```
set nu
set sw=4
set ts=4
set st=4
set bs=2
set cul
set ai
set ls=2
map <F5> gT
imap <F5> <ESC>gT
map <F6> gt
imap <F6> <ESC>gt
imap {<CR> {<CR><END><CR>}<UP><END>
au FileType cpp map <F9> <ESC>:w<CR>:!g++<Space>-Wall<Space>%&&./a.out<CR>
au FileType cpp imap <F9> <ESC>:w<CR>:!g++<Space>-Wall<Space>%&&./a.out<CR>
set encoding=UTF-8
```

code/.vimrc

```
const long long MOD = 1e9+7;
const int MAX = 1e5+1;

typedef long long T;
T inverse(T mod, T b) { /* return b^(-1) mod a */
    T k[2][2], n[2][2], u1, u2;

    k[0][0] = k[1][1] = 1;
    k[0][1] = k[1][0] = 0;

    u1 = mod, u2 = b;

    while (u2) {
        T div = u1/u2;
        T remind = u1%u2;

        n[0][0] = k[1][0];
        n[0][1] = k[1][1];
        n[1][0] = k[0][0] - k[1][0]*div;
        n[1][1] = k[0][1] - k[1][1]*div;

        for (T i = 0; i < 2; ++i) {
            for (T j = 0; j < 2; ++j) {
                k[i][j] = n[i][j];
            }
        }
        u1 = u2;
        u2 = remind;
    }

    if (k[0][1] < 0) k[0][1] += mod;
    return k[0][1];
}

T C(T n, T m, T mod) {
    if (m < 0) return 0;
    if (n < m) return 0;
    T ans = 1;
    T base = min(n-m, m);

    for (T i = 0; i < base; ++i) {
        ans = ans*(n-i)%mod;
    }

    T inv = 1;
    for (T i = 1; i <= base; ++i) {
```

```
        inv = inv*i%mod;
    }
    return ans*inverse(mod, inv)%mod;
}
```

code/combination.cpp

```
import java.io.*;
import java.util.*;

public class Main{

  public static void main(String[] args){
    Scan scan = new Scan();
    int testcases = scan.nextInt();
    while(testcases-- != 0){
      int n = scan.nextInt();
      Coordinate[] vertex = new Coordinate[n];
      for(int i=0;i<n;i++) vertex[i] = new Coordinate(scan.nextDouble(), scan.nextDouble());
      Arrays.sort(vertex);
//      for(Coordinate c : vertex){
//        System.out.println(c.x+" "+c.y);
//      }
      Coordinate[] list = new Coordinate[n+1];
      int index = 0;
      for(int i=0;i<n;i++){
        while(index >= 2 && ABcrossAC(list[index-2], list[index-1], vertex[i]) <= 0) index--;
        list[index++] = vertex[i];
      }
      int half_point = index+1;
      for(int i=n-2;i>=0;i--){
        while(index >= half_point && ABcrossAC(list[index-2], list[index-1], vertex[i]) <= 0)
    index--;
        list[index++] = vertex[i];
      }
      double result = 0.0;
      //System.out.println(list[0].x+" "+list[0].y);
      for(int i=1;i<index;i++){
        //System.out.println(list[i].x+" "+list[i].y);
        result += Math.sqrt((list[i].x-list[i-1].x)*(list[i].x-list[i-1].x) + (list[i].y-list[i
    -1].y)*(list[i].y-list[i-1].y));
      }
      System.out.println(result);
    }
  }

  static double ABcrossAC(Coordinate A, Coordinate B, Coordinate C){
    return (B.x-A.x) * (C.x-A.x) - (B.y-A.y) * (C.y-A.y);
  }

  static class Coordinate implements Comparable<Coordinate>{

    double x,y;

    Coordinate(double x, double y){
      this.x = x;
      this.y = y;
    }

    @Override
    public int compareTo(Coordinate o){
      if(x < o.x) return -1;
      if(x > o.x) return 1;
      if(y < o.y) return -1;
      if(y > o.y) return 1;
      return 0;
```

```
    }

  }


}

class Scan implements Iterator<String>{

  BufferedReader buffer;
  StringTokenizer tok;

  Scan(){
    buffer = new BufferedReader(new InputStreamReader(System.in));
  }


  @Override
  public boolean hasNext(){
    while(tok == null || !tok.hasMoreElements()){
      try{
        tok = new StringTokenizer(buffer.readLine());
      }catch(Exception e){
        return false;
      }
    }
    return true;
  }

  @Override
  public String next(){
    if(hasNext()) return tok.nextToken();
    return null;
  }

  @Override
  public void remove(){
    throw new UnsupportedOperationException();
  }

  int nextInt(){
    return Integer.parseInt(next());
  }

  long nextLong(){
    return Long.parseLong(next());
  }

  double nextDouble(){
    return Double.parseDouble(next());
  }

  String nextLine(){
    if(hasNext()) return tok.nextToken("\n");
    return null;
  }
}
```

code/ConvexHull.java

```
/* build: O(VlogV), query: O(logV) */
#include <iostream>
#include <vector>
#include <cstdio>
#define MAX 50010
```

```cpp
using namespace std;

int a[MAX][160]; /* 160 = log2(MAX/2) */
int parent[MAX], tin[MAX], tout[MAX];
int num, root, timestamp;
bool visit[MAX];
vector<int> adj[MAX];

int log2(int n) {
    int i = 0;
    while ((1<<i) <= n) ++i;
    return i - 1;
}

/* when x == y, it's be true */
bool ancestor(int x, int y) {
    return (tin[x] <= tin[y]) && (tout[x] >= tout[y]);
}

void dfs(int x, int px) {
    tin[x] = timestamp++;
    visit[x] = true;
    a[x][0] = px;
    for (int i = 1; i < log2(num); ++i) {
        a[x][i] = a[a[x][i-1]][i-1];
    }

    for (int i = 0; i < adj[x].size(); ++i) {
        int target = adj[x][i];
        if (!visit[target]) {
            parent[target] = x;
            dfs(target, x);
        }
    }
    tout[x] = timestamp++;
}

int lca(int x, int y) {
    if (ancestor(x, y)) return x;
    if (ancestor(y, x)) return y;

    for (int i = log2(num); i >= 0; --i) {
        if (!ancestor(a[x][i], y)) {
            x = a[x][i];
        }
    }
    return a[x][0];
}

int main () {
    timestamp = 0;

    /* init */
    for (int i = 0; i < num; ++i) {
        parent[i] = i;
        visit[i] = false;
        adj[i].clear();
    }

    for (int i = 0; i < num-1; ++i) {
        int x, y;
        scanf("%d%d", &x, &y);
        adj[x].push_back(y);
        adj[y].push_back(x);
    }
```

```
    dfs(0, 0);
    cin >> x >> y;
    cout << lca(x, y);
}
```

code/double_lca.cpp

```
#include <iostream>
#include <cmath>
#include <vector>
#include <cstdio>
#include <algorithm>
#define EPS 1e-10
#define LEFT_TOP POS(1000, 1000)
#define NO_INTERSECT POS(-1234, -1234)
#define PARALLEL POS(-1001, -1001)
#define COLINE POS(1234, 1234)

using namespace std;

typedef double T;

class POS {
public:
    T x, y;
    POS(const T& x = 0, const T& y = 0) : x(x), y(y) {}
    POS(const POS& x) : x(x.x), y(x.y) {}

    bool operator==(const POS& rhs) const {
        return x == rhs.x && y == rhs.y;
    }

    POS& operator+=(const POS& rhs) {
        x += rhs.x;
        y += rhs.y;
        return *this;
    }

    POS operator -() {
        POS tmp(-x, -y);
        return tmp;
    }

    double dist(const POS& rhs) const {
        T tmp_x = x-rhs.x, tmp_y = y-rhs.y;
        return sqrt(tmp_x*tmp_x+tmp_y*tmp_y);
    }

    friend ostream& operator<<(ostream& out, const POS& pos) {
        out << pos.x << " " << pos.y;
        return out;
    }
};

POS const operator+(const POS& lhs, const POS& rhs) {
    return POS(lhs) += rhs;
}

POS const operator-(const POS& lhs, const POS& rhs) {
    POS tmp = rhs;
    tmp = -tmp;
    return POS(lhs) += (tmp);
}

class LINE {
public:
```

```
POS start, end, vec;
LINE(const T& st_x, const T& st_y, const T& ed_x, const T& ed_y) :
    start(st_x, st_y), end(ed_x, ed_y), vec(end - start) {}

LINE(const POS& start, const POS& end) :
    start(start), end(end), vec(end - start) {}

LINE(const POS& end) : /* start point is origin */
    start(0, 0), end(end), vec(end) {}

LINE(const T a, const T b, const T c) : /* given line by ax+by+c = 0 */
    start(0, 0), end(0, 0), vec(-b, a) {
    if (a == 0) {
        start.y = end.y = -c/b;
        end.x = -b;
    }
    else if (b == 0) {
        start.x = end.x = -c/a;
        end.y = a;
    }
    else if (c == 0) {
        end.x = -b; end.y = a;
    }
    else {
        start.y = -c/b; end.x = -c/a;
        vec.x = -c/a; vec.y = c/b;
    }
}

LINE build_orthogonal(const POS& point) const {
    T c = -(vec.x*point.x + vec.y*point.y);
    return LINE(vec.x, vec.y, c);
}

T length2() const { /* square */
    T x = start.x - end.x, y = start.y - end.y;
    return x*x + y*y;
}

void modify(T x, T y) {
    this->end.x += x;
    this->end.y += y;
    this->vec.x += x;
    this->vec.y += y;
}

bool on_line(const POS& a) const {
    if (vec.x == 0) {
        if (start.x != a.x) return false;
        return true;
    }
    if (vec.y == 0) {
        if (start.y != a.y) return false;
        return true;
    }
    return ( (a.x-start.x)/vec.x*vec.y + start.y ) == a.y;
}

bool operator/(const LINE& rhs) const { /* to see if this line parallel to LINE rhs */
    return (vec.x*rhs.vec.y == vec.y*rhs.vec.x);
}

bool operator==(const LINE& rhs) const { /* to see if they are same line */
    return (*this/rhs) && (rhs.on_line(start));
}
```

```
    POS intersect(const LINE& rhs) const {
        if (*this==rhs) return COLINE; /* return co-line */
        if (*this/rhs) return PARALLEL; /* return parallel */

        double A1 = vec.y, B1 = -vec.x, C1 = end.x*start.y - start.x*end.y;
        double A2 = rhs.vec.y, B2 = -rhs.vec.x, C2 = rhs.end.x*rhs.start.y - rhs.start.x*rhs.end.y
    ;
        return POS( (B2*C1-B1*C2)/(A2*B1-A1*B2), (A1*C2-A2*C1)/(A2*B1-A1*B2) ); /* sometimes has
    -0 */
    }

    double dist(const POS& a) const {
        return fabs(vec.y*a.x - vec.x*a.y + vec.x*start.y - vec.y*start.x)/sqrt(vec.y*vec.y+vec.x*
    vec.x);
    }

    double dist(const LINE& rhs) const {
        POS intersect_point = intersect(rhs);
        if (intersect_point == PARALLEL) {
            return dist(rhs.start);
        }
        return 0;
    }

    friend ostream& operator<<(ostream& out, const LINE& line) {
        out << line.start << "-->" << line.end << " vec: " << line.vec << endl;
        return out;
    }
};

class LINESEG : public LINE {
public:
    LINESEG() : LINE(POS(0, 0)) {}
    LINESEG(const LINE& input) : LINE(input) {}
    LINESEG(const POS& start, const POS& end) : LINE(start, end) {}

    bool on_lineseg(const POS& a) const {
        if (!on_line(a)) return false;
        bool first, second;
        if (vec.x >= 0) first = (a.x >= start.x)&&(a.x <= end.x);
        else first = (a.x <= start.x)&&(a.x >= end.x);
        if (vec.y >= 0) second = (a.y >= start.y)&&(a.y <= end.y);
        else second = (a.y <= start.y)&&(a.y >= end.y);
        return first&&second;
    }

    bool operator==(const LINESEG& rhs) const {
        return ( (rhs.start == start && rhs.end == end) ||
              (rhs.start == end && rhs.end == start) );
    }

    bool operator==(const LINE& rhs) const {
        return this->LINE::operator==(rhs);
    }

    T dot(const LINESEG& rhs) const {
        return vec.x*rhs.vec.x + vec.y*rhs.vec.y;
    }

    T cross(const LINESEG& rhs) const {
        return vec.x*rhs.vec.y - vec.y*rhs.vec.x;
    }

    bool clockwise(const LINE& a) const { /* to see if LINE a is in b's clockwise way */
        return cross(a) > 0;
    }
```

```
    double dist(const POS& a) const {
        double ortho_dist = this->LINE::dist(a);
        LINE ortho_line = build_orthogonal(a);
        POS intersect_point = this->LINE::intersect(ortho_line);
        if (on_lineseg(intersect_point)) return ortho_dist;
        else return min(a.dist(this->start), a.dist(this->end));
    }

    double dist(const LINE& line) const {
        POS intersect_point = this->LINE::intersect(line);
        if (intersect_point == COLINE) return 0;
        if (intersect_point == PARALLEL) return dist(line.start);
        if (on_lineseg(intersect_point)) return 0;
        return min(line.dist(start), line.dist(end));
    }

    double dist(const LINESEG& line) const {
        return min( min(dist(line.start), dist(line.end)),
                    min(line.dist(start), line.dist(end)) );
    }

    POS intersect(const LINESEG& rhs) const {
        LINE a1b1(start, rhs.start);
        LINE a1b2(start, rhs.end);
        LINE b1a1(rhs.start, start);
        LINE b1a2(rhs.start, end);

        POS tmp(this->LINE::intersect(rhs));

        if (tmp == COLINE) {
            if ( (start==rhs.start) && (!rhs.on_lineseg(end)) && (!on_lineseg(rhs.end)) ) return
    start;
            if ( (start==rhs.end) && (!rhs.on_lineseg(end)) && (!on_lineseg(rhs.start)) ) return
    start;
            if ( (end==rhs.start) && (!rhs.on_lineseg(start)) && (!on_lineseg(rhs.end)) ) return
    end;
            if ( (end==rhs.end) && (!rhs.on_lineseg(start)) && (!on_lineseg(rhs.start)) ) return
    end;
            if (on_lineseg(rhs.start) || on_lineseg(rhs.end) || rhs.on_lineseg(start) || rhs.
    on_lineseg(end)) return COLINE;
            return NO_INTERSECT;
        }

        bool intersected =  ( (cross(a1b1)*cross(a1b2)<0) && (rhs.cross(b1a1)*rhs.cross(b1a2)<0) )
    ;
        if (!intersected) return NO_INTERSECT;
        if (!on_lineseg(tmp) || !rhs.on_lineseg(tmp)) return NO_INTERSECT;
        return tmp;
    }
};

class POLYGON {
public:
    vector<POS> point;
    vector<LINE> line;

    void add_points(const POS& x) {
        point.push_back(x);
    }

    void add_points(const int& x, const int& y) {
        point.push_back(POS(x,y));
    }

    void build_line() {
```

```
        if (line.size() != 0) return; /* if it has build */
        for (int i = 1; i < point.size(); ++i) {
            line.push_back(LINE(point[i], point[i-1]));
        }
        line.push_back(LINE(point[0], point[point.size()-1]));
    }

    double area() {
        double ans = 0;

        vector<LINESEG> tmp;
        for (int i = 0; i < point.size(); ++i) {
            tmp.push_back(LINESEG(point[i]));
        }
        tmp.push_back(LINESEG(point[0]));

        for (int i = 1; i < tmp.size(); ++i) {
            ans += tmp[i-1].cross(tmp[i]);
        }
        return 0.5*fabs(ans);
    }

    bool in_polygon(const POS& a, const POS& left_top = LEFT_TOP) {

        for (int i = 0; i < point.size(); ++i) {
            if (a == point[i]) return true; /* a is polygon's point */
        }

        build_line();
        for (int i = 0; i < line.size(); ++i) {
            if (line[i].on_line(a)) {
                return true; /* a is on polygon's line */
            }
        }

        POS endpoint(left_top); /* should be modified according to problem */
        LINESEG ray(a, endpoint);
        bool touch_endpoint = false;
        do {
            touch_endpoint = false;
            for (int i = 0; i < point.size(); ++i) {
                if (ray.on_lineseg(point[i])) {
                    touch_endpoint = true;
                    break;
                }
            }
            if (touch_endpoint) ray.modify(-1, 0); /* should be modified according to problem */
        } while (touch_endpoint);

        int times = 0;
        for (int i = 0; i < line.size(); ++i) {
            POS tmp(ray.intersect(line[i]));
            if (tmp == NO_INTERSECT || tmp == PARALLEL) {
                continue;
            }
            ++times;
        }
        return (times&1);
    }
};

int main() {
    return 0;
}
```

code/Geometry.cpp

```
int search(STATE& now, int g, int bound) {
    int f = g + now.heuri;
    if (f > bound) return f;
    if (is_goal(now)) return FOUND;

    int min = INF;
    for next in successors(now):
        int t = search(state, g+cost(now,next), bound);
        if (t == FOUND) return FOUND;
        if (t < min) min = t;
    }
    return min;
}

void IDAStar() {
    STATE init(input);
    int bound = init.heuri;
    while (bound <= MAXI) {
        int t = search(init, 0, bound);
        if (t == FOUND) return FOUND;
        if (t == INF) return NOT_FOUND;
        bound = t;
    }
}
```

code/IDAstar.cpp

```
#include <iostream>
#include <cstdio>
#include <algorithm>
#include <cstring>
#define MAX 404
#define INF 0x7fffffff

using namespace std;

int num; // total num of node
int path[MAX][MAX];
bool visit_x[MAX], visit_y[MAX];
int parent[MAX], weight_x[MAX], weight_y[MAX];

bool find(int i) {
    visit_x[i] = true;
    for (int j = 0; j < num; ++j) {
        if (visit_y[j]) continue;
        if (weight_x[i] + weight_y[j] == path[i][j]) {
            visit_y[j] = true;
            if (parent[j] == -1 || find(parent[j])) {
                parent[j] = i;
                return true;
            }
        }
    }
    return false;
}

int weighted_hangarian() {

    /* remember to initial weight_x (max weight of node's edge)*/
    /* initialize */
    for (int i = 0; i < num; ++i) {
        weight_y[i] = 0;
        parent[i] = -1;
    }
```

```
    for (int i = 0; i < num; ++i) {
        while (1) {
            memset(visit_x, false, sizeof(visit_x));
            memset(visit_y, false, sizeof(visit_y));
            if (find(i)) break;

            int lack = INF;
            for (int j = 0; j < num; ++j) {
                if (visit_x[j]) {
                    for (int k = 0; k < num; ++k) {
                        if (!visit_y[k]) {
                            lack = min(lack, weight_x[j] + weight_y[k] - path[j][k]);
                        }
                    }
                }
            }
            if (lack == INF) break;
            // renew label
            for (int j = 0; j < num; ++j) {
                if (visit_x[j]) weight_x[j] -= lack;
                if (visit_y[j]) weight_y[j] += lack;
            }

        }
    }

    int ans = 0;
    for (int i = 0; i < num; ++i) {
        ans += weight_x[i];
        ans += weight_y[i];
    }
    return ans;
}
```

code/km.cpp

```
struct SegmentTree{

  int rank, capacity, length;
  int *input, *tree;

    SegmentTree() {}
  void init(int* input, int length){
    this->input = input;
    this->length = length;
    rank = 1;
    while((1<<rank++) < length);
    capacity = 1<<(rank-1);
    tree = new int[capacity << 1];
    build(1, capacity, capacity<<1);
  }

  ~SegmentTree(){
        delete[] tree;
  }

  int build(int index, int left, int right){
    if(index >= left){
      return tree[index] = getInput(index);
    }
    int middle = (left+right) >> 1;
    int left_value = build(lc(index), left, middle);
    int right_value = build(rc(index), middle, right);
    return tree[index] = max(left_value, right_value);
  }
```

```
  int query(int start, int finish){
    return query(1, capacity, capacity<<1, capacity+start, capacity+finish+1);
  }

  int query(int index, int left, int right, int start, int finish){
    if(left == start && right == finish) return tree[index];
    int middle = (left+right) >> 1;
    if(finish <= middle) return query(lc(index), left, middle, start, finish);
    if(start >= middle) return query(rc(index), middle, right, start, finish);
    int left_value = query(lc(index), left, middle, start, middle);
    int right_value = query(rc(index), middle, right, middle, finish);
    return max(left_value, right_value);
  }

  int getInput(int index){
    index -= capacity;
    if(index < length) return input[index];
    return 0;
  }

  int lc(int x){
    return x<<1;
  }

  int rc(int x){
    return (x<<1)+1;
  }
};
```

code/SegmentTree.cpp

```
public class SplayTree{

  Node root;
  int size;

  SplayTree(){
    root = null;
    size = 0;
  }

  public boolean containsKey(int target){
    return splay(target);
  }

  public void add(int target){
//    System.out.println("add "+target);
    if(root == null){
      root = new Node(null, target);
      return;
    }
    Node now = root;
    while(true){
      if(now.key == target) break;
      if(target<now.key){
        if(now.lchild == null){
          now.lchild = new Node(now, target);
          break;
        }else now = now.lchild;
      }else{
        if(now.rchild == null){
          now.rchild = new Node(now, target);
          break;
        }else now = now.rchild;
      }
    }
```

```
    splay(target);
  }

  public void delete(int target){
//    System.out.println("delete "+target);
    if(!containsKey(target)) return;
    Node l = root.lchild;
    Node r = root.rchild;
    if(l == null){
      root = r;
    }else l.parent = null;
    if(r == null){
      root = l;
    }else r.parent = null;
    if(root==null || root.key != target) return;
    Node lMax = l;

    while(lMax.rchild != null) lMax = lMax.rchild;
    splay(lMax.key);
    lMax.rchild = r;
  }

  private boolean splay(int target){
//    System.out.println("splay "+target);
    while(true){
      if(root == null) return false;
      if(root.key == target) return true;
      if(target<root.key){
        if(root.lchild == null) return false;
        Node l = root.lchild;
        if(l.key == target){
          root = l;
          rightRoatation(l);
          return true;
        }
        if(target<l.key){
          if(l.lchild == null) return false;
          Node a = l.lchild;
          root = a;
          rightRoatation(l);
          rightRoatation(a);
        }else{
          if(l.rchild == null) return false;
          Node b = l.rchild;
          root = b;
          leftRoatation(b);
          rightRoatation(b);
        }
      }else{
        if(root.rchild == null) return false;
        Node r = root.rchild;
        if(r.key == target){
          root = r;
          leftRoatation(r);
          return true;
        }
        if(target>r.key){
          if(r.rchild == null) return false;
          Node d = r.rchild;
          root = d;
          leftRoatation(r);
          leftRoatation(d);
        }else{
          if(r.lchild == null) return false;
          Node c = r.lchild;
          root = c;
```

```
            rightRoatation(c);
            leftRoatation(c);
          }
        }
      }
    }

    void print(Node now){
      if(now == null){
        System.out.print("-1 ");
        return;
      }
      System.out.print(now.key+" ");
      print(now.lchild);
      print(now.rchild);
    }

    void rightRoatation(Node x){
      Node r = x.parent.parent;
      Node p = x.parent;
      Node b = x.rchild;

      x.rchild = p;
      if(p != null) p.parent = x;
      if(p != null) p.lchild = b;
      if(b != null) b.parent = p;
      x.parent = r;
      if(r != null) r.lchild = x;

    }

    void leftRoatation(Node x){
      Node r = x.parent.parent;
      Node p = x.parent;
      Node b = x.lchild;


      x.lchild = p;
      if(p != null) p.parent = x;
      if(p != null) p.rchild = b;
      if(b != null) b.parent = p;
      x.parent = r;
      if(r != null) r.rchild = x;
    }

    class Node{

      Node parent, lchild, rchild;
      int key;

      Node(Node parent, int key){
        this.parent = parent;
        lchild = rchild = null;
        this.key = key;
      }
    }
}
```

code/SplayTree.java

```
/* Time Complexity=2*n*log(n)*log(n) */
#include <cstdio>
#include <algorithm>
using namespace std;

class Weight{
```

```
public:
    Weight(int a=0,int b=0,int c=0):id(a),first(b),second(c){}
    int id,first,second;
    bool operator<(const Weight &rhs)const{
        return first<rhs.first||(first==rhs.first&&second<rhs.second);
    }
    bool operator==(const Weight &rhs)const{
        return first==rhs.first&&second==rhs.second;
    }
    bool operator!=(const Weight &rhs)const{
        return!((*this)==rhs);
    }
};

class SuffixArray{
public:
    SuffixArray(char *r):refer(r){
        for(length=0;refer[length]!='\0';length++);
        rankOfIndex=new int[length];
        indexOfRank=new int[length];
        texi=new Weight[length];//=
        firstsort();
        for(int know=1;know<=length;know<<=1) doublesort(know);
    }
    ~SuffixArray() {
        delete [] rankOfIndex;
        delete [] indexOfRank;
        delete [] texi;
    }

    void firstsort(){
        for(int i=0;i<length;i++){
            texi[i]=Weight(i,refer[i]);
        }
        sort(&texi[0],&texi[length-1]+1);

        indexOfRank[rankOfIndex[texi[0].id]=0]=texi[0].id;
        int current=0;
        for(int i=1;i<length;i++){
            if(texi[i]!=texi[i-1]) current++;
            indexOfRank[i]=texi[i].id;
            rankOfIndex[texi[i].id]=current;
        }
    }

    void doublesort(int known){
        for(int i=0;i<length;i++){
            texi[i]=Weight(i,rankOfIndex[i],(i+known<length)?rankOfIndex[i+known]:-1);
        }

        sort(&texi[0],&texi[length-1]+1);

        indexOfRank[rankOfIndex[texi[0].id]=0]=texi[0].id;
        int current=0;
        for(int i=1;i<length;i++){
            if(texi[i]!=texi[i-1]) current++;
            indexOfRank[i]=texi[i].id;
            rankOfIndex[texi[i].id]=current;
        }

    }

    void print(int i,bool newline=0){
        printf("%s",&refer[indexOfRank[i]]);
        if(newline) printf("\n");
    }
```

```
    void printall(){
        for(int i=0;i<length;i++) print(i,1);
    }

    int *indexOfRank,*rankOfIndex,length;
    char *refer;
    Weight *texi;
};

int main()
{
    char str[100];
    scanf("%s", str);
    SuffixArray a(str);
    a.printall();
    return 0;
}
```

code/SuffixArray.cpp

```
import java.util.*;

/**
 * A magical data structure.
 * Written on 103.08.19
 */
public class Treap<K, V>{

  Random priorityGenerator;
  int time, size;
  Entry root;

  /**
   * Default Constructor
   */
  Treap(){
    root = null;
    time = size = 0;
    priorityGenerator = new Random();
  }

  /**
   * Find the Entry associated with key
   * @param key the key of the entry you are looking for
   * @return Entry
   */
  Entry find(K key){
    Entry now = root;
    Comparable<? super K> cmp = (Comparable<? super K>)key;
    int situation;
    while((now != null) && (situation=cmp.compareTo(now.key)) != 0){
      if(situation == -1) now = now.lchild;
      else now = now.rchild;
    }
    return now;
  }

  /**
   * Split the treap based on the key
   * Behavior undefined if the specified key is already in the tree
   * @param cmp Comparable based on the key
   * @return an array consists of two elements, the left subtree and the right
   */
  Entry[] split(Comparable<? super K> cmp){
    Entry leftTree = null, rightTree = null, left = null, right = null;
```

```
  Entry current = root;
  while(current != null){
    if(cmp.compareTo(current.key) == -1){
      if(right == null){
        right = rightTree = current;
      }else{
        current.parent = right;
        right = right.lchild = current;
      }
      current = current.lchild;
      right.lchild = null;
      if(current != null) current.parent = null;
    }else{
      if(left == null){
        left = leftTree = current;
      }else{
        current.parent = left;
        left = left.rchild = current;
      }
      current = current.rchild;
      left.rchild = null;
      if(current != null) current.parent = null;
    }
  }
  return new Treap.Entry[]{leftTree, rightTree};
}

/**
 * Merge two Treaps into one.
 * All keys of the entries in the left must be smaller than all keys of the entries in the right
 * @param left the left Treap, it must be smaller than the right Treap
 * @param right the right Treap, it must be greater than the left Treap
 * @return root of the resulting Treap
 */
Entry merge(Entry left, Entry right){
  if(left == null) return right;
  if(right == null) return left;
  if(left.compareTo(right) == -1){
    if(right.lchild == null){
      right.lchild = left;
      left.parent = right;
    }else if(right.lchild.compareTo(left) == -1){
      Entry temp = right.lchild;
      right.lchild = left;
      left.parent = right;
      temp.parent = null;
      merge(left, temp);
    }else{
      merge(left, right.lchild);
    }
    return right;
  }else{
    if(left.rchild == null){
      left.rchild = right;
      right.parent = left;
    }else if(left.rchild.compareTo(right) == -1){
      Entry temp = left.rchild;
      left.rchild = right;
      right.parent = left;
      temp.parent = null;
      merge(temp, right);
    }else{
      merge(left.rchild, right);
    }
    return left;
  }
```

```
}

/**
 * Insert a new Entry into the Treap if the key doesn't exists
 * Else replace the value with the new one and return the old value
 * @param key the key of the entry to be inserted or modified
 * @param value the new value of the entry
 * @return The original value if entry already exists, else return null;
 */
V puts(K key, V value){
  if(root == null){
    root = new Entry(key, value);
    size++;
    return null;
  }
  Entry position = find(key);
  if(position != null){
    V temp = position.value;
    position.value = value;
    return temp;
  }
  Entry newEntry = new Entry(key, value);
  Comparable<? super K> cmp = ((Comparable<? super K>)key);
  Entry[] subtree = split(cmp);
  newEntry = merge(subtree[0], newEntry);
  root = merge(newEntry, subtree[1]);
  size++;
  return null;
}

/**
 * Remove the entry associated with the specified key
 * return the according value upon removing
 * @param key the key of the entry to be destroyed
 * @return the value associated with the specified key, return null if no such key exists
 */
V remove(K key){
  Entry target = find(key);
  if(target == null) return null;
  if(target.lchild!=null) target.lchild.parent = null;
  if(target.rchild!=null) target.rchild.parent = null;
  Entry child = merge(target.lchild, target.rchild);
  if(child != null) child.parent = target.parent;
  if(target.parent != null){
    if(target == target.parent.lchild) target.parent.lchild = child;
    else if(target == target.parent.rchild) target.parent.rchild = child;
    else throw new AssertionError("remove fail");
  }else if(root == target) root = child;
  else throw new AssertionError("What is this?");
  size--;
  return target.value;
}

/**
 * This is a debugger
 * @param now the node doing a in order traversal
 * @return the size of the subtree rooted at now
 */
int iterate(Entry now, Entry parent){
  if(now == null) return 0;
  //System.out.println("Iterate "+now.key);
  if(now.parent != parent) System.out.println("Parent Check Fail!!!");
  int result = 1;
  result += iterate(now.lchild, now);
  //System.out.println("Entry : "+now.key);
  result += iterate(now.rchild, now);
```

```
    return result;
  }

  /**
   * The class storing all the entries of Treap
   * each Entry consists of a key and a value and a random generated priority
   * also stores its parent and children as well
   */
  class Entry implements Comparable<Entry>{
    Entry parent, lchild, rchild;
    Integer priority, timestamp;
    K key;
    V value;

    Entry(K key, V value){
      this.key = key;
      this.value = value;
      parent = lchild = rchild = null;
      priority = priorityGenerator.nextInt();
      timestamp = time++;
    }

    @Override
    public int compareTo(Entry rhs){
      int result = priority.compareTo(rhs.priority);
      if(result == 0) return timestamp.compareTo(rhs.timestamp);
      return result;
    }
  }

}
```

code/Treap.java