = lca(x, y);
    while (top[x] != top[L]) {
        chg(dfn[x], dfn[top[x]], z);
        x = fa[top[x]];
    }
    chg(dfn[x], dfn[L], z);
    while (top[y] != top[L]) {
        chg(dfn[y], dfn[top[y]], z);
        y = fa[top[y]];
    }
    chg(dfn[y], dfn[L], z);
    chg(dfn[L], dfn[L], -z);
}

inline void case2() {
    int x, y;
    read(x), read(y);
    int L = lca(x, y);
    ll ans = 0;
    while (top[x] != top[L]) {
        ans += qry(dfn[x], dfn[top[x]]);
        x = fa[top[x]];
    }
    ans += qry(dfn[x], dfn[L]);
    while (top[y] != top[L]) {
        ans += qry(dfn[y], dfn[top[y]]);
        y = fa[top[y]];
    }
    ans += qry(dfn[y], dfn[L]);
    ans -= qry(dfn[L], dfn[L]);
    printf("%lld\n", ans % P);
}

inline void case3() {
    int x, z;
    read(x), read(z);
    chg(dfn[x], dfn[x] + siz[x] - 1, z);
}

inline void case4() {
    int x;
    read(x);
    ll ans = qry(dfn[x], dfn[x] + siz[x] - 1);
    printf("%lld\n", ans % P);
}

signed main() {
    cin >> n >> m >> r >> P;
    for (int i = 1; i <= n; ++i)
        read(w[i]);
    for (int i = 1, x, y; i < n; ++i) {
        read(x), read(y);
        add(x, y), add(y, x);
    }
    init();
    for (int i = 1, op; i <= m; ++i) {
        read(op);
        switch (op) {
            case 1: case1(); break;
            case 2: case2(); break;
            case 3: case3(); break;
            case 4: case4(); break;
        }
    }

    return 0;
}
```

#ST表

```
const int L = 22;
int Log[N], ma[N][L];
inline void init() {
    Log[1] = 0;
    for (int i = 2; i < N; ++i)
        Log[i] = Log[i >> 1] + 1;

    for (int i = 1; i <= n; ++i) ma[i][0] = a[i];
    for (int j = 1; j <= L; ++j)
        for (int i = 1; i + (1 << j) - 1 <= n; ++i)
            ma[i][j] = max(ma[i][j - 1], ma[i + (1 << j - 1)][j - 1]);
}

inline int qry_max(int l, int r) {
    int q = Log[r - l + 1];
    return max(ma[l][q], ma[r - (1 << q) + 1][q]);
}
```

#二叉搜索树 #平衡树
#AVL树
#Splay树

#可持久化数据结构

---

## 数论

#线性筛 #欧拉函数

```
int ps[N], phi[N], tot = 0;
bool vis[N];
inline void init() {
    for (int i = 2; i < N; ++i) {
        if (!vis[i])
            ps[++tot] = i, phi[i] = i - 1;
        for (int j = 1; j <= tot; ++j) {
            int p = ps[j], t = i * p;
            if (t >= N) break;
            vis[t] = true;
            if (i % p)
                phi[t] = phi[i] * phi[p];
            else {
                phi[t] = phi[i] * p;
                break;
            }
        }
    }
}
```

若要求$\varphi(x)$（$x$很大），则需要对$x$进行质因数分解，然后利用$\varphi(x)$的积性来计算，其中需要$N > \sqrt{x}$

#快速幂

```
ll qpow(ll a, ll n) {
    a %= P;
    ll re = 1;
    while (n) {
        if (n & 1) re = re * a % P;
        a = a * a % P;
        n >>= 1;
    }
    return re;
}
```

#EXGCD
求$ax + by = \gcd(a, b)$的一组可行解

```
int Exgcd(int a, int b, int& x, int& y) {
    if (!b) {
        x = 1;
        y = 0;
        return a;
    }
    int d = Exgcd(b, a % b, x, y);
    int t = x;
    x = y;
    y = t - (a / b) * y;
    return d;
}
```

函数返回的值为gcd，在这个过程中计算$x, y$即可

---

## 图论

#树上问题

#最短路

#分层图

#欧拉回路

## 动态规划

#背包DP
#01背包

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

滚动数组优化：

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

需要 `j` 从大到小遍历

```
for (int i = 1; i ≤ n; i++)
    for (int j = W; j ≥ w[i]; j--)
        f[j] = max(f[j], f[j - w[i]] + v[i]);
```

#完全背包

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

同样采用滚动数组优化， `j` 从小到大遍历

```
for (int i = 1; i ≤ n; i++)
    for (int j = w[i]; j ≤ W; j++)
        if (f[j - w[i]] + v[i] > f[j]) f[j] = f[j - w[i]] + v[i];
```

## 字符串

#字典树 #Trie

```
struct trie {
    int nex[100000][26], cnt;
    bool exist[100000];  // 该结点结尾的字符串是否存在

    void insert(char* s, int l) {  // 插入字符串
        int p = 0;
        for (int i = 0; i < l; i++) {
            int c = s[i] - 'a';
            if (!nex[p][c]) nex[p][c] = ++cnt;  // 如果没有，就添加结点
            p = nex[p][c];
        }
        exist[p] = 1;
    }
    bool find(char* s, int l) {  // 查找字符串
        int p = 0;
        for (int i = 0; i < l; i++) {
            int c = s[i] - 'a';
            if (!nex[p][c]) return 0;
            p = nex[p][c];
        }
        return exist[p];
    }
};
```

#前缀数组

$$\pi[i] = \max_{k=0,\dots,i} \{\, k \mid s[0..k-1] = s[i-k+1..i] \,\}$$

```
vector<int> prefix_function(string s) {
    int n = (int)s.length();
    vector<int> pi(n);
    for (int i = 1; i < n; i++) {
        int j = pi[i - 1];
        while (j > 0 && s[i] != s[j]) j = pi[j - 1];
        if (s[i] == s[j]) j++;
        pi[i] = j;
    }
    return pi;
}
```

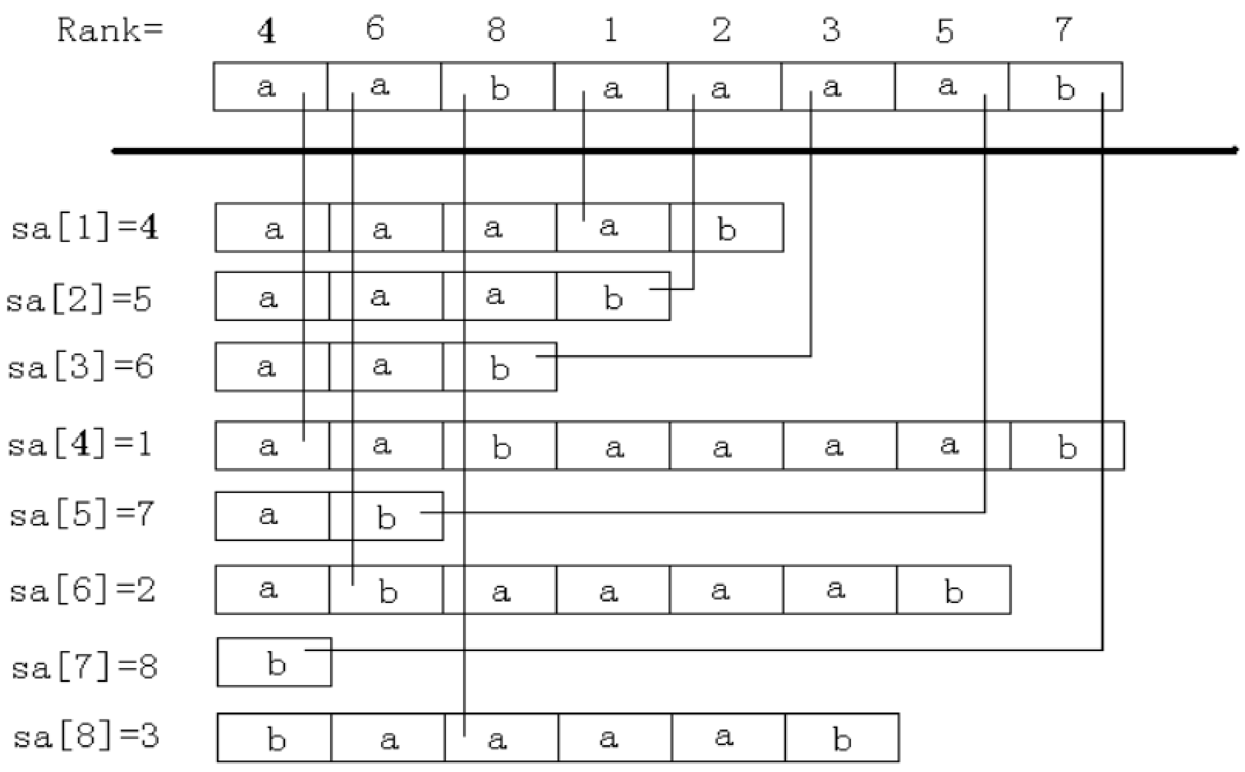
#KMP算法

```
vector<int> find_occurrences(string text, string pattern) {
    string cur = pattern + '#' + text;
    int sz1 = text.size(), sz2 = pattern.size();
    vector<int> v;
    vector<int> lps = prefix_function(cur);
    for (int i = sz2 + 1; i <= sz1 + sz2; i++) {
        if (lps[i] == sz2) v.push_back(i - 2 * sz2);
    }
    return v;
}
```

#AC自动机

#后缀数组

`sa[i]` 表示将所有后缀排序后第 `i` 小的后缀的编号
`rk[i]` 表示后缀 `i` 的排名，即 `sa[rk[i]] = rk[sa[i]] = i`



```cpp
#include <algorithm>
#include <cstdio>
#include <cstring>
#include <iostream>
using namespace std;

const int N = 1000010;

char s[N];
// key1[i] = rk[id[i]]（作为基数排序的第一关键字数组）
int n, sa[N], rk[N], oldrk[N << 1], id[N], key1[N], cnt[N];

bool cmp(int x, int y, int w) {
    return oldrk[x] == oldrk[y] && oldrk[x + w] == oldrk[y + w];
}

int main() {
    int i, m = 127, p, w;

    scanf("%s", s + 1);
    n = strlen(s + 1);
    for (i = 1; i ≤ n; ++i)
        ++cnt[rk[i] = s[i]];
    for (i = 1; i ≤ m; ++i)
        cnt[i] += cnt[i - 1];
    for (i = n; i ≥ 1; --i)
```

```
        sa[cnt[rk[i]]--] = i;

    for (w = 1;; w <<= 1, m = p) {  // m=p 就是优化计数排序值域
        for (p = 0, i = n; i > n - w; --i)
            id[++p] = i;
        for (i = 1; i <= n; ++i)
            if (sa[i] > w) id[++p] = sa[i] - w;

        memset(cnt, 0, sizeof(cnt));
        for (i = 1; i <= n; ++i)
            ++cnt[key1[i] = rk[id[i]]];
        // 注意这里px[i] != i，因为rk没有更新，是上一轮的排名数组

        for (i = 1; i <= m; ++i)
            cnt[i] += cnt[i - 1];
        for (i = n; i >= 1; --i)
            sa[cnt[key1[i]]--] = id[i];
        memcpy(oldrk + 1, rk + 1, n * sizeof(int));
        for (p = 0, i = 1; i <= n; ++i)
            rk[sa[i]] = cmp(sa[i], sa[i - 1], w) ? p : ++p;
        if (p == n) {
            break;
        }
    }

    for (i = 1; i <= n; ++i)
        printf("%d ", sa[i]);

    return 0;
}
```

#后缀自动机