# Índice

**1. Matemáticas** **2**
1.1. Aritmética . . . 2
1.1.1. Números de Catalan . . . 2
1.1.2. Sucesion de Fibonacci: formula . . . 2
1.1.3. Función de Euler . . . 2
1.2. Aritmética modular . . . 2
1.2.1. Fórmulas interestantes . . . 2
1.2.2. Exponenciación modular . . . 2
1.2.3. Algoritmo de Euclides extendido . . . 2
1.2.4. Factorización de un entero . . . 3
1.3. Probabilidad . . . 3
1.3.1. Propiedades . . . 3
1.4. Combinatoria . . . 4
1.4.1. Fórmulas interesantes . . . 4
1.4.2. Coeficiente binomial . . . 4
1.4.3. Números de Euler . . . 4
1.4.4. Logaritmo . . . 4

**2. Grafos** **4**
2.1. Clases base . . . 4
2.2. Algoritmos sobre grafos . . . 5
2.2.1. Camino euleriano . . . 5
2.2.2. Algoritmo de Dijkstra . . . 6
2.2.3. Algoritmo de Bellman-Ford . . . 6
2.2.4. Algoritmo de Floyd-Warshall . . . 7
2.2.5. Ordenacion topologica . . . 8
2.2.6. Algoritmo de Tarjan . . . 8
2.3. Recorridos de grafos . . . 9
2.3.1. BFS y DFS . . . 9
2.4. Árbol de recubrimiento de peso mínimo. Algoritmo de Kruskal. . . . 9
2.5. Flujo máximo en un grafo . . . 10
2.5.1. Algoritmo de Edmonds-Karp . . . 10
2.6. Emparejamiento de máximo/mínimo peso en un grafo bipartido . . . 12
2.6.1. Algoritmo de Kuhn-Munkres o algoritmo húngaro . . . 12

**3. Estructuras de datos** **14**
3.1. MFSets . . . 14
3.2. Prefix tree . . . 14
3.3. Segment tree . . . 15
3.4. Binary Search Tree . . . 16
3.4.1. Numero de secuencias que crean un BST . . . 18

**4. Programación Dinámica** **18**
4.1. Subsecuéncia común máxima (LCS) . . . 18
4.2. Subsecuencia creciente/decreciente máxima (LIS/LDS) . . . 18

**5. Algoritmos** **19**
5.1. Algoritmo del matrimonio estable . . . 19
5.2. Algoritmo de la mochila . . . 19
5.3. Algoritmo KMP (buscar subcadena en cadena) y Boyer-Moore . . . 20

**6. Geometría** **21**
6.1. Clases base . . . 21
6.2. Convex Hull . . . 23
6.3. Funciones . . . 24
6.3.1. Area de un Poligono . . . 24
6.3.2. Puntos enteros sobre un segmento . . . 24
6.4. Propiedades . . . 24
6.4.1. Teorema de Pick: . . . 24
6.4.2. Intersección entre línea y círculo: . . . 25

# 1. Matemáticas

## 1.1. Aritmética

$mcm(x,y) = \frac{x*y}{mcd(x,y)}$
$c^2 = a^2 + b^2 - 2*a*b*cos(C)$
$a = b*cos(C) + c*cos(B)$
$\frac{a}{sen(A)} = \frac{b}{sen(B)} = \frac{c}{sen(C)}$

### 1.1.1. Números de Catalan

$C_0 = 1; C_{n+1} = \frac{2(2n+1)}{n+2}C_n$

### 1.1.2. Sucesion de Fibonacci: formula

$F_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$

### 1.1.3. Función de Euler

Devuelve el número de números $\leq n$ que son coprimos con $n$.

```
int euler_function(int n) {
  int tot = n;
  for (int p = 2; p * p <= n; p++)
    if (n % p == 0) {
      tot /= p;
      tot *= (p - 1);
      while (n % p == 0) n /= p;
    }
  if (n > 1) {
    tot /= n;
    tot *= (n - 1);
  }
  return tot;
}
```

## 1.2. Aritmética modular

### 1.2.1. Fórmulas interestantes

$$(x+y)modn = ((xmodn) + (ymodn))modn$$
$$(x-y)modn = ((xmodn) - (ymodn))modn$$
$$(x*y)modn = ((xmodn) * (ymodn))modn$$
$$x^y modn = (xmodn)^y modn$$

Teorema chino del resto: Sean $n_1$, $n_2$, ..., $n_k$ primos dos a dos. Entonces el conjunto de ecuaciones $x_i \equiv (modn_i)$ tiene una única solución módulo $n = n_1 * n_2 * ... * n_k$ y es:

$$x = \Sigma(a_i * m_i * y_i)$$

Para todo i comprendido entre 1 y k, donde $m_i$ e $y_i$ se definen como:

$$m_i = n/n_i$$
$$y_i = m_i^{-1} modn_i$$

Para el cálculo de $y_i$ se debe usar el algoritmo de Euclides extendido, el cual también está en este documento.

Si $a_1 = b_1 modn$ y $a_2 = b_2 modn$, entonces: $a_1 + a_2 = b_1 + b_2 modn$ y $a_1 a_2 = b_1 b_2 modn$..

Si $a$ y $b$ son enteros, la congruencia: $ax = b(modn)$ tiene solución $x$ si y sólo si el máximo común divisor $(a,n)$ divide a $b$. En particular, existirán exactamente $d = mcd(a,n)$ soluciones en el conjunto de residuos $\{0, 1, 2, ..., n-1\}$.

$$x_0 + k\frac{n}{d}$$

Obtener $x_0$: $ax_0 \equiv bmodc \to ax_0 + cy = b$.

Una ecuación lineal diofántica de laforma $ax + by = n$ tiene solución entera $x_0, y_0$ si y sólo si el máximo común divisor de $a$ y $b$ es un divisor de $n$.

Definimos $d = gcd(a,b)$. La solución particular de dicha ecuación puede obtenerse de la siguiente forma:

$$x_0 = \frac{n}{d}p;\ y_0 = \frac{n}{d}q$$

Siendo: $d = p \times a + q \times b$.

### 1.2.2. Exponenciación modular

```
int expmod(int a, int n, int m) { // a^n mod m
    int i = n;
    int r = 1;
    int x = a;
    while (i > 0) {
        if (i%2 == 1) r = (r*x)%m;
        x = (x*x)%m;
        i /= 2;
    }
    return r;
}
```

```
#include <cstdlib>

bool Fermat(long long p, int iterations) {
    if (p == 1) {
        return false;
    }
    for (int i = 0; i < iterations; i++) {
        ll a = std::rand() % (p - 1) + 1;
        if (expmod(a, p - 1, p) != 1) {
            return false;
        }
    }
    return true;
}
```

### 1.2.3. Algoritmo de Euclides extendido

Calcula el máximo común divisor entre dos enteros $a$ y $b$ y devuelve los números $p$, $q$ tales que $a*p + b*q = mcd(a,b)$. Si $a$ y $b$ son relativamente primos, podemos obtener $b^{-1}$ mod $a$ con este algoritmo. El valor de $q$ devuelto será dicho inverso.

```cpp

#include <cmath>

struct GcdOutput {
    int gcd, p, q;
};
GcdOutput gcdExt(int a, int b) {
  GcdOutput solution;
  if (b > a) {
    solution = gcdExt(b, a);
    int s = solution.q;
        solution.q = solution.p;
    solution.p = s;
    return solution;
  }
  if (b == 0) {
        GcdOutput out;
    out.p = 1;
    out.q = 0;
        out.gcd = a;
    return out;
  }
  solution = gcdExt(b, a % b);
  int p1 = solution.p;
  int q1 = solution.q;
  solution.p = q1;
  solution.q = (p1 - ((int) floor(a / b)) * q1);

  return solution;
}
```

```cpp
// Solves x*a+y*b=n
struct Sol { int x, y; };
Sol solve_diophantine(int a, int b, int n) {
  GcdOutput sol = gcdExt(a,b);
  int d = sol.gcd, p = sol.p, q = sol.q;
    DiophanticSolution s;
  s.x = n / d * p;
  s.y = n / d * q;
  return s;
}
```

### 1.2.4. Factorización de un entero

Divide un número entero en sus factores primos. Hay $nfac$ factores distintos. factor[n] indica el n-ésimo factor y potencia[n] el número de veces que se repite. $n = factor[0]^{potencia[0]} \cdot factor[1]^{potencia[1]} \ldots factor[nfac-1]^{potencia[nfac-1]}$

```cpp
int factor[1000];
int potencia[1000];
int nfac = 0;
void factores(int res) {
    // potencias de 2
    if (res%2 == 0) {
        res /= 2;
        factor[0] = 2;
        potencia[0] = 1;
        nfac = 1;
        while (res%2 == 0) {
            potencia[0]++;
            res /= 2;
        }
    }
    // potencias impares
    for (int i = 3; i <= res; i+=2)
        if (res%i == 0) {
            res /= i;
            factor[nfac] = i;
            potencia[nfac] = 1;
            while (res%i == 0) {
                potencia[nfac]++;
                res /= i;
            }
            nfac++;
        }
}
```

## 1.3. Probabilidad

### 1.3.1. Propiedades

Si A y B son excluyentes:

$$P(A \cup B) = P(A) + P(B)$$

$$P(A \cap B) = 0$$

Suma de Sucesos:

$$P(A \cup B \cup C) = P(A) + P(B) + P(C) - P(A \cap B)$$

$$-P(A \cap C) - P(B \cap C) + P(A \cap B \cap C)$$

Producto de sucesos:

$$P(A \cap B) = P(A) \cdot P(B/A)$$

$$P(A \cap B) = P(B) \cdot P(A/B)$$

Si A y B son independientes:

$$P(A \cap B) = P(A) \cdot P(B)$$

Probabilidad Condicional:

$$P(A/B) = \frac{P(A \cap B)}{P(B)}$$

Teorema de Bayes:

$$P(A/B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A) \cdot P(B/A)}{\sum_{i=1}^{n} P(A_i) \cdot P(B/A_i)}$$

Teorema de probabilidad total:

$$P(B) = \sum_{i=1}^{n} P(A_i) \cdot P(B/A_i)$$

Esperanza Matematica: Para una variable aleatoria discreta con valores posibles $x_1, x_2, ...x_n$ y sus probabilidades representadas por la función de probabilidad $p(x_i)$ la esperanza se calcula como ejemplo:

$$E[X] = x_1 p(X = x_1) + ... + x_n p(X = x_n) = \sum_{i=1}^{n} x_i \cdot p(x_i)$$

## 1.4. Combinatoria

### 1.4.1. Fórmulas interesantes

Principio de inclusión - exclusión

$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$

Combinaciones y permutaciones para n elementos tomados en grupos de k Combinaciones sin repetición (no importa el orden):

$$\binom{n}{k} = \frac{n!}{k! \cdot (n-k)!}$$

Permutaciones sin repetición (sí importa el orden):

$$P(n,k) = \frac{n!}{(n-k)!}$$

Combinaciones con repetición (no importa el orden):

$$\frac{(n+k-1)!}{k!(n-1)!}$$

### 1.4.2. Coeficiente binomial

```
int bc2[1000][1000];     // tabla dp
int coeficiente_binomial2(int n, int k) {
    for (int i = 0; i <= n; i++)
        bc2[i][0] = bc2[i][i] = 1;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j < i; j++)
            bc2[i][j]=bc2[i-1][j-1]+bc2[i-1][j];
    return bc2[n][k];
}
```

```
int binlog(int bits) {
    // Output for negative numbers
    int log = 0; //is compiler-dependent (>>)
    if ( ( bits & 0xffff0000 ) != 0 ) {
        bits >>= 16; log = 16; }
    if ( bits >= 256 ) { bits >>= 8; log += 8; }
    if ( bits >= 16 ) { bits >>= 4; log += 4; }
    if ( bits >= 4 ) { bits >>= 2; log += 2; }
    return log + ( bits >> 1 );
}
```

Permutaciones con repetición (importa el orden):

$$n^k$$

Permutaciones con $r_i$ repeticiones para el i-ésimo elemento (hay k elementos y sí importa el orden):

$$\frac{n!}{r_1! \cdot r_2! \cdot \ldots \cdot r_k!}$$

Existen $\binom{n+k}{k}$ cadenas que continen k unos y n ceros Existen $\binom{n+1}{k}$ cadenas que continen k unos y n ceros tal que no hay dos unos adyacentes

### 1.4.3. Números de Euler

Cantidad de permutaciones de longitud $n$ que tienen $k$ sucesiones crecientes o series.

```
long euler(int n, int k) {
    // E(N,1) = E(N,N) = 1.
    // E(N,K) = 0 if K <= 0 or N < K.
    if (n == k || k == 1)
        return 1;
    if (k < 0 || n < k)
        return 0;

    long izq = k * euler(n-1, k);
    long der = (n - k + 1) * euler(n-1, k-1);

    return izq + der;
}
```

### 1.4.4. Logaritmo

# 2. Grafos

## 2.1. Clases base

```
#include <vector>
#include <sstream>

class Adyacente {
public:
    int dest, coste;
    Adyacente(int dest, int coste) {
        this->dest = dest;
        this->coste = coste;
    }
};

class Grafo {
public:
    int numNodos, numAristas;
    std::vector<int> gradosEntrada;
    std::vector<std::vector<Adyacente> > adyacentes;

    Grafo(int numNodos) {
```

```
        this->numNodos = numNodos;
        this->numAristas = 0;
        gradosEntrada.resize(numNodos, 0);
        adyacentes.resize(numNodos);
    }

    void insertarArista(int origen, int dest, int coste) {
        Adyacente ady(dest, coste);
        adyacentes[origen].push_back(ady);
        gradosEntrada[dest]++;
        numAristas++;
    }

    int getArista(int origen, int dest) {
        for (auto ady : adyacentes[origen]) {
            if (ady.dest == dest) {
                return ady.coste;
            }
        }
        return 0;
    }

    bool existeArista(int origen, int dest) {
        for (auto ady : adyacentes[origen]) {
            if (ady.dest == dest) {
                return true;
            }
        }
        return false;
    }

    void eliminarArista(int origen, int dest) {
        for (unsigned int i = 0; i < adyacentes[origen].size(); i++) {
            Adyacente ady = adyacentes[origen][i];
            if (ady.dest == dest) {
                adyacentes[origen].erase(adyacentes[origen].begin() + i);
                gradosEntrada[origen]--;
                numAristas--;
            }
        }
    }

    std::string toString() {
        std::stringstream res;
        for (int i = 0; i < numNodos; i++) {
            res << "Vertice: " << i;
            std::vector<Adyacente> l = adyacentes[i];
            if (l.empty()) {
                res << " sin adyacentes ";
            } else {
                res << " con adyacentes: ";
            }
            for (auto ady : l) {
                res << ady.dest << "(" << ady.coste << ") ";
            }
            res << "\n";
        }
        return res.str();
    }
};
```

## 2.2. Algoritmos sobre grafos

### 2.2.1. Camino euleriano

Camino euleriano a partir del nodo $n$. Devuelve en la lista $l$ el camino resultante. El camino es el lexicograficamente menor. AVISO: destruye el grafo que recibe como parámetro. El grafo tiene que ser no ponderado. $n$ ha de ser un nodo de grafo impar, solo puede haber 2 nodos de grado impar.

```
#include <vector>
```

```
vector<int> caminoEuleriano(bool dirigido) {
  vector<int> camino, caminoTemp;
  int i = 0;
  unsigned int x = 0;
  camino.push_back(0);

  while (true) {
    vector<Adyacente> adyacentes = g.adyacentes[i];
    if (!adyacentes.empty()) {
      caminoTemp.push_back(i);
      int j = adyacentes[0].dest;
      g.eliminarArista(i, j);
      if (!dirigido)
        g.eliminarArista(j, i);
      i = j;
    } else {
      if (caminoTemp.size() != 0) {
              for (vector<int>::iterator current = camino.begin();
                                 current != camino.end(); current++)
                  if (*current == caminoTemp.at(0)) {
                      camino.insert(current, caminoTemp.begin(), caminoTemp.end());
                      break;
                  }
        caminoTemp.clear();
        x = 0;
      } else
        x++;
      if (x >= camino.size())
        break;
      i = camino.at(x);
    }
  }

  return camino;
}
```

### 2.2.2. Algoritmo de Dijkstra

```
#include <vector,queue,utility,climits>
typedef pair<int, int> ii;
vector<int> distanciaMin(AdjList.size(), 1000000000);

void dijkstra(int origen) {
    priority_queue< ii,vector<ii>,greater<ii> > caminos;
    caminos.push(ii(0, origen));
    while (!caminos.empty()) {
        ii actual = caminos.top(); caminos.pop();

        if (actual.first < distanciaMin[actual.second]) {
            distanciaMin[actual.second] = actual.first;
            for (auto ady : AdjList[actual.second])
                caminos.push(ii(actual.first + ady.coste, ady.dest));
} } }
```

### 2.2.3. Algoritmo de Bellman-Ford

Calcula la distancia mínima entre los nodos ini y fin. Si no hay conexion entre ini y fin devuelve 99999999. El grafo es ponderado y puede tener aristas negativas pero no puede haber ciclos negativos. Para detectar ciclos negativos, pueden hacerse (|V|-1) iteraciones del bucle while(!salir) y luego una iteracion adicional. Si en la iteracion adicional hay cambios en el vector dist es porque hay ciclos de peso total negativo.

```
#include <vector>
#define MAX_VALUE 1e9

vector<vector<int>> BellmanFord(int origen) {
  vector<int> coste(AdjList.size());
  vector<int> pred(AdjList.size());

  for (int i = 0; i < AdjList.size(); i++) {
```

```
        coste[i] = MAX_VALUE >> 1;
        pred[i] = -1;
    }
    coste[origen] = 0;
    pred[origen] = origen;
    for (int i = 1; i < AdjList.size(); i++) {
            bool salir = true;
        for (int n = 0; n < AdjList.size(); n++) {
            if (coste[n] != MAX_VALUE >> 1) {
            for (auto ady : AdjList[n]) {
                if (coste[n] + ady.second < coste[ady.first]) {
                    coste[ady.first] = coste[n] + ady.second;
                    pred[ady.first] = n;
                            salir = false; // Modificado
                }}}} // 4
                if (salir) break;
        }

    for (int n = 0; n < AdjList.size(); n++) {
        if (coste[n] != MAX_VALUE >> 1) {
            for (auto ady : AdjList[n]) {
                if (coste[n] + ady.second < coste[ady.first]) {
                    // Si se cumple para algun vertice hay al menos un ciclo negativo
                }}}} // 4
    vector<vector<int>> res = {coste, pred};
    return res;
}
```

### 2.2.4. Algoritmo de Floyd-Warshall

Calcula la distancia mínima entre todos los nodos. El grafo es ponderado y puede tener aristas negativas pero no puede haber ciclos negativos.

```
#include <vector>
#define MAX_VALUE 1e9

vector<vector<int>> floydWarshall() {
    vector<vector<int>> costeMin(AdjList.size(), vector<int>(AdjList.size()));
    for (int i = 0; i < AdjList.size(); i++) {
        for (int j = 0; j < AdjList.size(); j++) {
            costeMin[i][j] = MAX_VALUE >> 1;
            for(int k = 0; k < AdjList[i].size(); k++)
                if(AdjList[i][k].first == j) {
                    costeMin[i][j] = AdjList[i][k].second; break; }
        }
    }
    for (int k = 0; k < AdjList.size(); k++)
        for (int i = 0; i < AdjList.size(); i++)
            for (int j = 0; j < AdjList.size(); j++)
                costeMin[i][j] = min(costeMin[i][j], (costeMin[i][k] + costeMin[k][j]));
    return costeMin;
}
```

### 2.2.5. Ordenacion topologica

```
#include <vector>
#include <stack>

std::vector<int> topologicalSort(const Grafo& g) {
    std::vector<int> ordenTopologico(g.numNodos);
    std::vector<int> gradosEntrada = g.gradosEntrada;
    int nodosVisitados = 0;
    int numAristas = g.numAristas;

    std::stack<int> pila;
    for (int n = 0; n < g.numNodos; n++) {
        if (gradosEntrada[n] == 0) {
            pila.push(n);
        }
    }

    while (!pila.empty()) {
        int actual = pila.top();
        pila.pop();
        ordenTopologico[nodosVisitados++] = actual;
        std::vector<Adyacente> adyacentes = g.adyacentes[actual];
        numAristas -= adyacentes.size();
        for (Adyacente ady : adyacentes) {
            if (--gradosEntrada[ady.dest] == 0) {
                pila.push(ady.dest);
            }
        }
    }

    if (numAristas > 0) {
        // Exisen ciclos
    }
    return ordenTopologico;
}
```

### 2.2.6. Algoritmo de Tarjan

```
#include <vector>
#include <stack>
#include <cmath>

typedef vector<int>& vi;
void strongConnect(int node, vi nodeIndex, vi lowlink, vi component,
                   int& index, int &components, Grafo& g, stack<int>& stk,
                   vector<bool>& stacked) {
    nodeIndex[node] = index;
    lowlink[node] = index;
    index++;

    stk.push(node);
    stacked[node] = true;

    for (Adyacente ady : g.adyacentes[node])
        if(nodeIndex[ady.dest] == 0) {
            strongConnect(ady.dest, nodeIndex, lowlink, component,
                          index, components, g, stk, stacked);
            lowlink[node] = std::min(lowlink[node], lowlink[ady.dest]);
        } else if (stacked[ady.dest])
            lowlink[node] = std::min(lowlink[node], nodeIndex[ady.dest]);

    if (nodeIndex[node] == lowlink[node]) {
        int w;
        components++;
        do {
            w = stk.top();
            stk.pop();
            stacked[w] = false;
            component[w] = components;
```

```
        } while (w != node);
    }

}

/*
 * Devuelve un vector con el id de la componente fuertemente conexa a la que
 * pertenece cada nodo.
 * Numero de componentes fuertemente conexas = valor maximo del vector resultado
 */
vector<int> tarjan(Grafo& g) {
    int index, components;
    vector<int> nodeIndex, lowlink, component;
    vector<bool> stacked;

    stack<int> stk;

    index = 1;
    components = 0;
    nodeIndex.resize(g.numNodos);
    lowlink.resize(g.numNodos);
    component.resize(g.numNodos);
    stacked.resize(g.numNodos);

    for (int i = 0; i < g.numNodos; i++)
        if (nodeIndex[i] == 0)
            strongConnect(i, nodeIndex, lowlink, component,
                          index, components, g, stk, stacked);

    return component;
}
```

## 2.3. Recorridos de grafos

### 2.3.1. BFS y DFS

```
#include <vector,queue,stack>
vector<int> bfs(int origen) {
    vector<bool> visitados(AdjList.size(), false);
    vector<int> ordenVisita;
    queue<int> colaBFS; //stack<int> pilaDFS;
    colaBFS.push(origen); //pilaDFS.push(origen);
    visitados[origen] = true;
    while(!/*pilaDFS*/colaBFS.empty() ) {
        int actual = colaBFS.front(); //int actual = pilaDFS.top();
        colaBFS.pop(); //pilaDFS.pop();
        ordenVisita.push_back(actual);
        for (auto ady : AdjList[actual]) {
            if (!visitados[ady.first]) {
                colaBFS.push(ady.first); //pilaDFS.push(ady.first);
                visitados[ady.first] = true;
            } } }
    return ordenVisita;
}
```

## 2.4. Árbol de recubrimiento de peso mínimo. Algoritmo de Kruskal.

```
// -- inside main, include <utility>
vector< pair<int, ii> > EdgeList;
EdgeList.push_back(make_pair(w, ii(u, v)));

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

int mst_cost = 0; // <-- result
UnionFind UF(V);
for (int i = 0; i < EdgeList.size(); i++){
    pair<int, ii> front = EdgeList[i];
    if (!UF.isSameSet(front.second.first, front.second.second)){
```

```
        mst_cost += front.first; // We can save chosen edge here
        UF.unionSet(front.second.first, front.second.second);
} }
```

## 2.5. Flujo máximo en un grafo

### 2.5.1. Algoritmo de Edmonds-Karp

```
#include <queue>
#include <vector>
#define MAX_VALUE 1e9
using std::vector;
using std::queue;
using std::min;

int bfsEdmondKarp(Grafo, int, int, vector<vector<int>>&, vector<int>&);
struct Enlace {
    int origen, dest, coste;
    Enlace(int origen, int dest, int coste) :
        origen(origen), dest(dest), coste(coste) {};
};

vector<Enlace> minCutResidualNetwork(Grafo g, Grafo r, int fuente, int sumidero);
/*
 * Calcula el flujo maximo de un grafo, si se quieren las aristas que forman
 * el minimo corte hay que cambiar el tipo de retorno y descomentar la parte
 * final del metodo
 */
/*vector<Enlace>*/ int EdmondsKarp(Grafo g, int fuente, int sumidero) {
  int maxFlow = 0;
  vector<int> parents;

  vector<vector<int>> flujo = vector<vector<int>>(g.numNodos, vector<int>(g.numNodos));
  while(true) {
    parents = vector<int>(g.numNodos);
    int m = bfsEdmondKarp(g, fuente, sumidero, flujo, parents);
    if (m == 0) break;
    maxFlow += m;
    int v = sumidero;
    while (v != fuente) {
      int u = parents[v];
      flujo[u][v] = flujo[u][v] + m;
      flujo[v][u] = flujo[v][u] - m;
      v = u;
    }
  }
  return maxFlow;

    /*
  Grafo residualNetwork(g.numNodos);
  for (unsigned i = 0; i < flujo.size(); i++) {
    for (unsigned j = 0; j < flujo.size(); j++) {
      if (g.existeArista(i, j) || flujo[i][j] < 0)
        residualNetwork.insertarArista(i, j, flujo[i][j]);
    }
  }
  return minCutResidualNetwork(g, residualNetwork, fuente, sumidero);
    */
}

/*
 * Metodo auxiliar para el calculo del flujo maximo
 */
int bfsEdmondKarp(Grafo g, int fuente, int sumidero,
                  vector<vector<int>>& flujo, vector<int>& parents) {
  vector<int> minCap = vector<int>(g.numNodos);
  int actual, destino;

  minCap[fuente] = MAX_VALUE;
  for (unsigned i = 0; i < minCap.size(); i++)
```

```
    parents[i] = -1;
  parents[fuente] = -2;

  queue<int> cola;
  cola.push(fuente);

  while (!cola.empty()) {
    actual = cola.front();
        cola.pop();
    vector<Adyacente> adyacentes = g.adyacentes[actual];
    for (unsigned i = 0; i < adyacentes.size(); i++) {
      destino = adyacentes.at(i).dest;
      if (
        adyacentes.at(i).coste - flujo[actual][destino] > 0 &&
        parents[destino] == -1
      ) {
        parents[destino] = actual;
        minCap[destino] = min(
          minCap[actual],
          adyacentes.at(i).coste - flujo[actual][destino]
        );
        if (destino == sumidero) return minCap[sumidero];
        cola.push(destino);
      }
    }

  }
  return 0;
}

/*
 * Metodo auxiliar que devuelve una lista de aristas de g que forman el minimo
 * corte en la red residual r obtenida con el flujo maximo de g desde la fuente
 * al sumidero
 */
vector<Enlace> minCutResidualNetwork(
  Grafo g, Grafo r, int fuente, int sumidero
) {
    vector<Enlace> minCut;
  vector<bool> visitados(g.numNodos);
  int actual;
  int destino;
  vector<Adyacente> adyacentes;

  queue<int> q;
  q.push(fuente);

  while (!q.empty()) { // Marcar los vertices
    actual = q.front();
        q.pop();
    visitados[actual] = true;

    adyacentes = r.adyacentes[actual];
    for (unsigned i = 0; i < adyacentes.size(); i++) {
      destino = adyacentes.at(i).dest;
      if ( !visitados[destino] && ((g.existeArista(actual, destino) &&
         adyacentes.at(i).coste < g.getArista(actual, destino)) ||
         !g.existeArista(actual, destino)))
        q.push(destino);
    }
  }

  // Anyadir las aristas que vayan de un vertice marcado a uno sin marcar
  for (unsigned i = 0; i < visitados.size(); i++) {
    if (visitados[i]) {
      adyacentes = g.adyacentes[i];
      for (unsigned j = 0; j < adyacentes.size(); j++) {
        if (!visitados[adyacentes.at(j).dest]) {
          minCut.push_back(Enlace(i, adyacentes.at(j).dest,
            adyacentes.at(j).coste));
        }
```

```
        }
      }
    }

    return minCut;
}
```

### 2.6. Emparejamiento de máximo/mínimo peso en un grafo bipartido

#### 2.6.1. Algoritmo de Kuhn-Munkres o algoritmo húngaro

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

#define TYPE long long
#define INF 2e15

class HungarianAlgorithm {
private:
  vector<std::vector<TYPE>> costMatrix;
  vector<TYPE> labelByWorker, labelByJob, minSlackValueByJob;
  vector<bool> committedWorkers;
  vector<int> minSlackWorkerByJob, parentWorkerByCommittedJob;
  vector<int> matchJobByWorker, matchWorkerByJob;
  int rows, cols, dim;
  void computeInitialFeasibleSolution() {
    for (int j = 0; j < dim; j++) labelByJob[j] = INF;
    for (int w = 0; w < dim; w++)
      for (int j = 0; j < dim; j++)
        if (costMatrix[w][j] < labelByJob[j]) labelByJob[j] = costMatrix[w][j];
  }
  void match(int w, int j) {
    matchJobByWorker[w] = j;
    matchWorkerByJob[j] = w;
  }
  void updateLabeling(double slack) {
    for (int w = 0; w < dim; w++)
      if (committedWorkers[w]) labelByWorker[w] += slack;
    for (int j = 0; j < dim; j++)
      if (parentWorkerByCommittedJob[j] != -1) labelByJob[j] -= slack;
      else minSlackValueByJob[j] -= slack;
  }
  void executePhase() {
    while (true) {
      int minSlackWorker = -1, minSlackJob = -1;
      TYPE minSlackValue = INF;
      for (int j = 0; j < dim; j++)
        if (parentWorkerByCommittedJob[j] == -1) {
          if (minSlackValueByJob[j] < minSlackValue) {
            minSlackValue = minSlackValueByJob[j];
            minSlackWorker = minSlackWorkerByJob[j];
            minSlackJob = j;
          }
          if (minSlackValue > 0) updateLabeling(minSlackValue);
          parentWorkerByCommittedJob[minSlackJob] = minSlackWorker;
          if (matchWorkerByJob[minSlackJob] == -1) {
            int committedJob = minSlackJob;
            int parentWorker = parentWorkerByCommittedJob[committedJob];
            while (true) {
              int temp = matchJobByWorker[parentWorker];
              match(parentWorker, committedJob);
              committedJob = temp;
              if (committedJob == -1) break;
              parentWorker = parentWorkerByCommittedJob[committedJob];
            }
            return;
          } else {
            int worker = matchWorkerByJob[minSlackJob];
            committedWorkers[worker] = true;
            for (int j = 0; j < dim; j++) {
```

```
            if (parentWorkerByCommittedJob[j] == -1) {
              TYPE slack = costMatrix[worker][j]
              - labelByWorker[worker] - labelByJob[j];
              if (minSlackValueByJob[j] > slack) {
                minSlackValueByJob[j] = slack;
                minSlackWorkerByJob[j] = worker;
              }}}}}}} //7
  int fetchUnmatchedWorker() {
    int w;
    for (w = 0; w < dim; w++) if (matchJobByWorker[w] == -1) break;
    return w;
  }
  void greedyMatch() {
    for (int w = 0; w < dim; w++)
      for (int j = 0; j < dim; j++)
        if (matchJobByWorker[w] == -1 && matchWorkerByJob[j] == -1
          && costMatrix[w][j] - labelByWorker[w] - labelByJob[j] == 0)
            match(w, j);
  }
  void initializePhase(int w) {
    for (int i = 0; i < dim; i++) {
      committedWorkers[i] = false;
      parentWorkerByCommittedJob[i] = -1;
    }
    committedWorkers[w] = true;
    for (int j = 0; j < dim; j++) {
      minSlackValueByJob[j] = costMatrix[w][j] - labelByWorker[w] - labelByJob[j];
      minSlackWorkerByJob[j] = w;
    }
  }
  void reduce() {
    for (int w = 0; w < dim; w++) {
      TYPE min = INF;
      for (int j = 0; j < dim; j++)
        if (costMatrix[w][j] < min) min = costMatrix[w][j];
      for (int j = 0; j < dim; j++) costMatrix[w][j] -= min;
    }
    vector<TYPE> min(dim, INF);
    for (int w = 0; w < dim; w++)
      for (int j = 0; j < dim; j++)
        if (costMatrix[w][j] < min[j]) min[j] = costMatrix[w][j];
    for (int w = 0; w < dim; w++)
      for (int j = 0; j < dim; j++) costMatrix[w][j] -= min[j];
  }
public:
  HungarianAlgorithm(const vector<vector<TYPE>>& costMatrix) {
    dim = max(costMatrix.size(), costMatrix[0].size());
    rows = costMatrix.size();
    cols = costMatrix[0].size();
    this->costMatrix.resize(dim);
    for (int r = 0; r < dim; r++)
      this->costMatrix[r].resize(dim);
      for (int c = 0; c < dim; c++)
        this->costMatrix[r][c] = costMatrix[r][c];
    labelByWorker.resize(dim);
    labelByJob.resize(dim);
    minSlackWorkerByJob.resize(dim);
    minSlackValueByJob.resize(dim);
    committedWorkers.resize(dim);
    parentWorkerByCommittedJob.resize(dim);
    matchJobByWorker.resize(dim);
    matchWorkerByJob.resize(dim);
    for (int i = 0; i < dim; i++) {
      matchJobByWorker[i] = -1;
      matchWorkerByJob[i] = -1;
    }
  }

  vector<int>  execute() {
    reduce();
    computeInitialFeasibleSolution();
```

```
    greedyMatch();
    int w = fetchUnmatchedWorker();
    while (w < dim) {
      initializePhase(w);
      executePhase();
      w = fetchUnmatchedWorker();
    }
    vector<int> result(matchJobByWorker);
    for (unsigned w = 0; w < result.size(); w++)
      if (result[w] >= cols) result[w] = -1;
    return result;
  }
};
```

# 3. Estructuras de datos

## 3.1. MFSets

```
struct MFSet {
    vector<int> p, rank, setSize;
    int numSets;

    MFSet(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 find(int i) { return (p[i] == i) ? i : (p[i] = find(p[i])); }
    void merge(int i, int j) {
        int x = find(i), y = find(j);
        if (x != y) { numSets--;
        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 sizeOfSet(int i) { return setSize[find(i)]; }
};
```

## 3.2. Prefix tree

```
#define ALPHABET_SIZE 26

struct node {
    int data;
    struct node* link[ALPHABET_SIZE];
};

struct node* root = NULL;

struct node* create_node() {
    struct node *q = new node();
    for(int x=0;x<ALPHABET_SIZE;x++)
        q->link[x] = NULL;
    q->data = -1;
    return q;
}

void insert_node(string key) {
    int length = key.length();
    int index;
    int level = 0;
    if(root == NULL)
        root = create_node();
    struct node *q = root;

    for(;level < length;level++) {
        index = key[level] - 'a';

        if(q->link[index] == NULL) {
            q->link[index] = create_node();
```

```
        }

        q = q->link[index];
    }
    q->data = level;
}

int search(string key) {
    struct node *q = root;
    int length = key.length();
    int level = 0;
    for(;level < length;level++) {
        int index = key[level]- 'a';
        if(q->link[index] != NULL)
            q = q->link[index];
        else
            break;
    }
    if(key[level] == '\0' && q->data != -1)
        return q->data;
    return -1;
}
```

## 3.3. Segment tree

```
typename vector<int> vi;

class SegmentTree {
private: vi st, A;
  int n;
  int left (int p) { return p << 1; }
  int right(int p) { return (p << 1) + 1; }

  void build(int p, int L, int R) {
    if (L == R) st[p] = L;
    else {
      build(left(p) , L, (L + R) / 2);
      build(right(p), (L + R) / 2 + 1, R);
      int p1 = st[left(p)], p2 = st[right(p)];
      st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  } }

  int rmq(int p, int L, int R, int i, int j) {
    if (i > R || j <  L) return -1;
    if (L >= i && R <= j) return st[p];

    int p1 = rmq(left(p) , L, (L+R) / 2, i, j);
    int p2 = rmq(right(p), (L+R) / 2 + 1, R, i, j);

    if (p1 == -1) return p2;
    if (p2 == -1) return p1;
    return (A[p1] <= A[p2]) ? p1 : p2; }

  int update_point(int p, int L, int R, int idx, int new_value) {
    int i = idx, j = idx;

    if (i > R || j < L) return st[p];

    if (L == i && R == j) {
      A[i] = new_value;
      return st[p] = L;
    }

    int p1, p2;
    p1 = update_point(left(p) , L, (L + R) / 2, idx, new_value);
    p2 = update_point(right(p), (L + R) / 2 + 1, R, idx, new_value);
    return st[p] = (A[p1] <= A[p2]) ? p1 : p2;
  }

public:
```

```
  SegmentTree(const vi &_A) {
    A = _A; n = (int)A.size();
    st.assign(4 * n, 0);
    build(1, 0, n - 1);
  }

  int rmq(int i, int j) { return rmq(1, 0, n - 1, i, j); }

  int update_point(int idx, int new_value) {
    return update_point(1, 0, n - 1, idx, new_value); }
};
```

### 3.4. Binary Search Tree

```
#include <iostream>
#include <vector>
#include <queue>

template <class T>
class BSTNode {
public:
    T data;
    BSTNode<T> *leftChild, *rightChild, *parent;
    BSTNode(T data) {
        this->data = data;
        leftChild = nullptr; rightChild = nullptr; parent = nullptr;
    }
    ~BSTNode() {
        if (leftChild) delete leftChild;
        if (rightChild) delete rightChild;
    }
};
template <class T>
class BST {
public:
    BSTNode<T> *root;
    int size;
    BST() {
        root = nullptr;
        size = 0;
    }
    ~BST() {
        if (root) delete root;
    }
    void insert(const T& data) {
        BSTNode<T> *newNode = new BSTNode<T>(data);
        if (!root) {
            root = newNode;
        } else {
            BSTNode<T> *current = root;
            BSTNode<T> *parent = nullptr;
            while (current) {
                parent = current;
                if (current->data == data) return; //No permite duplicados
                current = (data < current->data) ? current->leftChild : current->rightChild;
            }
            if (data < parent->data) {
                parent->leftChild = newNode;
            } else {
                parent->rightChild = newNode;
            }
            newNode->parent = parent;
        }
        size++;
    }
    BSTNode<T>* search(const T& data) {
        BSTNode<T> *current = root;
        while (current) {
            if (current->data == data) return current;
            current = (data < current->data) ? current->leftChild : current->rightChild;
```

```
        }
        return nullptr;
    }
    BSTNode<T>* findMin(BSTNode<T> *node) {
        if (!node) return nullptr; //Subarbol vacio
        while (node->leftChild) node = node->leftChild;
        return node;
    }
    void remove(const T& data) {
        BSTNode<T>* node = search(data);
        if (!node) return; // data no existe
        if (node->leftChild && node->rightChild) {
            BSTNode<T>* minInRigth = findMin(node->rightChild);
            // Remove minInRight from the tree
            if (minInRigth->rightChild) {
                minInRigth->rightChild->parent = minInRigth->parent;
            }
            if (minInRigth->parent->leftChild == minInRigth) {
                minInRigth->parent->leftChild = minInRigth->rightChild;
            } else {
                minInRigth->parent->rightChild = minInRigth->rightChild;
            }
            node->data = minInRigth->data;
            minInRigth->leftChild = minInRigth->rightChild = nullptr;
            delete minInRigth;
        } else {
            BSTNode<T>* aux = (node->leftChild) ? node->leftChild : node->rightChild;
            BSTNode<T>* parent = node->parent;
            if (aux) aux->parent = parent;
            if (!parent) {
                root = aux;
            } else if (parent->leftChild == node) {
                parent->leftChild = aux;
            } else {
                parent->rightChild = aux;
            }
            node->leftChild = node->rightChild = nullptr;
            delete node;
        }
        size--;
    }
    bool isValidBST(BSTNode<T>* node, T minValue, T maxValue) {
        if (!node) return true;
        bool isLeftValid = true;
        bool isRightValid = true;
        if (node->data <= minValue || node->data >= maxValue) return false;
        if (node->leftChild) {
            isLeftValid = node->leftChild->data < node->data
            && isValidBST(node->leftChild, minValue, node->data);
        }
        if (node->rightChild) {
            isRightValid = (node->rightChild->data > node->data)
            && isValidBST(node->rightChild, node->data, maxValue);
        }
        return isLeftValid && isRightValid;
    }
    void print() {
        queue<BSTNode<T>*> q;
        q.push(root);
        while (!q.empty()) {
            BSTNode<T>* current = q.front();
            q.pop();
            if (!current) continue;
            cout << current->data << "(";
            if (current->leftChild) {
                cout << current->leftChild->data << ", ";
            } else {
                cout << "-" << ", ";
            }
            if (current->rightChild) {
                cout << current->rightChild->data << ")";
```

```
            } else {
                cout << "-" << ")";
            }
            cout << endl;
            q.push(current->leftChild);
            q.push(current->rightChild);
        }
        cout << endl << endl;
    }
};
```

### 3.4.1. Numero de secuencias que crean un BST

```
template <class T>
result countSequences(BSTNode<T>* node) {
    int leftChilds = 0;
    int leftCount = 1;
    if (node->leftChild) {
        result leftRes = countSequences(node->leftChild);
        leftChilds = 1 + leftRes.childs;
        leftCount = leftRes.count;
    }
    int rightChilds = 0;
    int rightCount = 1;
    if (node->rightChild) {
        result rightRes = countSequences(node->rightChild);
        rightChilds = 1 + rightRes.childs;
        rightCount = rightRes.count;
    }
    int totalChilds = leftChilds + rightChilds;
    result ret;
    ret.count = leftCount * rightCount * bc2[totalChilds][leftChilds]; // binCoef Placeholder
    ret.childs = totalChilds;
    return ret;
}
```

# 4. Programación Dinámica

## 4.1. Subsecuéncia común máxima (LCS)

Calcula el tamaño de la LCS de l1 y l2 que tienen tamaños t1 y t2 respectivamente.

```
const int MAX = 1005;
int lcs = new int[MAX][MAX];
for ( int i = 0 ; i <= t1 ; i++ ) lcs[i][0] = 0;
for ( int i = 0 ; i <= t2 ; i++ ) lcs[0][i] = 0;

for ( int i = 1 ; i <= t1 ; i++ )
        for ( int j = 1 ; j <= t2 ; j++ )
                if ( l1[i-1] == l2[j-1] ) lcs[i][j] = lcs[i-1][j-1] + 1;
                else lcs[i][j] = max ( lcs[i-1][j] , lcs[i][j-1] );
```

## 4.2. Subsecuencia creciente/decreciente máxima (LIS/LDS)

```
const int MAX = 32800;
int s = new int[MAX];

int lds[n];
for ( int i = 0 ; i <= n ; i++ ) lcs[i] = 1;  //Coste == O(n^2)
for ( int i = 0 ; i <= n-1 ; i++ )
  for ( int j = i + 1 ; j <= n ; j++ )
    if ( s[j] < s[i] ) // Cambiar a > para LIS
      if ( lds[i] + 1 > lds[j] ) lds[j] = lds[i] + 1;
// Buscar maximo
int mx = 0;
for (int i = 1 ; i < n ; i++ ) if ( lds[mx] < lds[i] ) mx = i;
```

# 5. Algoritmos

## 5.1. Algoritmo del matrimonio estable

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

vector<int> stableMarriage(vector<vector<int>> important, vector<vector<int>> notImportant) {
    int N = important.size();
    vector<vector<bool>>  importantTriedNotImportant(N, vector<bool>(N));
    queue<int> freeImportant;
    vector<int> notImportantForImportant(N);
    vector<int> importantForNotImportant(N);
    for (int i = 0; i < N; i++) {
        freeImportant.push(i);
        notImportantForImportant[i] = -1;
        importantForNotImportant[i] = -1;
    }
    while (freeImportant.size() > 0) {
        int ie = freeImportant.front();
        freeImportant.pop();
        int nie = -1;
        for (auto pnie : important[ie])
            if (!importantTriedNotImportant[ie][pnie]) {
                nie = pnie;
                importantTriedNotImportant[ie][pnie] = true;
                break;
            }
        if (importantForNotImportant[nie] == -1) {
            notImportantForImportant[ie] = nie;
            importantForNotImportant[nie] = ie;
        }
        else {
            int pfnie = -1;
            int pfpie = -1;
            int pie = importantForNotImportant[nie];
            for (int i = 0; i < N; i++) {
                if (notImportant[nie][i] == ie)
                    pfnie = i;
                if (notImportant[nie][i] == pie)
                    pfpie = i;
            }
            if (pfnie < pfpie) {
                notImportantForImportant[pie] = -1;
                notImportantForImportant[ie] = nie;
                importantForNotImportant[nie] = ie;
                freeImportant.push(pie);
            } else {
                freeImportant.push(ie);
            }
        }
    }
    return notImportantForImportant;
}
```

## 5.2. Algoritmo de la mochila

```
int solveKnapsack(vector<int>& w, vector<int>& v, int W) {
  vector<int>* prev = new vector<int>(W + 1, 0);
  vector<int>* next = new vector<int>(W + 1, 0);
  vector<int>* exchange = NULL;

  for (unsigned i = 0; i < v.size(); i++) {
    for (int j = w[i]; j <= W; j++)
        (*next)[j] = max((*prev)[j], (*prev)[j - w[i]] + v[i]);
    exchange = prev; prev = next; next = exchange;
  }
  int result = (*prev)[W];
  delete prev; delete next;
```

```
    return result;
}
```

## 5.3. Algoritmo KMP (buscar subcadena en cadena) y Boyer-Moore

```
#include <vector>
#include <string>

//Boyer-Moore-Horspool
//Busca el patron pattern dentro del texto text
//Es mas eficiente que KMP para alfabetos y/o patrones grandes
class BMH {
private:
    static vector<int> preBoyerMooreHorspool(string P, string T, int size) {
        int pLength = P.length();
        int last = P.length() - 1;
        vector<int> R(size);
        for (int i = 0; i < size; i++) R[i] = pLength;
        for (int i = 0; i < last; i++) R[P[i]] = last - i;
        return R;
    }
public:
    static int boyerMooreHorspool(string pattern, string text) {
        string P = pattern;
        string T = text;
        int offset = 0;
        int scan = 0;
        int last = P.length() - 1;
        int maxoffset = T.length() - P.length();
        vector<int> R = preBoyerMooreHorspool(P, T, 256);
        // aun hay suficientes caracteres para comparar
        while (offset <= maxoffset) {
            // comparar de derecha a izquierda
            for (scan = last; P[scan] == T[scan + offset]; scan--)
                if(scan == 0) return offset; // encontrado
            offset += R[T[offset + last]]; // alinear el patron
        }
        return text.length(); // no encontrado
    }
};


//Knuth-Morris-Pratt
//Busca el patron pat dentro del texto txt
class KMP {
private:
    const int R;        // the radix
    vector<vector<int>> dfa; // the KMP automoton
    string pat;         // or the pattern string

public:
    KMP(string pat) : R(256) {
        this->pat = pat;
        int M = pat.size();
        dfa.resize(R, vector<int>(M));
        dfa[pat[0]][0] = 1;
        for (int X = 0, j = 1; j < M; j++) {
            for (int c = 0; c < R; c++)
                dfa[c][j] = dfa[c][X];      // Copy mismatch cases.
            dfa[pat[j]][j] = j+1;   // Set match case.
            X = dfa[pat[j]][X];     // Update restart state.
        }
    }

    // return offset of first match; N if no match
    int search(string txt) {
        int M = pat.size();
        int N = txt.size();
        int i, j;
        for (i = 0, j = 0; i < N && j < M; i++)
            j = dfa[txt[i]][j];
```

```
        if (j == M) return i - M;    // found
        return N;                    // not found
    }
};
```

# 6. Geometría

## 6.1. Clases base

```
#include <cmath>
class Point2D {
public:
  double x;
  double y;
  Point2D(double x, double y) {
    if (x == 0.0) x = 0.0;  // convert -0.0 to +0.0
    if (y == 0.0) y = 0.0;  // convert -0.0 to +0.0
    this->x = x;
    this->y = y;
  }
  // Returns the polar radius of this point
  double r() { return sqrt(x*x + y*y); }
  // Returns the angle of this point in polar coordinates
  double theta() { return atan2(y, x); }
  // Returns the angle between this point and that point
  double angleTo(const Point2D& that) const {
    double dx = that.x - this->x;
    double dy = that.y - this->y;
    return atan2(dy, dx);
  }
  double angle(Point2D p, Point2D q) {
    Point2D vec1(p.x - this->x, p.y - this->y);
    Point2D vec2(q.x - this->x, q.y - this->y);
    double dot = vec1.x*vec2.x + vec1.y*vec2.y;
    double mod1 = distanceTo(p);
    double mod2 = distanceTo(q);
    return acos(dot / (mod1*mod2));
  }
  double arc(Point2D p, Point2D q) {
    double angle = this->angle(p, q);
    return this->distanceTo(p)*angle;
  }
  /**
   * Is a->b->c a counterclockwise turn?
   * @return { -1, 0, +1 } if a->b->c is a { clockwise, collinear; counterclocwise }
   * turn.
   */
  static int ccw(Point2D a, Point2D b, Point2D c) {
    double area2 = (b.x-a.x)*(c.y-a.y) - (b.y-a.y)*(c.x-a.x);
    if      (area2 < 0) return -1;
    else if (area2 > 0) return +1;
    else                return  0;
  }
  // Returns twice the signed area of the triangle a-b-c
  static double area2(Point2D a, Point2D b, Point2D c) {
    return (b.x-a.x)*(c.y-a.y) - (b.y-a.y)*(c.x-a.x);
  }
  double distanceTo(const Point2D& that) const {
        return sqrt(this->distanceSquaredTo(that));
    }
    double distanceSquaredTo(const Point2D& that) const {
        double dx = this->x - that.x;
        double dy = this->y - that.y;
        return dx*dx + dy*dy;
    }
    /**
     * Compares this point to that point by y-coordinate, breaking ties by x-coordinate
     */
    bool operator<(const Point2D& that) const {
```

```
        if (this->y < that.y) return false;
        if (this->y > that.y) return true;
        if (this->x < that.x) return false;
        return true;
    }
    // compare other points relative to atan2 angle (bewteen -pi/2 and pi/2)they
    // make with this Point
    bool atan2Order(const Point2D& q1, const Point2D& q2) const {
        double angle1 = angleTo(q1);
        double angle2 = angleTo(q2);
        if (angle1 < angle2) return false;
        return true;
    }
    // compare other points relative to polar angle (between 0 and 2pi) they make
    // with this Point
    bool polarOrder(const Point2D& q1, const Point2D& q2) const {
        double dx1 = q1.x - x;
        double dy1 = q1.y - y;
        double dx2 = q2.x - x;
        double dy2 = q2.y - y;
        if      (dy1 >= 0 && dy2 < 0) return false;    // q1 above; q2 below
        else if (dy2 >= 0 && dy1 < 0) return true;    // q1 below; q2 above
        else if (dy1 == 0 && dy2 == 0) {             // 3-collinear and horizontal
            if      (dx1 >= 0 && dx2 < 0) return false;
            return true;
        }
        else return -ccw(*this, q1, q2) > 0;      // both above or below
        // Note: ccw() recomputes dx1, dy1, dx2, and dy2
    }
    // compare points according to their distance to this point
    bool distanceToOrder(const Point2D& p, const Point2D& q) const {
        double dist1 = distanceSquaredTo(p);
        double dist2 = distanceSquaredTo(q);
        if      (dist1 < dist2) return -1;
        else if (dist1 > dist2) return +1;
        else                    return  0;
    }
    static Point2D getMidpoint(Point2D p, Point2D q) {
        return Point2D((q.x + p.x)/2, (q.y + p.y)/2);
    }
    // Useful to check for points at infinity
    bool equals(Point2D p) {
        return this->x == p.x &&
            this->y == p.y;
    }
};
```

```
#include <limits>
#include "geo_Point2D.cpp"
class Line {
    public:
    double A, B, C;
    Line(double A, double B, double C) {
        this->A = A;
        this->B = B;
        this->C = C;
    }
    // Line containing p and q
    Line(Point2D p, Point2D q) {
        this->A = q.y - p.y;
        this->B = p.x - q.x;
        this->C = this->A*p.x + this->B*p.y;
    }
    // "Mediatriz"
    Line perpBisector(Point2D p, Point2D q) {
        Point2D mid = Point2D::getMidpoint(p, q);
        double D = -this->B*mid.x + this->A*mid.y;
        return Line(-this->B, this->A, D);
    }
    Point2D intersect(Line line) {
        double det = this->A*line.B - line.A*B;
```

```
        if(det == 0) return Point2D(std::numeric_limits<float>::infinity(),
                                    std::numeric_limits<float>::infinity()); // Parallel lines
        else {
            return Point2D((line.B * this->C - this->B * line.C)/det,
                           (this->A * line.C - line.A * this->C)/det);
        }
    }
};
```

Vector2D es una clase **inmutable**. Esto quiere decir que todos sus métodos devolverán un vector nuevo.

```
class Vector2D {
private: double x, y;
public:
    Vector2D(double x, double y) : x(x), y(y) {}
    double getX() { return this->x; }
    double getY() { return this->y; }
    Vector2D add(Vector2D other) {
        return Vector2D(this->x + other.getX(), this->y + other.getY());  }
    Vector2D subtract(Vector2D other){
        return Vector2D(this->x - other.getX(), this->y - other.getY());  }
    Vector2D unit(){  return this->scale(1 / this->length());  }
    Vector2D scale(double k){  return Vector2D(this->x * k, this->y * k);  }
    double length(){  return sqrt(this->lengthSquared());  }
    double lengthSquared(){  return this->dotProduct(*this);  }
    double dotProduct(Vector2D other){return this->x * other.getX() + this->y * other.getY();}
    Vector2D rotate90Right(){  return Vector2D(this->y, -this->x);  }
    void toString(){  printf("(%.2f, %.2f)", this->x, this->y);  }
};
```

## 6.2. Convex Hull

```
class Point    {
public:
    float x, y;
    bool operator < (Point b) {
        if (y != b.y) return y < b.y;
        return x < b.x;
    }
};

Point pivot;

int ccw(Point a, Point b, Point c) {
    float area = (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
    if (area > 0) return -1;
    else if (area < 0) return 1;
    return 0;
}

float sqrDist(Point a, Point b){
    float dx = a.x - b.x, dy = a.y - b.y;
    return dx * dx + dy * dy;
}

bool POLAR_ORDER(Point a, Point b){
    float order = ccw(pivot, a, b);
    if (order == 0)
        return sqrDist(pivot, a) < sqrDist(pivot, b);
    return (order == -1);
}

stack<Point> grahamScan(vector<Point>& points){
    stack<Point> hull;
    if (points.size() < 3) return hull;
    int leastY = 0;
    for (unsigned i = 1; i < points.size(); i++)
        if (points[i] < points[leastY]) leastY = i;

    Point temp = points[0];
```

```
    points[0] = points[leastY];
    points[leastY] = temp;
    pivot = points[0];
    sort(points.begin() + 1, points.end(), POLAR_ORDER);

    hull.push(points[0]);
    hull.push(points[1]);
    hull.push(points[2]);

    for (unsigned i = 3; i < points.size(); i++){
        Point top = hull.top();
        hull.pop();
        /* Lo siguiente puede fallar cuando hay puntos colineales.

           Se puede modificar con ccw(...) == 1 para obtener un convex
           hull con posibles vertices de 180 grados. */
        while (ccw(hull.top(), top, points[i]) != -1) {
            top = hull.top();
            hull.pop();
        }
        hull.push(top);
        hull.push(points[i]);
    }
    return hull;
}
```

## 6.3. Funciones

### 6.3.1. Area de un Poligono

Recive un vector con los vertices del poligono. Si se quita el fabs se puede utilizar para saber si los vertices estan ordenados clockwise o counter-clockwise segun el signo del area.

```
#include <vector>
#include <cmath>
double area(vector<Point> v) {
  double area = 0.0;
  for(unsigned int i = 0; i < v.size(); i++){
    int j = (i+1) % v.size();
    area += v[i].x * v[j].y - v[i].y * v[j].x;
  }                          // Negativo si CW
  return fabs(area / 2.0);} // Positivo si CCW
```

### 6.3.2. Puntos enteros sobre un segmento

Calcula el numero de puntos con ambas coordenadas enteras que estan en el segmento formado por dos puntos. Estos puntos tienen que tener las coordenadas enteras. El resultado no incluye a los puntos extremos (alg([0,0], [0,1])=0)

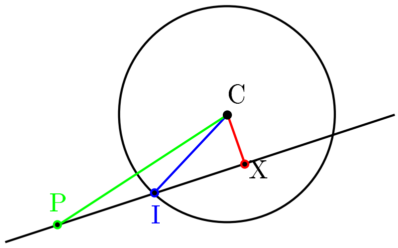
```
#include <cstdlib>
long long points_line(Point x1, Point x2) {
    return abs(gcd(x2.x - x1.x, x2.y - x1.y))-1;
}
```

## 6.4. Propiedades

### 6.4.1. Teorema de Pick:

Sea un polígono simple cuyos vértices tienen coordenadas enteras. Si B es el número de puntos enteros en el borde, I el número de puntos enteros en el interior del polígono, entonces el área A del polígono se puede calcular con la fórmula: $A = I + \frac{B}{2} - 1$ O lo que es lo mismo, el numero de puntos enteros interiores al poligono se puede calcular con: $I = \frac{2*A-B+2}{2}$

### 6.4.2. Intersección entre línea y círculo:




Empezamos con una línea representada por un punto P y un vector director unitario $\vec{v}$ (no representado, es equivalente a $\frac{\overrightarrow{PX}}{|\overrightarrow{PX}|}$); y con un círculo representado por el punto $C$ (centro) y su radio $r$.
Primero obtenemos X, el punto de la recta más cercano al centro del círculo: proyectamos $\overrightarrow{PC}$ sobre $\vec{v}$. Esta proyección ($\overrightarrow{PX}$) se obtiene de la siguiente forma: $|\overrightarrow{PX}| = \overrightarrow{PC} \cdot \vec{v}$, $\overrightarrow{PX} = \vec{v} \cdot |\overrightarrow{PX}|$. $\cdot$ representa el producto escalar entre vectores (salvo en el segundo caso, que representa el escalado de un vector por una constante).
La distancia entre la recta y $C$ es $|\overrightarrow{CX}|$ (en adelante $d$). Si $d = 0$ la recta pasa por el centro del círculo. Si $d < r$ la recta corta al círculo en dos puntos. Si $d = r$ la recta es tangente al círculo (lo corta en un punto). Si $d > r$ la recta no interseca el círculo.
$\widehat{PXC}$ es siempre un ángulo recto. Como el triángulo IXC es rectángulo, obtenemos el módulo de $\overrightarrow{IX}$ mediante el teorema de Pitágoras. $|\overrightarrow{IC}| = r$ (el radio del círculo). Si a X le restamos $\vec{v} \cdot |\overrightarrow{IX}|$ obtenemos I (la intersección). Si se lo sumamos, obtenemos el otro punto de intersección.

## 7. American Keyboard Layout