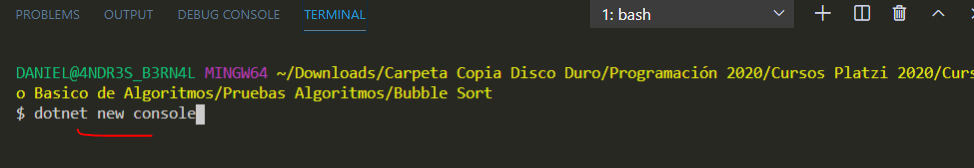
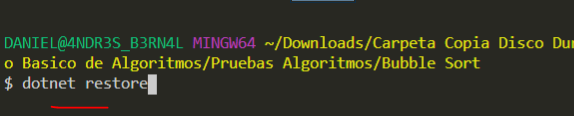
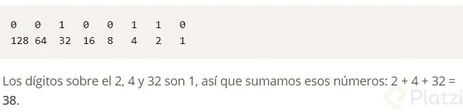
**Crear un nuevo programa de Consola.**



* Valida que el proyecto no tenga errores



**Números Binarios.**



**Lenguajes de programación**

Los lenguajes de programación son un *set* de instrucciones que usamos para enseñarle a las computadoras los pasos a seguir para resolver un problema.

Existen diversos tipos de lenguajes de programación y los podemos reducir en 2 grandes categorías: lenguajes de **bajo nivel** y de **alto nivel**. Esto según su complejidad y nivel de preocupación o acceso al hardware.

**Lenguajes de bajo nivel**: Son los que te permiten hacer modificaciones en el hardware. Entre más bajo es el nivel, más nos debemos preocupar los programadores por el hardware. Por ejemplo, tenemos el **lenguaje máquina** (solo código binario) como el de más alto nivel, siguiendo por **ensamblador** (con instrucciones más amigables) y otros lenguajes como C o C++.

**Lenguajes de alto nivel**: Son frecuentemente lenguajes más modernos en los que uno como programador se enfoca en el software, no tanto por el hardware. Por ejemplo, **Ruby**, **JavaScript** o **Perl**.

# Estructuras de datos

Las estructuras de datos son una forma eficiente de almacenar y organizar la información de nuestro programa. Existen diferentes estructuras que vamos a estudiar durante el curso, es muy importante trabajar con una buena estructura que nos ayude a resolver nuestros problemas de la mejor manera posible.

Podemos clasificar las estructuras en dos grandes grupos:

* **Lineales**: La información se guarda de forma secuencial y podemos personalizar el orden en que se guardan. Por ejemplo, de acuerdo a las fecha, relevancia de la información, entre otras.
* **No lineales**: No toda la información está al mismo nivel o almacenada con un orden especifico. Por ejemplo, en la estructura de árbol tenemos un tronco principal con diferentes ramificaciones que surgen a partir de este o, también, la estructura de grafos donde tenemos puntos de información dispersos pero interconectados entre sí.

# ¿Qué es un algoritmo?

Un **algoritmo** es un conjunto de **instrucciones** o pasos que **resuelven** un problema dado paso a paso y sin generar **ambigüedades**.

Las instrucciones de la computadora siempre deben dar los mismos resultados cuando tienen los mismos datos de entrada. Por ejemplo, no importa cuantas veces lo preguntes, la operación 5+5 siempre es igual a 10.

Además, podemos programar diferentes instrucciones que terminan con el mismo resultado. Nuestro trabajo también es descubrir cuál es la mejor manera de solucionar estos problemas.

Vamos a crear un diagrama de flujo para realizar un algoritmo que debe resolver el problema de cómo encontrar la ruta más corta para ir de un lugar a otro.

**Metodología para la construcción de un algoritmo**

Los pasos recomendados para desarrollar un algoritmo son los siguientes:

1. Definición del problema
2. Análisis del problema
3. Diseño del algoritmo
4. Verificación o pruebas

Para la definición del problema debemos obtener los siguientes elementos:

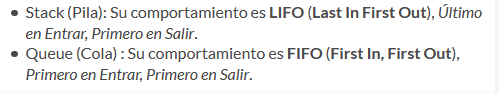
* **Entradas**: ¿Qué se necesita para realizar los pasos?
* **Salidas**: ¿Qué se obtiene al final del algoritmo?
* **Tipos de datos involucrados**: Textos, números, listas, etc.

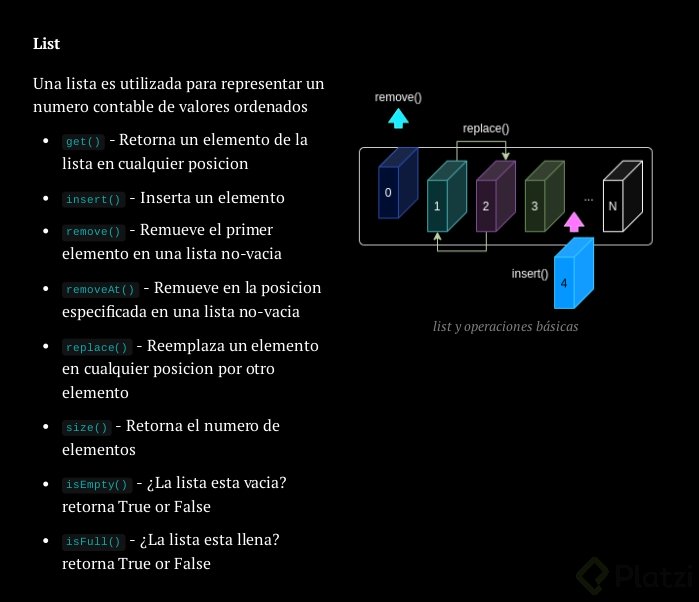
# Variables y tipos de datos

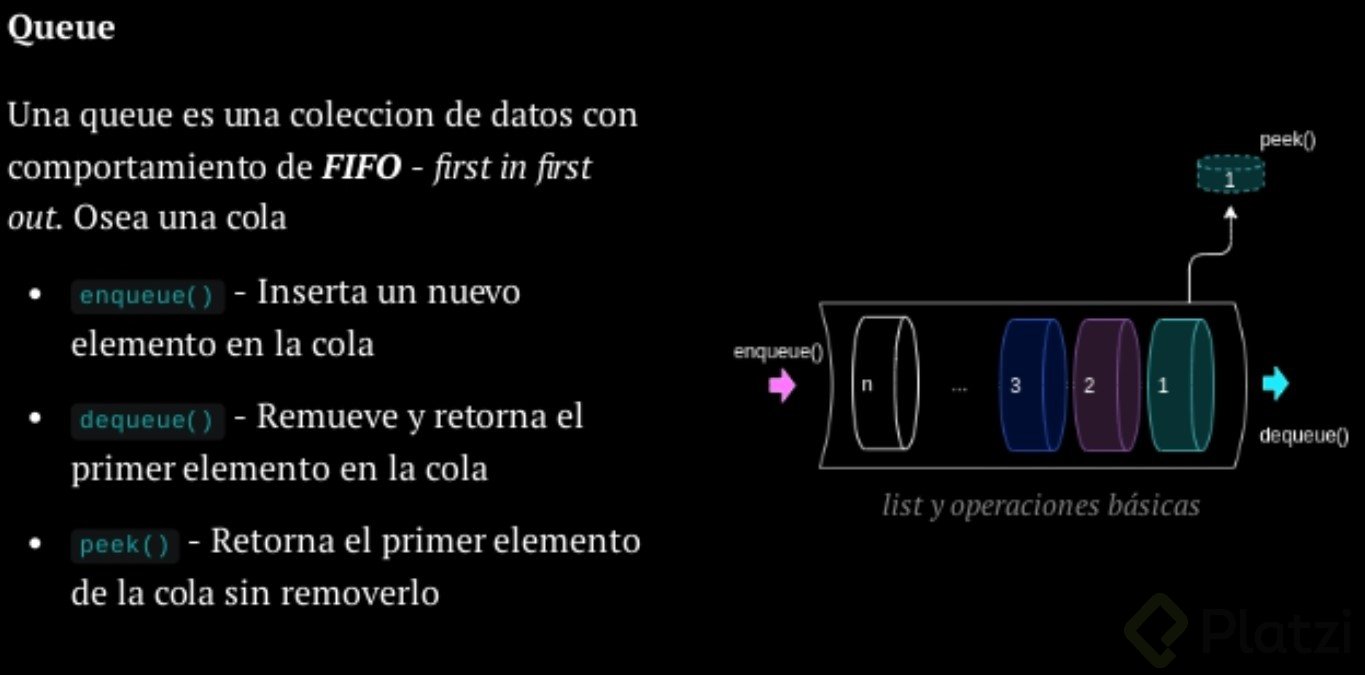
# 

# Abstract Data Types básicos: Lists, Stacks, Queues

Un **tipo de dato abstracto** (ADT) representa un set particular de comportamientos, aunque no define su implementación.  
Mientras que una **estructura de datos** es más concreta. Típicamente es la implementación de un ADT.  
Los ADT más básicos serían: Lists, Stacks, Queues, Diccionarios, Arrays, Linked Lists, Trees



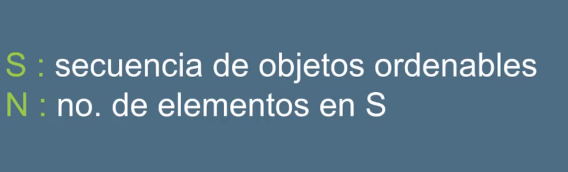




***Algoritmos de Ordenamiento.***

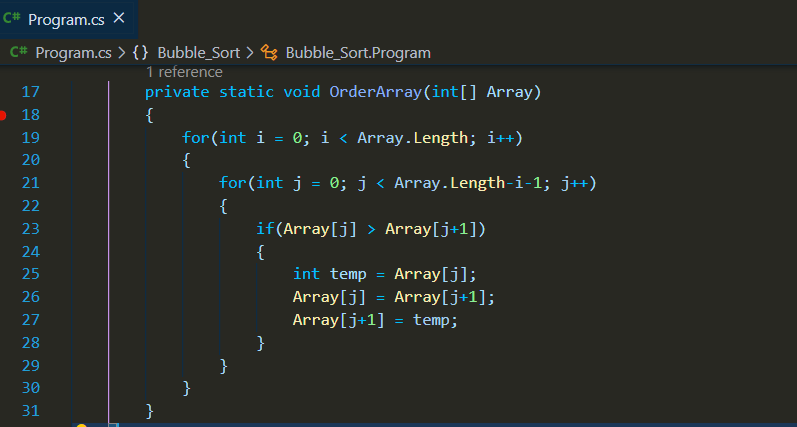
* **Bubble Sort.**

Para realizar este algoritmo necesitaremos un array con las posiciones a ordenar **S** y la cantidad de posiciones que va tener este array **N.**



**Bubble Sort** es el algoritmo de ordenamiento más rápido de implementar.

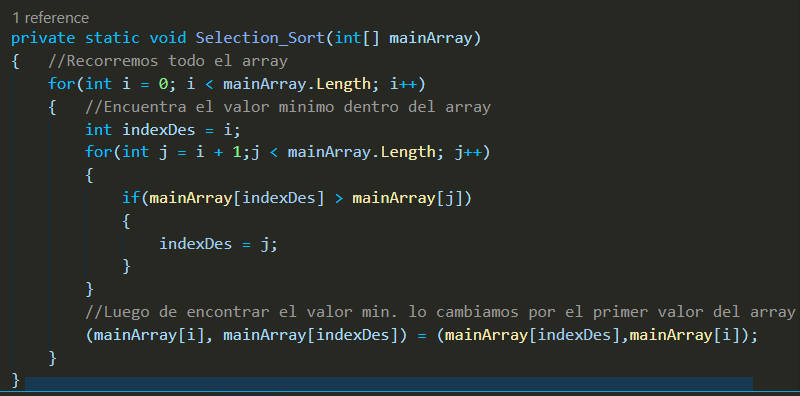
Se ordena por pares y se va repitiendo de 1 a 1 hasta que queda totalmente ordenado, es muy iterativo y muy poco eficiente (por la cantidad de iteraciones), pero al entenderlo estarás listo para poder pasar a algoritmos un poco más complejos.

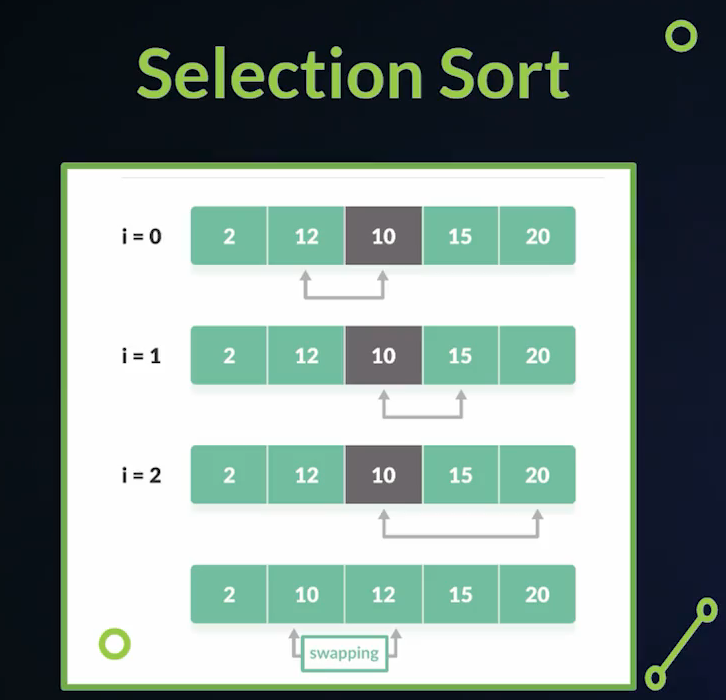


# Selection Sort.

Selection sort siempre va mandar el número más pequeño al principio.

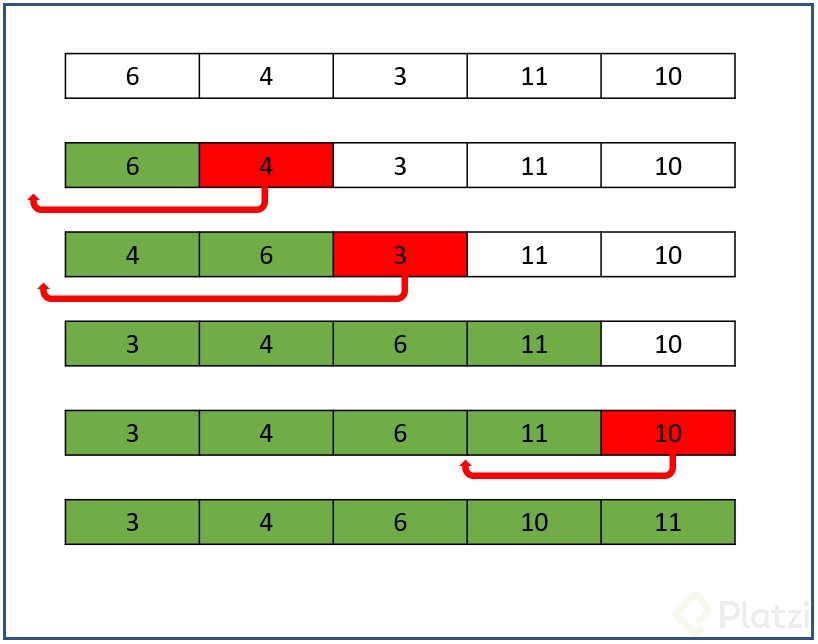
Selection sort en comparación con BubbleSort ambos son los mas sencillos de implementar, ambos con un sentido exponencial dependiendo de la cantidad de datos a ordenar pero, SelectionSort no hace recorridos innecesarios como los que llega a hacer BubbleSort por lo que en tiempo de ejecución resulta mas eficiente.



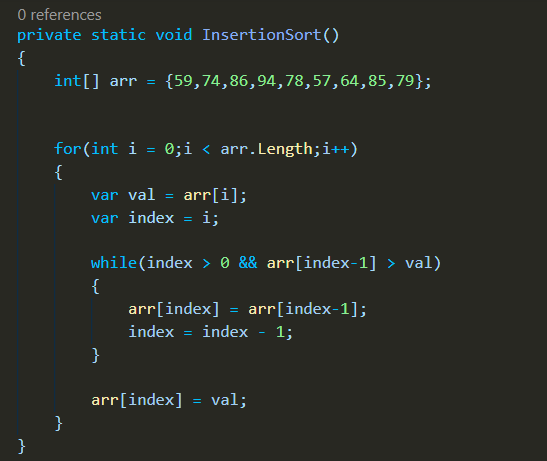


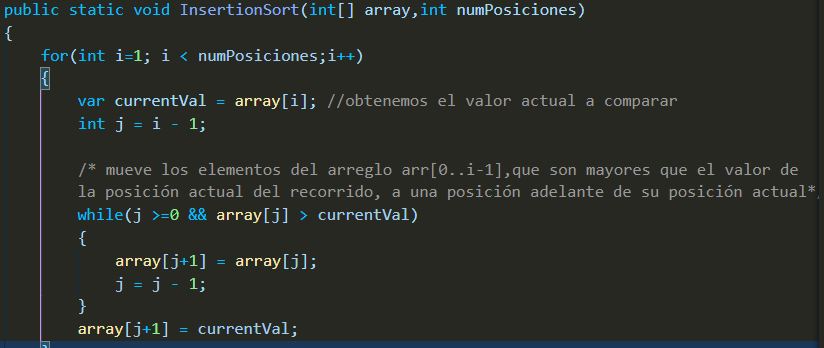
# Insertion sort .

# Insertion Sort es un algoritmo simple que ordena individualmente cada valor, como lo harías al ordenar un set de cartas del juego UNO en tu mano.



Como puedes observar en la imagen, recorremos nuestro set de datos posición por posición y comparamos el número con los valores anteriores, en caso de ser menor, lo colocamos en su posición indicada para ordenar de menor a mayor.

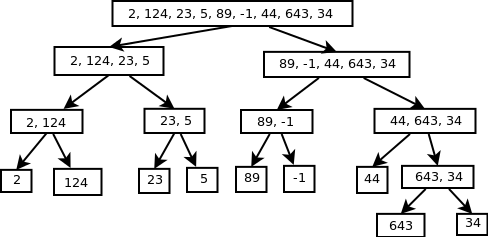




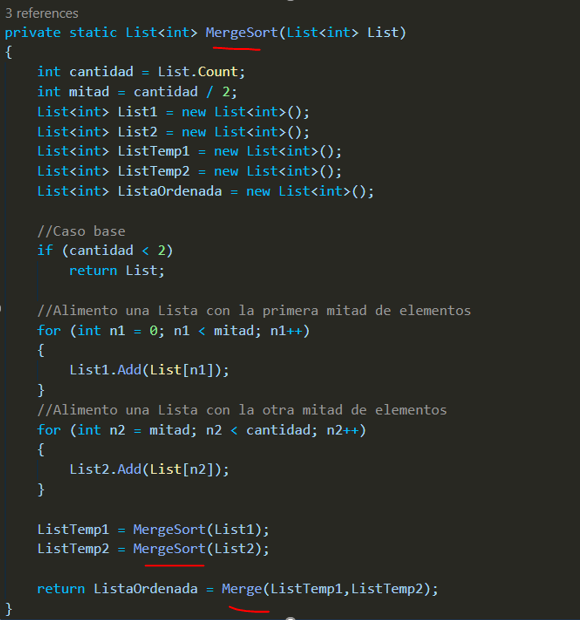
# MergeSort.

El algoritmo MergeSort es un algoritmo del tipo Divide and Conquer, en este dividimos el array de entrada en dos mitades, se invoca la función de manera recursiva dividiendo hasta que nuestra lista quede de un solo elemento, las cuales se van retornando secuencialmente hasta en que en el final del método se unan con un Merge().

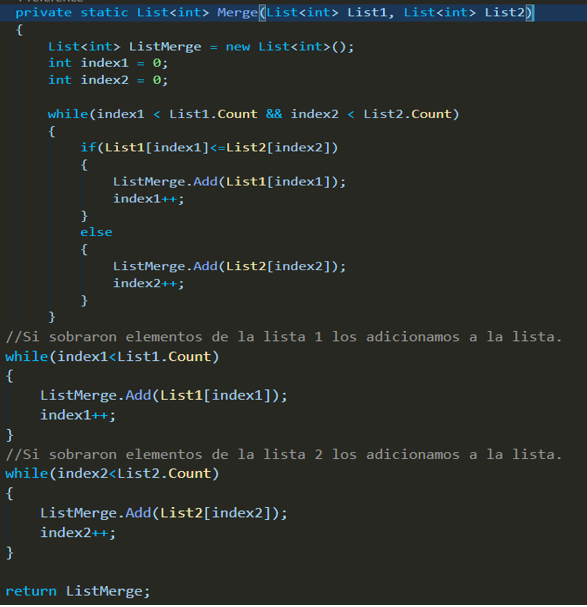
En cada una de esas mitades y luego une las dos mitades ordenadas, la función merge() se encargará de unir las dos mitades.



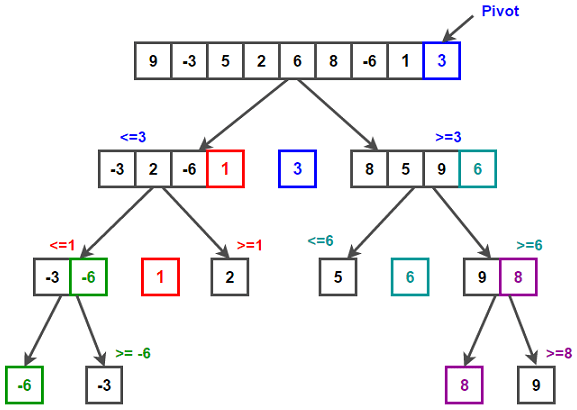
* *Metodo MergeSort()*



* *Metodo Merge()*



***QuickSort.***



Para esta clase aprenderemos a reordenar una serie de números mediante quick sort, que no es más que un algoritmo de divide y vencerás, nuestro algoritmo se dividirá en 3:

1. Necesitamos **dividir** nuestra lista.
2. Obtener un **punto pivotal**
3. **Recursivamente ordenar** cada **mitad** de mí array

# En primer lugar le enviaremos a nuestra función QuickSort() el array a ordenar y la cantidad de posiciones que tiene.

# Dentro de QuickSort() necesito saber el index de mi pivot a través de mi método Particion() y luego sabiendo esto de manera recursiva invoco nuevamente el metodo QuickSort() organizando mi parte derecha e izquierda de mi array.

# 

# 

# *Recursividad.*

La recursividad es la capacidad de una función de llamarse a si misma.

Las funciones recursivas tienen las siguientes caracteristicas.

* Se **llaman a si mismas**.
* Tienen de **argumento** un valor que cambio por cada iteración.
* **Regresan** algún valor definido en cada iteración.
* Tienen una **condicional** que define el fin del ciclo.
* Manejan un **stack** que es el órden de las ejecuciones de las iteraciones de la función, empezando por la última a llamar.

En este ejemplo usaremos recursividad para invertir una cadena de texto.

# 

# Qué es la programación dinámica (divide y vencerás v2.0)

Antes de implementar nuestro algoritmo quicksort tenemos que ver otro concepto súper importante, la **programación dinámica**.

La **programación dinámica** es el método para resolver **problemas complejos**, rompiéndolos en un conjunto de problemas simples

La diferencia con el divide y vencerás que aprendimos anteriormente es que cada uno de los **problemas que solucionamos** se van a ir **guardando** automáticamente y se va a ir **acomodando automáticamente**.

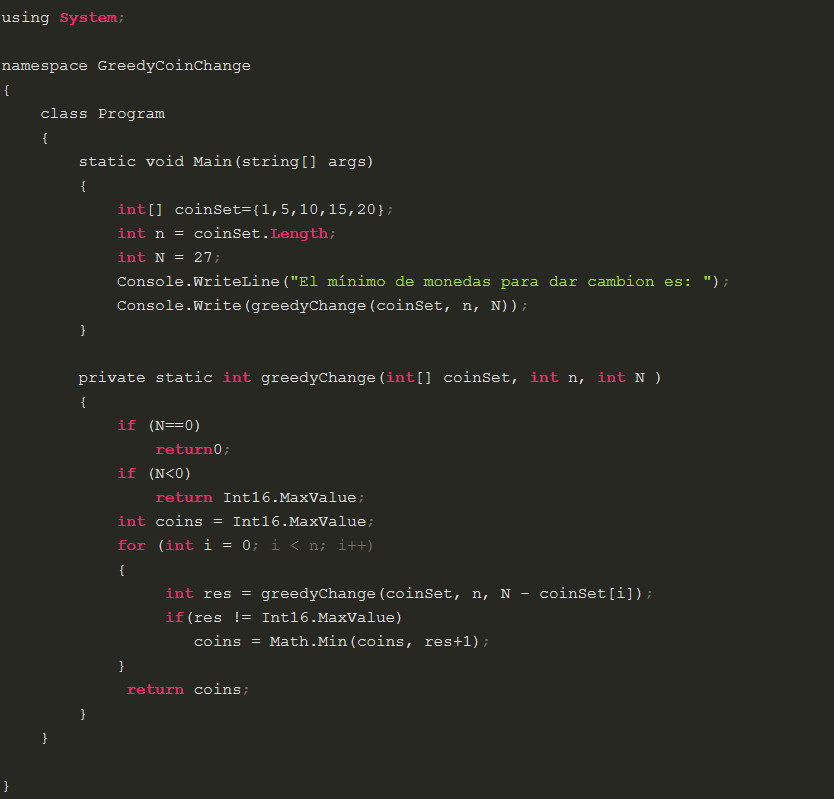
***Qué son los Greedy Algorithm***

Los *Greedy Algorithms* son algoritmos que van a buscar la **solución más óptima** para resolver un problema en **cada una de sus etapas**; vamos a tener varios subprocesos antes de llegar al resultado final.

Este tipo de algoritmos son muy “**codiciosos**” porque van a encontrar la mejor solución de cada paso pero, en conjunto, puede que no sean la mejor solución al problema completo.

Estos son algunos ejemplos de problemas que se resolverían con *Greedy Algorithms*:

1. Una **máquina que da cambio** porque va a dividir el dinero de una manera eficiente y correcta.
2. Un **juego** donde debemos calcular el mejor movimiento y podemos usar algoritmos de inteligencia artificial.
3. Una **aplicación de mapas** porque va a buscar la ruta con el menor consumo o la menor distancia.



# Grafos y sus aplicaciones.

Los graphos son mapas de **set de nodos** que tienen múltiples **relaciones** entre sí mediante **enlaces** (edges).

Los graphos no tienen estructura de cascada como los árboles binarios, pero comparten la lógica relacional.

Los graphos son definidos por **G=(V,E)**, donde:

* **G** = Representación del grapho
* **V** = Set de nodos (**vértices**)
* **E** = Enlaces (**edges**)

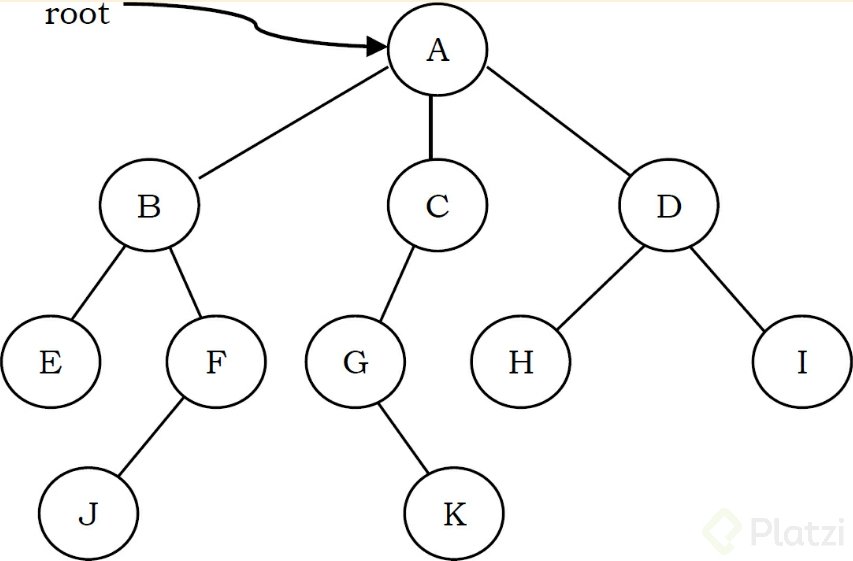
Los graphos son el pilar o los fundamentos que se ocupan en bases de datos modernas como es el caso de la base de datos de:

* Las relaciones de amistad de Facebook
* Las recomendaciones de Amazon
* Los seguidores de Twitter

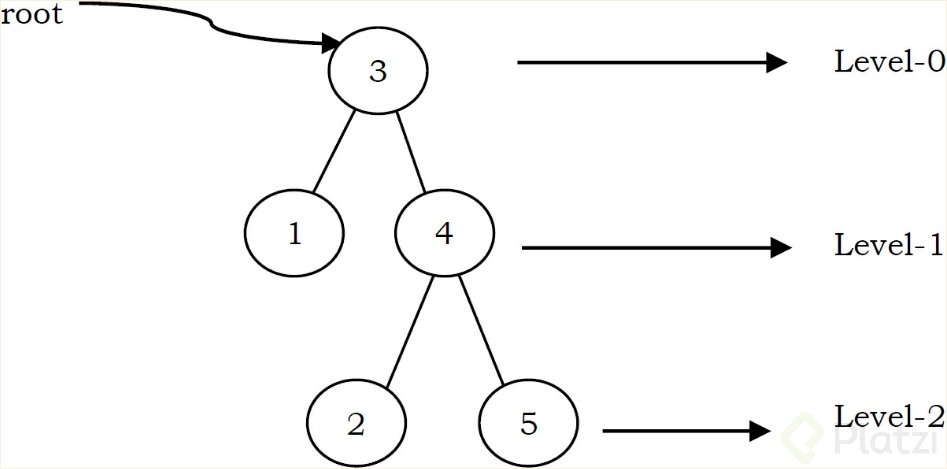
**Árboles.**

Un árbol es una estructura de datos similar a una lista pero en lugar de que cada nodo apunte al siguiente de forma lineal, cada nodo apunta a un número de nodos.

Un árbol es un ejemplo de una estructura de datos no lineal y es la forma de representar la naturaleza jerárquica de una estructura en forma gráfica.  
Los árboles también funcionan como un tipo de datos abstracto que cuenta con su definición y descripción de propiedades y operaciones, y se diferencia principalmente de las listas, queues (colas) y pilas porque a diferencia de estas no se utiliza para ordenar elementos de forma lineal.



* La raíz (root) de un árbol es el nodo sin nodos padres, un árbol puede tener máximo una raíz, como el nodo A en el ejemplo anterior.
* Un enlace o “edge” se refiere al enlace entre un nodo padre a un nodo hijo.
* Un nodo sin nodos hijos se conoce nodo “hoja” como los nodos E, J, K, H, I.
* Los nodos hijos del mismo padre se conocen como “hermanos” (Siblings) por ejemplo, los nodos B, C, D son nodos hermanos hijos de A.
* Por otro lado, los nodos anteriores a un nodo dado en la ruta hacia la raíz se conocen como ancestros, como G, C, y A para el nodo K en nuestra imágen de ejemplo.
* El set de todos los nodos a cualquier profundidad dada se conoce como “nivel del árbol” en nuestro ejemplo B, C, D son del mismo nivel (nivel 1) siendo la raíz el nivel 0.



* La profundidad de un nodo es la longitud del camino desde la raíz hasta el nodo, por ejemplo el nodo G tiene una profundidad de 2, A(1) - C(2) - G.
* La altura de un nodo es el largo del camino del ndo al nodo más profundo por ejemplo en nuestra imágen la altura de B es 2, B - F(1) - J(2).
* La altura del árbol es la altura máxima entre todos los nodos en un árbol, para un árbol la altura y la profundidad retornará el mismo valor, pero para nodos individuales podemos tener diferentes resultados, en el caso de nuestro árbol de ejemplo la altura y profundidad son de 3.