

Análisis de Algoritmos en Python

Comparación de Eficiencia: Recursión vs Slicing en Detección de Palíndromos

Alumnos:

Bruno Ludueña – brunoluduenaa@abc.gob.ar

Pablo Mariasch – pablomariasch85@gmail.com

Comisión: 16.

Materia: Programación I.

Profesora: Cinthia Rigoni.

Fecha de Entrega: 9 de junio de 2025.

Índice

1. Introducción
2. Marco Teórico
3. Caso Práctico
4. Metodología Utilizada
5. Resultados Obtenidos
6. Conclusiones
7. Bibliografía
8. Anexos

1. Introducción

El análisis de algoritmos es una disciplina fundamental en la programación, ya que permite evaluar y comparar diferentes métodos de resolución de problemas con el fin de seleccionar los más eficientes y adecuados para distintos contextos. En este trabajo se abordará la comparación de eficiencia entre dos enfoques para resolver un problema clásico: la detección de palíndromos. Para ello, se seleccionaron dos estrategias diferentes para resolver el problema: una recursiva y otra que aprovecha el slicing de cadenas.

La elección de este tema se debe a que permite observar claramente cómo distintas estrategias algorítmicas pueden afectar el rendimiento de un programa, especialmente a medida que se incrementa el tamaño de los datos. Además, el problema de detección de palíndromos es lo suficientemente simple como para centrarse en el análisis de eficiencia, sin que la complejidad del problema interfiera en la comparación.

La relevancia del tema en programación está en la necesidad de optimizar algoritmos para lograr aplicaciones más rápidas y escalables, especialmente frente a grandes volúmenes de datos.

El objetivo principal es comparar el rendimiento de ambos algoritmos con diferentes volúmenes de datos, para determinar cuál resulta más eficiente y en qué contextos.

2. Marco Teórico

¿Qué es el Análisis de Algoritmos? Es el estudio de cómo se comportan los algoritmos en función del tiempo que tardan en ejecutarse a medida que crece la cantidad de datos de entrada. Su objetivo principal es comparar distintas soluciones a un mismo problema para elegir la más eficiente según el contexto.

Conceptos Clave:

- **Notación Big-O:** es el análisis fundamental en programación para garantizar la escalabilidad y optimización de las aplicaciones. Representa el crecimiento de un algoritmo en función de la entrada.

Notación Big-O	Nombre	Descripción breve	Ejemplo de algoritmo
$O(1)$	Constante	El tiempo de ejecución no depende del tamaño de entrada	Acceso a un elemento en un array
$O(\log n)$	Logarítmica	Crece lentamente incluso si la entrada aumenta mucho	Búsqueda binaria
$O(n)$	Lineal	El tiempo crece proporcionalmente al tamaño de entrada	Recorrido de una lista
$O(n \log n)$	Linealítmica	Más rápido que cuadrático, pero más lento que lineal	Algoritmos de ordenamiento eficientes (Merge Sort)
$O(n^2)$	Cuadrática	Tiempo de ejecución crece rápidamente con la entrada	Algoritmos de ordenamiento simples (Bubble Sort)
$O(2^n)$	Exponencial	Tiempo se duplica con cada aumento de entrada	Problemas combinatorios (Fuerza bruta)
$O(n!)$	Factorial	Muy ineficiente; crece extremadamente rápido	Permutaciones de n elementos

- **Tiempo de ejecución real:** es una evaluación empírica que se realiza cronometrando el tiempo que tarda un algoritmo en ejecutarse. En Python, esto se puede hacer con el módulo **time**.
- **Palíndromo:** una palabra o frase que se lee igual de izquierda a derecha que de derecha a izquierda, ignorando mayúsculas, espacios y tildes.
- **Recursión:** es una técnica de programación en la cual una función se llama a sí misma para resolver un problema más pequeño en cada paso.
- **Slicing:** en esta técnica se utiliza un paso negativo (-1) para recorrer la secuencia del final al inicio, generando una copia invertida. Es más concisa y suele ofrecer mejor rendimiento gracias a optimizaciones internas del lenguaje.

3. Caso Práctico

Se implementan dos funciones en Python para detectar si una palabra es un palíndromo:

Se analiza la función Recursiva

Esta función resuelve el problema utilizando recursión, es decir, llamándose a sí misma con una versión reducida de la cadena.

```
5  # Algoritmo 1: Recursivo
6  def es_palindromo_recursiva(palabra):
7      palabra = palabra.lower().replace(" ", "")
8      if len(palabra) <= 1:
9          return True
10     if palabra[0] != palabra[-1]:
11         return False
12     return es_palindromo_recursiva(palabra[1:-1])
13
```

- Normalización

Convierte la palabra a minúsculas y elimina los espacios:

`palabra = palabra.lower().replace(" ", "")`

Por ejemplo: "Anita lava la tina" → "anitalavalatina"

- Caso base de la recursión

Si la longitud de la palabra es 0 o 1, es un palíndromo.

`if len(palabra) <= 1:`

`return True`

- Comparación de extremos

Compara el primer y último carácter. Si no coinciden, no es palíndromo.

`if palabra[0] != palabra[-1]:`

`return False`

- Llamada recursiva

Si los extremos son iguales, se repite el proceso con la subcadena sin el primer y último carácter:

`return es_palindromo_recursiva(palabra[1:-1])`

Se analiza la función Slicing

Este segundo enfoque utiliza el operador de slicing de Python (`[::-1]`) para invertir la cadena y compararla con la original.

```
14 # Algoritmo 2: Slicing
15 def es_palindromo_slice(palabra):
16     palabra = palabra.lower().replace(" ", "")
17     return palabra == palabra[::-1]
18
```

- Normalización

`palabra = palabra.lower().replace(" ", "")`

Convierte todo a minúsculas y elimina espacios.

Ejemplo: "Anita lava la tina" → "anitalavalatina"

- Comparación con reverso

`return palabra == palabra[::-1]`

Usa slicing (`[::-1]`) para invertir la palabra y la compara con la original.

Si son iguales, es un palíndromo.

Comparación de complejidad entre ambas funciones

Algoritmo	Tiempo (Big-O)	Comentario
es_palindromo_slice	$O(n)$	Es lineal porque invierte la cadena una vez usando slicing.
es_palindromo_rekursiva	$O(n^2)$	Es cuadrática porque crea una subcadena en cada llamada recursiva.

4. Metodología Utilizada

Se implementó una metodología de análisis empírico basada en las siguientes fases:

- Datos base: Se creó una lista inicial de 20 palabras (10 palíndromos y 10 no palíndromos).
- Escalado: La lista se duplicó en cada iteración, hasta alcanzar 10.240 palabras en la décima.
- Medición: Se usó `time.perf_counter()` para registrar el tiempo total que cada algoritmo tardó en procesar la lista.
- Comparación: En cada iteración se calculó:
 - La diferencia absoluta de tiempo entre ambos algoritmos.
 - La diferencia porcentual para evaluar la ventaja relativa.

Esta metodología permitió recopilar datos empíricos robustos sobre el rendimiento de cada algoritmo, facilitando la extracción de conclusiones sobre su eficiencia y escalabilidad.

5. Resultados Obtenidos

- Ambos métodos funcionaron correctamente, identificando de manera precisa si una palabra es palíndromo o no.
- A medida que crece la cantidad de palabras procesadas, el algoritmo basado en slicing demuestra ser significativamente más eficiente que el recursivo.
- En todos los casos, el tiempo de ejecución del método recursivo fue más alto que el del método con slicing.
- La diferencia de tiempo aumenta con el tamaño de la entrada, alcanzando en algunos casos una diferencia de más del 200 % en favor del método con slicing.
- El método recursivo, aunque funcional, genera una sobrecarga que restringe su escalabilidad.
- El slicing en Python aprovecha optimizaciones nativas, haciéndolo ideal para manipular secuencias eficientemente.

6. Conclusiones

Este trabajo permitió profundizar en el uso práctico de la notación Big-O para anticipar y evaluar el rendimiento del código. Al comparar dos enfoques distintos se evidenció claramente cómo la elección del algoritmo influye directamente en la eficiencia, sobre todo al trabajar con grandes volúmenes de datos.


Además, el tema estudio es sumamente útil para cualquier proyecto que implique procesamiento de datos, optimización de recursos o aplicaciones en tiempo real, donde la eficiencia es crucial.


Entre las posibles mejoras se podrían incluir pruebas con entradas aún más grandes, el uso de herramientas automatizadas de benchmarking y la comparación con otros lenguajes o paradigmas.


Durante el desarrollo, surgieron algunas dificultades, como errores lógicos en las primeras implementaciones y desafíos al medir el tiempo con precisión. Estos se resolvieron mediante pruebas controladas, revisión del código y el uso de bibliotecas estándar como time para registrar los tiempos de ejecución con mayor exactitud.


En resumen, el trabajo no solo reforzó conceptos fundamentales de programación, sino que también brindó una experiencia valiosa en diseño, evaluación y mejora de algoritmos.

7. Bibliografía

- Documentación oficial Python time:
 <https://docs.python.org/3/library/time.html>

- Operaciones comunes sobre slicing (str, list, etc.):
 <https://docs.python.org/3/library/stdtypes.html#common-sequence-operations>

- Definición de funciones en Python:
 <https://docs.python.org/3/tutorial/controlflow.html#defining-functions>

- Límite de recursión en Python (sys.setrecursionlimit):
 <https://docs.python.org/3/library/sys.html#sys.setrecursionlimit>

8. Anexos

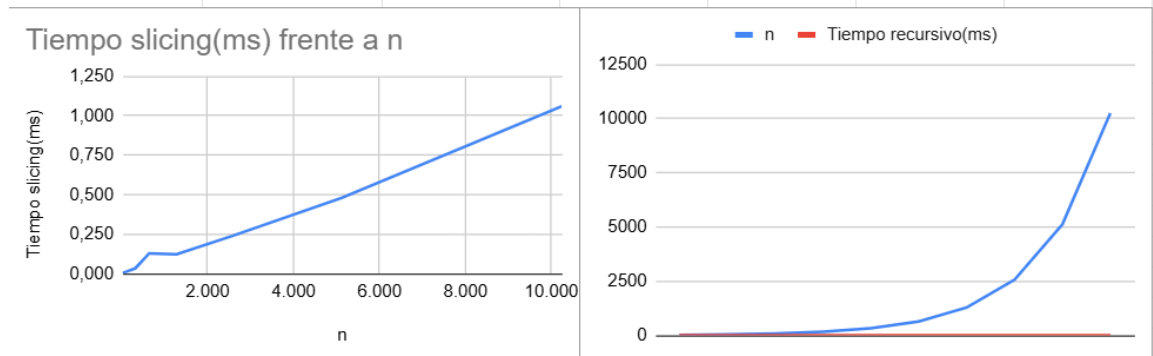
Captura del resultado de ejecución

1/Trabajos Integradores/Análisis de Algoritmos en Python/palindromo.py"				
===== Crecimiento del tiempo vs tamaño de lista (n) =====				
n	Tiempo Recursivo	Tiempo Slicing	Diferencia (s)	Diferencia (%)
20	0.000014 s	0.000006 s	0.000008 s	132.20 %
40	0.000013 s	0.000005 s	0.000008 s	157.14 %
80	0.000022 s	0.000008 s	0.000014 s	164.63 %
160	0.000042 s	0.000016 s	0.000026 s	167.52 %
320	0.000103 s	0.000032 s	0.000071 s	220.50 %
640	0.000337 s	0.000126 s	0.000212 s	168.34 %
1280	0.000349 s	0.000122 s	0.000227 s	186.31 %
2560	0.000655 s	0.000236 s	0.000419 s	177.62 %
5120	0.001302 s	0.000478 s	0.000823 s	172.12 %
10240	0.002611 s	0.001059 s	0.001552 s	146.64 %

Comparación del Tiempo de Ejecución:

Algoritmo Recursivo y Algoritmo Slicing para Detección de Palíndromos.

Tiempo recursivo(ms)	Tiempo slicing(ms)	n				
0,014	0,006	20				
0,013	0,005	40				
0,022	0,008	80				
0,042	0,016	160				
0,103	0,032	320				
0,337	0,126	640				
0,349	0,122	1.280				
0,655	0,236	2.560				
1,302	0,478	5.120				
2,611	1,059	10.240				



Repositorio en GitHub:

<https://github.com/Arteok/UTN-TUPaD-P1/tree/main/Trabajos%20Integradores/An%C3%A1lisis%20de%20Algoritmos%20en%20Python>

Video explicativo:

<https://youtu.be/NFsuYn1qK8E>