

# Guía del programador competitivo

*Programación  
competitiva para  
todos*

Diego Fernando Rodriguez

Castañeda

# 1 Tabla de contenido

<b>1</b>	<b>Tabla de contenido .....</b>	<b>1</b>
<b>2</b>	<b>Introducción.....</b>	<b>11</b>
<b>3</b>	<b>Información previa .....</b>	<b>12</b>
3.1	Reglas básicas.....	12
3.2	¿Qué es un problema de programación competitiva? .....	13
3.3	Estructura de un problema de programación competitiva .....	13
3.4	Como abordar un problema .....	14
3.5	Códigos dentro del libro.....	16
3.6	Manejo de archivos de código fuente.....	16
3.7	Casos de prueba y tipos de casos de prueba .....	17
3.8	Terminología básica .....	17
3.9	Librerías y funciones prohibidos.....	20
3.10	10 pasos para resolver cualquier problema .....	21
<b>4</b>	<b>Acerca de los lenguajes.....</b>	<b>23</b>
4.1	Diferencias generales de los lenguajes .....	25
<b>5</b>	<b>Maratones de programación.....</b>	<b>26</b>
<b>6</b>	<b>ACIS/REDIS y CCPL .....</b>	<b>27</b>
<b>7</b>	<b>ACM e ICPC.....</b>	<b>29</b>
<b>8</b>	<b>Google CodeJAM, HashMap y Kickstart .....</b>	<b>32</b>
<b>9</b>	<b>CodeChef.....</b>	<b>33</b>
<b>10</b>	<b>Online Judge.....</b>	<b>34</b>

<b>11</b>	<b>Comenzando con lenguajes de programación.....</b>	<b>36</b>
<b>11.1</b>	<b>Operadores lógicos, comparativos y matemáticos .....</b>	<b>36</b>
<b>11.2</b>	<b>Tipos de variables básicas .....</b>	<b>36</b>
<b>11.3</b>	<b>Lectura e impresión .....</b>	<b>37</b>
<b>11.4</b>	<b>Condicionales y ciclos .....</b>	<b>41</b>
<b>11.5</b>	<b>Control de excepciones .....</b>	<b>44</b>
<b>11.6</b>	<b>Complejidad de tiempo (Eficiencia Algorítmica).....</b>	<b>45</b>
<b>11.7</b>	<b>¿Qué afecta la complejidad de tiempo? .....</b>	<b>47</b>
<b>11.8</b>	<b>Conteo de tiempo de ejecución.....</b>	<b>48</b>
<b>11.9</b>	<b>String Matching y expresiones regulares .....</b>	<b>48</b>
<b>12</b>	<b>Estructuras de datos básicas .....</b>	<b>50</b>
<b>13</b>	<b>Identidades matemáticas y geométricas .....</b>	<b>61</b>
<b>14</b>	<b>Algoritmos de Búsquedas .....</b>	<b>63</b>
<b>14.1</b>	<b>Binary Search.....</b>	<b>63</b>
<b>14.2</b>	<b>Exponential Search .....</b>	<b>64</b>
<b>14.3</b>	<b>Array Max/Min Search.....</b>	<b>65</b>
<b>14.4</b>	<b>Fibonacci Search .....</b>	<b>66</b>
<b>14.5</b>	<b>Jump Search .....</b>	<b>68</b>
<b>14.6</b>	<b>Minimum Absolute Sum Pair.....</b>	<b>71</b>
<b>14.7</b>	<b>Missing Number Search .....</b>	<b>72</b>
<b>14.8</b>	<b>Difference Pair Search.....</b>	<b>72</b>
<b>14.9</b>	<b>Ternary Search.....</b>	<b>74</b>
<b>14.10</b>	<b>Problemas de repaso .....</b>	<b>75</b>

<b>15 Algoritmos de Ordenamiento.....</b>	<b>76</b>
<b>15.1 Recursive Bubble Sort .....</b>	<b>76</b>
<b>15.2 Quick Sort.....</b>	<b>77</b>
<b>15.3 Radix Sort.....</b>	<b>80</b>
<b>15.4 Shell Sort .....</b>	<b>81</b>
<b>15.5 Tim Sort.....</b>	<b>83</b>
<b>15.6 Tree Sort.....</b>	<b>86</b>
<b>15.7 Merge Sort .....</b>	<b>88</b>
<b>15.8 Binary Insertion Sort.....</b>	<b>91</b>
<b>15.9 Bitonic Sort.....</b>	<b>92</b>
<b>15.10 Cocktail Sort.....</b>	<b>94</b>
<b>15.11 Comb Sort .....</b>	<b>96</b>
<b>15.12 Counting Sort .....</b>	<b>97</b>
<b>15.13 Cycle Sort .....</b>	<b>99</b>
<b>15.14 3 Way Merge Sort.....</b>	<b>100</b>
<b>15.15 Pigeon Hole Sort.....</b>	<b>103</b>
<b>15.16 Problemas de repaso .....</b>	<b>104</b>
<b>16 Patrones y manejo de Strings.....</b>	<b>105</b>
<b>16.1 Aho-Corasick Algorithm .....</b>	<b>105</b>
<b>16.2 Anagrams Pattern .....</b>	<b>109</b>
<b>16.3 Boyer-Moore Algorithm.....</b>	<b>111</b>
<b>16.4 Finite Automata.....</b>	<b>114</b>
<b>16.5 Knuth-Morris-Pratt Matching.....</b>	<b>116</b>

16.6	Manacher Algorithm.....	119
16.7	Rabin-Karp Algorithm .....	121
16.8	Wildcard.....	124
16.9	Problemas de repaso .....	127
17	Matemática .....	128
17.1	GCD/LCM.....	128
17.2	Primality Test .....	129
17.3	Prime Factors.....	130
17.4	Divisibility By Small Numbers In Interval [1-12] .....	131
17.5	Number Of Divisors.....	133
17.6	Prime Numbers (Criba de Eratóstenes) .....	134
17.7	Fibonacci Last Digit .....	135
17.8	Long Fibonacci .....	136
17.9	Is Fibonacci Number?.....	137
17.10	All Permutations .....	137
17.11	Combinatory .....	139
17.12	Compose Combinatory .....	140
17.13	All Subsets Of a Set.....	140
17.14	Hanoi Towels.....	142
17.15	AX+BY=N .....	143
17.16	A%X=B .....	144
17.17	Factorial .....	145
17.18	Big Factorial .....	146

17.19	Binomial Coefficients.....	147
17.20	Cassini Identity.....	148
17.21	E pow X .....	148
17.22	Euclid-Euler Theorem .....	149
17.23	Euclidean Algorithm .....	151
17.24	Euler Totient .....	152
17.25	Number of Digits Factorial .....	152
17.26	Number of Digits Factorial Optimized .....	154
17.27	Fermat Little Theorem .....	155
17.28	Fibonacci Golden Ratio .....	156
17.29	Float Great Common Divisor.....	157
17.30	Fractions Product .....	158
17.31	Josephus .....	160
17.32	Cardinal/Ordinal Numbers.....	162
17.33	Roman Numbers .....	163
17.34	Multiple Great Common Divisor .....	165
17.35	Hardy-Ramanujan Theorem.....	166
17.36	Hoax Number .....	167
17.37	The Largest Pow That Divides a Factorial .....	169
17.38	Lower Common Multiplier in Array.....	170
17.39	Leonardo Number .....	172
17.40	Modular Exponentiation.....	172
17.41	Multiple Euler Totient .....	174

<b>17.42</b>	<b>Natural Coprime Sumatory .....</b>	<b>175</b>
<b>17.43</b>	<b>Zeckendorf Theorem .....</b>	<b>177</b>
<b>17.44</b>	<b>Rosser Theorem .....</b>	<b>178</b>
<b>17.45</b>	<b>Sieve Of Eratosthenes O(n) .....</b>	<b>180</b>
<b>17.46</b>	<b>Smith Number .....</b>	<b>181</b>
<b>17.47</b>	<b>Sphenic Number.....</b>	<b>184</b>
<b>17.48</b>	<b>Catalan Numbers.....</b>	<b>185</b>
<b>17.49</b>	<b>Charmichael Numbers .....</b>	<b>186</b>
<b>17.50</b>	<b>Stern-Brocot Sequence .....</b>	<b>187</b>
<b>17.51</b>	<b>Sylvester Sequence.....</b>	<b>189</b>
<b>17.52</b>	<b>Aliquot Sequence .....</b>	<b>190</b>
<b>17.53</b>	<b>Juggler Sequence.....</b>	<b>191</b>
<b>17.54</b>	<b>Moser de Bruijn Sequence .....</b>	<b>192</b>
<b>17.55</b>	<b>Newman-Conway Sequence .....</b>	<b>194</b>
<b>17.56</b>	<b>Padovan Sequence .....</b>	<b>195</b>
<b>17.57</b>	<b>Recaman Sequence .....</b>	<b>196</b>
<b>17.58</b>	<b>Problemas de repaso .....</b>	<b>197</b>
<b>18</b>	<b>Geometría .....</b>	<b>198</b>
<b>18.1</b>	<b>Arc Length .....</b>	<b>198</b>
<b>18.2</b>	<b>Area Of A Circular Sector.....</b>	<b>199</b>
<b>18.3</b>	<b>Arranged Coin Triangle .....</b>	<b>200</b>
<b>18.4</b>	<b>Circle Area .....</b>	<b>201</b>
<b>18.5</b>	<b>Circumscribed Circle Of Equilateral Triangle .....</b>	<b>201</b>

<b>18.6</b>	<b>Convex Hull .....</b>	<b>202</b>
<b>18.7</b>	<b>Cuts Sumatory .....</b>	<b>205</b>
<b>18.8</b>	<b>Heron Formula.....</b>	<b>205</b>
<b>18.9</b>	<b>Three Point Circle Radious .....</b>	<b>206</b>
<b>18.10</b>	<b>Dimensions Right Angled Triangle.....</b>	<b>206</b>
<b>18.11</b>	<b>Hexagon Area.....</b>	<b>207</b>
<b>18.12</b>	<b>Area Of Polygon .....</b>	<b>208</b>
<b>18.13</b>	<b>Line to Line Intersection .....</b>	<b>210</b>
<b>18.14</b>	<b>Line Mid Point .....</b>	<b>212</b>
<b>18.15</b>	<b>Line Through Two Points .....</b>	<b>213</b>
<b>18.16</b>	<b>Minimum Height Triangle .....</b>	<b>214</b>
<b>18.17</b>	<b>Perimeter Using Blocks.....</b>	<b>215</b>
<b>18.18</b>	<b>Point Inside Triangle.....</b>	<b>217</b>
<b>18.19</b>	<b>Ratio Divide Line .....</b>	<b>218</b>
<b>18.20</b>	<b>Rectangles In NxM.....</b>	<b>219</b>
<b>18.21</b>	<b>Squares In Triangle 2x2.....</b>	<b>219</b>
<b>18.22</b>	<b>Sum Of Manhattan .....</b>	<b>220</b>
<b>18.23</b>	<b>Three Points Collinear .....</b>	<b>221</b>
<b>18.24</b>	<b>Triangle Angles.....</b>	<b>222</b>
<b>18.25</b>	<b>Problemas de repaso .....</b>	<b>223</b>
<b>19</b>	<b>Manejo binario BitWise .....</b>	<b>225</b>
<b>19.1</b>	<b>Add One .....</b>	<b>225</b>
<b>19.2</b>	<b>Binary Palindrome .....</b>	<b>225</b>

<b>19.3</b>	<b>Booth Multiply.....</b>	<b>227</b>
<b>19.4</b>	<b>Closest Number With Same Bits .....</b>	<b>230</b>
<b>19.5</b>	<b>Gray Code to Binary /Binary To Gray Code .....</b>	<b>233</b>
<b>19.6</b>	<b>Count A to B .....</b>	<b>236</b>
<b>19.7</b>	<b>Count Sets Of Bits .....</b>	<b>237</b>
<b>19.8</b>	<b>Euclid No Mod And Divide.....</b>	<b>237</b>
<b>19.9</b>	<b>Find Duplicates In Bit Array.....</b>	<b>238</b>
<b>19.10</b>	<b>Gray Codes Of N .....</b>	<b>240</b>
<b>19.11</b>	<b>Karatsuba Multiply .....</b>	<b>241</b>
<b>19.12</b>	<b>Max Consecutive One.....</b>	<b>243</b>
<b>19.13</b>	<b>Max Subarray XOR .....</b>	<b>245</b>
<b>19.14</b>	<b>Multiply By Seven.....</b>	<b>245</b>
<b>19.15</b>	<b>Magic Number Of N.....</b>	<b>246</b>
<b>19.16</b>	<b>N Square No Pow .....</b>	<b>247</b>
<b>19.17</b>	<b>Odd Even Swap .....</b>	<b>248</b>
<b>19.18</b>	<b>Odd Ocurrence .....</b>	<b>250</b>
<b>19.19</b>	<b>Opposite Signs.....</b>	<b>251</b>
<b>19.20</b>	<b>Pandigital Concatenations .....</b>	<b>251</b>
<b>19.21</b>	<b>Power Of Four .....</b>	<b>253</b>
<b>19.22</b>	<b>Russian Peasant Multiply.....</b>	<b>254</b>
<b>19.23</b>	<b>Problemas de repaso .....</b>	<b>255</b>
<b>20</b>	<b>Grafos y arboles.....</b>	<b>256</b>
<b>20.1</b>	<b>Matrix of Adyacency .....</b>	<b>256</b>

20.2	List of Adyacency .....	257
20.3	BFS (Breath First Search) .....	259
20.4	BFS All Paths .....	261
20.5	Binary Graph Shortest Path.....	263
20.6	Count All Paths .....	265
20.7	Cycle In A Directed Graph .....	267
20.8	DFS (Depth First Search).....	269
20.9	DFS Transitive Clousure.....	272
20.10	Disconnected Graph BFS.....	273
20.11	Eulerian Cycle Directed Graph.....	275
20.12	Eulerian Path Undirected Graph .....	278
20.13	Floyd-Warshall All Pairs Shortest Paths.....	282
20.14	Hamiltonian Cycle .....	284
20.15	Kahn Algorithm For Topological Sorting .....	288
20.16	Knight Tour Backtraking .....	290
20.17	Kosaraju DFS Strongly Connected Graph .....	293
20.18	Kruskal Minimum Spanning Tree .....	296
20.19	Minimum Moves Knight .....	301
20.20	N Queen Problem .....	302
20.21	Prim Minimum Spanning Tree .....	305
20.22	Topological Sorting.....	311
20.23	Union Find.....	312
20.24	Domino DFS .....	315

20.25	Exit The Maze BFS .....	317
20.26	Bellman-Ford Shortest Path .....	321
20.27	Dijkstra Shortest Path.....	324
20.28	Bipartite Graph.....	328
20.29	Problemas de repaso .....	330
21	Otros algoritmos y programación dinámica .....	332
21.1	Knapsack 0/1 .....	332
21.2	Coin Exchange .....	334
21.3	Longest Increasing Subsequence .....	335
21.4	Longest Common SubString .....	337
21.5	Pascal Triangle.....	338
21.6	Problemas de repaso .....	341
22	Lista de tablas.....	342
23	Lista de ilustraciones .....	343
24	Bibliografía.....	346



## 2 Introducción

Las competencias de programación a nivel mundial han demostrado ser una valiosa herramienta para el aprendizaje y desarrollo de habilidades relacionadas a la programación en función de resolución de problemas, muchas de estas competencias emplean diversos aplicativos webs para el entrenamiento y evaluación de los estudiantes.

Es un concurso de programación algorítmica para estudiantes universitarios. Equipos de uno a tres integrantes, que representan a su universidad, trabajan para resolver los problemas de la vida real, fomentando la colaboración, la creatividad, la innovación y la capacidad de actuar bajo presión. A través del entrenamiento y la competencia, los equipos se desafían unos a otros para elevar la presión y la rapidez con el cual realizan los problemas.

## **3 Información previa**

### **3.1 Reglas básicas**

Durante las competencias, dependiendo de la modalidad de la misma, existen ciertas reglas que se deben cumplir para mantener la igualdad entre los competidores y evitar cualquier tipo de trampas.

#### **Competencias presenciales:**

- Durante las competencias está prohibido el uso de elementos electrónicos tales como celulares, dispositivos de almacenamiento extraíbles, reproductores de música, auriculares, entre otros. De ser encontrado con alguno de estos, el participante será retirado del recinto y perderá el derecho a continuar en la competencia por intento de fraude.
- El ingreso a páginas web distintas a la plataforma oficial de la competencia será penalizado con la descalificación inmediata de la competencia.
- El consumo de bebidas y alimentos dentro de las aulas no está permitido.
- Durante la competencia los equipos no podrán hablar ni intercambiar material entre ellos. De ser sorprendidos por primera vez incumpliendo esta norma, se les hará una advertencia pero si ya es la segunda vez serán descalificados de la competencia.
- La copia de código de fuentes externas como páginas web o redes sociales se considera plagio.
- Está permitido el uso de material como libros de texto, códigos impresos, textos matemáticos, diccionarios, entre otros.
- Cada equipo tendrá a su disposición una sola máquina durante la competencia.
- En la competencia los equipos serán conformados por máximo tres competidores.

#### **Competencias virtuales:**

- Durante la competencia los equipos no podrán hablar ni intercambiar material entre ellos. De ser sorprendidos por primera vez incumpliendo esta norma, se les hará una advertencia pero si ya es la segunda vez serán descalificados de la competencia.
- En la competencia los equipos serán conformados por máximo tres competidores.
- La copia de código de fuentes externas como páginas web o redes sociales se considera plagio.
- No se admite cualquier tipo de ataques informáticos que intenten vulnerar la integridad de los sistemas usados en las competencias.

Durante la mayoría de las competencias, la forma de entrada y salida de información de cada ejercicio se realiza por medio de la consola (STDIN/STDOUT), aunque algunas competencias si utilizan entornos gráficos y entornos web (Uva Online Judge, 2012).

## 3.2 ¿Qué es un problema de programación competitiva?

Un problema de programación competitiva es un pequeño ejercicio de programación que puede abordar cualquier tema de la vida cotidiana, ciencias puras como la matemática o la física, un juego de mesa como el ajedrez o un rompecabezas.

Utilizando la programación se pueden resolver estos ejercicios y buscar una solución óptima al mismo utilizando matemática, algoritmos matemáticos especializados o incluso lógica básica.

## 3.3 Estructura de un problema de programación competitiva

Un problema de estos se conforma de 4 partes principales:

1. Título: Contiene el nombre del problema, comúnmente da cierta aproximación al tema del problema

2. Descripción: Esta parte trae toda la información del problema, puede ser una historia, un evento, algo que le pasando a alguien y quiere que lo ayude a resolverlo. Puede contener algunas pistas de cómo resolver el ejercicio, variables que se pueden usar y de qué tipo pueden ser.
3. Variables: muestra el tipo de dato que entrara y el tipo de dato que debe salir, además del tamaño que puede tener la entrada.
4. Ejemplo de entrada y salida: Muestra uno de todos los posibles casos de prueba que pueden evaluarse en el ejercicio, tener en cuenta que los ejemplos en los textos de ejercicio nunca serán los únicos, se debe usar la parte de variables para calcular cual será el caso más fácil, un caso intermedio y el caso más difícil.

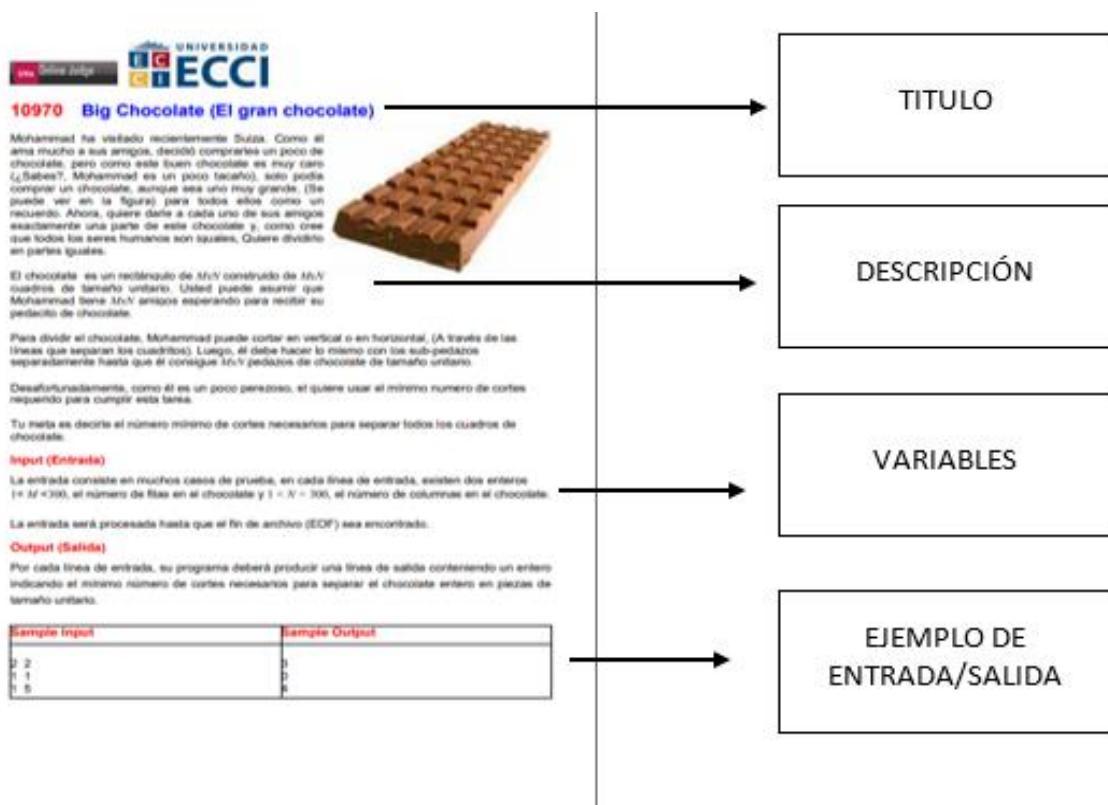


Ilustración 3-1: Partes principales de un problema de programación competitiva

## 3.4 Como abordar un problema

Comúnmente todos los ejercicios tienen dos o más formas de ser solucionados, utilizando diversos tipos de algoritmos o metodologías matemáticas de resolución de problemas, a continuación nombraremos algunas de las formas existentes para abordar un problema:

- Fórmulas matemáticas
- Fuerza brutal
- Aproximaciones de raíces
- Procesamiento de Strings
- Búsqueda de patrones
- Uso de estructuras de datos especializadas
- Algoritmos prediseñados modificados
- Enfoque geométrico
- Algoritmos ingenuos
- Uso de estructuras geometrías avanzadas
- Búsquedas
- Ordenamientos

Al leer toda la información de nuestro ejercicio, nuestro deber como competidores es identificar cual es la metodología más adecuada para la resolución de estos ejercicios, podemos apoyarnos de material previamente consultado. Entre cada metodología de resolución varia el uso de memoria, tiempo de ejecución y forma de entrada de la información a procesar, por lo que conocer los puntos fuertes y débiles de cada uno es crucial.

Por lo general las metodologías matemáticas son las más rápidas pero también suelen ser las más complicadas de implementar, ya que se basan en la búsqueda de sistemas de ecuaciones o el uso de teoremas matemáticos que son complicados y abstractos de entender, las metodologías de fuerza bruta son las más lentas pero las más seguras a la hora de encontrar una solución a un problema debido a que usan gran cantidad de recursos para asegurarse de dar con la respuesta, además de obtener las demás respuestas posibles a un problema cuando sus variables cambian.

## 3.5 Códigos dentro del libro

Los códigos y algoritmos que se encuentran dentro de este documento han sido investigados y basados en diferentes textos matemáticos y páginas web especializadas en programación competitiva como por ejemplo GeeksForGeeks®, LetCode® y Algorithmist®, los cuales sus códigos son de uso público y no contienen derechos de autor, han sido probados en distintos entornos de desarrollo, también en máquinas con distintas capacidades de procesamiento, cada código contiene comentarios sobre el uso o funcionalidad de cada sentencia, con el fin de que el usuario comprenda el fin de estas.

Cada código es la aproximación más cercana de cada algoritmo, y puede tener formas de ser optimizado, por lo que su efectividad no es del 100% para todos los casos, así que siempre se debe tener en cuenta que existe un margen de error, y debemos buscar medios para mitigarlos, ese es el fin de las competencias de programación, probar nuestra capacidad de resolución de todo tipo de problemas, ver con qué medios nos enfrentamos a ellos.

## 3.6 Manejo de archivos de código fuente

El nombre de los archivos fuentes es Main, teniendo en cuenta la extensión de cada lenguaje. Solo debe enviar el archivo fuente del ejercicio, no todo el proyecto.

Lenguaje	Nombre del archivo
JAVA	Main.java
PYTHON	Main.py
C++	Main.cpp
NOTA: En el caso específico de JAVA no se usa Package, coloque su archivo fuente en el default package	
NOTA: En algunas competencias, el nombre del archivo fuente tiene que ser como las instrucciones de la misma diga. Ejemplo: A.java	

Tabla 3-1 Ejemplo de nombres de archivo fuente validos

## 3.7 Casos de prueba y tipos de casos de prueba

Existen 4 tipos de casos de prueba:

Tipo de caso	Descripción	Ejemplo
Caso único	Solo se utilizará una entrada por problema	$N == 1$
Caso bandera	El ejercicio seguirá ejecutándose hasta que una condición se cumpla	Mientras $N$ sea diferente de 0
Caso fijo	Hay un número limitado de casos y el programa se ejecutara hasta que se cumpla este número	Mientras $N$ sea menor a 10 Acumulando 1 por ciclo
Caso ilimitado o fin de archivo	El ejercicio seguirá leyendo y procesando entradas hasta que se encuentre el banderín EOF, se considera ese tipo como ilimitado.	Mientras siga habiendo entradas

Tabla 3-2: Tipos de casos de prueba

Los ejercicios tienen un tiempo límite de ejecución, si al enviar el ejercicio al juzgador la respuesta es TIME LIMIT, su algoritmo no es lo suficientemente rápido para resolver el problema, intente reducir ciclos, mejorar la entrada de datos y usar algoritmos mejor optimizados.

## 3.8 Terminología básica

**Array:** Un arreglo es un conjunto de datos o una estructura de datos homogéneos que se encuentran ubicados en forma consecutiva en la memoria RAM, sirve para almacenar datos en forma temporal.

**Set:** Un set o conjunto es una colección (contenedor) de ciertos valores, sin ningún orden concreto ni valores repetidos. Su correspondencia en las matemáticas sería el conjunto finito.

**Variable:** Una variable está formada por un espacio en el sistema de almacenaje comúnmente en la memoria principal de un ordenador y un nombre simbólico el cual sirve identificador que está asociado a dicho espacio. Ese espacio contiene una cantidad de información conocida o desconocida, es decir un valor. El nombre de la variable es la forma usual de referirse al valor almacenado: esta separación entre nombre y contenido permite que el nombre sea usado independientemente de la información exacta que representa.

**Número:** Un número, es una abstracción que representa una cantidad o una magnitud. En matemáticas un número puede representar una cantidad métrica o más generalmente un elemento de un sistema numérico o un número ordinal que representará una posición dentro de un orden de una serie determinada.

**Algoritmo:** Conjunto ordenado de operaciones sistemáticas que permite hacer un cálculo y hallar la solución de un tipo de problemas.

**Programación:** La programación es un proceso que se utiliza para idear y ordenar las acciones que se realizarán en el marco de un proyecto; al anuncio de las partes que componen un acto o espectáculo; a la preparación de máquinas para que cumplan con una cierta tarea en un momento determinado; a la elaboración de programas para la resolución de problemas mediante ordenadores, y a la preparación de los datos necesarios para obtener una solución de un problema

**Tiempo de ejecución:** Se denomina tiempo de ejecución (runtime en inglés) al intervalo de tiempo en el que un programa de computadora se ejecuta en un sistema operativo. Este tiempo se inicia con la puesta en memoria principal del programa, por lo que el sistema operativo comienza a ejecutar sus instrucciones. El intervalo finaliza en el momento en que éste envía al sistema operativo la señal de terminación, sea ésta una terminación normal,

en que el programa tuvo la posibilidad de concluir sus instrucciones satisfactoriamente, o una terminación anormal, en el que el programa produjo algún error y el sistema debió forzar su finalización.

**Memoria:** La memoria es el dispositivo que retiene, memoriza o almacena datos informáticos durante algún período de tiempo. La memoria proporciona una de las principales funciones de la computación moderna: el almacenamiento de información y conocimiento. Es uno de los componentes fundamentales de la computadora, que interconectados a la unidad central de procesamiento y los dispositivos de entrada/salida.

**Compilador:** Un compilador es un tipo de traductor que transforma un programa entero de un lenguaje de programación (llamado código fuente) a otro. Usualmente el lenguaje objetivo es código máquina, aunque también puede ser traducido a un código intermedio (bytecode) o a texto. A diferencia de los intérpretes, los compiladores reúnen diversos elementos o fragmentos en una misma unidad (un programa ejecutable o una librería), que puede ser almacenada y reutilizada. Este proceso de traducción se conoce como compilación.

**Interprete:** Intérprete o interpretador es un programa informático capaz de analizar y ejecutar otros programas. Los intérpretes se diferencian de los compiladores o de los ensambladores en que mientras estos traducen un programa desde su descripción en un lenguaje de programación al código de máquina del sistema, los intérpretes sólo realizan la traducción a medida que sea necesaria, típicamente, instrucción por instrucción, y normalmente no guardan el resultado de dicha traducción.

**Fuerza bruta:** La búsqueda por fuerza bruta, búsqueda combinatoria, búsqueda exhaustiva o simplemente fuerza bruta, es una técnica trivial pero a menudo usada, que consiste en enumerar sistemáticamente todos los posibles candidatos para la solución de un problema, con el fin de chequear si dicho candidato satisface la solución al mismo.

**Algoritmo voraz:** Un algoritmo voraz (también conocido como ávido, devorador o greedy) es una estrategia de búsqueda por la cual se sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima.

Este esquema algorítmico es el que menos dificultades plantea a la hora de diseñar y comprobar su funcionamiento. Normalmente se aplica a los problemas de optimización.

**Divide y conquista:** El término divide y vencerás hace referencia a uno de los más importantes paradigmas de diseño algorítmico. El método está basado en la resolución recursiva de un problema dividiéndolo en dos o más subproblemas de igual tipo o similar. El proceso continúa hasta que éstos llegan a ser lo suficientemente sencillos como para que se resuelvan directamente. Al final, las soluciones a cada uno de los subproblemas se combinan para dar una solución al problema original.

**Programación dinámica:** La programación dinámica es una técnica matemática que se utiliza para la solución de problemas matemáticos seleccionados, en los cuales se toma una serie de decisiones en forma secuencial. Proporciona un procedimiento sistemático para encontrar la combinación de decisiones que maximice la efectividad total, al descomponer el problema en etapas, las que pueden ser completadas por una o más formas (estados), y enlazando cada etapa a través de cálculos recursivos.

**Package:** Un paquete en Java se usa para agrupar clases relacionadas. Piense en ello como una carpeta en un directorio de archivos. Usamos paquetes para evitar conflictos de nombres y para escribir un código mejor mantenible.

**Clase:** Una clase es una plantilla para la creación de objetos de datos según un modelo predefinido. Las clases se utilizan para representar entidades o conceptos, como los sustantivos en el lenguaje.

**Función:** Una función, una subrutina o subprograma, como idea general, se presenta como un subalgoritmo que forma parte del algoritmo principal, el cual permite resolver una tarea específica.

**Librería:** Las librerías son trozos de código hechas por terceros. Esto nos facilita mucho la programación y hace que nuestro programa sea más sencillo de hacer y luego de entender.

**Tipado:** Cualidad de los lenguajes de programación con la cual establecen el tipos de las variables.

## 3.9 Librerías y funciones prohibidos

En las competencias de programación hay cierto tipo de librerías que no se pueden usar debido a su finalidad, ya que pueden usar requerimientos de hardware o software que pueden generar inestabilidad en las maquinas o simplemente hacer trampa dentro de la competencia, nombraremos las más importantes.

- Librerías de creación y manipulación de hilos (Threads).
- Librerías de manipulación de memoria (Malloc).
- Se evita el uso de punteros de memoria en las variables.
- Librerías de entornos gráficos.
- Librerías que no sean de grupo estándar del lenguaje, es decir librerías hechas por terceros ajenos.
- Librería de manejo de procesos.

### 3.10 10 pasos para resolver cualquier problema

1. Lea el problema completamente al menos dos o tres veces (o la cantidad de veces que sean que te hagan sentir cómodo)
2. Identifique el tema al que pertenece el problema. ¿Es un problema de ordenamiento o coincidencia de patrones? ¿Puedo usar la teoría de grafos? ¿Está relacionado con la teoría de números? etc.
3. Intente resolver el problema manualmente considerando 3 o 4 sets de datos de prueba.
4. Luego concéntrese en optimizar los pasos manuales. Intenta hacerlo lo más simple posible.
5. Escriba el pseudocódigo y comentarios, además del código de cada paso. Una cosa que puede hacer es verificar después de escribir cada función. Use un buen IDE con un depurador, si es posible. No es necesario pensar mucho en la sintaxis. Solo concéntrese en la lógica y los pasos.

6. Reemplace los comentarios o pseudocódigo con código real. Siempre verifique si los valores y el código se comportan como se esperaba antes de pasar a la nueva línea de pseudocódigo.
7. Luego optimice el código real.
8. Cuide también las condiciones de restricción de cada variable.
9. Obtenga comentarios de sus compañeros de equipo, profesores y otros desarrolladores y, si es posible, haga su pregunta en StackOverflow. Intente aprender de las pautas de los demás y de lo que están manejando esos problemas. Un problema puede resolverse de varias maneras. Por lo tanto, no se decepcione si no puede pensar como un experto. Debe atenerse al problema y gradualmente será mejor y más rápido para resolver problemas como los demás.
10. Práctica, práctica y práctica.

## 4 Acerca de los lenguajes

En el mercado existen muchísimos lenguajes de programación que sirven para infinidad de áreas aplicadas, tales como la matemática, la medicina, la administración aeroespacial, el control de gobierno, entre otras, pero dentro de este tendremos en cuenta 3 lenguajes únicamente los cuales son:

**JAVA 1.8.0:** Es un lenguaje de programación de propósito general, concurrente, orientado a objetos, que fue diseñado específicamente para tener tan pocas dependencias de implementación como fuera posible. Su intención es permitir que los desarrolladores de aplicaciones escriban el programa una vez y lo ejecuten en cualquier dispositivo, lo que quiere decir que el código que es ejecutado en una plataforma no tiene que ser recompilado para correr en otra. Java es, a partir de 2012, uno de los lenguajes de programación más populares en uso, particularmente para aplicaciones de cliente-servidor de web, con unos diez millones de usuarios reportados (Oracle, 2019).

El entorno de desarrollo recomendado para este libro es Eclipse IDE 2019-06.

Link de descarga:

<https://www.eclipse.org/downloads>

**C++11:** Es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup. La intención de su creación fue extender al lenguaje de programación C mecanismos que permiten la manipulación de objetos. En ese sentido, desde el punto de vista de los lenguajes orientados a objetos, el C++ es un lenguaje híbrido.

Posteriormente se añadieron facilidades de programación genérica, que se sumaron a los paradigmas de estructurada y programación orientada a objetos. Por esto se suele decir que el C++ es un lenguaje de programación multiparadigma (CPP Reference, 2019) .

El entorno de desarrollo recomendado es Zinjai 20190808.

Link de descarga:

<http://jinjai.sourceforge.net/>

**PYTHON3:** Es un lenguaje de programación interpretado cuya filosofía hace hincapié en una sintaxis que favorezca un código legible.

Se trata de un lenguaje de programación multiparadigma, ya que soporta orientación a objetos, programación imperativa y, en menor medida, programación funcional. Es un lenguaje interpretado, usa tipado dinámico y es multiplataforma (Python Software Foundation., 2019) .

El entorno de desarrollo recomendado es Python 3.7 IDLE.

Link de descarga:

<https://www.python.org/downloads/>



Ilustración 4-1 Logos JAVA, C++ y Python, Tomados de: <https://www.theverge.com>, <https://adictoalcodigo.blogspot.com> y <https://anthononcode.com> respectivamente

Cada uno de estos lenguajes tiene fortalezas y debilidades cuando los comparamos en rendimiento, escalabilidad, plataforma, almacenamiento y tiempo de ejecución. Por lo que tener conocimiento de los 3 lenguajes es una gran ventaja ya que se tiene en cuenta cual es más apto para la resolución de un ejercicio, y también permite mayor competitividad a nivel laboral.

## 4.1 Diferencias generales de los lenguajes

**Velocidad de ejecución**

*JAVA < PYTHON < C ++*

**Cantidad de librerías**

*JAVA > PYTHON > C ++*

**Dificultad de código**

*JAVA < PYTHON < C ++*

## **5 Maratones de programación**

Las competencias de programación a nivel internacional son desarrolladas para medir la capacidad de los estudiantes de programación de las distintas universidades y entes estudiantiles, entre ellas se permite utilizar todo tipo de medios escritos en los cuales basarse para el desarrollo de diferentes soluciones a problemas de la vida real que requieren un algoritmo eficiente.

Es por este motivo que nace la idea de desarrollar un compilado más nutrido de la información que facilita el desarrollo de este tipo de algoritmos, de tal forma que cualquier persona pueda aprender y participar en este tipo de eventos.

Grandes empresas de desarrollo como Google, Microsoft, IBM entre otras utilizan estas competencias como principal medio de reclutamiento de personal, por lo que se hace evidente la necesidad de participación en ellas.

## 6 ACIS/REDIS y CCPL



Ilustración 6-1 Logos ACIS/REDIS Tomado de: <http://www.evaluamos.com/images2017A/16219-1.jpg>

La Asociación Colombiana de Ingenieros de Sistemas (ACIS) es una organización sin ánimo de lucro que agrupa a más de 1500 profesionales en el área de sistemas. ACIS nació en 1975 agrupando entonces a un número pequeño de profesionales en sistemas. Con el transcurrir de los años, así como el panorama profesional para el área de los sistemas ha ido evolucionando, ACIS ha experimentado un desarrollo paralelo. Es el gremio de los Ingenieros de Sistemas participante en el desarrollo nacional.

Hoy en día, además de organizar eventos académicos de gran importancia a nivel nacional en el área de informática, la Asociación Colombiana de Ingenieros de Sistemas ha multiplicado sus campos de acción involucrándose en la mayoría de los debates sobre el desarrollo tecnológico de Colombia. ACIS se ha constituido en los últimos años como el gestor de eventos de gran reconocimiento que buscan cubrir las diferentes áreas tecnológicas de la Ingeniería de Sistemas como son el Salón de Informática, las Jornadas de Gerencia de Proyectos de TI, las Jornadas de Seguridad Informática, Cursos de Capacitación, MoodleMoot Colombia, etc (Asociación Colombiana de Ingenieros de Sistemas., 2019).

La Red de Programas de Ingeniería de Sistemas y Afines (REDIS) es una agrupación de quienes actúan como autoridades máximas de los programas de ingeniería de sistemas, o nombres afines, ofrecidos por instituciones de educación superior. Fue creada en Bogotá en el año 2001 y desde entonces se reúne por lo menos una vez al mes a tratar asuntos

comunes de carácter académico y profesional, que implican colaboración y relación con los representantes de distintos sectores y gremios: Acis, Acofi, Fedesoft, Ministerio de las TIC, Ministerio de Industria y Comercio, y Ministerio de Educación Nacional, entre otros.



*Ilustración 6-2 Logo CCPL Tomado de: [www.programingleague.org](http://www.programingleague.org)*

Colombian Collegiate Programming League (CCPL) es la liga de competencia en programación más grande de Colombia, realizando diferentes rondas por año en donde las diferentes entidades de educación superior envían equipos de estudiantes a competir, las competencias se realizan en diferentes sedes cada ronda y funcionan como entrenamiento para las competencias clasificatorias nacionales en Colombia.

## 7 ACM e ICPC



Ilustración 7-1 Logo ACM-ICPC Tomado de: <http://www.cs.cornell.edu/acm/>

Association for Computing Machinery (ACM) reúne a educadores, investigadores y profesionales de computación para inspirar el diálogo, compartir recursos y abordar los desafíos del campo. Como la sociedad de computación más grande del mundo, ACM fortalece la voz colectiva de la profesión a través de un liderazgo sólido, la promoción de los más altos estándares y el reconocimiento de la excelencia técnica. ACM apoya el crecimiento profesional de sus miembros al brindar oportunidades para el aprendizaje de por vida, el desarrollo profesional y la creación de redes profesionales.

Fundada en los albores de la era de la computación, el alcance de ACM se extiende a cada parte del mundo, con más de la mitad de sus casi 100,000 miembros que residen fuera de los EE. UU. Su creciente membresía ha llevado a los Consejos en Europa, India y China, fomentando oportunidades de redes que fortalecen los lazos dentro y entre los países y las comunidades técnicas. Sus acciones mejoran la capacidad de ACM para aumentar el conocimiento de los importantes problemas técnicos, educativos y sociales de la informática en todo el mundo.

El Concurso Internacional de Programación Colegial (International Collegiate Programming Contest ICPC) es un concurso de programación algorítmica para estudiantes universitarios. Equipo de tres, que representan a su universidad, trabajan para resolver los problemas más reales del mundo, fomentando la colaboración, la creatividad, la innovación y la capacidad de desempeñarse bajo presión. A través del entrenamiento y la competencia, los equipos se desafían entre sí para elevar el nivel de lo posible. En pocas palabras, es el concurso de programación más antiguo, más grande y más prestigioso del mundo (Zhu Jieao, 2005).

El ICPC se remonta a 1970, cuando los pioneros del Capítulo Alfa de la Sociedad de Honor de Ciencias de la Computación de la UPE organizaron la primera competencia. La iniciativa se difundió rápidamente dentro de los Estados Unidos y Canadá como un programa innovador para aumentar la ambición, la capacidad de resolución de problemas y las oportunidades de los estudiantes más sólidos en el campo de la informática (Combéfis, 2014).

Con el tiempo, el concurso se convirtió en una competencia de varios niveles con la primera ronda de campeonato llevada a cabo en 1977. Desde entonces, el concurso se ha convertido en una colaboración mundial de universidades que organizan competiciones regionales que hacen avanzar a los equipos a la ronda de campeonatos mundial anual, Las finales mundiales de ICPC (ICPC World Finals) (Amraii, 2006).



Ilustración 7-2 ICPC WORLD FINALS 2018 Tomado de: [www.topcoder.com](http://www.topcoder.com)

El International Collegiate Programming Contest (ICPC) es el principal concurso mundial de programación realizado por y para las universidades del mundo. El ICPC está afiliado a la Fundación ICPC y tiene su sede en la Universidad de Baylor.

## 8 Google CodeJAM, HashMap y Kickstart



Ilustración 8-1 Inscripciones CODEJAM 2018 Tomado de: [https://www.youtube.com/watch?v=ipdUjbK1\\_h8](https://www.youtube.com/watch?v=ipdUjbK1_h8)

Google Code Jam, HashMap y Kickstart son competencias de programación internacional organizada y administrada por Google, son de modalidad solitario, en equipo y entrenamiento respectivamente. La competencia CodeJAM comenzó en 2003 como un medio para identificar a los mejores talentos de ingeniería para un empleo potencial en Google. La competencia consiste en un conjunto de problemas algorítmicos que deben resolverse en un período de tiempo fijo.

Los competidores pueden usar cualquier lenguaje de programación y entorno de desarrollo para crear sus soluciones. De 2003 a 2007, Google Code Jam se implementó en la plataforma de Topcoder y tenía unas reglas bastante diferentes. Desde 2008, Google ha desarrollado su propia infraestructura dedicada para el concurso.

## 9 CodeChef



Ilustración 9-1 Logo CodeChef y Directi, Tomado de: [www.codechef.com](http://www.codechef.com)

CodeChef es una iniciativa educativa sin fines de lucro de Directi, una compañía india de software. Es una comunidad de programación global que fomenta el aprendizaje y la competencia amistosa, construida sobre la plataforma de programación competitiva más grande del mundo. También hemos construido una gran comunidad de curadores de problemas.

CodeChef fue creado como una plataforma para ayudar a los programadores a triunfar en el mundo de los algoritmos, la programación de computadoras y los concursos de programación. Realizan tres concursos destacados cada mes y otorgan premios y regalos a los ganadores como estímulo. Aparte de esto, la plataforma está abierta a toda la comunidad de programación para organizar sus propios concursos. Las principales instituciones y organizaciones de todo el mundo utilizan la plataforma para organizar sus concursos (Directi, 2019).

En el siguiente link se podrá encontrar un video tutorial explicativo sobre cómo utilizar CodeChef para entrenamiento de programación:

Solving your first problem in Java on CodeChef:

<https://www.youtube.com/watch?v=gaPdjwuFZTs>

## 10 Online Judge



*Ilustración 10-1 Logo Online Judge 2019, Tomado de: <https://onlinejudge.org/>*

Online Judge (Anteriormente Uva Online Judge) es un juez automatizado en línea para problemas de programación anteriormente alojado por la Universidad de Valladolid. Su archivo de problemas tiene más de 14000 problemas y el registro de usuarios está abierto a todos. Actualmente hay más de 1000000 usuarios registrados. Un usuario puede enviar una solución en ANSI C (C89), C ++ (C ++ 98), Pascal, Java, C ++ 11 o Python.



*Ilustración 10-2 Anterior logo de UVa Online Judge, Tomado de: <http://nafischonchol.blogspot.com>*

UVa OJ también organiza concursos. En el entorno del concurso, el usuario tiene un tiempo limitado para resolver un pequeño conjunto de problemas. El UVa OJ fue creado en 1995 por Miguel Ángel Revilla, matemático que enseñaba algoritmos en la Universidad de

Valladolid en España, actualmente la plataforma se convirtió en independiente de esta universidad y es administrada por Miguel Ángel Revilla Jr.

Los siguientes links presentan video tutoriales explicando cómo buscar y realizar ejercicios en esta plataforma:

How to submit in UVa Online Judge:

<https://www.youtube.com/watch?v=i56-7MlzyHY>

Find Beginners Problem in UVa Online Judge:

<https://www.youtube.com/watch?v=MZhV8B9lnw>

# 11 Comenzando con lenguajes de programación

## 11.1 Operadores lógicos, comparativos y matemáticos

OPERADOR	DESCRIPCIÓN	EJEMPLO	RESULTADO
+	Suma	5+6	11
-	Resta	5-6	-1
*	Multiplicación	5*6	30
/	División	7/5	1
%	Modulo (Residuo)	7%5	2
&	AND	true & true	True
	OR	true   false	True
!	NOT	!true	False
^	XOR	true^true	False
&&	AND de condición	5>4&&6>7	False
	OR de condición	5>4  6>7	True
==	Igual que	7==38	False
!=	Distinto que	'a' != 'k'	True
<	Menor que	'G' < 'B'	False
>	Mayor que	'b' > 'a'	True
<=	Menor o igual que	7.5 <= 7.38	False
>=	Mayor o igual que	38 >= 7	true

Tabla 11-1 Operadores lógicos, comparativos y matemáticos con ejemplo

## 11.2 Tipos de variables básicas

Variable	Tipo	JAVA	C++	PYTHON
<b>Int</b>	Numérico entero	2^32 bits	2^32 bits	Infinito
<b>long</b>	Numérico entero	2^64 bits	2^64 bits	No aplica
<b>float</b>	Numérico decimal	2^32 bits	2^32 bits	Infinito
<b>double</b>	Numérico decimal	2^64 bits	2^64 bits	No aplica
<b>char</b>	Carácter	2^16 bits	2^8 bits	Infinito
<b>string</b>	Cadena de caracteres	Infinito	Infinito	Infinito
<b>byte</b>	Binario	8 bits	No aplica	No aplica

Tabla 11-2: Tipos de variable y sus tamaños en tres lenguajes de programación

## 11.3 Lectura e impresión

Las lecturas e impresiones de nuestros ejercicios de programación competitiva, sus algoritmos base y sus modificaciones se realizan por la consola (STDIN, STDOUT), por lo que la programación de entornos gráficos para nuestros códigos no es válida, los siguientes códigos muestran ejemplos de cómo se realizan las entradas y salidas de diferentes tipos de variables.

### Código

```
/*
Ejemplo de entrada
6555
73653736353
463553
345543234
a
holamundo
101
*/
```

## JAVA

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Scanner;

public class LecturaEscritura {
    //Variables Globales
    static int númeroentero;
    static long númeroenterolargo;
    static float númerodecimal;
    static double númerodecimallargo;
    static char carácter;
    static String cadena;
    static byte bits;

    public static void main(String[] args) throws IOException {
        System.out.println("Lectura\n");
        lecturaLenta();
        lecturaRapida();
        System.out.println("");
        System.out.println("Impresión \n");
        impresionNormal();
        impresionConFormato();
    }

    static void lecturaLenta() {
        //Objeto lector
        Scanner sc = new Scanner(System.in);
        //Lectura con tipo especificado
        númeroentero = sc.nextInt();
        númeroenterolargo = sc.nextLong();
        númerodecimal = sc.nextFloat();
        númerodecimallargo = sc.nextDouble();
        carácter = sc.next().charAt(0);
        cadena = sc.next();
        bits = sc.nextByte();

    }

    static void lecturaRapida() throws IOException {
        //Objeto lector
        BufferedReader br = new BufferedReader(new
        InputStreamReader(System.in));
        //Lectura como cadena y conversión a tipo
        númeroentero = Integer.parseInt(br.readLine());
        númeroenterolargo = Long.parseLong(br.readLine());
        númerodecimal = Float.parseFloat(br.readLine());
        númerodecimallargo = Double.parseDouble(br.readLine());
        carácter = br.readLine().charAt(0);
        cadena = br.readLine();
        bits = Byte.parseByte(br.readLine());
    }
}
```

```

    static void impresionNormal() {
        System.out.println(númeroentero);
        System.out.println(númeroenterolargo);
        System.out.println(númerodecimal);
        System.out.println(númerodecimallargo);
        System.out.println(caracter);
        System.out.println(cadena);
        System.out.println(bits);
    }

    static void impresionConFormato() {
        //Se especifica el tipo de dato o la forma en que se imprimira
        System.out.printf("%d \n",númeroentero);
        System.out.printf("%d \n",númeroenterolargo);
        System.out.printf("%f \n",númerodecimal);
        System.out.printf("%e \n",númerodecimal);
        System.out.printf("%f \n",númerodecimallargo);
        System.out.printf("%e \n",númerodecimallargo);
        System.out.printf("%s \n",caracter);
        System.out.printf("%s \n",cadena);
        System.out.printf("%s \n",bits);
    }
}

```

## C++

```

//Unicas 2 librerias que se usan en c++ (Contienen todas)
#include<bits/stdc++.h>
#include<cstdlib>
using namespace std;
//variables globales
int númeroentero;
long númeroenterolargo;
float númerodecimal;
double númerodecimallargo;
char caracter;
string cadena;

void lecturaLenta() {
    //Lectura con tipo especificado
    cin>>númeroentero;
    cin>>númeroenterolargo;
    cin>>númerodecimal;
    cin>>númerodecimallargo;
    cin>>caracter;
    cin>>cadena;
}

void lecturaRapida() {
    //Lectura como cadena y conversión a tipo (En c++ & es necesario)
    scanf("%d",&númeroentero);
    scanf("%ld",&númeroenterolargo);
    scanf("%f",&númerodecimal);
    scanf("%lf",&númerodecimallargo);
    scanf("%s",&caracter);
    scanf("%s",cadena.c_str());
}

```

```

void impresionNormal() {
    cout<<númeroentero<<endl;
    cout<<númeroenterolargo<<endl;
    cout<<númerodecimal<<endl;
    cout<<númerodecimallargo<<endl;
    cout<<caracter<<endl;
    cout<<cadena<<endl;
}
void impresionConFormato() {
    //Se especifica el tipo de dato o la forma en que se imprimira
    printf("%d \n",númeroentero);
    printf("%lld \n",númeroenterolargo);
    printf("%f \n",númerodecimal);
    printf("%e \n",númerodecimal);
    printf("%lf \n",númerodecimallargo);
    printf("%e \n",númerodecimallargo);
    printf("%c \n",caracter);
    printf("%s \n",cadena.c_str());
}

int main (int argc, char *argv[]) {
    cout<<"Lectura\n"<<endl;
    lecturaLenta();
    cout<<"Impresión \n"<<endl;
    impresionNormal();
    cout<<endl;
    cout<<"Lectura\n"<<endl;
    lecturaRapida();
    cout<<"Impresión \n"<<endl;
    impresionConFormato();
    return 0;
}

```

## PYTHON

```

#Los comentarios en Python se usa el # (Numeral)
#variables globales
#la identación define el orden
númeroentero=int(0)
númeroenterolargo=int(0)
númerodecimal=float(0)
númerodecimallargo=float(0)
caracter=''
cadena=""
print("Lectura")
#Lectura con tipo especificado
númeroentero=int(input())
númeroenterolargo=int(input())
númerodecimal=float(input())
númerodecimallargo=float(input())
caracter=input()
cadena=str(input())
print("Impresión")
print(númeroentero)
print(númeroenterolargo)

```

```

print(númerodecimal)
print(númerodecimallargo)
print(caracter)
print(cadena)
print("Impresión formateada")
#Se especifica el tipo de dato o la forma en que se imprimira
#obligatorio usar el %
print("%d" % númeroentero)
print("%d" % númeroenterolargo)
print("%f" %númerodecimal)
print("%e"%númerodecimal)
print("%f"%númerodecimallargo)
print("%e"%númerodecimallargo)
print("%c"%caracter)
print("%s"%cadena)

```

## 11.4 Condicionales y ciclos

Un condicional, como su nombre lo indica, es una condición para discernir entre una opción u otra, y en el proceso mental normalmente se manifiesta con un “Si”; por ejemplo: Si (va a llover), coge el paraguas. Sintácticamente, IF es la palabra reservada para desencadenar el poder de los condicionales en el código. ELSE expresa “en el caso contrario”. Siguiendo con el ejemplo anterior de la lluvia: if (va a llover) coge el paraguas else coge el bañador.

Un bucle o ciclo, en programación, es una secuencia que ejecuta repetidas veces un trozo de código, hasta que la condición asignada a dicho bucle deja de cumplirse. Los tres bucles más utilizados en programación son el bucle while, el bucle for y el bucle do-while.

/\*

Ejemplo de entrada

5

\*/

## JAVA

```

import java.util.Scanner;
public class CondicionalesCiclos {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        //Lectura

```

```

int número = sc.nextInt();
//Estructura de un condicional
//Dentro el parentesis va la condicion a revisar
    //Si se cumple se realiza las siguientes lineas
    //En caso contrario pasa al else
    //y se realiza las lineas siguientes del else
    if (número % 2 == 0) {
        System.out.println("Es par");
    } else {
        System.out.println("Es impar");
    }
    System.out.println("");
    System.out.println("Ciclo for ");
// variable iteradora, condicion , acumulador
    for (int i = 0; i < 10; i++) {
        System.out.println(i);
    }
    System.out.println("Ciclo while");
//Variable, condicion, al final acumulador
    int acumulador = 0;
    while (acumulador < 10) {
        System.out.println(acumulador);
        acumulador++;
    }
    System.out.println("Ciclo do while");
//Haga hasta que una condición se cumpla
    acumulador = 0;
    do {
        System.out.println(acumulador);
        acumulador++;
    } while (acumulador < 10);
    System.out.println("Ciclo for each");
//Iterar entre todos los elementos que tenga una estructura
//Sin importar el tamaño
    int números[] = {1,2,3,4,5,6,7,8,9,0};
for (int o : números) {
    System.out.println(o);
}

```

}

## C++

```

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

int main (int argc, char *argv[]) {

    int número;
    cin>>número;
    //Estructura de un condicional
    //Dentro el parentesis va la condicion a revisar

```

```

//Si se cumple se realiza las siguientes lineas
//En caso contrario pasa al else
//y se realiza las lineas siguientes del else
if (número % 2 == 0) {
    cout<<"Es par"<<endl;
} else {
    cout<<"Es impar"<<endl;
}
cout<<endl;
cout<<"Ciclo for "<<endl;
// variable iteradora, condicion , acumulador
for (int i = 0; i < 10; i++) {
    cout<<i<<endl;
}
cout<<"Ciclo while"<<endl;
//Variable, condicion, al final acumulador
int acumulador = 0;
while (acumulador < 10) {
    cout<<acumulador<<endl;;
    acumulador++;
}

cout<<"Ciclo do while"<<endl;
//Haga hasta que una condición se cumpla
acumulador = 0;
do {
    cout<<acumulador<<endl;
    acumulador++;
} while (acumulador < 10);
cout<<"Ciclo for each"<<endl;
//Iterar entre todos los elementos que tenga una estructura
//Sin importar el tamaño
int números[] = {1,2,3,4,5,6,7,8,9,0};
for (int o : números) {
    cout<<o<<endl;
}
return 0;
}

```

## PYTHON

```

numero=0;
numero =int(input())
#Estructura de un condicional
#Dentro el parentesis va la condicion a revisar
#Si se cumple se realiza las siguientes lineas
#En caso contrario pasa al else
#y se realiza las lineas siguientes del else
if (numero % 2 == 0):
    print("Es par")
else:
    print("Es impar")
print("Ciclo for ")
#variable iteradora, rango inicio,final
for i in range (0,10):
    print(i)

```

```

print("Ciclo while")
#Variable, condicion, al final acumulador
acumulador = 0;
while (acumulador < 10):
    print(acumulador)
    acumulador=acumulador+1;
print("Ciclo for each")
#Iterar entre todos los elementos que tenga una estructura
#Sin importar el tamaño
numeros =[1,2,3,4,5,6,7,8,9,0]
for o in numeros:
    print(o)

```

## 11.5 Control de excepciones

El manejo de excepciones es una técnica de programación que permite al programador controlar los errores ocasionados durante la ejecución de un programa informático. Cuando ocurre cierto tipo de error, el sistema reacciona ejecutando un fragmento de código que resuelve la situación, por ejemplo retornando un mensaje de error o devolviendo un valor por defecto.

```

/*
Ejemplo de entrada
5
*/

```

### JAVA

```

import java.util.Scanner;
public class ControlExcep {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int número;
        //Intenta guardar el dato en la variable
        //Si falla controla la excepción y
        //realiza otra cosa sin terminar la ejecución
        try {
            número=sc.nextInt();
            System.out.println(número);
        } catch (Exception e) {
            System.out.println("El dato insertado no es un número");
            System.out.println("0 es muy grande para un tipo int");
        }
    }
}

```

## C++

//No es necesario controlar las excepciones, no almacena nada nuevo por lo que el programa mostrara el valor que anteriormente tenía esa posición de memoria  
**PYTHON**

```
número=0
#Intenta guardar el dato en la variable
#Si falla controla la excepción y
#realiza otra cosa sin terminar la ejecución
try:
    número=int(input())
    print (número)
except:
    print ("El dato insertado no es un número")
    print ("0 es muy grande para un tipo int")
```

## 11.6 Complejidad de tiempo (Eficiencia Algorítmica)

En Ciencias de la Computación, el término eficiencia algorítmica es usado para describir aquellas propiedades de los algoritmos que están relacionadas con la cantidad de recursos utilizados por el algoritmo. Un algoritmo debe ser analizado para determinar el uso de los recursos que realiza. La eficiencia algorítmica puede ser vista como análogo a la ingeniería de productividad de un proceso repetitivo o continuo.

Notación	Nombre	Ejemplos
<b>O(1)</b>	Constante	Determinar si un número es par o impar. Usar una tabla de consulta que asocia constante/tamaño. Usar una hash para obtener un elemento
<b>O(Log n)</b>	Logarítmico	Buscar un elemento específico en un Array utilizando un árbol

		binario de búsqueda o un árbol de búsqueda balanceado, así como todas las operaciones en un Heap binomial.
$O(\sqrt{n})$	Radical	Buscar un elemento específico en un Array utilizando búsqueda por saltos
$O(n)$	Lineal	Buscar un elemento específico en una lista desordenada o en un árbol degenerado (peor caso).
$O(n \log n)$	Log Linear o Quasilinear	Ejecutar una transformada rápida de Fourier; Heap Sort, Quick Sort (caso mejor y promedio), o Merge Sort
$O(N^2)$	Cuadrático	Multiplicar dos números de $n$ dígitos por un algoritmo simple. Bubble Sort (caso peor o implementación sencilla), Shell Sort, Quick Sort (caso peor).
$O(C^n), C > 1$	Exponencial	Encontrar la solución exacta al problema del viajante utilizando programación dinámica. Determinar si dos sentencias lógicas son equivalentes utilizando una búsqueda por fuerza bruta

$O(N^3)$	Cubico	Algoritmos de búsqueda de ciclos triples, recorridos de grafos.
----------	--------	---

Tabla 11-3 Ejemplos de complejidad de tiempo

## 11.7 ¿Qué afecta la complejidad de tiempo?

- La cantidad de memoria requerida por el código del algoritmo.
- La cantidad de memoria requerida para almacenar los datos de entrada.
- La cantidad de memoria requerida para los datos de salida (algoritmos como los de ordenación suelen reorganizar los datos de entrada y por ello no necesitan memoria extra para la salida).
- La cantidad de memoria requerida en cuanto a espacio de trabajo del algoritmo para realizar los cálculos y asignaciones (tanto para variables como cualquier espacio necesario en la pila para almacenar llamadas a subrutinas, este espacio es particularmente significativo para algoritmos que utilizan técnicas recursivas).
- Memoria Caché (usualmente RAM-estática): esta opera a una velocidad comparable a la del CPU.
- Memoria física principal (usualmente RAM-dinámica): esta opera un tanto más lenta que el CPU.
- Memoria virtual (usualmente en disco): esta da la impresión de una gran cantidad de memoria utilizable y opera en el orden de los miles más lenta que el CPU.
- La velocidad de la CPU
- El factor de tiempo propio de cada lenguaje, recuerde que cada lenguaje tiene tiempos de ejecución diferentes dependiendo de cómo funcione, de si es interpretado y de la prioridad que tenga en tiempo del procesador

## 11.8 Conteo de tiempo de ejecución

El tiempo de ejecución es el período en el que un programa es ejecutado por el sistema operativo. El período comienza cuando el programa es llevado a la memoria primaria y comienzan a ejecutarse sus instrucciones. El período finaliza cuando el programa envía la señal de término (normal o anormal) al sistema operativo.

Suele decirse también que un programa se encuentra "corriendo" mientras está siendo ejecutado. Otros tiempos de un programa son el tiempo de compilación, el tiempo de enlazado y el tiempo de carga.

```
/*Calculo del tiempo de ejecución de un ciclo de 100000 números
Varia de maquina en maquina, y dependiendo de lo que se
encuentre en ejecución en la maquina*/
public class CalculoTiempo {

    public static void main(String[] args) {
        long timeBefore = System.currentTimeMillis();
        final int N = 100000;
        for (int i = 0; i <= N; i++) {
            System.out.println(i);
        }
        long timeAfter = System.currentTimeMillis();
        System.out.println((timeAfter - timeBefore) + "ms");
    }
}
```

## 11.9 String Matching y expresiones regulares

Una expresión regular, o expresión racional, es una secuencia de caracteres que conforma un patrón de búsqueda. También son conocidas como regex por su contracción de las palabras inglesas regular expression. Son principalmente utilizadas para la búsqueda de patrones de cadenas de caracteres u operaciones de sustituciones.

### Lógicos:

- $x|y$ :  $x$  o  $y$
- $xy$ :  $x$  seguido de  $y$
- $()$ : Agrupación

### **Intervalos de caracteres:**

- [abc]: Cualquiera de los caracteres entre corchetes. Pueden especificarse rangos, por ejemplo ([a-d] que equivale a [abcd]).
- [^abc]: Cualquier carácter que no esté entre los corchetes.
- [a-zA-Z]: a a la z o A a la Z (Rango).
- [a-zA-¶[def]]: d,e, o f (Intersección)
- [a-zA-¶[bc]]: (Substracción)

### **Intervalos de caracteres predefinidos:**

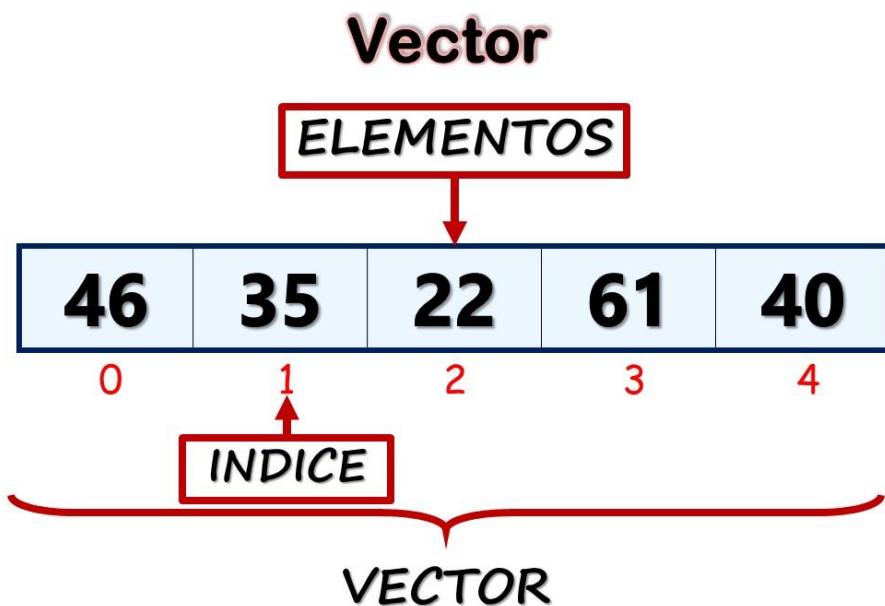
- .: Cualquier carácter individual, salvo el de salto de línea
- \d: Cualquier carácter de digito, equivalente a [0-9]
- \D: Cualquier carácter que no sea del digito, equivalente a [^0-9]
- \s: Cualquier carácter individual de espacio en blanco (Espacios, tabulaciones, saltos de página o saltos de línea)
- \S: Cualquier carácter individual que no sea un espacio en blanco
- \w: Cualquier carácter alfanumérico
- \W: Cualquier carácter que no sea alfanumérico

## 12 Estructuras de datos básicas

Lo habitual es que un vector tenga una cantidad fija de memoria asignada, aunque dependiendo del tipo de vector y del lenguaje de programación un vector podría tener una cantidad variable de datos. En este caso, se les denomina vectores dinámicos, en oposición, a los vectores con una cantidad fija de memoria asignada se los denomina vectores estáticos. El uso de vectores dinámicos requiere realizar una apropiada gestión de memoria dinámica. Un uso incorrecto de los vectores dinámicos, o mejor dicho, una mala gestión de la memoria dinámica, puede conducir a una fuga de memoria. Al utilizar vectores dinámicos siempre habrá que liberar la memoria utilizada cuando ésta ya no se vaya a seguir utilizando.

Lenguajes más modernos y de más alto nivel, cuentan con un mecanismo denominado recolector de basura que permiten que el programa decida si debe liberar el espacio basándose en si se va a utilizar en el futuro o no un determinado objeto.

**Vector tamaño fijo (Array):**



*Guía del programador competitivo.*

Ilustración 12-1 Ejemplo de vector

Se le denomina vector o formación (en inglés Array) a una zona de almacenamiento contiguo que contiene una serie de elementos del mismo tipo, los elementos de la matriz. Desde el punto de vista

lógico una matriz se puede ver como un conjunto de elementos ordenados en fila (o filas y columnas si tuviera dos dimensiones).

Estas estructuras de datos son adecuadas para situaciones en las que el acceso a los datos se realice de forma aleatoria e impredecible. Por el contrario, si los elementos pueden estar ordenados y se va a utilizar acceso secuencial sería más adecuado utilizar una lista, ya que esta estructura puede cambiar de tamaño fácilmente durante la ejecución de un programa.

### Vector tamaño dinámico (ArrayList):

En programación, un arreglo dinámico o Array dinámico, es un Array de elementos que crece o mengua dinámicamente conforme los elementos se agregan o se eliminan. Se suministra como librerías estándar en muchos lenguajes modernos de programación.

### Vector sin repetición (Set):

Set es una interfaz que amplía la colección. Es una colección desordenada de objetos en la que no se pueden almacenar valores duplicados. Set tiene varios métodos para agregar, eliminar, borrar, tamaño, etc.

### Matriz (Matrix):

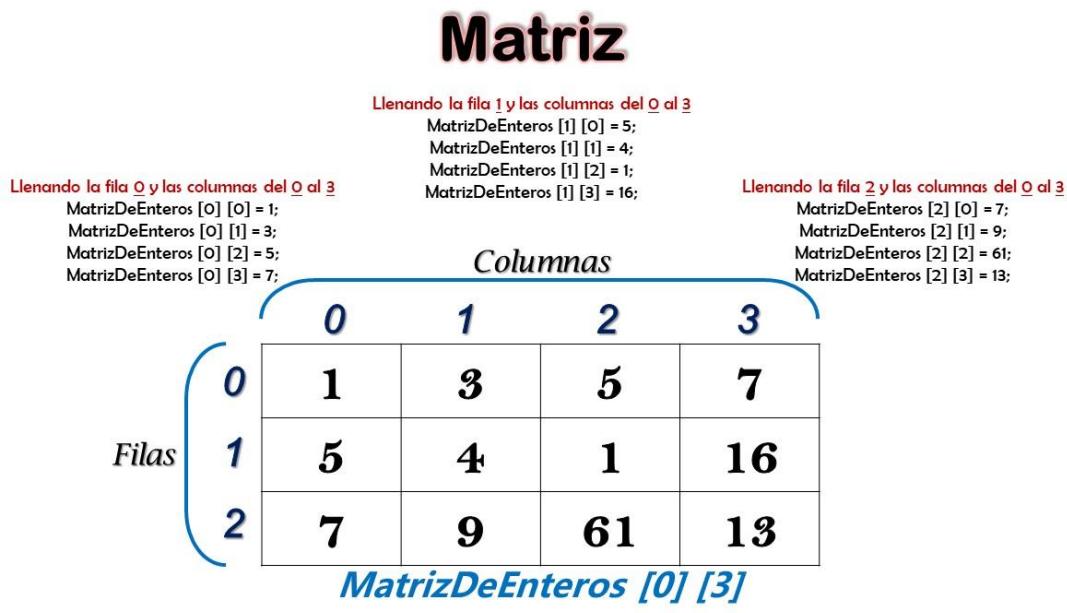


Ilustración 12-2 Ejemplo de matriz

Es una tabla bidimensional de números consistentes en cantidades abstractas con las que se pueden realizar diferentes operaciones, como por ejemplo la suma, multiplicación y descomposición de las mismas de varias formas, lo que también las hace un concepto clave en el campo del álgebra lineal. Las matrices se utilizan para describir sistema de ecuaciones lineales, realizar un seguimiento de los coeficientes de una aplicación lineal y registrar los datos que dependen de varios parámetros.

#### Diccionario o mapa (HashMap):

Los mapas son contenedores asociativos que almacenan elementos de forma mapeada. Cada elemento tiene un valor clave y un valor asignado. No hay dos valores asignados que puedan tener los mismos valores clave

#### Pila (Stack):

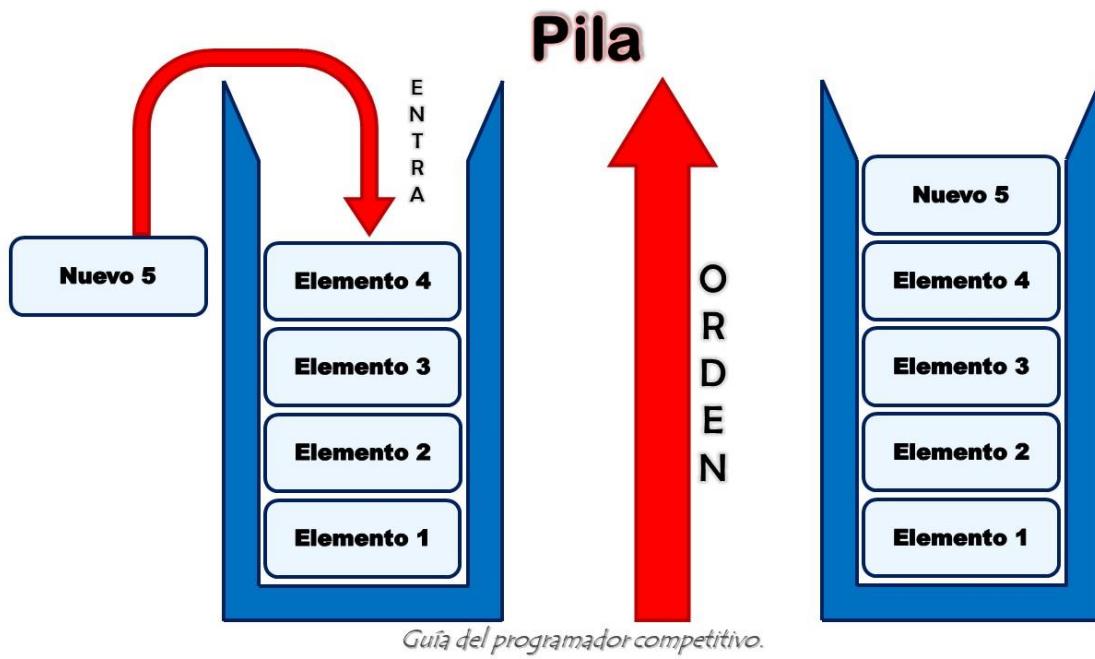


Ilustración 12-3 Ejemplo de pila

Es una lista ordenada o estructura de datos que permite almacenar y recuperar datos, el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In, First Out, «último en

entrar, primero en salir»). Esta estructura se aplica en multitud de supuestos en el área de informática debido a su simplicidad y capacidad de dar respuesta a numerosos procesos.

### Cola (Queue):

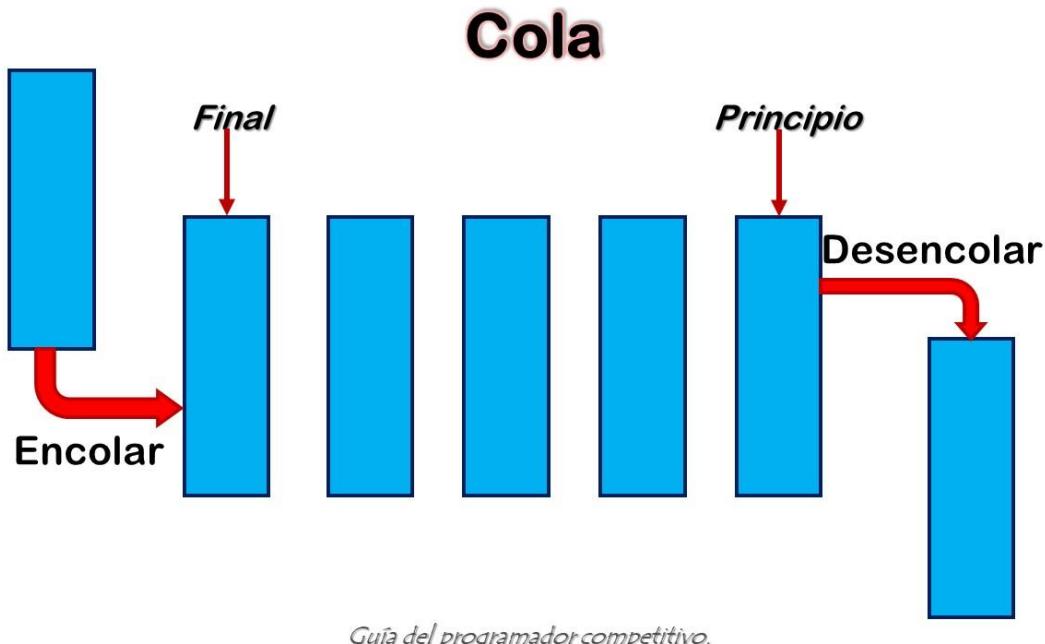


Ilustración 12-4 Ejemplo de cola

Es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción push se realiza por un extremo y la operación de extracción pop por el otro. También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.

## JAVA

```
import java.util.*;  
public class Main {  
    public static Random r = new Random();  
    public static int i, j;  
    public static void main(String args[]) {  
        Scanner sc = new Scanner(System.in);  
        System.out.println("Inserte cantidad de datos");
```

```

        int n = sc.nextInt();
        System.out.println("Que estructura quiere utilizar? (Los números son
aleatorios");
        System.out.println("1) Vector fijo ");
        System.out.println("2) Vector dinamico");
        System.out.println("3) Vector sin repeticion");
        System.out.println("4) Matriz");
        System.out.println("5) Diccionario");
        System.out.println("6) Pila");
System.out.println("7) Cola");
        int aux = sc.nextInt();
        switch (aux) {
            case 1:
                vectorfijo(n);
                break;
            case 2:
                vectordinamico(n);
                break;
            case 3:
                vectorsinrepeticion(n);
                break;
            case 4:
                matriz(n);
                break;
            case 5:
                Diccionario(n);
break;
            case 6:
                pila(n);
                break;
            case 7:
                cola(n);
                break;
        }

    }

public static void vectordinamico(int n) {
    ArrayList<Integer> arreglo = new ArrayList<>();
    for (i = 0; i < n; i++) {
        arreglo.add(r.nextInt());
    }
    for (i = 0; i < n; i++) {
        System.out.println(arreglo.get(i));
    }
}

public static void vectorfijo(int n) {
    int[] números = new int[n];
    for (i = 0; i < n; i++) {
        números[i] = r.nextInt();
    }
    for (i = 0; i < n; i++) {

```

```

        System.out.println(números[i]);
    }

}

public static void matriz(int n) {
    int[][] matrix = new int[n][n];
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            matrix[i][j] = r.nextInt(50);

        }
    }
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            System.out.print(matrix[i][j] + " ");
        }
        System.out.println("");
    }
}

public static void vectorsinrepeticion(int n) {
    HashSet<Integer> sinrep = new HashSet<>();
    for (i = 0; i < n; i++) {
        sinrep.add(r.nextInt(50));
    }
    Iterator it = sinrep.iterator();
    while (it.hasNext()) {
        System.out.println(it.next());
    }
}

public static void Diccionario(int n) {
    HashMap<Integer, String> dicc = new HashMap<>();
    char letra = 'a';
    for (i = 0; i < n; i++) {
        dicc.put(r.nextInt(), letra + "");
        int aux = (int) letra;
        aux++;
        letra = (char) aux;
    }
    for (i = 0; i < n; i++) {
        System.out.println(dicc.get(i) + "");
    }
}

public static void pila(int n) {
    Stack<Integer> mipila = new Stack<>();

    for (i = 0; i < n; i++) {
        mipila.push(r.nextInt(50));
    }
}

```

```

        while (!mipila.isEmpty()) {
            System.out.println(mipila.pop());
        }
    }

    public static void cola(int n) {
        Queue<Integer> mocola = new LinkedList<>();

        for (i = 0; i < n; i++) {
            mocola.offer(r.nextInt(50));
        }
        while (!mocola.isEmpty()) {
            System.out.println(mocola.poll());
        }
    }
}

```

## C++

```

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

int i, j;

void vectordinamico(int n) {
    vector<int> arreglo;
    for (i = 0; i < n; i++) {
        arreglo.push_back(rand()%100);
    }
    for (i = 0; i < n; i++) {
        cout<<arreglo.at(i)<<endl;
    }
}

void vectorfijo(int n) {
    int números[n];
    for (i = 0; i < n; i++) {
        números[i] = rand()%100;
    }
    for (i = 0; i < n; i++) {
        cout<<números[i]<<endl;;
    }
}

void matriz(int n) {
    int matrix [n][n];
    for (i = 0; i < n; i++) {

```

```

        for (j = 0; j < n; j++) {
            matrix[i][j] = rand()%100;
        }
    }
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            cout<<matrix[i][j] << " ";
        }
        cout<<endl;
    }
}

void vectorsinrepeticion(int n) {
    set<int> sinrep;
    for (i = 0; i < n; i++) {
        sinrep.insert(rand()%100);
    }
    for (auto itr = sinrep.begin(); itr != sinrep.end(); ++itr) {
        cout<<*itr<<endl;
    }
}

void Diccionario(int n) {
    map<int, char> dicc;
    char letra = 'a';
    for (i = 0; i < n; i++) {
        dicc.insert({rand()%100, letra});
        int aux = (int) letra;
        aux++;
        letra = (char) aux;
    }
    for (auto itr = dicc.begin(); itr != dicc.end(); ++itr) {
        cout << itr->first
            << '\t' << itr->second << '\n';
    }
}

void pila(int n) {
    stack<int> mipila;

    for (i = 0; i < n; i++) {
        mipila.push(rand() % 100);
    }
    while (!mipila.empty()) {
        cout<<mipila.top()<<endl;
        mipila.pop();
    }
}

void cola(int n) {
    queue<int> micola;
    for (i = 0; i < n; i++) {

```

```

        micola.push(rand() % 100);
    }
    while (!micola.empty()) {
        cout<<micola.front()<<endl;
        micola.pop();
    }
}

int main (int argc, char *argv[]) {
    cout<<"Inserte cantidad de datos"<<endl;
    int n ;
    cin>>n;
    cout<<"Que estructura quiere utilizar? (Los números son
aleatorios"<<endl;
    cout<<"1) Vector fijo "<<endl;
    cout<<"2) Vector dinamico"<<endl;
    cout<<"3) Vector sin repeticion"<<endl;
    cout<<"4) Matriz"<<endl;
    cout<<"5) Diccionario"<<endl;
    cout<<"6) Pila"<<endl;
    cout<<"7) Cola"<<endl;
    int aux;
    cin>>aux;
    switch (aux) {
        case 1:
            vectorfijo(n);
            break;
        case 2:
            vectordinamico(n);
            break;
        case 3:
            vectorsinrepeticion(n);
            break;
        case 4:
            matriz(n);
            break;
        case 5:
            Diccionario(n);
            break;
        case 6:
            pila(n);
            break;
        case 7:
            cola(n);
            break;
    }
    return 0;
}

```

## PYTHON

```

from collections import deque
import random
def vector(n):
    arreglo=[]
    for i in (0,n):
        arreglo.append(random.randint(0,100))

    for i in (arreglo):
        print(i)

def matriz(n):
    matrix = []
    for i in range(n):
        row = []
        for j in range(n):
            row.append(random.randint(0,100))
        matrix.append(row)
    for i in range(n):
        for j in range(n):
            print(matrix[i][j],end = " ")
    print()

def vectorsinrepeticion(n):
    sinrep={0}
    for i in range(n):
        sinrep.add(random.randint(0,100))
    sinrep.remove(0)
    for i in sinrep :
        print(i)

def Diccionario( n):
    dicc={}
    valor = 0
    for i in range(n):
        dicc[random.randint(0,100)]= valor
        valor=valor+1
    for x in dicc:
        print(str(x)+":"+str(dicc[x]))

def pila(n) :
    mipila=[]
    for i in range(n):
        mipila.append(random.randint(0,100))
    while(len(mipila)!=0):
        print(mipila.pop())

def cola(n):
    micola=deque([])
    for i in range(n):
        micola.append(random.randint(0,100))

    while(len(micola)!=0):
        print(micola.popleft())

```

```
print("Inserte cantidad de datos")
n=int(input())
#No existe switch en python
print("Vector")
vector(n)
print("Vector sin repetición")
vectorsinrepeticion(n)
print("Diccionario")
Diccionario(n)
print("Pila")
pila(n)
print("Cola")
cola(n);
print("Matriz")
matriz(n)
```

## 13 Identidades matemáticas y geométricas

- $A = \pi r^2$  Área del círculo
- $x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$  fórmula cuadrática
- $r = \frac{1}{2}at^2 + v_0t + r_0$  movimiento acelerado
- $\lim_{n \rightarrow \infty} \left(1 + \frac{1}{n}\right)^n = e$  límite
- $\frac{a}{b} \times \frac{c}{d} = \frac{ac}{bd}$  multiplicación de fracciones
- $\left(\frac{a}{b}\right)^n = \frac{a^n}{b^n}$  Potencia de un cociente
- $(a^n)^m = a^{nm}$  Potencia de una potencia
- $a^n a^m = a^{n+m}$  Primera ley de los exponentes
- $(ab)^n = a^n b^n$  Segunda ley de los exponentes
- $\frac{a}{b} + \frac{c}{d} = \frac{ad+bc}{bd}$  Suma de fracciones
- $S = 4\pi r^2$  Área de la superficie de una esfera
- $A = \frac{1}{2}bh$  Área de un triángulo
- $V = \frac{4}{3}\pi r^3$  Volumen de una esfera
- $\sum_{i=1}^n c = nc$  Sumatoria constante
- $\sum_{i=k}^n c = (n - k + 1)c$  Sumatoria constante desde k
- $\sum_{i=1}^n i = \frac{n(n+1)}{2}$  Sumatoria de 1 hasta n
- $\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6}$  Sumatoria de los cuadrados de 0 a n
- $\sum_{i=0}^n i^3 = \left(\frac{n(n+1)}{2}\right)^2$  Sumatoria de los cubos de 0 a n
- $\sum_{i=0}^n i^4 = \frac{n(n+1)(2n+1)(3n^2+3n-1)}{30}$  Sumatorias de las potencias de 4 de 0 a n
- $\sum_{i=0}^n 2^i = 2^{n+1} - 1$  Sumatoria de las potencias de 2 de 0 a n
- $\sum_{i=0}^n a^i = \frac{1+a^{n+1}}{1-a}$  Sumatoria de las potencias de a desde 0 a n
- $\log_b a = \frac{\log_c a}{\log_c b}$

- $\log \frac{a}{b} = \log a - \log b$
- $\log(axb) = \log a + \log b$
- $G = \left( \frac{x_1+x_2+x_3}{3}, \frac{y_1+y_2+y_3}{3} \right)$  Baricentro de un triángulo
- $P(n, r) = \frac{n!}{(n-r)!}$  Número de permutaciones de n objetos tomando r objetos sin repetición
- $C(n, r) = \frac{n!}{r!(n-r)!}$  Número de combinaciones de n objetos tomando r objetos
- $P(n, r) = n^r$  Número de permutaciones de n objetos tomando r objetos con repetición
- $C_n = \frac{(2n)!}{(n+1)!n!}$  n Número de Catalan
- $\frac{a}{\sin(A)} = \frac{b}{\sin(B)} = \frac{c}{\sin(C)}$  Ley del seno
- $c^2 = a^2 + b^2 - 2ab * \cos(C)$  Ley del coseno

# 14 Algoritmos de Búsquedas

## 14.1 Binary Search

### Búsqueda Binaria

0	1	2	3	4	5	6	7	8	9	
4	6	10	12	17	25	29	30	41	44	<b>28 &gt; 17</b>
					25	29	30	41	44	<b>28 &gt; 30</b>
					25	29				<b>28 &gt; 25</b>
						29				<b>28 != 39</b>

Se determinó que el elemento no está haciendo solo 4 comparaciones!

Guía del programador competitivo.

Ilustración 14-1 Ejemplo de búsqueda binaria

La búsqueda binaria, también conocida como búsqueda de intervalo medio o búsqueda logarítmica, es un algoritmo de búsqueda que encuentra la posición de un valor en un arreglo ordenado. Compara el valor con el elemento en el medio del arreglo, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda sigue en la mitad restante hasta que el valor se encuentre.

La búsqueda binaria es computada en el peor de los casos en un tiempo logarítmico, realizando  $O(\log n)$  comparaciones, donde  $n$  es el número de elementos del arreglo y  $\log$  es el logaritmo.

### Código

```
// Implementación Java de una búsqueda binaria recursiva  
public class BinarySearch {
```

```

/*Retorna el indice de x si se encuentra presente
en arr[1,2,...,r]*, si no retorna -1*/
static int binarySearch(int arr[], int l, int r, int x) {
    if (r >= 1) {
        int mid = l + (r - 1) / 2;
        /*Si el elemento esta presente en el medio*/
        if (arr[mid] == x) {
            return mid;
        }
        /*Si el elemento es más pequeño que la mitad
         entonces solo puede estar presente en el
         subarreglo izquierdo*/
        if (arr[mid] > x) {
            return binarySearch(arr, l, mid - 1, x);
        }
        //Si no el elemento solo puede estar presente
        //en el subarreglo derecho
        return binarySearch(arr, mid + 1, r, x);
    }
    //Si llegamos aqui, el elemento no esta presente
    //en el arreglo
    return -1;
}

public static void main(String args[]) {
    int arr[] = {2, 3, 4, 10, 40};
    int n = arr.length;
    int x = 10;
    int result = binarySearch(arr, 0, n - 1, x);
    if (result == -1) {
        System.out.println("Elemento no presente");
    } else {
        System.out.println("Elemento encontrado en la posición " + result);
    }
}

```

## 14.2 Exponential Search

El nombre de este algoritmo de búsqueda puede ser engañoso, ya que funciona en tiempo O (Log n). El nombre proviene de la forma en que busca un elemento. La búsqueda binaria exponencial es particularmente útil para búsquedas ilimitadas, donde el tamaño de la matriz es infinito.

### Código

```
//Programa Java para encontrar un elemento
```

```

//x en un array ordenado usando Exponential Search

import java.util.Arrays;

public class ExponentialSearch {

    /* Retorna posición de la primera ocurrencia
       de x en un arreglo*/
    static int exponentialSearch(int arr[],
        int n, int x) {
        // Si x esta presente en la primera localización en si misma
        if (arr[0] == x) {
            return 0;
        }
        /*Encuentra rango para la búsqueda binaria
           por repetidos dobles*/
        int i = 1;
        while (i < n && arr[i] <= x) {
            i = i * 2;
        }
        //llama la búsqueda binaria para el rango encontrado
        return Arrays.binarySearch(arr, i / 2,
            Math.min(i, n), x);
    }

    public static void main(String args[]) {
        int arr[] = {2, 3, 4, 10, 40};
        int x = 10;
        int result = exponentialSearch(arr, arr.length, x);

        System.out.println((result < 0)
            ? "El elemento no está presente en el array"
            : "Elemento encontrado en : "
            + result);
    }
}

```

## 14.3 Array Max/Min Search

Cuando tenemos un Array en donde queremos buscar el valor más pequeño y el valor más grande del mismo de forma eficiente haciendo la menor cantidad de comparaciones.

### Código

```

/* Programa Java que busca el maximo y el minimo
en un array*/
public class ArrayMaxMinSearch {
    /*Clase Pair es usada para retornar
    dos valores de getMinMax()*/
    static class Pair {

        int min;

```

```

        int max;
    }

    static Pair getMinMax(int arr[], int n) {
        Pair minmax = new Pair();
        int i;
/*Si es el unico elemento, lo retorna como min y max*/
        if (n == 1) {
            minmax.max = arr[0];
            minmax.min = arr[0];
            return minmax;
        }
/*Si hay más de un elemento,
   entonces inicializa min y más*/
        if (arr[0] > arr[1]) {
            minmax.max = arr[0];
            minmax.min = arr[1];
        } else {
            minmax.max = arr[1];
            minmax.min = arr[0];
        }
        for (i = 2; i < n; i++) {
            if (arr[i] > minmax.max) {
                minmax.max = arr[i];
            } else if (arr[i] < minmax.min) {
                minmax.min = arr[i];
            }
        }
        return minmax;
    }
    public static void main(String args[]) {
        int arr[] = {1000, 11, 445, 1, 330, 3000};
        int arr_size = 6;
        Pair minmax = getMinMax(arr, arr_size);
        System.out.printf("\nEl minimo elemento es %d", minmax.min);
        System.out.printf("\nEl maximo elemento es %d", minmax.max);
    }
}

```

## 14.4 Fibonacci Search

La búsqueda de Fibonacci es una técnica basada en la comparación que utiliza los números de Fibonacci para buscar un elemento en un array ordenado.

**Similitudes con la búsqueda binaria:**

- Trabaja con arrays ordenados.

- Es un algoritmo de dividir y conquistar.
- Tiene una complejidad de registro y tiempo.

### Diferencias con la búsqueda binaria:

- La búsqueda de Fibonacci divide un array dado en partes desiguales
- La búsqueda binaria usa el operador de división para dividir el rango. La búsqueda de Fibonacci no usa /, pero usa + y -. El operador de la división puede ser costoso en algunas CPU.
- La búsqueda de Fibonacci examina elementos relativamente más cercanos en pasos subsiguientes. Por lo tanto, cuando la matriz de entrada es grande y no cabe en la memoria caché de la CPU o incluso en la RAM, la búsqueda de Fibonacci puede ser útil.
- La idea es encontrar primero el número de Fibonacci más pequeño que sea mayor o igual a la longitud de la matriz dada.

## Código

```
// Programa Java para Fibonacci Search

public class FibonacciSearch {

    //Función de utilidad para buscar el minimo
    //de dos elementos
    public static int min(int x, int y) {
        return (x <= y) ? x : y;
    }

    /* Regresa el indice de x si esta presente, si no retorna -1*/
    public static int fibMonaccianSearch(int arr[],
    int x, int n) {
        /* Inicializa los números fibonacci*/
        int fibMMm2 = 0; // (m-2)esimo número fibonacci.
        int fibMMm1 = 1; // (m-1)esimo número fibonacci.
        int fibM = fibMMm2 + fibMMm1; // mesimo fibonacci.

        /* FibM va a ser almacenado como el más pequeño
           fibonacci más grande o igual a n*/
        while (fibM < n) {
            fibMMm2 = fibMMm1;
            fibMMm1 = fibM;
            fibM = fibMMm2 + fibMMm1;
        }
    }
}
```

```

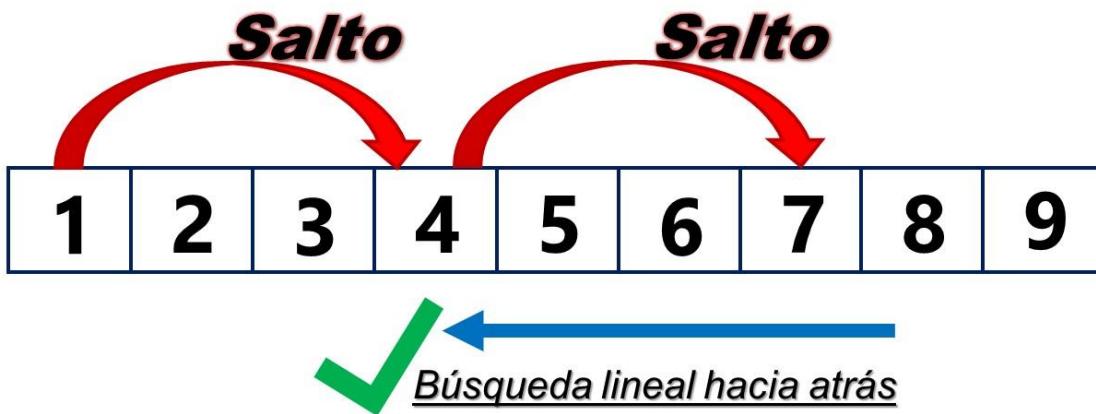
// Marca el rango eliminado del frente
int offset = -1;
/* Si bien hay elementos a inspeccionar.
Tenga en cuenta que comparamos arr [fibMm2] con x.
Cuando fibM se convierte en 1, fibMm2 se convierte en 0 */
while (fibM > 1) {
//Revisa si fibMm2 es una ubicación valida
int i = min(offset + fibMMm2, n - 1);
/*Si x es más grande que el valor
en el indice fibMm2, fuera del subarray
desde offset a i*/
if (arr[i] < x) {
    fibM = fibMMm1;
    fibMMm1 = fibMMm2;
    fibMMm2 = fibM - fibMMm1;
offset = i;
} /*Si x es más grande que el valor en indice
fibMm2, corta el subarreglo despues de i+1*/
else if (arr[i] > x) {
    fibM = fibMMm2;
    fibMMm1 = fibMMm1 - fibMMm2;
fibMMm2 = fibM - fibMMm1;
} /* Elemento encontrado, retorna indice*/
else {
    return i;
}
}
/* Comparando el ultimo elemento con x*/
if (fibMMm1 == 1 && arr[offset + 1] == x) {
return offset + 1;
}
/*Elemento no encontrado, retorna -1*/
return -1;
}

public static void main(String[] args) {
    int arr[] = {10, 22, 35, 40, 45, 50,
                80, 82, 85, 90, 100};
    int n = 11;
    int x = 85;
    System.out.println("Found at index: "
+ fibMonaccianSearch(arr, x, n));
}
}

```

## 14.5 Jump Search

# Jump Search



*Guía del programador competitivo.*

Ilustración 14-2 Ejemplo búsqueda por saltos

Al igual que Binary Search, Jump Search es un algoritmo de búsqueda para matrices ordenadas. La idea básica es verificar menos elementos (que la búsqueda lineal) saltando hacia delante con pasos fijos u salteando algunos elementos en lugar de buscar todos los elementos.

En el peor de los casos, tenemos que realizar saltos de  $n / m$  y, si el último valor verificado es mayor que el elemento a buscar, realizamos comparaciones de  $m-1$  más para la búsqueda lineal. Por lo tanto, el número total de comparaciones en el peor de los casos será  $((n / m) + m-1)$ . El valor de la función  $((n / m) + m-1)$  será mínimo cuando  $m = \sqrt{n}$ . Por lo tanto, el mejor tamaño de paso es  $m = \sqrt{n}$ .

## Puntos importantes:

- Trabaja solo arreglos ordenados.
- El tamaño óptimo de un bloque a saltar es  $(\sqrt{n})$ . Esto hace que la complejidad del tiempo de Jump Search  $O(\sqrt{n})$ .

- La complejidad temporal de la búsqueda por salto es entre la búsqueda lineal ( $O(n)$ ) y la búsqueda binaria ( $O(\log n)$ ).
- La búsqueda binaria es mejor que la búsqueda por salto, pero la búsqueda por salto tiene la ventaja de que retrocedemos solo una vez (la búsqueda binaria puede requerir saltos  $O(\log n)$ ), considere una situación en la que el elemento a buscar es el elemento más pequeño o más pequeño que el más pequeño). Entonces, en un sistema donde el salto hacia atrás es costoso, usamos Jump Search.

## Código

```
//Programa Java que implementa Jump Search
public class JumpSearch {

    public static int jumpSearch(int[] arr, int x) {
        int n = arr.length;
        //Buscando el tamaño del bloque que sera
        //saltado
        int step = (int) Math.floor(Math.sqrt(n));
        /* Buscando el bloque donde el elemento
           esta presente (Si esta presente)*/
        int prev = 0;
        while (arr[Math.min(step, n) - 1] < x) {
            prev = step;
            step += (int) Math.floor(Math.sqrt(n));
        if (prev >= n) {
            return -1;
        }
        }
        /*Realizando una busqueda linear para x en
           el bloque empezando con prev*/
        while (arr[prev] < x) {
            prev++;
        }
        /*Si nosotros alcanzamos el siguiente bloque
           o el fin del array el elemento no esta presente*/
        if (prev == Math.min(step, n)) {
            return -1;
        }
        }
        // Si el elemento fue encontrado
        if (arr[prev] == x) {
            return prev;
        }
        return -1;
    }

    public static void main(String[] args) {
        int arr[] = {0, 1, 1, 2, 3, 5, 8, 13, 21,
                    34, 55, 89, 144, 233, 377, 610};
        int x = 55;
```

```

//Encontrar el indice de 'x' usando Jump Search
int index = jumpSearch(arr, x);
// Imprime el indice donde x fue encontrado
System.out.println("\nNúmero " + x
+ " esta en el indice " + index);
}
}

```

## 14.6 Minimum Absolute Sum Pair

Dos elementos cuya suma es la más cercana a cero.

Para cada elemento, encuentre la suma con cada otro elemento en el array y compare las sumas. Por último, devolver la suma mínima.

### Código

```

//Programa JAVA que busque el par de números
//que sumados de la minima suma absoluta

public class MinAbsSumPair {
    //función que busca el par
    static void minAbsSumPair(int arr[], int arr_size) {
        int inv_count = 0;
        int l, r, min_sum, sum, min_l, min_r;
    /*El array debe tener al menos 2 elementos*/
        if (arr_size < 2) {
            System.out.println("Invalid Input");
            return;
        }
    /* Inicialización de los valores*/
        min_l = 0;
        min_r = 1;
        min_sum = arr[0] + arr[1];
        for (l = 0; l < arr_size - 1; l++) {
            for (r = l + 1; r < arr_size; r++) {
                sum = arr[l] + arr[r];
                if (Math.abs(min_sum) > Math.abs(sum)) {
                    min_sum = sum;
                    min_l = l;
                    min_r = r;
                }
            }
        }
        System.out.println(" Los dos elementos los cuales "
            + "tienen la suma minima son "
            + arr[min_l] + " y " + arr[min_r]);
    }

    public static void main(String[] args) {

```

```

        int arr[] = {1, 60, -10, 70, -80, 85};
        minAbsSumPair(arr, 6);
    }

}

```

## 14.7 Missing Number Search

Se le da una lista de  $n-1$  enteros y estos enteros están en el rango de 1 a  $n$ . No hay duplicados en la lista. Falta uno de los enteros en la lista.

- 1) XOR todos los elementos del array, que el resultado de XOR sea  $X_1$ .
- 2) XOR todos los números del 1 al  $n$ , sea XOR sea  $X_2$ .
- 3) XOR de  $X_1$  y  $X_2$  da el número que falta.

### Código

```

//Programa java para buscar el número perdido
//usando xor
public class MissingNoSearch {
    //Función que busca el número perdido
    static int getMissingNo(int a[], int n) {
        int x1 = a[0];
        int x2 = 1;
        /*For XOR para todos los elementos
        del arreglo*/
        for (int i = 1; i < n; i++) {
            x1 = x1 ^ a[i];
        }
        /* For xor de todos los elementos de 1 a n+1 */
        for (int i = 2; i <= n + 1; i++) {
            x2 = x2 ^ i;
        }
        return (x1 ^ x2);
    }

    public static void main(String args[]) {
        int a[] = {1, 2, 3, 4, 5, 7, 8};
        int miss = getMissingNo(a, 7);
        System.out.println(miss);
    }
}

```

## 14.8 Difference Pair Search

Dada una matriz sin orden y un número n, busque si existe un par de elementos en la matriz cuya diferencia es n.

El método más simple es ejecutar dos bucles, el bucle externo selecciona el primer elemento (elemento más pequeño) y el bucle interno busca el elemento seleccionado por el bucle externo más n. La complejidad del tiempo de este método es O ( $n^2$ ).

Podemos usar la ordenación y la búsqueda binaria para mejorar la complejidad del tiempo a O ( $n \log n$ ). El primer paso es ordenar el array en orden ascendente. Una vez que el array esté ordenado, recorra el array de izquierda a derecha y, para cada elemento arr [i], la búsqueda binaria de arr [i] + n en arr [i + 1...n-1]. Si se encuentra el elemento, devuelva el par.

Tanto el primer como el segundo paso toman O ( $n \log n$ ). Así que la complejidad global es O ( $n \log n$ ).

El segundo paso del algoritmo anterior se puede mejorar a O (n). El primer paso sigue siendo el mismo. La idea para el segundo paso es tomar dos variables de índice i y j, inicializarlas como 0 y 1 respectivamente. Ahora ejecuta un bucle lineal. Si arr [j] - arr [i] es más pequeño que n, debemos buscar un arr mayor [j], así que aumente j. Si arr [j] - arr [i] es mayor que n, debemos buscar un arr mayor [i], entonces incremente i.

## Código

```
// Programa Java para buscar un par
// Dada una diferencia
public class PairDifferenceSearch {
    //La función asume que el array esta ordenado
    static boolean findPair(int arr[], int n) {
        int size = arr.length;
        // Inicializa la posición de dos elementos
        int i = 0, j = 1;
        // Busca por el par
        while (i < size && j < size) {
            if (i != j && arr[j] - arr[i] == n) {
                System.out.print("Par Encontrado: "
                    + "(" + arr[i] + ", " + arr[j] + ")");
                return true;
            } else if (arr[j] - arr[i] < n) {
                i++;
            } else {
                j++;
            }
        }
        return false;
    }
}
```

```

j++;
    } else {
        i++;
    }
}
//No encuentra el par
System.out.print("No hay tal par");
return false;
}
public static void main(String[] args) {
    int arr[] = {1, 8, 30, 40, 100};
    int n = 60;
    findPair(arr, n);
}
}

```

## 14.9 Ternary Search

La búsqueda ternaria es un algoritmo de dividir y conquistar que se puede usar para encontrar un elemento en un array. Es similar a la búsqueda binaria donde dividimos el array en dos partes pero en este algoritmo. En esto, dividimos el array dado en tres partes y determinamos cuál tiene la clave (elemento buscado). Podemos dividir el array en tres partes tomando mid1 y mid2, que se pueden calcular como se muestra a continuación.

$$\text{mid1} = l + (r-l) / 3$$

$$\text{mid2} = r - (r-l) / 3$$

Inicialmente, l y r serán iguales a 0 y n-1 respectivamente, donde n es la longitud de la matriz.

El array debe ordenarse para realizar una búsqueda ternaria en ella.

### Código

```

//Programa Java para ilustrar
//recursivamente la aproximación
//de una búsqueda ternaria

public class TernarySearch {

    //Función que realiza la búsqueda ternaria
    static int ternarySearch(int l, int r, int key, int ar[]) {
        if (r >= l) {
            //encontrar el mid1 y mid 2
            int mid1 = l + (r - l) / 3;

```

```

        int mid2 = r - (r - 1) / 3;
//Verificar si la key esta presente en algun medio
if (ar[mid1] == key) {
    return mid1;
}
if (ar[mid2] == key) {
    return mid2;
}
/*Desde que la key no este presente en el mid
verifica en cada region si esta presente
luego repite la operación de búsqueda
en esa región*/
if (key < ar[mid1]) {
//La key yace entre 1 y mid1
return ternarySearch(l, mid1 - 1, key, ar);
} else if (key > ar[mid2]) {
//La key yace entre mid2 y r
return ternarySearch(mid2 + 1, r, key, ar);
} else {
//La key yace entre mid 1 y mid 2
return ternarySearch(mid1 + 1, mid2 - 1, key, ar);
}
}
// key no encontrada
return -1;
}

public static void main(String args[]) {
int l, r, p, key;
//Cree el array y ordenelo si no lo está
int ar[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
// Indice inicial
l = 0;
// Tamaño del arreglo
r = 9;
// Buscando por el 5
// key que va a ser buscada en el array
key = 5;
// Busca usando Ternary Search
p = ternarySearch(l, r, key, ar);
// Imprime el resultado
System.out.println("Indice de " + key + " es " + p);
// Buscando por el 50
// Key a ser buscada en el array
key = 50;
// Buscar usando Ternary Search
p = ternarySearch(l, r, key, ar);
// Imprime el resultado
System.out.println("Index of " + key + " is " + p);
}
}

```

## 14.10 Problemas de repaso

## Ejercicios en Online Judge

246-Count on Cantor

10277-Boastin' Red Socks

714-Copying Books

10611-The Playboy Chimp

978-Lemmings Battle!

10152-ShellSort

## 15 Algoritmos de Ordenamiento

### 15.1 Recursive Bubble Sort

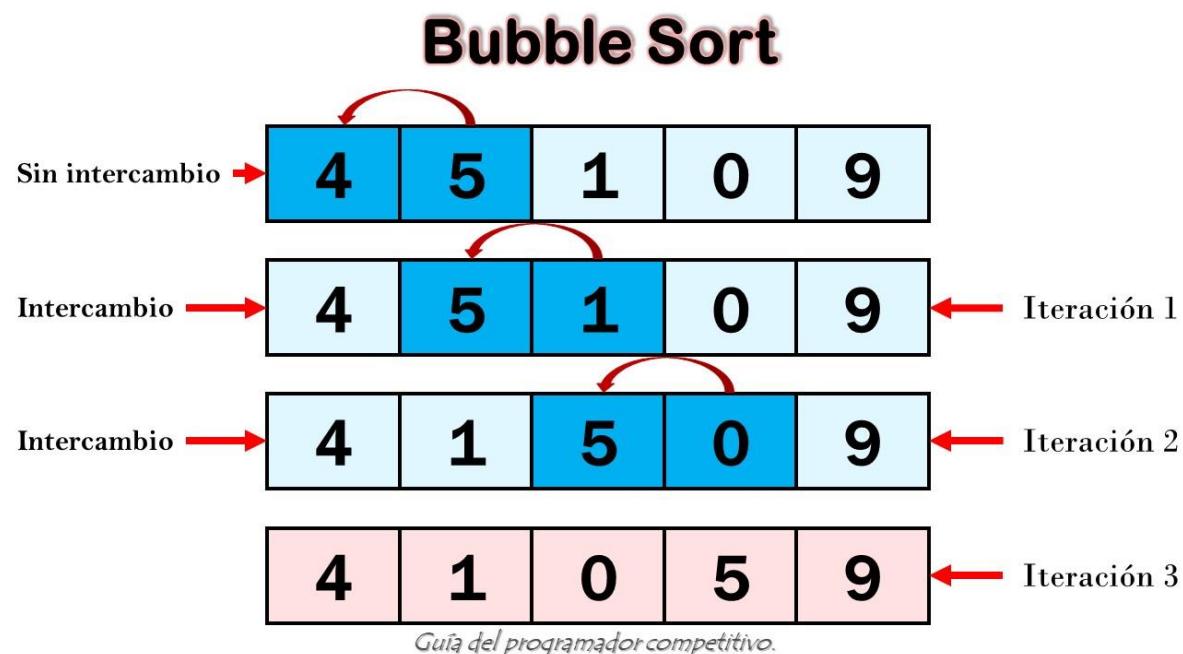


Ilustración 15-1 Ejemplo ordenamiento burbuja

Ordenamiento burbuja es el algoritmo de ordenamiento más simple que existe, funciona intercambiando repetidamente los elementos adyacentes si están en orden incorrecto.

### Código

```
//Programa java que realiza Bubble Sort Recursivo  
import java.util.Arrays;
```

```

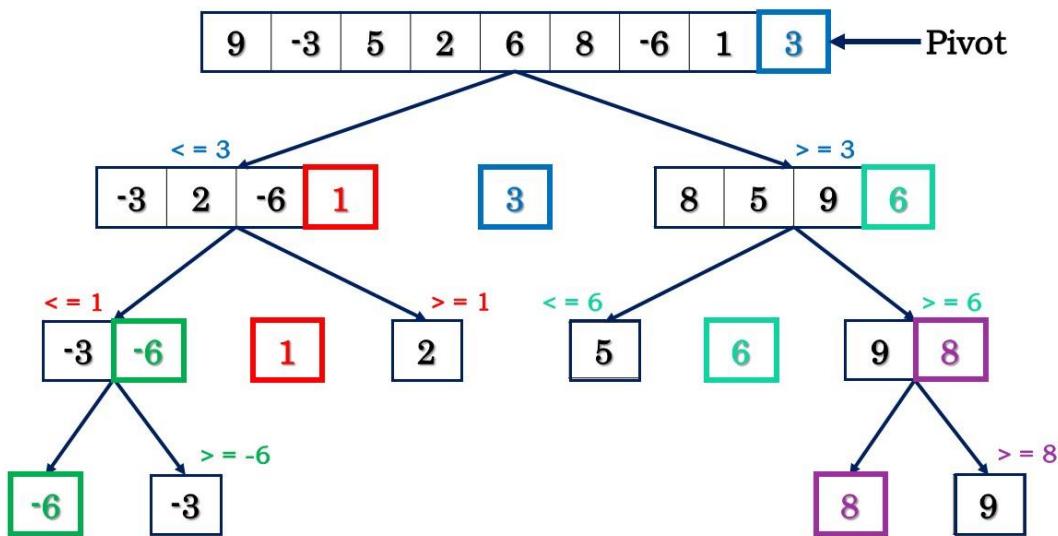
public class RecursiveBubbleSort
{
    static void bubbleSort(int arr[], int n)
    {
        // Caso base
        if (n == 1)
            return;
        //Un paso de Bubble Sort, luego de este
        //paso, el elemento más largo es movido
        // hasta el final
        for (int i=0; i<n-1; i++)
            if (arr[i] > arr[i+1])
                { //intercambia arr[i], arr[i+1]
                    int temp = arr[i];
                    arr[i] = arr[i+1];
                    arr[i+1] = temp;
                }
        bubbleSort(arr, n-1);
    }

    public static void main(String[] args)
    {
        int arr[] = {64, 34, 25, 12, 22, 11, 90};
        bubbleSort(arr, arr.length);
        System.out.println("Array ordenado : ");
        System.out.println(Arrays.toString(arr));
    }
}

```

## 15.2 Quick Sort

# Quicksort



Guía del programador competitivo.

Ilustración 15-2 Ejemplo ordenamiento rápido

Quick Sort es un algoritmo de Dividir y Conquistar. Selecciona un elemento como pivote y divide el array dado alrededor del pivote seleccionado. Hay muchas versiones diferentes de quick Sort que seleccionan pivote de diferentes maneras.

- Elije siempre el primer elemento como pivote.
- Siempre elije el último elemento como pivote.
- Elige un elemento aleatorio como pivote.
- Elije la mediana como pivote.

El proceso clave en quick Sort es partition (). El destino de las particiones es, dada un array y un elemento x del array como pivote, coloca x en su posición correcta en el array ordenado y coloca todos los elementos más pequeños (más pequeños que x) antes de x, y coloca todos los elementos mayores (mayores que x) después de X. Todo esto debe hacerse en tiempo lineal.

## Código

```
// Programa java para la implementación de QuickSort
```

```
public class QuickSort {
```

```

/* Esta función toma el ultimo elemento como
pivot, coloca el pivot en la posición correcta
en el array ordenado, y coloca todos los más
pequeños (Más pequeños que el pivot) a la izquierda
del pivot, y todos los más grandes a la derecha del
pivot*/
static int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1); // Indice del elemento más pequeño
    for (int j = low; j < high; j++) {
        /*Si el actual elemento es más pequeño
         * que el pivot*/
        if (arr[j] <= pivot) {
            i++;
            //intercambia arr[i] y arr[j]
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }

    // Intercambia arr[i+1] y arr[high] (o pivot)
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;

    return i + 1;
}

/*arr[] --> Array que sera ordenado,
low --> indice inicial,
high --> indice final */
static void sort(int arr[], int low, int high) {
    if (low < high) {
        /* pi esta particionando indices, arr[pi] is
           now at right place */
        int pi = partition(arr, low, high);
        // Recursivamente ordena elementos antes de
        // la partición y despues de la partición
        sort(arr, low, pi - 1);
        sort(arr, pi + 1, high);
    }
}

/* Imprimir array */
static void printArray(int arr[]) {
    int n = arr.length;
    for (int i = 0; i < n; ++i) {
        System.out.print(arr[i] + " ");
    }
    System.out.println();
}

public static void main(String args[]) {

```

```

        int arr[] = {10, 7, 8, 9, 1, 5};
        int n = arr.length;
        sort(arr, 0, n - 1);
        System.out.println("Array ordenado");
printArray(arr);
    }
}

```

## 15.3 Radix Sort

La idea de Radix Sort es hacer una clasificación de dígito por dígito desde el dígito menos significativo hasta el dígito más significativo. La ordenación de Radix usa la ordenación de conteo como una subrutina para ordenar.

Si tenemos bits  $\log_2 n$  para cada dígito, el tiempo de ejecución de Radix parece ser mejor que Quick Sort para una amplia gama de números de entrada. Los factores constantes ocultos en la notación asintótica son mayores para Radix Sort y Quick-Sort usa cachés de hardware de manera más efectiva.

### Código

```

//Implementación java de Radix Sort

import java.util.*;

public class RadixSort {

    // Una función de utilidad que obtiene
    // El maximo valor en arr[]
    static int getMax(int arr[], int n) {
        int mx = arr[0];
        for (int i = 1; i < n; i++) {
            if (arr[i] > mx) {
                mx = arr[i];
            }
        }
        return mx;
    }

    /*Una función que realiza conteo de ordenamiento en
    arr[] de acuerdo al digito
    representado como exp*/
    static void countSort(int arr[], int n, int exp) {
        int output[] = new int[n]; // Array de salida
        int i;
        int count[] = new int[10];

```

```

Arrays.fill(count, 0);
    // Almacena el conteo de las ocurrencias en count[]
for (i = 0; i < n; i++) {
    count[(arr[i] / exp) % 10]++;
}
// cambia count[i] de tal manera que ahora
    //contenga la actual posicion de este digito en
//output[]
    for (i = 1; i < 10; i++) {
        count[i] += count[i - 1];
    }
//Construye el array de salida
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
    }
count[(arr[i] / exp) % 10]--;
}
// copia el array de salida a arr[], ahora arr[]
    // contiene los números ordenados de acuerdo al digito
//actual
    for (i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}

static void radixsort(int arr[], int n) {
// Encuentra el maximo número para conocer el número
    // de digitos
    int m = getMax(arr, n);
for (int exp = 1; m / exp > 0; exp *= 10) {
    countSort(arr, n, exp);
}
}

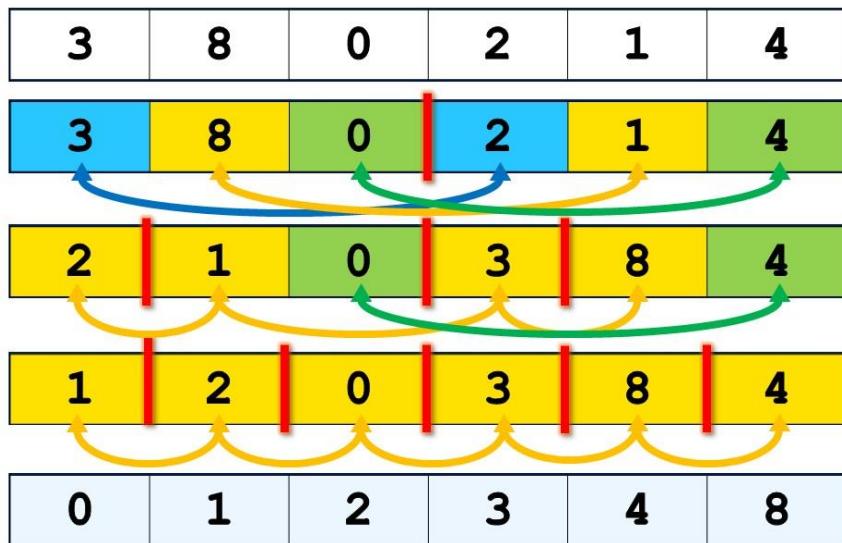
//Imprimir el array
static void print(int arr[], int n) {
    for (int i = 0; i < n; i++) {
        System.out.print(arr[i] + " ");
    }
}

public static void main(String[] args) {
    int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};
    int n = arr.length;
    radixsort(arr, n);
    print(arr, n);
}
}

```

## 15.4 Shell Sort

# Shell Sort



Guía del programador competitivo.

Ilustración 15-3 Ejemplo ordenamiento cascara

Shell Sort es principalmente una variación de Insertion Sort. En la ordenación por inserción, movemos los elementos solo una posición adelante. Cuando un elemento tiene que moverse mucho más adelante, hay muchos movimientos involucrados. La idea de shell Sort es permitir el intercambio de elementos lejanos. En shell Sort, hacemos el array h-ordenado para un gran valor de h. Continuamos reduciendo el valor de h hasta que se convierte en 1. Se dice que un array está ordenado por h si todas las sublistas de cada elemento h están ordenadas.

La complejidad de tiempo de la implementación de shell Sort es  $O(n^2)$ . En la implementación la brecha se reduce a la mitad en cada iteración.

## Código

```
// Implementación java de ShellSort
public class ShellSort
{ /*Imprimir el array */
    static void printArray(int arr[])
    {
        int n = arr.length;
        for (int i=0; i<n; ++i)
            System.out.print(arr[i] + " ");
        System.out.println();
    }
}
```

```

    }
    //Función ordenadora Shell Sort
static int sort(int arr[])
{
    int n = arr.length;
    //Inicia con un gran salto, luego reduce el salto
    for (int gap = n/2; gap > 0; gap /= 2)
    {
        //Realiza un insertion sort con salto
        // Los primeros elementos del sato a[0..gap-1]
        // estan ya en un orden de salto que sigue agregando
        //un elemento más hasta que el array entero este ordenado
    for (int i = gap; i < n; i += 1)
    {
        //Agrega a[i] a los elementos que estan en el ordenamiento
        // con salto
        int temp = arr[i];
        int j;
        for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)
            arr[j] = arr[j - gap];
        arr[j] = temp;
    }
}
return 0;
}

public static void main(String args[])
{
    int arr[] = {12, 34, 54, 2, 3};
    System.out.println("Array antes de ordenar");
    printArray(arr);
    sort(arr);
    System.out.println("Array despues de ordenar");
    printArray(arr);
}
}

```

## 15.5 Tim Sort

TimSort es un algoritmo de ordenamiento basado en Insertion Sort y Merge Sort.

- Es un algoritmo de ordenamiento estable funciona en tiempo  $O(n \log n)$
- Se utiliza en `Arrays.sort()` de Java, así como en Python `sorted()` y C++ `sort()`
- Primero clasifica las piezas pequeñas utilizando el ordenamiento de Inserción, luego fusiona las piezas utilizando la combinación de ordenación por fusión (Merge).

Dividimos el array en bloques conocidos como Run. Ordenamos esas ejecuciones utilizando la ordenación por inserción una por una y luego las combinamos utilizando la función de combinación utilizada en Merge Sort. Si el tamaño del array es menor que run, entonces el array se clasifica simplemente utilizando el orden de inserción.

El tamaño de run puede variar de 32 a 64, dependiendo del tamaño del array. Tenga en cuenta que la función de combinación funciona bien cuando los arreglos secundarios de tamaño son potencias de 2. La idea se basa en el hecho de que la ordenación por inserción funciona bien para arreglos pequeños.

## Código

```
// Programa Java que realiza TimSort

public class TimSort {

    static int RUN = 32;

    // esta función ordena el arreglo
    //desde el índice izquierdo hasta
    //a la derecha el índice que es de tamaño más alto RUN
    public static void insertionSort(int[] arr, int left, int right) {
        for (int i = left + 1; i <= right; i++) {
            int temp = arr[i];
            int j = i - 1;
            while (arr[j] > temp && j >= left) {
                arr[j + 1] = arr[j];
                j--;
            }
            arr[j + 1] = temp;
        }
    }

    public static void merge(int[] arr, int l,
    int m, int r) { //Array original esta separado en dos partes
        // array derecho e izquierdo
        int len1 = m - l + 1, len2 = r - m;
        int[] left = new int[len1];
        int[] right = new int[len2];
        for (int x = 0; x < len1; x++) {
            left[x] = arr[l + x];
        }
        for (int x = 0; x < len2; x++) {
            right[x] = arr[m + 1 + x];
        }

        int i = 0;
        int j = 0;
```

```

        int k = 1;
    // despues de comparar,
    // after comparing,unimos los dos array
    // en un subarray más largo
    while (i < len1 && j < len2) {
if (left[i] <= right[j]) {
            arr[k] = left[i];
            i++;
        } else {
            arr[k] = right[j];
}
j++;
    }
//Copia los elementos restantes de la izquierda, si hay
while (i < len1) {
    arr[k] = left[i];
k++;
    i++;
}

// Copia los elementos restantes de la derecha, si hay
while (j < len2) {
    arr[k] = right[j];
k++;
j++;
}
}

public static void timSort(int[] arr, int n) {
//Ordena individualmente los subarrays de
//tamaño RUN
for (int i = 0; i < n; i += RUN) {
    insertionSort(arr, i, Math.min((i + 31), (n - 1)));
}
//Comienza a unir de tamaño run, luego
//podrá unir de tamaño 64, 128, 256 y así...
for (int size = RUN; size < n; size = 2 * size) {
// Toma un punto inicial del subarray izquierdo
    // nosotros vamos a unir arr[left..left+size-1] y
//arr[left+size, left+2*size-1], luego de cada unión
// nosotros incrementamos izquierda en 2*size
    for (int left = 0; left < n; left += 2 * size) {
//Encontramos punto de finalización de
    // el subarray izquierdo, mid+1
    // mid+1 es el punto inicial del
//subarray derecho
        int mid = left + size - 1;
        int right = Math.min((left + 2 * size - 1), (n - 1));

    // une sub array arr[left.....mid] y
    // arr[mid+1...right]
        merge(arr, left, mid, right);
    }
}
}

```

```

}

// Imprimir el array
public static void printArray(int[] arr, int n) {
    for (int i = 0; i < n; i++) {
        System.out.print(arr[i] + " ");
    }
    System.out.print("\n");
}

public static void main(String[] args) {
    int[] arr = {5, 21, 7, 23, 19};
    int n = arr.length;
    System.out.print("Given Array is\n");
    printArray(arr, n);
    timSort(arr, n);
    System.out.print("After Sorting Array is\n");
    printArray(arr, n);
}
}

```

## 15.6 Tree Sort

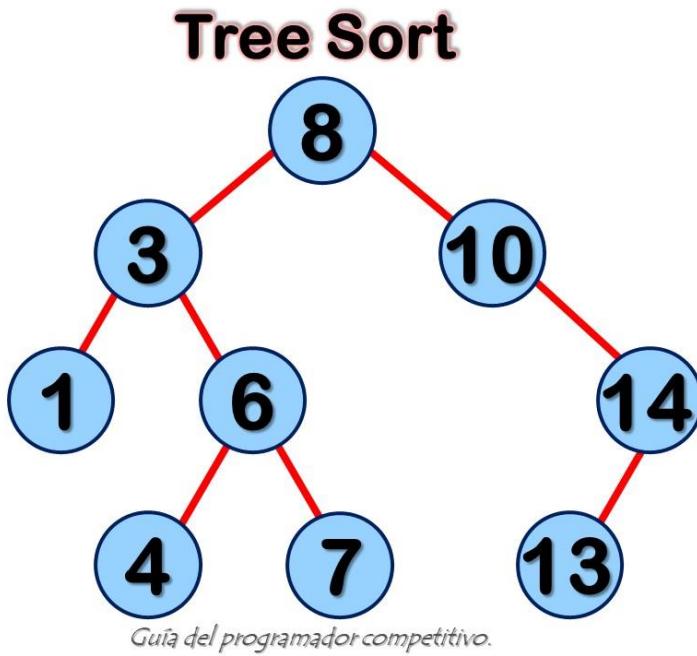


Ilustración 15-4 Vista abstracta de un árbol binario de búsqueda

Ordenamiento árbol es un algoritmo de clasificación que se basa en la estructura de datos del Binary Search Tree. Primero crea un árbol de búsqueda binario a partir de los elementos

de la lista de entrada o un array luego realiza un recorrido inorden en el árbol de búsqueda binario creado para obtener los elementos ordenados.

Su complejidad de tiempo es  $O(n\log n)$  en el mejor de los casos y  $O(n^2)$  en el peor.

## Código

```
//Código java que implementa Tree Sort
public class TreeSort {
    //Clase conteniendo hijo izquierdo
    //y derecho del nodo actual, y
    // el valor del nodo (Key)
    static class Node {
        int key;
        Node left, right;

        public Node(int item) {
            key = item;
            left = right = null;
        }
    }

    // Raíz del arbol de búsqueda
    //binaria
    static Node root;
    //Constructor inicial
    static void init() {
        root = null;
    }
    //Este metodo principalmente
    static void insert(int key) {
        root = insertRec(root, key);
    }
    /*Una función recursiva que inserta
     nuevos valores al arbol de búsqueda binaria*/
    static Node insertRec(Node root, int key) {
        /*Si el arbol esta vacio
         Retorne un nuevo nodo*/
        if (root == null) {
            root = new Node(key);
        }
        return root;
    }
    /*De otra forma, baja por el arbol*/
    if (key < root.key) {
        root.left = insertRec(root.left, key);
    } else if (key > root.key) {
        root.right = insertRec(root.right, key);
    }
    /*Retorna la raiz*/
    return root;
}
/*Una función que realiza recorrido
 inorden atraves del arbol*/
```

```

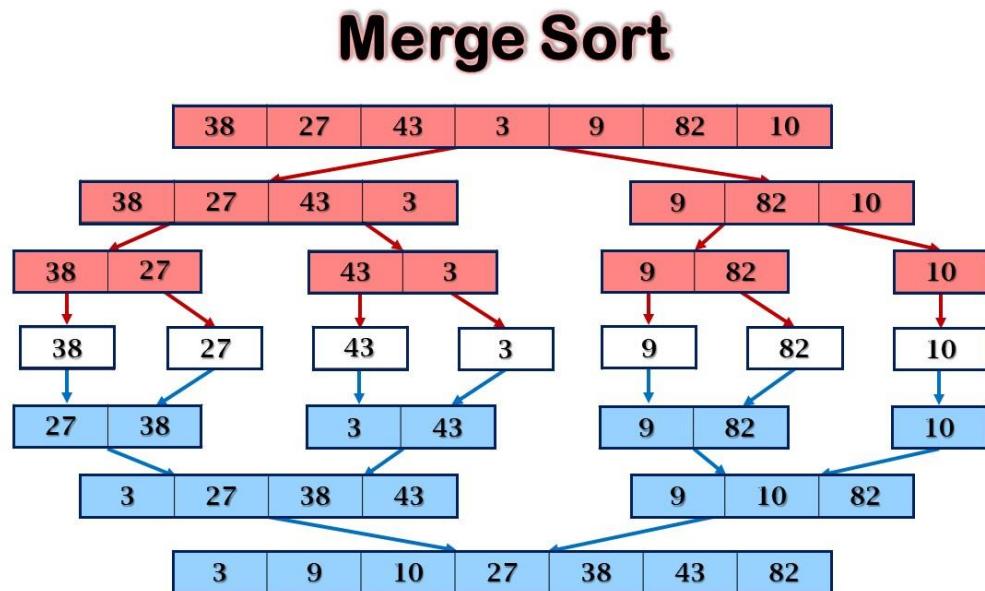
static void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.print(root.key + " ");
        inorderRec(root.right);
    }
}

static void treeins(int arr[]) {
    for (int i = 0; i < arr.length; i++) {
        insert(arr[i]);
    }
}

public static void main(String[] args) {
    int arr[] = {5, 4, 7, 2, 11};
    treeins(arr);
    inorderRec(root);
}
}

```

## 15.7 Merge Sort



*Guía del programador competitivo.*

Ilustración 15-5 Ejemplo de ordenamiento por unión

Al igual que Quick Sort, Merge Sort es un algoritmo de Dividir y Conquistar. Divide el arreglo de entrada en dos mitades, se llama a sí misma para las dos mitades y luego combina las dos mitades clasificadas. La función merge () se usa para fusionar dos mitades.

El merge (arr, l, m, r) es un proceso clave que asume que arr [l...m] y arr [m + 1...r] están ordenados y combina los dos subarreglos ordenados en uno solo.

Merge Sort es útil para ordenar listas enlazadas en tiempo O (nLogn). En el caso de listas enlazadas, el caso es diferente principalmente debido a la diferencia en la asignación de memoria de los arrays y las listas enlazadas. A diferencia de los arrays, los nodos de listas enlazadas pueden no estar adyacentes en la memoria. A diferencia del array, en la lista enlazada, podemos insertar elementos en el medio en O (1) espacio adicional y O (1) tiempo. Por lo tanto, la operación de fusión de la ordenación de fusión se puede implementar sin espacio adicional para las listas vinculadas.

En arrays, podemos hacer acceso aleatorio ya que los elementos son continuos en la memoria.

## Código

```
/*Programa java para Merge Sort*/
public class MergeSort
{
    /*Une dos subarrays de arr[]
     Primer subarray es arr[1...m]
     Segundo subarray es arr[m+1..r]*/
    static void merge(int arr[], int l, int m, int r)
    { //Encuentra tamaños de dos subarrays a ser unidos
        int n1 = m - l + 1;
        int n2 = r - m;
        /* Crear arrays temporales */
        int L[] = new int [n1];
        int R[] = new int [n2];
        /*Copia datos en los arrays temporales*/
        for (int i=0; i<n1; ++i)
            L[i] = arr[l + i];
        for (int j=0; j<n2; ++j)
            R[j] = arr[m + 1+ j];
        /*Une los arreglos temporales*/
        // Indices iniciales del los dos subarrays
        int i = 0, j = 0;
        //indice inicial de array unido
```

```

int k = 1;
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
        }
        j++;
    }
    /*Copia los elementos restantes de L[] si hay*/
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }
    //Copia los elementos restantes de R[] si hay
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}
static void sort(int arr[], int l, int r)
{
    if (l < r)
    {
        //Encuentra el punto medio
        int m = (l+r)/2;
        // Ordena los dos subarrays
        sort(arr, l, m);
        sort(arr , m+1, r);
        // Une los subarrays
        merge(arr, l, m, r);
    }
}

//Imprimir array
static void printArray(int arr[])
{
    int n = arr.length;
    for (int i=0; i<n; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}

public static void main(String args[])

```

```

{
    int arr[] = {12, 11, 13, 5, 6, 7};

    System.out.println("Array dado");
    printArray(arr);
    sort(arr, 0, arr.length-1);
    System.out.println("\nArray ordenado");
printArray(arr);
}
}

```

## 15.8 Binary Insertion Sort

Podemos usar la búsqueda binaria para reducir el número de comparaciones en la clasificación de inserción normal. La clasificación de inserción binaria utiliza la búsqueda binaria para encontrar la ubicación adecuada para insertar el elemento seleccionado en cada iteración.

En la clasificación de inserción normal, se necesitan comparaciones  $O(n)$  (en la iteración  $n$ ) en el peor de los casos. Podemos reducirlo a  $O(\log n)$  mediante la búsqueda binaria.

El algoritmo en su conjunto aún tiene un tiempo de ejecución en el peor de los casos de  $O(n^2)$  debido a la serie de intercambios necesarios para cada inserción.

### Código

```

// Programa java implementando
// Binary Insertion Sort

import java.util.Arrays;
public class BinaryInsertionSort
{
    public static void main(String[] args)
    {
        final int[] arr = {37, 23, 0, 17, 12, 72, 31,
                           46, 100, 88, 54 };

        sort(arr);
        for(int i=0; i<arr.length; i++)
            System.out.print(arr[i]+ " ");
    }

    static void sort(int array[])

```

```

{
    for (int i = 1; i < array.length; i++)
    {
        int x = array[i];
        // Encontrar ubicación para insertar usando
        // Binary Search
        int j = Math.abs(Arrays.binarySearch(array, 0, i, x) + 1);
        System.arraycopy(array, j, array, j+1, i-j);
        // Colocar elemento en su correcta localización
        array[j] = x;
    }
}

```

## 15.9 Bitonic Sort

Bitonic Sort es un algoritmo paralelo clásico para ordenamiento.

La ordenación bitónica hace comparaciones  $O(n \log 2n)$ .

El número de comparaciones realizadas por Bitonic Sort son más que los algoritmos de clasificación populares como Merge Sort [hace  $O(n \log n)$  comparaciones], pero Bitonic es mejor para la implementación en paralelo porque siempre comparamos elementos en una secuencia predefinida y la secuencia de comparación no. Depende de los datos. Por lo tanto, es adecuado para la implementación en hardware y array de procesado paralelo

Para entender el ordenamiento bitónico, primero debemos entender qué es la secuencia bitónica y cómo hacer una secuencia dada bitónica.

Una secuencia es bitónica si primero aumenta, luego disminuye. En otras palabras, una matriz arr [0...n-i] es Bitonic si existe un índice i donde  $0 \leq i \leq n-1$  tal que

$$x_0 \leq x_1 \dots \leq x_i \text{ y } x_i \geq x_{i+1} \dots \geq x_{n-1}$$

Para formar una secuencia ordenada de longitud n a partir de dos secuencias ordenadas de longitud  $n/2$ , se requieren comparaciones de  $\log(n)$  (por ejemplo:  $\log(8) = 3$  cuando el tamaño de la secuencia. Por lo tanto, el número de comparaciones  $T(n)$  de La clasificación completa está dada por:

$$T(n) = \log(n) + T(n/2)$$

La solución de esta ecuación de recurrencia es

$$T(n) = \log(n) + \log(n) - 1 + \log(n) - 2 + \dots + 1 = \log(n) \cdot (\log(n) + 1) / 2$$

Como, cada etapa de la red de clasificación consiste en  $n / 2$  comparaciones. Por lo tanto un total de  $O(n \log_2 n)$  comparaciones.

## Código

```
/* Programa java para Bitonic Sort*/
/* Nota: Este programa solo funciona si el tamaño
de la entrada es una potencia de 2*/
public class BitonicSort {
    /* El parámetro dir indica la dirección de ordenamiento,
       ASCENDIENDO o DESCENDIENDO; si (a [i]> a [j]) está de acuerdo
       con la dirección, entonces a[i] y a[j] son
    intercambiados*/
    static void compAndSwap(int a[], int i, int j, int dir) {
        if ((a[i] > a[j] && dir == 1)
            || (a[i] < a[j] && dir == 0)) {
            // intercambiando elementos
            int temp = a[i];
            a[i] = a[j];
            a[j] = temp;
        }
    }
    static void bitonicMerge(int a[], int low, int cnt, int dir) {
        if (cnt > 1) {
            int k = cnt / 2;
            for (int i = low; i < low + k; i++) {
                compAndSwap(a, i, i + k, dir);
            }
            bitonicMerge(a, low, k, dir);
            bitonicMerge(a, low + k, k, dir);
        }
    }
    static void bitonicSort(int a[], int low, int cnt, int dir) {
        if (cnt > 1) {
            int k = cnt / 2;
            // Ordena en orden ascendente, dir es 1
            bitonicSort(a, low, k, 1);
            // ordena en orden descendente, dir es 0
            bitonicSort(a, low + k, k, 0);
            // une toda la secuencia en orden ascendente
            bitonicMerge(a, low, cnt, dir);
        }
    }
    static void sort(int a[], int N, int up) {
        bitonicSort(a, 0, N, up);
    }

    /*Imprimir array */
    static void printArray(int arr[]) {
        int n = arr.length;
```

```

        for (int i = 0; i < n; ++i) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }

    public static void main(String args[]) {
        int a[] = {3, 7, 4, 8, 6, 2, 1, 5};
        int up = 1;
        sort(a, a.length, up);
        System.out.println("\nArray ordenado");
        printArray(a);
    }
}

```

## 15.10 Cocktail Sort

Cocktail Sort es una variación de Bubble Sort. El algoritmo de clasificación de burbuja siempre atraviesa elementos de la izquierda y mueve el elemento más grande a su posición correcta en la primera iteración y el segundo más grande en la segunda iteración, y así sucesivamente. La clasificación de cóctel atraviesa una matriz dada en ambas direcciones alternativamente.

Las complejidades de tiempo son iguales, pero Cocktail se desempeña mejor que Bubble Sort. Por lo general, el cóctel es menos de dos veces más rápido que el tipo burbuja. Considere el ejemplo (2, 3, 4, 5, 1). La clasificación de burbuja requiere cuatro recorridos de matriz para este ejemplo, mientras que la clasificación de cóctel requiere solo dos recorridos.

Tiempo del peor caso y caso promedio  $O(n * n)$ . Tiempo del mejor caso es  $O(n)$ . El mejor caso ocurre cuando el array ya está ordenado.

### Código

```

// Programa java implementando Cocktail sort
public class CocktailSort {

    static void cocktailSort(int a[]) {
        boolean swapped = true;
        int start = 0;
        int end = a.length;
        while (swapped == true) {

```

```

// Resetea la bandera intercambiada al entrar
    // en el ciclo, porque puede ser true de la
    // anterior iteración
    swapped = false;
// Ciclo del fondo a encima igual
// que bubble sort
    for (int i = start; i < end - 1; ++i) {
        if (a[i] > a[i + 1]) {
            int temp = a[i];
            a[i] = a[i + 1];
            a[i + 1] = temp;
        }
    swapped = true;
}
// Si nada fue movido, array esta ordenado
if (swapped == false) {
    break;
}
// de otra forma, resetea la bandera intercambiada
// de tal forma que pueda ser usada en el
//siguiente proceso
swapped = false;
// Mueve el punto final atras en uno, porque
// el item al final esta en la derecha completa
end = end - 1;
// from top to bottom, doing the
// same comparison as in the previous stage
for (int i = end - 1; i >= start; i--) {
    if (a[i] > a[i + 1]) {
        int temp = a[i];
        a[i] = a[i + 1];
        a[i + 1] = temp;
    }
}
// Incrementa el punto inicial, por que
// la ultima fase pudo haber movido el siguiente
// más pequeño número en la derecha completa
start = start + 1;
}
/*
 * Imprime el array*/
static void printArray(int a[]) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        System.out.print(a[i] + " ");
    }
    System.out.println();
}

public static void main(String[] args) {
;
int a[] = {5, 1, 4, 2, 8, 0, 2};
cocktailSort(a);

```

```

        System.out.println("Arreglo ordenado");
        printArray(a);
    }
}

```

## 15.11 Comb Sort

Comb Sort es principalmente una mejora sobre Bubble Sort. La ordenación de burbuja siempre compara valores adyacentes. Así que todas las inversiones se eliminan una por una. Comb Sort mejora en Bubble Sort usando un espacio de tamaño mayor que 1. El espacio comienza con un gran valor y se reduce en un factor de 1.3 en cada iteración hasta que alcanza el valor 1. Por lo tanto, Comb Sort elimina más de una cuenta de inversión con cada Intercambio y funciona mejor que Bubble Sort.

Aunque funciona mejor que Bubble Sort en promedio, el peor de los casos sigue siendo O ( $n^2$ ).

### Código

```

// Programa java implementando Comb Sort
public class CombSort
{
    // Para buscar espacio entre elementos
    static int getNextGap(int gap)
    {
        gap = (gap*10)/13;
        if (gap < 1)
            return 1;
        return gap;
    }
    static void sort(int arr[])
    {
        int n = arr.length;

        // Inicializa espacio
        int gap = n;
        boolean swapped = true;
        /*Mantiene ejecutando mientras gap es más que 1 y la
         ultima iteración causa un intercambio*/
        while (gap != 1 || swapped == true)
        {
            gap = getNextGap(gap);
            /*Inicializa swapped como falso, aso
             podemos verificar si el intercambio paso
             o no*/
            swapped = false;
            for (int i = 0; i < n - gap; i++)
            {
                if (arr[i] > arr[i + gap])
                {
                    swap(arr, i, i + gap);
                    swapped = true;
                }
            }
        }
    }
}

```

```

        //Compara todos los elementos con el espacio actual
for (int i=0; i<n-gap; i++)
{
    if (arr[i] > arr[i+gap])
    { //Intercambia arr[i] y arr[i+gap]
        int temp = arr[i];
        arr[i] = arr[i+gap];
        arr[i+gap] = temp;
        // Se intercambió
        swapped = true;
    }
}
}

public static void main(String args[])
{
    int arr[] = {8, 4, 1, 56, 3, -44, 23, -6, 28, 0};
    sort(arr);
    System.out.println("Array ordenado");
    for (int i=0; i<arr.length; ++i)
        System.out.print(arr[i] + " ");
}
}

```

## 15.12 Counting Sort

El ordenamiento por conteo es una técnica de ordenamiento basada en claves entre un rango específico. Funciona contando el número de objetos que tienen valores clave distintos (tipo de hashing). Luego, hacer algo de aritmética para calcular la posición de cada objeto en la secuencia de salida.

1. La ordenación de conteo es eficiente si el rango de datos de entrada no es significativamente mayor que el número de objetos a clasificar.
2. No es una comparación basada en la clasificación. La complejidad del tiempo de ejecución es  $O(n)$  con espacio proporcional al rango de datos.
3. A menudo se usa como una sub-rutina para otro algoritmo de clasificación como la clasificación por radix.

4. La ordenación de conteo usa un hashing parcial para contar la ocurrencia del objeto de datos en O (1).

5. La ordenación de conteo también se puede extender para que funcione con entradas negativas.

## Código

```
// Implementación java de Counting Sort
public class CountingSort
{
    static void sort(char arr[])
    {
        int n = arr.length;
        // El array de caracteres que sera el array
        //ordenado de salida
        char output[] = new char[n];
        // Crea un array de conteo para almacenar
        // conteo individual de caracteres e inicializar
        // array de conteo en 0
        int count[] = new int[256];
        for (int i=0; i<256; ++i)
            count[i] = 0;
        // Almacena el conteo de cada caracter
        for (int i=0; i<n; ++i)
            ++count[arr[i]];
        // Cambian count[i] de forma que ahora contenga la
        //posición actual de este caracter en el array de salida
        for (int i=1; i<=255; ++i)
            count[i] += count[i-1];
        // Construye el array de caracteres de salida
        // Para hacerlo estable lo hacemos en orden inverso
        for (int i = n-1; i>=0; i--)
        {
            output[count[arr[i]]-1] = arr[i];
            --count[arr[i]];
        }
        //Copia el array de salida a arr, asi arr ahora
        // contiene los caracteres ordenados
        for (int i = 0; i<n; ++i)
            arr[i] = output[i];
    }

    public static void main(String args[])
    {

        char arr[] = {'g', 'e', 'e', 'k', 's', 'f', 'o',
                      'r', 'g', 'e', 'e', 'k', 's'};
        sort(arr);
        System.out.print("array de caracteres ordenado: ");
    }
}
```

```

        for (int i=0; i<arr.length; ++i)
            System.out.print(arr[i]);
    }
}

```

## 15.13 Cycle Sort

El ordenamiento de ciclos es un algoritmo de ordenamiento en el lugar, una ordenación de comparación que es teóricamente óptima en términos del número total de escrituras en el array original.

Es óptimo en términos de cantidad de escrituras de memoria. Minimiza el número de escrituras de memoria para ordenar (cada valor se escribe cero veces, si ya está en su posición correcta, o se escribe una vez en su posición correcta).

### Código

```

// Java program to implement cycle sort

//Programa java implementando Cycle sort
public class CycleSort {
    public static void cycleSort(int arr[], int n)
    {   //Contador de escrituras en memoria
        int writes = 0;
    //Pasar por los elementos del array
        //y ponerlos en el lugar correcto
    for (int cycle_start = 0; cycle_start <= n - 2; cycle_start++) {
    // Inicializa item como punto inicial
        int item = arr[cycle_start];
    // Encuentra posicion donde nosotros pondremos el item
        // Basicamente contamos todos los elementos más pequeños
    //a la derecha del item
        int pos = cycle_start;
        for (int i = cycle_start + 1; i < n; i++)
            if (arr[i] < item)
                pos++;
    //Si item esta ya en la posicion correcta
    if (pos == cycle_start)
        continue;
    //Ignorar todos los elementos duplicados
        while (item == arr[pos])
            pos += 1;
    //Ponemos el item en su posicion correcta
    if (pos != cycle_start) {
        int temp = item;

```

```

        item = arr[pos];
        arr[pos] = temp;
        writes++;
    }
    //Rota el resto del ciclo
    while (pos != cycle_start) {
        pos = cycle_start;
        // Encuentra posición donde poner el elemento
    for (int i = cycle_start + 1; i < n; i++)
        if (arr[i] < item)
            pos += 1;
    //Ignora todos los duplicados
    while (item == arr[pos])
        pos += 1;

    // Ponemos el item en su posición correcta
    if (item != arr[pos]) {
        int temp = item;
        item = arr[pos];
        arr[pos] = temp;
        writes++;
    }
}
}

public static void main(String[] args)
{
    int arr[] = { 1, 8, 3, 9, 10, 10, 2, 4 };
    int n = arr.length;
    cycleSort(arr, n);
    System.out.println("Luego de ordenar : ");
    for (int i = 0; i < n; i++)
        System.out.print(arr[i] + " ");
}
}

```

## 15.14 3 Way Merge Sort

Merge sort envuelve separación recursiva del array en dos partes, ordenamiento y finalmente unión de las mismas. Una variante de merge sort es llamado merge sort de tres vías donde en vez de separar el array en dos partes se separa en tres. Merge sort recursivamente rompe los arrays en arrays de tamaño medio, Merge de tres vías hace lo mismo en arrays de tamaño de un tercio.

**Complejidad de tiempo:** En caso de Merge Sort se tiene la ecuación  $T(n) = 2T(n/2) + O(n)$ . Igualmente en caso de Merge de tres vías se tiene la ecuación  $(n) = 3T(n/3) + O(n)$  por lo que la aproximación más cercana de su complejidad es:  $O(n \log_3 n)$ .

```

//Programa java que realiza Merge Sort de tres vias
public class MergeSort3Way
{
    public static void mergeSort3Way(Integer[] gArray)
    {   //Si el array tiene tamaño 0, retorna null
        if (gArray == null)
            return;
        //Creando duplicado del array dado
        Integer[] fArray = new Integer[gArray.length];
        //Copiando elementos del array dado al
        //array duplicado
        System.arraycopy(gArray, 0, fArray, 0, fArray.length);
        mergeSort3WayRec(fArray, 0, gArray.length, gArray);
        //Copiando elementos del array duplicado al
        //array dado
        System.arraycopy(fArray, 0, gArray, 0, fArray.length);
    }
    /*Realizando el algoritmo Merge Sort en el array
     dado para los valores en el rango de los indices
     [low,high], low es el menor y high es el mayor
     Exclusivo*/
    public static void mergeSort3WayRec(Integer[] gArray,
                                         int low, int high, Integer[] destArray)
    {
        //Si el array tiene tamaño 1 no haga nada
        if (high - low < 2)
            return;
        // Separando el array en tres partes
        int mid1 = low + ((high - low) / 3);
        int mid2 = low + 2 * ((high - low) / 3) + 1;
        // Ordenando las tres partes recursivamente
        mergeSort3WayRec(destArray, low, mid1, gArray);
        mergeSort3WayRec(destArray, mid1, mid2, gArray);
        mergeSort3WayRec(destArray, mid2, high, gArray);
        // Uniendo las tres partes
        merge(destArray, low, mid1, mid2, high, gArray);
    }
    /*Une los rangos ordenados (low, mid1),(mid1,mid2)
     y (mid2, high), mid 1 es el primer indice punto
     medio en todo el rango para unir mid2 es segundo*/
    public static void merge(Integer[] gArray, int low,
                           int mid1, int mid2, int high,
                           Integer[] destArray)
    {
        int i = low, j = mid1, k = mid2, l = low;
        // escoje el más pequeño de los más pequeños en los
        //tres rangos
        while ((i < mid1) && (j < mid2) && (k < high))
        {
    }

```

```

        if (gArray[i].compareTo(gArray[j]) < 0)
        {
            if (gArray[i].compareTo(gArray[k]) < 0)
                destArray[l++] = gArray[i++];

            else
                destArray[l++] = gArray[k++];
        }
        else
        {
            if (gArray[j].compareTo(gArray[k]) < 0)
                destArray[l++] = gArray[j++];
            else
                destArray[l++] = gArray[k++];
        }
    }
    while ((i < mid1) && (j < mid2))
    {
        if (gArray[i].compareTo(gArray[j]) < 0)
            destArray[l++] = gArray[i++];
        else
            destArray[l++] = gArray[j++];
    }
    while ((j < mid2) && (k < high))
    {
        if (gArray[j].compareTo(gArray[k]) < 0)
            destArray[l++] = gArray[j++];
        else
            destArray[l++] = gArray[k++];
    }
    while ((i < mid1) && (k < high))
    {
        if (gArray[i].compareTo(gArray[k]) < 0)
            destArray[l++] = gArray[i++];
        else
            destArray[l++] = gArray[k++];
    }
    while (i < mid1)
        destArray[l++] = gArray[i++];
    while (j < mid2)
        destArray[l++] = gArray[j++];
    while (k < high)
        destArray[l++] = gArray[k++];
}

public static void main(String args[])
{
    Integer[] data = new Integer[] {45, -2, -45, 78,
                                    30, -42, 10, 19, 73, 93};
    mergeSort3Way(data);
    System.out.println("Despues de merge sort de 3 vias: ");
    for (int i = 0; i < data.length; i++)
}

```

```

        System.out.print(data[i] + " ");
    }
}

```

## 15.15 Pigeon Hole Sort

Ordenamiento nido de Paloma es un algoritmo de ordenamiento que es adecuado para ordenar listas de elementos donde el número de elementos y el número de valores de los mismos son aproximadamente lo mismo.

Este requiere un tiempo de  $O(n + \text{Rango})$  donde  $n$  es el número de elementos en el array de entrada y Rango es el número de posibles valores en el array. Este algoritmo es similar a Counting sort, pero difiere en la forma en que mueve los datos, dos veces.

```

/*Programa java que implemeta Pigeonhole Sort*/
import java.util.*;

public class PigeonholeSort {

    static void pigeonhole_sort(int arr[],
                                int n) {
        int min = arr[0];
        int max = arr[0];
        int range, i, j, index;
        for (int a = 0; a < n; a++) {
            if (arr[a] > max) {
                max = arr[a];
            }
            if (arr[a] < min) {
                min = arr[a];
            }
        }
        range = max - min + 1;
        int[] phole = new int[range];
        Arrays.fill(phole, 0);
        for (i = 0; i < n; i++) {
            phole[arr[i] - min]++;
        }
        index = 0;
        for (j = 0; j < range; j++) {
            while (phole[j]-- > 0) {
                arr[index++] = j + min;
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {8, 3, 2, 7, 4, 6, 8};

```

```

        System.out.print("Array ordenado : ");
        pigeonhole_sort(arr, arr.length);
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
    }
}

```

## 15.16 Problemas de repaso

### Ejercicios en Online Judge

450-Little Black Book

10041- Vito's Family

741-Burrows Wheeler Decoder

10107-What is the Median?

885-Telephone Directory Alphabetization

10152-CDVII

895- Word Problem

### Ejercicios en Online Judge

FLOW007

PBR01

FLOW017

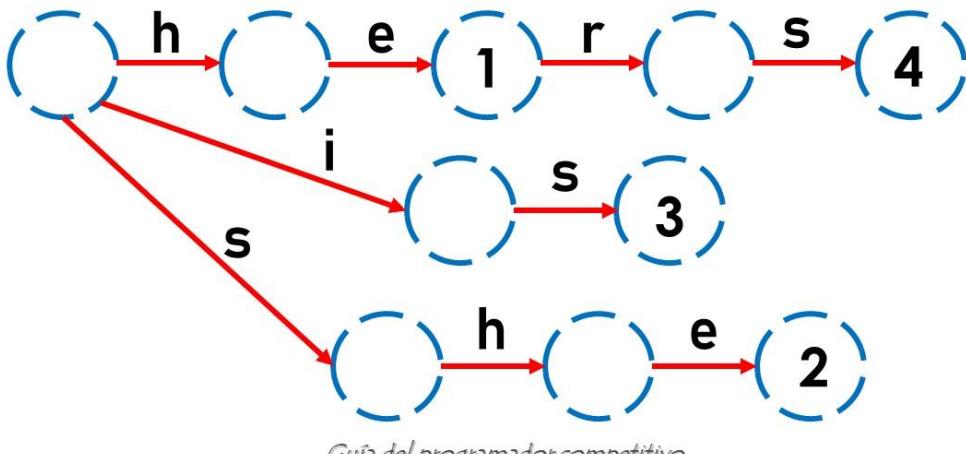
FLOW016

# 16 Patrones y manejo de Strings

## 16.1 Aho-Corasick Algorithm

### Aho corasick

Intentar con  $All[] = \{he, she, his, hers\}$



Guía del programador competitivo.

Ilustración 16-1 Ejemplo de búsqueda de patrones Aho-Corasick

Dada una entrada de texto y un array de k palabras, arr[], encontrar todas las ocurrencias de todas las palabras en el texto de entrada. Sea n la longitud del texto y m sea el número total de caracteres en todas las palabras, por ejemplo:  $m = \text{length}(\text{arr}[0]) + \text{length}(\text{arr}[1]) + \dots + \text{length}(\text{arr}[k-1])$ . Si usamos un algoritmo de tiempo lineal como KMP, entonces necesitaremos una búsqueda uno por uno de todas las palabras en text[]. Esto nos da un total de complejidad de tiempo de  $O(n + \text{length}(\text{word}[0]) + O(n + \text{length}(\text{word}[1]) + O(n + \text{length}(\text{word}[2]) + \dots + O(n + \text{length}(\text{word}[k-1]))$ . Esta complejidad de tiempo puede ser escrita como  $O(n*k + m)$ .

Aho-corasick encuentra todas las palabras en tiempo  $O(n+m+z)$  donde z es el número total de ocurrencias de las palabras en el texto, este algoritmo forma las bases del comando original de Unix fgrep.

Este algoritmo tiene dos pasos principales:

Preprocesado: Construye un automaton de todas las palabras en arr[].

Cotejamiento: Atraviesa el texto dado sobre el automaton formado para encontrar las palabras a cotejar. Por un estado s, los índices de todas las palabras terminados en s son almacenadas. Estos índices son almacenados en un map de BitWise (Haciendo O R a lo valores. Esto también computa usando búsqueda primera en anchura con fallo.

```
//Codigo JAVA que implementa el algoritmo
// Aho Corasick de busqueda de palabras
import java.util.LinkedList;
import java.util.Queue;

public class AhoCorasick {

    static class TrieNode {
        private final int SIZE = 4;
        TrieNode parent;
        TrieNode fall;
        TrieNode[] children;
        char c;
        TrieNode next;
        boolean isEnd;
        public TrieNode() {
            children = new TrieNode[SIZE];
        }

        public TrieNode(char c) {
            this();
            this.c = c;
        }
        public TrieNode child(char ch) {
            TrieNode node = children[ch % SIZE];
            if (node == null) {
                return null;
            } else {
                while (node.next != null) {
                    if (node.c == ch) {
                        return node;
                    }
                    node = node.next;
                }
                // Caracter estaba al final de la lista
                if (node.c == ch) {
                    return node;
                }
                return null;
            }
        }
    }
}
```

```

public void addWord(String word) {
    char firstChar = word.charAt(0);
    TrieNode child = child(firstChar);
    if (child == null) {
        child = new TrieNode(firstChar);
        child.parent = this;
        int pos = firstChar % SIZE;
        if (children[pos] == null) {
            children[pos] = child;
        } else {
            TrieNode node = children[pos];
            while (node.next != null) {
                node = node.next;
            }
            node.next = child;
        }
    }
    if (word.length() > 1) {
        child.addWord(word.substring(1));
    } else {
        child.isEnd = true;
    }
}

public boolean contains(String word) {
    if (word.length() == 0) {
        return isEnd;
    } else {
        TrieNode node = child(word.charAt(0));
        if (node == null) {
            return false;
        } else {
            return node.contains(word.substring(1));
        }
    }
}

public void constructFallLinks() {
    TrieNode root = this;
    root.fall = root;

    // BFS
    Queue<TrieNode> queue = new LinkedList<>();
    queue.add(root);
    while (!queue.isEmpty()) {
        TrieNode currentNode = queue.poll();

        // Agrega hijo
        for (int i = 0; i < SIZE; i++) {
            TrieNode node = currentNode.children[i];
            if (node != null) {
                while (node.next != null) {

```

```

                queue.add(node);
                node = node.next;
            }
            queue.add(node);
        }
    }

    if (currentNode != root) {
        // Atraviesa arriba la caida del padre para encontrar
        // el sufijo más largo
        //o hasta que alcanzamos la raiz del arbol
        TrieNode f = currentNode.parent.fall;
        while (f.child(currentNode.c) != null && f != root) {
            f = f.fall;
        }
        currentNode.fall = f.child(currentNode.c);

        if (currentNode.fall == null) {
            // Sufijo no encontrado, caemos la raiz
            currentNode.fall = root;
        }

        if (currentNode.fall == currentNode) {
            currentNode.fall = root;
        }
    }
}

/*Realiza una busqueda en el texto de las palabras
 contenidas en Trie*/
public void search(String text) {
    TrieNode root = this;
    TrieNode currentNode = root;
    TrieNode node;
    TrieNode n;
    char[] chars = text.toCharArray();
    for (char ch : chars) {
        node = currentNode;
        while (node.child(ch) == null && node != root) {
            node = node.fall;
        }
        if (node == root) {
            node = node.child(ch);
            if (node == null) {
                node = root;
            }
        } else {
            node = node.child(ch);
        }

        n = node;
        while (n != root) {
            if (n.isEnd) {

```

```

        System.out.println("Encontrada palabra que termina en "
+ n.c);
}
n = n.fall;
}
currentNode = node;
}
}
}

public static void main(String[] args) {
TrieNode trie = new TrieNode();
trie.addWord("soft");
trie.addWord("fo");
trie.constructFallLinks();
trie.search("abcfosoft");
}

}
}

```

## 16.2 Anagrams Pattern

Dado un texto txt [0...n-1] y un patrón pat [0...m-1], use una función que imprima todas las ocurrencias de pat [] y sus permutaciones (o anagramas) en txt [], se asume que n es menor que m.

La complejidad de tiempo esperada es de O(n).

Este problema es ligeramente diferente a la búsqueda de patrones estándar, aquí necesitamos buscar por anagramas también. Por lo tanto no podemos aplicar directamente la búsqueda de patrones estándar de algoritmos como KMP, Rabin Karp o Boyer Moore.

Podemos conseguir una complejidad de tiempo de O(n) asumiendo que el tamaño del alfabeto está arreglado en los 256 caracteres ASCII. La idea es usar dos arrays de conteo:

- 1) El primer array de conteo almacena la frecuencia de los caracteres en el patrón.
- 2) El Segundo array de conteo almacena la frecuencia en la actual ventana de texto.

Una cosa importante a tener en cuenta es, la complejidad de tiempo de comparar dos arrays de conteo es O(1) como el número de elementos en ellos. Estos son los pasos de este algoritmo:

- 1) Almacena conteos de frecuencia del patrón en el primer array de conteo countP [].
 

También almacena conteo de frecuencias de la primera ventana de texto en el array countTW[].
- 2) Ahora ejecuta un ciclo de i=M hasta N-1 haciendo lo siguiente en cada ciclo:
  - a) Si los dos arrays de conteo son idénticos, hemos encontrado una ocurrencia.
  - b) Incrementa el conteo del actual carácter del texto en countTW[]
  - c) Decrementa conteo del primer carácter en la ventana anterior en countWT[]
- 3) La última ventana no es revisada por este ciclo, se revisa explícitamente.

```

// Programa java que busca todos los anagramás
// de un patrón en un texto
public class AnagramsPattern
{
    static final int MAX = 256;
    // Esta función retorna true si los contenidos
    // de arr1[] y arr2[] son iguales, de otra forma es falso
    static boolean compare(char arr1[], char arr2[])
    {
        for (int i = 0; i < MAX; i++)
            if (arr1[i] != arr2[i])
                return false;
        return true;
    }
    // Esta función busca todas las permutaciones de
    // pat[] en txt[]
    static void search(String pat, String txt)
    {
        int M = pat.length();
        int N = txt.length();

        // countP[]: Almacena el conteo de todos
        // los caracteres del patrón
        // countTW[]: Almacena el conteo de los caracteres
        // la ventana de texto
        char[] countP = new char[MAX];
        char[] countTW = new char[MAX];
        for (int i = 0; i < M; i++)
        {
            (countP[pat.charAt(i)])++;
            (countTW[txt.charAt(i)])++;
        }
        //Atravesar atravez de los caracteres restantes de patrón
        for (int i = M; i < N; i++)
        {
            //Compara conteos de la ventana actual
            // de texto con los conteos de pattern[]
            if (compare(countP, countTW))
                System.out.println("Encontrado en indice " +
(i - M));
        }
    }
}

```

```

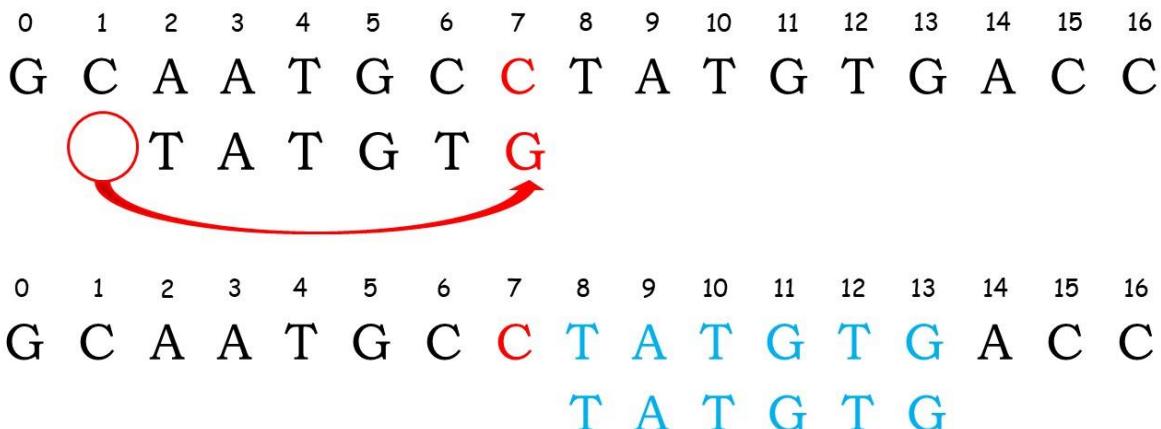
// Agrega el actual caracter a la ventana actual
    (countTW[txt.charAt(i)])++;
// Remueve el primer caracter de la anterior ventana
countTW[txt.charAt(i-M)]--;
}
// Revisa por la ultima ventana en el texto
if (compare(countP, countTW))
    System.out.println("Encontrado en indice " +
                        (N - M));
}

public static void main(String args[])
{
    String txt = "BACDGABCDA";
    String pat = "ABCD";
search(pat, txt);
}

```

## 16.3 Boyer-Moore Algorithm

### Boyer Moore



*Guía del programador competitivo.*

Ilustración 16-2 Ejemplo de búsqueda de patrón Boyer-Moore

Cuando realizamos una búsqueda de un string en Notepad, Word, buscador o una base de datos, los algoritmos de búsquedas de patrones son usados para buscar los resultados. Un enunciado de este problema podría ser:

Dado un texto  $txt[0...n-1]$  y un patrón  $pat[0...m-1]$  busque una función que imprima todas las ocurrencias de  $pat$  en  $txt$ , se asume que  $n$  es mayor que  $m$ .

Como KMP y Finite Automata, Boyer Moore también preprocesa el patrón, Boyer Moore es la combinación de las siguientes dos aproximaciones:

- 1) Heurística de caracteres malos
- 2) Heurística de buenos sufijos

Ambas de las dos heurísticas de arriba pueden también ser usadas independientemente para buscar un patrón en un texto. Primero entendamos como estas dos aproximaciones trabajan juntos en Boyer Moore. Si tomamos un vistazo a cualquier algoritmo ingenuo, desliza el patrón sobre el texto carácter por carácter. KMP hace preprocesado sobre el patrón de tal forma que el patrón puede ser cambiado más de una vez.

El algoritmo de Boyer Moore hace preprocesado por la misma razón. Este procesa el patrón y crea arrays diferentes para cada heurística. En cada paso desliza el patrón por el máximo de deslices sugeridos por las dos heurísticas, entonces usa la mejor de las dos heurísticas en cada paso.

A diferencia de los anteriores algoritmos de búsqueda de patrones, Boyer Moore comienza el cotejamiento desde el último carácter del patrón.

### **Heurística de caracteres malos**

La idea de esta heurística es simple. El carácter del texto que no coincide con el carácter actual del patrón es llamado un carácter malo. Con esta no coincidencia nosotros movemos el patrón hasta:

- 1) La no coincidencia se convierte en un cotejamiento positivo.
- 2) Patrón P se mueve después del carácter no coincidente.

#### **Caso 1:** No coincidencia se convierte en una coincidencia

Buscamos la posición de la última ocurrencia del carácter no coincidente en el patrón y si el carácter no coincidente existe en el patrón, entonces movemos el patrón de tal manera que quede alineado con el carácter no coincidente en el texto T.

#### **Caso 2:** El patrón se mueve pasado el carácter no coincidente

Nosotros buscamos la posición de la última ocurrencia del carácter no coincidente en el patrón y si el carácter no existe deberíamos mover el patrón pasado ese carácter.

```
/*Programa java para heuristicas de malos caracteres
usando el algoritmo de Boyer Moore*/

public class BoyerMoore{

    static int NO_OF_CHARS = 256;
    //Una función de utilidad para obtener el maximo de
    //dos enteros

    static int max (int a, int b) { return (a > b)? a: b; }
    //El preprocesado del algoritmo
    static void badCharHeuristic( char []str, int size,int badchar[])
    {
        int i;

        // Inicializa todas las ocurrencias en -1
        for (i = 0; i < NO_OF_CHARS; i++)
        badchar[i] = -1;
        /*Llena el actual valor de la ultima ocurrencia
        de un caracter*/
        for (i = 0; i < size; i++)
            badchar[(int) str[i]] = i;
    }
    /* Una función de búsqueda de patrón que usa
       la heuristicas de mal caracter*/
    static void search( char txt[],  char pat[])
    {
        int m = pat.length;
        int n = txt.length;

        int badchar[] = new int[NO_OF_CHARS];
        /* Llena el arreglo de malos caracteres llamando
           la función de preprocesado para el patrón dado*/
        badCharHeuristic(pat, m, badchar);
        int s = 0; // s es cambiado del patron con respecto al texto
        while(s <= (n - m))
        {
            int j = m-1;
            /* Mantiene reduciendo el indice j para el patrón
               mientras los caracteres del patrón y el texto
               estan coincidiendo en s*/
            while(j >= 0 && pat[j] == txt[s+j])
                j--;
            /*Si el patrón esta presente en el actual
               cambio, entonces el indice j se convertira en -1
               luego del ciclo de arriba*/
            if (j < 0)
            {
                System.out.println("Patrón encontrado en cambio = " + s);
                s += (s+m < n)? m-badchar[txt[s+m]] : 1;
            }
        }
    }
}
```

```

        }
    else
        s += max(1, j - badchar[txt[s+j]]);
    }
}

public static void main(String []args) {
    char txt[] = "ABAAABCDABCABC".toCharArray();
    char pat[] = "ABC".toCharArray();
    search(txt, pat);
}
}

```

## 16.4 Finite Automata

Dado un texto  $txt[0...n-1]$  y un patrón  $pat[0...m-1]$ , busque todas las ocurrencias de  $pat$  en  $txt$  e imprímolas, asuma que  $n$  es mayor que  $m$ .

En el algoritmo basado en Finite autómata, nosotros preprocesamos el patrón y construimos un array de dos dimensiones que represente un autómata finito. Construcción del FA es la parte complicada de este algoritmo. Una vez el FA este construido, la búsqueda es simple, En la búsqueda nosotros simplemente necesitamos iniciar desde el primer estado del autómata y el primer carácter del texto. En cada paso, consideramos el siguiente carácter del texto, miramos por el siguiente estado en el FA construido y nos movemos a un nuevo estado. Si alcanzamos el estado final, entonces el patrón fue encontrado en el texto.

La complejidad de tiempo en el proceso de búsqueda es de  $O(n)$ .

```

// Programa java para el algoritmo
// Finite Automata de búsqueda de patrones
public class FiniteAutomata {

    static int NO_OF_CHARS = 256;

    static int getNextState(char[] pat, int M,
    int state, int x) {
        // Si el carácter c es el mismo como el
        // siguiente en patrón, simplemente incrementa
        // state
        if (state < M && x == pat[state]) {

```

```

        return state + 1;
    }
    // ns almacena el resultado el cual es
    // el siguiente state
    int ns, i;
/* ns finalmente contiene el prefijo más largo
   el cual tambien es sufijo en "pat[0..state-1]c"
   Empieza desde el valor más largo posible
   y se detiene cuando se encuentra un prefijo el cual
   es tambien sufijo*/
    for (ns = state; ns > 0; ns--) {
if (pat[ns - 1] == x) {
    for (i = 0; i < ns - 1; i++) {
        if (pat[i] != pat[state - ns + 1 + i]) {
            break;
        }
    }
    if (i == ns - 1) {
        return ns;
    }
}
}
return 0;
}
/*Esta función construye la tabla TF la cual
   representa Finite Automata del patrón dado*/
static void computeTF(char[] pat, int M, int TF[][][]) {
    int state, x;
    for (state = 0; state <= M; ++state) {
        for (x = 0; x < NO_OF_CHARS; ++x) {
            TF[state][x] = getNextState(pat, M, state, x);
        }
    }
}
/* Imprime todas las ocurrencias de pat en txt*/
static void search(char[] pat, char[] txt) {
    int M = pat.length;
    int N = txt.length;
    int[][] TF = new int[M + 1][NO_OF_CHARS];
    computeTF(pat, M, TF);
// Procesa txt sobre FA.
    int i, state = 0;
    for (i = 0; i < N; i++) {
        state = TF[state][txt[i]];
        if (state == M) {
            System.out.println("Patrón encontrado "
                + "en indice " + (i - M + 1));
        }
    }
}

public static void main(String[] args) {
char[] pat = "AABAACAAADAAABAAABAA".toCharArray();
char[] txt = "AABA".toCharArray();

```

```

        search(txt, pat);
    }
}

```

## 16.5 Knuth-Morris-Pratt Matching

### KMP

Texto : A A B A A C A A D A A B A A B A

Modelo : A A B A

A A B A	A A B A
A A B A A C A A D A A B A A B A	
0    1    2    3    4    5    6    7    8    9    10    11    12    13    14    15	

- Patrón Encontrado desde las posiciones 0, 9 y 12.

A A B A

*Guía del programador competitivo.*

*Ilustración 16-3 Búsqueda de patrón KMP*

Dado un texto txt[0...n-1] y un patrón pat[0...m-1] busque todas las ocurrencias de pat en txt e imprímolas, asuma que n es mayor que m.

El algoritmo KMP usa la propiedad de degeneración (Patrón teniendo el mismo sub patrón apareciendo más de una vez en el patrón) del patrón y mejora el peor caso de complejidad de O(n). La idea básica detrás de KMP es: Cuando no detectemos una no coincidencia (luego de algunas coincidencias), ya sabemos algunos de los caracteres en el texto de la siguiente ventana. Tomamos ventaja de esta información para evadir coincidencias de caracteres que sabemos que de todas maneras coincidirán.

**Revisión del preprocessado:**

KMP preprocesa pat[] y construye un lps[] auxiliar de tamaño m (El mismo tamaño que el patrón) el cual es usado para saltar caracteres mientras se coteja.

El nombre lps indica el prefijo propio más largo el cual es también sufijo. Un prefijo propio es un prefijo con todo el string no permitido. Por ejemplo, prefijos de “ABC” son “”, “A”, “AB” Y “ABC”, prefijos propios son “”, “A” y “AB”. Sufijos del string son “”, “C”, “BC” y “ABC”.

Buscamos por lps en subpatrones. Más claramente nos enfocamos en los subsring de patrones que son prefijos y sufijos. Por cada subpatron pat[0...i] donde i=0 hasta m-1, lps[i] almacena la longitud del máximo prefijo propio coincidente el cual también es sufijo del subpatron pat[0...i].

#### **Algoritmo de búsqueda:**

A diferencia de los algoritmos ingenuos, donde deslizamos el patrón uno por uno y comparamos todos los caracteres en cada movimientos, usamos un valor de lps[] para decidir los nuevos caracteres que serán cotejados, la idea es no cotejar un carácter que sabemos que va a aparecer.

¿Cómo usamos lps [] para decidir las siguientes posiciones o número de caracteres que serán saltadas?

Comenzamos la comparación de pat [j] con j=0 con los caracteres de la actual ventana de texto. Seguimos cotejando caracteres de txt[i] y pat [j] y seguimos incrementando i y j mientras pat[j] y txt [i] sigan coincidiendo.

Cuando vemos una no coincidencia, sabemos que los caracteres pat[0...j-1] coinciden con txt[i-j.....i-1] (Nótese que j inicia en 0 e incrementa solo cuando hay una coincidencia). También sabemos que lps[j-1] está contando los caracteres de pat[0...j-1] que son prefijo propio y sufijo.

De estos puntos podemos concluir que nosotros no necesitaremos cotejar estos lps[j-1] caracteres con txt porque sabemos que de todas formas esos caracteres coincidirán.

```
/* Programa java implementando el algoritmo
de busqueda KMP*/
public class KMPStringMatching {

    static void KMPSearch(String pat, String txt) {
        int M = pat.length();
        int N = txt.length();
        // Crea lps[] el cual podra mantener el
        // más largo prefijo sufijo para patrón
        int lps[] = new int[M];
        int j = 0; // indice de pat[]
```

```

// Preprocesa el patrón (Calcula lps[])
computeLPSArray(pat, M, lps);
    int i = 0; // indice para txt[]
    while (i < N) {
        if (pat.charAt(j) == txt.charAt(i)) {
            j++;
            i++;
        }
        if (j == M) {
            System.out.println("Encontrado patrón "
+ "en indice " + (i - j));
            j = lps[j - 1];
        } // no coincide luego de j veces
        else if (i < N && pat.charAt(j) != txt.charAt(i)) {
/*No coincide lps[0..lps[j-1]] caracteres
   ellos coincidiran de todos modos*/
            if (j != 0) {
                j = lps[j - 1];
            } else {
                i = i + 1;
            }
        }
    }
}

static void computeLPSArray(String pat, int M, int lps[]) {
// Longitud del anterior prefijo sufijo más largo
    int len = 0;
    int i = 1;
    lps[0] = 0; // lps[0] es siempre 0

// el ciclo calcula lps[i] para i = 1 hasta M-1
    while (i < M) {
        if (pat.charAt(i) == pat.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else // (pat[i] != pat[len])
{
// Esto es difícil, considere el ejemplo
// AAACAAAA y i = 7. la idea es similar
// el paso de búsqueda
            if (len != 0) {
                len = lps[len - 1];
}
// También, note que no incrementamos
// i aquí
            } else // if (len == 0)
{
                lps[i] = len;
                i++;
}
    }
}
}

```

```

    public static void main(String args[]) {
        String txt = "ABABDABACDABABCABAB";
        String pat = "ABABCABAB";
        KMPSearch(pat, txt);
    }
}

```

## 16.6 Manacher Algorithm

Dado un string, encuentre el substring más largo que sea palíndromo.

Si el string dado es “abaaba” la salida deberá ser “abaaba”

Vamos a considerar el string “abababa”.

Aquí el centro del string es el 4to carácter con índice 3, si cotejamos más caracteres en la izquierda y derecha del centro todos los caracteres coinciden y el string es palíndromo.

Considere el string “abaaba” de tamaño par. Este string es palíndromo alrededor de la posición entre el 3er y 4to carácter.

Para encontrar el substring más largo palíndromo de un string de tamaño N, una vía es tomar cada posible  $2N+1$  centros (las N posiciones de caracteres, N-1 entre dos caracteres y dos posiciones en los fines de derecha e izquierda), haga que el carácter coincida en ambas direcciones en cada  $2N+1$  centros y siga rastreando por LPS.

Si el string dado es “abababa” la salida debe ser “abababa”

Si el string dado es “abcbabcbabcba” la salida debe ser “abcbabcba”.

```

// Programa java implementando el algoritmo de Manacher
// LPS: longest palindrome string
public class ManacherAlgorithm {

    static char text[];

    public static void main(String[] args) {
        text = "babcbabcbaccba".toCharArray();
        findLongestPalindromicString();
        text = "abaaba".toCharArray();
        findLongestPalindromicString();
        text = "abababa".toCharArray();
        findLongestPalindromicString();
        text = "abcbabcbabcba".toCharArray();
        findLongestPalindromicString();
        text = "caba".toCharArray();
        findLongestPalindromicString();
        text = "abacdfgdcaba".toCharArray();
    }
}

```

```

        findLongestPalindromicString();
        text = "abacdfgdcabba".toCharArray();
        findLongestPalindromicString();
        text = "abacdedcaba".toCharArray();
        findLongestPalindromicString();
    }

    static int min(int a, int b) {
        int res = a;
        if (b < a) {
            res = b;
        }
        return res;
    }

    static void findLongestPalindromicString() {
        int N = text.length;
        if (N == 0) {
            return;
        }
        N = 2 * N + 1; //Conteo de posición
        int L[] = new int[N]; //LPS tamaño de array
        L[0] = 0;
        L[1] = 1;
        int C = 1; //Posición central
        int R = 2; //posición Centro derecho
        int i = 0; //Posición actual derecho
        int iMirror; //Posición actual izquierda
        int maxLPSLength = 0;
        int maxLPSCenterPosition = 0;
        int start = -1;
        int end = -1;
        int diff = -1;
        //Descomentar para imprimir tamaño del arreglo LPS
        //printf("%d %d ", L[0], L[1]);
        for (i = 2; i < N; i++) {
            iMirror = 2 * C - i;
            L[i] = 0;
            diff = R - i;
            if (diff > 0) {
                L[i] = min(L[iMirror], diff);
            }
        }
        /*Intente expandir palíndromo centrado en currentRightPosition i
         Aquí para posiciones impares, comparamos caracteres y
         si coinciden, aumente la longitud de LPS en UNO
         Si la posición es igual, solo incrementamos LPS en UNO sin*/
        try {
            while (((i + L[i]) < N && (i - L[i]) > 0)
&& (((i + L[i] + 1) % 2 == 0)
|| (text[(i + L[i] + 1) / 2] == text[(i - L[i] - 1) / 2]))) {
                L[i]++;
            }
        } catch (Exception e) {
    }

```

```

//Comparación de cualquier carácter
        if (L[i] > maxLPSLength)
{
            maxLPSLength = L[i];
            maxLPSCenterPosition = i;
}
if (i + L[i] > R) {
    C = i;
    R = i + L[i];
}
//Descomentar para imprimir tamaño del arreglo LPS
//printf("%d ", L[i]);
}
start = (maxLPSCenterPosition - maxLPSLength) / 2;
end = start + maxLPSLength - 1;
System.out.println("LPS del string es " + String.valueOf(text) + ":");

for (i = start; i <= end; i++) {
    System.out.printf("%c", text[i]);
}

System.out.println("");
}
}

```

## 16.7 Rabin-Karp Algorithm

### Rabin Karp

Texto : A A B A A C A A D A A B A A B A

Modelo : A A B A

A A B A	A A B A
A A B A A C A A D A A B A A B A	
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15	

- Patrón Encontrado desde las posiciones 0, 9 y 12.

A A B A

*Guía del programador competitivo.*

*Ilustración 16-4 Búsqueda de patrón por Rabin-Karp*

Dado un texto  $txt[0...n-1]$  y un patrón  $pat[0...m-1]$  busque todas las ocurrencias de  $pat$  en  $txt$  e imprímolas, asuma que  $n$  es mayor que  $m$ .

Como los algoritmos ingenuos, Rabin-Karp también desliza el patrón uno por uno, pero a diferencia de este, RK coteja el valor hash del patrón con el valor hash del actual substring de texto y si los valores hash coinciden entonces solo empieza a cotejar caracteres individuales, entonces RK necesita calcular los valores hash de los siguientes strings.

- 1) El string patrón
- 2) Todos los substrings del texto de tamaño  $m$ .

Desde que necesitamos calcular eficientemente los valores hash de todos los substrings de tamaño  $m$  del texto, debemos tener una función hash la cual tenga la siguiente propiedad. El hash del siguiente cambio debe ser eficientemente calculable desde el hash actual y el siguiente carácter en el texto o podemos decir  $\text{hash}(txt[s+1..... s+m]) = \text{rehash}(\text{txt}[s+m], \text{hash}(\text{txt}[s....s+m-1]))$  y rehash debe ser una operación  $O(1)$ .

La función hash sugerida por Rabin y Karp calcula un valor entero, el valor entero para un string es un valor número de un string, por ejemplo el número si todos los posibles caracteres son de 1 a 10, el valor número de “122” sería 122. El número de posibles caracteres es mayor que 10 (256 en general) y la longitud el patrón puede ser larga. Entonces los valores numéricos no pueden ser prácticamente almacenados como un entero. Sin embargo el valor número es calculado usando matemática modular para asegurar que los valores hash pueden ser almacenados en una variable entera (puede caber en palabras de memoria). Para hacer rehashing, necesitamos tomar el más significante dígito y añadirlo nuestro nuevo dígito significante para el valor hash, el rehashing se realiza con la siguiente fórmula

- $\text{hash}(txt[s+1 .. s+m]) = (d(\text{hash}(txt[s .. s+m-1]) - \text{txt}[s]*h) + \text{txt}[s + m]) \bmod q$
- $\text{hash}(txt[s .. s+m-1])$ : Valor hash en cambio  $s$ .
- $\text{hash}(txt[s+1 .. s+m])$ : Valor hash en nuevo cambio ( $s+1$ )
- $d$ : Número de caracteres en el alfabeto
- $q$ : Un número primo

- $h: d^{(m-1)}$

Esto es matemática simple, calculamos el valor decimal de la actual ventana desde la ventana anterior.

Por ejemplo el tamaño del patrón es 3 y el string es "23456"

Se calcula el valor de la primera ventana el cual es 234 (String ventana es "234").

¿Cómo puedes calcular el valor de la siguiente ventana "345"? se puede hacer  $(234 - 2*100)*10 + 5$  y obtener 345.

El promedio y mejor tiempo de ejecución en un caso de RK es de  $O(n+m)$  pero su peor caso es  $O(nm)$ . El peor caso de Rabin-Karp ocurre cuando todos los caracteres del patrón y el texto tienen los mismos valores hash de todos los substrings de txt, por ejemplo `pat[]="AAA" y txt[]="AAAAAAA".`

```

/*Implementación java del algoritmo de
Rabin Karp*/
public class RabinKarp
{
    // d es el número de caracteres en el alfabeto de entrada
    public final static int d = 256;
    /* pat -> patrón
       txt -> texto
       q -> Un número primo
    */
    static void search(String pat, String txt, int q)
    {
        int M = pat.length();
        int N = txt.length();
        int i, j;
        int p = 0; // valor hash del patrón
        int t = 0; // valor hash del txt
        int h = 1;
        // el valor de h debe ser "pow(d, M-1)%q"
        for (i = 0; i < M-1; i++)
            h = (h*d)%q;
        /*Calcula el valor hash del patron y primera
        ventana de texto*/
        for (i = 0; i < M; i++)
        {
            p = (d*p + pat.charAt(i))%q;
            t = (d*t + txt.charAt(i))%q;
        }
        //Desliza el patrón por encima del texto uno por uno
        for (i = 0; i <= N - M; i++)
        {
            /*Verifica los valores hash de la actual ventana de text
               y patrón. Si el valor hash coincide entonces solo revisa
               los caratceres uno por uno*/

```

```

        if ( p == t )
            { /*Revisa por caracteres uno por uno*/
for (j = 0; j < M; j++)
{
    if (txt.charAt(i+j) != pat.charAt(j))
        break;
}
// Si p==t y pat[0...M-1] = txt[i, i+1, ...i+M-1]
if (j == M)
    System.out.println("Patron encontrado en el indice " + i);
}
// Calcula el valor hash de la siguiente ventana de texto
// Remueve el digito lider, y final
if ( i < N-M )
{
    t = (d*(t - txt.charAt(i)*h) + txt.charAt(i+M))%q;
/*Nosotros obtendremos un valor negativo de t
   convirtiendolo a positivo*/
    if (t < 0)
t = (t + q);
}
}
}

public static void main(String[] args)
{
    String txt = "EQUIPO ARTEMIS";
    String pat = "ARTE";
int q = 101; // un número primo
    search(pat, txt, q);
}
}

```

## 16.8 Wildcard

Dado un texto y un patrón de wildcards, se debe implementar un algoritmo que encuentre si el patrón wildcard concuerda con el texto, el cotejamiento debe cubrir todo el texto.

El patrón wildcard puede incluir los caracteres '?' y '\*':

'?' – Coteja cualquier carácter solitario.

'\*' – Coteja cualquier secuencia de caracteres (Incluyendo una secuencia vacía).

Cada ocurrencia de '?' en el patrón wildcard puede ser reemplazado con cualquier otro carácter y cada ocurrencia de '\*' con una secuencia de caracteres tales que el patrón de wildcard se convierta en idéntico al string de entrada luego del reemplazo.

Consideremos cualquier carácter en el patrón.

### Caso 1: El carácter es ‘\*’

Aquí dos cosas pueden pasar:

- Podemos ignorar el carácter ‘\*’ y movernos al siguiente carácter del patrón.
- El carácter ‘\*’ coincide con uno o más caracteres en el texto, aquí podemos movernos al carácter siguiente en el string.

### Caso 2: El carácter es ‘?’:

Podemos ignorar el actual carácter en el texto y movernos al siguiente carácter en el patrón y texto.

### Caso 3: El carácter no es un carácter wildcard

Si el carácter actual en el texto coincide con el actual carácter en el patrón, podemos movernos al siguiente carácter en el patrón y el texto, si no concuerdan, el patrón wildcard y el texto.

La complejidad de tiempo es  $O(m \times n)$

```
//Programa java que implementa el cotejamiento
// de patrones con Wildcard
import java.util.Arrays;

public class WildcardPattern {
    //Función que coteja str con el patron wildcard
    static boolean strmatch(String str, String pattern,
    int n, int m) {
        // Patron vacio colo puede coincidir
        // con string vacio
        if (m == 0) {
            return (n == 0);
        }
        // Tabla de busqueda para almacenar resultados
        // de subproblemás
        boolean[][] lookup = new boolean[n + 1][m + 1];
        //Inicializa la tabla en falso
        for (int i = 0; i < n + 1; i++) {
            Arrays.fill(lookup[i], false);
        }
        lookup[0][0] = true;
        //Solo '*' puede coincidri con string vacio
        for (int j = 1; j <= m; j++) {
            if (pattern.charAt(j - 1) == '*') {
                lookup[0][j] = lookup[0][j - 1];
            }
        }
        // Llena la ytabla
        for (int i = 1; i <= n; i++) {
```

```

        for (int j = 1; j <= m; j++) {
/*Dos casos que podemos ver de '*'
    a) Ignoramos '*' y pasamos al siguiente
    caracter en el patrón
    b) '*' coincide con un iesimo
    caracter en una entrada*/
}

if (pattern.charAt(j - 1) == '*') {
    lookup[i][j] = lookup[i][j - 1]
    || lookup[i - 1][j];
}
/*Actuales caracteres son considerados
como coincidentes en dos casos
    a) actual caracter de patron es '?'
    b) caracteres actuales coinciden*/
else if (pattern.charAt(j - 1) == '?'
    || str.charAt(i - 1) == pattern.charAt(j - 1)) {
    lookup[i][j] = lookup[i - 1][j - 1];
} // Si el caracter no coincide
else {
    lookup[i][j] = false;
}
}

return lookup[n][m];
}

public static void main(String args[]) {
    String str = "baaabab";
    String pattern = "*****ba*****ab";
//Casos de prueba
// String pattern = "ba*****ab";
// String pattern = "ba*ab";
// String pattern = "a*ab";
// String pattern = "a*****ab";
// String pattern = "*a*****ab";
// String pattern = "ba*ab****";
// String pattern = "****";
// String pattern = "*";
// String pattern = "aa?ab";
// String pattern = "b*b";
// String pattern = "a*a";
// String pattern = "baaabab";
// String pattern = "?baaabab";
// String pattern = "*baaaba*";

    if (strmatch(str, pattern, str.length(),
        pattern.length())) {
        System.out.println("Si");
    } else {
        System.out.println("No");
    }
}

```

}

## 16.9 Problemas de repaso

### Ejercicios en Online Judge

454-Anagrams

148-Anagram checker

271-Simply Syntax

642- Word Amalgamation

401-Palindromes

10343-Base64 Decoding

455-Periodic Strings

10058- Jimmi's Riddles

### Ejercicios en CodeChef

CHEFSTLT

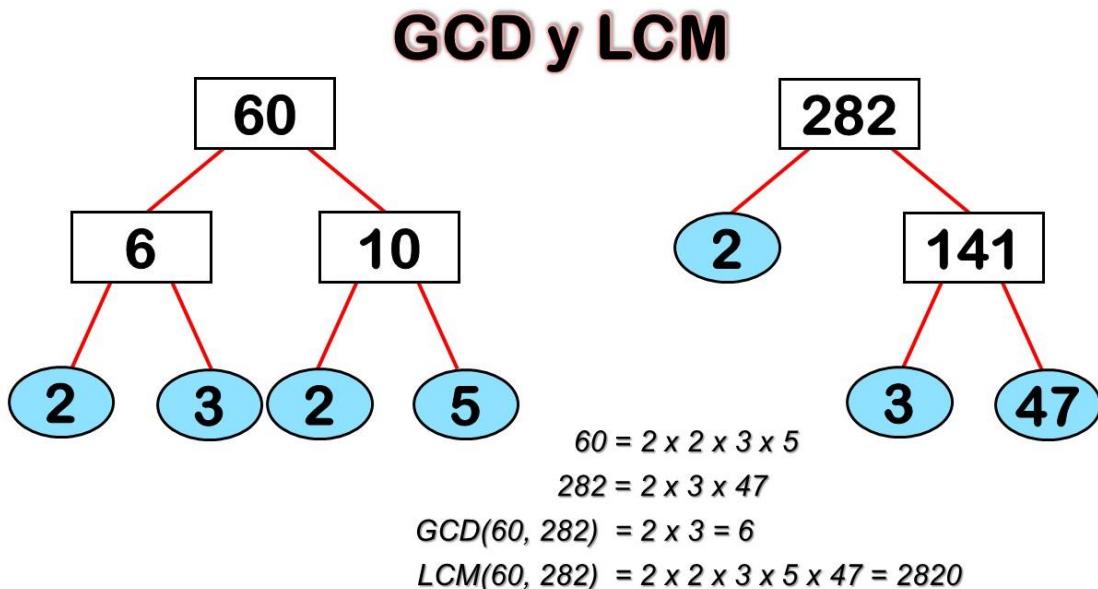
CHRL2

STRPALIN

PLAYST

# 17 Matemática

## 17.1 GCD/LCM



*Guía del programador competitivo.*

Ilustración 17-1 Ejemplo de GCD y LCM

El máximo común divisor (MCD) de dos o más número natural o enteros (no números con decimales) es el número más grande que les divide.

El mínimo común múltiplo de dos números a y b es el número más pequeño que es múltiplo de a y múltiplo de b.

```
//Programa java que realiza
//Maximo comun divisor y minimo comun multiplo
public class GCDLCM {

    public static void main(String[] args) {
        int a = 8, b = 12;
        System.out.println("GCD de a y b es :" + gcd(a, b));
        System.out.println("LCM de a y b es :" + lcm(a, b));
    }
    //Maximo Comun Divisor
    public static int gcd(int a, int b) {
        return b == 0 ? a : gcd(b, a % b);
    }
}
```

```

    }
    //Mínimo Común Multiplo
    public static int lcm(int a, int b) {
        return a * (b / gcd(a, b));
    }
}

```

## 17.2 Primality Test

Un número primo es un número natural mayor que 1 que tiene únicamente dos divisores distintos: él mismo y el 1. Por el contrario, los números compuestos son los números naturales que tienen algún divisor natural aparte de sí mismos y del 1, y, por lo tanto, pueden factorizarse. El número 1, por convenio, no se considera ni primo ni compuesto.

La propiedad de ser número primo se denomina primalidad. El estudio de los números primos es una parte importante de la teoría de números, rama de las matemáticas que trata las propiedades, básicamente aritméticas, de los números enteros.

El teorema fundamental de la aritmética establece que todo número natural tiene una representación única como producto de factores primos, salvo el orden. Un mismo factor primo puede aparecer varias veces. El 1 se representa entonces como un producto vacío.

```

//Programa java que verifica si un número
//es primo o no
public class PrimalityTest {

    public static void main(String[] args) {
        boolean isprime = false;
        int n = 5;
        if (isPrime(n)) {
            System.out.println("Es primo");
        } else {
            System.out.println("No es primo");
        }
    }

    //Prueba de primalidad
    public static boolean isPrime(int x) {
        if (x < 2) {
            return false;
        }
        if (x == 2) {
            return true;
        }
        if (x % 2 == 0) {
            return false;
        }
        ...
    }
}

```

```

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

```

## 17.3 Prime Factors

Los factores primos de un número entero son los números primos divisores exactos de ese número entero. El proceso de búsqueda de esos divisores se denomina factorización de enteros, o factorización en números primos.

Determinar el número de factores primos de un número es un ejemplo de problema matemático frecuentemente empleado para asegurar la seguridad de los sistemas criptográficos: se cree que este problema requiere un tiempo superior al tiempo polinómico en el número de dígitos implicados; de hecho, es relativamente sencillo construir un problema que precisaría más tiempo que la Edad del Universo si se intentase calcular con los ordenadores actuales utilizando algoritmos actuales.

```

//Programa que descompone un número n
// en sus factores primos
public class PrimeFactors {

    public static void main(String[] args) {
        primeFactors(12);
    }

    //Descomposicion en factores primos
    public static void primeFactors(int N) {
        for (long p = 2; p * p <= N; ++p) {
            while (N % p == 0) {
                System.out.println(p);
                N /= p;
            }
        }
        if (N > 1) {
            System.out.println(N);
        }
    }
}

```

## 17.4 Divisibility By Small Numbers In Interval [1-12]

Los criterios de divisibilidad son reglas que sirven para saber si un número es divisible por otro sin necesidad de realizar la división.

Estas son las reglas más comunes:

DIVISIBILIDAD POR 2: Un número es divisible por dos si termina en cero o en cifra par.

- 24 es divisible por 2 porque es par.
- 31 no es divisible por 2 porque no es par.

DIVISIBILIDAD POR 3: Un número es divisible por tres, si la suma de sus cifras es múltiplo de tres.

- 42 es divisible por 3 porque  $4 + 2 = 6$  es múltiplo de tres.
- 43 no es divisible por 3 porque  $4 + 3 = 7$  que no es múltiplo de tres.

DIVISIBILIDAD POR 5: Un número es divisible por cinco cuando acaba en cero o en cinco.

- 35 es divisible por 5 porque acaba en cinco.
- 540 es múltiplo de 5 porque acaba en cero.

DIVISIBILIDAD POR 9: Un número es divisible por nueve cuando la suma de sus cifras es múltiplo de nueve.

- 45 es divisible por 9 porque la suma de sus cifras es múltiplo de 9 ( $4 + 5 = 9$ )
- 738 es múltiplo de 9 porque  $7 + 3 + 8 = 18$ , que es múltiplo de 9.

DIVISIBILIDAD POR 10: Un número es divisible por 10 si termina en cero. De manera similar, si termina en 00 es divisible por 100; si termina en 000 es divisible por 1000.

- El número 70 es divisible por 10 porque termina en cero

```
//Programa que verifica la divisibilidad de un
//número entre los números 1 a 12
public class DivisibilityBySmallNumbers {

    public static void main(String[] args) {
        boolean[] res = divisility(12);
```

```

        for (int i = 0; i < res.length; i++) {
            System.out.println(i+ " : "+res[i]);
        }
    }

    static public boolean[] divisility(int num) {
        String M = String.valueOf(num);
        int n=M.length();
        boolean[] isMultipleOf = new boolean[13];
        int sum = 0;
        for (char c : M.toCharArray()) {
            sum += c - '0';
        }
        int lastDigit = M.charAt(n - 1) - '0';
//Divisibilidad entre 1
        isMultipleOf[1] = true;
//Divisibilidad entre 2
        isMultipleOf[2] = lastDigit % 2 == 0;
//Divisibilidad entre 3
        isMultipleOf[3] = sum % 3 == 0;
//Divisibilidad entre 4
        if (n > 1) {
            isMultipleOf[4] = Integer.parseInt(M.substring(n - 2, n)) % 4 == 0;
        } else {
            isMultipleOf[4] = lastDigit % 4 == 0;
        }
//Divisibilidad entre 5
        isMultipleOf[5] = lastDigit == 0 || lastDigit == 5;
//Divisibilidad entre 6
        isMultipleOf[6] = isMultipleOf[2] && isMultipleOf[3];
        int altSum = 0;
        int[] pattern = {1, 3, 2, -1, -3, -2};
        int j = 0;
        for (int i = n - 1; i >= 0; i--) {
            altSum += pattern[j] * (M.charAt(i) - '0');
        }
        j = (j + 1) % 6;
        }
//Divisibilidad entre 7
        isMultipleOf[7] = Math.abs(altSum) % 7 == 0;
//Divisibilidad entre 8
        if (n > 2) {
            isMultipleOf[8] = Integer.parseInt(M.substring(n - 3, n)) % 8 == 0;
        } else {
            isMultipleOf[8] = Integer.parseInt(M) % 8 == 0;
        }
//Divisibilidad entre 9
        isMultipleOf[9] = sum % 9 == 0;
//Divisibilidad entre 10
        isMultipleOf[10] = lastDigit == 0;
        altSum = 0;
        int s = 1;
        for (int i = n - 1; i >= 0; i--) {
            altSum += s * (M.charAt(i) - '0');
        }
        s = -s;
    }
}

```

```

//Divisibilidad entre 11
    isMultipleOf[11] = Math.abs(altSum) % 11 == 0;
    //Divisibilidad entre 12
isMultipleOf[12] = isMultipleOf[3] && isMultipleOf[4];
    return isMultipleOf;
}
}

```

## 17.5 Number Of Divisors

Los divisores de un número son aquellos valores que dividen al número en partes exactas.

Así, dado un número  $a$ , si la división  $a/b$  es exacta (el resto es cero), entonces se dice que  $b$  es divisor de  $a$ . También se puede decir que  $a$  es divisible por  $b$  o que  $a$  es un múltiplo de  $b$ .

Esto nos resulta útil, por ejemplo, a la hora de agrupar una cantidad de objetos en partes iguales sin que nos sobre ninguno.

Lógicamente, el 1 siempre es divisor de cualquier número, porque siempre podemos hacer paquetes individuales y no nos sobrará ninguno. De igual forma, todo número es divisible por sí mismo, lo que equivaldría a hacer un único paquete.

```

//Programa java que cuenta el número
// de divisores de x
public class NumberOfDivisors {

    public static void main(String[] args) {
        System.out.println(divisors(56));
    }

    //Todos los divisores de un número
    public static int divisors(int x) {
        int nDiv = 1;
        for (int p = 2; p * p <= x; ++p) {
            int cnt = 0;
            while (x % p == 0) {
                ++cnt;
                x /= p;
            }
            nDiv *= cnt + 1;
        }
        if (x > 1) {
            nDiv *= 2;
        }
        return nDiv;
    }
}

```

## 17.6 Prime Numbers (Criba de Eratóstenes)

### Criba de Eratóstenes

	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100

*Guía del programador competitivo.*

Ilustración 17-2 Vista en matriz de búsqueda de primos por medio de la criba de Eratóstenes

Criba de Eratóstenes es un algoritmo que permite hallar todos los números primos menores que un número natural dado  $n$ . Se forma una tabla con todos los números naturales comprendidos entre 2 y  $n$ , y se van tachando los números que no son primos de la siguiente manera:

Comenzando por el 2, se tachan todos sus múltiplos; comenzando de nuevo, cuando se encuentra un número entero que no ha sido tachado, ese número es declarado primo, y se procede a tachar todos sus múltiplos, así sucesivamente. El proceso termina cuando el cuadrado del siguiente número confirmado como primo es mayor que  $n$ .

```
//Programa java que encuentra todos los números
//primos bajo N
import java.util.Arrays;

public class PrimeNumbers {

    public static void main(String[] args) {
        boolean[] res = sieveEratostenes(100);
        for (int i = 0; i < res.length; i++) {
```

```

        System.out.println(i+" : "+res[i]);
    }
}

//Criba de Eratostenes (Todos los primos bajo un número)
public static boolean[] sieveEratostenes(int N) {
    boolean[] prime = new boolean[N + 1];
    Arrays.fill(prime, true);
    prime[0] = prime[1] = false;
    for (int p = 2; p * p <= N; p++) {
        if (prime[p]) {
            for (int i = p * p; i <= N; i += p) {
                prime[i] = false;
            }
        }
    }
    return prime;
}
}

```

## 17.7 Fibonacci Last Digit

La sucesión o serie de Fibonacci es la siguiente sucesión infinita de números naturales: 0 1 1 2 3 5 8 21 34 55 89....

La sucesión comienza con los números 0 y 1, y a partir de estos, cada término es la suma de los dos anteriores.

Los números de esta sucesión pueden llegar a ser muy grandes y calcular su último dígito se hace complicado en máquinas.

```

// Programa java que encuentra el ultimo dígito
// de un nesimo número fibonacci
public class FibonacciLastDigit {
    //Funcion que retorna el nesimo
    // Número de fibonacci
    static long fib(long n) {
        long F[][] = new long[][]{{1, 1}, {1, 0}};
        if (n == 0) {
            return 0;
        }
        power(F, n - 1);
        return F[0][0];
    }
    //Función que multiplica dos
    // matrices y almacera el resultado en la primera
    static void multiply(long F[][], long M[][])
    {
        long x = F[0][0] * M[0][0]

```

```

        + F[0][1] * M[1][0];
    long y = F[0][0] * M[0][1]
        + F[0][1] * M[1][1];
    long z = F[1][0] * M[0][0]
        + F[1][1] * M[1][0];
    long w = F[1][0] * M[0][1]
        + F[1][1] * M[1][1];
    F[0][0] = x;
    F[0][1] = y;
    F[1][0] = z;
    F[1][1] = w;
}
static void power(long F[][], long n) {
    if (n == 0 || n == 1) {
        return;
    }
    long M[][] = new long[][]{{1, 1}, {1, 0}};
    power(F, n / 2);
    multiply(F, F);
    if (n % 2 != 0) {
        multiply(F, M);
    }
}
// Retorna el ultimo digito
// nesimo numero fibonacci
public static long findLastDigit(long n) {
    return (fib(n) % 10);
}
public static void main(String[] args) {
    int n;
    n = 1;
    System.out.println(findLastDigit(n));
    n = 61;
    System.out.println(findLastDigit(n));
    n = 7;
    System.out.println(findLastDigit(n));
    n = 67;
    System.out.println(findLastDigit(n));
}
}

```

## 17.8 Long Fibonacci

```

//Programa java que busca el nesimo numero
// de fibonacci cuando n puede ser muy largo
import java.math.*;

public class LongFibonacci
{
    static BigInteger fib(int n)
    {
        BigInteger a = BigInteger.valueOf(0);

```

```

        BigInteger b = BigInteger.valueOf(1);
        BigInteger c = BigInteger.valueOf(1);
        for (int j=2 ; j<=n ; j++)
        {
            c = a.add(b);
            a = b;
            b = c;
        }
        return (a);
    }

    public static void main(String[] args)
    {
        int n = 1000;
        System.out.println("Fibonacci de " + n +
                           " termino" + " " +"es" +" " + fib(n));
    }
}

```

## 17.9 Is Fibonacci Number?

```

// Programa que verifica si x es número fibonacci
public class IsFiboNumber
{
    // Función que retorna si x es cuadrado perfecto
    static boolean isPerfectSquare(int x)
    {
        int s = (int) Math.sqrt(x);
        return (s*s == x);
    }
    //Función que verifica si es número fibonacci
    static boolean isFibonacci(int n)
    {
        // N es fibonacci si uno de 5*n*n+4 o or 5*n*n - 4
        // o ambos son cuadrados perfectos
        return isPerfectSquare(5*n*n + 4) ||
               isPerfectSquare(5*n*n - 4);
    }

    public static void main(String[] args)
    {
        for (int i = 1; i <= 10; i++)
            System.out.println(isFibonacci(i) ? i + " Es un número de
fibonacci" :
                               i + " No es un número de
fibonacci");
    }
}

```

## 17.10 All Permutations

Una permutación es la variación del orden o posición de los elementos de un conjunto ordenado o una tupla.

Si el orden no importa, es una combinación.

Si el orden sí importa es una permutación.

```
//Programa java que imprime todas las permutaciones
// de un conjunto
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class AllPermutations {

    static Set<String> permutations;
    static Set<String> result = new HashSet<String>();

    public static Set<String> permutation(String string) {
        permutations = new HashSet<>();

        int n = string.length();
        for (int i = n - 1; i >= 0; i--) {
            shuffle(string.charAt(i));
        }
        return permutations;
    }

    private static void shuffle(char c) {
        if (permutations.isEmpty()) {
            permutations.add(String.valueOf(c));
        } else {
            Iterator<String> it = permutations.iterator();
            for (int i = 0; i < permutations.size(); i++) {

                String temp1;
                for (; it.hasNext();) {
                    temp1 = it.next();
                    for (int k = 0; k < temp1.length() + 1; k += 1) {
                        StringBuilder sb = new StringBuilder(temp1);

                        sb.insert(k, c);
                        result.add(sb.toString());
                    }
                }
            }
            permutations = result;
            result = new HashSet<>();
        }
    }

    public static void main(String[] args) {
        Set<String> finalresult = permutation("1234");
    }
}
```

```

        System.out.println("\nHay en total " + finalresult.size() + " "
permutaciones:");
        Iterator<String> it = finalresult.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}

```

## 17.11 Combinatoria

La Combinatoria es la parte de las Matemáticas que estudia las diversas formas de realizar agrupaciones con los elementos de un conjunto, formándolas y calculando su número.

Existen distintas formas de realizar estas agrupaciones, según se repitan los elementos o no, según se puedan tomar todos los elementos de que disponemos o no y si influye o no el orden de colocación de los elementos.

```

//Programa java que imprime todas las combinaciones
// de tamaño r en un arreglo de tamaño n
public class Combination {

    /* arr[] ---> Array de entrada
    data[] ---> Array temporal que almacenara la combinación actual
    start & end ---> Iniciando y terminando indices en arr[]
    index ---> Actual indice en data[]
    r ---> Tamaño de una combinacion a ser impresa*/
    static void combinationUtil(int arr[], int data[], int start,
                                int end, int index, int r) {
        // Actual combinación esta lista para ser impresa, se imprime
        if (index == r) {
            for (int j = 0; j < r; j++) {
                System.out.print(data[j] + " ");
            }
            System.out.println("");
            return;
        }

        for (int i = start; i <= end && end - i + 1 >= r - index; i++) {
            data[index] = arr[i];
            combinationUtil(arr, data, i + 1, end, index + 1, r);
        }
    }
    static void printCombination(int arr[], int n, int r) {
        int data[] = new int[r];
        combinationUtil(arr, data, 0, n - 1, 0, r);
    }
    public static void main(String[] args) {
        int arr[] = {1, 2, 3, 4, 5};

```

```

        int r = 3;
        int n = arr.length;
printCombination(arr, n, r);
    }
}

```

## 17.12 Compose Combinatory

```

//Programa java que imprime todas
//las combinaciones que pueden componer un número dado
public class CombinationsCompose {
//Función imprime todas las combinaciones de números 1, 2, ...MAX_POINT
    // que su suma resulte n
    // i es usado en recursion para mantener revisión del indice
    // en arr[] donde el siguiente elemento sera añadido
    // Valor inicial de i debe ser pasado como 0

static void printCompositions(int arr[], int n, int i) {
    int MAX_POINT = 3;
    if (n == 0) {
        printArray(arr, i);
    } else if (n > 0) {
        for (int k = 1; k <= MAX_POINT; k++) {
            arr[i] = k;
            printCompositions(arr, n - k, i + 1);
        }
    }
}

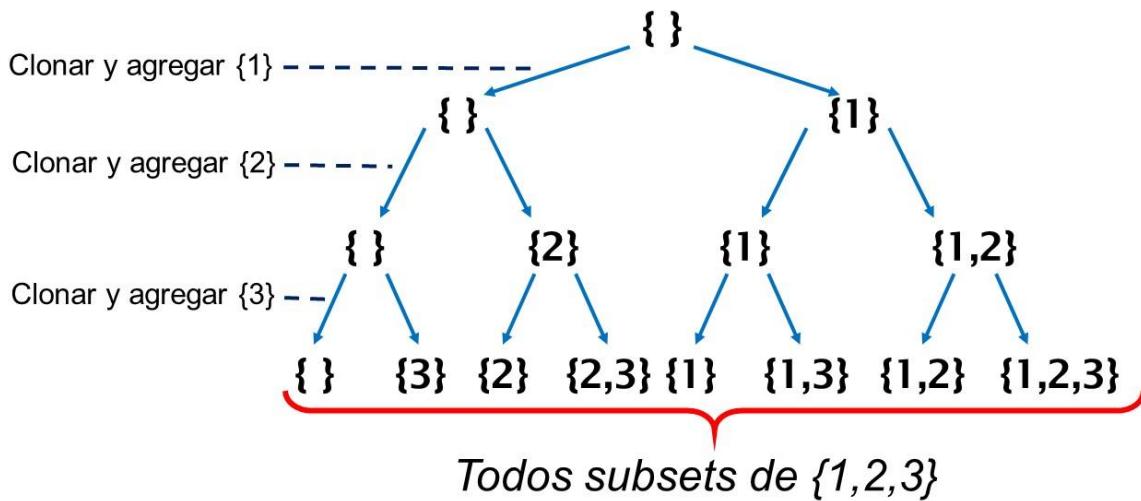
// Imprime array
static void printArray(int arr[], int m) {
    for (int i = 0; i < m; i++) {
        System.out.print(arr[i] + " ");
    }
    System.out.println();
}

public static void main(String[] args) {
    int n = 5;
    int size = 100;
    int[] arr = new int[size];
System.out.println("Diferentes composiciones formadas por 1, 2 y 3 de " + n + " "
son ");
    printCompositions(arr, n, 0);
}
}

```

## 17.13 All Subsets Of a Set

# All subsets



Guía del programador competitivo.

Ilustración 17-3 Búsqueda de todos los subconjuntos de un conjunto (Set)

Un conjunto es una colección de elementos con características similares considerada en sí misma como un objeto. Los elementos de un conjunto, pueden ser las siguientes: personas, números, colores, letras, figuras, etc.

Se dice que un elemento (o miembro) pertenece al conjunto si está definido como incluido de algún modo dentro de él.

```
//Programa java que imprime todos los subconjuntos de
//un conjunto
public class Allsets {
    static void printSubsets(char set[]) {
        int n = set.length;
        // Ejecuta un ciclo imprimiendo todos
        // los subconjuntos 2^n uno por uno
        for (int i = 0; i < (1 << n); i++) {
            System.out.print("{ ");
            //Imprime el subconjunto actual
            for (int j = 0; j < n; j++) // (1<<j) es un número con jesimo bit 1
            {
                if ((i & (1 << j)) > 0) {
                    System.out.print(set[j] + " ");
                }
            }
            System.out.println("}");
        }
    }
}
```

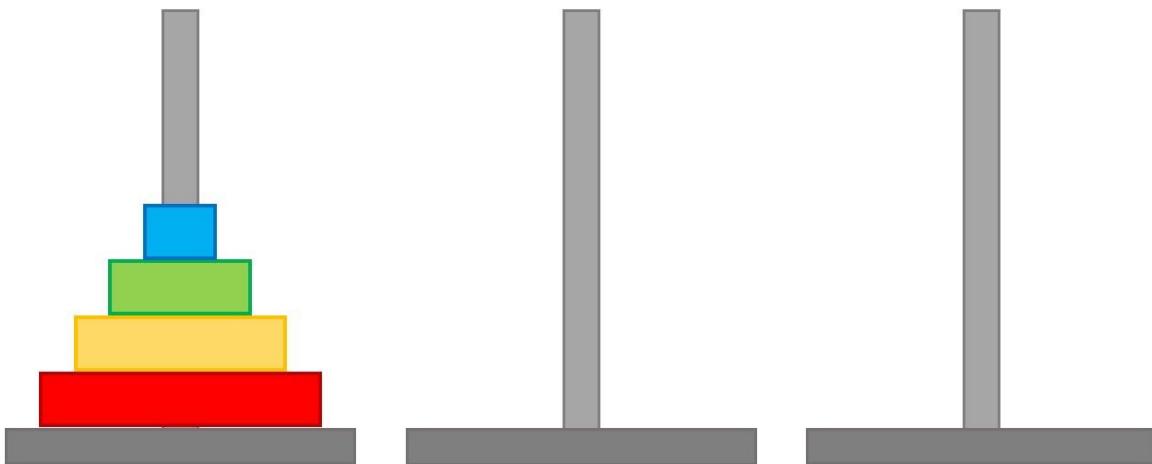
```

public static void main(String[] args) {
    char set[] = {'a', 'b', 'c'};
    printSubsets(set);
}
}

```

## 17.14 Hanoi Towels

### Torre De Hanoi



*Guía del programador competitivo.*

Ilustración 17-4 Juego "Las Torres de Hanoi"

Las Torres de Hanói es un rompecabezas o juego matemático inventado en 1883 por el matemático francés Édouard Lucas.<sup>1</sup> Este juego de mesa individual consiste en un número de discos perforados de radio creciente que se apilan insertándose en uno de los tres postes fijados a un tablero. El objetivo del juego es trasladar la pila a otro de los postes siguiendo ciertas reglas, como que no se puede colocar un disco más grande encima de un disco más pequeño. El problema es muy conocido en la ciencia de la computación y aparece en muchos libros de texto como introducción a la teoría de algoritmos.

La fórmula para encontrar el número de movimientos necesarios para transferir  $n$  discos desde un poste a otro es:  $2^n - 1$

```

//Programa java que calcula los movimientos
// necesarios para completar las torres de Hanoi

```

```

// Sin importar el número de discos n
public class HanoiTowels {

    public static void main(String[] args) {
        //Número de discos
        int n;
        n = 8;
        hanoi(n, "Primera torre", "Segunda torre", "Tercera torre");
    }
    static int paso = 1;
    //Función recursiva de búsqueda
    static void hanoi(int n, String from, String temp, String to) {
        if (n == 0) {
            return;
        }
        hanoi(n - 1, from, to, temp);
        System.out.println(paso + ": Mover disco " + n + " de " + from + " a " +
        to);
        paso++;
        hanoi(n - 1, temp, from, to);
    }
}

```

## 17.15 AX+BY=N

Dado a, b y n. Encuentre x y y que satisfaga  $ax+by=n$ , imprima cualquiera de las x y y que cumplan la ecuación.

Podemos verificar si alguna solución existe o no usando ecuaciones lineales de Diofanes, pero ahí necesitamos encontrar para esta ecuación, entonces podemos simplemente iterar por todos los posibles valores de 0 a n sin exceder n para esta ecuación. Entonces resolviendo esta ecuación con lápiz y papel obtenemos  $y=(n-ax)/b$  y similarmente obtenemos el otro número con  $x=(n-by)/a$ , si ninguno de los valores satisface la ecuación, al final imprimimos “Sin solución”.

```

//Programa que calcula la solución de
// ax + by = n

public class AXplusBYequalsN {

    static void solution(int a, int b, int n)
    {
        //A traves de todos los posibles valores
        for (int i = 0; i * a <= n; i++)
        {
            //Verifica si se satisface la ecuación

```

```

if ((n - (i * a)) % b == 0)
{
    System.out.println("x = " + i +
                       ", y = " +
                       (n - (i * a)) / b);
    return ;
}
}

System.out.println("Sin solución");
}

public static void main (String[] args)
{
    int a = 2, b = 3, n = 7;
    solution(a, b, n);

}
}

```

## 17.16 A%X=B

Dados dos números a y b, encontrar todos los x que permitan  $a \% x = b$ .

Existen tres casos:

- Si a es menor que b entonces no habrá respuesta.
- Si a es igual que b entonces todos los números más grandes que a, habrá infinitas soluciones.
- Si a es mayor que b, supone que x es una respuesta a nuestra ecuación. Entonces x divide (a-b) también desde  $a \% x = b$  entonces b es menor que x

```

/* Programa java que encuentra x tal que
a % x es igual b.*/
public class AmodXequalB {

    static void modularEquation(int a, int b) {
        // Si a es menor que b, entonces no hay solución
        if (a < b) {
            System.out.println("No solution possible ");
            return;
        }
        /*Si a es igual a b, entonces cada número
           más grande que a sera la solución, entonces

```

```

        es infinito*/
        if (a == b) {
System.out.println("Infinite Solution possible ");
return;
    }
/*todo el número resultante debe ser mayor
que b y (a-b) deben ser divisibles
por número resultante
variable count almacena el número de
valores posibles*/
int count = 0;
int n = a - b;
int y = (int) Math.sqrt(a - b);
for (int i = 1; i <= y; ++i) {
if (n % i == 0) {
/*Revisando por ambos divisor y
cociente cual divide (a-b) completamente
y mayor que b*/
    if (n / i > b) {
        count++;
    }
    if (i > b) {
        count++;
    }
}
}
/* Aqui y es añadido dos veces en la
ultima iteración entonces y deberia ser decrementado
para obtener la solución correcta*/
if (y * y == n && y > b) {
    count--;
}
System.out.println(count);
}

public static void main(String[] args) {
    int a = 21, b = 5;
    modularEquation(a, b);
}
}

```

## 17.17 Factorial

El factorial de un entero positivo  $n$ , el factorial de  $n$  o  $n$  factorial se define en principio como el producto de todos los números enteros positivos desde 1 (es decir, los números naturales) hasta  $n$ .

La operación de factorial aparece en muchas áreas de las matemáticas, particularmente en combinatoria y análisis matemático. De manera fundamental la factorial de n representa el número de formas distintas de ordenar n objetos distintos (elementos sin repetición).

```
// Programa java que encuentra el
// factorial de un número
public class Factorial {
    static int factorial(int n) {
        if (n == 0) {
            return 1;
        }

        return n * factorial(n - 1);
    }

    public static void main(String[] args) {
        int num = 5;
        System.out.println("Factorial de " + num + " es " + factorial(5));
    }
}
```

## 17.18 Big Factorial

```
// Programa java que calcula factoriales
// de números enormes

public class BigFactorial {

    static void factorial(int n)
    {
        int res[] = new int[500];
        res[0] = 1;
        int res_size = 1;
        // Aplicamos la formula normal de factorial
        // n! = 1 * 2 * 3 * 4...*n
        for (int x = 2; x <= n; x++)
            res_size = multiply(x, res, res_size);

        System.out.println("Factorial of given number is ");
        for (int i = res_size - 1; i >= 0; i--)
            System.out.print(res[i]);
    }

    static int multiply(int x, int res[], int res_size)
    {
        int carry = 0; // inicializar carry
        // Uno por uno multiplicamos n con
        // los dígitos individuales de res[]
        for (int i = 0; i < res_size; i++)
        {
```

```

        int prod = res[i] * x + carry;
    res[i] = prod % 10; //Almacenar ultimo digito de
    // 'prod' en res[]
        carry = prod/10; // Poner el resto de carry
    }
    //Pone el carry en res e incrementa el tamaño del resultado size
while (carry!=0)
{
    res[res_size] = carry % 10;
    carry = carry / 10;
    res_size++;
}
return res_size;
}

public static void main(String args[])
{
    factorial(100);
}
}

```

## 17.19 Binomial Coefficients

Los coeficientes binomiales, números combinatorios o combinaciones son números estudiados en combinatoria que corresponden al número de formas en que se puede extraer subconjuntos a partir de un conjunto dado.

Se tiene un conjunto con seis objetos diferentes {A, B, C, D, E, F}, de los cuales se desea escoger dos (sin importar el orden de elección). Existen 15 formas de efectuar tal elección:

```

// Programa java que calcula el valor
// de coeficientes binomiales
public class BinomialCoefficients {
// Retorna el valor del coeficiente binomial
// C(n, k)
    static int binomialCoeff(int n, int k)
    {

// Casos base
        if (k == 0 || k == n)
            return 1;

// Recursión
        return binomialCoeff(n - 1, k - 1) +
               binomialCoeff(n - 1, k);
    }

    public static void main(String[] args)
    {

```

```

        int n = 5, k = 2;
        System.out.printf("Valor de C(%d, %d) es %d ",
                           n, k, binomialCoeff(n, k));
    }
}

```

## 17.20 Cassini Identity

La identidad de Cassini y la identidad de Catalan son relaciones matemáticas ligadas con los números de la sucesión de Fibonacci, afirma que para cada número  $n$ -ésimo de la sucesión de Fibonacci, se cumple que:

- $F_{n-1} \times F_n + F_{n+1}^2 = (-1)^n$

```

//Programa java que demuestra
// la identidad de Cassini

public class CassiniIdentity
{
// Retorna (-1)^n
static int cassini(int n)
{
    return (n & 1) != 0 ? -1 : 1;
}
public static void main(String args[])
{
    int n = 5;
    System.out.println(cassini(n));
}
}

```

## 17.21 E pow X

El valor de la función exponencial  $e^x$  puede ser expresado usando la siguiente serie de Taylor:

- $e^x = 1 + x/1! + x^2/2! + x^3/3! + \dots$

¿Cómo calcular eficientemente la suma de la serie de arriba? Puede ser escrita de la siguiente forma:

- $e^x = 1 + (x/1) (1 + (x/2) (1 + (x/3) (\dots)))$

```

// Programa eficiente que calcula e elevado x
public class EpowX {
    // Función que retorna aproximado de e^x
    // Usando suma de los primeros n términos
    // de la serie de Taylor
    static float exponential(int n, float x) {
        float sum = 1;
        for (int i = n - 1; i > 0; --i) {
            sum = 1 + x * sum / i;
        }
        return sum;
    }

    public static void main(String[] args) {
        int n = 10;
        float x = 1;
        System.out.println("e^x = " + exponential(n, x));
    }
}

```

## 17.22 Euclid-Euler Theorem

De acuerdo con el teorema de Euclides-Euler, un número perfecto el cual es par, puede ser representado de la forma  $(2^n - 1) \cdot (2^n / 2)$  donde  $n$  es un número primo y  $2^n - 1$  es un número primo de Mersenne. Este es un producto de la potencia de 2 con un primo Mersenne, este teorema establece una conexión entre un número primo de Mersenne y un número par primo perfecto.

Algunos ejemplos de números perfectos los cuales satisfacen este teorema son:

- 6, 28, 496, 8128, 33550336, 8589869056, 137438691328

Explicación:

1) 6 es un número perfecto par.

Entonces puede ser escrito de la forma

- $(2^2 - 1) \cdot (2(2 - 1)) = 6$

Donde  $n = 2$  es un número primo y  $2^n - 1 = 3$  es un número primo de Mersenne

Toma cada número primo y forma un primo de Mersenne con él. El primo de Mersenne =  $2^n - 1$  donde n es primo. Ahora formamos el número  $(2^n - 1) * (2^{n-1})$  y verificamos si es par y perfecto.

```
//Programa que verifica el teorema de Euclides Euler
import java.util.ArrayList;

public class EuclidEulerTheorem {

    static ArrayList<Long> power2 = new ArrayList<Long>();

    public static void main(String[] args) {
        //Almacenando potencias de 2 para acceder
        // en tiempo O(1)
        for (int i = 0; i < 62; i++) {
            power2.add(0L);
        }

        for (int i = 0; i <= 60; i++) {
            power2.set(i, (1L << i));
        }
        System.out.println("Generando los primeros números que satisfacen el
teorema de Euclid Euler\n");
        for (long i = 2; i <= 25; i++) {
            long no = ((power2.get((int) i) - 1L) * (power2.get((int) (i -
1))));

            if (isperfect(no) && (no % 2 == 0)) {
                System.out.println("(2^" + i + " - 1) * (2^( " + i + " - 1)) = "
+ no + "\n");
            }
        }
    }

    static boolean isperfect(long n) // Números perfectos
    { /* Verifica si n es suma perfecta de divisores
       excepto por el número en si mismo*/
        long s = -n;
        for (long i = 1; i * i <= n; i++) {
            // es i un divisor de n
            if (n % i == 0) {
                long factor1 = i, factor2 = n / i;
                s += factor1 + factor2;
            // aqui i*i == n
                if (factor1 == factor2) {
                    s -= i;
                }
            }
        }
        return (n == s);
    }

    boolean isprime(long n) {
```

```

// Verifica cual número es primo o no
for (int i = 2; i * i <= n; i++) {
    if (n % i == 0) {
        return false;
    }
}
return true;
}

```

## 17.23 Euclidean Algorithm

El MCD de dos números es el número más grande que divide ambos. Una forma simple de encontrar este número es factoriar ambos números y multiplicar los factores comunes.

El algoritmo se basa en lo siguiente:

- Si restamos el número más pequeño del más grande, MCD (GCD) no cambia, entonces si seguimos restando repetidamente el más grande dos, terminamos con MCD.
- Ahora en vez de restar, si dividimos el número más pequeño, el algoritmo termina cuando encontramos residuo 0.
- 

La complejidad de tiempo es:  $O(\log \min(a, b))$

```

//Programa java que demuestra el algoritmo de Euclides
public class EuclideanAlgorithm
{
// Algoritmo de euclides extendido
public static int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b%a, a);
}

public static void main(String[] args)
{
    int a = 10, b = 15, g;
    g = gcd(a, b);
    System.out.println("GCD(" + a + " , " + b+ ") = " + g);

    a = 35; b = 10;
    g = gcd(a, b);
}

```

```

        System.out.println("GCD(" + a + " , " + b+ ") = " + g);

        a = 31; b = 2;
        g = gcd(a, b);
        System.out.println("GCD(" + a + " , " + b+ ") = " + g);

    }
}

```

## 17.24 Euler Totient

La función totient de Euler para una entrada n es un conteo de números desde 1 hasta n que son primos relativos con n, por ejemplo, los números cuyo GCD (Máximo común divisor) con n es 1.

El criptosistema RSA es basado en este teorema.

```

// Programa simple que calcula el valor
// de la función totient Euler
public class EulerTotient {
    // Función que retorna gcd de a y b
    static int gcd(int a, int b)
    {
        if (a == 0)
            return b;
        return gcd(b % a, a);
    }
    // Función que evalua función totient de euler
    static int phi(int n)
    {
        int result = 1;
        for (int i = 2; i < n; i++)
            if (gcd(i, n) == 1)
                result++;
        return result;
    }

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

        for (n = 1; n <= 10; n++)
            System.out.println("phi(" + n + ") = " + phi(n));
    }
}

```

## 17.25 Number of Digits Factorial

Dado un entero  $n$ , encuentre el número de dígitos que aparecen en este factorial, donde factorial es definido como,  $\text{factorial}(n) = 1*2*3*4.....*n$  y  $\text{factorial}(0) = 1$ .....

Una solución ingenua puede ser calcular  $n!$  primero y luego calcular el número de dígitos presentes en él, sin embargo el valor de  $n!$  puede ser muy largo. Se vuelve algo complicado de almacenar esta variable (A menos que estés trabajando en Python).

Una mejor solución puede ser usar las útiles propiedades de los logaritmos para calcular la respuesta.

Sabemos que:

$$- \log(a*b) = \log(a) + \log(b)$$

Por lo tanto:

$$- \log(n!) = \log(1*2*3.....*n) = \log(1) + \log(2) + ..... + \log(n)$$

Ahora, observamos que el valor piso del logaritmo base 10 incrementado 1 de cualquier número da el número de dígitos presentes en ese número.

Entonces la salida puede ser:  $\text{floor}(\log(n!)) + 1$ .

```
// Programa java que encuentra número de
// dígitos en un factorial
public class FactorialNumDigits
{
    // Retorna el número de dígitos
    // en n!
    static int findDigits(int n)
    {
        // Factorial existe solo para n>=0
        if (n < 0)
            return 0;
        // Caso base
        if (n <= 1)
            return 1;
        // si no itera atravez de n y calcula el valor
        double digits = 0;
        for (int i=2; i<=n; i++)
            digits += Math.log10(i);

        return (int)(Math.floor(digits)) + 1;
    }

    public static void main (String[] args)
    {
        System.out.println(findDigits(1));
```

```

        System.out.println(findDigits(5));
        System.out.println(findDigits(10));
        System.out.println(findDigits(120));
    }
}

```

## 17.26 Number of Digits Factorial Optimized

Si la solución anterior no es lo suficientemente rápida, podemos usar la fórmula de Kamenetsky para obtener la respuesta.

Se aproxima al número de dígitos en una factorial con:

$$- \quad f(x) = \log_{10}((n/e)^n) * \sqrt{2\pi n}$$

Además podemos fácilmente usar las propiedades de los logaritmos para obtener:

$$- \quad f(x) = n \cdot \log_{10}(n/e) + \log_{10}(2\pi n)/2$$

Esta solución puede manejar número muy grandes de entrada, que pueden caber en un entero de 32 bits, e incluso más que esto.

```

// Programa java que encuentra el número de dígitos
// en un factorial
public class FactorialNumDigitsOP {

    public static double M_E = 2.71828182845904523536;
    public static double M_PI = 3.141592654;
    /* Función que retorna el número de dígitos presente en
       n! desde que el resultado sea muy largo*/
    static long findDigits(int n) {

        if (n < 0) {
            return 0;
        }

        // caso base
        if (n <= 1) {
            return 1;
        }
        // Usamos la fórmula de Kamenetsky para
        // calcular el número de dígitos
        double x = (n * Math.log10(n / M_E)
        + Math.log10(2 * M_PI * n)
        / 2.0);

        return (long) Math.floor(x) + 1;
    }
}

```

```

public static void main(String[] args) {
    System.out.println(findDigits(1));
    System.out.println(findDigits(50000000));
    System.out.println(findDigits(100000000));
    System.out.println(findDigits(120));
}
}

```

## 17.27 Fermat Little Theorem

El pequeño teorema de Fermat dice que si  $p$  es un número primo, entonces para cualquier entero  $a$ , el número de  $p-a$  es un entero múltiplo de  $p$ .

Aquí  $p$  es un número primo:

- $a^p \equiv a \pmod{p}$ .

Caso especial: si  $a$  no es divisible por  $p$ , el pequeño teorema de Fermat es equivalente a la sentencia que  $a^{p-1} - 1$  es un entero múltiplo de  $p$ .

- $a^{p-1} \equiv 1 \pmod{p}$

O

- $a^{p-1} \% p = 1$  Aquí  $a$  no es divisible por  $p$ .

### Usos del pequeño teorema de Fermat

Si sabemos que  $m$  es primo, entonces podemos también usar el pequeño teorema de Fermat para buscar la inversa:

- $a^{m-1} \equiv 1 \pmod{m}$

Si nosotros multiplicamos ambos saldos con  $a-1$ , obtenemos:

- $a^{-1} \equiv a^{m-2} \pmod{m}$

```

/*Programa de Java para encontrar modular.
   inverso de un módulo bajo m
   utilizando el pequeño teorema de Fermat.
   Este programa funciona solo si m es primo.*/
public class FermatLittleTheorem
{

```

```

static int __gcd(int a, int b)
{
    if(b == 0)
    {
        return a;
    }
    else
    {
        return __gcd(b, a % b);
    }
}
// Computa x^y bajo modulo m
static int power(int x,int y,int m)
{
    if (y == 0)
        return 1;
    int p = power(x, y / 2, m) % m;
    p = (p * p) % m;

    return (y % 2 == 0) ? p : (x * p) % m;
}
// Función para encontrar modular
// inverso bajo un modulo m
// Asumimos m es primo
static void modInverse(int a, int m)
{
    if (__gcd(a, m) != 1)
        System.out.print("No existe inverso");

    else {
// Si a y m son primos relativos, entonces
// modulo inverso es a^(m-2) mod m
        System.out.print("Multiplicación modular inversa es "
+power(a, m - 2, m));
    }
}

public static void main (String[] args)
{
    int a = 3, m = 11;
    modInverse(a, m);
}
}

```

## 17.28 Fibonacci Golden Ratio

Existen diferentes métodos para encontrar el enésimo número de Fibonacci, una simple manera de encontrarlo es usando el ratio dorado.

Golden ratio (Ratio dorado):

$$\varphi = \frac{1 + \sqrt{5}}{2} = 1.6180339887 \dots$$

Golden ratio nos puede dar una respuesta incorrecta.

Podemos obtener la respuesta correcta si redondeamos hacia arriba el resultado de cada punto.

Este método puede calcular los primeros 34 números de Fibonacci correctamente, luego de esto puede haber diferencia con el valor correcto.

```
// Programa java que encuentra el
// nesimo número fibonacci
public class FiboGoldenRatio
{
    //Valor aproximado del golden ratio
    // Approximate value of golden ratio
    static double PHI = 1.6180339;
    // Números fibonacci hasta 5
    static int f[] = { 0, 1, 1, 2, 3, 5 };
    // Función que encuentra nesimo
    // número fibonacci
    static int fib (int n)
    {
        // Números fibonacci menores a 6
        if (n < 6)
            return f[n];
        // Si no comience conteo desde el quinto
        int t = 5;
        int fn = 5;
        while (t < n) {
            fn = (int)Math.round(fn * PHI);
            t++;
        }
        return fn;
    }

    public static void main (String[] args)
    {
        int n = 9;
        System.out.println(n + " número fibonacci= "
                           +fib(n));
    }
}
```

## 17.29     Float Great Common Divisor

Una aproximación simple para realizar GCD a números flotantes es:

- $a=1.20$
- $b=22.5$

Expresando cada uno de los números sin decimales como el producto de los primos obtenemos:

- $120=2^3 \cdot 3 \cdot 5$
- $2250=2 \cdot 3^2 \cdot 5^3$

H.C.F. de 120 y 2250 =  $2 \cdot 3 \cdot 5 = 30$

Por lo tanto, el H.C.F. de 1.20 y 22.5=0.30 (Tomando 2 dígitos decimales).

Podemos hacer esto usando el algoritmo de Euclides, Este algoritmo indica si el número más pequeño es restado del número más largo, el GCD de dos números no cambia.

```
// Programa java que encuentra el GCD de dos
// números flotantes
public class FloatGCD {
    // Funcion recursiva que retorna gcd de a y b
    static double gcd(double a, double b)
    {
        if (a < b)
            return gcd(b, a);

        // Caso base
        if (Math.abs(b) < 0.001)
            return a;

        else
            return (gcd(b, a -
                        Math.floor(a / b) * b));
    }

    public static void main(String args[])
    {
        double a = 1.20, b = 22.5;
        System.out.printf("%.1f" ,gcd(a, b));
    }
}
```

## 17.30 Fractions Product

Dados el numerador y el denominador de N fracciones, la tarea es encontrar el producto de N fracciones e imprimir la respuesta en forma reducida,

La idea es encontrar el producto de numeradores en una variable, como new\_num, ahora encontrar el producto de los denominadores en otra variable como new\_den.

Ahora para encontrar la respuesta en forma reducida, encontré el GCD de new\_num y new\_den y dividir el new\_num y new\_den por el GCD calculado.

La solución causa desbordamiento para números grandes, podemos evadir esto si encontramos los factores primos de todos los numeradores y denominadores, una vez hayamos encontrado los factores, podemos cancelar los factores primos comunes.

Cuando se solicita representar la respuesta de la forma {P \veces {Q} ^ {-1}} .Primero convierta el numerador y el denominador en forma reducible de P/Q. luego busque el multiplicativo inverso de Q con respecto a un número primo m (Generalmente 10^9 + 7) el cual es dado como pregunta, luego de encontrar el multiplicativo inverso de Q, multiplicarlo con P y tomar el modulo con el número primo m, el cual nos da nuestra salida requerida.

```
//Programa java que encuentra el producto
// de N fracciones en forma reducida

public class Fractionsproduct {
    // Función que retorna el gcd de a y b
    static int gcd(int a, int b) {
        if (a == 0) {
            return b;
        }
        return gcd(b % a, a);
    }

    static void productReduce(int n, int num[],
        int den[]) {
        int new_num = 1, new_den = 1;
        //Encontrando el producto de todos los N
        // numeradores y denominadores
        for (int i = 0; i < n; i++) {
            new_num *= num[i];
            new_den *= den[i];
        }
        // Encontrando GCD de nuevo numerados y denominador
        int GCD = gcd(new_num, new_den);
        // Convirtiendo en forma reducida
        new_num /= GCD;
        new_den /= GCD;
```

```

        System.out.println(new_num + "/" + new_den);
    }

    public static void main(String[] args) {
        int n = 3;
        int num[] = {1, 2, 5};
        int den[] = {2, 1, 6};
        productReduce(n, num, den);

    }
}

```

## 17.31 Josephus

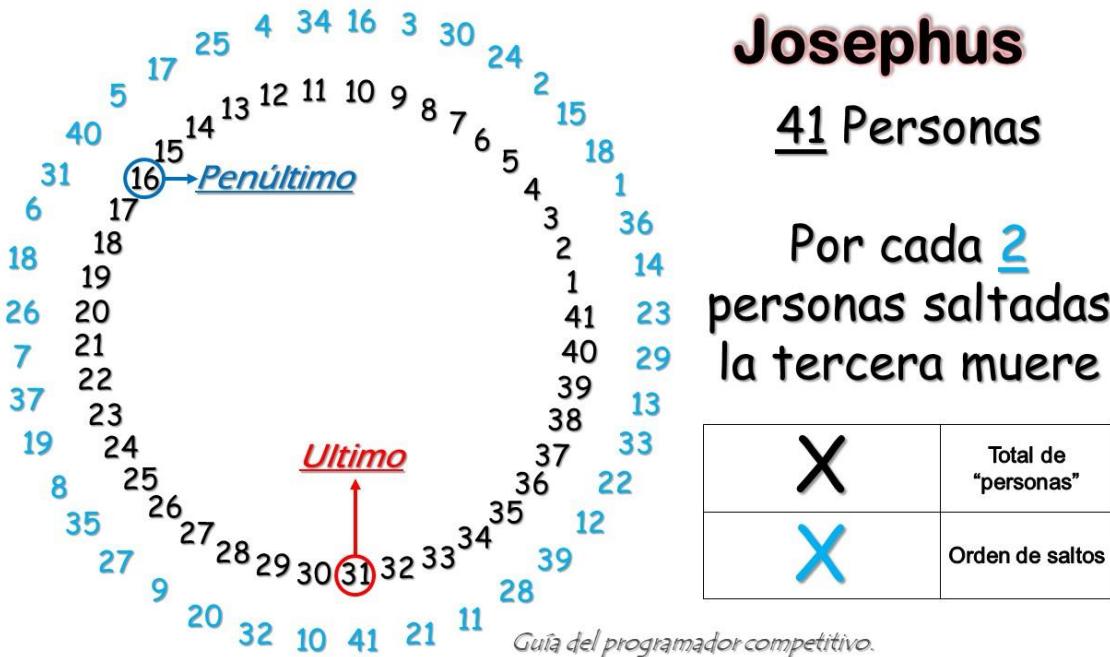


Ilustración 17-5 Flavio Josefo y sus compañeros soldados en el círculo

En las ciencias de la computación y las matemáticas, el problema Josephus (O permutación Josephus) es un problema teórico el cual su enunciado es el siguiente.

Hay  $n$  personas sentadas en círculo, esperando a ser ejecutadas. El conteo empieza en el mismo punto en el círculo y procede alrededor del círculo en dirección horaria, en cada paso un cierto número de personas son saltadas y la siguiente persona es ejecutada. La eliminación se realiza alrededor del círculo (el comienza a ser más pequeño y pequeño mientras la gente es ejecutada y removida), hasta que solo quede una persona, a quien se

le dará la libertad. Dado el total de personas  $n$  y un número  $k$  que indica que  $k-1$  personas serán saltadas y la kesima persona es asesinada en el círculo, la tarea es escoger el lugar en el círculo inicial de tal forma que sea la posición de la última persona que sobreviva.

Por ejemplo, si  $n = 5$  y  $k = 2$ , entonces la posición segura es 3. Primeramente la persona en la posición 2 es asesinada, luego la persona en la posición 4 es asesinada, luego la persona en la posición 1 es asesinada, finalmente la persona en la posición 5 es asesinada dejando a la persona en la 4 posición viva y con capacidad de disfrutar su libertad.

Si  $n=7$  y  $k=3$ , entonces la posición segura es 4, las personas en las posiciones 3,6,2,7,5,1 son asesinadas en ese orden y la 4 sobrevive.

```
//Implementación java de dos Algoritmos que realizan la
// busqueda josephus sabiendo número personas y tamaño
// de salto
public class Josephus {

    public static void main(String[] args) {
        System.out.println(josephus(6, 2));
        System.out.println(josephusModular(6, 2));
    }

    //Busqueda por algoritmo voraz
    static int josephus(int n, int k) {
        int d[] = new int[n + 1];
        d[0] = -1;
        for (int i = 1; i <= n; i++) {
            d[i] = i;
        }
        int i = n;
        int a = 0;
        while (i != 1) {
            i--;
            a = next(a, k, d);
            d[a] = -1;
            a++;
        }
        for (i = 0; d[i] == -1; i++);
        return d[i];
    }

    //Calcule la posición siguiente a caer
    static int next(int a, int k, int[] d) {
        int j = a - 1;
        for (int i = 0; i < k;) {
            j = (j + 1) % d.length;
            if (d[j] != -1) {
                i++;
            }
        }
        return j;
    }
}
```

```

        }
    //busqueda por matematica modular
    static int josephusModular(int n, int k) {
        int f = 0;
        for (int i = 1; i <= n; i++) {
            f = (f + k) % i;

        }
        return f + 1;
    }
}

```

## 17.32 Cardinal/Ordinal Numbers

Cuando usamos los números naturales para contar los elementos de un determinado conjunto los llamamos números cardinales. Imagina que tienes un conjunto de galletas como el que se muestra en la imagen de abajo. Si realizas el proceso de contar encontrarás que hay ocho galletas en total. Decimos entonces que el cardinal del conjunto es ocho, ya que este número representa la cantidad de elementos que tiene el conjunto.

En muchas ocasiones es necesario dar un orden a las cosas: las posiciones finales de una carrera o los pisos de un edificio son algunos ejemplos. Cuando usamos los números naturales para este ordenar los llamamos ordinales.

Para representar los números ordinales usamos los números naturales acompañados por una pequeña letra así: 1<sup>a</sup>, 2<sup>o</sup> etc. Cuando acompañamos el número por la letra a es para femenino, y con la letra o es para masculino. Así, si queremos decir que Anita es la número uno de la clase decimos que es la primera: 1<sup>a</sup>; y si queremos decir que Pablo ocupó el lugar número uno en la carrera decimos que fue el primero: 1<sup>o</sup>

```

//Programa java que convierte números ordinales
// en números cardinales en inglés
public class CardinalNumbers {

    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.println(i+cardinalNumber(i));
        }
    }

    static String cardinalNumber(int n) {
        if (n % 10 == 1 && n % 100 != 11) {

```

```

        return "st";
    }
    if (n % 10 == 2 && n % 100 != 12) {
        return "nd";
    }
    if (n % 10 == 3 && n % 100 != 13) {
        return "rd";
    }
    return "th";
}

```

## 17.33 Roman Numbers

### Números romanos

$1 = I$	$20 = XX$	$300 = CCC$	$4000 = M\bar{V}$
$2 = II$	$30 = XXX$	$400 = CD$	$5000 = \bar{V}$
$3 = III$	$40 = XL$	$500 = D$	$6000 = \bar{VI}$
$4 = IV$	$50 = L$	$600 = DC$	$7000 = \bar{VII}$
$5 = V$	$60 = LX$	$700 = DCC$	$8000 = \bar{VIII}$
$6 = VI$	$70 = LXX$	$800 = DCCC$	$9000 = \bar{IX}$
$7 = VII$	$80 = LXXX$	$900 = CM$	$10000 = \bar{X}$
$8 = VIII$	$90 = LC$	$1000 = M$	
$9 = IX$	$100 = C$	$2000 = MM$	
$10 = X$	$200 = CC$	$3000 = MMM$	

*Guía del programador competitivo.*

Ilustración 17-6 Principales componentes de los números Romanos

La numeración romana es un sistema de numeración que se desarrolló en la Antigua Roma y se utilizó en todo el Imperio romano, manteniéndose con posterioridad a su desaparición y todavía utilizado en algunos ámbitos.

Este sistema emplea algunas letras mayúsculas como símbolos para representar ciertos valores. Los números se escriben como combinaciones de letras. Por ejemplo, el año 2019 se escribe como MMXIX, donde cada M representa 1000 unidades, la X representa 10

unidades más y IX representa 9 unidades más (al ser X, que representa el 10, precedido por I, que representa el 1).

```
//Implementación java que convierte números decimales
// a números romanos y viceversa usando matemática modular
public class RomanNumbers {
    // Arrays constantes de letras romanas
    static String unit[] = {"", "I", "II", "III", "IV", "V", "VI", "VII",
    "VIII", "IX"};
    static String ten[] = {"", "X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX",
    "XC"};
    static String hnd[] = {"", "C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC",
    "CM"};
    public static void main(String[] args) {
        System.out.println(toRoman(122));
        System.out.println(toDecimal("CXXII"));
    }
    //Función que convierte a romano
    static String toRoman(int n) {
        //Los números romanos no tienen letras
        //más allá de los miles
        int a;
        StringBuilder sb = new StringBuilder();
        if (n >= 1000) {
            a = n / 1000;
            for (int i = 1; i <= a; i++) {
                sb.append("M");
            }
            n %= 1000;
        }
        //Letras de las centenas
        sb.append(hnd[n / 100]);
        n = n % 100;
        //Letras de las decenas
        sb.append(ten[n / 10]);
        //Letras de las unidades
        sb.append(unit[n % 10]);
        return sb.toString();
    }
    //Devuelve el valor de cada letra
    static int valor(char ch) {
        switch (ch) {
            case 'I':
                return 1;
            case 'V':
                return 5;
            case 'X':
                return 10;
            case 'L':
                return 50;
            case 'C':
                return 100;
            case 'D':
                return 500;
            case 'M':
                return 1000;
        }
    }
}
```

```

        return 500;
    case 'M':
return 1000;
}
return 0;
}
//Función que convierte a decimal
static int toDecimal(String num) {
    int sum = 0;
    int last = 0;
    int next;
//Toma cada letra y verifica si esta antes o despues
// suma si esta despues, resta si esta antes
for (int i = num.length() - 1; i >= 0; i--) {
    next = valor(num.charAt(i));
    if (last <= next) {
        sum += next;
    } else {
        sum -= next;
    }
    last = next;
}
return sum;
}
}

```

## 17.34 Multiple Great Common Divisor

Dado un array de números, encontrar GCD del array de elementos.

El GCD de tres o más números es igual que el producto de los factores primos comunes de todos los números, pero puede también ser calculado tomando repetidamente el GCD de los pares de números.

```

// Programa java que encuentra
// el maximo comun divisor de 2 o mas
//números
public class GCDMultiple {
// Funcion que retorna gcd de a y b
static int gcd(int a, int b)
{
    if (a == 0)
        return b;
    return gcd(b % a, a);
}
// Función que encuentra gcd de un arreglo de números
static int findGCD(int arr[], int n)
{
    int result = arr[0];

```

```

        for (int i = 1; i < n; i++)
            result = gcd(arr[i], result);

        return result;
    }

    public static void main(String[] args)
    {
        int arr[] = { 2, 4, 6, 8, 16 };
        int n = arr.length;
        System.out.println(findGCD(arr, n));
    }
}

```

## 17.35 Hardy-Ramanujan Theorem

El teorema de Hardy Ramanujan propone que el número de factores de  $n$  debe ser aproximadamente  $\log(\log(n))$  para la mayoría de números naturales  $n$ .

- 5192 tiene 2 factores primos distintos y  $\log(\log(5192)) = 2.1615$
- 51242183 tiene 3 factores primos distintos y  $\log(\log(51242183)) = 2.8765$

Este teorema es principalmente usado en algoritmos de aproximación y es prueba líder para conceptos más grandes de teoría de la probabilidad

```

// Programa java que cuenta todos los
// factores primos

public class HardyRamanujanTheorem {
    // Una función que cuenta factores primos de
    // un número n
    static int exactPrimeFactorCount(int n)
    {
        int count = 0;
        if (n % 2 == 0) {
            count++;
        while (n % 2 == 0)
            n = n / 2;
        }
        // n debe ser impar en este punto, así
        // podemos saltar un elemento (i=i+2)
        for (int i = 3; i <= Math.sqrt(n); i = i + 2)
        {
            if (n % i == 0) {
                count++;
                while (n % i == 0)
                    n = n / i;
            }
        }
    }
}

```

```

        }
        // Esta condición es para controlar el caso
        // cuando n es un factor primo más grande que 2
    if (n > 2)
        count++;
    return count;
}

public static void main (String[] args)
{
    int n = 51242183;
    System.out.println( "El número de diferentes "
        + " factores primos es "
        + exactPrimeFactorCount(n));
    System.out.println( "El valor de (log(n))"
        + " es " + Math.log(Math.log(n))) ;
}
}

```

## 17.36 Hoax Number

Dado un número  $n$ , verificar si es un número falso o no.

Un número falso es definido como un número compuesto, cuya suma de dígitos es igual a la suma de dígitos de sus distintos factores primos, 1 no es considerado un número primo, entonces no es incluido en la suma de dígitos de los distintos factores primos.

La definición de un número falso está cerca de la de la definición de un número de Smith, algunos de los números falsos son también números de Smith, es aparente que estos números falsos no tienen factores repetidos en su descomposición de primos.

Implementación

- 1) Primero generamos todos los distintos factores primos del número  $n$ .
- 2) Si el  $n$  no es un número primo, encuentre la suma de dígitos de los factores obtenidos en el paso 1
- 3) Encuentre la suma de dígitos de  $n$
- 4) Verifique si la suma obtenida en 2 y 3 son iguales o no.
- 5) Si las sumas son iguales, entonces  $n$  es un número falso.

```

// Programa java que verifica si un número es
// falso o no
import java.util.*;

```

```

public class HoaxNumber {
    /*Función que encuentra distintos factores primos
    dado un número n*/

    static List<Integer> primeFactors(int n)
    {
        List<Integer> res = new ArrayList<>();
        if (n % 2 == 0)
        {
            while (n % 2 == 0)
                n = n / 2;
        res.add(2);
        }
        /*N es impar en este punto
        desde que ya no sea divisible por 2
        entonces podemos probar solamente
        por números impares, mientras sean
        factores de n*/
        for (int i = 3; i <= Math.sqrt(n);
                i = i + 2)
        {
            //Verifica si i es factor primo
            if (n % i == 0)
            {
                while (n % i == 0)
                    n = n / i;
                res.add(i);
            }
        }
        /*Esra condición es para controlar
        el caso cuando n es un número primo
        más grande que 2*/
        if (n > 2)
        res.add(n);
        return res;
    }
    /*Función que calcula suma de digitos de distintos
     factores primos de dado n y la suma de digitos
     de número n. compara las sumás obtenidas*/
    static boolean isHoax(int n)
    {
        /*Distintos factores primos de n seran almacenados
         en vector pf*/
        List<Integer> pf = primeFactors(n);
        /* Si n es un número primo
         no puede ser un número falso*/
        if (pf.get(0) == n)
            return false;
        /*Encontrando suma de digitos de
         distintos factores primos de n*/
        int all_pf_sum = 0;
        for (int i = 0; i < pf.size(); i++)
        {
            // Encontramos la suma de digitos de
            // el actual factor primo pf[i]
    }

```

```

int pf_sum;
    for (pf_sum = 0; pf.get(i) > 0;
        pf_sum += pf.get(i) % 10,
        pf.set(i,pf.get(i) / 10));

    all_pf_sum += pf_sum;
}
// Encontrando suma de digitos de n
int sum_n;
    for (sum_n = 0; n > 0; sum_n += n % 10,
n /= 10);
// Comparando las dos sumas calculadas
return sum_n == all_pf_sum;
}

public static void main(String args[])
{
    int n = 84;
    if (isHoax(n))
System.out.print( "Un número falso\n");
    else
        System.out.print("No es número falso\n");
}
}

```

## 17.37 The Largest Pow That Divides a Factorial

Dados dos números, fact y n, encuentre la potencia más grande de n que divide fact! (Factorial de fact).

La idea es basada en la fórmula de Legendre la cual encuentra la potencia más grande de un número primo que divide fact!. Encontramos todos los factores primos de n. para cada factor primo encontramos la potencia más grande que divide fact! Y finalmente retornamos el mínimo de todas las potencias encontradas.

Si hay múltiples potencias de un factor primo presentes en n, entonces dividimos el conteo para obtener el valor de la máxima potencia de este factor.

```

/* Programa java que encuentra la potencia más larga de
un número (La cual puede ser compuesta) que divide
factorial*/
public class LargestPowDividesFactorial {

    /* For que encuentra la maxima potencia de número primo
       p que puede dividir un número factorial*/
    static int findPowerPrime(int fact, int p) {
        int res = 0;

```

```

        while (fact > 0) {
            res += fact / p;
        fact /= p;
    }

    return res;
}

// Retorna la suma de todos los factores de n
static int findPowerComposite(int fact, int n) {
    // Para almacenar el resultado ( Potencia minima de un
        // factor primo que divide fact)
    int res = Integer.MAX_VALUE;
    // Atravesar atraves de todos los factores
    // primos de n
    for (int i = 2; i <= Math.sqrt(n); i++) {
        // contador para contar la potencia del numero primo
        int count = 0;
        if (n % i == 0) {
            count++;
            n = n / i;
        }
        if (count > 0) {
            // Maxima potencia de i que divide
                // fact, dividimos por count para
                // manejar multiples ocurrencias de
            // un factor primo
            int curr_pow = findPowerPrime(fact, i) / count;
            res = Math.min(res, curr_pow);
        }
    }
    // Esta condición es para manejar
        // el caso cuando n es un número primo mayor
    // que 2
    if (n >= 2) {
        int curr_pow = findPowerPrime(fact, n);
        res = Math.min(res, curr_pow);
    }
    return res;
}

public static void main(String[] args) {
    int fact = 146, n = 5;
    System.out.println(findPowerComposite(fact, n));
}
}

```

## 17.38 Lower Common Multiplier in Array

Dado un array de n números, encontrar el LCM de ellos.

La idea es extender nuestra relación de más de dos números, vamos a decir que tenemos un array  $\text{arr}[]$  que contiene  $n$  elementos de los cuales necesitamos calcular su LCM.

Los pasos principales del algoritmo son:

- 1) Inicializa  $\text{ans} = \text{arr}[0]$
- 2) Iterar sobre todos los elementos del array, por ejemplo desde  $i=1$  a  $i= n-1$ , en la  $i$ esima iteración  $\text{ans} = \text{LCM}(\text{arr}[0], \text{arr}[1], \dots, \text{arr}[i-1])$ . Esto puede ser fácilmente hecho como  $\text{LCM}(\text{arr}[0], \text{arr}[1], \dots, \text{arr}[i]) = \text{LCM}(\text{ans}, \text{arr}[i])$ . Entonces en la  $i$ esima iteración tenemos que hacer  $\text{ans} = \text{LCM}(\text{ans}, \text{arr}[i]) = \text{ans} \times \text{arr}[i] / \text{gcd}(\text{ans}, \text{arr}[i])$

```
// Programa java que calcula el minimo comun multiplo
// de n elementos

public class Lcmofarrayelements {

    public static long lcm_of_array_elements(int[] element_array) {
        long lcm_of_array_elements = 1;
        int divisor = 2;
        while (true) {
            int counter = 0;
            boolean divisible = false;
            for (int i = 0; i < element_array.length; i++) {

                // lcm_of_array_elements (n1, n2, ... 0) = 0.
                // Para cada numero negativo lo convertimos
                // En positivo y calculamos lcm_of_array_elements.
                if (element_array[i] == 0) {
                    return 0;
                } else if (element_array[i] < 0) {
                    element_array[i] = element_array[i] * (-1);
                }
                if (element_array[i] == 1) {
                    counter++;
                }
            }
            /* Divide element_array por divisor si completa
            division */
            if (element_array[i] % divisor == 0) {
                divisible = true;
                element_array[i] = element_array[i] / divisor;
            }
        }
        /*Si el divisor es capaz de dividir completamente cualquier numero.
        de la matriz multiplicar con lcm_of_array_elements
        y almacenar en lcm_of_array_elements y continuar
        al mismo divisor para encontrar el siguiente factor.
        si no incrementar divisor*/
        if (divisible) {
            lcm_of_array_elements = lcm_of_array_elements * divisor;
        }
    }
}
```

```

    } else {
        divisor++;
    }
//Verifica si todo element_Array es 1 indicando
// encontramos todos los factores y terminamos el ciclo
if (counter == element_array.length) {
    return lcm_of_array_elements;
}
}

public static void main(String[] args) {
    int[] element_array = {2, 7, 3, 9, 4};
    System.out.println(lcm_of_array_elements(element_array));
}
}

```

## 17.39 Leonardo Number

Los números de Leonardo son una secuencia de números con la recurrencia:

- $L(0)=0$
- $L(1)=1$
- $L(n)=L(n-1)+L(n-2)+1$  si  $n>1$

Los primeros números de Leonardo son 1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753, 1219, 1973, 3193, 5167, 8361, …

Complejidad de tiempo: Exponencial

```

//Programa java que busca el nesimo número
// de Leonardo
public class LeonardoNumber {
    static int leonardo(int n)
    {
        if (n == 0 || n == 1)
            return 1;
        return (leonardo(n - 1) + leonardo(n - 2) + 1);
    }

    public static void main(String args[])
    {
        System.out.println(leonardo(3));
    }
}

```

## 17.40 Modular Exponentiation

Dados tres números x, y y p, calcule  $(x^y) \% p$ .

Bajo esta propiedad fundamental modular que es usada para computación eficiente, calcular la potencia usando matemática modular.

- $(ab) \bmod p = ((a \bmod p)(b \bmod p)) \bmod p$

Por ejemplo a = 50, b = 100, p = 13

- $50 \bmod 13 = 11$
- $100 \bmod 13 = 9$
- $(50 * 100) \bmod 13 = ((50 \bmod 13) * (100 \bmod 13)) \bmod 13$
- or  $(5000) \bmod 13 = (11 * 9) \bmod 13$
- or  $8 = 8$

Complejidad de tiempo: O(Log y).

```
// Programa iterativo que calcula
// potencia modular
public class ModularExponentiation {

    /* Función iterativa que calcula
        $(x^y) \% p$  in  $O(\log y)$  */
    static int power(int x, int y, int p)
    {
        // Inicializar resultado
        int res = 1;
        // Actualiza x si es más que
        // o igual a p
        x = x % p;
        while (y > 0)
        {
            // Si y es impar, multiplica x con res
            if((y & 1)==1)
                res = (res * x) % p;
        // y debe ser par ahora
            // y = y / 2
            y = y >> 1;
            x = (x * x) % p;
        }
        return res;
    }

    public static void main(String args[])
    {
        int x = 2;
        int y = 5;
        int p = 13;
        System.out.println("La potencia es " + power(x, y, p));
    }
}
```

```
    }  
}
```

## 17.41 Multiple Euler Totient

Función Totient de Euler de una entrada  $n$  es el conteo de números en  $\{1,2,3,\dots, n\}$  que sea primo relativo a  $n$ , por ejemplo, los números los cuales su GCD con  $n$  es 1.

En problemas donde tenemos que llamar a la función totient muchas veces como  $10^5$  veces, una solución simple puede retornar un TLE (time limit exceeded). La idea es usar la criba de Eratóstenes.

Encuentre todos los factores primos con límite en  $10^5$  usando la criba de Eratóstenes.

Para realizar este  $\Phi(n)$ , se hace lo siguiente.

- 1) Inicializa el resultado como  $n$ .
- 2) Itera a través de todos los primos más pequeños o iguales que la raíz cuadrada de  $n$ .  
Dejamos que el actual número primo sea  $p$ , revisamos si  $p$  divide  $n$ , si lo hace, removemos todas las ocurrencias de  $p$  de  $n$  dividiéndolo repetidamente por  $n$ , también reducimos nuestro resultado por  $n/p$ .
- 3) Finalmente retornamos nuestro resultado.

```
// Programa java que eficientemente calcula valores de  
// la formula totient de euler para multiples entradas  
  
import java.util.*;  
  
public class MultipleEulerTotient {  
  
    static int MAX = 100001;  
    // Almacena números primos arriba hasta MAX -1  
    static ArrayList<Integer> p = new ArrayList<Integer>();  
    // Encunrtra los números primos hasta MAX-1 Y  
    // los almacena en p  
  
    static void sieve() {  
        int[] isPrime = new int[MAX + 1];  
        for (int i = 2; i <= MAX; i++) {  
            // Si prime[i] no es marcado antes  
            if (isPrime[i] == 0) {  
                // Llena el vector para cada nuevo  
                // primo encontrado
```

```

        p.add(i);
    for (int j = 2; i * j <= MAX; j++) {
        isPrime[i * j] = 1;
    }
}
}

// Función que encuentra totient de n
static int phi(int n) {
int res = n;
// Este ciclo corre sqrt(n / ln(n)) veces
for (int i = 0; p.get(i) * p.get(i) <= n; i++) {
if (n % p.get(i) == 0) {
    // resta multiplos de p[i] de r
    res -= (res / p.get(i));
    // Remueve todas las ocurrencias de p[i] en n
while (n % p.get(i) == 0) {
n /= p.get(i);
}
}
}
// cuando n es un factor primo mayor
// que sqrt(n)
if (n > 1) {
    res -= (res / n);
}
return res;
}

public static void main(String[] args) {
//Preprocesa todos los primos hasta 10 ^ 5
sieve();
System.out.println(phi(11));
System.out.println(phi(21));
System.out.println(phi(31));
System.out.println(phi(41));
System.out.println(phi(51));
System.out.println(phi(61));
System.out.println(phi(91));
System.out.println(phi(101));
}
}

```

## 17.42 Natural Coprime Sumatory

Dado N y M, la tarea es encontrar cuales números de 1 a n pueden ser divididos en dos conjuntos los cuales su diferencia absoluta entre la suma de los dos sets es M y el GCD de la suma de los dos sets es 1.

Desde que tenemos 1 a N números, sabemos que la suma de todos los números es  $N*(N+1)/2$ . Dejamos S1 y S2 de esta manera:

$$1) \sum(S1) + \sum(S2) = N * (N + 1) / 2$$

$$2) \sum(S1) - \sum(S2) = M$$

Resolviendo estas dos ecuaciones podemos dar la suma de ambos conjuntos. Si  $\sum(S1)$  y  $\sum(S2)$  son enteros y ellos son coprimos (Su GCD es 1), entonces ahí existe una forma de separar el número en dos sets. De otra forma no hay forma de separar esos números N.

Complejidad de tiempo:  $O(\log(n))$

```
/* Código de Java para determinar si los números
1 a N se puede dividir en dos conjuntos
tal que la diferencia absoluta entre
la suma de estos dos conjuntos es M y estos
dos sumás son co-primos*/
public class NaturalCoprimeSum
{
    static int GCD (int a, int b)
    {
        return b == 0 ? a : GCD(b, a % b);
    }
    /*función que devuelve valor booleano
    sobre la base de si es posible
    dividir 1 a N números en dos conjuntos
    Que satisfacen las condiciones dadas.*/
    static boolean isSplittable(int n, int m)
    {
        // Inicializando suma total de 1
        //a n
        int total_sum = (n * (n + 1)) / 2;
        /*desde (1) total_sum = sum_s1 + sum_s2
        y (2) m = sum_s1 - sum_s2 asumiendo
        sum_s1 > sum_s2. resolviendo estas 2
        ecuaciones para obtener sum_s1 y sum_s2*/
        int sum_s1 = (total_sum + m) / 2;
        // total_sum = sum_s1 + sum_s2
        // y asi
        int sum_s2 = total_sum - sum_s1;
        /*Si la suma total es menor que la
        diferencia absoluta, no hay forma
        de que podamos dividir n números
        en dos conjuntos, así que devuelva falso*/
        if (total_sum < m)
            return false;
        /*Compruebe si estas dos sumás son
        enteros y se suman a
        suma total y también si su
```

```

        La diferencia absoluta es m.*/
if (sum_s1 + sum_s2 == total_sum &&
sum_s1 - sum_s2 == m)
// Ahora si las dos sumas son coprimos
    // Entonces retorna true, si no false
return (GCD(sum_s1, sum_s2) == 1);
/*si dos sumas no suman la suma total
   o si su diferencia absoluta.
   no es m, entonces no hay manera de
   dividir n numeros, por lo tanto retorna false*/
return false;
}

public static void main(String args[])
{
    int n = 5, m = 7;
    if (isSplittable(n, m))
        System.out.println("Si");
    else
        System.out.println("No");

}
}

```

## 17.43 Zeckendorf Theorem

El teorema de Zeckendorf indica que cada posible entero puede ser escrito como la suma de distintos números de Fibonacci no vecinos. Dos números Fibonacci son vecinos si uno viene luego del otro en la secuencia (0, 1, 1, 2, 3, 5, ..). Por ejemplo 3 y 5 son vecinos pero 2 y 5 no lo son.

Dado un número, encontrar la representación del número como la suma de números de Fibonacci no consecutivos.

- 1) Sea  $n$  el número de entrada
- 2) Mientras  $n \geq 0$ 
  - a) Encontrar el Fibonacci más grande que sea menor que  $n$ . Dejar que este número sea ' $f$ ', e imprimir  $f$ .
  - b)  $n = n - f$

```

/*Programa java para el Teorema de Zeckendorf,
encuentra la representación de n como suma de
números de fibonacci no vecinos*/
public class ZeckendorfTheorem {

```

```

        public static int nearestSmallerEqFib(int n)
        {
            // Casos base
            if (n == 0 || n == 1)
                return n;
            //Encuentra el mayor número fibonacci menor que n
            int f1 = 0, f2 = 1, f3 = 1;
            while (f3 <= n) {
                f1 = f2;
                f2 = f3;
                f3 = f1 + f2;
            }
            return f2;
        }
        // Imprime representación de fibonacci
        public static void printFibRepresentation(int n)
        {
            while (n > 0) {
                // Encuentra el mayor número fibonacci menor
                // o igual que n
                int f = nearestSmallerEqFib(n);
                // Imprime el número fibonacci encontrado
                System.out.print(f + " ");
                // Reduce n
                n = n - f;
            }
        }

        public static void main(String[] args)
        {
            int n = 30;
            System.out.println("Representación de fibonacci no vecinos "
                    + "de " + n + " es");
            printFibRepresentation(n);
        }
    }
}

```

## 17.44 Rosser Theorem

El teorema de Rosser dicta que el nésimo número es mayor que el producto de  $n$  y el logaritmo natural de  $n$  para todos los  $n$  mayores a 1.

Matemáticamente:

Para  $n \geq 1$ , si  $p_n$  es el nésimo número primo, entonces then:

- $p_n > n * (\ln n)$

Para  $n = 1$ , enésimo número primo = 2

-  $2 > 1 * \ln(1)$

Para  $n = 2$ , enésimo número primo = 3

-  $3 > 2 * \ln(2)$

Para  $n = 3$ , enésimo número primo = 5

-  $5 > 3 * \ln(3)$

Para  $n = 4$ , enésimo número primo = 7

-  $7 > 4 * \ln(4)$

Para  $n = 5$ , enésimo número primo = 11

-  $11 > 5 * \ln(5)$

Para  $n = 6$ , enésimo número primo = 13

-  $13 > 6 * \ln(6)$

```
// Programa java que verifica el Teorema de Rosser
import java.util.*;
public class RosserTheorem
{
    static ArrayList<Integer> prime=new ArrayList<Integer>();
    // Criba de Eratostenes
    static void sieve()
    {
        int n = 10000;
        boolean []isprime=new boolean[n+2];
        for(int i=0;i<n;i++)
            isprime[i]=true;
        isprime[0]=false;
        isprime[1] =false;
        for (int i = 2; i <= n; i++) {
            if (isprime[i]) {
                for (int j = i * i; j <= n; j += i)
                    isprime[j] =false;
            }
        }
        //Almacena primos en prime[]
        for (int i = 0; i <= n; i++)
            if (isprime[i])
                prime.add(i);
    }

    // Verifica el TEOREMA DE ROSSER para todos los números
}
```

```

// Menores a n
    static void verifyRosser(int n)
{
    System.out.println("TEOREMA DE ROSSER: nesimo número primo > n * (ln
n)");

    for (int i = 0; i < n; i++)
        if (prime.get(i) > (i + 1) * Math.log(i + 1)) {
            System.out.println( "para n = " + (i+1)
+ ", nesimo número primo = "
+ prime.get(i) + "\n\t"
+ prime.get(i) + " > " + (i + 1)
+ " * ln(" + (i + 1) + ")");
        }
    }

    public static void main(String [] args)
{
    sieve();
    verifyRosser(20);
}
}

```

## 17.45 Sieve Of Eratosthenes O(n)

La clásica criba de Eratóstenes toma  $O(N \log(\log N))$  para encontrar todos los números primos menores a  $N$ , este código es una versión modificada de esta criba que tiene una complejidad de tiempo de  $O(N)$ .

```

/*Programa java que genera todos los números primos
menores a N en O(N) Eratostenes Optimizado*/
import java.util.ArrayList;

public class SieveofEratosthenesOptimized
{
    static final int MAX_SIZE = 1000001;
    // isPrime[] : isPrime[i] es true si el número es primo
    // prime[] : Almacena todos los números primos menores a N
    // SPF[] Almacena los factores primos más pequeños de un número
    // [Por ejemplo : factor primo más pequeño que '8' y '16' es
    // '2' entonces nosotros ponemos SPF[8]=2 , SPF[16]=2 ]
    static ArrayList<Boolean>isprime = new ArrayList<>(MAX_SIZE);
    static ArrayList<Integer>prime = new ArrayList<>();
    static ArrayList<Integer>SPF = new ArrayList<>(MAX_SIZE);
    // Metodo que genera todos los factores primos menores de N
    static void manipulated_seive(int N)
    {
        // 0 y 1 no son primos
        isprime.set(0, false);
        isprime.set(1, false);

        // llena el resto de las entradas

```

```

for (int i=2; i<N ; i++)
{
    // si isPrime[i] == True entonces i es
        // número primo
    if (isprime.get(i))
    {
        // pone i dentro de prime[]
        prime.add(i);
        // un número primo es su propio factor primo
            // más pequeño
            SPF.set(i,i);
    }
    /*Eliminar todos los múltiplos de i * prime [j] que son
       no primos haciendo isPrime [i * prime [j]] = false
       y ponga el factor primo más pequeño de i * Prime [j] como prime [j]
       [Por ejemplo: dejemos i = 5, j = 0, prime[j] = 2 [i * prime [j] =
           10]
       por lo que el factor primo más pequeño de '10' es '2' que es prime
           [j]]
       este bucle se ejecuta solo una vez para el número que no es primo*/
}

for (int j=0;
        j < prime.size() &&
        i*prime.get(j) < N && prime.get(j) <= SPF.get(i);
        j++)
{
    isprime.set(i*prime.get(j),false);
    // Pone el factor primo más pequeño de i*prime[j]
    SPF.set(i*prime.get(j),prime.get(j)) ;
}
}

public static void main(String args[])
{
    int N = 13 ; //Debe ser menor que MAX_SIZE
    // inicializando isprime y SPF
    for (int i = 0; i < MAX_SIZE; i++){
        isprime.add(true);
        SPF.add(2);
    }
    manipulated_seive(N);
    //Imprima todos los números primos menores que n
    for (int i=0; i<prime.size() && prime.get(i) <= N ; i++)
        System.out.print(prime.get(i) + " ");
}
}

```

## 17.46 Smith Number

Dado un número  $n$ , la tarea encontrar si un número es número de Smith o no, un número de Smith es un número compuesto cuya suma de dígitos es igual a la suma de los dígitos en su factorización prima.

- $n = 4$
- Factorización prima = 2, 2 and  $2 + 2 = 4$
- Por lo tanto, 4 es un número de Smith

La idea es primero encontrar todos los factores primos por debajo de un límite usando la criba de Sundaram. (Esto es útil para buscar y verificar varios números de Smith). Ahora por cada entrada que será verificada como número de Smith, atravesamos por todos los factores primos en él, y encontramos la suma de los dígitos en cada factor primo. También buscamos la suma de los dígitos en el número dado. Finalmente comparamos las dos sumas, si son lo mismo, retornamos true.

```
// Programa java que verifica si un número es
// número de Smith o no
import java.util.ArrayList;

public class SmithNumber
{

    static int MAX  = 10000;
    //Array que almacena todos los primos menores o iguales
    // a 10^6
    static ArrayList <Integer> primes = new ArrayList<>();
    //Función de la criba de Sundaram
    static void sieveSundaram()
    {
        /*En general criva de Sundaram, produce primos más pequeños.
         * que  $(2 * x + 2)$  para un número dado el número  $x$ . Ya que
         * Queremos primos más pequeños que MAX, reducimos MAX a la mitad
         * Esta matriz se usa para separar números del forma
         *  $i + j + 2ij$  de otros donde  $1 \leq i \leq j$ /
        boolean marked[] = new boolean[MAX/2 + 100];
        //Lógica principal de Sundaram, marcar todos los números
        // loc cuales no generan número primo haciendo  $2*i+1$ 
        for (int i=1; i<=(Math.sqrt(MAX)-1)/2; i++)
            for (int j=(i*(i+1))<<1; j<=MAX/2; j=j+2*i+1)
                marked[j] = true;

        // 2s es número primo
        primes.add(2);
        // Imprime los otros primos, primos restantes son de la
        //forma  $2*i+1$  de tal manera que  $marked[i]$  es falso
```

```

for (int i=1; i<=MAX/2; i++)
    if (marked[i] == false)
        primes.add(2*i + 1);
}
// Retorna true si n es un número de Smith, si no falso
static boolean isSmith(int n)
{
    int original_no = n;
    // Encuentra la suma de los dígitos de los factores
    // primos de n
    int pDigitSum = 0;
    for (int i = 0; primes.get(i) <= n/2; i++)
    {
        while (n % primes.get(i) == 0)
    {   // Si primes[i] es un factor primo
        // agrega sus dígitos a pDigitSum
        int p = primes.get(i);
        n = n/p;
        while (p > 0)
        {
            pDigitSum += (p % 10);
            p = p/10;
        }
    }
    /*Si n!=1 entonces un primo sigue para ser sumado*/
    if (n != 1 && n != original_no)
    {
        while (n > 0)
    {
        pDigitSum = pDigitSum + n%10;
        n = n/10;
    }
    }
    // Todos los factores primos sumados
    // Ahora suma los dígitos del número original
    int sumDigits = 0;
    while (original_no > 0)
    {
        sumDigits = sumDigits + original_no % 10;
        original_no = original_no/10;
    }
    // Si la suma de los dígitos en factores primos
    // y la suma de dígitos en el número original son los mismos
    // entonces true, si no false
    return (pDigitSum == sumDigits);
}

public static void main(String[] args)
{   //Encuentra todos los números primos antes del límite
    // estos números son usados para encontrar factores primos
    sieveSundaram();
    System.out.println("Imprimiendo primeros números de Smith" +
" usando isSmith()");
}

```

```

        for (int i=1; i<500; i++)
            if (isSmith(i))
                System.out.print(i + " ");
    }
}

```

## 17.47 Sphenic Number

Un número escénico es un entero positivo el cual es el producto de exactamente tres primos distintos, los primeros números esfénicos son 30, 42, 66, 70, 78, 102, 105, 110, 114,...

Dado un número n, determine si es un número esfenico o no.

Un número esfenico puede ser verificado generando los últimos factores primos de los números hasta n.

Luego podemos simplemente dividir el número por sus factores primos y luego ese número por sus factores primos, y así en Adelante, y luego verificar si el número tiene exactamente 3 factores primos distintos.

Complejidad de tiempo: O (nlog(n))

```

// Programa JAVA que verifica si un número
// es esfenico o no

import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.Set;
public class SphenicNumber {

    static final int MAX = 1000;
    //Crea un vector donde almacenar primos
    // inicializa todas las entradas en 0
    static ArrayList<Integer> least_pf = new ArrayList<>(MAX);
    /* Esta función llena valores en least_pf[]
       de tal modo que el valor de least_pf[] almacene
       el factor primo más pequeño de i
       Se encuentra basado en la criba
       de Eratostenes*/
    public static void main(String[] args) {
        for (int i = 0; i < 1001; i++) {
            least_pf.add(0);
        }
        Collections.fill(least_pf, 0);
        leastPrimeFactor(MAX);
        for (int i = 1; i < 100; i++) {
            if (isSphenic(i)) {
                System.out.println(i + " ");
            }
        }
    }
    private static void leastPrimeFactor(int n) {
        for (int i = 2; i < n; i++) {
            if (least_pf.get(i) == 0) {
                for (int j = i; j < n; j += i) {
                    least_pf.set(j, i);
                }
            }
        }
    }
    private static boolean isSphenic(int n) {
        if (n < 1) {
            return false;
        }
        int count = 0;
        int current_prime = least_pf.get(1);
        int current_exponent = 1;
        while (n > 1) {
            if (current_prime * current_exponent > n) {
                break;
            }
            if (n % current_prime == 0) {
                count++;
                current_exponent++;
            } else {
                current_prime = least_pf.get(n);
                current_exponent = 1;
            }
        }
        return count == 3;
    }
}

```

```

        }
    }

}

/*Función que verifica si un número es esfenico*/
static boolean isSphenic(int n) {
    /*Almacena tres factores primos de n
     *tenemos al menos 3 elementos en s*/
    Set<Integer> s = new HashSet<>();
    // Sigue encontrando factores primos hasta que n sea 1
    while (n > 1) {
        // Encuentra al menos un factor del actual valor de n
        int lpf = least_pf.get(n);
        // Almacenamos actual tamaño de s para verificar si
        // algun factor primo se repite
        int init_size = s.size();
        // Inserta un valor primo al actual valor de n
        s.add(lpf);
        // si lpf repite un número o se pasa de 3, retorna falso.
        if (s.size() == init_size || s.size() > 3) // El mismo primo divide
        // un número más de una vez
        {
            return false;
        }

        // dividir n en lpf
        n /= lpf;
    }
    // True si el tamaño es 3
    return (s.size() == 3);
}

static void leastPrimeFactor(int n) {
    // un factor primo de 1 es 1
    least_pf.set(1, 1);
    // Almacena números primos para todos
    // los otros números
    for (int i = 2; i <= n; i++) {
        // least_pf[i] == 0 significa que i es primo
        if (least_pf.get(i) == 0) {
            least_pf.set(i, i);
            for (int j = 2 * i; j <= n; j += i) {
                if (least_pf.get(j) == 0) {
                    least_pf.set(j, i);
                }
            }
        }
    }
}

```

## 17.48 Catalan Numbers

En combinatoria, los números de Catalan forman una secuencia de números naturales que aparece en varios problemas de conteo que habitualmente son recursivos. Obtienen su nombre del matemático belga Eugène Charles Catalan (1814–1894).

El  $n$ -ésimo número de Catalan se obtiene, aplicando coeficientes binomiales, a partir de la siguiente fórmula:

$$C_n = \frac{(2n)!}{(n+1)! n!}$$

```
//Programa Java que recursivamente encuentra el
// nesimo número Catalan
public class CatalanNumber {

    public static int catalan(int n) {
        int res = 0;
        // Caso base
        if (n <= 1) {
            return 1;
        }
        for (int i = 0; i < n; i++) {
            res += catalan(i) * catalan(n - i - 1);
        }
        return res;
    }

    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            System.out.print(catalan(i) + " ");
        }
    }
}
```

## 17.49 Charmichael Numbers

Un número  $n$  es un número de Carmichael si satisface la siguiente condición aritmética modular:

- $\text{pow}(b, n-1) \bmod n = 1$ ,

Para todos los  $b$  en rango de 1 a  $n$  tal que  $b$  y  $n$  son relativos primos  $\text{gcd}(b, n) = 1$ .

Dado un entero positivo  $n$ , encontrar si es un número de Carmichael, estos números tienen importancia en el método de Fermat para el test de primalidad.

```
//Programa java que verifica si un número
```

```

// es un número de carmichael
public class CarmichaelNumbers {

    //función que busca el GCD de dos números
    static int gcd(int a, int b) {
        if (a < b) {
            return gcd(b, a);
        }
        if (a % b == 0) {
            return b;
        }
        return gcd(b, a % b);
    }

    //Función que busca el pow(x,y)
    // bajo un modulo mod
    static int power(int x, int y, int mod) {
        if (y == 0) {
            return 1;
        }
        int temp = power(x, y / 2, mod) % mod;
        temp = (temp * temp) % mod;
        if (y % 2 == 1) {
            temp = (temp * x) % mod;
        }
        return temp;
    }

    //Función que verifica si un número es número de charmichael
    static int isCarmichaelNumber(int n) {
        for (int b = 2; b < n; b++) {
            // Si 'b' es primo relativo de n
            if (gcd(b, n) == 1) // y pow(b, n-1)%n no es 1,
            // retorne falso
            {
                if (power(b, n - 1, n) != 1) {
                    return 0;
                }
            }
        }
        return 1;
    }

    public static void main(String args[]) {
        System.out.println(isCarmichaelNumber(500));
        System.out.println(isCarmichaelNumber(561));
        System.out.println(isCarmichaelNumber(1105));
    }
}

```

## 17.50 Stern-Brocot Sequence

La secuencia de Stern Brocot es similar la secuencia de Fibonacci, pero es diferente en la forma en que la secuencia es generada.

- 1) Primero y segundo elemento de la secuencia es 1 y 1.
- 2) Considere el segundo miembro de la secuencia, luego sume el miembro considerado de la secuencia y su predecesor, por ejemplo  $(1+1=2)$  ahora 2 es el siguiente elemento de nuestra serie, la secuencia será [1,1,2].
- 3) Después de este elemento, nuestro siguiente elemento en la secuencia será considerado el elemento en nuestro segundo paso, ahora la secuencia seria [1,1,2,1].
- 4) De nuevo realizamos el paso 2, pero ahora consideramos el elemento 2 (Tercer elemento), entonces nuestro siguiente número de la secuencia será la suma de os números considerados, y su predecesor( $2+1=3$ ), la secuencia ahora será [1,1,2,1,3]
- 5) Como en el paso 3, el siguiente elemento será considerado, por ejemplo 2, la secuencia será [1,1,2,1,3,2]
- 6) El proceso continua, nuestro elemento considerado será 1(Cuarto elemento).

```
// Programa java que imprime
// La secuencia de Stern Brocot
import java.util.*;

public class SternBrocotSequence {

    static void SternSequenceFunc(ArrayList<Integer>
        BrocotSequence, int n)
    {
        // Ciclo que crea la secuencia
        for (int i = 1; BrocotSequence.size() < n; i++)
        {
            int considered_element = BrocotSequence.get(i);
            int precedent = BrocotSequence.get(i-1);
            //Agregando la suma de los elementos considerados
            // y son precedentes
            BrocotSequence.add(considered_element + precedent);
        // Agregando siguiente elemento considerado
            BrocotSequence.add(considered_element);
        }
        // Imprimiendo secuencia
        for (int i = 0; i < 15; ++i)
            System.out.print(BrocotSequence.get(i) + " ");
    }

    public static void main (String[] args) {
```

```

        int n = 15;
        ArrayList<Integer> BrocotSequence = new ArrayList<Integer>();
// Agregando primer y segundo valor
// A la secuencia
BrocotSequence.add(1);
        BrocotSequence.add(1);
        SternSequenceFunc(BrocotSequence, n);
}
}

```

## 17.51 Sylvester Sequence

El en sistema numérico, La secuencia de Sylvester es una secuencia de enteros la cual cada miembro es un producto de los números previos, más uno, dado un entero positivo N, imprima los primeros N miembros de la secuencia.

Los números pueden ser muy largos, se usa %10^9 + 7.

La idea es correr un ciclo y tomar dos variables, inicializarlos como 1 y 2, uno para almacenar el producto hasta ahora, y el otro para almacenar el número actual el cual es el primer número +1 y por cada paso, multiplicar ambos usando aritmética modular, por ejemplo (a + b)%N = (a%N + b%N)%N donde N es un número modular.

```

//Implementación java de la secuencia Sylvester
public class SylvesterSequence {

    public static void printSequence(int n)
{
    int a = 1; // Para almacenar el producto
    int ans = 2; // Para almacenar el número actual
int N = 1000000007;
    //Ciclo hasta n
    for (int i = 1; i <= n; i++) {
        System.out.print(ans + " ");
        ans = ((a % N) * (ans % N)) % N;
        a = ans;
        ans = (ans + 1) % N;
    }
}

    public static void main(String[] args)
{
    int n = 6;
    printSequence(n);

}
}

```

## 17.52 Aliquot Sequence

Dado un número  $n$ , la tarea es imprimir su secuencia de Aliquot. La secuencia de Aliquot de un número empieza con el mismo, términos restantes de la secuencia son la suma de los divisores propios del término inmediatamente anterior.

Por ejemplo, la secuencia de Aliquot de 10, es 10,8,7,1,0. La secuencia puede repetir. Por ejemplo para 6, tenemos una secuencia infinita de todos los 6, En esos casos imprimimos el número repetido y paramos.

- Números los cuales tienen una secuencia de Aliquot repetitiva de longitud 1 son llamados números perfectos. Por ejemplo 6, suma de sus divisores propios es 6.
- Números que tienen una secuencia de Aliquot repetitiva de tamaño 2 son llamados números de Amicable. Por ejemplo 220 es un número de Amicable.
- Números que tienen secuencia de Aliquot repetitiva de tamaño 3 son llamados números sociales

Podemos generar la secuencia primero imprimiendo el número  $n$  y luego calculando los siguientes términos usando la suma de los divisores propios. Cuando computemos el siguiente término, verificamos si nosotros ya habíamos visto ese término o no, si el término aparece de nuevo tenemos una secuencia repetida, imprimimos el mismo y rompemos el ciclo.

```
//Implementación java de una apoximación
// de la secuencia Aliquot
import java.util.*;

public class AliquotSequence
{
    // Función que calcula suma de
    // los divisores adecuados
    static int getSum(int n)
    {
        int sum = 0; // 1 es un divisor adecuado
        for (int i = 1; i <= Math.sqrt(n); i++)
        {
            if (n % i == 0)
```

```

        {
            // Si divisores son iguales, solo toma uno
            if (n / i == i)
            {
                sum = sum + i;
            }
            else // Si no tome ambos
            {
                sum = sum + i;
                sum = sum + (n / i);
            }
        }

// Calcula la suma de los divisores adecuados
return sum - n;
}

static void printAliquot(int n)
{
    // Imprime el primer termino
    System.out.printf("%d ", n);
    TreeSet<Integer> s = new TreeSet<>();
    s.add(n);

    int next = 0;
    while (n > 0)
    {
        // Calcula siguiente termino desde el anterior
        n = getSum(n);
        if (s.contains(n) && n != s.last())
        {
            System.out.print("\nRepeats with " + n);
            break;
        }
        //Imprimir siguiente termino
        System.out.print(n + " ");
        s.add(n);
    }
}

public static void main(String[] args)
{
    printAliquot(12);
}
}

```

## 17.53 Juggler Sequence

La secuencia de Juggles es una serie de números enteros la cual su primer término comienza con un entero positivo y los términos restantes son generados de los anteriores inmediatos números.

La secuencia Juggler comenzando con el número 3: 5, 11, 36, 6, 2, 1

La secuencia Juggler comenzando desde el número 9: 9, 27, 140, 11, 36, 6, 2, 1

Dado un número n tenemos que imprimir la secuencia Juggler de este número como el primer número de la secuencia

- Los términos en la secuencia de Juggler primero crecen hasta un valor pico, y luego empiezan a decrecer.
- El último término de la secuencia de Juggler es siempre 1.

```
// Implementación java de la secuencia de Juggler
public class JugglerSequence {
    static void printJuggler(int n) {
        int a = n;
        // Imprime el primer termino
        System.out.print(a + " ");
        // Calcula terminos hasta que el ultimo no sea 1
        while (a != 1) {
            int b = 0;
            // Verifica si los previos terminos son pares o impares
            if (a % 2 == 0) // calcular siguiente termino
            {
                b = (int) Math.floor(Math.sqrt(a));
            } else // para impar anterior, calcular
            // Siguiente termino
            {
                b = (int) Math.floor(Math.sqrt(a)
                    * Math.sqrt(a) * Math.sqrt(a));
            }
            System.out.print(b + " ");
            a = b;
        }
    }

    public static void main(String[] args) {
        printJuggler(3);
        System.out.println();
        printJuggler(9);
    }
}
```

## 17.54 Moser de Bruijn Sequence

Dado un entero n, imprima los primeros n términos de la secuencia de Moser de Bruijn.

La secuencia de Moser de Bruijn es una secuencia obtenida adicionando las distintas potencias del número 4, por ejemplo 1,4,16,6..

Debe notarse aquí que cualquier número el cual es la suma de potencias de 4 no distintas no son parte de la secuencia, por ejemplo 8 no es parte de la secuencia debido a que es formado de la suma de no distintas potencias de 4, que son 4 y 4.

Por lo tanto cualquier número el cual no sea una potencia de 4 y está presente en la secuencia debe ser la suma de distintas potencias de 4.

Por ejemplo, 21 es parte de la secuencia, incluso a través de que no es una potencia de 4 porque es la suma de distintas potencias de 4, que son 1,4 y 16.

```
// Codigo java que genera los primeros n terminos
// de la secuencia de Moser-de Bruijn
```

```
public class MoserdeBruijnSequence {

    public static int gen(int n) {

        // S(0) = 0
        if (n == 0) {
            return 0;
        } // S(1) = 1
        else if (n == 1) {
            return 1;
        } // S(2 * n) = 4 * S(n)
        else if (n % 2 == 0) {
            return 4 * gen(n / 2);
        } // S(2 * n + 1) = 4 * S(n) + 1
        else if (n % 2 == 1) {
            return 4 * gen(n / 2) + 1;
        }
        return 0;
    }

    public static void moserDeBruijn(int n) {
        for (int i = 0; i < n; i++) {
            System.out.print(gen(i) + " ");
        }
        System.out.println();
    }

    public static void main(String args[]) {
        int n = 15;
        System.out.println("Primeros " + n
            + " terminos de la secuencia de "
            + "Moser-de Bruijn : ");
        moserDeBruijn(n);
    }
}
```

```
    }  
}
```

## 17.55 Newman-Conway Sequence

La secuencia de Newman-Conway es aquella que genera la siguiente secuencia de enteros:

1 1 2 2 3 4 4 5 6 7 7...

En términos matemáticos, la secuencia  $P(n)$  de Newman-Conway es definida por la siguiente relación de recurrencia:

- $P(n) = P(P(n - 1)) + P(n - P(n - 1))$  with seed values  $P(1) = 1$  and  $P(2) = 1$

Dado un número  $n$ , imprima el enésimo número de la secuencia Newman-Conway

Complejidad de tiempo:  $O(n)$

```
// Programa java que encuentra el nesimo  
// elemento de la secuencia de Newman-Conway  
public class NewmanConwaySequence {  
    static int sequence(int n)  
    {  
        if (n == 1 || n == 2)  
            return 1;  
        else  
            return sequence(sequence(n - 1))  
                + sequence(n - sequence(n - 1));  
    }  
    static intsequenceDP(intn)  
    {  
        intf[n + 1];  
        inti;  
        f[0] = 0;  
        f[1] = 1;  
        f[2] = 1;  
  
        for(i = 3; i <= n; i++)  
            f[i] = f[f[i - 1]] + f[i - f[i - 1]];  
  
        returnf[n];  
    }  
  
    public static void main(String args[])
```

```

{
    int n = 10;
    System.out.println(sequence(n));
}
}

```

## 17.56 Padovan Sequence

La secuencia de Padovan es similar a la secuencia de Fibonacci, con una secuencia recursiva similar cuya fórmula es:

- $P(n) = P(n-2) + P(n-3)$
- $P(0) = P(1) = P(2) = 1$

Secuencia Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55.....

Espiral de cuadrados con largo de lados los cuales siguen la secuencia de Fibonacci,

Secuencia de Padovan: 1, 1, 1, 2, 2, 3, 4, 5, 7, 9, 12, 16, 21, 28, 37,.....

Espiral de triángulos equiláteros con largos de lado que siguen la secuencia de Padovan.

```

// Programa JAVA que encuentra el enesimo termino
// de la secuencia de Padovan
// usando programación dinámica
public class PadovanSequence {
/* Función que calcula el número de Padovan*/
static int pad(int n)
{ //0,1 y 2 número de la serie es 1
int pPrevPrev = 1, pPrev = 1,
        pCurr = 1, pNext = 1;
    for (int i = 3; i <= n; i++) {
        pNext = pPrevPrev + pPrev;
        pPrevPrev = pPrev;
        pPrev = pCurr;
        pCurr = pNext;
    }
    return pNext;
}

public static void main(String args[])
{
    int n = 12;
    System.out.println(pad(n));
}
}

```

## 17.57 Recaman Sequence

Dado un entero  $n$ , imprima los primeros  $n$  elementos de la secuencia de Recaman

Es básicamente una función con dominio y co dominio como números naturales y 0, su recursividad se define de la siguiente manera.

Especificamente, dejamos  $a(n)$  denotar el  $(n+1)$ simo termino (0 ya está ahí).

La regla dice:

- $a(0) = 0,$
- Si  $n > 0$  y el número no está incluido en la secuencia
- $a(n) = a(n - 1) - n$
- si no
- $a(n) = a(n-1) + n.$

Complejidad de tiempo:  $O(n^2)$

```
// Programa java que imprime el nesimo numero
// de la secuencia de Recaman
public class RecamanSequence {

    static void recaman(int n)
    {   //Crea un array que almacenara los terminos
        int arr[] = new int[n];
        // Primer termino de la secuencia es siempre 0
        arr[0] = 0;
        System.out.print(arr[0]+",");
        // Llena terminos restantes usando formula recursiva
        for (int i = 1; i < n; i++)
        {
            int curr = arr[i - 1] - i;
            int j;
            for (j = 0; j < i; j++)
            {   // si arr[i-1] - i es negativo or ya existe
                if ((arr[j] == curr) || curr < 0)
                {
                    curr = arr[i - 1] + i;
                    break;
                }
            }
            arr[i] = curr;
            System.out.print(arr[i]+", ");
        }
    }
}
```

```
    public static void main (String[] args)
{
    int n = 17;
    recaman(n);
}
}
```

## 17.58 Problemas de repaso

### Ejercicios en Online Judge

440-Eeny Meeny Moo

10539- Almost Prime Numbers

355-The Bases Are Loaded

686-Goldbach's Conjecture (II)

498-Polly the Polynomial

847-A Multiplication Game

550- Multiplying by Rotation

10299-Relatives

919- Cutting Polyominones

10407-Simple division

### Ejercicios en CodeChef

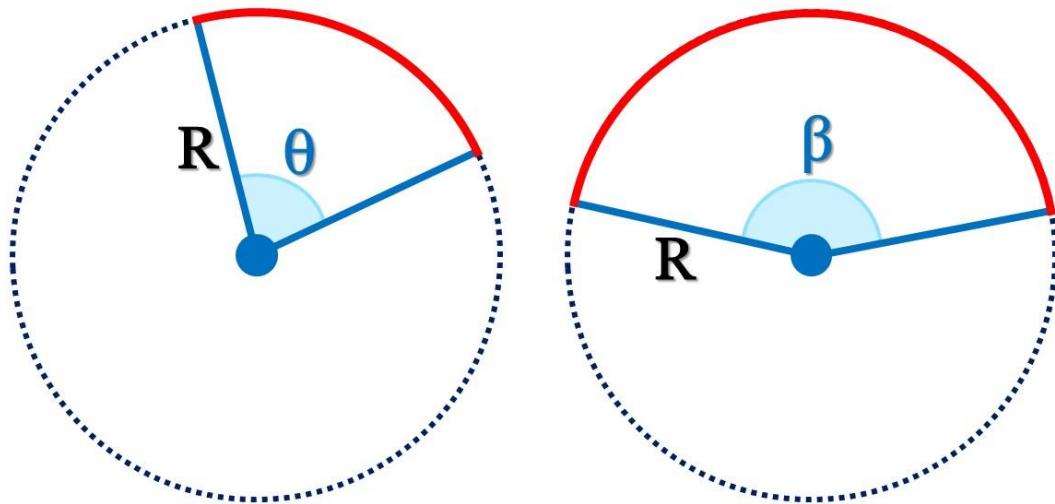
FCTRLZ

RPD

# 18 Geometría

## 18.1 Arc Length

### Longitud de Arco



*Guía del programador competitivo.*

Ilustración 18-1 Arco dentro de un círculo de radio R

Un ángulo es una figura geométrica cuando dos rayos se encuentran en un punto en un plano, esos rayos forman los lados del ángulo, y el punto de encuentro es referido como el vértice del ángulo. Hay que tener en cuenta que el plano que forma un ángulo no tiene que ser un plano Euclíadiano, ahora en un círculo el largo de un arco es una porción de la circunferencia.

Dado un ángulo y el diámetro de un círculo, podemos calcular el largo de un arco usando la fórmula:

- ArcLength =  $(2 * \pi * \text{radio}) * (\text{ángulo} / 360)$
- Donde  $\pi = 22/7$ ,
- Diámetro =  $2 * \text{radio}$ ,

- El ángulo está en grados.

Si el ángulo es mayor o igual a 360 grados, entonces el largo del arco no puede ser calculado desde que ningún ángulo es posible.

```
//Programa java que calcula
//la longitud de un arco

public class ArcLength {

    static double arcLength(double diameter,
                           double angle) {
        double pi = 22.0 / 7.0;
        double arc;
        if (angle >= 360) {
            System.out.println("Angulo no puede ser formado");
            return 0;
        } else {
            arc = (pi * diameter) * (angle / 360.0);
            return arc;
        }
    }

    public static void main(String args[]) {
        double diameter = 25.0;
        double angle = 45.0;
        double arc_len = arcLength(diameter, angle);
        System.out.println(arc_len);
    }
}
```

## 18.2 Area Of A Circular Sector

Un sector circular o un sector círculo, es la porción de un disco encerrado por dos radios y un arco, donde el área más pequeña es conocida como el sector menor y el grande como el sector mayor.

- Sector = $(\pi \cdot r^2) \cdot (\text{Angulo}/360)$

El área de un sector es similar al cálculo del área de un círculo, solo se le multiplica el área de un círculo con el ángulo del sector.

```
// Programa java que encuentra el area de un sector circular

public class AreaCircularSector
{
```

```

static void SectorArea(double radius,double angle)
{
    if(angle >= 360)
        System.out.println("Angle not possible");

// Calculando el area
    else
    {
        double sector =((22 * radius * radius) / 7)
                    * (angle / 360);
        System.out.println(sector);
    }
}

public static void main (String[] args)
{
    double radius = 9;
    double angle = 60;
    SectorArea(radius, angle);
}
}

```

## 18.3 Arranged Coin Triangle

Tenemos N monedas las cuales necesitamos ordenar en forma de triángulo, por ejemplo la primera fila podrá tener una moneda, la segunda fila dos monedas y así en adelante, necesitamos saber la máxima altura que podemos obtener usando esas N monedas.

Este problema puede ser resuelto encontrando la relación entre la altura del triángulo y el número de monedas, dejamos como la altura máxima como H, luego la suma total de monedas debe ser menos de N.

```

// Programa java que encuentra la maxima altura
// de un triángulo de monedas arregladas
public class ArrangedCoinTriangle {
/*Retorna el la raiz cuadrada de n
   Note como la función lo realiza*/
static float squareRoot(float n) {
/*Usamos n como aproximación inicial*/
    float x = n;
    float y = 1;
// e decide el nivel de precisión
    float e = 0.000001f;
    while (x - y > e) {
        x = (x + y) / 2;
        y = n / x;
    }
    return x;
}

```

```

        }
    //Metodo que encuentra la maxima altura
    //del arreglo de monedas
    static int findMaximumHeight(int N) {
    //Calculando la porción interna
        //de la raiz cuadrada
    int n = 1 + 8 * N;
        int maxH = (int) (-1 + squareRoot(n)) / 2;

        return maxH;
    }
    public static void main(String[] args) {
        int N = 12;

        System.out.print(findMaximumHeight(N));
    }
}

```

## 18.4 Circle Area

El área del círculo es igual a pi por el radio al cuadrado.

$$A=\pi r^2$$

```

//Programa java que encuentra el area de un circulo
public class CircleArea
{
    static final double PI = Math.PI;
    static double findArea(int r)
    {
        return PI * Math.pow(r, 2);
    }
    public static void main(String[] args)
    {
        System.out.println("Area is " + findArea(5));
    }
}

```

## 18.5 Circumscribed Circle Of Equilateral Triangle

Dado el largo de los lados de un triángulo equilátero, necesitamos encontrar el área de un circuncírculo del triángulo dado. Todos los lados del triángulo equilátero son de igual largo, y todos los ángulos interiores son de 60 grados.

Las propiedades de un circuncírculo son las siguientes:

- El centro del circuncírculo es el punto donde las medianas del triángulo equilátero se intersectan.
- El círculo circuncírculo es un triángulo equilátero es hecho a través de los tres vértices de un triángulo equilátero.
- El radio de un circuncírculo de un triángulo equilátero es igual a  $(a/\sqrt{3})$ , donde 'a' es el largo de los lados del triángulo equilátero

La fórmula usada para calcular el área de un círculo circuncírculo es:

- $(\pi*a^2)/3$

Donde a es el largo del lado del triángulo dado.

Sabemos que el área de un círculo es  $\pi*r^2$ , donde r es el radio del círculo dado.

También sabemos que el radio de un circuncírculo de un triángulo equilátero = (Lado del triángulo/ $\sqrt{3}$ ).

Por lo tanto, área =  $\pi*r^2 = \pi*a^2/3$ .

```
// código Java para encontrar el área de
// un círculo circunscrito a un triángulo equilátero
public class CircumscribedCircleOfEquilateral{

    static double PI = 3.14159265;
    //Función que encuentra el área
    // del círculo circunscrito
    public static double area_circumscribed(double a)
    {
        return (a * a * (PI / 3));
    }
    public static void main(String[] args)
    {
        double a = 6.0;
        System.out.println("Area of circumscribed circle is :"
+ area_circumscribed(a));
    }
}
```

## 18.6 Convex Hull

Se define como la envolvente convexa, envoltura convexa o cápsula convexa de un conjunto de puntos X de dimensión n como la intersección de todos los conjuntos convexos que contienen a X.

En el caso particular de puntos en un plano, si no todos los puntos están alineados, entonces su envolvente convexa corresponde a un polígono convexo cuyos vértices son algunos de los puntos del conjunto inicial de puntos.

Una forma intuitiva de ver la envolvente convexa de un conjunto de puntos en el plano, es imaginar una banda elástica estirada que los encierra a todos. Cuando se libere la banda elástica tomará la forma de la envolvente convexa.

```
// Programa java que halla el casco convexo de un set de puntos
import java.util.*;

public class ConvexHull {

    static class Point {

        int x, y;

        Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }

    // Para encontrar la orientación de la tripleta
    // la función retorna lo siguiente
    // 0 --> p, q y r son colineares
    // 1 --> Manecillas de reloj
    // 2 --> en contra de las manecillas
    static int orientation(Point p, Point q, Point r) {
        int val = (q.y - p.y) * (r.x - q.x)
                - (q.x - p.x) * (r.y - q.y);

        if (val == 0) {
            return 0; // colinear
        }
        return (val > 0) ? 1 : 2; // Manecillas
    }

    // Imprime el casco convexo del set de puntos
    static void convexHull(Point points[], int n) {
        // Debe haber al menos 3 puntos
        if (n < 3) {
            return;
        }

        // Inicializar resultado
        ArrayList<Point> hull = new ArrayList<>();
        // Encuentra el punto de más a la izquierda
```

```

int l = 0;
    for (int i = 1; i < n; i++) {
        if (points[i].x < points[l].x) {
l = i;
        }
    }
    // Comienza desde el punto más a la izquierda, sigue moviéndose.
    // en sentido antihorario hasta llegar al punto de inicio
    // otra vez. Este ciclo corre O (h) veces donde h es
    // número de puntos en resultado o salida
    int p = l, q;
    do {
// Agrega el punto actual al resultado
        hull.add(points[p]);
// Buscar un punto 'q' tal que
        // la orientación (p, x, q) es antihorario
        // para todos los puntos 'x'. La idea es mantener
        // pista de los últimos visitados más contra reloj
        // punto en q. Si algún punto 'i' es más
        // en sentido contrario a las agujas del reloj que q, luego
        //actualiza q.
q = (p + 1) % n;
        for (int i = 0; i < n; i++) {
//Si i es más antihorario que el actual q, actualice q
if (orientation(points[p], points[i], points[q])
== 2) {
            q = i;
        }
}
//Ahora q es el más antihorario con respecto
// a p, ubica p como q para la siguiente iteración
// asi q es agregado al casco resultado
p = q;
} while (p != l);
// Mientras no vengamos del primer punto
// Imprima resultado
for (Point temp : hull) {
    System.out.println("(" + temp.x + ", "
                      + temp.y + ")");
}
public static void main(String[] args) {

    Point points[] = new Point[7];
    points[0] = new Point(0, 3);
    points[1] = new Point(2, 3);
    points[2] = new Point(1, 1);
    points[3] = new Point(2, 1);
    points[4] = new Point(3, 0);
    points[5] = new Point(0, 0);
    points[6] = new Point(3, 3);
    int n = points.length;
    convexHull(points, n);
}
}

```

## 18.7 Cuts Sumatory

Dado el número de cortes, encuentre el máximo número de posibles piezas.

Este problema no es más que el problema del cartero flojo, y tiene la siguiente formula.

$$\text{Máximo número de piezas} = 1 + n * (n+1) / 2$$

```
//Programa que calcula el maximo número
// de piezas de pizza dados el número de cortes

public class CutsSumatory {

    static int findMaximumPieces(int n) {
        return 1 + n * (n + 1) / 2;
    }

    public static void main(String arg[]) {
        System.out.print(findMaximumPieces(3));
    }
}
```

## 18.8 Heron Formula

En geometría plana elemental la fórmula de Herón, cuya invención se atribuye al matemático griego Herón de Alejandría, da el área de un triángulo conociendo las longitudes de sus tres lados a, b y c:

$$Area = \sqrt{s(s - a)(s - b)(s - c)}$$

```
//Programa Java que calcula el area
// De un triángulo usando la formula de Herón

public class HeronFormula {

    public static void main(String[] args) {
        //Enviar las coordenadas de los vertices
        System.out.println(heron(1, 0, -1, 0, 0, 2));
    }

    //función que usa la formula de Herón
    static double heron(double x1, double y1, double x2,
                        double y2, double x3, double y3) {
        double a = Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
        double b = Math.sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3));
        double c = Math.sqrt((x3 - x2) * (x3 - x2) + (y3 - y2) * (y3 - y2));
    }
}
```

```

        double s = (a + b + c) / 2.0;
        double A = Math.sqrt(s * (s - a) * (s - b) * (s - c));
    return A;
    }

}

```

## 18.9 Three Point Circle Radious

```

//Programa JAVA que usa la formula de heron
//Para calcular el radio de un circulo dado por
//tres puntos (Circumradio)

public class RadiousCircleThreePoints {

    public static void main(String[] args) {
        //Se envian las coordenadas de los
        // vertices del triángulo
        System.out.println(circumradius(1, 0, -1, 0, 0, 2));
    }
    //Función que devuelve el tamaño del radio
    static double circumradius(double x1, double y1, double x2,
                               double y2, double x3, double y3) {
        double a = Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
        double b = Math.sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3));
        double c = Math.sqrt((x3 - x2) * (x3 - x2) + (y3 - y2) * (y3 - y2));
        double A = heron(x1, y1, x2, y2, x3, y3);
        double r = (a * b * c) / (4.0 * A);
        return r;
    }
    //Calcula el area con la formula de Herón
    static double heron(double x1, double y1, double x2,
                        double y2, double x3, double y3) {
        double a = Math.sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2));
        double b = Math.sqrt((x1 - x3) * (x1 - x3) + (y1 - y3) * (y1 - y3));
        double c = Math.sqrt((x3 - x2) * (x3 - x2) + (y3 - y2) * (y3 - y2));
        double s = (a + b + c) / 2.0;
        double A = Math.sqrt(s * (s - a) * (s - b) * (s - c));
        return A;
    }
}

```

## 18.10 Dimensions Right Angled Triangle

Se llama triángulo o trígono, en geometría plana, al polígonal de tres lados. Los puntos comunes a cada par de lados se denominan vértices del triángulo.

Un triángulo tiene tres ángulos interiores, tres pares congruentes de ángulos exteriores, tres lados y tres vértices entre otros elementos.

```
// Código java que encuentra las dimensiones
// de un triángulo por medio de altura y área
public class DimensionsRightAngledTriangle {

    // Función que calcula la dimensión
    static void findDimen(int H, int A)
    {

        // P^2+B^2 = H^2
        // P*B = 2*A
        // (P+B)^2 = P^2+B^2+2*P*B = H^2+4*A
        // (P-B)^2 = P^2+B^2-2*P*B = H^2-4*A
        // P+B = sqrt(H^2+4*A)
        // |P-B| = sqrt(H^2-4*A)
        if (H * H < 4 * A) {
            System.out.println("Not Possible");
        }
        return;
    }

    // Raíz cuadrada valor de H^2 + 4A y H^2 - 4A
    double apb = Math.sqrt(H * H + 4 * A);
    double asb = Math.sqrt(H * H - 4 * A);

    System.out.println("P = " + Math.round(((apb - asb) / 2.0) * 100.0) /
100.0);
    System.out.print("B = " + Math.round(((apb + asb) / 2.0) * 100.0) /
100.0);
}

public static void main(String[] args)
{
    int H = 5;
    int A = 6;
    findDimen(H, A);
}
```

## 18.11 Hexagon Area

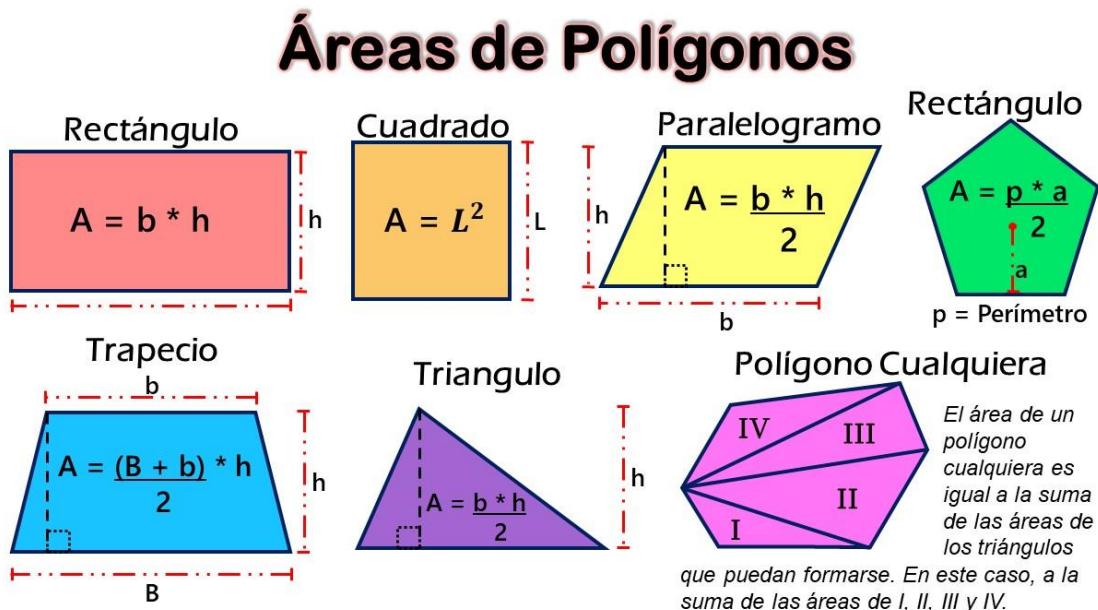
Un hexágono es una figura geométrica de 6 lados, en dos dimensiones, el total de la suma de los ángulos internos de cualquier hexágono es  $720^\circ$ . Un hexágono regular tiene 6 simetrías rotacionales, y 6 simetrías reflectivas, todos los ángulos internos son de  $120$  grados.

Aquí hay principalmente 6 triángulos equiláteros de lado  $n$  y el área de un triángulo equilátero es  $\sqrt{3}/4 * n * n$ . Desde el hexágono, hay en total 6 triángulos equiláteros con lado  $n$ , el área del hexágono se convierte en  $(3 * \sqrt{3}/2) * n * n$

```
//Programa Java que calcula el area de un Hexagono
public class HexagonArea {
    public static double hexagonArea(double s) {
        return ((3 * Math.sqrt(3)
            * (s * s)) / 2);
    }

    public static void main(String[] args) {
        // Largo de un lado
        double s = 4;
        System.out.print("Area: "
            + hexagonArea(s));
        System.out.println("");
    }
}
```

## 18.12 Area Of Polygon



Guía del programador competitivo.

Ilustración 18-2 Áreas de diferentes polígonos

Dadas ordenadamente las coordenadas de un polígono, con n vértices, encontrar el área del polígono. Aquí ordenado significa que las coordenadas son dadas en sentido horario o anti horario, desde el primer vértice hasta el último.

Podemos dividir un polígono en triángulos, la fórmula del área es derivada de tomar cada camino AB, y calcular el área del triángulo ABO, con un vértice de origen O, tomando el producto cruz (El cual da el área de un paralelogramo) y dividiendo por 2. Mientras pasemos alrededor del polígono, estos triángulos con área positiva o negativa se sobreponen, y las áreas en medio del origen y el polígono pueden ser canceladas y sumadas a 0, mientras solo haya área interna, el triángulo de referencia de mantiene.

```
//Programa Java que calcula el area de un poligono
import java.awt.Point;
import java.awt.Polygon;
import java.util.Arrays;

public class AreaOfPolygon {

    public static void main(String[] args) {
        //Crea el poligono con los puntos dados
        Polygon p = new Polygon();
        p.addPoint(0, 2);
        p.addPoint(2, 2);
        p.addPoint(2, 0);
        p.addPoint(0, 0);
        System.out.println(area(p));
    }
    //Por medio de triángulos va calculando el area completa
    static int signedTriangleArea(Point a, Point b, Point c) {
        return a.x * b.y - a.y * b.x + a.y * c.x - a.x * c.y + b.x * c.y - c.x *
        b.y;
    }

    static boolean ccw(Point a, Point b, Point c) {
        return signedTriangleArea(a, b, c) > 0;
    }
    // Verifica si los puntos son colineares
    static boolean collinear(Point a, Point b, Point c) {
        return signedTriangleArea(a, b, c) == 0;
    }
    //Calcula la distancia entre dos puntos
    static double distance(Point p1, Point p2) {
        double dx = p1.x - p2.x;
        double dy = p1.y - p2.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
    //Función que calcula el area de cualquier poligono
```

```

static double area(Polygon poly) {
    int N = poly.npoints;
    int[] x = poly.xpoints;
    int[] y = poly.ypoints;
    Point[] p = new Point[N];
    for (int i = 0; i < N; ++i) {
        p[i] = new Point(x[i], y[i]);
    }
    final Point first;
    int min = 0;
    for (int i = 1; i < N; i++) {
        if (p[i].y < p[min].y) {
            min = i;
        } else if (p[i].y == p[min].y) {
            if (p[i].x < p[min].x) {
                min = i;
            }
        }
    }
    first = p[min];
    p[min] = p[0];
    p[0] = first;
    //Usar expresion lambda en vez de comparador
    Arrays.sort(p, 1, N, (Point p1, Point p2) -> {
        if (collinear(first, p1, p2)) {
            return Double.compare(distance(first, p1), distance(first, p2));
        }
        if (ccw(first, p1, p2)) {
            return -1;
        } else {
            return 1;
        }
    });
    double A = 0.0;
    for (int i = 0; i < N; i++) {
        int j = (i + 1) % N;
        A += p[i].x * p[j].y - p[j].x * p[i].y;
    }
    return A / 2.0;
}
}

```

## 18.13 Line to Line Intersection

Dados puntos A y B, correspondientes a la línea AB y los puntos P y Q, correspondientes a la línea PQ, encuentre el punto de intersección de estas líneas. Los puntos están dados en un plano 2D con sus coordenadas X y Y.

Primero que todo, asumimos que tenemos dos puntos  $(x_1, y_1)$  y  $(x_2, y_2)$ . Ahora encontramos la ecuación de la línea formada por esos puntos.

Dejaremos que las líneas dadas sean:

- $a_1x + b_1y = c_1$
- $a_2x + b_2y = c_2$

Tenemos que ahora resolver estas dos ecuaciones para encontrar el punto de intersección, para resolver esto, multiplicamos  $a_1$  por  $b_2$  y  $a_2$  por  $b_1$ , esto nos da:

- $a_1b_2x + b_1b_2y = c_1b_2$
- $a_2b_1x + b_2b_1y = c_2b_1$

Restando esto obtenemos:

- $(a_1b_2 - a_2b_1)x = c_1b_2 - c_2b_1$

Esto nos da el valor de  $x$ . similarmente podemos encontrar el valor de  $y$ ,  $(x, y)$  nos da el punto de intersección.

Esto nos da el punto de intersección de dos líneas, pero si nos dan segmentos de línea en vez de líneas, tenemos que revisar el punto que computado yace en ambos segmentos de líneas,

Si el segmento de línea es especificado por los puntos  $(x_1, y_1)$  y  $(x_2, y_2)$ , entonces debemos verificar si  $(x, y)$  está en el segmento que tenemos de la siguiente manera:

- $\min(x_1, x_2) \leq x \leq \max(x_1, x_2)$
- $\min(y_1, y_2) \leq y \leq \max(y_1, y_2)$

```
//Implementación Java que encuentra el punto de
//intersección de dos líneas

public class LineLineIntersection {
    //Clase usada para almacenar las coordenadas X y la Y
    // de un punto respectivo

    static class Point {

        double x, y;

        public Point(double x, double y) {
```

```

        this.x = x;
        this.y = y;
    }

    // Metodo usado para imprimir las cordenadas
    // X y Y de un punto
    static void displayPoint(Point p) {
        System.out.println("(" + p.x + ", " + p.y + ")");
    }

    static Point lineLineIntersection(Point A, Point B, Point C, Point D) {
        // Linea AB representada como a1x + b1y=c1
        double a1 = B.y - A.y;
        double b1 = A.x - B.x;
        double c1 = a1 * (A.x) + b1 * (A.y);
        // Linea CD representada como a2x + b2y=c2
        double a2 = D.y - C.y;
        double b2 = C.x - D.x;
        double c2 = a2 * (C.x) + b2 * (C.y);
        double determinant = a1 * b2 - a2 * b1;
        if (determinant == 0) {
            //Las lineas son paralelas, esto es simplificado
            //Retornando un par de FLT_MAX
            return new Point(Double.MAX_VALUE, Double.MAX_VALUE);
        } else {
            double x = (b2 * c1 - b1 * c2) / determinant;
            double y = (a1 * c2 - a2 * c1) / determinant;
            return new Point(x, y);
        }
    }

    public static void main(String args[]) {
        Point A = new Point(1, 1);
        Point B = new Point(4, 4);
        Point C = new Point(1, 8);
        Point D = new Point(2, 4);
        Point intersection = lineLineIntersection(A, B, C, D);
        if (intersection.x == Double.MAX_VALUE
        && intersection.y == Double.MAX_VALUE) {
            System.out.println("la linea AB y CD son paralelas.");
        } else {
            System.out.print("La intersección de las lineas AB "
            + "y CD es: ");
            Point.displayPoint(intersection);
        }
    }
}

```

## 18.14 Line Mid Point

Dado dos coordenadas de una línea iniciando en  $(x_1, y_1)$  y terminando en  $(x_2, y_2)$  encontrar el punto medio de una línea.

El punto medio de dos puntos  $(x_1, y_1)$  y  $(x_2, y_2)$  es el punto M encontrado con la siguiente fórmula:

- $M = ((x_1+x_2)/2, (y_1+y_2)/2)$

```
//Programa java para buscar
//el punto medio de una linea
public class LineMidPoint
{
    static void midpoint(int x1, int x2,
                         int y1, int y2)
    {
        System.out.print((x1 + x2) / 2 +
                        " , " + (y1 + y2) / 2);
    }
    public static void main (String[] args)
    {
        int x1 = -1, y1 = 2 ;
        int x2 = 3, y2 = -6 ;
        midpoint(x1, x2, y1, y2);
    }
}
```

## 18.15 Line Through Two Points

Dados dos puntos P( $x_1, y_1$ ) y Q( $x_2, y_2$ ), encuentre la ecuación de la línea formado por esos dos puntos.

Cualquier línea puede ser representada como:

- $ax + by = c$

Dejamos dos puntos que satisfagan la línea dada, entonces tenemos:

- $ax_1 + by_1 = c$
- $ax_2 + by_2 = c$

Podemos cambiar los siguientes valores para que la ecuación de mantenga verdadera:

- $a = y_2 - y_1$

- $b = x_1 - x_2$
- $c = ax_1 + by_1$

```

//Implementación Java para encontrar la linea
//que pasa atraves de dos puntos
public class LineThroughTwoPoints {

    public static void main(String[] args) {
        Pair P = new Pair(3, 2);
        Pair Q = new Pair(2, 6);
        lineFromPoints(P, Q);
    }

    static void lineFromPoints(Pair P, Pair Q) {
        double a = Q.second - P.second;
        double b = P.first - Q.first;
        double c = a * (P.first) + b * (P.second);

        if (b < 0) {
            System.out.println("La linea que pasa a traves de los puntos P y Q
es: "
                + a + "x " + b + "y = " + c);

        } else {
            System.out.println("La linea que pasa a traves de los puntos P y Q
es: "
                + a + "x + " + b + "y = " + c);
        }
    }

    /* Este par es usado para almacenar la X y Y
     de un punto respectivamente/
    static class Pair {

        int first;
        int second;

        public Pair(int first, int second) {
            this.first = first;
            this.second = second;
        }

    }
}

```

## 18.16 Minimum Height Triangle

Dados dos números a y b, encontrar la altura más pequeña de un triángulo de área ‘a’ y base ‘b’.

La altura mínima de un triángulo con base 'b' y área 'a' puede ser evaluada teniendo el conocimiento de relación entre las tres.

La relación entre el área, la base y la altura es:

$$- \text{ área} = (1/2) * \text{base} * \text{altura}$$

Entonces la altura puede ser calculada como:

$$- \text{height} = (2 * \text{área}) / \text{base}$$

La altura mínima es el techo de la altura obtenida usando la fórmula de arriba.

```
//Código java que buscar la altura mínima  
//Codigo java que buscar la altura minima  
//usando la base y el area
```

```
public class MinimumHeightTriangle {  
  
    static double minHeight(double base, double area) {  
        double d = (2 * area) / base;  
        return Math.ceil(d);  
    }  
  
    public static void main(String[] args) {  
        double base = 4, area = 8;  
        System.out.println("Minimum height is "  
+ minHeight(base, area));  
    }  
}
```

## 18.17 Perimeter Using Blocks

Tenemos n bloques de tamaño 1x1, necesitamos encontrar el mínimo perímetro de una matriz hecha de estos bloques.

Tomemos un ejemplo para ver el patrón, tenemos 4 bloques, las siguientes son las diferentes posibilidades:

```
+---+---+---+  
| | | | | Perímetro = 10  
+---+---+---+  
  
+---+---+
```

| | | | Perímetro = 10

+---+---+

| |

+---+

+---+---+

| | | | Perímetro = 8

+---+---+

| | |

+---+---+

Si hacemos algunos ejemplos usando lápiz y papen, podemos notar que el perímetro se vuelve mínimo cuando la figura formada es más cercana a un cuadrado. La razón de esto es, que queremos el máximo de lados de bloques que miren dentro de la figura, entonces el perímetro de la figura se vuelve mínimo.

Si el número de bloques es un cuadrado perfecto, entonces el perímetro puede ser simplemente  $4 * \sqrt{n}$ .

Pero si el número de bloques no es una raíz cuadrada perfecta, entonces nosotros calculamos el número de filas y columnas cercanas a la raíz cuadrada, luego de arreglar los bloques en un rectángulo, y tenemos bloques restantes, simplemente podemos agregar 2 al perímetro porque solo 2 lados extra faltarían.

```
/*Código JAVA que permire encontrar el minimo
perímetro usando n bloques*/
public class PerimeterUsingBlocks {

    public static long minPerimeter(int n) {
        int l = (int) Math.sqrt(n);
        int sq = l * l;
        //si n es un cuadrado perfeto
        if (sq == n) {
            return l * 4;
        } else {
            //Número de filas
            long row = n / l;
            //Perímetro de la matriz rectangular
            long perimeter
                = 2 * (l + row);
            // Si hay bloques restantes
        }
    }
}
```

```

        if (n % 1 != 0) {
            perimeter += 2;
        }
        return perimeter;
    }
}

public static void main(String[] args) {
    int n = 10;
    System.out.println(minPerimeter(n));
}
}

```

## 18.18 Point Inside Triangle

Dados tres puntos esquina de un triángulo, y un punto más P, verifique si P yace dentro del triángulo o no.

Dejaremos las coordenadas de las tres esquinas ser  $(x_1, y_1)$ ,  $(x_2, y_2)$  y  $(x_3, y_3)$ , y las coordenadas de P ser  $(x, y)$ .

- 1) Calcular área del triángulo dado, por ejemplo el área del triángulo ACB: Area A =  $[x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)]/2$
- 2) Calcular el área del triángulo PAB. Podemos usar la misma fórmula para esto, dejamos esta área ser A1.
- 3) Calculamos el área del triángulo PBC, dejamos esta área ser A2.
- 4) Calcular el área del triángulo PAC, dejaremos esta área ser A3.
- 5) Si P yace dentro del triángulo, entonces  $A_1 + A_2 + A_3$  debe ser igual a A.

```

//Código Java que verifica cuando un punto
//yace dentro de un triángulo o no
public class PointInsideTriangle {
    /* Una función de utilidad que calcula el
       área del triángulo formado por (x1, y1) (x2, y2) y (x3, y3)*/
    static double area(int x1, int y1, int x2, int y2,
                      int x3, int y3)
    {
        return Math.abs((x1*(y2-y3) + x2*(y3-y1)+x3*(y1-y2))/2.0);
    }
    /* Una función que verifica cu cualquier punto P(x,y)
       yace dentro de un triángulo formado por (x1, y1),
       B(x2, y2) and C(x3, y3) */
    static boolean isInside(int x1, int y1, int x2,
                           int y2, int x3, int y3, int x, int y)

```

```

{
    /* Calcula el area del triángulo ABC */
    double A = area (x1, y1, x2, y2, x3, y3);

    /* Calcula el area del triángulo PCB */
    double A1 = area (x, y, x2, y2, x3, y3);

    /* Calcula el area del triángulo PAC */
    double A2 = area (x1, y1, x, y, x3, y3);

    /* Calcula el area del triángulo PAB */
    double A3 = area (x1, y1, x2, y2, x, y);
    /* verifica si la suma de A1,A2 y A3 es igual a A*/
    return (A == A1 + A2 + A3);
}

    public static void main(String[] args)
{   /* Verificamos si el punto P(10,15)
       yace dentro del triángulo formado por
       A(0, 0), B(20, 0) and C(10, 30)*/
    if (isInside(0, 0, 20, 0, 10, 30, 10, 15))
        System.out.println("Adentro");
    else
        System.out.println("Por fuera");
}
}

```

## 18.19 Ratio Divide Line

Dadas dos coordenadas  $(x_1, y_1)$  y  $(x_2, y_2)$ , y m y n, encuentre las coordenadas que dividen la línea juntando  $(x_1, y_1)$  y  $(x_2, y_2)$  en el ratio m:n.

La fórmula de la sección nos dice las coordenadas del punto que divide una línea segmento dada en dos partes las cuales tendrán la longitud del ratio m:n.

$$- \quad ((mx_2+nx_1/m+n),(my_2+ny_1/m+n))$$

```

// Programa java que encuentra el punto que divide una
// linea dada, dado un ratio
public class RatioDivideLine {
    static void section(double x1, double x2,
                       double y1, double y2,
                       double m, double n)
    {
        //Aplicando la formula de la sección
        double x = ((n * x1) + (m * x2)) /
        (m + n);
        double y = ((n * y1) + (m * y2)) /
        (m + n);
        // Imprimiendo resultado
    }
}

```

```

        System.out.println("(" + x + ", " + y + ")");
    }

    public static void main(String[] args)
    {
        double x1 = 2, x2 = 4, y1 = 4,
               y2 = 6, m = 2, n = 3;
        section(x1, x2, y1, y2, m, n);
    }
}

```

## 18.20 Rectangles In NxM

Tenemos una matriz N\*M, imprima el número de rectángulos en él.

- Si la matriz es 1x1, entonces habrá 1 rectángulo.
- Si la matriz es 2x1, entonces habrá  $2+1=3$  rectángulos
- Si la matriz es 3x1, entonces habrá  $3+2+1=6$  rectángulo.

Podemos decir que para N\*1 ahí habrá  $N+(N+1)+(N-2)\dots+1=(N)(N+1)/2$  rectángulos.

Si nosotros añadimos una columna más a Nx1, primero tendríamos tantos rectángulos en la segunda columna como en la primera, y luego tendríamos el mismo número de 2xM rectángulos, entonces  $Nx2=3(N)(N+1)/2$ , luego de deducir esto podemos decir que:

- Para N\*M nosotros tendríamos  $(M)(M+1)/2(N)(N+1)/2=M(M+1)(N)(N+1)/4$

Entonces la fórmula para el total de rectángulos es:

- $M(M+1)(N)(N+1)/4$

```

// Código Java que cuenta el número
// de rectángulos en una matriz N*M
public class RectanglesInNxM {

    public static long rectCount(int n, int m)
    {
        return (m * n * (n + 1) * (m + 1)) / 4;
    }
    public static void main(String[] args)
    {
        int n = 5, m = 4;
        System.out.println(rectCount(n, m));
    }
}

```

## 18.21 Squares In Triangle 2x2

Cuál es el máximo número de cuadrados de tamaño 2x2 unidades que pueden caber en un triángulo isósceles de ángulos correctos dada la base en unidades.

Un lado del cuadrado debe ser paralelo a la base del triángulo.

Desde que el triángulo es isósceles, la base dada será también igual a la altura. Ahora en la parte diagonal, podríamos siempre necesitar un largo extra de dos unidades en la altura y la base del triángulo para acomodar un triángulo. En la longitud restante de la base, podemos construir  $\text{largo}/2$  cuadrados. Desde que cada cuadrado es de dos unidades, lo mismo puede hacerse en el caso de la altura, ahí no hay necesidad de calcular eso de nuevo. Entonces, para cada nivel de la longitud dada, podemos construir “ $(\text{largo}-2)/2$ ” cuadrados. Esto nos da una base de “ $(\text{largo}-2)$ ” encima de él. Continuando con el proceso de obtener el número de cuadrados para toda la disponible “ $\text{largo}/2$ ” altura, podemos calcular los cuadrados.

Para una forma más eficiente, podemos usar la fórmula de la suma de AP  $n*(n+1)/2$ , donde  $n = \text{largo}-2$ .

```
// Programa Java que cuenta el número de cuadros2
// 2x2 que cabe en un triángulo isoceles

public class SquaresInTriangle2x2 {

    public static int numberSquares(int base) {
        // Removiendo la parte extra
        // que podamos necesitar
        base = (base - 2);
        // Desde cada cuadrado que tenga
        // base de tamaño 2
        base = base / 2;
        return base * (base + 1) / 2;
    }
    public static void main(String args[]) {
        int base = 8;
        System.out.println(numberSquares(base));
    }
}
```

## 18.22 Sum Of Manhattan

Dados n coordenadas enteras, encuentre la suma de la distancia de Manhattan entre todos los pares de coordenadas.

La distancia de Manhattan entre dos puntos  $(x_1, y_1)$  y  $(x_2, y_2)$  es:

$$- |x_1 - x_2| + |y_1 - y_2|$$

La idea es recorrer dos ciclos anidados, por ejemplo cada punto, encontrar la distancia de todos los otros puntos con este.

Complejidad de tiempo:  $O(n^2)$

```
/*Programa Java para encontrar la suma de
las distancias de Manhattan entre todos
los pares de puntos dados */

public class SumOfManhattan {
    /*Retorna la suma de la distancia entre todos
     los pares de puntos*/
    static int distancesum(int x[], int y[], int n)
    {
        int sum = 0;
        /* Por cada punto, encuentra la distancia
         al resto de puntos */
        for (int i = 0; i < n; i++)
            for (int j = i + 1; j < n; j++)
                sum += (Math.abs(x[i] - x[j]) +
Math.abs(y[i] - y[j]));
        return sum;
    }

    public static void main(String[] args)
    {
        int x[] = { -1, 1, 3, 2 };
        int y[] = { 5, 6, 5, 3 };
        int n = x.length;
        System.out.println(distancesum(x, y, n));
    }
}
```

## 18.23 Three Points Collinear

Dados tres puntos, verificar si estos puntos yacen en recta (colineales) o no.

Ejemplo:  $(1, 1), (1, 4), (1, 5)$

Los tres puntos yacen en una línea recta.

Tres puntos yacen en una línea recta si el área formada por un triángulo de estos tres puntos es cero.

```

/*Programa en java para verificar si
tres puntos son colineares
o no usando el area del triángulo*/
public class ThreePointsCollinear {
//Función que verifica si un punto es colineal o no
static void collinear(int x1, int y1, int x2,
int y2, int x3, int y3) {
/*Calcular el area del triángulo
Debemos omitir multiplicaciones con 0.5
para evitar calculos flotantes*/
    int a = x1 * (y2 - y3)
        + x2 * (y3 - y1)
    + x3 * (y1 - y2);

    if (a == 0) {
        System.out.println("Yes");
    } else {
        System.out.println("No");
    }
}

public static void main(String args[]) {
int x1 = 1, x2 = 1, x3 = 1,
    y1 = 1, y2 = 4, y3 = 5;

collinear(x1, y1, x2, y2, x3, y3);
}
}

```

## 18.24 Triangle Angles

Dadas las coordenadas de tres vértices de un triángulo en un plano 2D, encuentre sus tres ángulos.

$$- c^2 = a^2 + b^2 - 2(a)(b)(\cos \beta)$$

Luego del despeje:

$$- \beta = \arccos( (a^2 + b^2 - c^2) / (2ab) )$$

En trigonometría la ley del coseno cuenta que los largos de los lados de un triángulo con el coseno de uno de sus ángulos luego de un despeje nos da el ángulo.

Primero calculamos el largo de todos los lados, luego aplicamos la fórmula de arriba para obtener todos los ángulos en radianes, luego convertimos de radianes a grados.

```

/* Codigo JAVA para encontrar todos los angulos
de un triángulo dadas las coordenadas
de los tres vertices*/

```

```

import java.awt.Point;
import static java.lang.Math.PI;
import static java.lang.Math.sqrt;
import static java.lang.Math.acos;

public class TriangleAngles { // Regresa el cuadrado de la distancia b/w de
//dos puntos

    static int lengthSquare(Point p1, Point p2) {
        int xDiff = p1.x - p2.x;
        int yDiff = p1.y - p2.y;
        return xDiff * xDiff + yDiff * yDiff;
    }

    static void printAngle(Point A, Point B,
Point C) {
//Cuadrado de los tamaños de a2, b2, c2
    int a2 = lengthSquare(B, C);
        int b2 = lengthSquare(A, C);
        int c2 = lengthSquare(A, B);
// Longitus de los lados de a, b, c
    float a = (float) sqrt(a2);
    float b = (float) sqrt(b2);
    float c = (float) sqrt(c2);
// De la ley del coseno
    float alfa = (float) acos((b2 + c2 - a2) / (2 * b * c));
    float beta = (float) acos((a2 + c2 - b2) / (2 * a * c));
    float gamma = (float) acos((a2 + b2 - c2) / (2 * a * b));
//Conversión a grados
    alfa = (float) (alfa * 180 / PI);
    beta = (float) (beta * 180 / PI);
    gamma = (float) (gamma * 180 / PI);
//Imprimiendo los angulos
// printing all the angles
    System.out.println("alfa : " + alfa);
    System.out.println("beta : " + beta);
    System.out.println("gamma : " + gamma);
}

public static void main(String[] args) {
    Point A = new Point(0, 0);
    Point B = new Point(0, 1);
    Point C = new Point(1, 0);
    printAngle(A, B, C);
}
}

```

## 18.25 Problemas de repaso

### Ejercicios en Online Judge

184-Laser Lines	476-Points in Figures: Rectangles
378-Intersecting Lines	681-Convex Hull Finding
477-Points in Figures: Rectangles and Circles	10200-Prime Time
478-Points in Figures: Rectangles, Circles, Triangles	10283-The Kissing Circles
	10573-Geometry Paradox

## Ejercicios en CodeChef

TRICOIN

RECTAGL

# 19 Manejo binario BitWise

## 19.1 Add One

Sume uno a un número dado, el uso de los operadores como '+', '-','\*', '/', '++','—'.... Entre otros no está permitido.

La respuesta se consigue con algo de magia de bits.

Para agregar 1 a un número x (como 0011000111), voltee todos los bits luego del cero de más a la derecha (bit 0) (obtenemos 0011000000) finalmente, voltee el cero de más a la derecha también (obtenemos 0011001000) para obtener la respuesta.

```
// Programa Java en donde se agrega uno a
// un número dado
public class AddOne {

    static int addOne(int x) {
        int m = 1;
        //Voltea todos los sets de bits
        //hasta que encuentre un 0
        while ((int) (x & m) == 1) {
            x = x ^ m;
            m <= 1;
        }
        //Voltea el bit 0 de más de la derecha
        x = x ^ m;
        return x;
    }

    public static void main(String[] args) {
        System.out.println(addOne(13));
    }
}
```

## 19.2 Binary Palindrome

Encuentre el número cuya representación binaria es un palíndromo. No se consideran los ceros iniciales, Mientras se considera la representación binaria, considere el primer número cuya representación binaria es palíndroma como 1, en vez de 0.

Una aproximación ingenua puede ser atravesar a través de todos los enteros desde 1 hasta  $2^{31}-1$  e incrementar el conteo palíndromo, si el número es palíndromo, cuando el conteo palíndromo alcanza el n requerido, rompe el ciclo y retorna el actual entero.

La complejidad de tiempo de esta solución es  $O(x)$  donde x es el número resultado. Note que el valor de x es generalmente más grande que n.

```
/* Programa Java para buscar el Nesimo número el cual
su binario es un palindromo*/
public class BinaryPalindromeN {
    static int INT_MAX = 2147483647;
    /*Busca si el kesimo bit esta
    puesto en la representacion binaria*/
    static int isKthBitSet(int x, int k) {
        return ((x & (1
        << (k - 1))) > 0) ? 1 : 0;
    }
    /*Retorna la posicion de más a la izquierda
    el set de bits en la representacion binaria*/
    static int leftmostSetBit(int x) {
        int count = 0;
        while (x > 0) {
            count++;
            x = x >> 1;
        }
        return count;
    }
    /*Encuentra cuales sean los enteros en binario
    siendo plindromos o no*/
    static int isBinPalindrome(int x) {
        int l = leftmostSetBit(x);
        int r = 1;
        //Uno a uno se comparan los bits
        while (l > r) {
        //Comprara bits de izquierda y derecha
        // y converge
            if (isKthBitSet(x, l)
                != isKthBitSet(x, r)) {
                return 0;
            }
            l--;
            r++;
        }
        return 1;
    }
}
```

```

        static int findNthPalindrome(int n) {
    int pal_count = 0;
/*Comienza desde 1, atravieza por todos los
enteros*/
    int i = 0;
    for (i = 1; i <= INT_MAX; i++) {
        if (isBinPalindrome(i) > 0) {
            pal_count++;
        }
/*Si nosotros llegamos n
rompe el ciclo*/
    if (pal_count == n) {
        break;
    }
    return i;
}
public static void main(String[] args) {
    int n = 9;
    System.out.println(findNthPalindrome(n));
}
}

```

## 19.3 Booth Multiply

El algoritmo de Booth es un algoritmo de multiplicación que multiplica dos números binarios con signo en su notación de complemento a 2.

Se pone el multiplicando en BR y el multiplicador en QR y luego el algoritmo funciona con las siguientes condiciones:

- 1) Si  $Q_n$  y  $Q_{n+1}$  son lo mismo, por ejemplo 00 y 11 se realiza movimiento aritmético de un bit.
- 2) Si  $Q_n$  y  $Q_{n+1}=10$  se realiza  $A=A+BR$  y se realiza movimiento aritmético de un bit.
- 3) Si  $Q_n$  y  $Q_{n+1} = 01$  se realiza  $A=A-BR$  y se realiza movimiento aritmético de un bit

```

//Programa java para multiplicar dos número binarios usando
//el algoritmo de Booth
import java.util.Arrays;

public class BoothMultiply {
    public static void main(String[] args) {
        int mt[] = new int[10];
        int sc;
int brn, qrn;
// Número de bits del multiplicando
        brn = 4;

```

```

// multiplicando
    int br[] = {0, 0, 1, 0};
//Copia multiplicador al arreglo temporal mt[]
for (int i = brn - 1; i >= 0; i--) {
    mt[i] = br[i];
}
int[] aux1 = Arrays.copyOf(br, br.length);
for (int i = aux1.length - 1; i >= 0; i--) {
    br[aux1.length - i - 1] = aux1[i];
}
complement(mt, brn);
// Número de bits del multiplicador
qrn = 4;
// Conteo de secuencias
sc = qrn;
// multiplicador
    int qr[] = {0, 0, 0, 1};
    int[] aux2 = Arrays.copyOf(qr, qr.length);
    for (int i = aux2.length - 1; i >= 0; i--) {
        qr[aux2.length - i - 1] = aux2[i];
    }
boothAlgorithm(br, qr, mt, qrn, sc);
System.out.print("\nResult = ");
for (int i = qrn - 1; i >= 0; i--) {
    System.out.print(qr[i]);
}
System.out.println("");
}

static void add(int ac[], int x[], int qrn) {
    int i, c = 0;
    for (i = 0; i < qrn; i++) {
//Actualizando acumulador con A=A+BR
ac[i] = ac[i] + x[i] + c;
        if (ac[i] > 1) {
            ac[i] = ac[i] % 2;
            c = 1;
        } else {
            c = 0;
        }
    }
}

static void complement(int a[], int n) {
    int i;
    int x[] = new int[8];
    x[0] = 1;
    for (i = 0; i < n; i++) {
        a[i] = (a[i] + 1) % 2;
    }
    add(a, x, n);
}

static void rightShift(int ac[], int qr[], int qn, int qrn) {
    int temp, i;

```

```

        temp = ac[0];
        qn = qr[0];
        System.out.print("\t\trightShift\t");
        for (i = 0; i < qrn - 1; i++) {
            ac[i] = ac[i + 1];
            qr[i] = qr[i + 1];
        }
        qr[qrn - 1] = temp;
    }

    static void display(int ac[], int qr[], int qrn) {
        int i;
        for (i = qrn - 1; i >= 0; i--) {
            System.out.print(ac[i]);
        }
        System.out.print("\t");

        for (i = qrn - 1; i >= 0; i--) {
            System.out.print(qr[i]);
        }
    }

    static void boothAlgorithm(int br[], int qr[], int mt[], int qrn, int sc) {
        int qn = 0, ac[] = new int[10];
        int temp = 0;
        System.out.print("qn\tq[n+1]\t\tBR\t\tAC\tQR\t\tsc\n");
        System.out.print("\t\tinitial\t\t");
        display(ac, qr, qrn);
        System.out.println("\t\t" + sc);
        while (sc != 0) {
            System.out.print(qr[0] + "\t" + qn);

// Segunda condición
            if ((qn + qr[0]) == 1) {
                if (temp == 0) {
//Restar BR del acumulador
                add(ac, mt, qrn);
                    System.out.print("\t\tA = A - BR\t");
                    for (int i = qrn - 1; i >= 0; i--) {
                        System.out.print(ac[i]);
                    }
                    temp = 1;
                } // Tercera condicion
            else if (temp == 1) {
// Añade BR al acumulador
                add(ac, br, qrn);
                    System.out.print("\t\tA = A + BR\t");
                    for (int i = qrn - 1; i >= 0; i--) {
                        System.out.print(ac[i]);
                    }
                    temp = 0;
                }
            System.out.print("\n\t");
            rightShift(ac, qr, qn, qrn);
        } //Primera condicion
    else if (qn - qr[0] == 0) {

```

```

        rightShift(ac, qr, qn, qrn);
    }
    display(ac, qr, qrn);
    System.out.print("\t");
// Decrementar contador
    sc--;
    System.out.print("\t" + sc + "\n");
}
}
}

```

## 19.4 Closest Number With Same Bits

Dado un entero positive n, imprimir el siguiente más pequeño y el previo más largo que tiene el mismo número de bits 1 en su representación binaria.

Aproximación por fuerza bruta:

Una simple aproximación es contar el número de 1 en n, y luego incrementar o decrementar hasta que encontremos un número con el mismo número de 1.

Aproximación optimizada:

Vamos a inicial con el código de getNext, y luego nos movemos a getPrev.

Aproximación de manipulación de bits para obtención del siguiente número:

Si pensamos acerca cual será el siguiente número, podemos observar lo siguiente, dado el número 13948, su representación binaria es:

```

1 1 0 1 1 0 0 1 1 1 1 1 0 0
13 12 11 10 9 8 7 6 5 4 3 2 1 0

```

Buscamos hacer este número más grande, pero no muy grande, también necesitamos mantener el mismo número de unos.

Nota: Dado un número N y dos localizaciones de bits i y j, suponga que volteamos bit i desde 1 a 0, y bit j desde 0 a 1, si i>j, entonces n podrá decrementarse, si i<j entonces n podrá incrementarse.

Sabemos lo siguiente:

- Si volteamos un cero a un uno, debemos voltear un uno a un ero.

- El número (Luego de dos volteretas) podría ser más grande si y solo si el bit cero a uno fue el izquierdo del bit uno a cero.

Nosotros queremos hacer el número más grande, pero no necesariamente más grande, por lo tanto necesitamos voltear el cero de más a la derecha el cual tiene unos en el derecho de él.

Para poner esto en una forma diferente, nosotros estamos volteando el cero no final de más a la derecha, esto es usando el ejemplo de abajo, los ceros finales son en la primera y cero posición. El cero no final de más a la derecha es un bit 7, vamos a llamar esta posición p.

- p → posición de más a la derecha que no sea cero final.

Paso 1: volteamos el cero más a la derecha no final.

- 1 1 0 1 1 0 1 1 1 1 1 0 0
- 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Con este cambio, hemos incrementado el número de unos en n, podemos encojer el número reordenando todos los bits de la derecha del bit p tales que los ceros están en la izquierda y los unos están a la derecha, cuando se hace esto, se busca reemplazar uno de los unos con cero.

Una forma relativamente fácil de hacer esto es contando cuantos unos están a la derecha de p, despejar todos los bits desde 0 hasta p, y luego agregar de nuevo en c1-1 unos. Dejar c1 ser el número de unos de la derecha de p y c0 el número de ceros de la derecha de p.

Vamos a verificar esto con un ejemplo:

- c1 → Número de unos de la derecha de p
- c0 → Número de ceros de la derecha de p
- $p = c0 + c1$

Paso 2: despejar los bits de la derecha de p, como antes  $c0 = 2$ .  $c1 = 5$ .  $p = 7$ .

- 1 1 0 1 1 0 1 0 0 0 0 0 0 0
- 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Para despejar estos bits, necesitamos crear una máscara que sea una secuencia de unos, seguido por p ceros, podemos hacer esto de la siguiente forma:

// Todos los ceros excepto por un 1 en la posición p.

- $a = 1 \ll p;$

// Todos los ceros, seguidos por p unos.

- $b = a - 1;$

// Todos los unos, seguidos por p ceros.

- $mask = \sim b;$

// Despeja los p bits de más a la derecha.

- $n = n \& mask;$

O más concisamente, hacemos:

- $n \&= \sim ((1 \ll p) - 1).$

Paso 3: Agrega un  $c1=1$  unos.

Step 3: Add one  $c1 - 1$  ones.

- 1 1 0 1 1 0 1 0 0 0 1 1 1 1
- 13 12 11 10 9 8 7 6 5 4 3 2 1 0

Para insertar  $c1-1$  unos a la derecha, se realiza:

// Ceros con un uno en la posición  $c1-1$

- $a = 1 \ll (c1 - 1);$

// Ceros con unos en posiciones cero a través de  $c1-1$

- $b = a - 1;$

// Inserta unos en las posiciones 0 a través de  $c1-1$

- $n = n | b;$

O más concisamente:

```
- n |= (1 << (c1 - 1)) - 1;

/* Programa JAVA de búsqueda del siguiente número con
la misma cantidad de 1 que algun anterior*/
public class ClosestNumbersSamebits {
    static int getNext(int n) {
// Computa c0 y c1
        int c = n;
int c0 = 0;
        int c1 = 0;
        while (((c & 1) == 0)
&& (c != 0)) {
            c0++;
            c >>= 1;
        }
        while ((c & 1) == 1) {
            c1++;
            c >>= 1;
        }
/*Si no hay número más grande
con el número de 1 buscado*/
if (c0 + c1 == 31
            || c0 + c1 == 0) {
            return -1;
}
//Posicion de el más derecho cero
        int p = c0 + c1;
//Voltea el zero más derecho
        n |= (1 << p);
// Despeja todos los bits de la derecha de p
        n &= ~((1 << p) - 1);
//Inserta los (c1-1) a la derecha
n |= (1 << (c1 - 1)) - 1;
        return n;
}
public static void main(String[] args) {
    int n = 5;
    System.out.println(getNext(n));
    n = 8;
    System.out.println(getNext(n));
}
```

## 19.5 Gray Code to Binary /Binary To Gray Code

Números binarios es la forma por defecto para almacenar números, pero en muchas aplicaciones los números binarios son difíciles de usar y una variación de los números binarios es necesaria, aquí es cuando los códigos de Gray son muy útiles.

El código de gray tiene una propiedad, dos números sucesivos difieren en un solo bit porque esta propiedad permite tener ciclo a través de varios estados con un esfuerzo mínimo y son usados en mapas k, también permiten corrección de errores, comunicación entre muchas cosas más.

¿Cómo generar códigos de Gray de n bits?

La siguiente es la secuencia de dos bits (n=2)

- 00 01 11 10

La siguiente es la secuencia de 3 bits (n = 3)

- 000 001 011 010 110 111 101 100

Y la siguiente es la secuencia de 4 bits (n = 4)

- 0000 0001 0011 0010 0110 0111 0101 0100 1100 1101 1111 1110 1010 1011 1001  
1000

Los códigos de Gray de n bits pueden ser generados de la lista de (n-1) códigos de Gray con los siguientes pasos.

- Dejar la lista de (n-1)bits ser L1, crear otra lista L2 la cual es la reversa de L1
- Modificar la lista L1 usando prefijo '0' en todos los códigos de L1
- Concatene L1 y L2. La lista concatenada es la lista requerida de los códigos de Gray de n bits

En ciencias de la computación muchas veces necesitamos convertir de código binario a código de Gray y viceversa, esta conversion puede ser realizada bajo las siguientes reglas:

#### **Conversión de binario a Gray:**

El bit más significante (MSB most significant bit) del código de gray es siempre igual a el MSB del código binario dado.

Otros bits de la salida del código de gray pueden ser obtenidos realizando XOR al bit del código binario en ese índice y en el índice anterior.

#### Conversión de Gray a binario:

El MSB del código binario es siempre igual al MSB del código de Gray.

Los otros bits de la salida del código binario pueden ser obtenidos verificando el bit del código Gray en ese índice, si el bit actual es 0, entonces copia el anterior bit del código binario, si no copia el inverso del anterior bit del código binario.

```
// Programa en java para conversion binario - Gray e inverso
import java.io.*;

public class CodeConversionGrayToBinary {

    public static void main(String args[]) throws IOException {

        String binary = "01001";
        System.out.println("Codigo gray de " + binary + " is " +
binarytoGray(binary));

        String gray = "01101";
        System.out.println("Codigo Binario de " + gray + " is " +
graytoBinary(gray));
    }

    static char xor_c(char a, char b) {
        return (a == b) ? '0' : '1';
    }
    //Funcion para voltear el bit
    static char flip(char c) {
        return (c == '0') ? '1' : '0';
    }
    //Funcion binario a gray
    static String binarytoGray(String binary) {
        String gray = "";
        gray += binary.charAt(0);
        // Computa bits restantes, siguiente bit es conmutado haciendole
        // XOR del previo con el actual en binario
        for (int i = 1; i < binary.length(); i++) {
        /*Comcatena XOR del bit anterior con el actual*/
        gray += xor_c(binary.charAt(i - 1),
                      binary.charAt(i));
        }
        return gray;
    }
    //Funcion gray a binario
    static String graytoBinary(String gray) {
        String binary = "";
        binary += gray.charAt(0);
        // Computa bits restantes,
```

```
        for (int i = 1; i < gray.length(); i++) {
//Si el bit actual es 0, concatena el bit anterior
if (gray.charAt(i) == '0') {
binary += binary.charAt(i - 1);
}
//Sino, concatena invertidamente el bit anterior
else {
                binary += flip(binary.charAt(i - 1));
}
        }
return binary;
}
```

## 19.6 Count A to B

Dados dos números ‘a’ y ‘b’, contar el número de bits necesarios a voltear para convertir ‘a’ en ‘b’.

- 1) Calcular el XOR de A y B:  $a\_xor\_b = A \wedge B$
  - 2) Contar los sets de bits de lo de arriba

Resultado del XOR calculado:

countSetBits(a xor b)

XOR de dos números tendrá set de bits solo en esos lugares donde A difiera de B

```
//Contar el número de bits que van a ser volteados para convertir A en B

public class CountAtoB {
    public static int countSetBits(int n)
    {
        int count = 0;
        while (n != 0) {
            count += n & 1;
            n >>= 1;
        }
        return count;
    }
    public static int FlippedCount(int a, int b)
    { //Retorna el conteo del set de bits en a XOR b
    return countSetBits(a ^ b);
    }
    public static void main(String[] args)
    {
        int a = 10;
        int b = 20;
        System.out.print(FlippedCount(a, b));
    }
}
```

```
}
```

## 19.7 Count Sets Of Bits

Dado un entero positivo n, cuente el número total de sets de bits en representación binaria de todos los números de 1 a n.

Una solución simple es correr un ciclo desde 1 hasta n y sumar el conteo de sets de bits de todos los números de 1 a n.

```
//Un programa siempre que cuenta el set de bits
//en todos los números de 1 a n.
public class CountSetBits{
    static int countSetBits( int n)
{
// Inicializa el resultado
    int bitCount = 0;
for (int i = 1; i <= n; i++)
    bitCount += countSetBitsUtil(i);
    return bitCount;
}
//Funcion que cuenta el set de bits de un número x
static int countSetBitsUtil( int x)
{
    if (x <= 0)
        return 0;
    return (x % 2 == 0 ? 0 : 1) +
        countSetBitsUtil(x / 2);
}
public static void main(String[] args)
{
    int n = 4;
    System.out.print("Conteo total del set de bits es ");
    System.out.print(countSetBits(n));
}
```

## 19.8 Euclid No Mod And Divide

El algoritmo de Euclides es usado para encontrar el GCD de dos números.

La idea es usar operaciones de BitWise, podemos encontrar x/2 usando x>>1. Podemos verificar si x es par o impar usando x&1.

- $\text{gcd}(a, b) = 2 * \text{gcd}(a/2, b/2)$  si ambos a y b son pares.
- $\text{gcd}(a, b) = \text{gcd}(a/2, b)$  si a es par y b es impar.
- $\text{gcd}(a, b) = \text{gcd}(a, b/2)$  si a es impar y b es par.

```
//Programa Java eficiente para realizar maximo comun divisor sin % y /
public class EuclidNoModAndDivide {

    public static void main(String[] args) {
        System.out.println(gcd(8, 9));
    }

    static int gcd(int a, int b) {
        // Casos base
        if (b == 0 || a == b) {
            return a;
        }
        if (a == 0) {
            return b;
        }
        /*Si ambos a y b son pares, divide ambos por 2
        y multiplica el resultado con 2*/
        if ((a & 1) == 0 && (b & 1) == 0) {
            return gcd(a >> 1, b >> 1) << 1;
        }
        //Si a es par, y b es impar, divide a por 2
        if ((a & 1) == 0 && (b & 1) != 0) {
            return gcd(a >> 1, b);
        }
        //Si a es impar y b es par, divide b por 2
        if ((a & 1) != 0 && (b & 1) == 0) {
            return gcd(a, b >> 1);
        }
        /*Si ambos son impares, entonces aplica el algoritmo de
        resta normal, note que el caso impar-impar siempre
        convierte casos impar-par luego de una recursion*/
        return (a > b) ? gcd(a - b, b) : gcd(a, b - a);
    }
}
```

## 19.9 Find Duplicates In Bit Array

Se tiene un arreglo de N números, donde n es al menos 32000, el array puede tener entradas duplicadas y no se sabe que N es.

Con solo 4 kilobytes de memoria disponible, ¿cómo podría imprimir todos los elementos duplicados en el array?

Tenemos 4 kilobytes de memoria lo cual significa que podemos direccionar hasta  $8 \times 4 \times 2^{10}$  bits, note que  $32 \times 2^{10}$  es más grande que 32000, podemos crear un bit con 32000 bits, donde cada bit representa un entero.

Si se necesita crear un bit con más de 32000 bits entonces se puede crear fácilmente más y más de 32000.

Usando este vector de bits, podemos entonces iterar a través del arreglo, marcando cada elemento v poniendo el bit v en 1, cuando pasemos por un elemento duplicado, los imprimimos.

```
//Programa Java para imprimir todos los duplicados en un arreglo

public class FindDuplicatesBitArray {

    static class BitArray {
        int[] arr;
        // Constructor
        public BitArray(int n) {
            /*Divide por 32, para almacenar n bits, nosotros necesitamos
            n/32 +1 enteros (Asumiendo int esta almacenado usando 32 bits*/
            arr = new int[(n >> 5) + 1];
        }
        // Obtener el valor de un bit en una posicion dada
        boolean get(int pos) {
            //Divide por 32 para encontrar la posicion del entero
            int index = (pos >> 5);
            //Ahora encuentra el numero de bits en arr[index]
            int bitNo = (pos & 0x1F);
            //Encuentra el valor dado un numero bit en arr[index]
            return (arr[index] & (1 << bitNo)) != 0;
        }
        // Acomoda un bit en una posicion dada
        void set(int pos) {
            // Encuentra indice de una posicion de un bit
            int index = (pos >> 5);
            //Acomoda un numero bit es arr[index]
            int bitNo = (pos & 0x1F);
            arr[index] |= (1 << bitNo);
        }
        // Funcion de impresion de los duplicados
    static void checkDuplicates(int[] arr) {
        // Crea un bit con 320000 bits
        BitArray ba = new BitArray(320000);

        // Arreglo transverso de los elementos
        for (int i = 0; i < arr.length; i++) {
            // Indice de un arreglo de bits
            int num = arr[i] - 1;
            //Si numero ya se encuentra presente en el arreglo de bits
```

```

if (ba.get(num)) {
    System.out.print((num + 1) + " ");
} // Si no inserte el número
else {
ba.set(num);
}
}
}

public static void main(String[] args) {
int[] arr = {10, 10, 1, 1, 2, 2, 3, 3};
BitArray.checkDuplicates(arr);
}
}

```

## 19.10 Gray Codes Of N

Dado un número n, generar los patrones de bits de 0 a  $2^n-1$  de los cuales cada patrón sucesivo difiera en uno solo bit.

Los códigos de Gray de n bits pueden ser generados de la lista de (n-1) códigos de Gray con los siguientes pasos.

- Dejar la lista de (n-1)bits ser L1, crear otra lista L2 la cual es la reversa de L1
- Modificar la lista L1 usando prefijo '0' en todos los códigos de L1
- Concatene L1 y L2. La lista concatenada es la lista requerida de los códigos de Gray de n bits

Por ejemplo, los siguientes son los pasos para generar el código de Gray de 3 bits desde la lista de códigos de Gray de 2 bits.

- L1 = {00, 01, 11, 10} (Lista de Gray de dos bits)
- L2 = {10, 11, 01, 00} (Reversa de L1)
- Agrega prefijo a todas las entradas de L1 con '0', L1 se convierte en {000, 001, 011, 010}
- Agrega prefijo a todas las entradas de L2 con '1', L2 se convierte en {110, 111, 101, 100}
- Concatena L1 y L2, obtenemos {000, 001, 011, 010, 110, 111, 101, 100}

Para generar los códigos de Gray de n bits, empezamos desde la lista de Grays de un solo bit, la cual es {0,1}, repetimos los pasos de arriba para general la lista de 2 bits a partir de la de 1 solo bit, luego generamos la de 3 bits a partir de la de 2 bits, y así hasta que el número de bits sea igual a n.

```
/Programa Java para generar codigos de Gray Nesimos

import java.util.ArrayList;

public class GrayCodesOfN {

    static void generateGrayarr(int n) {
        // Caso base
        if (n <= 0) {
            return;
        }
        // 'arr' podria almacenar todos los codigos generados
        ArrayList<String> arr = new ArrayList<String>();
        // Comienza con un patron de un bit
        arr.add("0");
        arr.add("1");
        /* Cada iteracion de este ciclo genera 2*i codigos desde los
         i codigos generados previamente*/
        int i, j;
        for (i = 2; i < (1 << n); i = i << 1) {
            /*Entra los previamente generados codigos de nuevo a arr[]
             en orden reverso, arr[] tiene el doble de numero de codigos*/
            for (j = i - 1; j >= 0; j--) {
                arr.add(arr.get(j));
            }
            // Concatena 0 a la primera mitad
            for (j = 0; j < i; j++) {
                arr.set(j, "0" + arr.get(j));
            }
            // Concatena 1 a la segunda mitad
            for (j = i; j < 2 * i; j++) {
                arr.set(j, "1" + arr.get(j));
            }
        }
        // Imprime el contenido de arr
        for (i = 0; i < arr.size(); i++) {
            System.out.println(arr.get(i));
        }
    }

    public static void main(String[] args) {
        generateGrayarr(3);
    }
}
```

## 19.11 Karatsuba Multiply

Dados dos strings binarios (A bigIntegers) que representan el valor de dos enteros, encuentre el producto de los dos strings, por ejemplo, si el primer string de bits es “1100” y el segundo string de bits es “1010” la salida debe ser 120.

Por simplicidad, dejaremos la longitud de ambos strings ser igual y ser n.

- x=5678 y=1234
- a=56,b=78
- c=12,d=34

Paso 0 = m = n/2 + n%2

Paso 1 = a\*c

Paso 2 = b\*d

Paso 3 = (a + b)\*(c + d)

Paso 4 = 3) - 2) - 1)

Paso 5 = 1)\*pow(10, m\*2) + 2) + 4)\*pow(10, m)

```
//Programa JAVA que realiza multiplicaciones
//Por medio del algoritmo de Karatsuba
import java.math.BigInteger;
import java.util.Scanner;

public class KaratsubaMultiply {

    public static void main(String[] args) {
        BigInteger x, y;
        Scanner sc = new Scanner(System.in);
        x = sc.nextBigInteger();
        y = sc.nextBigInteger();

        BigInteger result = karatsuba(x, y);
        long result2 =karatsuba(x.longValue(), y.longValue());
        System.out.println(result);
        System.out.println(result2);
    }

    private static long karatsuba(long x, long y) {
        if (x < 10 && y < 10) {
            return x * y;
        }
        int n = Math.max(Long.valueOf(x).toString().length(),
        (Long.valueOf(y).toString().length()));
        int m = n / 2 + n % 2;
        long a = x / (long) Math.pow(10, m);
        long b = x % (long) Math.pow(10, m);
        long c = y / (long) Math.pow(10, m);
        long d = y % (long) Math.pow(10, m);

        long sumac = a + b;
        long sumad = c + d;
        long productac = a * c;
        long productad = a * d;
        long productbc = b * c;
        long productbd = b * d;

        long result = productac * Math.pow(10, 2 * m) +
        ((sumac * sumad) * Math.pow(10, m)) +
        productbd;
```

```

        long d = y % (long) Math.pow(10, m);
        long step1 = karatsuba(a, c);
        long step2 = karatsuba(b, d);
        long step3 = karatsuba(a + b, c + d);
        long step4 = step3 - step2 - step1;
        long step5 = step1 * (long) Math.pow(10, m * 2) + step2 + step4 * (long)
Math.pow(10, m);
        return step5;
    }

    private static BigInteger karatsuba(BigInteger x, BigInteger y) {
        if (x.compareTo(BigInteger.valueOf(10)) < 0 &&
y.compareTo(BigInteger.valueOf(10)) < 0) {
            return x.multiply(y);
        }
        int n = Math.max(x.toString().length(), y.toString().length());
        int m = n / 2 + n % 2;
        BigInteger[] a_b = x.divideAndRemainder(BigInteger.valueOf(10).pow(m));
        BigInteger a = a_b[0];
        BigInteger b = a_b[1];
        BigInteger[] c_d = y.divideAndRemainder(BigInteger.valueOf(10).pow(m));
        BigInteger c = c_d[0];
        BigInteger d = c_d[1];
        BigInteger step1 = karatsuba(a, c);
        BigInteger step2 = karatsuba(b, d);
        BigInteger step3 = karatsuba(a.add(b), c.add(d));
        BigInteger step4 = step3.subtract(step2).subtract(step1);
        BigInteger step5 = step1.multiply(BigInteger.valueOf(10).pow(m *
2)).add(step2)
            .add(step4.multiply(BigInteger.valueOf(10).pow(m)));
        return step5;
    }
}

```

## 19.12 Max Consecutive One

Dado un entero  $n$ , podemos voltear exactamente un bit, encuentre la longitud de la secuencia más larga de 1 que se pueda crear.

Una solución eficiente es andar a través de los bits en representación binaria del número dado, mantenemos rastreo de la longitud de la secuencia actual de unos, de la longitud de la secuencia de unos previa, cuando veamos un cero, actualizamos la longitud anterior:

- Si el siguiente bit es un 1, longitud anterior deberá ser ahora la longitud actual.
- Si el siguiente bit es un 0, entonces no podemos unir esas secuencias juntas, entonces la longitud previa es 0.

Actualizamos la longitud máxima comparando las dos siguientes cosas:

- Valor actual de max\_length
- Current-length+previos-length

Entonces

- result = return max-length+1 (// Agrega 1 para el Contador de bits volteados )

```
//Porgrama java para buscar el subset de 1 más largo
public class MaxConsecutiveOne {

    static int flipBit(int a) {
        /*Si todos los bits son 1, la representacion de 'a'
         tiene todos los unos*/

        if (~a == 0) {
            return 8 * sizeof();
        }

        int currLen = 0, prevLen = 0, maxLen = 0;
        while (a != 0) {
            /*Si el bit actual es un 1
             entonces incrementa currLen*/
            if ((a & 1) == 1) {
                currLen++;
            }
            /*Si el bit actual es un 0
             revisa el siguiente bit de a*/
            else if ((a & 1) == 0) {
                /*Actualiza prevLen a 0 (Si el siguiente bit es 0)
                 o currLen (Si el siguiente bit es 1)*/
                prevLen = (a & 2) == 0 ? 0 : currLen;
                /*Si dos bits consecutivos son 0
                 entonces currLen tambien sera 0*/
                currLen = 0;
            }
            // Actualiza maxLen si es requerido
            maxLen = Math.max(prevLen + currLen, maxLen);

            //Remueve el ultimo digito (Right shift)
            a >>= 1;
        }
        /*Nosotros siempre podremos tener un secuencia de
         al menos un 1, este es un bit volteado*/
        return maxLen + 1;
    }

    static byte sizeof() {
        byte sizeOfInteger = 8;
        return sizeOfInteger;
    }
}
```

```

    public static void main(String[] args) {
        System.out.println(flipBit(13));
        System.out.println(flipBit(1775));
        System.out.println(flipBit(15));
    }
}

```

## 19.13 Max Subarray XOR

Dado un array de enteros, encontrar el valor máximo del Subarray XOR en el array dado.

Una solución simple es usar dos ciclos para encontrar el XOR de todos los subarrays y retornar el máximo.

La complejidad de tiempo de esta solución es  $O(n^2)$ .

```

// Programa en JAVA para buscar el maximo subarreglo XOR
public class MaxSubarrayXOR {

    static int maxSubarrayXOR(int arr[], int n) {
        int ans = Integer.MIN_VALUE; // Inicializar resultado
        // Escogiendo puntos de inicio para los subarreglos
        for (int i = 0; i < n; i++) {
            // para guardar XOR del actual subarreglo
            int curr_xor = 0;
            // Escojiendo puntos finales de subarreglos empezando por i
            for (int j = i; j < n; j++) {
                curr_xor = curr_xor ^ arr[j];
                ans = Math.max(ans, curr_xor);
            }
        }
        return ans;
    }

    public static void main(String args[]) {
        int arr[] = {8, 1, 2, 12};
        int n = arr.length;
        System.out.println("Maximo subarray XOR es "
            + maxSubarrayXOR(arr, n));
    }
}

```

## 19.14 Multiply By Seven

Podemos multiplicar un número por 7 usando operadores BitWise, primero hacemos corrimiento izquierdo del número de 3 bits (Se obtendrá  $8n$ ) luego se resta la forma original del número corrido y se retorna la diferencia ( $8n-n$ ).

Complejidad de tiempo: O(1).

Funciona solo para enteros positivos.

El mismo concepto puede ser usado para multiplicación rápida por 9 u otros números.

```
// Programa Java para verificar si es
// multiplo de 7
public class MultiplyBySeven{

    static int multiplyBySeven(int n)
    {
        return ((n << 3) - n);
    }
    public static void main (String arg[])
    {
        int n = 4;
        System.out.println(multiplyBySeven(n));
    }
}
```

## 19.15 Magic Number Of N

Un número mágico es definido como un número el cual puede ser expresado como una potencia de 5 o la suma de potencias únicas de 5, algunos números mágicos son 5, 25, 30( $5 + 25$ ), 125, 130( $125 + 5$ ), ....

Si revisamos cuidadosamente los números mágicos, pueden ser representados como 001, 010, 011, 100, 101, 110, entre otros, donde 001 es  $0 * \text{pow}(5,3) + 0 * \text{pow}(5,2) + 1 * \text{pow}(5,1)$ , entonces básicamente necesitamos agregar potencias de 5 por cada bit dado en el entero dado  $n$ .

```
// Programa en java para buscar el Nsimo
// número magico, un número magico esta definido como un número el cual puede
// ser expresado como
// una potencia de 5 o suma de potencias unicas de 5
// Algunos primeros número magicos son: 5, 25, 30(5 + 25), 125, 130(125 + 5), ...
// en adelante
public class NMagicNumber {
    static int nthMagicNo(int n) {
        int pow = 1, answer = 0;
```

```

// Ir a traves de cada bit de n
    while (n != 0) {
        pow = pow * 5;
    // Si el ultimo bit de n esta puesto
    if ((int) (n & 1) == 1) {
        answer += pow;
    }
    //Proceder con el siguiente bit
    // 0 n= n/2
    n >>= 1;
}
return answer;
}

public static void main(String[] args) {
int n = 5;
System.out.println("Enesimo número"
+ " magico es " + nthMagicNo(n));
}
}

```

## 19.16 N Square No Pow

Dado un entero n, calcular el cuadrado de un número sin usar \*, / y pow().

Podemos hacerlo en tiempo O(Logn) usando operadores de BitWise, la idea está basada en el siguiente hecho:

- $\text{square}(n) = 0 \text{ if } n == 0$

Si n es par

- $\text{square}(n) = 4 * \text{square}(n/2)$

Si n es impar

- $\text{square}(n) = 4 * \text{square}(\text{floor}(n/2)) + 4 * \text{floor}(n/2) + 1$

Por ejemplo:

- $\text{square}(6) = 4 * \text{square}(3)$
- $\text{square}(3) = 4 * (\text{square}(1)) + 4 * 1 + 1 = 9$
- $\text{square}(7) = 4 * \text{square}(3) + 4 * 3 + 1 = 4 * 9 + 4 * 3 + 1 = 49$

Si n es par, puede ser escrito como:

- $n = 2*x$
- $n^2 = (2*x)^2 = 4*x^2$

Si  $n$  es impar, puede ser escrito como:

- $n = 2*x + 1$
- $n^2 = (2*x + 1)^2 = 4*x^2 + 4*x + 1$

$\text{floor}(n/2)$  puede ser calculado usando el operador de BitWise corrimiento derecho.

```
// Programa en java para calcular el cuadrado
// Sin usar * ni Pow()

public class NSquareNoPow {

    static int square(int n)
    {

        // Caso base
        if(n == 0)
            return 0;
        if(n < 0)
            n = -n;
        // Obtener floor(n/2) usando
        // right shift
        int x = n >> 1;
        // si n es par
        ;
        if(n % 2 != 0)
            return ((square(x) << 2)
                    + (x << 2) + 1);
        else// si n es impar
            return (square(x) << 2);
    }

    public static void main(String args[])
    {
        for(int n = 1; n <= 5; n++)
            System.out.println("n = "+ n +
                               " n^2 = "+
                               square(n));
    }
}
```

## 19.17 Odd Even Swap

Dado un entero sin signo, intercambia todos los bits pares con impares, por ejemplo si el número dado es 23 (00010111), puede ser convertido en 43 (00101011). Cada posición de bit par es intercambiada con el bit adyacente del lado de derecho, y cada posición impar es cambiada con el adyacente del lado izquierdo.

Si realizamos una revisamos en el ejemplo, podemos observar que básicamente necesitamos el corrimiento derecho (`>>`) de todos los bits pares por 1, entonces se convierten en bits impares, y corrimiento izquierdo (`<<`) todos los bits impares por 1 entonces se convierten en pares, la siguiente solución está basada en estas ideas, se asume que el número de entrada está almacenado usando 32 bits.

Dejemos la entrada ser `x`.

- 1) Obtener todos los bits pares de `x` realizando BitWise, y or de `x` con `0xAAAAAAA`, el número `0xAAAAAAA` es un entero de 32 bit con todos los bits pares ubicados en 1, y todos los impares en 0.
- 2) Obtener todos los bits impares usando BitWise, y or de `x` con `0x5555555`, el número `0x5555555` es un número entero de 32 bits con todos los bits impares en 1 y los pares en 0.
- 3) Corrimiento derecho de todos los bits pares
- 4) Corrimiento izquierdo de todos los bits impares
- 5) Combina los nuevos pares e impares y retorna

```
// Programa java para invertir bits pares
// e impares de un número dado
public class OddEvenSwap {

    static int swapBits(int x) {
        // Obtener todos los bits par de x
        int even_bits = x & 0xAAAAAAA;

        //Obtener todos los bits impar de x
        int odd_bits = x & 0x5555555;

        // Movimiento derecho de bits pares
        even_bits >>= 1;

        // Movimiento izquierdo de bits impares
        odd_bits <= 1;

        // combinar pares e impares
    }
}
```

```

        return (even_bits | odd_bits);
    }

    public static void main(String[] args) {
int x = 23; // 00010111
// La salida es 43 (00101011)
System.out.println(swapBits(x));
}
}

```

## 19.18 Odd Occurrence

Dado un array de enteros positivos, todos los números ocurren número par de veces excepto un número el cual ocurre impar veces, encuentre el número en O(n) tiempo y espacio constante.

Una solución simple es correr dos ciclos anidados, el ciclo externo toma todos los elementos uno por uno y el ciclo interno cuenta el número de las ocurrencias del elemento tomado en el ciclo externo, la complejidad de tiempo de esta solución es O( $n^2$ ).

```

//Programa JAVA para buscar la ocurrencia de un elemento
// número impar de veces
public class OddOccurrence {

static int getOddOccurrence(int arr[], int arr_size)
{
    int i;
    for (i = 0; i < arr_size; i++) {
        int count = 0;
        for (int j = 0; j < arr_size; j++) {
            if (arr[i] == arr[j])
                count++;
        }
        if (count % 2 != 0)
            return arr[i];
    }
    return -1;
}

public static void main(String[] args)
{
    int arr[] = new int[]{ 2, 3, 5, 4, 5, 2, 4, 3, 5, 2, 4, 4, 2 };
    int n = arr.length;
    System.out.println(getOddOccurrence(arr, n));
}
}

```

## 19.19 Opposite Signs

Dado dos enteros con signo, retorne true si los signos de los enteros dados son diferentes, de otra forma retorne falso, por ejemplo la función debe retornar true en -1 y +100, y debe retornar falso para -200 y -100. La función no debe usar ningún operador aritmético.

Dejaremos los enteros dados ser  $x$  y  $y$ , el bit de signo es 1 en números negativos, y 0 en números positivos, El XOR de  $x$  y  $y$  puede tener el bit de signo en 1 si ellos tienen signos opuestos, en otras palabras, XOR de  $x$  y  $y$  será un número negativo si  $x$  y  $y$  tienen signos opuestos.

```
//Programa Java para detectar si dos enteros
//tienen signos opuestos

public class OppositeSigns {

    static boolean oppositeSigns(int x, int y) {
        return ((x ^ y) < 0);
    }

    public static void main(String[] args) {
        int x = 100, y = -100;
        if (oppositeSigns(x, y) == true) {
            System.out.println("Signos opuestos");
        } else {
            System.out.println("Signos no opuestos");
        }
        x = 100;
        y = 100;
        if (oppositeSigns(x, y) == true) {
            System.out.println("Signos son opuestos");
        } else {
            System.out.println("Signos no son opuestos");
        }
    }
}
```

## 19.20 Pandigital Concatenations

Un par de string cuando son concatenados puede ser una concatenación Pandigital si su concatenación consiste en todos los dígitos de 0 a 9 en orden al menos una vez, la tarea es dado  $n$  strings, computar el número de pares que resultan en una concatenación Pandigital.

Una posible solución por fuerza bruta es forma todas las posibles concatenaciones formando todos los pares en  $O(n^2)$  y usando un array de frecuencia para los dígitos 0 a 9, verificamos cada digito si existe al menos una vez en cada concatenación formado por cada par.

```
//Programa JAVA para buscar todas las concatenaaciones
//Pandigitales de dos Strings
import java.util.ArrayList;

public class PandigitalConcatenations {

    static ArrayList<String> v = new ArrayList<String>();

    static int isPanDigital(String s) {
        int digits[] = new int[10];

        for (int i = 0; i < s.length(); i++) {
            digits[s.charAt(i)
                    - (int) '0'] = 1;
        }

        // Dígito i no está presente
        // entonces no es pandigital
        for (int i = 0; i <= 9; i++) {
            if (digits[i] == 0) {
                return 0;
            }
        }
        return 1;
    }

    // Retorna el número de pares de strings
    // resultado en las concatenaciones pandigitales
    static int countPandigitalPairs() {
        //Itera sobre todos los pares
        int pairs = 0;
        for (int i = 0; i < v.size(); i++) {
            for (int j = i + 1;
                 j < v.size(); j++) {
                if (isPanDigital(v.get(i)
                                + v.get(j)) == 1) {
                    pairs++;
                }
            }
        }
        return pairs;
    }

    public static void main(String args[]) {
        v.add("123567");
        v.add("098234");
        v.add("14765");
        v.add("19804");
    }
}
```

```

        System.out.println(countPandigitalPairs());
    }
}

```

## 19.21 Power Of Four

Dado un entero n, encontrar si es potencia de 4 o no.

- 1) Un método simple es tomar logaritmo del número dado en base 4, y si obtenemos un número entero entonces el número es potencia de 4.
- 2) Otra solución es mantenerse dividiendo el número por 4, por ejemplo haremos  $n=n/4$  iterativamente, si  $n \% 4$  se convierte en un no cero y n no es 1, entonces no es potencia de 4, de otra forma si lo es

```

// Código Java para verificar si un número
// es una potencia de 4 o no

public class PowerOfFour {

    static int isPowerOfFour(int n) {
        if (n == 0) {
            return 0;
        }
        while (n != 1) {
            if (n % 4 != 0) {
                return 0;
            }
            n = n / 4;
        }
        return 1;
    }

    public static void main(String[] args) {
        int test_no = 64;
        if (isPowerOfFour(test_no) == 1) {
System.out.println(test_no
                    + " es potencia de 4");
    } else {
        System.out.println(test_no
+ " no es potencia de 4");
    }
    test_no = 17;
if (isPowerOfFour(test_no) == 1) {
        System.out.println(test_no
+ " es potencia de 4");
    } else {
        System.out.println(test_no
+ " no es potencia de 4");
    }
}

```

```
        }
    }
}
```

## 19.22 Russian Peasant Multiply

Dados dos enteros, multiplicador sin usar el operador de multiplicación.

Una forma interesante es usar el algoritmo de la campesina rusa, la idea es duplicar el primer número y dividir en dos el segundo número repetidamente mientras el segundo número no se convierta en 1, en el proceso cuando el segundo número se vuelta impar, añadimos el primer número al resultado, el cual esta inicializado en 0.

El valor de  $a*b$  es el mismo que  $(a*2)*(b/2)$ , si  $b$  es par, de otra forma el valor es el mismo de  $((a*2)*(b/2) + a)$ , en el ciclo while, seguimos multiplicando ‘ $a$ ’ con dos y seguimos dividiendo ‘ $b$ ’ por 2, si ‘ $b$ ’ se convierte en impar en el ciclo, agregamos ‘ $a$ ’ a ‘ $res$ ’ cuando el calor de ‘ $b$ ’ se convierta en 1, el valor de ‘ $res$ ’ + ‘ $a$ ’ nos da el resultado.

Note que cuando ‘ $b$ ’ es una potencia de 2, el ‘ $res$ ’ puede mantenerse en 0 y ‘ $a$ ’ puede ser la multiplicación.

```
// Programa en java para multiplicar usando el algoritmo de Russian Peasant
public class RussianPeasantMultiply {
    // Funcion para multiplicar dos numeros

    static int russianPeasant(int a, int b) {
        // Inicializar resultado
        int res = 0;
        //Mientras el segundo no se convierta en 1
        while (b > 0) {
            // Si el segundo numero es impar,
            // añade el primer numero al resultado
            if ((b & 1) != 0) {
                res = res + a;
            }
            // El doble del primer numero
            // y la mitad del segundo numero
            a = a << 1;
            b = b >> 1;
        }
        return res;
    }
    public static void main(String[] args) {
        System.out.println(russianPeasant(18, 1));
    }
}
```

```
        System.out.println(russianPeasant(20, 12));  
    }  
}
```

## 19.23 Problemas de repaso

### Ejercicios en Online Judge

574-Sum It Up

10446-The Marriage Interview : - )

10176-Ocean Deep! Make it shallow!!

10664-Luggage

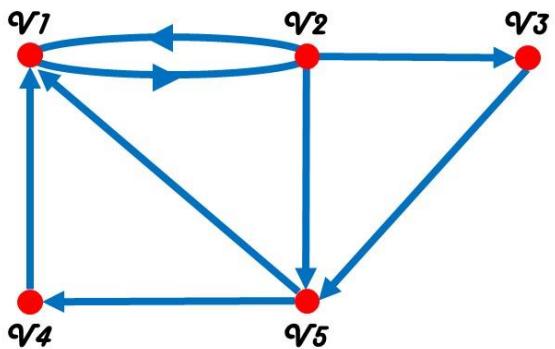
10469-To Carry or not to Carry

10911- Forming Quiz Teams

## 20 Grafos y arboles

### 20.1 Matrix of Adyacency

#### Matriz de Adyacencia



	V1	V2	V3	V4	V5
V1	0	1	0	0	0
V2	1	0	1	0	1
V3	0	0	0	0	1
V4	1	0	0	0	0
V5	1	0	0	1	0

*Guía del programador competitivo.*

Ilustración 20-1 Matriz de adyacencia de un grafo

La matriz de adyacencia es una matriz cuadrada que se utiliza como una forma de representar relaciones binarias. Se crea una matriz cero, cuyas columnas y filas representan los nodos del grafo.

- 1) Por cada arista que une a dos nodos, se suma 1 al valor que hay actualmente en la ubicación correspondiente de la matriz.
- 2) Si tal arista es un bucle y el grafo es no dirigido, entonces se suma 2 en vez de 1.
- 3) Finalmente, se obtiene una matriz que representa el número de aristas (relaciones) entre cada par de nodos (elementos).

Existe una matriz de adyacencia única para cada grafo (sin considerar las permutaciones de filas o columnas), y viceversa.

- Para un grafo no dirigido la matriz de adyacencia es simétrica.
- El número de caminos  $C_{i,j}(k)$ , atravesando  $k$  aristas desde el nodo  $i$  al nodo  $j$ , viene dado por un elemento de la potencia  $k$ -ésima de la matriz de adyacencia: -
- $C_{i,j}(k) = [A^k]_{ij}$

Existen otras formas de representar relaciones binarias, como por ejemplo los pares ordenados o los grafos

```
//Notacion java de una matriz de adyacencia
import java.util.Scanner;

public class AdyacencyMatrix {
//matriz del grafo
    static int[][] G;
    static Scanner sc = new Scanner(System.in);

    public static void main(String[] args) {
//Vertices y caminos
        int V = 4, E = 8;
        G = new int[V][V];
        for (int i = 0; i < V; i++) {
//desde hasta peso
            int aux1, aux2, aux3;
            aux1 = sc.nextInt();
            aux2 = sc.nextInt();
            aux3 = sc.nextInt();
            // solo este si es dirigido
            G[aux1][aux2] = aux3;
            // inverso si es no dirigido
            G[aux2][aux1] = aux3;
        }
        print(V);
    }
//Función que imprime la matriz de adyacencia
    static void print(int V){
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                System.out.print(G[i][j] + " ");
            }
            System.out.println("");
        }
    }
}
```

## 20.2 List of Adyacency

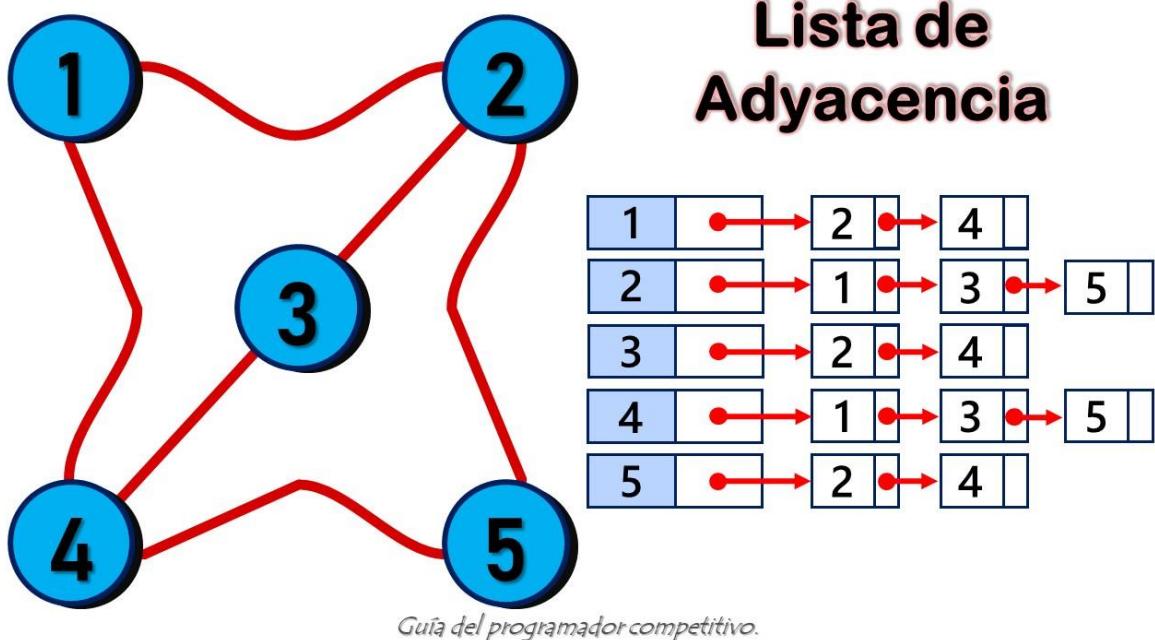


Ilustración 20-2 Lista de listas de adyacencia de un grafo

En teoría de grafos, una lista de adyacencia es una representación de todas las aristas o arcos de un grafo mediante una lista.

Si el grafo es no dirigido, cada entrada es un conjunto o multiconjunto de dos vértices conteniendo los dos extremos de la arista correspondiente. Si el grafo es dirigido, cada entrada es una tupla de dos nodos, uno denotando el nodo fuente y el otro denotando el nodo destino del arco correspondiente.

Típicamente, las listas de adyacentes no son ordenadas.

```
//Implementación java de una lista de adyacencia
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class ListAdyacency {

    static int N;
    //Grafo dado por vertices
    static Vertex[] G;
    static Scanner sc = new Scanner(System.in);
    public static void main(String[] args) {
        // Vertices y caminos
        int V = 4, E = 8;
        G = new Vertex[V];
        for (int i = 0; i < V; i++) {
```

```

//Inicializar cada lista de lista
        G[i].adj = new ArrayList<>();
    }
    for (int i = 0; i < E; i++) {
// de hasta peso
        int aux1,aux2,aux3;
        aux1=sc.nextInt();
        aux2=sc.nextInt();
aux3=sc.nextInt();
        //Dirijido
        G[aux1].adj.add(new Edge(aux2,aux3));
//No dirigido
        G[aux2].adj.add(new Edge(aux1,aux3));
    }
}

static class Vertex {
    List<Edge> adj;
    public Vertex() {
        adj = new ArrayList<>();
    }
}

static class Edge {
    int to, w;
    public Edge(int to, int w) {
        this.to = to;
        this.w = w;
    }
}
}

```

## 20.3 BFS (Breath First Search)

Búsqueda en anchura (en inglés BFS - Breadth First Search) es un algoritmo de búsqueda no informada utilizado para recorrer o buscar elementos en un grafo (usado frecuentemente sobre árboles). Intuitivamente, se comienza en la raíz (eliendo algún nodo como elemento raíz en el caso de un grafo) y se exploran todos los vecinos de este nodo. A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el árbol.

Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución. El algoritmo no usa ninguna estrategia heurística.

- Dado un vértice fuente  $s$ , Breadth-first search sistemáticamente explora los vértices de  $G$  para “descubrir” todos los vértices alcanzables desde  $s$ .
- Calcula la distancia (menor número de vértices) desde  $s$  a todos los vértices alcanzables.
- Después produce un árbol BF con raíz en  $s$  y que contiene a todos los vértices alcanzables.
- El camino desde  $dt$  a cada vértice en este recorrido contiene el mínimo número de vértices. Es el camino más corto medido en número de vértices.
- Su nombre se debe a que expande uniformemente la frontera entre lo descubierto y lo no descubierto. Llega a los nodos de distancia  $k$ , sólo tras haber llegado a todos los nodos a distancia  $k-1$ .

La complejidad computacional del algoritmo se puede expresar como  $O(|V|+|E|)$   $O(|V|+|E|)$ , donde  $|V|$   $|V|$  es el número de vértices y  $|E|$  es el número de aristas. El razonamiento es porque en el peor caso, cada vértice y cada arista serán visitados por el algoritmo.

```
// Programa java que imprime BFS (Busqueda en anchura) transverso
// desde un vertice (nodo) dado como inicio
// BFS(int s) atravesia vertices alcanzables desde s
import java.util.*;
//Esta clase representa un grafo dirigido usando listas de adyacencia

public class BFS {
    public static void main(String args[]) {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);
        System.out.println("Siguiendo su primera busqueda en anchura transverso"
            + "(iniciando desde el vertice 2)");
        //Vertice de inicio
        g.BFS(2);
    }

    static class Graph {
        private final int V;    // número de vertices
        private LinkedList<Integer> adj[]; //lista de ayacencia
        // Constructor
        Graph(int v) {
```

## 20.4 BFS All Paths

```
//Programa java que imprime todos los caminos
// desde un inicio hasta un destino usando BFS
import java.util.ArrayList;
import java.util.List;
//Un grafo dirigido usando lista de adyacencia
public class BFSAllPaths {
    // Número de vertices
        // No. of vertices in graph
    static int v;
    // lista de adyacencia
```

```

static ArrayList<Integer>[] adjList;
//Constructor
    static void initGraph(int vertices) {
//Inicializa número de vértices
        v = vertices;
// inicializa lista de adyacencia
        initAdjList();
}
    static void initAdjList() {
        adjList = new ArrayList[v];
        for (int i = 0; i < v; i++) {
            adjList[i] = new ArrayList<>();
        }
    }
//Agrega camino de u a v
    static void addEdge(int u, int v) {
// Agrega v a la lista de u
        adjList[u].add(v);
    }
//Imprime todos los caminos de
// 's' a 'd'
    static void printAllPaths(int s, int d) {
        boolean[] isVisited = new boolean[v];
        ArrayList<Integer> pathList = new ArrayList<>();
//Agrega origen al path[]
        pathList.add(s);
        printAllPathsUtil(s, d, isVisited, pathList);
}
//Función recursiva que imprime
    /*Todos los caminos de u a d.
    isVisited[] mantiene rastro de
    los vértices en el camino actual
    localPathList<> almacena vértices actuales
    en el camino actual*/
    static void printAllPathsUtil(Integer u, Integer d,
                                boolean[] isVisited,
                                List<Integer> localPathList) {
        // Marca el nodo actual
        isVisited[u] = true;
        if (u.equals(d)) {
            System.out.println(localPathList);
// Si coincidencia encontrada entonces
            // no se necesita atravesar más profundo
            isVisited[u] = false;
            return;
        }
// Recorre todos los vértices
//adyacentes al actual vértice
        for (Integer i : adjList[u]) {
            if (!isVisited[i]) {
// Almacena el nodo actual
                // en path[]
                localPathList.add(i);
                printAllPathsUtil(i, d, isVisited, localPathList);
            }
        }
    }

```

```

        localPathList.remove(i);
    }
}
// MArca el nodo actual
isVisited[u] = false;
}

public static void main(String[] args) {
//Crea el grafo
    initGraph(4);
//camino desde hasta
addEdge(0, 1);
    addEdge(0, 2);
    addEdge(0, 3);
    addEdge(2, 0);
    addEdge(2, 1);
addEdge(1, 3);
// inicio arbitrario
    int s = 0;
// destino arbitrario
    int d = 3;
    System.out.println("Los siguientes son todos los diferentes"
        + "caminos de " + s + " a " + d);
    printAllPaths(s, d);

}
}

```

## 20.5 Binary Graph Shortest Path

Dado un grafo donde cada camino tiene un peso de 0 o 1, un vértice inicial también es dado, encuentre el camino más corto desde el origen hasta cualquier otro vértice.

En el BFS normal de un grafo, todos los caminos son de igual peso, pero en BFS 0-1 ALGUNOS caminos pueden tener peso 0 y algunos 1, en este no podemos usar un array de booleanos para marcar los nodos visitados pero en cada paso podemos verificar la condición de distancia óptima, usamos una cola de doble fin para almacenar el nodo, mientras realizamos BFS si un camino es encontrado con peso 0 el nodo es empujado al frente de la doble cola y si el camino tiene peso 1 es empujado atrás en la doble cola.

Esta aproximación es similar a Dijkstra en que si la distancia más corta a un nodo es relajada por el nodo anterior entonces solo será empujada en la cola.

La idea de arriba trabaja en todos los casos, cuando sacamos un vértice, es el vértice de mínimo peso de todos los vértices restantes, si hay un vértice de peso 0 adyacente a el

entonces este adyacente tiene la misma distancia, si hay un adyacente de peso 1, entonces este adyacente tiene la máxima distancia de todos los vértices en la cola doble, (Porque todos los otros vértices son adyacentes de actual eliminado vértice o adyacente de los anteriores eliminados).

Este problema puede ser resuelto con Dijkstra pero el tiempo de complejidad seria de  $O(E + V \log V)$  donde por BFS puede ser de  $O(V+E)$ .

```
//Programa java que implementa el camino más corto
// en un grafo binario
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;

public class BinaryGraphShortestPath {
    //Número de vértices
    static int V = 9;
    // Lista de listas que almacena los caminos
    static ArrayList<ArrayList<node>> edges = new ArrayList<ArrayList<node>>();
    public static void main(String[] args) {
        for (int i = 0; i < V; i++) {
            edges.add(new ArrayList<>());
        }
        addEdge(0, 1, 0);
        addEdge(0, 7, 1);
        addEdge(1, 7, 1);
        addEdge(1, 2, 1);
        addEdge(2, 3, 0);
        addEdge(2, 5, 0);
        addEdge(2, 8, 1);
        addEdge(3, 4, 1);
        addEdge(3, 5, 1);
        addEdge(4, 5, 1);
        addEdge(5, 6, 1);
        addEdge(6, 7, 1);
        addEdge(7, 8, 1);
        int src = 0;//Nodo inicial
        zeroOneBFS(src);
    }

    static void zeroOneBFS(int src) {
        // Inicializa las distancias desde el nodo inicial

        int dist[] = new int[V];
        for (int i = 0; i < V; i++) {
            dist[i] = Integer.MAX_VALUE;
        }
        //cola doble para el BFS.
        Deque<Integer> Q = new ArrayDeque<>();
        dist[src] = 0;
```

```

Q.add(src);
while (!Q.isEmpty()) {
    int v = Q.getFirst();
    Q.removeFirst();
    for (int i = 0; i < edges.get(v).size(); i++) {
//Busca la distancia optima
        if (dist[edges.get(v).get(i).to] > dist[v] +
edges.get(v).get(i).weight) {
            dist[edges.get(v).get(i).to] = dist[v] +
edges.get(v).get(i).weight;
/* pone peso caminos de peso 0 al frente y 1 atras para
que los vertices puedan ser procesados en orden ascendente
por sus pesos*/
            if (edges.get(v).get(i).weight == 0) {
                Q.addFirst(edges.get(v).get(i).to);
            } else {
                Q.addLast(edges.get(v).get(i).to);
            }
        }
    }
}
//imprimiendo los caminos mas cortos
for (int i = 0; i < V; i++) {
System.out.print(dist[i] + " ");
}
System.out.println("");
}

static void addEdge(int u, int v, int wt) {
    edges.get(u).add(new node(v, wt));
    edges.get(v).add(new node(u, wt));
}

// una estructura que representa los caminos
static class node {
// Dos variables, una denora el nodo
// destino y otra el peso
    int to, weight;
    public node(int to, int weight) {
        this.to = to;
        this.weight = weight;
    }
}
}

```

## 20.6 Count All Paths

Cuento el número total de caminos o vías que existen entre dos vértices en un grafo dirigido, estos caminos no contienen un ciclo, la simple razón de esto es que un ciclo contiene infinito número de caminos y esto crea problema.

El problema puede ser resuelto usando backtracking, esto es si tomamos un camino y empezamos a andar por él, si nos lleva al vértice de destino entonces contamos el camino y nos devolvemos a tomar otro camino, si el camino no nos lleva al vértice destino, descartamos este camino.

```
// Programa java que cuenta todos los caminos de un
// inicio a un destino.

import java.util.Arrays;
import java.util.Iterator;
import java.util.LinkedList;

public class CountAllPaths {

    public static void main(String args[]) {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(0, 3);
        g.addEdge(2, 0);
        g.addEdge(2, 1);
        g.addEdge(1, 3);
        int s = 2, d = 3;
        System.out.println(g.countPaths(s, d));
    }

    static class Graph {

        // Número de vértices
        private int V;
        // array de listas de adyacencia
        private LinkedList<Integer> adj[];

        Graph(int v) {
            V = v;
            adj = new LinkedList[v];
            for (int i = 0; i < v; ++i) {
                adj[i] = new LinkedList<>();
            }
        }

        // Agregar caminos en un grafo
        void addEdge(int v, int w) {
            // Agrega w a las listas de v
            adj[v].add(w);
        }
    }
}
```

```

// Un metodo recursivo que cuenta
    // todos los caminos de u a d
int countPathsUtil(int u, int d,
boolean visited[],
        int pathCount) {
    //Marca el nodo actual como visitado
    //y lo imprime
    visited[u] = true;
    // Si el vertice actual es igual
    // al destino, incrementa el conteo
    if (u == d) {
        pathCount++;
    } // Recorre todos los vertices
    // adyacentes a este vertice
else {
    Iterator<Integer> i = adj[u].listIterator();
    while (i.hasNext()) {
        int n = i.next();
        if (!visited[n]) {
            pathCount = countPathsUtil(n, d,
                visited,
                pathCount);
        }
    }
    visited[u] = false;
    return pathCount;
}
// Retorna conteo de caminos desde s a d
int countPaths(int s, int d) {
//Marca todos los vertices como no visitados
boolean visited[] = new boolean[V];
    Arrays.fill(visited, false);
    int pathCount = 0;
    pathCount = countPathsUtil(s, d,
        visited,
        pathCount);
    return pathCount;
}
}
}

```

## 20.7 Cycle In A Directed Graph

Dado un grafo dirigido, verificar si el grafo contiene un ciclo o no, la función debe retornar true si el grafo dado contiene al menos un ciclo, de lo contrario retorne false, por ejemplo el siguiente grafo tiene tres ciclos,  $0 \rightarrow 2 \rightarrow 0$ ,  $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$  y  $3 \rightarrow 3$ , entonces la función debe retornar true.

DFS puede ser usado para detectar un ciclo en un grafo, DFS para un grafo conectado produce un árbol, hay un ciclo en un grafo solo si hay un camino de regreso presente en el grafo. Un camino de regreso es un camino de un nodo a sí mismo, o uno de sus antecesores en el árbol producido por el DFS.

Para un grado desconexo, tenemos el bosque DFS como salida, para detectar un ciclo, podemos verificar los arboles individuales en búsqueda de caminos de regreso.

Para detectar un camino de regreso, podemos rastrear los vértices actuales en una pila de recursión de la función de DFS transverso, si llegamos a un vértice que ya está en la pila de recursión entonces hay un ciclo en el árbol, el camino que conecta el vértice actual a un vértice en la pila recursión es el camino de regreso, Usamos recStack[] para mantener rastreado los vértices de la pila de recursión.

La complejidad de tiempo de este método es la misma complejidad de tiempo de un DFS transverso la cual es  $O(V+E)$  siendo V la cantidad de vértices y E la cantidad de caminos.

```
// Programa java que detecta ciclo en un grafo

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;

public class CycleInADirectedGraph {

    static class Graph {

        private final int V;
        private final List<List<Integer>> adj;

        public Graph(int V) {
            this.V = V;
            adj = new ArrayList<>(V);
            for (int i = 0; i < V; i++) {
                adj.add(new LinkedList<>());
            }
        }

        private boolean isCyclicUtil(int i, boolean[] visited,
                                     boolean[] recStack) {
            /* Marca el nodo actual como visitado
               y parte de la pila de recursión*/
            if (recStack[i]) {
                return true;
            }
            if (visited[i]) {
                return false;
            }
            visited[i] = true;
            recStack[i] = true;
            for (Integer j : adj.get(i)) {
                if (isCyclicUtil(j, visited, recStack)) {
                    return true;
                }
            }
            recStack[i] = false;
        }

        void printSolution() {
            System.out.println("Cyclic graph");
        }
    }
}
```

```

    }
    visited[i] = true;
    recStack[i] = true;
    List<Integer> children = adj.get(i);

    if (children.stream().anyMatch((c) -> (isCyclicUtil(c, visited,
recStack)))) {
        return true;
    }
    recStack[i] = false;
    return false;
}

private void addEdge(int source, int dest) {
    adj.get(source).add(dest);
}

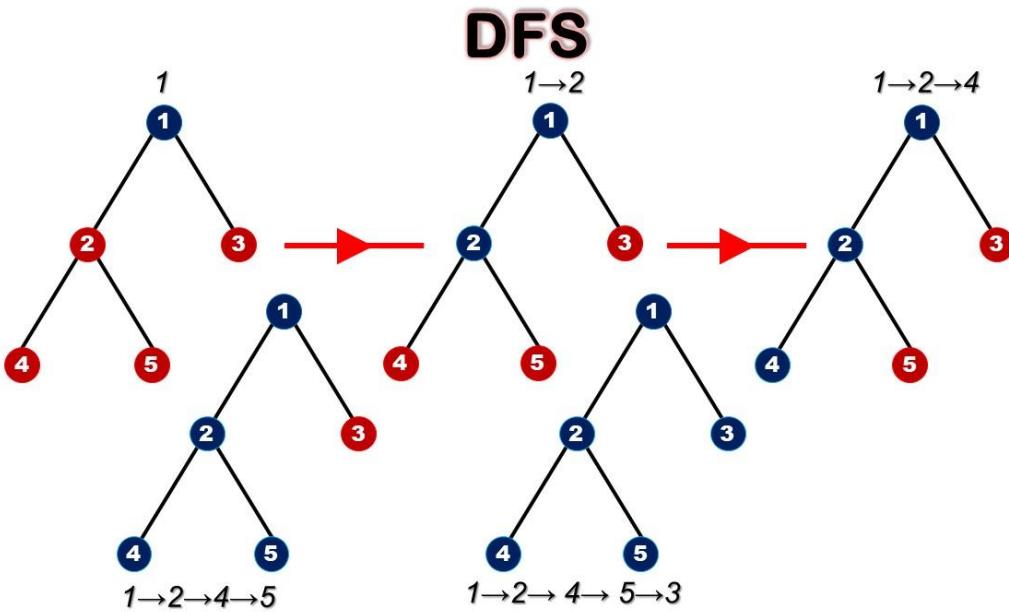
/* Retorna true si el grafo tiene un ciclo, si no falso*/
private boolean isCyclic() {
//Marca todos los vértices como no visitados
    // y no parte de la pila de recursión
boolean[] visited = new boolean[V];
    boolean[] recStack = new boolean[V];
    for (int i = 0; i < V; i++) {
        if (isCyclicUtil(i, visited, recStack)) {
            return true;
        }
    }

    return false;
}

public static void main(String[] args) {
    Graph graph = new Graph(4);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 2);
    graph.addEdge(2, 0);
    graph.addEdge(2, 3);
    graph.addEdge(3, 3);
    if (graph.isCyclic()) {
System.out.println("El grafo contiene un ciclo");
    } else {
        System.out.println("El grafo no contiene"
+ "un ciclo");
    }
}
}

```

## 20.8 DFS (Depth First Search)



*Guía del programador competitivo.*

Ilustración 20-3 Ejemplo de búsqueda en profundidad

Una Búsqueda en profundidad (en inglés DFS o Depth First Search) es un algoritmo de búsqueda no informada utilizado para recorrer todos los nodos de un grafo o árbol (teoría de grafos) de manera ordenada, pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto. Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

Completitud: DFS es completo si y solo si usamos búsqueda basada en grafos en espacios de estado finitos, pues todos los nodos serán expandidos.

Optimalidad: DFS en ningún caso asegura la optimalidad, pues puede encontrar una solución más profunda que otra en una rama que todavía no ha sido expandida.

Complejidad temporal: en el peor caso,  $O(b^m)$  siendo  $b$  el factor de ramificación (número promedio de ramificaciones por nodo) y  $m$  la máxima profundidad del espacio de estados.

Complejidad espacial:  $O(b^d)$  siendo  $b$  el factor de ramificación y  $d$  la profundidad de la solución menos costosa, pues cada nodo generado permanece en memoria, almacenándose la mayor cantidad de nodos en el nivel meta.

El tiempo de ejecución es  $O(|V|+|E|)$

```
// Programa java que imprime DFS transverso en un grafo
import java.util.*;
public class DFS {

    public static void main(String args[]) {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);
        System.out.println("Siguiendo la primera búsqueda en profundidad "
            + "(iniciando desde vértice 2)");
        g.DFS(2);
    }

    static class Graph {

        private int V; // número de vértices
        // Array de listas de adyacencia
        private LinkedList<Integer> adj[];
        Graph(int v) {
            V = v;
            adj = new LinkedList[v];
            for (int i = 0; i < v; ++i) {
                adj[i] = new LinkedList();
            }
        }

        //Agregando caminos
        void addEdge(int v, int w) {
            adj[v].add(w);
        }

        void DFSUtil(int v, boolean visited[]) {
            // Marca el nodo actual como visitado y lo imprime
            visited[v] = true;
            System.out.print(v + " ");
            // Recorre todos los vértices adyacentes a este vértice
            Iterator<Integer> i = adj[v].listIterator();
            while (i.hasNext()) {
                int n = i.next();
                if (!visited[n]) {
                    DFSUtil(n, visited);
                }
            }
        }

        void DFS(int v) {
            // Marca todos los vértices como no visitados (Falso)
            boolean visited[] = new boolean[V];
```

```

        DFSUtil(v, visited);
    }

}

```

## 20.9 DFS Transitive Clousure

Dado un grafo dirigido, encontrar si un vértice  $v$  es alcanzable desde otro vértice  $u$  para todos los pares de vértices  $(u,v)$  en el grafo dado, aquí alcanzable significa que existe un camino desde el vértice  $u$  a  $v$ , la matriz de habilidad de alcance es llamada cerramiento transitivo de un grafo.

La solución está basada en el algoritmo de Floyd Warshall,

Los pasos abstractos de este algoritmo son:

- Crear una matriz  $tc[V][V]$  que pueda tener finalmente el cerramiento transitivo de un grafo dado, inicializar todas sus entradas como 0.
- Llamar DFS por cada nodo del grado para marcar vértices alcanzables en  $tc[][]$ . En llamadas recursivas de DFS no podemos llamar DFS para un vértice adyacente si este ya fue marcado como alcanzable en  $tc[]$ .

El código usa listas de adyacencia para el grafo de entrada y construye una matriz  $tc[V][V]$  tal que  $tc[u][v]$  será true si  $v$  es alcanzable desde  $u$ .

```

//Programa java que imprime el cerramiento transitivo de un grafo
import java.util.ArrayList;
import java.util.Arrays;
public class DFSTransitiveClosure {
    public static void main(String[] args) {
        Graph g = new Graph(4);
        g.addEdge(0, 1);
        g.addEdge(0, 2);
        g.addEdge(1, 2);
        g.addEdge(2, 0);
        g.addEdge(2, 3);
        g.addEdge(3, 3);
        System.out.println("Matriz de cerramiento transitivo es ");
        g.transitiveClosure();
    }
    static class Graph {
        // número de vertices
        private int vertices;

```

```

// lista de adyacencia
    private ArrayList<Integer>[] adjList;
// para almacenar el cerramiento transitivo
    private int[][] tc;
// Constructor
    public Graph(int vertices) {
// inicializa el conteo de vertices
        this.vertices = vertices;
this.tc = new int[this.vertices][this.vertices];
// inicializa lista de adyacencia
        initAdjList();
    }
    private void initAdjList() {
        adjList = new ArrayList[vertices];
        for (int i = 0; i < vertices; i++) {
            adjList[i] = new ArrayList<>();
        }
    }
// Agregar caminos
    public void addEdge(int u, int v) {
adjList[u].add(v);
    }
// buscando cerramiento transitivo
public void transitiveClosure() {
    for (int i = 0; i < vertices; i++) {
        dfsUtil(i, i);
    }
    for (int i = 0; i < vertices; i++) {
        System.out.println(Arrays.toString(tc[i]));
    }
}
private void dfsUtil(int s, int v) {
// Marca alcance desde s a v como true
    tc[s][v] = 1;
// Encuentra todos los vertices alcanzables
// atraves de v
    for (int adj : adjList[v]) {
        if (tc[s][adj] == 0) {
dfsUtil(s, adj);
        }
    }
}
}
}

```

## 20.10 Disconnected Graph BFS

Por ejemplo asumamos que todos los vértices son alcanzables desde un vértice inicial, pero en el caso de un grafo desconexo o que cualquier vértice es inalcanzable desde todos os

vértices, un BFS normal no nos da la salida deseada, por lo que se utiliza esta modificación del BFS.

```
// Implementación de BFS modificado
import java.util.*;

public class DisconnectedGraphBFS {
    // Implementando grafo usando HashMap
    static HashMap<Integer, LinkedList<Integer>> graph = new HashMap<>();
    // Agregar caminos al grafo
    public static void addEdge(int a, int b) {
        if (graph.containsKey(a)) {
            LinkedList<Integer> l = graph.get(a);
            l.add(b);
            graph.put(a, l);
        } else {
            LinkedList<Integer> l = new LinkedList<>();
            l.add(b);
            graph.put(a, l);
        }
    }

    public static void bfshelp(int s, ArrayList<Boolean> visited) {
        // Crea una cola para el BFS
        LinkedList<Integer> q = new LinkedList<>();
        // Marca el nodo actual como visitado y lo encola
        q.add(s);
        visited.set(s, true);
        while (!q.isEmpty()) {
            // Desencola un vertice de la cola y la imprime
            int f = q.poll();
            System.out.print(f + " ");
            // Verifica cuando el nodo actual esta conectado
            // a otro nodo o no
            if (graph.containsKey(f)) {
                Iterator<Integer> i = graph.get(f).listIterator();
                // Obtiene todos los nodos adyacentes
                // del nodo desencolado f, si no ha sido visitado
                // lo marca y lo encola
                while (i.hasNext()) {
                    int n = i.next();
                    if (!visited.get(n)) {
                        visited.set(n, true);
                        q.add(n);
                    }
                }
            }
        }
    }

    // Función BFS que verifica cada nodo
    public static void bfs(int vertex) {
        ArrayList<Boolean> visited = new ArrayList<>();
        // Marcando cada nodo como no visitado
```

```

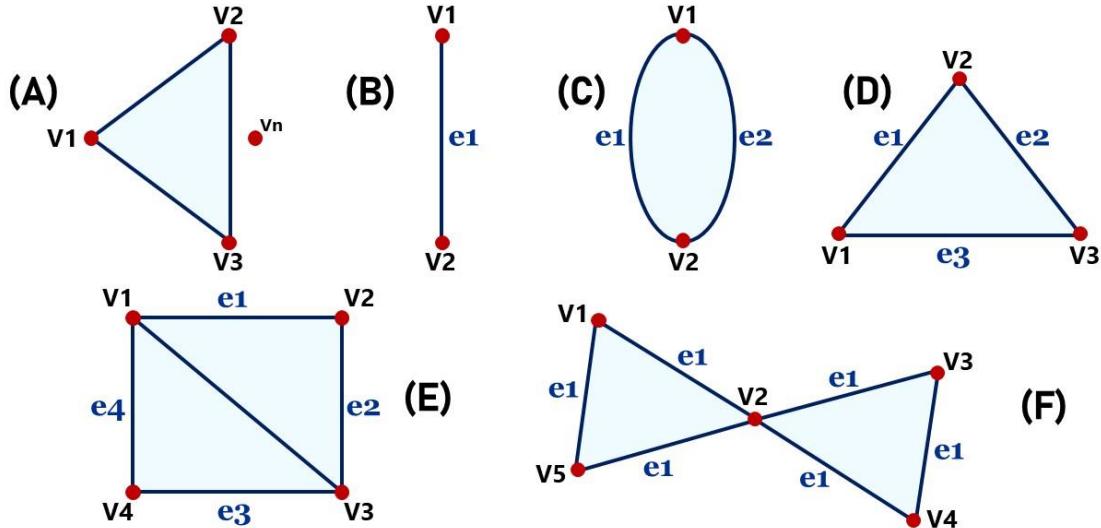
for (int i = 0; i < vertex; i++) {
    visited.add(i, false);
}
for (int i = 0; i < vertex; i++) {
//Verificando cuantos nodos no han sido visitados
if (!visited.get(i)) {
    bfshelp(i, visited);
}
}
}

public static void main(String[] args) {
int v = 5;
addEdge(0, 4);
addEdge(1, 2);
addEdge(1, 3);
addEdge(1, 4);
addEdge(2, 3);
addEdge(3, 4);
bfs(v);
}
}

```

## 20.11 Eulerian Cycle Directed Graph

### Ciclo de Euler



*Guía del programador competitivo.*

Ilustración 20-4 Ejemplo de diferentes grafos

Un camino de Euler es un camino en un grafo que visita cada arco exactamente una vez, el circuito de Euler es un camino de Euler que empieza y termina en el mismo vértice.

Un grafo es euleriano si tiene un ciclo de Euler.

Un grafo dirigido tiene un ciclo de Euler si las siguientes condiciones son verdaderas:

- 1) Todos los vértices con grado no cero pertenecen a una sola componente fuertemente conectada.
- 2) En los grados, el grado de entrada es igual al grado de salida

Podemos detectar componentes fuertemente conectadas usando el DFS de Kosaraju.

Para comparar los grados de entrada y salida, necesitamos almacenar los grados de entrada y salida de cada vértice, el grado de salida puede ser obtenido por el tamaño de la lista de adyacencia, el grado de entrada puede ser almacenado creando un array de igual tamaño al número de vértices.

La complejidad de tiempo de esta implementación es de  $O(V+E)$ , luego de correr el algoritmo de Kosaraju, atravesamos todos los vértices y comparamos los grados de salida y entrada, esto toma  $O(V)$  tiempo.

```
// Programa java que verifica si un grafo es Euleriano
import java.util.*;
import java.util.LinkedList;

public class EulerianCycleDirectedGraph {

    public static void main(String[] args) throws java.lang.Exception {
        Graph g = new Graph(5);
        g.addEdge(1, 0);
        g.addEdge(0, 2);
        g.addEdge(2, 1);
        g.addEdge(0, 3);
        g.addEdge(3, 4);
        g.addEdge(4, 0);
        if (g.isEulerianCycle()) {
            System.out.println("El grafo dado es euleriano ");
        } else {
            System.out.println("El grafo dado no es euleriano ");
        }
    }

    static class Graph {
        private int V;      // Número de vértices
```

```

private LinkedList<Integer> adj[];//Lista de adyacencia
    private int in[]; //Manteniendo los grados
//Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        in = new int[V];
        for (int i = 0; i < v; ++i) {
            adj[i] = new LinkedList();
            in[i] = 0;
        }
    }
//Agregar caminos
    void addEdge(int v, int w) {
        adj[v].add(w);
        in[w]++;
    }
    void DFSUtil(int v, Boolean visited[]) {
// Marca nodo actual como visitado
        visited[v] = true;
        int n;
// recorre todos los nodos adyacentes a este vertice
        Iterator<Integer> i = adj[v].iterator();
        while (i.hasNext()) {
            n = i.next();
            if (!visited[n]) {
                DFSUtil(n, visited);
            }
        }
    }
// Retornar el transpuesto de este grafo
Graph getTranspose() {
    Graph g = new Graph(V);
    for (int v = 0; v < V; v++) {
        Iterator<Integer> i = adj[v].listIterator();
        while (i.hasNext()) {
            g.adj[i.next()].add(v);
            (g.in[v])++;
        }
    }
    return g;
}
// Verifica si el grafo esta fuertemente conectado
Boolean isSC() {
/*Paso 1: Marca todos los vertices como
no visitados (Primer DFS)*/
    Boolean visited[] = new Boolean[V];
    for (int i = 0; i < V; i++) {
        visited[i] = false;
    }
/* Paso 2: Hace DFS transverso
    iniciando del primer vertice*/
    DFSUtil(0, visited);
// Si DFS no visita todos los nodos, retorna falso
    for (int i = 0; i < V; i++) {

```

## 20.12 Eulerian Path Undirected Graph

¿Es posible dibujar un grafo dado sin levantar el lápiz del papel y sin pasar por los caminos más de una vez?

Un grafo es llamado euleriano si tiene un ciclo de Euler y es llamado semi-euleriano si tiene un camino de Euler, el problema es similar al camino Hamiltoniano, podemos verificar si un grafo es euleriano o no en tiempo polinómico, de  $O(V+E)$ .

Las siguientes son propiedades interesantes de los grafos no dirigidos con un camino euleriano y ciclo, podemos usar estas propiedades para encontrar si un grafo es euleriano o no.

### Ciclo euleriano

Un grafo no dirigido tiene un ciclo de Euler si las siguientes dos condiciones son ciertas:

- Todos los vértices con grado no cero están conectados, no nos importa los vértices con grado cero porque no pertenecen a el ciclo de Euler o el camino, solo estamos considerando los caminos.
- Todos los vértices tienen grado par.

### Camino euleriano

Un grafo no dirigido tiene un camino euleriano si las dos siguientes condiciones se cumplen:

- La misma primera condición de un ciclo de Euler
- Si dos vértices tienen grado impar y todos los demás vértices tienen grado par, note que solo un vértice con grado impar no es posible en un grafo no dirigido, la suma de todos los grados es siempre par en un grafo no dirigido.

Note que un grafo sin caminos se considera euleriano porque no hay caminos que atravesar.

En el camino euleriano, cada vez que visitamos un vértice  $v$ , nosotros caminamos a través de dos caminos no visitados, con un punto de fin como  $v$ , por lo tanto todos los vértices medios en el camino euleriano deben tener grado par, para el ciclo euleriano cualquier vértice puede ser vértice medio, sin embargo todos los vértices deben tener grado par.

La complejidad de tiempo es:  $O(V+E)$

```
// Programa java que busca camino euleriano
// de un grafo
import java.util.*;
import java.util.LinkedList;

public class EulerianPathUndirectedGraph {
    public static void main(String args[]) {
```

```

// Creamos varios ejemplos de grafos para probar
Graph g1 = new Graph(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(2, 1);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.test();
Graph g2 = new Graph(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.addEdge(4, 0);
    g2.test();
Graph g3 = new Graph(5);
    g3.addEdge(1, 0);
    g3.addEdge(0, 2);
    g3.addEdge(2, 1);
    g3.addEdge(0, 3);
    g3.addEdge(3, 4);
    g3.addEdge(1, 3);
g3.test();
/* Crearemos un grafo con tres vertices
   conectados en forma de ciclo*/
Graph g4 = new Graph(3);
    g4.addEdge(0, 1);
    g4.addEdge(1, 2);
    g4.addEdge(2, 0);
    g4.test();
/* Creamos un grafo con vertices con grado cero*/
Graph g5 = new Graph(3);
    g5.test();
}

static class Graph {
    private int V;
    private LinkedList<Integer> adj[];
// Constructor
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i) {
            adj[i] = new LinkedList();
        }
    }
//Aregar caminos
    void addEdge(int v, int w) {
        adj[v].add(w);
            adj[w].add(v); //El grafo es no dirigido
    }
    void DFSUtil(int v, boolean visited[]) {
        visited[v] = true;
        Iterator<Integer> i = adj[v].listIterator();

```

```

        while (i.hasNext()) {
            int n = i.next();
            if (!visited[n]) {
                DFSUtil(n, visited);
            }
        }
    }
    boolean isConnected() {
        boolean visited[] = new boolean[V];
        int i;
        for (i = 0; i < V; i++) {
            visited[i] = false;
        }
        for (i = 0; i < V; i++) {
            if (!adj[i].isEmpty()) {
                break;
            }
        }
        //Si no hay caminos en el grafo, retorna true
        if (i == V) {
            return true;
        }
        DFSUtil(i, visited);
        for (i = 0; i < V; i++) {
            if (visited[i] == false && adj[i].size() > 0) {
                return false;
            }
        }
        return true;
    }

    int isEulerian() {
        if (isConnected() == false) {
            return 0;
        }
        // Cuenta vertices con grado impar
        int odd = 0;
        for (int i = 0; i < V; i++) {
            if (adj[i].size() % 2 != 0) {
                odd++;
            }
        }
        // Si cuenta es más de 2, el grafo no es euleriano
        if (odd > 2) {
            return 0;
        }
        //si odd es 2, es semieuleriano
        //Si odd es 0, es euleriano
        return (odd == 2) ? 1 : 2;
    }

    void test() {
        int res = isEulerian();
        switch (res) {

```

```
        case 0:
            System.out.println("Grafo no es euleriano");
            break;
        case 1:
            System.out.println("Grafo tiene un camino de euler");
            break;
        default:
            System.out.println("Grafo tiene ciclo de euler");
            break;
    }
}
```

## 20.13 Floyd-Warshall All Pairs Shortest Paths

El algoritmo de Floyd Warshall se usa para la resolución de todos los caminos más cortos de todos los pares, el problema es encontrar las distancias más pequeñas entre cada par de vértices dado un grafo de caminos con pesos.

Inicializamos la matriz de la solución igual que la matriz de entrada del grafo en el primer paso, luego actualizamos la matriz de solución considerando todos los vértices en un vértice intermedio, la idea es uno por uno tomar todos los vértices y actualizar todos los caminos más cortos los cuales incluyen el vértice seleccionado como el vértice intermedio en el camino más corto. Cuando tomamos el vértice de número  $k$  como un vértice intermedio ya estamos considerando vértices  $\{0,1,2,\dots,k-1\}$  como intermedios, para cada par  $(i,j)$  del origen al destino respectivamente existen dos posibles casos.

- 1)  $k$  no es un vértice intermedio en el camino más corto de  $i$  a  $j$ , mantenemos el valor de  $\text{dist}[i][j]$  como esta.
  - 2)  $K$  es un vértice intermedio en el camino más corto de  $i$  a  $j$ , actualizamos el valor de  $\text{dist}[i][j]$  como  $\text{dist}[i][k] + \text{dist}[k][j]$  if  $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$

La complejidad de tiempo es:  $O(V^3)$  o  $O(N^3)$ .

El programa solo imprime las distancias más cortas, podemos modificar la solución para imprimir el camino más corto ordenando la información del predecesor en una matriz separada.

El valor de INF puede ser tomado de INT\_MAX o Integer.MAX\_VALUE, lo que nos permite manejar los valores máximos posibles.

```
// Programa java que busca todos los caminos más cortos
// en un grafo

import java.util.Scanner;

public class FloydWarshal {

    static int INF = 9999;

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int[][] graph;
        int N = sc.nextInt();
        graph = new int[N][N];
        int E = sc.nextInt();
        //Inicializa todo en infinito
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                graph[i][j] = INF;
            }
        }
        // De un nodo al mismo nodo es 0
        for (int i = 0; i < N; i++) {
            graph[i][i] = 0;
        }
        for (int i = 0; i < E; i++) {
            graph[sc.nextInt()][sc.nextInt()] = sc.nextInt();
        }
        AllPairShortestPath a = new AllPairShortestPath();
        a.floydWarshall(graph);
    }

    static class AllPairShortestPath {

        final static int INF = 9999, V = 4;

        void floydWarshall(int graph[][]){
            int dist[][] = new int[V][V];
            int i, j, k;
            /*Inicialice la matriz de solución igual que
            la matriz del gráfico de entrada. O podemos
            decir que los valores iniciales de las distancias
            más cortas se basan en las rutas más cortas,
            considerando que no hay vértice intermedio. */
            for (i = 0; i < V; i++) {
                for (j = 0; j < V; j++) {
                    dist[i][j] = graph[i][j];
                }
            }
            /*Agregue todos los vértices uno por uno al conjunto
```

20.14 Hamiltonian Cycle

# Ciclo de Hamilton

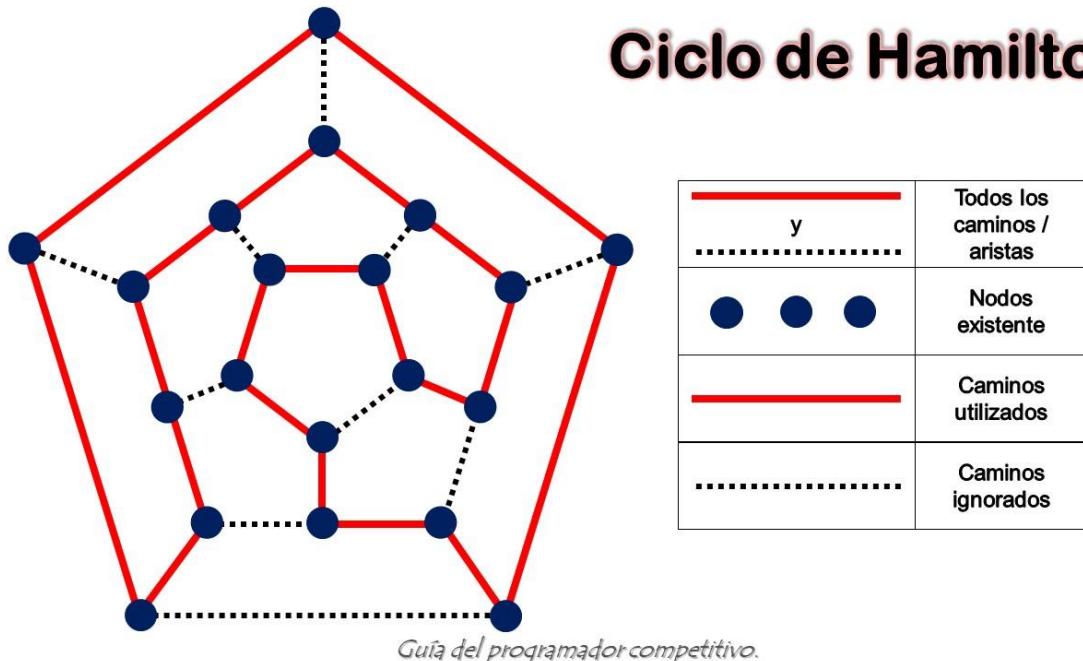


Ilustración 20-5 Ciclo Hamiltoniano en un grafo

El camino Hamiltoniano de un grafo no dirigido es un camino que visita cada vértice exactamente una vez, un ciclo Hamiltoniano es un camino Hamiltoniano que tiene un camino desde el último vértice al primer vértice del camino Hamiltoniano, debemos determinar si un grafo tiene ciclo Hamiltoniano o no, si lo contiene imprimir el camino.

Nuestra entrada será un array `graph[v][v]` donde `v` es el número de vértices en el grafo y `graph[][]` es la matriz de adyacencia representando el grafo, un valor `graph[i][j]` es 1 si existe un camino directo de `i` a `j`, de lo contrario será 0

Nuestra salida será un array `path[v]` que deberá contener el camino Hamiltoniano, `path[i]` puede representar el `i`ésimo vértice en el camino Hamiltoniano. El código debe también retornar si no existe ciclo Hamiltoniano en el grafo.

Se crea un array de camino vacío y se le agrega el vértice 0 a él, se agregan los otros vértices iniciando desde el vértice 1, antes de añadir un vértice, se verifica por cual es el adyacente del anterior añadido y si no se ha añadido ya, si encontramos tal vértice, añadimos el vértice como parte de la solución, si no lo encontramos retornamos false.

Note que el código siempre imprime el ciclo iniciando de 0, el punto de inicio no importa ya que el ciclo puede empezar de cualquier punto, si se quiere cambiar el punto de inicio, se deben hacer dos cambios al código de abajo.

Cambie "path[0] = 0;" por "path[0] = s;" donde s es el nuevo punto de inicio, también cambie el ciclo "for (int v = 1; v < V; v++)" en hamCycleUtil() por "for (int v = 0; v < V; v++)".

```
/* Programa java que soluciona el ciclo de Hamilton
usando backtracking*/

public class HamiltonianCycle {
    static final int V = 5;
    static int path[];

    /* Una función de utilidad para comprobar si el vértice v puede ser
       añadido en el índice 'pos' en el ciclo hamiltoniano
       construido hasta ahora (almacenado en 'path[]')*/
    static boolean isSafe(int v, int graph[][], int path[], int pos) {
        /*Verifica si este vertice es adyacente del
        anterior vertice */
        if (graph[path[pos - 1]][v] == 0) {
            return false;
        }
        /* Verifica si el vertice ya esta incluido*/
        for (int i = 0; i < pos; i++) {
            if (path[i] == v) {
                return false;
            }
        }
        return true;
    }
    static boolean hamCycleUtil(int graph[][], int path[], int pos) {
        /* Caso base: Si todos los vertices estan incluidos en
           el ciclo hamiltoniano*/
        if (pos == V) {
            // Y si hay un camino de el ultimo al primer vertice
            return graph[path[pos - 1]][path[0]] == 1;
        }
        /*Prueba diferentes vértices como próximo candidato
           en el ciclo hamiltoniano. No intentamos con 0,
           ya que incluimos 0 como punto de partida en hamCycle ()*/
        for (int v = 1; v < V; v++) {
            /* Verifica si este vertice puede ser añadido al
               ciclo hamiltoniano*/
            if (isSafe(v, graph, path, pos)) {
                path[pos] = v;
                /* Recorre hasta construir el camino*/
                if (hamCycleUtil(graph, path, pos + 1) == true) {
                    return true;
                }
                path[pos] = -1;
            }
        }
    }
}
```

```

        }
    /* Si no hay vertice para añadir al ciclo
       retorna falso*/
    return false;
}
/* Esta función resuelve el problema del ciclo hamiltoniano usando
   backtracking. Utiliza principalmente hamCycleUtil () para resolver el
   problema. Devuelve falso si no hay ciclo hamiltoniano.
   posible, de lo contrario devuelve verdadero e imprime la ruta.
   Tenga en cuenta que puede haber más de una solución,
   Esta función imprime una de las soluciones factibles.*/
static int hamCycle(int graph[][]) {
    path = new int[V];
    for (int i = 0; i < V; i++) {
path[i] = -1;
    }
/* Pongamos el vértice 0 como el primer vértice en el camino.
   Si hay un ciclo hamiltoniano, entonces el camino puede ser
   Comenzó desde cualquier punto del ciclo ya que la gráfica es
   no dirigido*/
    path[0] = 0;
    if (hamCycleUtil(graph, path, 1) == false) {
System.out.println("\nNo existe solución");
return 0;
    }
    printSolution(path);
    return 1;
}

// Imprimir solución
static void printSolution(int path[]) {
System.out.println("Solución existe: Este"
    + " es uno de los ciclos hamiltoniano");
for (int i = 0; i < V; i++) {
    System.out.print(" " + path[i] + " ");
}
System.out.println(" " + path[0] + " ");
}

public static void main(String args[]) {

/*Tenemos el siguiente grafo
    (0)--(1)--(2)
      |   / \ |
      |   / \ |
      |   / \ |
    (3)-----(4) */
    int graph1[][] = {{0, 1, 0, 1, 0},
{1, 0, 1, 1, 1},
{0, 1, 0, 0, 1},
{1, 1, 0, 0, 1},
{0, 1, 1, 1, 0},};
// Imprimir solución
hamCycle(graph1);
}

```

```

/*Tenemos el siguiente grafo
(0)---(1)---(2)
      |   / \ |
      |  /   \ |
(3)    (4)  */
int graph2[][] = {{0, 1, 0, 1, 0},
{1, 0, 1, 1, 1},
{0, 1, 0, 0, 1},
{1, 1, 0, 0, 0},
{0, 1, 1, 0, 0},};
// Imprimir solución
hamCycle(graph2);
}
}

```

## 20.15 Kahn Algorithm For Topological Sorting

Ordenamiento topológico para un grafo dirigido acíclico (DAG) es un ordenamiento lineal de vértices el cual por cada camino dirigido UV, el vértice U viene antes de V en el ordenamiento, el ordenamiento topológico no es posible si el grafo no es un DAG.

Los pasos de este algoritmo son los siguientes:

- 1) Calcule los grados de entrada de cada vértice en el DAG presente e inicialice el conteo de los nodos visitados como 0.
- 2) Tome todos los vértices con grado de entrada como 0 y agréguelos en una cola
- 3) Remueva un vértice de la cola y entonces incremente el conteo de nodos visitados por 1
- 4) Decremente el grado de entrada en 1 en todos los nodos vecinos, si el grado de entrada de los nodos vecinos es cero, agréguelos a la cola.
- 5) Repita el paso 3 hasta que la cola este vacía.
- 6) Si el conteo de nodos visitados no es igual al número de nodos en el grado entonces el ordenamiento topológico no es posible en este grafo.

### ¿Cómo encontrar el grado de entrada en cada nodo?

Existen dos vías para encontrar el grado de entrada de cada nodo.

Tomaremos un array de grado de entrada para mantener rastreo de estos.

1) Atraviese el array de nodos y simplemente incremente el conteo del nodo de destino en 1-

for each node in Nodes

    indegree[node] = 0;

for each edge(src,dest) in Edges

    indegree[dest]++

2) Atraviese la lista de cada nodo e incremente el grado de entrada de todos los nodos conectados con él en 1

for each node in Nodes

    If (list[node].size()!=0) then

        for each dest in list

            indegree[dest]++;

La complejidad de tiempo se da por: el ciclo externo será ejecutado V número de veces y el interno será ejecutado E número de veces, la complejidad resultante será de O(V+E).

```
// Programa java que imprime el ordenamiento topológico
// de un grafo
import java.util.*;

public class KahnAlgorithmTopologicalSorting {

    public static void main(String args[]) {
        Graph g = new Graph(6);
        g.addEdge(5, 2);
        g.addEdge(5, 0);
        g.addEdge(4, 0);
        g.addEdge(4, 1);
        g.addEdge(2, 3);
        g.addEdge(3, 1);
        System.out.println("Siguiendo su ordenamiento topológico ");
        g.topologicalSort();
    }

    static class Graph {
        int V;
        List<Integer> adj[];
        public Graph(int V)// Constructor
        {
            this.V = V;
            adj = new ArrayList[V];
            for (int i = 0; i < V; i++) {
                adj[i] = new ArrayList<>();
```

```

        }
    }
// Agregar caminos
    public void addEdge(int u, int v) {
        adj[u].add(v);
    }

    public void topologicalSort() {
// Crea un array para almacenar los indegrees
// de todos los vertices e inicializa en 0
int indegree[] = new int[V];
// Atraviesa las listas de adyacencia para llenar
// grados de los vertices
        for (int i = 0; i < V; i++) {
            ArrayList<Integer> temp = (ArrayList<Integer>) adj[i];
            temp.forEach((node) -> {
                indegree[node]++;
            });
        }
/* Crea una cola y encola todos los vertices
con grado 0*/
        Queue<Integer> q = new LinkedList<>();
        for (int i = 0; i < V; i++) {
            if (indegree[i] == 0) {
                q.add(i);
            }
        }
// Inicializa el conteo de vertices visitados
        int cnt = 0;
// Crea un vector que almacena el resultado
ArrayList<Integer> topOrder = new ArrayList<Integer>();
        while (!q.isEmpty()) {
            int u = q.poll();
            topOrder.add(u);
            adj[u].stream().filter((node) -> (--indegree[node] ==
0)).forEachOrdered((node) -> {
                q.add(node);
            });
            cnt++;
        }
// Verifica si hay ciclo
if (cnt != V) {
            System.out.println("There exists a cycle in the graph");
            return;
        }
//Imprime el ordenamiento topologico
        for (int i : topOrder) {
            System.out.print(i + " ");
        }
    }
}

```

## 20.16 Knight Tour Backtracking

El backtracking es un paradigma algorítmico que intenta diferentes soluciones hasta que encuentra una solución que “Funciona”. Problemas los cuales son típicamente resueltos con técnicas de backtracking tienen una propiedad en común, estos problemas puede solo ser resueltos intentando cada posible configuración y cada configuración es intentada una sola vez, una solución ingenua para estos problemas es intentar todas las configuraciones e imprimir una configuración que siga las restricciones dadas por el problema. Backtracking funciona en forma incremental y es una optimización sobre las soluciones ingenuas donde todas las posibles configuraciones son generadas e intentadas.

Backtracking funciona de forma incremental para atacar problemas, típicamente iniciamos desde un vector de solución vacío, y uno por uno vamos agregando ítems, cuando agregamos un ítem podemos verificar si agregando el siguiente ítem violamos alguna de las restricciones del problema, si lo hace eliminamos ese elemento e intentamos otras alternativas. Si ninguna de las alternativas funciona entonces volvemos a la fase previa y removemos el ítem anterior dado en la fase anterior. Si alcanzamos la fase inicial entonces decimos que no existe solución, si agregamos un valor que no viola alguna restricción entonces recursivamente agregamos ítems uno por uno, si el vector de solución se completa imprimimos la solución.

El siguiente es el backtracking del problema del tour del caballo en el tablero de ajedrez.

- Si todos los cuadrados son visitados imprima la solución
- Si no
  - a) Agregue uno de los siguientes movimientos posibles al vector de solución y recursivamente verifique si este movimiento lleva a una solución (Un caballo puede hacer máximo 8 movimientos, aquí escogemos alguno de esos 8 movimientos).
  - b) Si el movimiento escogido arriba no lleva a una solución entonces removemos este movimiento del vector de solución e intentamos otros movimientos alternativos.

- c) Si ninguna de las alternativas funciona, retornamos falso (Retornando falso podemos remover el anterior ítem agregado en recursión y si el falso es retornado a la recursión inicial entonces no existe solución.

```

Programa java para el problema del tour del caballo
public class KnightTourBacktracking {
    static int N = 8;
    /* Una función que verifica si i,j es
       indice valido para un tablero n*n*/
    static boolean isSafe(int x, int y, int sol[][]) {
        return (x >= 0 && x < N && y >= 0
        && y < N && sol[x][y] == -1);
    }
    /* Imprimir la solución*/
    static void printSolution(int sol[][]) {
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++) {
                System.out.print(sol[x][y] + "\t");
            }
            System.out.println();
        }
    }
    /*Esta función resuelve el problema de Knight Tour.
       utilizando Backtracking. Esta función principalmente
       utiliza solveKTUtil () para resolver el problema. Eso
       devuelve falso si no es posible realizar un recorrido completo,
       De lo contrario, devuelve true e imprime el recorrido.
       Tenga en cuenta que puede haber más de una
       soluciones, esta función imprime una de las
       soluciones viables.*/
    static boolean solveKT() {
        int sol[][] = new int[8][8];
        /* Inicializando la matriz de solución*/
        for (int x = 0; x < N; x++) {
            for (int y = 0; y < N; y++) {
                sol[x][y] = -1;
            }
        }
        /* xMove[] y yMove[] define el siguiente movimiento del caballo
           xMove[] para siguiente valor en x
           yMove[] para siguiente valor en y */
        int xMove[] = {2, 1, -1, -2, -2, -1, 1, 2};
        int yMove[] = {1, 2, 2, 1, -1, -2, -2, -1};
        //Desde que el caballero inicie en el primer bloque
        sol[0][0] = 0;
        if (!solveKTUtil(0, 0, 1, sol, xMove, yMove)) {
            System.out.println("No existe la solución");
            return false;
        } else {
            printSolution(sol);
        }
    }
}

```

```

        }
        return true;
    }
    static boolean solveKTUtil(int x, int y, int movei,
        int sol[][], int xMove[],
        int yMove[]) {
        int k, next_x, next_y;
        if (movei == N * N) {
            return true;
        }
        /* Intenta todos los movimientos desde la coordenada
        x y y*/
        for (k = 0; k < 8; k++) {
            next_x = x + xMove[k];
            next_y = y + yMove[k];
            if (isSafe(next_x, next_y, sol)) {
                sol[next_x][next_y] = movei;
                if (solveKTUtil(next_x, next_y, movei + 1,
                    sol, xMove, yMove)) {
                    return true;
                } else {
                    sol[next_x][next_y] = -1;// backtracking
                }
            }
        }
        return false;
    }

    public static void main(String args[]) {
solveKT();
    }
}

```

## 20.17 Kosaraju DFS Strongly Connected Graph

Dado un grafo dirigido, encontrar si el grafo se encuentra fuertemente conectado o no, un grado es fuertemente conectado si hay un camino entre cualquier par de vértices.

Esto es fácil para un grafo no dirigido, solo tenemos que hacer BFS y DFS comenzando desde cualquier vértice, si BFS o DFS visita todos los vértices, entonces el grado no dirigido dado está conectado, esta aproximación no funciona con un grafo dirigido.

Una simple idea es usar un algoritmo de todos los caminos más cortos entre todos los pares como Floyd Warshall o encontrar el cerramiento transitivo del grafo, la complejidad de tiempo de este método es de  $O(v^3)$ .

Podemos también hacer DFS V veces iniciando desde cada vértice, si alguno de los DFS no visita todos los vértices entonces el grafo no es fuertemente conectado. Este algoritmo toma  $O(V^*(V+E))$  en complejidad de tiempo, el cual puede ser el mismo del cerramiento transitivo para un grafo denso.

Una mayor idea puede ser el algoritmo de componentes fuertemente conectadas (SCC), podemos encontrar todos los SCC en  $O(V+E)$  tiempo, si el número de SCC es 1, entonces el grafo es fuertemente conectado, el algoritmo de SCC hace trabajo extra cuando busca todos los SCC.

El siguiente es un algoritmo simple basado en el DFS de Kosaraju el cual realiza dos DFS transversos en el grafo.

- 1) Inicializa todos los vértices como no visitados.
- 2) Hace un DFS transverso del grado iniciando desde cualquier vértice v, si este DFS transverso no visita todos los vértices, entonces retorna falso.
- 3) Reversa todos los arcos (O encuentra transpuesta o reversa de grafo)
- 4) Marque todos los vértices como no visitados en el grafo reverso
- 5) Hacer un DFS transverso del grafo reversado iniciando desde el mismo vértice v, si el DFS transverso no visita todos los vértices entonces retorna falso, de otro modo retorna true.

La idea es, si cada nodo puede ser alcanzado de un vértice v, y cada nodo puede alcanzar v, entonces el grafo es fuertemente conectado. En el paso dos podemos verificar si todos los vértices son alcanzables desde v, en el paso 4 verificamos si todos los vértices pueden alcanzar v (En el grafo reverso, si todos los vértices son alcanzables desde v entonces todos los vértices pueden alcanzar v en el grafo original).

La complejidad de tiempo de esta implementación es la misma de DFS, el cual es  $O(V+E)$  si el grafo está representado usando listas de adyacencia.

```
// Programa java que verifica si un grafo es
// fuertemente conectado
import java.util.*;
import java.util.LinkedList;

public class KosarajuDFSStronglyConnected {
```

```

public static void main(String args[]) {
    Graph g1 = new Graph(5);
    g1.addEdge(0, 1);
    g1.addEdge(1, 2);
    g1.addEdge(2, 3);
    g1.addEdge(3, 0);
    g1.addEdge(2, 4);
    g1.addEdge(4, 2);
    if (g1.isSC()) {
        System.out.println("Si");
    } else {
        System.out.println("No");
    }
    Graph g2 = new Graph(4);
    g2.addEdge(0, 1);
    g2.addEdge(1, 2);
    g2.addEdge(2, 3);
    if (g2.isSC()) {
        System.out.println("Si");
    } else {
        System.out.println("No");
    }
}

static class Graph {
    private int V;
    private LinkedList<Integer> adj[];
    Graph(int v) {
        V = v;
        adj = new LinkedList[v];
        for (int i = 0; i < v; ++i) {
            adj[i] = new LinkedList();
        }
    }
    void addEdge(int v, int w) {
        adj[v].add(w);
    }
    void DFSUtil(int v, Boolean visited[]) {
        visited[v] = true;
        int n;
        Iterator<Integer> i = adj[v].iterator();
        while (i.hasNext()) {
            n = i.next();
            if (!visited[n]) {
                DFSUtil(n, visited);
            }
        }
    }
    // Obtiene el transpuesto de un grafo
    Graph getTranspose() {
        Graph g = new Graph(V);
        for (int v = 0; v < V; v++) {
            Iterator<Integer> i = adj[v].listIterator();
            while (i.hasNext()) {

```

```

                g.adj[i.next()].add(v);
            }
        }
        return g;
    }
    //Verifica si el grafo esta fuertemente conectado
    Boolean isSC() {
        /*Paso 1: marcar todos los vertices como no visitados
        (primer DFS)*/
        Boolean visited[] = new Boolean[V];
        for (int i = 0; i < V; i++) {
            visited[i] = false;
        }
        //Paso 2: DFS transverso desde el primer vertice
        DFSUtil(0, visited);
        // si no visita todos, retorna falso
        for (int i = 0; i < V; i++) {
            if (visited[i] == false) {
                return false;
            }
        }
        // Paso 3: crear grafo transpose
        Graph gr = getTranspose();
        // Paso 4: marcar todos los vertices como no visitados
        // (Segundo DFS)
        for (int i = 0; i < V; i++) {
            visited[i] = false;
        }
        /* Paso 5: hacer DFS en el grafo reversado*/
        gr.DFSUtil(0, visited);
        /* Si todos los vertices no son visitados, retorne falso*/
        for (int i = 0; i < V; i++) {
            if (visited[i] == false) {
                return false;
            }
        }
        return true;
    }
}

```

## 20.18 Kruskal Minimum Spanning Tree

## MST

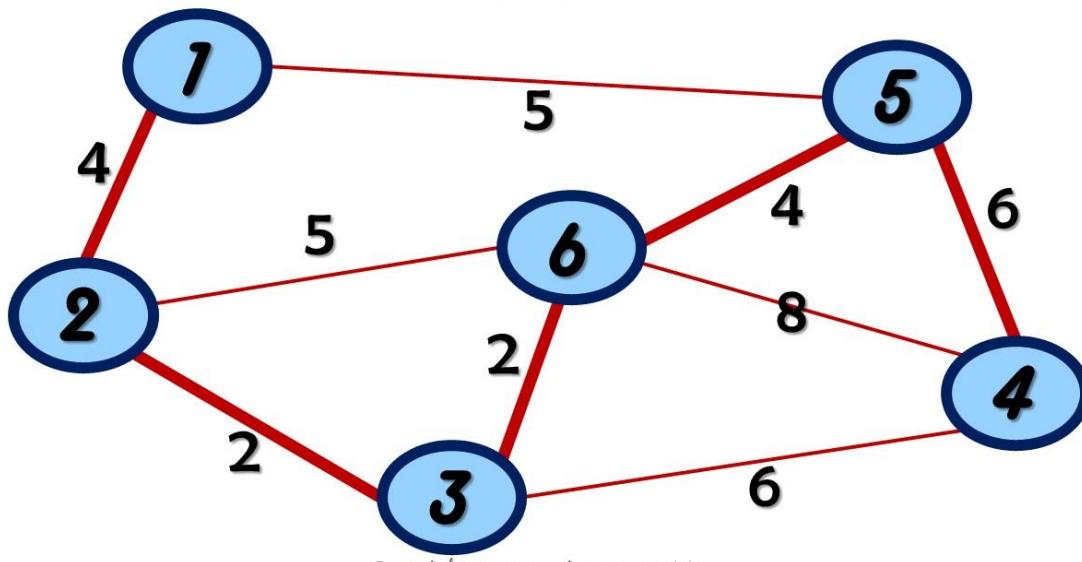


Ilustración 20-6 Árbol de expansión mínima en un grafo

Dado un grafo conexo y no dirigido, un árbol de expansión del grado es el subgrafo que es el árbol que conecta todos los vértices juntos, un solo grafo puede tener muchos árboles de expansión diferentes, un árbol de expansión mínima (MST) o árbol de expansión de peso mínimo para un grafo ponderado, conexo y no dirigido es el árbol de expansión con el peso menos o igual a cualquier otro árbol de expansión. El peso de un árbol de expansión es la suma de los pesos dados en cada camino del árbol de expansión.

¿Cuántos caminos tiene un árbol de expansión mínima?

Un MST tiene  $(V-1)$  caminos donde  $V$  es el número de vértices en el grafo dado.

Los siguientes son los pasos para encontrar el MST con el algoritmo de Kruskal.

- 1) ordene todos los caminos en orden incremental dado su peso.
- 2) Tome el camino más pequeño, verifique si forma un ciclo con el árbol formado hasta ahora, si el ciclo no es formado, incluya este camino, si no descártelo
- 3) Repita el paso 2 hasta que haya  $(V-1)$  caminos en el árbol de expansión.

El paso 2 usa el algoritmo Union-Find para detectar ciclos.

El algoritmo es un algoritmo voraz, la voracidad escogida es tomar el camino de tamaño más pequeño lo que no causa un ciclo en el MST construido hasta ese punto

La complejidad de tiempo de este algoritmo es de  $O(E\log E)$  or  $O(E\log V)$ . Ordenar los caminos toma  $O(E\log E)$  tiempo, luego de ordenar iteramos a través de todos los caminos y aplicamos el algoritmo Union-Find, Las operaciones de unir y encontrar puede tomar al menos  $O(\log V)$  tiempo, entonces la complejidad completa es  $O(E\log E + E\log V)$  tiempo. El valor de W puede ser al menos  $O(V^2)$  entonces  $O(\log V)$  son  $O(\log E)$  iguales.

```
//Implementación java de busqueda del
// arbol de expansión mínima usando el algoritmo de Kruskal
/*
EJEMPLO DE INPUT
9 14
1 2 4
1 8 9
2 3 9
2 8 11
3 4 7
3 9 2
3 6 4
4 5 10
4 6 15
5 6 11
6 7 2
7 8 1
7 9 6
8 9 7

EJEMPLO VERIFICACION DE MST
9 11
1 2 4
1 8 9
2 3 9
2 8 11
3 9 2
7 8 1
7 9 6
8 9 7
4 5 10
4 6 15
5 6 11
*/
import java.util.Arrays;
import java.util.Comparator;
import java.util.Scanner;

public class KruskalMST {

    static final int MAX = 1005; //maximo número de vértices
```

```

///UNION-FIND
static int padre[] = new int[MAX]; //Este arreglo contiene el padre del i-
esimo nodo

//Método de inicialización
static void MakeSet(int n) {
    for (int i = 1; i <= n; ++i) {
        padre[i] = i;
    }
}

//Método para encontrar la raíz del vértice actual X
static int Find(int x) {
    return (x == padre[x]) ? x : (padre[x] = Find(padre[x]));
}

//Método para unir 2 componentes conexas
static void Union(int x, int y) {
    padre[Find(x)] = Find(y);
}

//Método que me determina si 2 vértices están o no en la misma componente conexa
static boolean sameComponent(int x, int y) {
    if (Find(x) == Find(y)) {
        return true;
    }
    return false;
}
///FIN UNION-FIND

static int V, E; //número de vértices y aristas
//Estructura arista( edge )

    static class Edge implements Comparator<Edge> {

        int origen; //Vértice origen
        int destino; //Vértice destino
        int peso; //Peso entre el vértice origen y destino

        Edge() {
        }
        //Comparador por peso, me servirá al momento de ordenar lo realizara en orden
        //ascendente
        //Ordenar de forma descendente para obtener el árbol de expansión máxima

        @Override
        public int compare(Edge e1, Edge e2) {
            //return e2.peso - e1.peso; //Árbol de expansión máxima
            return e1.peso - e2.peso; //Árbol de expansión mínima
        }
    };

    static Edge arista[] = new Edge[MAX]; //Arreglo de aristas para el uso en
    kruskal

```

```

static Edge MST[] = new Edge[MAX]; //Arreglo de aristas del MST encontrado

    static void KruskalMST() {
int origen, destino, peso;
    int total = 0; //Peso total del MST
    int numAristas = 0; //Número de Aristas del MST

    MakeSet(V); //Inicializamos cada componente
    Arrays.sort(arista, 0, E, new Edge()); //Ordenamos las aristas por su
comparador
for (int i = 0; i < E; ++i) {
    origen = arista[i].origen; //Vértice origen de la arista actual
    destino = arista[i].destino; //Vértice destino de la arista actual
    peso = arista[i].peso; //Peso de la arista actual

//Verificamos si estan o no en la misma componente conexa
    if (!sameComponent(origen, destino)) { //Evito ciclos
        total += peso; //Incremento el peso total del MST
        MST[numAristas++] = arista[i]; //Agrego al MST la arista actual
        Union(origen, destino); //Union de ambas componentes en una sola
    }
}

//Si el MST encontrado no posee todos los vértices mostramos mensaje de error
//Para saber si contiene o no todos los vértices basta con que el número
//de aristas sea igual al número de vertices - 1
if (V - 1 != numAristas) {
    System.out.println("No existe MST valido para el grafo ingresado, el
grafo debe ser conexo.");
    return;
}
System.out.println("----El MST encontrado contiene las siguientes
aristas----");
for (int i = 0; i < numAristas; ++i) {
    System.out.printf("( %d , %d ) : %d\n", MST[i].origen,
MST[i].destino, MST[i].peso);
}
System.out.printf("El costo minimo de todas las aristas del MST es :
%d\n", total);
}

//( vertice u , vertice v ) : peso
public static void main(String[] args) {
Scanner sc = new Scanner(System.in); //para lectura de datos
V = sc.nextInt();
E = sc.nextInt();

//Realizamos el ingreso del grafo, almacenando las aristas en un arreglo con los
datos respectivos
for (int i = 0; i < E; ++i) {
    arista[i] = new Edge();
    arista[i].origen = sc.nextInt();
    arista[i].destino = sc.nextInt();
    arista[i].peso = sc.nextInt();
//arista[ i ] = new Arista( sc.nextInt() , );
}

```

```

    }
    KruskalMST();
}
}

```

## 20.19 Minimum Moves Knight

Dado una mesa de ajedrez cuadrada de tamaño NxN, la posición de un caballo y la posición objetivo, necesitamos encontrar la mínima cantidad de pasos que un caballo toma para llegar a la posición objetivo.

Este problema puede verse como el camino más corto en un grafo sin pesos, sin embargo usamos BFS para resolver este problema, intentamos todos las 8 posibles posiciones donde un caballo puede llegar desde su posición, si la posición alcanzable no ha sido visitada ya y está dentro del tablero, agregamos este estado dentro de la cola con una distancia de 1 más que su estado padre, finalmente retornamos la distancia de la posición objetivo cuando sale de la cola.

El siguiente código implementa BFS para la búsqueda a través de las celdas, donde cada celda contiene sus coordenadas y distancias desde el nodo inicial, en el peor de los casos el código visita todas las celdas del tablero, haciendo que el la complejidad del peor de los casos sea  $O(n^2)$ .

```

//Programa java que encuentra la minima cantidad de pasos
// para alcanzar una celda especifica con un caballo
import java.util.Vector;

public class MinimumMovesKnight {
    /*Clase que almacena los datos de una celda*/
    static class cell {
        int x, y;
        int dis;
        public cell(int x, int y, int dis) {
            this.x = x;
            this.y = y;
            this.dis = dis;
        }
    }
    /* Utilidad que retorna s (x,y) yace dentro del tablero*/
    static boolean isInside(int x, int y, int N) {
        return x >= 1 && x <= N && y >= 1 && y <= N;
    }
    /* Retorna los minimos pasos para llegar al objetivo*/

```

```

static int minStepToReachTarget(int knightPos[], int targetPos[],
int N) {
    // dirección x y y, donde el caballo puede llegar
    int dx[] = {-2, -1, 1, 2, -2, -1, 1, 2};
    int dy[] = {-1, -2, -2, -1, 1, 2, 2, 1};
    // Vector para almacenar los estados del caballo
    Vector<cell> q = new Vector<>();
    // Agrega la posicion inicial con distancia 0
    q.add(new cell(knightPos[0], knightPos[1], 0));
    cell t;
    int x, y;
    boolean visit[][] = new boolean[N + 1][N + 1];
    //Hacer todas las celdas sin visitar
    for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
            visit[i][j] = false;
        }
    }

    // visitar estado inicial
    visit[knightPos[0]][knightPos[1]] = true;
    // ciclo hasta que quede solo un valor
    while (!q.isEmpty()) {
        t = q.firstElement();
        q.remove(0);
        /* Si la celda actual es igual al objetivo
        retorne su distancia*/
        if (t.x == targetPos[0] && t.y == targetPos[1]) {
            return t.dis;
        }
        // Ciclo de todos los estados alcancables
        for (int i = 0; i < 8; i++) {
            x = t.x + dx[i];
            y = t.y + dy[i];
            if (isInside(x, y, N) && !visit[x][y]) {
                visit[x][y] = true;
                q.add(new cell(x, y, t.dis + 1));
            }
        }
    }
    return Integer.MAX_VALUE;
}

public static void main(String[] args) {
    int N = 30;
    int knightPos[] = {1, 1};
    int targetPos[] = {30, 30};
    System.out.println(minStepToReachTarget(knightPos, targetPos, N));
}
}

```

## 20.20 N Queen Problem

El problema de las N reinas es aquel en donde se colocan N reinas en un tablero de NxN de tal manera que dos reinas no puedan atacarse una con otra.

La idea es colocar reinas una por una en diferentes columnas, iniciando desde la columna de más a la izquierda, cuando colocamos una reina en una columna, verificamos por colisiones con las reinas ya colocadas, en la columna actual, si encontramos una fila en la cual no hay colisión marcamos esta columna y fila como parte de la solución, si nosotros no encontramos tal fila en donde haya colisión entonces se retrocede y se retorna falso.

- 1) Iniciar desde la columna de más a la izquierda.
- 2) Si todas las reinas han sido colocadas, retornar true
- 3) Intentar todas las filas de la la columna actual

Realizar lo siguiente para cada columna intentada

- a) si la reina puede ser colocada seguramente en esta fila entonces marcar esta [fila, columna] como parte de la solución y recursivamente verificar si colocando una reina aquí se llega a la solución.
  - b) Si colocando una reina en [fila,columna] se llega a una solución, entonces retorne true.
  - c) Si colocando una reina no se llega a la solución entonces desmarque esta fila y columna (Paso atrás) y vaya al paso A para intentar otras filas.
- 4) Si todas las filas han sido probadas y ninguna funciona retorne falso para activar el backtracking.

```
/* Programa JAVA que resuelve el problema de las N reinas
usando backtracking*/
public class NQueenProblem {
    final int N = 4;
/* A utility function to print solution */
    void printSolution(int board[][])
    {
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                System.out.print(" " + board[i][j]
+ " ");
            }
            System.out.println();
        }
    }/*Una función de utilidad para comprobar si una reina puede
    colocarse en board[row][col]. Tenga en cuenta que
    La función se llama cuando "col" reinas ya están
```

```

        Colocadas en columnas de 0 a col -1. Así que necesitamos
        para comprobar sólo el lado izquierdo para las reinas atacantes*/
boolean isSafe(int board[][], int row, int col) {
    int i, j;
    /*Verifica esta fila en el lado izquierdo*/
    for (i = 0; i < col; i++) {
        if (board[row][i] == 1) {
            return false;
        }
    }
    /*Verifica la diagonal superior de la izquierda*/
    for (i = row, j = col; i >= 0 && j >= 0; i--, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }
    /*Diagonal baja desde la izquierda*/
    for (i = row, j = col; j >= 0 && i < N; i++, j--) {
        if (board[i][j] == 1) {
            return false;
        }
    }
    return true;
}
boolean solveNQUtil(int board[][], int col) {
/* Caso base: Si todas las reinas estan
puestas, retorno true*/
    if (col >= N) {
        return true;
    }
    /*Considere esta columna e intente colocando esta
     reina en todas las filas una por una*/
    for (int i = 0; i < N; i++) {
        /* Verifica si la reina puede ser puesta
        en board[i][col] */
        if (isSafe(board, i, col)) {
            board[i][col] = 1;
            /*Recursivamente pone todas las reinas*/
            if (solveNQUtil(board, col + 1) == true) {
                return true;
            }
            board[i][col] = 0; // BACKTRACK
        }
    }
    /*Si la reina no puede ser puesta en
     ninguna fila en esta columna, r retorna falso*/
    return false;
}
/*Esta función resuelve el problema de N Queen usando
 Backtracking. Utiliza principalmente solveNQUtil()
 para resolver el problema. Devuelve false si no se
 pueden colocar las reinas; de lo contrario, devuelve
 true e imprime la ubicación de las reinas en forma de
 1s. Tenga en cuenta que puede haber más de una solución,
 esta función imprime una de las soluciones posibles.*/

```

```

boolean solveNQ() {
    int board[][] = {{0, 0, 0, 0},
    {0, 0, 0, 0},
    {0, 0, 0, 0},
    {0, 0, 0, 0}
    };
    if (solveNQUtil(board, 0) == false) {
        System.out.print("Solución no existe");
        return false;
    }
    printSolution(board);
    return true;
}

public static void main(String args[]) {
    NQueenProblem Queen = new NQueenProblem();
    Queen.solveNQ();
}
}

```

## 20.21 Prim Minimum Spanning Tree

El algoritmo de Prim es también un algoritmo voraz, comienza con un árbol de expansión vacío, la idea es mantener dos conjuntos de vértices, el primero contiene los vértices ya incluidos en el MST, la otra contiene los vértices que no han sido incluidos aun, en cada paso, se considera todos los caminos que conectan dos sets, y toma el camino con mínimo peso de esos caminos, luego de tomar el camino, se mueve el otro punto final del camino al set conteniendo el MST.

Un grupo de caminos que conectan dos sets de vértices en un grafo es llamado corte en la teoría de grafos, entonces en cada paso del algoritmo de Prim, nosotros buscamos un corte (de dos conjuntos, uno contiene los vértices ya en el MST y el otro el resto de vértices), tome el camino de mínimo peso del corte e incluye este vértice al conjunto MST.

¿Cómo funciona el algoritmo de Prim? La idea detrás del algoritmo de Prim es simple, un árbol de expansión significa que todos los vértices deben estar conectados, entonces los dos conjuntos disjuntos de vértices deben estar conectados para hacer un árbol de expansión. Y ellos deben estar conectados con el camino de peso mínimo para realizar un árbol de expansión mínima.

- 1) Crear un set mstSet que mantenga rastro de los vértices ya incluidos en el MST.
- 2) Asignar un valor key a todos los vértices en el grafo de entrada, inicializar los valores key como INFINITO, asignar el valor key como 0 para el primer vértice entonces este es tomado primero.
- 3) Mientras mstSet no incluya todos los vértices.
  - a) Tomar un vértice u el cual no esté en el mstSet y tenga el valor key mínimo
  - b) Incluir u al mstSet
  - c) Actualizar el valor key de todos los vértices adyacentes de u, para actualizar los valores key, se itera a través de todos los vértices adyacentes v, si el peso del camino u-v es menos que el valor key anterior de v, actualice el valor key como peso de u-v.

La idea de usar valores key es tomar el camino de mínimo peso como el corte, los valores key son usados únicamente por los vértices que no han sido incluidos en el MST, el valor key de estos vértices indican los caminos de peso mínimo conectándolos con el set de vértices incluidos en el MST.

Por ejemplo:

El set mstSet esta inicialmente vacío y los keys asignados a los vértices son {0, INF, INF, INF, INF, INF, INF} donde INF indica infinito, ahora tomamos el vértice con el valor key menor el vértice 0 es tomado, se incluye en el mstSetm entonces mstSet se convierte en {0}, luego de incluir al mstSet, se actualizan los valores key de los vértices adyacentes, los vértices adyacentes de 0 son 1 y 7, los valores key de 1 y 7 son actualizados como 4 y 8.

Se toma el vértice con el menor valor key y no incluido ya en el MST, (no en mstSet), el vértice 1 es tomado y añadido al mstSet por lo que se convierte en {0,1}, se actualiza los valores key de los vértices adyacentes de 1, el valor key del vértice 2 se convierte en 8.

Se toma el vértice con el valor key mínimo y no incluido en MST, podemos tomar el vértice 7 o 2, tomaremos el 7, entonces el mstSet es {0,1,7}, actualizamos los valores key de los vértices adyacentes de 7, el valor key del vértice 6 y 8 se convierten en finitos, en 1 y 7.

Tome el vértice con el valor key mínimo, y no incluido ya en MST, el vértice 6 es tomado entonces el mstSet se convierte en {0,1,7,6}, se actualizan los valores key y los vértices adyacentes de 6m el valor de 5 y 8 es actualizado.

Repetimos los pasos de arriba hasta que mstSet incluya todos los vértices del grafo dado.

```
//Programa java que calcula el MST de un grafo
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Comparator;

public class PrimMSTFULL {

    public static void main(String[] args) {
        int V = 9;
        Graph graph = new Graph(V);
        addEdge(graph, 0, 1, 4);
        addEdge(graph, 0, 7, 8);
        addEdge(graph, 1, 2, 8);
        addEdge(graph, 1, 7, 11);
        addEdge(graph, 2, 3, 7);
        addEdge(graph, 2, 8, 2);
        addEdge(graph, 2, 5, 4);
        addEdge(graph, 3, 4, 9);
        addEdge(graph, 3, 5, 14);
        addEdge(graph, 4, 5, 10);
        addEdge(graph, 5, 6, 2);
        addEdge(graph, 6, 7, 1);
        addEdge(graph, 6, 8, 6);
        addEdge(graph, 7, 8, 7);
        prims_mst(graph);
    }

    static class node1 {
        int dest;
        int weight;
        node1(int a, int b) {
            dest = a;
            weight = b;
        }
    }

    static class Graph {
        int V;
        LinkedList<node1>[] adj;
        Graph(int e) {
            V = e;
            adj = new LinkedList[V];
            for (int o = 0; o < V; o++) {
                adj[o] = new LinkedList<>();
            }
        }
    }

    private void addEdge(Graph graph, int u, int v, int w) {
        node1 n1 = new node1(v, w);
        node1 n2 = new node1(u, w);
        graph.adj[u].add(n1);
        graph.adj[v].add(n2);
    }

    private void prims_mst(Graph graph) {
        int V = graph.V;
        int[] key = new int[V];
        int[] parent = new int[V];
        boolean[] mstSet = new boolean[V];
        for (int i = 0; i < V; i++) {
            key[i] = Integer.MAX_VALUE;
            parent[i] = -1;
            mstSet[i] = false;
        }
        key[0] = 0;
        PriorityQueue<node1> pq = new PriorityQueue<>(new Comparator<node1>() {
            @Override
            public int compare(node1 n1, node1 n2) {
                return n1.weight - n2.weight;
            }
        });
        pq.add(new node1(0, 0));
        while (!pq.isEmpty()) {
            node1 n = pq.poll();
            int u = n.dest;
            if (mstSet[u]) {
                continue;
            }
            mstSet[u] = true;
            for (node1 adj : graph.adj[u]) {
                int v = adj.dest;
                int weight = adj.weight;
                if (!mstSet[v] && key[v] > weight) {
                    key[v] = weight;
                    parent[v] = u;
                    pq.add(new node1(v, weight));
                }
            }
        }
        System.out.println("MST edges:");
        for (int i = 1; i < V; i++) {
            System.out.println("Edge " + parent[i] + " -> " + i + " weight: " + key[i]);
        }
    }
}
```

```

    static class node {
        int vertex;
            int key;
        }
    /* Comparador de la priorityQueue
        retorna 1 si node0.key > node1.key
        retorna 0 si node0.key < node1.key y
    retorna -1 otherwise */
    static class comparator implements Comparator<node> {
        @Override
        public int compare(node node0, node node1) {
            return node0.key - node1.key;
        }
    }
    static void addEdge(Graph graph, int src, int dest, int weight) {
        node node0 = new node(dest, weight);
        node node = new node(src, weight);
        graph.adj[src].addLast(node0);
        graph.adj[dest].addLast(node);
    }

// Buscar MST
static void prims_mst(Graph graph) {
    Boolean[] mstset = new Boolean[graph.V];
    node[] e = new node[graph.V];
    int[] parent = new int[graph.V];
    for (int o = 0; o < graph.V; o++) {
        e[o] = new node();
    }
    for (int o = 0; o < graph.V; o++) {
//inicializar en falso
        mstset[o] = false;
// Inicial valores key en infinito
        e[o].key = Integer.MAX_VALUE;
        e[o].vertex = o;
        parent[o] = -1;
    }

// incluir el vertice inicial en el MST
mstset[0] = true;
e[0].key = 0;
PriorityQueue<node> queue = new PriorityQueue<>
(graph.V, new comparator());
for (int o = 0; o < graph.V; o++) {
queue.add(e[o]);
}
while (!queue.isEmpty()) {
node node0 = queue.poll();
mstset[node0.vertex] = true;
for (node1 iterator : graph.adj[node0.vertex]) {
if (mstset[iterator.dest] == false) {
if (e[iterator.dest].key > iterator.weight) {
queue.remove(e[iterator.dest]);
e[iterator.dest].key = iterator.weight;
}
}
}
}
}

```

```

queue.add(e[iterator.dest]);
parent[iterator.dest] = node0.vertex;
}
}
}
}
// Imprimir el par de vertices del mst
for (int o = 1; o < graph.V; o++) {
System.out.println(parent[o] + " "
+ "-" +
+ " " + o);
}
}
//Porgrama java que busca el arbol de expansión
// minima de minimo costo
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;
import java.util.Scanner;

public class PrimMSTMinValue {

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        N = sc.nextInt();
        G = new Vertex[N];
        for (int i = 0; i < G.length; i++) {
            G[i] = new Vertex();
        }
        E = sc.nextInt();
        for (int i = 0; i < E; i++) {
            int from, to, w;
            from = sc.nextInt();
            to = sc.nextInt();
            w = sc.nextInt();
            G[from].adj.add(new Edge(to, w));
            G[to].adj.add(new Edge(from, w));
        }
        System.out.println(prim());
    }

    static PriorityQueue<QueueItem> Q;
    static int E;
    static boolean[] intree;
    static int N;
    static Vertex[] G;

    static class QueueItem implements Comparable<QueueItem> {

        int v, w;

        public QueueItem(int v, int w) {
            this.v = v;
        }
    }
}

```

```

        this.w = w;
    }

    @Override
    public int compareTo(QueueItem q) {
        if (this.w != q.w) {
            return this.w - q.w;
        }
        return this.v - q.v;
    }
}

static void process(int u) {
    intree[u] = true;
    G[u].adj.forEach((e) -> {
        int v = e.to, w = e.w;
        if (!intree[v]) {
            Q.offer(new QueueItem(v, w));
        }
    });
}

static int prim() {
    intree = new boolean[N];
    Q = new PriorityQueue<>();
    int cost = 0;
    process(0);
    while (!Q.isEmpty()) {
        QueueItem qi = Q.poll();
        int v = qi.v, w = qi.w;
        if (!intree[v]) {
            cost += w;
            process(v);

        }
    }
    return cost;
}

static class Vertex {

    List<Edge> adj;

    public Vertex() {
        adj = new ArrayList<>();
    }

}

static class Edge {

    int to, w;

    public Edge(int to, int w) {

```

```

        this.to = to;
        this.w = w;
    }

}
}

```

## 20.22 Topological Sorting

Ordenamiento topológico de un grafo dirigido acíclico (DAG) es un ordenamiento lineal de vértices tales que por cada camino dirigido  $uv$ , el vértice  $u$  venga antes de  $v$  en el orden, si el grafo no es DAG no es posible el ordenamiento topológico.

En DFS imprimimos un vértice y luego recursivamente llamamos DFS para los vértices adyacentes, en el ordenamiento topológico necesitamos imprimir un vértice antes de sus vértices adyacentes.

```

// Programa java que imprime el ordenamiento topologico
// de un grafo dirigido
import java.util.*;

public class TopologicalSorting {

    public static void main(String args[]) {
        // Create a graph given in the above diagram
        Graph g = new Graph(6);
        g.addEdge(5, 2);
        g.addEdge(5, 0);
        g.addEdge(4, 0);
        g.addEdge(4, 1);
        g.addEdge(2, 3);
        g.addEdge(3, 1);
        System.out.println("Siguiendo el ordenamiento topologico "
            + "del grafo dado ");
        g.topologicalSort();
    }

    static class Graph {
        private int V;
        private LinkedList<Integer> adj[];
        Graph(int v) {
            V = v;
            adj = new LinkedList[v];
            for (int i = 0; i < v; ++i) {
                adj[i] = new LinkedList();
            }
        }
    }
}

```

```

        void addEdge(int v, int w) {
            adj[v].add(w);
        }
        void topologicalSortUtil(int v, boolean visited[],
                                  Stack stack) {
            visited[v] = true;
            Integer i;
            Iterator<Integer> it = adj[v].iterator();
            while (it.hasNext()) {
                i = it.next();
                if (!visited[i]) {
                    topologicalSortUtil(i, visited, stack);
                }
            }
            stack.push(new Integer(v));
        }
        void topologicalSort() {
            Stack stack = new Stack();
            boolean visited[] = new boolean[V];
            for (int i = 0; i < V; i++) {
                visited[i] = false;
            }
            for (int i = 0; i < V; i++) {
                if (visited[i] == false) {
                    topologicalSortUtil(i, visited, stack);
                }
            }
            // Imprime el contenido de la pila
            while (stack.empty() == false) {
                System.out.print(stack.pop() + " ");
            }
        }
    }
}

```

## 20.23 Union Find

Una estructura de datos para conjuntos disjuntos, es una estructura de datos que mantiene un conjunto de elementos particionados en un número de conjuntos disjuntos no se solapan los conjuntos). Un algoritmo Unión-Buscar es un algoritmo que realiza dos importantes operaciones en esta estructura de datos:

Buscar: Determina a cual subconjunto pertenece un elemento. Esta operación puede usarse para verificar si dos elementos están en el mismo conjunto.

Union: Une dos subconjuntos en uno solo.

La otra operación importante CrearConjunto es generalmente trivial, esta crea un conjunto con un elemento dado. Con estas tres operaciones, muchos problemas prácticos de particionamiento pueden ser resueltos.

Con el fin de definir estas operaciones más precisamente, es necesario representar los conjuntos de alguna manera. Una aproximación común es seleccionar un elemento fijo de cada conjunto, llamado el representativo, para representar el conjunto como un todo. Entonces Buscar(x) retorna el elemento representativo del conjunto al cual x pertenece, y Unión toma como argumento dos elementos representativos de dos conjuntos respectivamente.

```
//Programa java que usa el algoritmo UNION FIND
// para verificar componentes conexas
/*
INPUT
9 7
2 0
7 0
3 1
1 0
6 4
8 5
4 5
*/
import java.util.*;

public class UnionFind {

    static final int MAX = 10005; //maximo número de vértices
    static int padre[] = new int[MAX]; //Este arreglo contiene el parente del i-
esimo nodo
    static int rango[] = new int[MAX]; //profundidad de cada vértice

    //Método de inicialización
    static void MakeSet(int n) {
        for (int i = 0; i < n; ++i) {
            padre[i] = i; //Inicialmente el parente de cada vértice es el mismo vértice
            rango[i] = 0; //Altura o rango de cada vértice es 0
        }
    }

    //Método para encontrar la raíz del vértice actual X
    static int Find(int x) {
        if (x == padre[x]) { //Si estoy en la raíz
            return x; //Retorno la raíz
        } //else return Find( padre[ x ] ); //De otro modo busco el parente del
        //vértice actual, hasta llegar a la raíz.
        else {

```

```

        return padre[x] = Find(padre[x]); //Compresion de caminos
    }
}

//Método para unir 2 componentes conexas
static void Union(int x, int y) {
    int xRoot = Find(x); //Obtengo la raiz de la componente del vértice X
    int yRoot = Find(y); //Obtengo la raiz de la componente del vértice Y
    padre[xRoot] = yRoot; //Mezclo ambos arboles o conjuntos, actualizando
    su padre de alguno de ellos como la raiz de otro
}

//Método para unir 2 componentes conexas usando sus alturas (rangos)
static void UnionbyRank(int x, int y) {
    int xRoot = Find(x); //Obtengo la raiz de la componente del vértice X
    int yRoot = Find(y); //Obtengo la raiz de la componente del vértice Y
    if (rango[xRoot] > rango[yRoot]) { //en este caso la altura de la
        componente del vértice X es
        //mayor que la altura de la componente del vértice Y.
        padre[yRoot] = xRoot; //el padre de ambas componentes será el de
        mayor altura
    } else { //en este caso la altura de la componente del vértice Y es
        mayor o igual que la de X
        padre[xRoot] = yRoot; //el padre de ambas componentes será el de
        mayor altura
        if (rango[xRoot] == rango[yRoot]) { //si poseen la misma altura
            rango[yRoot]++; //incremento el rango de la nueva raíz
        }
    }
}

static int root[] = new int[MAX]; //tendra las raices de las componentes
conexas luego de aplicar el método
static int numComponentes; //variable para el número total de componentes
conexas
//Método para obtener el número de componentes conexas luego de realizar las
conexiones respectivas

static int getNumberConnectedComponents(int n) {
    numComponentes = 0;
    for (int i = 0; i < n; ++i) {
if (padre[i] == i) { //Si el padre del vértice i es el mismo vértice entonces es
    raíz
        //if( Find( i ) == i ){ //podemos usamos find para el mismo
        proposito y
        //para que se realice compresion de caminos
        root[numComponentes++] = i; //almaceno la raiz de cada nueva
        componente
        // numComponentes++;
    }
}
    return numComponentes;
}

```

```

        static int numVertices[] = new int[MAX]; //almacenara la cantidad de
vértices para la i-esima raíz.
//Método para obtener la raíz y el número de vértices de cada componente conexa
//será necesario primero tener la cantidad de componentes conexas
//podemos llamar 1ero al metodo getNumberConnectedComponents o incluir
porcion de su código en este

static void getNumberNodes(int n) {
    Arrays.fill(numVertices, 0); //inicializo mi contador de vértices
    for (int i = 0; i < n; ++i) {
        numVertices[Find(i)]++; //incremento la raíz del vértice i
    }
    for (int i = 0; i < numComponentes; ++i) {
        System.out.printf("Componente %d: Raiz = %d , Nro nodos = %d.\n", i
+ 1, root[i], numVertices[root[i]]);
    }
}

//Método que me determina si 2 vértices están o no en la misma componente conexa
static boolean sameComponent(int x, int y) {
    if (Find(x) == Find(y)) {
        return true; //si poseen la misma raíz
    }
    return false;
}

public static void main(String[] args) {
    int V, E, origen, destino;
    Scanner sc = new Scanner(System.in);
    V = sc.nextInt();
    E = sc.nextInt(); //tengamos número de vértices y aristas
    MakeSet(V); //inicializamos los conjuntos
    for (int i = 0; i < E; ++i) {
        origen = sc.nextInt();
        destino = sc.nextInt();
        UnionbyRank(origen, destino); //unión de elementos
    }
    System.out.printf("El número de componentes conexas es: %d\n",
getNumberConnectedComponents(V));
    getNumberNodes(V);
}
}

```

## 20.24 Domino DFS

El efecto dominó o reacción en cadena es el efecto acumulativo producido cuando un acontecimiento origina una cadena de otros acontecimientos similares.

Se produce cuando un pequeño cambio origina un cambio similar a su lado, que a su vez causa otro similar, y así sucesivamente en una secuencia lineal. Recibe este nombre, por analogía con la caída de una hilera de fichas de dominó colocadas en posición vertical. El efecto dominó también puede hacer referencia a una cadena de acontecimientos no materiales.

El término, en sus distintos usos, se ha hecho popular por su analogía al efecto mecánico, una fila de fichas de dominó al caer una ficha detrás de otra, aunque típicamente se refiere a una secuencia enlazada de acontecimientos donde el tiempo entre acontecimientos sucesivos es relativamente pequeño. Puede ser utilizado literalmente (una serie observada de colisiones reales) o de forma metafórica (conexiones causales dentro de sistemas como la política o las finanzas globales).

```
//Programa java que usando DFS busca cuantos dominos
// caen desde un origen
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Scanner;

/* EJEMPLO DE INGRESO
9 6 3
1 2
2 5
5 3
4 3
6 7
7 8
1
6
4*/
public class DominoDFS {

    static final int MAX = 10001;
    static ArrayList<ArrayList<Integer>> ady =
        new ArrayList<ArrayList<Integer>>(MAX);
    static Scanner sc = new Scanner(System.in);
    //la cantidad total de dominos que caerán
    static int total;
    //arreglo de domino caido
    static boolean visitado[] = new boolean[MAX];

    public static void main(String[] args) {
        //número de dominos, cantidad de enlaces, consultas
        int V, M, Q, x, y, origen;
        V = sc.nextInt();
        M = sc.nextInt();
        Q = sc.nextInt();
```

```

        for (int i = 0; i < V; i++) {
            ady.add(new ArrayList<>());
        }
        while (M > 0) {
//domino x hace caer a domino y
x = sc.nextInt();
            y = sc.nextInt();
            ady.get(x).add(y);
M--;
        }
        while (Q > 0) {
//domino origen
            origen = sc.nextInt();
            total = 0;
            Arrays.fill(visitado, false);
            dfs(origen);
System.out.printf("%d\n", total);
Q--;
    }

}

static void dfs(int u) { //domino origen
    //aumento en mi respuesta la caida de un domino
    total++;
//domino "u" cayo
    visitado[u] = true;
//verifico los demás posibles
    //domino que caeran si impulso "u"
for (int v = 0; v < ady.get(u).size(); ++v) {
//si el domino adyacente no
    //cayó entonces es el siguiente a evaluar
    if (!visitado[ady.get(u).get(v)]) {
//recursivamente veo que dominos
    //caeran a partir del adyacente de "u"
dfs(ady.get(u).get(v));
}
}
}
}

```

## 20.25 Exit The Maze BFS

Un laberinto es un pasatiempo gráfico consistente en trazar una línea desde un punto de origen situado en el exterior de un laberinto a uno de destino situado generalmente en el centro o bien en el lado opuesto. La dificultad consiste en encontrar un camino directo hasta

el lugar deseado. El laberinto, por su propia configuración, contiene diferentes vías sin salida (de mayor o menor longitud) y solo un recorrido correcto.

```
// Programa java que realiza laberintos con BFS
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class ExitTheMazeBFS {
    /* Ejemplo de ingreso
    8 8
    ....I
    ######
    .#.....
    .#.S...S
    .###.#.#
    .#...#.#
    .#.###.#
    .....
    */
    //máximo número de filas y columnas del laberinto
    static final int MAX = 100;
    static Scanner sc = new Scanner(System.in);
    //laberinto
    static char ady[][][] = new char[MAX][MAX];
    //arreglo de estados visitados
    static boolean visitado[][][] = new boolean[MAX][MAX];
    //incremento en coordenada x
    static int dx[] = {0, 0, 1, -1};
    //incremento en coordenada y
    static int dy[] = {1, -1, 0, 0};
    //altura y ancho del laberinto
    static int h, w;
    //Arreglo para mostrar la ruta que se siguió
    static Estado prev[][][] = new Estado[MAX][MAX];
    static class Estado {
        int x; // Fila del estado
        int y; // Columna del estado
        int d; // Distancia del estado
    }
    // Constructor
    Estado(int x1, int y1, int d1) {
        this.x = x1;
        this.y = y1;
        this.d = d1;
    }
    Estado() {
    }
}

public static void main(String[] args) {
    int x=0, y=0;
    System.out.println("Ingrese altura del laberinto: ");
    h = sc.nextInt();
```

```

        System.out.println("Ingrese ancho del laberinto: ");
        w = sc.nextInt();
        sc.nextLine();
        System.out.printf("\nIngrese el laberinto, con un solo "
                + "valor inicial I, valor final sera S: \n");
        for (int i = 0; i < h; ++i) {
            String aux = sc.nextLine();
            for (int j = 0; j < w; ++j) {
                ady[i][j] = aux.charAt(j);
        //obtengo coordenada de valor inicial
                if (ady[i][j] == 'I') {
                    x = i;
                    y = j;
                }
            }
        }

//MOSTRAMOS LABERINTO
for (int i = 0; i < h; ++i) {
    for (int j = 0; j < w; ++j) {
        System.out.printf("%c", ady[i][j]);
    }
    System.out.printf("\n");
}
int min = BFS(x, y, h, w);
if (min != -1) {
    System.out.printf("Menor número de pasos: %d\n", min);
} else {
    System.out.println("No se pudo llegar al destino");
}

}

//Funcion para imprimir la ruta encontrada
//Ingresan las coordenadas del nodo final

static void print(int x, int y) {
//El arreglo prev posee las coordenadas del nodo
    //anterior, por ello empezamos desde el final
    //El proceso termina al momento de preguntar
    //por el anterior del nodo inicial, como pusimos -1
    //Preguntamos hasta que nuestro anterior sea diferente de -1
    for (int i = x, j = y; prev[i][j].d != -1; i =
    prev[x][y].x, j = prev[x][y].y) {
        ady[i][j] = '*';
        x = i;
        y = j;
    }

    System.out.printf("Camino con menor número de pasos\n");
    for (int i = 0; i < h; ++i) {
        for (int j = 0; j < w; ++j) {
            System.out.printf("%c", ady[i][j]);
        }
        System.out.printf("\n");
    }
}

```

```

        }
    //coordenadas de inicial "I" y dimensiones de laberinto
    static int BFS(int x, int y, int h, int w) {
        //Estado inicial, distancia = 0
        Estado inicial = new Estado(x, y, 0);
        //Cola de todos los posibles Estados por
        //los que se pase para llegar al destino
        Queue<Estado> Q = new LinkedList<>();
        //Insertamos el estado inicial en la Cola.
        //marcamos como no visitado
        Q.offer(inicial);
        for (int i = 0; i < MAX; i++) {
            Arrays.fill(visitado[i], false);
        }
        //el inicial no tiene una ruta anterior puesto que es primero
        prev[x][y] = new Estado(-1, -1, -1);
        //Mientras cola no este vacia
        while (!Q.isEmpty()) {
            //Obtengo de la cola el estado actual,
            //en un comienzo sera el inicial
            Estado actual = Q.peek();
            //Saco el elemento de la cola
            Q.poll();
            //Si se llego al destino (punto final)
            if (ady[actual.x][actual.y] == 'S') {
                //imprimo la ruta del camino mas corto
                print(actual.x, actual.y);
                //Retornamos distancia recorrida hasta ese momento
                return actual.d;
            }
            //Marco como visitado dicho estado para no volver a recorrerlo
            visitado[actual.x][actual.y] = true;
            //Recorremos hasta 4 porque tenemos 4 posibles adyacentes
            for (int i = 0; i < 4; ++i) {
                //nx y ny tendran la coordenada adyacente
                int nx = dx[i] + actual.x;
                //ejemplo en i=0 y actual
                //(3,4) -> 3+dx[0]=3+0=3,
                //4+dy[0]=4+1=5, nueva coordenada (3,5)
                int ny = dy[i] + actual.y;
                //aqui comprobamos que la coordenada
                //adyacente no sobrepase las dimensiones del laberinto
                //ademas comprobamos que no sea
                //pared "#" y no este visitado
                if (nx >= 0 && nx < h && ny >= 0 &&
                    ny < w && ady[nx][ny] != '#' && !visitado[nx][ny]) {
                    //Creamos estado adyacente aumento en
                    //1 la distancia recorrida
                    Estado adyacente = new Estado(nx, ny, actual.d + 1);
                    //Agregamos adyacente a la cola
                    Q.offer(adyacente);
                    //El previo del nuevo nodo es el actual.
                    prev[nx][ny] = actual;
                }
            }
        }
    }
}

```

```

        }
        return -1;
    }
}

```

## 20.26 Bellman-Ford Shortest Path

Dado un grafo y un vértice de origen  $src$  en el grafo, encontrar los caminos más cortos desde  $src$  a todos los vértices en el grafo dado, el grafo puede contener caminos con pesos negativos. Si hay un ciclo de peso negativo, entonces las distancias más cortas no son calculadas, se reporta el ciclo negativo.

- 1) Este paso inicializa las distancias desde el origen de todos los vértices como infinito y la distancia al origen en si como 0, crea un array  $dist[]$  de tamaño  $V$  con todos los valores como infinito excepto  $dist[src]$  donde  $src$  es el vértice origen.
- 2) Este paso calcula las distancias más cortas, esto se realiza  $V-1$  veces.
- 3) si  $dist[v] > dist[u] + \text{peso del camino } uv$ , entonces actualice  $dist[]$  en  $dist[v] = dist[u] + \text{weight of edge } uv$
- 4) Este paso verifica si hay un ciclo negativo en el grafo, se realiza lo siguiente:
- 5) si  $dist[v] > dist[u] + \text{peso del camino } uv$ , entonces “El grafo contiene un ciclo negativo”

La idea del paso 3 es, el paso 2 garantiza las distancias más cortas si el grafo no contiene un ciclo de peso negativo, si iteramos a través de todos los caminos una vez más y obtenemos un camino más corto para cualquier vértice, entonces ahí hay un ciclo negativo.

¿Cómo funciona esto? Como en otros problemas de programación dinámica, el algoritmo calcula los caminos más cortos de manera del atrás hacia adelante, primero calcula las distancias más cortas las cuales tienen al menos una arista en el camino, luego calcula los caminos más cortos con al menos dos aristas, y así en adelante, luego de la  $i$ esima iteración del ciclo exterior, los caminos más cortos con al menos  $i$  aristas son calculados, ahí puede haber un máximo de  $V-1$  aristas en un camino simple, por eso el ciclo externo se corre  $V-1$  veces, la idea es, asumiendo que ahí no hay ciclo negativo, si calculamos los caminos los

caminos más cortos con al menos  $i$  aristas, entonces una interacción sobre todos las aristas garantiza darnos el camino más corto con al menos  $i+1$  artistas.

Ejemplo:

Dado el vértice origen 0, inicializamos todas las distancias como infinito, excepto la distancia al origen mismo, el número total de vértices en el grafo es 5 y todos los caminos deben ser procesados 4 veces.

Todos las aristas son procesadas en el siguiente orden, (B, E), (D, B), (B, D), (A, B), (A, C), (D, C), (B, C), (E, D). Tenemos las siguientes distancias cuando todos las aristas son procesadas por primera vez, la primera fila

La primera iteración garantiza darnos todos los caminos más cortos los cuales tienen un largo de una arista, obtenemos las distancias siguientes cuando todos las aristas son procesadas por segunda vez.

La segunda iteración garantiza darnos todos los caminos más cortos que sean de 2 aristas más larga, el algoritmo procesa los caminos 2 veces más, las distancias son minimizadas luego de la segunda iteración, y la tercera y cuarta no actualiza distancias...

- 1) Pesos negativos son encontrados en varias aplicaciones de grafos, por ejemplo en vez de pagar el costo por un camino, podemos obtener ventaja si seguimos el camino.
- 2) Bellman-Ford trabaja mejor que Dijkstra para sistemas distribuidos, a diferencia de Dijkstra en donde necesitamos encontrar el valor menor de todos los vértices, en Bellman-Ford necesitamos considerar uno por uno.

```
// Implementación java del algoritmo de BellmanFord
// para la búsqueda del camino más corto de un vértice al
// resto y con capacidad de detección de ciclo negativo
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class BellmanFordSP {
```

```
    static final int MAX = 105;
    static final int INF = 1 << 30;
    static int[] previo = new int[MAX];
```

```

static int[] distancia = new int[MAX];
static int vertices;
static List<List<Node>> adyacencia = new ArrayList<List<Node>>();
static Scanner sc = new Scanner(System.in);
static void inicializacion() {
    for (int i = 0; i <= vertices; i++) {
        distancia[i] = INF;
        previo[i] = -1;
    }
}
static class Node {
    int first, second;
    public Node(int destino, int peso) {
        this.first = destino;
        this.second = peso;
    }
}
static void print(int destino) {
if (previo[destino] != -1) {
    print(previo[destino]);
}
System.out.printf("%d ", destino);
}
static boolean relajacion(int actual, int adyacente, int peso) {
if (distancia[actual] + peso < distancia[adyacente]) {
    distancia[adyacente] = distancia[actual] + peso;
    previo[adyacente] = actual;
return true;
}
return false;
}
static void bellmanFord(int inicial) {
inicializacion();
distancia[inicial] = 0;
for (int i = 0; i <= vertices - 1; i++) {
    for (int actual = 0; actual <= vertices; actual++) {
        for (int j = 0; j < adyacencia.get(actual).size(); j++) {
            int adyacente = adyacencia.get(actual).get(j).first;
            int peso = adyacencia.get(actual).get(j).second;
relajacion(actual, adyacente, peso);
        }
    }
}
for (int actual = 0; actual <= vertices; actual++) {
for (int j = 0; j < adyacencia.get(actual).size(); j++) {
    int adyacente = adyacencia.get(actual).get(j).first;
    int peso = adyacencia.get(actual).get(j).second;
if (relajacion(actual, adyacente, peso)) {
        System.out.println("Existe ciclo negativo");
        return;
    }
}
}
System.out.println("No existe ciclo negativo");
}

```

```

        System.out.printf("Distancias más cortas iniciando en el nodo %d\n",
inicial);
for (int i = 0; i <= vertices; i++) {
System.out.printf("Nodo %d , distancia más corta = %d\n", i, distancia[i]);
}
System.out.println("\n _____Camino más corto____");
System.out.println("Ingrese vertice destino: ");
int destino = sc.nextInt();
print(destino);
System.out.println("");
}
public static void main(String[] args) {
int E, origen, destino, peso, inicial;
vertices = sc.nextInt();
E = sc.nextInt();
for (int i = 0; i < 10; i++) {
    adyacencia.add(new ArrayList<>());
}
for (int i = 0; i < E; i++) {
    origen = sc.nextInt();
    destino = sc.nextInt();
    peso = sc.nextInt();
    adyacencia.get(origen).add(new Node(destino, peso));
}
System.out.printf("Ingrese el nodo inicial: ");
inicial = sc.nextInt();
bellmanFord(inicial);
}
}

```

## 20.27 Dijkstra Shortest Path

Dado un grafo y un vértice origen en el grafo, encuentre los caminos más cortos del origen a todos los vértices en el grafo dado.

La complejidad de tiempo de esta implementación es de  $O(V^*E)$  si el grafo de entrada está representado usando una lista de adyacencia. Tener en cuenta que Dijkstra no procesa pesos negativos y no detecta ciclos negativos.

```

// Implementación java del algoritmo de Dijkstra que
// busca el camino más corto de un nodo al resto
/*
EJEMPLO DE INPUT
5 9
1 2 7
1 4 2
2 3 1

```

```

2 4 2
3 5 4
4 2 3
4 3 8
4 5 5
5 3 5
1
*/
import java.util.*;

public class DijkstraSP {

    //similar a los defines de C++
    static final int MAX = 10005; //maximo numero de vértices
    static final int INF = 1 << 30; //definimos un valor
        //grande que represente la distancia infinita
        //inicial, basta con que sea superior al maximo
        //valor del peso en alguna de las aristas
        //En el caso de java usamos una clase que
        //representara el pair de C++

    static class Node implements Comparable<Node> {

        int first, second;

        Node(int d, int p) { //constructor
            this.first = d;
            this.second = p;
        }

        @Override
        public int compareTo(Node other) { //es necesario
            //definir un comparador para el
            //correcto funcionamiento del PriorityQueue
            if (second > other.second) {
                return 1;
            }
            if (second == other.second) {
                return 0;
            }
            return -1;
        }
    };

    static Scanner sc = new Scanner(System.in); //para lectura de datos
    static List< List< Node>> ady = new ArrayList< List< Node>>(); //lista de
adyacencia
    static int distancia[] = new int[MAX]; // distancia de vértice inicial
//vértice con ID = u
    static boolean visitado[] = new boolean[MAX]; //para vértices visitados
    static PriorityQueue< Node> Q = new PriorityQueue<Node>();
//usamos el comparador definido para
//que el de menor valor este en el tope
    static int V; //número de vertices
    static int previo[] = new int[MAX]; //para la impresion de caminos
}

```

```

//función de inicialización
static void init() {
    for (int i = 0; i <= V; ++i) {
        distancia[i] = INF; //inicializamos todas
        //las distancias con valor infinito
        visitado[i] = false; //inicializamos todos
        //los vértices como no visitados
        previo[i] = -1; //inicializamos el previo
        //del vertice i con -1
    }
}

//Paso de relajacion
static void relajacion(int actual, int adyacente, int peso) {
//Si la distancia del origen al vertice actual +
//peso de su arista es menor a la distancia del
//origen al vertice adyacente
    if (distancia[actual] + peso < distancia[adyacente]) {
//relajamos el vertice actualizando
//la distancia
        distancia[adyacente] = distancia[actual] + peso;
//a su vez actualizamos el vertice
//previo
        previo[adyacente] = actual;
//agregamos adyacente a la cola
//de prioridad
        Q.add(new Node(adyacente, distancia[adyacente]));
    }
}

//Impresion del camino más corto desde el vertice inicial y final ingresados
static void print(int destino) {
if (previo[destino] != -1) //si aun poseo un vertice previo
{
    print(previo[destino]); //recursivamente sigo explorando
}
//terminada la recursion imprimo los vertices
//recorridos
System.out.printf("%d ", destino);
}

static void dijkstra(int inicial) {
init(); //inicializamos nuestros arreglos
//Insertamos el vértice inicial en la Cola de Prioridad
Q.add(new Node(inicial, 0));
//Este paso es importante, inicializamos la distancia del
//inicial como 0
distancia[inicial] = 0;
int actual, adyacente, peso;
while (!Q.isEmpty()) { //Mientras cola no este vacia
    //Obtengo de la cola el nodo con menor peso, en un
    //comienzo será el inicial
    actual = Q.element().first;
    Q.remove(); //Sacamos el elemento de la cola
}
}

```

```

        if (visitado[actual]) {
            continue; //Si el vértice actual ya fue visitado entonces sigo
        } //sacando elementos de la cola
        visitado[actual] = true; //Marco como visitado el vértice actual
        //reviso sus adyacentes del
        //vertice actual
    for (int i = 0; i < ady.get(actual).size(); ++i) {
        adyacente = ady.get(actual).get(i).first; //id del vertice
        adyacente
        //peso de la arista que une actual
        //con adyacente ( actual , adyacente )
        peso = ady.get(actual).get(i).second;
        //si el vertice adyacente no fue visitado
        if (!visitado[adyacente]) {
            //realizamos el paso de relajacion
            relajacion(actual, adyacente, peso);
        }
    }
}

System.out.printf("Distancias más cortas iniciando en vertice %d\n",
inicial);
for (int i = 1; i <= V; ++i) {
System.out.printf("Vertice %d , distancia más corta = %d\n", i, distancia[i]);
}

System.out.println("\n*****Impresion de camino más
corto*****");
System.out.printf("Ingrese vertice destino: ");
int destino;
destino = sc.nextInt();
print(destino);
System.out.printf("\n");
}

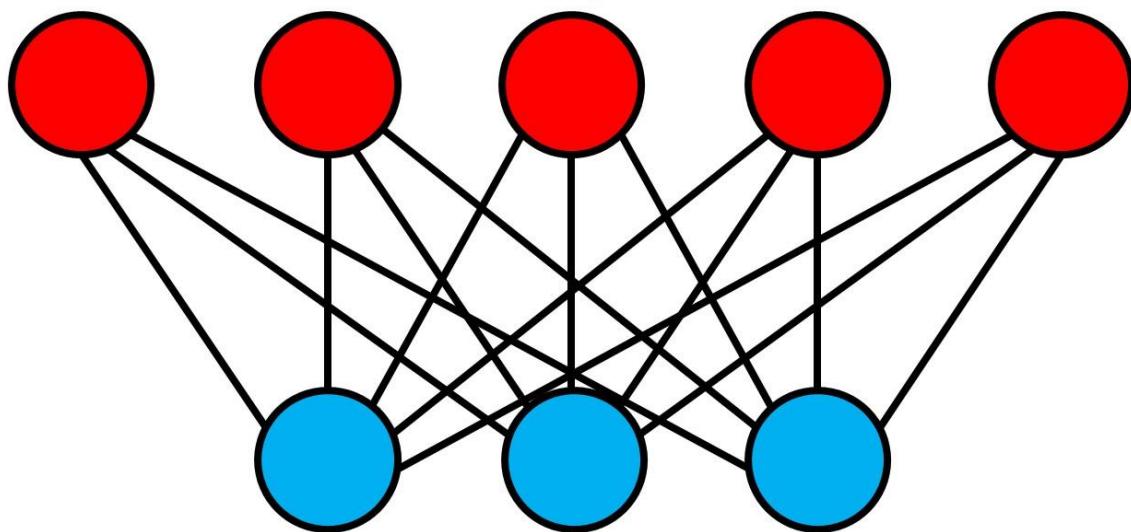
public static void main(String[] args) {
int E, origen, destino, peso, inicial;

V = sc.nextInt();
E = sc.nextInt();
for (int i = 0; i <= V; ++i) {
ady.add(new ArrayList<Node>()); //inicializamos lista de
} //adyacencia
for (int i = 0; i < E; ++i) {
origen = sc.nextInt();
destino = sc.nextInt();
peso = sc.nextInt();
ady.get(origen).add(new Node(destino, peso)); //grafo diridigo
//ady.get( destino ).add( new Node( origen , peso ) ); //no dirigido
}
System.out.print("Ingrese el vertice inicial: ");
inicial = sc.nextInt();
dijkstra(inicial);
}
}

```

## 20.28 Bipartite Graph

### Grafo Bipartido



*Guía del programador competitivo.*

Ilustración 20-7 Ejemplo de coloración de un grafo bipartito

Un grafo bipartito es un grafo cuyos vértices pueden ser divididos en dos sets independientes,  $U$  y  $V$  en donde cada arista  $(u, v)$  conecta un vértice de  $U$  a  $V$  o un vértice de  $V$  a  $U$ . En otras palabras para cada arista  $(u, v)$   $u$  pertenece a  $U$  y  $v$  pertenece a  $V$ , o viceversa, también podemos decir que no hay aristas que conecte vértices del mismo set. Un grafo bipartito es posible si el coloramiento del grafo es posible usando dos colores los cuales los vértices de un set son coloreados del mismo color.

Una aproximación para verificar si un grafo es bipartito es verificar si el grafo es coloreable o no usando backtracking.

El siguiente es un algoritmo simple de verificación de bipartito usando BFS.

- 1) Asigna el color ROJO al vértice origen (Poniéndolo en el set  $U$ )
- 2) Colorea todos los vecinos con color AZUL (poniéndolos en el set  $V$ )
- 3) Colorear todos los vecinos del vecino anterior de color ROJO (Poniéndolos en  $U$ )

- 4) De esta forma se asigna color a todos los vértices los cuales satisfacen todas las restricciones del coloramiento.
- 5) Mientras asignamos colores, si encontramos un vecino que esta coloreado del mismo color del actual vértice entonces el grafo no puede ser coloreado, es decir no es bipartito.

El algoritmo solo funciona si el grafo es fuertemente conectado, en el código siempre empezamos desde el origen 0 y asumimos que los vértices son visitados desde el, una observación importante es un grafo sin aristas es también bipartito,

La complejidad de tiempo de esta aproximación es la misma del BFS,  $O(V^2)$  si el grafo es representado con listas de adyacencia, se convierte en  $O(V+E)$ .

```
//Programa java que encuentra si un grafo es
//bipartito o no
import java.util.*;

public class BipartiteGraph {
    final static int V = 4;
    static boolean isBipartite(int G[][], int src) {
        /*Crear una matriz de colores para almacenar los
         * colores asignados a todas las verificaciones.
         * El número de vértice se utiliza como índice en
         * esta matriz. El valor '-1' de colorArr [i] se
         * usa para indicar que no se asigna ningún color
         * al vértice 'i'. El valor 1 se utiliza para indicar
         * que el primer color está asignado y el valor 0
         * indica que el segundo color está asignado.*/
        int colorArr[] = new int[V];
        for (int i = 0; i < V; ++i) {
            colorArr[i] = -1;
        }
        //Asigna primer color al origen
        colorArr[src] = 1;
        //Crea una cola de número de vértices
        // y encola el vértice origen
        LinkedList<Integer> q = new LinkedList<>();
        q.add(src);
        while (!q.isEmpty()) {
            // Descola un vértice de la cola
            int u = q.poll();
            //Retorna falso si hay un autociclo
            if (G[u][u] == 1) {
                return false;
            }
            //Encuentra todos los vértices adyacentes sin color
            for (int v = 0; v < V; ++v) {
                // Un camino de u a v existe
                if (G[u][v] == 1 && colorArr[v] == -1) {
                    colorArr[v] = 1 - colorArr[u];
                    q.add(v);
                }
            }
        }
        return true;
    }
}
```

```

        // y destino v no esta coloreado
        if (G[u][v] == 1 && colorArr[v] == -1) {
//Asigna color alternativo para esta adyacencia
            colorArr[v] = 1 - colorArr[u];
            q.add(v);
        }
/*Un camino de u a v existe y el destino
   esta del mismo color que u*/
else if (G[u][v] == 1 && colorArr[v] == colorArr[u]) {
    return false;
}
}
}
}
// Si llegamos aqui, todos los vertices adyacentes pueden
// ser coloreados con color alternativo
return true;
}

public static void main(String[] args) {
    int G[][] = {{0, 1, 0, 1},
    {1, 0, 1, 0},
    {0, 1, 0, 1},
    {1, 0, 1, 0}
    };
    if (isBipartite(G, 0)) {
        System.out.println("Si");
    } else {
System.out.println("No");
    }
}
}

```

## 20.29 Problemas de repaso

### Ejercicios en Online Judge

314-Robot

341-Non-Stop Travel

315-Network

492-Pig-Latin

331-Mapping the Swaps

439- Knight Moves

334-Identifying Concurrent Events

558-Wormholes

352-The Seasonal War

125-Numbering Paths

382- Perfection



# 21 Otros algoritmos y programación

## dinámica

### 21.1 Knapsack 0/1

Dados pesos y valores de n ítems, ponga esos ítems en una mochila de capacidad W, para obtener el máximo valor total en la mochila, en otras palabras dados dos arrays de enteros val[0..n-1] y wt[0...-1] los cuales representan los valores y pesos asociados a los n ítems respectivamente. También dado un entero W el cual representa la capacidad de la mochila, encuentre el máximo valor subset de val[] tal que esa suma de los pesos de este subset es menor o igual a w, no se puede romper un ítem, se tiene que tomar o no tomar, (propiedad 0-1).

Una simple solución es considerar todos los subsets de ítems y calcular el peso total y valor de todos los subsets, considere solo los subsets los cuales el peso total es menor a W, de todos los subsets tome el que tenga el máximo valor de peso.

Considere todos los subsets de ítems, puede haber dos casos para cada ítem:

- 1) El ítem está incluido en el subset optimo
- 2) El ítem no está incluido en el subset optimo

Con esto, el máximo valor puede ser obtenido de n ítems es máximo siguiendo los dos siguientes valores,

- 1) El máximo valor obtenido por n-1 ítems y peso W (Excluyendo el enésimo ítem)
- 2) Valor del enésimo ítem más el máximo valor obtenido por n-1 ítems y W menos el peso del enésimo ítem (Incluyendo el enésimo ítem).

Si el peso del enésimo ítem es mayor que W, entonces el enésimo ítem no puede ser incluido y el caso 1 es la única posibilidad.

Desde que los subproblemas son evaluados de nuevo, este problema tiene la propiedad de sobreponer subproblemas. Entonces el problema de la mochila 0-1 tiene ambas propiedades de un problema de programación dinámica, como otros problemas de programación dinámica, recomputaciones de subproblemas iguales pueden ser evadidos construyendo una matriz temporal desde el fondo, el código usa esta metodología.

La complejidad de tiempo de este algoritmo es de  $O(nW)$  donde  $n$  es el número de ítems y  $W$  el peso de la mochila.

```
/*Implementación java del problema 0-1 Knapsack */
public class KnapsackRecursive {
    //Función de utilidad que retorna el mayor de dos números
    static int max(int a, int b) { return (a > b) ? a : b; }
    // Retorna el maximo valor que puede ser
    //Puesto en una knapsack de capacidad w
    static int knapSack(int W, int wt[], int val[], int n)
    {
        // Caso base
        if (n == 0 || W == 0)
            return 0;
        /*Si el peso de el nesimo item es más
         que la capacidad el knapsack, entonces
         este item no puede ser incluido en una
         solución optima*/
        if (wt[n - 1] > W)
            return knapSack(W, wt, val, n - 1);
        // Retorna el maximo de dos casos:
        //1) nesimo termino incluido
        //2) no incluido
        else
            return max(val[n - 1] + knapSack(W - wt[n - 1], wt, val, n - 1),
                      knapSack(W, wt, val, n - 1));
    }

    public static void main(String args[])
    {
        int val[] = new int[] { 60, 100, 120 };
        int wt[] = new int[] { 10, 20, 30 };
        int W = 50;
        int n = val.length;
        System.out.println(knapSack(W, wt, val, n));
    }
}

// Implementación java que usa programación dinamica
// para solucionar el problema de 0-1 Knapsack
public class KnapsackDP {
    //Función de utilidad que retorna el maximo de dos enteros
    static int max(int a, int b) {
        return (a > b) ? a : b;
}
```

```

// Retorna el maximo valor que puede ser puesto en un
//knapsack de capacidad w
    static int knapSack(int W, int wt[], int val[], int n) {
        int i, w;
        int K[][] = new int[n + 1][W + 1];
        // Construye la tabla K[][]de abajo hacia arriba
        for (i = 0; i <= n; i++) {
            for (w = 0; w <= W; w++) {
                if (i == 0 || w == 0) {
                    K[i][w] = 0;
                } else if (wt[i - 1] <= w) {
                    K[i][w] = max(val[i - 1] +
                                   K[i - 1][w - wt[i - 1]], K[i - 1][w]);
                } else {
                    K[i][w] = K[i - 1][w];
                }
            }
        }
        return K[n][W];
    }

    public static void main(String args[]) {
        int val[] = new int[]{60, 100, 120};
        int wt[] = new int[]{10, 20, 30};
        int W = 50;
        int n = val.length;
        System.out.println(knapSack(W, wt, val, n));
    }
}

```

## 21.2 Coin Exchange

Dado un valor N, si queremos hacer el cambio de N centavos y tenemos suministros infinitos de cada uno de las  $S = \{S_1, S_2, \dots, S_m\}$  monedas con valor, ¿de cuantas formas podemos hacer el cambio? El orden de las monedas no importa

Dado un valor N, queremos hacer el cambio por N centavos, y tenemos suministros de cada una, por ejemplo, para  $N=4$ , y  $S=\{1,2,3\}$ , existen 4 soluciones  $\{1,1,1,1\}, \{1,1,2\}, \{2,2\}, \{1,3\}$ . Entonces la salida debe ser 4, para  $N=10$ , y  $S=\{2,5,3,6\}$  existen 6 soluciones  $\{2,2,2,2,2\}, \{2,2,3,3\}, \{2,2,6\}, \{2,3,5\}$  y  $\{5,5\}$ . Entonces la salida debe ser 5.

Para contar el total el números de soluciones podemos dividir todos los sets solución en dos sets.

- 1) Soluciones que no contienen emesima moneda o  $S_m$ .

## 2) Soluciones que al menos contienen una Sm.

Dejaremos ser  $\text{count}(S[], m, n)$  la función de conteo del número de soluciones, luego estas pueden ser escritas como suma de  $\text{count}(S[], m-1, n)$  y  $\text{count}(S[], m, n-S_m)$ .

Por lo tanto, el problema tiene una propiedad de subestructura óptima haciendo que el problema pueda ser resuelto usando soluciones a subproblemas.

Complejidad de tiempo:  $O(mn)$

```
// Programa java que resuelve el problema
// del cambio de monedas
public class CoinExchange {
    // Retorna el conteo de formas que podemos
    // sumar S[0...m-1] monedas para obtener n
    static int count( int S[], int m, int n )
    {
        // Si n es 0 entonces 1 es la solución
        // (No incluir ninguna moneda)
        if (n == 0)
            return 1;
        // Si n es menor que 0 entonces no hay solución
        if (n < 0)
            return 0;
        /*Si no hay monedas y n es mayor que 0, no existe solución*/
        if (m <= 0 && n >= 1)
            return 0;
        // count es la suma de las soluciones (i)
        // incluyendo S[m-1] (ii) excluyendo S[m-1]
        return count( S, m - 1, n ) +
               count( S, m, n-S[m-1] );
    }

    public static void main(String[] args)
    {
        int arr[] = {1, 2, 3};
        int m = arr.length;
        System.out.println( count(arr, m, 4));
    }
}
```

## 21.3 Longest Increasing Subsequence

El problema de la secuencia incremental más larga (LIS), es encontrar la longitud de la subsecuencia más larga dada una secuencia la cual todos los elementos de la subsecuencia están ordenados en orden ascendente, por ejemplo la longitud de LIS para {10, 22, 9, 33, 21, 50, 41, 60, 80} es 6 y LIS es {10, 22, 33, 50, 60, 80}.

Dejaremos arr [0...n-1] ser el array de entrada y L(i) ser la longitud de la LIS finalizando en el índice I tal que arr[i] es el último elemento de la LIS.

Entonces L(i) puede ser recursivamente escrito como:

$$L(i) = 1 + \max( L(j) ) \text{ donde } 0 < j < i \text{ y } arr[j] < arr[i];$$

O

$$L(i) = 1, \text{ Si no existe tal } j.$$

Para encontrar la LIS de un array dado, necesitamos retornar  $\max(L(i))$  donde  $0 < i < n$ .

Por lo tanto vemos que el problema LIS satisface la propiedad de la subestructura óptima como el problema principal puede ser resuelto usando soluciones a subproblemas.

Note que la complejidad de tiempo de esta programación dinámica es  $O(n\log n)$ .

```
//Programa java que implementa la LIS

public class LongestIncreasingSubsequence {

    static int max_ref;
    /* Para hacer uso de llamadas recursivas, esta función debe devolver.
    dos cosas:
    1) Longitud de LIS que termina con el elemento arr [n-1]. Usamos
        max_ending_here para este propósito
    2) Máximo global ya que el LIS puede terminar con un elemento
        antes de arr [n-1] max_ref se usa para este propósito.
    El valor de LIS de la matriz completa de tamaño n se almacena en
    * max_ref cual es nuestro resultado final*/
    static int _lis(int arr[], int n) {
        // Caso base
        if (n == 1) {
            return 1;
        }
        // 'max_ending_here' es el tamaño de LIS
        // terminando con arr[n-1]
        int res, max_ending_here = 1;
        for (int i = 1; i < n; i++) {
            res = _lis(arr, i);
            if (arr[i - 1] < arr[n - 1] && res + 1 > max_ending_here) {
                max_ending_here = res + 1;
            }
        }
        if (max_ref < max_ending_here) {
            max_ref = max_ending_here;
        }
        return max_ending_here;
    }

    static int lis(int arr[], int n) {
```

```

// Guarda el resultado
    max_ref = 1;
//Almacena su resultado en max
    _lis(arr, n);
//Retorna el maximo
    return max_ref;
}

public static void main(String args[]) {
    int arr[] = {10, 22, 9, 33, 21, 50, 41, 60};
    int n = arr.length;
System.out.println("Tamaño de la subsecuencia incremental más larga"
    + " es " + lis(arr, n) + "\n");
}
}

```

## 21.4 Longest Common SubString

Dados dos strings X y Y encuentre el substring común más largo. Dejaremos que M y N sean las longitudes del primer y segundo string respectivamente.

Una simple solución es uno por uno considerar todos los substrings del primer string y por cada substring verificar si es un substring en el segundo substring, mantenemos rastro del string de longitud máxima, ahí pueden haber  $O(m^2)$  substrings y podemos encontrar si un string es substring de otro string en tiempo  $O(n)$ , entonces el tiempo general de este método será  $O(n * m^2)$

Programación dinámica puede ser usada para encontrar el string común más largo en  $O(n*m)$  tiempo, la idea es encontrar el sufijo común más largo para todos los substrings de ambos strings y almacena estas longitudes en la tabla.

El sufijo común más largo tiene propiedad de subestructura propia.

Si el último carácter coincide, entonces reducimos ambas longitudes en 1.

- $LCSuff(X, Y, m, n) = LCSuff(X, Y, m-1, n-1) + 1 \text{ if } X[m-1] = Y[n-1]$

Si el último carácter no coincide, entonces el resultado es 0, por ejemplo:

- $LCSuff(X, Y, m, n) = 0 \text{ if } (X[m-1] \neq Y[n-1])$

Ahora consideramos sufijos de diferentes substrings terminando en diferentes índices, el sufijo común más largo, su longitud es el substring común más largo.

- $\text{LCSubStr}(X, Y, m, n) = \text{Max}(\text{LCSuff}(X, Y, i, j))$  donde  $1 \leq i \leq m$  y  $1 \leq j \leq n$

Complejidad de tiempo:  $O(m*n)$

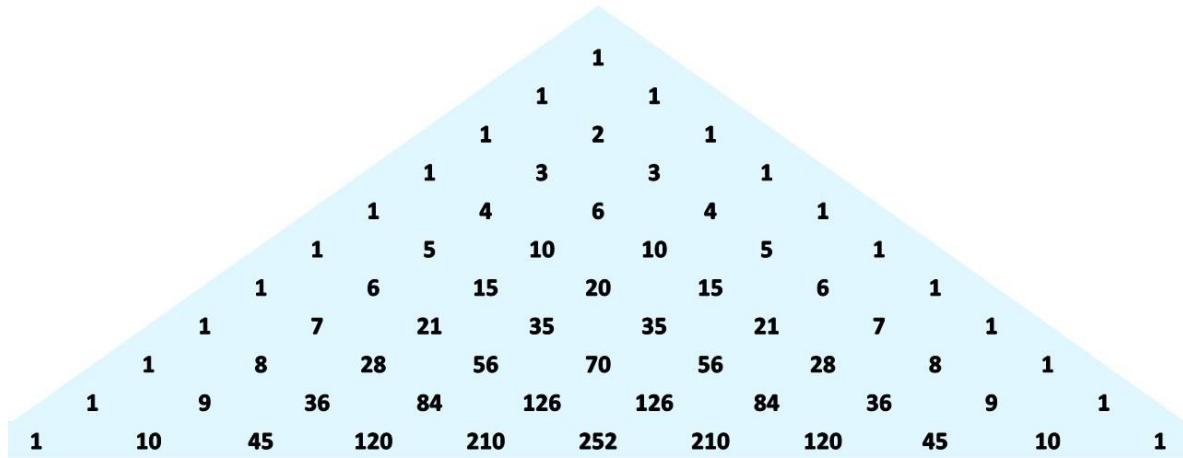
```
/* Implementación java que encuentra el substring comun más
largo usando programación dinamica*/
public class LongestCommonSubString
{
    static int LCSubStr(char X[], char Y[], int m, int n)
    {

        int LCStuff[][] = new int[m + 1][n + 1];
        int result = 0; //Para almacenar el tamaño del substring
        for (int i = 0; i <= m; i++)
        {
            for (int j = 0; j <= n; j++)
            {
                if (i == 0 || j == 0)
                    LCStuff[i][j] = 0;
                else if (X[i - 1] == Y[j - 1])
                {
                    LCStuff[i][j] = LCStuff[i - 1][j - 1] + 1;
                    result = Integer.max(result, LCStuff[i][j]);
                }
                else
                    LCStuff[i][j] = 0;
            }
        }
        return result;
    }

    public static void main(String[] args)
    {
        String X = "hola mundo";
        String Y = "mundo";
        int m = X.length();
        int n = Y.length();
        System.out.println("El tamaño del substring común más largo es "
        + LCSubStr(X.toCharArray(), Y.toCharArray(), m, n));
    }
}
```

## 21.5 Pascal Triangle

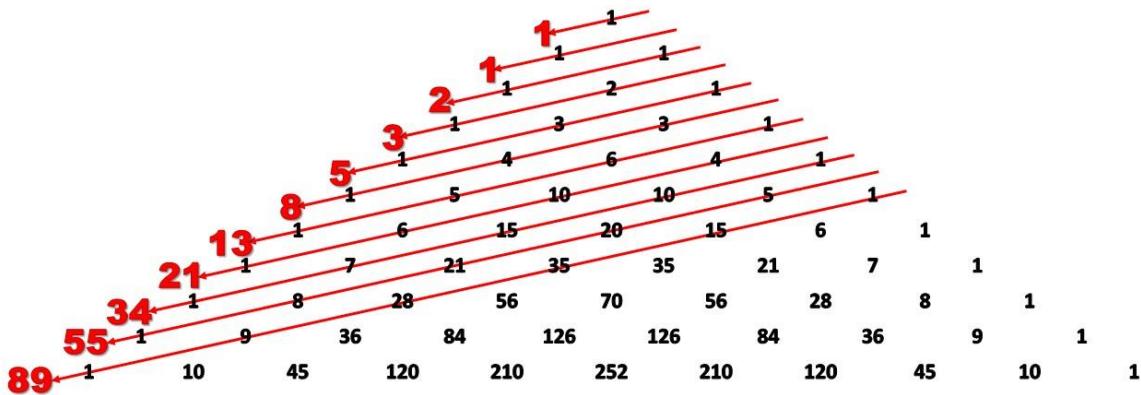
# Triángulo de Pascal



*Guía del programador competitivo.*

Ilustración 21-1 Triángulo de pascal de  $N=10$

## Sucesión de Fibonacci en el triángulo de Pascal



*Guía del programador competitivo.*

Ilustración 21-2 Números de Fibonacci en el triángulo de Pascal

El triángulo de Pascal es un triángulo de números enteros, infinito y simétrico. Se empieza con un 1 en la primera fila, y en las filas siguientes se van colocando números de forma que cada uno de ellos sea la suma de los dos números que tiene encima. Se supone que los

lugares fuera del triángulo contienen ceros, de forma que los bordes del triángulo están formados por unos.

- Los números del triángulo de Pascal coinciden con los números combinatorios.
- El número combinatorio  $C_m n$  ( $n$  sobre  $m$ ) se encuentra en el triángulo en la fila  $n+1$ , en el lugar  $m+1$ .
- El número combinatorio  $C_m n$  ( $n$  sobre  $m$ ) que representa el número de grupos de  $m$  elementos que pueden hacerse de entre un conjunto de  $n$  (por ejemplo, (4 sobre 2) nos da el número de parejas distintas que podrían hacerse en un grupo de cuatro personas), se encuentra en el triángulo en la fila  $n+1$ , en el lugar  $m+1$ .
- Podemos saber que el número de parejas posibles que decíamos antes es 6 si miramos el tercer número de la quinta fila.

Esto hace que el triángulo sea útil como representación de estos números, y proporciona una buena forma de intuir sus propiedades.

- La fórmula general del llamado Binomio de Newton  $(a + b)^n$  está formada por unos coeficientes que coinciden con la línea número  $n+1$  del triángulo de Pascal (la que empieza por 1 y  $n$ ).
  - Si el primer elemento de una fila es un número primo, todos los números de esa fila serán divisibles por él (menos el 1, claro). Así, en la fila 7: (1 7 21 35 35 21 7 1), los números 7, 21 y 35 son divisibles por 7.
  - La suma de los elementos de cualquier fila es el resultado de elevar 2 al número que define a esa fila.
- La serie de Fibonacci puede ser encontrada también en el triángulo de Pascal. Dividiendo al mismo según las líneas que mostramos en el diagrama, los números atrapados entre ellas suman cada uno de los elementos de esta sucesión.

```
//Código java que realiza el triángulo de Pascal
public class PascalTriangle {

    static void printPascal(int n) {
        /* Itera atraves de cada linea y la imprime
        con sus entradas*/
    }
}
```

```

        for (int line = 0; line < n; line++) {
// Cada linea tiene un número de enteros
// igual al número de linea
            for (int i = 0; i <= line; i++) {
                System.out.print(binomialCoeff(line, i) + " ");
            }
            System.out.println();
        }
    }

    static int binomialCoeff(int n, int k) {
        int res = 1;
        if (k > n - k) {
            k = n - k;
        }
        for (int i = 0; i < k; ++i) {
            res *= (n - i);
            res /= (i + 1);
        }
        return res;
    }

    public static void main(String args[]) {
int n = 7;
        printPascal(n);
    }
}

```

## 21.6 Problemas de repaso

### Ejercicios en Online Judge

147-Dollars

10130-SuperSale

357-Let Me Count The Ways

1196-Tiling Up Blocks

990-Diving for Gold

10029-Edit Step Ladders

1213-Sum Of Different Primes

10534-Wavio Sequence

## **22 Lista de tablas**

Tabla 2-1 Ejemplo de nombres de archivo fuente validos .....	16
Tabla 2-2: Tipos de casos de prueba .....	17
Tabla 9-1 Operadores lógicos, comparativos y matemáticos con ejemplo .....	36
Tabla 9-2: Tipos de variable y sus tamaños en tres lenguajes de programación.....	37
Tabla 9-3 Ejemplos de complejidad de tiempo .....	47

## 23 Lista de ilustraciones

Ilustración 2-1: Partes principales de un problema de programación competitiva.....	14
Ilustración 3-1 Logos JAVA, C++ y Python, Tomados de: <a href="https://www.theverge.com">https://www.theverge.com</a> , <a href="https://adictoalcodigo.blogspot.com">https://adictoalcodigo.blogspot.com</a> y <a href="https://anthoncode.com">https://anthoncode.com</a> respectivamente.....	24
Ilustración 5-1 Logos ACIS/REDIS Tomado de: <a href="http://www.evaluamos.com/images2017A/16219-1.jpg">http://www.evaluamos.com/images2017A/16219-1.jpg</a> .....	27
Ilustración 5-2 Logo CCPL Tomado de: <a href="http://www.programingleague.org">www.programingleague.org</a> .....	28
Ilustración 6-1 Logo ACM-ICPC Tomado de: <a href="http://www.cs.cornell.edu/acm/">http://www.cs.cornell.edu/acm/</a> .....	29
Ilustración 6-2 ICPC WORLD FINALS 2018 Tomado de: <a href="http://www.topcoder.com">www.topcoder.com</a> .....	30
Ilustración 7-1 Inscripciones CODEJAM 2018 Tomado de: <a href="https://www.youtube.com/watch?v=ipdUjbK1_h8">https://www.youtube.com/watch?v=ipdUjbK1_h8</a> .....	32
Ilustración 8-1 Logo CodeChef y Directi, Tomado de: <a href="http://www.codechef.com">www.codechef.com</a> .....	33
Ilustración 9-1 Logo Online Judge 2019, Tomado de : <a href="https://onlinejudge.org/">https://onlinejudge.org/</a> .....	34
Ilustración 9-2 Anterior logo de UVa Online Judge, Tomado de: <a href="http://nafischonchol.blogspot.com">http://nafischonchol.blogspot.com</a> .....	34
Ilustración 10-1 Ejemplo de vector.....	50
Ilustración 10-2 Ejemplo de matriz .....	52
Ilustración 10-3 Ejemplo de pila .....	52
Ilustración 10-4 Ejemplo de cola .....	53
Ilustración 12-1 Ejemplo de búsqueda binaria.....	63
Ilustración 12-2 Ejemplo búsqueda por saltos .....	69
Ilustración 13-1 Ejemplo ordenamiento burbuja .....	76

Ilustración 13-2 Ejemplo ordenamiento rápido .....	78
Ilustración 13-3 Ejemplo ordenamiento cascara.....	82
Ilustración 13-4 Vista abstracta de un árbol binario de búsqueda .....	86
Ilustración 13-5 Ejemplo de ordenamiento por unión.....	88
Ilustración 14-1 Ejemplo de búsqueda de patrones Aho-Corasick .....	105
Ilustración 14-2 Ejemplo de búsqueda de patrón Boyer-Moore .....	111
Ilustración 14-3 Búsqueda de patrón KMP.....	116
Ilustración 14-4 Búsqueda de patrón por Rabin-Karp.....	121
Ilustración 15-1 Ejemplo de GCD y LCM .....	128
Ilustración 15-2 Vista en matriz de búsqueda de primos por medio de la criba de Eratóstenes .....	134
Ilustración 15-3 Búsqueda de todos los subconjuntos de un conjunto (Set).....	141
Ilustración 15-4 Juego "Las Torres de Hanoi" .....	142
Ilustración 15-5 Flavio Josefo y sus compañeros soldados en el circulo.....	160
Ilustración 15-6 Principales componentes de los números Romanos .....	163
Ilustración 16-1 Arco dentro de un circulo de radio R .....	198
Ilustración 16-2 Áreas de diferentes polígonos.....	208
Ilustración 18-1 Matriz de adyacencia de un grafo .....	256
Ilustración 18-2 Lista de listas de adyacencia de un grafo .....	258
Ilustración 18-3 Ejemplo de búsqueda en profundidad .....	270
Ilustración 18-4 Ejemplo de diferentes grafos .....	275
Ilustración 18-5 Ciclo Hamiltoniano en un grafo.....	285
Ilustración 18-6 Árbol de expansión mínima en un grafo .....	297

Ilustración 18-7 Ejemplo de coloración de un grafo bipartito .....	328
Ilustración 19-1 Triangulo de pascal de N=10 .....	339
Ilustración 19-2 Números de Fibonacci en el triángulo de Pascal .....	339

## 24 Bibliografía

- 1) Amraii, S. A. (2006). Observations on Teamwork Strategies in the. *Crossroads, The ACM Student Magazine*, 1-3.
- 2) Asociación Colombiana de Ingenieros de Sistemas. (10 de Septiembre de 2019). *ACIS*. Recuperado el 11 de Septiembre de 2019, de <https://acis.org.co/portal/>
- 3) Combéfis, S. (2014). Programming Trainings and Informatics Teaching. *Olympiads in Informatics*, 21-34.
- 4) CPP Reference. (2019). *Cpp Reference*. Recuperado el 10 de Septiembre de 2019, de <https://en.cppreference.com/w/>
- 5) Directi. (10 de Septiembre de 2019). *About CodeChef*. Obtenido de <https://www.codechef.com/aboutus>
- 6) Francesc Comellas, J. F. (2001). *Matemática Discreta*. Catalunya: Edicions de la Universitat Politècnica de Catalunya, SL.
- 7) GeeksforGeeks. (10 de Septiembre de 2019). *GeeksforGeeks About US*. Obtenido de <https://www.geeksforgeeks.org/about/>
- 8) Johnsonbaugh, R. (2005). *Matematicas Discretas 6ta edición*. Ciudad de Mexico, Mexico: Prentice Hall.
- 9) Oracle. (Septiembre de 2019). *Oracle*. Recuperado el 10 de Septiembre de 2019, de [https://www.java.com/es/download/help/index\\_using.xml](https://www.java.com/es/download/help/index_using.xml)
- 10) Python Software Foundation. (9 de Septiembre de 2019). *Docs Python*. Obtenido de <https://docs.python.org/3/>
- 11) Rob Hoogerwoord, H. Z. (2016). *Discrete Structures*. 2IT50.
- 12) Uva Online Judge. (12 de Julio de 2012). *UVa OJ Board*. Recuperado el 10 de Septiembre de 2019, de <https://uva.onlinejudge.org/board/viewtopic.php?t=71042>

- 13) Zhu Jie-ao, S. M. (2005). Learning Software Engineering through Experience of ACM-ICPC Training and Practicing Exercises. *ACM/ICPC*.