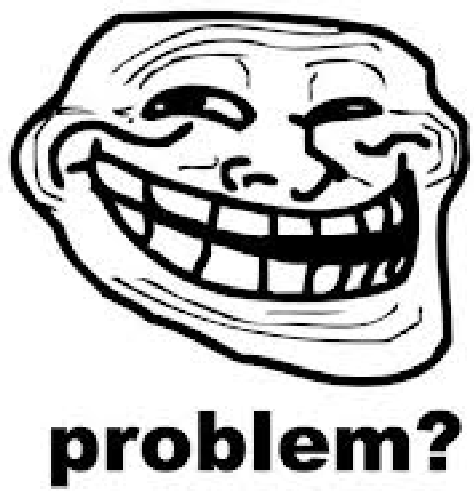
jhtan Teambook

Jhonatan I. Castro Rocabado

August 30, 2014



Contents

Chapter 1

Number Theory

1.1 Prime sieves

1.1.1 Sieve of Eratosthenes

```
#include <cstdio>
#include <iostream>
#include <bitset>
#include <vector>

using namespace std;

typedef long long          ll;
typedef vector<int>        vi;

ll _sieve_size;
bitset<10000010> bs;
vi primes;

// Sieve of Eratosthenes.
void sieve(ll upperbound) {
  _sieve_size = upperbound + 1;
  bs.set();
  bs[0] = bs[1] = 0;

  for(ll i=2; i<=_sieve_size; i++) {
    if(bs[i]) { // This is a prime! :D
      for(ll j = i*i; j <= _sieve_size; j += i)
  bs[j] = 0; // Mark the composite numbers

      primes.push_back((int)i); // Save the prime number in a vector.
    }
  }
}

// Test if a number is prime.
bool isPrime(ll n) {
  if(n <= _sieve_size)
    return bs[n];

  // Checks if the number is prime, when this exceeds the _sieve_size value.
  for(int i=0; i<(int)primes.size(); i++) {
    if(n % primes[i] == 0)
      return false;
  }

  return true;
}
```

```
int main() {

  sieve(10000000);

  cout << isPrime(2147483647) << endl; // 10-digits prime
  cout << isPrime(136117223861LL) << endl; // not a prime, 104729*1299709
}
```

Chapter 2

Data Structures

import karma

jhtan

2.1 Segment tree

```
int tree[400000];
int v[100000];

void init(int node, int a, int b) {
  if (a == b) {
    tree[node] = v[a];
    return;
  }
  init(2*node+1, a, (a + b)/2);
  init(2*node+2, (a+b)/2+1, b);
  tree[node] = tree[2*node+1] + tree[2*node+2];
}

int query(int node, int a, int b, int p, int q) {
  if (q < a || b < p) return 0; // return 0 for sum, 1 for product
  if (p <= a && b <= q) return tree[node];
  return query(2*node+1, a, (a+b)/2, p, q) + query(2*node+2, (a+b)/2+1, b, p, q);
}

void update(int node, int a, int b, int p, int val) {
  if (p < a || b < p) return;
  if (a == b) {
    tree[node] = val;
    return;
  }
  update(2*node+1, a, (a+b)/2, p, val);
  update(2*node+2, (a+b)/2+1, b, p, val);
  tree[node] = tree[2*node+1] + tree[2*node+2];
}

int tree[400000];
int v[100000];

void init(int node, int a, int b) {
  if (a == b) {
    tree[node] = v[a];
    return;
  }
  init(2*node+1, a, (a + b)/2);
  init(2*node+2, (a+b)/2+1, b);
```

```
  tree[node] = tree[2*node+1] + tree[2*node+2];
}

int query(int node, int a, int b, int p, int q) {
  if (q < a || b < p) return 0; // return 0 for sum, 1 for product
  if (p <= a && b <= q) return tree[node];
  return query(2*node+1, a, (a+b)/2, p, q) + query(2*node+2, (a+b)/2+1, b, p, q);
}

void update(int node, int a, int b, int p, int val) {
  if (p < a || b < p) return;
  if (a == b) {
    tree[node] = val;
    return;
  }
  update(2*node+1, a, (a+b)/2, p, val);
  update(2*node+2, (a+b)/2+1, b, p, val);
  tree[node] = tree[2*node+1] + tree[2*node+2];
}
```

2.2 Trie

```
#include <iostream>
#include <cstdio>

using namespace std;

struct trie{
  int words;
  int prefixes;
  struct trie *edges[26];
};

void init(trie *vertex) {
  vertex->words = 0;
  vertex->prefixes = 0;
  for(int i=0; i<26; i++) {
    vertex->edges[i] = NULL;
  }
}

void addWord(trie *vertex, string word) {
  if(word.length() == 0) {
    vertex->words++;
  } else {
    vertex->prefixes++;
    int k = word[0] - 'a';
    if(!vertex->edges[k]) {
      vertex->edges[k] = new trie();
      init(vertex->edges[k]);
    }
    addWord(vertex->edges[k], word.substr(1));
  }
}

int countWords(trie vertex, string word) {
  int k = word[0] - 'a';
  if(word.length() == 0) {
    return vertex.words;
  } else if(!vertex.edges[k]) {
    return 0;
  } else {
    return countWords(*vertex.edges[k], word.substr(1));
  }
}

int countPrefixes(trie vertex, string prefix) {
```

```
  int k = prefix[0] - 'a';
  if(prefix.length() == 0) {
    return vertex.prefixes;
  } else if(!vertex.edges[k]) {
    return 0;
  } else {
    return countPrefixes(*vertex.edges[k], prefix.substr(1));
  }
}

int main() {
  trie index;
  init(&index);

  int n;
  scanf("%d", &n);

  string s;
  for(int i=0; i<n; i++) {
    cin >> s;
    addWord(&index, s);
  }

  cout << "There are " << countWords(index, "lol") << " lol words." << endl;
  cout << "There are " << countPrefixes(index, "lol") << " lol prefixes." << endl;

  return 0;
}
```

2.3 Union-Find-Disjoint set

```
#include <cstdio>

using namespace std;

#define MAX 1000000

int p[MAX], num_sets;
void initSet(int n) {
  for (int i = 0; i < n ; i++) p[i] = i;
  num_sets = n;
}
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)) num_sets--;
  p[findSet(i)] = findSet(j);
}

int main() {
  int n, m;
  scanf("%d %d", &n, &m);

  initSet(n);

  int a, b;
  for(int i=0; i<m; i++) {
    scanf("%d %d", &a, &b);
    unionSet(a-1, b-1);
  }

  int q, x;
  scanf("%d", &q);
```

```
  for(int i=0; i<q; i++) {
    scanf("%d %d", &a, &b);
    if(isSameSet(a-1, b-1))
      printf("%d and %d are in the same set.\n", a, b);
    else
      printf("%d and %d are not in the same set.\n", a, b);
  }

  printf("There are %d sets.\n", num_sets);

  return 0;
}
```

Chapter 3

Graphs

3.1 Depth First Search (DFS)

```
#define VISITED 1
#define NOT_VISITED 0
int n, e; // number of nodes and edges
vector<vi> graph; // adjacency list of the graph
int dfsm[MAX]; // max number of vertices in the graph

void dfs(int start) {
  dfsm[start] = VISITED;
  DBG(start);
  for (int i = 0; i < graph[start].size(); i++) {
    if(dfsm[graph[start][i]] == NOT_VISITED) {
      dfs(graph[start][i]);
    }
  }
}

int main() {
  scanf("%d %d", &n, &e);
  graph = vector<vi>(n);
  int ns, nt;
  while(e--) {
    scanf("%d %d", &ns, &nt);
    graph[ns].push_back(nt);
  }
  memset(dfsm, NOT_VISITED, sizeof dfsm);
  dfs(0);
  return 0;
}
```

3.1.1 Finding Connected Components in Undirect Graph

```
int n, e;
vector<vi> graph;
int dfsm[MAX];

void dfs(int start) {
  dfsm[start] = VISITED;
  cout << start << " ";
  for (int i = 0; i < graph[start].size(); i++) {
    if (dfsm[graph[start][i]] == NOT_VISITED) {
      dfs(graph[start][i]);
    }
  }
}

int main() {
```

```
  memset(dfsm, NOT_VISITED, sizeof dfsm);
  int numCC = 0;
  for (int i = 0; i < n; i++) {
    if (dfsm[i] == NOT_VISITED) {
      printf("Component %d:", ++numCC);
      dfs(i);
      printf("\n");
    }
  }
  return 0;
}
```

3.1.2 Flood Fill

```
#include <cstring>
#include <cstdio>

using namespace std;

int n;
int M[1001][1001];
bool B[1001][1001];

void ff(int i, int j) {
  B[i][j] = true;
  int X[8] = {0, 1, 1, 1, 0, -1, -1, -1};
  int Y[8] = {-1, -1, 0, 1, 1, 1, 0, -1};

  for(int k=0; k<8; k++) {
    int a = X[k] + i;
    int b = Y[k] + j;
    if(a >= 0 && b >= 0 && a < n && b < n) {
      if(!B[a][b] && M[i][j])
        ff(a, b);
    }
  }
}

int main() {
  scanf("%d", &n);

  for(int i=0; i<n; i++)
    for(int j=0; j<n; j++)
      scanf("%d", &M[i][j]);

  memset(B, false, sizeof(B));
  int c = 0;
  for(int i=0; i<n; i++) {
    for(int j=0; j<n; j++) {
      if(!B[i][j] && M[i][j]) {
        ff(i, j);
        c++;
      }
    }
  }

  printf("%d\n", c);
  return 0;
}
```

3.1.3 Finding Articulation Points and Bridges [Hopcroft and Tarjan]

```
int dfsn[MAX];
int dfsl[MAX];
int dfsp[MAX];
int aVertex[MAX];
int dfsNumberCounter = 0;
```

```
int dfsRoot;
int rootChildren;

void articulationPointAndBridges(int u) {
  dfsl[u] = dfsn[u] = dfsNumberCounter++;
  for (int j = 0; j < graph[u].size(); j++) {
    if (dfsn[graph[u][j]] == NOT_VISITED) {
      dfsp[graph[u][j]] = u;
      if (u == dfsRoot) rootChildren++;
      articulationPointAndBridges(graph[u][j]);
      if (dfsl[graph[u][j]] >= dfsn[u])
        aVertex[u] = 1;
      if (dfsl[graph[u][j]] > dfsn[u])
        DBG(graph[u][j] << " " << u); // u and graph[u][j] are a bridge
      dfsl[u] = min(dfsl[u], dfsl[graph[u][j]]);
    } else if(graph[u][j] != dfsp[u]) {
      dfsl[u] = min(dfsl[u], dfsn[graph[u][j]]);
    }
  }
}

int main() {
  memset(dfsn, 0, sizeof dfsn);
  memset(dfsl, 0, sizeof dfsl);
  memset(dfsp, 0, sizeof dfsp);
  memset(aVertex, 0, sizeof aVertex);
  dfsNumberCounter = 0;
  for (int i = 0; i < n; i++) {
    if (dfsn[i] == NOT_VISITED) {
      dfsRoot = i;
      rootChildren = 0;
      articulationPointAndBridges(i);
      aVertex[dfsRoot] = (rootChildren > 1);
    }
  }
  for (int i = 0; i < n; i++) {
    if (aVertex[i])
      DBG(i); // i is a articulation point
  }
  return 0;
}

#include <cstdio>
#include <vector>

using namespace std;

typedef pair<int, int> ii;
typedef vector<ii> vii;
typedef vector<int> vi;
#define DFS_WHITE -1
#define DFS_BLACK 1

vector<vii> G;
vi dfs_num;
vi dfs_parent;
vi dfs_low;
vi articulation_vertex;
int dfsNumberCounter, dfsRoot, rootChildren;

void articulationPointAndBridge(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;
  for(int j=0; j<G[u].size(); j++) {
    ii v = G[u][j];
    if(dfs_num[v.first] == DFS_WHITE) {
      dfs_parent[v.first] = u;
      if(u == dfsRoot)
  rootChildren++;
```

```
      articulationPointAndBridge(v.first);

      if(dfs_low[v.first] >= dfs_num[u])
  articulation_vertex[u] = true;
      if(dfs_low[v.first] > dfs_num[u])
  printf(" Edge (%d, %d) is a bridge\n", u, v.first);
      dfs_low[u] = min(dfs_low[u], dfs_low[v.first]);
    } else if(v.first != dfs_parent[u])
      dfs_low[u] = min(dfs_low[u], dfs_num[v.first]);
  }
}

int main() {
  int v, e;
  scanf("%d %d", &v, &e);

  G.assign(v, vii());
  int a, b;
  for(int i=0; i<e; i++) {
    scanf("%d %d", &a, &b);
    G[a].push_back(ii(b, 1));
    G[b].push_back(ii(a, 1));
  }

  dfsNumberCounter = 0;
  dfs_num.assign(v, DFS_WHITE);
  dfs_low.assign(v, 0);
  dfs_parent.assign(v, 0);
  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);
    }
  }

  printf("Articulation Points:\n");
  for(int i=0; i<v; i++) {
    if(articulation_vertex[i])
      printf(" Vertex %d\n", i);
  }

  return 0;
}
```

3.1.4 Finding Strongly Connected Components in Directed Graph [Tarjan]

```
vi dfs_num, dfs_low, S,visited;
int dfsNumberCounter, numSCC;

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;
  S.push_back(u);
  visited[u] = 1;
  for (int j = 0; j < g[u].size(); j++) {
    int v = g[u][j];
    if (dfs_num[v] == DFS_WHITE) tarjanSCC(v);
    if (visited[v]) dfs_low[u] = min(dfs_low[u], dfs_low[v]);
  }
  if (dfs_low[u] == dfs_num[u]) {
    cout << "SCC " << ++numSCC << ":";
    while (1) {
      int v = S.back();
      S.pop_back();
```

```
      visited[v] = 0;
      cout << " " << v;
      if (u == v) break;
    }
    cout << endl;
  }
}

int main() {
  /* Build graph */
  dfs_num.assign(n, DFS_WHITE);
  dfs_low.assign(n,0);
  visited.assign(n, 0);
  dfsNumberCounter = numSCC = 0;
  for (int i = 0; i < n; i++)
    if (dfs_num[i] == DFS_WHITE)
      tarjanSCC(i);
  return 0;
}
```

```
#include <cstdio>
#include <vector>

using namespace std;

typedef long long ll;
typedef pair<int, int> ii;
typedef vector<int> vi;
typedef vector<ii> vii;
#define UNVISITED -1
#define VISITED 1

vector<vi> G;
vi S, visited, dfs_num, dfs_low;
int dfsNumberCounter, numSCC;

void tarjanSCC(int u) {
  dfs_low[u] = dfs_num[u] = dfsNumberCounter++;
  S.push_back(u);
  visited[u] = 1;

  for(int j=0; j<G[u].size(); j++) {
    int v = G[u][j];
    if(dfs_num[v] == UNVISITED)
      tarjanSCC(v);
    if(visited[v])
      dfs_low[u] = min(dfs_low[u], dfs_low[v]);
  }

  if(dfs_low[u] == dfs_num[u]) {
    numSCC++;
    printf("SCC %d:", numSCC);

    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, e;
  scanf("%d %d", &v, &e);
```

```
  G.assign(v, vi());

  int a, b;
  for(int i=0; i<e; i++) {
    scanf("%d %d", &a, &b);
    G[a].push_back(b);
  }

  dfs_num.assign(v, UNVISITED);
  dfs_low.assign(v, 0);
  visited.assign(v, 0);
  dfsNumberCounter = numSCC = 0;

  for(int i=0; i<v; i++) {
    if(dfs_num[i] == UNVISITED)
      tarjanSCC(i);
  }

  return 0;
}
```

3.2 Breadth First Search (BFS)

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

typedef vector<int> vi;
#define pb push_back

vector<vi> G;
int dist[10010];
int parent[10010];

void bfs(int n) {
  queue<int> q;
  q.push(n);
  memset(dist, -1, sizeof(dist));
  memset(dist, -1, sizeof(parent));
  dist[n] = 0;

  while(!q.empty()) {
    int u = q.front();
    q.pop();

    for(int i=0; i<G[u].size(); i++) {
      if(dist[G[u][i]] == -1) {
        dist[G[u][i]] = dist[u] + 1;
        parent[G[u][i]] = u;
        q.push(G[u][i]);
      }
    }
  }
}

int main() {
  int v, e;
  scanf("%d %d", &v, &e);

  G.assign(v, vi());

  int a, b;
  for(int i=0; i<e; i++) {
    scanf("%d %d", &a, &b);
```

```
    G[a].pb(b);
  }

  bfs(0);

  printf("Distances\n");
  for(int i=0; i<v; i++)
    printf("%d ", dist[i]);
  printf("\n");

  printf("Parents\n");
  for(int i=0; i<v; i++)
    printf("%d ", parent[i]);
  printf("\n");

  return 0;
}
```

3.2.1 Graph Bicoloring

```
#include <cstdio>
#include <vector>
#include <queue>
#include <cstring>

using namespace std;

typedef long long           ll;
typedef vector<int>         vi;
#define pb push_back

int main() {
  int n, m;
  scanf("%d", &n);

  while(n) {
    scanf("%d", &m);

    vector<vi> G(n);

    int a, b;
    for(int i=0; i<m; i++) {
      scanf("%d %d", &a, &b);
      G[a].push_back(b);
      G[b].push_back(a);
    }

    bool sw = true;

    // BFS
    queue<int> Q;
    vi color(n, -1);
    Q.push(0);
    color[0] = 0;
    while(!Q.empty()) {
      int u = Q.front();
      Q.pop();

      for(int i=0; i<G[u].size(); i++) {
        if(color[G[u][i]] == -1) {
          color[G[u][i]] = (color[u]+1)%2;
          Q.push(G[u][i]);
        } else {
          if(color[G[u][i]] == color[u]) {
            sw = false;
            break;
          }
        }
```

```
      }

      if(!sw)
        break;
    }

    if(sw)
      printf("BICOLORABLE.\n");
    else
      printf("NOT BICOLORABLE.\n");

    scanf("%d", &n);
  }

  return 0;
}
```

3.2.2 Finding Connected Components in Undirect Graph

```
int n, e;
vector<vi> graph;
int dfsm[MAX];

void dfs(int start) {
  dfsm[start] = VISITED;
  cout << start << " ";
  for (int i = 0; i < graph[start].size(); i++) {
    if (dfsm[graph[start][i]] == NOT_VISITED) {
      dfs(graph[start][i]);
    }
  }
}

int main() {
  memset(dfsm, NOT_VISITED, sizeof dfsm);
  int numCC = 0;
  for (int i = 0; i < n; i++) {
    if (dfsm[i] == NOT_VISITED) {
      printf("Component %d:", ++numCC);
      dfs(i);
      printf("\n");
    }
  }
  return 0;
}
```

3.2.3 [Kruskal's] Algorithm

```
#include <cstdio>
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

typedef pair<int, int> ii;
#define MAX 1000000

int p[MAX], num_sets;
void initSet(int n) {
  for (int i = 0; i < n ; i++) p[i] = i;
  num_sets = n;
}
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)) num_sets--;
  p[findSet(i)] = findSet(j);
}

int main() {
  int v, e;
  scanf("%d %d", &v, &e);

  vector<pair<int, ii> > K;

  int a, b, c;
  for(int i=0; i<e; i++) {
    scanf("%d %d %d", &a, &b, &c);
    K.push_back(make_pair(c, ii(a, b)));
  }

  sort(K.begin(), K.end());

  int mst = 0;
  initSet(v);

  for(int i=0; i<e; i++) {
    if(!isSameSet(K[i].second.first, K[i].second.second)) {
      unionSet(K[i].second.first, K[i].second.second);
      mst += K[i].first;
    }
  }

  printf("mst = %d\n", mst);

  return 0;
}
```

3.2.4 [Prim's] Algorithm

```
#include <vector>
#include <queue>
#include <cstdio>
#include <cstring>

using namespace std;

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

vi taken;
priority_queue<ii> pq;
vector<vii> G;

void process(int vtx) {
  taken[vtx] = 1;
  for(int j=0; j<G[vtx].size(); j++) {
    ii v = G[vtx][j];
    if (!taken[v.first]) pq.push(ii(-v.second, -v.first));
  }
}

int main() {
  int v, e;
  scanf("%d %d", &v, &e);
  G.assign(v, vii());

  int a, b, c;
```

```
  for(int i=0; i<e; i++) {
    scanf("%d %d %d", &a, &b, &c);
    G[a].push_back(ii(b, c));
    G[b].push_back(ii(a, c));
  }

  taken.assign(v, 0);
  process(0);
  int mst_cost = 0;
  while(!pq.empty()) {
    ii front = pq.top(); pq.pop();
    int u = -front.second;
    int w = -front.first;
    if(!taken[u]) {
      mst_cost += w;
      process(u);
    }
  }

  printf("mst = %d\n", mst_cost);

  return 0;
}
```

3.3 Single-Source Shortest Path (SSSP)

3.3.1 SSSP on Unweighted Graph

```
void printPath(int u) {
  if (u == s) {
    cout << s << " ";
    return;
  }
  printPath(p[u]);
  cout << u << " ";
}

int main() {
  /* Build Graph */
  // SSPU
  // s: start, t: target
  map<int, int> dist;
  dist[s] = 0;
  queue<int> q;
  q.push(s);
  while(!q.empty()) {
    int u = q.front(); q.pop();
    for (int j = 0; j < g[u].size(); j++) {
      int v = g[u][j];
      if (!dist.count(v)) {
        dist[v] = dist[u] + 1;
        p[v] = u;
        q.push(v);
      }
    }
  }
  DBG(dist[t]);
  printPath(t);
  return 0;
}
```

3.3.2 SSSP on Weighted Graph [Dijkstra's]

```
#include <vector>
#include <queue>
#include <algorithm>
```

```
#include <cstdio>
#include <cstring>

using namespace std;

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

vector<vii> G;
vi dist(10001, INF);
vi parent(10001, 0);

void dijkstra(int n) {
  dist[n] = 0;
  priority_queue<ii, vector<ii>, greater<ii> > pq;
  pq.push(ii(0, n));

  while(!pq.empty()) {
    ii front = pq.top();
    pq.pop();
    int u = front.second, d = front.first;

    if(d > dist[u])
      continue;

    for(int i=0; i<G[u].size(); i++) {
      ii v = G[u][i];
      if(dist[u] + v.second < dist[v.first]) {
        dist[v.first] = dist[u] + v.second;
  parent[v.first] = u;
        pq.push(ii(dist[v.first], v.first));
      }
    }
  }
}

int main() {
  int v, e;
  scanf("%d %d", &v, &e);

  G.assign(v, vii());
  int a, b, c;
  for(int i=0; i<e; i++) {
    scanf("%d %d %d", &a, &b, &c);
    G[a].pb(ii(b, c));
  }

  dijkstra(0);

  printf("Distances\n");
  for(int i=0; i<v; i++)
    printf("%d ", dist[i]);
  printf("\n");

  printf("Parents\n");
  for(int i=0; i<v; i++)
    printf("%d ", parent[i]);
  printf("\n");

  return 0;
}
```

3.3.3 Finding Connected Components in Undirect Graph

```
int n, e;
```

```
vector<vi> graph;
int dfsm[MAX];

void dfs(int start) {
  dfsm[start] = VISITED;
  cout << start << " ";
  for (int i = 0; i < graph[start].size(); i++) {
    if (dfsm[graph[start][i]] == NOT_VISITED) {
      dfs(graph[start][i]);
    }
  }
}

int main() {
  memset(dfsm, NOT_VISITED, sizeof dfsm);
  int numCC = 0;
  for (int i = 0; i < n; i++) {
    if (dfsm[i] == NOT_VISITED) {
      printf("Component %d:", ++numCC);
      dfs(i);
      printf("\n");
    }
  }
  return 0;
}
```

3.3.4 [Floyd Warshall's] Algorithm

```
#include <cstdio>
#include <algorithm>

using namespace std;

#define INF 1000000000

int main() {
  int v, e;
  scanf("%d %d", &v, &e);

  int M[v][v];
  for(int i=0; i<v; i++) {
    for(int j=0; j<v; j++)
      M[i][j] = INF;
    M[i][i] = 0;
  }

  int a, b, c;
  for(int i=0; i<e; i++) {
    scanf("%d %d %d", &a, &b, &c);
    M[a][b] = c;
  }

  // Floyd Warshall Algotithm.
  for(int k=0; k<v; k++)
    for(int i=0; i<v; i++)
      for(int j=0; j<v; j++)
  M[i][j] = min(M[i][j], M[i][k] + M[k][j]);

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

  return 0;
}
```

3.4 Maximum Flow

3.4.1 [Edmonds Karp's] Algorithm

```
int res[100][100];
int mf, f;
vi p;
int s, t;

void augment(int v, int minEdge) {
  if (v == s) {
    f = minEdge;
    return;
  } else if (p[v] != -1) {
    augment(p[v], min(minEdge, res[p[v]][v]));
    res[p[v]][v] -= f;
    res[v][p[v]] += f;
  }
}

int main() {
  /* Build adjacency matrix res */
  scanf("%d %d", &s, &t);
  mf = 0;
  while (1) {
    f = 0;
    vi dist(n, INT_INF);
    dist[s] = 0;
    queue<int> q; q.push(s);
    p.assign(n, -1);
    while (!q.empty()) {
      int u = q.front(); q.pop();
      if (u == t) break;
      for (int v = 0; v < n; v++)
        if (res[u][v] > 0 && dist[v] == INT_INF)
          dist[v] = dist[u] + 1, q.push(v), p[v] = u;
    }
    augment(t, INT_INF);
    if (f == 0) break;
    mf += f;
  }
  DBG(mf);
  return 0;
}
```

Chapter 4

String

4.1 KMP's Algorithm

```
#include <cstdio>
#include <iostream>
#include <vector>

using namespace std;

typedef vector<int> vi;
#define pb push_back

string s, t;
vi P;
vi M;

void KMPPreprocess() {
  P.assign(t.size() + 1, -1);
  for(int i=1; i<=t.size(); i++) {
    int pos = P[i-1];
    while(pos != -1 && t[pos] != t[i-1]) pos = P[pos];
    P[i] = pos + 1;
  }
}

void KMPSearch() {
  M.clear();
  for(int sp=0, kp=0; sp<s.size(); sp++) {
    while(kp != -1 && (kp == t.size() || t[kp] != s[sp]))
      kp = P[kp];
    kp++;
    if(kp == t.size()) M.pb(sp + 1 - t.size());
  }
}

int main() {
  cin >> s >> t;

  KMPPreprocess();
  KMPSearch();

  for(int i=0; i<M.size(); i++)
    printf("%d\n", M[i]);

  return 0;
}
```

Chapter 5

Computational Geometry

I never program geometry problems, because
there are better things to do with my life

Fidel Schaposnik

5.1 Geometry objects 2D

5.1.1 Point

```
#include <iostream>
#include <cstdio>
#include <cmath>

using namespace std;

#define EPS 1e-8
#define PI acos(-1)

struct point {
  double x, y;
  point(double _x, double _y) {
    x = _x, y = _y;
  }
  bool operator < (point other) {
    if (fabs(x - other.x) < EPS)
      return x < other.x;
    return y < other.y;
  }
};

bool areSame(point p1, point p2) {
  return fabs(p1.x - p2.x) < EPS && fabs(p1.y - p2.y) < EPS;
}

double dist(point p1, point p2) {
  return hypot(p1.x - p2.x, p1.y - p2.y);
}

// cross product between 3 points
double cross(point p, point q, point r) {
  return (r.x - q.x) * (p.y - q.y) - (r.y - q.y) * (p.x - q.x);
}

// return true if point r is on the same line as the line pq
bool collinear (point p, point q, point r) {
```

```
  return fabs(cross(p,q,r)) < EPS;
}

// return true if point r is on the left side of line pq
bool ccw(point p, point q, point r) {
  return cross(p, q, r) > 0; // can be modified to accept collinear points
}

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

point rotate(point p, double theta) {
  double rad = DEG_to_RAD(theta);
  return point(p.x * cos(rad) - p.y * sin(rad), p.x * sin(rad) + cos(rad));
}


int main() {
  point r(3,0);
  point q(6,0);
  point p(6,4);
  printf("%f\n", cross(p,q,r));
  if (collinear(p,q,r))
    printf("Collinear\n");
  else
    printf("No collinear\n");
  if (ccw(p,q,r))
    printf("CCW\n");
  else
    printf("No CCW\n");
  return 0;
}
```

5.1.2 Line

```
struct line {
  double a, b, c;
};

// the answer is stored in the third parameter (pass by reference)
void pointsToLine(point p1, point p2, line *l) {
  if (p1.x == p2.x) { // vertical line is handled nicely here
    l->a = 1.0, l->b = 0.0; l->c = -p1.x;
  } else {
    l->a -(double)(p1.y - p2.y) / (p1.x - p2.x);
    l->b = 1.0;
    l->c = -(double)(l->a * p1.x) - (l->b * p1.y);
  }
}

// my implementation of pointsToLine
void pointsToLine2(point p1, point p2, line *l) {
  if (p1.x == p2.x) {
    l->a = 1.0, l->b = 0.0, l->c = -p1.x;
  } else {
    l->a = -(p2.y - p1.y) / (p2.x - p1.x);
    l->b = 1.0;
    l->c = -(l->a * p1.x) - p1.y;
  }
}

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

bool areSame(line l1, line l2) {
  return areParallel(l1, l2) && (fabs(l1.c - l2.c) < EPS);
```

```
}

bool areIntersect(line l1, line l2, point *p) {
  if (areSame(l1, l2)) return false;
  if (areParallel(l1, l2)) return false;
  //solve system of 2 linear algebraic equation with 2 unknowns
  p->x = (l2.b * l1.c * l2.c) / (l2.a * l1.b - l1.a * l2.b);
  if (f(abs(l1.b) > EPS)
    p->y = -(l1.a * p->x + l1.c) / l1.b;
  else
    p->y = -(l2.a * p->x + l2.c) / l2.b;
}
```

5.1.3 Polygon

```
// return the perimeter, which is the sum of Euclidian distances
// of consecutive line segments (polygon edges)
double perimeter(vector<point> P) {
  double result = 0.0;
  for (int i = 0; i < P.size() - 1; i++)
    result += dist(P[i],P[i+1]);
  return result;
}

// returns the area, which is half the determinant
double area(vector<point> P) {
  double result = 0.0;
  double x1, y1, x2, y2;
  for (int i = 0; i < P.size() - 1; i++) {
    x1 = P[i].x; x2 = P[i+1].x;
    y1 = P[i].y; y2 = P[i+1].y;
    result += (x1 * y2 - x2 * y1);
  }
  return fabs(result)/ 2.0;
}

// returns true if all three consecutive vertices of P form the same turns
bool isConvex(vector<point> P) {
  int sz = P.size() - 1;
  if (sz < 3)
    return false;
  bool isLeft = ccw(P[0], P[1], P[2]);
  for (int i = 1; i < P.size(); i++)
    if (ccw(P[i], P[(i+1) % sz], P[(i+2) % sz]) != isLeft)
      return false;
  return true;
}

double angle(point a, point b, point c) {
  double ux = b.x - a.x, uy = b.y - a.y;
  double vx = c.x - a.x, vy = c.y - a.y;
  return acos((ux*vx + uy*vy)/ sqrt((ux*ux + uy*uy)*(vx*vx+vy*vy)));
}

// returns true if point p is in either convex/concave polygon P
bool inPolygon(point p, vector<point> P) {
  if (P.size() == 0) return false;
  for (int i = 0; i < P.size(); i++) // point is in P
    if (fabs(P[i].x - p.x) < EPS && fabs(P[i].y - p.y) < EPS)
      return true;
  double sum = 0;
  for (int i = 0; i < P.size() - 1; i++)
    if (cross(p, P[i], P[i+1]) < 0)
      sum -= angle(p, P[i], P[i+1]);
    else
      sum += angle(p, P[i], P[i+1]);
  return (fabs(sum - 2*PI) < EPS || fabs(sum + 2*PI) < EPS);
}
```

```
int main() {
  vector<point> P;
  P.push_back(point(0,0));
  P.push_back(point(0,10));
  P.push_back(point(10,0));
  P.push_back(point(0,0));
  point p(5,5);

  if (inPolygon(p, P))
    cout << "IN POLYGON" << endl;
  else
    cout << "NOT IN POLYGON" << endl;
  return 0;
}
```

5.2 Convex hull

5.2.1 Graham's Algorithms

```
point pivot(0,0);

bool angleCmp(point a, point b) {
  if (collinear(pivot, a, b))
    return dist(pivot, a) < dist(pivot, b); // determine wich one is closer
  double d1x = a.x - pivot.x, d1y = a.y - pivot.y;
  double d2x = b.x - pivot.x, d2y = b.y - pivot.y;
  return (atan2(d1y, d1x) - atan2(d2y, d2x)) < 0;
}

vector<point> CH(vector<point> P) {
  int i;
  int N = P.size();
  if (N < 3) return P; // special case, the CH is P itself

  // first, find P0 = point with lowest Y and if tie, rightmost X
  int P0 = 0;
  for (i = 1; i < N; i++)
    if (P[i].y < P[P0].y || P[i].y == P[P0].y && P[i].x > P[P0].x)
      P0 = i;
  // swap selected vertex with P[0]
  point temp = P[0];
  P[0] = P[P0];
  P[P0] = temp;

  // second, sort points by angle w.r.t. pivot P0
  pivot = P[0]; // use this global variable as reference
  sort(++P.begin(), P.end(), angleCmp); // notice that we does not sort P[0]

  // third, the ccw tests
  point prev(0,0), now(0,0);
  stack<point> S; S.push(P[N-1]); S.push(P[0]); // initial content of stack S
  i = 1;  // then, we check the rest
  while (i < N) {
    now = S.top();
    S.pop(); prev = S.top(); S.push(now); // get 2nd from top
    if (ccw(prev, now, P[i])) S.push(P[i++]); // left turn, accept
    else S.pop(); // otherwise, pop the top of stack S until we have a left turn
  }

  vector<point> ConvexHull;
  while (!S.empty()) {
    ConvexHull.push_back(S.top());
    S.pop();
  }
  return ConvexHull;
}
```

```
int main() {
  vector<point> P;
  P.push_back(point(0,0));
  P.push_back(point(1,0));
  P.push_back(point(2,0));
  P.push_back(point(2,2));
  P.push_back(point(0,2));
  vector<point> R = CH(P);
  for(vector<point>::iterator it = R.begin(); it != R.end(); it++)
    cout << it->x << " " << it-> y << endl;
  return 0;
}
```