

# Resumen de algoritmos para maratones de programación

Manuel Felipe Pineda - Diego Alejandro Martínez

2 de octubre de 2012

## Índice

<b>1. Plantilla</b>	<b>1</b>	4.6. Determinar si un punto está dentro de un polígono cualquiera	6
<b>2. Grafos</b>	<b>2</b>	4.7. Intersección de dos rectas . . . . .	6
2.1. Depth First Search . . . . .	2	4.8. Intersección de dos segmentos . . . . .	6
2.2. Breadth First Search . . . . .	2	4.9. Determinar si dos segmentos se intersectan o no . . . . .	6
2.3. Dijkstra . . . . .	2	4.10. Centro del círculo que pasa por tres puntos . . . . .	6
2.4. Bellman-Ford . . . . .	3	4.11. Par de puntos más cercanos . . . . .	6
2.5. Floyd-Warshall . . . . .	3	4.12. Par de puntos más alejados . . . . .	6
2.6. Johnson . . . . .	3	4.13. Área de un polígono . . . . .	6
2.7. Minimum Spanning Tree: Kruskal . . . . .	3	4.14. Convexhull . . . . .	6
2.8. Minimum Spanning Tree: Prim . . . . .	3	<b>5. Strings</b>	<b>6</b>
2.9. Strongly Connected Components . . . . .	3	5.1. Knuth-Morris-Pratt KMP . . . . .	6
2.10. Puntos de articulación . . . . .	3	5.2. Aho-Corasick . . . . .	6
2.11. 2-SAT . . . . .	3	5.3. Suffix Array . . . . .	6
2.12. Maximum bipartite matching . . . . .	3	<b>6. Teoría de Juegos</b>	<b>6</b>
2.13. Flujo Máximo . . . . .	3	<b>7. Estructuras de Datos</b>	<b>6</b>
2.14. Lowest Common Ancestor: TarjanOLCA . . . . .	3	7.1. Prefix Tree - Trie . . . . .	6
<b>3. Matemáticas</b>	<b>3</b>	7.2. Fenwick Tree . . . . .	6
3.1. Aritmética Modular . . . . .	3	7.3. Interval Tree . . . . .	6
3.2. Mayor exponente de un primo que divide a $n!$ . . . . .	5	<b>8. Hashing</b>	<b>6</b>
3.3. Potencia modular . . . . .	6	8.1. FNV Hash . . . . .	6
3.4. Criba de Eratóstenes . . . . .	6	8.2. JSW Hash . . . . .	6
<b>4. Geometría</b>	<b>6</b>	<b>9. Misceláneo</b>	<b>6</b>
4.1. Utilidades Geometría . . . . .	6	9.1. Bitwise operations . . . . .	6
4.2. Distancia mínima: Punto-Segmento . . . . .	6		
4.3. Distancia mínima: Punto-Recta . . . . .	6		
4.4. Determinar si un polígono es convexo . . . . .	6		
4.5. Determinar si un punto está dentro de un polígono convexo . . . . .	6		

# 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> vpri;

int main(){

    return 0;
}

.....
```

## 2. Grafos

### 2.1. Depth First Search

Es un algoritmo para recorrer o buscar en un grafo. Empieza visitando la raíz 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  $O(E \log V)$

```

/**
 * 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.

```

.....

- 2.4. Bellman-Ford
- 2.5. Floyd-Warshall
- 2.6. Johnson
- 2.7. Minimum Spanning Tree: Kruskal
- 2.8. Minimum Spanning Tree: Prim
- 2.9. Strongly Connected Components
- 2.10. Puntos de articulación
- 2.11. 2-SAT
- 2.12. Maximum bipartite matching
- 2.13. Flujo Máximo
- 2.14. Lowest Common Ancestor: TarjanOLCA

### 3. Matemáticas

#### 3.1. Aritmética Modular

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);
}

```

.....

3.3. Potencia modular

3.4. Criba de Eratóstenes

## 4. Geometría

4.1. Utilidades Geometría

4.2. Distancia mínima: Punto-Segmento

4.3. Distancia mínima: Punto-Recta

4.4. Determinar si un polígono es convexo

4.5. Determinar si un punto está dentro de un polígono convexo

4.6. Determinar si un punto está dentro de un polígono cualquiera

4.7. Intersección de dos rectas

4.8. Intersección de dos segmentos

4.9. Determinar si dos segmentos se intersectan o no

4.10. Centro del círculo que pasa por tres puntos

4.11. Par de puntos más cercanos

4.12. Par de puntos más alejados

4.13. Área de un polígono

4.14. Convexhull

## 5. Strings

5.1. Knuth-Morris-Pratt KMP

5.2. Aho-Corasick

5.3. Suffix Array

## 6. Teoría de Juegos

## 7. Estructuras de Datos

<sup>6</sup> 7.1. Prefix Tree - Trie

7.2. Fenwick Tree

```
typedef unsigned int uint;

//retorna el siguiente entero con la misma
//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;
}

.....
```