Índice

1. Mapas

Estructura de datos que guarda pares *(clave, valor)*. El HashMap no pone las claves en ningún orden en particular. TreeMap ordena las claves de acuerdo a su orden natural. LinkedHashMap pone las claves en el orden en que se ingresen.

Las operaciones .put(), .get() y .containsKey() son $O(1)$ en HashMap y LinkedHashMap, y $O(\log n)$ en TreeMap.

Ejemplo: Contar cuántas veces aparece cada palabra en un String.

```java
public static void main(String args[]){
  HashMap<String, Integer> map = new HashMap<String, Integer>();
  //TreeMap<String, Integer> map = new TreeMap<String, Integer>();
  //LinkedHashMap<String, Integer> map = new LinkedHashMap<String, Integer>();

  String s = "tres tristes tigres tragaban trigo en un trigal en tres tristes trastos";
  String palabras[] = s.split(" ");

  for(int i=0; i<palabras.length; i++){
    if(!map.containsKey(palabras[i])){
      map.put(palabras[i], 1);
    }else{
      map.put(palabras[i], map.get(palabras[i])+1);
    }
  }
```

```java

    //Obtener un elemento
    System.out.println(map.get("tres"));

    //Recorrer el mapa
    for(Entry<String, Integer> e : map.entrySet()){
      System.out.println(e.getKey() + " : " + e.getValue());
    }
  }
```

2. Sets

Estructura de datos que actúa como “bolsa” donde se almacenan elementos, pero no puede almacenar elementos duplicados.

En HashSet .add() y .contains() son $O(1)$, mientras que en TreeSet son $O(\log n)$. Sin embargo, en el TreeSet los elementos quedan ordenados.

```java
  public static void main(String[] args) {
    HashSet<String> hs = new HashSet<String>();
    //TreeSet<String> ts = new TreeSet<String>();

    hs.add("Hola");
    hs.add("Hola");
    hs.add("Mundo");
    //Imprime 2, porque no se aceptan repetidos
    System.out.println(hs.size());

    //Recorrido
    for(String s: hs){
      System.out.println(s);
    }
  }
```

3. Union-Find

Estructura de datos que soporta las siguientes operaciones eficientemente:

- Unir los conjuntos de los elementos p, q
- Determinar si los elementos p, q pertenecen al mismo conjunto o no

```java
class UnionFind{
  private int[] parent, size;
  private int components;

  // n = Numero de nodos
  public UnionFind(int n){
    components = n;
    parent = new int[n];
    size = new int[n];
    for(int i=0; i<n; i++){
      parent[i] = i;
      size[i] = 1;
    }
  }

  private int root(int p){
    while(p != parent[p]){
      parent[p] = parent[parent[p]];
      p = parent[p];
    }
    return p;
  }

  //Une los nodos p,q
  public void union(int p, int q){
    int rootP = root(p);
    int rootQ = root(q);
    if(rootP != rootQ){
      if(size[rootP] < size[rootQ]){
        parent[rootP] = rootQ;
        size[rootQ] = size[rootQ] + size[rootP];
      }else{
        parent[rootQ] = rootP;
        size[rootP] = size[rootP] + size[rootQ];
      }
      components--;
    }
  }

  //Retorna true si p,q estan conectados
  public boolean connected(int p, int q){
    return root(p) == root(q);
  }

```

```
  //Retorna el numero de componentes conexas
  public int getComponents(){
    return components;
  }
}

class Main {
  public static void main(String[] args){
    UnionFind uf = new UnionFind(5);
    uf.union(0, 2);
    uf.union(1, 0);
    uf.union(3, 4);

    //El numero de componentes es
    int comp = uf.getComponents();

    //Dos nodos estan conectados?
    boolean connected = uf.connected(0, 3);
  }
}
```

4. Grafos

4.1. BFS y DFS

Recorren un grafo a partir de un nodo origen y visitan todos los nodos alcanzables desde éste. Ambos algoritmos tienen una complejidad de $O(n+m)$ donde n es el número de nodos y m es el número de aristas del grafo. El siguiente ejemplo está con DFS pero funciona igual con BFS.

```
  static ArrayList<Integer> g[];
  static boolean seen[];

  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();

    seen = new boolean[n];
    g = new ArrayList[n];
    for(int i = 0; i < n; i++){
      g[i] = new ArrayList<Integer>();
    }

    while(sc.hasNextInt()){
      int u = sc.nextInt();
      int v = sc.nextInt();
      g[u].add(v);
      // Si el grafo es no-dirigido, tambien se agrega arista de v a u
      g[v].add(u);
    }

    //Visita solo los nodos que son alcanzables desde el nodo 's'
    int s = 0;
    dfs(s);

    //Con el vector 'seen' vemos cuales son estos nodos
    for(int i=0; i<n; i++){
      if(seen[i]){
        //'i' es alcanzable desde 's'
      }
    }

    //Si queremos visitar todos los nodos
    for(int u=0; u<n; u++){
      if(!seen[u]){
        //Si no hemos visitado 'u', hacer DFS en 'u'
        dfs(u);
      }
    }
  }

  private static void dfs(int u){
    seen[u] = true;
    int len = g[u].size();
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(!seen[v]){
        dfs(v);
      }
    }
  }

  private static void bfs(int u){
    seen[u] = true;
    Queue<Integer> q = new LinkedList<Integer>();
    q.add(u);
```

```java
    while(!q.isEmpty()){
      u = q.poll();
      int len = g[u].size();
      for(int i=0; i<len; i++){
        int v = g[u].get(i);
        if(!seen[v]){
          seen[v] = true;
          q.add(v);
        }
      }
    }
}
```

4.2. Shortest Hop

Modificación de BFS que calcula el camino más corto desde un nodo origen s a todos los demás. Sólo funciona cuando el peso de todas las aristas es 1. Su complejidad es la misma de BFS: $O(n+m)$.

```java
static ArrayList<Integer> g[];
static boolean seen[];
static int dist[];

public static void main(String[] args) {
  int n = 10;

  seen = new boolean[n];
  dist = new int[n];
  g = new ArrayList[n];
  for(int i=0; i<n; i++){
    g[i] = new ArrayList<Integer>();
  }

  int s = 0;
  shortestHop(s);
  //Despues de llamar este metodo, en dist[i] esta la distancia mas corta (s,i)
}

public static void shortestHop(int u){
  int n = g.length;

  //Distancia "infinita" hacia todos los nodos
  Arrays.fill(dist, Integer.MAX_VALUE);
  //Distancia 0 hacia el nodo de origen
  dist[u] = 0;

  //BFS "modificado"
  seen[u] = true;
  Queue<Integer> q = new LinkedList<Integer>();
  q.add(u);
  while(!q.isEmpty()){
    u = q.poll();
    int len = g[u].size();
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(!seen[v]){
        seen[v] = true;
        q.add(v);
        //Lo unico que cambia es que se calcula el dist[v]
        dist[v] = dist[u] + 1;
      }
    }
  }
}
```

4.3. Ordenamiento topológico

Todo grafo dirigido acíclico (DAG) tiene un ordenamiento topológico. Esto significa que para todas las aristas (u,v), u aparece en el ordenamiento antes que v. Visualmente es como si se pusieran todos los nodos en línea recta y todas las aristas fueran de izquierda a derecha, ninguna de derecha a izquierda.

El algoritmo para hallar dicho ordenamiento es una modificación de DFS y complejidad es la misma: $O(n+m)$. El método retorna falso si detecta un ciclo en el grafo, ya que en este caso no existe ordenamiento topológico posible.

```java
static ArrayList<Integer> g[];
static int seen[];
static LinkedList<Integer> topoSort;

public static void main(String[] args) {
  int n = 10;

  seen = new int[n];
  topoSort = new LinkedList<Integer>();
  g = new ArrayList[n];
  for(int i = 0; i < n; i++){
```

```java
      g[i] = new ArrayList<Integer>();
    }

    boolean sinCiclo = true;

    //Es necesario hacer el ciclo para visitar todos los nodos
    for(int u=0; u<n; u++){
      if(seen[u] == 0){
        sinCiclo = sinCiclo && topoDfs(u);
      }
    }

    if(sinCiclo){
      //La lista 'topoSort' contiene los nodos en su orden topologico
    }else{
      //Hay un ciclo
    }
  }

  private static boolean topoDfs(int u){
    //DFS "modificado" para hacer ordenamiento topologico
    //Se marca 'u' como 'gris'
    seen[u] = 1;
    int len = g[u].size();
    boolean sinCiclo = true;
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(seen[v] == 0){
        sinCiclo = sinCiclo && topoDfs(v);
      }else if(seen[v] == 1){
        //Hay un ciclo, retorna falso
        sinCiclo = false;
      }
    }
    //Se agrega el nodo 'u' al inicio de la lista y se marca 'negro'
    seen[u] = 2;
    topoSort.addFirst(u);
    return sinCiclo;
  }
```

4.4. Componentes fuertemente conexas (Algoritmo de Tarjan)

Calcula la componente fuertemente conexa a la que pertenece cada nodo de un grafo dirigido. Si dos nodos u, v están en la misma componente, significa que existe un camino de u a v y uno de v a u. Su complejidad es $O(n+m)$.

```java
static ArrayList<Integer> g[];
static boolean[] seen, stackMember;
static int[] disc, low, scc;
static Stack<Integer> st;
static int time, component;

public static void main(String[] args) {
  int n = 10;

  seen = new boolean[n];
  stackMember = new boolean[n];
  disc = new int[n];
  low = new int[n];
  scc = new int[n];
  st = new Stack<Integer>();
  time = 0;
  component = 0;
  g = new ArrayList[n];
  for(int i = 0; i < n; i++){
    g[i] = new ArrayList<Integer>();
  }

  for(int u=0; u<n; u++){
    if(!seen[u]){
      tarjan(u);
    }
  }
  //scc[i]==x significa que 'i' pertenece a la componente 'x'
}

private static void tarjan(int u){
  seen[u] = true;
  st.add(u);
  stackMember[u] = true;
  disc[u] = time;
  low[u] = time;
  time++;
```

```java

    int len = g[u].size();
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(!seen[v]){
        tarjan(v);
        low[u] = Math.min(low[u], low[v]);
      }else if(stackMember[v]){
        low[u] = Math.min(low[u], disc[v]);
      }
    }

    if(low[u] == disc[u]){
      int w;
      do{
        w = st.pop();
        stackMember[w] = false;
        scc[w] = component;
      }while(w != u);
      component++;
    }
  }
```

4.5. Puntos de articulación

Halla los puntos de articulación de un grafo. Un punto de articulación es un nodo del grafo que si se quitara causaría que el grafo se “desconectara”. Si el grafo no era conexo en un principio, un punto de articulación es un nodo que, si se quitara, incrementaría el número de componentes conexas. La complejidad del algoritmo es $O(n+m)$.

```java
  static ArrayList<Integer> g[];
  static boolean[] seen, ap;
  static int[] disc, low, parent;
  static int time;

  public static void main(String[] args) {
    int n = 10;

    g = new ArrayList[n];
    seen = new boolean[n];
    ap = new boolean[n];
    disc = new int[n];
    low = new int[n];
    parent = new int[n];
    time = 0;
    for(int i = 0; i < n; i++){
      g[i] = new ArrayList<Integer>();
      parent[i] = -1;
    }

    for(int u=0; u<n; u++){
      if(!seen[u]){
        articulationPoints(u);
      }
    }
    //Si ap[i]==true, 'i' es un punto de articulacion
  }

  private static void articulationPoints(int u){
    seen[u] = true;
    disc[u] = time;
    low[u] = time;
    time++;
    int children = 0;

    int len = g[u].size();
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(!seen[v]){
        children++;
        parent[v] = u;
        articulationPoints(v);
        low[u] = Math.min(low[u], low[v]);
        if(parent[u] == -1 && children > 1){
          ap[u] = true;
        }else if(parent[u] != -1 && low[v] >= disc[u]){
          ap[u] = true;
        }
      }else if(v != parent[u]){
        low[u] = Math.min(low[u], disc[v]);
      }
    }
  }
```

4.6. Puentes

Halla los puentes de un grafo. Un puente es una arista del grafo que si se quitara causaría que el grafo se "desconectara". Si el grafo no era conexo en un principio, un puente es una arista que, si se quitara, incrementaría el número de componentes conexas. La complejidad del algoritmo es $O(n + m)$.

```
class Bridge {
  public int u, v;
  public Bridge(int u, int v){
    this.u = u;
    this.v = v;
  }
}

public class GraphBridges {

  static ArrayList<Integer> g[];
  static boolean[] seen;
  static int[] disc, low, parent;
  static int time;
  static ArrayList<Bridge> bridgeEdges;

  public static void main(String[] args) {
    int n = 10;

    g = new ArrayList[n];
    seen = new boolean[n];
    disc = new int[n];
    low = new int[n];
    parent = new int[n];
    time = 0;
    bridgeEdges = new ArrayList<Bridge>();
    for(int i = 0; i < n; i++){
      g[i] = new ArrayList<Integer>();
      parent[i] = -1;
    }

    for(int u=0; u<n; u++){
      if(!seen[u]){
        bridges(u);
      }
    }
    //'bridgeEdges' contiene objetos tipo Bridge que indican que la arista u,v es un puente
  }

  private static void bridges(int u){
    seen[u] = true;
    disc[u] = time;
    low[u] = time;
    time++;

    int len = g[u].size();
    for(int i=0; i<len; i++){
      int v = g[u].get(i);
      if(!seen[v]){
        parent[v] = u;
        bridges(v);
        low[u] = Math.min(low[u], low[v]);
        if(low[v] > disc[u]){
          Bridge b = new Bridge(u, v);
          bridgeEdges.add(b);
        }
      }else if(v != parent[u]){
        low[u] = Math.min(low[u], disc[v]);
      }
    }
  }
}
```

4.7. Minimum Spanning Tree (Algoritmo de Kruskal)

Halla el árbol de cubrimiento mínimo de un grafo no-dirigido y conexo. Si no está garantizado de antemano que el grafo sea conexo, hay que verificarlo antes de correr este algoritmo.

Utiliza la estructura de datos Union-Find discutida anteriormente. El grafo debe ser representado como una lista de objetos tipo Arista. Tiene una complejidad de $O(m \log n)$.

```
//Se necesita implementar tambien la clase UnionFind

class Arista implements Comparable<Arista>{
  public int u, v, costo;
  public Arista(int u, int v, int costo){
    this.u = u;
    this.v = v;
    this.costo = costo;
  }
```

```java
  public int compareTo(Arista o) {
    return this.costo - o.costo;
  }
}

public class Kruskal {

  public static void main(String[] args) {
    int n = 10; //Cantidad de nodos del grafo
    ArrayList<Arista> aristas = new ArrayList<Arista>();
    int mst = kruskal(aristas, n);
  }

  public static int kruskal(ArrayList<Arista> aristas, int n){
    Collections.sort(aristas);

    UnionFind uf = new UnionFind(n);
    int costoMST = 0;
    int i = 0;
    while(uf.getComponents() != 1){
      Arista a = aristas.get(i);
      if(!uf.connected(a.u, a.v)){
        uf.union(a.u, a.v);
        costoMST += a.costo;
      }
      i++;
    }

    return costoMST;
  }
}
```

4.8. Algoritmo de Dijkstra

Halla la distancia más corta desde un nodo origen *src* hacia todos los demás nodos. Funciona con grafos dirigidos y no dirigidos, siempre y cuando los pesos de las aristas sean no-negativos. Se debe representar el grafo tanto en lista como en matriz de adyacencia. Su complejidad es $O(m + n \log n)$.

```java
class Nodo implements Comparable<Nodo>{
  int id, distancia;
  public Nodo(int id, int distancia){
    this.id = id;
    this.distancia = distancia;
  }
  public int compareTo(Nodo o) {
    return this.distancia-o.distancia;
  }
}

public class Dijkstra {

  static ArrayList<Integer> g[];
  static int[][] p;
  static int[] distancias, padre;
  static boolean[] visitado;
  static PriorityQueue<Nodo> proximo;

  public static void main(String[] args) {
    int n = 8;

    g = new ArrayList[n];
    p = new int[n][n];
    distancias = new int[n];
    padre = new int[n];
    visitado = new boolean[n];
    proximo = new PriorityQueue<Nodo>();
    for (int i=0; i<n; i++) {
      g[i] = new ArrayList<Integer>();
      distancias[i] = Integer.MAX_VALUE;
    }

    int src = 0;
    dijkstra(src);

    //El vector 'nodos' contiene la menor distancia de 'src' a todos los nodos
    //Por ejemplo, la menor distancia de 'src' a 4 es:
    int menorDist = distancias[4];

    //Para hallar el camino como tal entre 'src' y un nodo
    LinkedList<Integer> camino = new LinkedList<Integer>();
    int r = 4;
    camino.add(r);
    while(r != src){
      r = padre[r];
      camino.addFirst(r);
```

```java
    }
  }

  public static void dijkstra(int src){
    distancias[src] = 0;
    proximo.add(new Nodo(src, 0));
    padre[src] = src;

    while(!proximo.isEmpty()) {
      Nodo u = proximo.poll();
      if(!visitado[u.id]){
        visitado[u.id] = true;
        int len = g[u.id].size();
        for (int i=0; i<len; i++) {
          int v = g[u.id].get(i);
          if(u.distancia + p[u.id][v] < distancias[v]){
            distancias[v] = u.distancia + p[u.id][v];
            proximo.add(new Nodo(v, distancias[v]));
            padre[v] = u.id;
          }
        }
      }
    }
  }
}
```

4.9. Algoritmo de Floyd-Warshall

Halla la distancia más corta desde todos los nodos hacia todos los demás. El grafo debe estar representado en matriz de adyacencia, y puede tener aristas con peso negativo. Sin embargo, no puede tener ciclos de peso negativo. En caso de que exista un ciclo negativo, el algoritmo lo detectará. Si todas las artistas son no-negativas, se puede omitir la comprobación del ciclo negativo.

La matriz de adyacencia debe armarse así:

$$\text{grafo}(i,j) = \begin{cases} 0 & si \quad i = j \\ c_{i,j} & si \quad \text{existe una arista de } i \text{ a } j \text{ con costo } c_{i,j} \\ \infty & si \quad i \neq j \text{ y no existe arista de } i \text{ a } j \end{cases}$$

La complejidad del alrgoritmo es $O(n^3)$.

```java
  public static void main(String[] args) {
    int n = 5;
    int grafo[][] = new int[n][n];
    int A[][] = new int[n][n];

    for(int i=0; i<n; i++){
      for(int j=0; j<n; j++){
        if(i == j){
          grafo[i][j] = 0;
        }else{
          grafo[i][j] = Integer.MAX_VALUE;
        }
      }
    }

    //Tener la matriz del grafo llena antes de hacer esto
    for(int i=0; i<n; i++){
      for(int j=0; j<n; j++){
        A[i][j] = grafo[i][j];
      }
    }

    for(int k=0; k<n; k++){
      for(int i=0; i<n; i++){
        for(int j=0; j<n; j++){
          int option1 = A[i][j];
          int option2;
          if(A[i][k] == Integer.MAX_VALUE || A[k][j] == Integer.MAX_VALUE){
            option2 = Integer.MAX_VALUE;
          }else{
            option2 = A[i][k] + A[k][j];
          }
          A[i][j] = Math.min(option1, option2);
        }
      }
    }

    //Verificar si el grafo tiene un ciclo negativo
    boolean negativeCycle = false;
    for(int i=0; i<n && !negativeCycle; i++){
      if(A[i][i] < 0){
        negativeCycle = true;
      }
    }
  }
```

4.10. Máximo flujo y mínimo corte (Algoritmo de Edmonds-Karp)

Halla el máximo flujo que se puede emitir desde un origen s hacia un destino t, dado que los enlaces (aristas) tienen una capacidad dada.

El algoritmo de Edmonds-Karp es una implementación del método de Ford-Fulkerson que usa BFS para hallar los caminos en el grafo residual. Su complejidad es de $O(nm^2)$.

Variantes de este problema:

- Mínimo corte: El máximo flujo es igual al mínimo corte (esto es un teorema). Por ende, este algoritmo halla también el mínimo costo de cortar aristas de manera que s y t queden desconectados.
- Si hay varios orígenes $\{s_1, s_2, ...\}$, se pone un "super origen" s, y se agregan aristas (s, s_i) con capacidad infinita. Análogamente, si hay varios destinos $\{t_1, t_2, ...\}$, se agrega un "super destino" t y se agregan aristas (t_i, t) con capacidad infinita.
- Nodos con capacidad: Si el nodo u tiene también una capacidad c_u, se divide en dos nodos. Un nodo u_l que recibe todas las aristas que entran a u, y un nodo u_r del que salen todas las aristas que salen de u. Posteriormente, se agrega una arista (u_l, u_r) con capacidad c_u.

```java
static ArrayList<Integer> g[];
static int[] parent;
static int[][] cap;
static int[][] flow;

public static void main(String[] args) {
  Scanner sc = new Scanner(System.in);
  int n = sc.nextInt();
  int m = sc.nextInt();
  g = new ArrayList[n];
  for(int i = 0; i < n; i++){
    g[i] = new ArrayList<Integer>();
  }
  parent = new int[n];
  cap = new int[n][n];
  flow = new int[n][n];

  for(int i=0; i<m; i++){
    int u = sc.nextInt();
    int v = sc.nextInt();
    int c = sc.nextInt();
    g[u].add(v);
    // Siempre se agrega esta arista, aunque el grafo sea dirigido
    g[v].add(u);
    cap[u][v] = c;
    // La siguiente linea solo se agrega si el grafo es no-dirigido
    // cap[v][u] = c;
  }

  int s = sc.nextInt();
  int t = sc.nextInt();
  int max = maxFlow(s, t);
}

public static int maxFlow(int s, int t) {
  int ans = 0;
  while(true) {
    Arrays.fill(parent, -1);
    Queue<Integer> q = new LinkedList<Integer>();
    q.add(s);

    while(!q.isEmpty()) {
      int u = q.poll();
      if(u == t) break;

      int len = g[u].size();
      for(int i=0; i<len; i++){
        int v = g[u].get(i);
        if(parent[v] == -1 && cap[u][v] - flow[u][v] > 0) {
          q.add(v);
          parent[v] = u;
        }
      }
    }
    if(parent[t] == -1) break;

    int bottleneck = Integer.MAX_VALUE;
    int end = t;
    while(end != s) {
      int start = parent[end];
      bottleneck = Integer.min(bottleneck, cap[start][end] - flow[start][end]);
```

```
        end = start;
      }

      end = t;
      while(end != s) {
        int start = parent[end];
        flow[start][end] += bottleneck;
        flow[end][start] = -flow[start][end];
        end = start;
      }

      ans += bottleneck;
    }
    return ans;
  }
```

4.10.1. Máximo matching de un grafo bipartito

Un matching de un grafo es un subconjunto de aristas tal que en cada nodo incida máximo una de ellas. Si se trata de un grafo bipartito, encontrar el máximo matching (aquel con mayor cardinalidad) se puede modelar como un problema de máximo flujo:

Sea el grafo original $G = (V, E)$, donde $V = L \cup R$ (es decir, los vértices se separan en dos subconjuntos, ya que el grafo es bipartito). Se construye un nuevo grafo G', con los mismos vértices y aristas del grafo original. Se agregan a G' dos nuevos vértices s y t. Posteriormente se agregan aristas de (s, l_i) para los vértices $l_i \in L$, y aristas (u_i, t) para los vértices $u_i \in U$. Todas las aristas se ponen con capacidad 1.

El máximo matching en G es equivalente al máximo flujo entre s y t en G'.

Esta no es la forma más eficiente de resolver este problema, ya que hay algoritmos específicos para él que no lo modelan como un máximo flujo (por ejemplo, el algoritmo de Hopcroft-Karp).

5. KMP

Algoritmo para buscar una cadena *pattern* dentro de una cadena *text*. Su complejidad es de $O(m+n)$ donde m es la longitud de *text* y n es la longitud de *pattern*. Retorna la posición donde inicia la primera ocurrencia de *text* dentro de *pattern*, o -1 si no existe.

```
private static int[] computeTemporaryArray(String pattern
    ){
  int lps[] = new int[pattern.length()];
  int index = 0;
  int i = 1;
  while(i < pattern.length()){
    if(pattern.charAt(i) == pattern.charAt(index)){
      lps[i] = index + 1;
      index++;
      i++;
    }else{
      if(index != 0){
        index = lps[index-1];
      }else{
        lps[i] = 0;
        i++;
      }
    }
  }
  return lps;
}

public static int KMP(String text, String pattern){
  int lps[] = computeTemporaryArray(pattern);
  int i=0;
  int j=0;
  while(i < text.length() && j < pattern.length()){
    if(text.charAt(i) == pattern.charAt(j)){
      i++;
      j++;
    }else{
      if(j!=0){
        j = lps[j-1];
      }else{
        i++;
      }
    }
  }
  if(j == pattern.length()){
    return i-j;
  }
  return -1;
}

public static void main(String[] args) {
  String text = "ABABABABC";
  String pattern = "BABABC";
  int index = KMP(text, pattern);
```

```
}
```

6. Programación dinámica

6.1. Longest Increasing Subsequence

Halla la longitud de la subsecuencia creciente más larga que hay en un vector (o String). También halla los elementos que pertenecen a dicha subsecuencia, por si llega a ser necesario. Su complejidad es $O(n^2)$.

```java
public static void main(String[] args) {
  int array[] = {3, 4, -1, 0, 6, 2, 3};
  int n = array.length;

  int T[] = new int[n];
  int previous[] = new int[n];
  for(int i=0; i<n; i++){
    T[i] = 1;
    previous[i] = i;
  }

  for(int i=1; i<n; i++){
    for(int j=0; j<i; j++){
      if(array[i] > array[j]){
        if(T[j] + 1 > T[i]){
          T[i] = T[j] + 1;
          previous[i] = j;
        }
      }
    }
  }

  int maxIndex = 0;
  for(int i=0; i<n; i++){
    if(T[i] > T[maxIndex]){
      maxIndex = i;
    }
  }

  //Longitud de la LIS
  int lisLength = T[maxIndex];

  //La subsecuencia como tal
  int t = maxIndex;
  int newT = maxIndex;
  LinkedList<Integer> subsequence = new LinkedList<
    ↪ Integer>();
  do{
    t = newT;
    subsequence.addFirst(array[t]);
    newT = previous[t];
  }while(t != newT);
}
```

6.2. Longest Common Subsequence

Halla la longitud de la subsecuencia común más larga entre dos Strings (o vectores). También halla los elementos que pertenecen a dicha subsecuencia, por si llega a ser necesario. Su complejidad es $O(mn)$, donde m y n son las longitudes de los Strings.

```java
public static void main(String[] args) {
  String str1 = "ABCDGHLQR";
  String str2 = "AEDPHR";

  char x[] = str1.toCharArray();
  char y[] = str2.toCharArray();
  int T[][] = new int[x.length + 1][y.length + 1];

  for(int i=1; i<=x.length; i++){
    for(int j=1; j<=y.length; j++){
      if(x[i-1] == y[j-1]){
        T[i][j] = T[i-1][j-1] + 1;
      }else{
        T[i][j] = Math.max(T[i][j-1], T[i-1][j]);
      }
    }
  }

  //Longitud de la LCS
  int lcsLength = T[x.length][y.length];

  //La LCS como tal
  int i = x.length;
  int j = y.length;
  StringBuilder sb = new StringBuilder();
  while(i > 0 && j > 0){
    if(T[i][j] == T[i-1][j]){
```

```java
            i--;
        }else if(T[i][j] == T[i][j-1]){
            j--;
        }else{
            sb.append(x[i-1]);
            i--;
            j--;
        }
    }
    String lcs = sb.reverse().toString();
}
```

6.3. Edit Distance

Halla el número mínimo de operaciones necesarias para transformar un String en otro. Las operaciones permitidas son eliminar, insertar o modificar un caracter del String. La complejidad del algoritmo es $O(mn)$, donde m y n son las longitudes de los Strings.

```java
public static int editDistance(String original, String
    ↪ destination) {
  int sizex = destination.length();
  int sizey = original.length();
  int[][] changes = new int[sizex + 1][sizey + 1];

  for (int i = 0; i < Integer.max(sizex + 1, sizey + 1);
      ↪ i++) {
    if (i < sizex + 1) {
      changes[i][0] = i;
    }
    if (i < sizey + 1) {
      changes[0][i] = i;
    }
  }

  for (int i = 1; i < sizex + 1; i++) {
    for (int j = 1; j < sizey + 1; j++) {
      char x = destination.charAt(i - 1);
      char y = original.charAt(j - 1);
      if (x != y) {
        changes[i][j] = 1 + Integer.min(changes[i - 1][j
            ↪ ], Integer.min(changes[i][j - 1], changes[i
            ↪ - 1][j - 1]));
      } else {
        changes[i][j] = changes[i - 1][j - 1];
      }
    }
  }

  return changes[sizex][sizey];
}

public static void main(String[] args) {
  String original = "icpcsucks";
  String destination = "icpcrocks";
  int changes = editDistance(original, destination);
  System.out.println(changes);
}
```

6.4. Coin Change Problem

Se tienen monedas de n denominaciones diferentes. Se requiere encontrar el mínimo número de monedas tales que su valor sume exactamente W. Su complejidad es $O(nW)$.

```java
static int result;
static int[] coinsResult;

public static void coinChange(int sum, int[] denomination
    ↪ ) {
  int size = denomination.length;

  int[] minimal = new int[sum + 1];
  int[] coins = new int[sum + 1];

  Arrays.fill(minimal, Integer.MAX_VALUE);
  Arrays.fill(coins, -1);
  minimal[0] = 0;

  for (int i = 0; i < size; i++) {
    int coin = denomination[i];
    for (int j = coin; j < sum + 1; j++) {
      if (minimal[j - coin] != Integer.MAX_VALUE) {
        int choose = Integer.min(minimal[j], minimal[j -
            ↪ coin] + 1);
        if (choose < minimal[j]) {
          minimal[j] = choose;
          coins[j] = coin;
```

```java
          }
        }
      }
    }

    if(minimal[sum] == Integer.MAX_VALUE) {
      result = -1;
    }else {
      result = minimal[sum];

      // Si se quiere hallar exactamente cuales monedas se escogieron
      coinsResult = new int[result];
      int pointer = sum;
      for (int i = 0; i < result; i++) {
        coinsResult[i] = coins[pointer];
        pointer = pointer - coins[pointer];
      }
    }
  }

  public static void main(String[] args) {
    int sum = 800;
    int[] denominations = { 100, 200, 500 };
    coinChange(sum, denominations);
    if(result == -1) {
      // No es posible
    }else {
      System.out.println(result);
      for(int c : coinsResult) {
        System.out.println(c);
      }
    }
  }
}
```

6.5. El problema de la mochila (Knapsack)

Se tiene una mochila con capacidad W, y n items con un peso w_i y un valor v_i cada uno. Se quiere hallar el conjunto de items tal que la suma de sus pesos no exceda W, y que la suma de sus valores sea lo más grande posible. Su complejidad es $O(nW)$. Es posible indicar cuál es el mayor valor posible, y con un ciclo adicional, indicar exactamente cuáles items se seleccionaron.

```java
public static void main(String[] args) {
  int n = 4;
  int W = 8;
  int values[] = {15, 10, 9, 5};
  int weights[] = {1, 5, 3, 4};

  //Tener cuidado: En la matriz los items se numeran 1...n y la capacidad de la mochila 1...W
  int A[][] = new int[n+1][W+1];

  //Aca se resuelve el problema. Asegurarse de tener los values y weights
  for (int i=1; i<=n; i++) {
    for (int x=0; x<=W; x++) {
      if (weights[i-1] > x) {
        A[i][x] = A[i-1][x];
      } else {
        int p = A[i-1][x];
        int q = A[i-1][x-weights[i-1]] + values[i-1];
        A[i][x] = (p > q) ? p : q;
      }
    }
  }

  //El valor maximo que se puede obtener es A[n][W]
  int solution = A[n][W];

  //Si se quiere determinar cuales items se incluyeron
  boolean chosen[] = new boolean[n];
  int i = n;
  int j = W;

  while(i>0){
    if(A[i][j] == A[i-1][j]){
      i--;
    }else{
      chosen[i-1] = true;
      i--;
      j = j-weights[i];
    }
  }
  //Si chosen[i]==true es porque i se incluyo
}
```

7. Range Minimum Query

8. Teoría de números

8.1. Algoritmo de Euclides

Se utiliza para hallar el máximo común divisor (MCD) entre dos números. También se puede usar para hallar el mínimo común múltiplo (MCM).

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

//Dividir primero para evitar overflow en a*b
public static int mcm(int a, int b){
  return a * (b / mcd(a, b));
}
```

8.2. Verificar si un número es primo

Dependiendo del problema, puede que nos sirva la forma “fuerza bruta”. Esta forma tiene una complejidad de $O(\sqrt{n})$. Sin embargo, si tenemos números de más de 64 bits (que no caben en un *long*) ya esta forma no es viable.

La clase BigInteger provee un método probabilístico para determinar si un número es primo. Si el número es compuesto, el método retona *false* siempre. Si el método retorna *true*, hay una probabilidad de $1 - \frac{1}{2^x}$ de que el número sea primo, donde x es un parámetro que se le pasa a la función. Generalmente un valor de $x = 10$ está bien.

```
public static boolean isPrime(int x){
  if(x == 1) return false;
  if(x == 2) return true;
  if(x % 2 == 0) return false;

  int s = (int) Math.ceil(Math.sqrt(x));

  for(int i=3; i<=s; i+=2){
    if(x % i == 0) return false;
  }

  return true;
}

public static void main(String[] args) {
  isPrime(63); //true
  isPrime(99); //false

  //De hecho estos si caben en un int pero los pongo como ejemplo
  BigInteger a = new BigInteger("104723");
  BigInteger b = new BigInteger("104727");
  a.isProbablePrime(10); //true
  b.isProbablePrime(10); //false
}
```

8.3. Criba de Eratóstenes

Algoritmo para hallar los números primos menores o iguales a n. Su complejidad es $O(n \log \log n)$.

```
public static ArrayList<Integer> criba(int n){
  boolean marked[] = new boolean[n+1];
  marked[0] = true;
  marked[1] = true;
  ArrayList<Integer> primos = new ArrayList<Integer>();

  for(int i=2; i<=n; i++){
    if(!marked[i]){
      primos.add(i);
      //OJO: Si esta teniendo problemas de overflow, cambie la siguiente linea por j = 2*i
      int j = i*i;
      while(j <= n){
        marked[j] = true;
        j = j+i;
      }
    }
  }

  return primos;
}
```

8.4. Factorización prima de un número

Se busca expresar un número n como una multiplicación de factores primos, de la forma:

$$n = \prod p_i^{a_i} = p_1^{a_1} \cdot p_2^{a_2} \cdot p_3^{a_3} ... p_k^{a_k}$$

Previamente se debe hacer una Criba de Eratóstenes modificada. Verifique en la especificación de la entrada del problema cuál es el máximo número x que tendrá que factorizar, y haga la Criba hasta $\sqrt{x}$.

El método retorna un HashMap donde la *clave* es el factor primo p_i y el *valor* su multiplicidad a_i. Se puede modificar fácilmente para retornar una lista de todos los factores, o retornar la cantidad de factores.

Con algunas modificaciones, puede funcionar más o menos hasta $n = 10^{12}$.

```java
static int menorFactor[];
static ArrayList<Integer> primos;

public static void main(String[] args) {
  //Por ejemplo, si el mayor valor posible es 10000, se hace la criba hasta 100
  cribaFactores(100);
  HashMap<Integer, Integer> fac = factorizar(895);
  System.out.println(fac.toString());
}

public static void cribaFactores(int n){
  menorFactor = new int[n+1];
  Arrays.fill(menorFactor, -1);
  primos = new ArrayList<Integer>();

  for(int i=2; i<=n; i++){
    if(menorFactor[i] == -1){
      primos.add(i);
      //OJO: Si esta teniendo problemas de overflow, cambie la siguiente linea por j = 2*i
      int j = i*i;
      while(j <= n){
        if(menorFactor[j] == -1){
          menorFactor[j] = i;
        }
        j = j+i;
      }
    }
  }
}

public static HashMap<Integer, Integer> factorizar(int n) {
  HashMap<Integer, Integer> factores = new HashMap<Integer, Integer>();

  if(n >= menorFactor.length){
    for(int p : primos){
      if(n %p == 0){
        int count = 0;
        while(n %p == 0){
          n = n/p;
          count++;
        }
        factores.put(p, count);
      }
      if(n < menorFactor.length) break;
    }
    if(n >= menorFactor.length){
      factores.put(n, 1);
      return factores;
    }
  }

  while(n > 1){
    int f = menorFactor[n];
    if(f == -1){
      f = n;
    }
    if(factores.containsKey(f)){
      factores.put(f, factores.get(f)+1);
    }else{
      factores.put(f, 1);
    }
    n = n / f;
  }

  return factores;
}
```

8.5. Fórmulas

Para $n \geq 2$ es posible calcular algunas cosas partiendo de la factorización prima de n:

$$n = \prod p_i^{a_i} = p_1^{a_1} \cdot p_2^{a_2} \cdot p_3^{a_3} ... p_k^{a_k}$$

$n = 1$ es un caso especial:

$$d(1) = \sigma(1) = \varphi(1) = 1$$

8.5.1. Cantidad de divisores

$$d(n) = \prod (a_i + 1)$$

8.5.2. Suma de divisores

$$\sigma(n) = \prod \frac{p_i^{a_i+1} - 1}{p_i - 1}$$

Esta función toma todos los divisores. Por ejemplo, los divisores de 12 son $\{1, 2, 3, 4, 6, 12\}$. Por ende, $\sigma(12) = 1 + 2 + 3 + 4 + 6 + 12 = 28$. Si se quiere la suma de los divisores propios (es decir, los divisores excluyendo a n), basta con hallar:

$$s(n) = \sigma(n) - n$$

En el ejemplo anterior, $s(12) = 28 - 12 = 16$.

8.5.3. Función φ de Euler

Dos números son relativamente primos (o coprimos) si no tienen divisores en común (es decir, si su MCD es 1). $\varphi(n)$ se define como la cantidad de enteros positivos menores a n y coprimos a n.

$$\varphi(n) = \prod (p_i - 1) p_i^{a_i - 1}$$

9. Combinatoria

9.1. Permutaciones

Un conjunto de n elementos diferentes tiene $n!$ permutaciones. El número máximo cuyo factorial cabe en un *long* de 64 bits es $n = 20$. Más allá, se requiere usar BigInteger.

9.2. Subconjuntos

Un conjunto de n elementos tiene 2^n posibles subconjutos.

Si se busca generarlos, cada subconjunto puede representarse como un número b de n bits. El elemento k pertenece al subconjunto si el bit k de b está en 1. No es viable hacerlo para $n > 30$.

```java
public static void main(String[] args) {
  String elements[] = {"A", "B", "C", "D"};
  int n = elements.length;

  for(int b=0; b < (1 << n); b++){
    ArrayList<String> subset = new ArrayList<String>();
    for(int i=0; i<n; i++){
      if((b & (1 << i)) != 0){
        subset.add(elements[i]);
      }
    }
    System.out.println(subset.toString());
  }
}
```

9.3. Coeficientes binomiales

El número de subconjuntos de tamaño k de un conjunto de tamaño n, está dado por:

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

Sin embargo no es muy práctico usar esta fórmula ya que involucra factoriales. Se puede utilizar la recurrencia $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ para generar todos los coeficientes binomiales $\binom{i}{j}$ para $0 \leq i, j \leq n$.

```java
public static void main(String[] args) {
  int n = 10;
  long[][] C = new long[n][n];

  for(int i=0; i<n; i++){
    C[i][0] = 1;
    for(int j=1; j<=i; j++){
      C[i][j] = C[i-1][j-1] + C[i-1][j];
    }
  }
}
```

Este código funciona para $n \leq 67$. Más allá, se requiere usar BigInteger.

9.4. Multiconjuntos

Un multiconjunto es un conjunto que permite elementos repetidos. El número de multiconjuntos de cardinalidad k, con elementos tomados de un conjunto de cardinalidad n, se puede contar como:

$$\left(\!\!\binom{n}{k}\!\!\right) = \binom{k+n-1}{k}$$

También se puede usar una recurrencia:

$$\left(\!\!\binom{n}{0}\!\!\right) = 1, \quad \left(\!\!\binom{0}{k}\!\!\right) = 0 \text{ para } k > 0$$

$$\left(\!\!\binom{n}{k}\!\!\right) = \left(\!\!\binom{n}{k-1}\!\!\right) + \left(\!\!\binom{n-1}{k}\!\!\right)$$

9.5. Particiones

Una partición de un conjunto es una colección de subconjuntos disjuntos, tales que la unión de todos ellos es el conjunto original. La cantidad de maneras diferentes de particionar un conjunto de n elementos en k partes se denota como $S(n,k)$ (números de Stirling de tipo 2).

$$S(n,k) = \begin{cases} 1 & n = 0 \\ 1 & k = 0 \\ S(n-1,k-1) + k \cdot S(n-1,k) & n \geq 1, k \geq 1 \end{cases}$$

9.6. Desarreglos

Un desarreglo es una permutación que no “mapea” ningún elemento a sí mismo. Por ejemplo, 231 es un desarreglo de 123, pero 132 no lo es (ya que el 1 queda en la misma posición).

El número de desarreglos de un conjunto de n elementos es:

$$D_n = n! \cdot \sum_{k=0}^{n} \frac{(-1)^k}{k!} \approx \frac{n!}{e}$$

Puede ser más práctico usar la siguiente recurrencia:

$$D_n = \begin{cases} 1 & n = 0 \\ 0 & n = 1 \\ (n-1)(D_{n-1} + D_{n-2}) & n \geq 2 \end{cases}$$

El máximo D_n que cabe en un *long* es con $n = 20$.

9.7. Números de Catalan

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

Sin embargo, puede ser más práctico usar la siguiente recurrencia:

$$C_0 = 1, \quad C_n = \frac{2(2n-1)C_{n-1}}{(n+1)}$$

El número C_n tiene muchas interpretaciones. Entre ellas:

- El número de árboles binarios diferentes con n vértices
- El número formas de hacer una expresión con n pares de paréntesis balanceados
- El número de formas de triangular un polígono con $n+2$ lados
- El número de caminos monótonos que no pasan sobre la diagonal en una cuadrícula de $n \cdot n$.

El C_n más grande que cabe en un *long* es con $n = 33$.

10. Otros

10.1. Ordenamiento de Arrays y Listas

Cuando necesite ordenar un vector o una lista, utilice los métodos .sort() que tiene Java. El algoritmo que utilizan es QuickSort y su complejidad es $O(n \log n)$.

```java
public static void main(String[] args) {
  int n = 10;
  String v[] = new String[n];
  ArrayList<Integer> l = new ArrayList<Integer>();

  Arrays.sort(v);
  Collections.sort(l);
  //Collections.sort() tambien ordena LinkedList
}
```

10.2. Cola de Prioridad

Cola en la que la cabeza siempre es el menor elemento presente. Las operaciones add() y poll() son $O(\log n)$. La operación peek() es $O(1)$.

También se puede invertir, haciendo que la cabeza siempre sea el mayor elemento, pasando el parámetro Collections.reverseOrder().

```java
public static void main(String[] args) {
  PriorityQueue<Integer> pq = new PriorityQueue<Integer>();
  pq.add(3);
  pq.add(5);
  pq.add(1);
  pq.peek(); //No saca el 1 de la cola
  pq.poll(); //Saca el 1 de la cola y el 3 queda en la cabeza

  //Orden inverso
  PriorityQueue<Integer> pqInversa = new PriorityQueue<Integer>(Collections.reverseOrder());
  pqInversa.add(3);
  pqInversa.add(1);
  pqInversa.poll(); //Saca el 3
}
```

10.3. Interfaz Comparable

En ocasiones se puede necesitar ordenar un vector o lista de un tipo de datos definido por el usuario (clase), o utilizar una cola de prioridad. Para hacer esto, la clase debe implementar la interfaz Comparable de Java.

El método compareTo() debe retornar:

- Negativo si *this* $<$ *obj*.
- 0 si *this* $=$ *obj*.
- Positivo si *this* $>$ *obj*.

Por ejemplo, se tiene una clase Persona con dos campos: nombre y edad. Se quiere ordenar una lista de objetos tipo Persona de menor a mayor edad.

```java
class Persona implements Comparable<Persona>{
  public String nombre;
  public int edad;
  public Persona(String nombre, int edad){
    this.nombre = nombre;
    this.edad = edad;
  }
  public int compareTo(Persona obj) {
    return this.edad - obj.edad;
  }
}

class Main {
  public static void main(String[] args) {
    ArrayList<Persona> p = new ArrayList<Persona>();
    p.add(new Persona("Carlos", 20));
    p.add(new Persona("Juan", 20));
    Collections.sort(p);
  }
}
```

10.4. Imprimir números decimales redondeados

Generalmente basta con esta función de Java para redondear correctamente números decimales.

```java
public static void main(String[] args) {
  double d = 9.2651659;
  //Por ejemplo, para redondear a 4 decimales
  System.out.format("%.4f\n", d);
  //Hay que poner el \n si se quiere imprimir tambien un salto de linea
}
```

10.5. BufferedReader y BufferedWriter

Scanner es sencillo de utilizar pero es lento. Se recomienda utilizar siempre *BufferedReader* para leer entradas.

En algunas ocasiones también se necesitará un modo más rápido que *System.out.println()* para imprimr. *BufferedWriter* es más rápido, nunca está de más usarlo.

```java
public static void main(String[] args) throws IOException {
  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  //Solo lee por lineas
  //Ciclo hasta end of input
  String s;
  while((s = br.readLine()) != null){
```

```java
    String l[] = s.split(" ");
  }

  //Ciclo con numero de casos
  int t = Integer.parseInt(br.readLine());
  for(int i=0; i<t; i++){
    String l[] = br.readLine().split(" ");
  }

  BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));
  //No pone un salto de linea al final. Por tanto, se debe poner \n cuando sea necesario
  bw.write("Hola mundo\n");
  //El flush es el que realmente imprime en consola. En lo posible, hacer flush solo una vez, al final de procesar todos los casos
  bw.flush();
}
```