Complejidad en Algoritmos

```
CREAR UN ALGORITMO QUE LEA UN NÚMERO ALEATORIO Y LUEGO PERMITA AL USUARIO
ADIVINARLO INDICANDO CADA VEZ SI ESTÁ MUY LEJOS, CERCA O CASI E INDICAR
CUÁNTAS VECES REALIZO E INTENTO

INICIO

    entero c = 0
    logico encontro = Falso
    LEER numero

    MIENTRAS ( NO encontro)

        c = c + 1
        LEER num

        SI (numero > num) ENTONCES
            SI(numero - num < 5)
                IMPRIMA "Casi"
            SINO SI (numero - num < 10) ENTONCES
                     IMPRIMA "Cerca"
                  SINO
                      IMPRIMA "Lejos"
                 FIN SI
            FIN SI
        SINO SI(numero < num) ENTONCES
               SI(num - numero < 5)
                   IMPRIMA "Casi"
               SINO SI (num - numero < 10) ENTONCES
                        IMPRIMA "Cerca"
                    SINO
                        IMPRIMA "Lejos"
                    FIN SI
               FIN SI
            SINO
               encontro = Verdadero
            FIN SI
        FIN SI

    FIN MIENTRAS

    IMPRIMIR "La pesona realizo", c, " intentos"
FIN
```


Universidad Católica ANDRÉS BELLO UCAB

Algoritmos y Programación

Complejidad en Algoritmos y Programas

Complejidad en Algoritmos

Características de los Algoritmos

- **Son una secuencia de instrucciones Ordenada y de Fácil comprensión.**
- **Son una secuencia finita de instrucciones.**
- **Su orden de ejecución es Secuencial.**
- **Son instrucciones Precisas que tienen su equivalente en cada lenguaje de Programación.**
- **Un Programa es un Algoritmo expresado en un Lenguaje de Programación específico.**
- **Todo Algoritmo debe responder del mismo modo ante las mismas condiciones (Determinismo).**

Complejidad en Algoritmos

Criterios para Evaluar un Programa

- ✓ **Eficiencia**
- ✓ **Portabilidad**
- ✓ **Eficacia**
- ✓ **Robustez**

Tipos de Complejidad Tiempo y Espacio

- ❖ **Complejidad Temporal** (esta relacionada con el tiempo de CPU utilizado)
  Es la que se refiere al tiempo de ejecución del Algoritmo y depende de las entradas

- ❖ **Complejidad Espacial** (esta relacionada con el espacio utilizado en Memoria RAM)
  Es la que se refiere al espacio en Memoria RAM que se reserva para las variables, constantes, arreglos, matrices, registros , apuntador o dirección.

Complejidad en Algoritmos

Criterios para Evaluar un Algoritmo

Preguntas en otros ámbitos

Los Ingenieros Civiles

¿ Cuanto peso soporta un terreno para construir un edificio?

Una Empresa de Construcción

¿Cuántos bloques se pueden fabricar en un día ?

Los Ingenieros de Informática

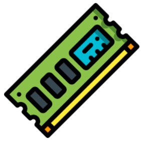
¿ Cual es el tiempo de respuesta de un programa y que recursos de hardware y software necesita ?

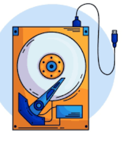
> Run

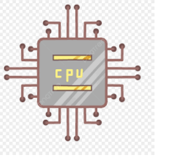
Tiempo

+

Recursos







Complejidad en Algoritmos

Criterios para Evaluar un Algoritmo

Preguntas para Evaluar nuestros Algoritmos o Programas

- **¿Cual es el tamaño de los datos que soporta?**
- **¿Cuantos usuarios soporta?**
- **¿Cuantas transacciones se pueden procesar?**
- **¿Cual es el tamaño de las Bases de Datos?**
- **¿Cuántos Servidores se necesitan?**

En cuanto a la complejidad Temporal

- **¿Cómo medir el tiempo que tardan mis Algoritmos o Programas?**

. - Midiendo el tiempo de ejecución

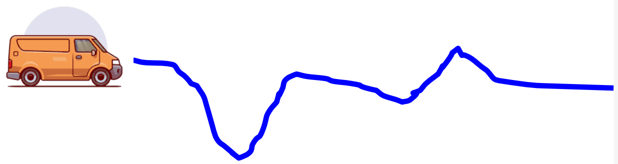
Si con un dato tarda 2 µs con dos datos ¿ 4 µs ?

Complejidad en Algoritmos

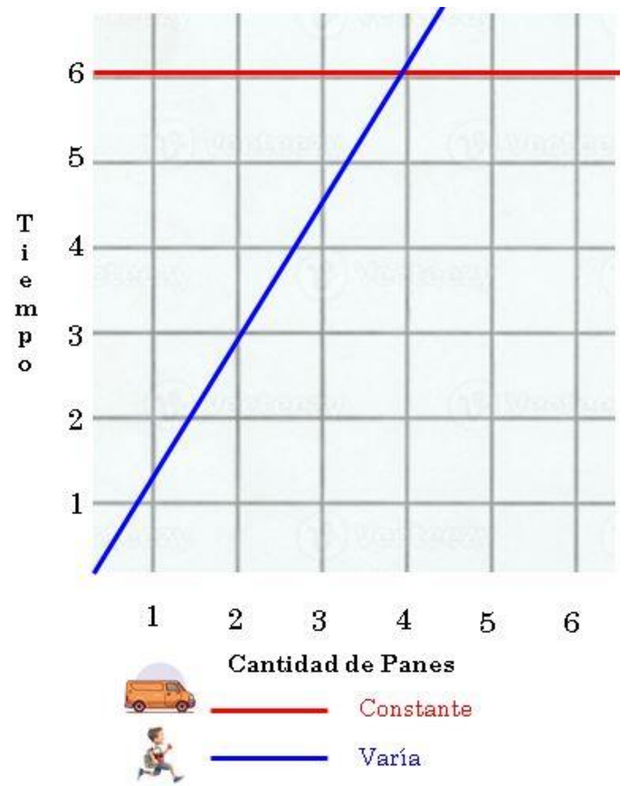
Ejemplo

Llevar panes al Restaurant

1) Van de carga (Tiempo 6 min)

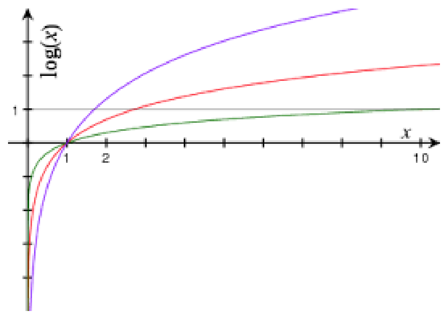


2) Trotando (Tiempo 3 min)



Criterios para Evaluar un Algoritmo

. – Determinar Variaciones de la Función



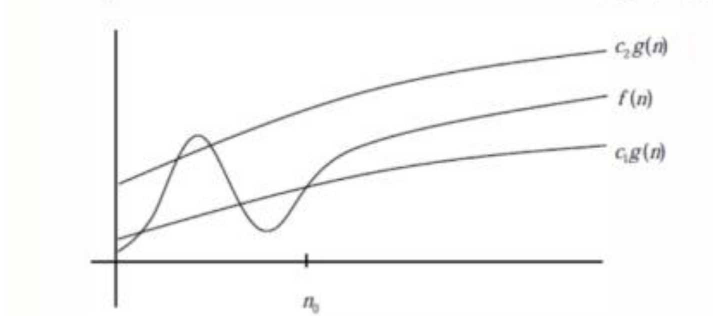
Tomar el máximo valor (peor de los casos)

-) Puede correr a 65 Mph

-) Corre hasta 90 Mph

. – Ajustarlas a un Orden de comportamiento Asintótico (Big O)

Comportamiento Asintótico $\Theta(g(n))$



$\Theta(g(n)) = \{\, f(n) :$

si existen constantes positivas c_1, c_2 y n_0 tal que $0 \leq$

$c_1 g(n) \leq f(n) \leq c_2 g(n)$ *para todo* $n \geq n_0 \,\}$

Complejidad Temporal y Espacial en Algoritmos

Notación Big O

La Notación Big – O es una notación matemática que se utiliza para :

.- Describir la Rapidez o Velocidad de Procesamiento de un Algoritmo o Programa
.- La Complejidad (lógica coherente y fácil) del mismo.
.- Determinar el Impacto en los recursos Hardware y Software

Esta Notación hace referencia y se fundamenta en el orden de crecimiento asintótico (comportamiento de la función) de la función que mejor describe a cada **herramienta Algorítmica** y **Estructuras de Datos** tomando como referencia el peor escenario que se puede dar (**peor caso**).

Reglas de la Notación O

Definición 1: Sean ***f*** y ***g*** dos funciones. Se dice que ***f*** = **O**(***g***) (*f* es de orden *g*) **si y sólo si** existe ***c*** ϵ *a* ***R*** **+** y ***n0*** ϵ a **N** tal que para un ***n*** >***n0*** se cumpla ***f*** (***n***) < = ***c***.*g* (***n***). (**N** no incluye el **0**).
La relación **O** denota una ***dominancia de funciones***, en donde la función ***f*** está acotada superiormente por un múltiplo de la función ***g****, por lo que* ***f*** =**O**(***g***) refleja que el **orden de crecimiento asintótico** de la función ***f*** es < = la función ***g***. **Siempre f (n) < = N * g(n)**

Definición 2: Sean ***f*** y ***g*** dos funciones se dice que ***f*** y ***g*** tienen igual orden de crecimiento (*f*=Q(*g*)) **si y sólo si** existe ***c***, ***d*** *pertenecientes a* ***R+*** y ***n0*** perteneciente a **N** tal que ***n*** > ***n0*** *y* se cumpla ***d***.***g*** (*n*) < = ***f*** (***n***) < = ***c***.***g*** (***n***). **Acotada**

Definición 3: Se realiza la instrucción **A**, como la secuencia dos acciones **A1** y **A2** de **complejidades temporales T1**(**n**) y **T2** (**n**) respectivamente. Si **T1**(**n**) es **O**(**f**(**n**)) y **T2**(**n**) es **O**(**g**(**n**)) entonces **A** es de complejidad **O**(**max**(**f** (**n**), **g** (**n**)). **Peor de los casos**

Complejidad en Algoritmos

Análisis de Complejidad de Tiempo

El tiempo de ejecución depende principalmente de un factor:

.- El relacionado con la **entrada de datos** del programa(**cantidad de datos**)

Tasa de Crecimiento

La tasa de crecimiento obtenida para hallar el orden de complejidad en tiempo de un Algoritmo o Programa, permite entre otras cosas:

.- Determinar el comportamiento del Algoritmo en función del tamaño del problema.

.- Determinar cuánto tiempo de cómputo aumenta al incrementar el tamaño del problema.

.- Determinar el impacto en los recursos Hardware y software.

.- Facilita la comparación de Algoritmos para determinar cuál será el más eficiente (el que tenga menor tasa de crecimiento).

Complejidad en Algoritmos

O(1) : Constante

Tiempo no depende de las cantidades de datos de entrada

Ejemplo una Lista

A = [9,1,5,8,2,3]

x = **A . POP ( )** igual a 3 no importa el tamaño de la Lista

O(N) : Lineal

Directamente proporcional al tamaño de las Entradas

Ejemplo un **Ciclo Iterativo Simple**

```
for i in range (0, N, 1 )
     print(40)          N
```

O(N^2) : Cuadrática

Ofrece proporcionalidad al cuadrado

Ejemplo **Ciclos iterativos Anidados**

```
for i in range (0, N, 1 )          #  Aporta N
     for j in range (0, N, 1 )     #  Aporta N
          print (65)               #  N x N = N²
```

O(Log N) : Logarítmica

Divide el problema en 2 (Búsqueda Binaria)

Entradas	Tiempo	
2	1	
4	2	
8	3	
16	4	**Log N**

O(2^N) : Exponencial

Menos Eficiente que la cuadrática

Ejemplo Secuencia de Fibonacci **2^N**

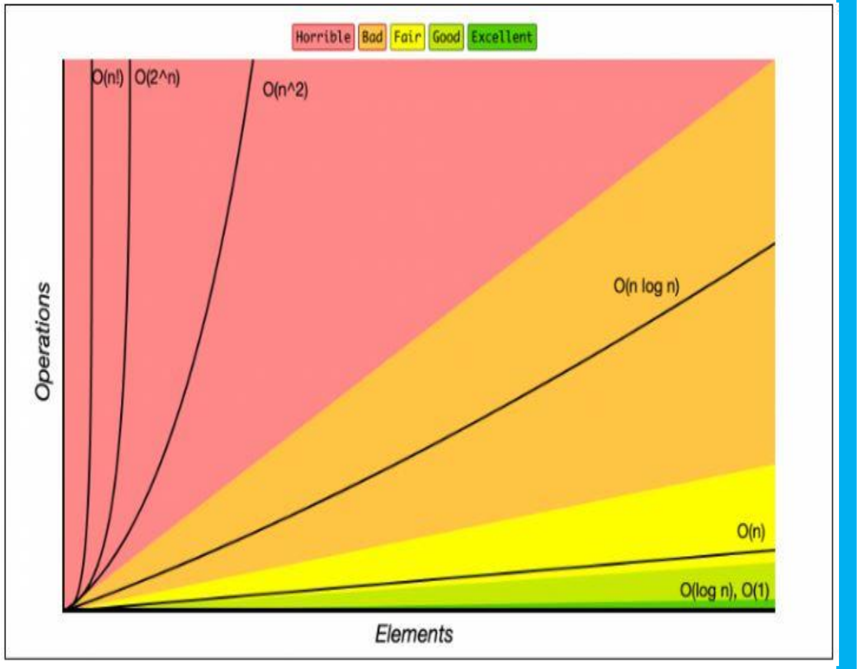
1, 2, 3, 5, 8, 13, 21, 34,.. Generar Secuencias dep datos

O(N!) : Factorial

Mucho más ineficiente que todas (**Recursivos**)

Ejemplo Factorial Recursivo

10! = 3.628.800 fun(3)=fun(3)+fun(2) +fun(1) **N !**



Complejidad en Algorítmos

UCAB Universidad Católica Andrés Bello

Notación Big - O

- **O(1) : Constante**
  - **Instrucciones simples**
- **O(N) : Lineal**
  - **Ciclo Iterativo Simple**
- **O(N^2) : Cuadrática**
  - **Ciclos iterativos Anidados**
- **O(Log N) : Logarítmica**
  - **Divide el problema en 2**
- **O(2^N) : Exponencial**
  - **Secuencias numéricas con valores dependientes**
- **O(N!) : Factorial**
  - **Recursivo y Funciones**

Cálculo Big-O

Instrucciones simples

Toda línea de código se considera Big O de 1, siempre y cuando NO sea un ciclo, recursión o función no constante

Algoritmo

```
    n1,n2,n3,n4:entero;
Inicio                                  Cálculo Big-O
    Escribir ( " Indique Nº  " ) ;
    Leer( n1 ) ;                        # O ( 1 )
    n2 ←2;                              # O ( 1 )
    n3←(n1+ n2);                        # O ( 1 )
    Si ( n3 > 30 ) entonces             # O ( 1 )
       n4←n3;                           # O ( 1 )
       SINO
          n4←n2;                        # O ( 1 )
    FSi
Fin                                     5O( 1 )
```

Programa **Phyton**

```
def main( ) :                                          Cálculo Big-O
    n1 = int(input("indique Nº "))   # O( 1 )
    n2 = 2                           # O( 1 )
    n3 = (n1+ n2) ;                  # O( 1 )
    if ( n3 > 30 ) :                 # O( 1 )
        n4=n3                        # O( 1 )
    else:
        n4=n2                        #  O( 1 )

main( )                                                5O( 1)
```

Complejidad en Algoritmos

Cálculo Big-O

Condicionales

Toda línea de código se considera Big O de 1, siempre y cuando NO sea un ciclo, recursión o función no constante

Algoritmo

```
        n1,n2,n3,n4:entero;
Inicio                                   Cálculo Big-O
        Escribir ( " Indique Nº  " ) ;
        Leer( n1 ) ;                     # O ( 1 )
        n2 ←2;                           # O ( 1 )
        n3←(n1+ n2);                     # O ( 1 )
        Si ( n3 > 30 ) entonces          # O ( 1 )
            n4←n3;                       # O ( 1 )
            SINO
                n4←n2;                   # O ( 1 )
                n4←(n4+n3);              # O ( 1 )

        FSi
Fin                                      O = 6
```

Programa **Phyton**

```
def main( ) :                            Cálculo Big-O
    n1 = int(input("indique Nº "))       # O( 1 )
    n2 = 2                               # O( 1 )
    n3 = (n1+ n2) ;                      # O( 1 )
    if ( n3 > 30 ) :                     # O( 1 )
        n4=n3                            # O( 1 )
    else:
        n4=n2                            # O( 1 )
        n4 +=n3                          # O( 1 )

main( )                                  O = 6
```

Complejidad en Algoritmos

Cálculo Big-O

Ciclos Iterativos simples

Cuando esta presente un Ciclo Iterativo simple en el cual se relaciona el ciclo con todos los datos de entrada, en la mayoría de los casos el orden de la **Notación Big-O** es **O** (n) donde n es el conjunto de datos a ser tratados y el cual esta relacionado con los datos de Entrada (**N**)

Algoritmo

```
Algoritmo
      N, i,sum::entero;
Inicio                                Cálculo Big-O
      Escribir("Indique Nº ");
      Leer ( N ) ;                    #  O ( 1 )
      sum←0 ;                         #  O ( 1 )
      Para i←0 hasta  N hacer         #  O ( N )
          sum←(sum + i )              #  O ( 1 )
      FPara

      i←0 ;                           # O ( 1 )
      sum←0 ;                         # O ( 1 )
      Mientras ( i < = N ) hacer      # O ( N )
          sum←( sum + i ) ;           # O ( 1 )
          i←( i + 1 ) ;               # O ( 1 )
      FMientras
Fin                                   2N + 7
                                        N
                                      O ( N )
```

Programa **Phyton**

```
def main( ) :                             Cálculo Big-O
    N = int(input("indique Nº ")   # O( 1 )
    sum=0                          # O( 1 )
    for i in range (0,N+1,1)       # O( N )
        sum += i                   # O( 1 )

    i = 0                          # O( 1 )
    sum=0                          # O( 1 )
    while ( i < = N ) :            # O( N )
        sum += i                   # O( 1 )
        i += 1                     # O( 1 )

main( )                            2N + 7
                                     N
                                   O ( N )
```

Complejidad en Algoritmos

Casos Especiales en Ciclos Iterativos simples

Caso 1

Cuando el Ciclo **NO** esta relacionado con los datos de Entrada (**N**)
Entonces el Orden Big-O es igual **O** (1)

Algoritmo

```
        N, i,sum::entero;
Inicio                              Cálculo Big-O
       Escribir("Indique Nº ");
       Leer ( N ) ;                      #  O ( 1 )
        sum←0 ;                          #  O ( 1 )
        Para i←0 hasta  4 hacer          #  O ( 1 )
            sum←( sum +  2 )             #  O ( 1 )
        FPara
Fin                                    4O ( 1 )
                                        O ( 1 )
```

Caso 2

Cuando el Ciclo **se detiene antes de llegar al final** por cumplirse una condición, no trata a todos los datos de Entrada (**N**)

Uso del **break**

Entonces el Orden Big-O es igual **O** (n) porque se debe tomar el peor de los casos, es decir que la condición se cumpla con el último elemento.

Programa **Phyton**

```
 import numpy as np
def main( ) :
      a=np.array( [ 1,2,3,4,5 ] )
      n1 = int(input("indique Nº "))
       for i in range(0,5,1):
           if ( n1== a[ i ] ) :
               print("Nº Encontrado")
               break
             else:
               print("Nº NO Encontrado")
 main( )
```

Complejidad en Algoritmos

UCAB Universidad Católica Andrés Bello

Cálculo Big-O

Ciclos Iterativos simples

Algoritmo

```
        N, i,n1::entero;
Inicio                                      Cálculo Big-O
        Escribir("Indique Nº ");
        Leer ( N ) ;                        # O ( 1 )
        n1←0;                               # O ( 1 )
        # Orden Progresivo con Para
        Para i←0 hasta  N hacer             # O ( n )
            n1←( n1 + i );                  # O ( 1 )
        FPara
        # Orden Regresivo con Para
        Para i←N hasta  0 hacer             # O ( n )
            n1←( n1+ i ) ;                  # O ( 1 )
        FPara
        # Orden Progresivo con Mientras
        i←0 ;                               # O ( 1 )
        Mientras ( i < = N ) hacer          # O ( n )
            n1←( n1 + i );                  # O ( 1 )
            i←( i + 1 );                    # O ( 1 )
        FMientras
        # Orden Regresivo con Mientras
        i←N ;                               # O ( 1 )
        Mientras ( i > 0 ) hacer            # O ( n )
            n1←( n1 + i );                  # O ( 1 )
            i←( i - 1 );                    # O ( 1 )
        FMientras
Fin                                         4n + 10
                                               n
```

Programa **Phyton**

```
def main( ) :                                    Cálculo Big-O
    N = int(input("indique Nº "))                # O( 1 )
    n1=0                                         # O( 1 )
    # Orden Progresivo con For
    for i in range (0,N+1,1)                     # O( n )
        n1 += i                                  # O( 1 )
    # Orden Regresivo con For
    for i in range (N,-1,-1)                     # O( n )
        n1 += i                                  # O( 1 )
    # Orden Progresivo con While
    i = 0                                        # O( 1 )
    while ( i < = N ) :                          # O( n )
        n1 += i                                  # O( 1 )
        i += 1                                   # O( 1 )
    # Orden Regresivo con While
    i = N                                        # O( 1 )
    while ( i > N ) :                            # O( n )
        n1 += i                                  # O( 1 )
        i - = 1                                  # O( 1 )
main( )                                          4n + 10
                                                    n
```

Complejidad en Algoritmos

Cálculo Big-O

Ciclos Iterativos Anidados

Cuando esta presentes Ciclo Iterativo Anidados en los cuales se relaciona cada ciclo con todos los datos de entrada, en la mayoría de los casos el orden de la **Notación Big-O** es **O** (n) en cada ciclo y n es el conjunto de datos a ser tratados y el cual esta relacionado con los datos de Entrada (**N**) entonces el Orden de la Notación Big–O es **O** (n^2)

Programa **Phyton**

```
def main( ) :                          Cálculo Big-O
     N = int(input("indique Nº ")  # O( 1 )
     sum= 0                        # O( 1 )
     for i in range (1,N,1)        # O( n )
        for j in range(N,-1,-1)    # O( n )   n * n = O( n²  )
          sum=(sum + ( i * j ))    # O( 1 )
        print(sum)

main( )                                O(n²) ) +  3O(1)
                                         n²
```

UCAB Universidad Católica ANDRÉS BELLO

Casos Especiales en Ciclos Iterativos anidados

Caso 1

Cuando el Ciclo **NO** esta relacionado con los datos de Entrada (**n**)
Entonces el Orden Big-O es igual **O** (1)

Programa **Phyton**

```
def main( ) :                                 Cálculo Big-O
      N = int(input("indique Nº ")   # O( 1 )
      sum = 0                         # O( 1 )
      for i in range (1,N,1) :        # O( n )
          for cont in range(1,3,1) :  # O( 1 )
            sum=(sum + cont)          # O( 1 )
          x= ( sum * i )              # O( 1 )
      for j in range (N, 0, -1):      # O( n )
          z= ( j +  5 )               # O( 1 )

main( )                               2 O(n) + 6 O(1)
                                        O ( n )
```

Importante :Cuando los ciclos anidados hacen referencia a cantidades de elementos Distintas cada uno pero relacionadas con **n** (conjunto de datos de entrada), tiene en la **Notación Big-O** un orden de **O**(n) cada uno dando como resultado al involucrar a los dos ciclos un orden de **O**(n^2)

Complejidad en Algoritmos

Cálculo Big-O

Reducción o división del conjunto de Datos de Entrada

Cuando en un problema para conseguir su solución se reduce en cada momento el conjunto de Datos de Entrada (n) hasta finalizar el problema o conseguir la solución entonces su Orden en Notación Big O se comportara como log(n)

Búsqueda Binaria (siempre sobre arreglos Ordenados)

Dado un Arreglo de 8 posiciones que almacena números enteros DISTINTOS positivos mayores que cero y el cual esta ORDENADO en orden creciente se desea indicar si se encuentra el número **36** entre los elementos del Arreglo .

num = **36** (Nº a Buscar)

Forma Elemental de Búsqueda

i

36

6	9	13	18	25	36	41	54

n [i]

```
for i in range ( 0, 8, 1) :
    if ( num = = n [ i ] ) :
        print ( " El numero " , num, " fue encontrado " )
        break
```

¿ Cual es el **MEJOR** y **PEOR** de los casos ?

MEJOR = primera posición
PEOR = última posición

¿ Que sucede con arreglos muy grandes ?

Complejidad en Algoritmos

Búsqueda Binaria (siempre sobre arreglos Ordenados)

num=**36**

3

0 ((Inicio + Fin) **DIV** 2) Len (n) = 7

Inicio Mitad Fin

0	1	2	3	4	5	6	7
6	9	13	18	25	36	41	54

n [**i**]

```
Si( n [mitad] < num) entonces
        inicio←( mitad + 1 )
        SINO
            Si ( n [mitad ] > num ) entonces
                fin←( mitad - 1)
                SINO
                     Escribir( " El número esta en el Arreglo " )
            FSi
FSi
```

((Inicio + Fin) **DIV** 2)

(3 + 1)

0 4 5 7

Inicio Inicio Mitad Fin

0	1	2	3	4	5	6	7
6	9	13	18	25	36	41	54

n [**i**]

36 El número esta en el Arreglo

Complejidad en Algoritmos

Búsqueda Binaria (siempre sobre arreglos Ordenados

```
Import numpy
import numpy as np

def main():
   a=np.array([2,5,8,12,16,23,38,56,72,91])

    print("El arreglos es: ")
   print(a)
   print("indique Nº a buscar: ")
   buscar=int(input())
   # Busqueda Binaria
   inicio=0
   fin=len(a)
   mitad=0
   while(inicio<=fin):
      mitad=((inicio +fin )//2)
      if(a[mitad] < buscar):
         inicio = (mitad+1)
      elif(a [mitad] > buscar):
         fin = (mitad-1)
      else:
         print("El numero buscado es ",buscar, " y fue encontrado)
         break
main( )
```

Orden en la **Notación Big O** es **log (n)**

Complejidad en Algoritmos

UCAB Universidad Católica Andrés Bello

Cálculo Big-O

Recursividad

Cuando dentro de una función se invoca o llama nuevamente a la función modificando los parámetros, este tipo de algoritmos proporciona en el Orden en Notación Big O dependiendo si se realiza un solo llamado a la función su Orden será **O (n)** , en cambio si son dos (2) llamados a la función el Orden es 2^n (Orden exponencial).

Caso 1

```
def recursiva( n ) :
    if ( n < = 0 ) :
        return 1
    else :
        return  1 + recursiva ( n – 1 )
```

O (n)

1
1
1

Pila

Caso 2

```
def recursiva1 ( n, x, y ) :
    if ( n < = 0 ) :
        print ( n, x, y )
    else :
        recursiva1 ( n - 1, x+1, y )
        recursiva 1( n - 1, x, y+1 )
```

O (2^n)