Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América

Análisis y diseño de algoritmos

Semana 1

Datos del Docente

Javier Antonio Prudencio Vidal

jprudenciov@unmsm.edu.pe
Cel 996577810

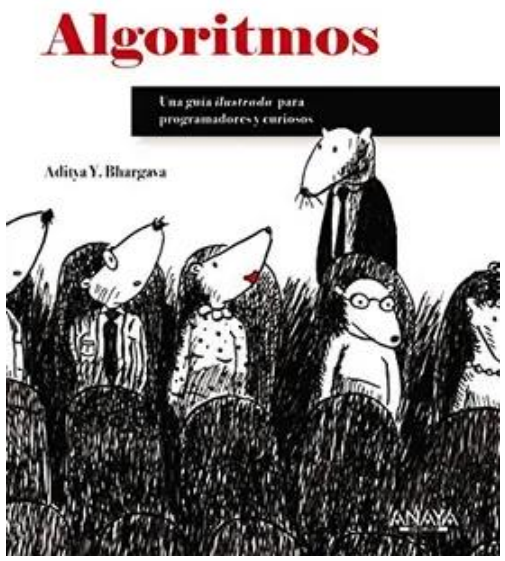
Breve Experiencia Personal

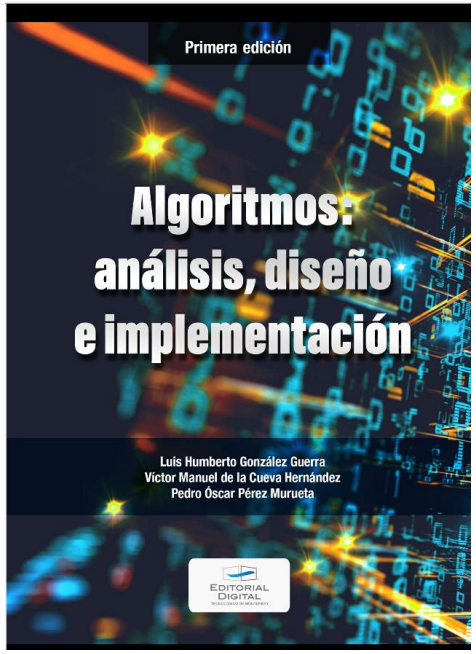
- Magister en Administración mención en Sistemas de Información y Telecomunicaciones/Ing. Electrónico: 28 años de Experiencia Profesional
- Docente Universitario desde el 2010
- Experto Telefónica del Perú S.A.A: Dirección de Tecnología y Excelencia Operacional (2000-Actualidad)
- Ingeniero de Soporte Técnico Unibanca (1996-1999)

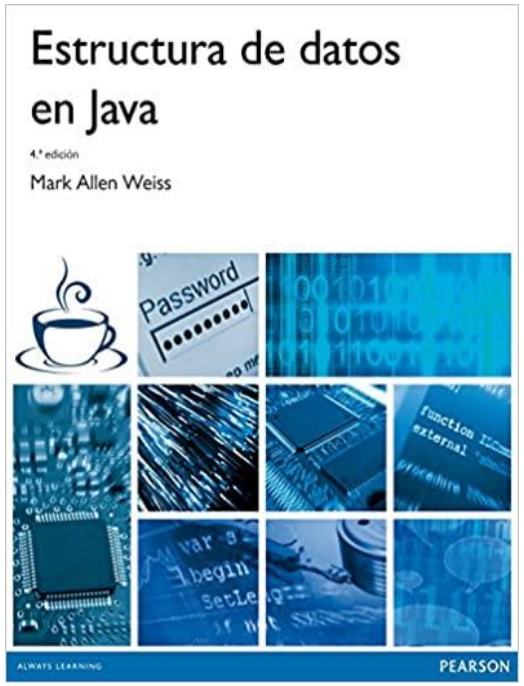
Presentación del estudiante

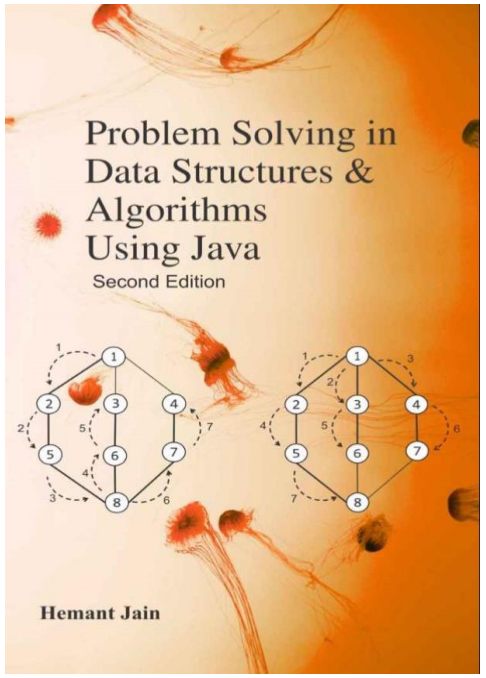
- Nombre y Edad
- Experiencia Personal/laboral
- Expectativas del Curso

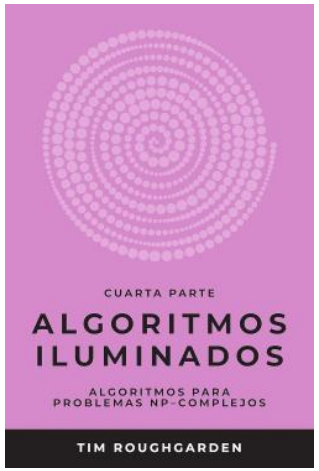
Libros Guía

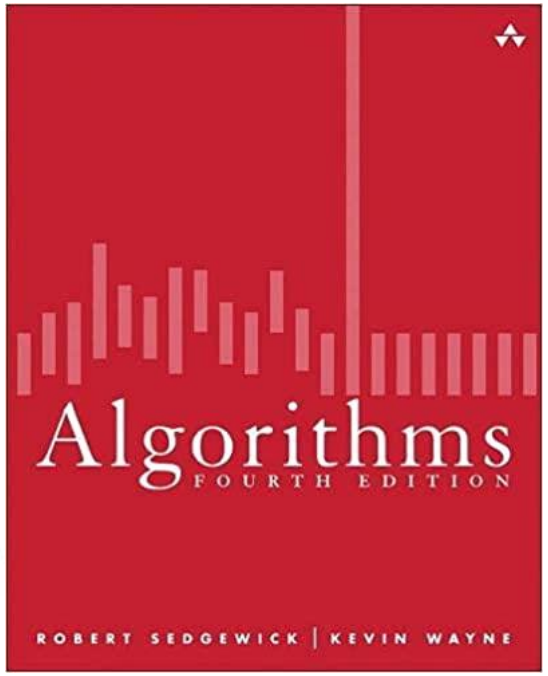


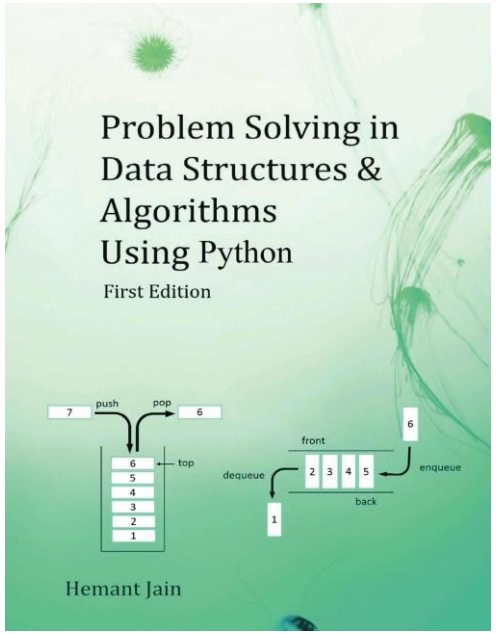












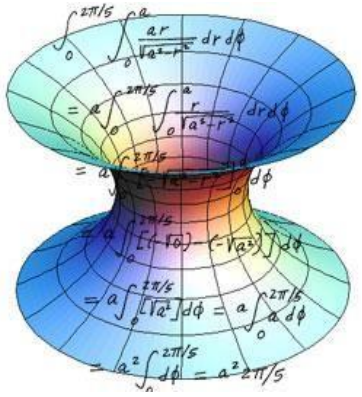
UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América

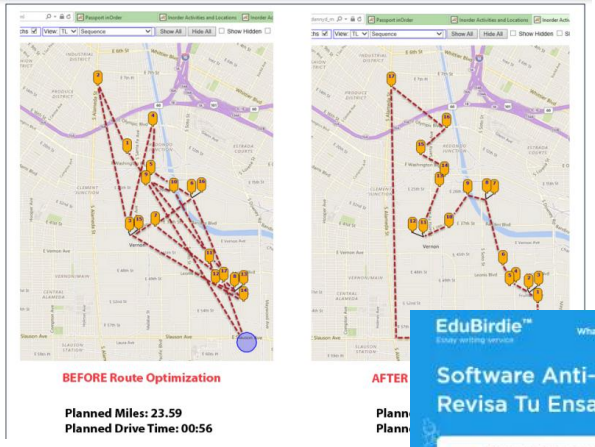
Revisión del Sílabo, elección del delegado y formación de grupos

Proyecto fin de curso

Calculadora científica



WorkForce Management



Sistema antiplagio

UNMSM

Rúbrica

Criterios	Logrado	En proceso	En inicio
Utilización de diferentes tipos de algoritmos	Al menos utiliza 3 tipos de algoritmos desarrollados en el curso. **(5 puntos)**	Sólo utiliza 2 tipos de algoritmos desarrollados en clase. **(3.5 puntos)**	Sólo utiliza un tipo de algoritmo desarrollado en clase. **(2 puntos)**
Utilización de diferentes estructuras de datos	Al menos utiliza 3 tipos de estructura de datos desarrollados en el curso. **(5 puntos)**	Sólo utiliza 2 tipos de estructuras de datos desarrollados en clase. **(3.5 puntos)**	Sólo utiliza un tipo de estructura de datos desarrollado en clase.. **(2 puntos)**
Se presenta un análisis de complejidad del código de software	Describe cuál es el nivel de complejidad temporal y espacial. **(4 puntos)**	Describe sólo un tipo de complejidad. **(3 puntos)**	No describe ningún análisis de complejidad del código de software. **(0.5 puntos)**
Comportamiento de la aplicación ante diferentes valores de entrada	El programa de software no incrementa su complejidad en forma notable cuando la entrada es muy grande. **(3 puntos)**	El programa de software incrementa su complejidad si la entrada crece a valores enormes y que pueden ocurrir en la realidad. **(1.5 puntos)**	Si la entrada crece a valores grandes y reales la aplicación no funciona correctamente. **(0.5 punto)**
Utilidad de la Aplicación	El trabajo presenta la solución a un problema existente en la realidad y los algoritmos y estructura de datos se pueden utilizar en grandes sistemas. Se puede utilizar por el usuario común **(3 puntos)**	El trabajo no presenta la solución a un problema existente en la realidad, los algoritmos y estructura de datos se pueden utilizar en grandes sistemas El trabajo se puede utilizar por un usuario común **(2 puntos)**	El trabajo no presenta la solución a un problema existente en la realidad, los algoritmos y estructura de datos no se pueden utilizar en grandes sistemas. El trabajo se puede utilizar por un usuario común. **(1 punto)**

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América

Logro de la sesión

Al finalizar la sesión, el estudiante:

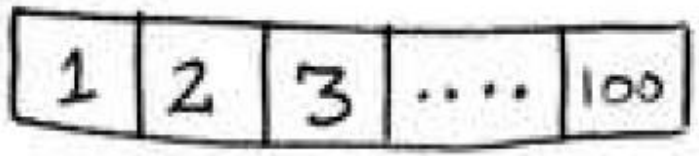
- **Determina empíricamente y teóricamente la complejidad temporal de algoritmos simples**

Algoritmos presentes en la vida diaria: Búsqueda binaria en Java/Python

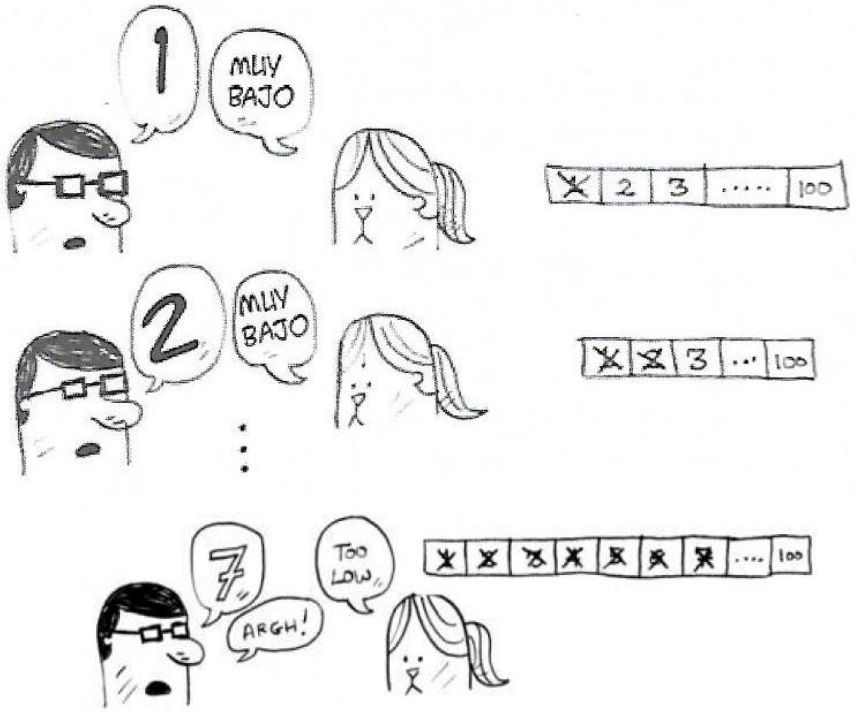
Algoritmos presentes en la vida diaria

Búsqueda Binaria: Dinámica adivina el número

Estoy pensando en un número entre 1 y 100. (El 57 para este ejemplo)



Debes tratar de adivinar mi número en la menor cantidad posible de intentos. En cada intento diré si tu respuesta es demasiado baja, demasiado alta o correcta. Supongamos que comienza a adivinar de esta manera: 1, 2, 3, 4 .... Así es como funcionaría.



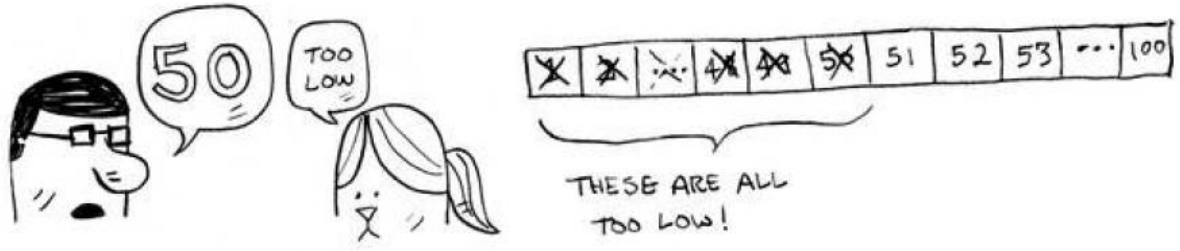
Un mal enfoque para adivinar el número.

Esta es una búsqueda simple (tal vez una búsqueda estúpida sería un término mejor). Con cada intento, estás eliminando un solo número. Si mi número era 99, ¡podría llevarte 99 intentos llegar ahí!

Algoritmos presentes en la vida diaria

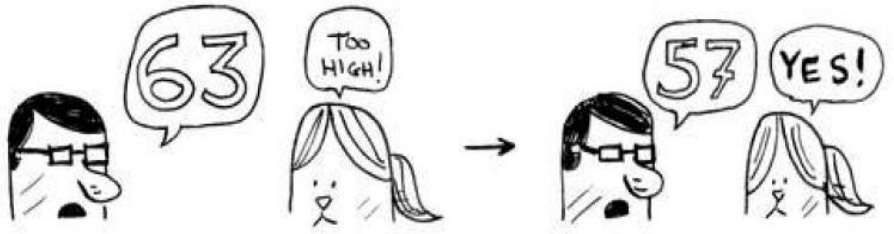
Una mejor forma de buscar

Aquí hay una mejor técnica. Comience con 50.



Demasiado bajo, pero acabas de eliminar la mitad de los números! Ahora sabes que 1-50 son muy bajos. Siguiente respuesta: 75.

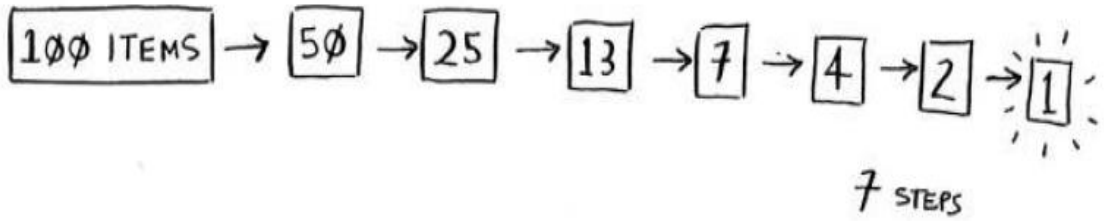
Demasiado alto, pero nuevamente se reduce a la mitad las posibles respuestas. Con la búsqueda binaria, seleccionas el número del medio y se elimina la mitad de los números restantes. El siguiente es 63 (la mitad entre 50 y 75).



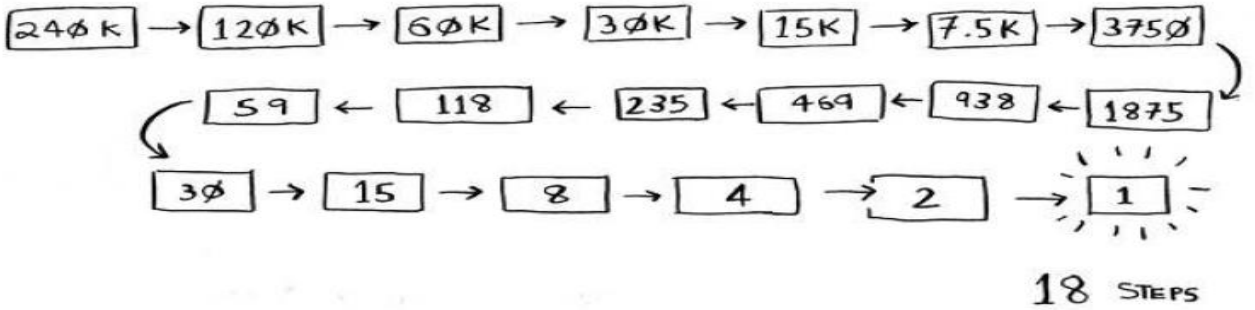
Esta es una búsqueda binaria. ¡Vamos a implementar este algoritmo en Java!

Algoritmos presentes en la vida diaria

Cualquiera que sea el número en el que estoy pensando, puedes adivinarlo en un máximo de siete oportunidades, ¡porque eliminas tantos números en cada oportunidad!



Y una lista de 240,000 números???



La búsqueda binaria solo funciona cuando su lista está ordenada. Por ejemplo, los nombres en una guía telefónica están ordenados alfabéticamente, por lo que puede usar la búsqueda binaria para buscar un nombre. ¿Qué pasaría si los nombres no estuvieran clasificados?

Algoritmos presentes en la vida diaria

Escribir el algoritmo de búsqueda binaria en Java o Python

Para representar la lista de números, usaremos un arreglo de números, los cuales son una secuencia de celdas consecutivas donde podemos almacenar los números. Las celdas se numeran empezando por 0: la primera celda está en la posición # 0, la segunda es # 1, la tercera es # 2, y así sucesivamente.

Escribiremos en Java el método busqueda_binaria que toma un arreglo ordenado (lista) y el elemento a buscar (elemento). Si el elemento está en el arreglo, el método retorna su posición. Harás un seguimiento de qué parte de la matriz o arreglo tienes que buscar.

Al principio, este es todo el arreglo

```
menor = 0
mayor = lista.length - 1
```

Cada intento de búsqueda escogería el elemento ubicado en la mitad del arreglo y se haría hasta que se haya reducido la búsqueda a un solo elemento:

```
medio = (menor+mayor)/2
estimado = lista[medio]
```

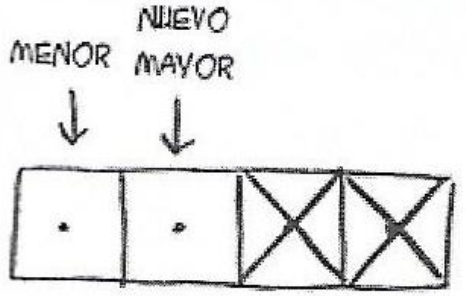
medio debe ser redondeado si `(menor+mayor)` *no es un número par*

Si el número encontrado es muy bajo, actualiza la variable menor, para intentar ahora con un arreglo reducido:

```
if (estimado < elemento){
menor = medio + 1
}
```

Si el número encontrado es muy alto, actualiza la variable mayor, para intentar ahora con un arreglo reducido:

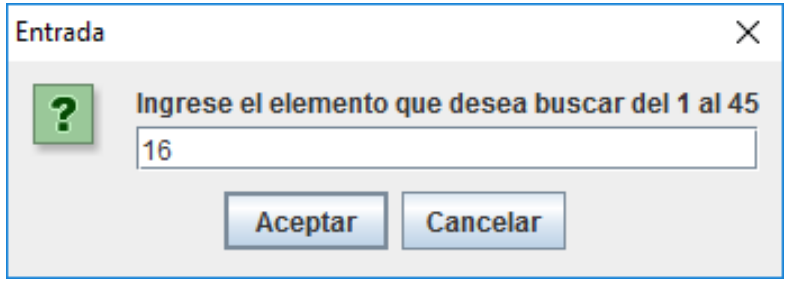
```
if (estimado > elemento){
mayor = medio - 1
}
```

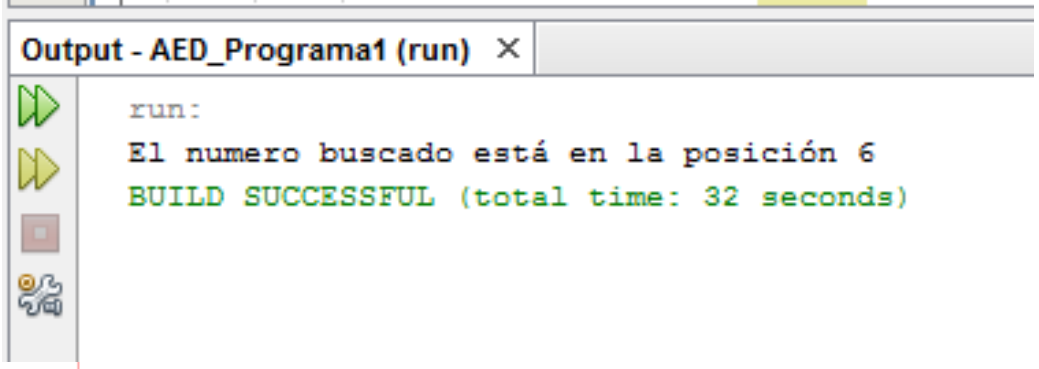


Algoritmos presentes en la vida diaria

El algoritmo de búsqueda binaria en Java

```java
package Semana1;
import javax.swing.JOptionPane;
public class AED_Programa1 {
    public static void main(String[] args) {
        int[] lista={1,3,5,7,9,10,16,19,21,24,27,29,31,31,33,36,39,41,43,45};
        AED_Programa1 ejemplo1=new AED_Programa1();
        int elemento=Integer.parseInt(JOptionPane.showInputDialog("Ingrese el elemento que desea buscar del 1 al 45"));
        int resultado=ejemplo1.busqueda_binaria(lista, elemento);
        if(resultado!=-1){
        System.out.println("El numero buscado está en la posición "+ resultado);
        }else{
            System.out.println("El numero buscado no está en la lista");
        }
    }
    private int busqueda_binaria(int[] lista, int elemento){
        int menor=0,medio;
        int mayor=lista.length - 1;
        while(menor<=mayor){
            medio=menor+(mayor-menor)/2;
            if(lista[medio]==elemento){
                return medio;
            }else if(lista[medio]>elemento){
                mayor=medio-1;
            }else{
                menor=medio+1;
            }
        }
        return -1;
    }
}
```





Algoritmos presentes en la vida diaria

Escribir el algoritmo de búsqueda binaria en Python

```
def busqueda_binaria(lista, elemento):
    menor = 0
    mayor = len(lista)- 1
    while menor <= mayor:
        medio = (menor + mayor)/2
        estimado = lista[medio]
        if estimado == elemento:
            return medio
        if estimado > elemento:
            mayor = medio - 1
        else:
             menor = medio + 1
    return None
```

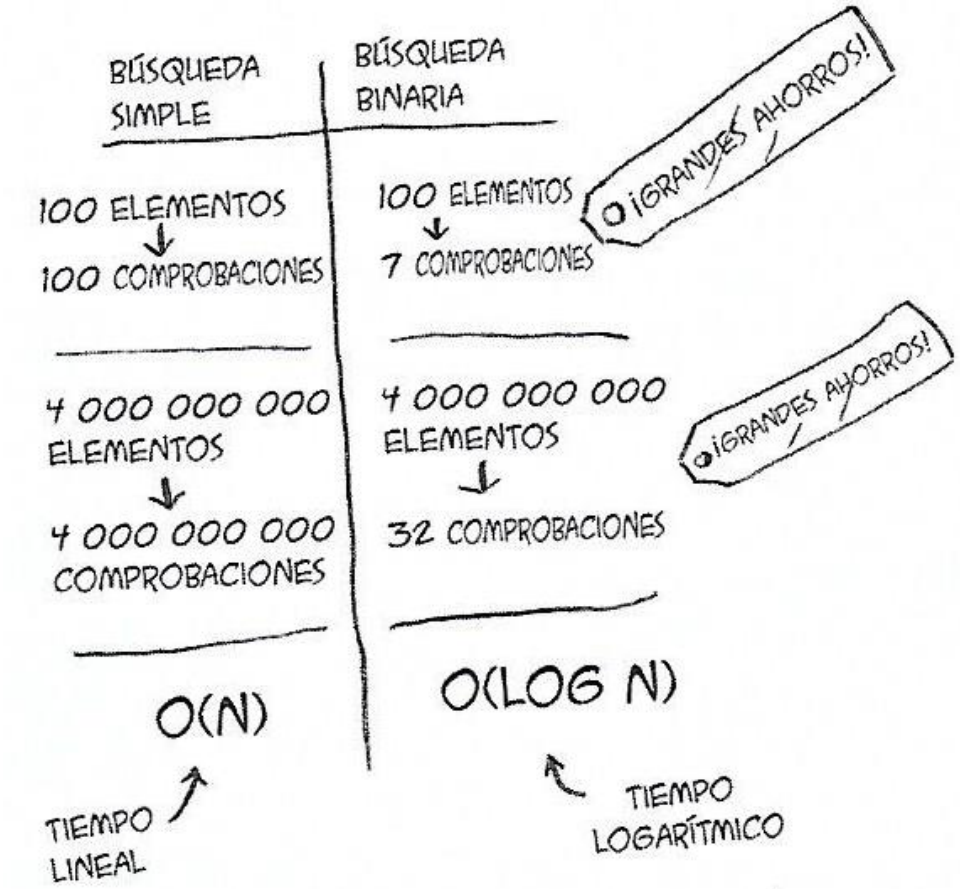
```
my_list = [1,3,5,7,9]
>>> print(busqueda_binaria(my_list,3))
1
>>> print(busqueda_binaria(my_list,-1))
None
>>>
```

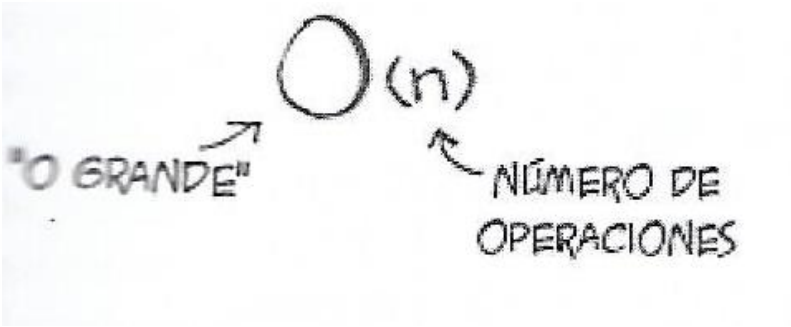
Tiempo de ejecución de un Algoritmo: Notación Big O

Tiempo de Ejecución

Por lo general cuando seleccionamos un algoritmo para resolver un problema lo hacemos porque intentamos optimizar el tiempo de ejecución o el espacio de memoria

Búsqueda binaria o búsqueda simple?





Notación Big O

Tiempo de Ejecución

Ejemplo

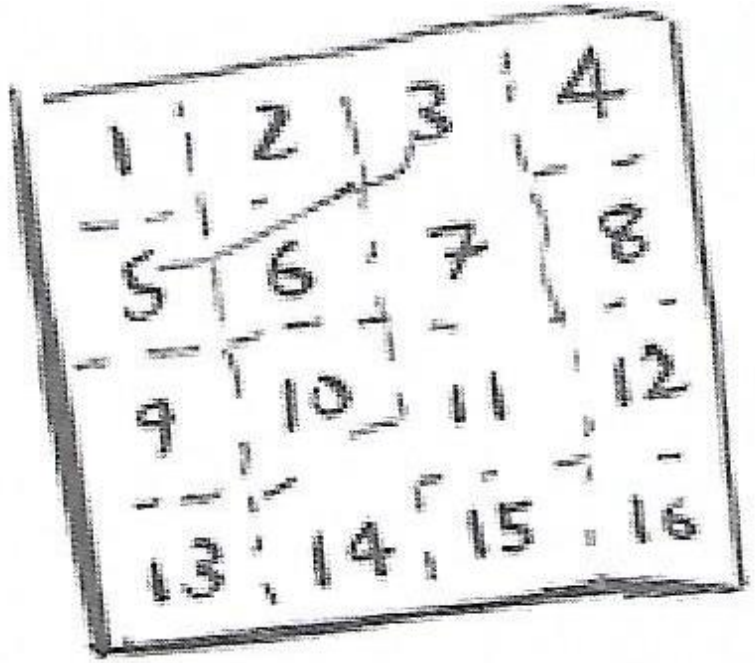
Bob está escribiendo un algoritmo de búsqueda para la NASA. Su algoritmo se activará cuando un cohete esté a punto de aterrizar en la Luna, y ayudará a calcular dónde aterrizar. Y Bob tiene solo 10 segundos para averiguar dónde aterrizar; de lo contrario, el cohete saldrá de su trayectoria. Supongamos que se necesita 1 milisegundo para verificar un elemento. No obstante tiene que verificar mil millones de posiciones donde aterrizar. ¿Cúanto tomará con búsqueda simple? Y con búsqueda binaria?

	BÚSQUEDA SIMPLE	BÚSQUEDA BINARIA
100 ELEMENTOS	100 MS	7 MS
10,000 ELEMENTOS	10 SEGUNDOS	14 MS
1.000.000.000 ELEMENTOS	11 DÍAS	32 MS

Tiempo de Ejecución

Mas ejemplos

Aquí hay un ejemplo práctico que puede seguir en casa con unos cuantos pedazos de papel y un lápiz. Suponga que tiene que dibujar una cuadrícula de 16 cuadros.



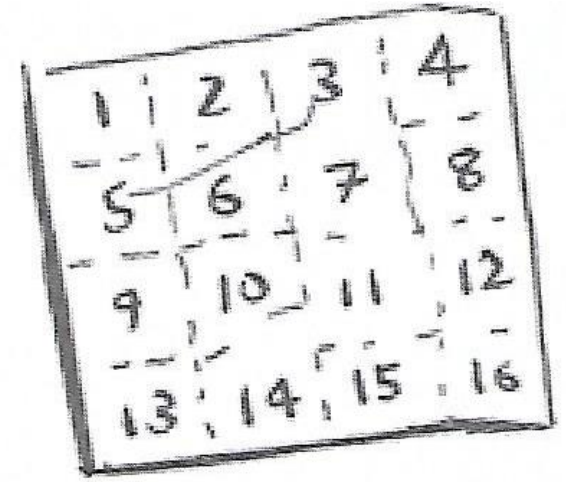
Algoritmo 1

1. Una forma de hacerlo es dibujar 16 cuadros, uno a la vez. Recuerde, la notación Big O cuenta el número de operaciones. En este ejemplo, dibujar un cuadro es una operación. Tienes que dibujar 16 cajas. ¿Cuántas operaciones tomará, dibujando una caja a la vez? Se necesitan 16 pasos para dibujar 16 cuadros. ¿Cuál es el tiempo de ejecución de este algoritmo?

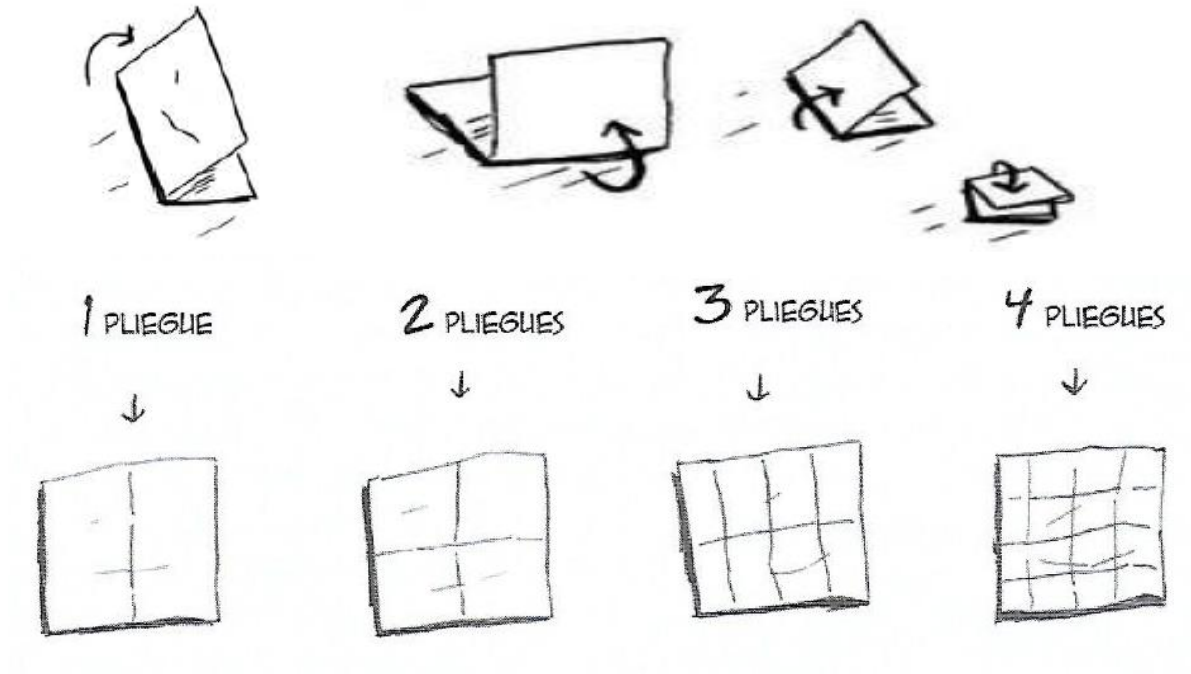
Tiempo de Ejecución

Mas ejemplos

Aquí hay un ejemplo práctico que puede seguir en casa con unos cuantos pedazos de papel y un lápiz. Suponga que tiene que dibujar una cuadrícula de 16 cuadros.



Algoritmo 2



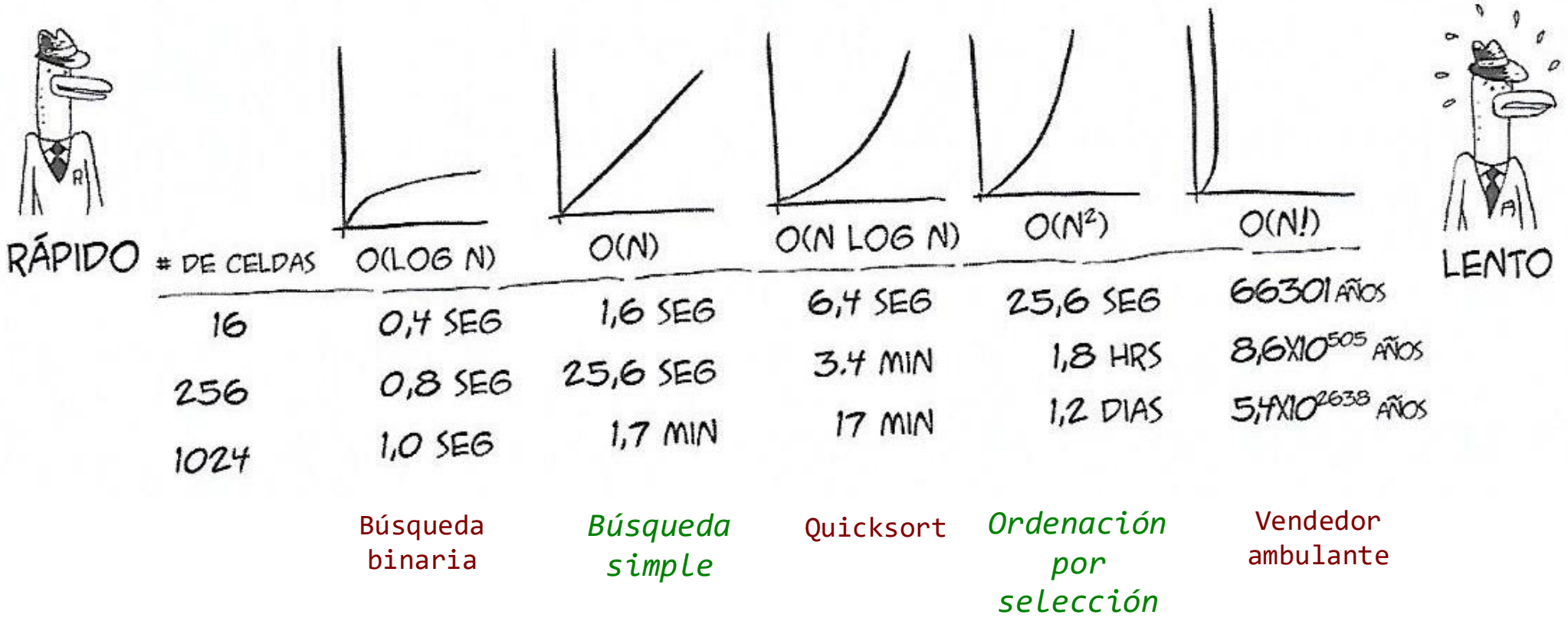
Dobla el papel. En este ejemplo, doblar el papel una vez es una operación. Acabas de crear ¡Dos celdas con esa operación!
Dobla el papel otra vez y otra vez y otra vez.
Despliégalo después de cuatro pliegues, ¡y tendrás una hermosa matriz!
Cada pliegue duplica el número de celdas. ¡Hiciste 16 cajas con 4 operaciones! ¿Cuál es el tiempo de ejecución de este algoritmo?

El Algoritmo 1 necesita un tiempo de ejecución O(n) y el Algoritmo 2 necesita O(log n).

Tiempo de Ejecución

La notación Big O establece el tiempo de ejecución del peor caso

En el ejercicio anterior si suponemos que podemos hacer 10 operaciones por segundo, la siguiente figura nos muestra 5 tiempos Big O que encontraremos en los algoritmos que trabajaremos.



Búsqueda binaria

Búsqueda simple

Quicksort

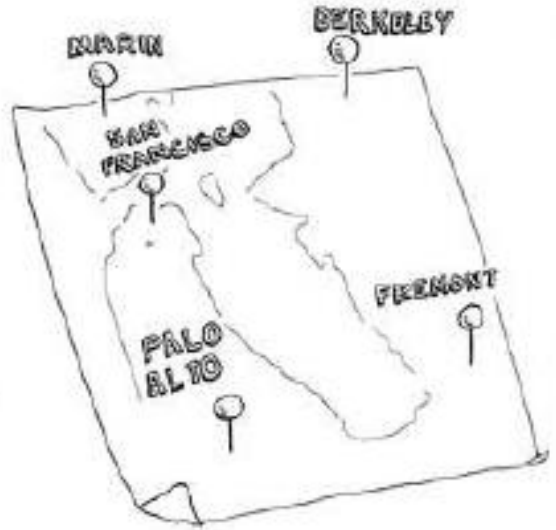
Ordenación por selección

Vendedor ambulante

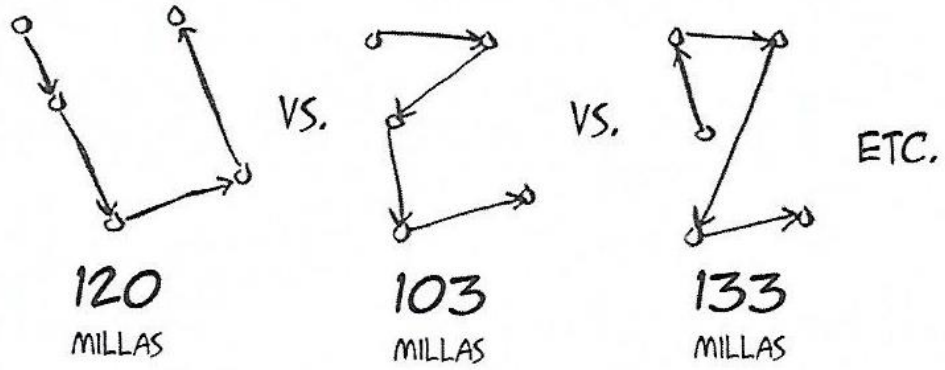
Tiempo de Ejecución

El vendedor ambulante o el problema del viajante de comercio

El vendedor tiene que visitar 5 ciudades



Este vendedor, quien se llamará Opus, quiere ir a todas las ciudades recorriendo la menor distancia posible. Aquí hay una forma de hacerlo: buscar cada posible orden en el que pudiera viajar a las ciudades y seleccionar la menor.



CIUDADES	OPERACIONES
6	720
7	5040
8	40 320
...	...
15	1 307 674 368 000
...	...
30	2 652 52 0 859 812 191 058 636 308 480 000 000

En general, para n elementos, se necesitará n! (n factorial) operaciones para calcular el resultado. Entonces este es el tiempo O (n!).

Hay 120 permutaciones con 5 ciudades, por lo que se necesitarán 120 operaciones para resolver el problema en 5 ciudades. Para 6 ciudades, se necesitarán 720 operaciones (hay 720 permutaciones). ¡Para 7 ciudades, se necesitarán 5.040 operaciones!

https://www.youtube.com/watch?v=iehqnwhKzNc

Análisis de Algoritmos: Análisis Empírico

Análisis de Algoritmos

- Un **algoritmo** es un conjunto de pasos (instrucciones) para resolver un problema.
  - Debe ser robusta y correcta!!!.
  - Debe ser eficiente!!!.
- El análisis de algoritmos nos va a permitir estimar cómo de eficiente es un algoritmo.
- Un problema puede tener diferentes soluciones (algoritmos)
- El análisis de algoritmos nos va a permitir **comparar algoritmos** y **elegir el más eficiente**.

Análisis de Algoritmos

- ¿Cómo estudiar el rendimiento de un algoritmo?:
  - **Complejidad temporal**: estima el tiempo de ejecución requerido por un algoritmo.
  - **Complejidad espacial**: estima el espacio en memoria principal que necesita utilizar el algoritmo.
- Enfoque basado en el tiempo: ¿cómo estimar el tiempo requerido para un algoritmo?
  - Análisis empírico
  - Análisis teórico

Análisis Empírico de Algoritmos

En el análisis empírico, se calcula cuánto tiempo tarda un algoritmo en resolver un problema para distintos tamaños de entrada. Pasos:

1. Implementar el algoritmo
2. Incluir instrucciones que permitan para medir el tiempo de ejecución
3. Ejecutar el programa con entradas de diferentes tamaños y obtener los tiempos de ejecución para cada tamaño
4. Representa gráficamente los resultados: Por ejemplo, importar los resultados a un fichero Excel y mostrarlos en un gráfico XY (dispersión)

Análisis Empírico de Algoritmos

- Dado un número n, desarrolle un método para sumar de 1 a n

1. Implementa el algoritmo:

```java
public static long sum(long n) {
    long result=0;
    for (long i=1; i<=n; i++) {
        result = result + i;
    }
    return result;
}
```

```python
def sum_n(n: int) -> int:
    """returns the sum of the first n numbers"""

    if not isinstance(n, int) or n < 0:
        return None

    result = 0
    for i in range(n+1):
        result += i
    return result


if __name__ == '__main__':
    print("sumN({})={}".format(10, sum_n(10)))
    print("sumN({})={}".format(100, sum_n(100)))
```

Análisis Empírico de Algoritmos

2. Incluye instrucciones para medir el tiempo de ejecución

```java
long startTime = System.currentTimeMillis();

//code lines whose time you want to measure

long endTime = System.currentTimeMillis();
System.out.println("Took "+(endTime - startTime) + " ms");
```

En java Use System.currentTimeMillis() or System.nanoTime().

```python
import time


def sum_n(n: int) -> (int, float):
    """returns the sum of the first n numbers"""

    if not isinstance(n, int) or n < 0:
        return None

    result = 0
    start = time.time()  # returns the time in milliseconds
    for i in range(n+1):
        result += i
    end = time.time()
    return result, end-start  # we also return the time


if __name__ == '__main__':
    for input_size in [10, 100]:
        result_sum, execution_time = sum_n(input_size)
        print("sumN({})={}, time={} ms".format(input_size, result_sum, execution_time))
```

Análisis Empírico de Algoritmos

2. Incluye instrucciones para medir el tiempo de ejecución

```
public static long sum(long n) {
    long startTime = System.nanoTime();

    long result=0;
    for (long i=1; i<=n; i++) {
        result = result + i;
    }

    long endTime = System.nanoTime();
    long total=endTime - startTime;

    System.out.println("sum("+n+")  took "+total + " ns");

    return result;
}
```

Análisis Empírico de Algoritmos

3. Ejecuta el programa con entradas de diferentes tamaños

```
long MAX=1000000000;
for (int n=1000; n<=MAX; n=n*10)
    sum(n);
```

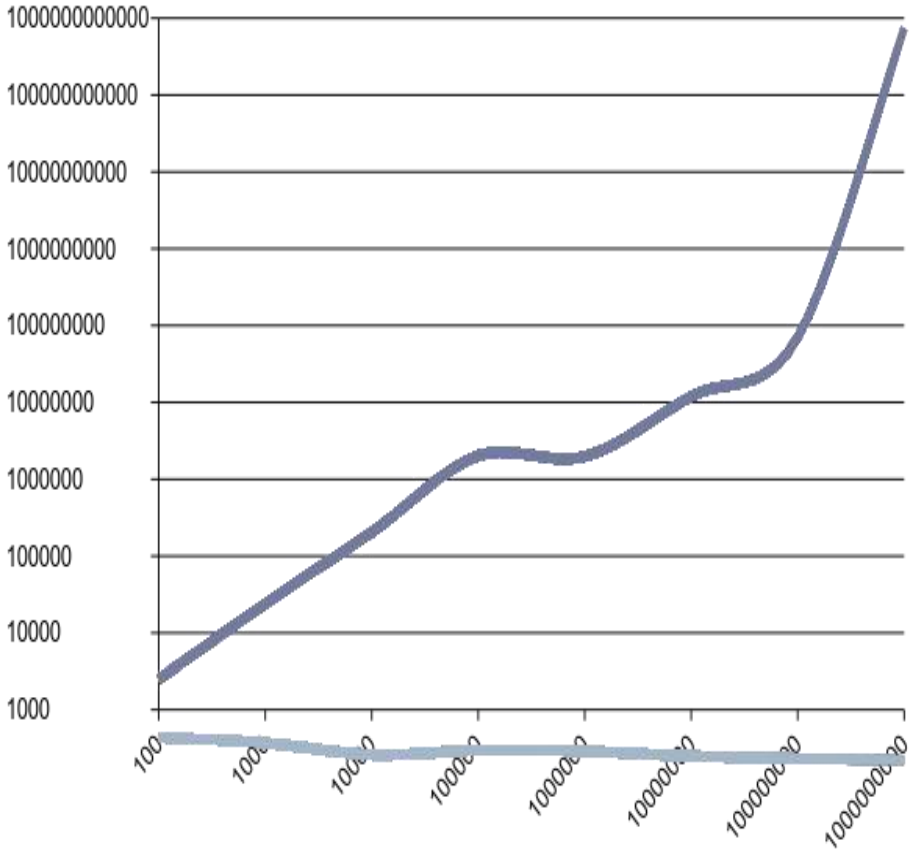
n	tiempo (ns)
100	2485
1.000	23996
10.000	204102
100.000	2022441
1.000.000	1973428
10.000.000	12012791
100.000.000	69984715
1.000.000.000	743431482

```
if __name__ == '__main__':
    for i in range(1, 8):
        input_size = pow(10, i)
        result_sum, execution_time = sum_n(input_size)
        print("sumN({})={}, time={} ms".format(input_size, result_sum, execution_time))
```

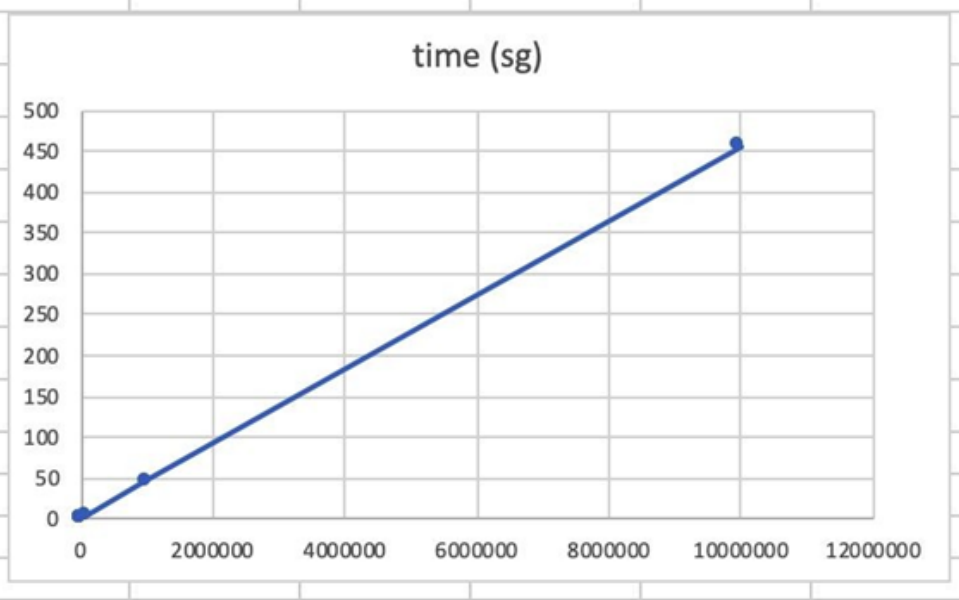
```
sumN(10)=55, time=9.5367431640625e-07 ms
sumN(100)=5050, time=3.814697265625e-06 ms
sumN(1000)=500500, time=4.291534423828125e-05 ms
sumN(10000)=50005000, time=0.0004360675811767578 ms
sumN(100000)=5000050000, time=0.004608869552612305 ms
sumN(1000000)=500000500000, time=0.04788565635681152 ms
sumN(10000000)=50000005000000, time=0.4881730079650879 ms
```

Análisis Empírico de Algoritmos

4. Representa gráficamente los resultados



n	time (sg)
10	0,000953674
100	0,00500679
1000	0,043869019
10000	0,468254089
100000	4,755973816
1000000	47,31917381
10000000	455,6391239

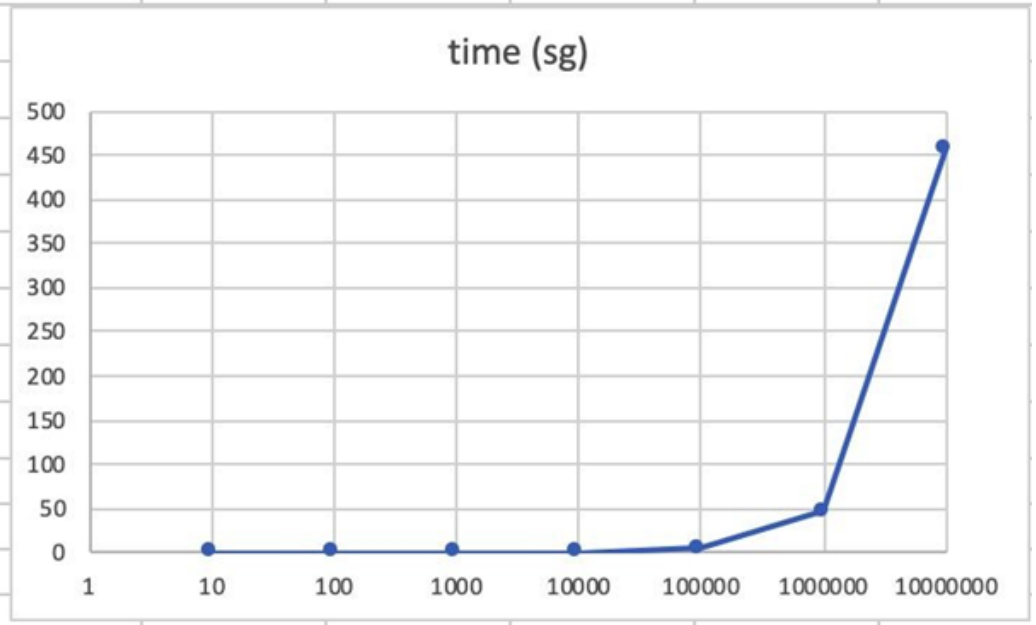


Análisis Empírico de Algoritmos

- Cuando se necesita mostrar rangos muy grandes (como en el ejemplo anterior), transforma tu gráfico a la escala logarítmica
- ¿Cómo se puede hacer un gráfico logarítmico en Excel?

1. En el gráfico XY (dispersión), hacer doble clic en la escala de cada eje.
2. En el cuadro Formato de ejes, seleccionar la pestaña Escala y luego verifique la escala logarítmica

n	time (sg)
10	0,000953674
100	0,00500679
1000	0,043869019
10000	0,468254089
100000	4,755973816
1000000	47,31917381
10000000	455,6391239



Análisis Empírico de Algoritmos

- ¿Existen otros algoritmos que resuelven este problema?

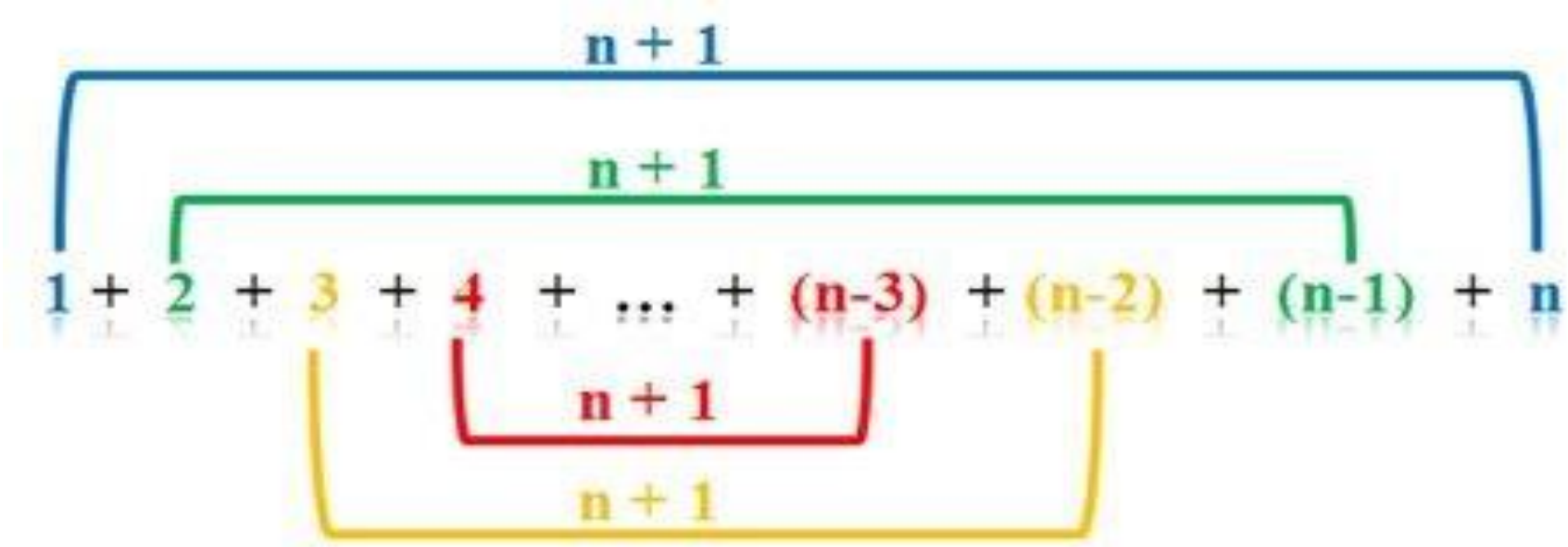
```
public static long sum(long n) {
    long result=0;
    for (long i=1; i<=n; i++) {
        result = result + i;
    }
    return result;
}
```

- La solución de Gauss para sumar números del 1 al n

$$\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

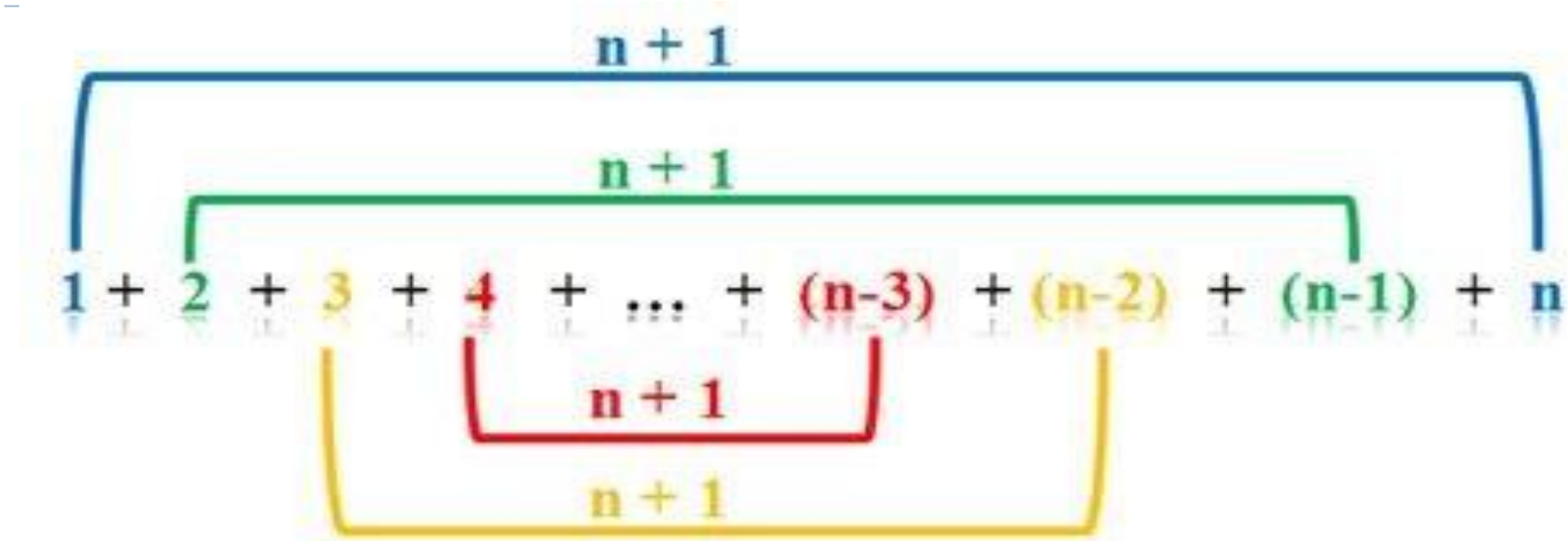
Nota: Se puede encontrar una explicación fácil en:
http://mathandmultimedia.com/2010/09/15/sum-first-n-positive-integers/

Análisis Empírico de Algoritmos



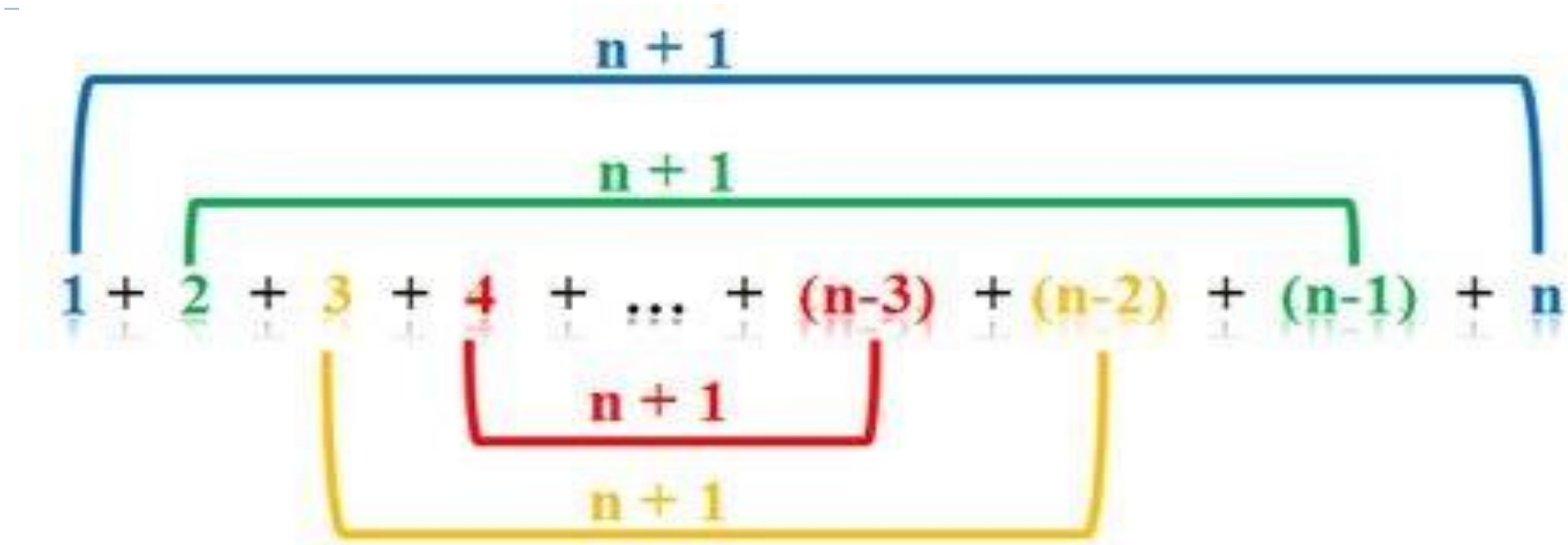
Si n es par, hay n/2 pares
Si n es impar, hay (n-1)/2 pares
Cada par suma n + 1

Análisis Empírico de Algoritmos



Si n es par, hay n/2 pares => $\sum_{k=1}^{n} k = \frac{n(n+1)}{2}$

Análisis Empírico de Algoritmos



Si n es impar, hay (n-1)/2 pares =>

$$\sum_{k=1}^{n} k = (n-1)(n+1)/2 + (n+1)/2 = n(n+1)/2$$

Análisis Empírico de Algoritmos

- Realicemos elu análisis empírico siguiendo los 4 pasos anteriores

1) Implementa el algoritmo

2) Incluye instrucciones para medir el tiempo:

```java
public static long sumGauss(long n) {
    long startTime = System.currentTimeMillis();

    long result=n*(n+1)/2;

    long endTime = System.currentTimeMillis();
    long total=endTime - startTime;

    System.out.println("sum("+n+")  took "+ total + " ms");

    return result;
}
```

```python
def sum_gauss(n: int) -> (int, float):
    """returns the sum of the first n numbers, by
    applying Gauss sum"""

    if not isinstance(n, int) or n < 0:
        return None

    start = time.time()
    result = n*(n+1) / 2
    end = time.time()
    # we return the result and the execution time in seconds
    return result, (end-start)*1000
```

Análisis Empírico de Algoritmos

3) Ejecuta el programa para entradas de tamaños diferentes

```
long MAX=1000000000;
for (int n=100; n<=MAX; n=n*10)
    sumGauss(n);
```

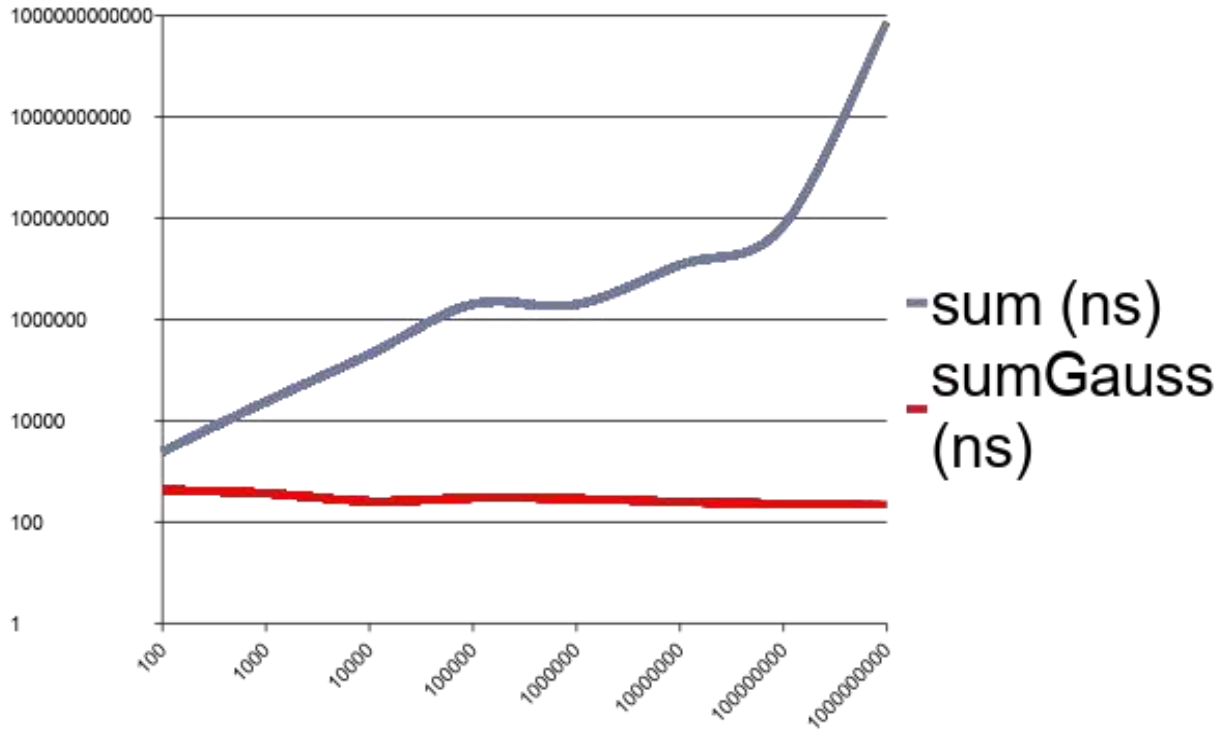
```
if __name__ == '__main__':
    for i in range(1, 8):
        input_size = pow(10, i)
        result_sum, execution_time = sum_gauss(input_size)
        print("sum_gauss({})={}, time={} sg".format(input_size, result_sum, execution_time)
```

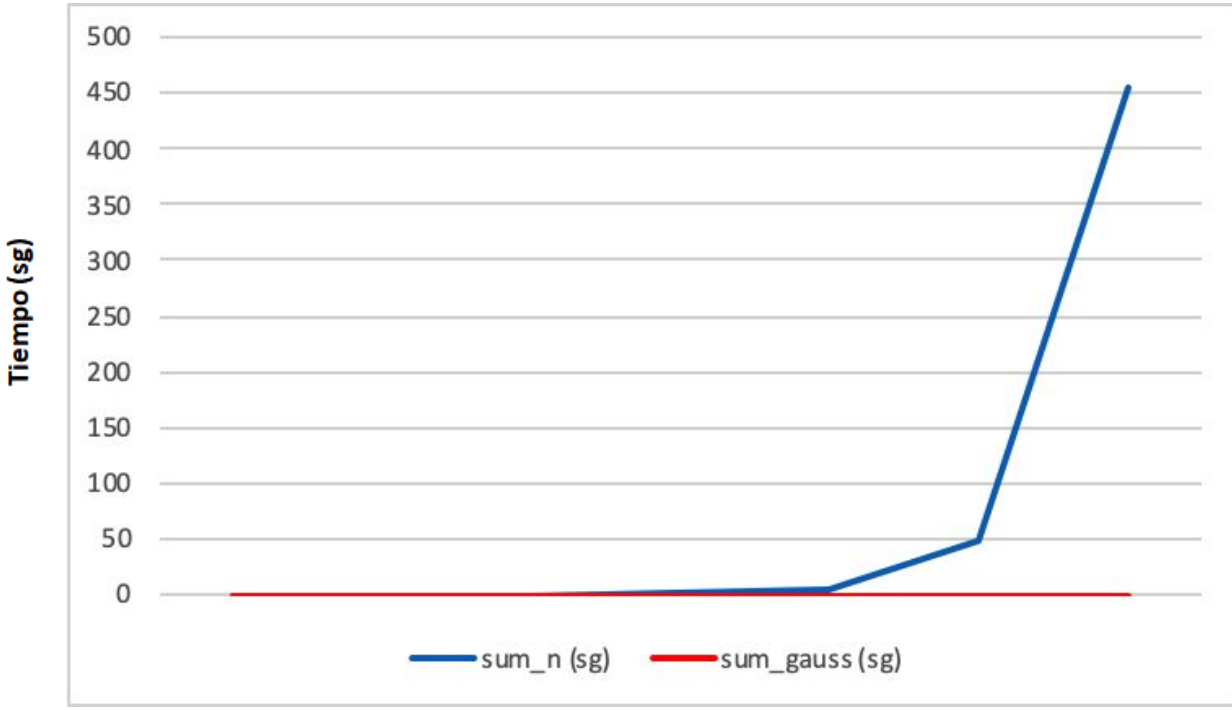
n	tiempo (ns)
100	436
1.000	371
10.000	259
100.000	298
1.000.000	290
10.000.000	250
100.000.000	233
1.000.000.000	222

```
sum_gauss(10)=55.0, time=0.0 ms
sum_gauss(100)=5050.0, time=0.0011920928955078125 ms
sum_gauss(1000)=500500.0, time=0.0 ms
sum_gauss(10000)=50005000.0, time=0.0 ms
sum_gauss(100000)=5000050000.0, time=0.0 ms
sum_gauss(1000000)=500000500000.0, time=0.0 ms
sum_gauss(10000000)=50000005000000.0, time=0.0009536743164062 5 ms
```

Análisis Empírico de Algoritmos

3) Representa gráficamente los resultados





La gráfica muestra claramente que la solución de Gauss es más eficiente que el primer algoritmo.

Análisis Empírico de Algoritmos

- El análisis empírico nos permite obtener gráficas para razonar sobre el tiempo de ejecución de un algoritmo.

- Estas gráficas nos van a permitir comparar fácilmente el tiempo de ejecución de distintos algoritmos para un mismo problema.

- Sin embargo, algunas **desventajas**:
  - Es necesario implementar los algoritmos.
  - Mismo entorno para comparar los algoritmos.
  - Los resultados podrían no ser representativos para todas las posibles entradas.

Análisis de Algoritmos: Análisis Teórico

Análisis Teórico de Algoritmos

- Se puede usar sólo pseudocódigo sin necesidad de ejecutarse en un entorno de programación.
- Consiste en obtener la **función T(n)**, que representa el **número de operaciones** que deben ser ejecutadas para una **entrada de tamaño n**.
- Al ser una función, es posible considerar todas las posibles entradas (tamaños).
- Como no se miden tiempos de ejecución, es independiente del entorno Hardware/Software.

Análisis Teórico – función temporal

- ¿Cómo contamos operaciones/instrucciones en un algoritmo?.
- En cualquier algoritmo, nos podemos encontrar con los siguientes tipos de operaciones:
  - Operaciones primitivas
  - Operaciones bucles
  - Operaciones selectivas o condicionales (if)

Análisis Teórico – Operaciones primitivas

- Las operaciones primitivas toman una cantidad constante de tiempo
- Ejemplos:
  - Declaración de una variable: *int x;*
  - Evaluar una expresión aritmética: *x+3*
  - Asignar un valor a una variable: *x=2*
  - Indexar un elemento en un array: *vector[3]*
  - Llamar a un método: *sumGauss(n)*
  - Retorno de un método: *return x*
  - Evaluar una expresión lógica: *0 < i < index*

Este tipo de operaciones se van a contabilizar como una operación.

Análisis Teórico – Operaciones primitivas

- ¿Qué ocurre si una operación primitiva se puede descomponer en varias operaciones primitivas?

  return vector[3] + vector[5]

- En este caso, podríamos contar:
  - 1 operación por acceder a vector[3]
  - 1 operación por acceder a vector[5]
  - 1 operación por la operación de suma
  - 1 operación por el return.
- Es decir, en total tendríamos cuatro operaciones.

Análisis Teórico – Operaciones primitivas

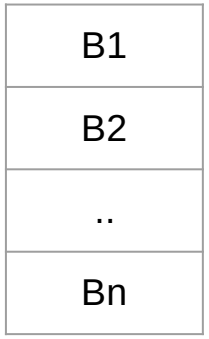
Operación	**# operaciones primitivas**
x = 2	1
x = y + 3	2 (1 suma + 1 asignación)
return a[0] + 1	3 (1 acceso + 1 suma + 1 return)
return node is None or node.elem >5	5 (1 node is None + 1 obtener node.elem + 1 node.elem >5 + 1 or + 1 return)
print(“Hola”)	1

Análisis Teórico – Operaciones primitivas

- Por simplificar el cálculo de la función T(n), siempre podríamos contabilizar este tipo de operaciones primitivas (que se componen de otras primitivas) como una única operación (veremos más adelante por qué podemos hacer esta simplificación).

- La función temporal de una operación primitiva siempre tiene un valor constante, que no depende de n, el tamaño de la entrada.

- Si nuestro algoritmo se compone de varios bloques, su función T(n), será la suma de las funciones T(n) de cada bloque.



$$T(n)=T(B1)+T(B2)+...+T(Bn)$$

Análisis Teórico

Algorithm *swap*(a,b)	**# operations**
temp=a	1
a=b	1
b=temp	1

T(n) = 1+1+1 = 3

Análisis Teórico - bucle

- La función T(n) para un bucle se calculará como el número de veces que se ejecuta el bucle (número de iteraciones) por la función T(n) del bloque interno B.

while condition: B

for x in … : B

T_{loop} (n) = T(B) * número de iteraciones

Análisis Teórico - bucle

- ¿Cuántas veces se ejecuta el bucle (número de iteraciones)?.

```
for i in range(1,n+1):
    result = result + i
```

- La función range(1, n+1) devuelve los siguientes valores:1,2, …, n. Por tanto, el número de iteraciones será n.
- La función T(n) del bloque interno `result=result+i` es T(n) = 2.
- Por tanto, la función **T(n) = n * 2**.
- En este caso, la función temporal **sí depende del valor de n.**

Análisis Teórico - bucles anidados

- Dado el siguiente bucle anidado:

```
for i in range(n):
    for i in range(n):
        print(i*j)  #T(n) = 2
```

- Su función temporal T(n) es: $T(n) = n * n * 2 = 2n^2$

- Es decir, 2 de la instrucción print(i*j), por el número de iteraciones del bucle interno, por el número de iteraciones del bucle externo.

Análisis Teórico - bucles anidados

- ¿Qué pasa si cada bucle se ejecuta un número distinto de veces?

```
for i in range(n1):
    for i in range(n2):
        print(i*j)
```

- En este caso, vamos a definir **n = max(n_1, n_2),**

- Podemos hacer la siguiente simplificación:

T(n) = n_1 * n_2 * 2 < n * n * 2 = **$2n^2$**

Análisis Teórico - operaciones condicionales

- **If-Else**: Sólo uno de los bloques (B_1, B_2, …B_k) se ejecutará.

```
if condition1:
        B1
elif condition2:
        B2
        ...
else:
        Bk
```

$$T_{if\text{-}else}(n) = \max(T_{B1}(n), T_{B2}(n), \ldots, T_{Bk}(n))$$

Análisis Teórico - operaciones condicionales

- En el ejemplo anterior, estamos ignorando las funciones temporales de las condiciones (normalmente tienen un valor constante que no depende de n).
- Para ser más rigurosos, supongamos que Bj es el bloque con la mayor función temporal.
- Entonces, la función temporal final será:

$$T_{if\text{-}else}(n) = T_{condition1}(n) + \ldots + T_{conditionj}(n) + T_{Bj}(n)$$

Análisis Teórico - operaciones condicionales

```
if opc = "inc":
   n = n + 1                    #B1, TB1(n)= 2
elif opc = "dec":
   n = n - 1                    #B2, TB2(n)= 2
elif opc = "mostrar":
   for i in range(1, n+1):      #B3, TB3(n)= n*1
      print(i)
else:
   print("Error: opc!!")        #B4, TB4(n)= 1
```

- En este ejemplo, el bloque con la mayor función temporal es B_3

Análisis Teórico - operaciones condicionales

```
if opc = "inc":
   n = n + 1                    #B1, TB1(n)= 2
elif opc = "dec":
   n = n - 1                    #B2, TB2(n)= 2
elif opc = "mostrar":
   for i in range(1, n+1):      #B3, TB3(n)= n*1
      print(i)
else:
      print("Error: opc!!")     #B4, TB4(n)= 1
```

- Por tanto, T(n) = 1 (evaluar la primera condición) + 1 (evaluar la segunda condición)+ 1 (evaluar la tercera condición) + n (función temporal B3)

Análisis Teórico - operaciones condicionales

```
if opc = "inc":
   n = n + 1                    #B1, TB1(n)= 2
elif opc = "dec":
   n = n - 1                    #B2, TB2(n)= 2
elif opc = "mostrar":
   for i in range(1, n+1):      #B3, TB3(n)= n*1
      print(i)
else:
   print("Error: opc!!")        #B4, TB4(n)= 1
```

- Por tanto, T(n) = 3 + n
- Más tarde veremos que en realidad T(n) = 3 + n ó T(n) = n (si no hubiéramos tenido en cuenta las condiciones), implican el mismo coste.

Análisis Teórico

- Las funciones de tiempo de ejecución nos permiten comparar algoritmos, sin necesidad de implementarlos.

- Vamos a comparar las funciones de los algoritmos sum_n y sum_gauss (aunque estás ya las tenemos implementadas).

Análisis Teórico de algoritmos

```
public static long sum(long n) {
    long result=0;
    for (long i=1; i<=n; i++) {
        result = result + i;
    }
    return result;
}
```

operaciones
2
2+n+1+2*n
2*n
1

$$T_{Sum}(n) = 5n + 6$$

Este algoritmo requiere 5n + 6 para una entrada de tamaño n

Análisis Teórico de algoritmos

```
def sum_n(n: int) -> int:
    """returns the sum of the first n numbers"""
    result=0                        # 1
    for i in range(1,n+1):          # loop will be executed n times
        result += i                 # 2

    return result                   # 1
```

- Por tanto, T(n) = 2n + 2
- Depende del valor de n.

Análisis Teórico de algoritmos

 # operaciones

```
public static long sumGauss(long n) {
    long result=n*(n+1)/2;
    return result;
}
```

5

1

$$T_{Gauss}(n) = 6$$

La solución de Gauss requiere 6 operaciones para cualquier entrada
La línea long result=n*(n+1)/2 tiene 5 operaciones : 1 por declaración, 1 por asignación, 1 por multiplicación, 1 por suma, 1 por división.

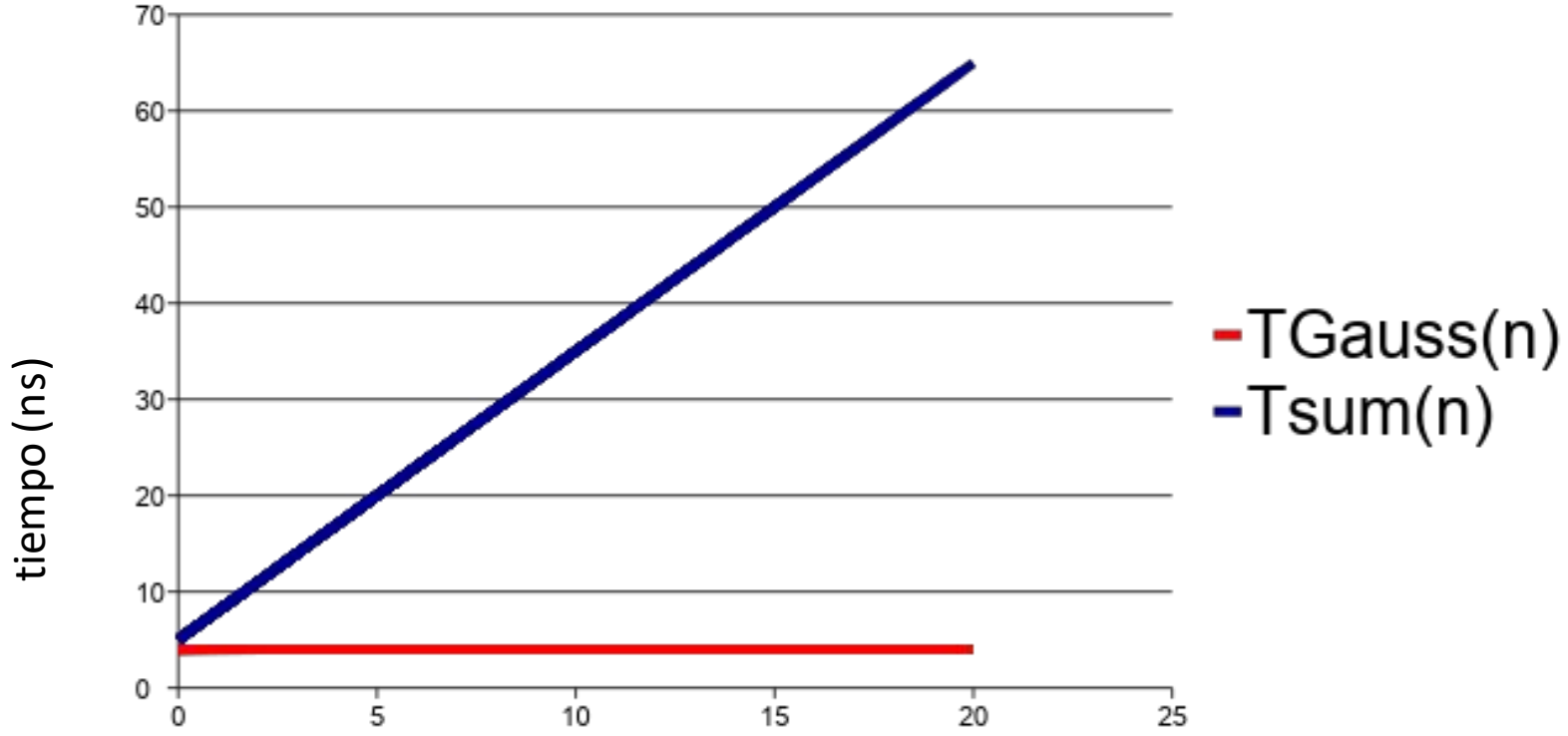
Análisis Teórico de algoritmos

```python
def sum_gauss(n: int) -> int:
    """returns the sum of the first n numbers, by
    applying Gauss sum"""
    return n*(n+1) // 2
```

- En este caso, T(n) = 1 + (suma)
  1 + (multiplicación)
  1 + (división entera)
  1 (return)
  = 4
- T(n) = 4, no depende del valor de n.

Análisis teórico de algoritmos

Los requisitos de tiempo en función del tamaño del problema n

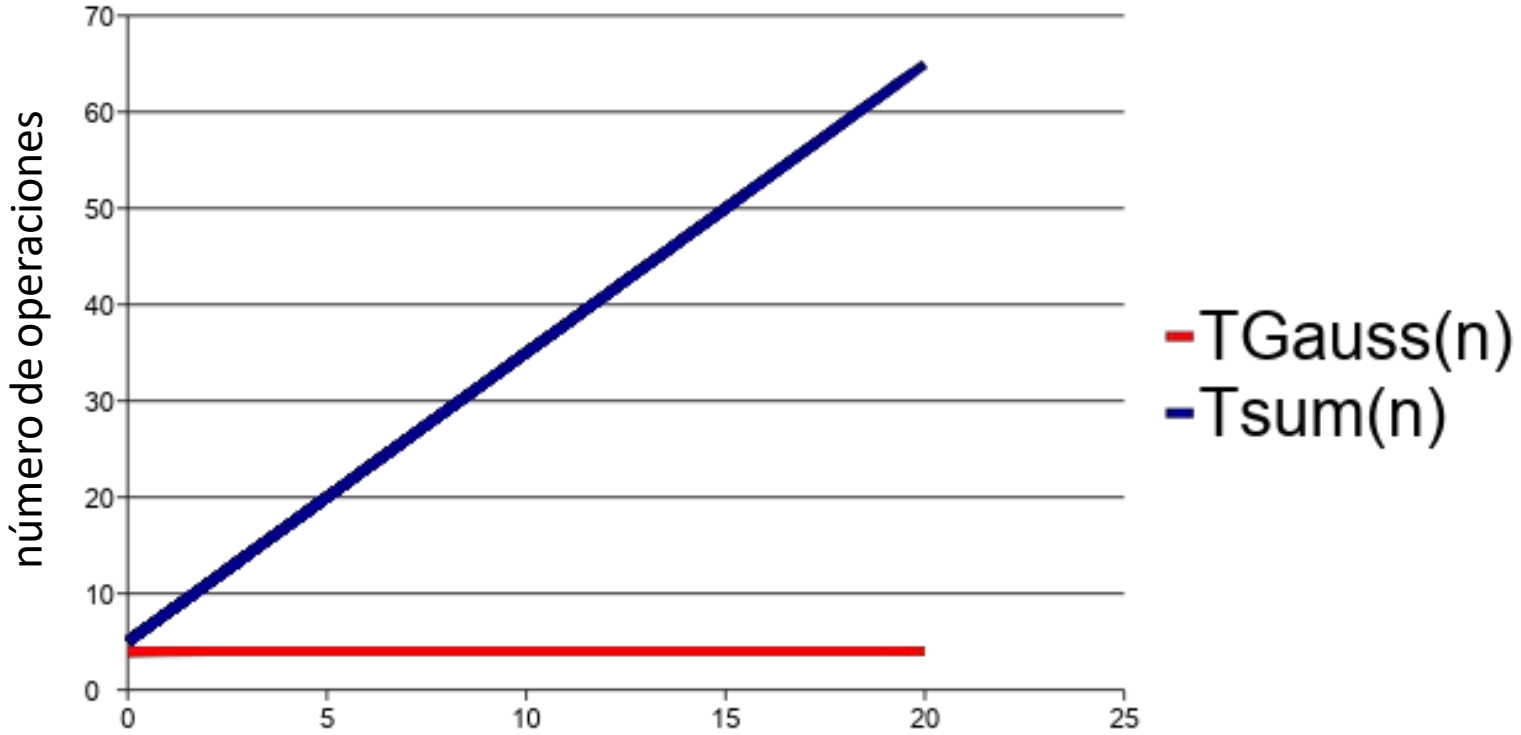


$T_{sum}(n)=5n+6$

$T_{Gauss}(n)=6$

Análisis teórico de algoritmos

Comparativa de las funciones temporales de la suma de gauss y la suma con bucle



$T_{sum}(n)=2n+2$

$T_{Gauss}(n)=4$

Análisis teórico de algoritmos (Python)

- Si suponemos que cada operación tarda un tiempo constante, por ejemplo, c=1 ns, entonces, podemos estimar que la solución de gauss siempre tardará 4 ns, para cualquier tamaño n, mientras que la solución basada en el bucle tardará 2n+2 ns.

- Podemos afirmar que el algoritmo de gauss es más eficiente porque su tiempo de ejecución será constante, mientras que el tiempo de ejecución del algoritmo basado en el bucle, siempre dependerá del valor de n.

Ejercicios propuestos

Problema 1. **Escribir un método, llamado sumPair0, que acepta un array de enteros, vector, como parámetro y devuelve el número de pares (i, j), que cumplen vector [i] + vector [j]= 0. Calcular la función del tiempo de ejecución T (n).**

Problema 2. **Escribir un método, llamado sumTriple0, que acepte un array de enteros, vector, como parámetro y devuelva el número de triples (i, j, k), que cumplen vector [i] + vector [j] + vector [k] = 0 tal que i<j<k. Calcular la función del tiempo de ejecución T (n).**

¿Preguntas?

Universidad Nacional Mayor de San Marcos

Universidad del Perú. Decana de América

Resumiendo y Repasando…

- Búsqueda binaria es mucho más rápida que búsqueda simple
- O(log n) es más rápido que O(n) pero se hace aún más rápido a medida que la lista de elementos en la que estás buscando crece
- La velocidad de un algoritmo no se mide en segundos sino en el crecimiento del número de operaciones necesarias
- Los tiempos de ejecución de escriben en notación O grande

Resumiendo y Repasando…

- Reglas generales para la estimación:
  - Declaraciones consecutivas: el tiempo de ejecución es igual a la suma de los tiempos de ejecución de las distintas instrucciones.
  - Bucles: el tiempo de ejecución de un bucle es como máximo el tiempo de ejecución de las instrucciones dentro de ese bucle multiplicado por el número de iteraciones

Resumiendo y Repasando…

- Reglas generales para la estimación:
  - Bucles anidados: el tiempo de ejecución de un bucle anidado que contiene una instrucción en el bucle más interno es el tiempo de ejecución de la instrucción multiplicado por el producto del tamaño de todos los bucles
  - If/Else: el tiempo de ejecución de una instrucción selectiva es, como máximo, el tiempo de las condiciones y el mayor de los tiempos de ejecución de los bloques de instrucciones asociadas

Universidad Nacional Mayor de San Marcos

Universidad del Perú. Decana de América

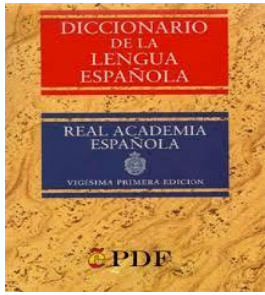
Algoritmos

Entonces los Algoritmos:

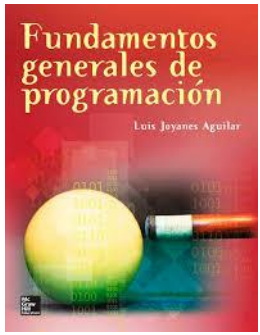
- Sirven para resolver un tipo de problema especifico.
- Son secuencias de pasos concretos.
- Requiere la definición de la entrada y la salida.
- Adecuados para ser ejecutados por un computador

Definiciones de Algoritmos

- Según el Diccionario de la lengua española de la Real Academia Española:
  - ✓ **"Conjunto ordenado y finito de operaciones que permite hallar la solución de un problema"**



- Según Luis Joyanes:
  - ✓ **"Secuencia ordenada de pasos sin ambigüedades que conducen a la solución de un problema dado y expresado en lenguaje natural."**

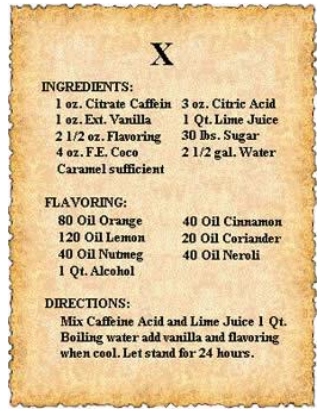


Algoritmo: Conjunto de instrucciones que especifica la secuencia de operaciones a realizar, en orden, para resolver un problema específico.

Algoritmos presentes en la vida diaria

La definición de un algoritmo debe describir tres partes: **Entrada, Proceso y salida.**

Por ejemplo en un algoritmo de receta de cocina se tendrá:
Entrada: Ingredientes y utensilios empleados.
Proceso: Elaboración de la receta en la cocina.
Salida: Terminación del plato (Ejemplo sancochado).



Entradas → PROCESO → Salida

Algoritmos presentes en la vida diaria

Estructura de un Algoritmo: Todo algoritmo consta de tres secciones principales:

Entrada: Es la introducción de datos para ser transformados.

Proceso: Es el conjunto de operaciones a realizar para dar solución al problema.

Salida: Son los resultados obtenidos a través del proceso.

Ejemplo 1:

Se desea elaborar la planilla de pagos de una empresa(sin entrar en detalles de manejo de base de datos).
ENTRADA: *Los datos de cada uno de los empleados, su monto de sueldo básico, reglas de pagos adicionales, reglas de descuentos de sueldos, datos de asistencia y tardanzas, datos de bonos adicionales.*
PROCESO: *La fórmula matemática para calcular los sueldos al revisar cada uno de los empleados será:*
Sueldo básico + pagos adicionales (Comisiones, gratificaciones) – descuentos (considerar: AFP, EsSalud, Tardanzas y faltas).
SALIDA: *reporte de pagos.*

Ejemplo 2:

El cálculo del área del rectángulo se puede dividir en:

- *Entrada de datos (altura, base)*
- *Proceso: Cálculo del área (= base x altura)*
- *Salida de datos (área)*

Definición de Algoritmo

Un Algoritmo es un conjunto ordenado de pasos ejecutables y no ambiguos, que definen un proceso finito con un fin determinado.

Hacer una lista de todos los enteros positivos

La ejecución de cada paso de un algoritmo no requiere de ninguna capacidad creativa

Esta sentencia no ejecutable, no podría ser parte de un algoritmo

Es sólo seguir las instrucciones que se proporcionan

Se muestran 6 características:

PRECISO

Cada paso debe estar especificado con claridad, sin ambigüedad.

FINITO

Al realizar seguimiento del algoritmo, debe finalizar, es decir, debe tener un número finito de pasos.

DEFINIDO

Si se sigue varias veces el algoritmo, ingresando los mismos datos, se debe obtener los mismos resultados.

EFICACIA

Todas las operaciones a realizar deben ser suficientemente básicas.

ENTRADA

El algoritmo tiene cero o más entradas.

SALIDA

Un algoritmo tiene una o más salidas.

¿En qué sentido no se ajustan a la definición técnica de algoritmo los pasos descritos por la siguiente lista de instrucciones?
Paso 1. Sacar una moneda del bolsillo y ponerla sobre la mesa
Paso 2. Volver al paso 1.

Descubrimiento de Algoritmos

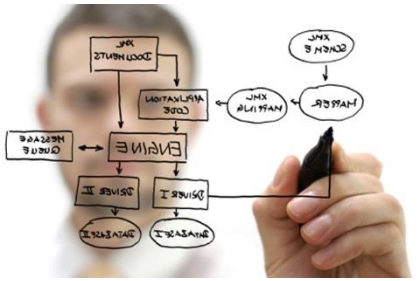
El descubrimiento de algoritmos suele ser el paso más desafiante en el proceso de desarrollo de software. Después de todo, descubrir un algoritmo para resolver un problema requiere encontrar un método de resolución de ese problema. Por tanto, comprender cómo se descubren los algoritmos equivale a comprender el proceso de resolución de problemas.

Resolución de problemas

El acto de encontrar una solución a una pregunta desconcertante, angustiosa, molesta o todavía sin respuesta

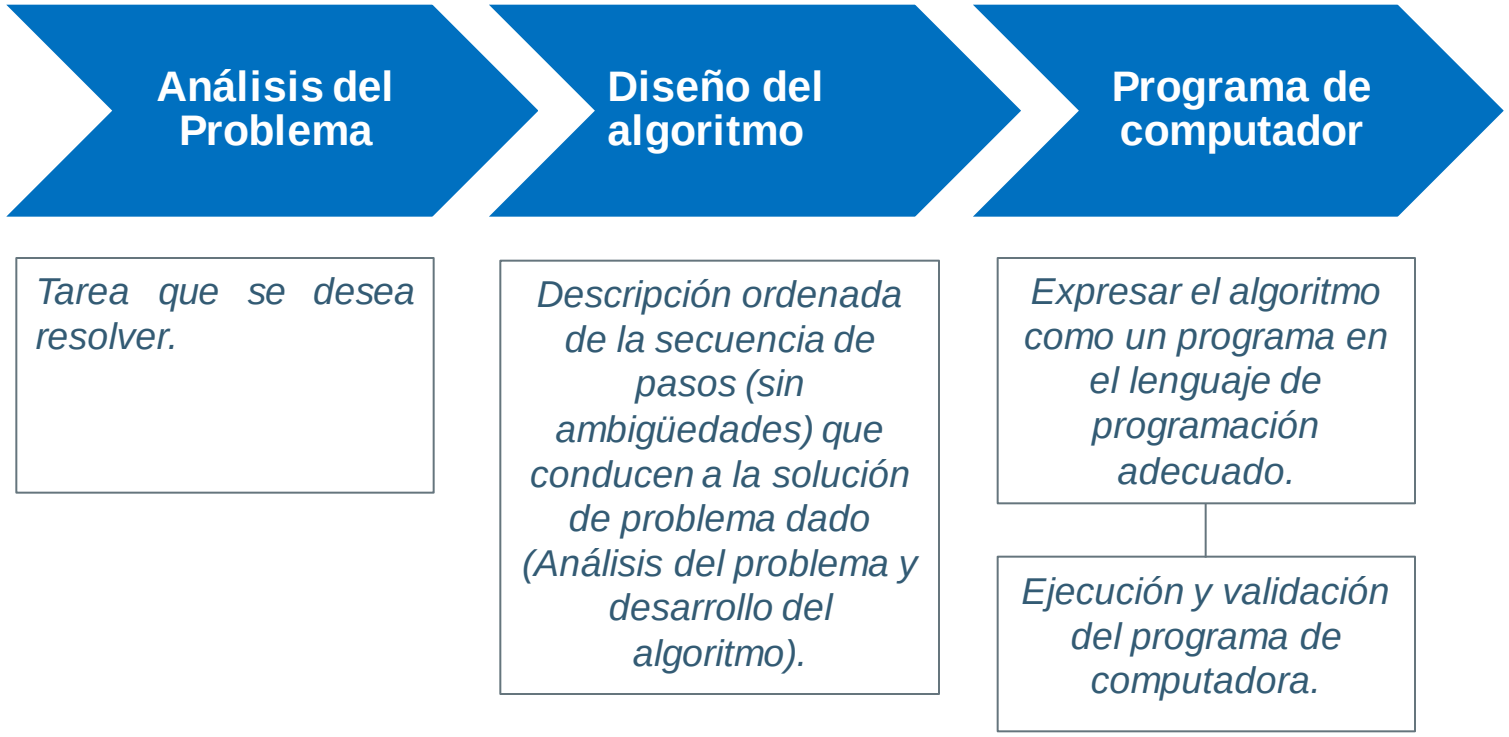
Fases en el desarrollo de un Algoritmo

Análisis del problema → **Diseño del algoritmo** → **Implementación del algoritmo**



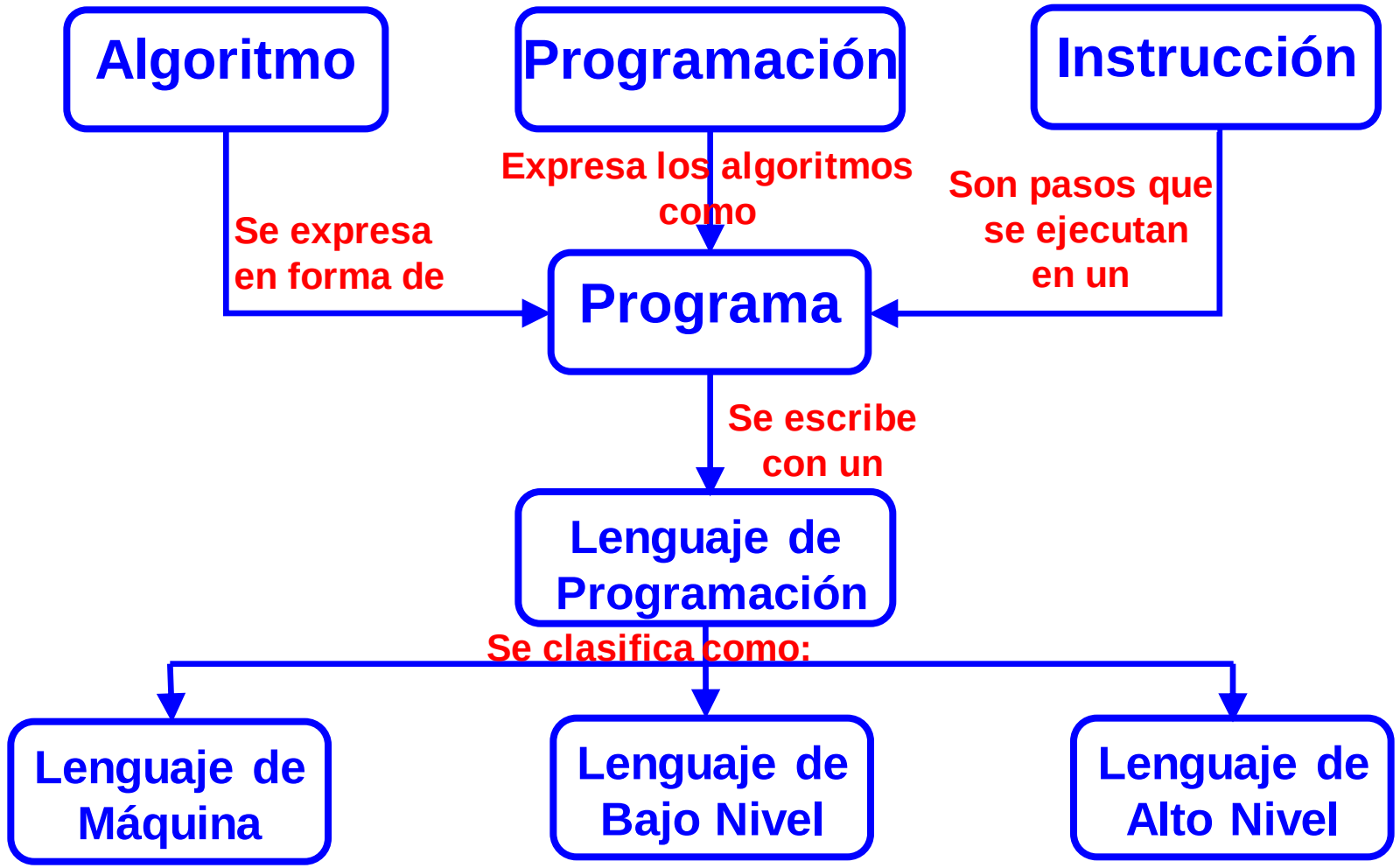
Algoritmos presentes en la vida diaria

Metodología de programación: Consiste en la metodología aplicada para la resolución de problemas mediante programas.



VIDEOS SOBRE ALGORITMOS: https://www.youtube.com/watch?v=8shdU_ABT5w

Implementación del Algoritmo: Lenguajes de Programación



Principales lenguajes utilizados en la actualidad:
- Lenguaje de máquina.
- Lenguaje de bajo nivel.
- Lenguaje de alto nivel

Para implementar el algoritmo en una computadora el algoritmo ha de expresarse en una forma que recibe el nombre de programa.
Un programa se escribe en un lenguaje de programación.

INSTRUCCIÓN

Es cada uno de los pasos que se ejecutan en el programa.

PROGRAMACIÓN

Actividad que consiste en expresar un algoritmo en forma de programa.