Algoritmos de referencia para competencias ICPC (C++)

Grupo de Estudio en Programación Competitiva
Semillero de Investigación en Linux y Software libre SILUX
Gerson Lázaro - Melissa Delgado

26 de junio de 2017

Índice

1. Bonus: Input Output

1.1. scanf y printf

```
#include <cstdio>

scanf("%d",&value); //int
scanf("%ld",&value); //long y long int
scanf("%c",&value); //char
scanf("%f",&value); //float
scanf("%lf",&value); //double
scanf("%s",&value); //char*
scanf("%lld",&value); //long long int
scanf("%x",&value); //int hexadecimal
scanf("%o",&value); //int octal
```

2. Data Structures

2.1. Disjoint Set

```
Estructura de datos para modelar una colección de conjuntos disyuntos.
    Permite determinar de manera eficiente a que conjunto pertenece un
    elemento, si dos elementos se encuentran en un mismo conjunto y unir
    dos conjuntos disyuntos en un conjunto mayor.

const int MAX = 10001; //Cantidad máxima de conjuntos disyuntos
int parent[MAX]; //estructura de DS
int size[MAX]; //Estructura para almacenar el tamaño de los conjuntos.
int cantSets; //Cantidad de conjuntos disyuntos existentes

/* Recibe la cantidad de conjuntos disyuntos iniciales */
void init( int n ){
    cantSets = n;
    for( int i = 0; i <= n; i++ ){
        parent[i] = i;
        size[i] = 1;
    }
}

int find(int i){
  parent[i] = ( parent[i] == i ) ? i : find(parent[i]);
  return parent[i];
}

void unionFind(int x, int y){
  x = find(x);
  y = find(y);

  if( x != y ){
      cantSets--;
      parent[x] = y;
      size[y] += size[x];
  }
}

int sizeOfSet( int i ){
    return size[ find(i) ];
}
```

2.2. RMQ

```
Range minimum query. Recibe como parametro en el constructor un array de
    valores. Las consultas se realizan con el método rmq(indice_inicio,
    indice_final) y pueden actualizarse los valores con
    update_point(indice, nuevo_valor)


class SegmentTree {
private: vector<int> 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 vector<int> &_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); }
};

int main() {
  int arr[] = { 18, 17, 13, 19, 15, 11, 20 };
  vector<int> A(arr, arr + 7);
  SegmentTree st(A);

  return 0;
}
```

---

3. Dynamic Programming

3.1. Knapsack

```
Dados N articulos, cada uno con su propio valor y peso y un tamaño maximo
    de una mochila, se debe calcular el valor maximo de los elementos que
    es posible llevar.
Debe seleccionarse un subconjunto de objetos, de tal manera que quepan en
    la mochila y representen el mayor valor posible.

#include <algorithm>

const int MAX_WEIGHT = 40;//Peso maximo de la mochila
const int MAX_N = 1000; //Numero maximo de objetos
int N;//Numero de objetos
int prices[MAX_N];//precios de cada producto
int weights[MAX_N];//pesos de cada producto
int memo[MAX_N][MAX_WEIGHT];//tabla dp

//El metodo debe llamarse con 0 en el id, y la capacidad de la mochila en
    w
int knapsack(int id, int w) {
  if (id == N || w == 0) return 0;
  if (memo[id][w] != -1) return memo[id][w];
  if (weights[id] > w) memo[id][w] = knapsack(id + 1, w);
  else memo[id][w] = max(knapsack(id + 1, w), prices[id] + knapsack(id +
      1, w - weights[id]));
  return memo[id][w];
}
```

```
//La tabla memo debe iniciar en -1
memset(memo, -1, sizeof(memo[0][0]) * MAX_N * MAX_WEIGHT);
```

3.2. Longest Common Subsequence

```
Dados dos Strings, encuentra el largo de la subsecuencia en común mas
    larga entre ellas.

const int M_MAX = 20; // Máximo size del String 1
const int N_MAX = 20; // Máximo size del String 2
int m, n; // Size de Strings 1 y 2
string X; // String 1
string Y; // String 2
int memo[M_MAX + 1][N_MAX + 1];

int lcs (int m, int n) {
  for (int i = 0; i <= m; i++) {
    for (int j = 0; j <= n; j++) {
      if (i == 0 || j == 0) memo[i][j] = 0;
      else if (X[i - 1] == Y[j - 1]) memo[i][j] = memo[i - 1][j - 1] + 1;
      else memo[i][j] = max(memo[i - 1][j], memo[i][j - 1]);
    }
  }
  return memo[m][n];
}
```

3.3. Longest Increasing Subsequence

```
Halla la longitud de la subsecuencia creciente mas larga. MAX debe
    definirse en el tamaño limite del array, n es el tamaño del array.
    Puede aplicarse también sobre strings, cambiando el parametro int s[]
    por string s. Si debe ser estrictamente creciente, cambiar el <= de
    s[j] <= s[i] por <

const int MAX = 1005;
int memo[MAX];

int longestIncreasingSubsequence(int s[], int n){
	memo[0] = 1;
	int output = 1;
	for (int i = 1; i < n; i++){
		memo[i] = 1;
		for (int j = 0; j < i; j++){
			if (s[j] <= s[i] && memo[i] < memo[j] + 1){
				memo[i] = memo[j] + 1;
			}
		}
		output = max(output, memo[i]);
	}
	return output;
}
```

3.4. Max Range Sum

```
Dada una lista de enteros, retorna la máxima suma de un rango de la lista.

#include <algorithm>

int maxRangeSum(vector<int> a){
	int sum = 0, ans = 0;
	for (int i = 0; i < a.size(); i++){
		if (sum + a[i] >= 0) {
			sum += a[i];
		  ans = max(ans, sum);
	   } else sum = 0;
	}
	return ans;
}
```

4. Geometry

4.1. Angle

```
Dados 3 puntos A, B, y C, determina el valor del angulo ABC (origen en B)
    en radianes. IMPORTANTE: Definir la estructura point y vec (Geometric
    Vector). Si se desea convertir a grados sexagesimales, revisar
    Sexagesimal degrees and radians.

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

```
double angle(point a, point b, point c) {
  vec ba = toVector(b, a);
  vec bc = toVector(b, c);
  return acos((ba.x * bc.x + ba.y * bc.y) / sqrt((ba.x * ba.x + ba.y *
      ba.y) * (bc.x * bc.x + bc.y * bc.y)));
}
```

4.2. Area

```
Calcula el area de un polígono representado como un vector de puntos.
    IMPORTANTE: Definir P[0] = P[n-1] para cerrar el polígono. El
    algorítmo utiliza el metodo de determinante de la matriz de puntos de
    la figura. IMPORTANTE: Debe definirse previamente la estructura point.

#include <vector>
#include <cmath>

double area(vector<point> P) {
       double result = 0.0;
  for (int i = 0; i < P.size()-1; i++) {
       result += ((P[i].x * P[i + 1].y) - (P[i + 1].x * P[i].y));
  }
       return fabs(result) / 2.0;
}
```

4.3. Collinear Points

```
Determina si el punto r está en la misma linea que los puntos p y q.
    IMPORTANTE: Deben incluirse las estructuras point y vec.


double cross(vec a, vec b) {
       return a.x * b.y - a.y * b.x;
}
bool collinear(point p, point q, point r) {
       return fabs(cross(toVector(p, q), toVector(p, r))) < 1e-9;
}
```

4.4. Convex Hull

```
Retorna el polígono convexo mas pequeño que cubre (ya sea en el borde o
    en el interior) un set de puntos. Recibe un vector de puntos, y
    retorna un vector de puntos indicando el polígono resultante. Es
    necesario que esten definidos previamente:

Estructuras: point y vec
Métodos: collinear, euclideanDistance, ccw (de inPolygon) y angle.


#include <cmath>
#include <algorithm>
#include <vector>

point pivot;
bool angleCmp(point a, point b) {
       if (collinear(pivot, a, b)) return euclideanDistance(pivot, a) <
           euclideanDistance(pivot, b);
  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> convexHull(vector<point> P) {
  int i, j, n = P.size();
  if (n <= 3) {
       if (!(P[0] == P[n-1])) P.push_back(P[0]);
       return P;
       }
  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;
  }
  point temp = P[0]; P[0] = P[P0]; P[P0] = temp;
       pivot = P[0];
  sort(++P.begin(), P.end(), angleCmp);
       vector<point> S;
  S.push_back(P[n-1]);
  S.push_back(P[0]);
       S.push_back(P[1]);
  i = 2;
  while (i < n) {
       j = S.size()-1;
```

```
    if (ccw(S[j-1], S[j], P[i])) S.push_back(P[i++]);
    else S.pop_back();
  }
  return S;
}
```

4.5. Euclidean Distance

```
Halla la distancia euclideana de 2 puntos en dos dimensiones (x,y). Para
    usar el primer método, debe definirse previamente la estructura point

#include <cmath>

/*Trabajando con estructuras de tipo punto*/
double euclideanDistance(point p1, point p2) {
  return hypot(p1.x - p2.x, p1.y - p2.y);
}

/*Trabajando con los valores x y y de cada punto*/
double euclideanDistance(double x1, double y1, double x2, double y2){
  return hypot(x1 - x2, y1 - y2);
}
```

4.6. Geometric Vector

```
Dados dos puntos A y B, crea el vector A->B. IMPORTANTE: Debe definirse
    la estructura point. Es llamado vec para no confundirlo con el vector
    propio de c++.


struct vec {
       double x, y;
  vec(double _x, double _y) : x(_x), y(_y) {}
};

vec toVector(point a, point b) {
       return vec(b.x - a.x, b.y - a.y);
}
```

4.7. Perimeter

```
Calcula el perímetro de un polígono representado como un vector de
    puntos. IMPORTANTE: Definir P[0] = P[n-1] para cerrar el polígono. La
    estructura point debe estar definida, al igual que el método
    euclideanDistance.

#include <vector>

double perimeter(vector<point> P) {
       double result = 0.0;
  for (int i = 0; i < P.size()-1; i++){
       result += euclideanDistance(P[i], P[i+1]);
  }
  return result;
}
```

4.8. Point in Polygon

```
Determina si un punto pt se encuentra en el polígono P. Este polígono se
    define como un vector de puntos, donde el punto 0 y n-1 son el mismo.
    IMPORTANTE: Deben incluirse las estructuras point y vec, ademas del
    método angle, y el método cross que se encuentra en Collinear Points.

#include <cmath>

bool ccw(point p, point q, point r) {
  return cross(toVector(p, q), toVector(p, r)) > 0;
}

bool inPolygon(point pt, vector<point> P) {
       if (P.size() == 0) return false;
  double sum = 0;
  for (int i = 0; i < P.size()-1; i++) {
    if (ccw(pt, P[i], P[i+1])) sum += angle(P[i], pt, P[i+1]);
    else sum -= angle(P[i], pt, P[i+1]);
  }
  return fabs(fabs(sum) - 2*acos(-1.0)) < 1e-9;
}
```

4.9. Point

```
La estructura punto será la base sobre la cual se ejecuten otros
    algoritmos.

#include <cmath>

struct point {
	double x, y;
	point() { x = y = 0.0; }
	point(double _x, double _y) : x(_x), y(_y) {}
	bool operator == (point other) const {
		return (fabs(x - other.x) < 1e-9 && (fabs(y - other.y) <
		    1e-9));
	}
};
```

4.10. Sexagesimal degrees and radians

```
Conversiones de grados sexagesimales a radianes y viceversa.

#include <cmath>

double DegToRad(double d) {
	return d * acos(-1.0) / 180.0;
}

double RadToDeg(double r) {
	return r * 180.0 / acos(-1.0);
}
```

5. Graph

5.1. BFS

```
Algoritmo de búsqueda en anchura en grafos, recibe un nodo inicial s y
    visita todos los nodos alcanzables desde s. BFS también halla la
    distancia más corta entre el nodo inicial s y los demás nodos si
    todas las aristas tienen peso 1.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

int v, e; //vertices, arcos
const int MAX=100005; //Cantidad máxima de nodos del grafo
vector<int> ady[MAX]; //lista de Adyacencia del grafo
long long dist[MAX]; //Estructura auxiliar para almacenar la distancia a
    cada nodo.

/*Este método se llama con el indice del nodo desde el que se desea
    comenzar
el recorrido.*/
void bfs(int s){
  queue<int> q;
  q.push(s); //Inserto el nodo inicial
  dist[s] = 0;
  int actual, i, next;

  while( q.size() > 0 ){
    actual = q.front();
    q.pop();

    for( i = 0; i < ady[actual].size(); i++){
      next = ady[actual][i];
      if( dist[next] == -1 ){
        dist[next] = dist[actual] + 1;
        q.push(next);
      }
    }
  }

}
```

5.2. Bipartite Check

```
Algoritmo para la detección de grafos bipartitos. Modificación de BFS.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

int v, e; //vertices, arcos
const int MAX=100005; //Cantidad máxima de nodos del grafo
vector<int> ady[MAX]; //lista de Adyacencia del grafo
int color[MAX]; //Estructura auxiliar para almacenar la distancia a cada
    nodo.
bool bipartite;
```

```
/*Este método se llama con el indice del nodo desde el que se desea
    comenzar
el recorrido.*/
void bfs(int s){
  queue<int> q;
  q.push(s);
  color[s] = 0;
  int actual, i, next;

  while( q.size() > 0 ){
    actual = q.front();
    q.pop();

    for( i = 0; i < ady[actual].size(); i++){
      next = ady[actual][i];
      if( color[next] == -1 ){
        color[next] = 1 - color[actual];
        q.push(next);
      }else if( color[next] == color[actual] ){
        bipartite = false;
        return;
      }
    }
  }

}
```

5.3. DFS

```
Algoritmo de búsqueda en profundidad para grafos. Parte de un nodo
    inicial s visita a todos sus vecinos. DFS puede ser usado para contar
    la cantidad de componentes conexas en un grafo y puede ser modificado
    para que retorne información de los nodos dependiendo del problema.
    Permite hallar ciclos en un grafo.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

int v, e; //vertices, arcos
const int MAX=100005; //Cantidad máxima de nodos del grafo
vector<int> ady[MAX]; //lista de Adyacencia del grafo
bool marked[MAX]; //Estructura auxiliar para marcar los nodos ya visitados

/*Este método se llama con el indice del nodo desde el que se desea
    comenzar
el recorrido.*/
static void dfs(int s){
        marked[s] = 1;
        int i, next;

        for( i = 0; i < ady[s].size(); i++ ){
                next = ady[s][i];
                if( !marked[next] )   dfs(next);
        }
}
```

5.4. Dijkstra's Algorithm

```
Algoritmo que dado un grafo con pesos no negativos halla la ruta mínima
    entre un nodo inicial s y todos los demás nodos.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

#define Node pair<int,long long> //(Vertice adyacente, peso)

int v,e; //v = cantidad de nodos, e = cantidad de aristas
const int MAX = 100001; //Cantidad Máxima de Nodos
vector<Node> ady[MAX]; //Lista de Adyacencia del grafo
bool marked[MAX]; //Estructura auxiliar para marcar los nodos visitados
long long dist[MAX]; //Estructura auxiliar para llevar las distancias a
    cada nodo
int previous[MAX]; //Estructura auxiliar para almacenar las rutas

class cmp{
public:
  bool operator()(Node n1, Node n2){
    return (n1.second>n2.second);
};

//El método debe llamarse con el indice del nodo inicial.
void dijkstra( int s ){
    priority_queue< Node , vector<Node> , cmp > pq;
    pq.push( Node(s, 0) );
    dist[s] = 0;
    int actual, j, adjacent;
    long long weight;

    while( !pq.empty() ){
      actual = pq.top().first;
```

```
      pq.pop();

      if( !marked[actual] ){
        marked[actual] = 1;
        for( j = 0; j < ady[actual].size(); j++ ){
          adjacent = ady[actual][j].first;
          weight = ady[actual][j].second;
          if( !marked[adjacent] ){
            if( dist[adjacent] > dist[actual] + weight ){
              dist[adjacent] = dist[actual] + weight;
              previous[adjacent] = actual;
              pq.push(Node( adjacent, dist[adjacent] ));
            }
          }
        }
      }
    }

}

int main(){
  int origen, destino;
  dijkstra(origen);
  //Para imprimir la distancia más corta desde el nodo inicial al nodo
      destino
  dist[destino];
  //Para imprimir la ruta más corta se debe imprimir de manera recursiva
      la estructura previous.
}
```

5.5. Edge

Estructura Edge con su comparador. Usada en algoritmos como Kruskal y Puentes e Itmos.

```
struct Edge{

  int source, dest, weight;

  bool operator != (const Edge& rhs) const{
    if(rhs.source != source || rhs.dest != dest || rhs.weight != weight){
      return true;
    }
      return false;
    }

};

/* Comparador de Edges */
int cmp(const void* a, const void* b){
  struct Edge* a1 = (struct Edge*)a;
  struct Edge* b1 = (struct Edge*)b;
  return a1->weight > b1->weight;
}
```

5.6. Flood Fill

Dado un grafo implicito colorea y cuenta el tamaño de las componentes conexas. Normalmente usado en rejillas 2D.

```
//aka Coloring the connected components

const int tam = 1000; //Máximo tamaño de la rejilla
int dy[] = {1,1,0,-1,-1,-1, 0, 1}; //Estructura auxiliar para los
    desplazamientos (8 direcciones)
int dx[] = {0,1,1, 1, 0,-1,-1,-1}; //Estructura auxiliar para los
    desplazamientos (8 direcciones)
char grid[tam][tam]; //Matriz de caracteres
int X, Y; //Tamaño de la matriz

/*Este método debe ser llamado con las coordenadas x, y donde se inicia el
recorrido. c1 es el color que estoy buscando, c2 el color con el que se va
a pintar. Retorna el tamaño de la componente conexa*/
int floodfill(int y, int x, char c1, char c2) {
        if (y < 0 || y >= Y || x < 0 || x >= X) return 0;

        if (grid[y][x] != c1) return 0; // base case

        int ans = 1;
        grid[y][x] = c2; // se cambia el color para prevenir ciclos

        for (int i = 0; i < 8; i++)
                ans += floodfill(y + dy[i], x + dx[i], c1, c2);

        return ans;
}
```

5.7. Floyd Warshall

```
Algoritmo para grafos que halla la distancia mínima entre cualquier par
    de nodos. ady[i][j] guardará la distancia mínima entre el nodo i y el
    j.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

int v, e; //vertices, arcos
const int MAX = 505; //Cantidad máxima de nodos del grafo
int ady[505][505]; //Matriz de adyacencia del grafo

void floydWarshall(){
   int k, i ,j;

    for( k = 0; k < v; k++ ){
        for( i = 0; i < v; i++ ){
            for( j = 0; j < v; j++ ){
                ady[i][j] = min( ady[i][j], ( ady[i][k] + ady[k][j] ) );
            }
        }
    }
}
```

5.8. Init

```
Método para la limpieza de TODAS las estructuras de datos utilizadas en
    TODOS los algoritmos de grafos.
Copiar solo las necesarias, de acuerdo al algoritmo que se este
    utilizando.

#define INF 1000000000

/*Debe llamarse al iniciar cada caso de prueba luego de haber leido la
    cantidad de nodos v
Limpia todas las estructuras de datos.*/
void init() {
    long long max = LLONG_MAX;
    rta = 0; //Prim
    cont = dfsRoot = rootChildren = 0; //Puentes
    bridges.clear(); //Puentes
    topoSort.clear(); //Topological Sort
    loops = false; //Loop Check
    cantSCC = 0; //Tarjan
    bipartite = true; //Bipartite Check

    for( int j = 0; j <= v; j++ ) {
        dist[j] = -1; //Distancia a cada nodo (BFS)
        dist[j] = max; //Distancia a cada nodo (Dijkstra)
        ady[j].clear(); //Lista de Adyacencia
        marked[j] = 0; //Estructura auxiliar para marcar los nodos ya
            visitados
        previous[j] = -1; //Estructura auxiliar para almacenar las rutas
        parent[j] = j; //Estructura auxiliar para DS
        dfs_num[j] = -1;
        dfs_low[j] = 0;
        itsmos[j] = 0;
        color[j] = -1; //Bipartite Check

        for(j = 0; j < v; j++) ady[i][j] = INF; //Warshall
    }
}
```

5.9. Kruskal

```
Algoritmo para hallar el arbol cobertor mínimo de un grafo no dirigido y
    conexo. Utiliza la técnica de Union-Find(Conjuntos disjuntos) para
    detectar que aristas generan ciclos.
Requiere de la struct Edge.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

int v, e; //v = nodos, e = arcos
const int MAX = 10001; //Cantidad máxima de NODOS
const int MAXE = 10001; //Cantidad máxima de ARCOS
int parent[MAX]; //estructura de DS
Edge edges[MAXE]; //Lista de arcos del grafo
Edge answer[MAX]; //Lista de arcos del arbol cobertor mínimo

/*      Métodos Disjoint Set       */
int find(int i){
  parent[i] = ( parent[i] == i ) ? i : find(parent[i]);
  return parent[i];
}

void unionFind(int x, int y){
  parent[ find(x) ] = find(y);
}
```

```
/*El arbol cobertor mínimo del grafo queda almacenado en el
vector de arcos answer*/
void kruskall(){
  Edge actual;
  int aux = 0;
  int i = 0;
  int x, y;
  qsort( edges, e, sizeof(edges[0]), cmp);

  while(aux < v-1 && i < edges.size() ){
    actual = edges[i];
    x = find( actual.source );
    y = find( actual.dest );

    if(x != y){
      answer[aux] = actual;
      aux++;
      unionFind(x, y);
    }
    i++;
  }
}

int main(){
  int s, d, w;
  //Los arcos se inicializan así
  edges[i].source = s;
  edges[i].dest = d;
  edges[i].weight = w;

  kruskall();
}
```

5.10. LoopCheck

```
Determina si un Grafo DIRIGIDO tiene o no ciclos.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

const int MAX = 10010; //Cantidad maxima de nodos
int v; //Cantidad de Nodos del grafo
vector<int> ady[MAX]; //Estructura para almacenar el grafo
int dfs_num[MAX];
bool loops; //Bandera de ciclos en el grafo

/* DFS_NUM STATES
        2 - Explored
        3 - Visited
        -1 - Unvisited
*/

/*
Este metodo debe ser llamado desde un nodo inicial u.
Cortara su ejecucion en el momento que encuentre algun ciclo en el grafo.
*/
void graphCheck( int u ){
        int j, next;

        if( loops ) return;

        dfs_num[u] = 2;

        for(j = 0; j < ady[u].size(); j++ ){
                next = ady[u][j];

                if( dfs_num[next] == -1 )    graphCheck( next );
                else if( dfs_num[next] == 2 ){
                        loops = true;
                        break;
                }
        }

        dfs_num[u] = 3;
}


int main(){
        for( int s = 1; s <= v && !loops; s++ ){ //Por si el grafo es NO
            conexo
                if( dfs_num[s] == -1 ) graphCheck(s);
        }
}
```

5.11. Maxflow

```
Dado un grafo, halla el máximo flujo entre una fuente s y un sumidero t.
```

SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```
vector<int> ady [105];
int capacity [105] [105]; //Capacidad de aristas de la red
int flow [105] [105]; //Flujo de cada arista
int previous [105];

void connect(int i, int j, int cap){
    ady[i].push_back(j);
    ady[j].push_back(i);
    capacity[i][j] += cap;
    //Si el grafo es dirigido no hacer esta linea
    //capacity[j][i]+=cap;
}

int maxflow(int s, int t, int n){ //s=fuente, t=sumidero, n=numero de
    nodos
    int i, j, maxFlow, u, v, extra, start, end;
    for( i = 0; i <= n; i++ ){
        for( j = 0; j <= n; j++ ){
            flow[i][j]=0;
        }
    }

    maxFlow = 0;

    while( true ){
        for( i = 0; i <= n; i++ ) previous[i] = -1;

        queue<int> q;
        q.push(s);
        previous[s] = -2;

        while( q.size() > 0 ){
            u = q.front();
            q.pop();
            if( u == t ) break;
            for( j = 0; j < ady[u].size(); j++){
                v = ady[u][j];
                if( previous[v] == -1 && capacity[u][v] - flow[u][v] > 0 ){
                    q.push(v);
                    previous[v] = u;
                }
            }
        }
        if( previous[t] == -1 ) break;

        extra = 1 << 30;
        end = t;
        while( end != s){
            start = previous[end];
            extra = min( extra, capacity[start][end]-flow[start][end] );
            end = start;
        }

        end = t;
        while( end != s){
            start = previous[end];
            flow[start][end] += extra;
            flow[end][start] = -flow[start][end];
            end = start;
        }

        maxFlow += extra;
    }

    return maxFlow;
}

int main(){
    //Para cada arista
    connect( s, d, f); //origen, destino, flujo
}
```

5.12. Prim

Algoritmo para hallar el arbol cobertor mínimo de un grafo no dirigido y conexo.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

```
#define Node pair < int, long long > //(Vertice adyacente, peso)

int v, e; //vertices, arcos
const int MAX = 100005;
vector < Node > ady[MAX];
bool marked[MAX];
int rta;
```

```
class cmp {
  public:
    bool operator()(Node n1, Node n2) {
      return (n1.second > n2.second);
};

static void prim() {
  priority_queue < Node, vector < Node > , cmp > pq;
  int u, w, i, v;

  marked[0] = true;
  for (i = 0; i < ady[0].size(); i++) {
    v = ady[0][i].first;
    if ( ! marked[v]) pq.add(Node(v, ady[u][i].second));
        }

        while ( ! pq.empty()) {
                u = pq.top().first;
                w = pq.top().second;

                pq.pop();

                if ( !marked[u]) {
                        rta += w;
                        marked[u] = true;
      for (i = 0; i < ady[u].size(); i++) {
        v = ady[u][i].first;
        if ( ! marked[v]) pq.add(Node(v, ady[u][i].second));
      }
                }
        }
}
```

5.13. Puentes itmos

```
Algoritmo para hallar los puentes e itsmos en un grafo no dirigido.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

vector<int> ady[1010];
int marked[1010];
int previous[1010];
int dfs_low[1010];
int dfs_num[1010];
bool itsmos[1010];
int n, e;
int dfsRoot,rootChildren,cont;
vector< pair<int,int> > bridges;

void dfs(int u){
    dfs_low[u] = dfs_num[u] = cont;
    cont++;
    marked[u] = 1;
    int j, v;

    for(j = 0; j < ady[u].size(); j++){
        v = ady[u][j];
        if( marked[v] == 0 ){
            previous[v] = u;
            //para el caso especial
            if( u == dfsRoot ) rootChildren++;
            dfs(v);

            //Itsmos
            if( dfs_low[v] >= dfs_num[u] ) itsmos[u] = 1;

            //Bridges
            if( dfs_low[v] > dfs_num[u] )
                bridges.push_back(make_pair(min(u,v),max(u,v)));

            dfs_low[u] = min(dfs_low[u], dfs_low[v]);
        }else if( v != previous[u] ) dfs_low[u] = min(dfs_low[u],
             dfs_num[v]);
    }
}

int main(){
    //Antes de ejecutar el Algoritmo
    cont = dfsRoot = rootChildren = 0;
    bridges.clear();
    dfs( dfsRoot );
    /* Caso especial */
    itmos[dfsRoot] = ( itmos[ dfsRoot ] == 1 && rootChildren > 1 ) ? 1 :
        0;
}
```

5.14. Tarjan

```
Algoritmo para hallar componentes fuertemente conexas(SCC) en grafos
    dirigidos.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

int v, e;
const int MAX = 5000; // Máxima cantidad de nodos
int dfs_low[MAX];
int dfs_num[MAX];
bool marked[MAX];
vector<int> s;
int dfsCont, cantSCC;
vector<int> ady[];

void tarjanSCC( int u ){
       dfs_low[u] = dfs_num[u] = dfsCont;
       dfsCont++;
       s.push_back(u);
       marked[u] = true;

       int j, v;

       for( j = 0; j < ady[u].size(); j++ ){
              v = ady[u][j] );

              if( dfs_num[v] == -1 ){
                     tarjanSCC( v );
              }

              if( marked[v] ){
                     dfs_low[u] = min( dfs_low[u], dfs_low[v] );
              }
       }

       if( dfs_low[u] == dfs_num[u] ){
              cantSCC++;
       /* ********************************************************* */
       /* Esta seccion se usa para imprimir las componentes conexas */
              cout << "COMPONENTE CONEXA #" << cantSCC << "\n";
              while( true ){
                     v = s.back();
                     s.pop_back();
                     marked[v] = false;
                     cout << v << "\n";
                     if( u == v ) break;
              }
       /* ********************************************************* */
       }
}

int main (){
   for( int i = 0; i < v; i++ ){ //Por si el grafo no es conexo
      if( dfs_num[i] == -1 ){
         dfsCont = 0;
         s.clear();
         tarjanSCC(i);
      }
   }
}
```

5.15. Topological Sort

```
Dado un grafo acíclico y dirigido, ordena los nodos linealmente de tal
    manera que si existe una arista entre los nodos u y v entonces u
    aparece antes que v.
Este ordenamiento es una manera de poner todos los nodos en una línea
    recta de tal manera que las aristas vayan de izquierda a derecha.
SE DEBEN LIMPIAR LAS ESTRUCTURAS DE DATOS ANTES DE UTILIZARSE

int v; //Cantidad de nodos del grafo
const int MAX=100005; //Cantidad máxima de nodos del grafo
vector<int> topoSort; //Ordenamiento topologico del grafo
vector<int> ady[MAX]; //Lista de adyacencia
bool marked[MAX]; //Estructura auxiliar para marcar los grafos visitados

//Recibe un nodo inicial u
void dfs( int u ){
       int i, v;
       marked[u] = 1;
       for( i = 0; i < ady[u].size(); i++){
              v = ady[u][i];
              if( !marked[v] ) dfs(v);
       }
       topoSort.push_back(u);
}

int main(){
       for(i=0; i<v; i++){
```

```
            if( !marked[i] )      dfs(i);
        }
        //imprimir topoSort en reversa :3
}
```

6. Math

6.1. Binary Exponentiation

```
Realiza a^b y retorna el resultado módulo c


long long binaryExponentiation(long long a, long long b, long long c){
  if (b == 0) return 1;
  if (b % 2 == 0) {
    long long temp = binaryExponentiation(a, b/2, c);
    return (temp * temp) % c;
  } else {
    long long temp = binaryExponentiation(a, b-1, c);
    return (temp * a) % c;
  }
}
```

6.2. Binomial Coefficient

```
Calcula el coeficiente binomial nCr, entendido como el número de
    subconjuntos de k elementos escogidos de un conjunto con n elementos.

long long binomialCoefficient(long long n, long long r) {
  if (r < 0 || n < r) return 0;
  r = min(r, n - r);
  long long ans = 1;
  for (int i = 1; i <= r; i++) {
    ans = ans * (n - i + 1) / i;
  }
  return ans;
}
```

6.3. Catalan Number

```
Guarda en el array Catalan Numbers los numeros de Catalan hasta MAX.

const int MAX = 30;
long long catalanNumbers[MAX+1];

void catalan(){
        catalanNumbers[0] = 1;
        for(int i = 1; i <= MAX; i++){
                catalanNumbers[i] = (long
                    long)(catalanNumbers[i-1]*((double)(2*((2 * i)- 1))/(i
                    + 1)));
        }
}
```

6.4. Euler Totient

```
Función totient o indicatriz de Euler. Para cada posición n del array
    result retorna el número de enteros positivos menores o iguales a n
    que son coprimos con n (Coprimos: MCD=1)

#include <string.h>

const int MAX = 100;
int result[MAX];

void totient () {
        bool temp[MAX];
        int i,j;
        memset(temp,1,sizeof(temp));
        for (i = 0; i < MAX; i++) {
                result[i] = i;
        }
        for (i = 2; i < MAX; i++){
                if (temp[i]) {
                        for (j = i; j < MAX ; j += i){
                                temp[j] = false;
                                result[j] = result[j] - (result[j]/i) ;
                        }
                        temp[i] = true ;
                }
        }
```

```
}
```

6.5. Gaussian Elimination

```
Resuelve sistemas de ecuaciones lineales por eliminación Gaussiana.
    matrix contiene los valores de la matriz cuadrada y result los
    resultados de las ecuaciones. Retorna un vector con el valor de las n
    incongnitas. Los resultados pueden necesitar redondeo.

#include <vector>
#include <algorithm>
#include <limits>
#include <cmath>

const int MAX = 100;
int n = 3;
double matrix[MAX][MAX];
double result[MAX];

vector<double> gauss() {
       vector<double> ans(n, 0);
  double temp;
       for (int i = 0; i < n; i++) {
    int pivot = i;
         for (int j = i + 1; j < n; j++) {
              temp = fabs(matrix[j][i]) - fabs(matrix[pivot][i]);
           if (temp > numeric_limits<double>::epsilon()) {
              pivot = j;
           }
         }
         swap(matrix[i], matrix[pivot]);
         swap(result[i], result[pivot]);
         if (!(fabs(matrix[i][i]) < numeric_limits<double>::epsilon())) {
           for (int k = i + 1; k < n; k++) {
                  temp = -matrix[k][i] / matrix[i][i];
                  matrix[k][i] = 0;
                  for (int l = i + 1; l < n; l++) {
                      matrix[k][l] += matrix[i][l] * temp;
                  }
                      result[k] += result[i] * temp;
                }
         }
  }
  for (int m = n - 1; m >= 0; m--) {
       temp = result[m];
       for (int i = n - 1; i > m; i--) {
              temp -= ans[i] * matrix[m][i];
       }
       ans[m] = temp / matrix[m][m];
  }
  return ans;
}
```

6.6. Greatest common divisor

```
Calcula el máximo común divisor entre a y b mediante el algoritmo de
    Euclides

int mcd (int a, int b) {
       while (b != 0){
              a %= b;
              swap(a, b);
       }
       return a;
}
```

6.7. Lowest Common Multiple

```
Calculo del mínimo común múltiplo usando el máximo común divisor.
    REQUIERE mcd(a,b)

int mcm (int a, int b) {
       return a * b / mcd(a, b);
}
```

6.8. Miller-Rabin

```
La función de Miller-Rabin determina si un número dado es o no un número
    primo. IMPORTANTE: Debe utilizarse el método binaryExponentiation y
    Modular Multiplication.

#include <cstdlib>
```

```
bool miller (long long p) {
  if (p < 2 || (p != 2 && p % 2 == 0)) return false;
  long long s = p - 1;
  while (s % 2 == 0) s /= 2;
  for (int i = 0; i < 5; i++){
    long long a = rand() % (p - 1) + 1;
    long long temp = s;
    long long mod = binaryExponentiation(a, temp, p);
    while (temp != p - 1 && mod != 1 && mod != p - 1){
      mod = mulmod(mod, mod, p);
      temp *= 2;
    }
    if (mod != p - 1 && temp % 2 == 0) return false;
  }
  return true;
}
```

6.9. Modular Multiplication

Realiza la operación (a * b) % mod minimizando posibles desbordamientos.

```
long long mulmod (long long a, long long b, long long mod) {
  long long x = 0;
  long long y = a % mod;
  while (b > 0){
    if (b % 2 == 1) x = (x + y) % mod;
    y = (y * 2) % mod;
    b /= 2;
  }
  return x % mod;
}
```

6.10. Pollard Rho

La función Rho de Pollard calcula un divisor no trivial de n. IMPORTANTE: Deben implementarse Modular Multiplication y Gratest Common Divisor (para long long).

```
long long pollardRho (long long n) {
  int i = 0, k = 2;
  long long d, x = 3, y = 3;
  while (true) {
    i++;
    x = (mulmod(x, x, n) + n - 1) % n;
    d = mcd(abs(y - x), n);
    if (d != 1 && d != n) return d;
    if (i == k) {
      y = x;
      k *= 2;
    }
  }
}
```

6.11. Prime Factorization

Guarda en primeFactors la lista de factores primos del value de menor a mayor. IMPORTANTE: Debe ejecutarse primero la criba de Eratostenes. La criba debe existir al menos hasta la raiz cuadrada de value (se recomienda dejar un poco de excedente).

```
#include <vector>

vector <long long> primeFactors;

void calculatePrimeFactors(long long value){
	primeFactors.clear();
	long long temp = value;
	int factor;
	for (int i = 0; (long long)primes[i] * primes[i] <= value; ++i){
		factor = primes[i];
		while (temp % factor == 0){
			primeFactors.push_back(factor);
			temp /= factor;
		}
	}
	if (temp != 1) primeFactors.push_back(temp);
}
```

6.12. Sieve of Eratosthenes

Guarda en primes los números primos menores o iguales a MAX

```
#include <vector>

const int MAX = 10000000;
vector<int> primes;
bool sieve[MAX+5];

void calculatePrimes() {
  sieve[0] = sieve[1] = 1;
  int i;
  for (i = 2; i * i <= MAX; i++) {
    if (!sieve[i]) {
      primes.push_back(i);
      for (int j = i * i; j <= MAX; j += i) sieve[j] = true;
    }
  }
  for(;i <= MAX; i++){
        if (!sieve[i]) {
      primes.push_back(i);
        }
  }
}
```

7. String

7.1. KMP's Algorithm

```
Encuentra si el string pattern se encuentra en el string cadena.

#include <vector>

vector<int> table(string pattern){
        int m=pattern.size();
        vector<int> border(m);
        border[0]=0;

        for(int i=1; i<m; ++i){
                border[i]=border[i-1];
                while(border[i]>0 && pattern[i]!=pattern[border[i]]){
                        border[i]=border[border[i]-1];
                }
                if(pattern[i] == pattern[border[i]]){
                        border[i]++;
                }
        }
        return border;
}

bool kmp(string cadena, string pattern){
        int n=cadena.size();
        int m=pattern.size();
        vector<int> tab=table(pattern);
        int seen=0;

        for(int i=0; i<n; i++){
                while(seen>0 && cadena[i]!=pattern[seen]){
                        seen=tab[seen-1];
                }
                if(cadena[i]==pattern[seen])
                        seen++;
                if(seen==m){
                        return true;
                }
        }
        return false;
}
```

8. Tips and formulas

8.1. ASCII Table

Caracteres ASCII con sus respectivos valores numéricos.

No.	ASCII	No.	ASCII
0	NUL	16	DLE
1	SOH	17	DC1
2	STX	18	DC2
3	ETX	19	DC3
4	EOT	20	DC4
5	ENQ	21	NAK
6	ACK	22	SYN
7	BEL	23	ETB
8	BS	24	CAN
9	TAB	25	EM

10	LF	26	SUB
11	VT	27	ESC
12	FF	28	FS
13	CR	29	GS
14	SO	30	RS
15	SI	31	US

No.	**ASCII**	**No.**	**ASCII**
32	(space)	48	0
33	!	49	1
34	”	50	2
35	#	51	3
36	$	52	4
37	%	53	5
38	&	54	6
39	’	55	7
40	(	56	8
41	)	57	9
42	*	58	:
43	+	59	;
44	,	60	¡
45	-	61	=
46	.	62	¿
47	/	63	?

No.	**ASCII**	**No.**	**ASCII**
64	@	80	P
65	A	81	Q
66	B	82	R
67	C	83	S
68	D	84	T
69	E	85	U
70	F	86	V
71	G	87	W
72	H	88	X
73	I	89	Y
74	J	90	Z
75	K	91	[
76	L	92	\
77	M	93	]
78	N	94	ˆ
79	O	95	_

No.	**ASCII**	**No.**	**ASCII**
96	‘	112	p
97	a	113	q
98	b	114	r
99	c	115	s
100	d	116	t
101	e	117	u
102	f	118	v
103	g	119	w
104	h	120	x
105	i	121	y
106	j	122	z
107	k	123	{
108	l	124	|
109	m	125	}
110	n	126	˜
111	o	127	

8.2. Formulas

PERMUTACIÓN Y COMBINACIÓN	
Combinación (Coeficiente Binomial)	Número de subconjuntos de k elementos escogidos de un conjunto con n elementos. $\binom{n}{k} = \binom{n}{n-k} = \frac{n!}{k!(n-k)!}$
Combinación con repetición	Número de grupos formados por n elementos, partiendo de m tipos de elementos. $CR_m^n = \binom{m+n-1}{n} = \frac{(m+n-1)!}{n!(m-1)!}$

Continúa en la siguiente columna

Permutación	Número de formas de agrupar n elementos, donde importa el orden y sin repetir elementos $P_n = n!$
Permutación múltiple	Elegir r elementos de n posibles con repetición n^r
Permutación con repetición	Se tienen n elementos donde el primer elemento se repite a veces , el segundo b veces , el tercero c veces, ... $PR_n^{a,b,c...} = \frac{P_n}{a!b!c!...}$
Permutaciones sin repetición	Núumero de formas de agrupar r elementos de n disponibles, sin repetir elementos $\frac{n!}{(n-r)!}$
DISTANCIAS	
Distancia Euclideana	$d_E(P_1, P_2) = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$
Distancia Manhattan	$d_M(P_1, P_2) = \|x_2 - x_1\| + \|y_2 - y_1\|$
CIRCUNFERENCIA Y CÍRCULO	
Considerando r como el radio, α como el ángulo del arco o sector, y (R, r) como radio mayor y menor respectivamente.	
Área	$A = \pi * r^2$
Longitud	$L = 2 * \pi * r$

Continúa en la siguiente columna

Longitud de un arco	$L = \frac{2 * \pi * r * \alpha}{360}$
Área sector circular	$A = \frac{\pi * r^2 * \alpha}{360}$
Área corona circular	$A = \pi(R^2 - r^2)$
TRIÁNGULO	
Considerando b como la longitud de la base, h como la altura, letras minúsculas como la longitud de los lados, letras mayúsculas como los ángulos, y r como el radio de círcunferencias asociadas.	
Área conociendo base y altura	$A = \frac{1}{2} b * h$
Área conociendo 2 lados y el ángulo que forman	$A = \frac{1}{2} b * a * sin(C)$
Área conociendo los 3 lados	$A = \sqrt{p(p-a)(p-b)(p-c)}$ con $p = \frac{a+b+c}{2}$
Área de un triángulo circunscrito a una circunferencia	$A = \frac{abc}{4r}$

Continúa en la siguiente columna

Área de un triángulo inscrito a una circunferencia	$A = r(\frac{a+b+c}{2})$
Área de un triangulo equilátero	$A = \frac{\sqrt{3}}{4}a^2$
RAZONES TRIGONOMÉTRICAS	
Considerando un triangulo rectángulo de lados a, b y c, con vértices A, B y C (cada vértice opuesto al lado cuya letra minuscula coincide con el) y un ángulo α con centro en el vertice A. a y b son catetos, c es la hipotenusa:	
$sin(\alpha) = \frac{cateto\ opuesto}{hipotenusa} = \frac{a}{c}$	
$cos(\alpha) = \frac{cateto\ adyacente}{hipotenusa} = \frac{b}{c}$	
$tan(\alpha) = \frac{cateto\ opuesto}{cateto\ adyacente} = \frac{a}{b}$	
$sec(\alpha) = \frac{1}{cos(\alpha)} = \frac{c}{b}$	
$csc(\alpha) = \frac{1}{sin(\alpha)} = \frac{c}{a}$	
$cot(\alpha) = \frac{1}{tan(\alpha)} = \frac{b}{a}$	

Continúa en la siguiente columna

PROPIEDADES DEL MÓDULO (RESIDUO)	
Propiedad neutro	(a % b) % b = a % b
Propiedad asociativa en multiplicación	(ab) % c = ((a % c)(b % c)) % c
Propiedad asociativa en suma	(a + b) % c = ((a % c) + (b % c)) % c
CONSTANTES	
Pi	$\pi = acos(-1) \approx 3{,}14159$
e	$e \approx 2{,}71828$
Número áureo	$\phi = \frac{1+\sqrt{5}}{2} \approx 1{,}61803$

8.3. Sequences

Listado de secuencias mas comunes y como hallarlas.

Estrellas octangulares	0, 1, 14, 51, 124, 245, 426, 679, 1016, 1449, 1990, 2651, ...
	$f(n) = n * (2 * n^2 - 1)$.
Euler totient	1, 1, 2, 2, 4, 2, 6, 4, 6, 4, 10, 4, 12, 6,...
	$f(n)$ = Cantidad de números naturales $\leq n$ coprimos con n.
Números de Bell	1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147, 115975, ...
	Se inicia una matriz triangular con f[0][0] = f[1][0] = 1. La suma de estos dos se guarda en f[1][1] y se traslada a f[2][0]. Ahora se suman f[1][0] con f[2][0] y se guarda en f[2][1]. Luego se suman f[1][1] con f[2][1] y se guarda en f[2][2] trasladandose a f[3][0] y así sucesivamente. Los valores de la primera columna contienen la respuesta.
Números de Catalán	1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, ...
	$f(n) = \frac{(2n)!}{(n+1)!n!}$
Números de Fermat	3, 5, 17, 257, 65537, 4294967297, 18446744073709551617, ...
	$f(n) = 2^{(2^n)} + 1$
Números de Fibonacci	0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...
	$f(0) = 0$; $f(1) = 1$; $f(n) = f(n-1) + f(n-2)$ para $n > 1$
Números de Lucas	2, 1, 3, 4, 7, 11, 18, 29, 47, 76, 123, 199, 322, ...
	$f(0) = 2$; $f(1) = 1$; $f(n) = f(n-1) + f(n-2)$ para $n > 1$
Números de Pell	0, 1, 2, 5, 12, 29, 70, 169, 408, 985, 2378, 5741, 13860, ...
	$f(0) = 0; f(1) = 1; f(n) = 2f(n-1) + f(n-2)$ para $n > 1$

Continúa en la siguiente columna

Números de Tribonacci	0, 0, 1, 1, 2, 4, 7, 13, 24, 44, 81, 149, 274, 504, ...
	$f(0) = f(1) = 0; f(2) = 1; f(n) = f(n-1) + f(n-2) + f(n-3)$ para $n > 2$
Números factoriales	1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, ...
	$f(0) = 1; f(n) = \prod_{k=1}^{n} k$ para $n > 0$.
Números piramidales cuadrados	0, 1, 5, 14, 30, 55, 91, 140, 204, 285, 385, 506, 650, ...
	$f(n) = \frac{n * (n+1) * (2 * n + 1)}{6}$
Números primos de Mersenne	3, 7, 31, 127, 8191, 131071, 524287, 2147483647, ...
	$f(n) = 2^{p(n)} - 1$ donde p representa valores primos iniciando en $p(0) = 2$.
Números tetraedrales	0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, ...
	$f(n) = \frac{n * (n+1) * (n+2)}{6}$
Números triangulares	0, 1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78, 91, 105, ...
	$f(n) = \frac{n(n+1)}{2}$
OEIS A000127	1, 2, 4, 8, 16, 31, 57, 99, 163, 256, 386, 562, ...
	$f(n) = \frac{(n^4 - 6n^3 + 23n^2 - 18n + 24)}{24}$.
Secuencia de Narayana	1, 1, 1, 2, 3, 4, 6, 9, 13, 19, 28, 41, 60, 88, 129, ...
	$f(0) = f(1) = f(2) = 1; f(n) = f(n-1) + f(n-3)$ para todo $n > 2$.

Continúa en la siguiente columna

Secuencia de Silvestre	2, 3, 7, 43, 1807, 3263443, 10650056950807, ...
	$f(0) = 2; f(n+1) = f(n)^2 - f(n) + 1$
Secuencia de vendedor perezoso	1, 2, 4, 7, 11, 16, 22, 29, 37, 46, 56, 67, 79, 92, 106, ...
	Equivale al triangular(n) + 1. Máxima número de piezas que se pueden formar al hacer n cortes a un disco. $f(n) = \frac{n(n+1)}{2} + 1$
Suma de los divisores de un número	1, 3, 4, 7, 6, 12, 8, 15, 13, 18, 12, 28, 14, 24, ...
	Para todo $n > 1$ cuya descomposición en factores primos es $n = p_1^{a_1} p_2^{a_2} ... p_k^{a_k}$ se tiene que: $f(n) = \frac{p_1^{a_1+1} - 1}{p_1 - 1} * \frac{p_2^{a_2+1} - 1}{p_2 - 1} * ... * \frac{p_k^{a_k+1} - 1}{p_k - 1}$

8.4. Time Complexities

Aproximación del mayor número n de datos que pueden procesarse para cada una de las complejidades algoritmicas. Tomar esta tabla solo como referencia.

Complexity	**n**
$O(n!)$	11
$O(n^5)$	50
$O(2^n * n^2)$	18
$O(2^n * n)$	22
$O(n^4)$	100
$O(n^3)$	500
$O(n^2 \log_2 n)$	1.000
$O(n^2)$	10.000
$O(n \log_2 n)$	10^6
$O(n)$	10^8
$O(\sqrt{n})$	10^{16}
$O(\log_2 n)$	-
$O(1)$	-