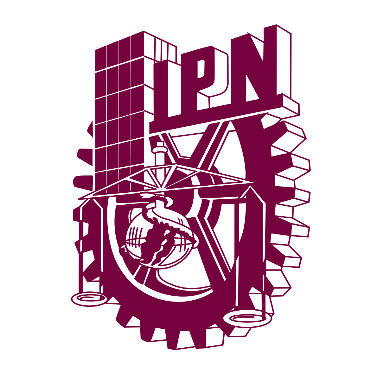
****

**Instituto Politécnico Nacional**

**Escuela Superior de Computo**

*Alumno:*

* Monroy Ramírez Oscar G.

***Grupo:*** *2CM23*

***Unidad de Aprendizaje:*** *Algoritmos y Estructuras de Datos*

***Evidencia:*** *Reporte 2do Parcial*

***Docente:*** *De Luna Caballero Roberto*

***Fecha****: 24 de noviembre del 2020*

Contenido

[Introducción 3](#_Toc57058710)

[Marco Teórico 4](#_Toc57058711)

[Algoritmia 4](#_Toc57058712)

[Características de los algoritmos y tipos. 4](#_Toc57058713)

[-Tipos 5](#_Toc57058714)

[Representación de Algoritmos y Pseudocódigo 5](#_Toc57058715)

[Abstracción y tipo de dato abstracto 8](#_Toc57058716)

[Tablas Hash 13](#_Toc57058717)

[Contexto de uso de una tabla hash 13](#_Toc57058718)

[Posibles implementaciones 14](#_Toc57058719)

[Tablas hash 14](#_Toc57058720)

[Gestión de colisiones mediante listas encadenadas 15](#_Toc57058721)

[Recursividad 15](#_Toc57058722)

[Ejemplo sencillo 16](#_Toc57058723)

[Factorial 17](#_Toc57058724)

[Backtracking 18](#_Toc57058725)

[Concepto 18](#_Toc57058726)

[Enfoque 19](#_Toc57058727)

[Diseño e implementación 19](#_Toc57058728)

[Heurísticas 20](#_Toc57058729)

[Ejemplos de aplicación de backtracking 20](#_Toc57058730)

[HAMILTON CYCLE (VIAJANTE DE COMERCIO) 21](#_Toc57058731)

[EXACT COVER 21](#_Toc57058732)

[Ejemplos de problemas comunes resueltos usando Vuelta Atrás 22](#_Toc57058733)

[Problema de las N Reinas 22](#_Toc57058734)

[Problema de la mochila 0,1 23](#_Toc57058735)

[Problema del laberinto 23](#_Toc57058736)

[Backtracking para la enumeración 23](#_Toc57058737)

[Aplicaciones 24](#_Toc57058738)

[Conclusión. 24](#_Toc57058739)

[Referencias Bibliográficas: 24](#_Toc57058740)

# Introducción

Una computadora es una maquina que maneja información. El estudio de la ciencia de la computación incluye saber la forma en que se organiza la información en una computadora, como puede manejarse y la manera en que puede utilizarse esa información. Por tanto, es de suma importancia para un estudiante de la computación, entender los conceptos de organización y manejo de la información para continuar con el estudio de nuestra diciplina.

Si la ciencia de la computación es fundamental el estudio de la información, la primera pregunta que surge es: ¿qué es la información? Por desgracia, aunque ese concepto es la piedra angular de la diciplina, la interrogante anterior no puede contestar con precisión. En que sentido, el concepto de la información en esta diciplina es similar a los conceptos de punto, recta y plano para la geometría: son un grupo de términos indefinidos con los que se pueden hacer proposiciones, pero no pueden definirse con conceptos más elementales.

En la geometría es posible hablar de la longitud de una recta, a pesar de que el concepto de recta es en si mismo indefinido. La longitud de una recta es una medida cuantitativa. De manera similar, en la ciencia de la computación podemos hablar de cantidades de información. La unidad básica de información es el ***bit*** cuyo valor confirma una de dos posibilidades excluyentes. Por ejemplo, un conmutador puede estar en una de dos posiciones pero no ambas al mismo tiempo pero el hecho es que al estar en una posición representa un bit.

Pues asi es la parte estructurada de la computacion, Que nos permite encontrar diversas formas métodos, con los cual es podemos avanzar en diversas partes yasea para usarlas como herramientas que funcionen como grandes precursores en todas las operaciones básicas que realizamos a la hora de realizar diferentes operaciones en nuestros propios programas, siendo así la estructura de datos una manera de construir un mapa para no perderse a la hora de El manejo de esa intensa y extensa información, métodos que veremos a lo largo del desarrollo de este trabajo.

# Marco Teórico

## Algoritmia

### Características de los algoritmos y tipos.

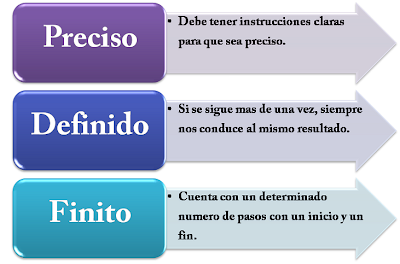
SegúnAlgoritmia y Programacion (2017) se denomina algoritmo a un grupo finito de operaciones organizadas de manera lógica y ordenada que permite solucionar un determinado problema. Se trata de una serie de instrucciones o reglas establecidas que, por medio de una sucesión de pasos, permiten arribar a un resultado o solución. Los algoritmos son el objeto de estudio de la algoritmia**.[ 1]**

SegúnAlgoritmia y Programacion (2017) se emplean algoritmos frecuentemente para resolver problemas. Algunos ejemplos son los manuales de usuario, que muestran algoritmos para usar un aparato, o las instrucciones que recibe un trabajador por parte de su patrón. Algunos ejemplos en matemática son el algoritmo de multiplicación, para calcular el producto, el algoritmo de la división para calcular el cociente de dos números, el algoritmo de Euclides para obtener el máximo común divisor de dos enteros positivos, o el método de Gauss para resolver un sistema de ecuaciones lineales. **[1]**

SegúnAlgoritmia y Programacion (2017) los algoritmos pueden ser expresados de muchas maneras, incluyendo el lenguaje natural, pseudocódigo, diagramas de flujo y lenguajes de programación, entre otros. Las descripciones en lenguaje natural tienden a ser ambiguas y extensas. El usar pseudocódigo y diagramas de flujo evita muchas ambigüedades del lenguaje natural. Dichas expresiones son formas más estructuradas para representar algoritmos; no obstante, los algoritmos son independientes de los lenguajes de programación. En cada problema el algoritmo puede escribirse y luego ejecutarse en un lenguaje de diferente programación. El algoritmo es la infraestructura de cualquier solución, escrita luego en cualquier lenguaje de programación**. [1]**

#### -Características

Según Delgado A. (2017) Las características de los algoritmos son:

* “Un algoritmo **debe ser preciso**: tiene que indicar el orden de realización de cada paso**.”[2]**
* “Un algoritmo **debe estar definido:** Si se sigue un algoritmo dos veces, se debe obtener el mismo resultado cada vez.”**[2]**
* “Un algoritmo **debe ser finito:** el algoritmo se debe terminar en algún momento; o sea, debe tener un número finito de pasos.” **[2]**

**Imagen 1 “Características iniciales” [2]**

* “Un algoritmo **debe ser legible:**El texto que lo describe debe ser claro, tal que permita entenderlo y leerlo fácilmente. **“[2]**
* “Un algoritmo **debe definir tres partes:** Entrada, Proceso y Salida”**[2]**

### -Tipos

Según Algorítmica y Programación (2017) Los tipos de Algoritmos son:

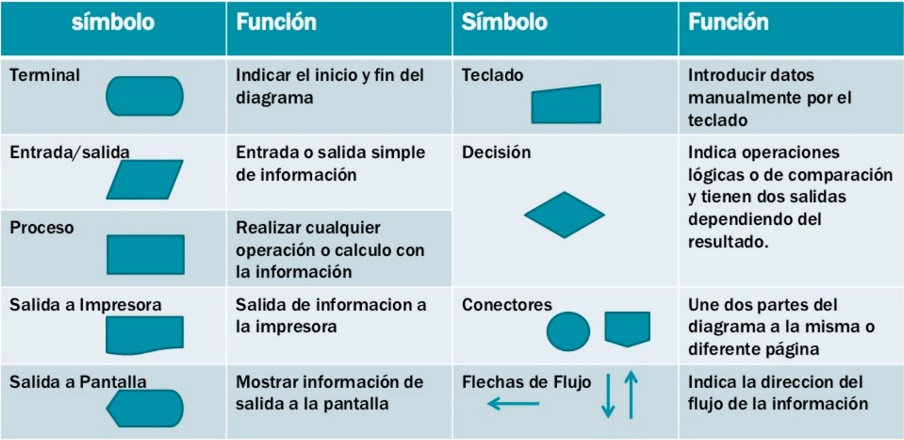
* **“Algoritmo computacional:** Es un algoritmo que puede ser ejecutado en una computadora. Ejemplo: Fórmula aplicada para un cálculo de la raíz cuadrada de un valor x.”**[1]**
* **“Algoritmo no computacional:** Es un algoritmo que no requiere de una computadora para ser ejecutado. Ejemplo: Instalación de un equipo de sonido.**”[1]**
* **“Algoritmo cualitativo:**Un algoritmo es cualitativo cuando en sus pasos o instrucciones no están involucrados cálculos numéricos. Ejemplos: Las instrucciones para desarrollar una actividad física, encontrar un tesoro.” **[1]**
* **“Algoritmo cuantitativo:** Un algoritmo es cuantitativo cuando en sus pasos o instrucciones involucran cálculos numéricos. Ejemplo: Solución de una ecuación de segundo grado.” **[1]**

### Representación de Algoritmos y Pseudocódigo

Según Tecnochiapa (2012) para representar un algoritmo se debe utilizar algún método que permita independizar dicho algoritmo del lenguaje de programación elegido. Ello permitir que un algoritmo puede ser codificado indistintamente en cualquier lenguaje. Para conseguir este objetivo se precisa que el algoritmo sea representado grafica o numéricamente de modo que los sucesivo acciones no dependen de la sintaxis de ningún lenguaje de programación, sino que la descripción puede servir fácilmente para su transformación en un programa, ese decir su codificación.[3]

#### -Diagramas de Flujo

A continuación se muestran una serie de símbolos útiles para llevar a cabo este tipo de representaciones.



**Imagen 2 “Diagramas de flujo” [4]**

#### -

#### Pseudocódigo

Según José M. (2020) “El Pseudocódigo es sin duda de las representaciones más utilizadas. Es una forma de expresar el algoritmo utilizando el lenguaje natural, comprensible para cualquier persona, pero añadiendo ciertas instrucciones típicas de los lenguajes de programación**.”[4]**

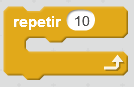
Según José M. (2020) “No existe una sintaxis estándar para el pseudocódigo, pero como hemos comentado, en el pseudocódigo se reflejan las instrucciones típicas de los lenguajes de programación, como las instrucciones **condicionales”**

**SI** condición **ENTONCES**  
    instrucciones/pasos a realizar si se cumple la condición  
**SI NO**  
    instrucciones/pasos a realizar si NO se cumple la condición  
**FIN SI**

****

**Imagen 3 “Armando Seudocódigo” [4]**

Y las instrucciones **repetitivas**:

**REPETIR** n veces  
    instrucciones/pasos a realizar  
**FIN REPETIR**

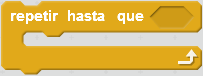
**Imagen 4 “Armando Seudocódigo 2” [4]**

**REPETIR HASTA** condición de salida  
    instrucciones/pasos a realizar hasta que se cumpla la condición de salida del bucle  
**FIN REPETIR**

Según José M. (2020) “En definitiva, el pseudocódigo se trata de un**falso lenguaje**, ya que apela a las normas de estructura de un lenguaje de programación aunque está pensado para que pueda **ser leído por un ser humano** y no interpretado por una máquina.” **[4]**

**Imagen 5 “Armando Seudocódigo 2” [4]**

Según José M. (2020) el pseudocódigo, en este sentido, está considerado como una descripción de un algoritmo que resulta independiente de otros lenguajes de programación. Para que una persona pueda leer e interpretar el código en cuestión, se excluyen diversos datos que no son clave para su entendimiento. Veamos un par de ejemplos orientados a crear algoritmos matemáticos. **[4]**



**Ejemplo: Realizar el pseudocódigo de un programa que permita calcular el área de un rectángulo. Se debe introducir la base y la altura para poder realizar el cálculo.**

Programa: área

Entorno: BASE, ALTURA, AREA son números enteros

Algoritmo:

escribir "Introduzca la base y la altura"

leer BASE, ALTURA

calcular AREA = BASE \* ALTURA

escribir "El área del rectángulo es "AREA"

Finprograma

**Ejemplo: Realizar el pseudocódigo que permita al usuario introducir por teclado dos notas, calculando la suma y el producto de las notas.**

Programa: Suma Producto

Entorno: NOTA1, NOTA2, SUMA, PRODUCTO son números enteros

Algoritmo:

escribir "Introduzca las notas"

leer NOTA1, NOTA2

calcular SUMA = NOTA1 + NOTA2

calcular PRODUCTO = NOTA1 \* NOTA2

escribir "La suma de las dos notas es: "SUMA"

escribir "El producto de las dos notas es: "PRODUCTO".

Finprograma

Ningún ordenador podría interpretar estas instrucciones. Para crear un programa a partir del algoritmo, una vez refinado el pseudocódigo, deberíamos reescribirlo en un leguaje de programación: C, C++, Java, Scratch... Ejemplos según José M. (2020)**[4]**

### Abstracción y tipo de dato abstracto

Según Sirus (2020) un Tipo de dato abstracto  (TDA) es un conjunto de datos u objetos al cual se le asocian operaciones. El TDA provee de una interfaz con la cual es posible realizar las operaciones permitidas, abstrayéndose de la manera en como estén implementadas dichas operaciones. Esto quiere decir que un mismo TDA puede ser implementado utilizando distintas estructuras de datos y proveer la misma funcionalidad.**[5]**

El paradigma de orientación a objetos permite el encapsulamiento de los datos y las operaciones mediante la definición de clases e interfaces, lo cual permite ocultar la manera en cómo ha sido implementado el TDA y solo permite el acceso a los datos a través de las operaciones provistas por la interfaz. (Sirus, 2020) **[5]**

#### -Tipo de Dato Abstracto

##### -Pila

Una pila es una colección ordenada de elementos en la que pueden insertarse y suprimirse por un extremo, llamado ***tope,***  nuevos elementos. Se puede representar una pila como en la figura No. 6 (“Representacion de pila”).

A diferencia del arrglo, la definicion de pila incorpor la insercion y supresion de elementos, de tal manera que esta es un objeto dinamico constantemente variable. En consecuencia, surge una pregunta ¿Cómo cambia la pila?. La definicion especifica que un extremo de la fila se designa como el tope de la misma. Pueden agregarse nuevos elementos en el tope de la pila (en cuyo caso éste se mueve hacia arriba para corresponder al nuevo elemento que ocupa la cima), o pueden quitarse los elementos que están en el tope (y entonces el tope se mueve hacia abajo para corresponder al nuevo elemento que se encuentra hacia arriba). Para responder la pregunta cuál camino es hacia arriba? hj tenemos que definir a 1 de los extremos de la pila como su tope; es decir, en qué extremo pueden colocar suprimirse elementos. En la figura 6 se observa que el hecho que la F esté físicamente más arriba que el resto de los elementos de la pila, implica que es el elemento tope actual. Sí se agregan nuevos elementos a la pila, estos deben colocarse encima de F, y si se suprimen algunos, F será el primero en ser borrado. Esto sí indica también por las líneas verticales, las cual es se extienden más allá de los elementos de la pila en dirección del tope de la misma.

|  |
| --- |
| F |
| A |
| B |
| G |

**Imagen 6 “Representación de Pila” [Elaboración propia]**

La representación de una pila como un tipo de dato abstracto es directa. Usamos ***eltype*** Para denotar el tipo del elemento de la pila y parametrizamos el tipo de pila con ***eltype.***

**abstract tydef** <<eltype>> STACK (eltype);

**abstract** empty(s)

STACK (eltype)s;

postcondition empty==(len(s)==0);

**abstract** eltype pop(s)

*STACK* (eltype) s;

precondition empty==FALSE;

**postcondition** pop == first(s’);

s==sub(s’, 1, len(s’)-1);

***abstract*** *push (s, elt)*

*STACK (eltype) s;*

*eltype elt;*

***postcondition*** *s==<elt>+s’*

Antes de programar la solución de un problema que usa una pila, debe decidirse como representar una pila mediante las estructuras de datos existentes en nuestro lenguaje de programación. Cómo se verá, hay varias maneras de representar una pila en lenguaje C. Considere C ahora la más simple:

Una pila es una colección ordenada de elementos, el lenguaje se incluye un tipo de datos que es una colección ordenada de elementos: el arreglo. Por ello, excepto por comenzar un programa con la declaración de una variable *pila* cómo arreglo. siempre que la solución de un problema requiere el uso de una pila. Sin embargo, una pila y un arreglo son dos cosas por completo diferentes. El número de elementos de un arreglo es fijo y se asigna mediante su declaración. Y en general, el usuario no puede cambiar dicho número. Por otra parte, una pila es lo fundamental un objeto dinámico cuyo tamaño cambia constantemente tanto se le agreguen o quiten elementos.

Sin embargo aunque una regla no puede ser una pila, puede usarse como tal. Es decir, puede declararse un arreglo lo bastante grande para admitir el tamaño máximo de la pila. Durante la ejecución del programa, la pila puede crecer y contraerse dentro de su espacio reservado. El fondo fijo de la pila es un extremo del arreglo mientras que su tope cambia en forma constante cuando se agregan o quitan elementos. Así, es necesario otro campo para que en cada paso de la ejecución del programa, registre la posición actual del elemento tope de la pila.

En consecuencia se puede declarar una pila el lenguaje C, com una estructura que contiene dos datos: un arreglo para guardar los elementos de la pila y un entero para indicar la posición del elemento tope actual dentro del arreglo; lo cual puede hacerse para una pila de enteros por medios de las declaraciones:

**#define** STACKSIZE 100

STRUCT stack {

**int** top;

**int** items[STACKSIZE];

};

Una vez hecho puede declararse una pila S,por medio de:

**Struct** stack s;

##### -Colas

La **cola** es una colección ordenada de elementos de la que se pueden borrar elementos en un extremo (llamado **frente** de la cola) o insertarlos en el otro (llamado **final** de la cola).

La figura 7 (“Colas y sus representaciones”) se ve en **a)** que contiene los elementos **A B y C,** A está en el frente de la cola y C en el final. En **b)** se borró un elemento de la cola. Como los elementos solo se pueden borrar desde el frente de la cola a eliminar A, B pasa a ser el nuevo frente. En **c),** cuando se insertaron los elementos D y E, fue necesario insertarlos desde el final de la cola.

Como D se insertó en la cola antes que E, será eliminada primero. El primer elemento insertado en una cola es el primero en ser eliminado. Por esta razón algunas veces las colas son denominadas listas ***fifo*** en la oposición a la pila que es una lista ***lifo***.

|  |  |  |
| --- | --- | --- |
| **A** | **B** | **C** |

**a)**

|  |  |
| --- | --- |
| **A** | **B** |

**b)**

|  |  |  |  |
| --- | --- | --- | --- |
| **A** | **B** | **C** | **D** |

**c)**

**Imagen 7 “Representación de Cola” [Elaboración propia]**

La representación de una cola como un tipo de datos abstracto es directa. *eltype* se usa para denotar el tipo de elementos de la cola y parametrizar el tipo de cola.

**abstract tydef** <<eltype>> QUEUE (eltype);

**abstract** empty(q)

QUEUE (eltype)s;

postcondition empty==(len(q)==0);

**abstract** eltype remove(q)

QUEUE (eltype) q;

precondition empty==FALSE;

**postcondition** remove == first(s’);

q==sub(q’, 1, len(s’)-1);

***abstract*** *insert (q, elt)*

QUEUE *(eltype) q;*

*eltype elt;*

***postcondition*** *q==q’+ <elt>;*

Una de las posibilidades para representar una cola el lenguaje C ex es usar un arreglo para guardar los elementos de la cola y usar dos variables para guardar las posiciones en el arreglo del último y el primer elemento de la cola dentro del arreglo. Es posible declarar una cola de enteros q mediante:

**#define** STACKSIZE 100

STRUCT queue {

**int** items [MAXQUEUE];

**int** front, rear;

}q;

Por supuesto que el uso de un arreglo para guardar una cola introduce la posibilidad de desborde si la cola rebasa el tamaño del arreglo punto si se hace a un lado por el momento la posibilidad de desborde y su sub desborde coma puede implantarse la operación insert (q,x) mediante las instrucciones:

Entonces la operación **insert** comprende una verificación de desborde qué ocurre cuando todo el arreglo está ocupado por elementos de la cola y se intenta insertar otro.

Tanto la pila como la cola son estructuras de datos cuyos elementos están ordenados con base en la secuencia en que se insertaron. La operación **front** recupera el último elemento insertado, en tanto que la operación toma el primer elemento que se insertó. Si hay un orden intrínseco entre los elementos, las operaciones de la pila o la cola lo ignoran, es aquí donde surge la cola de prioridad.

La cola de prioridad es una estructura de datos en la que el ordenamiento intrínseco de los elementos determinan los resultados de sus operaciones básicas, hay dos tipos de colas de prioridad: la cola de prioridad ascendente y la cola de prioridad descendente.

##### -Listas

Si las listas no sólo son importantes como un medio para implementar pilas y colas, sino como estructuras de datos por derecho propio. En una lista ligada, la forma de accesar un elemento y recorriendo la lista desde el principio punto la implantación con arreglo permite accesar el enésimo elemento de un grupo por medio de una sola operación, mientras que no implantación con lista se requiere de n-operaciones. Es necesario pasar a través de los primeros n-1 elementos antes de alcanzar el elemento enésimo de una lista, ya que no hay relación entre la localidad de memoria que ocupa un elemento de una lista y su posición en la misma

ahora la implantación con lista de colas de prioridad, considerando una lista ordenada que puede usarse para representar una cola de prioridad. Para una cola de prioridad ascendente sin planta la inserción mediante la operación place, la que mantiene en orden la lista coma y la eliminación del elemento mínimo mediante la operación pop, la que elimina el primer elemento de la lista la cola te prioridad descendente puede implantarse guardando la lista en orden descendente coma en lugar de ascendente, o usando otras medidas que ya se vieron anteriormente en la parte de colas de prioridad.

# Tablas Hash

## Contexto de uso de una tabla hash

Supongamos que en una aplicación se deben manipular un número muy elevado de pares (clave, valor). Estos pares son creados bajo demanda por la aplicación, se manipulan, y se destruyen. La manipulación consiste principalmente en buscar si la aplicación contiene ya un par (clave, valor) dado, y si no es así, se añade a la tabla. **[6]**

Por ejemplo, se dispone de una aplicación un teléfono móvil que es capaz de reconocer mediante la cámara los números de un pasaporte. Para cada pasaporte se almacena un mensaje de texto con un comentario. La aplicación debe realizar las siguientes operaciones: **[6]**

* Crear una tabla de pasaportes vacía. **[6]**
* Dado un número de pasaporte, buscar si está almacenado ya en la tabla. **[6]**
* Dado un número de pasaporte y un comentario, almacenar este par (clave, valor) en la tabla. Se asume que no hay información previa sobre este número de pasaporte. **[6]**

## Posibles implementaciones

Como el número de pares a manipular es indeterminado, se necesita una estructura de datos dinámica. Como primera solución podemos considerar una lista encadenada de pares. La operación de inserción es sencilla. Dado un número de pasaporte y una cadena, se crea un elemento nuevo en la tabla y se inserta. La operación de búsqueda necesita atravesar la lista hasta que, o se encuentra la clave dada (número de pasaporte), o se llega al final (la clave no está en la tabla). Cuando el número de pares es muy elevado, esta búsqueda es muy ineficiente, pues hay que procesar un gran número de elementos. **[6]**

Una segunda opción para tener una búsqueda muy rápida podría ser almacenar todos los datos en una tabla y utilizar el número de pasaporte como su índice. En este caso, ver si un número está en la tabla consiste simplemente en ver si tiene una cadena asociada. El insertar un nuevo par (pasaporte, mensaje) también sería muy rápido pues consistiría en guardar el mensaje dado en la posición correspondiente al número. Pero el problema con esta solución es que el número de pasaportes puede ser un entero muy largo y por tanto la tabla que se necesita definir está más allá de lo aceptable. **[6]**

La tabla hash ofrece un compromiso para esta situación. Los pares (clave, valor) se guardan en una tabla, pero con un tamaño menor del ideal. Para nuestro ejemplo, se utiliza una tabla, pero no tiene como tamaño el número máximo de pasaportes posibles, sino un número más pequeño. **[6]**

## Tablas hash

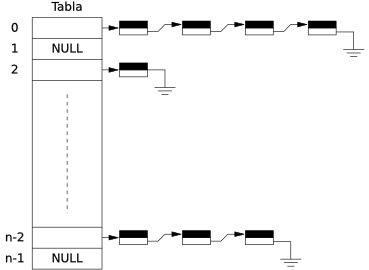
Las tablas hash son uno de los mecanismos más utilizados en el desarrollo de aplicaciones (haz una búsqueda en internet del término hash table y mira número de enlaces que se devuelven). Existen múltiples librerías en casi todos los lenguajes de programación que proporcionan implementaciones muy eficientes de estas tablas. **[6]**

* La implementación de una tabla hash está basada en los siguientes elementos:
* Una tabla de un tamaño razonable para almacenar los pares (clave, valor).
* Una función “hash” que recibe la clave y devuelve un índice para acceder a una posición de la tabla.
* Un procedimiento para tratar los casos en los que la función anterior devuelve el mismo índice para dos claves distintas. Esta situación se conoce con el nombre de **colisión**. **[6]**

Las posibles implementaciones de cada uno de estos tres elementos se traducen en una infinidad de formas de implementar una tabla hash. A continuación se detalla una solución concreta para el problema de las colisiones. **[6]**

## Gestión de colisiones mediante listas encadenadas

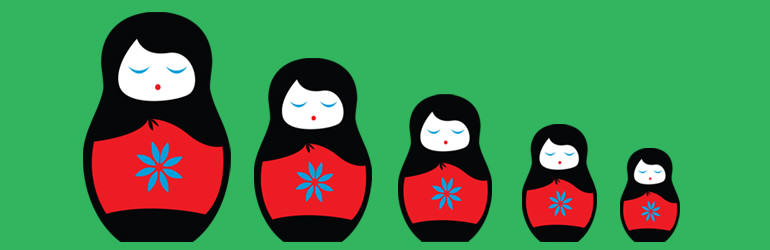
La solución propuesta para implementar la tabla hash combina la estructura de tabla con la de lista encadenada. Cada posición de la tabla no almacena un único elemento sino la cabeza de una lista encadenada que a su vez contiene todos aquellos elementos cuya función de hash ha devuelto idéntico resultado. Una posición de la tabla en la que no se haya insertado ningún elemento, contiene un puntero a NULL. La siguiente figura muestra una tabla de tamaño **n** y los elementos almacenados en las posiciones 0, 1, 2, n-2 y n-1. **[6]**



**Imagen 7 “Representación de Lista encadenada” [6]**

Fíjate que diferentes posiciones de la tabla pueden tener listas de diferente longitud, o incluso vacías. Dada una clave, el proceso de búsqueda consiste en calcular primero su índice mediante la función de hash, y a continuación buscar esa clave en la lista de colisiones. **[6]**

# Recursividad



Es una técnica utilizada en programación que nos permite que un bloque de instrucciones se ejecute un cierto número de veces (el que nosotros determinemos). A veces es algo complicado de entender, pero no os preocupéis. Cuando veamos los ejemplos estará clarísimo. En Java, como en otros muchos lenguajes, los métodos pueden llamarse a sí mismos. Gracias a esto, podemos utilizar a nuestro favor la recursividad en lugar de la iteración para resolver determinados tipos de problemas. **[7]:**

### Ejemplo sencillo

Vamos a ver un pequeño ejemplo que no hace absolutamente nada. Es un método cuyo único objetivo es llamarse a sí mismo: **[7]:**

void cuentaRegresiva () {

cuentaRegresiva();

}

 Si ejecutáis esto, os va a dar un error en la pila (mítico StackOverflow Error, Biblia de los programadores).

Como podemos ver, se ha llamado al método **cuentaRegresiva**porque vamos a mostrar por pantalla la cuenta atrás de un número que nosotros pasemos como parámetro a la función. Por ejemplo, para hacer la cuenta atrás de 10 sin recursividad, haríamos: **[7]:**

for( int i = 10; i >= 0; i--) {

System.out.println(i);

}

 Ahora, para hacerlo de manera recursiva, tendríamos que pasar como parámetro un número. Además, tras imprimir ese número, llamaremos a la misma función con el número actual restando uno: **[7]:**

void cuentaRegresiva(int numero) {

System.out.println(numero);

cuentaRegresiva(numero - 1);

}

Es lo que os he comentado arriba. Llamamos a la función con un 10. Imprimimos el 10 y llamamos a la función con un 9. Imprimimos el 9 y llamamos a la función con un 8. Así hasta el fin de los días. Digo hasta el fin de los días porque os va a saltar error si ejecutáis esto así directamente: **[7]:**

public class Recursividad {

static void cuentaRegresiva(int numero) {

System.out.println(numero);

cuentaRegresiva(numero - 1);

}

public static void main(String[] args) {

cuentaRegresiva(10);

}

}

 Problema: llamada infinita. Para ello, lo que tenemos que hacer es que cuando el número sea 0, deje de llamar a la función. Para eso, metemos una [estructura condicional](http://geekytheory.com/tutorial-3-java-estructuras-condicionales-y-excepciones/) de toda la vida. Gracias al condicional, dejará de ejecutarse a partir de 0: **[7]:**

void cuentaRegresiva(int numero) {

System.out.println(numero);

if(numero > 0) {

cuentaRegresiva(numero - 1);

}

}

## Factorial

Calcular el factorial de un número con recursividad es el típico ejemplo para explicar este método de programación. Recordad que el factorial de un número es multiplicar dicho número por todos sus anteriores hasta llegar a 1. Se representa con una exclamación. **[7]:**

Por ejemplo:

**5! = 54321 = 120** El recorrer los números hacia atrás ya lo tenemos hecho. Ahora lo que queda es multiplicar ese número por su anterior, y así sucesivamente:

int factorial(int n) {

if (n == 0) {

return 1;

} else {

return n \* factorial(n - 1);

}

}

# Backtracking

## Concepto

En su forma básica, la idea de backtracking se asemeja a un recorrido en profundidad dentro de un grafo dirigido. El grafo en cuestión suele ser un árbol, o por lo menos no contiene ciclos. Sea cual sea su estructura, existe sólo implícitamente. El objetivo del recorrido es encontrar soluciones para algún problema. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. El recorrido tiene éxito si, procediendo de esta forma, se puede definir por completo una solución. En este caso el algoritmo puede bien detenerse (si lo único que se necesita es una solución del problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas). Por otra parte, el recorrido no tiene éxito si en alguna etapa la solución parcial construida hasta el momento no se puede completar. En tal caso, el recorrido vuelve atrás exactamente igual que en un recorrido en profundidad, eliminando sobre la marcha los elementos que se hubieran añadido en cada fase. Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución. **[8]**

Algoritmo de Backtracking

proc Backtracking (↕X[1 . . . i ]: TSolución, ↑ok: B)

variables L: ListaComponentes

inicio

si EsSolución (X) entonces ok CIERTO

en otro caso

ok FALSO

L=Candidatos (X)

mientras ¬ok ^ ¬Vacía (L) hacer

X[i + 1] Cabeza (L); L Resto (L)

Backtracking (X, ok)

finmientras

finsi

fin

Podemos visualizar el funcionamiento de una técnica de backtracking como la exploración en profundidad de un grafo. **[8]**

Cada vértice del grafo es un posible estado de la solución del problema. Cada arco del grafo representa la transición entre dos estados de la solución (i.e., la toma de una decisión). **[8]**

Típicamente el tamaño de este grafo será inmenso, por lo que no existirá de manera explícita. En cada momento sólo tenemos en una estructura los nodos que van desde el estado inicial al estado actual. Si cada secuencia de decisiones distinta da lugar a un estado diferente, el grafo es un árbol (el árbol de estados). **[8]**

## Enfoque

Los problemas que deben satisfacer un determinado tipo de restricciones son problemas completos, donde el orden de los elementos de la solución no importa. Estos problemas consisten en un conjunto (o lista) de variables a la que a cada una se le debe asignar un valor sujeto a las restricciones del problema. La técnica va creando todas las posibles combinaciones de elementos para obtener una solución. Su principal virtud es que en la mayoría de las implementaciones se puede evitar combinaciones, estableciendo funciones de acotación (o poda) reduciendo el [tiempo](https://www.ecured.cu/index.php?title=Eficiencia_de_los_algoritmos&action=edit&redlink=1) de ejecución. **[8]**

Vuelta atrás está muy relacionado con la [Búsqueda combinatoria](https://www.ecured.cu/index.php?title=B%C3%BAsqueda_combinatoria&action=edit&redlink=1).

## Diseño e implementación

Esencialmente, la idea es encontrar la mejor combinación posible en un momento determinado, por eso, se dice que este tipo de algoritmo es una [Búsqueda en profundidad](https://www.ecured.cu/B%C3%BAsqueda_en_profundidad). Durante la búsqueda, si se encuentra una alternativa incorrecta, la búsqueda retrocede hasta el paso anterior y toma la siguiente alternativa. Cuando se han terminado las posibilidades, se vuelve a la elección anterior y se toma la siguiente opción (hijo [si nos referimos a un árbol]). Si no hay más alternativas la búsqueda falla. De esta manera, se crea un árbol implícito, en el que cada nodo es un estado de la solución (solución parcial en el caso de nodos interiores o solución total en el caso de los nodos hoja). **[8]**

Normalmente, se suele implementar este tipo de algoritmos como un procedimiento [recursivo](https://www.ecured.cu/Recursividad). Así, en cada llamada al procedimiento se toma una variable y se le asignan todos los valores posibles, llamando a su vez al procedimiento para cada uno de los nuevos estados. La diferencia con la [Búsqueda en profundidad](https://www.ecured.cu/B%C3%BAsqueda_en_profundidad) es que se suelen diseñar funciones de cota, de forma que no se generen algunos estados si no van a conducir a ninguna solución, o a una solución peor de la que ya se tiene. De esta forma se ahorra espacio en memoria y tiempo de ejecución. **[8]**

## Heurísticas

Algunas heurísticas son comúnmente usadas para acelerar el proceso. Como las variables se pueden procesar en cualquier orden, generalmente es más eficiente intentar ser lo más restrictivo posible con las primeras (esto es, las primeras con menores valores posibles). Este proceso poda el [árbol de búsqueda](https://www.ecured.cu/index.php?title=%C3%81rbol_(estructura_de_datos)&action=edit&redlink=1) antes de que se tome la decisión y se llame a la subrutina recursiva. **[8]**

Cuando se elige qué valor se va a asignar, muchas implementaciones hacen un examen hacia delante (FC, Forward Checking), para ver qué valor restringirá el menor número posible de valores, de forma que se anticipa en a) preservar una posible solución y b) hace que la solución encontrada no tenga restricciones destacadas. **[8]**

Algunas implementaciones muy sofisticadas usan una función de cotas, que examina si es posible encontrar una solución a partir de una solución parcial. Además, se comprueba si la solución parcial que falla puede incrementar significativamente la eficiencia del algoritmo. Por el uso de estas funciones de cota, se debe ser muy minucioso en su implementación de forma que sean poco costosas computacionalmente hablando, ya que lo más normal es que se ejecuten en para cada nodo o paso del algoritmo. Cabe destacar, que las cotas eficaces se crean de forma parecida a las funciones [Heurísticas](https://www.ecured.cu/Heur%C3%ADstica), esto es, relajando las restricciones para conseguir mayor eficiencia. **[8]**

Con el objetivo de mantener la solución actual con coste mínimo, los algoritmos vuelta atrás mantienen el coste de la mejor solución en una variable que va variando con cada nueva mejor solución encontrada. Así, si una solución es peor que la que se acaba de encontrar, el algoritmo no actualizará la solución. De esta forma, devolverá siempre la mejor solución que haya encontrado. **[8]**

## Ejemplos de aplicación de backtracking

SATISFABILITY

Inicialmente A contiene la expresión booleana que constituye el problema.

Elegir subproblema de A, p.ejemplo : (x+y+z)(x'+y)(y'+z)(z'+x)(x'+y'+z').

Elegir una cláusula con mínimo número de literales.

Elegir una variable x, y, z,... dentro de la cláusula y crear 2 subproblemas reemplazando x=V y x=F.

En el caso x=V

Omitir las cláusulas donde aparece x.

Omitir x' en las cláusulas que aparece x'.

En el caso x=F

Omitir las cláusulas donde aparece x'.

Omitir x en las cláusulas que aparece x.

Test

Si no quedan cláusulas. STOP. (solución encontrada).

Si hay una cláusula vacía. DROP.

En otro caso añadir a A

Nota: Observemos que si encontramos a A vacío entonces la expresión booleana no puede ser satisfecha.

### HAMILTON CYCLE (VIAJANTE DE COMERCIO)

En este caso los subproblemas S son caminos que parten de a y llegan a b a través de un sucesión de nodos T. (b es el mismo a lo largo de todo el algoritmo).

Inicialmente A contiene solamente el camino (a, vacío, b).

Elegimos un subproblema S cualquiera de A (y lo borramos de A) y añadimos ramas (c, a) del grafo (las c's son las adyacentes de a) . Estos caminos extendidos son los hijos. Ahora cada c juega el rol de a.

Examinamos c/ hijo:

Test:

1)Si G-T forma un camino hamiltoniano STOP (solución hallada)

2)Si G-T tiene un nodo de grado uno (excepto a y b) o si G-T-{a, b} es disconexo entonces DROP este subproblema .

3) Si 1) y 2) fallan add subproblema en A.

### EXACT COVER

Dado un conjunto finito U y una familia se subconjuntos {Tj} de U definimos una matriz A donde cada fila se corresponde con un elemento ui de U y cada columna de A con un subconjunto Tj . Ponemos aij=1 si ui∈U pertenece a Tj y aij=0 en caso contrario. Interpretamos que xj=1 significa que elegimos Tj y 0 en caso contrario. **[8]**

Se trata de averiguar si es factible Ax=1 donde A y x son binarias y las componentes de 1 son unos. **[8]**

S0= un vector de ceros (raíz del árbol)

Cada nodo S del árbol es una sucesión x cuyas primeras k componentes le han sido asignados un 1 o un 0 y el resto de componentes son ceros. Reemplazamos S por 2 subproblemas Si (i=1,2) poniendo xk+1 =1 y xk+1=0 respectivamente. **[8]**

Test

if Ax=1 STOP

if Ax>1 DROP Si

if Ax<1 add Si to A

## Ejemplos de problemas comunes resueltos usando Vuelta Atrás

### Problema de las N Reinas

Disponemos de un tablero de ajedrez de tamaño NxN, y se trata de colocar en él N reinas de manera que no se amenacen según las normas del ajedrez.

proc NReinas (↕[1 . . . i ]: TSolución, ↓N: N, ↑ok: B)

variables j : N

inicio

si i=N entonces ok=CIERTO

en otro caso

ok=FALSO

j=1

mientras ¬ok ^ (j≤N) hacer

si EsFactible (R, j) entonces

R[i + 1]= j

NReinas (R, N, ok)

finsi

j=j+1

finmientras

finsi

fin

func EsFactible (↓R[1 . . . i ]: TSolución, ↓j : N): B

variables factible: B

inicio

factible=CIERTO

k=1

mientras factible ^ (k≤i) hacer

si (j=R[k])\/(i+1−k= |j−R[k]|) entonces

factible=FALSO

finsi

k=k+1

finmientras

devolver factible

fin

### Problema de la mochila 0,1

Dados n elementos e1,e2,...,en con pesos p1,p2,...,pn y beneficios b1,b2,...,bn, y dada una mochila capaz de albergar hasta un máximo de peso M (capacidad de la mochila), queremos encontrar cuáles de los n elementos hemos de introducir en la mochila de forma que la suma de los beneficios de los elementos escogidos sea máxima, sujeto a la restricción de que tales elementos no pueden superar la capacidad de la mochila.

### Problema del laberinto

Se tiene una matriz bidimensional de nxn casillas para representar un laberinto cuadrado. Cada casilla está marcada como visitada o no visitada. Se debe ir desde la casilla (1,1) a la (n, n) haciendo movimientos horizontales y verticales.

## Backtracking para la enumeración

El problema de la enumeración consiste en encontrar todas las soluciones del problema, es por ello que tendremos que recorrer el árbol de estados al completo.

Algoritmo de Backtracking para la enumeración:

proc Bactracking Enum(↕X[1 . . . i ]: TSolución, ↑num: N)

variables L: ListaComponentes

inicio

si EsSolución (X) entonces num num+1

EscribeSolución (X)

en otro caso

L Candidatos (X)

mientras ¬Vacía (L) hacer

X[i + 1] Cabeza (L); L Resto (L)

BacktrackingEnum (X, num)

finmientras

finsi

fin

## Aplicaciones

Vuelta atrás se usa en la implementación de los [Lenguajes de programación](https://www.ecured.cu/Lenguajes_de_programaci%C3%B3n) tales como [Lenguaje de programación Planner](https://www.ecured.cu/Planner) y [Prolog](https://www.ecured.cu/Prolog). Además, se usa en los análisis sintácticos de los compiladores. Su uso en [Inteligencia artificial](https://www.ecured.cu/Inteligencia_artificial) ha sido muy importante, dando lugar a nuevos tipos de búsquedas como el [A estrella](https://www.ecured.cu/index.php?title=A*&action=edit&redlink=1). **[8]**

# Conclusión.

Es de esta forma que podemos concluir que todos los que nos podemos encontrar en los diversos programas que realizamos, tienen diversas formas de resolverse sin perder de vista que cada una de estas formas tienen diversas desventajas y ventajas en su haber, parte de esto es el gran trabajo del programador descubrir cuáles son y de qué manera utilizarlas en su beneficio o como herramienta, para hacer de su trabajo más fácil sencillo y eficiente, pues el código por sí mismo es solo código, es la lógica del programador la que logra que todo forme parte de un ciclo en el cual se realizan diversas partes y tareas específicas.

# Referencias Bibliográficas:

**[1]:**Algorítmica y Programación. (2017). Algoritmo. Retrieved November 3, 2020, from Google.com website: https://sites.google.com/site/portafoliocarlosmacallums/unidad-i/algoritmo

**[2]:**Delgado Arturo. (2017). Características de un algoritmo. Retrieved November 3, 2020, from Google.com website: https://sites.google.com/site/portafoliocarlosmacallums/unidad-i/caracteristicasdeunalgoritmo

**[3]:**Guillermo, J. (2018, June 23). ¿Qué es la complejidad algorítmica y con qué se come? Retrieved November 4, 2020, from Medium website: https://medium.com/@joseguillermo\_/qu%C3%A9-es-la-complejidad-algor%C3%ADtmica-y-con-qu%C3%A9-se-come-2638e7fd9e8c

**[4]:**José, M. (2020). Representación de un algoritmo | Algoritmos y pseudocódigo. Retrieved November 3, 2020, from Intef.es website: http://formacion.intef.es/pluginfile.php/87713/mod\_imscp/content/13/representacin\_de\_un\_algoritmo.html#:~:text=El%20Pseudoc%C3%B3digo%20es%20sin%20duda,de%20los%20lenguajes%20de%20programaci%C3%B3n.

**[5]:**Programación II. (2015). Tipo de Dato Abstracto. Retrieved November 3, 2020, from Google.com website: https://sites.google.com/site/programacioniiuno/temario/unidad-2---tipo-abstracto-de-dato/tipo-de-dato-abstracto

**[6]:**Funciones Hash (teoría y ejemplo)*: “Programación en C: Metodología, algoritmos y estructura de datos”*

**[7]:** Geeky Theory, & Geeky Theory. (2020). ¿Qué es la recursividad? Retrieved November 24, 2020, from Geeky Theory website: https://geekytheory.com/que-es-la-recursividad

‌**[8]:** Vuelta atrás (backtracking) - EcuRed. (2020). Retrieved November 24, 2020, from Ecured.cu website: https://www.ecured.cu/Vuelta\_atr%C3%A1s\_(backtracking)

‌