TRABAJO ESPECIAL DE LA
LICENCIATURA EN CS. DE LA COMPUTACIÓN

Análisis de Estructuras de Sufijos de Strings

Presentada por

Marcos Kolodny

Director: Dr. Luis Ferroni
Profesor representante: Dr. Nicolás Wolovick

Trabajo especial 2021

Introducción

El problema de desarrollar algoritmos que decidan si un cierto patrón o palabra aparece o no en un determinado texto es fundamental en ciencias de la computación. Diversos algoritmos se han desarrollado en las últimas décadas para resolver este problema (y sus múltiples variantes). Un análisis detallado de las complejidades temporales y espaciales de dichos algoritmos revela que, en la práctica, teniendo en cuenta el volumen de información y la potencia de los procesadores al día de la fecha, algoritmos de fuerza bruta no son viables en la mayoría de los casos.

La estructura de datos *suffix tree* es una forma eficiente para efectuar búsqueda on-line de patrones en un texto. Un suffix tree para un texto T de longitud n sobre un alfabeto Σ puede ser construido en $\mathcal{O}(n \log |\Sigma|)$ tiempo y $\mathcal{O}(n)$ memoria (ver por ejemplo [Wei73]). Tal estructura permite responder en $\mathcal{O}(m \log |\Sigma|)$ si un string S de longitud m tiene una ocurrencia en T. Tal estructura ha sido estudiada y refinada de manera extensiva a lo largo del último medio siglo, obteniéndose diversas generalizaciones que se pueden utilizar para resolver dicho problema de manera off-line, o con múltiples instancias simultáneamente.

En [MM93] Manber y Myers introdujeron una estructura de datos llamada *suffix array* que reemplaza al suffix tree, e incluso aunque teniendo la misma complejidad teórica, es sensiblemente superior cuando el texto T a considerar es grande. Este algoritmo ha sido durante las últimas décadas objeto de estudio y diversas mejoras han sido desarrolladas de manera análoga a lo sucedido con suffix tree. Entre muchas otras ventajas, poder representar con un arreglo de números algo que de otro modo requeriría una estructura más compleja como un árbol, sugiere que la exposición y el estudio de la teoría de suffix arrays es mucho más simple que la de suffix tree.

Existe otra estructura, conocida como el *longest common prefix array* (llamado usualmente lcp array), introducida también por Manber y Myers, y luego optimizada por Kasai et al. en [KLA+01], que al ser combinada con el suffix array de una palabra, permite obtener soluciones eficientes a distintos problemas, potenciando así el poder de suffix array. Subsecuentemente han aparecido diversas optimizaciones en las últimas décadas para estas construcciones (por ejemplo, ver [Man04]).

El objetivo de esta tesis es exponer de manera completa los algoritmos de suffix array y lcp array, explorando varias de sus aplicaciones. El desarrollo es completamente autocontenido y se ha dado prioridad a los ejemplos. Hemos incluido pseudocódigo para todas las rutinas y gráficos comparando la performance

temporal de los algoritmos de fuerza bruta con los que se han presentado en esta tesis. La computadora utilizada para este trabajo posee un procesador Intel Core i5-8250U 1.60GHz, además de una memoria RAM DDR4 de 4gb.

Organización del trabajo

En el Capítulo 1 establecemos la notación necesaria para el desarrollo del trabajo y la terminología básica de strings. Además, presentamos los problemas particulares a abordar, proponiendo soluciones de fuerza bruta que los resuelven y motivando la búsqueda de algoritmos más eficientes.

En el Capítulo 2 introducimos el algoritmo propuesto por Manber y Myers en [MM93] que construye el suffix array de una palabra, y luego realizamos un análisis detallado de la complejidad temporal y espacial del mismo. Además, proponemos distintas optimizaciones, que permiten alcanzar complejidades aún mejores.

En el Capítulo 3 explicamos de manera minuciosa cómo construir el longest common prefix array de un string extendiendo el método propuesto en el capítulo anterior. Además, describimos el algoritmo de Kasai et al. (ver [KLA$^+$01]), que alcanza una complejidad óptima para la construcción del arreglo deseado.

En el Capítulo 4 planteamos soluciones que, utilizando las estructuras propuestas en los Capítulos 2 y 3, mejoran la complejidad espacial y temporal de los métodos detallados en el Capítulo 1. Además explicamos cómo optimizar aún más estos algoritmos mediante el uso de estructuras de datos adicionales.

Concluimos con un Apéndice en el que describimos, de manera simplificada, dos estructuras de datos que son necesarias en el desarrollo del trabajo para realizar cálculos intermedios y subrutinas eficientes en los algoritmos y optimizaciones propuestas.

Índice general

Capítulo 1

Preliminares

1.1. Algunas definiciones

1.1.1. El concepto de string

Para comenzar nuestra disertación, introducimos un poco de terminología. Un *alfabeto* es un conjunto finito, cuyos elementos serán llamados *símbolos*. Una lista o sucesión finita de elementos en un alfabeto será llamada *palabra* o *string*. En general utilizaremos la letra Σ para denotar alfabetos. Usualmente también utilizaremos letras como S, T, α o β para denotar strings.

Fijo un alfabeto Σ, el conjunto de todas las palabras que se pueden obtener en dicho alfabeto será denotado por Σ^*. En particular, Σ^* contiene a una única sucesión que consiste de 0 símbolos, la cual vamos a denotar por ε.

Ejemplo 1.1.1. Consideremos $\Sigma = \{\texttt{a}\}$. En tal caso, tenemos que

$$\Sigma^* = \{\varepsilon, \texttt{a}, \texttt{aa}, \texttt{aaa}, \ldots\}.$$

Durante este trabajo, a menos que se mencione explícitamente lo contrario, asumiremos que el alfabeto a considerar es el de las letras minúsculas del alfabeto de la lengua inglesa (26 letras, desde la `a` hasta la `z`). Esta hipótesis no es restrictiva en lo absoluto, dado que, cuando se aborden los resultados principales, quedará claro que los mismos se pueden adaptar para cualquier alfabeto con cualquier cantidad (finita) de símbolos.

Definición 1.1.2. La *longitud* de un string S, la cual será denotada por $|S|$, está definida como la cantidad de elementos de S. Dado $0 \leq i < |S|$, denotaremos por S_i al i-ésimo símbolo de S.[1] Cuando $i < 0$ o $i \geq |S|$, definimos $S_i = \omega$, donde ω es un símbolo arbitrario que asumimos que no pertenece a Σ.

Ejemplo 1.1.3. Dado $S = \varepsilon$ y $T = \texttt{banana}$, tenemos $|S| = 0$ y $|T| = 6$. Además observemos que $S_0 = S_{-1} = S_1 = \omega$, así como también se cumple que $T_2 = \texttt{n}$ y $T_6 = \omega$.

[1] Durante este trabajo vamos a utilizar siempre indexación desde 0.

Definición 1.1.4. Un *substring* de un string S está dado por una sucesión correspondiente a un rango continuo de símbolos pertenecientes a S. Es decir, el substring determinado por el rango $[i, j) := \{k \in \mathbb{Z} : i \leq k < j\}$, el cual denotaremos por $S_{i,j}$, está dado de la siguiente manera

- Si $j > |S|$, $S_{i,j} = S_{i,|S|}$
- Si $i < 0$, $S_{i,j} = S_{0,j}$
- Si $i \geq j$, $S_{i,j} = \varepsilon$
- Si $0 \leq i < j \leq |S|$, $S_{i,j} = S_i S_{i+1} \dots S_{j-1}$

Ejemplo 1.1.5. Dado $S = \texttt{banana}$, tenemos que:

$$S_{0,6} = S_{-1,10} = \texttt{banana}$$
$$S_{-2,2} = \texttt{ba}$$
$$S_{-2,0} = S_{3,3} = \varepsilon$$

Una operación fundamental para este trabajo está dada por la *concatenación* de dos strings α y β, denotada por $\alpha\beta$. La misma está definida de la siguiente manera:

$$\alpha\beta = \alpha_0\alpha_1 \dots \alpha_{|\alpha|-1}\beta_0\beta_1 \dots \beta_{|\beta|-1},$$

si $|\alpha|, |\beta| \geq 1$. Además, definimos que $\varepsilon\alpha = \alpha$ y que $\alpha\varepsilon = \alpha$ para todo α. Observar que, en particular, $\varepsilon\varepsilon = \varepsilon$.

Dado un entero positivo n, expresaremos la concatenación de n copias de un string α como α^n. Por definición, establecemos que $\alpha^0 = \varepsilon$ para todo string α. Al momento de escribir pseudocódigo, y con el objetivo de lograr mayor claridad, denotaremos a la concatenación de α con β como $\alpha + \beta$.

Ejemplo 1.1.6.

$$\texttt{hola}^3 = \texttt{holaholahola}$$
$$\texttt{hola}^0 = \varepsilon$$
$$\varepsilon^0 = \varepsilon^5 = \varepsilon$$

Definición 1.1.7. Sea α un string en Σ^*.

- Un *sufijo* de α es un string β tal que
$$\alpha = \gamma\beta,$$
para algún $\gamma \in \Sigma^*$.
- Un *prefijo* de α es un string β tal que
$$\alpha = \beta\gamma,$$
para algún $\gamma \in \Sigma^*$.

notar que ε es a la vez un sufijo y un prefijo de cualquier string. Dado que a menudo consideraremos el substring de un string S que se obtiene de tomar todos los símbolos desde la posición i en adelante, introducimos la notación

$$S_{i\rightarrow} \doteq S_{i,|S|}$$

1.1.2. Orden lexicográfico

La idea en esta subsección es establecer de manera rigurosa las propiedas intuitivas del orden lexicográfico, el cual es, ni más ni menos, el orden "alfabético" de las palabras.

Sea Σ un alfabeto y $\leq'$ una relación de orden total establecida sobre $\Sigma \sqcup \{\omega\}$ que cumpla que $\omega \leq' x$, para todo $x \in \Sigma$. Dados c_1,$c_2 \in \Sigma$, diremos que $c_1 <' c_2$ si vale que $c_1 \leq' c_2$ y $c_1 \neq c_2$. Procedemos a definir una relación binaria $\leq$ sobre el conjunto Σ^*, a la cual llamaremos *orden lexicográfico*.

Definición 1.1.8. Sean α y β strings pertenecientes a Σ^*. Definimos la función *longest common prefix*, la cual denotamos por lcp, a la asignación:

$$\mathrm{lcp}(\alpha, \beta) = |\gamma|,$$

donde γ es el string de mayor longitud que es simultáneamente prefijo de α y β.

Observación 1.1.9. Notar que conocer la longitud de un prefijo de un string α es equivalente a determinar cuál es tal prefijo. En particular, conocer $\mathrm{lcp}(\alpha, \beta)$ (es decir, la longitud del prefijo común más largo) es equivalente a conocer a tal prefijo. A menudo usaremos de manera indistinta $\mathrm{lcp}(\alpha, \beta)$ para denotar al string o a su longitud. Este abuso de notación no debería causar confusiones pues del contexto se infiere si estamos hablando del string o de su longitud.

El orden prefijado en el conjunto de elementos $\Sigma \sqcup \{\omega\}$ induce un orden en el conjunto Σ^* como describimos a continuación. Dados dos strings $\alpha, \beta \in \Sigma^*$, y $\gamma = \mathrm{lcp}(\alpha, \beta)$, con $\ell = |\gamma|$, definimos que $\alpha \leq \beta$ si y sólo si $\alpha = \gamma$ o $\alpha_\ell \leq' \beta_\ell$.

Lema 1.1.10. *Sean α, β strings de Σ^* y $\ell = \mathrm{lcp}(\alpha, \beta)$. Se cumple $\alpha_i \leq' \beta_i$ para todo $i \in [0, \ell]$.*

Demostración. Si $i = \ell$, la condición vale por definición de lcp. Si $0 \leq i < \ell$, tenemos que $\alpha_i = \beta_i$. Pero como $\leq'$ es un orden total, es reflexivo. Esto quiere decir que $\alpha_i \leq' \alpha_i$, y entonces $\alpha_i \leq' \beta_i$. □

Lema 1.1.11. *La relación $\leq$ es transitiva.*

Demostración. Sean α, β y δ strings de Σ^* tal que $\alpha \leq \beta$ y $\beta \leq \delta$. Si $\alpha = \beta$ o $\beta = \delta$, el lema queda trivialmente demostrado. Supongamos ahora que $\alpha \neq \beta$ y $\beta \neq \delta$. Además, supongamos $l_{ab} = \mathrm{lcp}(\alpha, \beta)$, $l_{bc} = \mathrm{lcp}(\beta, \delta)$ y $l_{ac} = \mathrm{lcp}(\alpha, \delta)$. Hay dos casos:

- Si $|l_{ab}| \leq |l_{bc}|$, sea $k = |l_{ab}|$. Por definición de lcp, $\alpha_k \leq' \beta_k$. Además, por Lema 1.1.10, $\beta_k \leq' \delta_k$. Como $\leq'$ es un orden total, cumple transitividad, por lo que $\alpha_k \leq' \delta_k$. Como $\alpha \neq \beta$, tenemos que $\alpha_k \neq \beta_k$, y por ende $\alpha_k \neq \delta_k$. Además, $|l_{ab}| \leq |l_{bc}| \Rightarrow l_{ac} = l_{ab}$, por lo que $\alpha_{0,k} = \delta_{0,k}$. Esto nos dice que $\mathrm{lcp}(\alpha, \delta) = \alpha_{0,k}$, y como $\alpha_{|\,\mathrm{lcp}(\alpha,\delta)|} \leq' \delta_{|\,\mathrm{lcp}(\alpha,\delta)|}$, concluimos que $\alpha \leq \delta$.

- Si $l_{ab}| > |l_{bc}|$, la prueba es análoga al caso anterior.

□

Lema 1.1.12. *La relación $\leq$ es reflexiva.*

Demostración. Sea α un string de Σ^*. Notemos que $\text{lcp}(\alpha, \alpha) = \alpha$. Pero entonces, por definición de $\leq$, vale que $\alpha \leq \alpha$. □

Lema 1.1.13. *La relación $\leq$ es antisimétrica.*

Demostración. Sean α, β strings de Σ^*, y $\gamma = \text{lcp}(\alpha, \beta)$. Asumiendo que vale que $\alpha \leq \beta$ y $\beta \leq \alpha$, intentaremos demostrar que $\alpha = \beta$. Para ello, supongamos que esto no ocurre. Dado que $\alpha \leq \beta$ y $\alpha \neq \beta$, tenemos que $\alpha_{|\gamma|} \leq' \beta_{|\gamma|}$. De manera similar se puede mostrar que $\beta_{|\gamma|} \leq' \alpha_{|\gamma|}$. Como $\leq'$ es un orden total, vale entonces que $\alpha_{|\gamma|} = \beta_{|\gamma|}$. Pero si esto ocurre, entonces el string $\alpha_{0,|\gamma|+1}$ es un prefijo en común entre α y β de mayor longitud que $\text{lcp}(\alpha, \beta)$. Esta contradicción completa la prueba. □

Corolario 1.1.14. *La relación $\leq$ es un orden parcial sobre Σ^*.*

Demostración. Es consecuencia directa de los Lemas 1.1.11, 1.1.12, 1.1.13. □

No sólo tenemos que $\leq$ es un orden parcial, sino que todo par de elementos de Σ^* se puede comparar en el orden $\leq$. En otras palabras, tenemos el siguiente resultado.

Lema 1.1.15. *La relación $\leq$ es un orden total.*

Demostración. Sean α, β strings de Σ^* tales que $\alpha \not\leq \beta$, y sea $\gamma = \text{lcp}(\alpha, \beta)$. Si $\gamma = \beta$, entonces $\beta \leq \alpha$ por definición de lcp. Supongamos entonces que $\gamma \neq \beta$. Dado que $\alpha \not\leq \beta$ tenemos que $\alpha_{|\gamma|} \not\leq' \beta_{|\gamma|}$, pero al ser $\leq'$ una relación de orden total, debe valer que $\beta_{|\gamma|} \leq' \alpha_{|\gamma|}$. Concluimos entonces que $\beta \leq \alpha$. □

Al haber demostrado que $\leq$ es un orden total, podemos definir sin ambigüedad el concepto de mínimo y máximo de dos strings de manera intuitiva:

$$\text{mín}(S, T) = \begin{cases} S & \text{si } S \leq T \\ T & \text{caso contrario} \end{cases}$$

$$\text{máx}(S, T) = \begin{cases} S & \text{si } S \geq T \\ T & \text{caso contrario} \end{cases}$$

Ejemplo 1.1.16. Se cumple que

$$\texttt{abc} \leq \texttt{abcde} \leq \texttt{bcd}.$$

Además, tenemos que

$$\text{mín}(\texttt{abc}, \texttt{bcd}) = \texttt{abc},$$

$$\text{máx}(\texttt{abcde}, \texttt{bcd}) = \texttt{bcd}.$$

1.1.3. String matching

En esta subsección vamos a introducir formalmente al principal problema que abordaremos en esta tesis, que es el llamado problema de *string matching*. Dado que existen diversas variantes del mismo, mencionaremos las dos más frecuentes en el uso cotidiano. Previo a esto, introducimos brevemente algunos conceptos que serán de utilidad para entender el problema a definir:

Definición 1.1.17. La *distancia de edición*, también llamada *edit distance*, entre dos strings S y T es la mínima cantidad de operaciones (borrar, agregar y/o substituir símbolos de las palabras) necesaria para lograr que $S = T$.

Ejemplo 1.1.18. Sea $S = \texttt{hola}$ y $T = \texttt{ala}$, tenemos que la distancia de edición entre S y T es igual a 2. Una forma de lograr la igualdad mediante 2 operaciones está dada por:

$$\texttt{hola} \longrightarrow \texttt{ola} \longrightarrow \texttt{ala}.$$

Se puede notar que una única operación no es suficiente para lograr que $S = T$.

Definición 1.1.19. Dados strings S y T, diremos que:

- S *ocurre exactamente* en T si $S = T$.
- S *ocurre aproximadamente* en T, si la distancia de edición entre S y T es menor a un valor establecido.

Definición 1.1.20. Dados dos strings S y T, y un entero i, definimos que S *ocurre a partir de* i en T, si S es una ocurrencia exacta de $T_{i,i+|S|}$.

El problema de *string matching* consiste en encontrar una o varias ocurrencias de un string P (patrón) como substrings de otro string T (texto). En el ámbito de estudio de este problema, los algoritmos se clasifican en dos categorías según la clase de ocurrencia que se considere: matching exacto (ocurrencias exactas) o aproximado (ocurrencias aproximadas).

Durante el desarrollo de este trabajo, nos enfocaremos en explicar y analizar con profundidad algunos algoritmos del primer tipo, los cuales se utilizarán para la resolución de distintos problemas que tienen gran influencia en situaciones de la vida real. A pesar de esto, cabe destacar que el análisis de algoritmos del segundo tipo es igual de importante en el área de estudio, pues en reiteradas ocasiones se debe trabajar con textos en donde, por ejemplo, se debe tener en cuenta posibles errores o diferencias entre el string original y el que se utiliza en la comparación (debido a la pérdida de información durante transmisión de datos, equivocaciones humanas al redactar, etc.). Para un abordaje más detallado de este tipo de problemas, recomendamos [Ukk85].

1.2. Algunos problemas relacionados

A continuación, abordaremos algunos de los problemas más reconocidos en el área de strings, proponiendo soluciones y analizando el rendimiento de las mismas,

tanto en complejidad temporal como espacial. La intención es poner de manifiesto la necesidad de proveer algoritmos rápidos para resolver estos problemas. Además, la formulación de nuestros problemas es intencionalmente concreta, de modo que queda claro que la performance de estos algoritmos tiene impacto en problemas que suceden de manera muy frecuente en la vida real.

En todos los escenarios propuestos durante el trabajo, asumiremos que la complejidad temporal de comparar si dos strings S y T son iguales será igual a $\mathcal{O}(\text{mín}(|S|, |T|))$. Más concretamente, asumiremos que determinar la relación $\leq'$ entre dos símbolos de un alfabeto Σ tiene complejidad $\mathcal{O}(1)$.

1.2.1. Contar ocurrencias de un patrón en un texto

Uno de los problemas principales del área de string matching es el de computar cuántas ocurrencias hay de un patrón S en un texto T. Definamos $n = |S|$ y $m = |T|$. Si vale que $m < n$, entonces S no puede ocurrir en T. Es por esto que, por simplicidad, asumiremos siempre que $m \geq n$.

Una solución posible es iterar sobre todo posible candidato a posición inicial i de T, y observar si vale que $S = T_{i,i+n}$. Como hay m posiciones candidatas, y chequear la igualdad de strings para cada una de ellas tiene complejidad $\mathcal{O}(n)$, la complejidad temporal del algoritmo completo es $\mathcal{O}(n \cdot m)$.

Evidentemente, una posible optimización consiste en no considerar como candidatos a las posiciones $i \in [m - n + 1, m)$, pues en ese escenario $|T_{i,i+n}| < |S|$, por lo que nunca será considerado una ocurrencia de S en T. Gracias a esto, la cantidad de candidatos se reduce a $m-n+1$, y el costo total es: $\mathcal{O}((m-n+1)n)$. Se puede observar que cuando $m - n$ es de un orden de magnitud muy pequeño (digamos, una constante), el algoritmo realizará $\mathcal{O}(n)$ operaciones. Observar, no obstante, que si $n \approx m/2$, el costo de cómputo es $\mathcal{O}(m^2)$, por lo que la optimización propuesta solo mejora la complejidad del algoritmo por un factor constante.

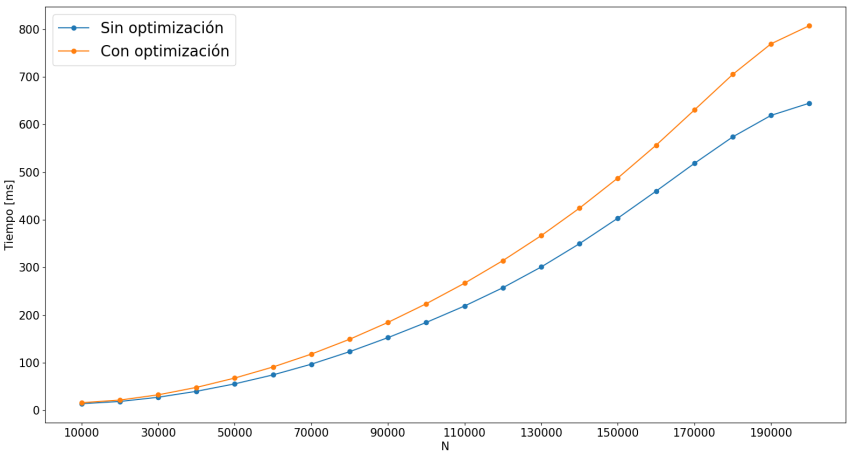

Figura 1.1: Tiempo de ejecución de la fuerza bruta original vs. la optimizada, considerando $m = 2n$, $S = \mathtt{a}^n$ y $T = \mathtt{a}^m$.

Algoritmo 1 Contar ocurrencias de S en T en $\mathcal{O}(n \cdot m)$

```
1: total ← 0
2: for i ← 0 to m − 1 do
3:     if S = T_{i,i+n} then
4:         total ← total + 1
5:     end if
6: end for
7: return total
```

1.2.2. Rotación lexicográficamente mínima de un string

Definimos como *shift cíclico* a la operación de mover el último símbolo de un string S al principio del mismo (para el caso del string ε, el shift cíclico es simplemente la función identidad). Denotaremos por shift$(\cdot)$ a esta función. Por ejemplo, shift(`banana`) = `abanan`.

Definición 1.2.1. Un string T es una *rotación* de S si existe un entero no negativo j tal que

$$\text{shift}^j(S) = T,$$

donde shift^j es la identidad si $j = 0$ y shift^j = shift $\circ$ shift^{j-1} para $j \geq 1$.

Ejemplo 1.2.2. Sea S = `aacaab`, la lista de strings distintos que se pueden obtener mediante aplicar reiteradas veces un shift cíciclo a S está dada por

$$[\texttt{aacaab}, \texttt{baacaa}, \texttt{abaaca}, \texttt{aabaac}, \texttt{caabaa}, \texttt{acaaba}]$$

Se puede observar allí que la rotación lexicográficamente mínima de S es `aabaac`.

El problema de hallar la rotación lexicográficamente mínima de un string (también conocida como *lyndon word*) tiene variadas aplicaciones en la vida real, abarcando distintas áreas como combinatoria y álgebra [Lot83, Reu93].

Un algoritmo sencillo para computar la rotación mínima de un string S de longitud n consiste en concatenar a la palabra consigo misma, y luego encontrar allí el menor substring de longitud n. La cantidad de candidatos total es exactamente n, y para cada uno de ellos se realiza una comparación lexicográfica de strings, la cual requiere $\mathcal{O}(n)$ operaciones. Esto resulta en un algoritmo de complejidad temporal $\mathcal{O}(n^2)$.

Algoritmo 2 Rotación lexicográficamente mínima de S en $\mathcal{O}(n^2)$

```
1: lyndon ← S
2: T ← S + S
3: for i ← 0 to n − 1 do
4:     lyndon ← mín(lyndon, T_{i,i+n})
5: end for
6: return lyndon
```

Observar que, si por ejemplo el string considerado contuviera una letra `a`, sería razonable considerar rotaciones que comiencen solamente con el símbolo `a`, pues evidentemente cualquier rotación que comience con otra letra será automáticamente mayor. Más generalmente, el algoritmo propuesto se puede optimizar a través de solo considerar como candidatas a aquellas rotaciones que comiencen con el mínimo símbolo de la palabra. Sin embargo, notar que en el caso en que el string posee muchas repeticiones del menor símbolo, se siguen realizando $\mathcal{O}(n^2)$ operaciones. No obstante, a pesar de este posible peor escenario, si uno tiene como información que la palabra es una cadena aleatoria de símbolos de Σ, no es difícil ver que la cantidad esperada de ocurrencias del mínimo símbolo es $\frac{n}{|\Sigma|}$. En particular, si Σ contiene una cantidad de símbolos significativa, esta optimización, a pesar de ser inocente, puede generar mejoras apreciables.

1.2.3. Largo máximo de un substring en común entre dos strings

En el área de estudio que abarca el análisis de textos, en reiteradas ocasiones se debe calcular distintas medidas de similitud entre dos strings. Entre ellas se encuentra la longitud máxima de un string que ocurra exactamente en ambas palabras. Por ejemplo, podemos observar que `grama` es el patrón más largo que ocurre simultáneamente en las palabras `programar` y `diagramas`.

Más rigurosamente, el problema consiste de, dados strings S y T de largo n y m respectivamente, calcular la mayor longitud de un string α tal que existen β_1, β_2, γ_1 y γ_2 que cumplan $S = \beta_1\alpha\beta_2$ y $T = \gamma_1\alpha\gamma_2$.

Para resolver este problema, se puede iterar sobre la longitud de β_1 y γ_1, y luego calcular $\text{lcp}(S_{|\beta_1|,|S|}, T_{|\gamma_1|,|T|})$ para conocer la máxima cantidad de símbolos que se pueden incluir en α para ese caso. La cantidad de posibles longitudes de β_1 y γ_1 son n y m respectivamente, y asumiendo que la complejidad de calcular el lcp entre dos strings A y B es $\mathcal{O}(\text{mín}(|A|, |B|))$, la cantidad de operaciones total es: $\sum_{i=0}^{n-1}\sum_{j=0}^{m-1}\text{mín}(n-i, m-j)$, por lo que el algoritmo tiene una complejidad igual a $\mathcal{O}(n \cdot m \cdot \text{mín}(n, m))$.

Para el cálculo del lcp entre dos strings S y T, de largo n y m respectivamente, en complejidad $\mathcal{O}(\text{mín}(n, m))$ se puede plantear la siguiente función recursiva:

$$\text{lcp}(S, T) = \begin{cases} 1 + \text{lcp}(S_{1,n}, T_{1,m}) & \text{si } S \neq \varepsilon \wedge T \neq \varepsilon \wedge S_0 = T_0 \\ 0 & \text{caso contrario} \end{cases}$$

Ejemplo 1.2.3.

$$\text{lcp}(\texttt{hola}, \texttt{horas}) = 1 + \text{lcp}(\texttt{ola}, \texttt{oras})$$

$$\text{lcp}(\texttt{ola}, \texttt{oras}) = 1 + \text{lcp}(\texttt{la}, \texttt{ras})$$

$$\text{lcp}(\texttt{la}, \texttt{ras}) = 0$$

Es posible extender la definición de la función lcp, agregando como parámetros la posición inicial de cada uno de los dos strings a considerar. Es decir, podemos

considerar la función $\text{lcp}(S, T, i, j) \doteq \text{lcp}(S_{i\rightarrow}, T_{j\rightarrow})$. Se puede aprovechar esta función más general para el cálculo del lcp sobre cualquier sufijo de las palabras:

$$\text{lcp}(S, T, i, j) = \begin{cases} 1 + \text{lcp}(S, T, i+1, j+1) & \text{si } i \in [0, n) \wedge j \in [0, m) \wedge S_i = T_j \\ 0 & \text{caso contrario} \end{cases}$$

Utilizando la definición anterior, podemos proceder ahora a plantear el pseudocódigo de nuestro algoritmo:

Algoritmo 3 Máximo substring en común entre S y T en $\mathcal{O}(n \cdot m \cdot \text{mín}(n, m))$

```
1: max_length ← 0
2: for i ← 0 to n − 1 do
3:     for j ← 0 to m − 1 do
4:         max_length ← máx(max_length, lcp(S, T, i, j))
5:     end for
6: end for
7: return max_length
```

Definamos a continuación la función $\text{rec\_path}(S, T, i, j)$, que retorna el conjunto de pares de posiciones considerados en cada una de las llamadas recursivas realizadas al ejecutar $\text{lcp}(S, T, i, j)$. Por ejemplo, si $S =$ `abcd`, $T =$ `cde`, $i = 2$ y $j = 0$, tenemos que:

$$\text{rec\_path}(S, T, i, j) = \{(2, 0), (3, 1), (4, 2)\}$$

Ahora bien, notar que si $\text{lcp}(S, T, i, j)$ realiza una llamada recursiva, vale que:

$$\text{rec\_path}(S, T, i+1, j+1) = \text{rec\_path}(S, T, i, j) \setminus \{(i, j)\}$$

Esto nos permite pensar en el uso de programación dinámica para realizar el cálculo de la función $\text{lcp}(S, T, i, j)$, para todo $i \in [0, n]$, $j \in [0, m]$. Para esto, podemos utilizar una matriz de $(n+1) \times (m+1)$ números, tal que la celda en la fila r y la columna c (denotada por $\text{dp}_{r,c}$), representa el valor de $\text{lcp}(S, T, r, c)$. El cómputo de cada celda de la matriz dp se realiza de una manera similar a como hemos definido a la función lcp, requiriendo $\mathcal{O}(1)$ operaciones. Esto nos lleva a concluir que la complejidad total del cálculo de la matriz es $\mathcal{O}(n \cdot m)$.

Algoritmo 4 Cómputo de la matriz dp en $\mathcal{O}(n \cdot m)$

```
1: dp ← matrix(n + 1, m + 1)                ▷ Matriz de tamaño (n + 1) × (m + 1)
2: for i ← n down to 0 do
3:     for j ← m down to 0 do
4:         if i = n or j = m or S_i ≠ T_j then
5:             dp_{i,j} ← 0
6:         else
7:             dp_{i,j} ← 1 + dp_{i+1,j+1}
8:         end if
9:     end for
10: end for
11: return dp
```

0	4	1	0	1	0
0	1	3	0	1	0
4	0	0	2	0	0
0	3	1	0	1	0
0	1	2	0	1	0
1	0	0	1	0	0
0	0	0	0	0	0

Cuadro 1.1: Matriz dp para los strings $S =$ `aabaab` y $T =$ `baaba`

La matriz propuesta nos es de utilidad en el Algoritmo 3 para evitar el llamado a $\text{lcp}(S, T, i, j)$, reemplazándolo por su equivalente: $\text{dp}_{i,j}$. De esta manera, el programa que calcula la máxima longitud de un substring en común entre dos strings de largo n y m resulta en una complejidad temporal y espacial de $\mathcal{O}(n \cdot m)$.

La solución propuesta para este problema se puede extender de manera sencilla si se requiere computar la máxima longitud de un substring en común entre k strings $S_1, S_2, \ldots, S_k$, resultando en una complejidad temporal y espacial de $\mathcal{O}(n_1 \cdot n_2 \cdot \ldots \cdot n_k)$, donde $n_i = |S_i|$.

Capítulo 2

Suffix array

2.1. Introducción

Vamos a introducir notación y terminología esencial para el desarrollo del trabajo.

Una permutación de longitud n es una lista P de los primeros n enteros no negativos en algún orden. Definimos la *permutación inversa de* P, denotada por $\text{inv}(P)$, a la única permutación P' tal que $P'_{P_i} = i$, es decir que P'_i es la posición de P que posee el valor i.

Ejemplo 2.1.1. Consideramos la permutación dada por $P = [0, 3, 1, 2]$. En este caso, la permutación inversa está dada por $\text{inv}(P) = [0, 2, 3, 1]$. Si $Q = [0, 1, 2]$, entonces tenemos que $\text{inv}(Q) = [0, 1, 2] = Q$.

A pesar de que no utilizaremos resultados de teoría de grupos en generalidad, vale la pena mencionar que el conjunto de todas las permutaciones de los elementos $\{0, 1, \ldots, n\}$ es un grupo utilizando la composición. Tal grupo se denomina el *grupo simétrico* $\mathbb{S}_{n+1}$. En particular, $\text{inv}(P)$ denota al inverso de la permutación P en tal grupo.

Además, denotamos con $\text{iden}(n)$ a la permutación ordenada de longitud n. En otras palabras, $\text{iden}(n)$ es el elemento neutro del grupo mencionado anteriormente.

Sea Σ' un alfabeto, y sea $\omega \notin \Sigma'$. A partir de ahora trabajaremos con el alfabeto extendido $\Sigma = \Sigma' \sqcup \{\omega\}$, donde además vamos a fijar un orden total $\leq'$ arbitrario sobre Σ que satisface que ω es el menor elemento.

Observación 2.1.2. Dado que hemos prefijado un orden total para los símbolos de Σ, no se pierde generalidad en trabajar con su posición relativa en este orden. Por ejemplo, si $\Sigma' = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ y $\Sigma = \Sigma' \sqcup \{\omega\}$, al prefijar el orden dado por

$$\omega < \texttt{a} < \texttt{b} < \texttt{c},$$

tenemos que la palabra `bbbbaca` se puede pensar como la lista de números representada por $[2, 2, 2, 2, 1, 3, 1]$.

Definición 2.1.3. Sea $S \in \Sigma^*$ un string. Dado $0 \leq i < |S|$, expresaremos como $\text{ord}_i(S)$ a la cantidad de sufijos de S que son estrictamente menores a $S_{i\rightarrow}$.

Equivalentemente, $\text{ord}_i(S)$ es la cantidad de posiciones $j \in [0, |S|)$ que cumplen $S_{j\rightarrow} < S_{i\rightarrow}$. Usaremos la convención de que $\text{ord}_i(S) = -1$ cuando $i \geq n$.

Ejemplo 2.1.4. Sea Σ como en la Observación 2.1.2. Consideremos $S = \texttt{abac}$. Entonces tenemos que la lista de sufijos de S es

$$[\texttt{abac}, \texttt{bac}, \texttt{ac}, \texttt{c}].$$

En este escenario, $\text{ord}_2(S) = 1$, pues tenemos que

$$\texttt{abac} < \texttt{ac} \qquad \texttt{bac} \not< \texttt{ac} \qquad \texttt{ac} \not< \texttt{ac} \qquad \texttt{c} \not< \texttt{ac}$$

Notar que siempre ocurre que $0 \leq \text{ord}_i(S) < |S|$ para todo $0 \leq i < |S|$, pues hay $|S|$ sufijos en total, y evidentemente tenemos que $S_{i\rightarrow} \not< S_{i\rightarrow}$.

En reiteradas ocasiones durante el desarrollo de este trabajo deberemos trabajar con listas de números. Con el objetivo de simplificar la redacción, observemos que se puede manipular a una lista de enteros de la misma manera que lo hemos hecho con strings. Es decir, todas las relaciones, operaciones y definiciones que fueron introducidas para las palabras (tales como concatenación, prefijo, sufijo o substring) también pueden ser utilizadas para listas de números sin causar ambigüedades.

Definición 2.1.5. El *suffix array* de un string (o una lista de números) S es una lista lexicográficamente ordenada de todos sus sufijos distintos de ε.

Ejemplo 2.1.6. El suffix array de la palabra `banana` está dado por

$$A = [\texttt{a}, \texttt{ana}, \texttt{anana}, \texttt{banana}, \texttt{na}, \texttt{nana}]$$

Durante el desarrollo de este capítulo, abordaremos el problema de computar de manera eficiente el suffix array de un string, considerando siempre la complejidad temporal y espacial. Esto tendrá diversas consecuencias en los problemas que hemos mencionado en el capítulo anterior. En el Capítulo 4 veremos cómo utilizar el suffix array de un string para resolver tales problemas.

2.2. Una implementación naïve

Un método simple para la construcción del suffix array de un string S de longitud n consiste en crear una lista con todos sus sufijos, y luego ordenar la misma utilizando el algoritmo de merge sort [Knu97].

Observar que para poder almacenar la lista de sufijos, requerimos $\mathcal{O}(n^2)$ memoria y la ejecución de merge sort realiza $\mathcal{O}(n \cdot \log(n))$ comparaciones de sufijos, por lo que la complejidad temporal resultante es $\mathcal{O}(n^2 \cdot \log(n))$. Además, las listas auxiliares requeridas por merge sort hacen que también la complejidad espacial sea $\mathcal{O}(n^2 \log(n))$.

Es de esperar que el algoritmo anterior admita mejoras. Como primer paso en nuestro intento de optimizar el método propuesto, intentaremos mejorar su complejidad espacial. Para ello, una posibilidad es representar a un sufijo con el índice de la posición inicial del mismo.

Ejemplo 2.2.1. El suffix array del string `banana` ahora será representado por la lista de números $[5, 3, 1, 0, 4, 2]$. En el caso general, esto nos permite utilizar $\mathcal{O}(n)$ memoria para representar la lista de sufijos, en vez de $\mathcal{O}(n^2)$.

Otra optimización es la siguiente. Al momento de determinar el menor entre dos sufijos durante la ejecución del algoritmo de ordenación, se puede comparar ambos sin necesidad de crear copias de los mismos mediante un chequeo *in place*. Es decir, al momento de hacer la comparación, iterar carácter a carácter mientras los mismos coincidan. Esto resulta en una complejidad espacial de $\mathcal{O}(n \log(n))$, debido a que las listas auxiliares de strings utilizadas internamente por merge sort son ahora listas de números. En otras palabras, en vez de computar la lista A como en el Ejemplo 2.1.6, obtendremos el arreglo de enteros dado en el Ejemplo 2.2.1.

Existen incluso técnicas de optimización de este algoritmo que requieren una cantidad de memoria extra constante [HL92], logrando así que la complejidad espacial resultante sea $\mathcal{O}(n)$.

2.3. Construcción en $\mathcal{O}(n \log^2(n))$

El objetivo en esta sección es describir un algoritmo que sea capaz de construir el suffix array de un string de longitud n en complejidad temporal $\mathcal{O}(n \log^2(n))$. Veremos que esta versión del algoritmo tiene complejidad espacial $\mathcal{O}(n)$, pero vamos a poner foco primero únicamente en la complejidad temporal.

Definición 2.3.1. Dado un string $S \in \Sigma^*$, denotaremos por $\text{ord}(S)$ a la lista de enteros dada por

$$\text{ord}(S)_i = \text{ord}_i(S)$$

Ejemplo 2.3.2. Sea $S =$ `abac`, tenemos que

$$\text{ord}(S) = [0, 2, 1, 3]$$

Dado que la construcción del suffix array tiene ciertos tecnicismos, es necesario probar algunos resultados preliminares que nos serán de utilidad. Asumiremos siempre que el string S tiene longitud n.

Lema 2.3.3. *Para todo string $S \in \Sigma^*$ se cumple que*

$$S_{i\rightarrow} \leq S_{j\rightarrow} \iff \text{ord}_i(S) \leq \text{ord}_j(S).$$

Demostración.

($\Rightarrow$) Sea $k \in [0, n)$, vamos a analizar los siguientes casos.

- Si $S_{k\rightarrow} < S_{i\rightarrow}$, como $\leq$ es una relación transitiva, vale que $S_{k\rightarrow} < S_{j\rightarrow}$, y por lo tanto, $S_{k\rightarrow}$ es considerado en $\text{ord}_i(S)$ y $\text{ord}_j(S)$.
- Si $S_{i\rightarrow} \leq S_{k\rightarrow} < S_{j\rightarrow}$, entonces $S_{k\rightarrow}$ es considerado en $\text{ord}_j(S)$, pero no en $\text{ord}_i(S)$.

- Si $S_{j\rightarrow} \leq S_{k\rightarrow}$, entonces $S_{k\rightarrow}$ no es considerado en $\text{ord}_i(S)$ ni en $\text{ord}_j(S)$.

Se puede observar entonces que todo sufijo considerado en $\text{ord}_i(S)$ es también considerado en $\text{ord}_j(S)$, por lo que $\text{ord}_i(S) \leq \text{ord}_j(S)$.

($\Leftarrow$) Supongamos que $S_{j\rightarrow} < S_{i\rightarrow}$. Mediante un análisis de casos similar al realizado previamente, obtenemos que $\text{ord}_j(S) < \text{ord}_i(S)$. Entonces concluimos que $S_{i\rightarrow} \leq S_{j\rightarrow}$. Esto contradice la hipótesis inicial, de modo que hemos llegado a una contradicción y el resultado sigue. □

Corolario 2.3.4. *El suffix array de S es igual al suffix array de* $\text{ord}(S)$.

Demostración. Es un resultado directo del Lema 2.3.3. □

Ejemplo 2.3.5. Sea $S = \texttt{abac}$, el suffix array de S es

$$[\texttt{abac}, \texttt{ac}, \texttt{bac}, \texttt{c}],$$

o bien, equivalentemente,

$$[0, 2, 1, 3].$$

Además, considerando $\text{ord}(S)$ como vimos en el Ejemplo 2.3.2, el suffix array de esta lista de números es

$$[[0, 2, 1, 3], [1, 3], [2, 1, 3], [3]],$$

o bien, equivalentemente,

$$[0, 2, 1, 3].$$

Notar que la lista resultante en ambos casos es idéntica.

Utilizando lo obtenido en el Corolario 2.3.4, cuando deseemos obtener el suffix array de la palabra S, computaremos en cambio el de la lista de números $\text{ord}(S)$. Esto nos permite que la construcción de la estructura sea totalmente independiente de Σ, ya que siempre se manipularán números enteros en el rango $[0, n)$ durante el proceso.

2.3.1. Algoritmo de Manber y Myers

En el año 1990, Udi Manber y Gene Myers introdujeron un método eficiente para la construcción del suffix array de un string [MM93]. El mismo fue el precursor de distintas investigaciones y publicaciones, en las que se ha utilizado el algoritmo como base para optimizar ciertos escenarios particulares, o bien obtener relaciones y aprovechar similitudes con otras estructuras de datos para lograr la construcción del arreglo ordenado de sufijos en complejidades aún mejores que la propuesta en el trabajo original.

Previo a describir el algoritmo de Manber y Myers, debemos introducir algunas definiciones que simplificarán la explicación.

Definición 2.3.6. Dado un alfabeto Σ. Sean S y T strings de Σ^*, y k un entero positivo, denotaremos con $\leq_k$ a la relación sobre Σ^* tal que

$$S \leq_k T \iff S_{0,k} \leq T_{0,k}$$

De manera análoga definimos a las relaciones $<_k$ y $=_k$.

Ejemplo 2.3.7. Sea $S =$ `abaco` y $T =$ `abeja`. Tenemos que $S =_2 T$ y que $S <_3 T$.

Definición 2.3.8. Denotaremos como $\text{rel}_k(S)$ a la lista de números tal que $\text{rel}_k(S)_i$ representa la cantidad de posiciones $0 \leq j < n$ tales que $S_{j\rightarrow} <_k S_{i\rightarrow}$. Además, asumiremos que $\text{rel}_k(S)_i = -1$ en caso de que $i \geq n$.

En otras palabras $\text{rel}_k(S)$ es un arreglo de enteros no negativos tal que su posición i-ésima cuenta cuántos sufijos de S son estrictamente menores que el i-ésimo sufijo, cuando uno sólo compara mirando los primeros k carácteres.

Ejemplo 2.3.9. Sea $S =$ `abaab`, tenemos que

$$\text{rel}_1(S) = [0, 3, 0, 0, 3],$$
$$\text{rel}_2(S) = [1, 4, 0, 1, 3],$$
$$\text{rel}_3(S) = [2, 4, 0, 1, 3],$$
$$\text{rel}_4(S) = [2, 4, 0, 1, 3],$$
$$\text{rel}_5(S) = [2, 4, 0, 1, 3].$$

Lema 2.3.10. *Consideremos $S \in \Sigma^*$. Para todo $k \geq |S|$ vale que*

$$\text{rel}_k(S)_i = \text{ord}_i(S).$$

Demostración. Asumamos que $|S| = n$. Para $0 \leq i < n$ tenemos que $S_{i,i+k} = S_{i\rightarrow}$. En particular, usando las definiciones obtenemos que la relación $\leq_k$ es equivalente a $\leq$. Se obtiene entonces que $\text{rel}_k(S)_i = \text{ord}_i(S)$. □

Asumamos que al calcular $\text{rel}_k(S)$, obtenemos una permutación de largo n. Entonces, poseemos suficiente información para determinar el menor entre cualquier par de sufijos: de hecho, el suffix array resultante será $\text{inv}(\text{rel}_k(S))$.

Ejemplo 2.3.11. Supongamos que $S =$ `arbol`, tenemos que $\text{rel}_1(S) = [0, 4, 1, 3, 2]$. La permutación inversa, en este caso, está dada por $\text{inv}(\text{rel}_1(S)) = [0, 2, 4, 3, 1]$.

0:	`arbol`
2:	`bol`
4:	`l`
3:	`ol`
1:	`rbol`

Suffix array para la palabra $S =$ `arbol`, donde se observa la equivalencia del mismo con $\text{inv}(\text{rel}_1(S))$.

En cambio, si lo que propusimos anteriormente no ocurre, existen al menos dos posiciones i y j en $\text{rel}_k(S)$ que tienen el mismo valor. Esto quiere decir que no hay suficiente información para generar el arreglo de sufijos si consideramos únicamente los primeros k símbolos de cada uno de ellos, pues $S_{i\rightarrow} =_k S_{j\rightarrow}$.

Ejemplo 2.3.12. Si la palabra a considerar es $S =$ `texto`, al calcular $\text{rel}_1(S)$ obtenemos la lista $[2, 0, 4, 2, 1]$, donde $S_{0\rightarrow} =_1 S_{3\rightarrow}$. En este caso, debemos incrementar el valor de k para ser capaces de determinar el menor entre estos dos sufijos.

El algoritmo propuesto por Manber y Myers para la construcción del suffix array [MM93] se aprovecha del Lema 2.3.10 para el cómputo del mismo. Se comienza por calcular $\text{rel}_1(S)$, y se inicializa $k = 1$. Luego, mientras $\text{rel}_k(S)$ no sea una permutación de los enteros entre 0 y $n - 1$, se procede a calcular $\text{rel}_{2k}(S)$ utilizando la información provista por $\text{rel}_k(S)$ para evitar el uso de comparaciones entre strings.

2.3.2. Cómputo de $\text{rel}_1(S)$

Si llamamos $\text{letras}(S)$ a la lista ordenada de todos los símbolos que ocurren en S, un posible algoritmo para realizar el cálculo de $\text{rel}_1(S)$ consiste en computar $\text{letras}(S)$ (ordenando el string S mediante *merge sort* y eliminando del mismo elementos repetidos). Luego, para conocer el valor de $\text{rel}_1(S)_i$ se realiza una búsqueda binaria con el objetivo de obtener la posición del símbolo S_i en la lista computada. El algoritmo propuesto tiene una complejidad temporal igual a $\mathcal{O}(n \cdot \log(n))$, y utiliza $\mathcal{O}(n)$ memoria extra.

Algoritmo 5 Cálculo de $\text{rel}_1(S)$ en $\mathcal{O}(n \cdot \log(n))$

```
1:  A ← letras(S)
2:  rel_1 ← list(n)                          ▷ Lista de tamaño n
3:  for i ← 0 to n − 1 do
4:      l ← 0
5:      r ← |A|
6:      while l + 1 < r do
7:          mid ← ⌊(l+r)/2⌋
8:          if S_i ≤' A_mid then
9:              l ← mid
10:         else
11:             r ← mid
12:         end if
13:     end while
14:     rel_{1 i} ← l                        ▷ S_i = A_l ⇒ ord(S_i) = l
15: end for
16: return rel_1
```

2.3.3. Cómputo de $\text{rel}_{2k}(S)$ utilizando $\text{rel}_k(S)$

Procederemos a describir un método que, dados enteros $i, j \in [0, n)$, nos permitirá comparar $S_{i\rightarrow}$ y $S_{j\rightarrow}$ utilizando la relación $\leq_{2k}$. El mismo utilizará información provista por la lista $\text{rel}_k(S)$ para realizar el cómputo necesario. Existen distintos escenarios a considerar:

(1) Si $\text{rel}_k(S)_i > \text{rel}_k(S)_j$, tenemos que $S_{i\rightarrow} >_k S_{j\rightarrow}$. Esto implica que la longitud de $\text{lcp}(S_{i\rightarrow}, S_{j\rightarrow})$ es menor a k, por lo que concluimos que $S_{i\rightarrow} >_{2k} S_{j\rightarrow}$.

(2) Si $\text{rel}_k(S)_i < \text{rel}_k(S)_j$, de manera similar al escenario anterior, se puede ver que $S_{i\rightarrow} <_{2k} S_{j\rightarrow}$.

(3) Si $\text{rel}_k(S)_i = \text{rel}_k(S)_j$, entonces este escenario es más desafiante, pues no es suficiente con considerar los primeros k símbolos de los sufijos para determinar el menor entre ambos. Es por esto que debemos utilizar los k símbolos siguientes de cada sufijo para determinar la relación entre ellos. La observación principal es que para este caso también poseemos toda la información necesaria en $\text{rel}_k(S)$: es suficiente con comparar $\text{rel}_k(S)_{i+k}$ con $\text{rel}_k(S)_{j+k}$. La relación entre estos dos números determinará la relación entre $S_{i\rightarrow}$ y $S_{j\rightarrow}$ si consideramos $\leq_{2k}$ en vez de $\leq_k$.

Ejemplo 2.3.13. Sea $S =$ `abdaabdb`, observemos que

$$\text{rel}_1(S) = [0, 3, 6, 0, 0, 3, 6, 3],$$

$$\text{rel}_2(S) = [1, 4, 6, 0, 1, 4, 7, 3],$$

$$\text{rel}_4(S) = [1, 4, 6, 0, 2, 5, 7, 3].$$

Podemos notar que $\text{rel}_2(S)_0$ será igual a $\text{rel}_2(S)_4$, pues tenemos que se cumple $\text{rel}_1(S)_0 = \text{rel}_1(S)_4$, y además $\text{rel}_1(S)_1 = \text{rel}_1(S)_5$.

Además, observemos que se cumple $\text{rel}_4(S)_0 < \text{rel}_4(S)_4$. Esto es gracias a que $\text{rel}_2(S)_0 = \text{rel}_2(S)_4$, y además $\text{rel}_2(S)_2 = \text{rel}_2(S)_6$.

Si miramos detenidamente las comparaciones que hemos realizado en cada uno de los escenarios, queda a la vista que el método descrito es equivalente a comparar pares de números utilizando el orden lexicográfico para ellos. Más precisamente, si denotamos como $\text{order}(i, k)$ al par de elementos $(\text{rel}_k(S)_i, \text{rel}_k(S)_{i+k})$, lo que estamos haciendo es esencialmente comparar lexicográficamente $\text{order}(i, k)$ con $\text{order}(j, k)$. El hecho fundamental es que para efectuar esta comparación se realizan $\mathcal{O}(1)$ operaciones.

Sea A la lista obtenida al ordenar $\text{iden}(n)$ mediante el algoritmo *merge sort*, utilizando como comparador al método recién descripto (es decir, $i < j$ si vale que $\text{order}(i, k) < \text{order}(j, k)$). Como A_0 representa la posición inicial del mínimo sufijo si consideramos la relación $\leq_{2k}$, tenemos que $\text{rel}_{2k}(S)_{A_0} = 0$. Para el resto de los elementos de A, hay dos casos:

- Si $\text{order}(A_i, k) = \text{order}(A_{i-1}, k)$, como los sufijos no tienen diferencias en los primeros $2k$ símbolos, obtenemos que $\text{rel}_{2k}(S)_{A_i} = \text{rel}_{2k}(S)_{A_{i-1}}$.

- Si $\text{order}(A_i, k) > \text{order}(A_{i-1}, k)$, entonces $\text{rel}_{2k}(S)_{A_i} = \text{rel}_{2k}(S)_{A_{i-1}} + 1$.

Algoritmo 6 Cómputo de $\text{rel}_{2k}(S)$ utilizando $\text{rel}_k(S)$ en $\mathcal{O}(n \cdot \log(n))$

```
1: A ← iden(n)
2: sort_by_order_k(A)
3: rel2k ← list(n)                          ▷ Lista de n enteros
4: rel2k_{A_0} ← 0
5: for i ← 1 to n − 1 do
6:     if order(A_i, k) = order(A_{i−1}, k) then
7:         rel2k_{A_i} ← rel2k_{A_{i−1}}
8:     else
9:         rel2k_{A_i} ← rel2k_{A_{i−1}} + 1
10:    end if
11: end for
12: return rel2k
```

2.3.4. Análisis de complejidad

Como el suffix array de un string S es una permutación, el Corolario 2.3.4 nos dice que $\text{ord}(S)$ también lo es. Entonces, gracias al Lema 2.3.10 tenemos que para $k \geq n$, $\text{ord}_k(S)$ es una permutación.

Dado que el algoritmo propuesto duplica el valor de k luego de calcular $\text{rel}_k(S)$, la cantidad de pasos realizados hasta que k supere a n está acotada por $\lceil \log(n) \rceil$. Como en cada uno de los cálculos de las listas $\text{rel}_k(S)$ se realizan $\mathcal{O}(n \cdot \log(n))$ operaciones, la complejidad total del algoritmo es $\mathcal{O}(n \cdot \log^2(n))$.

2.4. Construcción en $\mathcal{O}(n \cdot \log(n))$

Los algoritmos de ordenación basados en comparaciones, como *merge sort*, tienen una complejidad temporal de $\Omega(n \cdot \log(n))$ [Cor01]. A pesar de esto, existen escenarios en los que es posible ordenar una lista de elementos mediante el uso de otros métodos, que alcanzan incluso la complejidad óptima de $\mathcal{O}(n)$.

2.4.1. Counting sort

Sea L una lista de n números enteros. Definimos como $\text{máx}(L)$ al máximo elemento de L, y a $\text{mín}(L)$ como el mínimo de la misma. Como primera instancia, y a modo de simplificación, restaremos $\text{mín}(L)$ a cada elemento de la lista. De esta manera, todos los elementos de L se encontrarán ahora en el rango $[0, k)$, donde $k = \text{máx}(L) - \text{mín}(L) + 1$.

Counting sort [Cor01] es un algoritmo de ordenación para listas de números que posee una complejidad temporal de $\mathcal{O}(n+k)$, utilizando $\mathcal{O}(k)$ memoria extra. La idea detrás del mismo es computar la lista freq de tamaño k, tal que freq_i representa la cantidad de posiciones j de L que cumplen $L_j = i$.

Denotaremos con L' a la lista que resulta de ordenar L. Definimos al arreglo prev, de tamaño $k+1$, tal que $\text{prev}_i = \sum_{j=0}^{i-1} \text{freq}_j$. La observación principal del algoritmo consiste en notar que $L'_j = i$ si ocurre que $\text{prev}_i \leq j < \text{prev}_{i+1}$. Utilizando esto, se puede computar L' iterando el valor de i, y utilizando la información guardada en prev para llenar las distintas posiciones de la lista ordenada.

Ejemplo 2.4.1. Sea $L = [1, 0, 3, 2, 0, 2]$, las listas computadas por counting sort son:

$$\text{freq} = [2, 1, 2, 1]$$
$$\text{prev} = [0, 2, 3, 5, 6]$$
$$L' = [0, 0, 1, 2, 2, 3]$$

Notar, por ejemplo, que las posiciones de L' que tienen el valor 2 son las del rango $[\text{prev}_2, \text{prev}_3)$

En este trabajo, utilizaremos una idea muy similar para ordenar la lista $\text{iden}(n)$, considerando que i se ubicará antes que j si $L_i < L_j$, o bien $L_i = L_j$ e $i < j$. Notemos que, en caso de que freq_i sea mayor a 0, prev_i es la primer posición en la lista final, donde el valor guardado será un índice p tal que $L_p = i$. Esta información nos permitirá asignar secuencialmente valores a las distintas posiciones de L', usando en todo momento prev_i como indicador de la menor posición aún no utilizada en L' donde el valor final será un índice de L que contenga el valor i.

El algoritmo propuesto iterará la lista L de izquierda a derecha. Sea p la posición actual de L y $x = L_p$. Sabemos que, si ordenáramos la lista L, la posición prev_x poseería el valor x. Es por esto, que podemos realizar la asignación $L'_{\text{prev}_x} = p$, y luego incrementar en 1 el valor de prev_x. De esta manera, la próxima posición de L que contenga el valor x se ubicará en una posición contigua a la de p en L'. Podemos notar que el método descrito mantiene el orden relativo entre posiciones de L que contengan valores idénticos.

Ejemplo 2.4.2. Sea $L = [1, 0, 2, 1, 0, 1]$, se muestran a continuación los sucesivos pasos realizados por el método propuesto para generar la lista L'. En la izquierda, se muestra en rojo el elemento que se itera actualmente en L, mientras que en la derecha se observa la lista L' a medida que avanza la ejecución del algoritmo.

$$[\mathbf{1}, 0, 2, 1, 0, 1] \to [-, -, \underbrace{0}_{\text{prev}_1}, -, -, -]$$
$$[1, \mathbf{0}, 2, 1, 0, 1] \to [\underbrace{1}_{\text{prev}_0}, -, 0, -, -, -]$$
$$[1, 0, \mathbf{2}, 1, 0, 1] \to [1, -, 0, -, -, \underbrace{2}_{\text{prev}_2}]$$
$$[1, 0, 2, \mathbf{1}, 0, 1] \to [1, -, 0, \underbrace{3}_{\text{prev}_1}, -, 2]$$
$$[1, 0, 2, 1, \mathbf{0}, 1] \to [1, \underbrace{4}_{\text{prev}_0}, 0, 3, -, 2]$$
$$[1, 0, 2, 1, 0, \mathbf{1}] \to [1, 4, 0, 3, \underbrace{5}_{\text{prev}_1}, 2]$$

Algoritmo 7 Ordenamiento de la lista L utilizando *counting sort* en $\mathcal{O}(n+k)$

```
1: freq ← list(k + 1)
2: prev ← list(k + 1)
3: for i ← 0 to n − 1 do
4:     freq_{L_i} ← freq_{L_i} + 1
5: end for
6: prev_values ← 0
7: for i ← 0 to k do
8:     prev_i ← prev_values
9:     prev_values ← prev_values + freq_i
10: end for
11: L' ← list(n)
12: for i ← 0 to n − 1 do
13:     L'_{prev_{L_i}} ← i
14:     prev_{L_i} ← prev_{L_i} + 1
15: end for
16: return L'
```

2.4.2. Radix sort

El algoritmo de counting sort es útil cuando es aceptable una complejidad espacial de $\mathcal{O}(k)$. Es intuitivo plantearse si se puede utilizar el mismo método para ordenar lexicográficamente d-uplas (tuplas con d elementos) tal que todos sus valores sean menores a k. Recordemos que el método antes descrito ordena listas de números. Si quisiéramos plantear una biyección entre d-uplas y números, necesitaríamos k^d valores distintos, por lo que la complejidad espacial resulta exponencial en d. Por ejemplo, un mapeo posible entre los dos conjuntos consiste en relacionar un número x con su representación en base k (las coordenadas de una d-upla representan los coeficientes de un número en la misma base).

Sea L una lista de d-uplas con coordenadas menores a k, denotaremos con L_i^j a la j-ésima coordenada de la tupla ubicada en la posición i de L. El algoritmo de ordenamiento conocido como *radix sort* consiste en correr counting sort d veces sobre la lista L. En la ejecución número i del algoritmo, el criterio de comparación entre dos tuplas es el valor de la coordenada $d-i$ de cada una de ellas, ignorando el resto de los valores.

Lema 2.4.3. *Sea L una lista de d-uplas. Luego de ejecutar las d llamadas a counting sort, el algoritmo de radix sort ordena la lista L lexicográficamente.*

Demostración. Probaremos por medio de inducción que luego de i ejecuciones de counting sort, si consideramos la lista L' de i-uplas compuestas por las últimas i coordenadas de cada tupla de L, entonces L' está ordenada lexicográficamente.

Si suponemos que $i = 1$ el lema es trivialmente verdadero, pues L' es una lista de números, por lo que counting sort realiza el ordenamiento de manera correcta.

Supongamos como hipótesis inductiva que radix sort ordenó lexicográficamente las últimas $x-1$ coordenadas de cada d-upla con las primeras $x-1$ llamadas de

counting sort ($x \geq 2$). Sean $i, j \in [0, n)$ tal que $L_i^{d-x} \leq L_j^{d-x}$. Al ejecutar el algoritmo una vez más, existen dos escenarios:

- Si $L_i^{d-x} \neq L_j^{d-x}$, counting sort considerará entonces a i antes de j en el ordenamiento. Esto es correcto, pues el orden lexicográfico prioriza la primera coordenada antes que las siguientes.
- Si $L_i^{d-x} = L_j^{d-x}$, como hemos observado en el Algoritmo 7, el par de índices a comparar en counting sort almacenan el mismo valor, por lo que se preserva el orden relativo calculado en los $x-1$ pasos anteriores, el cual por hipótesis inductiva, es lexicográficamente correcto. □

Podemos observar que la complejidad temporal total de ejecutar radix sort para ordenar a la lista L es de $\mathcal{O}((n+k) \cdot d)$. Si en una llamada de counting sort se reutilizan los arreglos freq y prev de la ejecución anterior, la memoria utilizada por el algoritmo es $O(k)$.

2.4.3. Optimizando el algoritmo de Manber y Myers

El algoritmo descrito en la Sección 2.3.1 calcula las listas $\text{rel}_k(S)$ para $\mathcal{O}(\log(n))$ valores distintos de k. En cada uno de ellos, utiliza merge sort para ordenar las listas de pares $\text{order}(i, k)$. Por definición de $\text{rel}_k(S)$, cada coordenada de los pares a ordenar tiene valores que se encuentran en el rango $[0, n)$. Esto permite que radix sort compute la lista ordenada de estos pares utilizando $\mathcal{O}(n)$ memoria extra, y logrando una complejidad temporal de $\mathcal{O}((n+n) \cdot 2) = \mathcal{O}(n)$.

Gracias a la optimización propuesta, cada uno de los $\mathcal{O}(\log(n))$ pasos ejecutados por el algoritmo de Manber y Myers puede ser realizado con una cantidad lineal de operaciones, por lo que el cómputo del suffix array del string S alcanza una complejidad temporal final de $\mathcal{O}(n \cdot \log(n))$, utilizando $\mathcal{O}(n)$ memoria extra.

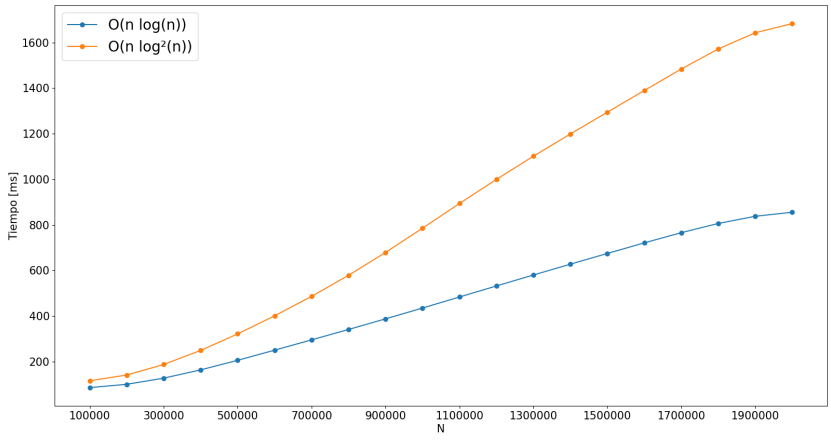

Figura 2.1: Tiempo de ejecución de la construcción de suffix array para el string $S = \mathtt{a}^n$. Este es el peor caso con respecto a la complejidad temporal del algoritmo, pues $\text{rel}_k(S)$ no es una permutación para ningún k tal que $k < n$.

2.5. Construcción en $\mathcal{O}(n)$

Existen métodos aún más veloces que realizan el cómputo de suffix array, logrando una complejidad temporal de $\mathcal{O}(n)$. Uno de ellos se enfoca en construir primero una estructura distinta sobre el string S, llamada *suffix tree* [Wei73], y luego generar el suffix array a partir de la información provista por ella [KSB06]. Hay otros algoritmos que incluso logran realizar el cómputo necesario en complejidad lineal sin depender de otras estructuras, como el algoritmo de Kim [KSPP05].

A pesar de que la descripción y el análisis de este método excede los objetivos propuestos en este trabajo, asumiremos de ahora en más que cuando sea necesaria la construcción del suffix array de un string S de longitud n, tanto el costo computacional como la memoria extra requerida será de $\mathcal{O}(n)$.

Capítulo 3

Longest Common Prefix array

3.1. Introducción

Sea P una permutación. Con el objetivo de simplificar la redacción, nos referiremos a $\text{inv}(P)$ como *pos*, y además denotaremos como $\text{lcp}_{i,j}(S)$ al valor de $\text{lcp}(S_{i\rightarrow}, S_{j\rightarrow})$.

Definición 3.1.1. Dado un alfabeto Σ. Sea S un string de longitud n en Σ^* y A su respectivo suffix array. El *longest common prefix array* (abreviado como *lcp array*) de S es una lista de números H de longitud n, la cual contiene el valor de la función lcp para sufijos de S que se encuentran en posiciones consecutivas del suffix array, es decir:

- $H_0 = 0$
- $H_i = \text{lcp}_{A_{i-1},A_i}(S)$

Ejemplo 3.1.2. Sea $S = \texttt{abbaab}$. La lista ordenada de sufijos es

$$[\texttt{aab}, \texttt{ab}, \texttt{abbaab}, \texttt{b}, \texttt{baab}, \texttt{bbaab}].$$

En este caso, el lcp array de S estará representado por la lista

$$H = [0, 1, 2, 0, 1, 1].$$

Se puede notar, por ejemplo, que $H_2 = 2$, pues tenemos que $\text{lcp}(\texttt{ab}, \texttt{abbab}) = 2$.

La construcción del lcp array H de un string S está íntimamente relacionado con el suffix array A del mismo: existen algoritmos que lo computan durante la ejecución del algoritmo de Manber y Myers descrito en la Sección 2.3.1, mientras que otros utilizan la información de A luego de que esta lista ha sido ya construida.

A pesar de solo contener información que relaciona sufijos consecutivos de A, el arreglo H permite el cálculo de la función lcp entre cualquier par de sufijos de S. Esto, como veremos más adelante, nos permitirá optimizar la complejidad de soluciones a distintos problemas.

Lema 3.1.3. *Sea S un string de largo n, su lcp array H y el inverso de su suffix array* pos. *Dadas dos posiciones i, $j \in [0, n)$ tales que* $\text{pos}_i < \text{pos}_j$, *se cumple que* $\text{lcp}_{i,j}(S) \leq H_k$, *para todo* $k \in [\text{pos}_i + 1, \text{pos}_j]$.

Demostración. Consideremos un valor de k arbitrario que se encuentre en el rango $[\text{pos}_i + 1, \text{pos}_j]$, y supongamos que $\text{lcp}_{i,j}(S) > H_k$. Como $S_{i\rightarrow}$ y $S_{j\rightarrow}$ comparten al menos sus primeros $H_k + 1$ símbolos, todos los sufijos que se encuentren entre ellos en el suffix array deben cumplir la misma condición, pues si esto no ocurriese para una posición inicial x de un sufijo, entonces valdría que $\text{pos}_x < \text{pos}_i$ o $\text{pos}_x > \text{pos}_j$, y en consecuencia $S_{x\rightarrow}$ no se encontraría entre $S_{i\rightarrow}$ y $S_{j\rightarrow}$ en A. Pero entonces, en particular, tenemos que los sufijos $S_{A_{k-1}\rightarrow}$ y $S_{A_k\rightarrow}$ comparten al menos sus primeros $H_k + 1$ símbolos. Esta contradicción nos permite concluir entonces que $\text{lcp}_{i,j}(S)$ no puede ser mayor a H_k. □

Corolario 3.1.4. *Sea S un string de largo n, su lcp array H y el inverso de su suffix array* pos. *Dadas dos posiciones i, $j \in [0, n)$ tales que* $\text{pos}_i < \text{pos}_j$, *se cumple que* $\text{lcp}_{i,j}(S) = \min\limits_{k=\text{pos}_i+1}^{\text{pos}_j} H_k$.

Demostración. Sea $m = \min\limits_{k=\text{pos}_i+1}^{\text{pos}_j} H_k$. Todo sufijo $S_{k\rightarrow}$ tal que $k \in [\text{pos}_i, \text{pos}_j]$ debe compartir los primeros m símbolos, pues si eso no ocurriera para un valor de k, entonces $H_k < m$ (lo cual es absurdo dado que m es, por definición, el mínimo de los valores de H en el rango). Pero entonces tenemos que, en particular, $S_{i\rightarrow}$ y $S_{j\rightarrow}$ comparten sus primeros m símbolos, por lo que $\text{lcp}_{i,j}(S) \geq m$. Además, gracias al Lema 3.1.3, sabemos que m es también una cota superior para $\text{lcp}_{i,j}(S)$. Utilizando las observaciones realizadas, podemos concluir finalmente que se cumple la igualdad $\text{lcp}_{i,j}(S) = m$. □

3.2. Construcción en $\mathcal{O}(n \cdot \log(n))$

Hemos mencionado previamente que existe una importante relación entre la construcción del lcp array de un string S, y el suffix array del mismo. Es natural entonces intentar aprovechar el algoritmo propuesto por Manber y Myers para computar, de forma simultánea, el arreglo de lcp.

Sea k tal que $\text{rel}_k(S)$ es una de las $\lceil \log(n) + 1 \rceil$ listas calculadas en la construcción del suffix array de la palabra. Además, denotemos como P^k a la permutación retornada por el algoritmo de counting sort propuesto en el Algoritmo 7 al ejecutarlo para ordenar $\text{rel}_k(S)$. Si ocurre que $\text{rel}_k(S)_{P_i^k} \neq \text{rel}_k(S)_{P_{i+1}^k}$, entonces $\text{lcp}(S_{P_i^k\rightarrow}, S_{P_{i+1}^k\rightarrow}) < k$. En el caso contrario, sabemos que el valor de la función lcp para el par de sufijos es al menos k.

Observación 3.2.1. Sea $x_1 = P_i^k$, $y_1 = P_{i+1}^k$, $x_2 = P_i^{2k}$, $y_2 = P_{i+1}^{2k}$, se cumple que:

$$\text{rel}_k(S)_{x_1} \neq \text{rel}_k(S)_{y_1} \implies \text{rel}_{2k}(S)_{x_2} \neq \text{rel}_{2k}(S)_{y_2}$$

Además, de forma análoga, tenemos que:

$$\text{rel}_{2k}(S)_{x_2} = \text{rel}_{2k}(S)_{y_2} \implies \text{rel}_k(S)_{x_1} = \text{rel}_k(S)_{y_1}$$

Procederemos a describir un método para calcular el valor de cada posición del lcp array de manera recursiva. Durante el proceso, y luego de obtener $\text{rel}_k(S)$, se mantendrá como invariante que H_j ya ha sido computado previamente si $H_j < k$.

Luego de calcular $\text{rel}_1(S)$, es fácil notar que si $P^1_{i-1} \neq P^1_i$, entonces $H_i = 0$, pues el primer símbolo de los sufijos es distinto. Además asumiremos que, inicialmente, $H_0 = 0$ y $H_i = \infty$ si $i > 0$ y $P^1_{i-1} = P^1_i$.

Sea $i \in [1, n)$. Por simplicidad, denotaremos $v_1 = \text{lcp}(S_{A_{i-1}\rightarrow}, S_{A_i\rightarrow})$. Supongamos que, luego de calcular $\text{rel}_{2k}(S)$ tenemos que $P^{2k}_{i-1} \neq P^{2k}_i$, y además $P^k_{i-1} = P^k_i$. Gracias a esto, concluimos que $k \leq v_1 < 2k$, por lo que, para mantener la invariante propuesta, el valor de H_i debe ser computado antes de proceder con el siguiente paso del algoritmo que construye el suffix array de S. Como $v_1 \geq k$, vale que:

$$v_1 = k + \text{lcp}(S_{A_{i-1}+k\rightarrow}, S_{A_i+k\rightarrow})$$

Sea $v_2 = \text{lcp}(S_{A_{i-1}+k\rightarrow}, S_{A_i+k\rightarrow})$, $a = \text{pos}_{A_{i-1}+k}$ y $b = \text{pos}_{A_i+k}$. Como $v_1 < 2k$, v_2 debe ser estrictamente menor a k. Utilizando la igualdad introducida en el Corolario 3.1.4:

$$v_2 = \min_{j=\text{mín}(a,b)+1}^{\text{máx}(a,b)} H_j \tag{3.1}$$

Gracias a la invariante propuesta, todos los valores de $j \in [0, n)$ que cumplen $H_j < k$ fueron calculados en pasos previos. En particular, el mínimo del rango $[\text{mín}(a, b) + 1, \text{máx}(a, b)]$ en el arreglo final es uno de ellos, por lo que el valor de v_2 computado en (3.1) será correcto y consistente.

El valor de H_j se calcula sólo cuando $P^{2k}_{i-1} \neq P^{2k}_i$ y $P^k_{i-1} = P^k_i$. Gracias a la Observación 3.2.1, ambas condiciones se cumplen para un único valor de k, por lo que H_j nunca se calculará más de una vez.

Al momento de computar H_j para distintos valores de j, debemos ser capaces de realizar reiteradas veces dos tipos de operaciones sobre el arreglo H:

- query(l, r): Obtener el valor del mínimo en el rango $[l, r]$ de la lista.
- set(x, v): Asignar el valor v a la posición x de la lista.

Iterar todas las posiciones de un rango para obtener el mínimo valor del mismo causa que la cantidad de operaciones realizadas para obtener el valor de H_j sean $\mathcal{O}(n)$, logrando así una complejidad temporal de $\mathcal{O}(n^2)$ para computar el lcp array, sin la necesidad de utilizar memoria extra.

La estructura de datos conocida como *segment tree* (ver el Apéndice A.2) permite realizar ambos tipos de operaciones sobre una lista de tamaño n, utilizando en total $\mathcal{O}(n)$ memoria extra, con un costo operacional de $\mathcal{O}(\log(n))$ por cada una de ellas. De esta manera, el algoritmo propuesto para el cómputo del lcp array durante la creación del suffix array resulta en una complejidad temporal final de $\mathcal{O}(n \cdot \log(n))$.

Algoritmo 8 Cómputo de posiciones j del lcp array tales que $k \leq H_j < 2k$

```
1: for i ← 1 to n − 1 do
2:     if P^k_{i−1} = P^k_i and P^{2k}_{i−1} ≠ P^{2k}_i then
3:         a ← pos_{A_{i−1}+k}
4:         b ← pos_{A_i+k}
5:         if a > b then
6:             swap(a, b)
7:         end if
8:         H_i ← query(a + 1, b)
9:         set(i, H_i)
10:    end if
11: end for
```

3.3. Construcción en $\mathcal{O}(n)$

En los últimos años se han logrado importantes avances en el estudio de suffix array, descubriendo métodos que construyen el mismo en $\mathcal{O}(n)$. Esto motiva a intentar construir el lcp array en la misma complejidad, y así lograr un algoritmo completamente lineal (tanto en cantidad de operaciones como memoria extra utilizada) que calcule ambas estructuras.

El algoritmo propuesto por Kasai et al. [KLA+01] es uno de los más simples y de los primeros en lograr computar el lcp array con una complejidad temporal y espacial de $\mathcal{O}(n)$. A diferencia del método explicado previamente, el actual utiliza la información provista por el suffix array en su versión final, en vez de construir ambas listas al mismo tiempo.

Lema 3.3.1. *Sea S un string de largo n, su lcp array H, y su suffix array A. Dado $i \in [0, n-1)$. Si $H_{\text{pos}_i} = k$, con $k > 0$, entonces $H_{\text{pos}_{i+1}} \geq k - 1$.*

Demostración. Denotemos $x = \text{pos}_i$. Al haber asumido que $H_x > 0$, tenemos que $x \neq 0$. Como $\text{lcp}(S_{A_x \rightarrow}, S_{A_{x-1} \rightarrow}) = k$, si borráramos el primer símbolo de ambos sufijos, el valor del lcp entre este nuevo par de strings sería igual a $k - 1$.

Sean l y r las posiciones en A donde ocurren los sufijos resultantes de remover la primer letra de $S_{A_{x-1}}$ y S_{A_x} respectivamente, es decir $l = \text{pos}_{A_{x-1}+1}$ y $r = \text{pos}_{i+1}$. El orden relativo entre los sufijos no cambia al eliminar la primer posición de cada uno de ellos, porque al ser $k > 0$, el símbolo inicial en ambas palabras era idéntico. Entonces podemos asumir que $l < r$.

Como el valor de lcp entre ambas palabras es $k - 1$, gracias al Corolario 3.1.4 tenemos que:

$$\min_{j=l+1}^{r} H_j = k - 1$$

Es decir, $k-1$ es una cota inferior para el valor de todas las posiciones de H en el rango $[l+1, r]$. En particular, esto vale para H_r. Pero como $r = \text{pos}_{i+1}$, obtenemos entonces que $H_{\text{pos}_{i+1}} \geq k - 1$. □

El método propuesto para la construcción del lcp array computa inicialmente el valor de la función lcp entre el sufijo de largo n y el que se ubica en la posición anterior adyacente en el suffix array. Luego, considerando el Lema 3.3.1, prosigue a computar el resto de los sufijos en orden decreciente de longitud, reutilizando el valor del lcp obtenido en pasos previos para evitar realizar operaciones redundantes. Notar que, dado un sufijo de largo L, si se desea obtener el de tamaño $L-1$, es suficiente con eliminar el primer símbolo del mismo.

Algoritmo 9 Algoritmo de Kasai et al. para el cómputo del lcp array de S

```
1:  k ← 0
2:  H ← list(n)
3:  for i ← 0 to n do                          ▷ Computar lcp_{i,A_{pos_i−1}}(S)
4:      if pos_i = 0 then
5:          k ← 0
6:      else
7:          j ← A_{pos_i −1}
8:          while i + k < n and j + k < n and S_{i+k} = S_{j+k} do
9:              k ← k + 1
10:         end while
11:     end if
12:     H_{pos_i} ← k
13:     k ← máx(k − 1, 0)        ▷ borrar el primer valor de S_{i→} ⇒ lcp ≥ k − 1
14: end for
```

Para analizar la complejidad temporal del algoritmo, es suficiente con notar que todas las operaciones realizadas consisten en incrementar o decrementar el valor de la variable k. Es trivial que k no puede ser nunca mayor o igual a n o menor a 0. En cada una de las n iteraciones, el valor de k decrece en a lo sumo 1 (excepto cuando $\text{pos}_i = 0$, en donde se le asigna el número 0). Pero entonces la cantidad total de veces que se puede incrementar el valor de k en el ciclo de la linea 8 en el Algoritmo 9 es a lo sumo $3n$. Concluimos entonces que la cantidad de operaciones realizadas por el método es $\mathcal{O}(n)$.

Ejemplo 3.3.2. Sea $S = \texttt{aabaabba}$. El suffix array de S está determinado por la lista de números

$$[7, 0, 3, 1, 4, 6, 2, 5].$$

Además, el lcp array de S se representa con la lista

$$[0, 1, 3, 1, 2, 0, 2, 1].$$

Podemos observar allí que se cumple la propiedad $H_{\text{pos}_i} \geq H_{\text{pos}_{i-1}} - 1$ para todo $i \in [1, |S|]$.

Capítulo 4

Aplicaciones

4.1. Introducción

Durante el desarrollo de esta sección, abordaremos distintos casos de uso y aplicaciones en las que las estructuras previamente descritas juegan un rol fundamental. En particular, se plantearán soluciones a los problemas introducidos en la Sección 1.2.

Como ha sido discutido previamente, y a modo de recordatorio, en esta sección asumiremos que, a pesar de solo haber mencionado su existencia sin entrar en detalles, los algoritmos utilizados para la construcción del suffix array y el longest common prefix array de un string S de longitud n tendrán una complejidad temporal y espacial de $\mathcal{O}(n)$. Además siempre asumiremos, sin pérdida de generalidad, que Σ estará compuesto únicamente de las letras minúsculas del alfabeto inglés.

4.2. Contar ocurrencias de un patrón en un texto

El problema principal que motivó el desarrollo y la creación del suffix array fue el de, dados strings S y T de largo n y m respectivamente, contar la cantidad de ocurrencias exactas de S en T. La solución propuesta en la Sección 1.2.1 posee una complejidad temporal de $\mathcal{O}(n \cdot m)$. Intentaremos utilizar los algoritmos descritos en este trabajo para alcanzar una mejora en el rendimiento.

4.2.1. Una propuesta de complejidad lineal: $\mathcal{O}(n + m)$

Sea `#` un símbolo que no se encuentra en Σ, el cual cumple que `#` $<'$ c para todo $c \in \Sigma$. Fijos S y T como antes, denotaremos con S' al string perteneciente a $(\Sigma \sqcup \{\texttt{\#}\})^*$ que cumple $S' = S + \texttt{\#} + T$, y nos referiremos a `#` como el símbolo *separador* entre S y T.

Sean A' y H' el suffix array y el lcp array correspondientes al string S'. Además, pos$'$ denotará a la permutación inversa de A'. Notar que los sufijos de T se en-

cuentran representados en A' por los valores en el rango $[n+1, n+m]$, pues los primeros $n+1$ representan sufijos que tienen su posición inicial en S, o bien en el símbolo especial utilizado como separador entre S y T.

Denotamos como p a la posición del suffix array en donde se encuentra el sufijo que representa la palabra S' (es decir, $p = \text{pos}'_0$). Dado que el símbolo separador no pertenece a Σ, el valor de H'_p no puede exceder n. En particular, si x es una posición inicial de una ocurrencia de S en T que cumple $\text{pos}'_x < p$, se debe cumplir que

$$\min_{j=\text{pos}'_x+1}^{p} H'_j = n.$$

En otras palabras, si hay una ocurrencia de S en T a partir de x, todo elemento del rango $[\text{pos}'_x+1, p]$ en H' debe ser al menos n. Un escenario completamente análogo ocurre para las ocurrencias tal que $\text{pos}'_x > p$. Es por esto que, sin pérdida de generalidad, sólo observaremos los sufijos que se encuentran en posiciones previas a p en A'.

Gracias a la observación realizada, para encontrar todas las ocurrencias de S en T a partir de posiciones que se encuentran a la izquierda de p en A', basta con recorrer los índices del suffix array en orden decreciente desde p, mientras que H' en esos lugares no sea menor a n. Cada posición visitada en A' que posea un índice de S' mayor a n representará una ocurrencia de S en T.

Al momento de analizar la cantidad de operaciones realizadas por método propuesto, basta con observar que cada posición de A' se visita a lo sumo una vez. Como $|A'| = n+m+1$, concluímos que la complejidad temporal del algoritmo es $\mathcal{O}(n+m)$.

Algoritmo 10 Ocurrencias de S en T a partir de posiciones menores a p en $\mathcal{O}(n+m)$

```
1: total ← 0
2: for i ← p − 1 down to 0 do
3:     if H'_{i+1} < n then                ▷ lcp_{A_i,0}(S') < n, no hay más ocurrencias
4:         break
5:     end if
6:     if A'_i > n then                    ▷ S ocurre en T a partir de A'_i − n − 1
7:         total ← total + 1
8:     end if
9: end for
10: return total
```

Ejemplo 4.2.1. Sea $S = \texttt{aba}$ y $T = \texttt{ababa}$, tenemos $n = 3$ y $m = 5$. El suffix array y el lcp array de S' son

$$A' = [\texttt{\#ababa}, \texttt{a}, \texttt{a\#ababa}, \texttt{aba}, \overbrace{\texttt{aba\#ababa}}^{p}, \texttt{ababa}, \texttt{ba}, \texttt{ba\#ababa}, \texttt{baba}],$$

$$H' = [0, 0, 1, 1, 3, 3, 0, 2, 2]$$

En rojo se destacan los sufijos que representan las respectivas ocurrencias de S en T, donde se puede observar que $H'_4 = H'_5 = n$.

4.2.2. Un enfoque con búsqueda binaria: $\mathcal{O}(n + \log(m))$

Como primera instancia, es útil notar una propiedad importante que posee el suffix array de un string:

Lema 4.2.2. *Dados strings S y T en Σ^* de largos n y m respectivamente, y sea A el suffix array de T, se cumple que el conjunto de posiciones iniciales de todas las ocurrencias de S en T se encuentran en un único rango continuo de A.*

Demostración. Sean $i, j \in [0, m)$ tal que $\text{pos}_i < \text{pos}_j$ y que S ocurre en T a partir de i y j. Para probar el lema, es suficiente mostrar que para todo $k \in [\text{pos}_i, \text{pos}_j]$, S ocurre en T a partir de A_k.

Sea k tal que $\text{pos}_i \leq k \leq \text{pos}_j$. Como $T_{i,i+n} = T_{j,j+n} = S$, tenemos que $\text{lcp}_{i,j}(T) \geq n$. Entonces, gracias al Corolario 3.1.4, podemos concluir que vale la desigualdad $\text{lcp}_{i,A_k}(T) \geq n$. Es decir, $T_{i\rightarrow}$ y $T_{A_k\rightarrow}$ comparten al menos sus primeros n símbolos, por lo que tenemos que S ocurre en T a partir de A_k. □

Ejemplo 4.2.3. Sea $S =$ `aba` y $T =$ `ababacaba`, el arreglo ordenado de sufijos de T es

$$[\texttt{a}, \texttt{aba}, \texttt{ababacaba}, \texttt{abacaba}, \texttt{acaba}, \texttt{ba}, \texttt{babacaba}, \texttt{bacaba}, \texttt{caba}].$$

Se muestran en color rojo los sufijos que representan el comienzo de una ocurrencia de S en T. Se puede observar que todas las ocurrencias se encuentran agrupadas.

Denotemos como *st* y *en* respectivamente a la primera y la última posición del suffix array de T que representen sufijos donde S ocurre como prefijo. Hemos visto previamente que los sufijos que se encuentren entre medio de esas posiciones en el suffix array también tendrán a S como prefijo. Gracias a esto, la cantidad de ocurrencias de S en T es igual a la cantidad de posiciones en el rango $[st, en]$. Es decir, asumiendo que hay al menos una ocurrencia, la respuesta a nuestro problema es igual a

$$en - st + 1.$$

A continuación, nos enfocaremos en intentar calcular eficientemente los valores de *st* y *en*. En particular, describiremos en detalle un algoritmo que calcule únicamente *st*, pues el que computa el valor restante es completamente análogo. Al momento de realizar el análisis de complejidad temporal del mismo, asumiremos que la construcción del suffix array y el lcp array de T (denotados por A y H respectivamente) fue realizada previamente, por lo que el costo computacional de su ejecución no será tenido en cuenta.

Observación 4.2.4. Dados dos strings S y T, y sea A el suffix array de T. Para todo entero positivo i menor a $|T|$ se cumple (por transitividad de la relación $<$) que

$$T_{A_i\rightarrow} < S \implies T_{A_{i-1}\rightarrow} < S.$$

Lema 4.2.5. *Sean S y T strings de largo n y m respectivamente, y sea A el suffix array de T. Supongamos que S ocurre en T al menos una vez, y denotemos con p a la primera posición de A tal que $S \leq T_{A_p\rightarrow}$. Entonces, vale que $p =$ st.*

Demostración. Sea $L = T_{A_p\rightarrow}$ y $R = T_{A_{st}\rightarrow}$. Gracias a la Observación 4.2.4, tenemos que $p \leq \text{st}$ (es decir, vale que $L \leq R$). Supongamos que $p \neq \text{st}$. Por hipótesis del lema, vale que $S \leq L$. Sabemos que st representa una ocurrencia de S en T, por lo que $\text{lcp}(S, R) \geq n$. Pero entonces, al tener que $S \leq L \leq R$, observamos que $\text{lcp}(S, L) \geq n$, por lo que S ocurre en T a partir de A_p. Esto es una contradicción, pues st es, por definición, la menor posición que cumple esto, y $p < \text{st}$. □

El algoritmo propuesto a continuación computará la primer posición p en el suffix array de T tal que el sufijo ubicado allí sea mayor o igual que S. Gracias al resultado obtenido en el lema anterior podemos garantizar que, en caso de que S ocurra en T, el valor retornado será igual a st.

La Observación 4.2.4 es condición suficiente para realizar una búsqueda binaria en el suffix array de T con el objetivo de calcular p. Al momento de evaluar un candidato durante la ejecución de un paso de la búsqueda, podemos realizar la comparación de strings en $\mathcal{O}(n)$ para determinar el menor entre las dos palabras.

Para analizar el costo operacional del algoritmo basta con notar que, al efectuar la búsqueda binaria, se consideran a lo sumo $\lceil \log(m) \rceil + 1$ candidatos durante la ejecución. Es por este motivo que la complejidad temporal del algoritmo propuesto es de $\mathcal{O}(n \cdot \log(m))$.

Algoritmo 11 Cómputo de st en $\mathcal{O}(n \cdot \log(m))$

```
1:  if S ≤ A_0 then                          ▷ S ≤ T_{i→}, para todo 0 ≤ i < m
2:      return 0
3:  end if
4:  if S > A_{m−1} then                      ▷ T_{i→} < S, para todo 0 ≤ i < m
5:      return m
6:  end if
7:  l ← 0
8:  r ← m − 1
9:  while l + 1 ≤ r do
10:     mid ← ⌊(l+r)/2⌋
11:     if S ≤ T_{A_mid→} then
12:         r ← mid
13:     else
14:         l ← mid
15:     end if
16: end while
17: return r
```

Notar que en el caso en el que no existan ocurrencias de S en T, el algoritmo recién descrito retornará el valor m, o bien el valor de p no será consistente con una ocrrencia de S en T. Además, si las condiciones de las líneas 1 y 4 no se cumplen, se mantiene durante toda la ejecución que

$$T_{l\rightarrow} < S \wedge S \leq T_{r\rightarrow}. \tag{4.1}$$

Dado que las condiciones iniciales se pueden verificar de manera directa, asumiremos siempre el caso en el que no son verdaderas, por lo que tomaremos a (4.1) como invariante.

Ejemplo 4.2.6. Sea $S =$ aba y $T =$ ababacaba, los candidatos del suffix array de T considerados durante la ejecución de cada paso de la búsqueda binaria se muestran a continuación en rojo, mientras que los extremos del rango se escriben en azul:

$$[\mathtt{a}, \mathtt{aba}, \mathtt{ababacaba}, \mathtt{abacaba}, \overbrace{\mathtt{acaba}}^{S \leq T_{A_4}\rightarrow}, \mathtt{ba}, \mathtt{babacaba}, \mathtt{bacaba}, \mathtt{caba}]$$

$$[\mathtt{a}, \mathtt{aba}, \overbrace{\mathtt{ababacaba}}^{S \leq T_{A_2}\rightarrow}, \mathtt{abacaba}, \mathtt{acaba}, \mathtt{ba}, \mathtt{babacaba}, \mathtt{bacaba}, \mathtt{caba}]$$

$$[\mathtt{a}, \overbrace{\mathtt{aba}}^{S \leq T_{A_1}\rightarrow}, \mathtt{ababacaba}, \mathtt{abacaba}, \mathtt{acaba}, \mathtt{ba}, \mathtt{babacaba}, \mathtt{bacaba}, \mathtt{caba}]$$

$$[\mathtt{a}, \overbrace{\mathtt{aba}}^{\mathrm{st}}, \mathtt{ababacaba}, \mathtt{abacaba}, \mathtt{acaba}, \mathtt{ba}, \mathtt{babacaba}, \mathtt{bacaba}, \mathtt{caba}]$$

Como hemos mencionado durante el desarrollo de este trabajo, al momento de determinar el menor entre dos strings S y T utilizando la relación $\leq$ se comparan los símbolos uno a uno, comenzando por el primero e iterando de izquierda a derecha, hasta encontrar la primer posición i donde $S_i \neq T_i$. Mostraremos a continuación que las comparaciones realizadas en la línea 11 del Algoritmo 11 en cada paso de la búsqueda binaria no necesariamente deben comenzarse desde la primer posición de los strings involucrados. En cambio, podemos utilizar información obtenida en pasos anteriores para ahorrar operaciones, y de esta manera mejorar la complejidad del método propuesto.

Sean l y r los valores de los extremos del intervalo considerado durante un paso de la búsqueda binaria en el Algoritmo 11, y sea mid $= \lfloor \frac{l+r}{2} \rfloor$. Además, dado $i \in [0, m)$, utilizaremos la notación

$$v_i = \mathrm{lcp}(S, T_{A_i\rightarrow}).$$

Por definición de la relación $\leq$, para determinar el menor entre los strings S y $T_{A_{\mathrm{mid}}\rightarrow}$, basta con comparar los símbolos ubicados en la posición v_{mid} en cada palabra. Si logramos mantener de forma consistente el valor de v_{mid} durante cada paso de la búsqueda, tendremos una forma efectiva de computar la relación $\leq$ entre ambos strings utilizando una única comparación de símbolos.

Lema 4.2.7. *En cualquier escenario de ejecución de la búsqueda binaria del Algoritmo 11 en el que $l + 1 < r$, dado un valor fijo de mid, existe un único par de valores posibles para l y r.*

Demostración. Supongamos valores fijos de l, mid y r en un momento de la ejecución del ciclo que representa a la búsqueda binaria. Si luego de ejecutar una vez el ciclo obtenemos que la longitud del intervalo de búsqueda es menor a 3,

en el siguiente paso el algoritmo habrá terminado. En el caso contrario valdrá que $l = \text{mid}$ o $r = \text{mid}$, pero como el intervalo resultante considerará al menos 3 posiciones, la posición del nuevo candidato nunca podrá ser la de uno de los extremos. Es decir, el valor actual de mid no se repetirá en los siguientes pasos del algoritmo. □

Para alcanzar el objetivo propuesto, necesitaremos de información que debemos computar antes de ejecutar nuestro algoritmo: el valor de la función lcp entre el sufijo con posición inicial en A_{mid}, y los que comienzan en A_l y A_r. Consideremos todas las posibles tuplas de valores (l, mid, r) que pueden ocurrir durante la búsqueda binaria en el Algoritmo 11. Notemos que el valor de mid siempre se encuentra en el rango $[1, m-2]$, y que, gracias a lo obtenido en el Lema 4.2.7, para cada uno ellos existen únicos l y r que conforman una tupla válida. Es decir, hay exactamente $m-2$ tuplas que son de nuestro interés.

Dado $i \in [1, m-2]$, denotamos por l_i y r_i a los valores que toman l y r cuando $\text{mid} = i$. Procederemos a computar los arreglos Llcp y Rlcp, tales que

$$\text{Llcp}_i = \text{lcp}_{A_{l_i}, A_i}(T),$$

$$\text{Rlcp}_i = \text{lcp}_{A_i, A_{r_i}}(T).$$

Utilizando el resultado obtenido en el Corolario 3.1.4, definiremos una función (denotada por recLcp) que se encargará de computar los arreglos Llcp y Rlcp de manera recursiva. Más específicamente, $\text{recLcp}(l, r)$ calculará los valores correspondientes a las posiciones de ambas listas que se encuentren en el rango $[l, r]$. Además, con el objetivo de simplificar la implementación, la función retornará el valor de $\text{lcp}_{A_l, A_r}(T)$.

Consideremos los valores de l y r en una de las llamadas recursivas de la función recLcp. Existen varios escenarios posibles:

- Si $l = r$, no se ejecutará un nuevo paso de la búsqueda binaria. Además el valor de la función lcp será simplemente la longitud del sufijo $T_{A_l \rightarrow}$.
- Si $l + 1 = r$, tenemos un caso similar al anterior. La única diferencia yace en que el valor a retornar será $\text{lcp}(T_{A_l \rightarrow}, T_{A_r \rightarrow})$ (es decir, H_r).
- Si $l + 1 < r$, aprovecharemos que la función recursiva retorna el valor de la función lcp para el rango representado por sus parámetros. Gracias a esto, podemos concluir que

$$\text{Llcp}_{\text{mid}} = \text{recLcp}(l, \text{mid}),$$

$$\text{Rlcp}_{\text{mid}} = \text{recLcp}(\text{mid}, r).$$

  Además, el valor a retornar por la función será $\text{mín}(\text{Llcp}_{\text{mid}}, \text{Rlcp}_{\text{mid}}, H_{\text{mid}})$.

Para analizar la cantidad de operaciones realizadas al ejecutar la llamada a la función $\text{recLcp}(0, m-1)$, observemos que solo se realizan llamadas recursivas cuando la tupla (l, mid, r) es una de las $m-2$ que pueden ser utilizadas en la

búsqueda binaria. Además, la cantidad de casos en los que $l = r$ y $l + 1 = r$ son a lo sumo m y $m - 1$ respectivamente, pues $0 \leq l, r < m$. Concluimos entonces que la complejidad temporal resultante es $\mathcal{O}(m)$.

Algoritmo 12 Función recLcp para el cómputo de Llcp y Rlcp en $\mathcal{O}(m)$

```
1: if l = r then
2:     return m − A_l
3: end if
4: if l + 1 = r then
5:     return H_r
6: end if
7: mid ← ⌊(l+r)/2⌋
8: Llcp_mid ← recLcp(l, mid)
9: Rlcp_mid ← recLcp(mid, r)
10: return mín(Llcp_mid, Rlcp_mid, H_mid)
```

Observación 4.2.8. Dados strings S y T, sea A el suffix array de T. La secuencia de valores $\text{lcp}(S, T_{A_i\rightarrow})$ se compone de un prefijo no decreciente, seguido por un sufijo no creciente. Más formalmente, existe $j \in [0, m)$ tal que

$$\forall i : 0 < i \leq j : \text{lcp}(S, T_{A_i\rightarrow}) \geq \text{lcp}(S, T_{A_{i-1}\rightarrow})$$

$$\forall i : j < i < m : \text{lcp}(S, T_{A_i\rightarrow}) \leq \text{lcp}(S, T_{A_{i-1}\rightarrow})$$

Ejemplo 4.2.9. Sea $S = \texttt{abab}$ y $T = \texttt{aababaa}$, tenemos que

$$A = [\texttt{a}, \texttt{aa}, \texttt{aababaa}, \texttt{abaa}, \texttt{ababaa}, \texttt{baa}, \texttt{babaa}].$$

Sea L la lista de tamaño $|T|$ tal que $L_i = v_i$, vale que

$$L = [1, 1, 1, 3, 4, 0, 0]$$

En donde observamos en rojo el prefijo no decreciente, y en azul el sufijo no creciente de la misma.

Dados los extremos l y r de un paso de la búsqueda binaria del Algoritmo 11, consideremos los valores respectivos de v_l y v_r. Sin pérdida de generalidad, asumiremos que $v_l \geq v_r$. Si deseamos calcular v_{mid}, tenemos distintos escenarios:

- Si $\text{Llcp}_{\text{mid}} > v_l$, tenemos que $T_{A_{\text{mid}}\rightarrow} =_{v_l+1} T_{A_l\rightarrow} \neq_{v_l+1} S$. En particular, gracias a la primera desigualdad de la invariante obtenida en (4.1), podemos concluir que st $>$ mid, y a su vez $v_{\text{mid}} = v_l$.
- Si $\text{Llcp}_{\text{mid}} = v_l$, los primeros v_l símbolos entre $T_{A_{\text{mid}}\rightarrow}$ y S son iguales. Es por esto que podemos comenzar a comparar una a una las siguientes posiciones, comenzando desde $v_l + 1$, hasta encontrar la primera en la que difieran. El símbolo que se encuentre allí determinará si st se encuentra en mitad

izquierda o derecha del intervalo de la búsqueda binaria. Además, asumiendo que se compararon x nuevos símbolos, el valor de v_l (o v_r si el símbolo distinto era mayor al de la posición correspondiente en S) incrementará su valor en x.

- Si $\text{Llcp}_{\text{mid}} < v_l$, S coincide en menos símbolos con $T_{A_{\text{mid}}\rightarrow}$ que con $T_{A_l\rightarrow}$. Entonces, gracias a la Observación 4.2.8, tenemos que st $<$ mid. Además, el nuevo valor de v_r será Llcp_{mid}.

Cabe destacar que el caso en el que $v_l < v_r$ es completamente análogo al recién descrito, utilizando los valores de Rlcp en vez de los de Llcp.

Sea mx = máx(v_l, v_r). Es claro que en el primer y tercer escenario se realizan $\mathcal{O}(1)$ operaciones, pues todos los valores que se requieren ya fueron previamente computados. Además, el valor de mx no se ve afectado. En el segundo caso, mx incrementa su valor por la cantidad de nuevas comparaciones que se realizan hasta encontrar el siguiente símbolo que difiere entre $T_{A_{\text{mid}}\rightarrow}$ y S. Dado que mx nunca decrece, y además se encuentra acotado por n, podemos concluir que la cantidad de veces que el valor de mx aumenta durante todos los pasos de la búsqueda binaria es a lo sumo n.

Como observamos que la cantidad de operaciones realizadas en cualquiera de los tres escenarios es a lo sumo n en total, obtenemos finalmente que la complejidad temporal del algoritmo resulta ser $\mathcal{O}(n + \log(m))$.

n	m	$\mathcal{O}(n \cdot \log(m))$	$\mathcal{O}(n + \log(m))$
$5 \cdot 10^5$	$5 \cdot 10^6$	15.02	0.96
10^6	10^7	16.27	1.48
$5 \cdot 10^6$	$5 \cdot 10^7$	91.08	6.95
10^7	10^8	181.33	13.93

Cuadro 4.1: Tiempo de ejecución en milisegundos de los métodos de búsqueda binaria propuestos, donde $S = \mathtt{a}^n$ y $T = \mathtt{a}^m$.

Cabe destacar que el cómputo de las listas Llcp y Rlcp demandan $\mathcal{O}(m)$ operaciones computacionales, por lo que la complejidad final considerando esto termina siendo muy similar a la del primer algoritmo propuesto en esta sección. Sin embargo, el poder de este método yace en que, una vez calculadas ambas listas, se puede utilizar las mismas para realizar múltiples consultas utilizando distintos patrones, permitiendo de esta manera lograr una ventaja considerable en comparación a las versiones anteriores.

Algoritmo 13 Cómputo de st en $\mathcal{O}(n + \log(m))$

$v_l \leftarrow \text{lcp}(T_{A_0}, S)$
$v_r \leftarrow \text{lcp}(T_{A_{m-1}}, S)$
if $v_l = n$ **or** $S_{v_l} \leq T_{A_0+v_l}$ **then** ▷ $S \leq T_{i\rightarrow}$, para todo $0 \leq i < m$
    return 0
end if
if $v_r < n$ **and** $S_{v_r} > T_{A_{m-1}+v_r}$ **then** ▷ $T_{i\rightarrow} < S$, para todo $0 \leq i < m$
    return m
end if
$l \leftarrow 0$
$r \leftarrow m - 1$
while $l + 1 < r$ **do**
    $\text{mid} \leftarrow \lfloor \frac{l+r}{2} \rfloor$
    if $v_l \geq v_r$ **then**
        if $\text{Llcp}_{\text{mid}} \geq v_l$ **then** ▷ Encontrar primer símbolo que difiere
            $v_{\text{mid}} \leftarrow v_l + \text{lcp}(T_{A_{\text{mid}+v_l}\rightarrow}, S_{v_l\rightarrow})$
        else
            $v_{\text{mid}} \leftarrow \text{Llcp}_{\text{mid}}$
        end if
    else
        if $\text{Rlcp}_{\text{mid}} \geq v_r$ **then**
            $v_{\text{mid}} \leftarrow v_r + \text{lcp}(T_{A_{\text{mid}+v_r}\rightarrow}, S_{v_r\rightarrow})$
        else
            $v_{\text{mid}} \leftarrow \text{Rlcp}_{\text{mid}}$
        end if
    end if
    if $v_{\text{mid}} = n$ **or** $S_{v_{\text{mid}}} \leq T_{A_{\text{mid}}+v_{\text{mid}}}$ **then**
        $r \leftarrow \text{mid}$
        $v_r \leftarrow v_{\text{mid}}$
    else
        $l \leftarrow \text{mid}$
        $v_l \leftarrow v_{\text{mid}}$
    end if
end while
return r

4.3. Rotación lexicográficamente mínima de un string

Como hemos mencionado previamente en nuestro trabajo, distintas áreas de estudio utilizan el concepto de rotación lexicográficamente mínima de una palabra. Muchas veces, la longitud de los strings involucrados suele ser de un tamaño considerablemente grande. Esto motiva a intentar obtener un método que posea una complejidad mejor que la propuesta realizada en el Algoritmo 2.

Sea un string S de longitud n contenido en Σ^*, a continuación describiremos un método para computar su rotación lexicográficamente mínima, el cual tendrá una complejidad temporal de $\mathcal{O}(n)$.

Denotemos con S' al string en Σ^* que cumple $S' = S + S$. Sea A el suffix array correspondiente a S', y pos $= \text{inv}(A)$. Nos será útil notar que toda posible rotación cíclica de S se encuentra representada como substring de S'. En particular, para todo i que cumple $0 \leq i < n$, tenemos que

$$\text{shift}^i(S) = (S'_{n-i\rightarrow})_{0,n}.$$

Es decir, si consideramos los primeros n símbolos de cada sufijo de S' con posiciones iniciales en el rango $[1, n]$, podemos obtener cualquier rotación de S.

Observación 4.3.1. Sean i, j enteros no negativos. Se cumple que

$$S'_{i\rightarrow} \leq S'_{j\rightarrow} \implies S'_{i\rightarrow} \leq_n S'_{j\rightarrow}$$

Gracias a la observación previa podemos notar que, al computar el suffix array A de S', además estamos calculando el orden relativo entre las rotaciones cíclicas de S. Es decir, si tomamos $i, j \in [1, n]$ tal que $\text{pos}_i \leq \text{pos}_j$, tenemos entonces que $\text{shift}^{n-i}(S) \leq \text{shift}^{n-j}(S)$.

El algoritmo propuesto resulta bastante simple e intuitivo luego de haber mostrado las propiedades de posee el arreglo A: para encontrar la rotación lexicográficamente mínima de S, solo debemos encontrar el mínimo índice de A que posea un valor entre 1 y n. Para ello podemos iterar las posiciones del suffix array de izquierda a derecha hasta encontrar una que cumpla la condición deseada. Dado que $|A| = 2n$, es fácil notar que la complejidad temporal y espacial del método será $\mathcal{O}(n)$.

Ejemplo 4.3.2. Sea $S =$ `acaab`. La rotación lexicográficamente mínima de S está representada por

$$\text{shift}^3(S) = \texttt{aabac}.$$

Tenemos que $S' = S + S =$ `acaabacaab`, y el suffix array A de S' es

[`aab`, `aabac`aab, `ab`, `abaca`ab, `acaab`, `acaabacaab`, `b`, `bacaa`b, `caab`, `caaba`caab].

En rojo se destacan los primeros n símbolos de los sufijos que representan todas las rotaciones cíclicas de S. En particular se puede observar que en el primero de ellos, es decir A_1, el prefijo de tamaño n coincide con $\text{shift}^3(S)$.

4.4. Largo máximo de un substring en común entre dos strings

Como hemos visto anteriormente, el concepto de lcp array hace que el potencial del suffix array de una palabra sea mucho mayor, permitiendo el desarrollo de soluciones para variados problemas que tienen gran impacto en la vida real. Uno de ellos es el de encontrar la palabra de mayor longitud que ocurre como substring de dos strings. Describiremos en detalle como resolver este problema mediante el uso de las estructuras que han sido introducidas en este trabajo.

4.4.1. Primer propuesta: $\mathcal{O}((n+m)\cdot\log(n+m))$

Dados strings S_1, S_2 en Σ^* de longitudes n y m respectivamente, nuestro objetivo será el de encontrar la mayor longitud de una palabra P que ocurra en S_1 y S_2 al mismo tiempo.

Sea S el string en $(\Sigma \sqcup \{\#\})^*$ tal que $S = S_1 + \# + S_2$, donde $\# <' c$ para todo $c \in \Sigma$, y denotemos al suffix array y al lcp array de S como A y H respectivamente. Notemos que las posiciones de S que se encuentren en el rango $[0, n)$ representarán sufijos de S_1, mientras que las que pertenezcan al intervalo $[n+1, n+m+1)$ corresponderán a sufijos de S_2. Definimos la lista rep de tamaño $n+m+1$ tal que

$$\text{rep}_i = \begin{cases} 1 & \text{si } 0 \le A_i < n \\ 2 & \text{si } n < A_i \le n+m \\ -1 & \text{caso contrario} \end{cases}$$

El problema a resolver es equivalente al de seleccionar dos posiciones i_1, i_2 del suffix array A que cumplan $\text{rep}_{i_1} = 1$ y $\text{rep}_{i_2} = 2$, tal que maximicen el valor de $\text{lcp}(S_{i_1\to}, S_{i_2\to})$. Asumiremos, sin pérdida de generalidad, que $i_1 < i_2$.

Observación 4.4.1. Dado un valor fijo de i_2, si deseamos maximizar el valor de la función lcp es óptimo considerar como i_1 a la mayor posición a la izquierda de i_2 tal que $\text{rep}_{i_1} = 1$, pues el valor de $\text{lcp}(S_{i_1\to}, S_{i_2\to})$ es no creciente a medida que decrementa el valor de i_1.

Podemos utilizar la propiedad recién observada para obtener un método simple que nos permita computar el mejor candidato i_1 para un valor determinado de i_2. En particular, procederemos de la misma manera para obtener i_2 dado un valor fijo de i_1, en caso de que se cumpla $i_2 < i_1$.

El algoritmo propuesto consiste en iterar las posiciones de A de izquierda a derecha, manteniendo dos variables last_1 y last_2 que representan la última posición observada previamente donde se cumple que $\text{rep}_{\text{last}_1} = 1$ y $\text{rep}_{\text{last}_2} = 2$ respectivamente.

Sea j la posición que está siendo iterada en un momento determinado de la ejecución. Supongamos, sin pérdida de generalidad, que $\text{rep}_j = 2$. Entonces, utilizando el Corolario 3.1.4, tenemos que el máximo valor de la función lcp cuando $i_2 = j$ (al que denotaremos como best_j) cumplirá que

$$\text{best}_j = \min_{k=\text{last}_1+1}^{j} H_k.$$

En el caso en que no existan candidatos para i_1 que se encuentren a la izquierda de j en A, asumiremos que $\text{last}_1 = -1$. Esto causará que $\text{best}_j = 0$, pues por definición de H, tenemos que $H_0 = 0$.

Si logramos computar eficientemente los distintos valores de best_j para cada valor que toma j durante la ejecución del algoritmo, tendremos una solución al problema.

Necesitamos de una estructura que sea capaz de computar el mínimo valor en distintos rangos de la lista H. En particular, podemos utilizar un segment tree

(A.2) que represente a H, y de esta manera obtener los valores deseados mediante la función query del mismo.

Durante la ejecución del método, realizamos una única consulta, de complejidad $\mathcal{O}(\log(n+m))$, para cada valor posible de j. Es por esto que concluimos que la complejidad temporal final del método es igual a $\mathcal{O}((n+m)\cdot\log(n+m))$, mientras que se utiliza $\mathcal{O}(n+m)$ memoria extra para construir el segment tree sobre H.

Ejemplo 4.4.2. Sea $S_1 = \texttt{caba}$ y $S_2 = \texttt{acab}$. En el suffix array de S, se muestran en color rojo y azul las palabras que representan sufijos de S_1 y S_2 respectivamente. Además, con el objetivo de simplificar la explicación, no mostraremos los símbolos que siguen al separador # en los sufijos de S que se correspondan con los de S_1.

En el caso en que $p = 4$, tenemos:

$$A = [\texttt{a}, \texttt{ab}, \overbrace{\underbrace{\texttt{aba}}_{\text{last}_1}, \texttt{acab}, \underbrace{\texttt{b}}_{p}}^{\text{lcp(aba,b)=1}}, \texttt{ba}, \texttt{cab}, \texttt{caba}].$$

Además, el valor óptimo se alcanza cuando $p = 7$:

$$A = [\texttt{a}, \texttt{ab}, \texttt{aba}, \texttt{acab}, \texttt{b}, \texttt{ba}, \overbrace{\underbrace{\texttt{cab}}_{\text{last}_2}, \underbrace{\texttt{caba}}_{p}}^{\text{lcp(cab,caba)=3}}],$$

pues el susbtring más largo en común entre S_1 y S_2 es cab.

Algoritmo 14 String más largo en común entre S_1 y S_2 en $\mathcal{O}((n+m)\cdot\log(n+m))$

```
1: for i ← 0 to |H| do
2:     set(i, H_i)
3: end for
4: last_1 ← −1
5: last_2 ← −1
6: maxLength ← 0
7: for j ← 0 to |H| do
8:     if rep_j = 1 then
9:         maxLength ← máx(maxLength, query(last_2 +1, j))
10:        last_1 ← j
11:    end if
12:    if rep_j = 2 then
13:        maxLength ← máx(maxLength, query(last_1 +1, j))
14:        last_2 ← j
15:    end if
16: end for
17: return maxLength
```

4.4.2. Adaptando la solución para k strings

Dados $k \geq 2$ strings $S_1, S_2, \ldots, S_k$ de longitudes $n_1, n_2, \ldots, n_k$ respectivamente, intentaremos a continuación obtener un algoritmo que compute eficientemente el máximo largo de un string que ocurra en todos ellos al mismo tiempo.

En el caso de que $k = 2$, el método propuesto anteriormente realiza el trabajo que buscamos. Es razonable intentar adaptar el mismo para obtener un algoritmo genérico que funcione para cualquier valor que pueda tomar k.

Sean $\#_1, \#_2, \ldots, \#_{k-1}$ símbolos distintos que no pertenecen a Σ, y denotaremos con S a la palabra en $(\Sigma \sqcup \{\#_i \mid 0 \leq i < k\})^*$ tal que

$$S = S_1 + \#_1 + S_2 + \#_2 + \cdots + \#_{k-1} + S_k$$

Asumiremos que $\#_i <' c$, para todo $c \in \Sigma$, y que además $\#_i <' \#_{i+1}$ para todo $i \in [0, k-1)$.

Sea $\ell = |S|$. Denotemos con A y H al suffix array y al lcp array de S. De manera análoga al problema anterior, definiremos la lista rep de tamaño $|S|$ que cumpla

$$\text{rep}_i = \begin{cases} 1 & \text{si } 0 \leq A_i < n_1 \\ 2 & \text{si } n_1 < A_i \leq n_1 + n_2 \\ \vdots & \\ k & \text{si } \sum_{j=0}^{k-1} n_i < A_i \leq \sum_{j=0}^{k} n_i \\ -1 & \text{caso contrario} \end{cases}$$

Sean $i_1, i_2, \ldots i_k$ índices arbitrarios en A tal que $\text{rep}_{i_j} = j$ para todo j. Utilizaremos las notaciones $\text{st} = \text{mín}(i_1, i_2, \ldots, i_k)$ y $\text{en} = \text{máx}(i_1, i_2, \ldots, i_k)$. Es intuitivo notar que el máximo prefijo en común entre los k sufijos se encuentra representado por $\text{lcp}_{A_{st}, A_{en}}(S)$. Es decir, nos interesa computar el mínimo del arreglo H en el rango $[st + 1, en]$.

Sea p un valor fijo de en. Similarmente a lo descrito en la Observación 4.4.1, si deseamos obtener el mayor valor posible de la función lcp, es óptimo maximizar el valor de st. Para lograr esto, mantendremos una lista de k valores, denotada por last, tal que last_i es igual a la máxima posición de A menor o igual a p que cumple $\text{rep}_{\text{last}_i} = i$. Es fácil observar que $\text{st} = \text{mín}(\text{last}_1, \text{last}_2, \ldots, \text{last}_k)$.

El método propuesto es una adaptación del que ha sido introducido en el Algoritmo 14. Mientras iteramos el valor de en de izquierda a derecha en A, mantendremos un puntero que represente el valor de st. Para que el mismo sea consistente durante toda la ejecución del programa, nos ayudaremos de la información provista por la lista last para poder saber cuando debemos incrementar el valor de st. Luego de recalcular la posición que representa a st, y de manera análoga a la solución anterior, el valor del máximo substring en común si $\text{en} = p$ lo obtendremos mediante una consulta de mínimo en rango en el segment tree que representa al arreglo H.

Cabe notar que si el valor de p es demasiado chico, puede ocurrir que no existan k posiciones menores o iguales a p representando a cada uno de los k strings involucrados. Si nos encontramos en esta situación, omitiremos el valor

actual de p, pues no es un candidato válido para tomar el valor de en. En términos de implementación, asumiremos que inicialmente $\text{last}_i = -1$ para todo i. Durante la ejecución del algoritmo mantendremos un contador de cuántas posiciones en last son distintas de -1 (denotado por *total*). Si el mismo alcanza el valor de k, sabremos que nos encontramos con un candidato válido para en.

Algoritmo 15 String más largo en común entre k strings en $\mathcal{O}(\ell \cdot \log(\ell))$

```
1:  for i ← 0 to |H| do
2:      set(i, H_i)
3:  end for
4:  last ← list(k, −1)          ▷ Lista de tamaño k con valor −1 en cada posición
5:  total ← 0
6:  maxLength ← 0
7:  st ← 0
8:  for en ← 0 to |H| do
9:      if last_rep_en = −1 then
10:         total ← total + 1
11:     end if
12:     last_rep_en ← en
13:     while last_rep_st ≠ st do          ▷ Actualización del valor de st
14:         st ← st + 1
15:     end while
16:     if total = k then
17:         maxLength ← máx(maxLength, query(st +1, en))
18:     end if
19: end for
20: return maxLength
```

Notemos que cada vez que la condición del ciclo en la línea 13 es verdadera, la variable st incrementa su valor en uno. Además, st y en se encuentran acotadas por ℓ, por lo que el ciclo se ejecutará a lo sumo ℓ veces en total. Esto nos permite concluir que la complejidad temporal amortizada que posee el algoritmo propuesto es $\mathcal{O}(\ell \cdot \log(\ell))$, utilizando $\mathcal{O}(\ell)$ memoria extra para mantener el segment tree de H y la lista last.

4.4.3. Optimizando la complejidad temporal

El método propuesto previamente para la resolución del problema involucra el uso de un segment tree sobre el arreglo H. La gran ventaja que proporciona esta estructura en comparación a otras es su capacidad de soportar cambios en la lista de valores. En nuestro caso, H es un arreglo estático. Es decir, no se requieren modificaciones en el mismo durante la ejecución del algoritmo. Esto nos lleva a pensar en la posibilidad de usar otras estructuras más eficientes a la hora de responder el mínimo en un rango, aprovechando las propiedades y condiciones de nuestro problema particular.

Durante toda la ejecución del Algoritmo 15, se mantiene como invariante que st $\leq$ en. Además, ambos punteros nunca decrecen en sus valores. Estas propiedades nos permiten notar que el intervalo [st, en] simula el comportamiento de la estructura de datos conocida como cola: st representa el valor en la posición inicial de ella, mientras que en apunta a su elemento final. Además, el incrementar el valor de st y en imita el comportamiento de la cola al realizar las operaciones pop y push respectivamente.

El objetivo del método es poder computar el mínimo del rango representado por los distintos valores que toman st y en. Gracias a lo observado previamente, podemos utilizar una cola monotónica (ver el Apéndice A.1) para mantener consistentemente los valores de H que se encuentren en el rango [st, en].

Al momento de incrementar el valor de en, insertamos $H_{\text{en}+1}$ en la cola. De manera análoga, cuando se requiere aumentar st, removemos $H_{\text{st}+1}$ de la misma. Es fácil observar que, de esta manera, solo mantenemos en la cola monotónica los valores de H que son necesarios para computar el valor de $\text{lcp}_{A_{\text{st}}, A_{\text{en}}}(S)$.

Como la cantidad de veces que realizamos la operación *push* en la cola es exactamente ℓ, obtenemos finalmente un algoritmo con complejidad temporal y espacial de $\mathcal{O}(\ell)$.

Algoritmo 16 String más largo en común entre k strings en $\mathcal{O}(\ell)$

```
1: last ← list(k, −1)          ▷ Lista de tamaño k con valor −1 en cada posición
2: total ← 0
3: maxLength ← 0
4: st ← 0
5: lcpQueue ← monotonicQueue()
6: for en ← 0 to |H| do
7:     if last_rep_en = −1 then
8:         total ← total + 1
9:     end if
10:    last_rep_en ← en
11:    while last_rep_st ≠ st do          ▷ Quitar de la cola al actualizar st
12:        st ← st + 1
13:        LcpQueue.pop()
14:    end while
15:    if total = k then
16:        maxLength ← máx(maxLength, LcpQueue.min())
17:    end if
18:    if en ≠ |H| then
19:        LcpQueue.push(H_{en+1})          ▷ Agregar a la cola al actualizar en
20:    end if
21: end for
22: return maxLength
```

Apéndice A

Consultas de mínimo en rango

Durante el desarrollo de este trabajo, en reiteradas ocasiones nos encontramos con la necesidad de contar con un método para computar de manera eficiente el mínimo valor en un rango continuo de posiciones de una lista de números enteros. A continuación describiremos, de manera breve y concisa, dos estructuras de datos que son utilizadas por los distintos algoritmos aquí propuestos.

A.1. Cola monotónica

El objetivo de esta sección será el de introducir una estructura de datos que implementará la interfaz de una cola, con la propiedad adicional de almacenar información relacionada al mínimo valor que se encuentra en ella en un momento determinado de tiempo. Esta estructura es conocida como *cola monotónica*, y provee la capacidad de realizar las siguientes operaciones sobre una lista de números:

- push(x): agregar un elemento al final de la lista con valor x.
- pop(): en caso de que la cola no se encuentre vacía, remover el elemento ubicado en su primer posición.
- min(): consultar por el mínimo valor ubicado en alguna posición de la lista. Si la cola no posee elementos, se retorna el valor ∞.

Para proveer una implementación de las operaciones previamente descritas, contaremos con una estructura auxiliar que nos será de mucha utilidad: la cola doblemente terminada (también conocida como *deque*). El poder de la deque yace en que provee la posibilidad de agregar y borrar elementos de cualquier extremo de la cola, gracias a las operaciones pushBack, pushFront, popBack y popFront. Además, mediante el uso de front y back se puede consultar el valor ubicado en la primera y en la última posición de la cola respectivamente. Todas las operaciones mencionadas poseen una complejidad temporal de $\mathcal{O}(1)$.

En este trabajo, implementaremos la cola monotónica como una deque de pares de enteros. El par ubicado en la primer posición de la misma contendrá

en su primera coordenada el mínimo actual en la cola, mientras que la segunda representa cuántas operaciones pop son necesarias para que el valor del menor elemento de ella aumente. Además, dado dos pares consecutivos en la deque, se cumplirá siempre que el primero de ellos tiene un valor menor que el segundo en su primer coordenada.

Al momento de realizar la operación push(x), mantendremos un contador para representar la cantidad de veces que x será el mínimo luego de que todos los valores menores a él hayan sido quitados de la cola. Inicialmente el mismo será igual a 1. Además, cuando haya un par (x_1, y_1) en la cola tal que $x \leq x_1$, notemos que x reemplazará a x_1 como menor valor en las y_1 ocasiones en las que x_1 iba a ser el mínimo de la lista. Esto es porque, para borrar el nuevo elemento que está siendo insertado, primero se deben quitar todas las ocurrencias de x_1 que se encontraban en la cola previamente. Gracias a la condición de que los pares de la deque se encuentran ordenados con respecto a su primera coordenada, podemos borrar reiteradas veces el elemento final de la misma mientras que se cumpla que x es menor o igual al primer valor de ese par.

La operación pop consiste en decrementar el valor de la segunda coordenada para el primer elemento (x_1, y_1) de la deque. En caso que el mismo llegue a cero, significa que el valor del mínimo de la cola ha aumentado, pues todas las ocurrencias de x_1 en ella han sido quitadas. En ese escenario, deberemos borrar de la cola al primer par de la misma, mediante la operación popFront.

Gracias a que la deque cumple que la primer posición representa al mínimo actual en la cola monotónica, realizar la operación min se reduce simplemente a retornar su primer coordenada.

Ejemplo A.1.1. Dada una cola monotónica inicialmente vacía, se muestra a continuación el estado de la deque luego de cada operación listada:

push(2)	$\longrightarrow$	[(2,1)]
push(3)	$\longrightarrow$	[(2,1), (3,1)]
pop()	$\longrightarrow$	[(3,1)]
push(1)	$\longrightarrow$	[(1,2)]
push(3)	$\longrightarrow$	[(1,2), (3,1)]
push(1)	$\longrightarrow$	[(1,4)]
pop()	$\longrightarrow$	[(1,3)]

Notar que en cada momento, la primer coordenada del elemento inicial de la deque coincide con el mínimo en la cola monotónica.

La implementación de cada una de las operaciones soportadas por la cola monotónica consiste únicamente en insertar y borrar elementos de la deque. Denotemos como k a la cantidad de veces que se ejecutó push. Como cada una de ellas agrega exactamente un elemento a la deque, la cantidad de veces que se borra un elemento de la misma también se encuentra acotado por k. Es por esto que la complejidad temporal y espacial final resulta en $\mathcal{O}(k)$.

Algoritmo 17 Implementación de la interfaz de una cola monotónica

```
1:  procedure push(x)
2:      total ← 1
3:      while |deque| > 0 and deque.back()_0 ≥ x do
4:          total ← total + deque.back()_1
5:          deque.popBack()
6:      end while
7:      deque.pushBack((x, total))
8:  end procedure

9:  procedure POP()
10:     if |deque| > 0 then
11:         deque.front()_1 ← deque.front()_1 − 1
12:         if deque.front()_1 = 0 then
13:             deque.popFront()
14:         end if
15:     end if
16: end procedure

17: procedure MIN()
18:     if |deque| = 0 then
19:         return ∞
20:     else
21:         return deque.front()_0
22:     end if
23: end procedure
```

A.2. Segment tree

En muchos escenarios, las consultas de mínimo en rango se deben realizar sobre una lista que recibe modificaciones en los valores de sus distintas posiciones. A continuación, describiremos una estructura de datos que permite resolver este problema de manera eficiente.

Definición A.2.1. Un árbol binario con raíz se denomina *estricto* si cumple la propiedad de que cada uno de sus nodos posee exactamente cero o dos hijos. A su vez, los nodos que no tienen hijos se llaman *hojas*.

Observación A.2.2. Un árbol binario estricto con k hojas, posee exactamente $2k - 1$ nodos en total. Este hecho se desprende de que, si tomamos dos hojas que poseen el mismo padre y las eliminamos del árbol, k decrece su valor en exactamente 1. Este proceso puede ser repetido mientras que $k > 1$, por lo que la cantidad de nodos que inicialmente no eran hojas es exactamente $k - 1$.

Sea A una lista de n números. Dados enteros l y r tales que $0 \leq l < r \leq n$, definiremos la función recursiva $\text{init}(l, r)$, la cual generará un árbol binario estricto que almacenará en su raíz al valor del mínimo del rango $[l, r)$ en A:

- Si $l + 1 = r$, el rango representa a un solo elemento de la lista A, por lo que el árbol se compone de un único nodo hoja, el cual contendrá el valor A_l.
- Si $l + 1 < r$, sea $m = \lfloor \frac{l+r}{2} \rfloor$. La raíz del árbol poseerá como hijos izquierdo y derecho a las raíces de los árboles generados por las llamadas recursivas a $\text{init}(l, m)$ y a $\text{init}(m, r)$ respectivamente. Podemos observar que en este caso, la posición del mínimo del rango $[l, r)$ se encuentra contenida en el rango representado por uno de los hijos de la raíz. Es por esto que el valor que contendrá la raíz será simplemente el mínimo de los valores almacenados en sus hijos.

El *segment tree* que representa a A es el árbol binario estricto generado por la llamada a la función $\text{init}(0, n)$. Cabe notar que la cantidad de hojas en el mismo es exactamente n. Gracias a la Observación A.2.2, podemos concluir entonces que la cantidad de nodos en total es $2n - 1$. Como en cada ejecución de una llamada a la función init se crea exactamente un nodo nuevo, la complejidad temporal y espacial de computar el segment tree de A es $\mathcal{O}(n)$.

Algoritmo 18 Definición recursiva de $\text{init}(l, r)$

```
1:  root ← node()
2:  if l + 1 = r then                          ▷ Nodo hoja con valor A_l
3:      root.left ← null
4:      root.right ← null
5:      root.value ← A_l
6:  else                          ▷ Construcción recursiva de los hijos de la raíz
7:      m ← ⌊(l+r)/2⌋
8:      root.left ← init(l, m)
9:      root.right ← init(m, r)
10:     root.value ← mín(root.left.value, root.right.value)
11: end if
12: return root
```

Observación A.2.3. La profundidad del segment tree de A es $\mathcal{O}(\log(n))$. Esto es gracias a que la raíz del árbol representa al rango $[0, n)$, y que cada vez que descendemos desde un nodo hacia uno de sus hijos, la longitud del intervalo representado por el nodo hijo es a lo sumo la mitad que la del padre.

A.2.1. Consultas

Dado que los nodos del segment tree sólo almacenan el mínimo en rangos específicos de A, deberemos proponer un método que combine la información alojada en ellos para poder computar el menor elemento en cualquier rango.

Al momento de recibir una consulta acerca de un rango $[l, r)$, recorreremos recursivamente los nodos del árbol, comenzando desde la raíz. Supongamos que el nodo actual posee la información acerca del mínimo en el rango $[a, b)$, existen 3 posibles casos:

- Si $r \leq a$ o $b \leq l$, ningún elemento considerado por el nodo se encuentra en el rango de interés, por lo que se debe devolver el valor ∞.
- Si $l \leq a < b \leq r$, el rango del nodo se encuentra completamente contenido en el rango de la consulta. Es decir, todos los elementos considerados por el nodo actual deben ser tomados en cuenta para computar el mínimo en el rango de la consulta. Por este motivo, simplemente podemos retornar el valor alojado en el mismo sin realizar ninguna llamada recursiva.
- Si no ocurre ninguno de los casos anteriores, los rangos del nodo y de la consulta se intersectan parcialmente. En este caso, no tenemos otra opción que recursionar en ambos hijos del nodo actual, y computar el mínimo de los valores obtenidos en ambas llamadas.

En términos de implementación, definiremos una función recursiva denotada por query(root, a, b, l, r), la cual computará el mínimo en el rango $[l, r)$, considerando que la raíz del árbol es el nodo root, el cual posee información acerca del rango $[a, b)$ de la lista. Si denotamos como p a la raíz del segment tree de A, la respuesta a la consulta será obtenida mediante la llamada a la función query$(p, 0, n, l, r)$.

Algoritmo 19 Definición recursiva de query(root, a, b, l, r)

```
1: if r ≤ a or b ≤ l then                    ▷ Rango disjunto con el de la consulta
2:     return ∞
3: end if
4: if l ≤ a and b ≤ r then                   ▷ Rango contenido en la consulta
5:     return root.value
6: end if
7: m ← ⌊(a+b)/2⌋                             ▷ Recursión a ambos hijos de root
8: val_l ← query(root.left, a, m, l, r)
9: val_r ← query(root.right, m, b, l, r)
10: return mín(val_l, val_r)
```

Lema A.2.4. *Sea p la raíz del segment tree que representa una lista A de longitud n. La cantidad de nodos visitados en cada nivel de profundidad del arbol al realizar la llamada a la función* query$(p, 0, n, l, r)$ *es a lo sumo* 4.

Demostración. Probaremos esta propiedad por inducción en el nivel de profundidad del árbol. En el nivel inicial solo hay un único nodo, por lo que en este caso el lema se cumple trivialmente.

Supongamos a continuación que en un nivel determinado se visitaron $k \leq 4$ nodos. Cada nodo que realiza una llamada recursiva, visitará a sus dos hijos en el siguiente nivel, por lo que si $k \leq 2$, la propiedad será verdadera. Supongamos entonces que $k > 2$. Como un nivel del árbol representa una partición en rangos de la lista A, si observamos los nodos del mismo de izquierda a derecha, los k nodos visitados se encontrarán en posiciones consecutivas. En particular, los rangos representados por los $k - 2$ nodos que no sean el extremo izquierdo y derecho se encontrarán completamente contenidos en el rango de la consulta, por lo que estos nodos no realizarán llamadas recursivas. Concluimos entonces que a lo sumo dos de ellos visitarán a sus hijos, por lo que la cantidad total de nodos visitados en el próximo nivel no puede superar 4. □

Corolario A.2.5. *La complejidad computacional total de calcular el valor de* $\text{query}(p, 0, n, l, r)$ *es* $\mathcal{O}(\log(n))$.

Demostración. La cantidad de operaciones realizadas al visitar un nodo del árbol es $\mathcal{O}(1)$, por lo que la complejidad de la llamada inicial es equivalente a la cantidad de nodos visitados por la misma. Por la Observación A.2.3, la profundidad del segment tree de una lista de longitud n es $\mathcal{O}(\log(n))$, y gracias al Lema A.2.4, podemos concluir entonces que la cantidad de nodos visitados en total será igual a

$$\mathcal{O}(4 \cdot \log(n)) = \mathcal{O}(\log(n)).$$

□

A.2.2. Modificaciones en la lista

Supongamos a continuación que deseamos cambiar el valor de la posición i de A por x. Es decir, queremos realizar la asignación $A_i = x$, y reconstruir el segment tree de A de modo que refleje correctamente los cambios necesarios.

Observación A.2.6. Como cada nivel del segment tree representa una partición del arreglo A en rangos disjuntos entre sí, la posición i se encuentra contenida en exactamente uno de ellos, por lo que el resto de los nodos del nivel no se verán afectados por la modificación realizada. En otras palabras, solo debemos recomputar el valor de un único nodo por nivel.

Definiremos una función recursiva que recalcule el valor almacenado en los nodos del segment tree que contengan a i en el rango que representan. La misma será denotada por $\text{update}(\text{root}, a, b, i, x)$, de manera similar al caso de la consulta. Existen dos posibles escenarios:

- Si $a + 1 < b$, como i se encuentra contenida en el rango de solo uno de los dos hijos de root, no es necesario recalcular el valor contenido en el otro; Solo realizaremos una llamada recursiva en el hijo que represente a i, y

luego el nuevo mínimo del intervalo $[a, b)$ será el menor entre los valores ya actualizados de los hijos de root.

- Si $a + 1 = b$, nos encontraremos en la hoja del árbol que representa a la posición i del arreglo, por lo que solo deberemos cambiar el valor del nodo actual por x.

Sea p la raíz del segment tree de A. Dado que la cantidad de niveles es $\mathcal{O}(\log(n))$, podemos concluir gracias a la Observación A.2.6 que la cantidad de nodos visitados por la llamada a update$(p, 0, n, i, x)$, y por consecuente la complejidad temporal de la misma, es igual a $\mathcal{O}(\log(n))$.

Algoritmo 20 Definición recursiva de update$(\text{root}, a, b, i, x)$

```
1:  if a + 1 = b then                          ▷ Hoja que representa a A_i
2:      root.value ← x
3:  else                                       ▷ Llamada recursiva al hijo correspondiente
4:      m ← ⌊(a+b)/2⌋
5:      if i < m then
6:          update(root.left, a, m, i, x)
7:      else
8:          update(root.right, m, b, i, x)
9:      end if
10:     root.value ← mín(root.left.value, root.right.value)
11: end if
```

Bibliografía

[Cor01] Thomas Cormen. *Introduction to algorithms.* MIT Press, Cambridge, Mass, 2001.

[HL92] B-C. Huang and M. A. Langston. Fast Stable Merging and Sorting in Constant Extra Space*. *The Computer Journal*, 35(6):643–650, 12 1992.

[KLA+01] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In Amihood Amir, editor, *Combinatorial Pattern Matching*, pages 181–192, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[Knu97] Donald E. Knuth. *The art of computer programming. Vol. 3: Sorting and searching.* Bonn: Addison-Wesley, 1997.

[KSB06] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[KSPP05] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Constructing suffix arrays in linear time. *Journal of Discrete Algorithms*, 3(2):126–142, 2005. Combinatorial Pattern Matching (CPM) Special Issue.

[Lot83] M. Lothaire. *Combinatorics on words. Foreword by Roger Lyndon*, volume 17. Cambridge University Press, Cambridge, 1983.

[Man04] Giovanni Manzini. Two space saving tricks for linear time LCP array computation. In *Algorithm theory—SWAT 2004*, volume 3111 of *Lecture Notes in Comput. Sci.*, pages 372–383. Springer, Berlin, 2004.

[MM93] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. Comput.*, 22(5):935–948, 1993.

[Reu93] Christophe Reutenauer. *Free Lie algebras*, volume 7. Oxford: Clarendon Press, 1993.

[Ukk85] Esko Ukkonen. Algorithms for approximate string matching. *Inf. Control*, 64:100–118, 1985.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *14th Annual IEEE Symposium on Switching and Automata Theory (Iowa City, Iowa, 1973)*, pages 1–11. University of Iowa, 1973.