Posgrado de Ingeniería de Sistemas

Análisis de algoritmos de palíndromos

Alberto Benavides
17 de abril de 2020

1. Algoritmos de palíndromos

Para esta tarea se desarrollan dos algoritmos de palíndromos en `Python`, uno iterativo y uno recursivo, presentes en los códigos 1 y 2. Ambos reciben una cadena de texto, eliminan los espacios que contenga y retornan $\top$ en caso de tratarse de un palíndromo y $\bot$ de otra manera.

Código 1: Algoritmo iterativo para encontrar palíndromos.

```
1  def iterativo(s):
2      s = s.replace(' ', '')
3      mitad = int(len(s) / 2)
4      if mitad == 0:
5          return True
6      for i in range(mitad):
7          print(s[i] + s[-1 - i])
8          if s[i] != s[-1 - i]:
9              return False
10     return True
```

Código 2: Algoritmo recursivo para encontrar palíndromos.

```
1  def recursivo(s):
2      s = s.replace(' ', '')
3      mitad = int(len(s) / 2)
4      if mitad == 0:
5          return True
6      if(s[0] != s[-1]):
7          return False
8      return recursivo(s[1:-1])
```

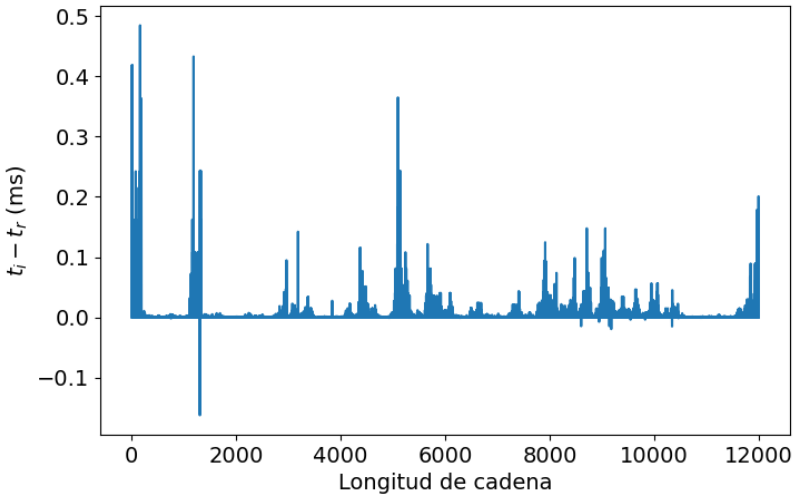

Figura 2.1: Diferencias en milisegundos entre los tiempos de cómputo del algoritmo iterativo (t_i) y los del recursivo (t_r).

2. Diseño de experimentos

Con el fin de intuir las diferencias entre tiempos de cómputo, se realiza un experimento computacional en el que se generan $N = 6\,000$ palíndromos de longitudes de cadena $n = \{0, 2, 4, \ldots, 2N\}$ obtenidos a partir de concatenar a $n/2$ caracteres tomados al azar del alfabeto `ascii` de la librería `string` [1] con los mismos caracteres en orden reverso. Por cada palíndromo, se realizaron $r = 1\,000\,000$ repeticiones con la librería `timeti` [2]. El tiempo que tomó completar estas r repeticiones para cada cadena de longitud n fue dividido entre r para obtener el promedio de milisegundos de cada operación.

Los resultados de estos experimentos computacionales demuestran que el algoritmo iterativo toma más tiempo que el recursivo. Las diferencias en milisegundos entre los tiempos t_i del algoritmo iterativo con respecto a los tiempos t_r del algoritmo recursivo se muestran en la figura 2.1. Muy pocos tiempos de cómputo del algoritmo recursivo son superiores a los del algoritmo iterativo para la misma cadena de texto, sin embargo esto puede deberse a los distintos procesos que se ejecutan por el sistema operativo donde se realizaron estos experimentos. Una discusión al respecto puede leerse en `https://stackoverflow.com/q/52251680`.

3. Análisis de los algoritmos

El análisis de los algoritmos se realiza siguiendo los ejemplos del material de curso compartido por Schaeffer [3] en sus notas de curso.

3.1. Algoritmo iterativo

Aquí, se tiene una cadena s de entrada, de longitud l, compuesta por l caracteres hubicados en una posición $s_i, i \in \{0, 1, 2, \ldots, l\}$. La complejidad computacional de este algoritmo viene dada por

- f_0: Lectura de la cadena s, $\mathcal{O}(1)$;
- f_1: Obtención de la longitud l de s, $\mathcal{O}(1)$;
- f_2: Asignación de $m = \lfloor l/2 \rfloor$ como mitad de la longitud de s, $\mathcal{O}(1)$;
- f_3: Condición que evalúa $m = 0$, con $\mathcal{O}(1)$;
- f_4: Condición que evalúa $s_i = s_{l-1-i}$, $\mathcal{O}(1)$;
- f_5: Ciclo que itera sobre $i \in \{0, 1, 2, \ldots, m\}$, $\mathcal{O}(m + f_4)$;

de donde se tiene que la complejidad computacional de este algoritmo es

$$\mathcal{O}(f_0 + f_1 + f_2 + f_3 + f_5) = \mathcal{O}(4 + n/2). \tag{3.1}$$

3.2. Algoritmo recursivo

La función R para el algoritmo recursivo de detección de palíndromos recibe también una cadena de caracteres s de longitud l de caracteres con c_i con posiciones $i \in \{0, 1, 2, \ldots, l-1\}$ cuya función es

$$R(s) = \begin{cases} \top & \lfloor l/2 \rfloor = 0 \\ \bot & c_0 \neq c_{l-1} \\ R(s(0, l-1)) & \lfloor l/2 \rfloor \neq 0 \wedge c_0 = c_{l-1} \end{cases} \tag{3.2}$$

Esta función puede simplificarse en la función modificada R^* que toma como parámetro inicial l

$$R^*(n) = \begin{cases} 1 & n \leq 1 \\ R^*(n-2) & n > 1 \end{cases} \tag{3.3}$$

de donde obtenemos

$$\begin{aligned} R^* &= R^*(n-2) \\ &= R^*((n-2)-2) \\ &= R^*(((n-2)-2)-2) \\ &= \ldots \end{aligned} \tag{3.4}$$

y por expansión

$$\begin{aligned} R^*(n) &= R^*(n-2) \\ R^*(n-2) &= R^*((n-2)-2) = R^*(n-4) \\ R^*(n-4) &= R^*(((n-2)-2)-2) = R^*(n-6) \\ &\dots \\ R^*(n-2i) &= R^*((n-2i)-2) \\ &\dots \\ R^*(n-2\cdot\lfloor n/2\rfloor) &= \begin{cases} R^*(1) \\ R^*(0) \end{cases} \end{aligned} \tag{3.5}$$

o sea

$$R^*(n) = R^*(1) + \lfloor n/2 \rfloor \tag{3.6}$$

lo que deja $\mathcal{O}(1 + \lfloor n/2 \rfloor)$.

Referencias

[1] Python Software Foundation. `string` – Common string operations. `https://docs.python.org/3/library/string.html`, 2020. [Accedido 16/abril/2020].

[2] Python Software Foundation. `timeit` – Measure execution time of small code snippets. `https://docs.python.org/3.8/library/timeit.html`, 2020. [Accedido 16/abril/2020].

[3] Elisa Schaeffer. Complejidad computacional de problemas y el análisis y diseño de algoritmos. [No publicado], 2019.