Carpeta de Investigación: Análisis de Algoritmos en Python

Análisis de la eficiencia algorítmica en Python

Alumnos:
Andrés Lettieri (andreslettieri@yahoo.com.ar)
Enzo Leiva (enzomauleiva@gmail.com)

Materia: Programación I
Profesor: Alberto Cortez

Fecha de Entrega: 9 de junio de 2025

Introducción

El algoritmo es una secuencia de pasos ordenados que resuelven correctamente un problema o realizan una tarea. En programación, los algoritmos permiten automatizar procesos y optimizar recursos.

Marco teórico

El análisis de los algoritmos es un aspecto fundamental en ciencias de la computación que evalúa la performance de los algoritmos y los programas. La eficiencia de un algoritmo evalúa principalmente tiempo de ejecución y uso de memoria

En este trabajo analizaremos dos formas de convertir números binarios a decimales. Una mediante iteración y otra mediante recursividad. La conversión binaria es una operación fundamental en ciencias de la computación, ya que el sistema binario es la base del procesamiento digital.

Para pasar de binario a decimal, se utiliza la fórmula de expansión de potencias de base 2 aplicada a cada bit

La versión iterativa del algoritmo utiliza un ciclo para recorrer la cadena binaria, acumulando el valor decimal a medida que procesa cada dígito.

La versión recursiva divide el problema en subproblemas más pequeños, resolviéndolos mediante llamadas sucesivas a la misma función.

Si bien ambas estrategias son funcionales, su rendimiento difiere en escenarios de gran escala. Python impone un límite a la profundidad de llamadas recursivas, lo cual puede ocasionar errores cuando se procesan grandes volúmenes de datos. Por este motivo, la eficiencia algorítmica, entendida como la relación entre los recursos consumidos y la escala del problema, es un criterio clave en la elección de una solución

El análisis comparativo de ambas estrategias permite ilustrar los impactos prácticos de decisiones de diseño algorítmico, y justifica la importancia de considerar no solo la corrección funcional, sino también el comportamiento temporal de un algoritmo frente a grandes entradas de datos.

Caso practico

Se analizan y comparan dos algoritmos que transforman un numero binario en decimal aplicando cargas masivas sobre los mismos.

Algoritmo #1: conversión iterativa

```
def binario_a_decimal_for(binarios):
        for b in binarios:
                decimal = 0
                for i in range(len(b)):
                        bit = int(b[i])
                        potencia = len(b) - 1 - i
                        decimal += bit * (2 ** potencia)
```

Algoritmo #2: conversión recursiva

```
def binario_a_decimal_recursivo(binario):
        if binario == "":
                return 0
        return int(binario[0]) * (2 ** (len(binario) - 1)) + binario_a_decimal_recursivo(binario[1:])
```

Version del código completa

https://github.com/AndresLettieri/UTN-P1/blob/main/Trabajo%20integrador/TP%20integrador.py

Metodología utilizada

- Análisis teórico de la complejidad de cada algoritmo:

Iterativa: T(n) = (12*m + 2)*n:

```
def binario_a_decimal_for(binarios):
    for b in binarios: #n veces
        decimal = 0 # 1 asignacion
        for i in range(len(b)): # m veces,  1 len
            bit = int(b[i]) # 1 asignacion, 1 conversión, 1 acceso
            potencia = len(b) - 1 - i # 1 asignacion, 1 len, 2 resta
            decimal += bit * (2 ** potencia) # 1 asignacion, 1 suma, 1 multiplicación, 1 potencia
        resultados_for.append(decimal) # 1 append
```

Recursiva: T(n) = (11*m + 2)*n

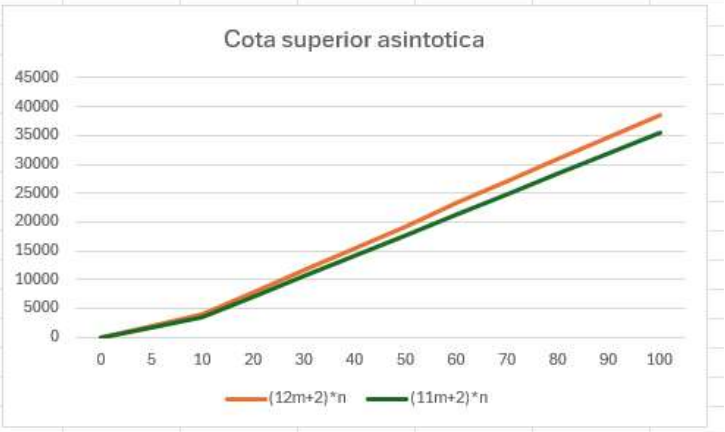
```
def convertir_recursivo(binarios):
    for b in binarios: # n veces
        resultados_recursivo.append(binario_a_decimal_recursivo(b)) # 1 append, 1 llamada a funcion


def binario_a_decimal_recursivo(binario):
    if binario == "": # 1 comparación
        return 0 # 1 retorno
    return int(binario[0]) * (2 ** (len(binario) - 1)) + binario_a_decimal_recursivo(binario[1:])
    #1 retorno, 1 conversion, 1 acceso a lista, 1 multiplacacion, 1 potencia, 1 len, 1 resta, 1 llamada recursiva, 1 acceso a lista
```

Analizando ambas funciones podemos asumir que el comportamiento es lineal y similar entre las dos. Con lo cual, podemos decir que son funciones asintóticas.

En el siguiente grafico podemos apreciar como la función (12m+2)*n es de orden superior.

n	(12m+2)*n	(11m+2)*n
0	0	0
5	1930	1770
10	3860	3540
20	7720	7080
30	11580	10620
40	15440	14160
50	19300	17700
60	23160	21240
70	27020	24780
80	30880	28320
90	34740	31860
100	38600	35400



Aclaración. Para el calculo del grafico se tomo m como valor de 32 ya que el código utilizado ofrece conversión de binarios en 32 bits.

BigO.

Tomamos TA(n) = (12m+2).n => 12mn+2n y TB(n)= (11m+2).n => 11mn+2n.

TA(n) = 12mn (Se elimina 2n por ser de menor magnitud)

TA(n) = O(mn) (Se elimina el 12 por ser una constante)

TB(n) = 11mn (Se elimina 2n por ser de menor magnitud)

TB(n) = O(mn) (Se elimina el 11 por ser una constante)

Con esta comparativa podemos identificar que el tiempo de ejecución para ambos algoritmos crece de forma lineal en grandes escalas.

- Todo el código fue realizado en un archivo Python utilizando VS Code como IDE y GitHub para versionado. Para la medición del tiempo se utilizó la librería *time* y para generar números aleatorios librería *random*.

Resultados obtenidos

10000 elementos - Iterativo: 0.10518122s - Recursivo: 0.13312197s

30000 elementos - Iterativo: 0.31265044s - Recursivo: 0.39998531s

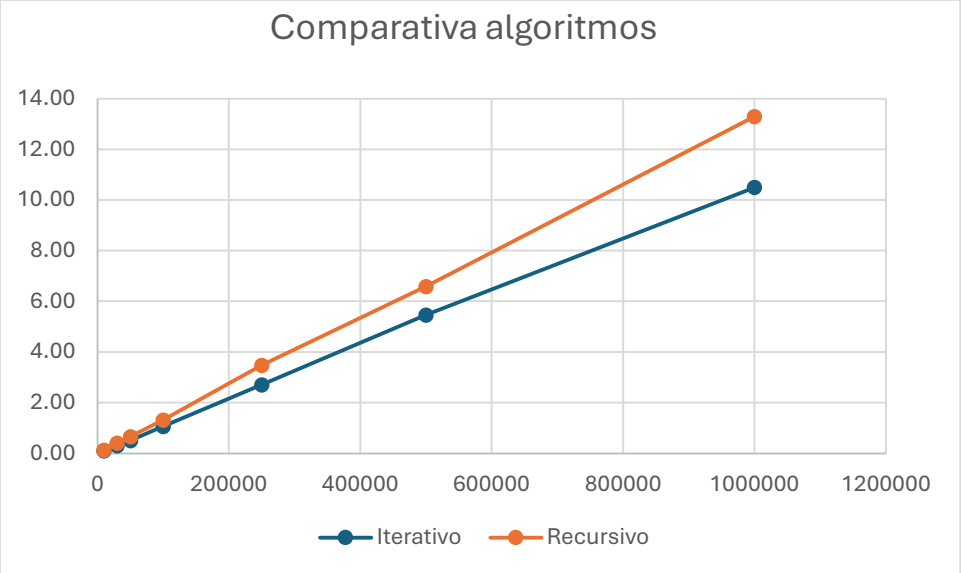
50000 elementos - Iterativo: 0.51416302s - Recursivo: 0.66601443s

100000 elementos - Iterativo: 1.06579399s - Recursivo: 1.32428527s

250000 elementos - Iterativo: 2.69909549s - Recursivo: 3.48016071s

500000 elementos - Iterativo: 5.46410227s - Recursivo: 6.59224916s

1000000 elementos - Iterativo: 10.50524879s - Recursivo: 13.30425715s



- Agregando una función validamos que ambos algoritmos convierten correctamente el binario a decimal.
- El algoritmo por iteración es aproximadamente 30% más rápido.

- La diferencia en tiempos es constante a medida que se van incrementando la cantidad de números a procesar.

Conclusiones:

Se observa que el algoritmo implementado con iteración es más eficiente que el algoritmo recursivo, principalmente porque utiliza menos memoria y evita el uso de la pila de llamadas. En el proceso iterativo, el programa procesa cada bit del número binario secuencialmente y acumula directamente el resultado decimal, sin requerir almacenamiento adicional entre pasos.

En cambio, el algoritmo recursivo necesita conservar en memoria el estado de cada llamada hasta llegar al caso base. Esto genera un mayor consumo de recursos, especialmente con binarios largos.

Aunque ambos algoritmos presentan una complejidad asintótica de O(mn), la implementación recursiva es menos eficiente en la práctica debido al uso intensivo de la pila de llamadas que declina en un uso de memoria mayor a la versión recursiva.

Bibliografía

- GeeksforGeeks. *Analysis of Algorithms*. https://www.geeksforgeeks.org/analysis-of-algorithms/
- Tutorialspoint. Convert *Binary to Decimal*.
- https://www.tutorialspoint.com/how-to-convert-binary-to-decimal
- Python Software Foundation. *System-specific parameters and functions*. https://docs.python.org/3/library/sys.html#sys.getrecursionlimit
- Real Python. (2022). *Understanding Big O Notation*. https://campus.datacamp.com/courses/data-structures-and-algorithms-in-python/work-with-linked-lists-and-stacks-and-understand-big-o-notation?ex=5

- ChatGPT, consultas sobre uso de librerías. https://chatgpt.com/