



**UNIVERSIDAD JUÁREZ AUTONOMA DE TABASCO**  
**DIVISIÓN ACADÉMICA DE CIENCIAS BÁSICAS**



PROGRAMA EDUCATIVO

**LIC. CIENCIAS COMPUTACIONALES**

PROFESOR

**DR. ABDIEL EMILIO CACERES GONZALEZ**

EXPERIENCIA EDUCATIVA

**ANALISIS DE ALGORITMOS**

TRABAJO

**TAREA 1**

ESTUDIANTE

**RODRIGUEZ TORRES KEVIN NICK**

**CARDENAS, TAB.**

**10 DE MARZO DEL 2021**

## EJERCICIOS

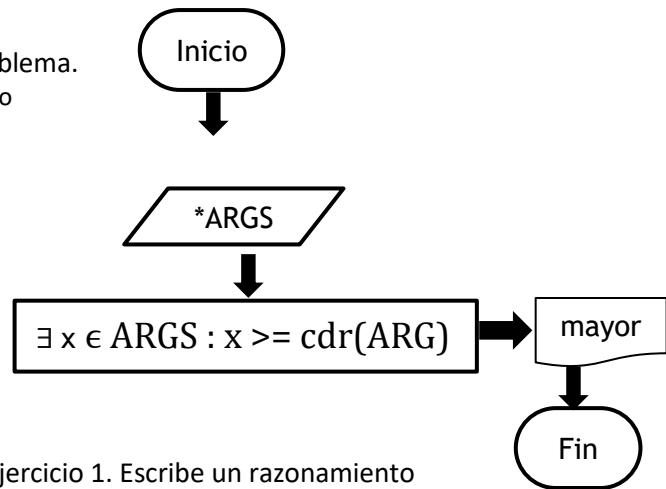
1. Escribe un algoritmo que resuelva el siguiente problema.

PROBLEMA: Determinar el número que es mayor dentro de una colección de 4 números.

ENTRADA: Cuatro números enteros.

SALIDA: El número mayor de los cuatro de entrada.

Entrada	Salida
3 6 9 1	9
1 3 3 3	3
4 5 -6 0	5



2. Considera nuevamente el algoritmo hecho en el ejercicio 1. Escribe un razonamiento sobre las 5 características importantes de todo algoritmo [pagina 2]. ¿Tu algoritmo tiene estas características? ¿Por qué?

**Finitud:** ya que el número de instrucciones es una cantidad mínima.

**Especificidad:** es un algoritmo claro ya que se expresa en un lenguaje formal.

**Entrada:** consta de una entrada de 4 números enteros

**Salida:** devuelve un numero entero.

**Efectividad:** este algoritmo esta pensado para ejecutar el menor número de pasos posibles.

-----Ejercicio 3 -----

### Algoritmos recursivos:

```

int fibonacci(int n){
  if(n==1 || n==2) {
    return 1;
  }
  else{
    return fibonacci(n-1)+fibonacci(n-2);
  }
}
  
```

### Algoritmos divide y vencerás:

```

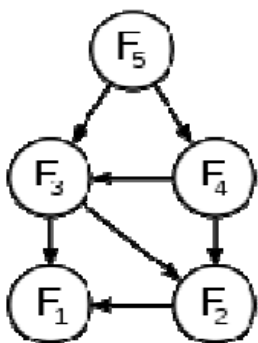
function raíz(n){
  numIter++;
  let r = 0;
  while (r<n){
    numIter++;
    r = r + 1;
    if (r**2 <= n && n < (r+1)**2){
      break;
    }
  }
  return r;}
  
```

## Algoritmos de programación dinámica:

### Sucesión de Fibonacci

$$fib(n) = fib(n-1) + fib(n-2)$$

- Implementación recursiva:  $O(\varphi^n)$
- Implementación usando programación dinámica:  $\Theta(n)$



```
if (n == 0) return 0;
else if (n == 1) return 1;
else {
    previo = 0; actual = 1;
    for (i=1; i<n; i++) {
        fib = previo + actual;
        previo = actual; actual = fib;
    }
    return actual;
}
```



## Algoritmos voraces:

```
funcion voraz(C:conjunto):conjunto
{ C es el conjunto de todos los candidatos }
S <= vacio { S es el conjunto en el que se construye la solucion}
mientras  $\neg$ solucion(S) y C <> vaciohacer
    x <= el elemento de C que maximiza seleccionar(x)
    C <= C \ {x}
    si completable(S U {x}) entonces S <= S U {x}
si solucion(S)
    entonces devolver S
    si no devolver no hay solucion
```

### Algoritmos de fuerza bruta:

#### #CIFRADO CESAR

```
texto=input("Tu texto: ")

abc="ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789"

k=int(input("Valor de desplazamiento: "))
cifrad=""

for c in texto:
    if c in abc:
        cifrad += abc[(abc.index(c)+k)%(len(abc))]
    else:
        cifrad+=c

print("Texto cifrado: ",cifrad)
```

### Algoritmos de vuelta atrás:

**Paso 1.**  $S \leftarrow \{\text{vinicial}\}$  //Inicialmente S contendrá el vértice //origen

**Paso 2.** Para cada  $v \in V$ ,  $v \neq \text{vinicial}$ , hacer

2.1.  $D[v] \leftarrow C[\text{vinicial}, v]$  //Inicialmente el costo del //camino mínimo de vinicial a v es lo contenido en //la matriz de costos

2.2.  $P[v] \leftarrow \text{vinicial}$  //Inicialmente, el //predecesor de v en el camino mínimo construido //hasta el momento es vinicial

**Paso 3.** Mientras  $(V - S \neq \emptyset)$  hacer //Mientras existan vértices para //los cuales no se ha determinado el //camino mínimo

3.1. Elegir un vértice  $w \in (V - S)$  tal que  $D[w]$  sea el mínimo.

3.2.  $S \leftarrow S \cup \{w\}$  //Se agrega w al conjunto S, pues ya se //tiene el camino mínimo hacia w

3.3. Para cada  $v \in (V - S)$  hacer

3.3.1.  $D[v] \leftarrow \min(D[v], D[w] + C[w, v])$  //Se escoge, entre //el camino mínimo hacia v que se tiene //hasta el momento, y el camino hacia v //pasando por w mediante su camino mínimo, //el de menor costo.

3.3.2. Si  $\min(D[v], D[w] + C[w, v]) = D[w] + C[w, v]$  entonces  $P[v] \leftarrow w$  //Si se escoge ir por w entonces //el predecesor de v por el momento es w

**Paso 4.** Fin

4. Dentro de las razones para utilizar Python se mencionó que es multiparadigma. En este ejercicio debes leer, comprender y explicar que es y para qué sirve cada uno de los paradigmas mencionados [orientado a objetos, imperativo y funcional].

Paradigma de la Programación Orientada a Objetos:

El paradigma de la programación orientada a objetos nació en 1969 de la mano de Kristin Nygaard. Desde entonces, las tecnologías orientadas a objetos han evolucionado mucho.

De acuerdo con Booch (1995), un objeto es algo que tiene *estado*, *comportamiento* e *identidad*. Por ejemplo, en el mundo real, un auto (objeto), tiene un determinado, color, cantidad de puertas, cantidad de velocidades, capacidad de carga, etc. (estado), y es capaz de avanzar, frenar, cambiar de velocidad (comportamiento). Aunque dos autos tengan características muy parecidas, en realidad son diferentes (identidad). En un diseño orientado a objetos se crea una abstracción (o modelo simplificado) del auto basado en sus estado y comportamiento. Un objeto conoce cómo ejecutar acciones que alteren sus propios datos. Un objeto consta de datos y acciones que se pueden ejecutar.

Un objeto es transitorio: se pierde cuando el programa termina, a menos que pueda guardar sus datos en un disco o almacenamiento.

En conclusión:

Un objeto (POO) trata de emular una entidad del mundo real, sea ésta una entidad física, conceptual o de software. Tiene estado (un conjunto de datos), comportamiento (acciones que se pueden (ejecutar) e identidad.

Referencias:

1. Booch (1995) **Análisis y diseño orientado a objetos con aplicaciones.**

## Paradigma Funcional:

La programación funcional se basa como su nombre lo indica en la construcción de programas utilizando funciones puras, sin efectos secundarios este método surgió desde 1958 por John McCarthy y cumple con:

1. Evaluación por reducción funcional. Técnicas: recursividad, parámetros acumuladores, CPS, Mónadas.
2. Los programas se componen de funciones, es decir, implementaciones de comportamiento que reciben un conjunto de datos de entrada y devuelven un valor de salida.
3. Las funciones son elementos de primer orden
4. Basado en los modelos de cómputo cálculo lambda (Lisp, Scheme) y lógica combinatoria (familia ML, Haskell)

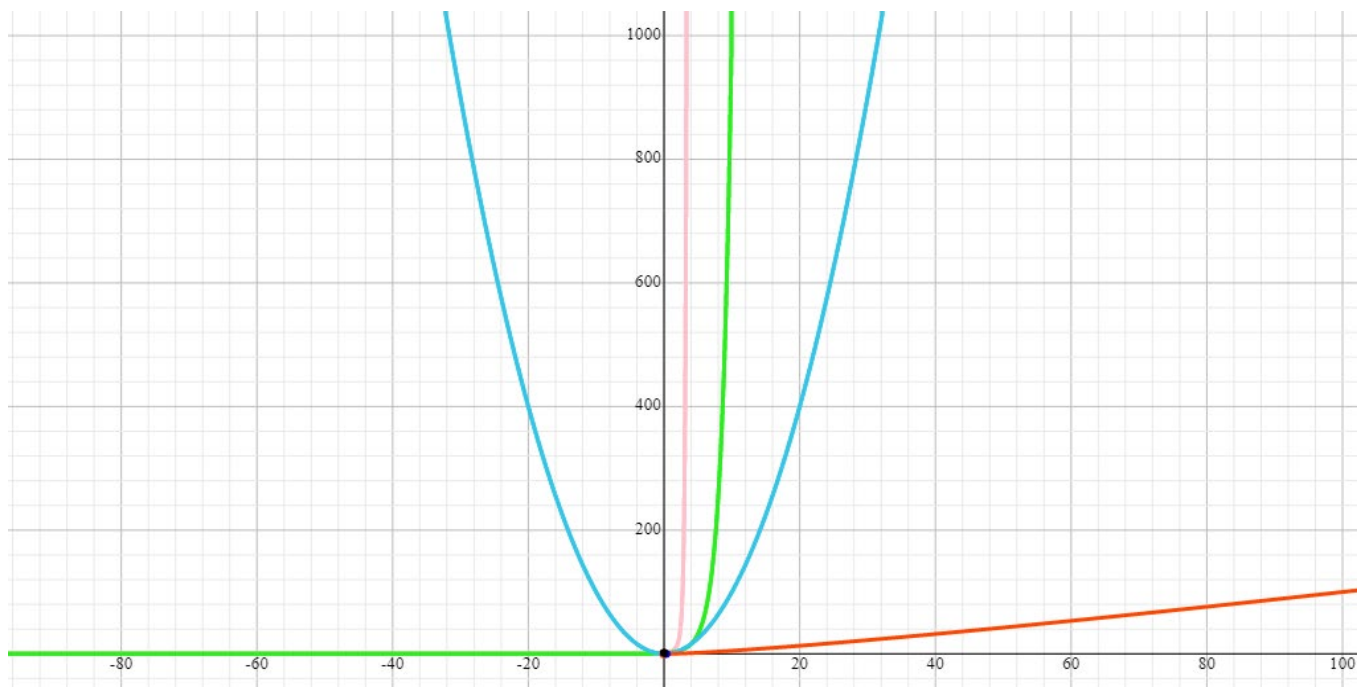
Y esto surge para resolver dos problemas uno la representación matemática de orden superior y para resolver el problema del paradigma línea que se convertía en un espagueti.

## Paradigma Imperativo:

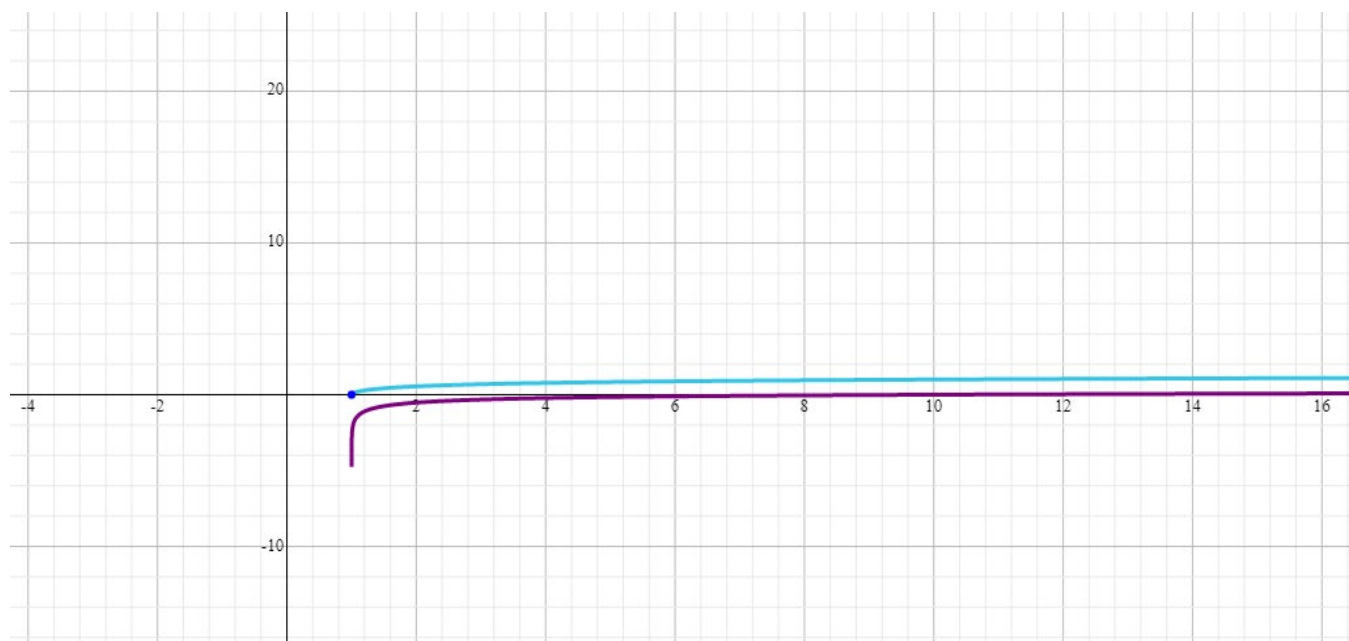
La programación imperativa (del latín *imperare* = ordenar) es el paradigma de programación más antiguo. De acuerdo con este paradigma, un programa consiste en una **secuencia claramente definida de instrucciones para un ordenador**.

El código fuente de los lenguajes imperativos encadena instrucciones una detrás de otra que determinan lo que debe hacer el ordenador en cada momento para alcanzar un resultado deseado. Los valores utilizados en las variables se modifican durante la ejecución del programa. Para gestionar las instrucciones, se **integran estructuras de control como bucles o estructuras anidadas en el código**.

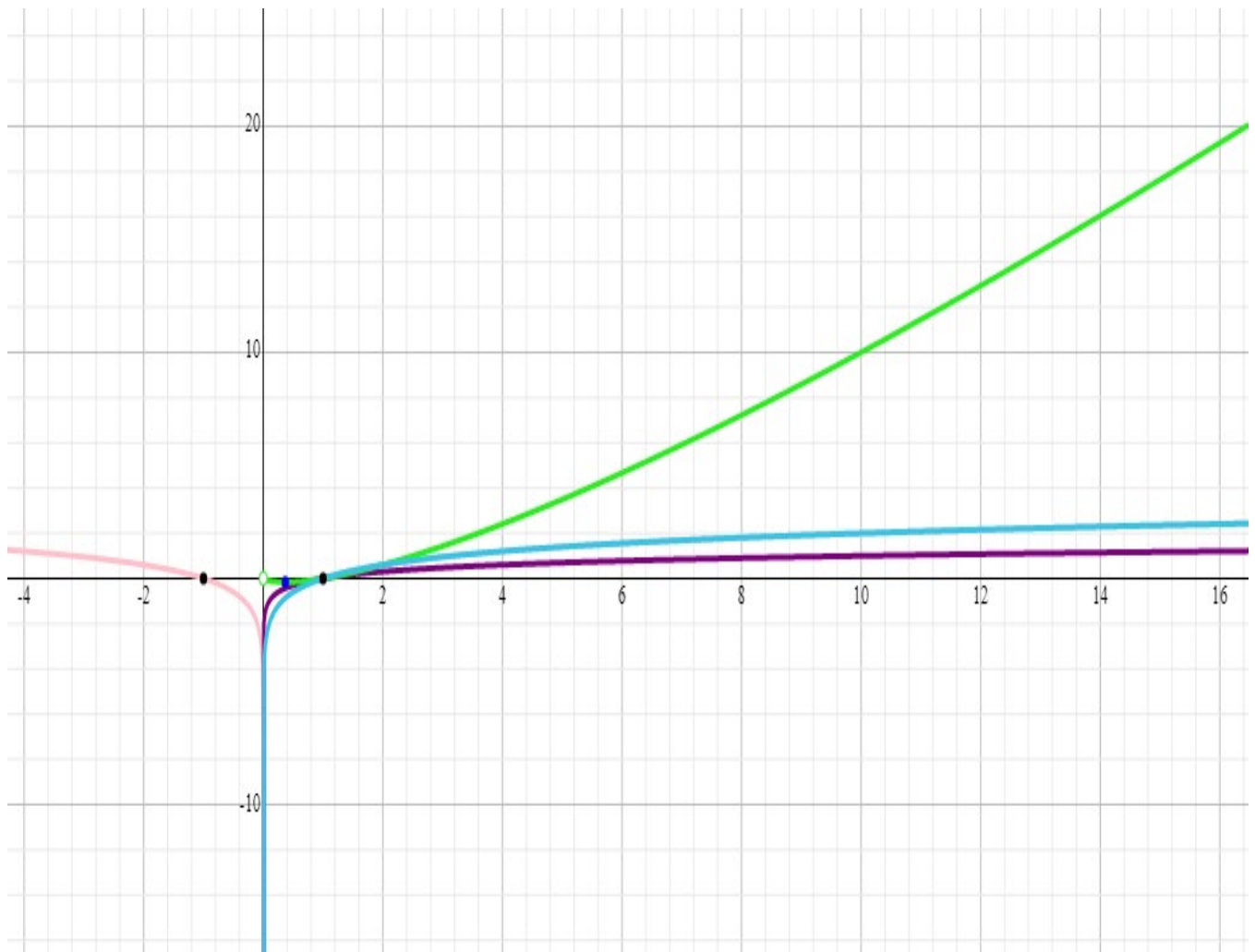
Los lenguajes de programación imperativa son muy concretos y trabajan cerca del sistema. De esta forma, el código es, por un lado, fácilmente comprensible, pero, por el otro, requiere **muchas líneas de texto fuente** para describir lo que en los lenguajes de la programación declarativa se consigue con solo una parte de las instrucciones.







<span style="color: pink;">●</span>	$2^{2^n}$
<span style="color: green;">●</span>	$2^n$
<span style="color: blue;">●</span>	$n^2$
<span style="color: orange;">●</span>	$n \log_{10}(n)$



<span style="color: purple;">●</span>	$\log_{10}(\log_{10}(n))$
<span style="color: cyan;">●</span>	$\sqrt{\log_{10}(n)}$



	$\log_{10}(n)^2$
	$\log_{10}(n)$
	$n \log_{10}(n)$
	$2 \log_{10}(n)$