Universidad Simón Bolívar
CI-5651 - Diseño de Algoritmos I
Prof. Ricardo Monascal

Resuelto por:
Christopher Gómez

Tarea 7: Cadenas de caracteres/Geometría computacional

1. Sea n una cadena de caracteres que tiene su número de carné (sin el guión):
   (a) Construya un árbol de sufijos para n.
   (b) Construya un arreglo de sufijos a partir del árbol de sufijos para n.
   (c) Para cada posición k, calcule los valores para $\mathrm{PLCP}[k]$ (el prefijo común permutado más largo) y $\mathrm{LCP}[k]$ (el prefijo común más largo), como fue visto en clase.

Para este caso, la cadena de caracteres n es `1810892`, con lo que el árbol de sufijos sería el siguiente:

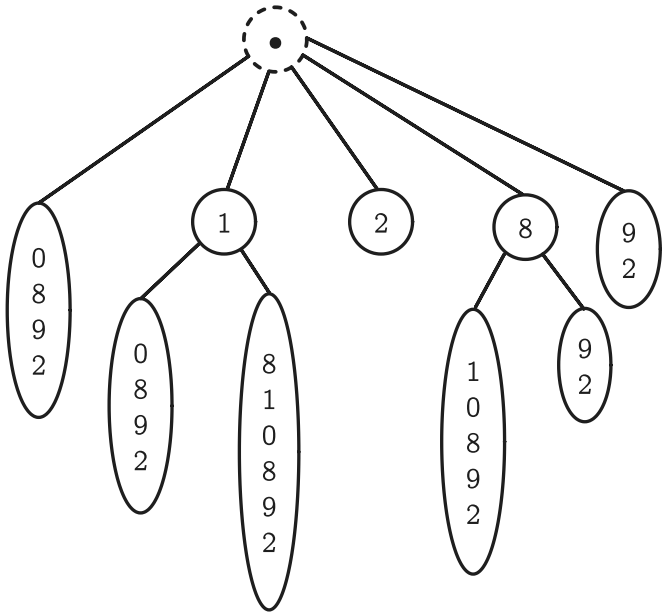

Figura 1: Árbol de sufijos para `1810892`

A partir de esto, podemos construir el arreglos de sufijos:

i	Sufijo
0	1810892
1	810892
2	10892
3	0892
4	892
5	92
6	2

7	

Y tras ordenar los sufijos lexicográficamente, obtenemos el siguiente arreglo, que conforma SA y desde el cual podemos aprovechar y llenar LCP:

i	SA[i]	LCP[i]	Sufijo
0	7	0	
1	3	0	0892
2	2	0	10892
3	0	1	1810892
4	6	0	2
5	1	0	810892
6	4	1	892
7	5	0	92

Y con esto podemos construir finalmente PLCP:

SA[i]	Phi[SA[i]]	PLCP[SA[i]]	Sufijo
0	2	1	1810892
1	6	0	810892
2	3	0	10892
3	7	0	0892
4	1	1	892
5	4	0	92
6	0	0	2
7	-1	0	

2. Sea $P = \{p_1, p_2, ..., p_n\}$ un conjunto de puntos en el plano cartesiano. Definimos una **capa** como aquellos puntos que forman parte del polígono convexo más pequeño que rodea a todos los puntos en P. Pero los puntos son como los ogros o las cebollas y pueden tener más de una capa. En particular, se puede remover la capa para P y obtener un conjunto de puntos P' a los que también se les puede calcular su capa. ¿Cuántas capas tiene el conjunto de puntos P?

   Diseñe un algoritmo que pueda responder a esta consulta usando tiempo $O(n^2 \log n)$ y memoria $O(n)$.

   Sabemos que podemos hallar un *Convex Hull* (o capa) de un conjunto de puntos en tiempo $O(n \log n)$ usando el algoritmo de *Graham Scan*. Con esto, un algoritmo que resuelva el problema sería el siguiente:

```
def capas(puntos: Conjunto[Punto]) → Entero:
    capas = 0
    while !puntos.vacío():
        capas += 1
        capa = graham_scan(puntos)
        puntos.remover(capa)

    return capas
```

   Veamos, la cantidad máxima de capas está acotada por n (no pudiese haber más capas que puntos), por lo que si el cuerpo del ciclo no tarda más de $O(n \log n)$, el algoritmo completo tendrá una complejidad de $O(n^2 \log n)$.

   En este punto hay que tener especial cuidado en la implementación, ya que si la verificación de existencia en cada capa no es constante, la operación `remover` tendría en el peor caso que recorrer `capa` tantas veces como elementos haya en la capa, tomándose tiempo $O(n^2)$ en lugar de $O(n)$. Así, para cumplir con la complejidad esperada, se debe retornar una estructura como Conjunto que permita verificar existencia en tiempo constante, así cada iteración del ciclo tomará tiempo $O(n + n \log n) = O(n \log n)$, y el algoritmo completo tomará tiempo $O(n^2 \log n)$.

   Luego, es trivial que la memoria utilizada por el algoritmo es $O(n)$, ya que *Graham Scan* no necesita polinomialmente más memoria que la cantidad de puntos que recibe como entrada.

   Una implementación de este algoritmo en Scala se puede encontrar aquí.

3. Considere una cadena de caracteres S, de longitud n. Se desea hallar la subcadena T de S más grande, tal que:

   - T sea prefijo de S (la cadena S *empieza* con T).
   - T sea sufijo de S (la cadena S *termina* con T).
   - $T \neq S$.

   Considere los siguientes ejemplos:

   - Para la cadena `ABRACADABRA`, la respuesta sería `ABRA`.
   - Para la cadena `AREPERA`, la respuesta sería `A`.
   - Para la cadena `ALGORITMO`, la respuesta sería λ. (la cadena vacía).

   Diseñe un algoritmo que pueda responder a esta consulta usando tiempo $O(n)$.

   Notemos que a la hora de hacer el preprocesamiento de una cadena dada S en el algoritmo de KMP, se calcula una tabla de saltos que se puede interpretar exactamente como lo que se pide (y de hecho, es ese el invariante que se mantiene en cada iteración de la construcción): la longitud del prefijo de $S[0..i]$ más largo que también es sufijo de $S[0..i]$ sin ser $S[0..i]$.

   De esta forma, el algoritmo que resolvería este problema consiste simplemente en preprocesar de dicha manera S y retornar una subcadena en el rango $[0, r)$, donde r es el último elemento de la tabla de saltos.

   El algoritmo sería entonces el siguiente:

```
def prefix_suffix(s: Cadena) → Cadena:
    tabla = preprocesar(s)

    return s.subcadena(0, tabla.último())
```

   Una implementación de este algoritmo se utilizó al compilar el código fuente de este documento y mostrar los resultados de los ejemplos en el enunciado, se puede encontrar aquí.