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].begin());
        }
        //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(){};
```

```
13          node(int _x)
14          {
15                  x=_x, y=vrand(), c=0;
16                  L=R=NULL;
17          }
18  };
19
20  int cnt(node *root)
21  {
22          return root?root->c:0;
23  }
24
25  void upd_cnt(node *root)
26  {
27          if(root)
28                  root->c=1+cnt(root->L)+cnt(root->R);
29  }
30
31  void split(node *root, int x, node *&L, node *&R)
32  {
33          if(!root)
34                  L=R=NULL;
35          else if(x < root->x)
36                  split(root->L, x, L, root->L), R=root;
37          else
38                  split(root->R, x, root->R, R), L=root;
39          upd_cnt(root);
40  }
41
42  void insert(node *&root, node *it)
43  {
44          if(!root)
45                  root=it;
46          else if(it->y > root->y)
47                  split(root, it->x, it->L, it->R), root=it;
48          else
49                  insert(it->x < root->x? root->L:root->R, it);
50          upd_cnt(root);
51  }
52
53  void merge(node *&root, node *L, node *R)
54  {
55          if(!L || !R)
56                  root=L?L:R;
57          else if(L->y > R->y)
58                  merge(L->R, L->R, R), root=L;
59          else
60                  merge(R->L, L, R->L), root=R;
61          upd_cnt(root);
62  }
63
64  void erase(node *&root, int x)
65  {
66          if(root->x==x)
67                  merge(root, root->L, root->R);
68          else
69                  erase(x < root->x? root->L:root->R, x);
70          upd_cnt(root);
71  }
72
73  node *unite(node *L, node *R)
74  {
75          if(!L || !R)
76                  return L?L:R;
77          if(L->y < R->y)
78                  swap(L, R);
79          node *Lt, *Rt;
80          split(R, L->x, Lt, Rt);
81          L->L=unite(L->L, Lt);
82          L->R=unite(L->R, Rt);
83          return L;
84  }
85
86  int find(node *root, int x)
87  {
88          if(!root)
89                  return 0;
90          if(root->x==x)
91                  return 1;
```

```
92          if(x > root->x)
93                  return find(root->R, x);
94          else
95                  return find(root->L, x);
96  }
97
98  int findkth(node *root, int x)
99  {
100         if(!root)
101                 return -1;
102         int Lc=cnt(root->L);
103         if(x-Lc-1==0)
104                 return root->x;
105         if(x>Lc)
106                 return findkth(root->R, x-Lc-1);
107         else
108                 return findkth(root->L, x);
109 }
```

1.3.2 Implicit Cartesian Tree

```
1   //srand(time(NULL))
2   int vrand()
3   {
4           return abs(rand()<<(rand()%31));
5   }
6
7   struct node
8   {
9           //basic treap: x=key, y=priority key, c=tree count;
10          int x, y, c;
11          //treap operations: v=max(x), lazy=lazy value of propagation, rev=reversed
12          int v, lazy, rev;
13
14          node *L, *R;
15          node(){};
16          node(int _x)
17          {
18                  x=_x, y=vrand();
19                  L=R=NULL;
20                  v=x;
21                  lazy=0;
22                  rev=0;
23          }
24  };
25
26  //updating functions
27  inline int get_cnt(node *root)
28  {
29          return root?root->c:0;
30  }
31
32  inline void upd_cnt(node *root)
33  {
34          if(root)
35                  root->c=1+get_cnt(root->L)+get_cnt(root->R);
36  }
37
38  inline void push(node *&root)
39  {
40          if(root && root->rev)
41          {
42                  root->rev=0;
43                  swap(root->L, root->R);
44                  if(root->L)
45                          root->L->rev^=1;
46                  if(root->R)
47                          root->R->rev^=1;
48          }
49  }
50
51  inline void propagate(node *&root)
52  {
53          if(root)
54          {
55                  if(!root->lazy)
56                          return;
57                  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)
```

```
137                     return;
138             int Lc=get_cnt(root->L);
139             if(x-Lc-1==0)
140                     merge(root, root->L, root->R);
141             else if(x>Lc)
142                     erase_kth(root->R, x-Lc-1);
143             else
144                     erase_kth(root->L, x);
145             update(root);
146     }
147
148     //add x to [l,r]
149     inline void paint(node *&root, int l, int r, int x)
150     {
151             node *R1, *R2, *R3;
152             split(root, R1, R2, l);
153             split(R2, R2, R3, r-l+1);
154             R2->lazy=x;
155             propagate(R2);
156
157             merge(root, R1, R2);
158             merge(root, root, R3);
159     }
160
161     //max range query [l,r]
162     inline int rquery(node *&root, int l, int r)
163     {
164             node *R1, *R2, *R3;
165             split(root, R1, R2, l);
166             split(R2, R2, R3, r-l+1);
167             int ret=R2->v;
168             merge(root, R1, R2);
169             merge(root, root, R3);
170             return ret;
171     }
172
173     inline void reverse(node *&root, int l, int r)//reverse elements [l, r]
174     {
175             node *R1, *R2, *R3;
176             split(root, R1, R2, l);
177             split(R2, R2, R3, r-l+1);
178             R2->rev^=1;
179             merge(root, R1, R2);
180             merge(root, root, R3);
181     }
182
183     //output functions
184     int poscnt=0;
185     void output_all(node *root)
186     {
187             if(!root)
188                     return;
189             update(root);
190             push(root);
191             output_all(root->L);
192             printf("[%d] %d\n", poscnt++, root->x);
193             output_all(root->R);
194     }
195
196     int output_kth(node *root, int x)
197     {
198             if(!root)
199                     return -1;
200             update(root);
201             push(root);
202             int Lc=get_cnt(root->L);
203             if(x-Lc-1==0)
204                     return root->x;
205             if(x>Lc)
206                     return output_kth(root->R, x-Lc-1);
207             else
208                     return output_kth(root->L, x);
209     }
```

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);
```

```
46          }
47          dr[u]=++cur;
48          di[cur]=u;
49  }
50
51  inline int getLca(int u, int v)
52  {
53          if(h[u]>h[v])
54                  swap(u, v);
55          for(int i=M-1; i>=0; i--)
56                  if(h[v]-(1<<i)>=h[u])
57                          v=lca[v][i];
58          if(u==v)
59                  return u;
60          for(int i=M-1; i>=0; i--)
61          {
62                  if(lca[u][i]!=lca[v][i])
63                  {
64                          u=lca[u][i];
65                          v=lca[v][i];
66                  }
67          }
68          return lca[u][0];
69  }
70
71  map<string,int>remap;
72  int data[N], ans[N], vis[N], freq[N], cnt;
73  inline void update(int u)
74  {
75          int x=data[u];
76          if(vis[u] && (--freq[ data[u] ]==0))
77                  cnt--;
78          else if(!vis[u] && (freq[ data[u] ]++==0))
79                  cnt++;
80          vis[u]^=1;
81  }
82
83  int main()
84  {
85          scanf("%d %d", &n, &q);
86          for(int i=1; i<=n; i++)
87          {
88                  char temp[25];
89                  scanf("%s", temp);
90                  string temp2=string(temp);
91                  if(!remap.count(temp2))
92                          remap[temp2]=remap.size();
93                  data[i]=remap[temp2];
94          }
95          for(int i=1; i<n; i++)
96          {
97                  int u, v;
98                  scanf("%d %d", &u, &v);
99                  g[u].pb(v);
100                 g[v].pb(u);
101         }
102         dfs(1, 0);
103
104         vector<mo>query;
105         for(int i=0; i<q; i++)
106         {
107                 int u, v;
108                 scanf("%d %d", &u, &v);
109                 int lc=getLca(u, v);
110                 if(dl[u]>dl[v])
111                         swap(u, v);
112                 query.pb(mo(u==lc?dl[u]:dr[u], dl[v], lc, i));
113         }
114         sort(ALL(query));
115
116         int l=query[0].l, r=query[0].l-1;
117         cnt=0;
118         for(int i=0; i<q; i++)
119         {
120                 int li=query[i].l;
121                 int ri=query[i].r;
122                 int lc=query[i].lc;
123                 int ii=query[i].i;
124                 while(l>li)
```

```
125                         update(di[--l]);
126                 while(r<ri)
127                         update(di[++r]);
128                 while(l<li)
129                         update(di[l++]);
130                 while(r>ri)
131                         update(di[r--]);
132
133                 int u=di[l], v=di[r];
134                 if(lc!=u && lc!=v)
135                         update(lc);
136                 ans[ii]=cnt;
137                 if(lc!=u && lc!=v)
138                         update(lc);
139         }
140         for(int i=0; i<q; i++)
141                 printf("%d\n", ans[i]);
142         return 0;
143 }
```

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

```
1  const int MN=1e+3;
2  vector<int>g[MN];
3  int match[MN], rmatch[MN], vis[MN];
4  int findmatch(int u)
5  {
6          if(vis[u])
7                  return 0;
8          vis[u]=true;
9          for(int v:g[u])
10         {
11                 if(match[v]==-1 || findmatch(match[v]))
12                 {
13                         match[v]=u;
14                         rmatch[u]=v;
15                         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(){};
```

```
48                  node(int _u, CTYPE _d)
49                  {
50                          u=_u;
51                          d=_d;
52                  }
53                  bool operator <(const node &foo) const
54                  {
55                          return d>foo.d;
56                  }
57          };
58          graph(){};
59          vector<int>adj[MN];
60          vector<Edge>edge;
61          inline void set(int _n)
62          {
63                  n=_n;
64          }
65          inline void reset()
66          {
67                  for(int i=0; i<MN; i++)
68                          adj[i].clear();
69                  edge.clear();
70          }
71          inline void add_edge(int u, int v, FTYPE c, FTYPE cst)
72          {
73                  adj[u].push_back(edge.size());
74                  edge.push_back(Edge(v, c, cst));
75                  adj[v].push_back(edge.size());
76                  edge.push_back(Edge(u, 0, -cst));
77          }
78
79          pfc dijkstra(int s, int t)
80          {
81                  for(register int i=0; i<n; i++)
82                          dist[i]=CINF;
83                  dist[s]=0;
84                  flow[s]=FINF;
85                  priority_queue<node>heap;
86                  heap.push(node(s, 0));
87                  while(!heap.empty())
88                  {
89                          int u=heap.top().u;
90                          CTYPE d=heap.top().d;
91                          heap.pop();
92                          if(d>dist[u])
93                                  continue;
94                          for(int i=0; i<adj[u].size(); i++)
95                          {
96                                  int idx=adj[u][i];
97                                  int v=edge[idx].to;
98                                  CTYPE w=edge[idx].cost;
99                                  if(!edge[idx].cap || dist[v]<=d+w+pot[u]-pot[v])
100                                         continue;
101                                 if(d+w<dist[v])
102                                 {
103                                         dist[v]=d+w;
104                                         prev[v]=u;
105                                         eidx[v]=idx;
106                                         flow[v]=min(flow[u], edge[idx].cap);
107                                         heap.push(node(v, d+w));
108                                 }
109                         }
110                 }
111                 if(dist[t]==CINF)
112                         return mp(FINF, CINF);
113                 pfc ret=mp(flow[t], 0);
114                 for(int u=t; u!=s; u=prev[u])
115                 {
116                         int idx=eidx[u];
117                         edge[idx].cap-=flow[t];
118                         edge[idx^1].cap+=flow[t];
119                         ret.second+=flow[t]*edge[idx].cost;
120                 }
121                 return ret;
122         }
123
124         inline pfc mfmc(int s, int t)
125         {
126                 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
```

```
47  //of edge weights
48  int X[N], Y[N], W[N];
49
50  //hld
51  int chainInd[N], chainSize[N], chainHead[N], chainPos[N], chainNo, posInBase[N];
52  int ptr;
53
54  void dfs(int u, int l)
55  {
56          trSz[u]=1;
57          lca[u][0]=l;
58          for(int i=1; i<M; i++)
59                  lca[u][i]=lca[ lca[u][i-1] ][i-1];
60          for(int i=0; i<g[u].size(); i++)
61          {
62                  int v=g[u][i].first;
63                  if(v==l)
64                          continue;
65                  h[v]=h[u]+1;
66                  dfs(v, u);
67                  trSz[u]+=trSz[v];
68          }
69  }
70
71  inline int getLca(int u, int v)
72  {
73          if(h[u]>h[v])
74                  swap(u, v);
75          for(int i=M-1; i>=0; i--)
76                  if(h[v]-(1<<i)>=h[u])
77                          v=lca[v][i];
78          if(u==v)
79                  return u;
80          for(int i=M-1; i>=0; i--)
81          {
82                  if(lca[u][i]!=lca[v][i])
83                  {
84                          u=lca[u][i];
85                          v=lca[v][i];
86                  }
87          }
88          return lca[u][0];
89  }
90
91  //dont use 'c' if the weight is on the vertex
92  //instead of the edge
93  inline void hld(int u, int l, int c)
94  {
95          if(chainHead[chainNo]==-1)
96                  chainHead[chainNo]=u;
97          chainInd[u]=chainNo;
98          chainPos[u]=chainSize[chainNo]++;
99          tr.update(1, 0, n, ptr, c);
100         posInBase[u]=ptr++;
101
102         int msf, idx;
103         msf=idx=-1;
104         for(int i=0; i<g[u].size(); i++)
105         {
106                 int v=g[u][i].first;
107                 if(v==l)
108                         continue;
109                 if(trSz[v]>msf)
110                 {
111                         msf=trSz[v];
112                         idx=i;
113                 }
114         }
115         if(idx>=0)
116                 hld(g[u][idx].first, u, g[u][idx].second);
117         for(int i=0; i<g[u].size(); i++)
118         {
119                 if(i==idx)
120                         continue;
121                 int v=g[u][i].first;
122                 int w=g[u][i].second;
123                 if(v==l)
124                         continue;
125                 chainNo++;
```

```
126                     hld(v, u, w);
127             }
128     }
129
130     inline int query_up(int u, int v)
131     {
132             int uchain=chainInd[u];
133             int vchain=chainInd[v];
134             int ret=0;
135             while(true)
136             {
137                     uchain=chainInd[u];
138                     if(uchain==vchain)
139                     {
140                             ret+=tr.query(1, 0, n, posInBase[v]+1, posInBase[u]);
141                             break;
142                     }
143                     int head=chainHead[uchain];
144                     ret+=tr.query(1, 0, n, posInBase[head],posInBase[u]);
145                     u=head;
146                     u=lca[u][0];
147             }
148             return ret;
149     }
150
151     //returns sum of all edges weights
152     //from 'u' to 'v'
153     inline int query(int u, int v)
154     {
155             if(u==v)
156                     return 0;
157             int l=getLca(u, v);
158             return query_up(u, l)+query_up(v, l);
159     }
160
161     //set and edge to value 'val'
162     inline void update(int u, int val)
163     {
164             int x=X[u], y=Y[u];
165             if(lca[x][0]==y)
166                     tr.update(1, 0, n, posInBase[x], val);
167             else
168                     tr.update(1, 0, n, posInBase[y], val);
169     }
170
171     void clearHld()
172     {
173             //tr.reset();
174             for(int i=0; i<=n; i++)
175             {
176                     g[i].clear();
177                     chainHead[i]=-1;
178                     chainSize[i]=0;
179             }
180             ptr=1;
181             chainNo=0;
182     }
183
184     int main()
185     {
186             scanf("%d", &n);
187             clearHld();
188             for(int i=1; i<n; i++)
189             {
190                     scanf("%d %d %d", &X[i], &Y[i], &W[i]);
191                     g[ X[i] ].push_back({Y[i], W[i]});
192                     g[ Y[i] ].push_back({X[i], W[i]});
193             }
194             dfs(1, 0);
195             hld(1, 0, 0);
196             int q;
197             scanf("%d", &q);
198             while(q--)
199             {
200                     int o, x, y;
201                     scanf("%d %d %d", &o, &x, &y);
202                     if(o==1)
203                             printf("%d\n", query(x, y));
204                     else
```

```
205                             update(x, y);
206             }
207             return 0;
208     }
```

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

```
1  /// David Mateus Batista <david.batista3010@gmail.com>
2  /// Computer Science - Federal University of Itajuba - Brazil
3  /// Uri Online Judge - 2475
4  #include <bits/stdc++.h>
5
6  using namespace std;
7
8  typedef long long ll;
9  typedef unsigned long long ull;
10 typedef long double ld;
11 typedef pair<int,int> pii;
12 typedef pair<ll,ll> pll;
13
14 #define INF 0x3F3F3F3F
15 #define LINF 0x3F3F3F3F3F3F3F3FLL
16 #define DINF (double)1e+30
17 #define EPS (double)1e-9
18 #define PI (double)acos(-1.0)
19 #define RAD(x) (double)(x*PI)/180.0
20 #define PCT(x,y) (double)x*100.0/y
21 #define pb push_back
22 #define mp make_pair
23 #define pq priority_queue
24 #define F first
25 #define S second
26 #define D(x) x&(-x)
27 #define ALL(x) x.begin(),x.end()
28 #define SET(a,b) memset(a, b, sizeof(a))
29 #define DEBUG(x,y) cout << x << y << endl
30 #define gcd(x,y) __gcd(x, y)
31 #define lcm(x,y) (x/gcd(x,y))*y
32 #define bitcnt(x) __builtin_popcountll(x)
33 #define lbit(x) 63-__builtin_clzll(x)
34 #define zerosbitll(x) __builtin_ctzll(x)
35 #define zerosbit(x) __builtin_ctz(x)
36
37 enum {North, East, South, West};
38 //{0, 1, 2, 3}
39 //{Up, Right, Down, Left}
40
41 int mi[] = {-1, 0, 1, 0, -1, 1, 1, -1};
42 int mj[] = {0, 1, 0, -1, 1, 1, -1, -1};
43
44 const int MN=1e+4+35;
45 const int MN2=535;
46 int p, a;
47 ll data[MN];
48
49 inline ll getValue(int l, int r)
50 {
51         return (r-l+1)*(data[r]-data[l-1]);
52 }
```

```
53
54  ll dp[MN2][MN];
55  inline void solve(int k, int l, int r, int L, int R)
56  {
57          if(l>r)
58                  return;
59          int m=(l+r)/2;
60          int s=L;
61          dp[k][m]=LINF;
62          for(int i=max(m, L); i<=R; i++)
63          {
64                  if(dp[k][m]>dp[k-1][i+1]+getValue(m+1, i+1))
65                  {
66                          dp[k][m]=dp[k-1][i+1]+getValue(m+1, i+1);
67                          s=i;
68                  }
69          }
70          solve(k, l, m-1, L, s);
71          solve(k, m+1, r, s, R);
72  }
73
74  int main()
75  {
76          scanf("%d %d", &p, &a);
77          for(int i=1; i<=p; i++)
78          {
79                  ll x;
80                  scanf("%lld", &x);
81                  data[i]=data[i-1]+x;
82          }
83
84          for(int i=0; i<=p; i++)
85                  dp[0][i]=LINF;
86          for(int i=0; i<=a; i++)
87                  dp[i][p]=0;
88          for(int i=1; i<=a; i++)
89                  solve(i, 0, p-1, 0, p-1);
90          printf("%lld\n", dp[a][0]);
91          return 0;
92  }
```

3.1.2 Convex Hull I

Linear

```
1   //Original recurrence:
2   //        dp[i]=min(dp[j]+b[j]*a[i]) for j<i
3   //Condition:
4   //        b[j]>=b[j+1]
5   //        a[i]<=a[i+1]
6   // Solution:
7   // Hull cht=Hull();
8   // cht.insertLine(b[0], dp[0])
9   // for(int i=1; i<n; i++)
10  // {
11  //        dp[i]=cht.query(a[i]);
12  //        cht.insertLine(b[i], dp[i])
13  // }
14  // answer is dp[n-1]
15
16  class Hull
17  {
18          const static int CN=1e+5+35;
19  public:
20          long long a[CN], b[CN];
21          double x[CN];
22          int head, tail;
23          Hull():head(1), tail(0){};
24
25          long long query(long long xx)
26          {
27                  if(head>tail)
28                          return 0;
29                  while(head<tail && x[head+1]<=xx)
30                          head++;
31                  x[head]=xx;
32                  return a[head]*xx+b[head];
33          }
```

```
34
35          void insertLine(long long aa, long long bb)
36          {
37                  double xx=-1e18;
38                  while(head<=tail)
39                  {
40                          if(aa==a[tail])
41                                  return;
42                          xx=1.0*(b[tail]-bb)/(aa-a[tail]);
43                          if(head==tail || xx>=x[tail])
44                                  break;
45                          tail--;
46                  }
47                  a[++tail]=aa;
48                  b[tail]=bb;
49                  x[tail]=xx;
50          }
51  };
```

Dynamic

```
1   //Original recurrence:
2   //        dp[i]=min(dp[j]+b[j]*a[i]) for j<i
3   //Condition:
4   //        b[j]>=b[j+1]
5   //        a[i]<=a[i+1]
6   // Solution:
7   // HullDynamic cht;
8   // cht.insertLine(b[0], dp[0])
9   // for(int i=1; i<n; i++)
10  // {
11  //        dp[i]=cht.query(a[i]);
12  //        cht.insertLine(b[i], dp[i])
13  // }
14  // answer is dp[n-1]
15
16  const long long is_query=-(1LL<<62);
17  class Line
18  {
19  public:
20          long long m, b;
21          mutable function<const Line*()>succ;
22          bool operator < (const Line &rhs) const
23          {
24                  if(rhs.b!=is_query)
25                          return m<rhs.m;
26                  const Line *s=succ();
27                  if(!s)
28                          return 0;
29                  long long x=rhs.m;
30                  return (b-s->b)<((s->m-m)*x);
31          }
32  };
33
34  class HullDynamic: public multiset<Line>
35  {
36  public:
37          void clear()
38          {
39                  clear();
40          }
41          bool bad(iterator y)
42          {
43                  auto z=next(y);
44                  if(y==begin())
45                  {
46                          if(z==end())
47                                  return 0;
48                          return (y->m==z->m && y->b<=z->b);
49                  }
50                  auto x=prev(y);
51                  if(z==end())
52                          return (y->m == x->m && y->b<=x->b);
53                  return ((x->b - y->b)*(z->m - y->m) >= (y->b - z->b)*(y->m - x->m));
54          }
55          void insertLine(ll m, ll b)
56          {
57                  auto y=insert({m, b});
58                  y->succ=[=]
```

```
59                  {
60                          return next(y)==end()?0:&*next(y);
61                  };
62                  if(bad(y))
63                  {
64                          erase(y);
65                          return;
66                  }
67                  while(next(y)!=end() && bad(next(y)))
68                          erase(next(y));
69                  while(y!=begin() && bad(prev(y)))
70                          erase(prev(y));
71          }
72          long long query(long long x)
73          {
74                  auto ret=*lower_bound((Line){x, is_query});
75                  return ret.m*x+ret.b;
76          }
77  };
```

3.1.3 Convex Hull II

3.1.4 Knuth Optimization

3.2 Digits

3.3 Grundy Numbers

Chapter 4

String

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;
```

```
35  }
36
37  vector<int>fft_multiply(vector<int>&a, vector<int>&b)
38  {
39          vector<base>fa(a.begin(), a.end());
40          vector<base>fb(b.begin(), b.end());
41          int n=1;
42          while(n<max(a.size(), b.size()))
43                  n<<=1;
44          n<<=1;
45          fa.resize(n);
46          fb.resize(n);
47          fft(fa, false);
48          fft(fb, false);
49          for(int i=0; i<n; i++)
50                  fa[i]*=fb[i];
51          fft(fa, true);
52
53          vector<int>ret(n);
54          for(int i=0; i<n; i++)
55                  ret[i]=(int)(fa[i].real()+0.5);
56
57          int carry=0;
58          for(int i=0; i<n; i++)
59          {
60                  ret[i]+=carry;
61                  carry=ret[i]/10;
62                  ret[i]%=10;
63          }
64          return ret;
65  }
66
67  int main()
68  {
69          int n, m;
70          scanf("%d␣%d", &n, &m);
71          vector<int>a,b;
72
73          for(int i=0; i<n; i++)
74          {
75                  int x;
76                  scanf("%d", &x);
77                  a.pb(x);
78          }
79
80          for(int i=0; i<m; i++)
81          {
82                  int x;
83                  scanf("%d", &x);
84                  b.pb(x);
85          }
86          reverse(a.begin(), a.end());
87          reverse(b.begin(), b.end());
88
89          vector<int>ans=fft_multiply(a, b);
90          reverse(ans.begin(), ans.end());
91          bool flag=false;
92          for(int i=0; i<ans.size(); i++)
93          {
94                  if(ans[i])
95                          flag=true;
96                  if(flag)
97                          printf("%d", ans[i]);
98          }
99          printf("\n");
100         return 0;
101 }
```

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