Contents

1 Basic Template and snippets

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef vector<ll>vll;
typedef pair<ll,ll> pll;
#define xx first
#define yy second
#define rep(n) for(i=0;i<n;i++)
#define pb push_back
#define mp make_pair
#define clr(a) memset(a, 0, sizeof a)
#define reset(a) memset(a, -1, sizeof a)
#define Clr(a) fill(a.begin(),a.end(),0)
#define Reset(a) fill(a.begin(),a.end(),-1)
#define tr(c, it) \
  for(typeof(c.begin()) it=c.begin(); it!=c.end(); it++)
  void debug(vector<ll> v)
{
    for(int i=0;i<v.size();i++)
        cout<<v[i]<<" ";
    cout<<"\n";
    // call debug({i,j,k})
}
int main()
{
    ll t,z,i,j,k,n,m,p,q,r,s,ans;
    scanf("%lld",&t);
    for(z=1;z<=t;z++)
    {
        scanf("%lld",&n);
        printf("Case %lld: \n",z);
    }
    return 0;
}
```

```cpp
ll steps[][2]={{1,0},{-1,0},{0,1},{0,-1}};
 bool isvalid(int i,int j,ll n,ll m)
    {
        return (i>=0&&j>=0&&i<n&&j<m);
    }
  struct node
    {
        ll n;
        ll cost;
        node(){}
        node(ll n,ll cost) {this->n = n; this->cost = cost;}
        bool operator < (const node &nd) const {
         if(cost!=nd.cost)
         return cost > nd.cost;
    }
    };
```

2 C++ Ref.

2.1 String

1. to_string(val) returns string equivalent of val,val is a number
2. sto i/l/ul/ll/ull stoi(string,nullptr,2); returns number; 2 is base default 10;
3. erase

```cpp
str.erase (10,8); //start position, length
str.erase (str.begin()+9);
str.erase (str.begin()+5, str.end()-9);
```

4.
   - find
   - find_first_of
   - find_first_not_of
   - find_last_of
   - find_last_not_of

```cpp
found=str.find("haystack",0);
 if (found!=string::npos)
```

5. substr(start,length) make a substring

2.2 Algorithm

1. lower_bound (v.begin(), v.end(), 20); returns iterator having element >=20
2. upper_bound (v.begin(), v.end(), 20); returns iterator having element >20
3. binary_search (v.begin(), v.end(), 3); returns true if 3 is present
4.

3 Max-Flow and Bipartite Matching

3.1 Dinic'c Flow algorithm

```
// Dinic's blocking flow algorithm
// Running time:
// * general networks: O(|V|^2 |E|)
// * unit capacity networks: O(E min(V^(2/3), E^(1/2)))
// * bipartite matching networks: O(E sqrt(V))

const int INF = 2000000000;

struct Edge {
  int from, to, cap, flow, index;
  Edge(int from, int to, int cap, int flow, int index) :
    from(from), to(to), cap(cap), flow(flow), index(index) {}
};

struct Dinic {
  int N;
  vector<vector<Edge> > G;
  vector<Edge *> dad;
  vector<int> Q;

  // N = number of vertices
  Dinic(int N) : N(N), G(N), dad(N), Q(N) {}

  // Add an edge to initially empty network. from, to are 0-based
  void AddEdge(int from, int to, int cap) {
    G[from].push_back(Edge(from, to, cap, 0, G[to].size()));
    if (from == to) G[from].back().index++;
    G[to].push_back(Edge(to, from, 0, 0, G[from].size() - 1));
  }

  long long BlockingFlow(int s, int t) {
    fill(dad.begin(), dad.end(), (Edge *) NULL);
    dad[s] = &G[0][0] - 1;

    int head = 0, tail = 0;
    Q[tail++] = s;
    while (head < tail) {
      int x = Q[head++];
      for (int i = 0; i < G[x].size(); i++) {
        Edge &e = G[x][i];
        if (!dad[e.to] && e.cap - e.flow > 0) {
          dad[e.to] = &G[x][i];
          Q[tail++] = e.to;
        }
      }
    }
    if (!dad[t]) return 0;

    long long totflow = 0;
    for (int i = 0; i < G[t].size(); i++) {
      Edge *start = &G[G[t][i].to][G[t][i].index];
      int amt = INF;
      for (Edge *e = start; amt && e != dad[e->from]; e = dad[e->from]) {
        if (!e) { amt = 0; break; }
        amt = min(amt, e->cap - e->flow);
      }
      if (amt == 0) continue;
      for (Edge *e = start; amt && e != dad[e->from]; e = dad[e->from]) {
        e->flow += amt;
        G[e->to][e->index].flow -= amt;
      }
      totflow += amt;
    }
    return totflow;
  }

  // Call this to get the max flow. s, t are 0-based.
  // Note, you can only call this once.
  // To obtain the actual flow values, look at all edges with
  // capacity > 0 (zero capacity edges are residual edges).

  long long GetMaxFlow(int s, int t) {
    long long totflow = 0;
    while (long long flow = BlockingFlow(s, t))
      totflow += flow;
    return totflow;
  }
};
```

3.2 Maximum Bipartite Matching

```
// This code performs maximum bipartite matching.
//
// Running time: O(|E| |V|) -- often much faster in practice
// For larger input, consider Dinic, which runs in O(E sqrt(V))
//
//   INPUT: w[i][j] = edge between row node i and column node j
//   OUTPUT: mr[i] = assignment for row node i, -1 if unassigned
//           mc[j] = assignment for column node j, -1 if unassigned
//           function returns number of matches made

typedef vector<vll> vvl;

bool FindMatch(ll i, const vvl &w, vl &mr, vl &mc, vl &seen) {
  for (ll j = 0; j < w[i].size(); j++) {
    if (w[i][j] && !seen[j]) {
      seen[j] = true;
      if (mc[j] < 0 || FindMatch(mc[j], w, mr, mc, seen)) {
        mr[i] = j;
        mc[j] = i;
        return true;
      }
    }
```

```
  }
  return false;
}

ll BipartiteMatching(const vvl &w, vl &mr, ↩
    vl &mc) {
  mr = vl(w.size(), -1);
  mc = vl(w[0].size(), -1);

  ll ct = 0;
  for (ll i = 0; i < w.size(); i++) {
    vl seen(w[0].size());
    if (FindMatch(i, w, mr, mc, seen)) ct↩
        ++;
  }
  return ct;
}
```

3.3 Minimum cost Bipartite Matching

```
// Min cost bipartite matching via shortest↩
     augmenting paths
//
// This is an O(n^3) implementation of a ↩
    shortest augmenting path
// algorithm for finding min cost perfect ↩
    matchings in dense
// graphs.  In practice, it solves 1000↩
    x1000 problems in around 1
// second.
//
//   cost[i][j] = cost for pairing left ↩
    node i with right node j
//   Lmate[i] = index of right node that ↩
    left node i pairs with
//   Rmate[j] = index of left node that ↩
    right node j pairs with
//
// The values in cost[i][j] may be positive↩
     or negative.  To perform
// maximization, simply negate the cost[][]↩
     matrix.

typedef vector<double> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

double MinCostMatching(const VVD &cost, VI ↩
    &Lmate, VI &Rmate) {
  int n = int(cost.size());

  // construct dual feasible solution
  VD u(n);
  VD v(n);
  for (int i = 0; i < n; i++) {
    u[i] = cost[i][0];
    for (int j = 1; j < n; j++) u[i] = min(↩
        u[i], cost[i][j]);
  }
  for (int j = 0; j < n; j++) {
    v[j] = cost[0][j] - u[0];
    for (int i = 1; i < n; i++) v[j] = min(↩
        v[j], cost[i][j] - u[i]);
  }

  // construct primal solution satisfying ↩
      complementary slackness
  Lmate = VI(n, -1);
  Rmate = VI(n, -1);
  int mated = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      if (Rmate[j] != -1) continue;
      if (fabs(cost[i][j] - u[i] - v[j]) < ↩
          1e-10) {
        Lmate[i] = j;
        Rmate[j] = i;
        mated++;
        break;
      }
    }
  }

  VD dist(n);
  VI dad(n);
  VI seen(n);

  // repeat until primal solution is ↩
      feasible
  while (mated < n) {

    // find an unmatched left node
    int s = 0;
    while (Lmate[s] != -1) s++;

    // initialize Dijkstra
    fill(dad.begin(), dad.end(), -1);
    fill(seen.begin(), seen.end(), 0);
    for (int k = 0; k < n; k++)
      dist[k] = cost[s][k] - u[s] - v[k];

    int j = 0;
    while (true) {

      // find closest
      j = -1;
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        if (j == -1 || dist[k] < dist[j]) j↩
            = k;
      }
      seen[j] = 1;

      // termination condition
      if (Rmate[j] == -1) break;

      // relax neighbors
      const int i = Rmate[j];
      for (int k = 0; k < n; k++) {
        if (seen[k]) continue;
        const double new_dist = dist[j] + ↩
            cost[i][k] - u[i] - v[k];
        if (dist[k] > new_dist) {
          dist[k] = new_dist;
          dad[k] = j;
        }
      }
    }

    // update dual variables
    for (int k = 0; k < n; k++) {
      if (k == j || !seen[k]) continue;
      const int i = Rmate[k];
      v[k] += dist[k] - dist[j];
      u[i] -= dist[k] - dist[j];
    }
    u[s] += dist[j];

    // augment along path
```

```
    while (dad[j] >= 0) {
      const int d = dad[j];
      Rmate[j] = Rmate[d];
      Lmate[Rmate[j]] = j;
      j = d;
    }
    Rmate[j] = s;
    Lmate[s] = j;

    mated++;
  }

  double value = 0;
  for (int i = 0; i < n; i++)
    value += cost[i][Lmate[i]];

  return value;
}
```

3.4 Konig's Theorem

In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover. To exhibit the vertex cover:

1. Find a maximum matching
2. Change each edge **used** in the matching into a directed edge from **right to left**
3. Change each edge **not used** in the matching into a directed edge from **left to right**
4. Compute the set T of all vertices reachable from unmatched vertices on the left (including themselves)
5. The vertex cover consists of all vertices on the right that are **in** T, and all vertices on the left that are **not in** T

3.5 Minimum Edge Cover

If a minimum edge cover contains C edges, and a maximum match- ing contains M edges, then C +M = |V |. To obtain the edge cover, start with a maximum matching, and then, for every vertex not matched, just select some edge incident upon it and add it to the edge set.

4 Data Structures

4.1 BIT

```
// Binary indexed tree supporting binary ↩
    search.
struct BIT {
    ll n;
    vector<ll> bit;
    // BIT can be thought of as having ↩
        entries f[1], ..., f[n]
    // which are 0-initialized
    BIT(ll n):n(n), bit(n+1) {}
    // returns f[1] + ... + f[idx-1]
    // precondition idx <= n+1
    ll read(ll idx) {
    ll res = 0;
        while(idx > 0) res += bit[idx] , ↩
            idx -= idx & -idx;
        return res;
    }
    // returns f[idx1] + ... + f[idx2]
    // precondition idx1 <= idx2 <= n+1
    ll read2(ll idx1, ll idx2) {
        return read(idx2) - read(idx1-1);
    }
    // adds val to f[idx]
    // precondition 1 <= idx <= n (there is↩
         no element 0!)
    void update(ll idx, ll val) {
        while (idx <= n) bit[idx] += val,  ↩
            idx += idx & -idx;
    }
    // returns smallest positive idx such ↩
        that read(idx) >= target
    ll lower_bound(ll target) {
        if (target <= 0) return 1;
        ll pwr = 1; while (2*pwr <= n) pwr↩
            *=2;
        ll idx = 0; ll tot = 0;
        for (; pwr; pwr >>= 1) {
            if (idx+pwr > n) continue;
            if (tot + bit[idx+pwr] < target↩
                ) {
                tot += bit[idx+=pwr];
            }
        }
        return idx+1;
    }
    // returns smallest positive idx such ↩
        that read(idx) > target
    ll upper_bound(ll target) {
        if (target < 0) return 1;
        ll pwr = 1; while (2*pwr <= n) pwr↩
            *=2;
        ll idx = 0; ll tot = 0;
        for (; pwr; pwr >>= 1) {
            if (idx+pwr > n) continue;
            if (tot + bit[idx+pwr] <= ↩
                target) {
                tot += bit[idx+=pwr];
            }
        }
        return idx+1;
    }
};
/*For range update and range query
To add v in range [a, b]: Update(a, v), ↩
    Update(b+1, -v)first BIT and Update(a, ↩
    v*(a-1)) and Update(b+1, -v*b) second ↩
    BIT.
To get sum in range [0, x]: you simply do ↩
    Query_BIT1(x)*x - Query_BIT2(x);
 */
```

4.2 Segment Tree

```
class segtree{
public:
    ll size;
    const ll inf=0x7fffffff;
    ll MAX;
    vector<ll>tree;
    vector<ll>lazy;
    vector<ll>arr;
    segtree(ll n)
    {
        size=n;
        MAX=4*n;
            tree=vll(MAX,0);
            lazy=tree;
            arr=vll(size);
    }
    void build_tree(ll node,ll a, ll b)
    {
        if(a > b) return; // Out of range

        if(a == b) { // Leaf node
                tree[node] = arr[a]; // ↩
                    Init value
            return;
        }
```

```
        build_tree(node*2, a, (a+b)/2); // Init left child
        build_tree(node*2+1, 1+(a+b)/2, b); // Init right child
        tree[node] = max(tree[node*2], tree[node*2+1]); // Init root value
    }
    void update_tree(ll node, ll a, ll b, ll i, ll j, ll value)
        {
        if(lazy[node] != 0) { // This node needs to be updated
            tree[node] += lazy[node]; // Update it

            if(a != b) {
                lazy[node*2] += lazy[node]; // Mark child as lazy
                    lazy[node*2+1] += lazy[node]; // Mark child as lazy
            }

            lazy[node] = 0; // Reset it
        }

        if(a > b || a > j || b < i) // Current segment is not within range [i, j]
            return;

        if(a >= i && b <= j) { // Segment is fully within range
            tree[node] += value;

            if(a != b) { // Not leaf node
                lazy[node*2] += value;
                lazy[node*2+1] += value;
            }
            return;
        }
    /*   if(a == b) { // comment it out if lazy propagation
        tree[node] += value;
        return;
            }*/
        update_tree(node*2, a, (a+b)/2, i, j, value); // Updating left child
        update_tree(1+node*2, 1+(a+b)/2, b, i, j, value); // Updating right child

        tree[node] = max(tree[node*2], tree[node*2+1]); // Updating root with max value
    }
    ll query_tree(ll node, ll a, ll b, ll i, ll j)
    {
        if(a > b || a > j || b < i) return -inf; // Out of range

        if(lazy[node] != 0) { // This node needs to be updated
            tree[node] += lazy[node]; // Update it

            if(a != b) {
                lazy[node*2] += lazy[node]; // Mark child as lazy
                lazy[node*2+1] += lazy[node]; // Mark child as lazy
            }

            lazy[node] = 0; // Reset it
        }

        if(a >= i && b <= j) // Current segment is totally within range [i, j]
            return tree[node];

        ll q1 = query_tree(node*2, a, (a+b)/2, i, j); // Query left child
        ll q2 = query_tree(1+node*2, 1+(a+b)/2, b, i, j); // Query right child

        ll res = max(q1, q2); // Return final result

        return res;
    }
    void update(ll st,ll end,ll value)
    {
        update_tree(1,0,size-1,st,end,value);
    }
    void update(ll point,ll value)
    {
        update_tree(1,0,size-1,point,point,value);
    }
    ll query(ll st,ll end)
    {
        return query_tree(1,0,size-1,st,end);
    }
    void build()
    {
        build_tree(1,0,size-1);
    }
};
```

4.3 Suffix Array

```
// Suffix array construction in O(L log^2 L) time.  Routine for
// computing the length of the longest common prefix of any two
// suffixes in O(log L) time.
//
// INPUT:   string s
//
// OUTPUT:  array suffix[] such that suffix[i] = index (from 0 to L-1)
//          of substring s[i...L-1] in the list of sorted suffixes.
//          That is, if we take the inverse of the permutation suffix[],
//          we get the actual suffix array.
struct SuffixArray {
  const int L;
  string s;
  vector<vector<int> > P;
  vector<pair<pair<int,int>,int> > M;

  SuffixArray(const string &s) : L(s.length()), s(s), P(1, vector<int>(L, 0)), M
```

```
    (L) {
  for (int i = 0; i < L; i++) P[0][i] = ↩
      int(s[i]);
  for (int skip = 1, level = 1; skip < L;↩
       skip *= 2, level++) {
    P.push_back(vector<int>(L, 0));
    for (int i = 0; i < L; i++)
           M[i] = mp(mp(P[level-1][i], i ↩
               + skip < L ? P[level-1][i ↩
               + skip] : -1000), i);
    sort(M.begin(), M.end());
    for (int i = 0; i < L; i++)
           P[level][M[i].yy] = (i > 0 && ↩
               M[i].xx == M[i-1].xx) ? P[↩
               level][M[i-1].yy] : i;
  }
}

vector<int> GetRankArray() { return P.↩
    back(); }
vector<int> GetSuffixArray(){
  vector<int>&rank=P.back();
  int n=rank.size();
  vector<int>sa(n,0);
  for(int i=0;i<n;i++)sa[rank[i]]=i;
  return sa;
}
vector<int> LCP()
  {
      int n=s.size(),k=0;
      vector<int> lcp(n,0);
      vector<int>& rank=P.back();
      vector<int> sa=GetSuffixArray();
      for(int i=0; i<n; i++, k?k--:0)
      {
          if(rank[i]==n-1) {k=0; continue↩
              ;}
          int j=sa[rank[i]+1];
          while(i+k<n && j+k<n && s[i+k↩
              ]==s[j+k]) k++;
          lcp[rank[i]]=k;
      }
      return lcp;
  }
```

```
  // returns the length of the longest ↩
      common prefix of s[i...L-1] and s[j↩
      ...L-1]
  int LongestCommonPrefix(int i, int j) {
    int len = 0;
    if (i == j) return L - i;
    for (int k = P.size() - 1; k >= 0 && i ↩
        < L && j < L; k--) {
      if (P[k][i] == P[k][j]) {
        i += 1 << k;
        j += 1 << k;
        len += 1 << k;
      }
    }
    return len;
  }
};
```

4.4 Union Find

```
   ll par[100];
   void Union(ll x,ll y)
   {
    par[x]=y;
   }
   ll find(ll x)
   {
    if(par[x]==x) return x;
    else return par[x]=find(par[x]);
   }
```

4.5 Palindrome Tree

```
/

    Palindrome tree. Useful structure to ↩
        deal with palindromes in strings. O↩
        (N)/




#include <iostream>
#include <cstdio>
#include <cstdlib>
#include <algorithm>
#include <vector>
#include <set>
#include <map>
#include <string>
#include <utility>
#include <cstring>
#include <cassert>
#include <cmath>
#include <stack>
#include <queue>

using namespace std;

const int MAXN = 105000;

struct node {
    int next[26];
    int len;
    int sufflink;
    int num;
};

int len;
char s[MAXN];
node tree[MAXN];
int num;            // node 1 - root with ↩
    len -1, node 2 - root with len 0
int suff;           // max suffix ↩
    palindrome
long long ans;

bool addLetter(int pos) {
    int cur = suff, curlen = 0;
    int let = s[pos] - 'a';

    while (true) {
        curlen = tree[cur].len;
        if (pos - 1 - curlen >= 0 && s[pos ↩
            - 1 - curlen] == s[pos])
            break;
        cur = tree[cur].sufflink;
    }
    if (tree[cur].next[let]) {
        suff = tree[cur].next[let];
        return false;
    }

    num++;
```

```cpp
    suff = num;
    tree[num].len = tree[cur].len + 2;
    tree[cur].next[let] = num;

    if (tree[num].len == 1) {
        tree[num].sufflink = 2;
        tree[num].num = 1;
        return true;
    }

    while (true) {
        cur = tree[cur].sufflink;
        curlen = tree[cur].len;
        if (pos - 1 - curlen >= 0 && s[pos ↩
            - 1 - curlen] == s[pos]) {
            tree[num].sufflink = tree[cur].↩
                next[let];
            break;
        }
    }

    tree[num].num = 1 + tree[tree[num].↩
        sufflink].num;

    return true;
}

void initTree() {
    num = 2; suff = 2;
    tree[1].len = -1; tree[1].sufflink = 1;
    tree[2].len = 0; tree[2].sufflink = 1;
}

int main() {
    //assert(freopen("input.txt", "r", ↩
        stdin));
    //assert(freopen("output.txt", "w", ↩
        stdout));

    gets(s);
    len = strlen(s);

    initTree();

    for (int i = 0; i < len; i++) {
        addLetter(i);
```

```cpp
        ans += tree[suff].num;
    }

    cout << ans << endl;

    return 0;
}
```

4.6 Z Function

```cpp
#include<bits/stdc++.h>

using namespace std;

const int MAXN = 1000100;

string s;
int n;
int z[MAXN];
int l, r;

int main() {
    getline(cin, s);
    n = (int) s.length();

    l = r = 0;
    for (int i = 2; i <= n; i++) {
        int cur = 0;
        if (i <= r)
            cur = min(r - i + 1, z[i - l + ↩
                1]);
        while (i + cur <= n && s[i + cur - ↩
            1] == s[cur])
            cur++;
        if (i + cur - 1 > r) {
            l = i; r = i + cur - 1;
        }
        z[i] = cur;
    }

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

    return 0;
}
```

5 Graph Algorithms

5.1 BFS

```cpp
vector<vector<ll> >graph(110);
void bfs(ll root,ll n)
{
    vll visited(n,0);
    ll v,i;
    visited[root]=1;
    queue<ll> q;
    q.push(root);
    while(!q.empty())
    {
        ll u=q.front();
        q.pop();
        for(i=0;i<graph[u].size();i++)
        {
            v=graph[u][i];
            if(!visited[v])
            {
                visited[v]=1;
                q.push(v);
            }
        }
    }
}
```

```cpp
bool valid(ll x,ll y,ll p, ll q)
{
    return (x>=0&&x<p&&y>=0&&y<q);
}
int steps↩
    [][2]={{1,0},{0,1},{-1,0},{0,-1},{0,0}};↩

 char mat[15][21][21];
 ll bfs(ll p,ll q,ll r)
 {
    queue<ll> st,end,tm;
    int x,y,j,k;
    bool visited[15][21][21]={0};
    st.push(0);
    end.push(0);
    tm.push(0);
    visited[0][0][0]=1;
    if(mat[0][0][0]=='1')return -1;
    while(!st.empty())
```

```
    {
        x=st.front();
        y=end.front();
        k=tm.front();
        st.pop();
        end.pop();
        tm.pop();
        if(x==p-1&&y==q-1)
            return k;
        k++;
        j=(k)%r;
        for(int i=0;i<5;i++)
        {
            x+=steps[i][0];
            y+=steps[i][1];
            if(valid(x,y,p,q)&&!visited[j][x][y]&&mat[j][x][y]=='0')
            {
                visited[j][x][y]=1;
                st.push(x);
                end.push(y);
                tm.push(k);
            }
            x-=steps[i][0];
            y-=steps[i][1];
        }
    }
    return -1;
 }
ans=bfs(p,q,r);
```

5.2 DFS

```
bool visited[1010];
void dfs(ll num,ll ind)
{
    //cout<<ind<<endl;
    visited[ind]=true;
    cnt[num]++;
    coins[num]+=arr[ind];
    for(int i=0;i<graph[ind].size();i++)
    {
        if(!visited[graph[ind][i]])
        dfs(num,graph[ind][i]);
    }
}
```

5.3 Dijkstra

```
#define SIZE 10010
struct node
{
    ll n;
    ll cost;
    node(){}
    node(ll n,ll cost) {this->n = n;this->cost = cost;}
    bool operator < (const node &nd) const
        {
        if(cost!=nd.cost)
        return cost > nd.cost;
}
};
 struct edge
{
    ll u;
    ll v;
    ll w;
    edge(){}
    edge(ll u,ll v,ll w) {this->u = u; this->v=v;this->w = w;}
    bool operator < (const edge &nd) const
        {
        return this->w>nd.w;
}
};
ll dist[505][505];
vector< vector<pll> >graph;
ll anscnt,ansdist,s;
bool visited[SIZE];
ll weight[SIZE];
void dijkstra(ll st)
{
    priority_queue<node>q;
    clr(visited);
    reset(weight);
    weight[st]=0;
    q.push(node(st,0));
    node x,z;
    pll y;
    while(!q.empty())
    {
        x=q.top();
        q.pop();
        if(visited[x.n])
            continue;
        else visited[x.n]=true;
        dist[st][x.n]=x.cost;
        for(int i=0;i<graph[x.n].size();i++)
        {
            y=graph[x.n][i];
            if(weight[y.yy]==-1||weight[y.yy]>x.cost+y.xx)
            {
                weight[y.yy]=x.cost+y.xx;
                q.push(node(y.yy,weight[y.yy]));
            }
        }
    }
}
scanf("%lld %lld %lld",&n,&m,&s);
for(i=0;i<s;i++)
    scanf("%lld",&arr[i]);

ll wt;
fill(&dist[0][0],&dist[505][0],1e10);
graph.clear();
graph.resize(n+1);
for(i=0;i<n;i++)dist[i][i]=0;
for(i=0;i<m;i++)
{
    scanf("%lld %lld %lld",&p,&q,&wt);
    graph[p].push_back(mp(wt,q));
}
dijkstra(0);
```

5.4 Krushkals

```
vector<ll> par;
void make_set(ll n)
{
    par.clear();
    par.resize(n);
    for(ll i=0;i<n;i++)
        par[i]=i;
}
ll find(ll ch)
{
    if(ch==par[ch])
    {
        return ch;
```

```
    }else{
        par[ch]=find(par[ch]);
        return par[ch];
    }
}
ll xunion(ll x,ll y)
{
    par[x]=y;
}
vector<pair<ll,pll> > edges;
//main
edges.clear();
edges.push_back(mp(p,mp(i,j)));
sort(edges.begin(),edges.end());
ll size=edges.size();
ll cnt=0;
make_set(n);
for(i=0;i<size;i++)
{
    if(find(edges[i].yy.xx)!=find(edges[i].yy.yy))
    {
        //cout<<find(edges[i].yy.xx)<<find(edges[i].yy.yy)<<endl;
        xunion(find(edges[i].yy.xx),find(edges[i].yy.yy));
        ans-=edges[i].xx;
        cnt++;
        //cout<<i<<" "<<edges[i].xx<<endl;
        if(cnt==n-1)
            break;
    }
}
if(cnt==n-1)
printf("Case %lld: %lld\n",z,ans);
else
printf("Case %lld: -1\n",z);
```

5.5 SCC

```
vector<vll>graph,graph2;
vector<bool>visited;
stack<ll>nodes,nodes2;
ll scc[20010];
void dfs(ll num)
{
    visited[num]=1;
    for(int i=0;i<graph[num].size();i++)
    {
        if(!visited[graph[num][i]])
            dfs(graph[num][i]);
    }
    nodes.push(num);
}
void dfs2(ll num,ll cnt)
{
    visited[num]=1;
    scc[num]=cnt;
    for(int i=0;i<graph[num].size();i++)
    {
        if(!visited[graph[num][i]])
            dfs2(graph[num][i],cnt);
    }
}
int main()
{
    ll t,z,i,j,k,n,m,p,q,r,s,ans;
    scanf("%lld",&t);
    for(z=1;z<=t;z++)
    {
        scanf("%lld %lld",&n,&m);
        graph.clear();
        graph2.clear();
        graph.resize(n+1);
        graph2.resize(n+1);
        nodes=stack<ll>();
        visited.resize(n+1);
        fill(visited.begin(),visited.end(),0);
            for(j=0;j<m;j++)
            {
                scanf("%lld %lld",&p,&q);
                p--;q--;
               graph[p].push_back(q);
               graph2[q].push_back(p);
            }
        ll cnt=0;
        for(i=0;i<n;i++)
            if(!visited[i])
                {dfs(i);
                }
        nodes2=nodes;
        swap(graph,graph2);
        fill(visited.begin(),visited.end(),0);
        cnt=0;
        while(!nodes2.empty())
        {
            p=nodes2.top();
            nodes2.pop();
            if(!visited[p])
            {
                dfs2(p,cnt);
                cnt++;
            }
        }
        swap(graph,graph2);
        graph2.clear();
        graph2.resize(cnt+2);
        ll src[20010]={0},sink[20010]={0};
        for(int i=0;i<n;i++)
        {
            for(int j=0;j<graph[i].size();j++)
            {
                ll v=graph[i][j];
                //cout<<scc[i]<<" "<<scc[v]<<endl;
                if(scc[i]!=scc[v])
                {
                    //graph2[scc[i]].push_back(scc[v]);
                    src[scc[v]]=1;
                    sink[scc[i]]=1;
                }
            }
        }
        ll sc=0,sk=0;
        for(i=0;i<cnt;i++)
        {
            if(!src[i])sc++;
            if(!sink[i])sk++;
        }
        if(cnt==1)sc=sk=0;
        printf("Case %lld: %lld\n",z,max(sc,sk));
    }
    return 0;
}
```

5.6 Eulerian Path

```
// Eulerian path/circuit in an undirected ↩
    graph. TODO: Does this handle self-↩
    edges?
// NOTE(Brian): This looks like it could ↩
    theoretically degrade to quadratic time↩
     in, say, a graph where we keep going ↩
    back and forth between two vertices; in↩
     this case a lot of time could be ↩
    wasted searching for an unused edge.
struct EulerianPath {
    int n;
    vector<vector<int> > adj;
    vector<pair<int, int> > edges;
    vector<int> valid;
    vector<int> circuit;

    EulerianPath(int n): n(n), adj(n) {}

    // Call this to construct the graph.
    // Edges are zero-based and undirected ↩
        (only add each edge once!)
    void add_edge(int x, int y) {
        adj[x].push_back(edges.size());
        adj[y].push_back(edges.size());
        edges.push_back(make_pair(x, y));
        valid.push_back(1);
    }

    void find_path(int x){
      for(int i = 0; i < adj[x].size(); i↩
          ++){
        int e = adj[x][i];
        if(!valid[e]) continue;
        int v = edges[e].first;
        if(v == x) v = edges[e].second;
        valid[e] = 0;
        find_path(v);
      }
      circuit.push_back(x);
    }

    // Call this to find the path/circuit (↩
        autodetects)
    // Returns the path/circuit itself in "↩
        circuit" variable
    // Initial node is repeated at end if ↩
        it's a circuit.
    void find_euler_path() {
      circuit.clear();
      //supposes graph is connected and has↩
          correct degree
      for(int i = 0; i < n; i++)
        if(adj[i].size()%2){
          find_path(i);
          return;
        }
      find_path(0);
    }
};
```

5.7 Bridges

```
// Finds bridges and cut vertices
// Receives:
// N: number of vertices
// l: adjacency list
// Gives:
// vis, seen, par (used to find cut ↩
    vertices)
// ap - 1 if it is a cut vertex, 0 ↩
    otherwise
// brid - vector of pairs containing the ↩
    bridges
typedef pair<int, int> PII;

int N;
vector <int> l[MAX];
vector <PII> brid;
int vis[MAX], seen[MAX], par[MAX], ap[MAX];
int cnt, root;

void dfs(int x){
  if(vis[x] != -1)
    return;
  vis[x] = seen[x] = cnt++;

  int adj = 0;
  for(int i = 0; i < (int)l[x].size(); i++)↩
      {
    int v = l[x][i];
    if(par[x] == v)
      continue;
    if(vis[v] == -1){
      adj++;
      par[v] = x;
      dfs(v);
      seen[x] = min(seen[x], seen[v]);
      if(seen[v] >= vis[x] && x != root)
        ap[x] = 1;
      if(seen[v] == vis[v])
        brid.push_back(make_pair(v, x));
    }
    else{
      seen[x] = min(seen[x], vis[v]);
      seen[v] = min(seen[x], seen[v]);
    }
  }
  if(x == root) ap[x] = (adj>1);
}

void bridges(){
  brid.clear();
  for(int i = 0; i < N; i++){
    vis[i] = seen[i] = par[i] = -1;
    ap[i] = 0;
  }
  cnt = 0;
  for(int i = 0; i < N; i++)
    if(vis[i] == -1){
      root = i;
      dfs(i);
    }
}
```

6 Number Theory

6.1 Formulas

1. Eulers Formulas $a^{\varphi(m)} \equiv 1 \pmod{m}$ where $\varphi(m)$ is Euler's totient function
2. Cayley's Formula: There are n n2 spanning trees of a complete graph with n labeled vertices. 2.Derangement: $der(n) = (n-1)(der(n-1) + der(n-2))$
3. Euler's Formula for Planar Graph 6 : V E + F = 2, where F is the number of faces 7 of the Planar Graph.
4. Pick's theorem: the area A of a polygon in terms of the number i of lattice points in the interior located in the polygon and the number b of lattice points on the boundary placed on the polygon's perimeter: $A = i + \frac{b}{2} - 1$
5. A complete bipartite graph K(m,n) has mˆ(n-1) nˆ(m-1) spanning trees.

6.2 Math Library

```
ll mod_inverse ( ll a , ll n ) {
ll x , y ;
ll d = extended_euclid (a , n , x , y ) ;
if ( d > 1) return -1;
return mod (x , n ) ;
}
i64 nCr[MAX][MAX], fact[MAX];
fact[0] = nCr[0][0] = 1;
    for(i = 1; i < MAX; i++) {
        fact[i] = (fact[i-1] * i) % MOD;
        nCr[i][0] = nCr[i][i] = 1;
    }
    for(i = 1; i < MAX; i++)
        for(k = 1; k < i; k++)
            nCr[i][k] = (nCr[i-1][k] + nCr[↩
                i-1][k-1]) % MOD;
ll modpow(ll a,ll b,ll mod)
{
    ll res=1;
    while(b)
    {   if(b&1)res=(res*a)%mod;
        a=(a*a)%mod;
        b>>=1;
    }
    return res;
}
ll func(ll x,ll mod)
{
    return ((x)%mod+mod)%mod;
}
long long inv(ll n, ll MOD=md)
{
    return modpow(n,MOD-2,MOD);
}
void sieve(vll &imap,ll u)
{
    ll j;
    vector<bool>prime(u+1,true);
    for(ll i = 2; i <=u; i++)
    {
        if(prime[i])
        {
            j=i;
            imap.push_back(i);
            while(j<=u)
            {
                prime[j]=false;
                j+=i;
            }
        }
    }
}
```

6.3 Extended GCD

```
int extGcd(int  a, int b, int &x, int &y){
    if(b == 0){
        x = 1;
        y = 0;
        return a;
    }

    int g = extGcd(b,a % b,y,x);
    y -= a / b * x;
    return g;
}

int modInv(int a, int m){
    int x,y;
    extGcd(a, m, x, y);
    return (x % m + m) % m;
}
```

6.4 Chineese Remainder Theorem

```
// rem y mod tienen el mismo numero de ↩
    elementos
long long chinese_remainder(vector<int> rem↩
    , vector<int> mod){
    long long ans = rem[0],m = mod[0];
    int n = rem.size();

    for(int i=1;i<n;++i){
        int a = modular_inverse(m,mod[i]);
        int b = modular_inverse(mod[i],m);
        ans = (ans*b*mod[i]+rem[i]*a*m)%(m*↩
            mod[i]);
        m *= mod[i];
    }

    return ans;
}
```

6.5 Heavy Light Decomposition

```
/*↩
    *****************************************


    Heavy-light decomposition with segment ↩
        trees in paths.
    Used for finding maximum on the path ↩
        between two vertices.
    O(N) on building, O(logN ^ 2) on query.

    Based on problem 1553 from acm.timus.ru
    http://acm.timus.ru/problem.aspx?space↩
        =1&num=1553

*****************************************
    */

#include <iostream>
#include <fstream>
#include <cmath>
#include <algorithm>
#include <vector>
#include <set>
#include <map>
#include <stack>
#include <queue>
#include <cstdlib>
#include <cstdio>
#include <string>
#include <cstring>
#include <cassert>
#include <utility>
#include <iomanip>

using namespace std;

const int MAXN = 105000;

struct SegmentTree {
    int * tree;
    int size;

    void init(int sz) {
        tree = new int[4 * sz];
```

```
        memset(tree, 0, 4 * sz * sizeof(int));
        size = sz;
    }

    int getMax(int v, int from, int to, int l, int r) {
        if (l > to || r < from)
            return 0;
        if (from == l && to == r)
            return tree[v];
        int mid = (from + to) / 2;
        int res = getMax(v * 2, from, mid, l, min(r, mid));
        res = max(res, getMax(v * 2 + 1, mid + 1, to, max(l, mid + 1), r));
        return res;
    }

    int getMax(int l, int r) {
        return getMax(1, 1, size, l, r);
    }

    void update(int v, int from, int to, int pos, int val) {
        if (pos > to || pos < from)
            return;
        if (from == pos && to == pos) {
            tree[v] = val;
            return;
        }
        int mid = (from + to) / 2;
        update(v * 2, from, mid, pos, val);
        update(v * 2 + 1, mid + 1,  to, pos, val);
        tree[v] = max(tree[v * 2], tree[v * 2 + 1]);
    }

    void update(int pos, int val) {
        update(1, 1, size, pos, val);
    }
};

int n, qn;
char q;
int a, b;
int timer;
int sz[MAXN];
int tin[MAXN], tout[MAXN];
int val[MAXN];
vector <int> g[MAXN];
int p[MAXN];
int chain[MAXN], chainRoot[MAXN];
int chainSize[MAXN], chainPos[MAXN];
int chainNum;
SegmentTree tree[MAXN];

bool isHeavy(int from, int to) {
    return sz[to] * 2 >= sz[from];
}

void dfs(int v, int par = -1) {
    timer++;
    tin[v] = timer;
    p[v] = par;
    sz[v] = 1;

    for (int i = 0; i < (int) g[v].size(); i++) {
        int to = g[v][i];
        if (to == par)
            continue;
        dfs(to, v);
        sz[v] += sz[to];
    }

    timer++;
    tout[v] = timer;
}

int newChain(int root) {
    chainNum++;
    chainRoot[chainNum] = root;
    return chainNum;
}

void buildHLD(int v, int curChain) {
    chain[v] = curChain;
    chainSize[curChain]++;
    chainPos[v] = chainSize[curChain];

    for (int i = 0; i < g[v].size(); i++) {
        int to = g[v][i];
        if (p[v] == to)
            continue;
        if (isHeavy(v, to))
            buildHLD(to, curChain);
        else
            buildHLD(to, newChain(to));
    }
}

bool isParent(int a, int b) {
    return tin[a] <= tin[b] && tout[a] >= tout[b];
}

int getMax(int a, int b) {
    int res = 0;
    while (true) {
        int curChain = chain[a];
        if (isParent(chainRoot[curChain], b))
            break;
        res = max(res, tree[curChain].getMax(1, chainPos[a]));
        a = p[chainRoot[curChain]];
    }
    while (true) {
        int curChain = chain[b];
        if (isParent(chainRoot[curChain], a))
            break;
        res = max(res, tree[curChain].getMax(1, chainPos[b]));
        b = p[chainRoot[curChain]];
    }
    int from = chainPos[a], to = chainPos[b];
    if (from > to)
        swap(from, to);
    res = max(res, tree[chain[a]].getMax(from, to));
    return res;
}
```

```
int main() {
    //assert(freopen("input.txt","r",stdin)↩
        );
    //assert(freopen("output.txt","w",↩
        stdout));

    scanf("%d", &n);
    for (int i = 1; i < n; i++) {
        int from, to;
        scanf("%d %d", &from, &to);
        g[from].push_back(to);
        g[to].push_back(from);
    }

    dfs(1);
    buildHLD(1, newChain(1));

    for (int i = 1; i <= chainNum; i++) {
        tree[i].init(chainSize[i]);
    }

    scanf("%d\n", &qn);
    for (int i = 1; i <= qn; i++) {
        scanf("%c %d %d\n", &q, &a, &b);
        if (q == 'I') {
            val[a] += b;
            tree[chain[a]].update(chainPos[↩
                a], val[a]);
        }
        else {
            printf("%d\n", getMax(a, b));
        }
    }

    return 0;
}
```

6.6 Lucas

```
//`$num[i] = i!$`
int comLucus(int n,int m,int p) {
  int ans=1;
  for (; n && m && ans; n/=p,m/=p) {
    if (n%p>=m%p)
      ans = ans*num[n%p]%p*getInv(num[m%p]%↩
          p)%p
          *getInv(num[n%p-m%p])%p;
    else
      ans=0;
  }
  return ans;
}
```

6.7 Primitive Root

```
int getPriRoot(int p) {
  if (p==2) return 1;
  int phi = p - 1;
  getFactor(phi);
  for (int g = 2; g < p; ++g) {
    bool flag=1;
    for (int i = 0; flag && i < N; ++i)
      if (power(g, phi/fac[i], p) == 1)
        flag=0;
    if (flag)
      return g;
  }
}
```

6.8 Simplex Algorithm

```
// Two-phase simplex algorithm for solving ↩
    linear programs of the form
//
//     maximize     c^T x
//     subject to   Ax <= b
//                  x >= 0
//
// INPUT: A -- an m x n matrix
//        b -- an m-dimensional vector
//        c -- an n-dimensional vector
//        x -- a vector where the optimal ↩
    solution will be stored
//
// OUTPUT: value of the optimal solution (↩
    infinity if unbounded
//          above, nan if infeasible)
//
// To use this code, create an LPSolver ↩
    object with A, b, and c as
// arguments.  Then, call Solve(x).

typedef long double DOUBLE;
typedef vector<DOUBLE> VD;
typedef vector<VD> VVD;
typedef vector<int> VI;

const DOUBLE EPS = 1e-9;

struct LPSolver {
  int m, n;
  VI B, N;
  VVD D;

  LPSolver(const VVD &A, const VD &b, const↩
      VD &c) :
    m(b.size()), n(c.size()), N(n+1), B(m),↩
        D(m+2, VD(n+2)) {
    for (int i = 0; i < m; i++) for (int j ↩
        = 0; j < n; j++) D[i][j] = A[i][j];
    for (int i = 0; i < m; i++) { B[i] = n+↩
        i; D[i][n] = -1; D[i][n+1] = b[i]; ↩
        }
    for (int j = 0; j < n; j++) { N[j] = j;↩
        D[m][j] = -c[j]; }
    N[n] = -1; D[m+1][n] = 1;
  }

  void Pivot(int r, int s) {
    for (int i = 0; i < m+2; i++) if (i != ↩
        r)
      for (int j = 0; j < n+2; j++) if (j ↩
          != s)
        D[i][j] -= D[r][j] * D[i][s] / D[r↩
            ][s];
    for (int j = 0; j < n+2; j++) if (j != ↩
        s) D[r][j] /= D[r][s];
    for (int i = 0; i < m+2; i++) if (i != ↩
        r) D[i][s] /= -D[r][s];
    D[r][s] = 1.0 / D[r][s];
    swap(B[r], N[s]);
  }

  bool Simplex(int phase) {
    int x = phase == 1 ? m+1 : m;
    while (true) {
      int s = -1;
      for (int j = 0; j <= n; j++) {
        if (phase == 2 && N[j] == -1) ↩
            continue;
        if (s == -1 || D[x][j] < D[x][s] ||↩
            D[x][j] == D[x][s] && N[j] < N↩
```

```
          [s]) s = j;
      }
      if (D[x][s] >= -EPS) return true;
      int r = -1;
      for (int i = 0; i < m; i++) {
        if (D[i][s] <= 0) continue;
        if (r == -1 || D[i][n+1] / D[i][s] < D[r][n+1] / D[r][s] ||
            D[i][n+1] / D[i][s] == D[r][n+1] / D[r][s] && B[i] < B[r]) r = i;
      }
      if (r == -1) return false;
      Pivot(r, s);
    }
  }

  DOUBLE Solve(VD &x) {
    int r = 0;
    for (int i = 1; i < m; i++) if (D[i][n+1] < D[r][n+1]) r = i;
    if (D[r][n+1] <= -EPS) {
      Pivot(r, n);
      if (!Simplex(1) || D[m+1][n+1] < -EPS) return -numeric_limits<DOUBLE>::infinity();
      for (int i = 0; i < m; i++) if (B[i] == -1) {
        int s = -1;
        for (int j = 0; j <= n; j++)
          if (s == -1 || D[i][j] < D[i][s] || D[i][j] == D[i][s] && N[j] < N[s]) s = j;
        Pivot(i, s);
      }
    }
    if (!Simplex(2)) return numeric_limits<DOUBLE>::infinity();
    x = VD(n);
    for (int i = 0; i < m; i++) if (B[i] < n) x[B[i]] = D[i][n+1];
    return D[m][n+1];
  }
};
```

6.9 Eulers Totient Function

```
// This code took less than 0.5s to calculate with MAX = 10^7
#define MAX 10000000

int phi[MAX];
bool pr[MAX];

void totient(){
  for(int i = 0; i < MAX; i++){
    phi[i] = i;
    pr[i] = true;
  }
  for(int i = 2; i < MAX; i++)
    if(pr[i]){
      for(int j = i; j < MAX; j+=i){
        pr[j] = false;
        phi[j] = phi[j] - (phi[j] / i);
      }
      pr[i] = true;
    }
}
```

7 Geometry

7.1 2-D Geometry

```
// C++ routines for computational geometry.

double INF = 1e100;
double EPS = 1e-12;

struct PT {
  double x, y;
  PT() {}
  PT(double x, double y) : x(x), y(y) {}
  PT(const PT &p) : x(p.x), y(p.y)    {}
  PT operator + (const PT &p)  const { return PT(x+p.x, y+p.y); }
  PT operator - (const PT &p)  const { return PT(x-p.x, y-p.y); }
  PT operator * (double c)     const { return PT(x*c,   y*c  ); }
  PT operator / (double c)     const { return PT(x/c,   y/c  ); }
};

double dot(PT p, PT q)     { return p.x*q.x+p.y*q.y; }
double dist2(PT p, PT q)   { return dot(p-q,p-q); }
double cross(PT p, PT q)   { return p.x*q.y-p.y*q.x; }
ostream &operator<<(ostream &os, const PT &p) {
  os << "(" << p.x << "," << p.y << ")";
}

// rotate a point CCW or CW around the origin
PT RotateCCW90(PT p)   { return PT(-p.y,p.x); }
PT RotateCW90(PT p)    { return PT(p.y,-p.x); }
PT RotateCCW(PT p, double t) {
  return PT(p.x*cos(t)-p.y*sin(t), p.x*sin(t)+p.y*cos(t));
}

// project point c onto line through a and b
// assuming a != b
PT ProjectPointLine(PT a, PT b, PT c) {
  return a + (b-a)*dot(c-a, b-a)/dot(b-a, b-a);
}

// project point c onto line segment through a and b
// if the projection doesn't lie on the segment, returns closest vertex
PT ProjectPointSegment(PT a, PT b, PT c) {
  double r = dot(b-a,b-a);
  if (fabs(r) < EPS) return a;
  r = dot(c-a, b-a)/r;
  if (r < 0) return a;
  if (r > 1) return b;
  return a + (b-a)*r;
}

// compute distance from c to segment between a and b
double DistancePointSegment(PT a, PT b, PT c) {
```

```
  return sqrt(dist2(c, ProjectPointSegment(a, b, c)));
}

// determine if lines from a to b and c to d are parallel or collinear
bool LinesParallel(PT a, PT b, PT c, PT d) {
  return fabs(cross(b-a, c-d)) < EPS;
}

bool LinesCollinear(PT a, PT b, PT c, PT d) {
  return LinesParallel(a, b, c, d)
      && fabs(cross(a-b, a-c)) < EPS
      && fabs(cross(c-d, c-a)) < EPS;
}

// determine if line segment from a to b intersects with
// line segment from c to d
bool SegmentsIntersect(PT a, PT b, PT c, PT d) {
  if (LinesCollinear(a, b, c, d)) {
    if (dist2(a, c) < EPS || dist2(a, d) < EPS ||
      dist2(b, c) < EPS || dist2(b, d) < EPS) return true;
    if (dot(c-a, c-b) > 0 && dot(d-a, d-b) > 0 && dot(c-b, d-b) > 0)
      return false;
    return true;
  }
  if (cross(d-a, b-a) * cross(c-a, b-a) > 0) return false;
  if (cross(a-c, d-c) * cross(b-c, d-c) > 0) return false;
  return true;
}

// compute intersection of line passing through a and b
// with line passing through c and d, assuming that unique
// intersection exists; for segment intersection, check if
// segments intersect first
PT ComputeLineIntersection(PT a, PT b, PT c, PT d) {
  b=b-a; d=c-d; c=c-a;
  assert(dot(b, b) > EPS && dot(d, d) > EPS);
  return a + b*cross(c, d)/cross(b, d);
}

// determine if c and d are on same side of line passing through a and b
bool OnSameSide(PT a, PT b, PT c, PT d) {
  return cross(c-a, c-b) * cross(d-a, d-b) > 0;
}

// compute center of circle given three points
PT ComputeCircleCenter(PT a, PT b, PT c) {
  b=(a+b)/2;
  c=(a+c)/2;
  return ComputeLineIntersection(b, b+RotateCW90(a-b), c, c+RotateCW90(a-c));
}

// determine if point is in a possibly non-convex polygon (by William
// Randolph Franklin); returns 1 for strictly interior points, 0 for
// strictly exterior points, and 0 or 1 for the remaining points.
// Note that it is possible to convert this into an *exact* test using
// integer arithmetic by taking care of the division appropriately
// (making sure to deal with signs properly) and then by writing exact
// tests for checking point on polygon boundary
bool PointInPolygon(const vector<PT> &p, PT q) {
  bool c = 0;
  for (int i = 0; i < p.size(); i++){
    int j = (i+1)%p.size();
    if ((p[i].y <= q.y && q.y < p[j].y ||
      p[j].y <= q.y && q.y < p[i].y) &&
      q.x < p[i].x + (p[j].x - p[i].x) * (q.y - p[i].y) / (p[j].y - p[i].y))
      c = !c;
  }
  return c;
}

// determine if point is on the boundary of a polygon
bool PointOnPolygon(const vector<PT> &p, PT q) {
  for (int i = 0; i < p.size(); i++)
    if (dist2(ProjectPointSegment(p[i], p[(i+1)%p.size()], q), q) < EPS)
      return true;
    return false;
}

// compute intersection of line through points a and b with
// circle centered at c with radius r > 0
vector<PT> CircleLineIntersection(PT a, PT b, PT c, double r) {
  vector<PT> ret;
  b = b-a;
  a = a-c;
  double A = dot(b, b);
  double B = dot(a, b);
  double C = dot(a, a) - r*r;
  double D = B*B - A*C;
  if (D < -EPS) return ret;
  ret.push_back(c+a+b*(-B+sqrt(D+EPS))/A);
  if (D > EPS)
    ret.push_back(c+a+b*(-B-sqrt(D))/A);
  return ret;
}

// compute intersection of circle centered at a with radius r
// with circle centered at b with radius R
vector<PT> CircleCircleIntersection(PT a, PT b, double r, double R) {
  vector<PT> ret;
  double d = sqrt(dist2(a, b));
```

```
  if (d > r+R || d+min(r, R) < max(r, R)) return ret;
  double x = (d*d-R*R+r*r)/(2*d);
  double y = sqrt(r*r-x*x);
  PT v = (b-a)/d;
  ret.push_back(a+v*x + RotateCCW90(v)*y);
  if (y > 0)
    ret.push_back(a+v*x - RotateCCW90(v)*y);
  return ret;
}

// This code computes the area or centroid of a (possibly nonconvex)
// polygon, assuming that the coordinates are listed in a clockwise or
// counterclockwise fashion.  Note that the centroid is often known as
// the "center of gravity" or "center of mass".
double ComputeSignedArea(const vector<PT> &p) {
  double area = 0;
  for(int i = 0; i < p.size(); i++) {
    int j = (i+1) % p.size();
    area += p[i].x*p[j].y - p[j].x*p[i].y;
  }
  return area / 2.0;
}

double ComputeArea(const vector<PT> &p) {
  return fabs(ComputeSignedArea(p));
}

PT ComputeCentroid(const vector<PT> &p) {
  PT c(0,0);
  double scale = 6.0 * ComputeSignedArea(p);
  for (int i = 0; i < p.size(); i++){
    int j = (i+1) % p.size();
    c = c + (p[i]+p[j])*(p[i].x*p[j].y - p[j].x*p[i].y);
  }
  return c / scale;
}

// tests whether or not a given polygon (in CW or CCW order) is simple
bool IsSimple(const vector<PT> &p) {
  for (int i = 0; i < p.size(); i++) {
    for (int k = i+1; k < p.size(); k++) {
      int j = (i+1) % p.size();
      int l = (k+1) % p.size();
      if (i == l || j == k) continue;
      if (SegmentsIntersect(p[i], p[j], p[k], p[l]))
        return false;
    }
  }
  return true;
}
```

7.2 3-D Geometry

```
#define LINE 0
#define SEGMENT 1
#define RAY 2

struct point{
    double x, y, z;
    point(){};
    point(double _x, double _y, double _z){ x=_x; y=_y; z=_z; }
    point operator+ (point p) { return point(x+p.x, y+p.y, z+p.z); }
    point operator- (point p) { return point(x-p.x, y-p.y, z-p.z); }
    point operator* (double c) { return point(x*c, y*c, z*c); }
};

double dot(point a, point b){
    return a.x*b.x + a.y*b.y + a.z*b.z;
}

point cross(point a, point b) {
    return point(a.y*b.z-a.z*b.y,
                 a.z*b.x-a.x*b.z,
                 a.x*b.y-a.y*b.x);
}

double distSq(point a, point b){
    return dot(a-b, a-b);
}
```

```
// compute a, b, c, d such that all points lie on ax + by + cz = d. TODO: test this
double planeFromPts(point p1, point p2, point p3, double& a, double& b, double& c, double& d) {
    point normal = cross(p2-p1, p3-p1);
    a = normal.x; b = normal.y; c = normal.z;
    d = -a*p1.x-b*p1.y-c*p1.z;
}

// project point onto plane. TODO: test this
point ptPlaneProj(point p, double a, double b, double c, double d) {
    double l = (a*p.x+b*p.y+c*p.z+d)/(a*a+b*b+c*c);
    return point(p.x-a*l, p.y-b*l, p.z-c*l);
}

// distance from point p to plane aX + bY + cZ + d = 0
double ptPlaneDist(point p, double a, double b, double c, double d){
    return fabs(a*p.x + b*p.y + c*p.z + d) / sqrt(a*a + b*b + c*c);
}

// distance between parallel planes aX + bY + cZ + d1 = 0 and
// aX + bY + cZ + d2 = 0
double planePlaneDist(double a, double b, double c, double d1, double d2){
    return fabs(d1 - d2) / sqrt(a*a + b*b + c*c);
}

// square distance between point and line, ray or segment
double ptLineDistSq(point s1, point s2, point p, int type){
    double pd2 = distSq(s1, s2);
    point r;
```

```
    if(pd2 == 0)
    r = s1;
    else{
    double u = dot(p-s1, s2-s1) / pd2;
    r = s1 + (s2 - s1)*u;
    if(type != LINE && u < 0.0)
        r = s1;
    if(type == SEGMENT && u > 1.0)
        r = s2;
    }
    return distSq(r, p);
}

// Distance between lines ab and cd. TODO: Test this
double lineLineDistance(point a, point b, point c, point d) {
    point v1 = b-a;
    point v2 = d-c;
    point cr = cross(v1, v2);
    if (dot(cr, cr) < EPS) {
        point proj = v1*(dot(v1, c-a)/dot(v1, v1));
        return sqrt(dot(c-a-proj, c-a-proj));
    } else {
        point n = cr/sqrt(dot(cr, cr));
        point p = dot(n, c - a);
        return sqrt(dot(p, p));
    }
}

// Distance between line segments ab and cd (translated from Java)
double segmentSegmentDistance(point a, point b, point c, point d) {
    point u = b - a, v = d - c, w = a - c;
    double a = dot(u, u), b = dot(u, v), c = dot(v, v), d = dot(u, w), e = dot(v, w);
    double D = a*c-b*b;
    double sc, sN, sD = D;
    double tc, tN, tD = D;

    // compute the line parameters of the two closest points
    if (D < EPS) { // the lines are almost parallel
        sN = 0.0;         // force using point P0 on segment S1
        sD = 1.0;         // to prevent possible division by 0.0 later
        tN = e;
        tD = c;
    } else {                 // get the closest points on the infinite lines
        sN = (b*e - c*d);
        tN = (a*e - b*d);
        if (sN < 0.0) {        // sc < 0 => the s=0 edge is visible
            sN = 0.0;
            tN = e;
            tD = c;
        }
        else if (sN > sD) {  // sc > 1 => the s=1 edge is visible
            sN = sD;
            tN = e + b;
            tD = c;
        }
    }

    if (tN < 0.0) {            // tc < 0 => the t=0 edge is visible
        tN = 0.0;
        // recompute sc for this edge
        if (-d < 0.0)
            sN = 0.0;
        else if (-d > a)
            sN = sD;
        else {
            sN = -d;
            sD = a;
        }
    }
    else if (tN > tD) {      // tc > 1 => the t=1 edge is visible
        tN = tD;
        // recompute sc for this edge
        if ((-d + b) < 0.0)
            sN = 0;
        else if ((-d + b) > a)
            sN = sD;
        else {
            sN = (-d + b);
            sD = a;
        }
    }
    // finally do the division to get sc and tc
    sc = (abs(sN) < EPS ? 0.0 : sN / sD);
    tc = (abs(tN) < EPS ? 0.0 : tN / tD);

    // get the difference of the two closest points
    point dP = w + (sc * u) - (tc * v);  // = S1(sc) - S2(tc)
    return sqrt(dot(dP, dP));   // return the closest distance
}

double signedTetrahedronVol(point A, point B, point C, point D) {
    double A11 = A.x - B.x;
    double A12 = A.x - C.x;
    double A13 = A.x - D.x;
    double A21 = A.y - B.y;
    double A22 = A.y - C.y;
    double A23 = A.y - D.y;
    double A31 = A.z - B.z;
    double A32 = A.z - C.z;
    double A33 = A.z - D.z;
    double det =
        A11*A22*A33 + A12*A23*A31 +
        A13*A21*A32 - A11*A23*A32 -
        A12*A21*A33 - A13*A22*A31;
    return det / 6;
}

// Parameter is a vector of vectors of points - each interior vector
// represents the 3 points that make up 1 face, in any order.
// Note: The polyhedron must be convex, with all faces given as triangles.
double polyhedronVol(vector<vector<point> > poly) {
```

```
    int i,j;
    point cent(0,0,0);
    for (i=0; i<poly.size(); i++)
        for (j=0; j<3; j++)
            cent=cent+poly[i][j];
    cent=cent*(1.0/(poly.size()*3));
    double v=0;
    for (i=0; i<poly.size(); i++)
        v+=fabs(signedTetrahedronVol(cent,↩
            poly[i][0],poly[i][1],poly[i↩
            ][2]));
    return v;
}
```

7.3 Closest Pair of Points

```
/*↩
    ************************************************************************************↩


    Finding the closest pair of points. O(↩
        NlogN), divide-and-conquer.
    Based on http://www.spoj.com/problems/↩
        CLOPPAIR/

************************************************************************************↩
    */

#include <iostream>
#include <fstream>
#include <cmath>
#include <algorithm>
#include <vector>
#include <set>
#include <map>
#include <stack>
#include <queue>
#include <cstdlib>
#include <cstdio>
#include <string>
#include <cstring>
#include <cassert>
#include <utility>
#include <iomanip>

using namespace std;

#define sqr(x) ((x) * (x))

const double inf = 1e100;
const int MAXN = 105000;

struct point {
    double x, y;
    int ind;
};

bool cmp(point a, point b) {
    return (a.x < b.x || (a.x == b.x && a.y↩
        < b.y));
}

double dist(point a, point b) {
    return sqrt(sqr(a.x - b.x) + sqr(a.y - ↩
        b.y));
}

int n;
int a[MAXN];
point p[MAXN], tmp[MAXN];
double ans = inf;
int p1, p2;

void updateAnswer(point a, point b) {
    double d = dist(a, b);
    if (d < ans) {
        ans = d;
        p1 = a.ind; p2 = b.ind;
    }
}

void closestPair(int l, int r) {
    if (l >= r)
        return;

    if (r - l == 1) {
        if (p[l].y > p[r].y)
            swap(p[l], p[r]);
        updateAnswer(p[l], p[r]);
        return;
    }

    int m = (l + r) / 2;
    double mx = p[m].x;

    closestPair(l, m);
    closestPair(m + 1, r);

    int lp = l, rp = m + 1, sz = 1;
    while (lp <= m || rp <= r) {
        if (lp > m || ((rp <= r && p[rp].y ↩
            < p[lp].y))) {
            tmp[sz] = p[rp];
            rp++;
        }
        else {
            tmp[sz] = p[lp];
            lp++;
        }
        sz++;
    }

    for (int i = l; i <= r; i++)
        p[i] = tmp[i - l + 1];

    sz = 0;
    for (int i = l; i <= r; i++)
        if (abs(p[i].x - mx) < ans) {
            sz++;
            tmp[sz] = p[i];
        }

    for (int i = 1; i <= sz; i++) {
        for (int j = i - 1; j >= 1; j--) {
            if (tmp[i].y - tmp[j].y >= ans)
                break;
            updateAnswer(tmp[i], tmp[j]);
        }
    }
}

int main() {
    //assert(freopen("input.txt","r",stdin)↩
        );
    //assert(freopen("output.txt","w",↩
        stdout));

    scanf("%d", &n);
    for (int i = 1; i <= n; i++) {
        scanf("%lf %lf", &p[i].x, &p[i].y);
```

```
        p[i].ind = i - 1;
    }

    sort(p + 1, p + n + 1, cmp);

    closestPair(1, n);

    printf("%d %d %.6lf\n", min(p1, p2), ↩
        max(p1, p2), ans);

    return 0;
}
```

7.4 Convex Hull

```
// O(N log N) Monotone Chains algorithm for↩
    2d convex hull.
// Gives the hull in counterclockwise order↩
    from the leftmost point, which is ↩
    repeated at the end. Minimizes the ↩
    number of points on the hull when ↩
    collinear points exist.
long long cross(pair<int, int> A, pair<int,↩
    int> B, pair<int, int> C) {
    return (B.first - A.first)*(C.second - ↩
        A.second)
          - (B.second - A.second)*(C.first -↩
              A.first);
}
// The hull is returned in param "hull"
void convex_hull(vector<pair<int, int> > ↩
    pts, vector<pair<int, int> >& hull) {
    hull.clear(); sort(pts.begin(), pts.end↩
        ());
    for (int i = 0; i < pts.size(); i++) {
        while (hull.size() >= 2 && cross(↩
            hull[hull.size()-2], hull.back↩
            (), pts[i]) <= 0) {
            hull.pop_back();
        }
        hull.push_back(pts[i]);
    }
    int s = hull.size();
    for (int i = pts.size()-2; i >= 0; i--)↩
        {
        while (hull.size() >= s+1 && cross(↩
            hull[hull.size()-2], hull.back↩
            (), pts[i]) <= 0) {
            hull.pop_back();
        }
        hull.push_back(pts[i]);
    }
}
```

8 Miscellaneous

8.1 Dates

```
// Routines for performing computations on ↩
    dates.  In these routines,
// months are expressed as integers from 1 ↩
    to 12, days are expressed
// as integers from 1 to 31, and years are ↩
    expressed as 4-digit
// integers.

#include <iostream>
#include <string>

using namespace std;

string dayOfWeek[] = {"Mon", "Tue", "Wed", ↩
    "Thu", "Fri", "Sat", "Sun"};

// converts Gregorian date to integer (↩
    Julian day number)
int dateToInt (int m, int d, int y){
  return
    1461 * (y + 4800 + (m - 14) / 12) / 4 +
    367 * (m - 2 - (m - 14) / 12 * 12) / 12↩
        -
    3 * ((y + 4900 + (m - 14) / 12) / 100) ↩
        / 4 +
    d - 32075;
}

// converts integer (Julian day number) to ↩
    Gregorian date: month/day/year
void intToDate (int jd, int &m, int &d, int↩
    &y){
  int x, n, i, j;

  x = jd + 68569;
  n = 4 * x / 146097;
  x -= (146097 * n + 3) / 4;
  i = (4000 * (x + 1)) / 1461001;
  x -= 1461 * i / 4 - 31;
  j = 80 * x / 2447;
  d = x - 2447 * j / 80;
  x = j / 11;
  m = j + 2 - 12 * x;
  y = 100 * (n - 49) + i + x;
}

// converts integer (Julian day number) to ↩
    day of week
string intToDay (int jd){
  return dayOfWeek[jd % 7];
}

int main (int argc, char **argv){
  int jd = dateToInt (3, 24, 2004);
  int m, d, y;
  intToDate (jd, m, d, y);
  string day = intToDay (jd);

  // expected output:
  //    2453089
  //    3/24/2004
  //    Wed
  cout << jd << endl
    << m << "/" << d << "/" << y << endl
    << day << endl;
}
```

8.2 KMP

```
/*
Searches for the string w in the string s (↩
    of length k). Returns the
0-based index of the first match (k if no ↩
    match is found). Algorithm
runs in O(k) time.
*/
void buildTable(string& w, vll& pre)
{
 pre = vll(w.length());
  ll i = 2, j = 0;
 pre[0] = -1;pre[1] = 0;

  while(i < w.length())
  {
    if(w[i-1] == w[j]) {pre[i] = j+1; i++; ↩
        j++; }
```

```
    else if(j > 0) j =pre[j];
    else {pre[i] = 0; i++; }
  }
}
ll KMP(string& s, string& w)
{
  ll m = 0, i = 0;
  vll pre;
  buildTable(w,pre);
  while(m+i < s.length())
  {
    if(w[i] == s[m+i])
    {
      i++;
      if(i == w.length()) return m;
    }
    else
    {
      m += i-pre[i];
      if(i > 0) i =pre[i];
    }
  }
  return s.length();
}
```

8.3 2-SAT

```
// 2-SAT solver based on Kosaraju's algorithm.
// Variables are 0-based. Positive variables are stored in vertices 2n, corresponding negative variables in 2n+1
// TODO: This is quite slow (3x-4x slower than Gabow's algorithm)
struct TwoSat {
    int n;
    vector<vector<int> > adj, radj, scc;
    vector<int> sid, vis, val;
    stack<int> stk;
    int scnt;

    // n: number of variables, including negations
    TwoSat(int n): n(n), adj(n), radj(n), sid(n), vis(n), val(n, -1) {}

    // adds an implication
    void impl(int x, int y) { adj[x].push_back(y); radj[y].push_back(x); }
    // adds a disjunction
    void vee(int x, int y) { impl(x^1, y); impl(y^1, x); }
    // forces variables to be equal
    void eq(int x, int y) { impl(x, y); impl(y, x); impl(x^1, y^1); impl(y^1, x^1); }
    // forces variable to be true
    void tru(int x) { impl(x^1, x); }

    void dfs1(int x) {
        if (vis[x]++) return;
        for (int i = 0; i < adj[x].size(); i++) {
            dfs1(adj[x][i]);
        }
        stk.push(x);
    }

    void dfs2(int x) {
        if (!vis[x]) return; vis[x] = 0;
        sid[x] = scnt; scc.back().push_back(x);
        for (int i = 0; i < radj[x].size(); i++) {
            dfs2(radj[x][i]);
        }
    }

    // returns true if satisfiable, false otherwise
    // on completion, val[x] is the assigned value of variable x
    // note, val[x] = 0 implies val[x^1] = 1
    bool two_sat() {
        scnt = 0;
        for (int i = 0; i < n; i++) {
            dfs1(i);
        }
        while (!stk.empty()) {
            int v = stk.top(); stk.pop();
            if (vis[v]) {
                scc.push_back(vector<int>());
                dfs2(v);
                scnt++;
            }
        }
        for (int i = 0; i < n; i += 2) {
            if (sid[i] == sid[i+1]) return false;
        }
        vector<int> must(scnt);
        for (int i = 0; i < scnt; i++) {
            for (int j = 0; j < scc[i].size(); j++) {
                val[scc[i][j]] = must[i];
                must[sid[scc[i][j]^1]] = !must[i];
            }
        }
        return true;
    }
};
```

8.4 Binary Search

```
// Binary search. This is included because binary search can be tricky.
// n is size of array A, c is value we're searching for. Semantics follow those of std::lower_bound and std::upper_bound
int lower_bound(int A[], int n, int c) {
    int l = 0;
    int r = n;
    while (l < r) {
        int m = (r-l)/2+l; //prevents integer overflow
        if (A[m] < c) l = m+1; else r = m;
    }
    return l;
}

int upper_bound(int A[], int n, int c) {
    int l = 0;
    int r = n;
    while (l < r) {
        int m = (r-l)/2+l;
        if (A[m] <= c) l = m+1; else r = m;
```

```
    }
    return 1;
}
```

8.5 FFT

```
struct cpx
{
  cpx(){}
  cpx(double aa):a(aa){}
  cpx(double aa, double bb):a(aa),b(bb){}
  double a;
  double b;
  double modsq(void) const
  {
    return a * a + b * b;
  }
  cpx bar(void) const
  {
    return cpx(a, -b);
  }
};

cpx operator +(cpx a, cpx b)
{
  return cpx(a.a + b.a, a.b + b.b);
}

cpx operator *(cpx a, cpx b)
{
  return cpx(a.a * b.a - a.b * b.b, a.a * b↩
      .b + a.b * b.a);
}

cpx operator /(cpx a, cpx b)
{
  cpx r = a * b.bar();
  return cpx(r.a / b.modsq(), r.b / b.modsq↩
      ());
}

cpx EXP(double theta)
{
  return cpx(cos(theta),sin(theta));
}

const double two_pi = 4 * acos(0);

// in:     input array
// out:    output array
// step:   {SET TO 1} (used internally)
// size:   length of the input/output {MUST↩
    BE A POWER OF 2}
// dir:    either plus or minus one (↩
    direction of the FFT)
// RESULT: out[k] = \sum_{j=0}^{size - 1} ↩
    in[j] * exp(dir * 2pi * i * j * k / ↩
    size)
void FFT(cpx *in, cpx *out, int step, int ↩
    size, int dir)
{
  if(size < 1) return;
  if(size == 1)
  {
    out[0] = in[0];
    return;
  }
  FFT(in, out, step * 2, size / 2, dir);
  FFT(in + step, out + size / 2, step * 2, ↩
      size / 2, dir);
  for(int i = 0 ; i < size / 2 ; i++)
  {
    cpx even = out[i];
    cpx odd = out[i + size / 2];
    out[i] = even + EXP(dir * two_pi * i / ↩
        size) * odd;
    out[i + size / 2] = even + EXP(dir * ↩
        two_pi * (i + size / 2) / size) * ↩
        odd;
  }
}

// Usage:
// f[0...N-1] and g[0..N-1] are numbers
// Want to compute the convolution h, ↩
    defined by
// h[n] = sum of f[k]g[n-k] (k = 0, ..., N↩
    -1).
// Here, the index is cyclic; f[-1] = f[N↩
    -1], f[-2] = f[N-2], etc.
// Let F[0...N-1] be FFT(f), and similarly,↩
     define G and H.
// The convolution theorem says H[n] = F[n]↩
    G[n] (element-wise product).
// To compute h[] in O(N log N) time, do ↩
    the following:
//   1. Compute F and G (pass dir = 1 as ↩
    the argument).
//   2. Get H by element-wise multiplying F↩
     and G.
//   3. Get h by taking the inverse FFT (↩
    use dir = -1 as the argument)
//       and *dividing by N*. DO NOT FORGET ↩
    THIS SCALING FACTOR.
// To compute an *acyclic* convolution, pad↩
     f and g to the right with zeroes.
```

8.6 All nearest smaller values

```
// Linear time all nearest smaller values, ↩
    standard stack-based algorithm.
// ansv_left stores indices of nearest ↩
    smaller values to the left in res. -1 ↩
    means no smaller value was found.
// ansv_right likewise looks to the right. ↩
    v.size() means no smaller value was ↩
    found.
void ansv_left(vector<int>& v, vector<int>&↩
     res) {
    stack<pair<int, int> > stk; stk.push(↩
        make_pair(INT_MIN, v.size()));
    for (int i = v.size()-1; i >= 0; i--) {
        while (stk.top().first > v[i]) {
            res[stk.top().second] = i; stk.↩
                pop();
        }
        stk.push(make_pair(v[i], i));
    }
    while (stk.top().second < v.size()) {
        res[stk.top().second] = -1; stk.pop↩
            ();
    }
}

void ansv_right(vector<int>& v, vector<int↩
    >& res) {
    stack<pair<int, int> > stk; stk.push(↩
        make_pair(INT_MIN, -1));
    for (int i = 0; i < v.size(); i++) {
        while (stk.top().first > v[i]) {
            res[stk.top().second] = i; stk.↩
                pop();
```

```
        }
        stk.push(make_pair(v[i], i));
    }
    while (stk.top().second > -1) {
        res[stk.top().second] = v.size(); ↩
            stk.pop();
    }
}
```

8.7 Manacher's Algorithm

```
// Manacher's algorithm: finds maximal ↩
    palindrome lengths centered around each
// position in a string (including ↩
    positions between characters) and ↩
    returns
// them in left-to-right order of centres. ↩
    Linear time.
// Ex: "opposes" -> [0, 1, 0, 1, 4, 1, 0, ↩
    1, 0, 1, 0, 3, 0, 1, 0]
vector<ll> fastLongestPalindromes(string ↩
    str) {
    ll i=0,j,d,s,e,lLen,palLen=0;
    vector<ll> res;
    while (i < str.length()) {
        if (i > palLen && str[i-palLen-1] ↩
            == str[i]) {
            palLen += 2; i++; continue;
        }
        res.push_back(palLen);
        s = res.size()-2;
        e = s-palLen;
        bool b = true;
        for (j=s; j>e; j--) {
            d = j-e-1;
            if (res[j] == d) { palLen = d; ↩
                b = false; break; }
            res.push_back(min(d, res[j]));
        }
        if (b) { palLen = 1; i++; }
    }
    res.push_back(palLen);
    lLen = res.size();
    s = lLen-2;
    e = s-(2*str.length()+1-lLen);
    for (i=s; i>e; i--) { d = i-e-1; res.↩
        push_back(min(d, res[i])); }
    return res;
}
```

8.8 Matrix Library

```
struct Matrix{
    ll r,c;
    vector<vector<ll> >matrix;
    Matrix(ll r,ll c,ll def=0):r(r),c(c),↩
        matrix(r,vll(c,def)){}

    void unit(){for(ll i=0;i<r;i++)matrix[i↩
        ][i]=1;}

    void add(Matrix &b) //a=a+b
    {
        for(ll i=0;i<r;i++)
            for(ll j=0;j<c;j++)
                matrix[i][j]+=b.matrix[i][j↩
                    ];
    }

    Matrix mult(Matrix &b)  //return this*b
    {
        Matrix a(r,b.c,0);
        for(ll i=0;i<r;i++)
            for(ll j=0;j<b.c;j++)
                for(ll k=0;k<c;k++)
                    a.matrix[i][j]+=matrix[↩
                        i][k]*b.matrix[k][j↩
                        ];
        return a;
    }

    void pow(ll b)      //a=a^b
    {
        Matrix a(r,c,0);
        a.unit();
        while(b)
        {
            if(b&1)a=a.mult(*this);
            *this=this->mult(*this);
            b>>=1;
        }
        *this=a;
    }
    void display()
    {
        for(ll i=0;i<r;i++)
            for(ll j=0;j<c;j++)
                printf("%lld%c",matrix[i][j↩
                    ]," \n"[j==c-1]);
    }
};
```