

RESUMEN ALGORITMOS Y ESTRUCTURAS DE DATOS

Ficha 1 a 11

Ficha 1: Fundamentos de programación

CÁLCULOS Y FÓRMULAS NECESARIAS:

Segundos = horas * minutos

Tiempo = velocidad / distancia

Promedio = suma de datos / cantidad de datos. Ej: notas: $(8 + 9 + 4) / 3$

Porcentaje = (valor x * 100) / total. Ej: votos a favor = $(favor * 100) / \text{total de votos}$

Cantidad real de un porcentaje = porcentaje (%) * valor / 100

ejemplo: 30% de 120 : $(30 * 120) / 100 = 36$ (el 30% de 120 es 36.)

Área de un rectángulo = base * altura

Área de un triángulo = base * altura / 2

Área de un cuadrado = lado * lado

Perímetro = suma de los lados

Superficie = largo * ancho

Horas a min = horas * 60min

Min a segs = minutos * 60s

Horas a segs = horas * 3600s

Segs a min = min / 60s

Min a hrs = hs / 60m

Algunas definiciones.....

ALGORITMOS: Conjunto finito y ordenado de pasos que tienen como objetivo RESOLVER UN PROBLEMA. (receta de cocina, manual de instrucciones)

PROGRAMA: Es un algoritmo definido en los términos de un lenguaje de programación.

ESTRUCTURAS DE ALGORITMOS:

Datos → Procesos → Resultados

PASOS PARA RESOLVER UN PROBLEMA:

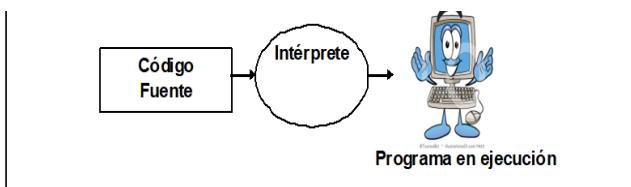
1. Entender el problema
2. Especificar un algoritmo
3. Desarrollar un programa
4. Verificar y ejecutar el programa

LENGUAJE DE PROGRAMACIÓN: Conjunto de símbolos, palabras y reglas para combinar esos símbolos y palabras de forma muy precisa y detallada, con los que pueden escribirse programas de computadoras.

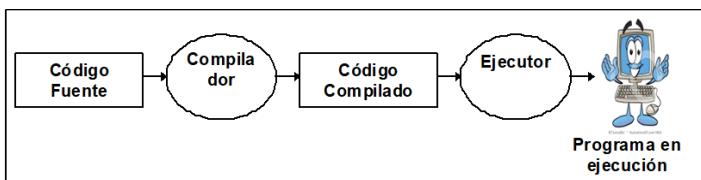
SINTAXIS DEL LENGUAJE: Exige que esas palabras, signos y operadores se escriban de una forma determinada, para que la instrucción sea válida.

CÓDIGO FUENTE: Es el programa que escribe un programador utilizando lenguaje de programación.

INTÉPRETE: Es un programa que verifica el archivo de código fuente de otro programa, detectando errores. Este ejecuta el programa línea por línea. Si la línea que está analizando no tiene errores, la ejecuta y pasa a la siguiente, y así hasta llegar al final o detectar un error, en el caso del error, la ejecución se interrumpe mostrando el error.



COMPILADOR: La breve diferencia que tiene este con el intérprete, es que este no puede ejecutar de una el código fuente, sino que genera otro archivo llamado código compilado y si está bien, recién ahí se ejecutará. Este es más rápido, pero no funciona para todas las plataformas.



ELEMENTOS BÁSICOS DE PROGRAMACIÓN EN PYTHON:

VARIABLE:

- Es la unidad mínima de almacenamiento en memoria
- En python se crean automáticamente al asignarles un valor
- Se puede usar y modificar su contenido
- Está asociada a un identificador

- Ej: edad = 12

Tabla de tipos de datos elementales

TIPO	DESCRIPCIÓN	ESCRITURA	RANGO
bool	valores lógicos (True, False)	T y F mayús.	True, False
int	números enteros	Se escribe el número	ilimitado
float	números reales (coma flotante)	Número con punto	hasta 15 decimales
str	cadena de caracteres (" ")	Con comillas	Unicode

Print(): permite mostrar en pantalla el contenido de una variable, como también mensajes formados por cadenas de caracteres. `print(a)`

None: valor de una variable, que es como nulo o False

Reglas para las variables:

- Sólo puede contener letras del alfabeto inglés (mayúsculas y/o minúsculas, o también dígitos (0 al 9), o también el guión bajo (_).
- El nombre de una variable *no debe comenzar* con un dígito.
- Las palabras reservadas del lenguaje Python no pueden usarse como nombres de variables.
- El nombre de una variable puede contener cualquier cantidad de caracteres de longitud.
- Es *case sensitive*: Python hace diferencia entre minúsculas y mayúsculas, por lo que toma como diferentes a dos nombres de variables que no sean exactamente iguales.
- El nombre de una variable se escribirá siempre en minúscula salvo que sea una variable de algún valor muy conocido y escrito en mayúscula (G gravedad).

Input():

- Permite realizar operaciones de carga de datos desde el teclado mientras el programa se está ejecutando .
- Siempre asigna un str (cadena de caracteres)
- Habilita el teclado hasta que alguien escriba algo o asigne un valor
- nom = input('Ingrese su nombre: ')
- Para cargar por teclado un número entero o un número en coma flotante:
 - **int(cadena):** retorna el número entero representado por la cadena tomada como parámetro.
`n = int(input('Ingrese un valor entero: '))`
 - **float(cadena):** retorna el número en coma flotante representado por la cadena tomada como parámetro.
`x = float(input('Ingrese un valor en coma flotante: '))`

Tabla de operadores aritméticos:

Operador	Significado	Ejemplo de uso
+	suma	$a = b + c$
-	resta	$a = b - c$
*	producto	$a = b * c$
/	división de coma flotante	$a = b / c$
//	división entera	$a = b // c$
%	resto de una división	$a = b \% c$
**	potencia	$a = b ** c$
$\sqrt{}$	raíz	$a = n ** 0.5$

Shell o IDLE de Python: Programa que permite editar, depurar, ejecutar y testear programas de forma básica.

Ficha 2: Estructuras Secuenciales

Estructuras secuenciales:

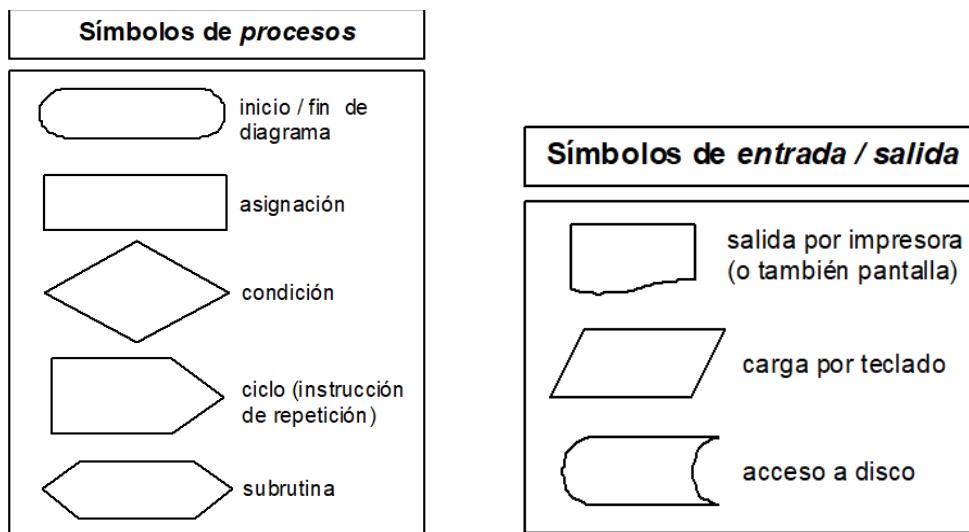
Analizar y resolver problemas simples de lógica lineal, es decir, problemas que no son divididos en subproblemas.

Estos pueden ser resueltos mediante la aplicación de secuencias de instrucciones simples, una debajo de la otra, de tal forma que una se ejecuta después de la otra. Esto se conoce como estructura secuencial de instrucciones.

Diagramas de flujo:

Es una técnica auxiliar para todos los lenguajes, de la representación general de algoritmos, para poder reflejar con más claridad la lógica general del algoritmo.

Es un gráfico que permite representar en forma clara y directa el algoritmo para resolver un problema. Se basa en el uso de símbolos unidos por líneas rectas descendentes.



Representan una operación para transformar datos en resultados.

.

Representan una acción o instrucción.
Operaciones de carga de datos o salida de resultados.

Pseudocódigo:

- Técnica auxiliar para el planteo de un algoritmo. Es una forma de código informal.
- Plantea el algoritmo escribiendo cada acción o paso en lenguaje natural. Está pensado para ser leído y entendido por una persona.
- Pasos:
 - Comenzar indicando el nombre del proceso seguido de 2 puntos
 - Enumerar cada paso en forma correlativa. Si hay subpasos, enumerarlos en base al principio.
 - Mantener la indentación general
 - Mantener consistencia. Designar los pasos de la misma forma y estilo.
- Ejemplo: Algoritmo:
 - 1) Cargar en cn la cantidad de enfermos del país
 - 2) Cargar en cc la cantidad de enfermos de la ciudad.
 - 3) Calcular y asignar en pc el porcentaje: $pc=cc*100/cn$
 - 4) Mostrar el porcentaje pc.

Operadores y precedencia:

- Primeros operadores de *multiplicación* (*), *división* (//, /), *resto* (%) y *potencia* (**).
- Luego, operadores de *suma* (+) y *resta* (-).
- Si dos operadores tienen la misma precedencia, se aplica primero el que esté más a la izquierda.
- Cualquier expresión encerrada entre paréntesis se ejecutará primero. Los paréntesis se usan para cambiar la precedencia de ejecución de algún operador.

Ficha 3: Tipos Estructurados Básicos

ESTRUCTURAS SECUENCIALES

TIPOS DE DATOS:

Tipos Simples : (int, float y bool) Una variable pueda contener *un único valor de ese tipo*.

Tipos Compuestos o estructurados : *Una misma variable puede contener varios valores al mismo tiempo.*

Secuencias de datos mutables: Los valores individuales pueden modificarse. (Listas)

Secuencias de datos inmutables: Los valores no pueden modificarse. (Tuplas, str, rangos, secuencias binarias.)

Tuplas:

Es una secuencia inmutable de datos que pueden ser de tipos diferentes.

t = a, b, c ó t = tuple(a,b,c).

Expresión de Asignación con tuplas

- El operador coma (,) actúa como un empaquetador cuando está a la derecha de la asignación, produciendo la tupla con los valores obtenidos.
- El operador coma (,) actúa como un desempaquetador si está a la izquierda de la expresión

LEN(): Puede usarse para determinar la cantidad de elementos que tiene una secuencia (tupla, lista, cadena). Cuenta los elementos de una tupla

TUPLE(): Convierte en tupla

SECUENCIA DE ÍNDICES: Cada casillero de una secuencia, está asociado a un número de orden llamado índice, mediante el cual se puede acceder al casillero en forma individual. La secuencia de índices comienza desde 0 y se separan con comas.

Para acceder a un elemento individual de una secuencia, escribir el identificador y entre corchetes el número de índice del casillero.

Ej: t = hola

```
print( t[0] )      # muestra = h
```

INTERCAMBIO DE VALORES: Intercambio entre a y b:

```
#valores iniciales
```

```
a = 12
```

```
b = 4
```

```
#intercambio con una tupla
```

```
a, b = b, a
```

Cadenas de caracteres:

Secuencia inmutable de caracteres. Se puede entender como un contenedor dividido en tantas casillas como caracteres tenga esa cadena.

Un literal es delimitado por comillas ("").

El operador suma (+) permite concatenar dos o más cadenas.

El producto (*) permite repetir una cadena varias veces, concatenando el resultado

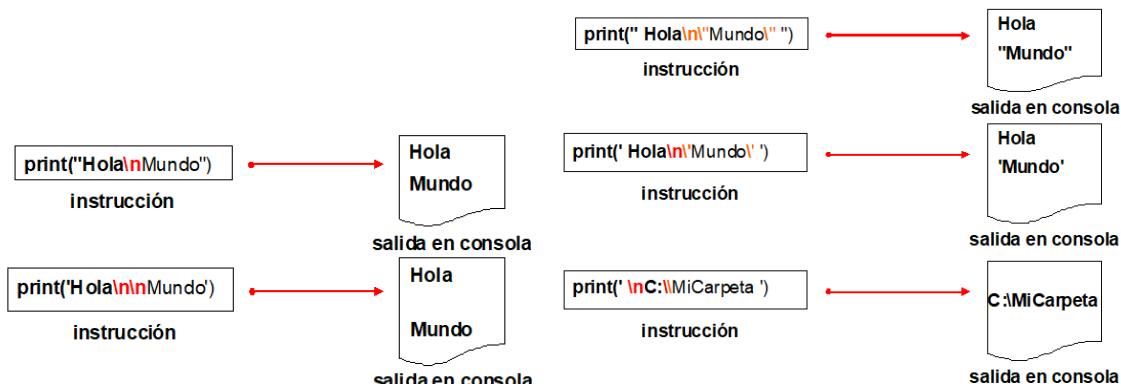
Caracteres de escape / control:

Se forman con una \ seguida de algún carácter especial que actúa en forma de código:

\n = SALTO DE LÍNEA (ENTER) Ej: ("hola\nmundo") ó ("hola", "\n", "mundo") o ("Hola \n\nmundo"),

\t = 4 ESPACIOS EN LA MISMA LÍNEA. Ej: ("hola\tmundo")

'\'' = REPRESENTAN LITERALMENTE LAS COMILLAS. Ej: "Anthony \"Tony\" Hoare)



Principales funciones de la librería:

- **Float():** Permite convertir una cadena a un número en coma flotante.
- **Input():** Permite tomar desde el teclado una carga de una cadena.
- **Int():** Para convertir una cadena en un número.
- **Abs():** Toma como parámetro un número y devuelve el valor absoluto de ese número.
- **Bin():** Toma como parámetro un número entero y devuelve el mismo número pero convertido en binario. (Ej: x = 8 s = bin(x) print(s) #muestra: 0b1000.)
- **Chr():** Toma como parámetro un número de orden y devuelve el carácter de la tabla unicode que corresponde a ese número. (Ej: i = 65 c = chr(i) print(c) #muestra: A)
- **Divmod(a,b):** Retorna 2 valores el cociente y el resto de la división entera entre los parámetros a y b.

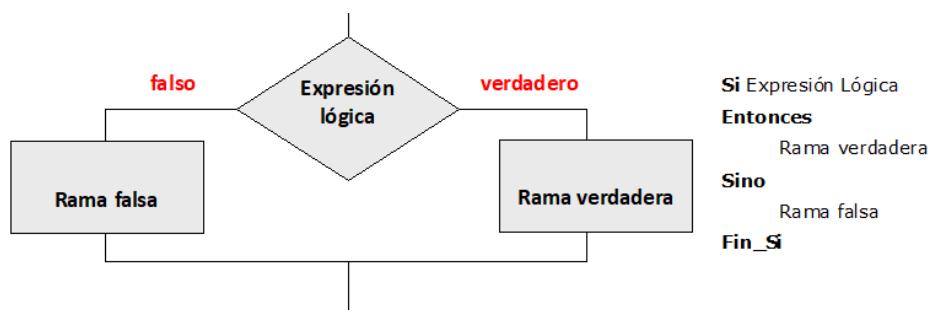
- **Hex():** Permite obtener la conversión hexadecimal de un valor tomado como parámetro. (Ej: `x = 124 s = hex(x) print(s) #muestra: 0x7c`)
- **Len(s):** Para tomar la longitud de una secuencia.
- **Max(a,b, * args):** Toma como parámetro una secuencia de valores y devuelve el valor más grande.
- **Min(a,b, * args):** Toma como parámetro una secuencia de valores y devuelve el valor menor que le dimos como parámetro.
- **Pow(x, y):** Retorna el valor de x elevado a la y. Es lo mismo que `**`. (potencia)
- **Round(x,n):** Retorna el número flotante, pero redondeado a n dígitos los decimales.
Ej: `x = 4.1485 r = round(x,2) print(r) #muestra: 4.15.`
- **Oct():** Toma un número en base 10 y devuelve la conversión octal (8) de ese número. Ej: `x=124 s=oct(x) print(s) #muestra: 0o174`
- **Ord():** Le damos como parámetro un carácter y devuelve el número de orden de ese símbolo en la tabla unicode. (Ej: `c = "A" i = ord(c) print(i) #muestra: 65`).
- **nombre.lower():** Transformar a minúscula

Subconjunto de la guía PEP 8:

- Coloque espacios alrededor de los operadores en un expresión, y después de las comas en una enumeración, pero no inmediatamente a los lados de cada paréntesis cuando los use: `a = f(1, 2) + g(3, 4)`
- Agregar espacios alrededor de los operadores que tengan menor prioridad, y eliminar los espacios en los que tengan prioridad mayor. Sin embargo, nunca coloque más de un espacio y siempre coloque la misma cantidad de espacios antes y después del mismo operador.
- Corte sus líneas de código en no más de 79 caracteres por línea.
- Use líneas en blanco para separar funciones, clases y largos bloques de código dentro de una función o una secuencia.
- Cuando sea posible, coloque sus comentarios en líneas específicas para esos comentarios (no los agregue en la misma línea de una instrucción).
- Utilice cadenas de documentación (docstrings): mantenga actualizada la documentación de sus programas.
- No utilice codificaciones de caracteres demasiado extrañas.
- Evite el uso de caracteres no-ASCII en el nombre de un identificador.
- Comillas dobles o simples: si comenzó con una forma de hacerlo, apeguese a ella y no la cambie a cada momento.
- Cuando necesite incluir un tipo de comilla (simple o doble) en una cadena, use la otra para delimitar.
- Nunca utilice la l ('letra ele' minúscula) ni la O ('letra o' mayúscula) ni la I ('letra i' mayúscula) como nombre simple de una variable.

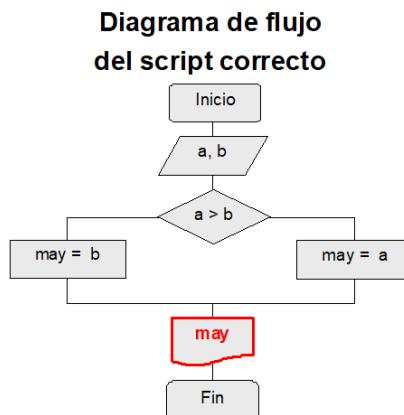
Ficha 4: Estructuras Condicionales

- Contiene una expresión lógica que puede ser evaluada por verdadero o falso.
- Contiene 2 bloques de instrucciones adicionales designados rama falsa y rama verdadera.
- Diseñada para el chequeo de una o más condiciones y ejecutar un conjunto de instrucciones, según el valor de la condición.
- Requiere comprobar el valor de alguna condición, y en función de ellos proceder a dividir la lógica del algoritmo en 2 o más ramas de ejecución.
- NUNCA se ejecutan ambas ramas
- “if” : da inicio a la estructura condicional y es la rama verdadera
- “else”: rama falsa



- Ejemplo: if a > b :

```
may = a
else:
    may= b
```



INDENTACIÓN: Python identifica a las instrucciones que pertenecen a un mismo bloque de acuerdo a su **encolumnado o nivel de indentación**. PARA INDENTAR UTILIZAMOS TAB.

EXPRESIONES LÓGICAS: Operadores relacionales y conectores lógicos:

Una **expresión** es una fórmula compuesta por variables y constantes (**operandos**) y por símbolos que indican la aplicación de una acción (**operadores**).

Expresión Aritmética:

- Su resultado es un número.
- Utilizan operadores aritméticos: suma, resta, producto, etc.

Expresión Lógica:

- Su resultado es un valor de verdad (True o False).
- Utilizan operadores relacionales y lógicos.

TABLA DE OPERADORES DE COMPARACIÓN EN PYTHON:

Operador	Significado	Ejemplo	Observaciones
<code>==</code>	igual que	<code>a == b</code>	retorna <i>True</i> si <i>a</i> es igual que <i>b</i> , o <i>False</i> en caso contrario
<code>!=</code>	distinto de	<code>a != b</code>	retorna <i>True</i> si <i>a</i> es distinto de <i>b</i> , o <i>False</i> en caso contrario
<code>></code>	mayor que	<code>a > b</code>	retorna <i>True</i> si <i>a</i> es mayor que <i>b</i> , o <i>False</i> en caso contrario
<code><</code>	menor que	<code>a < b</code>	retorna <i>True</i> si <i>a</i> es menor que <i>b</i> , o <i>False</i> en caso contrario
<code>>=</code>	mayor o igual que	<code>a >= b</code>	retorna <i>True</i> si <i>a</i> es mayor o igual que <i>b</i> , o <i>False</i> en caso contrario
<code><=</code>	menor o igual que	<code>a <= b</code>	retorna <i>True</i> si <i>a</i> es menor o igual que <i>b</i> , o <i>False</i> en caso contrario

También permiten comparar cadenas de caracteres(str). Python comparará de acuerdo al alfabeto (las primeras letras).

CONECTORES LÓGICOS

- Un conector lógico u operador **booleano** permite **encadenar** la comprobación de dos o más expresiones lógicas y obtener un resultado único.
- Convieren la expresión analizada en una expresión lógica o booleana.
- La expresión lógica encadenada por un conector lógico se designa como **proposición lógica**
- Para unir 2 o más proposiciones en una sola expresión se usan operadores llamados conectores lógicos u operadores lógicos.
- And y Or se aplican a dos o más proposiciones (operador binario)
- Not se aplica a una sola (operador unitario)
- AND: Será verdadera si todas son verdaderas
- OR : Será falsa si todas son falsas

TABLA DE CONECTORES LÓGICOS

Operador	Significado	Ejemplo
and	conjunción lógica (y)	a == b and y != x
or	disyunción lógica (o)	n == 1 or n == 2
not	negación lógica (no)	not x > 7

PRECEDENCIA DE OPERADORES

- conectores lógicos < op. de comparación < aritméticos
- Los operadores de comparación tienen la misma precedencia entre ellos y se aplican de izquierda a derecha.
- En los conectores lógicos la precedencia es:
precedencia(**or**) < precedencia(**and**) < precedencia(**not**)

RANDOM: GENERADOR DE VALORES ALEATORIOS

- Genera números al azar
- Al inicio del script escribir “import random”
- **random.random()** : núm en coma flotante (entre [0,1] no incluye al 1)
- **random.randrange()**: número entero (excluye al número final)
- **random.randint()**: número entero (incluye al final)
- **random.choice()**: acepta una secuencia, una tupla, cadena, lista, etc.
- **random.seed()**: mantiene los mismos números

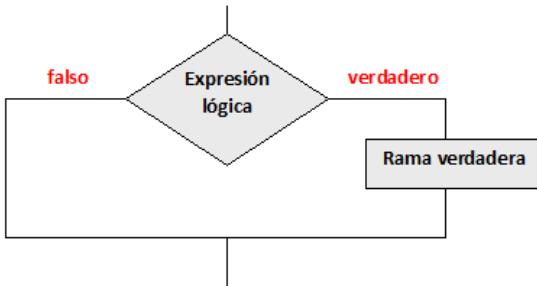
IN:

El operador In (en) devuelve True si un elemento se encuentra dentro de otro o False si no se encuentra.

Ficha 5: Estructuras Condicionales. Variantes

CONDICIÓN SIMPLE:

- Es una instrucción condicional, pero solo contiene la rama verdadera y omite la falsa.
- No se especifica el “else”.
- if expresión lógica:
 - instrucciones de la rama verdadera
 - Continúa el programa.



ANIDAMIENTO DE CONDICIONES:

Son varias instrucciones condicionales dentro de otras. En la rama falsa y/o verdadera de una instrucción condicional, es posible plantear una nueva instrucción condicional.

ELIF:

Variante para la instrucción condicional, que permite evitar el anidamiento excesivo de condiciones.

Variante if - elif. Equivale a un “else” que contenga a su vez otra condición.

Expresiones de conteo y acumulación:

Variables que en una expresión de asignación aparecen en ambos miembros: la misma variable se usa para hacer un cálculo y para recibir la asignación del resultado de ese cálculo.

variable = variable (cuyo valor se actualiza) - **operador (+ - * /)** - **expresión** (constante)

VARIABLE DE CONTEO:

- Actualiza su valor en términos de su propio valor anterior y de una constante.
- Lo uso para contar
- Para los procesos de conteo (por ejemplo, determinar cuántos productos se compraron), se utiliza la **variable de conteo o contador**.
- **Forma general:** contador = valor inicial
contador = contador + paso
- Ej: cuántos números negativos hay:

a = 0
if num < 0:
 a = a + 1
- Cada vez que alguno de los números cargados en la variable num sea negativo, se ejecuta a = a + 1
- El valor final de a indicará cuántos números negativos se ingresaron.
- a = a (+ - / *) 1

VARIABLE DE ACUMULACIÓN:

- Permite sumar los valores que va asumiendo otra variable.
- Similar al contador, pero ahora se suma una variable “x”.
- Lo uso para acumular
- *Para los procesos de sumarización* (por ejemplo, determinar cuánto es el importe total de una compra, es decir la sumatoria de todos los importes), se utiliza la **variable de acumulación o acumulador**.
- Ej: ir sumando valores que adquiere la variable x.

$s = 0$

$x = \text{num}$

$s = s + x$

Forma general	Forma resumida
$a = a + 1$	$a += 1$
$b = b - 1$	$b -= 1$
$c = c + 2$	$c += 2$
$d = d * 3$	$d *= 3$
$e = e / 4$	$e /= 4$
$s = s + x$	$s += x$
$b = b * z$	$b *= z$
$a = a - x$	$a -= x$
$p = p / t$	$p /= t$
$c = c + 2*x$	$c += 2*x$

Variables centinela o banderas:

- Marcar un suceso ocurrido a lo largo de la ejecución de un programa, para chequear en distintos puntos del programa si se cumplió o no.
- De acuerdo al valor que tenga en un momento determinado se puede determinar si se ha ejecutado o no el proceso que la utilizó.
- El desarrollador elige los valores posibles que puede asumir la bandera en cada momento de ejecución.
- Ej: notas = False
 $\text{if } n1 \geq 7 \text{ and } n3 \geq 7 \text{ and } no \geq 8:$
 $\quad \quad \quad \text{notas} = \text{True}$

Ficha 6: Estructuras repetitivas: WHILE

- Un ciclo o estructura repetitiva es una instrucción compuesta, que permite la repetición controlada de la ejecución de cierto conjunto de instrucciones en un programa, determinando si la repetición debe detenerse o continuar.
- Permiten repetir la ejecución de un bloque de instrucciones.

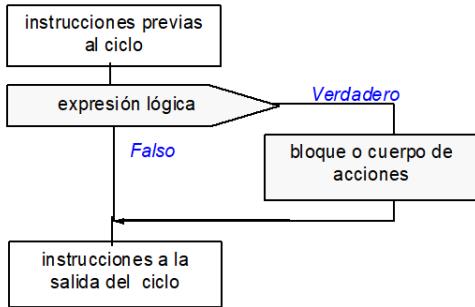
- En estas se encuentran el ciclo “for” y el ciclo “while”.
- Compuesto por:
 - La **cabecera del ciclo**, que incluye una **condición de control** y/o elementos adicionales en base a los que se determina si el ciclo continúa o se detiene.
 - El **bloque de acciones o cuerpo del ciclo**, que es el conjunto de instrucciones cuya ejecución se pide repetir.

Ciclo While: (cuando...)

- Plantear un ciclo que ejecute en forma repetida un bloque de acciones, sin conocer o conociendo la cantidad de vueltas a realizar.
- Es una estructura que repite un bloque de acciones mientras se cumpla una determinada condición.
- No se indica la cantidad de repeticiones, se indica alguna condición que especifique cómo debe cortar el ciclo, “**condición de corte**”.
- En un ciclo bien planteado, el valor de la o las variables que se usan en la expresión lógica de la cabecera debe cambiar adecuadamente dentro del bloque, para no generar un ciclo infinito.
- **Cabecera:**
 - contiene una expresión lógica que es evaluada
 - si esta es verdadera, el ciclo se ejecutará repetidamente, pero si es falsa esta se detiene o no se ejecuta.
- **while expresión lógica:**
 - bloque de acciones
- **Ejemplo** sin conocer el valor de vueltas:

```
a = 0
num = int(input("Ingrese un número o cero para terminar"))
while num != 0:
    if num < 0:
        a += 1
    num = int(input("Ingrese un número o cero para terminar"))
```
- **Ejemplo** sabiendo la cant. de vueltas:

```
x = int(input("Ingrese un número "))
cant_vueltas = 1
while cant_vueltas <= 10:
    print(x, "*", cant_vueltas, "=", x*i)
    cant_vueltas += 1
```

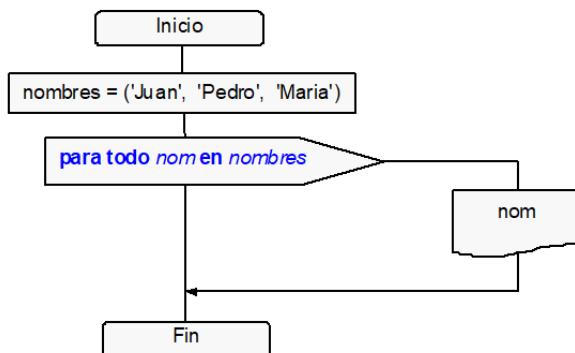


Ficha 7: Estructuras repetitivas: Ciclo FOR

Ciclo for: (para todo...)

- Repite un bloque de acciones un **número especificado** de veces.
- Se utiliza cuando se conoce previamente la cantidad de repeticiones a realizar.
- Diseñado para *recorrer secuencias* como *tuplas*, *cadenas de caracteres*, *rangos*, *listas*, etc . **Recuperando de a un elemento por vez**
- Controla el número de iteraciones o pasos a través de la **variable iteradora** o de control de ciclo.
- **Cabecera:**
 - Primera línea
 - **for nom in nombres**
 - Define una variable “iteradora” (**nom**), que recorre uno a uno los valores que se recuperen desde la estructura de datos (variable) que se pretenda recorrer (**nombres**) y el ciclo va a finalizar cuando no queden más valores por recorrer.
- **Bloque de acciones:**
 - Secuencia de instrucciones que el ciclo debe ejecutar en forma repetida.
 - Escribirse indentado a la derecha.
 - Ej: **for nom in nombres:**

```
print(nom)
```



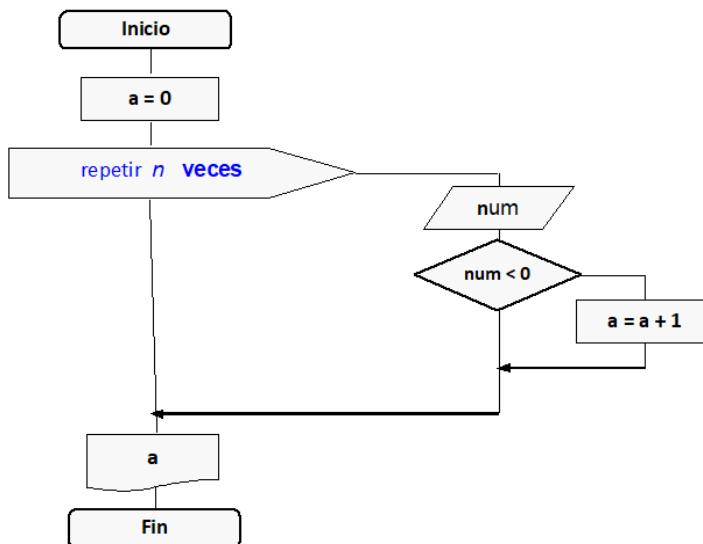
ITERAR: Es la operación de recorrer una estructura y procesar de a uno sus elementos.

VARIABLE ITERADORA: Es la variable que se usa para ir tomando uno a uno los valores de la estructura, a medida que se itere sobre ella. Ej: el ciclo *for* itera sobre la tupla *nombres* y está usando la variable iteradora *nom*.

RANGO():

- Es un tipo de secuencia numérica, de forma que el ciclo determine la cantidad de vueltas.
- Es una sucesión inmutable de valores numéricos en un **intervalo definido**, que puede ser creada con la función **range()**.
- **Funciones range()**
 - Formato general: (inicio, fin, paso)
 - **range(0,5) → (0,1,2,3,4)**
 - **range(5) → (0,1,2,3,4) →** se toma como límite inferior al 0
 - **range(1,6) → (1,2,3,4,5)**
 - **range(5,0,-1) → (5,4,3,2,1) →** orden descendente -1
 - **range(0,10,2) → (0,2,4,6,8) →** paso de 2 en 2
 - **range(n) →** (muestra lo que se cargue en la variable n)
- Ejemplo:

```
for num in range(5):
    print(num)
    #muestra los números del 0 al 4
```



COMPARACIÓN WHILE Y FOR:

while	<pre>secuencia = 'Hola' i = 0 while i < len(secuencia): print(secuencia[i]) i += 1</pre>	El desarrollador controla el contador del ciclo
for	<pre>secuencia = 'Hola' for car in secuencia: print(car)</pre>	El ciclo for controla el contador del ciclo

While se usa cuando no se conoce la cantidad de vueltas (si no que el ciclo finaliza cuando no se cumple la condición)

For se usa cuando se conoce la cantidad de vueltas (generalmente con un range, una tupla, cadena, etc).

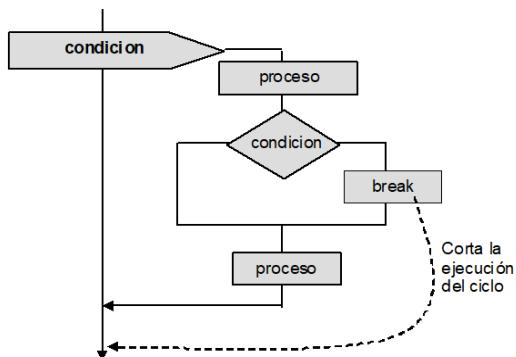
Ficha 8: Estructuras repetitivas: VARIANTES

BREAK: Corta el ciclo.

Instrucción para cortar el ciclo de inmediato sin retornar a la cabecera para evaluar la expresión lógica de control.

Ejemplo: Cargar por teclado cinco números positivos y calcular la suma de todos ellos. El ciclo termina si se carga por teclado un número cero o negativo.

```
suma = 0 - i = 1
while i <= 5:
    n = int(input('Ingrese un número mayor a cero: '))
    if n <= 0:
        break
    suma += n
    i += 1
```

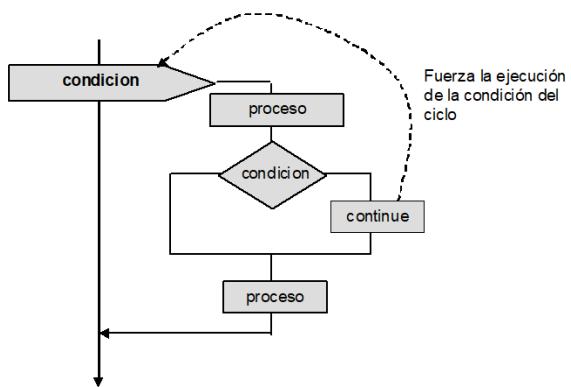


CONTINUE: Ignora instrucciones y vuelve al principio del ciclo.

Instrucción para *forzar una repetición del ciclo* sin terminar de ejecutar las instrucciones que queden por debajo de la invocación a **continue**.

Ejemplo: Cargar por teclado cinco números positivos y calcular la suma de todos ellos. El ciclo siempre pedirá los cinco números positivos.

```
suma = 0
i = 1
while i <= 5:
    n = int(input('Ingrese un número mayor a cero: '))
    if n <= 0 :
        continue
    suma += n
    i += 1
```



PASS: Saltear un código (auxiliar)

Se utiliza cuando el programador necesita *dejar vacío* un bloque de acciones.

La instrucción *pass* sirve para indicar al intérprete que simplemente considere vacío el bloque que la contiene.

Nunca deje instrucciones *pass* dentro del código de un programa que ya está terminado.
Por ejemplo, no hacer esto:

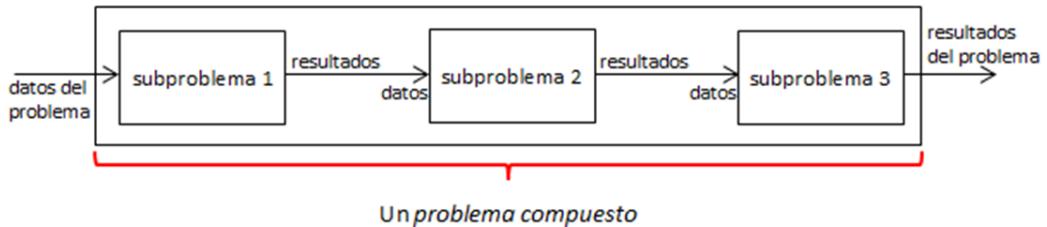
```
# una condición con rama verdadera en blanco...
if n1 > n2:
    pass
else:
    print('El primero no es el mayor')
```

Ficha 9: Subproblemas y Funciones

SUBPROBLEMA:

- Un subproblema es un problema incluido dentro de otro de estructura más compleja.

- Los problemas de estructura compuesta, se dividen en subproblemas de menor complejidad, hasta llegar a problemas simples.
- *Problema Compuesto*: Un problema que puede ser dividido en subproblemas.
- *Problema Simple*: no se puede seguir dividiendo en subproblemas.
- Cada subproblema simple es también un problema con datos, procesos y resultados.
- Los subproblemas, se implementan mediante funciones (print, pow, input, etc).
- Ej: 3 subproblemas



¿Por qué dividimos en subproblemas?

- El programador puede focalizarse en la solución de un problema más chico y puntual.
- Es más fácil realizar pruebas.
- Es más fácil descubrir y corregir errores.
- Es más fácil realizar cambios. (Adaptabilidad).
- Es factible programar un subprocesso una sola vez y usarlo muchas veces. (Reusabilidad)

PROGRAMACIÓN ESTRUCTURADA:

- La **programación estructurada** centra el análisis de un problema en los subproblemas que pudiera contener.
- Se descompone un problema en problemas menores, se programan por separado los procesos que resuelven esos subproblemas, y luego se une todo para formar el programa completo.
- Un problema compuesto se dividirá en tantos subproblemas como se hayan detectado y cada uno de ellos requerirá datos y entregará resultados. Los resultados que cada uno genere, serán datos para los subproblemas siguientes o ya serán los resultados finales esperados.

SUBRUTINAS:

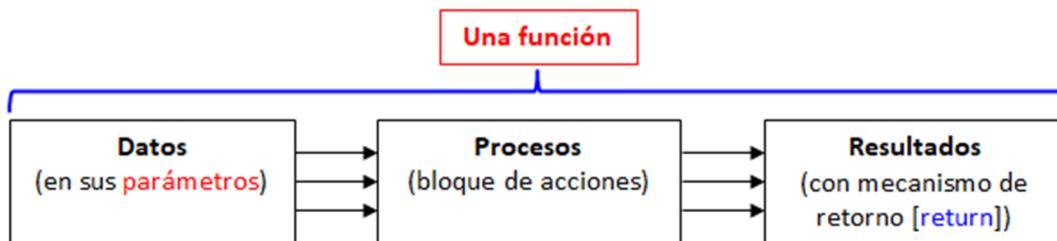
- Una subrutina es un segmento de un programa que se escribe en forma separada del programa principal.
- Las subrutas se implementan mediante funciones, y se definen a medida que se detectan subproblemas.

FUNCIONES:

Una función es un subprocesso que se escribe en forma autónoma y separada. (print, int, input, pow, etc).

Con un nombre asociado para poder activarla en cualquier momento.

- En Python, una función tiene dos partes:
- La **cabecera**: Es la primera línea de la función. En ella se indica el nombre y los parámetros.
“def nombre_funcion(): ”
- El **bloque de acciones**: Es la sección donde se indican los procesos que lleva a cabo la función.
 - Indentando hacia la derecha del comienzo de la cabecera
 - Declaraciones, asignaciones, condiciones, etc.
 - Es típico que el bloque finalice con una instrucción de retorno de valor, llamada *return*.
- Las funciones deben escribirse antes del script o bloque de código que las invoque. Por eso el "script principal" de nuestro programa está al final del código fuente.
- Una función es un proceso separado / caja negra, que acepta **entradas** o **datos (parámetros)**, procesa esos datos para obtener uno o más **resultados** o **salidas**, y devuelve esos **resultados** mediante la instrucción *return*.



- EJEMPLO: Función para el subprocesso (ordenar)

```
def ordenar(a, b):  
    if a > b:  
        men, may = b, a  
    else:  
        men, may = a, b  
    return men, may
```

Cabecera de la función: Las variables *a* y *b* encerradas entre los paréntesis se designan como **parámetros**, y contienen **los datos** que la función debe procesar.

Bloque de acciones: Las instrucciones que la función ejecuta. Los resultados se devuelven con la instrucción *return* (que en Python puede incluir varios valores).

PARÁMETROS:

Es una variable que contiene los datos que la función debe procesar.

Es lo que va entre () .

El uso de parámetros permite que una misma función pueda ser usada en forma generalizada, sobre variables diferentes que entran como datos.

No importa como se llamen las variables en el momento de llamar a la función, ya que esta reemplaza esos nombres por los nombres de sus propios parámetros.

- **Parámetros formales:** son las variables que se declaran en la cabecera de una función.

```
def suma(num1, num2):
```

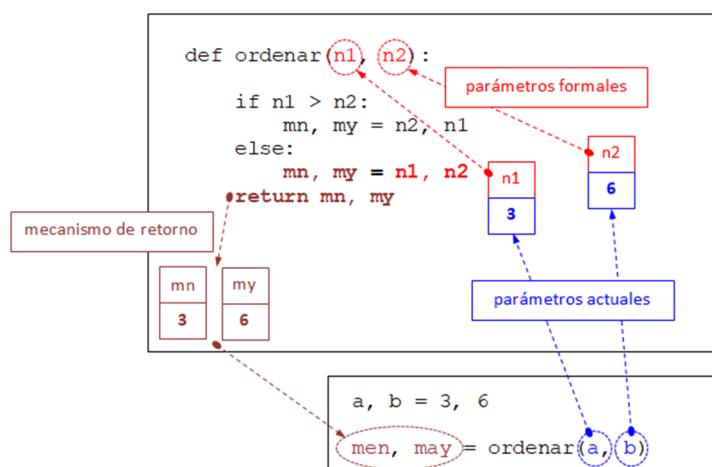
- **Parámetros actuales:** Son las variables que se escriben entre los paréntesis de una función al invocarla. (fuera de la función)

```
suma (2, 5)
```

```
suma (7, 9)
```

REGLAS DE PARÁMETROS:

1. Para invocar a una función, deben enviarse tantos *parámetros actuales* como *parámetros formales* tenga definidos la función en su cabecera. Los valores de los *actuales*, serán automáticamente asignados en los *formales*, en orden de aparición de izquierda a derecha.
2. El tipo de valor pasado en cada *parámetro actual*, debería coincidir con el tipo esperado para cada *parámetro formal*. (int, str, float, etc)
3. Los nombres de las variables de los *parámetros actuales* no tienen porqué coincidir con los nombres de los *formales*, aunque también podrían coincidir. No tiene importancia alguna si esos nombres coinciden o no.
4. En su bloque de acciones, la función opera con los *parámetros formales* (que son los *datos* que la función tiene).
5. Si se modifica el valor de algún *parámetro formal*, ese cambio no afecta al valor del *parámetro actual* (recuerde: los *formales* son COPIAS de los *actuales*).



RETORNOS:

- Funciones con retorno de valor: Son aquellas que *devuelven un valor en forma de tupla* como resultado de la acción que realizan, de forma tal que ese valor puede volver a usarse en alguna otra operación.
- Las *funciones con retorno de valor* deben incluir en alguna parte de su bloque de acciones la instrucción **return**, indicando en la misma el valor que la función debe retornar.
- La instrucción **return** es **cancelativa**: al ejecutar esta instrucción, la ejecución de la función que la contiene se *da por finalizada*, aun cuando no se haya llegado al final de la misma.
- Utilizar **return** siempre al final de la función.
- Funciones sin retorno de valor: son aquellas que realizan alguna acción pero no se espera que retornen valor alguno como resultado de la misma. (ej: `print()`)

Ejemplo de uso	Explicación
<code>y = pow(x, n)</code>	El valor devuelto se asigna en una variable <i>y</i> .
<code>print(pow(x, n))</code>	El valor devuelto se muestra directamente en consola.
<code>y = 2 * pow(x, n)</code>	El valor devuelto se multiplica por dos y el resultado se asigna en <i>y</i> .
<code>y = pow(pow(x, n), 3)</code>	Calcula x^n , eleva el resultado al cubo, y asigna el resultado final en <i>y</i> .

INVOCACIÓN O LLAMADA:

ÁMBITO DE UNA VARIABLE:

- En general la región de un programa donde una variable es reconocida y utilizable se denomina **ámbito de la variable**.
- Toda función define un espacio de nombres, que es un bloque de código en el cual las variables que se definen pertenecen a ese espacio y no pueden ser utilizadas fuera de él.

Variables locales

- Variable que se inicializa dentro del bloque de una función se considera como variable local.
- Una variable local es aquella que solo puede usarse dentro del bloque de la función y no existe o no es reconocida fuera del mismo.

Variables globales

- No están encerradas en ningún bloque de función.
- Pueden ser utilizadas en cualquier parte.

- Se recomienda no usar.
- Si se necesita que la variable local fuera considerada como global, puede usarse la palabra reservada global para marcar como globales a una o más variables.

PROGRAMA COMPLETO

```
# función para calcular menor y mayor
def ordenar(n1, n2):
    if n1 > n2
        mn, my = n2, n1
    else:
        mn, my = n1, n2
    return mn, my

# función para calcular las áreas
def areas(mn, my):
    a1 = my * my
    a2 = 3.1415 * mn * mn
    return a1, a2

# programa principal...
# títulos y carga de datos...
print('Cálculo de áreas...')
a = int(input('Primer número: '))
b = int(input('Segundo número: '))

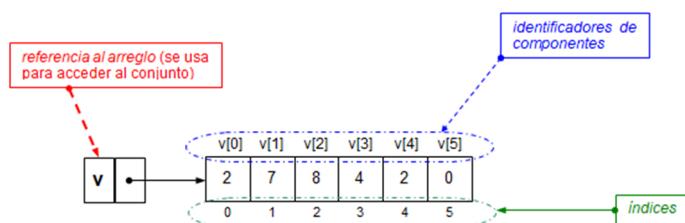
# invocación a las funciones...
men, may = ordenar(a, b)
acuad, acirc = areas(men, may)

# visualización de resultados...
print('Área del cuadrado:', acuad)
print('Área del círculo:', acirc)
```

Ficha 12: Arreglos unidimensionales

Arreglos unidimensionales/vectores:

Son variables que pueden contener una secuencia de datos, cada uno en una casilla que se identifica con un índice, comenzando con el índice cero. En Python, los arreglos (de cualquier dimensión) se representan con variables del tipo list, que representan secuencias mutables.



Algunos usos:

Arreglo vacío : v[]

Arreglo: v = [4, 7, 2]

Modificar arreglo:

```
v[1] = 3
#muestra
v = [4, 3, 2]
```

Arreglo de tamaño conocido:

ceros = 10 * [0]

```
print(ceros)
# muestra: [0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Carga de un arreglo:

```
v = []
v.append(4, 5, 6)
#muestra: v = [4, 5, 6]
```

Recorrer arreglo:

```
n = 4
v = n * [0]
for i in range(n):
    v[i] = int(input("Cargar:"))
```

Eliminar componentes:

```
v = [1, 2, 3, 4, 5]
del v[2]
#muestra: v = [1, 2, 4, 5]
```

Cortes de índices:

```
v = [1, 2, 3, 4, 5]
nueva = v[1:4] (se accede a la sublista que comienza en la casilla 1 incluida y termina en la 4 sin incluir)
#muestra: nueva[2, 3, 4]
```

Ficha 13: Arreglos - Algoritmos y técnicas básicas

Ordenamiento secuencial:

```
def ordenar(v):
    n = len(v)
    for i in range(n-1):
        for j in range(i+1, n):
            if v[i] > v[j]:
                v[i], v[j] = v[j], v[i]
    return v
```

Búsqueda secuencial:

Consiste en recorrer todo el arreglo desde la casilla 0 en adelante, hasta encontrar el valor buscado, o bien hasta llegar al final sin encontrarlo.

```
def linear_search(v, x):
    # test de pertenencia...
    for i in range(len(v)):
        if x == v[i]:
            return True
    return False
```

```
def linear_search(v, x):
    # retorno de posición...
    for i in range(len(v)):
        if x == v[i]:
            return i
    return -1
```

Búsqueda binaria:

SÓLO SI EL ARREGLO ESTÁ ORDENADO

- Buscar un punto medio en el arreglo
- Ver si es el valor q busco

- Si es, break
- Si el n° es menor al que buscamos aumentamos el inicio 1 sobre el puntero ($izq = c + 1$)
- Si el n° es mayor al que buscamos disminuimos el final 1 bajo el puntero ($der = c - 1$)

```
def binary_search(v, x):
    # busqueda binaria... asume arreglo ordenado...
    izq, der = 0, len(v) - 1
    while izq <= der:
        c = (izq + der) // 2
        if x == v[c]:
            return c
        if x < v[c]:
            der = c - 1
        else:
            izq = c + 1
    return -1
```

Add in order secuencial:

```
def add_in_order(v, reg):
    n = len(v)
    pos = n
    for i in range(n):
        if reg.algo < v[i].algo:
            pos = i
            break
    v[pos:pos] = [reg]
```

Cargar arreglo ordenado:

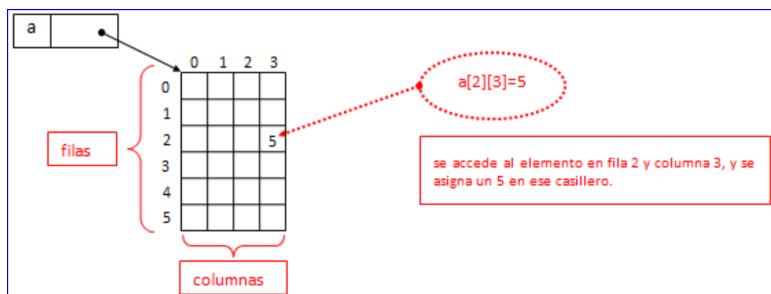
```
def cargar_arreglo():
    v = []
    n = int(input("Cantidad de...."))
    for i in range(n):
        id = random.....
        reg = Clase(id....)
        add_in_order(v, reg)
    return p
```

Add in order binario

```
def add_in_order(v, reg):
    n = len(v)
    pos = n
    izq, der = 0, n-1
    while izq <= der:
        c = (izq + der) // 2
        if v[c].algo == reg.algo:
            pos = c
            break
        if reg.algo < v[c].algo:
            der = c - 1
        else:
            izq = c + 1
    if izq > der:
        pos = izq
    v[pos:pos] = [reg]
```

Ficha 16: Arreglos Bidimensionales (Matrices)

Dimensión: es la cantidad de índices que se requieren para entrar a una casilla.



Creación de una matriz con valores fijos:

```
m0 = [ [1, 3, 4], [3, 5, 2], [4, 7, 1] ]  
print('Matriz con valores fijos:', m0)
```

Creación de una matriz:

```
n = filas - m = columnas  
def cargar_matriz(v):  
    mat = [[0] * m for i in range(n)]  
    for op in v:  
        fila = op.tipo - 1  
        columna = op.campo  
        mat[fila][columna] += 1  
  
    return mat
```

Recorrido y carga de una matriz:

```
# un recorrido por filas  
for f in range(len(a)):  
    for c in range(len(a[f])):  
        a [f] [c] = int(input('Valor: '))
```

```
# recorrido por columnas...  
filas = len(a), columnas = len(a[0])  
for c in range(columnas):  
    for f in range(filas):  
        a[f][c] = int(input('Valor: '))
```

len(a): muestra las filas - len(a[f]): muestra las columnas de la fila f

Mostrar matriz:

```
def mostrar_matriz(mat):  
    v1 = int(input("Ingrese un valor: "))  
    v2 = int(input("Ingrese otro valor: "))  
    for f in range(len(mat)):  
        for c in range(len(mat[f])):  
            if v1 <= mat[f][c] and v2 >= mat[f][c]:  
                print("Tipo: ", f, "Campo: ", c, "Total de op: ", mat[f][c])
```

Ficha 18: Registros

- Un **registro** es un conjunto *mutable* de valores que pueden ser de distintos tipos. Cada componente de un registro se denomina **campo** (o **atributo**), y cada campo se designa con un *nombre* o *identificador* (y no con índices).
- Para crear un registro utilizo “**class** Nombre: ”
- La creación y el acceso a los campos, se hace con el operador “punto”: un punto colocado entre el nombre de la variable que apunta al registro y el nombre del campo a crear/acceder. (nombre.apellido)

- Función para la creación de campos:

```
class Empleado:
    def __init__(self, leg, nom, direc, suel, ant):
        self.legajo = leg
        self.nombre = nom
        self.direccion = direc
        self.sueldo = suel
        self.antiguedad = ant

    def test():
        e1 = Empleado(1, 'Juan', 'Calle 1', 10000, 10)
```

La función constructora se invoca en forma implícita al crear un **Empleado**. Y también va en forma implícita el parámetro **self** (no debe ser nombrado al crear el registro!)

Arreglos de registros:

- Si se necesita trabajar con muchos registros a la vez, creo un vector para almacenarlos.
- **v = []**

Cargar arreglo:

```
for i in range(n):
    leg = int(input('Legajo[' + str(i) + ']: '))
    nom = input('Nombre: ')
    pro = float(input('Promedio: '))
    v[i] = Estudiante(leg, nom, pro)
```

Ficha 19: Pilas y colas

Muchas veces el programador necesita organizar su datos en alguna forma que quizás no esté prevista por el lenguaje.

Estructuras nativas: son las estructuras de datos que el lenguaje posee

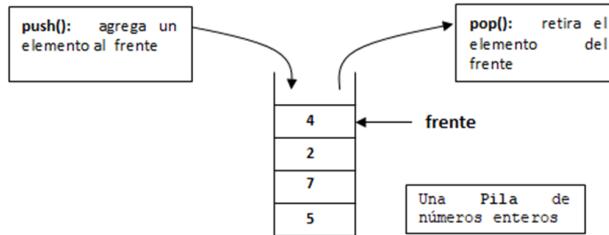
Estructuras abstractas: son las estructuras de datos que diseña el programador.

Implementación del tipo abstracto: Consta de dos partes: captar y aislar los **datos relevantes** (o **abstracción de datos**); e identificar los **procesos aplicables sobre esos datos** (lo que se designa como **abstracción funcional**).

Pilas:

- Son estructuras lineales.
- El primer elemento en entrar, queda al fondo de la pila, y el último que ingresa queda arriba o al frente (y por lo tanto será el primero en salir cuando se retire un valor).

- Se dice que una pila es entonces una estructura tipo **LIFO** (Last In – First Out).



- Lo típico es usar un **vector** para simular la propia pila.

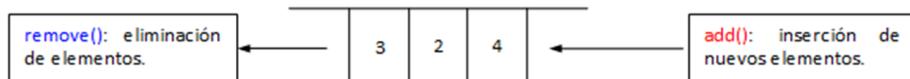
```
# Inserta un elemento x al frente de la pila.
def push(pila, x):
    pila.append(x)
```

```
# Crea y retorna una pila vacía
def init():
    pila = []
    return pila

# Chequea si la pila está vacía.
def is_empty(pila):
    n = len(pila)
    return n == 0
```

Colas:

- Estructura lineal. Lo opuesto a la pila
- El primer elemento en entrar, queda adelante de la cola (y por lo tanto será el primero en salir cuando se retire un valor), y el último que ingresa queda atrás o al fondo. Se dice que una cola es entonces una estructura tipo **FIFO** (First In – First Out).



- Lo típico es también usar un **vector** para representar la propia cola, y funciones como **add()** para agregar un valor al final (la última casilla del vector), o **remove()** para eliminar del valor de adelante (que ahora será la casilla cero).

```
# Crea y retorna una cola vacía
def init():
    cola = []
    return cola

# Chequea si la cola está vacía.
def is_empty(cola):
    n = len(cola)
    return n == 0
```

```
# Inserta un elemento x al fondo de la cola.
def add(cola, x):
    cola.append(x)
```

Ficha 20 y 21: Análisis de algoritmos

- Técnicas para medir la eficiencia de un algoritmo. (Comparar algoritmos que resuelven el mismo problema).
- Los factores que se miden suelen ser el **tiempo de ejecución** y el **consumo de memoria**.
- Se suelen tener en cuenta “el mejor caso” (en búsqueda secuencial, encontrar a la primera), “el peor caso” (encontrar al último o no encontrar) o el “caso promedio” (encontrar en cualquier orden).
- La meta es plantear **fórmulas** generales
- Para estas fórmulas se toma como entrada la cantidad de datos “n” que el algoritmo debe procesar y se intenta detectar la **operación crítica** (aquella que por repetirse muchas veces, hace que el algoritmo demore lo que demora)
- Por ejemplo en ordenamiento por selección simple la operación crítica sería la comparación de dos elementos.
- **Análisis de conteo riguroso/exhaustivo:** fórmula específica para calcular el tiempo de ejecución de un problema.
- **Análisis asintótico:** forma general de variación de la función (cuadrática, lineal, exponencial....). Notación “**O mayúscula**” o “**big O**” .

Conteo exhaustivo:

$$t(n) = \frac{1}{2} n^2 - \frac{1}{2} n$$

Análisis asintótico (notación Big O):

$$t(n) = O(n^2)$$

-

Función	Significado (cuando mide tiempo de ejecución)	Casos típicos
$O(1)$	Orden constante. El tiempo de ejecución es constante, sin importar si crece el volumen de datos.	Acceso directo a un componente de un arreglo.
$O(\log(n))$	Orden logarítmico. Surge típicamente en algoritmos que dividen sucesivamente por dos un lote de datos, desechar una parte y procesando la otra.	Búsqueda binaria.
$O(n)$	Orden lineal. Se da cuando cada uno de los datos debe ser procesado una vez.	Búsqueda secuencial. Recorrido completo de un arreglo.
$O(n * \log(n))$	Surge típicamente en algoritmos que dividen el lote de datos, procesando cada partición sin desechar ninguna, y combinando los resultados al final. No hemos analizado aún algoritmos que respondan a este orden.	Ordenamiento Rápido (Quick Sort).
$O(n^2)$	Orden cuadrático. Típico de algoritmos que combinan dos ciclos de n vueltas cada uno.	Ordenamiento por Selección Directa.
$O(n^3)$	Orden cúbico. Típico de algoritmos que combinan tres ciclos de n repeticiones cada uno. No hemos analizado aún algoritmos que respondan a ese orden	Multiplicación de matrices.
$O(2^n)$	Orden exponencial. Algoritmos que deben explorar una por una todas las posibles combinaciones de soluciones cuando el número de soluciones crece en forma exponencial.	Problema del viajante. Solución recursiva de la Sucesión de Fibonacci.

- de más rápido a más lento
- Consideraciones prácticas:
 1. Tener claro el factor a analizar (tiempo o memoria)
 2. Determinar el tamaño del problema (cuantos datos serán procesados “n”).
 3. Si se necesita un conteo riguroso utilizar formas matemáticas para dar la fórmula.

4. Si necesito un asintótico y ya tiene función rigurosa, pasarla a Big O.
5. Si no tiene función rigurosa, utilizar las ayudas de la tabla para identificar la estructura del algoritmo (un solo ciclo de repeticiones: $t(n) = O(n)$)

Ficha 22 y 25: Archivos

- Conjunto de datos persistentes
- Se almacenan en dispositivos de almacenamiento externos
- Su contenido no se pierde
- Los datos que se graban en un archivo se representan en sistema binario y se utilizan bytes para almacenarlos.
- El tamaño de un archivo es la cantidad total de bytes que contiene el archivo.

Archivos de texto:

- Todos los bytes son interpretados como caracteres y pueden ser visualizados en pantalla.
- Es también un archivo binario
- “read()” - LEER
- “write()” - ESCRIBIR

Método	Acción
m.write(cad)	Graba la cadena <i>cad</i> en el archivo, sin agregar un salto de línea al final de la misma (puede agregarlo el programador).
cad = m.read()	Lee y retorna el contenido completo del archivo, como una única cadena.
cad = m.readline()	Lee una cadena simple desde el archivo. Comienza en la posición del file pointer y termina al encontrar un salto de línea o el final del archivo. Mantiene el salto de línea en la cadena retornada.
cad = m.readlines()	Lee todas las líneas del archivo, y retorna una lista o arreglo con todas ellas (una por cada casilla del arreglo). Mantiene los saltos de línea al final de cada cadena leída.

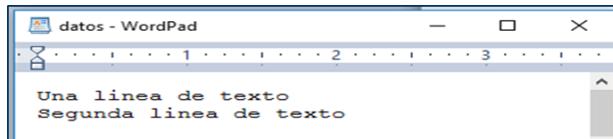
-

```
# 3. Uso de read()...
m = open('datos.txt', 'r')
cad = m.read()
print('Contenido completo:')
print(cad)
m.close()
```

Contenido completo:
Una linea de texto
Segunda linea de texto

-

```
# 2. Uso de write() (con "\n")...
m = open('datos.txt', 'wt')
m.write('Una linea de texto\n')
m.write('Segunda linea de texto')
m.close()
```



- OTROS MÉTODOS:

Método o Atributo	Aplicación
closed	Es un <u>atributo</u> o <u>campo</u> (no un método) cuyo valor es <i>True</i> si el archivo está cerrado.
flush()	Vuelca al archivo los buffers de grabación, si corresponde. Aplicable sólo para archivos abiertos para grabar (no hace nada en caso contrario).
readable()	Retorna <i>True</i> si el archivo está disponible para ser leído.
truncate(size=None)	Trunca el contenido del archivo a una cantidad igual a <i>size</i> bytes. El archivo puede <i>aumentar</i> o <i>disminuir</i> su tamaño.
writable()	Retorna <i>True</i> si el archivo está disponible para ser grabado.
writelines(lines)	Graba el contenido de la lista <i>lines</i> en el archivo, por defecto <i>sin</i> incluir separadores de línea (por lo cual, el programador debe indicar ese separador al final de cada elemento de la lista <i>lines</i> si quiere los separadores).

Archivos binarios:

- Representan información de cualquier tipo (números binarios, caracteres, etc).
- No se asume que representan un carácter
- El programador determina cómo interpretar el contenido
- “**open(nombre del archivo, modo)**” - ABRIR UN ARCHIVO

```
m1 = open("datos.dat", "w")      # la variable m1 es un file object...
m2 = open("c:\\prueba.txt", "r")  # la variable m2 es otro file object...
```

- “**close()**” - CERRAR UN ARCHIVO cuando se deja de utilizar o cuando quiero cambiar de modo de apertura.

```
m1 = open("datos.dat", "wb")    # se crea y se abre el archivo...
m1.close()                     # se cierra el archivo...
m1 = open("datos.dat", "rb")    # se abre el archivo en otro modo...
m1.close()                      # se vuelve a cerrar...
```

- Modos de apertura:

Modo	Significado
r (o rt)	El archivo se abre como <i>archivo de texto en modo de solo lectura</i> (no está permitido grabar). No será creado en caso de no existir previamente. Este es el modo por defecto si se invoca a <i>open()</i> sin especificar modo de apertura alguno.
w (o wt)	El archivo se abre como <i>archivo de texto en modo de solo grabación</i> . Si ya existía, su contenido se perderá y se abrirá vacío. Si el archivo no existía, será creado.
a (o at)	El archivo se abre como <i>archivo de texto en modo de solo append</i> (todas las grabaciones se hacen al final del archivo, <i>preservando</i> su contenido previo si el archivo ya existía). Si no existía, será creado.
r+ (o r+t)	El archivo se abre como <i>archivo de texto en modo de lectura y grabación</i> . El archivo debe existir previamente: no será creado en caso de no existir.
w+ (o w+t)	El archivo se abre como <i>archivo de texto en modo de grabación y lectura</i> . Si ya existía su contenido será eliminado y abierto vacío. Si no existía, será creado.
a+ (o a+t)	El archivo se abre como <i>archivo de texto en modo de lectura y de append</i> (todas las <i>grabaciones</i> se hacen al final del archivo, preservando su contenido previo). Si no existía, será creado.
rb	El archivo se abre como <i>archivo binario en modo de sólo lectura</i> . No será creado en caso de no existir previamente.
wb	El archivo se abre como <i>archivo binario en modo de sólo grabación</i> . Si ya existía su contenido será eliminado. Si no existía, será creado.
ab	El archivo se abre como <i>archivo binario en modo de sólo append</i> (todas las grabaciones se hacen al final del archivo, preservando su contenido previo si el archivo ya existía). Si no existía, será creado.
r+b	El archivo se abre como <i>archivo binario en modo de lectura y grabación</i> . El archivo debe existir previamente: no será creado en caso de no existir.
w+b	El archivo se abre como <i>archivo binario en modo de grabación y lectura</i> . Si ya existía su contenido será eliminado. Si no existía, será creado.
a+b	El archivo se abre como <i>archivo binario en modo de lectura y de append</i> (todas las <i>grabaciones</i> se hacen al final del archivo, preservando su contenido previo si ya existía). Si no existía, será creado.

- Import pickle:
- “**pickle.dump(x (a grabar), m (donde se graban))**” - GRABAR

```
import pickle
edad, nombre = 23, "Ana"
m1 = open("datos.dat", "wb")      # se crea y se abre el archivo para grabar...
pickle.dump(edad, m1)            # se graba el número 23 en el archivo...
pickle.dump(nombre, m1)          # se graba la cadena "Ana" en el archivo...
m1.close()                      # se cierra el archivo...
```

- “**pickle.load(m)**” - LEER

```
import pickle
m1 = open("datos.dat", "rb")      # se abre el archivo para leer...
edad = pickle.load(m1)            # se lee el número 23 desde el archivo...
nombre = pickle.load(m1)          # se lee la cadena "Ana" desde el archivo...
m1.close()                      # se cierra el archivo...
print("Nombre:", nombre, "- Edad:", edad)
```

Recorrido secuencial y acceso directo:

- Cada byte tiene un índice que lo identifica.
- El número total de bytes coincide con el número del 1er byte fuera del arreglo.
- “file pointer” - variable que almacena el número de byte en el que está posicionado el archivo.
- Si la función load() intenta leer bytes ubicados desp del final del archivo, da error el programa.
- Para el control de la lectura del archivo se utiliza (es decir, que el file pointer no se pase) :

```
import pickle
import os.path
t = os.path.getsize("datos.dat")    # se toma el tamaño en bytes...
m1 = open("datos.dat", "rb")        # se abre el archivo para leer...
pos = m1.tell()                   # se toma el valor del file pointer...
print("Valor del file pointer:", pos)
m1.close()
```

- **os.path.getsize()** - RETORNA EL TAMAÑO DE BYTES DEL ARCHIVO
- **tell()** - RETORNA EL VALOR DEL FILE POINTER

```
m = open(fd, 'rb')
t = os.path.getsize(fd)
while m.tell() < t:
    lib = pickle.load(m)
    display(lib)
    m.close()
```

El tamaño del archivo equivale al número del primer byte que está *frente* del archivo (y **getsize()** retorna ese número).

Si el **file pointer** en alguna vuelta llega al final del archivo, la condición será *False* y el ciclo se detendrá.

Reposición del File pointer:

```
m1.seek(offset, from_what)
```

Constante	Valor	Significado
io.SEEK_SET	0	Reposicionar comenzando desde el principio del archivo. El valor del primer parámetro (offset) puede ser 0 o positivo (pero no negativo).
io.SEEK_CUR	1	Reposicionar comenzando desde la posición actual del puntero de registro activo. El valor de offset puede ser entonces negativo, 0 o positivo.
io.SEEK_END	2	Reposicionar comenzando desde el final del archivo. El valor de offset típicamente es negativo, aunque puede ser 0 o positivo.

Ficha 23: Gestión ABM

Gestión ABM es el nombre abreviado que suele darse a un programa completo cuyo objetivo es el de permitir agregar registros a un archivo (**alta** de registros), eliminar registros (**baja** de registros) o cambiar valores de registros que ya están en el archivo (**modificación** de registros).

Baja de registros:

El proceso de eliminación o baja de un registro en un archivo consiste en buscar el registro que se quiere eliminar, y proceder a removerlo del archivo.

Alta de registros:

La operación de agregar un nuevo registro se designa como **alta** de un registro. La lógica del proceso general depende de si se aceptan registros "repetidos" (por ejemplo, estudiantes con el mismo legajo) o no. En nuestro modelo supondremos que NO se aceptan repeticiones.

En ese caso, el proceso de alta consiste en buscar en el archivo el legajo (o valor que sea que se use para identificar a un registro en forma primaria). Si no se encuentra ya cargado en el archivo un registro con ese identificador o clave, cargar el resto de los datos, crear un registro nuevo con esos datos, poner el campo activo de ese registro en True, y agregar el registro al final del archivo.

Modificación de registros:

En principio se trata de buscar el registro que se quiere modificar, y una vez encontrado ofrecer al usuario un menú para cargar los nuevos datos de los campos que quiera cambiar.

Lo típico es que NO se permita el cambio del campo que sirve como clave o identificador del registro.

Ficha 26: Estrategias de resolución de problemas: Recursividad

RECURSIVIDAD:

- Se dice que una definición es recursiva si el objeto que se está definiendo aparece a su vez en la propia definición.
- Es la propiedad que permite que un proceso se invoque a sí mismo una o más veces como parte de la solución.

- Ej: Una **frase** es un conjunto de palabras que puede estar vacía o bien puede contener una palabra seguida a su vez de otra **frase**.
- Una definición recursiva bien planteada, debe cumplir con 2 requisitos
 - 1- La definición debe agregar conocimiento del objeto definido (Una frase es una frase, esto está mal)
 - 2- La definición debe evitar la recursión infinita, que surge cuando no incluye elementos que permitan cerrarla lógicamente.

PROGRAMACIÓN RECURSIVA:

Prácticamente todos los lenguajes de programación modernos soportan la recursividad, a través del planteo de **funciones recursivas**: funciones que incluyen **una o más invocaciones a sí mismas** en sus bloques de acciones.

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

¡Correcto!: agrega conocimiento (un factorial es un producto), y el proceso recursivo no es infinito (si n es cero, la función termina y retorna 1, sin activar un nuevo proceso recursivo...) ☺

SEGUIMIENTO DE LA RECUSIÓN:

- **Stack Segment** (segmento pila): bloque de memoria en el cual se almacenan las variables locales y las direcciones de retorno de las funciones
- Si una función a su vez invoca a otra, esa otra también recibe un bloque en el **Stack**, que se ubicará encima del bloque anterior, en modo LIFO.
- La última función en invocarse estará en la cima del stack.
- **Stack Overflow**: el stack es un segmento de poco tamaño, una secuencia muy larga de funciones que se invoquen entre ellas, podría desbordarlo e interrumpir la secuencia.

CONCLUSIONES:

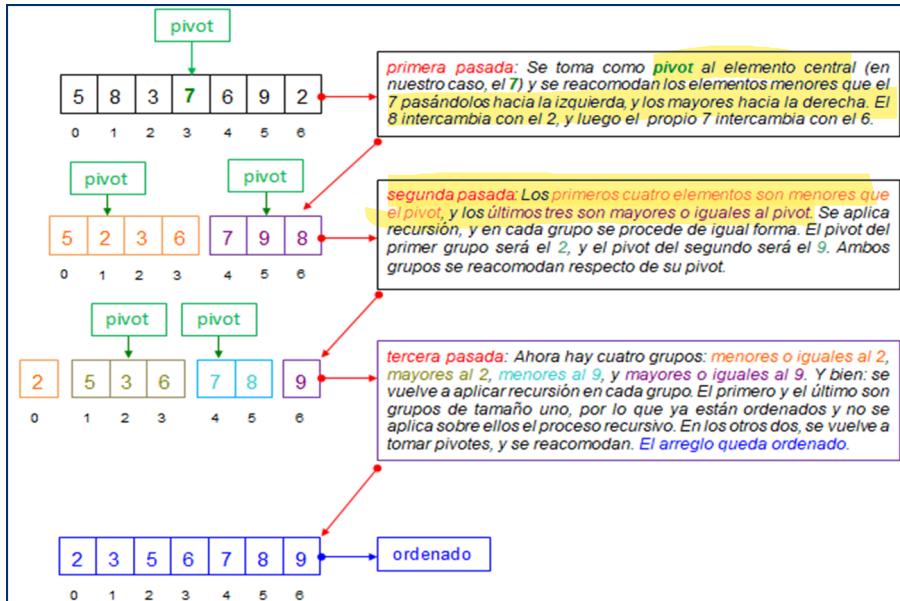
- La recursividad es muy útil para el planteo de algoritmos, pero debe ser usada con cuidado y con conocimiento adecuado en cuanto a la forma de estimar el uso de recursos de tiempo y memoria.
- Generalmente no es conveniente aplicar recursividad.

Ficha 27: Estrategias de resolución de problemas: Divide y vencerás

QUICKSORT:

- Algoritmo de ordenamiento de tiempo promedio $O(n \log n)$.
- La idea es tomar un elemento a modo de **pivot** (por ejemplo, el elemento central) y luego recorrer el vector desde cada extremo, pasando a la parte derecha del vector los elementos mayores que el pivot, y a la parte izquierda los elementos menores. Esta operación general se designa como "**pivotear**" el arreglo.

- Al terminar el pivoteo, el vector tendrá dos particiones (que en **promedio** puede esperarse que sean de tamaño $n/2$ cada una)... y se aplica entonces recursivamente la misma idea en cada una de esas particiones. El proceso se repite recursivamente, hasta que cada partición sea de tamaño 1, y en ese momento el vector estará ordenado...



ESTRATEGIA DIVIDE Y VENCERÁS:

- La estrategia aplicada en el quicksort se conoce como DyV.
- "n" datos se procesan y se dividen. Cada partición se vuelve a procesar y a dividir. Cuando ya no es posible seguir dividiendo, se detiene el proceso y la unión de todas las particiones es el resultado general buscado(vencerás).
- Caso promedio: $O(n \log n)$ - todas las particiones tienen tamaño similar
- Peor caso: $O(n^2)$ - El pivot que se toma es el menor o el mayor de esa partición

MEDIANA DE TRES:

- Estrategia para evitar el peor caso
- Toma 3 pivotes, el primer elemento, el central y el último.
- Ordena esos tres y tomar el pivot que quedó en la posición central

- Al tomar tres elementos de la partición y ordenarlos, se garantiza que no se tomará nunca el menor ni el mayor de esa partición (a menos que todos sean iguales...)

- Si la partición tiene n elementos, este proceso toma 3 de esos n , y siempre tres, por lo que el tiempo para ejecutarse **no depende de n** . No importa cuán grande sea n , el tiempo **de este proceso** será siempre el que tome hacer estas tres condiciones, por lo que queda $t(n) = O(1)$ (constante).

Ficha 28: Estrategias de resolución de problemas: Ávidos, dinámica y Backtracking

1. Fuerza Bruta: Consiste en explorar y aplicar sistemáticamente una por una, todas y cada una de las posibles combinaciones de solución para el problema dado. Ejemplo, Ordenamiento de **Selección Directa**. Son simples de comprender e implementar (de hecho, un algoritmo de fuerza bruta suele ser lo primero que se le ocurre a un programador) pero por otra parte suelen ser muy ineficientes ya que la esencia del planteo consiste justamente en no ahorrar pasos.

2. Recursión: Es la propiedad que permite que un proceso se invoque a sí mismo una o más veces como parte de la solución. muy claro para entender e implementar, pero utiliza mucha memoria adicional en el segmento de stack del computador (y algo de tiempo extra para gestionar el apilamiento en ese stack). **En muchos problemas ese costo no es aceptable (factorial, Fibonacci)**, pero en otros la recursión permite el planteos claros y concisos, frente a planteos no recursivos extensos, intrincados y sumamente difíciles de mantener frente a cambios en los requerimientos. Casos en que la recursión es apropiada, se dan en algoritmos para generar **gráficas fractales** o en algoritmos de **inserción y borrado en árboles de búsqueda equilibrados**.

3. Backtracking (o Vuelta Atrás): Es una técnica que permite explorar en forma incremental un conjunto de potenciales soluciones parciales a un problema, de forma que si se detecta que una solución parcial no puede ser una solución válida, se la descarta junto a todas las candidatas que podrían haberse propuesto a partir de ella (vuelta atrás). Típicamente, se implementa mediante **recursión** generando un árbol de invocaciones recursivas en el que cada nodo constituye una solución parcial. Cuando es aplicable, el backtracking suele ser más eficiente que la enumeración por fuerza bruta de todas las soluciones, ya que con backtracking pueden eliminarse muchas soluciones sin tener que analizarlas. Se usa en problemas que admiten la idea de solución parcial, siempre y cuando se pueda comprobar en forma aceptablemente rápida si una solución parcial es válida o no. Ejemplos: **Problema de las Ocho Reinas** (ubicar 8 reinas en un tablero de ajedrez de tal forma que ninguna de ellas se ataquen entre si), **Palabras Cruzadas**, **Sudoku** y en general, en **problemas de optimización combinatoria (encontrar la mejor combinación)**.

4. Greedy Algorithms (o Algoritmos Ávidos): Un **algoritmo ávido** es aquel que aplica una regla intuitivamente válida en cada paso local del proceso, con la esperanza de obtener finalmente una solución global óptima que resuelva el problema original. La ventaja de un algoritmo ávido (cuando es correcto) es que por lo general lleva a una solución simple de entender, muy directa de implementar y razonablemente eficiente. Pero la desventaja es que como no siempre lleva a una solución correcta, se debe realizar una demostración de la validez del algoritmo que podría no ser sencilla de hacer.

Ejemplos: **Algoritmos de Prim y de Kruskal para el Árbol de Expansión Mínimo de un Grafo, o el Algoritmo de Dijkstra para el Camino más Corto entre Nodos de un Grafo.**

Ej: en el problema de la moneda, elijo sistemáticamente la moneda de mayor valor, devolver tantas de ella como se pueda y continuar así hasta cubrir el valor pedido.

`coins = [1, 5, 10, 25]` y el valor a cambiar es $x = 63$, elegimos dos veces la moneda de 25, una vez la de 10 y tres la de 1, lo cual da un total de 6 monedas (que efectivamente, en el contexto planteado es la mínima cantidad de monedas para llegar a cubrir $x = 63$).

El problema con esta idea, es que si la base de monedas contiene algún valor "extraño", entonces la estrategia ávida **podría no funcionar**.

Supongamos que el conjunto `coins` fuese de la forma `coins = [1, 5, 10, 23, 50]` y queremos cambio de $x = 69$. La estrategia ávida calcularía 7 monedas (una de 50, una de 10, una de 5 y cuatro de 1)... pero la solución correcta es 3 (tres monedas de 23...)

En este caso, algoritmo ávido no funciona, y el conjunto `coins` mostrado **NO ES** entonces una base canónica.

5. Divide y Vencerás: Consiste en tratar de dividir el lote de datos en dos o más subconjuntos de tamaños aproximadamente iguales, procesar cada subconjunto por separado y finalmente unir los resultados para obtener la solución final. Normalmente, se aplica recursión para procesar a cada subconjunto y el proceso total podrá ser más o menos eficiente en cuanto a tiempo de ejecución dependiendo de tres factores: la cantidad de invocaciones recursivas que se hagan, el factor de achicamiento del lote de datos (por cuanto se divide al lote en cada pasada) y el tiempo que lleve procesar en forma separada a un subconjunto.

Ejemplos: **Algoritmos Quicksort y Mergesort para ordenamiento de arreglos.**

6. Programación Dinámica: Esta técnica sugiere almacenar en una tabla las soluciones obtenidas previamente para los subproblemas que pudiera tener un problema mayor, de forma que cada subproblema se resuelva sólo una vez y luego simplemente se obtengan sus soluciones consultando la tabla si esos subproblemas volvieran a presentarse. Esto tiene mucho sentido: en muchos planteos originalmente suele ocurrir que al dividir un problema en subproblemas se observe que varios de estos últimos se repiten más de una vez, con la consecuente pérdida de tiempo que implicaría el tener que volver a resolverlos.

Ejemplos: **Problema de Alineación de Secuencias** (encontrar la mínima cantidad de cambios que se requieren para que una secuencia de entrada se convierta en otra).

Ej: En el problema de las monedas, se usa una *tabla* (un arreglo) para guardar los resultados para todos los valores **menores a x** . Se va progresando en el cálculo del cambio óptimo para cada uno de esos valores, y cada vez que se tenga un nuevo caso resuelto, se guarda su resultado en la tabla. Esta estrategia de solución basada en tablas de resultados previos, se conoce como programación dinámica.

7. Algoritmos Randomizados: Para algunos problemas se suele intentar plantear algoritmos que ya no sean deterministas (la misma entrada produce la misma salida, sino **randomizados o de base aleatoria** (la misma entrada podría producir salidas diferentes ya que el siguiente paso a aplicar surge de algún tipo de selección aleatoria). Y está claro que si interviene el azar, **entonces es posible que el algoritmo no llegue eventualmente a una solución correcta**. La idea es que aplicar esta técnica debe implicar alguna ganancia en eficiencia, se debe poder calcular la probabilidad de que algoritmo falle y diseñarlo de forma esa probabilidad sea realmente muy baja, y finalmente sería deseable que el algoritmo sea simple de implementar).

Ejemplo: **Algoritmo de Karger para el Problema del Corte Mínimo en un Grafo.**