Contents

Chapter 1

Data Structure

1.1 Segment Tree

1.1.1 Segment Tree & Lazy Propagation

```
class segtree
{
    const static int N=100000;
    int tr[4*N], lazy[4*N];
public:
    segtree(){};
    void clear()
    {
        memset(tr, 0, sizeof(tr));
        memset(lazy, 0, sizeof(lazy));
    }
    void build(int no, int l, int r, vector<int>&data)
    {
        if(l==r)
        {
            tr[no]=data[l];
            return;
        }
        int nxt=no*2;
        int mid=(l+r)/2;
        build(nxt, l, mid, data);
        build(nxt+1, mid+1, r, data);
        tr[no]=tr[nxt]+tr[nxt+1];
    }
    void propagate(int no, int l, int r)
    {
        if(!lazy[no])
            return;

        tr[no]+=(r-l+1)*lazy[no];
        if(l!=r)
        {
            int nxt=no*2;
            lazy[nxt]+=lazy[no];
            lazy[nxt+1]+=lazy[no];
        }
        lazy[no]=0;
    }
```

```
    void update(int no, int l, int r, int i, int j, int x)
    {
        propagate(no, l, r);
        if(l>j || r<i)
            return;
        if(l>=i && r<=j)
        {
            lazy[no]=x;
            propagate(no, l, r);
            return;
        }
        int nxt=no*2;
        int mid=(l+r)/2;
        update(nxt, l, mid, i, j, x);
        update(nxt+1, mid+1, r, i, j, x);
        tr[no]=tr[nxt]+tr[nxt+1];
    }
    int query(int no, int l, int r, int i, int j)
    {
        propagate(no, l, r);
        if(l>j || r<i)
            return 0;
        if(l>=i && r<=j)
            return tr[no];
        int nxt=no*2;
        int mid=(l+r)/2;
        int ql=query(nxt, l, mid, i, j);
        int qr=query(nxt+1, mid+1, r, i, j);
        return (ql+qr);
    }
};
```

1.1.2 Quadtree

```
class quadtree
{
    //needs to be NxN
    const static int N=100000;
    int tr[16*N];
public:
    quadtree(){};
    void build(int node, int l1, int r1, int l2, int r2, vector< vector<int> >data)
    {
        if(l1==l2 && r1==r2)
        {
            tr[node]=data[l1][r1];
            return;
        }
        int nxt=node*4;
        int midl=(l1+l2)/2;
        int midr=(r1+r2)/2;

        build(nxt-2, l1, r1, midl, midr, data);
        build(nxt-1, midl+1, r1, l2, midr, data);
        build(nxt, l1, midr+1, midl, r2, data);
        build(nxt+1, midl+1, midr+1, l2, r2, data);

        tr[node]=tr[nxt-2]+tr[nxt-1]+tr[nxt]+tr[nxt+1];
    }
    void update(int node, int l1, int r1, int l2, int r2, int i, int j, int x)
```

```
    {
        if(l1>l2 || r1>r2)
            return;
        if(i>l2 || j>r2 || i<l1 || j<r1)
            return;
        if(i==l1 && i==l2 && j==r1 && j==r2)
        {
            tr[node]=x;
            return;
        }
        int nxt=node*4;
        int midl=(l1+l2)/2;
        int midr=(r1+r2)/2;

        update(nxt-2, l1, r1, midl, midr, i, j, x);
        update(nxt-1, midl+1, r1, l2, midr, i, j, x);
        update(nxt, l1, midr+1, midl, r2, i, j, x);
        update(nxt+1, midl+1, midr+1, l2, r2, i, j, x);

        tr[node]=tr[nxt-2]+tr[nxt-1]+tr[nxt]+tr[nxt+1];
    }
    int query(int node, int l1, int r1, int l2, int r2, int i1, int j1, int i2, int j2)
    {
        if(i1>l2 || j1>r2 || i2<l1 || j2<r1 || i1>i2 || j1>j2)
            return 0;
        if(i1<=l1 && j1<=r1 && l2<=i2 && r2<=j2)
            return tr[node];
        int nxt=node*4;
        int midl=(l1+l2)/2;
        int midr=(r1+r2)/2;

        int q1=query(nxt-2, l1, r1, midl, midr, i1, j1, i2, j2);
        int q2=query(nxt-1, midl+1, r1, l2, midr, i1, j1, i2, j2);
        int q3=query(nxt, l1, midr+1, midl, r2, i1, j1, i2, j2);
        int q4=query(nxt+1, midl+1, midr+1, l2, r2, i1, j2, i2, j2);
    }
};
```

1.1.3 Mergesort Segtree

```
class mergesort_segtree
{
    const static int N=100000;
    vector<int>tr[4*N];
public:
    mergesort_segtree(){};
    void build(int no, int l, int r, vector<int>&data)
    {
        if(l==r)
        {
            tr[no].push_back(data[l]);
            return;
        }
        int nxt=no*2;
        int mid=(l+r)/2;
        build(nxt, l, mid, data);
        build(nxt+1, mid+1, r, data);
        tr[no].resize(tr[nxt].size()+tr[nxt+1].size());
        merge(tr[nxt].begin(), tr[nxt].end(), tr[nxt+1].begin(), tr[nxt+1].end(), tr[no].begi
    }
```

```
    //how many numbers in (i, j) are greater or equal than k
    int query(int no, int l, int r, int i, int j, int k)
    {
        if(r<i || l>j)
            return 0;
        if(l>=i && r<=j)
            return (int)(tr[no].end()-upper_bound(tr[no].begin(), tr[no].end(), k));
        int nxt=no*2;
        int mid=(l+r)/2;
        int ql=query(nxt, l, mid, i, j, k);
        int qr=query(nxt+1, mid+1, r, i, j, k);
        return ql+qr;
    }
};
```

1.1.4 Persistent Segtree

```
class persistent_segtree
{
    const static int N=100000;
    int n;
    int tr[N];
    int root[N], L[N], R[N];
    int cnt, id;
public:
    persistent_segtree(){};
    void set(int _n)
    {
        memset(tr, 0, sizeof(tr));
        memset(root, 0, sizeof(root));
        memset(L, 0, sizeof(L));
        memset(R, 0, sizeof(R));
        id=0;
        cnt=1;
        n=_n;
    }
    void build(int no, int l, int r, vector<int>&data)
    {
        if(l==r)
        {
            tr[no]=data[l];
            return;
        }
        int mid=(l+r)/2;
        L[no]=cnt++;
        R[no]=cnt++;
        build(L[no], l, mid, data);
        build(R[no], mid+1, r, data);
        tr[no]=tr[ L[no] ]+tr[ R[no] ];
    }
    int update(int no, int l, int r, int i, int x)
    {
        int newno=cnt++;
        tr[newno]=tr[no];
        L[newno]=L[no];
        R[newno]=R[no];
        if(l==r)
        {
            tr[newno]=x;
            return newno;
```

```
        }
        int mid=(l+r)/2;
        if(i<=mid)
            L[newno]=update(L[newno], l, mid, i, x);
        else
            R[newno]=update(R[newno], mid+1, r, i, x);
        tr[newno]=tr[ L[newno] ]+tr[ R[newno] ];
        return newno;
    }
    int query(int no, int l, int r, int i, int j)
    {
        if(r<i || l>j)
            return 0;
        if(l>=i && r<=j)
            return tr[no];
        int mid=(l+r)/2;
        int ql=query(L[no], l, mid, i, j);
        int qr=query(R[no], mid+1, r, i, j);
        return ql+qr;
    }
    //update the i-th value to x.
    void update(int i, int x)
    {
        root[id+1]=update(root[id], 0, n-1, i, x);
    }
    //returns sum(l, r) after the k-th update.
    int query(int l, int r, int k)
    {
        return query(root[k], 0, n-1, l, r);
    }
};
```

1.2 Fenwick Tree

1.2.1 Fenwick Tree 1D

```
class fenwicktree
{
    #define D(x) x&(-x)
    const static int N=100000;
    int tr[N], n;
public:
    fenwicktree(){};
    void build(int _n)
    {
        n=_n;
        memset(tr, 0, sizeof(tr));
    }
    void update(int i, int x)
    {
        for(i++; i<=n; i+=D(i))
            tr[i]+=x;
    }
    int query(int i)
    {
        int ret=0;
        for(i++; i>0; i-=D(i))
            ret+=tr[i];
        return ret;
```

```
    }
    int rquery(int l, int r)
    {
        return query(r)-query(l-1);
    }
    void set(int i, int x)
    {
        update(i, -rquery(i, i)+x);
    }
    void rset(int l, int r, int x)
    {
        update(l, x);
        update(r+1, -x);
    }
};
```

1.2.2 Fenwick Tree 2D

```
class fenwicktree
{
    #define D(x) x&(-x)
    const static int N=1000;
    int tr[N][N], n, m;
public:
    fenwicktree(){};
    void build(int _n, int _m)
    {
        n=_n, m=_m;
        memset(tr, 0, sizeof(tr));
    }
    void update(int r, int c, int x)
    {
        for(int i=r+1; i<=n; i+=D(i))
            for(int j=c+1; j<=m; j+=D(j))
                tr[i][j]+=x;
    }
    int query(int r, int c)
    {
        int ret=0;
        for(int i=r+1; i>0; i-=D(i))
            for(int j=c+1; j>0; j-=D(j))
                ret+=tr[i][j];
        return ret;
    }
    int rquery(int r1, int c1, int r2, int c2)
    {
        if((r1>r2 && c1>c2) || (r1==r2 && c1>c2) || (r1>r2 && c1==c2))
        {
            swap(r1, r2);
            swap(c1, c2);
        }
        else if(r1<r2 && c1>c2)
        {
            swap(c1,c2);
        }
        else if(r1>r2 && c1<c2)
        {
            swap(r1,r2);
        }
        return query(r2, c2)-query(r1-1, c2)-query(r2, c1-1)+query(r1-1, c1-1);
```

```
    }
    void set(int r, int c, int x)
    {
        update(r, c, -rquery(r, c, r, c)+x);
    }
};
```

1.3 Cartesian Tree

1.3.1 Cartesian Tree

```
//srand(time(NULL))
int vrand()
{
    return abs(rand()<<(rand()%31));
}

struct node
{
    //x=key, y=priority key, c=tree count
    int x, y, c;
    node *L, *R;
    node(){};
    node(int _x)
    {
        x=_x, y=vrand(), c=0;
        L=R=NULL;
    }
};

int cnt(node *root)
{
    return root?root->c:0;
}

void upd_cnt(node *root)
{
    if(root)
        root->c=1+cnt(root->L)+cnt(root->R);
}

void split(node *root, int x, node *&L, node *&R)
{
    if(!root)
        L=R=NULL;
    else if(x < root->x)
        split(root->L, x, L, root->L), R=root;
    else
        split(root->R, x, root->R, R), L=root;
    upd_cnt(root);
}

void insert(node *&root, node *it)
{
    if(!root)
        root=it;
    else if(it->y > root->y)
        split(root, it->x, it->L, it->R), root=it;
    else
```

```
        insert(it->x < root->x? root->L:root->R, it);
    upd_cnt(root);
}

void merge(node *&root, node *L, node *R)
{
    if(!L || !R)
        root=L?L:R;
    else if(L->y > R->y)
        merge(L->R, L->R, R), root=L;
    else
        merge(R->L, L, R->L), root=R;
    upd_cnt(root);
}

void erase(node *&root, int x)
{
    if(root->x==x)
        merge(root, root->L, root->R);
    else
        erase(x < root->x? root->L:root->R, x);
    upd_cnt(root);
}

node *unite(node *L, node *R)
{
    if(!L || !R)
        return L?L:R;
    if(L->y < R->y)
        swap(L, R);
    node *Lt, *Rt;
    split(R, L->x, Lt, Rt);
    L->L=unite(L->L, Lt);
    L->R=unite(L->R, Rt);
    return L;
}

int find(node *root, int x)
{
    if(!root)
        return 0;
    if(root->x==x)
        return 1;
    if(x > root->x)
        return find(root->R, x);
    else
        return find(root->L, x);
}

int findkth(node *root, int x)
{
    if(!root)
        return -1;
    int Lc=cnt(root->L);
    if(x-Lc-1==0)
        return root->x;
    if(x>Lc)
        return findkth(root->R, x-Lc-1);
    else
        return findkth(root->L, x);
}
```

1.3.2 Implicit Cartesian Tree

```
//srand(time(NULL))
int vrand()
{
    return abs(rand()<<(rand()%31));
}

struct node
{
    //basic treap: x=key, y=priority key, c=tree count;
    int x, y, c;
    //treap operations: v=max(x), lazy=lazy value of propagation, rev=reversed
    int v, lazy, rev;

    node *L, *R;
    node(){};
    node(int _x)
    {
        x=_x, y=vrand();
        L=R=NULL;
        v=x;
        lazy=0;
        rev=0;
    }
};

//updating functions
inline int get_cnt(node *root)
{
    return root?root->c:0;
}

inline void upd_cnt(node *root)
{
    if(root)
        root->c=1+get_cnt(root->L)+get_cnt(root->R);
}

inline void push(node *&root)
{
    if(root && root->rev)
    {
        root->rev=0;
        swap(root->L, root->R);
        if(root->L)
            root->L->rev^=1;
        if(root->R)
            root->R->rev^=1;
    }
}

inline void propagate(node *&root)
{
    if(root)
    {
        if(!root->lazy)
            return;
        int lazy=root->lazy;
        root->x+=lazy;

        if(root->L)
```

```
            root->L->lazy=lazy;
        if(root->R)
            root->R->lazy=lazy;
        root->lazy=0;
    }
}

inline int get_max(node *root)
{
    return root?root->v:-INF;
}

inline void upd_max(node *root)
{
    if(root)
        root->v=max(root->x, max(get_max(root->L), get_max(root->R)));
}

inline void update(node *root)
{
    propagate(root);
    upd_cnt(root);
    upd_max(root);
}

void merge(node *&root, node *L, node *R)
{
    push(L);
    push(R);
    if(!L || !R)
        root=L?L:R;
    else if(L->y > R->y)
        merge(L->R, L->R, R), root=L;
    else
        merge(R->L, L, R->L), root=R;
    update(root);
}

void split(node *root, node *&L, node *&R, int x, int add=0)
{
    if(!root)
        return void(L=R=NULL);
    push(root);
    int ix=add+get_cnt(root->L); //implicit key
    if(x<=ix)
        split(root->L, L, root->L, x, add), R=root;
    else
        split(root->R, root->R, R, x, add+1+get_cnt(root->L)), L=root;
    update(root);
}

//insert function
void insert(node *&root, int pos, int x)//(insert x at position pos)
{
    node *R1, *R2;
    split(root, R1, R2, pos);
    merge(R1, R1, new node(x));
    merge(root, R1, R2);
}

//erase value x
void erase_x(node *&root, int x)
```

```
{
    if(!root)
        return;
    if(root->x==x)
        merge(root, root->L, root->R);
    else
        erase_x(x < root->x? root->L:root->R, x);
    update(root);
}

//erase kth value
void erase_kth(node *&root, int x)
{
    if(!root)
        return;
    int Lc=get_cnt(root->L);
    if(x-Lc-1==0)
        merge(root, root->L, root->R);
    else if(x>Lc)
        erase_kth(root->R, x-Lc-1);
    else
        erase_kth(root->L, x);
    update(root);
}

//add x to [l,r]
inline void paint(node *&root, int l, int r, int x)
{
    node *R1, *R2, *R3;
    split(root, R1, R2, l);
    split(R2, R2, R3, r-l+1);
    R2->lazy=x;
    propagate(R2);

    merge(root, R1, R2);
    merge(root, root, R3);
}

//max range query [l,r]
inline int rquery(node *&root, int l, int r)
{
    node *R1, *R2, *R3;
    split(root, R1, R2, l);
    split(R2, R2, R3, r-l+1);
    int ret=R2->v;
    merge(root, R1, R2);
    merge(root, root, R3);
    return ret;
}

inline void reverse(node *&root, int l, int r)//reverse elements [l, r]
{
    node *R1, *R2, *R3;
    split(root, R1, R2, l);
    split(R2, R2, R3, r-l+1);
    R2->rev^=1;
    merge(root, R1, R2);
    merge(root, root, R3);
}

//output functions
int poscnt=0;
```

```
void output_all(node *root)
{
    if(!root)
        return;
    update(root);
    push(root);
    output_all(root->L);
    printf("[%d] %d\n", poscnt++, root->x);
    output_all(root->R);
}

int output_kth(node *root, int x)
{
    if(!root)
        return -1;
    update(root);
    push(root);
    int Lc=get_cnt(root->L);
    if(x-Lc-1==0)
        return root->x;
    if(x>Lc)
        return output_kth(root->R, x-Lc-1);
    else
        return output_kth(root->L, x);
}
```

1.4 Merge Sort & Swap Count

1.4.1 Merge Sort & Vector

```
#define INF 0x3F3F3F3F
int mergesort(vector<int>&data)
{
    if(data.size()==1)
        return 0;
    vector<int>L, R;
    int t=data.size();
    for(int i=0; i<t/2; i++)
        L.push_back(data[i]);
    for(int i=t/2; i<t; i++)
        R.push_back(data[i]);
    int ret=mergesort(L)+mergesort(R);
    for(int i=0, j=0, k=0; j<L.size() || k<R.size(); i++)
    {
        int x=j<L.size()?L[j]:INF;
        int y=k<R.size()?R[k]:INF;
        if(x<y)
        {
            data[i]=x;
            j++;
        }
        else
        {
            data[i]=y;
            k++;
            ret+=(L.size()-j);
        }
    }
    return ret;
```

```
}
```

1.4.2 Merge Sort

```
#define INF 0x3F3F3F3F
int temp[100000];
int mergesort(int data[], int l, int r)
{
    if(abs(l-r)<=1)
        return 0;
    int mid=(l+r)/2;
    int ret=mergesort(data, l, mid)+mergesort(data, mid, r);
    for(int i=l; i<r; i++)
        temp[i]=data[i];
    for(int i=l, j=l, k=mid; j<mid || k<r; i++)
    {
        int x=j<mid?temp[j]:INF;
        int y=k<r?temp[k]:INF;
        if(x<y)//x<=y
        {
            data[i]=x;
            j++;
        }
        else
        {
            data[i]=y;
            k++;
            ret+=(mid-j);
        }
    }
    return ret;
}
```

1.5 Sparse Table

```
class sparsetable
{
    #define lbit(x) 63-__builtin_clzll(x);
    const static int N=100000, LN=20;
    int data[N][LN], n, ln;
public:
    sparsetable(){};
    void clear()
    {
        memset(data, 0, sizeof(data));
    }
    void build(vector<int>&foo)
    {
        n=foo.size();
        ln=lbit(n);
        for(int i=0; i<n; i++)
            data[i][0]=foo[i];
        for(int j=1; j<=ln; j++)
            for(int i=0; i<n-(1<<j)+1; i++)
                data[i][j]=max(data[i][j-1], data[i+(1<<(j-1))][j-1]);
    }
    int query(int l, int r)
```

```
    {
        int i=abs(l-r)+1;
        int j=lbit(i);
        return max(data[l][j], data[l-(1<<j)+1][j]);
    }
};
```

1.6 SQRT Decomposition

1.6.1 Array

```
const int N=100000;
int SN=sqrt(N);

class mo
{
public:
    int l, r, i;
    mo(){};
    mo(int _l, int _r, int _i)
    {
        l=_l, r=_r, i=_i;
    }
    bool operator <(const mo &foo) const
    {
        if((r/SN)!=(foo.r/SN))
            return (r/SN)<(foo.r/SN);
        if(l!=foo.l)
            return l<foo.l;
        return i<foo.i;
    }
};

int data[N], freq[N], ans[N];
int cnt=0;
void update(int p, int s)
{
    int x=data[p];
    if(s==1)
    {
        if(freq[x]==0)
            cnt++;
    }
    else
    {
        if(freq[x]==1)
            cnt--;
    }
    freq[x]+=s;
}

int main()
{
    int n;
    scanf("%d", &n);
    for(int i=1; i<=n; i++)
        scanf("%d", &data[i]);

    int q;
```

```
    scanf("%d", &q);
    vector<mo>querys;
    for(int i=0; i<q; i++)
    {
        int l, r;
        scanf("%d %d", &l, &r);
        querys.push_back(mo(l, r, i));
    }
    sort(querys.begin(), querys.end());

    int l=1, r=1;
    cnt=0;
    memset(freq, 0, sizeof(freq));
    update(l, 1);
    for(int i=0; i<q; i++)
    {
        int li=querys[i].l;
        int ri=querys[i].r;
        int ii=querys[i].i;
        while(l>li)
            update(--l, 1);
        while(r<ri)
            update(++r, 1);
        while(l<li)
            update(l++, -1);
        while(r>ri)
            update(r--, -1);
        ans[ii]=cnt;
    }
    for(int i=0; i<querys.size(); i++)
        printf("%d\n", ans[i]);
    return 0;
}
```

1.6.2 Tree

```
#define pb push_back
#define ALL(x) x.begin(),x.end()

const int N=1e+5+35;
const int M=20;
const int SN=sqrt(2*N)+1;

class mo
{
public:
    int l, r, i, lc;
    mo(){};
    mo(int _l, int _r, int _lc, int _i)
    {
        l=_l, r=_r, lc=_lc, i=_i;
    }
    bool operator <(const mo &foo) const
    {
        if((r/SN)!=(foo.r/SN))
            return (r/SN)<(foo.r/SN);
        if(l!=foo.l)
            return l<foo.l;
        return i<foo.i;
    }
```

```
};

int n, q;
int h[N], lca[N][M];
vector<int>g[N];
int dl[N], dr[N], di[2*N], cur;

void dfs(int u, int p)
{
    dl[u]=++cur;
    di[cur]=u;
    lca[u][0]=p;
    for(int i=1; i<M; i++)
        lca[u][i]=lca[ lca[u][i-1] ][i-1];
    for(int i=0; i<g[u].size(); i++)
    {
        int v=g[u][i];
        if(v==p)
            continue;
        h[v]=h[u]+1;
        dfs(v, u);
    }
    dr[u]=++cur;
    di[cur]=u;
}

inline int getLca(int u, int v)
{
    if(h[u]>h[v])
        swap(u, v);
    for(int i=M-1; i>=0; i--)
        if(h[v]-(1<<i)>=h[u])
            v=lca[v][i];
    if(u==v)
        return u;
    for(int i=M-1; i>=0; i--)
    {
        if(lca[u][i]!=lca[v][i])
        {
            u=lca[u][i];
            v=lca[v][i];
        }
    }
    return lca[u][0];
}

map<string,int>remap;
int data[N], ans[N], vis[N], freq[N], cnt;
inline void update(int u)
{
    int x=data[u];
    if(vis[u] && (--freq[ data[u] ]==0))
        cnt--;
    else if(!vis[u] && (freq[ data[u] ]++==0))
        cnt++;
    vis[u]^=1;
}

int main()
{
    scanf("%d %d", &n, &q);
    for(int i=1; i<=n; i++)
```

```
    {
        char temp[25];
        scanf("%s", temp);
        string temp2=string(temp);
        if(!remap.count(temp2))
            remap[temp2]=remap.size();
        data[i]=remap[temp2];
    }
    for(int i=1; i<n; i++)
    {
        int u, v;
        scanf("%d %d", &u, &v);
        g[u].pb(v);
        g[v].pb(u);
    }
    dfs(1, 0);

    vector<mo>query;
    for(int i=0; i<q; i++)
    {
        int u, v;
        scanf("%d %d", &u, &v);
        int lc=getLca(u, v);
        if(dl[u]>dl[v])
            swap(u, v);
        query.pb(mo(u==lc?dl[u]:dr[u], dl[v], lc, i));
    }
    sort(ALL(query));

    int l=query[0].l, r=query[0].l-1;
    cnt=0;
    for(int i=0; i<q; i++)
    {
        int li=query[i].l;
        int ri=query[i].r;
        int lc=query[i].lc;
        int ii=query[i].i;
        while(l>li)
            update(di[--l]);
        while(r<ri)
            update(di[++r]);
        while(l<li)
            update(di[l++]);
        while(r>ri)
            update(di[r--]);

        int u=di[l], v=di[r];
        if(lc!=u && lc!=v)
            update(lc);
        ans[ii]=cnt;
        if(lc!=u && lc!=v)
            update(lc);
    }
    for(int i=0; i<q; i++)
        printf("%d\n", ans[i]);
    return 0;
}
```

Chapter 2

Graph

2.1 Components

2.1.1 Articulations, Bridges & Cycles

2.1.2 Strongly Connected Components

2.1.3 Semi-Strongly Connected Components

2.2 Single Source Shortest Path

2.2.1 Dijkstra

2.2.2 Bellmanford

2.3 All Pairs Shortest Path

2.3.1 Floyd Warshall

2.4 Minimum Spannig Tree

2.4.1 Kruskal

2.4.2 Prim

2.5 Flow

2.5.1 Maximum Bipartite Matching

```
const int MN=1e+3;
vector<int>g[MN];
int match[MN], rmatch[MN], vis[MN];
int findmatch(int u)
{
    if(vis[u])
        return 0;
    vis[u]=true;
    for(int v:g[u])
```

```
    {
        if(match[v]==-1 || findmatch(match[v]))
        {
            match[v]=u;
            rmatch[u]=v;
            return 1;
        }
    }
    return 0;
}

int maxMatch(int n)
{
    int ret=0;
    memset(match, -1, sizeof(match));
    for(int i=0; i<n; i++)
    {
        memset(vis, false, sizeof(vis));
        ret+=findmatch(i);
    }
    return ret;
}
```

2.5.2 Maximum Flow

Dinic

```
class graph
{
    const static int N=100000;
public:
    vector< pair<int,int> >edge;
    vector<int>adj[N];
    int ptr[N];
    int dist[N];

    graph(){};
    void clear()
    {
        for(int i=0; i<N; i++)
            adj[i].clear();
        edge.clear();
    }
    void add_edge(int u, int v, int c)
    {
        adj[u].push_back(edge.size());
        edge.push_back(mp(v, c));
        adj[v].push_back(edge.size());
        edge.push_back(mp(u, 0)); //(u, c) if is non-directed
    }
    bool dinic_bfs(int s, int t)
    {
        memset(dist, -1, sizeof(dist));
        dist[s]=0;

        queue<int>bfs;
        bfs.push(s);
        while(!bfs.empty() && dist[t]==-1)
        {
            int u=bfs.front();
```

```
            bfs.pop();
            for(int i=0; i<adj[u].size(); i++)
            {
                int idx=adj[u][i];
                int v=edge[idx].F;

                if(dist[v]==-1 && edge[idx].S>0)
                {
                    dist[v]=dist[u]+1;
                    bfs.push(v);
                }
            }
        }
        return dist[t]!=-1;
    }
    int dinic_dfs(int u, int t, int flow)
    {
        if(u==t)
            return flow;
        for(int &i=ptr[u]; i<adj[u].size(); i++)
        {
            int idx=adj[u][i];
            int v=edge[idx].F;
            if(dist[v]==dist[u]+1 && edge[idx].S>0)
            {
                int cf=dinic_dfs(v, t, min(flow, edge[idx].S));
                if(cf>0)
                {
                    edge[idx].S-=cf;
                    edge[idx^1].S+=cf;
                    return cf;
                }
            }
        }
        return 0;
    }
    int maxflow(int s, int t)
    {
        int ret=0;
        while(dinic_bfs(s, t))
        {
            memset(ptr, 0, sizeof(ptr));
            int cf=dinic_dfs(s, t, INF);
            if(cf==0)
                break;
            ret+=cf;
        }
        return ret;
    }
};
```

2.5.3 Minimum Cost Maximum Flow

Dijkstra

```
/*
undirected graph:
u->uu(flow, 0)
uu->vv(flow, cost)
vv->v(flow, 0)
```

```
v->xx(flow, 0)
vv->u(flow, 0)
*/
typedef int FTYPE; //type of flow
typedef int CTYPE; //type of cost
typedef pair<FTYPE,CTYPE>pfc;
const CTYPE CINF=INF;
const FTYPE FINF=INF;

void operator+=(pfc &p1, pfc &p2)
{
    p1.F+=p2.F;
    p1.S+=p2.S;
}

class graph
{
    const static int MN=1e+4;
public:
    int n;
    FTYPE flow[MN];
    CTYPE dist[MN], pot[MN];
    int prev[MN], eidx[MN];

    struct Edge
    {
        int to;
        FTYPE cap;
        CTYPE cost;
        Edge(){};
        Edge(int _to, FTYPE _cap, CTYPE _cost)
        {
            to=_to;
            cap=_cap;
            cost=_cost;
        }//
    };
    struct node
    {
        int u;
        CTYPE d;
        node(){};
        node(int _u, CTYPE _d)
        {
            u=_u;
            d=_d;
        }
        bool operator <(const node &foo) const
        {
            return d>foo.d;
        }
    };
    graph(){};
    vector<int>adj[MN];
    vector<Edge>edge;
    inline void set(int _n)
    {
        n=_n;
    }
    inline void reset()
    {
        for(int i=0; i<MN; i++)
```

```
        adj[i].clear();
    edge.clear();
}
inline void add_edge(int u, int v, FTYPE c, FTYPE cst)
{
    adj[u].push_back(edge.size());
    edge.push_back(Edge(v, c, cst));
    adj[v].push_back(edge.size());
    edge.push_back(Edge(u, 0, -cst));
}

pfc dijkstra(int s, int t)
{
    for(register int i=0; i<n; i++)
        dist[i]=CINF;
    dist[s]=0;
    flow[s]=FINF;
    priority_queue<node>heap;
    heap.push(node(s, 0));
    while(!heap.empty())
    {
        int u=heap.top().u;
        CTYPE d=heap.top().d;
        heap.pop();
        if(d>dist[u])
            continue;
        for(int i=0; i<adj[u].size(); i++)
        {
            int idx=adj[u][i];
            int v=edge[idx].to;
            CTYPE w=edge[idx].cost;
            if(!edge[idx].cap || dist[v]<=d+w+pot[u]-pot[v])
                continue;
            if(d+w<dist[v])
            {
                dist[v]=d+w;
                prev[v]=u;
                eidx[v]=idx;
                flow[v]=min(flow[u], edge[idx].cap);
                heap.push(node(v, d+w));
            }
        }
    }
    if(dist[t]==CINF)
        return mp(FINF, CINF);
    pfc ret=mp(flow[t], 0);
    for(int u=t; u!=s; u=prev[u])
    {
        int idx=eidx[u];
        edge[idx].cap-=flow[t];
        edge[idx^1].cap+=flow[t];
        ret.second+=flow[t]*edge[idx].cost;
    }
    return ret;
}

inline pfc mfmc(int s, int t)
{
    pfc ret=mp(0, 0);
    pfc got;
    while((got=dijkstra(s, t)).first!=FINF)
        ret+=got;
```

```
        return ret;
    }
};
```

Bellmanford

2.5.4 Minimum Cut

2.6 Tree

2.6.1 Lowest Common Ancestor

2.6.2 Centroid Decomposition

```
const int N=1e+5;
const int M=log2(N)+1;

set<int>g[N]; //graph
int h[N]; //heigh of nodes
int trSz[N], sz; //tree subsize, size of current tree
int lca[N][M]; //lca sparse table
int cg[N]; //centroid graph

void dfs(int u, int l)
{
    lca[u][0]=l;
    for(int i=1; i<M; i++)
        lca[u][i]=lca[ lca[u][i-1] ][i-1];
    for(auto v:g[u])
    {
        if(v==l)
            continue;
        h[v]=h[u]+1;
        dfs(v, u);
    }
}

inline int getLca(int u, int v)
{
    if(h[u]>h[v])
        swap(u, v);
    for(int i=M-1; i>=0; i--)
        if(h[v]-(1<<i)>=h[u])
            v=lca[v][i];
    if(u==v)
        return u;
    for(int i=M-1; i>=0; i--)
    {
        if(lca[u][i]!=lca[v][i])
        {
            u=lca[u][i];
            v=lca[v][i];
        }
    }
    return lca[u][0];
}

inline int getDist(int u, int v)
{
```

```
    return h[u]+h[v]-2*h[getLca(u, v)];
}

void centDfs(int u, int l)
{
    trSz[u]=1;
    sz++;
    for(auto v:g[u])
    {
        if(v==l)
            continue;
        centDfs(v, u);
        trSz[u]+=trSz[v];
    }
}

int findCentroid(int u, int l)
{
    for(auto v:g[u])
    {
        if(v==l)
            continue;
        if(trSz[v]*2>=sz)
            return findCentroid(v, u);
    }
    return u;
}

inline void buildCentroid(int u, int l)
{
    sz=0;
    centDfs(u, u);
    int c=findCentroid(u, u); //actual centroid
    cg[c]=(u==l?c:l);
    for(auto v:g[c])
    {
        g[v].erase(g[v].find(c));
        buildCentroid(v, c);
    }
    g[c].clear();
}
```

2.6.3 Heavy Light Decomposition on Edges

```
class segtree
{
    const static int N=1e+5;
public:
    int tr[4*N];
    segtree(){};
    void reset()
    {
        memset(tr, 0, sizeof(tr));
    }
    void update(int no, int l, int r, int i, int val)
    {
        if(r<i || l>i)
            return;
        if(l>=i && r<=i)
        {
```

```
            tr[no]=val;
            return;
        }
        int nxt=(no<<1);
        int mid=(l+r)>>1;
        update(nxt, l, mid, i, val);
        update(nxt+1, mid+1, r, i, val);
        tr[no]=tr[nxt]+tr[nxt+1];
    }
    int query(int no, int l, int r, int i, int j)
    {
        if(r<i || l>j)
            return 0;
        if(l>=i && r<=j)
            return tr[no];
        int nxt=(no<<1);
        int mid=(l+r)>>1;
        return query(nxt, l, mid, i, j)+query(nxt+1, mid+1, r, i, j);
    }
};

const int N=1e+5;
const int M=log2(N)+1;
int n;
segtree tr;
vector< pair<int,int> >g[N];
int lca[N][M];
int h[N], trSz[N];

//in - use X[], Y[] in case
//of edge weights
int X[N], Y[N], W[N];

//hld
int chainInd[N], chainSize[N], chainHead[N], chainPos[N], chainNo, posInBase[N];
int ptr;

void dfs(int u, int l)
{
    trSz[u]=1;
    lca[u][0]=l;
    for(int i=1; i<M; i++)
        lca[u][i]=lca[ lca[u][i-1] ][i-1];
    for(int i=0; i<g[u].size(); i++)
    {
        int v=g[u][i].first;
        if(v==l)
            continue;
        h[v]=h[u]+1;
        dfs(v, u);
        trSz[u]+=trSz[v];
    }
}

inline int getLca(int u, int v)
{
    if(h[u]>h[v])
        swap(u, v);
    for(int i=M-1; i>=0; i--)
        if(h[v]-(1<<i)>=h[u])
            v=lca[v][i];
    if(u==v)
```

```
        return u;
    for(int i=M-1; i>=0; i--)
    {
        if(lca[u][i]!=lca[v][i])
        {
            u=lca[u][i];
            v=lca[v][i];
        }
    }
    return lca[u][0];
}

//dont use 'c' if the weight is on the vertex
//instead of the edge
inline void hld(int u, int l, int c)
{
    if(chainHead[chainNo]==-1)
        chainHead[chainNo]=u;
    chainInd[u]=chainNo;
    chainPos[u]=chainSize[chainNo]++;
    tr.update(1, 0, n, ptr, c);
    posInBase[u]=ptr++;

    int msf, idx;
    msf=idx=-1;
    for(int i=0; i<g[u].size(); i++)
    {
        int v=g[u][i].first;
        if(v==l)
            continue;
        if(trSz[v]>msf)
        {
            msf=trSz[v];
            idx=i;
        }
    }
    if(idx>=0)
        hld(g[u][idx].first, u, g[u][idx].second);
    for(int i=0; i<g[u].size(); i++)
    {
        if(i==idx)
            continue;
        int v=g[u][i].first;
        int w=g[u][i].second;
        if(v==l)
            continue;
        chainNo++;
        hld(v, u, w);
    }
}

inline int query_up(int u, int v)
{
    int uchain=chainInd[u];
    int vchain=chainInd[v];
    int ret=0;
    while(true)
    {
        uchain=chainInd[u];
        if(uchain==vchain)
        {
            ret+=tr.query(1, 0, n, posInBase[v]+1, posInBase[u]);
```

```
            break;
        }
        int head=chainHead[uchain];
        ret+=tr.query(1, 0, n, posInBase[head],posInBase[u]);
        u=head;
        u=lca[u][0];
    }
    return ret;
}

//returns sum of all edges weights
//from 'u' to 'v'
inline int query(int u, int v)
{
    if(u==v)
        return 0;
    int l=getLca(u, v);
    return query_up(u, l)+query_up(v, l);
}

//set and edge to value 'val'
inline void update(int u, int val)
{
    int x=X[u], y=Y[u];
    if(lca[x][0]==y)
        tr.update(1, 0, n, posInBase[x], val);
    else
        tr.update(1, 0, n, posInBase[y], val);
}

void clearHld()
{
    //tr.reset();
    for(int i=0; i<=n; i++)
    {
        g[i].clear();
        chainHead[i]=-1;
        chainSize[i]=0;
    }
    ptr=1;
    chainNo=0;
}

int main()
{
    scanf("%d", &n);
    clearHld();
    for(int i=1; i<n; i++)
    {
        scanf("%d %d %d", &X[i], &Y[i], &W[i]);
        g[ X[i] ].push_back({Y[i], W[i]});
        g[ Y[i] ].push_back({X[i], W[i]});
    }
    dfs(1, 0);
    hld(1, 0, 0);
    int q;
    scanf("%d", &q);
    while(q--)
    {
        int o, x, y;
        scanf("%d %d %d", &o, &x, &y);
        if(o==1)
```

```
            printf("%d\n", query(x, y));
        else
            update(x, y);
    }
    return 0;
}
```

2.6.4 Heavy Light Decomposition on Vertex

2.6.5 All-Pairs Distance Sum

2.7 MISC

2.7.1 2-SAT

Chapter 3

Dynamic Programming

3.1 Optimizations

3.1.1 Divide and Counquer

```
/// David Mateus Batista <david.batista3010@gmail.com>
/// Computer Science - Federal University of Itajuba - Brazil
/// Uri Online Judge - 2475
#include <bits/stdc++.h>

using namespace std;

typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
typedef pair<int,int> pii;
typedef pair<ll,ll> pll;

#define INF 0x3F3F3F3F
#define LINF 0x3F3F3F3F3F3F3F3FLL
#define DINF (double)1e+30
#define EPS (double)1e-9
#define PI (double)acos(-1.0)
#define RAD(x) (double)(x*PI)/180.0
#define PCT(x,y) (double)x*100.0/y
#define pb push_back
#define mp make_pair
#define pq priority_queue
#define F first
#define S second
#define D(x) x&(-x)
#define ALL(x) x.begin(),x.end()
#define SET(a,b) memset(a, b, sizeof(a))
#define DEBUG(x,y) cout << x << y << endl
#define gcd(x,y) __gcd(x, y)
#define lcm(x,y) (x/gcd(x,y))*y
#define bitcnt(x) __builtin_popcountll(x)
#define lbit(x) 63-__builtin_clzll(x)
#define zerosbitll(x) __builtin_ctzll(x)
#define zerosbit(x) __builtin_ctz(x)

enum {North, East, South, West};
//{0, 1, 2, 3}
```

```
//{Up, Right, Down, Left}

int mi[] = {-1, 0, 1, 0, -1, 1, 1, -1};
int mj[] = {0, 1, 0, -1, 1, 1, -1, -1};

const int MN=1e+4+35;
const int MN2=535;
int p, a;
ll data[MN];

inline ll getValue(int l, int r)
{
    return (r-l+1)*(data[r]-data[l-1]);
}

ll dp[MN2][MN];
inline void solve(int k, int l, int r, int L, int R)
{
    if(l>r)
        return;
    int m=(l+r)/2;
    int s=L;
    dp[k][m]=LINF;
    for(int i=max(m, L); i<=R; i++)
    {
        if(dp[k][m]>dp[k-1][i+1]+getValue(m+1, i+1))
        {
            dp[k][m]=dp[k-1][i+1]+getValue(m+1, i+1);
            s=i;
        }
    }
    solve(k, l, m-1, L, s);
    solve(k, m+1, r, s, R);
}

int main()
{
    scanf("%d %d", &p, &a);
    for(int i=1; i<=p; i++)
    {
        ll x;
        scanf("%lld", &x);
        data[i]=data[i-1]+x;
    }

    for(int i=0; i<=p; i++)
        dp[0][i]=LINF;
    for(int i=0; i<=a; i++)
        dp[i][p]=0;
    for(int i=1; i<=a; i++)
        solve(i, 0, p-1, 0, p-1);
    printf("%lld\n", dp[a][0]);
    return 0;
}
```

3.1.2 Convex Hull I

Linear

```
//Original recurrence:
```

```
//     dp[i]=min(dp[j]+b[j]*a[i]) for j<i
//Condition:
//     b[j]>=b[j+1]
//     a[i]<=a[i+1]
// Solution:
// Hull cht=Hull();
// cht.insertLine(b[0], dp[0])
// for(int i=1; i<n; i++)
// {
//     dp[i]=cht.query(a[i]);
//     cht.insertLine(b[i], dp[i])
// }
// answer is dp[n-1]

class Hull
{
    const static int CN=1e+5+35;
public:
    long long a[CN], b[CN];
    double x[CN];
    int head, tail;
    Hull():head(1), tail(0){};

    long long query(long long xx)
    {
        if(head>tail)
            return 0;
        while(head<tail && x[head+1]<=xx)
            head++;
        x[head]=xx;
        return a[head]*xx+b[head];
    }

    void insertLine(long long aa, long long bb)
    {
        double xx=-1e18;
        while(head<=tail)
        {
            if(aa==a[tail])
                return;
            xx=1.0*(b[tail]-bb)/(aa-a[tail]);
            if(head==tail || xx>=x[tail])
                break;
            tail--;
        }
        a[++tail]=aa;
        b[tail]=bb;
        x[tail]=xx;
    }
};
```

Dynamic

```
//Original recurrence:
//     dp[i]=min(dp[j]+b[j]*a[i]) for j<i
//Condition:
//     b[j]>=b[j+1]
//     a[i]<=a[i+1]
// Solution:
// HullDynamic cht;
// cht.insertLine(b[0], dp[0])
```

```
// for(int i=1; i<n; i++)
// {
//     dp[i]=cht.query(a[i]);
//     cht.insertLine(b[i], dp[i])
// }
// answer is dp[n-1]

const long long is_query=-(1LL<<62);
class Line
{
public:
    long long m, b;
    mutable function<const Line*()>succ;
    bool operator < (const Line &rhs) const
    {
        if(rhs.b!=is_query)
            return m<rhs.m;
        const Line *s=succ();
        if(!s)
            return 0;
        long long x=rhs.m;
        return (b-s->b)<((s->m-m)*x);
    }
};

class HullDynamic: public multiset<Line>
{
public:
    void clear()
    {
        clear();
    }
    bool bad(iterator y)
    {
        auto z=next(y);
        if(y==begin())
        {
            if(z==end())
                return 0;
            return (y->m==z->m && y->b<=z->b);
        }
        auto x=prev(y);
        if(z==end())
            return (y->m == x->m && y->b<=x->b);
        return ((x->b - y->b)*(z->m - y->m) >= (y->b - z->b)*(y->m - x->m));
    }
    void insertLine(ll m, ll b)
    {
        auto y=insert({m, b});
        y->succ=[=]
        {
            return next(y)==end()?0:&*next(y);
        };
        if(bad(y))
        {
            erase(y);
            return;
        }
        while(next(y)!=end() && bad(next(y)))
            erase(next(y));
        while(y!=begin() && bad(prev(y)))
            erase(prev(y));
```

```
    }
    long long query(long long x)
    {
        auto ret=*lower_bound((Line){x, is_query});
        return ret.m*x+ret.b;
    }
};
```

3.1.3 Convex Hull II

3.1.4 Knuth Optimization

3.2 Digits

3.3 Grundy Numbers

Chapter 4

String

4.1 Hash

4.2 KMP

4.3 Aho Corasick

4.4 Manacher

4.5 Z-Algorithm

4.6 Suffix Array & LCP

4.7 Suffix Tree

Chapter 5

Mathematic

5.1 Prime Numbers

5.1.1 Erastotenes Sieve

5.1.2 Linear Sieve

5.1.3 Miller Rabin

5.1.4 BPSW

5.1.5 Primality Test

5.2 Chinese Remainder Theorem

5.3 Fast Fourier Transformation

```
#define PI (double)acos(-1.0)
typedef complex<double> base;
void fft(vector<base>&data, bool invert)
{
    int n=data.size();
    for(int i=1, j=0; i<n; i++)
    {
        int bit=n>>1;
        for(; j>=bit; bit>>=1)
            j-=bit;
        j+=bit;
        if(i<j)
            swap(data[i], data[j]);
    }

    for(int len=2; len<=n; len<<=1)
    {
        double ang=2*PI/len*(invert?-1:1);
        base wlen(cos(ang), sin(ang));
        for(int i=0; i<n; i+=len)
        {
            base w(1);
            for(int j=0; j<len/2; j++)
            {
```

```
                base u=data[i+j], v=data[i+j+len/2]*w;
                data[i+j]=u+v;
                data[i+j+len/2]=u-v;
                w*=wlen;
            }
        }
    }
    if(invert)
        for(int i=0; i<n; i++)
            data[i]/=n;
}

vector<int>fft_multiply(vector<int>&a, vector<int>&b)
{
    vector<base>fa(a.begin(), a.end());
    vector<base>fb(b.begin(), b.end());
    int n=1;
    while(n<max(a.size(), b.size()))
        n<<=1;
    n<<=1;
    fa.resize(n);
    fb.resize(n);
    fft(fa, false);
    fft(fb, false);
    for(int i=0; i<n; i++)
        fa[i]*=fb[i];
    fft(fa, true);

    vector<int>ret(n);
    for(int i=0; i<n; i++)
        ret[i]=(int)(fa[i].real()+0.5);

    int carry=0;
    for(int i=0; i<n; i++)
    {
        ret[i]+=carry;
        carry=ret[i]/10;
        ret[i]%=10;
    }
    return ret;
}

int main()
{
    int n, m;
    scanf("%d %d", &n, &m);
    vector<int>a,b;

    for(int i=0; i<n; i++)
    {
        int x;
        scanf("%d", &x);
        a.pb(x);
    }

    for(int i=0; i<m; i++)
    {
        int x;
        scanf("%d", &x);
        b.pb(x);
    }
    reverse(a.begin(), a.end());
```

```
    reverse(b.begin(), b.end());

    vector<int>ans=fft_multiply(a, b);
    reverse(ans.begin(), ans.end());
    bool flag=false;
    for(int i=0; i<ans.size(); i++)
    {
        if(ans[i])
            flag=true;
        if(flag)
            printf("%d", ans[i]);
    }
    printf("\n");
    return 0;
}
```

5.4 Modular Math

5.4.1 Multiplicative Inverse

5.4.2 Linear All Multiplicative Inverse

5.5 Gaussian Elimination

5.6 Combinatorics

Chapter 6

Geometry

6.1 2d Template

6.2 3d Template

6.3 Polygon Template

6.4 Convex Hull

6.4.1 Graham Scan

6.4.2 Monotone Chain

6.5 Rotating Calipers

6.6 KD Tree

6.7 Range Tree

6.8 Circle Sweep