

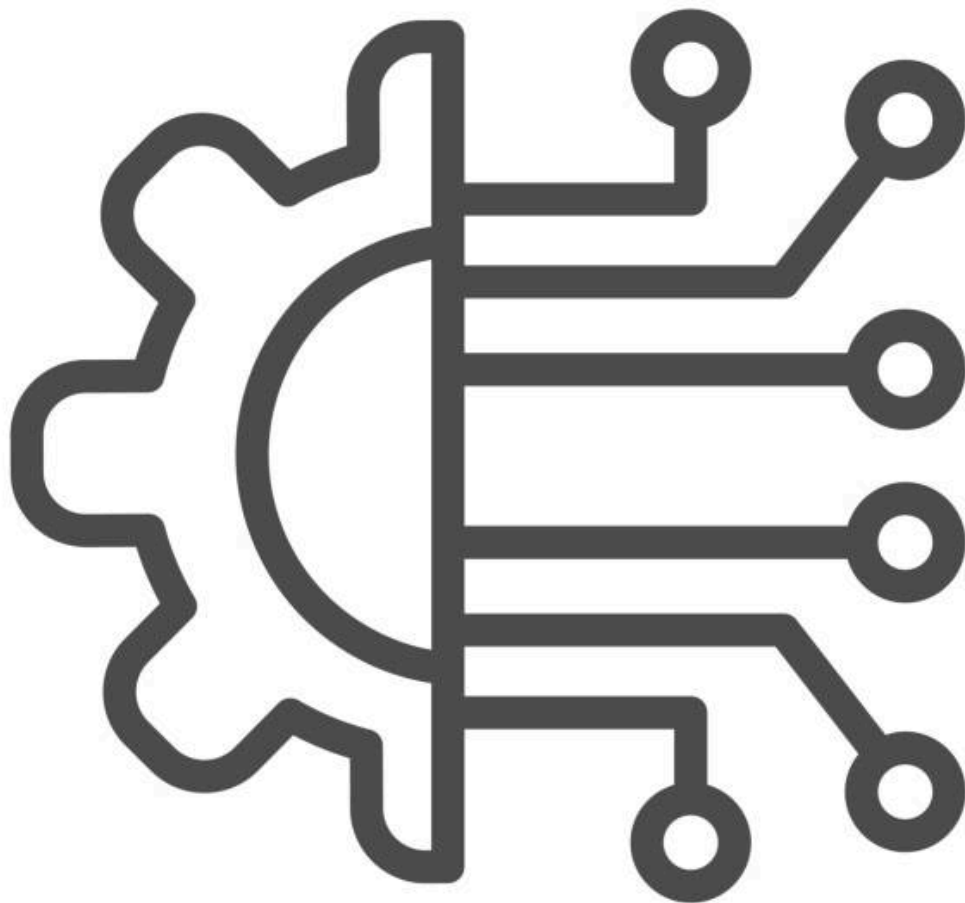
Untitled

Portada



211 DISEÑO ESTRUCTURADO DE ALGORITMOS 25-1

PROF.-DANIEL GUILLERMO CONRADO MOGUEL



Datos del Estudiante

- Nombre: Jesus Uriel Santana Oliva
- Grado : 1 -B
- Institución: Tec Playacar
- Fecha: Thursday, November 28th, 2024
- Ubicacion : Playa del Carmen
-

****Actividad**









- ✓ ~~Revisión de formato APA.~~ ✓ ~~2024-11-28~~
- ✓ ~~Finalizar la bibliografía.~~ ✓ ~~2024-11-28~~
- ✓ ~~Verificar coherencia en la argumentación.~~ ✓ ~~2024-11-28~~
- ✓ ~~Insertar gráficos relevantes.~~ ✓ ~~2024-11-28~~

✓ Terminada

Tarea Terminada


Investigación sobre Estructuras Algorítmicas y su Aplicación

Introducción


Las estructuras algorítmicas son el  cimiento de la  programación y el  diseño de soluciones. Estas  estructuras definen cómo se organiza un  algoritmo para resolver un  problema de manera eficiente y efectiva. La elección de una estructura algorítmica adecuada puede ser crucial para optimizar recursos como  tiempo y  memoria.

Clasificación de las Estructuras Algorítmicas

1 Estructuras Secuenciales

Las estructuras secuenciales representan el flujo  lineal de ejecución, donde las instrucciones se procesan en el orden en que se presentan.

Características:

- ✓ Simples y fáciles de implementar.
-  Ideales para problemas sin decisiones o iteraciones.

 Ejemplo:

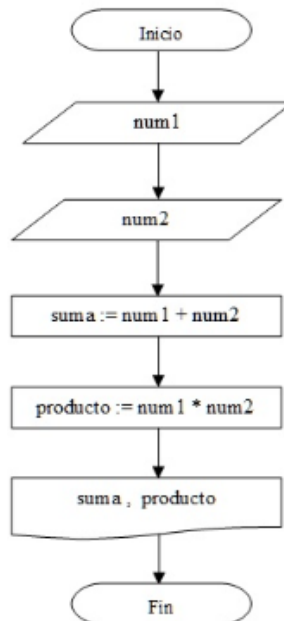
```

arreglo = [1, 2, 3, 4, 5]
suma = 0
for elemento in arreglo:
    suma += elemento
print(suma)

```

Estructura Secuenciales Aplicado: ✓

Estructuras Secuenciales



Tenemos dos entradas num1 y num2, dos operaciones: calcular la suma y el producto de los valores ingresados y dos salidas, que son los resultados de la suma y el producto de los valores ingresados. En el símbolo de impresión podemos indicar una o más salidas, eso queda a criterio del programador, lo mismo para indicar las entradas por teclado.

```

program Proyecto4;

{$APPTYPE CONSOLE}

var
    num1, num2: Integer;
    suma, producto: Integer;

begin
    Write('Ingrese el primer valor:');
    ReadLn(num1);
    Write('Ingrese el segundo valor:');
    ReadLn(num2);
    suma := num1 + num2;
    producto := num1 * num2;
    WriteLn('La suma de los dos valores ingresados es:', suma);
    WriteLn('El producto de los dos valores ingresados es:', producto);
    ReadLn;

```

end.

Previo al bloque begin end debemos definir todas las variables que hemos definido en nuestro diagrama de flujo. Podemos definir cada variable en una línea o agruparlas como hemos hecho en este problema para que se lea mejor el programa:

```
var
  num1, num2: Integer;
  suma, producto: Integer;
```

Dentro del bloque begin end es donde disponemos nuestro algoritmo.

Mostramos un mensaje por pantalla indicando al operador que cargue el primer valor empleando el procedimiento Write:

```
Write('Ingrese el primer valor:');
```

Para la entrada de datos por teclado utilizamos el procedimiento ReadLn donde obligatoriamente indicamos el nombre de la variable a cargar :

```
ReadLn(num1);
```

Los mismos pasos efectuamos para la carga del segundo número:

```
Write('Ingrese el segundo valor:');
ReadLn(num2);
```

Las operaciones las codificamos en forma idéntica a como lo indicamos en el diagrama de flujo. Recordar que siempre una operación debe tener el operador de asignación ":= "

```
suma := num1 + num2;
producto := num1 * num2;
```

Podemos utilizar el procedimiento Write y WriteLn para mostrar mensajes y contenidos de variables simplemente separando cada una por una coma:



```
WriteLn('La suma de los dos valores ingresados es:', suma);
WriteLn('El producto de los dos valores ingresados es:', producto);
```

La diferencia entre Write y WriteLn es que este último luego de imprimir deja el cursor en la siguiente línea, es decir hace un salto de línea.



Siempre al final del programa llamamos al procedimiento ReadLn para que el operador presione una tecla y finalice recién el programa:

2 Estructuras de Selección

Estas estructuras permiten tomar decisiones en base a condiciones, usando

 if-else o  switch-case.

☀ Características:

-  Permiten el control del flujo del programa.
-  Fundamentales para evaluar varias condiciones.

Ejemplo:

```
numero = int(input("Ingrese un número: "))
if numero > 0:
    print("El número es positivo.")
elif numero < 0:
    print("El número es negativo.")
else:
    print("El número es cero.")
```

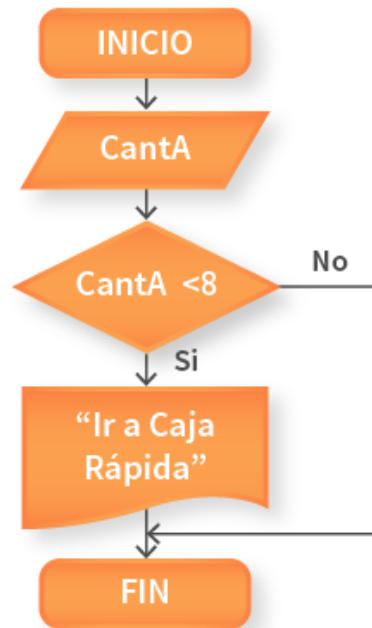
Construya un diagrama de flujo tal que dado como dato la cantidad de artículos a comprar, escriba “Ir a Caja Rápida” en caso de que este número sea menor a 8 artículos.

Datos de entrada:

1. Cantidad de artículos a comprar.
2. (Definición de variable) CantA:
3. Variable de tipo real que representa la cantidad de artículos a comprar.
4. Resultado esperado o datos de salida:
5. El enunciado “Ir a Caja Rápida” en caso de que el número de artículos a comprar sea menor a 8.
6. (Definición de variable)
7. No se requieren variables se imprimirá una cadena de caracteres.

Proceso: (Algoritmo)

1. Inicio
2. Leer la cantidad de artículos a comprar. CantA.
3. Evaluar si la cantidad de artículos a comprar es menor a 8.
 1. Si la cantidad de artículos a comprar es menor a 8, imprimir “Ir a la Caja Rápida”.
4. Fin



```

Inicio
  Leer CantA
  Si CantA<8 entonces
    Imprimir "Ir a Caja Rápida"
  Fin Si

Fin
  
```

En la siguiente tabla observamos el seguimiento del diagrama de flujo para diferentes corridas. Cabe aclarar que una corrida es una ejecución del programa.

NUMERO DE CORRIDA	DATOS	RESULTADO
	CantA	
1	5	"Ir a la Caja Rápida"
2	7	"Ir a la Caja Rápida"
3	14	
4	8	
5	1	"Ir a la Caja Rápida"

La parte sombreada expresa los valores que se imprimen.

La casilla en blanco significa que no se imprime ningún valor. Ya que no se cumplió con la condición.

3 Estructuras Repetitivas (Bucles)

Permiten ejecutar un bloque de código varias veces, ya sea con `for` o `while`.

Estructuras Repetitivas (Bucles)

Un bucle o lazo (Loop) es un segmento de un algoritmo o programa, cuya instrucciones se repiten un número determinado de veces mientras se cumple una determinada condición (existe o es verdadera la condición). SE debe establecer un mecanismo para determinar las tareas repetitivas. Este mecanismo es una condición que puede ser verdadera o falsa y que se comprueba una vez a cada paso o iteración del bucle (total de instrucciones que se repiten en el bucle).

Un bucle consta de tres partes:

- Decisión,
- cuerpo del bucle,
- salida del bucle.

El bucle de la siguiente figura es infinito, ya que las instrucciones (1), (2) y (3) se ejecutan indefinidamente, pues no existe salida del bucle, al no cumplirse una determinada condición.

☀ Características:

- 🔄 Útiles para procesamiento repetitivo.
- ⚙ Combinables con estructuras de selección para más flexibilidad.

💻 Ejemplo:

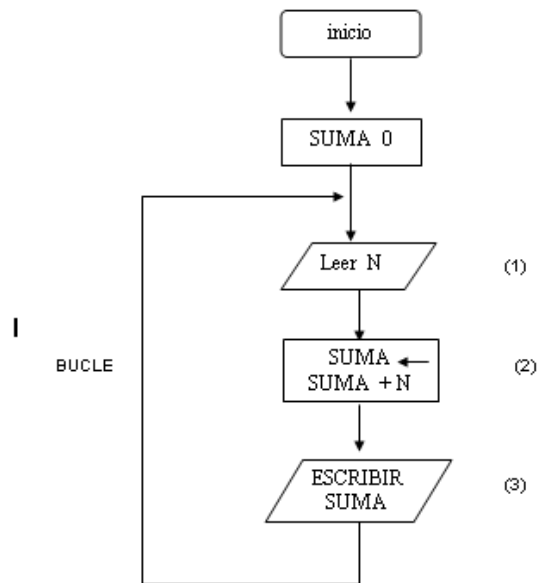
```
for i in range(1, 11):  
    print(i)
```

OBJETIVO ESPECÍFICO

Emplear las estructuras repetitivas para seleccionar la más adecuada y aplicarlas en la solución de problemas utilizando algoritmos específicos.

CONTENIDO

- Ciclos iterativos
- Contadores y acumuladores
- Ciclos de la estructura tipo FOR - NEXT - END FOR
- Ciclos de la estructura tipo DO - UNTIL
- Ciclos de la estructura tipo WHILE - END WHILE



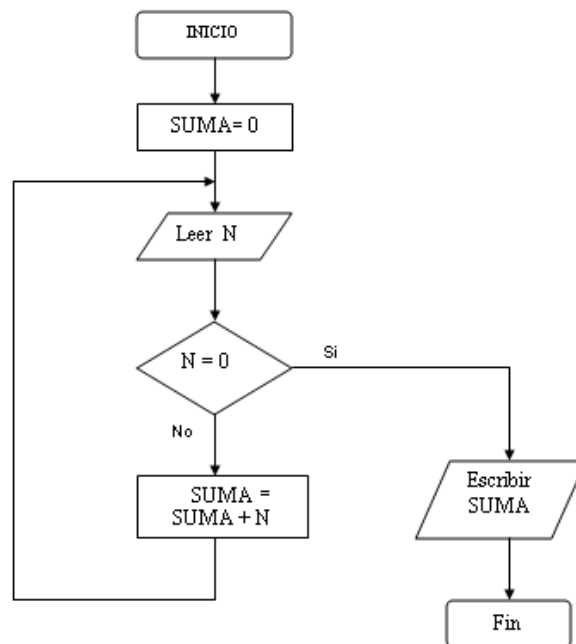
Si tras la lectura de la variable N se coloca una condición, el bucle dejará de ser infinito y tendrá fin cuando la condición sea verdadera.

El diagrama de flujo escrito en pseudo código es aproximadamente el siguiente:

```

Inicio
  SUMA 0
  1: leer N
  Si N = 0 entonces
    Escribir SUMA
    Ir_a fin
  Si_no
    Suma suma + N
  FIN_SI
  IR_A 1
FIN

```

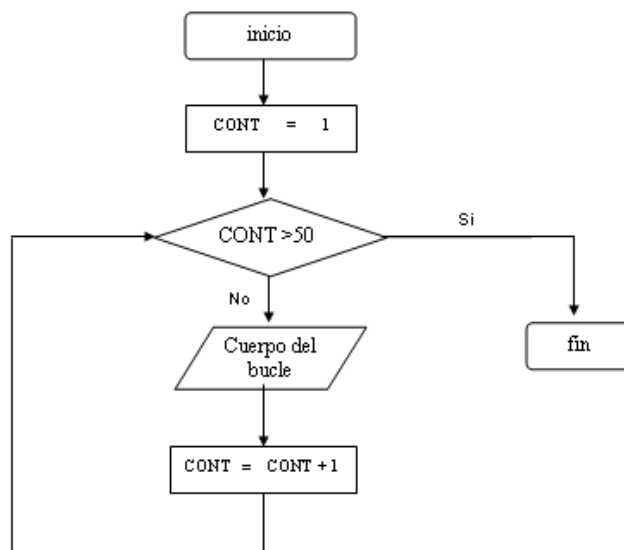
BUCLES ANIDADOS

Un bucle puede anidarse dentro de otro como se vio en clase con los condicionales anidados (un si fin_si dentro de otro si Fin_si)

Contadores

Un contador es una variable cuyo valor se incrementa o decremento en una cantidad constante en cada vuelta.

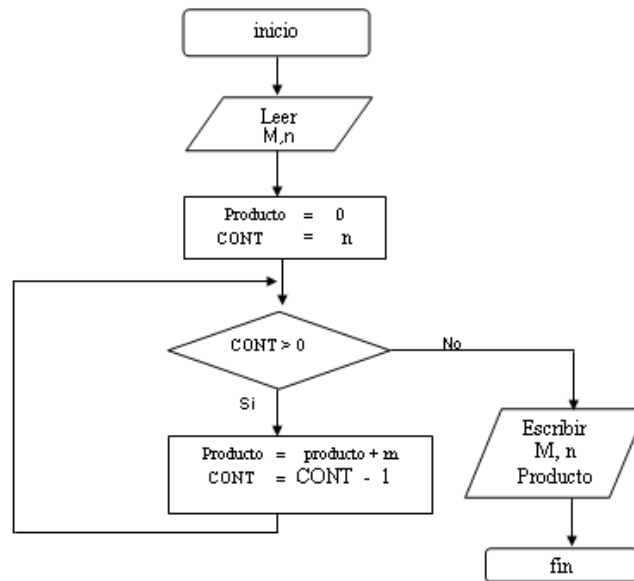
La siguiente figura



contar del 1 al 50

presenta un diagrama de flujo para un algoritmo que se desea repetir 50 veces; el contador se representa en este ejemplo con la variable CONT. La instrucción que representa a un contador es la asignación $CONT = CONT + 1$.

La siguiente figura



Decrementar desde N hasta 0

es otro ejemplo de un diagrama de flujo con contador; es este caso, negativo. Se dice también descontar.

El contador puede ser positivo (incrementos, uno en uno) o negativo (decrementos, uno en uno).

Importante acerca de incrementos y decrementos:

En la primera Figura el contador cuenta desde 1 al 50 y deja de contar cuando la variable CONT toma el valor 51 y se termina el bucle.

En la segunda Figura el contador cuenta negativamente, o lo que es lo mismo, descuenta o decrementa; comienza a contar en n y se decrementando hasta llegar a cero, en cuyo caso se termina el bucle y se realiza la acción escribir.

Como se vio anteriormente la condición permite terminar el bucle cuando ésta es verdadera (si)

ahora veremos las estructuras repetitivas que se usarán en el curso de algoritmos:

Tipos de estructuras repetitivas

Mientras Condicion Hacer

Fin_mientras

Desde Variable=inicio Hasta Variable=Final hacer

Fin_desde

(La anterior también puede ser usada con la palabra PARA en vez de DESDE

ejemplo:

PARA Variable=inicio Hasta Variable=Final hacer

Fin_PARA)

Repetir

Hemos visto que las Estructura repetitivas son aquellas en las que especialmente se diseña para todas aquellas

aplicaciones en las cuales una operación o conjunto de ellas deben repetirse muchas veces.

asi los Bucles (lazos o L00Ps) Son estructuras que repiten una secuencia de instrucciones un numero determinado de veces.

Interacción: Es el hecho de repetir la ejecución de una secuencia de acciones; en otras palabras el algoritmo repite muchas veces las acciones.

Al utilizar un bucle para sumar una lista de números, se necesita saber cuantos números se han de sumar, para poder detenerlo en el momento preciso; las dos principales preguntas ha realizarse en el diseño de un bucle son:

¿Que contiene el bucle? y ¿Cuántas veces se debe repetir?

Casos Generales de Estructuras repetitivas

1. La condición de Salida del bucle se realiza al principio del bucle (estructura mientras) también llamada PRE-CONDICIONAL
2. La condición de Salida se origina al final del bucle; el bucle se verifica hasta que se verifique una cierta condición

también llamada POST-CONDICIONAL (estructura Repetir Hasta).

3. La condición de salida se realiza con un contador que cuente el numero de interacciones. (i es un contador que cuenta desde el valor inicial (vi.) hasta el valor final (vf) con los incrementos que se consideren.) (estructura DESDE o PARA)

Estructura mientras ("while")

Es aquella en que el cuerpo del bucle se repite mientras se cumple una determinada condición. Cuando se ejecuta la acción mientras, la primera cosa que sucede es que se evalúa la condición (una expresión booleana que devuelve Verdadero o Falso), si se evalúa falsa ninguna acción se tomara y el programa en la siguiente instrucción del bucle; si la expresión booleana es verdadera, entonces se ejecuta el cuerpo del bucle, depuse del cual se evalúa de nuevo la expresión booleana.

Esta expresión booleana se repite una y otra vez mientras la expresión booleana (condición) sea verdadera

Estructura repetir ("repeat").

Si el valor de la expresión booleana es inicialmente falso, el cuerpo del Bucle no se ejecutara, por ello se necesitan de otros tipos de estructuras. Dicha estructura se ejecuta hasta que cumpla una condición determinada que se comprueba hasta el final del bucle

Diferencias entre las estructuras mientras y repetir

- La estructura **mientras** termina cuando la condición es falsa, mientras que **repetir** termina cuando la condición es verdadera.
- En la estructura **repetir** el cuerpo del bucle se ejecuta siempre al menos una sola vez; por el contrario **mientras** es mas general y permite la posibilidad de que el bucle pueda no ser ejecutado.
- Para usar la estructura **repetir** debe estar seguro de que el cuerpo del bucle se repetirá al menos una sola vez.

Estructura desde/para ("for").

Son el numero total de veces que se desea ejecutar las acciones del Bucle (numero de interacciones fijo), este ejecuta las acciones del cuerpo o del Bucle un numero especifico de veces y de modo automático controla el numero de Interacciones o pasos a través del cuerpo del bucle.

Ejemplos con Mientras

Forma de Uso

```
Mientras condicion hacer
```

```
    acción 1
```

```
    acción 2
```

```
    acción 3
```

```
    ....
```

```
    acción n
```

```
Fin_Mientras
```

1.- Hacer un programa que cuente del uno al 10

```
Inicio
```

```
    x= 1
```

```
    Mientras x ≤ 10 hacer
```

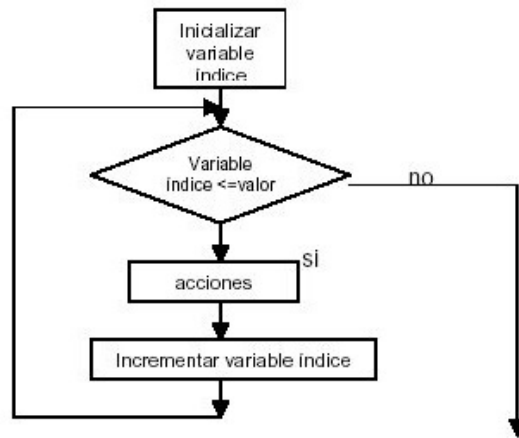
```
        Escribir x
```

```
        x= x + 1
```

```
    Fin_Mientras
```

```
Final
```

Esta estructura implica dividir un 🧩 problema en subproblemas, resolverlos de forma independiente y combinar las soluciones.



☀ Características:

- ⚡ Eficiente para problemas descompuestos recursivamente.
- 📦 Requiere diseñar algoritmos que equilibren la división y la combinación.

💻 Ejemplo:

```
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        izquierda = arr[:mid]
        derecha = arr[mid:]

        merge_sort(izquierda)
        merge_sort(derecha)

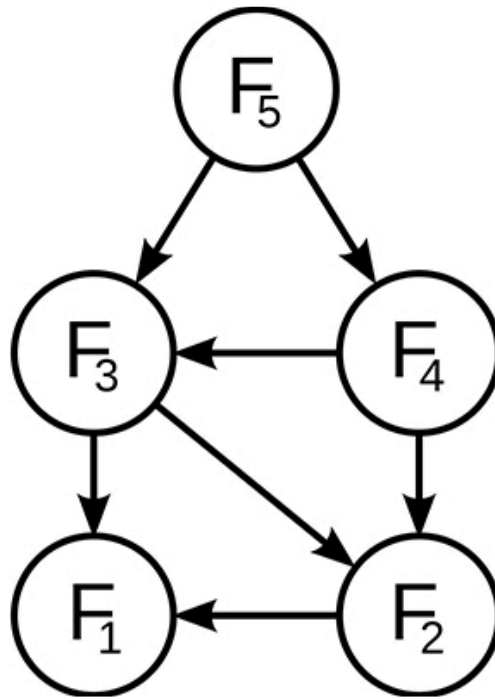
    i = j = k = 0
    while i < len(izquierda) and j < len(derecha):
        if izquierda[i] < derecha[j]:
            arr[k] = izquierda[i]
            i += 1
        else:
            arr[k] = derecha[j]
            j += 1
        k += 1

    while i < len(izquierda):
        arr[k] = izquierda[i]
        i += 1
        k += 1

    while j < len(derecha):
        arr[k] = derecha[j]
        j += 1
        k += 1
```

5 Programación Dinámica

Se basa en resolver problemas dividiéndolos en subproblemas y reutilizar soluciones calculadas.



☀ Características:

- ⚙ Eficiente para problemas con subproblemas superpuestos.
- 📊 Común en optimización.

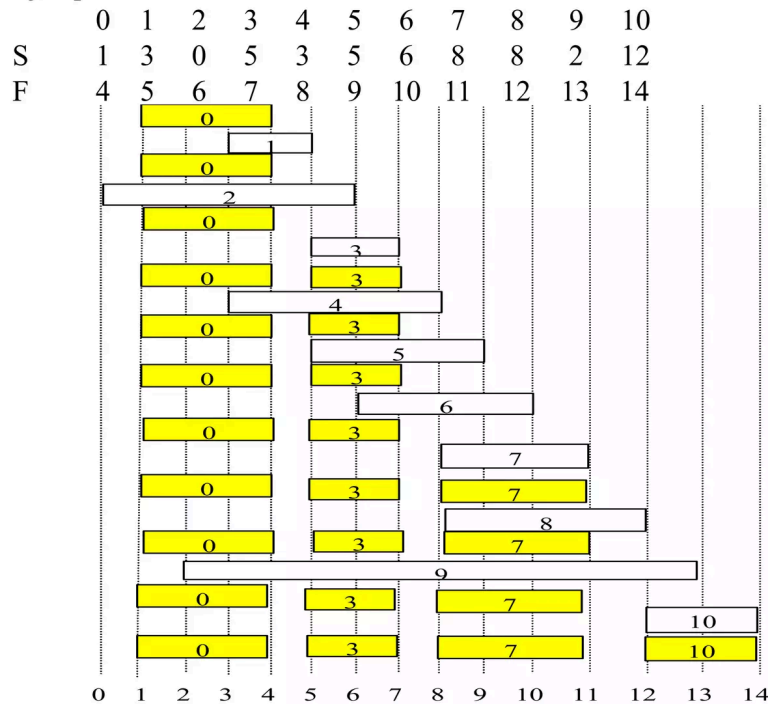
💻 Ejemplo:

```
fib_cache = {}  
def fibonacci(n):  
    if n in fib_cache:  
        return fib_cache[n]  
    if n ≤ 2:  
        return 1  
    fib_cache[n] = fibonacci(n-1) + fibonacci(n-2)  
    return fib_cache[n]
```

6 Algoritmos Greedy (Avaros)

Ejemplo

- Se puede demostrar que la solución aquí propuesta es óptima.
- La estrategia fue asignar la actividad que dejara el recurso libre la mayor cantidad de tiempo en el futuro y que fuera compatible con las anteriores.
- Ejemplo: sea



4

☀ Características:
















- 🛠 Simples de implementar.
- ❌ No siempre garantizan la solución óptima.

💻 Ejemplo:

```
def mochila_greedy(capacidad, objetos):
    objetos.sort(key=lambda x: x[1]/x[0], reverse=True)
    peso_total = 0
    beneficio = 0
    for peso, valor in objetos:
        if peso_total + peso <= capacidad:
            peso_total += peso
            beneficio += valor
        else:
            fraccion = (capacidad - peso_total) / peso
            beneficio += valor * fraccion
            break
    return beneficio
```



Comparación entre Estructuras Algorítmicas

 Estructura	 Ventajas	 Desventajas
Secuenciales	 Simples y directas.	 No aplicables a problemas complejos.
Selección	 Flexibilidad en el flujo.	 Complicadas con múltiples condiciones.
Repetitivas	 Procesamiento automatizado de datos.	 Riesgo de ciclos infinitos.
Dividir y vencer	 Eficiencia en problemas recursivos.	 Más memoria para subproblemas.
Programación dinámica	 Evita redundancias.	 Complejidad en su implementación.
Greedy	 Fácil de entender y codificar.	 No siempre óptimo.