# Resumen de algoritmos para maratones de programación

Manuel Felipe Pineda - Diego Alejandro Martínez

2 de octubre de 2012

## Índice

# 1. Plantilla

```
#include <cstdio>
#include <cmath>
#include <functional>
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
#include <queue>
#include <stack>
#include <list>
#include <map>

using namespace std;

#define all(x) x.begin(),x.end()
#define rep(i,a,b) for(int i=a;i<b;i++)
#define REP(i,n) rep(i,0,n)
#define foreach(x, v) for (typeof (v).begin() x = (v).begin(); \
x != (v).end(); ++x)
#define D(x) cout << #x " = " << x << endl;

typedef long long int lld;
typedef pair<int,int> pii;
typedef vector<int> vi;
typedef vector<pii> vpii;

int main(){

  return 0;
}
```

.....

# 2. Grafos

## 2.1. Depth First Search

Es un algoritmo para recorrer o buscar en un grafo. Empieza visitando la raiz y luego explora a fondo cada rama antes de hacer el backtraing.

Complejidad $O(|V| + |E|)$

```
/*
recordar
#define all(x) x.begin(),x.end()
typedef vector<int> vi;
typedef vector<vi> vvi;
*/
int N; // Número de vertices.
vvi G; // Grafo.
vi visited; // Vector de visitados.

void dfs(int i) {
  if(!visited[i]) {
    visited[i] = true;
    for_each(all(G[i]), dfs);
  }
}

bool check_graph_connected_dfs() {
  int start_vertex = 0;
  visited = vi(N, false);
  dfs(start_vertex);
  return (find(all(V), 0) == V.end());
}
```

.....

## 2.2. Breadth First Search

Es un algoritmo de búsqueda que empieza en la raíz y explora todos los nodos vecinos. Luego para cada vecino repite el proceso hasta encontrar la meta.
Complejidad $O(|V| + |E|)$

```
/*
Se considera un Grafo NO dirigido.
*/

int N; // number of vertices
vvi G; // lists of adjacent vertices
```

```
void bfs(){
  int start_vertex = 0;
  vi V(N, false);
  queue<int> Q;
  Q.push(start_vertex);
  V[start_vertex] = true;
  while(!Q.empty()) {
    int i = Q.front();
    Q.pop();
    foreach(G[i], it) {
      if(!V[*it]) {
        V[*it] = true;
        Q.push(*it);
      }
    }
  }
}
```

## 2.3. Dijkstra

Calcula la ruta más corta a todos los nodos desde un origen.
Todas las aristas deben ser NO negativas, en ese caso usar Bellman-ford.
Complejidad $O(E\ logV)$

```
/***
* Recordar, typedef pair<int,int> pii;
* typedef vector<pii> vii;
* typedef vector<vii> vvii;
***/

vi D (N, 987654321);
vi P (N, -1);
// D[i] es la distancia desde el vértice inicial hasta i
// P[i] es el predecesor de i en la ruta más corta
priority_queue < pii, vector < pii >, greater < pii > >Q;
//Cola de prioridad con comparador Mayor,
//top() retorna la distancia más lejana.
//Inicializar el vértice inicial.
D[0] = 0;
Q.push (pii (0, 0));
while (!Q.empty ()) {
  pii top = Q.top ();
  Q.pop ();
  int v = top.second, d = top.first;
  if (d <= D[v]) {
    foreach(G[v], it) {
      int v2 = it->first, cost = it->second;
      if (D[v2] > D[v] + cost) {
        D[v2] = D[v] + cost;
        P[v2] = v;
        Q.push (pii (D[v2], v2));
      }
    }
  }
}

//Código basado en TopCoder Algorithms tutorials.
```

Colección de códigos útiles para aritmética modular

```
int gcd (int a, int b) {
  int tmp;
  while (b) {
    a %= b;
    tmp = a;
    a = b;
    b = tmp;
  }
  return a;
}

// a % b (valor positivo)
int mod (int a, int b) {
  return ((a % b) + b) % b;
}

// returns d = gcd(a,b); finds x,y such that d = ax + by
int extended_euclid (int a, int b, int &x, int &y) {
  int xx = y = 0;
  int yy = x = 1;
  while (b) {
    int q = a / b;
    int t = b;
    b = a % b;
    a = t;
    t = xx;
    xx = x - q * xx;
    x = t;
    t = yy;
    yy = y - q * yy;
    y = t;
  }
  return a;
}


int lcm (int a, int b) {
  return a / gcd (a, b) * b;
}

// finds all solutions to ax = b (mod n)
vi modular_linear_equation_solver (int a, int b, int n) {
  int x, y;
  vi solutions;
  int d = extended_euclid (a, n, x, y);
  if (!(b % d)) {
    x = mod (x * (b / d), n);
    for (int i = 0; i < d; i++)
      solutions.push_back (mod (x + i * (n / d), n));
  }
  return solutions;
}

// computes b such that ab = 1 (mod n), returns -1 on failure
int mod_inverse (int a, int n) {
  int x, y;
```

```
  int d = extended_euclid (a, n, x, y);
  if (d > 1)
    return -1;
  return mod (x, n);
}

// Chinese remainder theorem (special case): find z such that
// z % x = a, z % y = b. Here, z is unique modulo M = lcm(x,y).
// Return (z,M). On failure, M = -1.
pii chinese_remainder_theorem (int x, int a, int y, int b) {
  int s, t;
  int d = extended_euclid (x, y, s, t);
  if (a % d != b % d)
    return make_pair (0, -1);
  return make_pair \
  (mod (s * b * x + t * a * y, x * y) / d, x * y / d);
}

// Chinese remainder theorem: find z such that
// z % x[i] = a[i] for all i. Note that the solution is
// unique modulo M = lcm_i (x[i]). Return (z,M). On
// failure, M = -1. Note that we do not require the a[i]'s
// to be relatively prime.
pii chinese_remainder_theorem (const vi & x, const vi & a) {
  pii ret = make_pair (a[0], x[0]);
  for (int i = 1; i < x.size (); i++) {
    ret = chinese_remainder_theorem \
    (ret.first, ret.second, x[i], a[i]);
    if (ret.second == -1) break;
  }
  return ret;
}

// computes x and y such that ax + by = c; on failure, x = y =-1
void linear_diophantine (int a, int b, int c, int &x, int &y) {
  int d = gcd (a, b);
  if (c % d) {
    x = y = -1;
  }
  else {
    x = c / d * mod_inverse (a / d, b / d);
    y = (c - a * x) / b;
  }
}
```

.....

.....

### 3.2. Mayor exponente de un primo que divide a n!

```
int pow_div_fact(int n, int p) {
  int sd = 0;
  for (int t = n; t > 0; t /= p)
    sd += t % p;
  return (n-sd)/(p-1);
}
```

.....

```
typedef unsigned int uint;

//retorna el siguiente entero con la mimsma
//cantidad de 1's en la representación binaria

uint next_popcount(uint n){
  uint c = (n & -n);
  uint r = n+c;
  return (((r ^ n) >> 2) / c) | r;
}

//retorna el primer entero con n 1's en binario
uint init_popcount(int n){
  return (1 << n) - 1;
}
```

.....