Algorithm Notebook in C++ and Java

Luis Miguel Báez Aponte - Universidad Nacional de colombia

28 de febrero de 2022

Índice

1. Java: Input And Output

1.1. Java - Scanner and PrintStream

```java
import java.io.PrintStream;
import java.util.Scanner;

public class IO {
    static PrintStream out = System.out;
    static Scanner in = new Scanner(System.in);
    public static void main(String[] args) {
        // INPUT
        String s = in.next();
        int x =   in.nextInt();
        short y = in.nextShort();
        long z =  in.nextLong();
        float a = in.nextFloat();
        double b = in.nextDouble();
        //OUTPUT
        out.println("Hello World: " + x);
        out.print(y);
        out.printf("%d %d %d = %s", x, y, z, s);
    }
}
```

2. Graph

2.1. Disjoint Set Union

```cpp
#include <bits/stdc++.h>
using namespace std;
// Implementación sin compresión de rango
class DisjointSet{
public:
    vector<int> parent;
    DisjointSet(int n): parent(n) {
        for(int i = 0; i < n; ++i) parent[i] = i;
    }
    void join(int a, int b) {
        parent[find(b)] = find(a);
    }
    int find(int a){
        return (a == parent[a]) ? a : parent[a] = find(parent[a]);
    }
    bool check(int a, int b){
        return find(a) == find(b);
    }
};
//Implementación con compresión de rango
class DisjointSet {
public:
    vector<int> parent;
    vector<int> ranks;
    DisjointSet(int n): parent(n), ranks(n) {
        for(int i = 0; i < n; ++i) parent[i] = i;
    }
    int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
    void join(int x, int y) {
        int xRoot = find(x);
        int yRoot = find(y);
        if (xRoot == yRoot){
            return;
        }
        if (ranks[xRoot] < ranks[yRoot]){
            parent[xRoot] = yRoot;
        } else if (ranks[yRoot] < ranks[xRoot]) {
            parent[yRoot] = xRoot;
        } else {
            parent[yRoot] = xRoot;
            ranks[xRoot] = ranks[xRoot] + 1;
```

```
        }
    };
    bool check(int a, int b) {
        return find(a) == find(b);
    }
};

int main() {

    int n = 5;
    DisjointSet dsu(n);
    dsu.join(0, 2);
    dsu.join(4, 2);
    dsu.join(3, 1);

    if (dsu.check(4, 0)){
        cout<<"YES"<<endl;
    } else {
        cout<<"NO"<<endl;
    }
    if (dsu.check(1, 0)) {
        cout<<"YES"<<endl;
    } else {
        cout<<"NO"<<endl;
    }
    // Out:
    // YES
    // NO
    return 0;
}
```

2.2. Algoritmo de Kruskal - Minimum Spanning Tree

```
#include <bits/stdc++.h>
using namespace std;

struct Edge {
    int  u, v, w;
    bool operator<(struct Edge other) {
        return w < other.w;
    }
};
// Disjoint Set Union
class Kruskal {
public:
    vector<struct Edge> E;
    vector<struct Edge> KruskalVector;
    int totalWeightKruskal = 0;
    int vertexNumber, edgeNumber;
    Kruskal(int v, int e): vertexNumber(v), edgeNumber(e) {}
    void addEdge(struct Edge edge) {
        E.push_back(edge);
    }
    int build() {
        DisjointSet dsu(edgeNumber);
        sort(E.begin(), E.end());
        int totalWeight = 0;
        for(struct Edge e : E) {
            if(dsu.find(e.u) != dsu.find(e.v)) {
                KruskalVector.push_back(e);
                totalWeight += e.w;
                dsu.join(e.u, e.v);
            }
        }
        totalWeightKruskal = totalWeight;
        return totalWeight;
    }
};
int main() {
    int vertexNumber = 9, edgeNumber = 14;
    Kruskal kruskal(vertexNumber, edgeNumber);
    kruskal.addEdge({0, 1, 4});
    kruskal.addEdge({0, 7, 8});
    kruskal.addEdge({1, 2, 8});
    kruskal.addEdge({1, 7, 11});
    kruskal.addEdge({2, 3, 7});
    kruskal.addEdge({2, 8, 2});
    kruskal.addEdge({2, 5, 4});
    kruskal.addEdge({3, 4, 9});
    kruskal.addEdge({3, 5, 14});
    kruskal.addEdge({4, 5, 10});
    kruskal.addEdge({5, 6, 2});
    kruskal.addEdge({6, 7, 1});
    kruskal.addEdge({6, 8, 6});
    kruskal.addEdge({7, 8, 7});
    int totalWeight = kruskal.build();
    cout<<"Total Weight : "<<totalWeight<<endl;
    for(struct Edge e: kruskal.KruskalVector) {
```

```
        cout<<"Edge("<<e.u<<", "<<e.v<<", "<<e.w<<")"<<endl;
    }
    return EXIT_SUCCESS;
}
// Total Weight : 37
// Edge(6, 7, 1)
// Edge(2, 8, 2)
// Edge(5, 6, 2)
// Edge(0, 1, 4)
// Edge(2, 5, 4)
// Edge(2, 3, 7)
// Edge(0, 7, 8)
// Edge(3, 4, 9)
```

2.3. Algoritmo de Dijkstra

```
#define INF INT_MAX
struct Node {
    int to;
    int dist;
    bool operator>(const Node& node) const {
        return dist > node.dist;
    }
};

int main() {
    int n = 5;
    int A = 0, B = 1, C = 2, D = 3, E = 4;
    int start = A;
    vector<vector<Node>> G(n);
    G[A].push_back({B, 6});G[B].push_back({A, 6});
    G[A].push_back({D, 1});G[D].push_back({A, 1});
    G[B].push_back({D, 2});G[D].push_back({B, 2});
    G[E].push_back({D, 1});G[D].push_back({E, 1});
    G[B].push_back({E, 2});G[E].push_back({B, 2});
    G[B].push_back({C, 5});G[C].push_back({B, 5});
    G[C].push_back({E, 5});G[E].push_back({C, 5});
    priority_queue<Node, vector<Node>, greater<Node>> Q;
    vector<int> DIST(n, INF);
    DIST[start] = 0;
    //      to   dist
    Q.push({start, 0});
    while (!Q.empty()) {
        int toNode = Q.top().to;
        Q.pop();
        for (Node e: G[toNode]) {
            int newDist = DIST[toNode] + e.dist;
            int to = e.to;
            if (DIST[to] > newDist) {
                Q.push({to, newDist});
                DIST[to] = newDist;
            }
        }
    }
    for(int i = 0; i < n; ++i) {
        cout<<"To: ["<<i<<"] Distance: ["<<DIST[i]<<"]"<<endl;
    }
    // To: [A] Distance: [0]
    // To: [B] Distance: [3]
    // To: [C] Distance: [7]
    // To: [D] Distance: [1]
    // To: [E] Distance: [2]
    return 0;
}
```

Time Complexity: $O\left(|E| + |V| .log\left(|V|\right)\right)$

2.4. Algoritmo de Bellman Ford

```
#define INF INT_MAX
struct edge {
    int to;
    int from;
    int dist;
};
int main() {
    int A = 0, B = 1, C = 2, D = 3, E = 4;
    int n = 5;
    vector<edge> e = {
    //  from  to  distance
            {A, B, -1},
            {A, C, 4},
            {B, C, 3},
            {D, C, 5},
            {D, B, 1},
            {B, D, 2},
            {B, E, 2},
```

```
        {E, D, -3}
    };
    vector<int> d(n, INF);
    int start = A;
    d[start] = 0;
    for(int i = 0; i < n - 1; ++i) {
        for(edge x: e) {
            if(d[x.to] + x.dist < d[x.from] && d[x.to] != INF) {
                d[x.from] = d[x.to] + x.dist;
            }
        }
    }
    cout<<"From 0"<<endl;
    cout<<"To   Min Distance"<<endl;
    for(int i = 0; i < n; ++i) {
        cout<<i<<" -> "<<d[i]<<endl;
    }
    return 0;
}
```

2.5. Algoritmo de Floyd-Warshall

```
#define INF 200000000
typedef vector<vector<int>> vvi;
class FloydWarshall {
public:
    vvi build(vvi graph) {
        int n = graph.size();
        for(int k = 0; k < n; ++k) {
            for(int i = 0; i < n; ++i) {
                for(int j = 0; j < n; ++j) {
                    graph[i][j] = min(graph[i][j], graph[i][k] +
                        graph[k][j]);
                }
            }
        }
        return graph;
    }
};
int main() {
    vvi graph = {
        {INF, 4, INF, INF, INF, INF, INF, 8, INF },
        {4, INF, 8, INF, INF, INF, INF, 11, INF },
        {INF, 8, INF, 7, INF, 4, INF, INF, 2 },
        {INF, INF, 7, INF, 9, 14, INF, INF, INF },
        {INF, INF, INF, 9, INF, 10, INF, INF, INF },
        {INF, INF, 4, 14, 10, INF, 2, INF, INF },
        {INF, INF, INF, INF, INF, 2, INF, 1, 6 },
        {8, 11, INF, INF, INF, INF, 1, INF, 7 },
        {INF, INF, 2, INF, INF, INF, 6, 7, INF }
    };
    FloydWarshall fw;
    graph = fw.build(graph);
    // Imprimir Grafo
    for(int i = 0; i < graph.size(); ++i) {
        cout<<i<<" |";
        for(int j = 0; j < graph.size(); ++j) {
            cout<<graph[i][j]<<" ";
        }
        cout<<endl;
    }
    // 0 |8 4 12 19 21 11 9 8 14
    // 1 |4 8 8 15 22 12 12 11 10
    // 2 |12 8 4 7 14 4 6 7 2
    // 3 |19 15 7 14 9 11 13 14 9
    // 4 |21 22 14 9 18 10 12 13 16
    // 5 |11 12 4 11 10 4 2 3 6
    // 6 |9 12 6 13 12 2 2 1 6
    // 7 |8 11 7 14 13 3 1 2 7
    // 8 |14 10 2 9 16 6 6 7 4
    return 0;
}
```

3. Number Theory

3.1. GCD - Algoritmo de Euclides - Euclid's algorithm

```
// Recursive
public int gcd (int a, int b) {
    if (b == 0) {
        return a;
    } else {
        return gcd(b, a % b);
    }
}
```

```java
// Iterative
public int gcd (int a, int b) {
    int tmp = 0;
    while (b != 0){
        tmp = a;
        a = b;
        b = tmp % b;
    }
    return a;
}
```

3.2. Least Common Multiple

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

3.3. Algoritmo de Euclides Extendido - Ecuacion Diofantica

```java
public static Tuple extended_euclidean(int a, int b) {
    if(a == 0) return new Tuple(b, 0, 1);
    Tuple tuple = extended_euclidean(b % a, a);
    int gcd=tuple.gcd, x=tuple.x, y=tuple.y;
    return new Tuple(gcd, (x - (b/a) * y), y);
}
static class Tuple {
    int gcd;
    int x;
    int y;
    Tuple(int gcd, int y, int x) {
        this.gcd = gcd;
        this.x = x;
        this.y = y;
    }
}
```

3.4. Criba de Eratostenes - Sieve of Eratosthenes

```java
public List<Integer> criba(int n) {
    boolean[] isPrime = new boolean[n];
    for(int i = 4; i <= n; i += 2) {
        isPrime[i] = true;
    }
    for(int p = 3; p <= n; p += 2) {
        if(!isPrime[p]) {
            for(int j = 2*p; j < n;j += p) {
                isPrime[j] = true;
            }
        }
    }
    List<Integer> primes = new ArrayList<>();
    for(int i = 2; i < isPrime.length; ++i) {
        if(!isPrime[i]) {
            primes.add(i);
        }
    }
    return primes;
}
```

3.5. Factores Primos - Prime Factors

```java
public static List<Pair> primeFactors(int number) {
    // criba:
    // Todos los number primos desde [2, number]
    List<Integer> primes = criba(number);
    List<Pair> factors = new ArrayList<>();
    for(Integer prime: primes) {
        if(number % prime == 0) {
            int count = 0;
            while(number % prime == 0) {
                number /= prime;
                count++;
            }
            factors.add(new Pair(prime, count));
        }
    }
    return factors;
}
```

3.6. Test de Primalidad

```
public static boolean isPrime(int number) {
    if(number <= 0) return false;
    else if(number <= 3) return true;
    if(number%2==0 || number%3==0) return false;
    for(int i = 5; i*i <= number; i += 6) {
        if(number%i==0 || number%(i+2)==0) {
            return false;
        }
    }
    return true;
}
```

Time Complexity: $O\left(\sqrt{n}\right)$

4. Bit Mask

4.1. Count Bits - C++

```
__builtin_clz // El número de ceros al comienzo del número.
__builtin_ctz // El número de ceros al final del número
__builtin_popcount // el número de unos en el número
__builtin_parity // La paridad (par o impar) del número de unos (1: Par,
    0: Impar)
```

4.2. Bit menos significativo - (Least Significant Bit)

```
int x = 100; // 0b1100100
cout<<(x & -x)<<endl;
// 4 -> '0b100'
```

5. Operations with Ranges

5.1. Segment Tree

```
// Limite de la longitud del Array
const int N = 100000;
int n; // Tamaño del Arreglo
// Maximo Tamaño del Arbol
int tree[2 * N];
// Funcion que Construye el Arbol
void build( int arr[]) {
    for (int i=0; i<n; i++){
        tree[n+i] = arr[i];
    }
    // construye el árbol calculando a los padres
    for (int i = n - 1; i > 0; --i) {
        tree[i] = tree[i<<1] + tree[i<<1 | 1];
    }
}
// Función para actualizar un nodo de árbol
void updateTreeNode(int index, int value) {
    // Establecer el valor en la posición p
    tree[index+n] = value;
    index = index + n;
    // Moverse hacia arriba y actualizar a los padres
    for (int i = index; i > 1; i >>= 1){
        tree[i>>1] = tree[i] + tree[i^1];
    }
}
// función para obtener la suma en el intervalo [l, r)
int query(int l, int r) {
    int res = 0;
    // bucle para encontrar la suma en el rango
    for (l += n, r += n; l < r; l >>= 1, r >>= 1) {
        if (l&1) {
            res += tree[l++];
        }
        if (r&1) {
            res += tree[--r];
        }
    }
    return res;
}

int main() {
    // Index    [0  1  2  3  4  5  6  7  8  9   10  11]
    int array[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
    // n es global
    n = sizeof(array) / sizeof(array[0]);
    // Construir el Arbol de Segmentos
    build(array);
```

```
    // Imprimir la suma en rango [1,2)
    cout << query(1, 3)<<endl;
    // Ans: 5
    // Modificar el indice 2 por el valor 1
    updateTreeNode(2, 1);
    // Imprimir la suma en rango [1,2)
    cout << query(1, 3)<<endl;
    // Ans: 3
    return 0;
}
```

5.2. Spanse Table - Tabla Dispersa

```
#define MAXN 8 // Longitud del Arreglo
#define LOGN 3 // = log2(MAXN)
int ST[LOGN][MAXN];
void build(int A[MAXN], int n) {
    int h = floor(log2(n));
    for (int i = 0; i < MAXN; i++){
        ST[i][0] = A[i];
    }
    for (int j = 1; j <= h; j++) {
        for (int i = 0; i + (1 << j) <= MAXN; i++) {
            ST[i][j] = ST[i][j-1] + ST[i + (1 << (j - 1))][j - 1];
        }
    }
}
int query(int L, int R) {
    // query in range [l,r)
    int sum = 0;
    for (int j = LOGN; j >= 0; j--) {
        if ((1 << j) <= R - L + 1) {
            sum += ST[L][j];
            L += 1 << j;
        }
    }
    return sum;
}

int main() {
    //   index    0  1  2  3  4  5  6   7
    int arr[MAXN] = {3, 1, 5, 3, 4, 7, 6, 1};
    build(arr, MAXN);
    cout<<query(0, 2)<<endl; // [0, 7]
    // 9
    return 0;
}
```

5.3. Descomposición SQRT - SQRT Decomposition

```
int main() {
    // input data
    // Index          [0  1  2  3  4  5  6  7  8]
    vector<int> array = {1, 5, 2, 4, 6, 1, 3, 5, 7};
    int n = array.size();
    // Raiz Cuadrada de n:
    int len = (int) sqrt (n) + 1;
    vector<int> squareRootRange (len);
    // Preprocesamiento
    for (int i=0; i < n; ++i){
        squareRootRange[i / len] += array[i];
    }
    // Queries
    int l = 3, r = 8;
    int sum = 0;
    for (int i = l; i <=r ; ){
        if (i % len == 0 && i + len - 1 <= r) {
            // Si todo el bloque que comienza en i pertenece a [l; r]
            // Suma todo el Bloque
            sum += squareRootRange[i / len];
            i += len;
        } else {
            // Suma Uno a Uno
            sum += array[i];
            ++i;
        }
    }
    cout<<sum<<endl;
    // Answer: 26
    return 0;
}
```

6. Computational Geometry

6.1. Macros

```
#include<complex>
using namespace std;
typedef long long ll;
typedef complex<ll> point;
#define x(p) real(p)
#define y(p) imag(p)
#define dot(p1, p2) x(conj(p1) * p2)
#define cross(p1, p2) y(conj(p1) * p2)
#define line(p1, p2) p2 - p1
#define PI acos(0) * 2
#define PI 3.141592653589793238462643383279502884L
#define angle180(p1) arg(p1)*(180/PI)
```

6.2. Add Vectors

```
point p1(2, 4);
point p2(4, 2);
point pt = p1 + p2;
cout<<"("<<x(pt)<<", "<<y(pt)<<")"<<endl;
// (6, 6)
```

6.3. Subtract Vectors

```
point p1(2, 4);
point p2(4, 2);
point pt = p1 - p2;
cout<<"("<<x(pt)<<", "<<y(pt)<<")"<<endl;
// (-2, 2)
```

6.4. Producto Punto - Dot Product

```
point p1(2, 4);
point p2(4, 2);
ll ans = dot(p1, p2);
cout<<ans<<endl;
// 16
```

6.5. Producto Cruz - Cross Product

```
point p1(2, 4);
point p2(4, 2);
ll ans = cross(p1, p2);
cout<<ans<<endl;
// -12 en termino de vectores seria (0, 0, -12)
```

6.6. Distance between two points

```
point p1(2, 4);
point p2(4, 2);
point pt = line(p1, p2);
cout<<pt<<endl;
// (2,-2)
```

6.7. Normal Two Vectors

Magnitud de la direfencia de los Vector

```
point p1(1.0, 2.0);
point p2(2.0, 4.0);
// el tipo de dato tiene que ser ld (long double)
// para que la norma funcione
cout<<setprecision(10)<<abs(line(p1, p2))<<endl;
// 2.236067977
cout<<setprecision(10)<<abs(point{4.0, 2.0})<<endl;
// 4.472135955
```

6.8. Rotate a vector Degrees counterclockwise - Rotar un vector Grados en sentido antihorario

```
point p1(1, 1);
ld theta = PI/2.0;
// Nota:
```

```cpp
// polar<Type> y complex<Type> deben tener el mismo Type
// Para poder hacer la operación
point rotated = p1 * polar<ld>(1.0, theta);
cout<<rotated<<endl;
// (-1,1)
```

6.9. Angle (in Degrees) of a Vector - Angulo (en Grados) de un Vector

```cpp
#define angle180(p1) arg(p1)*(180/PI)
ld angle360(point pt) {
  ld out = 0.0;
  ld tmp = angle180(pt);

  if(tmp >= 0 && tmp <= 90) {
    out = tmp;
  } else if(tmp > 90 && tmp <= 180) {
    out = tmp;
  } else if(tmp < 0 && tmp >= -90) {
    out = 180.0 + (180.0 - abs(tmp));
  } else if(tmp < -90 && tmp >= -180) {
    out = 270.0 + (90.0 - abs(tmp));
  }
  return out;
}
int main() {
  point p1 (1, 1);
  cout<<angle360(p1)<<endl;
  point p2 (-1, 1);
  cout<<angle360(p2)<<endl;
  point p3 (-1, -1);
  cout<<angle360(p3)<<endl;
  point p4 (1, -1);
  cout<<angle360(p4)<<endl;
  // 45
  // 135
  // 225
  // 315
}
```

6.10. Argument - Angle (in Radians) of a Vector - Argumento - Angulo (en Radianes) de un Vector

```cpp
point p1(1 ,1);
cout<<arg(p1)<<" Radianes"<<endl;
// 0.785398 Radianes
// Lo equivalente en grados es 45
```

6.11. Pendiente de la recta

```cpp
point p1(1 ,1);
point p2(6 ,8);
cout<<tan(arg(p2 -p1))<<endl;
// 1.4
```

6.12. Area of a triangle with vectors - Área de Un triángulo con Vectores

```cpp
double area(point p1, point p2, point p3) {
  point l1 = line(p1, p2);
  point l2 = line(p1, p3);
  // El Producto Cruz de dos vectores es el area
  // que forman entre ellos
  double ans = cross(l1, l2);
  // La Area del Triangulo es la mitad del Area del paralelogramo
  double out = double(ans) / 2.0;
  return abs(out);
}
```

6.13. Radius given 2 Points - Radio dado 2 Puntos

```cpp
typedef long double ld;
typedef complex<ld> point;
#define x(p) p.real()
#define y(p) p.imag()
ld radio(point p1, point p2) {
    return sqrt(
        pow((x(p2) - x(p1)), 2) +
```

```
        pow((y(p2) - y(p1)), 2)
    );
}
int main() {
    point p1(1, 1);
    point p2(5, 5);
    ld ans = radio(p1, p2);
    cout<<ans<<endl;
    // 5.65685
    return 0;
}
```

6.14. Check if one circle is inside the other - Revisar si un círculo está dentro del otro

```
typedef complex<ld> point;
bool contains(point p1, ll r1, point p2, ll r2) {
    ld dist = abs(p2 - p1);
    if (r1 > r2 + dist) {
        return true;
    } else {
        return false;
    }
}
int main() {
    point p1(0, 0);
    ll r1 = 10;
    point p2(1, 1);
    ll r2 = 2;
    bool ans = contains(p1, r1, p2, r2);
    if(ans) {
        cout<<"P1 Contiene a P2"<<endl;
    } else {
        cout<<"P1 No Contiene a P2"<<endl;
    }
    // P1 Contiene a P2
    return 0;
}
```

6.15. Check if two circles do not intersect - Revisar si dos círculos no se interceptan

```
bool disjoint(point p1, ll r1, point p2, ll r2) {
    ld dist = abs(p2 - p1);
    if(dist >= r2 + r1) {
        return false;
    } else {
        return true;
    }
}
int main() {
    point circle1(0, 0);
    ll radio1 = 2;
    point circle2(3, 3);
    ll radio2 = 2;
    cout<<disjoint(circle1, radio1, circle2, radio2)<<endl;
    // False
    return 0;
}
```

7. Tree

7.1. Binary Search Tree

```
// Insertar
public void insert(T key) {
    if(this.root == null) {
        this.root = new TreeNode(key);
    } else {
        this.root = insert(this.root, key);
    }
}
private TreeNode insert(TreeNode<T> node, T key) {
    if(node == null) {
        return new TreeNode(key);
    }
    if(key.compareTo(node.value) < 0) { // key < node.value
        node.left = insert(node.left, key);
    } else if( key.compareTo(node.value) > 0) { // key > node.value
        node.right = insert(node.right, key);
    }
```

```java
    return node;
}

// Buscar
public T search(T key) {
    TreeNode<T> ans = search(this.root, key);
    if(ans == null) {
        return null;
    } else {
        return ans.value;
    }
}
private TreeNode<T> search(TreeNode<T> node, T key) {
    if(node == null || key.compareTo(node.value) == 0) {
        return node;
    }
    if(key.compareTo(node.value) < 0) { // key < node.value
        return search(node.left, key);
    }
    return search(node.right, key);
}
// Eliminar
public TreeNode<T> delete(T key) {
    this.root = delete(this.root, key);
    return this.root;
}

private TreeNode<T> delete(TreeNode<T> node, T key) {
    if(isEmpty(node)) return null;

    if(key.compareTo(node.value) < 0) { // key < node.value
        node.left = delete(node.left, key);
    } else if(key.compareTo(node.value) > 0) { // key > node.value
        node.right = delete(node.right, key);
    } else {
        if(isLeaf(node)) {
            node = null;
        } else if(hasOneChild(node)) {
            if(node.left!=null) {
                node = node.left;
            } else { // node.right != null
                node = node.right;
            }
        } else { // hasTwoChild
            node.value = (T) minNode(node.right);
            node.right = delete(node.right, node.value);
        }
    }
    return node;
}

public T minNode(TreeNode<T> node) {
    TreeNode<T> ans = node;
    T out = node.value;

    while(ans != null) {
        out = ans.value;
        ans = ans.left;
    }
    return out;
}

private boolean isEmpty(TreeNode node) {
    return node == null;
}

private boolean isLeaf(TreeNode node) {
    if(isEmpty(node)) return false;
    return node.left == null && node.right == null;
}

private boolean hasOneChild(TreeNode node) {
    if(isEmpty(node)) return false;
    return
        (node.left!=null&&node.right==null)||(node.left==null&&node.right!=null);
}
```

7.2. Binary Search Tree - Depth-first Search - PreOrder

```java
private void preOrder(TreeNode node) {
    if(node == null) {
        return;
    }
    System.out.print(node.value + " ");
    preOrder(node.left);
    preOrder(node.right);
}
```

7.3. Binary Search Tree - Depth-first Search - InOrder

```
private void inOrder(TreeNode node) {
    if(node == null) {
        return;
    }
    inOrder(node.left);
    System.out.print(node.value+" ");
    inOrder(node.right);
}
```

7.4. Binary Search Tree - Depth-first Search - PostOrder

```
private void postOrder(TreeNode node) {
    if(node == null) {
        return;
    }
    postOrder(node.left);
    postOrder(node.right);
    System.out.print(node.value+" ");
}
```

7.5. Binary Search Tree - Breadth-first Search

```
private void bfs(TreeNode<T> node) {
    if(isEmpty(node)) return;
    Queue<TreeNode<T>> queue = new LinkedList<>();
    queue.add(node);
    while(!queue.isEmpty()) {
        TreeNode<T> tmp = queue.remove();
        System.out.print(tmp.value + " ");
        if(!isEmpty(tmp.left)) {
            queue.add(tmp.left);
        }
        if(!isEmpty(tmp.right)) {
            queue.add(tmp.right);
        }
    }
    System.out.println();
}
```

```
public void bfs() {
    System.out.print("BFS: ");
    bfs(this.root);
    System.out.println();
}
```

7.6. Lowest Common Ancestor - BST

```
public TreeNode LCA(TreeNode root, int p, int q) {
    if(isEmpty(root)) return root;
    int valueRoot = root.val;
    if(p < valueRoot && q < valueRoot) {
        return LCA(root.left, p, q);
    } else if(p > valueRoot && q > valueRoot) {
        return LCA(root.right, p, q);
    }
    return root;
}
```

7.7. N-ary Tree

7.8. N-ary Tree - Depth-first Search - Pre-Order (Recursive)

```
public static void preOrden(TreeNode node) {
    if(isEmpty(node)) return;
    System.out.println(node.value);
    for(TreeNode child: node.children) {
        preOrden(child);
    }
}
```

7.9. N-ary Tree - Depth-first Search - Pre-Order (Iterative)

```
public List<Integer> preOrder(TreeNode root) {
    List<Integer> output = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    stack.push(root);
    TreeNode tmp = null;
```

```
    while(!stack.isEmpty()) {
        tmp = stack.pop();
        if(!isEmpty(tmp)) {
            output.add(tmp.val);
            List<TreeNode> children = tmp.children;
            for(int i = children.size() - 1; i >= 0; --i) {
                if(children.get(i) != null) {
                    stack.push(children.get(i));
                }
            }
        }
    }
    return output;
}
```

7.10. N-ary Tree - Depth-first Search - Post-Order (Recursive)

```
public static void postOrden(TreeNode node) {
    if(isEmpty(node)) return;
    for(TreeNode child: node.children) {
        postOrden(child);
    }
    System.out.println(node.value);
}
```

7.11. N-ary Tree - Depth-first Search - Post-Order (Iterative)

```
public static void postOrden(TreeNode root) {
    List<Integer> output = new ArrayList<>();
    Stack<TreeNode> stack = new Stack<>();
    stack.add(root);
    TreeNode tmp = root;
    while(!stack.isEmpty()) {
        tmp = stack.pop();
        if(!isEmpty(tmp)) {
            output.add(tmp.value);
            for(TreeNode node: tmp.children) {
                stack.add(node);
            }
        }
    }
    Collections.reverse(output);
    for(Integer number: output) {
        System.out.println(number);
    }
}
```

7.12. N-ary Tree - Breadth-first Search

```
public void bfs(TreeNode root) {
    List<List<Integer>> output = new ArrayList<>();
    Queue<TreeNode> queue = new LinkedList<>();
    queue.add(root);
    TreeNode tmp = null;
    while(!queue.isEmpty()) {
        tmp = queue.remove();
        if(!isEmpty(tmp)) {
            System.out.println(tmp.value);
            for(TreeNode node: tmp.children) {
                queue.add(node);
            }
        }
    }
}
```

8. Binary Search

8.1. Binary Search (Iterative)

```
public int search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    int mid = 0;
    while (left <= right) {
        mid = left + (right - left ) / 2;
        if(nums[mid] == target) {
            return mid;
        }
        if(target < nums[mid]) {
```

```
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }
    // Not Found
    return -1;
}
```

8.2. Binary Search (Recursiva)

```
public int search(int[] nums, int target) {
    return search(nums, 0, nums.length - 1, target);
}
private int search(int[] nums, int left, int right, int target) {
    if(left <= right) {
        int mid = left + (right - left / 2);
        if(nums[mid] == target) {
            return mid;
        }
        if(target < nums[mid]) {
            return search(nums, left, mid - 1, target);
        } else {
            return search(nums, mid + 1, right, target);
        }
    }
    // Not Found
    return -1;
}
```

9. String

9.1. Levenshtein Distance

```
public int minDistance(String word1, String word2) {
    int size1 = word1.length();
    int size2 = word2.length();
    int[][] dp = new int[size1+1][size2+1];
    char c1=' ', c2=' ';
    int indicator = 0;
    // Llenar la columna 0
    for(int i = 0; i <= size1; ++i) {
        dp[i][0] = i;
    }
    // Llenar la fila 0
    for(int j = 0; j <= size2; ++j) {
        dp[0][j] = j;
    }
    for(int i = 1; i <= size1; ++i) {
        c1 = word1.charAt(i-1);
        for(int j = 1; j <= size2; ++j) {
            c2 = word2.charAt(j-1);
            // Si son iguales no hay que cambiar nada
            if(c1 == c2) indicator = 0;
            else indicator = 1;
            dp[i][j] = min(
                dp[i - 1][j] + 1, // Deletion
                dp[i][j - 1] + 1, // Insertion
                dp[i - 1][j - 1] + indicator // Substitution
            );
        }
    }
    return dp[size1][size2];
}
private int min(int x, int y, int z) {
    return Math.min(x, Math.min(y, z));
}
```

9.2. Trie (Prefix Tree)

```
// Node
static class TrieNode {
    private final int ALPHABET = 26;
    public TrieNode[] children;
    public boolean isEndWord;
    TrieNode() {
        this.children = new TrieNode[ALPHABET];
        this.isEndWord = false;
    }
}
// Insertar
public void insert(String word) {
    if(word.length() == 0) return;
```

```java
    char curr;
    int index = 0;
    TrieNode tmp = root;
    for(int i = 0; i < word.length(); ++i) {
        curr = word.charAt(i);
        index = curr - 'a';
        if(tmp.children[index] == null) {
        }
        tmp = tmp.children[index];
    }
    tmp.isEndWord = true;
}
// Buscar
public boolean search(String word) {
    if(word.length() == 0) return false;
    char curr;
    int index = 0;
    TrieNode tmp = root;
    for(int i = 0; i < word.length(); ++i) {
        curr = word.charAt(i);
        index = curr - 'a';
        if(tmp.children[index] == null) {
            return false;
        }
        tmp = tmp.children[index];
    }
    return tmp!=null?tmp.isEndWord:false;
}
// Inicia Con?
public boolean startsWith(String prefix) {
    if(prefix.length() == 0) return false;
    char curr;
    int index = 0;
    TrieNode tmp = root;
    for(int i = 0; i < prefix.length(); ++i) {
        curr = prefix.charAt(i);
        index = curr - 'a';
        if(tmp.children[index] == null) {
            return false;
        }
        tmp = tmp.children[index];
    }
    return true;
}
// Obtener todas las palabras de trie
public List<String> getWords() {
    List<String> words = new ArrayList<>();
    dfs(this.root, "", words);
    return words;
}
public void dfs(TrieNode node, String characters, List<String> words) {
    if(isEmpty(node)) return;
    if(node.isEndWord) {
        words.add(characters);
    }
    String newWord = "";
    for(int i = 0; i < node.children.length; ++i) {
        if(!isEmpty(node.children[i])) {
            newWord = characters + (char)(i+'a')+"";
            dfs(node.children[i], newWord, words);
        }
    }
}
// Obtener Todas las Palabras que Empiezan con un Prefijo
public List<String> getWordsWithPrefix(String prefix) {
    List<String> words = new ArrayList<>();
    if(prefix.length() == 0) return words;
    char curr;
    int index = 0;
    TrieNode tmp = this.root;
    for(int i = 0; i < prefix.length(); ++i) {
        curr = prefix.charAt(i);
        index = curr - 'a';
        if(tmp.children[index] == null) {
            return words;
        }
        tmp = tmp.children[index];
    }
    dfs(tmp, prefix, words);
    return words;
}
```

9.3. Algoritmo KMP

```java
// Search
public List<Integer> search(String txt, String pat) {
    List<Integer> output = new ArrayList<>();
    int N = txt.length();
```

```
    int M = pat.length();
    if(M > N) return output;
    // Longest Prefix Suffix
    int lps[] = new int[M];
    int j = 0; // index for pat[]
    // Calcular el array con los datos del 'Longest Prefix Suffix'
    LPS(pat, lps); // LPS
    int i = 0; // index for txt[]
    while (i < N) {
        if (pat.charAt(j) == txt.charAt(i)) {
            j++;
            i++;
        }
        if (j == M) {
            // Found pattern at index (i-j)
            output.add(i-j);
            j = lps[j - 1];
        } else if (i < N && pat.charAt(j) != txt.charAt(i)) {
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i = i + 1;
            }
        }
    }
    return output;
}
// Longest Prefix Suffix Array
private void LPS(String pat, int lps[]) {
    int M = pat.length();
    int len = 0;
    int i = 1;
    lps[0] = 0; // lps[0] siempre es 0
    // Calcular lps[i] para i = 1 to M-1
    while (i < M) {
        if (pat.charAt(i) == pat.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1];
            } else {
                lps[i] = len;
                i++;
                }
            }
        }
    }
```

9.4. Longest Common SubString