

**Universidad Nacional Mayor de San Marcos**

**FACULTAD DE INGENIERÍA DE SISTEMAS E INFORMÁTICA**

**Informe Final PROYECTO**

Logotipo, nombre de la empresa

Descripción generada automáticamente

**Curso:** Análisis y Diseño de Algoritmos **Profesor :** Robert Elías Espinoza Domínguez

**Escuela:** Ingeniería de Software

**GRUPO 5**

**Integrantes:**

* Caicedo Porras Jose Ricardo
* Jauregui Romero Eduardo Rafael
* Cachique Falcon Williams Eduardo
* Otoya Chaname Miguel Antonio
* Ocaña Lujan Jose Esteban

FISI

**Lima - Perú**

**Julio, 2021**

**Índice**

[**1.** **Objetivos del proyecto** 3](#_Toc81575441)

[1.1. Antecedentes 3](#_Toc81575442)

[1.2. Descripción del problema 3](#_Toc81575443)

[1.3. Formulación del problema 3](#_Toc81575444)

[1.4. Justificación 3](#_Toc81575445)

[**2.** **Descripción del proyecto** 4](#_Toc81575446)

[2.1. Alcances 4](#_Toc81575447)

[2.2. Limitaciones 4](#_Toc81575448)

[**3.** **Requerimientos** 5](#_Toc81575449)

[**4.** **DIAGRAMA DE CLASES** 1](#_Toc81575450)

[**5.** **DICCIONARIO DE CLASES** 1](#_Toc81575451)

[**6.** **Algoritmos Utilizados** 1](#_Toc81575452)

[6.1. Problema del viajante 1](#_Toc81575453)

[6.2. Tablas Hash - Colisión por encadenamiento 2](#_Toc81575454)

[6.3. Algoritmo de Dijkstra 4](#_Toc81575455)

[6.4. Algoritmo de Floyd-Warshall 6](#_Toc81575456)

[6.5. Programación dinámica 9](#_Toc81575457)

[6.6. Programación Voraz 10](#_Toc81575458)

[**7.** **Análisis de Complejidad** 11](#_Toc81575459)

[7.1.1. Análisis de complejidad algoritmo Dijkstra 11](#_Toc81575460)

[7.1.2. Análisis de complejidad algoritmo Floyd-Warshall 11](#_Toc81575461)

[11](#_Toc81575462)

[7.1.3. Análisis de complejidad tabla hash 12](#_Toc81575463)

[**8.** **MANUAL DE USUARIO** 12](#_Toc81575464)

[**9.** **CONCLUSIONES Y RECOMENDACIONES** 12](#_Toc81575465)

[**10.** **BIBLIOGRAFÍA** 13](#_Toc81575466)

# **Objetivos del proyecto**

## Antecedentes

En los últimos meses ir a una planta abastecedora de oxígeno lo más rápido posible puede significar la vida o la muerte para uno de nuestros compatriotas. En las redes sociales se comparten distintos puntos en el cual uno puede ir a recargar su balón de oxígeno, sin embargo, no existe un mapeo que nos muestre cual es el más cercano en relación con nuestra ubicación actual. En el Perú existen algunas plantas de oxígeno que no son muy conocidas debido a que la información, no está organizada en un software específico.

## Descripción del problema

A causa del problema sanitario del COVID 19, se implementaron en hospitales, espacios distritales, parroquias, etc. puntos de abastecimiento de oxígeno. Las personas que necesitan recargar los balones de oxígeno se deben acercar a los abastecimientos y hacer la recarga de sus balones. En esta situación el tiempo juega un papel importante para la vida de sus familiares, y el no saber cómo acudir de la forma más rápida y eficiente ralentizará el llenado, lo cual desencadenará que su familiar se quede sin este recurso primordial, esto podría desencadenar pérdidas de vidas humanas. La falta de un sistema de rutas mínimas hacia los lugares de abastecimiento se vuelve fundamental por la importancia que tiene el oxígeno en épocas de pandemia

## Formulación del problema

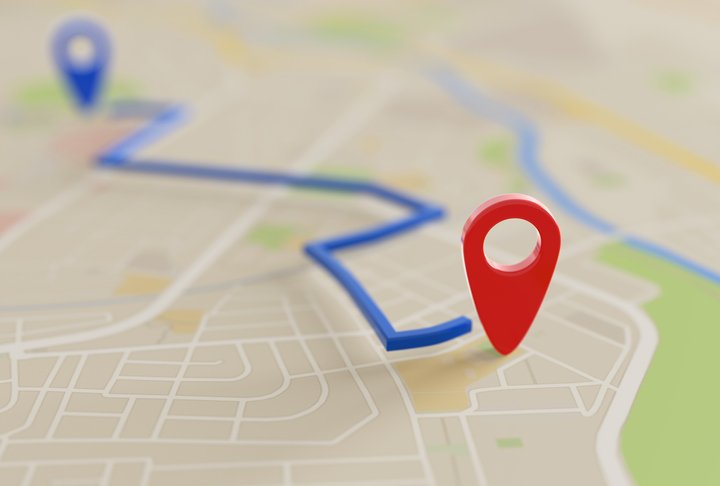
¿Cómo desarrollar un software que permita a un usuario encontrar la ruta más corta para llegar a un centro de abastecimiento de oxígeno dependiendo de su ubicación?

## Justificación

Se hace necesario realizar este proyecto para los ciudadanos de Lima con el fin de agilizar el tiempo de búsqueda, ubicación y traslado de las personas a los centros de abastecimiento de oxígeno. La tecnología siempre ayuda a mejorar los procesos y si se aplica al ámbito de la Salud. El fin de este proyecto es proporcionar a cualquier persona de Lima la posibilidad de visualizar la ruta más corta sin necesidad de recorrer tramos innecesarios, de esta manera podrá saber cuál ruta le resulta más eficiente, esto hará que ahorre mucho tiempo y pueda salvar vidas.

# **Descripción del proyecto**

El Qhapaq Ñan del oxígeno es un software que radica en ubicar los lugares de abastecimiento de una manera correcta, ordenada dentro de la aplicación. La aplicación permitirá al usuario logueado poder ubicarse dentro de su distrito y seleccionar el centro de abastecimiento de oxígeno al cual quiera dirigirse. Le mostrará la ruta más corta que este debe seguir para dirigirse al centro seleccionado. También ofrecerá una información detallada de cada centro de abastecimiento.



## Alcances

* El presente software permite ubicar los centros de abastecimiento de oxígeno.
* El presente software tomará en consideración a los centros de abastecimiento de oxígeno más resaltantes hasta agosto de 2021.
* El programa será exclusivo para escritorio.

## Limitaciones

* El equipo de desarrollo cuenta con más responsabilidades por lo que el tiempo para el desarrollo se vuelve más limitado.
* Escasez de información confiable obtenida en la recolección de datos en los centros de abastecimiento de oxígeno.
* El equipo de desarrollo cuenta con poco conocimiento sobre el uso de librerías gráficas como el swing.

# **Requerimientos**

|  |  |  |
| --- | --- | --- |
| **ID** | **REQUISITO FUNCIONAL** | **DESCRIPCION** |
| **RF—01** | ***REGISTRARSE*** | El sistema debe permitir al usuario  crearse una cuenta con su DNI y contraseña |
| **RF—02** | ***INICIAR SESION*** | El sistema debe permitir al usuario  iniciar sesión con su DNI y contraseña escogidos |
| **RF—03** | ***SELECCIONAR DISTRITO*** | El aplicativo permitirá al usuario  seleccionar el distrito en el cual se encuentra |
| **RF—04** | ***SELECCIONAR UBICACIÓN*** | El software permitirá al usuario seleccionar la ubicación más  próxima a él en función a centros referenciales. |
| **RF—05** | ***SELECCIONAR DISTRIBUIDORA*** | El aplicativo permitirá al usuario establecer seleccionar la  distribuidora de su preferencia en pantalla |
| **RF—06** | ***VER RUTA MINIMA*** | La aplicación mostrará mediante pantalla la ruta mínima para llegar  desde su ubicación a centro de distribución de oxígeno seleccionado previamente |
| **RF—07** | ***VER INFORMACION*** | El software permitirá al usuario ver  la información detallada del centro distribuidor. |
| **RF—08** | ***CERRAR SESION*** | El software permitirá al usuario cerrar sesión  en el sistema |

# Imagen que contiene Texto Descripción generada automáticamente**DIAGRAMA DE CLASES**

# **DICCIONARIO DE CLASES**

PARA EL DICCIONARIO DE DATOS UTILIZAMOS JAVA DOCS, EL RESULTADO SE PUEDE VER EN EL SIGUIENTE VINCULO:

[**Overview (dunned.github.io)**](https://dunned.github.io/)

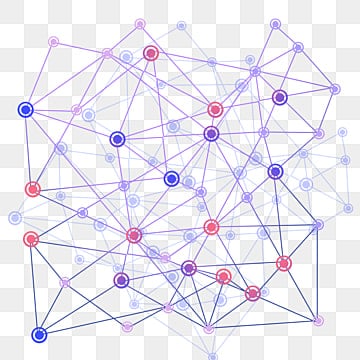
# **Algoritmos Utilizados**

## Problema del viajante

El problema del viajante fue definido en los años 1800s por el matemático irlandés W. R. Hamilton y por el matemático británico Thomas Kirkman. Parece ser que la forma general del problema del viajante fue estudiada, por primera vez por matemáticos en Viena y Harvard, durante los años 1930s.

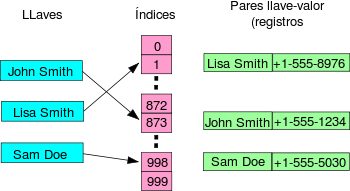
El problema del viajante da respuesta a la interrogante de teniendo una lista de ciudades y las distancias entre ellas, ¿cuál sería la ruta más corta que visita cada ciudad exactamente una vez y al finalizar regresa a la ciudad origen? Es usado como prueba para muchos métodos de optimización. Aunque el problema es computacionalmente complejo, se conoce gran cantidad de heurísticas y métodos exactos, así que es posible resolver planteamientos concretos del problema desde cien hasta miles de ciudades.

El problema del viajante se puede modelar como un grafo, de manera que las ciudades sean los vértices del grafo, los caminos son las aristas y las distancias de los caminos son los pesos de las aristas. Esto es un problema de minimización que comienza y termina en un vértice específico y se visita el resto de los vértices exactamente una vez. Con frecuencia, el modelo es un grafo completo.



## Tablas Hash - Colisión por encadenamiento

La tabla hash es una estructura de datos que pretende la inserción, búsqueda y borrado de elementos en tiempo constante. La manera más simple de resolver una colisión es construir, para cada localización de la tabla, una lista enlazada de registros cuyas claves caigan en esa dirección. Este método se conoce normalmente con el nombre de *encadenamiento separado* y obviamente la cantidad de tiempo requerido para una búsqueda dependerá de la longitud de las listas y de las posiciones relativas de las claves en ellas.



**registro** par {llave, valor }

**var** *vector de pares* casilla [0..numcasillas-1]

**function** buscacasilla(llave) {

i:= hash(llave) módulo de numcasillas

**loop** {

**if** casilla[i] esta libre **or** casilla[i]. llave = llave

**return** i

i:= (i + 1) módulo de numcasillas

}

}

**function** búsqueda(llave)

i:= buscacasilla(llave)

**if** casilla[i] está ocupada *// llave está en la tabla*

**return** casilla[i]. valor

**else** *// llave no está en la tabla*

**return** no encontrada

**function** asignar (llave, valor) {

i:= buscacasilla(llave)

**if** casilla[i] está ocupada

casilla[i]. valor := valor

**else** {

**if** tabla casi llena {

hacer tabla más grande *(nota 1)*

i:= buscacasilla(llave)

}

casilla[i]. llave := llave

casilla[i]. valor := valor

}

}

**CODIGO EN JAVA**

Texto

Descripción generada automáticamenteTexto

Descripción generada automáticamenteTexto

Descripción generada automáticamente**CODIGO EN JAVA**

## Algoritmo de Dijkstra

Fue diseñado por Edsger Wybe Dijkstra en 1959. Es un ejemplo típico de algoritmo ávido (voraz) que selecciona en cada paso la solución óptima para resolver el problema. El algoritmo de Dijkstra determina el camino más corto desde un vértice al resto de los vértices del grafo, dado un grafo dirigido G=(V, A) valorado y con factores de peso no negativos, donde la longitud de un camino es la suma de los pesos de las aristas que lo forman.

La idea subyacente en este algoritmo consiste en ir explorando todos los caminos más cortos que parten del vértice origen y que llevan a todos los demás vértices; cuando se obtiene el camino más corto desde el vértice origen hasta el resto de los vértices que componen el grafo, el algoritmo se detiene. Se trata de una especialización de labúsqueda del costo uniforme y, como tal, no funciona en grafos con aristas de coste negativo (al elegir siempre el nodo con distancia menor, pueden quedar excluidos de la búsqueda nodos que en próximas iteraciones bajaría el costo general del camino al pasar por una arista con costo negativo).

***Características del algoritmo***

• Es un algoritmo voraz

• Trabaja por etapas, y toma en cada etapa la solución óptima.

• El algoritmo puede modificar el óptimo encontrado, si

posteriormente surge una solución mejor

Pasos del algoritmo

**Algoritmo de Dijkstra.** **Inicialización.**

* Sea V un conjunto de vértices de un grafo.
* Sea C una matriz de costos de las aristas del grafo, donde en C[u, v] se almacena el costo de la arista entre u y v.
* Sea S un conjunto que contendrá los vértices para los cuales ya se tiene determinado el camino mínimo.
* Sea D un arreglo unidimensional tal que D[v] es el costo del camino mínimo del vértice origen al vértice v.
* Sea P un arreglo unidimensional tal que P[v] es el vértice predecesor de v en el camino mínimo que se tiene construido.
* Sea vinicial el vértice origen. Recordar que el Algoritmo Dijkstra determina los caminos mínimos que existen partiendo de un vértice origen al resto de los vértices.

**función Dijkstra (Grafo G, nodo\_salida s)**

**//Usaremos un vector para guardar las distancias del nodo salida al resto entero distancia[n]**

**//Inicializamos el vector con distancias iniciales booleano visto[n]**

**//vector de boleanos para controlar los vertices de los que ya tenemos la distancia mínima**

**para cada w ∈ V[G] hacer**

**Si (no existe arista entre s y w) entonces**

**distancia[w] = Infinito //puedes marcar la casilla con un -1 por ejemplo**

**Si\_no**

**distancia[w] = peso (s, w)**

**fin si**

**fin para**

**distancia[s] = 0**

**visto[s] = cierto**

**//n es el número de vertices que tiene el Grafo**

**mientras que (no\_esten\_vistos\_todos) hacer**

**vertice = coger\_el\_minimo\_del\_vector distancia y que no esté visto;**

**visto[vertice] = cierto;**

**para cada w ∈ sucesores (G, vertice) hacer**

**sí distancia[w]>distancia[vertice]+peso (vertice, w) entonces**

**distancia[w] = distancia[vertice]+peso (vertice, w)**

**fin si**

**fin para**

**fin mientras**

**fin función**

**CODIGO EN JAVA**

Texto

Descripción generada automáticamente

Texto

Descripción generada automáticamente

## Algoritmo de Floyd-Warshall

En informática, el algoritmo de Floyd-Warshall, descrito en 1959 por Bernard Roy, es un algoritmo de análisis sobre grafos para encontrar el camino mínimo en grafos dirigidos ponderados. Este algoritmo nos ayuda a encontrar el camino entre todos los pares de vértices en una única ejecución. El algoritmo de Floyd-Warshall es un ejemplo de programación dinámica.

Es un algoritmo de análisis de grafos para encontrar el camino mínimo en grafos dirigidos o no dirigidos ponderados.

El problema que intenta resolver este algoritmo es el de encontrar el camino más corto entre todos los pares de nodos o vértices de un grafo. Esto es semejante a construir una tabla con todas las distancias mínimas entre pares de ciudades de un mapa, indicando además la ruta a seguir para ir de la primera ciudad a la segunda. Este es uno de los problemas más interesantes que se pueden resolver con algoritmos de grafos.

Existen varias soluciones a este problema y los algoritmos a aplicar dependen también de la existencia de arcos con pesos o costes negativos en el grafo. En el caso de no existir pesos negativos, sería posible ejecutar V veces el algoritmo de Dijkstra para el cálculo del camino mínimo, donde V es el número de vértices o nodos del grafo. Esto conlleva un tiempo de ejecución de O (V^3) (aunque se puede reducir). Si existen arcos con pesos negativos, se puede ejecutar también V veces el Algoritmo de Bellman-Ford, una vez para cada nodo del grafo. Para grafos densos (con muchas conexiones o arcos) esto conlleva un tiempo de ejecución de O (V^3).

***Características del algoritmo***

●Obtiene la mejor ruta entre todo par de nodos.

●Trabaja con la matriz D inicializada con las distancias directas entre todo

par de nodos.

●La iteración se produce sobre nodos intermedios, o sea para todo elemento

de la matriz se prueba si lo mejor para ir de i a j es a través de un nodo

intermedio elegido o como estaba anteriormente, y esto se prueba con todos

los nodos de la red. Una vez probados todos los nodos de la red como nodos

intermedios, la matriz resultante da la mejor distancia entre todo par de

nodos.

●El algoritmo da sólo la menor distancia; se debe manejar información

adicional para encontrar tablas de encaminamiento.

●Hasta no hallar la última matriz no se encuentran las distancias mínimas.

●Su complejidad es del orden de O(n^3).

Descripción de los datos

* Matriz C = Es el conjunto de distancias entre cada par de puntos, algunos de ellos no cuentan con un camino directo al cual poder acceder (por lo que se agrega una distancia infinita para representarlo).
* Matriz Z = Es la matriz de nodos antecesores, en donde cada elemento es el punto directo anterior con la distancia más corta, determina las rutas mínimas.
* [*i*, *j*] = es la ubicación de un elemento en una matriz, donde ***i*** es la fila a la que pertenece y***j*** la columna.
* *k* = la iteración que se está llevando a cabo, para saber en qué parte del proceso nos encontramos.

Método Floyd (L, n)

// Inicializa las matrices

Para i desde 1 hasta n hacer

Para j desde 1 hasta n hacer

D [i, j]  L[i, j]

P [i, j]  0

Fin Para

Fin Para

// Cargar matrices D y P en cada iteración k

Para k desde 1 hasta n hacer

Para i desde 1 hasta n hacer

Para j desde 1 hasta n hacer

Si (D [i, k]+ D[k, j]) < D[i, j] entonces

D [i, j]  D[i, k]+ D[k, j]

P [i, j]  k

Fin Si

Fin Para

Fin Para

Fin Para

Fin Método

Texto

Descripción generada automáticamente**CODIGO EN JAVA**

**CODIGO PARA HALLAR MATRIZ DE CAMINOS**

Texto

Descripción generada automáticamente

**CODIGO PARA RECUPERAR EL RECORRIDO**

Texto

Descripción generada automáticamente

## Programación dinámica

La programación dinámica es un método para reducir el tiempo de ejecución de un algoritmo mediante la utilización de subproblemas superpuestos y subestructuras óptimas, como se describe a continuación.

Esta técnica se aplica sobre problemas que presentan las siguientes características:

* Subproblemas optímales: La solución óptima a un problema puede ser definida en función de soluciones óptimas a subproblemas de menor tamaño.
* Solapamiento entre subproblemas: Al plantear la solución recursiva del problema, un mismo problema se resuelve más de una vez.
* Uso de la programación dinámica
  + Caracterizar la estructura de una solución óptima
  + Definir de forma recursiva la solución óptima
  + Calcular la solución óptima de forma ascendente
  + Construir la solución óptima a partir de los datos almacenados al obtener soluciones parciales

Al igual que la técnica de Divide y Vencerás, la programación dinámica es una técnica para resolver problemas a partir de la solución de subproblemas y la combinación de esas soluciones. A diferencia de Divide y vencerás, la programación dinámica es aplicable cuando los subproblemas no son independientes. Un algoritmo que sigue esta técnica resuelve cada subproblema una sola vez y guarda su respuesta en una tabla.

Para que un problema pueda ser solucionado por esta técnica debe cumplir dos condiciones:

* La solución al problema ha de ser alcanzada mediante una secuencia de decisiones, una en cada etapa
* Dicha secuencia debe cumplir el principio de óptimo.

## Programación Voraz

Un algoritmo voraz (*greedy*) es un algoritmo que encuentra una solución globalmente óptima a un problema a base de hacer elecciones localmente óptimas. Es decir: el algoritmo siempre hace lo que “parece” mejor en cada momento, sin tener nunca que reconsiderar sus decisiones, y acaba llegando directamente a la mejor solución posible. Dado un conjunto finito de entradas C, un [algoritmo](https://www.ecured.cu/Algoritmo) voraz devuelve un conjunto S (seleccionados) tal que y que además cumple con las restricciones del problema inicial. Cada conjunto S que satisfaga las restricciones se le suele denominar prometedor, y si éste además logra que la función objetivo se minimice o maximice (según corresponda) diremos que S es una *solución óptima*.

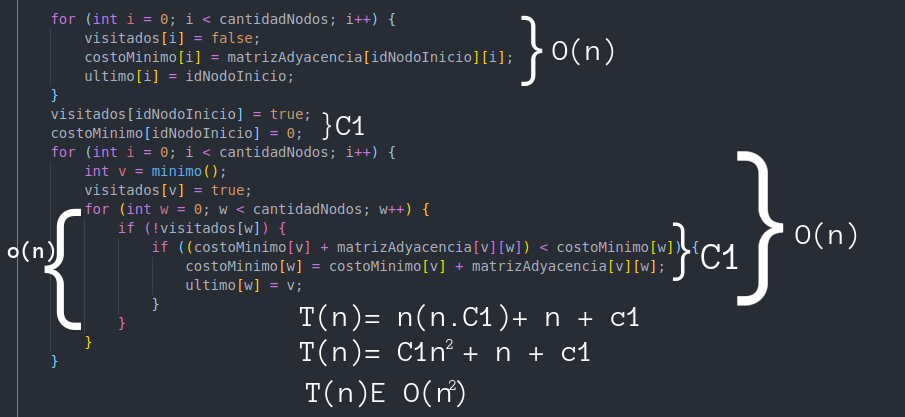
Elementos de los que consta la técnica:

* El conjunto C de **candidatos**, entradas del problema.
* Función **solución**. Comprueba, en cada paso, si el subconjunto actual de candidatos elegidos forma una solución (no importa si es óptima o no lo es).
* Función de **selección**. Informa de cuál es el elemento más prometedor para completar la solución. Éste no puede haber sido rechazado o escogido con anterioridad. Cada elemento es considerado una sola vez. Luego, puede ser rechazado o aceptado y pertenecerá a.
* Función de **factibilidad**. Informa si a partir de un conjunto se puede llegar a una solución. Lo aplicaremos al conjunto de seleccionados unido con el elemento más prometedor.
* Función **objetivo**. Es aquella que queremos maximizar o minimizar, el núcleo del problema.



## **Análisis de Complejidad**

### Análisis de complejidad algoritmo Dijkstra



### Análisis de complejidad algoritmo Floyd-Warshall

### 

### Análisis de complejidad tabla hash

# **MANUAL DE USUARIO**

SE ADJUNTO EL DOCUMENTO DE MANUAL DE USUARIO:

[MANUAL DE USUARIO GOOGLE DRIVE](https://drive.google.com/file/d/1GVUA-cO0DS68O_tGPvkbCCkl0p5fAnK_/view?usp=sharing)

# **CONCLUSIONES Y RECOMENDACIONES**

* Una tabla hash tiene como principal ventaja que el acceso a los datos suele ser muy rápido si se cumplen que una razón de ocupación no muy elevada (a partir del 75% de ocupación se producen demasiadas colisiones y la tabla se vuelve ineficiente) y una función resumen que distribuya uniformemente las claves. Si la función está mal diseñada, se producirán muchas colisiones.
* Los algoritmos Voraces son la técnica más sencilla de resolución de problemas. Normalmente se trata de algoritmos que resuelven problemas de optimización, es decir, tenemos un problema que queremos resolver de manera óptima
* Programación dinámica es una técnica muy útil para tomar una sucesión de decisiones interrelacionadas. Requiere la formulación de una relación recursiva apropiada para cada problema individual
* El algoritmo de Dijkstra es un algoritmo eficiente (de complejidad O(n2) donde n es el número de vértices) que sirve para encontrar el camino de coste mínimo desde un nodo origen a todos los demás nodos del grafo.
* El algoritmo de Floyd-Warshall es un algoritmo de análisis de grafos para que, de forma eficiente y simultánea, encuentre los caminos más cortos dentro de un grafo en el cual las aristas tengan un costo

# **BIBLIOGRAFÍA**

* JOYANES AGUILAR, L. (2008). ESTRUCTURA DE DATOS EN JAVA (1a. ed., 1a. reimp.). MADRID: MCGRAW-HILL INTERAMERICANA.
* MARIO STORTI (2009) Algoritmos y Estructuras de Datos (1a. ed., 1a. reimp.). Santa Fe: Universidad Nacional del Litoral
* ROSA GUEREQUETA (1997) Técnicas de Diseño de Algoritmos
* Aho, Hopcroft, Ullman (1998). Estructuras de datos y Algoritmos. Addison-Wesley.
* Bailey, Duanes (1999). Data Structures in Java. McGraw-Hill.
* Barnes, David (2000). Object-Oriented Programming in Java: An introduction. Prentice-Hall.
* Budd, Timoty (2002). Object-Oriented Programming. Third Edition. Addison-Wesley.
* Deitel, Harvey (2002). JAVA. How to program. Fourth edition, Prentice-Hall.
* Heileman, G. (1998). Estructuras de datos, algoritmos y programación orientada a objetos. McGraw-Hill,
* Horstmann, Cornell (2006). Core Java 2 Volumen I–Fundamentos. Pearson Educación.
* Hubbard, John R. (2001). Data Structures with Java. McGraw-Hill, Schaum ́s OuTlines.
* Joyanes, Luis (2003). Fundamentos de programación. 3a edición; McGraw-Hill.
* Kuth, Donald (1985). Algoritmos fundamentales volumen I. Editorial Reverté.
* Lafore, Robert (2003). Data Structures & Algorithms in Java. Second Edition. Sams.
* Lipschutz, S (1989). Estructuras de datos. McGraw-Hill.
* Weiss, M.A. (1995). Data Structures and Algorithm Análisis. Benjamín/Cummings.
* Weiss, M.A (1999). Data Structures & Algorithm Analysis in Java. Addison-Wesley.
* Weiss, M.A (2000). Estructuras de datos en Java. Addison-Wesley.
* Zukowski, John (1999). Programación en Java 2. Anaya Multimedia.