```
// Collecting Beepers
// DP TSP

#include <algorithm>
#include <cmath>
#include <cstdio>
#include <cstring>
using namespace std;

int i, j, TC, xsize, ysize, n, x[11], y[11], dist[11][11], memo[11][1 <<
11]; // Karel + max 10 beepers

int tsp(int pos, int bitmask) { // bitmask stores the visited coordinates
  if (bitmask == (1 << (n + 1)) - 1)
    return dist[pos][0]; // return trip to close the loop
  if (memo[pos][bitmask] != -1)
    return memo[pos][bitmask];

  int ans = 2000000000;
  for (int nxt = 0; nxt <= n; nxt++) // O(n) here
    if (nxt != pos && !(bitmask & (1 << nxt))) // if coordinate nxt is
not visited yet
      ans = min(ans, dist[pos][nxt] + tsp(nxt, bitmask | (1 << nxt)));
  return memo[pos][bitmask] = ans;
}

int main() {
  scanf("%d", &TC);
  while (TC--) {
    scanf("%d %d", &xsize, &ysize); // these two values are not used
    scanf("%d %d", &x[0], &y[0]);
    scanf("%d", &n);
    for (i = 1; i <= n; i++) // karel's position is at index 0
      scanf("%d %d", &x[i], &y[i]);

    for (i = 0; i <= n; i++) // build distance table
      for (j = 0; j <= n; j++)
        dist[i][j] = abs(x[i] - x[j]) + abs(y[i] - y[j]); // Manhattan
distance

    memset(memo, -1, sizeof memo);
    printf("The shortest path has length %d\n", tsp(0, 1)); // DP-TSP
  }

  return 0;
}
```

```
/* How do you add? */

// top-down

/*
#include <cstdio>
#include <cstring>
using namespace std;

int N, K, memo[110][110];

int ways(int N, int K) {
  if (K == 1) // only can use 1 number to add up to N
    return 1; // the answer is definitely 1, that number itself
  else if (memo[N][K] != -1)
    return memo[N][K];

  // if K > 1, we can choose one number from [0..N] to be one of the
number and recursively compute the rest
  int total_ways = 0;
  for (int split = 0; split <= N; split++)
    total_ways = (total_ways + ways(N - split, K - 1)) % 1000000; // we
just need the modulo 1M
  return memo[N][K] = total_ways; // memoize them
}

int main() {
  memset(memo, -1, sizeof memo);
  while (scanf("%d %d", &N, &K), (N || K)) // some recursion formula +
top down DP
    printf("%d\n", ways(N, K));
  return 0;
}
*/



// bottom-up

#include <cstdio>
#include <cstring>
using namespace std;

int main() {
  int i, j, split, dp[110][110], N, K;

  memset(dp, 0, sizeof dp);

  for (i = 0; i <= 100; i++) // these are the base cases
    dp[i][1] = 1;

  for (j = 1; j < 100; j++) // these three nested loops form the correct
topological order
    for (i = 0; i <= 100; i++)
      for (split = 0; split <= 100 - i; split++) {
        dp[i + split][j + 1] += dp[i][j];
        dp[i + split][j + 1] %= 1000000;
      }

  while (scanf("%d %d", &N, &K), (N || K))
```

```
    printf("%d\n", dp[N][K]);

  return 0;
}
```

```
// Cutting Sticks
// Top-Down DP

#include <algorithm>
#include <cstdio>
#include <cstring>
using namespace std;

int l, n, A[55], memo[55][55];

int cut(int left, int right) {
  if (left + 1 == right)       return 0;
  if (memo[left][right] != -1) return memo[left][right];

  int ans = 2000000000;
  for (int i = left + 1; i < right; i++)
    ans = min(ans, cut(left, i) + cut(i, right) + (A[right]-A[left]));
  return memo[left][right] = ans;
}

int main() {
  while (scanf("%d", &l), l) {
    A[0] = 0;
    scanf("%d", &n);
    for (int i = 1; i <= n; i++) scanf("%d", &A[i]);
    A[n + 1] = l;

    memset(memo, -1, sizeof memo);
    printf("The minimum cutting is %d.\n", cut(0, n + 1)); // start with
left = 0 and right = n + 1
  }

  return 0;
}
```

```
#include <algorithm>
#include <cstdio>
#include <vector>
using namespace std;

typedef pair<int, int> ii;      // In this chapter, we will frequently
use these
typedef vector<ii> vii;      // three data type shortcuts. They may look
cryptic
typedef vector<int> vi;   // but shortcuts are useful in competitive
programming

#define DFS_WHITE -1 // normal DFS, do not change this with other values
(other than 0), because we usually use memset with conjunction with
DFS_WHITE
#define DFS_BLACK 1

vector<vii> AdjList;

void printThis(char* message) {
  printf("================================\n");
  printf("%s\n", message);
  printf("================================\n");
}

vi dfs_num;     // this variable has to be global, we cannot put it in
recursion
int numCC;

void dfs(int u) {          // DFS for normal usage: as graph traversal
algorithm
  printf(" %d", u);                                    // this vertex is
visited
  dfs_num[u] = DFS_BLACK;      // important step: we mark this vertex as
visited
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];                       // v is a (neighbor,
weight) pair
    if (dfs_num[v.first] == DFS_WHITE)         // important check to
avoid cycle
      dfs(v.first);      // recursively visits unvisited neighbors v of
vertex u
} }

// note: this is not the version on implicit graph
void floodfill(int u, int color) {
  dfs_num[u] = color;                             // not just a generic
DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      floodfill(v.first, color);
} }

vi topoSort;             // global vector to store the toposort in
reverse order

void dfs2(int u) {    // change function name to differentiate with
original dfs
  dfs_num[u] = DFS_BLACK;
```

```
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      dfs2(v.first);
  }
  topoSort.push_back(u); }                    // that is, this is the only
change

#define DFS_GRAY 2              // one more color for graph edges
property check
vi dfs_parent;      // to differentiate real back edge versus
bidirectional edge

void graphCheck(int u) {                // DFS for checking graph edge
properties
  dfs_num[u] = DFS_GRAY;   // color this as DFS_GRAY (temp) instead of
DFS_BLACK
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {     // Tree Edge, DFS_GRAY to
DFS_WHITE
      dfs_parent[v.first] = u;                   // parent of this
children is me
      graphCheck(v.first);
    }
    else if (dfs_num[v.first] == DFS_GRAY) {             // DFS_GRAY to
DFS_GRAY
      if (v.first == dfs_parent[u])          // to differentiate these
two cases
        printf(" Bidirectional (%d, %d) - (%d, %d)\n", u, v.first,
v.first, u);
      else  // the most frequent application: check if the given graph is
cyclic
        printf(" Back Edge (%d, %d) (Cycle)\n", u, v.first);
    }
    else if (dfs_num[v.first] == DFS_BLACK)             // DFS_GRAY to
DFS_BLACK
      printf(" Forward/Cross Edge (%d, %d)\n", u, v.first);
  }
  dfs_num[u] = DFS_BLACK;     // after recursion, color this as DFS_BLACK
(DONE)
}

vi dfs_low;       // additional information for articulation
points/bridges/SCCs
vi articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <=
dfs_num[u]
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE) {                          // a
tree edge
      dfs_parent[v.first] = u;
      if (u == dfsRoot) rootChildren++;  // special case, count children
of root

      articulationPointAndBridge(v.first);
```

```
      if (dfs_low[v.first] >= dfs_num[u])              // for
articulation point
        articulation_vertex[u] = true;           // store this
information first
      if (dfs_low[v.first] > dfs_num[u])                            // for
bridge
        printf(" Edge (%d, %d) is a bridge\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);       // update
dfs_low[u]
    }
    else if (v.first != dfs_parent[u])       // a back edge and not
direct cycle
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);       // update
dfs_low[u]
} }

vi S, visited;                                       // additional global
variables
int numSCC;

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;      // dfs_low[u] <=
dfs_num[u]
  S.push_back(u);           // stores u in a vector based on order of
visitation
  visited[u] = 1;
  for (int j = 0; j < (int)AdjList[u].size(); j++) {
    ii v = AdjList[u][j];
    if (dfs_num[v.first] == DFS_WHITE)
      tarjanSCC(v.first);
    if (visited[v.first])                                  // condition for
update
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
  }

  if (dfs_low[u] == dfs_num[u]) {         // if this is a root (start) of
an SCC
    printf("SCC %d:", ++numSCC);            // this part is done after
recursion
    while (1) {
      int v = S.back(); S.pop_back(); visited[v] = 0;
      printf(" %d", v);
      if (u == v) break;
    }
    printf("\n");
} }

int main() {
  int V, total_neighbors, id, weight;

  /*
  // Use the following input:
  // Graph in Figure 4.1
  9
  1 1 0
  3 0 0 2 0 3 0
  2 1 0 3 0
  3 1 0 2 0 4 0
  1 3 0
```

```
  0
  2 7 0 8 0
  1 6 0
  1 6 0

  // Example of directed acyclic graph in Figure 4.4 (for toposort)
  8
  2 1 0 2 0
  2 2 0 3 0
  2 3 0 5 0
  1 4 0
  0
  0
  0
  1 6 0

  // Example of directed graph with back edges
  3
  1 1 0
  1 2 0
  1 0 0

  // Left graph in Figure 4.6/4.7/4.8
  6
  1 1 0
  3 0 0 2 0 4 0
  1 1 0
  1 4 0
  3 1 0 3 0 5 0
  1 4 0

  // Right graph in Figure 4.6/4.7/4.8
  6
  1 1 0
  5 0 0 2 0 3 0 4 0 5 0
  1 1 0
  1 1 0
  2 1 0 5 0
  2 1 0 4 0

  // Directed graph in Figure 4.9
  8
  1 1 0
  1 3 0
  1 1 0
  2 2 0 4 0
  1 5 0
  1 7 0
  1 4 0
  1 6 0
  */

  freopen("in_01.txt", "r", stdin);

  scanf("%d", &V);
  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to
AdjList
  for (int i = 0; i < V; i++) {
    scanf("%d", &total_neighbors);
    for (int j = 0; j < total_neighbors; j++) {
```

```
      scanf("%d %d", &id, &weight);
      AdjList[i].push_back(ii(id, weight));
    }
  }

  printThis("Standard DFS Demo (the input graph must be UNDIRECTED)");
  numCC = 0;
  dfs_num.assign(V, DFS_WHITE);    // this sets all vertices' state to
DFS_WHITE
  for (int i = 0; i < V; i++)                  // for each vertex i in
[0..V-1]
    if (dfs_num[i] == DFS_WHITE)            // if that vertex is not
visited yet
      printf("Component %d:", ++numCC), dfs(i), printf("\n");   // 3
lines here!
  printf("There are %d connected components\n", numCC);

  printThis("Flood Fill Demo (the input graph must be UNDIRECTED)");
  numCC = 0;
  dfs_num.assign(V, DFS_WHITE);
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      floodfill(i, ++numCC);
  for (int i = 0; i < V; i++)
    printf("Vertex %d has color %d\n", i, dfs_num[i]);

  // make sure that the given graph is DAG
  printThis("Topological Sort (the input graph must be DAG)");
  topoSort.clear();
  dfs_num.assign(V, DFS_WHITE);
  for (int i = 0; i < V; i++)            // this part is the same as
finding CCs
    if (dfs_num[i] == DFS_WHITE)
      dfs2(i);
  reverse(topoSort.begin(), topoSort.end());                 // reverse
topoSort
  for (int i = 0; i < (int)topoSort.size(); i++)       // or you can
simply read
    printf(" %d", topoSort[i]);           // the content of `topoSort'
backwards
  printf("\n");

  printThis("Graph Edges Property Check");
  numCC = 0;
  dfs_num.assign(V, DFS_WHITE); dfs_parent.assign(V, -1);
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      printf("Component %d:\n", ++numCC), graphCheck(i);       // 2 lines
in one

  printThis("Articulation Points & Bridges (the input graph must be
UNDIRECTED)");
  dfsNumberCounter = 0; dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V,
0);
  dfs_parent.assign(V, -1); articulation_vertex.assign(V, 0);
  printf("Bridges:\n");
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE) {
      dfsRoot = i; rootChildren = 0;
      articulationPointAndBridge(i);
```

```
      articulation_vertex[dfsRoot] = (rootChildren > 1); }       //
special case
  printf("Articulation Points:\n");
  for (int i = 0; i < V; i++)
    if (articulation_vertex[i])
      printf(" Vertex %d\n", i);

  printThis("Strongly Connected Components (the input graph must be
DIRECTED)");
  dfs_num.assign(V, DFS_WHITE); dfs_low.assign(V, 0); visited.assign(V,
0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < V; i++)
    if (dfs_num[i] == DFS_WHITE)
      tarjanSCC(i);

  return 0;
}
```

```
/* Wetlands of Florida */

// classic DFS flood fill

#include <cstdio>
#include <cstring>
using namespace std;

#define REP(i, a, b) \
  for (int i = int(a); i <= int(b); i++)

char line[150], grid[150][150];
int TC, R, C, row, col;

int dr[] = {1,1,0,-1,-1,-1, 0, 1}; // S,SE,E,NE,N,NW,W,SW
int dc[] = {0,1,1, 1, 0,-1,-1,-1}; // neighbors
int floodfill(int r, int c, char c1, char c2) {
  if (r<0 || r>=R || c<0 || c>=C) return 0; // outside
  if (grid[r][c] != c1) return 0; // we want only c1
  grid[r][c] = c2; // important step to avoid cycling!
  int ans = 1; // coloring c1 -> c2, add 1 to answer
  REP (d, 0, 7) // recurse to neighbors
    ans += floodfill(r + dr[d], c + dc[d], c1, c2);
  return ans;
}

// inside the int main() of the solution for UVa 469 - Wetlands of
Florida
int main() {
  // read the implicit graph as global 2D array 'grid'/R/C and (row, col)
query coordinate
  sscanf(gets(line), "%d", &TC);
  gets(line); // remove dummy line

  while (TC--) {
    R = 0;
    while (1) {
      gets(grid[R]);
      if (grid[R][0] != 'L' && grid[R][0] != 'W') // start of query
        break;
      R++;
    }
    C = (int)strlen(grid[0]);

    strcpy(line, grid[R]);
    while (1) {
      sscanf(line, "%d %d", &row, &col); row--; col--; // index starts
from 0!
      printf("%d\n", floodfill(row, col, 'W', '.')); // change water 'W'
to '.'; count size of this lake
      floodfill(row, col, '.', 'W'); // restore for next query
      gets(line);
      if (strcmp(line, "") == 0 || feof(stdin)) // next test case or last
test case
        break;
    }

    if (TC)
      printf("\n");
  }
```

```
  return 0;
}
```

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;

// Union-Find Disjoint Sets Library written in OOP manner, using both
path compression and union by rank heuristics
class UnionFind {                                               // OOP
style
private:
  vi p, rank, setSize;                          // remember: vi is
vector<int>
  int numSets;
public:
  UnionFind(int N) {
    setSize.assign(N, 1); numSets = N; rank.assign(N, 0);
    p.assign(N, 0); for (int i = 0; i < N; i++) p[i] = i; }
  int findSet(int i) { return (p[i] == i) ? i : (p[i] = findSet(p[i])); }
  bool isSameSet(int i, int j) { return findSet(i) == findSet(j); }
  void unionSet(int i, int j) {
    if (!isSameSet(i, j)) { numSets--;
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y]) { p[y] = x; setSize[x] += setSize[y]; }
    else                   { p[x] = y; setSize[y] += setSize[x];
                             if (rank[x] == rank[y]) rank[y]++; } } }
  int numDisjointSets() { return numSets; }
  int sizeOfSet(int i) { return setSize[findSet(i)]; }
};

vector<vii> AdjList;
vi taken;                                    // global boolean flag to
avoid cycle
priority_queue<ii> pq;            // priority queue to help choose
shorter edges

void process(int vtx) {    // so, we use -ve sign to reverse the sort
order
  taken[vtx] = 1;
  for (int j = 0; j < (int)AdjList[vtx].size(); j++) {
    ii v = AdjList[vtx][j];
    if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
} }                                  // sort by (inc) weight then by (inc)
id

int main() {
  int V, E, u, v, w;

  /*
  // Graph in Figure 4.10 left, format: list of weighted edges
  // This example shows another form of reading graph input
  5 7
  0 1 4
  0 2 4
  0 3 6
```

```
  0 4 6
  1 2 2
  2 3 8
  3 4 9
  */

  freopen("in_03.txt", "r", stdin);

  scanf("%d %d", &V, &E);
  // Kruskal's algorithm merged with Prim's algorithm
  AdjList.assign(V, vii());
  vector< pair<int, ii> > EdgeList;   // (weight, two vertices) of the
edge
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);            // read the triple: (u, v,
w)
    EdgeList.push_back(make_pair(w, ii(u, v)));                // (w, u,
v)
    AdjList[u].push_back(ii(v, w));
    AdjList[v].push_back(ii(u, w));
  }
  sort(EdgeList.begin(), EdgeList.end()); // sort by edge weight O(E log
E)
                      // note: pair object has built-in comparison
function

  int mst_cost = 0;
  UnionFind UF(V);                      // all V are disjoint sets
initially
  for (int i = 0; i < E; i++) {                       // for each edge,
O(E)
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.second)) {  //
check
      mst_cost += front.first;                // add the weight of e to
MST
      UF.unionSet(front.second.first, front.second.second);    // link
them
  } }                       // note: the runtime cost of UFDS is very
light

  // note: the number of disjoint sets must eventually be 1 for a valid
MST
  printf("MST cost = %d (Kruskal's)\n", mst_cost);



// inside int main() --- assume the graph is stored in AdjList, pq is
empty
  taken.assign(V, 0);                // no vertex is taken at the
beginning
  process(0);   // take vertex 0 and process all edges incident to vertex
0
  mst_cost = 0;
  while (!pq.empty()) {  // repeat until V vertices (E=V-1 edges) are
taken
    ii front = pq.top(); pq.pop();
    u = -front.second, w = -front.first;  // negate the id and weight
again
```

```
    if (!taken[u])                  // we have not connected this vertex
yet
      mst_cost += w, process(u); // take u, process all edges incident to
u
  }                                          // each edge is in pq only
once!
  printf("MST cost = %d (Prim's)\n", mst_cost);

  return 0;
}
```

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;      // In this chapter, we will frequently
use these
typedef vector<ii> vii;      // three data type shortcuts. They may look
cryptic
typedef vector<int> vi;   // but shortcuts are useful in competitive
programming

int V, E, a, b, s;
vector<vii> AdjList;
vi p;                                  // addition: the predecessor/parent
vector

void printPath(int u) {    // simple function to extract information from
`vi p'
  if (u == s) { printf("%d", u); return; }
  printPath(p[u]);   // recursive call: to make the output format: s ->
... -> t
  printf(" %d", u); }

int main() {
  /*
  // Graph in Figure 4.3, format: list of unweighted edges
  // This example shows another form of reading graph input
  13 16
  0 1    1 2    2  3   0  4   1  5   2  6    3  7   5  6
  4 8    8 9    5 10   6 11   7 12   9 10   10 11  11 12
  */

  freopen("in_04.txt", "r", stdin);

  scanf("%d %d", &V, &E);

  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to
AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d", &a, &b);
    AdjList[a].push_back(ii(b, 0));
    AdjList[b].push_back(ii(a, 0));
  }

  // as an example, we start from this source, see Figure 4.3
  s = 5;

  // BFS routine
  // inside int main() -- we do not use recursion, thus we do not need to
create separate function!
  vi dist(V, 1000000000); dist[s] = 0;      // distance to source is 0
(default)
  queue<int> q; q.push(s);                                  // start from
source
  p.assign(V, -1); // to store parent information (p must be a global
variable!)
  int layer = -1;                            // for our output printing
purpose
```

```
  bool isBipartite = true;       // addition of one boolean flag,
initially true

  while (!q.empty()) {
    int u = q.front(); q.pop();                          // queue: layer by
layer!
    if (dist[u] != layer) printf("\nLayer %d: ", dist[u]);
    layer = dist[u];
    printf("visit %d, ", u);
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                              // for each
neighbors of u
      if (dist[v.first] == 1000000000) {
        dist[v.first] = dist[u] + 1;                     // v unvisited +
reachable
        p[v.first] = u;          // addition: the parent of vertex v-
>first is u
        q.push(v.first);                                 // enqueue v for
next step
      }
      else if ((dist[v.first] % 2) == (dist[u] % 2))              // same
parity
        isBipartite = false;
  } }

  printf("\nShortest path: ");
  printPath(7), printf("\n");
  printf("isBipartite? %d\n", isBipartite);

  return 0;
}
```

```
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1000000000

int main() {
  int V, E, s, u, v, w;
  vector<vii> AdjList;

  /*
  // Graph in Figure 4.17
  5 7 2
  2 1 2
  2 3 7
  2 0 6
  1 3 3
  1 4 6
  3 4 5
  0 4 1
  */

  freopen("in_05.txt", "r", stdin);

  scanf("%d %d %d", &V, &E, &s);

  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to
AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjList[u].push_back(ii(v, w));                              //
directed graph
  }

  // Dijkstra routine
  vi dist(V, INF); dist[s] = 0;                    // INF = 1B to avoid
overflow
  priority_queue< ii, vector<ii>, greater<ii> > pq; pq.push(ii(0, s));
                            // ^to sort the pairs by increasing distance
from s
  while (!pq.empty()) {                                                //
main loop
    ii front = pq.top(); pq.pop();     // greedy: pick shortest unvisited
vertex
    int d = front.first, u = front.second;
    if (d > dist[u]) continue;   // this check is important, see the
explanation
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];                        // all outgoing edges
from u
      if (dist[u] + v.second < dist[v.first]) {
        dist[v.first] = dist[u] + v.second;                // relax
operation
        pq.push(ii(dist[v.first], v.first));
  } } }  // note: this variant can cause duplicate items in the priority
queue
```

```
  for (int i = 0; i < V; i++) // index + 1 for final answer
    printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

  return 0;
}
```

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define INF 1000000000

int main() {
  int V, E, s, a, b, w;
  vector<vii> AdjList;

  /*
  // Graph in Figure 4.18, has negative weight, but no negative cycle
  5 5 0
  0 1 1
  0 2 10
  1 3 2
  2 3 -10
  3 4 3

  // Graph in Figure 4.19, negative cycle exists
  3 3 0
  0 1 1000
  1 2 15
  2 1 -42
  */

  freopen("in_06.txt", "r", stdin);

  scanf("%d %d %d", &V, &E, &s);

  AdjList.assign(V, vii()); // assign blank vectors of pair<int, int>s to
AdjList
  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &a, &b, &w);
    AdjList[a].push_back(ii(b, w));
  }

  // Bellman Ford routine
  vi dist(V, INF); dist[s] = 0;
  for (int i = 0; i < V - 1; i++)  // relax all E edges V-1 times,
overall O(VE)
    for (int u = 0; u < V; u++)                         // these two loops
= O(E)
      for (int j = 0; j < (int)AdjList[u].size(); j++) {
        ii v = AdjList[u][j];        // we can record SP spanning here if
needed
        dist[v.first] = min(dist[v.first], dist[u] + v.second);
// relax
      }

  bool hasNegativeCycle = false;
  for (int u = 0; u < V; u++)                           // one more pass
to check
    for (int j = 0; j < (int)AdjList[u].size(); j++) {
      ii v = AdjList[u][j];
```

```
      if (dist[v.first] > dist[u] + v.second)                 // should
be false
        hasNegativeCycle = true;     // but if true, then negative cycle
exists!
    }
  printf("Negative Cycle Exist? %s\n", hasNegativeCycle ? "Yes" : "No");

  if (!hasNegativeCycle)
    for (int i = 0; i < V; i++)
      printf("SSSP(%d, %d) = %d\n", s, i, dist[i]);

  return 0;
}
```

```
#include <algorithm>
#include <cstdio>
using namespace std;

#define INF 1000000000

int main() {
  int V, E, u, v, w, AdjMatrix[200][200];

  /*
  // Graph in Figure 4.30
  5 9
  0 1 2
  0 2 1
  0 4 3
  1 3 4
  2 1 1
  2 4 1
  3 0 1
  3 2 3
  3 4 5
  */

  freopen("in_07.txt", "r", stdin);

  scanf("%d %d", &V, &E);
  for (int i = 0; i < V; i++) {
    for (int j = 0; j < V; j++)
      AdjMatrix[i][j] = INF;
    AdjMatrix[i][i] = 0;
  }

  for (int i = 0; i < E; i++) {
    scanf("%d %d %d", &u, &v, &w);
    AdjMatrix[u][v] = w; // directed graph
  }

  for (int k = 0; k < V; k++) // common error: remember that loop order
is k->i->j
    for (int i = 0; i < V; i++)
      for (int j = 0; j < V; j++)
        AdjMatrix[i][j] = min(AdjMatrix[i][j], AdjMatrix[i][k] +
AdjMatrix[k][j]);

  for (int i = 0; i < V; i++)
    for (int j = 0; j < V; j++)
      printf("APSP(%d, %d) = %d\n", i, j, AdjMatrix[i][j]);

  return 0;
}
```

```
#include <algorithm>
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef vector<int> vi;

#define MAX_V 40 // enough for sample graph in Figure 4.24/4.25/4.26/UVa
259
#define INF 1000000000

int res[MAX_V][MAX_V], mf, f, s, t;                          // global
variables
vi p;

void augment(int v, int minEdge) {     // traverse BFS spanning tree from
s to t
  if (v == s) { f = minEdge; return; }  // record minEdge in a global
variable f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v])); //
recursive
                         res[p[v]][v] -= f; res[v][p[v]] += f; }       //
update
}

int main() {
  int V, k, vertex, weight;

  /*
  // Graph in Figure 4.24
  4 0 1
  2 2 70 3 30
  2 2 25 3 70
  3 0 70 3 5 1 25
  3 0 30 2 5 1 70

  // Graph in Figure 4.25
  4 0 3
  2 1 100 3 100
  2 2 1 3 100
  1 3 100
  0

  // Graph in Figure 4.26.A
  5 1 0
  0
  2 2 100 3 50
  3 3 50 4 50 0 50
  1 4 100
  1 0 125

  // Graph in Figure 4.26.B
  5 1 0
  0
  2 2 100 3 50
  3 3 50 4 50 0 50
  1 4 100
  1 0 75
```

```
  // Graph in Figure 4.26.C
  5 1 0
  0
  2 2 100 3 50
  2 4 5 0 5
  1 4 100
  1 0 125
  */

  freopen("in_08.txt", "r", stdin);

  scanf("%d %d %d", &V, &s, &t);

  memset(res, 0, sizeof res);
  for (int i = 0; i < V; i++) {
    scanf("%d", &k);
    for (int j = 0; j < k; j++) {
      scanf("%d %d", &vertex, &weight);
      res[i][vertex] = weight;
    }
  }

  mf = 0;                                                  // mf stands for
max_flow
  while (1) {              // O(VE^2) (actually O(V^3E) Edmonds Karp's
algorithm
    f = 0;
    // run BFS, compare with the original BFS shown in Section 4.2.2
    vi dist(MAX_V, INF); dist[s] = 0; queue<int> q; q.push(s);
    p.assign(MAX_V, -1);           // record the BFS spanning tree, from
s to t!
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;      // immediately stop BFS if we already reach
sink t
      for (int v = 0; v < MAX_V; v++)                 // note: this part
is slow
        if (res[u][v] > 0 && dist[v] == INF)
          dist[v] = dist[u] + 1, q.push(v), p[v] = u;
    }
    augment(t, INF);     // find the min edge weight `f' along this path,
if any
    if (f == 0) break;      // we cannot send any more flow (`f' = 0),
terminate
    mf += f;                 // we can still send a flow, increase the
max flow!
  }

  printf("%d\n", mf);                                  // this is the max
flow value

  return 0;
}



/*

#include <algorithm>
#include <bitset>
```

```
#include <cstdio>
#include <vector>
#include <queue>
using namespace std;

typedef vector<int> vi;

#define MAX_V 40 // enough for sample graph in Figure 4.24/4.25/4.26/UVa
259
#define INF 1000000000

int res[MAX_V][MAX_V], mf, f, s, t;                              // global
variables
vi p;
vector<vi> AdjList;

void augment(int v, int minEdge) {     // traverse BFS spanning tree from
s to t
  if (v == s) { f = minEdge; return; }  // record minEdge in a global
variable f
  else if (p[v] != -1) { augment(p[v], min(minEdge, res[p[v]][v])); //
recursive
                         res[p[v]][v] -= f; res[v][p[v]] += f; }       //
update
}

int main() {
  int V, k, vertex, weight;

  scanf("%d %d %d", &V, &s, &t);

  memset(res, 0, sizeof res);
  AdjList.assign(V, vi());
  for (int i = 0; i < V; i++) {
    scanf("%d", &k);
    for (int j = 0; j < k; j++) {
      scanf("%d %d", &vertex, &weight);
      res[i][vertex] = weight;
      AdjList[i].push_back(vertex);
    }
  }

  mf = 0;
  while (1) {                    // now a true O(VE^2) Edmonds Karp's
algorithm
    f = 0;
    bitset<MAX_V> vis; vis[s] = true;            // we change vi dist to
bitset!
    queue<int> q; q.push(s);
    p.assign(MAX_V, -1);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (int j = 0; j < (int)AdjList[u].size(); j++) {  // we use
AdjList here!
        int v = AdjList[u][j];
        if (res[u][v] > 0 && !vis[v])
          vis[v] = true, q.push(v), p[v] = u;
      }
    }
```

```
    augment(t, INF);
    if (f == 0) break;
    mf += f;
  }

  printf("%d\n", mf);                                   // this is the max
flow value

  return 0;
}

*/
```

```
#include <cstdio>
#include <iostream>
#include <vector>
using namespace std;

typedef pair<int, int> ii;
typedef vector<int> vi;

vector<vi> AdjList;
vi match, vis;                                          // global
variables

int Aug(int l) {                 // return 1 if an augmenting path is
found
  if (vis[l]) return 0;                                 // return 0
otherwise
  vis[l] = 1;
  for (int j = 0; j < (int)AdjList[l].size(); j++) {
    int r = AdjList[l][j];
    if (match[r] == -1 || Aug(match[r])) {
      match[r] = l; return 1;                           // found 1
matching
  } }
  return 0;                                                 // no
matching
}

bool isprime(int v) {
  int primes[10] = {2,3,5,7,11,13,17,19,23,29};
  for (int i = 0; i < 10; i++)
    if (primes[i] == v)
      return true;
  return false;
}

int main() {
// inside int main()
  // build bipartite graph with directed edge from left to right set

/*
  // Graph in Figure 4.40 can be built on the fly
  // we know there are 6 vertices in this bipartite graph, left side are
numbered 0,1,2, right side 3,4,5
  int V = 6, Vleft = 3, set1[3] = {1,7,11}, set2[3] = {4,10,12};

  // Graph in Figure 4.41 can be built on the fly
  // we know there are 5 vertices in this bipartite graph, left side are
numbered 0,1, right side 3,4,5
  //int V = 5, Vleft = 2, set1[2] = {1,7}, set2[3] = {4,10,12};

  // build the bipartite graph, only directed edge from left to right is
needed
  AdjList.assign(V, vi());
  for (int i = 0; i < Vleft; i++)
    for (int j = 0; j < 3; j++)
      if (isprime(set1[i] + set2[j]))
        AdjList[i].push_back(3 + j);
*/
```

```
  // For bipartite graph in Figure 4.44, V = 5, Vleft = 3 (vertex 0
unused)
  // AdjList[0] = {} // dummy vertex, but you can choose to use this
vertex
  // AdjList[1] = {3, 4}
  // AdjList[2] = {3}
  // AdjList[3] = {}   // we use directed edges from left to right set
only
  // AdjList[4] = {}

  int V = 5, Vleft = 3;                                  // we ignore vertex
0
  AdjList.assign(V, vi());
  AdjList[1].push_back(3); AdjList[1].push_back(4);
  AdjList[2].push_back(3);

  int MCBM = 0;
  match.assign(V, -1);    // V is the number of vertices in bipartite
graph
  for (int l = 0; l < Vleft; l++) {         // Vleft = size of the left
set
    vis.assign(Vleft, 0);                    // reset before each
recursion
    MCBM += Aug(l);
  }
  printf("Found %d matchings\n", MCBM);  // the answer is 2 for Figure
4.42

  return 0;
}
```

```
#include <bitset>   // compact STL for Sieve, more efficient than
vector<bool>!
#include <cmath>
#include <cstdio>
#include <map>
#include <vector>
using namespace std;

typedef long long ll;
typedef vector<int> vi;
typedef map<int, int> mii;

ll _sieve_size;
bitset<10000010> bs;   // 10^7 should be enough for most cases
vi primes;   // compact list of primes in form of vector<int>


// first part

void sieve(ll upperbound) {          // create list of primes in
[0..upperbound]
  _sieve_size = upperbound + 1;                   // add 1 to include
upperbound
  bs.set();                                                // set all
bits to 1
  bs[0] = bs[1] = 0;                                     // except index
0 and 1
  for (ll i = 2; i <= _sieve_size; i++) if (bs[i]) {
    // cross out multiples of i starting from i * i!
    for (ll j = i * i; j <= _sieve_size; j += i) bs[j] = 0;
    primes.push_back((int)i);  // also add this vector containing list of
primes
} }                                             // call this method in main
method

bool isPrime(ll N) {                 // a good enough deterministic prime
tester
  if (N <= _sieve_size) return bs[N];                   // O(1) for small
primes
  for (int i = 0; i < (int)primes.size(); i++)
    if (N % primes[i] == 0) return false;
  return true;                    // it takes longer time if N is a large
prime!
}                     // note: only work for N <= (last prime in vi
"primes")^2


// second part

vi primeFactors(ll N) {   // remember: vi is vector of integers, ll is
long long
  vi factors;                    // vi `primes' (generated by sieve) is
optional
  ll PF_idx = 0, PF = primes[PF_idx];     // using PF = 2, 3, 4, ..., is
also ok
  while (N != 1 && (PF * PF <= N)) {   // stop at sqrt(N), but N can get
smaller
    while (N % PF == 0) { N /= PF; factors.push_back(PF); }    // remove
this PF
```

```
    PF = primes[++PF_idx];                              // only consider
primes!
  }
  if (N != 1) factors.push_back(N);     // special case if N is actually
a prime
  return factors;         // if pf exceeds 32-bit integer, you have to
change vi
}


// third part

ll numPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF * PF <= N)) {
    while (N % PF == 0) { N /= PF; ans++; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}

ll numDiffPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF * PF <= N)) {
    if (N % PF == 0) ans++;                             // count this pf
only once
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans++;
  return ans;
}

ll sumPF(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 0;
  while (N != 1 && (PF * PF <= N)) {
    while (N % PF == 0) { N /= PF; ans += PF; }
    PF = primes[++PF_idx];
  }
  if (N != 1) ans += N;
  return ans;
}

ll numDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;              // start from
ans = 1
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;                                           // count
the power
    while (N % PF == 0) { N /= PF; power++; }
    ans *= (power + 1);                                 // according to the
formula
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= 2;             // (last factor has pow = 1, we add 1
to it)
  return ans;
}
```

```
ll sumDiv(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = 1;             // start from
ans = 1
  while (N != 1 && (PF * PF <= N)) {
    ll power = 0;
    while (N % PF == 0) { N /= PF; power++; }
    ans *= ((ll)pow((double)PF, power + 1.0) - 1) / (PF - 1);         //
formula
    PF = primes[++PF_idx];
  }
  if (N != 1) ans *= ((ll)pow((double)N, 2.0) - 1) / (N - 1);        //
last one
  return ans;
}

ll EulerPhi(ll N) {
  ll PF_idx = 0, PF = primes[PF_idx], ans = N;             // start from
ans = N
  while (N != 1 && (PF * PF <= N)) {
    if (N % PF == 0) ans -= ans / PF;                // only count unique
factor
    while (N % PF == 0) N /= PF;
    PF = primes[++PF_idx];
  }
  if (N != 1) ans -= ans / N;                                      // last
factor
  return ans;
}

int main() {
  // first part: the Sieve of Eratosthenes
  sieve(10000000);                         // can go up to 10^7 (need few
seconds)
  printf("%d\n", isPrime(2147483647));                          // 10-
digits prime
  printf("%d\n", isPrime(136117223861LL));        // not a prime,
104729*1299709


  // second part: prime factors
  vi res = primeFactors(2147483647);   // slowest, 2147483647 is a prime
  for (vi::iterator i = res.begin(); i != res.end(); i++) printf(">
%d\n", *i);

  res = primeFactors(136117223861LL);   // slow, 2 large pfactors
104729*1299709
  for (vi::iterator i = res.begin(); i != res.end(); i++) printf("#
%d\n", *i);

  res = primeFactors(142391208960LL);   // faster, 2^10*3^4*5*7^4*11*13
  for (vi::iterator i = res.begin(); i != res.end(); i++) printf("!
%d\n", *i);

  //res = primeFactors((ll)(1010189899 * 1010189899)); // "error"
  //for (vi::iterator i = res.begin(); i != res.end(); i++) printf("^
%d\n", *i);


  // third part: prime factors variants
  printf("numPF(%d) = %lld\n", 50, numPF(50)); // 2^1 * 5^2 => 3
```

```
  printf("numDiffPF(%d) = %lld\n", 50, numDiffPF(50)); // 2^1 * 5^2 => 2
  printf("sumPF(%d) = %lld\n", 50, sumPF(50)); // 2^1 * 5^2 => 2 + 5 + 5
= 12
  printf("numDiv(%d) = %lld\n", 50, numDiv(50)); // 1, 2, 5, 10, 25, 50,
6 divisors
  printf("sumDiv(%d) = %lld\n", 50, sumDiv(50)); // 1 + 2 + 5 + 10 + 25 +
50 = 93
  printf("EulerPhi(%d) = %lld\n", 50, EulerPhi(50)); // 20 integers < 50
are relatively prime with 50

  return 0;
}
```

```
// Pseudo-Random Numbers

#include <cstdio>
#include <iostream>
using namespace std;

typedef pair<int, int> ii;

int caseNo = 1, Z, I, M, L;

int f(int x) { return (Z * x + I) % M; }

ii floydCycleFinding(int x0) {  // function int f(int x) is defined
earlier
  // 1st part: finding k*mu, hare's speed is 2x tortoise's
  int tortoise = f(x0), hare = f(f(x0));    // f(x0) is the node next to
x0
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(f(hare)); }
  // 2nd part: finding mu, hare and tortoise move at the same speed
  int mu = 0; hare = x0;
  while (tortoise != hare) { tortoise = f(tortoise); hare = f(hare);
mu++; }
  // 3rd part: finding lambda, hare moves, tortoise stays
  int lambda = 1; hare = f(tortoise);
  while (tortoise != hare) { hare = f(hare); lambda++; }
  return ii(mu, lambda);
}

int main() {
  while (scanf("%d %d %d %d", &Z, &I, &M, &L), (Z || I || M || L)) {
    ii result = floydCycleFinding(L);
    printf("Case %d: %d\n", caseNo++, result.second);
  }
  return 0;
}
```

```
#include <algorithm>
#include <ctype.h> // no equivalent C++ version... note that C++ can use
C features...
#include <iostream>
#include <map>
#include <fstream>
#include <sstream>
#include <string> // string class
#include <string.h>
#include <vector>
using namespace std;

int isvowel(char ch) { // make sure ch is in lowercase
  char vowel[6] = "aeiou";
  for (int j = 0; vowel[j]; j++)
    if (vowel[j] == ch)
      return 1;
  return 0;
}

int main() {
  int i, pos, digits, alphas, vowels, consonants;
  bool first = true, prev_dash, this_dash;
  char str[10010], line[110], *p;

  freopen("ch6.txt", "r", stdin);

  strcpy(str, "");
  first = true; // technique to differentiate first line with the other
lines
  prev_dash = this_dash = false; // to differentiate whether the previous
line contains a dash or not
  while (1) {
    fgets(line, 100, stdin);
    line[(int)strlen(line) - 2] = 0; // delete dummy char
    if (strncmp(line, ".......", 7) == 0) break;
    if (line[(int)strlen(line) - 1] == '-') {
      line[(int)strlen(line) - 1] = 0; // if the last character is '-',
delete it by moving the NULL (0) one character forward
      this_dash = true;
    }
    else
      this_dash = false;
    if (!first && !prev_dash)
      strcat(str, " "); // only append " " if this line is the second one
onwards
    first = false;
    strcat(str, line);
    prev_dash = this_dash;
  }

  for (i = digits = alphas = vowels = consonants = 0; str[i]; i++) { //
we can use str[i] as terminating condition as string in C++ is also
terminated with NULL (0)
    str[i] = tolower(str[i]); // make each character lower case
    digits += isdigit(str[i]) ? 1 : 0;
    alphas += isalpha(str[i]) ? 1 : 0;
    vowels += isvowel(str[i]); // already returns 1 or 0
  }
  consonants = alphas - vowels;
```

```
  printf("%s\n", str);
  printf("%d %d %d\n", digits, vowels, consonants);
  int hascs3233 = (strstr(str, "cs3233") != NULL);

  vector<string> tokens;
  map<string, int> freq;
  for (p = strtok(str, " ."); p; p = strtok(NULL, " .")) {
    tokens.push_back(p); // casting from C string to C++ string is
automatic
    freq[p]++;
  }

  sort(tokens.begin(), tokens.end());
  printf("%s %s\n", tokens[0].c_str(), tokens[(int)tokens.size() -
1].c_str()); // to cast C++ string to C string, we need to use c_str()
  printf("%d\n", hascs3233);

  int ans_s = 0, ans_h = 0, ans_7 = 0;
  char ch;
  while (scanf("%c", &ch), ch != '\n') {
         if (ch == 's') ans_s++;
    else if (ch == 'h') ans_h++;
    else if (ch == '7') ans_7++;
  }
  printf("%d %d %d\n", ans_s, ans_h, ans_7);

  return 0;
}
```

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

class ch6_01_basic_string {
  static int isvowel(char ch) { // make sure ch is in lowercase
    String vowel = "aeiou";
    for (int j = 0; j < 5; j++)
      if (vowel.charAt(j) == ch)
        return 1;
    return 0;
  }

  public static void main(String[] args) throws Exception {
    int i, pos, digits, alphas, vowels, consonants;
    Boolean first, prev_dash, this_dash;
    String str = "";
    first = true; // technique to differentiate first line with the other
lines
    prev_dash = this_dash = false; // to differentiate whether the
previous line contains a dash or not

    File f = new File("ch6.txt");
    Scanner sc = new Scanner(f);
    while (sc.hasNext()) {
      String line = sc.nextLine();
      if (line.equals(".......")) break;
      if (line.charAt(line.length() - 1) == '-') {
        line = line.substring(0, line.length() - 1); // if the last
character is '-', delete it
        this_dash = true;
      }
      else
        this_dash = false;
      if (!first && !prev_dash)
        str = str + " "; // only append " " if this line is the second
one onwards
      first = false;
      str = str + line;
      prev_dash = this_dash;
    }

    char[] temp = str.toCharArray();
    for (i = digits = alphas = vowels = consonants = 0; i < str.length();
i++) { // we can use str[i] as terminating condition as string in C++ is
also terminated with NULL (0)
      temp[i] = Character.toLowerCase(temp[i]); // make each character
lower case
      digits += Character.isDigit(temp[i]) ? 1 : 0;
      alphas += Character.isLetter(temp[i]) ? 1 : 0;
      vowels += isvowel(temp[i]); // already returns 1 or 0
    }
    consonants = alphas - vowels;
    str = new String(temp);
    System.out.println(str);
    System.out.printf("%d %d %d\n", digits, vowels, consonants);
    int hascs3233 = (str.indexOf("cs3233") != -1) ? 1 : 0;

    Vector<String> tokens = new Vector<String>();
    TreeMap<String, Integer> freq = new TreeMap<String, Integer>();
    StringTokenizer st = new StringTokenizer(str, " .");
```

```
    while (st.hasMoreTokens()) {
      String p = st.nextToken();
      tokens.add(p);
      if (!freq.containsKey(p)) freq.put(p, 1);
      else                      freq.put(p, freq.get(p) + 1);
    }

    Collections.sort(tokens);
    System.out.println(tokens.get(0) + " " + tokens.get(tokens.size() -
1));
    System.out.printf("%d\n", hascs3233);

    int ans_s = 0, ans_h = 0, ans_7 = 0;
    String lastline = sc.nextLine();
    for (i = 0; i < lastline.length(); i++) {
      char ch = lastline.charAt(i);
           if (ch == 's') ans_s++;
      else if (ch == 'h') ans_h++;
      else if (ch == '7') ans_7++;
    }
    System.out.printf("%d %d %d\n", ans_s, ans_h, ans_7);
  }
}
```

```
#include <cstdio>
#include <cstring>
#include <time.h>
using namespace std;

#define MAX_N 100010

char T[MAX_N], P[MAX_N]; // T = text, P = pattern
int b[MAX_N], n, m; // b = back table, n = length of T, m = length of P

void naiveMatching() {
  for (int i = 0; i < n; i++) { // try all potential starting indices
    bool found = true;
    for (int j = 0; j < m && found; j++) // use boolean flag `found'
      if (i + j >= n || P[j] != T[i + j]) // if mismatch found
        found = false; // abort this, shift starting index i by +1
    if (found) // if P[0 .. m - 1] == T[i .. i + m - 1]
      printf("P is found at index %d in T\n", i);
} }

void kmpPreprocess() { // call this before calling kmpSearch()
  int i = 0, j = -1; b[0] = -1; // starting values
  while (i < m) { // pre-process the pattern string P
    while (j >= 0 && P[i] != P[j]) j = b[j]; // if different, reset j
using b
    i++; j++; // if same, advance both pointers
    b[i] = j; // observe i = 8, 9, 10, 11, 12 with j = 0, 1, 2, 3, 4
} }           // in the example of P = "SEVENTY SEVEN" above

void kmpSearch() { // this is similar as kmpPreprocess(), but on string T
  int i = 0, j = 0; // starting values
  while (i < n) { // search through string T
    while (j >= 0 && T[i] != P[j]) j = b[j]; // if different, reset j
using b
    i++; j++; // if same, advance both pointers
    if (j == m) { // a match found when j == m
      printf("P is found at index %d in T\n", i - j);
      j = b[j]; // prepare j for the next possible match
} } }

int main() {
  strcpy(T, "I DO NOT LIKE SEVENTY SEV BUT SEVENTY SEVENTY SEVEN");
  strcpy(P, "SEVENTY SEVEN");
  n = (int)strlen(T);
  m = (int)strlen(P);

  //if the end of line character is read too, uncomment the line below
  //T[n-1] = 0; n--; P[m-1] = 0; m--;

  printf("T = '%s'\n", T);
  printf("P = '%s'\n", P);
  printf("\n");

  clock_t t0 = clock();
  printf("Naive Matching\n");
  naiveMatching();
  clock_t t1 = clock();
  printf("Runtime = %.10lf s\n\n", (t1 - t0) / (double) CLOCKS_PER_SEC);

  printf("KMP\n");
```

```
  kmpPreprocess();
  kmpSearch();
  clock_t t2 = clock();
  printf("Runtime = %.10lf s\n\n", (t2 - t1) / (double) CLOCKS_PER_SEC);

  printf("String Library\n");
  char *pos = strstr(T, P);
  while (pos != NULL) {
    printf("P is found at index %d in T\n", pos - T);
    pos = strstr(pos + 1, P);
  }
  clock_t t3 = clock();
  printf("Runtime = %.10lf s\n\n", (t3 - t2) / (double) CLOCKS_PER_SEC);

  return 0;
}

/*
Sample extreme test case.
The first line is T, the second line is P

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAB
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAB

*/

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

int main() {
  char A[20] = "ACAATCC", B[20] = "AGCATGC";
  int n = (int)strlen(A), m = (int)strlen(B);
  int i, j, table[20][20]; // Needleman Wunsnch's algorithm

  memset(table, 0, sizeof table);
  // insert/delete = -1 point
  for (i = 1; i <= n; i++)
    table[i][0] = i * -1;
  for (j = 1; j <= m; j++)
    table[0][j] = j * -1;

  for (i = 1; i <= n; i++)
    for (j = 1; j <= m; j++) {
      // match = 2 points, mismatch = -1 point
      table[i][j] = table[i - 1][j - 1] + (A[i - 1] == B[j - 1] ? 2 : -
1); // cost for match or mismatches
      // insert/delete = -1 point
      table[i][j] = max(table[i][j], table[i - 1][j] - 1); // delete
      table[i][j] = max(table[i][j], table[i][j - 1] - 1); // insert
    }

  printf("DP table:\n");
  for (i = 0; i <= n; i++) {
    for (j = 0; j <= m; j++)
      printf("%3d", table[i][j]);
    printf("\n");
  }
  printf("Maximum Alignment Score: %d\n", table[n][m]);

  return 0;
}
```

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

typedef pair<int, int> ii;

#define MAX_N 100010                          // second approach: O(n log
n)
char T[MAX_N];                    // the input string, up to 100K
characters
int n;                                         // the length of input
string
int RA[MAX_N], tempRA[MAX_N];         // rank array and temporary rank
array
int SA[MAX_N], tempSA[MAX_N];     // suffix array and temporary suffix
array
int c[MAX_N];                                     // for counting/radix
sort

char P[MAX_N];                   // the pattern string (for string
matching)
int m;                                        // the length of pattern
string

int Phi[MAX_N];                       // for computing longest common
prefix
int PLCP[MAX_N];
int LCP[MAX_N];  // LCP[i] stores the LCP between previous suffix T+SA[i-
1]
                                              // and current suffix
T+SA[i]

bool cmp(int a, int b) { return strcmp(T + a, T + b) < 0; }      //
compare

void constructSA_slow() {               // cannot go beyond 1000
characters
  for (int i = 0; i < n; i++) SA[i] = i; // initial SA: {0, 1, 2, ..., n-
1}
  sort(SA, SA + n, cmp); // sort: O(n log n) * compare: O(n) = O(n^2 log
n)
}

void countingSort(int k) {                                              //
O(n)
  int i, sum, maxi = max(300, n);   // up to 255 ASCII chars or length of
n
  memset(c, 0, sizeof c);                          // clear frequency
table
  for (i = 0; i < n; i++)       // count the frequency of each integer
rank
    c[i + k < n ? RA[i + k] : 0]++;
  for (i = sum = 0; i < maxi; i++) {
    int t = c[i]; c[i] = sum; sum += t;
  }
  for (i = 0; i < n; i++)          // shuffle the suffix array if
necessary
    tempSA[c[SA[i]+k < n ? RA[SA[i]+k] : 0]++] = SA[i];
```

```
  for (i = 0; i < n; i++)                       // update the suffix array
SA
    SA[i] = tempSA[i];
}

void constructSA() {         // this version can go up to 100000
characters
  int i, k, r;
  for (i = 0; i < n; i++) RA[i] = T[i];                 // initial
rankings
  for (i = 0; i < n; i++) SA[i] = i;     // initial SA: {0, 1, 2, ..., n-
1}
  for (k = 1; k < n; k <<= 1) {       // repeat sorting process log n
times
    countingSort(k);  // actually radix sort: sort based on the second
item
    countingSort(0);          // then (stable) sort based on the first
item
    tempRA[SA[0]] = r = 0;             // re-ranking; start from rank r =
0
    for (i = 1; i < n; i++)                     // compare adjacent
suffixes
      tempRA[SA[i]] = // if same pair => same rank r; otherwise, increase
r
       (RA[SA[i]] == RA[SA[i-1]] && RA[SA[i]+k] == RA[SA[i-1]+k]) ? r :
++r;
    for (i = 0; i < n; i++)                      // update the rank array
RA
      RA[i] = tempRA[i];
    if (RA[SA[n-1]] == n-1) break;               // nice optimization
trick
} }

void computeLCP_slow() {
  LCP[0] = 0;                                           // default
value
  for (int i = 1; i < n; i++) {                // compute LCP by
definition
    int L = 0;                                       // always reset L to
0
    while (T[SA[i] + L] == T[SA[i-1] + L]) L++;      // same L-th char,
L++
    LCP[i] = L;
} }

void computeLCP() {
  int i, L;
  Phi[SA[0]] = -1;                                      // default
value
  for (i = 1; i < n; i++)                           // compute Phi in
O(n)
    Phi[SA[i]] = SA[i-1];    // remember which suffix is behind this
suffix
  for (i = L = 0; i < n; i++) {             // compute Permuted LCP in
O(n)
    if (Phi[i] == -1) { PLCP[i] = 0; continue; }            // special
case
    while (T[i + L] == T[Phi[i] + L]) L++;       // L increased max n
times
    PLCP[i] = L;
```

```
    L = max(L-1, 0);                                // L decreased max n
times
  }
  for (i = 0; i < n; i++)                              // compute LCP in
O(n)
    LCP[i] = PLCP[SA[i]];   // put the permuted LCP to the correct
position
}

ii stringMatching() {                         // string matching in O(m log
n)
  int lo = 0, hi = n-1, mid = lo;               // valid matching = [0..n-
1]
  while (lo < hi) {                                    // find lower
bound
    mid = (lo + hi) / 2;                               // this is round
down
    int res = strncmp(T + SA[mid], P, m);  // try to find P in suffix
'mid'
    if (res >= 0) hi = mid;         // prune upper half (notice the >=
sign)
    else          lo = mid + 1;            // prune lower half including
mid
  }                                       // observe `=' in "res >= 0"
above
  if (strncmp(T + SA[lo], P, m) != 0) return ii(-1, -1);    // if not
found
  ii ans; ans.first = lo;
  lo = 0; hi = n - 1; mid = lo;
  while (lo < hi) {             // if lower bound is found, find upper
bound
    mid = (lo + hi) / 2;
    int res = strncmp(T + SA[mid], P, m);
    if (res > 0) hi = mid;                             // prune upper
half
    else         lo = mid + 1;             // prune lower half including
mid
  }                            // (notice the selected branch when res ==
0)
  if (strncmp(T + SA[hi], P, m) != 0) hi--;                 // special
case
  ans.second = hi;
  return ans;
} // return lower/upperbound as first/second item of the pair,
respectively

ii LRS() {                 // returns a pair (the LRS length and its
index)
  int i, idx = 0, maxLCP = -1;
  for (i = 1; i < n; i++)                          // O(n), start from i =
1
    if (LCP[i] > maxLCP)
      maxLCP = LCP[i], idx = i;
  return ii(maxLCP, idx);
}

int owner(int idx) { return (idx < n-m-1) ? 1 : 2; }

ii LCS() {                 // returns a pair (the LCS length and its
index)
```

```
  int i, idx = 0, maxLCP = -1;
  for (i = 1; i < n; i++)                          // O(n), start from i =
1
    if (owner(SA[i]) != owner(SA[i-1]) && LCP[i] > maxLCP)
      maxLCP = LCP[i], idx = i;
  return ii(maxLCP, idx);
}

int main() {
  //printf("Enter a string T below, we will compute its Suffix
Array:\n");
  strcpy(T, "GATAGACA");
  n = (int)strlen(T);
  T[n++] = '$';
  // if '\n' is read, uncomment the next line
  //T[n-1] = '$'; T[n] = 0;

  constructSA_slow();                                          // O(n^2 log
n)
  printf("The Suffix Array of string T = '%s' is shown below (O(n^2 log
n) version):\n", T);
  printf("i\tSA[i]\tSuffix\n");
  for (int i = 0; i < n; i++) printf("%2d\t%2d\t%s\n", i, SA[i], T +
SA[i]);

  constructSA();                                               // O(n log
n)
  printf("\nThe Suffix Array of string T = '%s' is shown below (O(n log
n) version):\n", T);
  printf("i\tSA[i]\tSuffix\n");
  for (int i = 0; i < n; i++) printf("%2d\t%2d\t%s\n", i, SA[i], T +
SA[i]);

  computeLCP();                                                //
O(n)

  // LRS demo
  ii ans = LRS();                 // find the LRS of the first input
string
  char lrsans[MAX_N];
  strncpy(lrsans, T + SA[ans.second], ans.first);
  printf("\nThe LRS is '%s' with length = %d\n\n", lrsans, ans.first);

  // stringMatching demo
  //printf("\nNow, enter a string P below, we will try to find P in
T:\n");
  strcpy(P, "A");
  m = (int)strlen(P);
  // if '\n' is read, uncomment the next line
  //P[m-1] = 0; m--;
  ii pos = stringMatching();
  if (pos.first != -1 && pos.second != -1) {
    printf("%s is found SA[%d..%d] of %s\n", P, pos.first, pos.second,
T);
    printf("They are:\n");
    for (int i = pos.first; i <= pos.second; i++)
      printf("  %s\n", T + SA[i]);
  } else printf("%s is not found in %s\n", P, T);

  // LCS demo
```

```
  //printf("\nRemember, T = '%s'\nNow, enter another string P:\n", T);
  // T already has '$' at the back
  strcpy(P, "CATA");
  m = (int)strlen(P);
  // if '\n' is read, uncomment the next line
  //P[m-1] = 0; m--;
  strcat(T, P);                                                  // append
P
  strcat(T, "#");                                  // add '$' at the
back
  n = (int)strlen(T);                                            // update
n

  // reconstruct SA of the combined strings
  constructSA();                                               // O(n log
n)
  computeLCP();                                                    //
O(n)
  printf("\nThe LCP information of 'T+P' = '%s':\n", T);
  printf("i\tSA[i]\tLCP[i]\tOwner\tSuffix\n");
  for (int i = 0; i < n; i++)
    printf("%2d\t%2d\t%2d\t%2d\t%s\n", i, SA[i], LCP[i], owner(SA[i]), T
+ SA[i]);

  ans = LCS();         // find the longest common substring between T and
P
  char lcsans[MAX_N];
  strncpy(lcsans, T + SA[ans.second], ans.first);
  printf("\nThe LCS is '%s' with length = %d\n", lcsans, ans.first);

  return 0;
}
```

```
#include <algorithm>
#include <cstdio>
#include <cmath>
#include <vector>
using namespace std;

#define INF 1e9
#define EPS 1e-9
#define PI acos(-1.0) // important constant; alternative #define PI (2.0
* acos(0.0))

double DEG_to_RAD(double d) { return d * PI / 180.0; }

double RAD_to_DEG(double r) { return r * 180.0 / PI; }

// struct point_i { int x, y; };    // basic raw form, minimalist mode
struct point_i { int x, y;     // whenever possible, work with point_i
  point_i() { x = y = 0; }                      // default constructor
  point_i(int _x, int _y) : x(_x), y(_y) {} };         // user-defined

struct point { double x, y;   // only used if more precision is needed
  point() { x = y = 0.0; }                      // default constructor
  point(double _x, double _y) : x(_x), y(_y) {}        // user-defined
  bool operator < (point other) const { // override less than operator
    if (fabs(x - other.x) > EPS)                 // useful for sorting
      return x < other.x;          // first criteria , by x-coordinate
    return y < other.y; }          // second criteria, by y-coordinate
  // use EPS (1e-9) when testing equality of two floating points
  bool operator == (point other) const {
   return (fabs(x - other.x) < EPS && (fabs(y - other.y) < EPS)); } };

double dist(point p1, point p2) {                // Euclidean distance
                    // hypot(dx, dy) returns sqrt(dx * dx + dy * dy)
  return hypot(p1.x - p2.x, p1.y - p2.y); }           // return double

// rotate p by theta degrees CCW w.r.t origin (0, 0)
point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);    // multiply theta with PI / 180.0
  return point(p.x * cos(rad) - p.y * sin(rad),
               p.x * sin(rad) + p.y * cos(rad)); }

struct line { double a, b, c; };          // a way to represent a line

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line &l) {
  if (fabs(p1.x - p2.x) < EPS) {              // vertical line is fine
    l.a = 1.0;   l.b = 0.0;   l.c = -p1.x;           // default values
  } else {
    l.a = -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l.b = 1.0;              // IMPORTANT: we fix the value of b to 1.0
    l.c = -(double)(l.a * p1.x) - p1.y;
} }

// not needed since we will use the more robust form: ax + by + c = 0
(see above)
struct line2 { double m, c; };      // another way to represent a line

int pointsToLine2(point p1, point p2, line2 &l) {
 if (abs(p1.x - p2.x) < EPS) {          // special case: vertical line
   l.m = INF;                    // l contains m = INF and c = x_value
```

```
    l.c = p1.x;                  // to denote vertical line x = x_value
    return 0;   // we need this return variable to differentiate result
  }
  else {
    l.m = (double)(p1.y - p2.y) / (p1.x - p2.x);
    l.c = p1.y - l.m * p1.x;
    return 1;     // l contains m and c of the line equation y = mx + c
} }

bool areParallel(line l1, line l2) {       // check coefficients a & b
  return (fabs(l1.a-l2.a) < EPS) && (fabs(l1.b-l2.b) < EPS); }

bool areSame(line l1, line l2) {           // also check coefficient c
  return areParallel(l1 ,l2) && (fabs(l1.c - l2.c) < EPS); }

// returns true (+ intersection point) if two lines are intersect
bool areIntersect(line l1, line l2, point &p) {
  if (areParallel(l1, l2)) return false;            // no intersection
  // solve system of 2 linear algebraic equations with 2 unknowns
  p.x = (l2.b * l1.c - l1.b * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  // special case: test for vertical line to avoid division by zero
  if (fabs(l1.b) > EPS) p.y = -(l1.a * p.x + l1.c);
  else                  p.y = -(l2.a * p.x + l2.c);
  return true; }

struct vec { double x, y;  // name: `vec' is different from STL vector
  vec(double _x, double _y) : x(_x), y(_y) {} };

vec toVec(point a, point b) {       // convert 2 points to vector a->b
  return vec(b.x - a.x, b.y - a.y); }

vec scale(vec v, double s) {        // nonnegative s = [<1 .. 1 .. >1]
  return vec(v.x * s, v.y * s); }               // shorter.same.longer

point translate(point p, vec v) {        // translate p according to v
  return point(p.x + v.x , p.y + v.y); }

// convert point and gradient/slope to line
void pointSlopeToLine(point p, double m, line &l) {
  l.a = -m;                                                 // always -m
  l.b = 1;                                                   // always 1
  l.c = -((l.a * p.x) + (l.b * p.y)); }                // compute this

void closestPoint(line l, point p, point &ans) {
  line perpendicular;         // perpendicular to l and pass through p
  if (fabs(l.b) < EPS) {              // special case 1: vertical line
    ans.x = -(l.c);   ans.y = p.y;      return; }

  if (fabs(l.a) < EPS) {            // special case 2: horizontal line
    ans.x = p.x;      ans.y = -(l.c);   return; }

  pointSlopeToLine(p, 1 / l.a, perpendicular);          // normal line
  // intersect line l with this perpendicular line
  // the intersection point is the closest point
  areIntersect(l, perpendicular, ans); }

// returns the reflection of point on a line
void reflectionPoint(line l, point p, point &ans) {
  point b;
  closestPoint(l, p, b);                    // similar to distToLine
```

```
  vec v = toVec(p, b);                              // create a vector
  ans = translate(translate(p, v), v); }         // translate p twice

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

double norm_sq(vec v) { return v.x * v.x + v.y * v.y; }

// returns the distance from p to the line defined by
// two points a and b (a and b must be different)
// the closest point is stored in the 4th parameter (byref)
double distToLine(point p, point a, point b, point &c) {
  // formula: c = a + u * ab
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  c = translate(a, scale(ab, u));                   // translate a to c
  return dist(p, c); }           // Euclidean distance between p and c

// returns the distance from p to the line segment ab defined by
// two points a and b (still OK if a == b)
// the closest point is stored in the 4th parameter (byref)
double distToLineSegment(point p, point a, point b, point &c) {
  vec ap = toVec(a, p), ab = toVec(a, b);
  double u = dot(ap, ab) / norm_sq(ab);
  if (u < 0.0) { c = point(a.x, a.y);                    // closer to a
    return dist(p, a); }         // Euclidean distance between p and a
  if (u > 1.0) { c = point(b.x, b.y);                    // closer to b
    return dist(p, b); }         // Euclidean distance between p and b
  return distToLine(p, a, b, c); }          // run distToLine as above

double angle(point a, point o, point b) {  // returns angle aob in rad
  vec oa = toVec(o, a), ob = toVec(o, b);
  return acos(dot(oa, ob) / sqrt(norm_sq(oa) * norm_sq(ob))); }

double cross(vec a, vec b) { return a.x * b.y - a.y * b.x; }

//// another variant
//int area2(point p, point q, point r) { // returns 'twice' the area of
this triangle A-B-c
//  return p.x * q.y - p.y * q.x +
//         q.x * r.y - q.y * r.x +
//         r.x * p.y - r.y * p.x;
//}

// note: to accept collinear points, we have to change the `> 0'
// returns true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(toVec(p, q), toVec(p, r)) > 0; }

// returns true if point r is on the same line as the line pq
bool collinear(point p, point q, point r) {
  return fabs(cross(toVec(p, q), toVec(p, r))) < EPS; }

int main() {
  point P1, P2, P3(0, 1); // note that both P1 and P2 are (0.00, 0.00)
  printf("%d\n", P1 == P2);                                  // true
  printf("%d\n", P1 == P3);                                 // false

  vector<point> P;
  P.push_back(point(2, 2));
  P.push_back(point(4, 3));
```

```
  P.push_back(point(2, 4));
  P.push_back(point(6, 6));
  P.push_back(point(2, 6));
  P.push_back(point(6, 5));

  // sorting points demo
  sort(P.begin(), P.end());
  for (int i = 0; i < (int)P.size(); i++)
    printf("(%.2lf, %.2lf)\n", P[i].x, P[i].y);

  // rearrange the points as shown in the diagram below
  P.clear();
  P.push_back(point(2, 2));
  P.push_back(point(4, 3));
  P.push_back(point(2, 4));
  P.push_back(point(6, 6));
  P.push_back(point(2, 6));
  P.push_back(point(6, 5));
  P.push_back(point(8, 6));

  /*
  // the positions of these 7 points (0-based indexing)
  6   P4      P3  P6
  5           P5
  4   P2
  3       P1
  2   P0
  1
  0 1 2 3 4 5 6 7 8
  */

  double d = dist(P[0], P[5]);
  printf("Euclidean distance between P[0] and P[5] = %.2lf\n", d); //
should be 5.000

  // line equations
  line l1, l2, l3, l4;
  pointsToLine(P[0], P[1], l1);
  printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l1.a, l1.b, l1.c); //
should be -0.50 * x + 1.00 * y - 1.00 = 0.00

  pointsToLine(P[0], P[2], l2); // a vertical line, not a problem in "ax
+ by + c = 0" representation
  printf("%.2lf * x + %.2lf * y + %.2lf = 0.00\n", l2.a, l2.b, l2.c); //
should be 1.00 * x + 0.00 * y - 2.00 = 0.00

  // parallel, same, and line intersection tests
  pointsToLine(P[2], P[3], l3);
  printf("l1 & l2 are parallel? %d\n", areParallel(l1, l2)); // no
  printf("l1 & l3 are parallel? %d\n", areParallel(l1, l3)); // yes, l1
(P[0]-P[1]) and l3 (P[2]-P[3]) are parallel

  pointsToLine(P[2], P[4], l4);
  printf("l1 & l2 are the same? %d\n", areSame(l1, l2)); // no
  printf("l2 & l4 are the same? %d\n", areSame(l2, l4)); // yes, l2
(P[0]-P[2]) and l4 (P[2]-P[4]) are the same line (note, they are two
different line segments, but same line)

  point p12;
```

```
  bool res = areIntersect(l1, l2, p12); // yes, l1 (P[0]-P[1]) and l2
(P[0]-P[2]) are intersect at (2.0, 2.0)
  printf("l1 & l2 are intersect? %d, at (%.2lf, %.2lf)\n", res, p12.x,
p12.y);

  // other distances
  point ans;
  d = distToLine(P[0], P[2], P[3], ans);
  printf("Closest point from P[0] to line         (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, d);
  closestPoint(l3, P[0], ans);
  printf("Closest point from P[0] to line V2      (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, dist(P[0], ans));

  d = distToLineSegment(P[0], P[2], P[3], ans);
  printf("Closest point from P[0] to line SEGMENT (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to A (or P[2]) =
(2.00, 4.00)
  d = distToLineSegment(P[1], P[2], P[3], ans);
  printf("Closest point from P[1] to line SEGMENT (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to midway between AB
= (3.20, 4.60)
  d = distToLineSegment(P[6], P[2], P[3], ans);
  printf("Closest point from P[6] to line SEGMENT (P[2]-P[3]): (%.2lf,
%.2lf), dist = %.2lf\n", ans.x, ans.y, d); // closer to B (or P[3]) =
(6.00, 6.00)

  reflectionPoint(l4, P[1], ans);
  printf("Reflection point from P[1] to line      (P[2]-P[4]): (%.2lf,
%.2lf)\n", ans.x, ans.y); // should be (0.00, 3.00)

  printf("Angle P[0]-P[4]-P[3] = %.2lf\n", RAD_to_DEG(angle(P[0], P[4],
P[3]))); // 90 degrees
  printf("Angle P[0]-P[2]-P[1] = %.2lf\n", RAD_to_DEG(angle(P[0], P[2],
P[1]))); // 63.43 degrees
  printf("Angle P[4]-P[3]-P[6] = %.2lf\n", RAD_to_DEG(angle(P[4], P[3],
P[6]))); // 180 degrees

  printf("P[0], P[2], P[3] form A left turn? %d\n", ccw(P[0], P[2],
P[3])); // no
  printf("P[0], P[3], P[2] form A left turn? %d\n", ccw(P[0], P[3],
P[2])); // yes

  printf("P[0], P[2], P[3] are collinear? %d\n", collinear(P[0], P[2],
P[3])); // no
  printf("P[0], P[2], P[4] are collinear? %d\n", collinear(P[0], P[2],
P[4])); // yes

  point p(3, 7), q(11, 13), r(35, 30); // collinear if r(35, 31)
  printf("r is on the %s of line p-r\n", ccw(p, q, r) ? "left" :
"right"); // right

  /*
  // the positions of these 6 points
     E<--  4
           3       B D<--
           2   A C
           1
  -4-3-2-1 0 1 2 3 4 5 6
          -1
```

```
          -2
   F<--   -3
  */

  // translation
  point A(2.0, 2.0);
  point B(4.0, 3.0);
  vec v = toVec(A, B); // imagine there is an arrow from A to B (see the
diagram above)
  point C(3.0, 2.0);
  point D = translate(C, v); // D will be located in coordinate (3.0 +
2.0, 2.0 + 1.0) = (5.0, 3.0)
  printf("D = (%.2lf, %.2lf)\n", D.x, D.y);
  point E = translate(C, scale(v, 0.5)); // E will be located in
coordinate (3.0 + 1/2 * 2.0, 2.0 + 1/2 * 1.0) = (4.0, 2.5)
  printf("E = (%.2lf, %.2lf)\n", E.x, E.y);

  // rotation
  printf("B = (%.2lf, %.2lf)\n", B.x, B.y); // B = (4.0, 3.0)
  point F = rotate(B, 90); // rotate B by 90 degrees COUNTER clockwise, F
= (-3.0, 4.0)
  printf("F = (%.2lf, %.2lf)\n", F.x, F.y);
  point G = rotate(B, 180); // rotate B by 180 degrees COUNTER clockwise,
G = (-4.0, -3.0)
  printf("G = (%.2lf, %.2lf)\n", G.x, G.y);

  return 0;
}
```