Proyecto #1 DAA: Procastinación ++

Lázaro Daniel González Martínez y Alejandra Monzón Peña

Desccripción del problema

Sean las cadenas S, T y una cadena vacía A, se construyen nuevas cadenas de la siguiente forma:

- ◇ Quitar la primera letra de S y ponerla al inicio de la cadena A.
- ◇ Quitar la primera letra de S y ponerla al final de la cadena A.

Estas operaciones se pueden realizar hasta que S quede sin caracteres.

Se desea contar cuántas cadenas diferentes se pueden crear con estas operaciones tales que T sea prefijo de la nueva cadena (A).

Definimos por comodidad la construcción de cadenas combinando de cualquier forma posible las dos operaciones permitidas como *construcción por aburrimiento*.

Solución Backtrack

Una primera solución al problema planteado se puede obtener haciendo una exploración por todas las posibles cadenas que se pueden formar y en cada caso comprobar si T es prefijo de dicha cadena.

```
def Procastinacion(S,T):
    if len(T) > len(S):
        return 0
    return Procastinacion2(S,T,'',0)

def Procastinacion2(S,T,A,i):
    m = len(T)
    k = len(A)
    if len(S) <= i:
        return T == A[0:m]
    return Procastinacion2(S,T,f'{A}{S[i]}',i+1)
                + Procastinacion2(S,T,f'{S[i]}{A}',i+1)
                    + ((m <= k) and (T == A[0:m]))
```

Figura 1: Código python del backtrack.

Este algoritmo de solución, aunque efectivo es demasiado costoso computacionalmente, sean $|S| = n$ y $|T| = m$, se tiene que:

$$T(n, m) = 2T(n-1) + m$$

de donde, al utilizar el Teorema Maestro para funciones decrecientes, se obtiene para este problema que $T(n, m) = \Theta(2^n + m)$.

Explotando características del problema

Una primera mejora que se puede hacer, se basa en la idea de que la primera letra de S al poder colocarse tanto al inicio como al final de la cadena vacía A, se generan así dos árboles de cadenas exactamente iguales (con las mismas cadenas), solo que estas se pueden considerar "diferentes" por haber tenido una decisión inicial distinta.

Por tanto una mejora inicial, consiste en no duplicar innecesariamente el espacio de búsqueda, sino asumir que en A inicialmente está el primer caracter de S y duplicar el resultado final. Aunque esta idea no mejora la complejidad temporal, reduce a la mitad la cantidad de operaciones a realizar.

Solución con Programación Dinámica

Representando las cadenas S y T por sus caracteres, tenemos:

$T = T_0T_1...T_{m-1}$

$S = S_0S_1...T_{n-1}$

Entonces T_i (S_i) denota al i-ésimo más un cararacter de $T(S)$, de igual modo $T_{i...j}(S_{i...j})$ denota a la subcadena $T_iT_{i+1}...T_j$ ($S_iS_{i+1}...S_j$) de la cadena T (S).

Definamos la función $f(i, j)$ como la cantidad de cadenas $A_{i...j}$ *construidas por aburrimiento* con los primeros $j - i + 1$ caracteres de S (es decir con los caracteres de $S_{0...j-i}$) donde para:

i $j < m, \quad A_{i...j} = T_{i...j}$

ii $j \geq m$ y $i < m, \quad A_{i...j} = T_{i...m-1}A_{m...j}$

iii $i > m, \quad A_{i...j} = A_{i...j}$

Luego $\sum_{j=m-1}^{n-1} f(0, j)$ es la cantidad total de cadenas que se pueden *construir por aburrimiento* que tienen como prefijo a T.

Notemos qué, para $i < m$ se cumple que:

$$f(i, i) = \begin{cases} 2, & T_i = S_0 \\ 0, & eoc \end{cases} \tag{1}$$

Esto se debe a que, como queremos formar subcadenas de tamaño 1 de T con el primer caracter de S, tenemos en los casos que hay coincidencia (2), dos

maneras de colocar el caracter, (por delante y por detrás) y en los restantes casos se tiene 0 puesto que no se tiene ninguna subcadena de T de longitud 1 al tomar ese caracter (3).

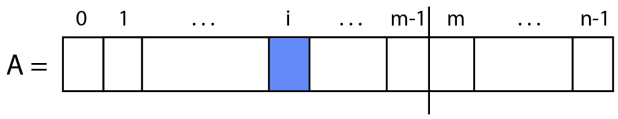

Figura 2:

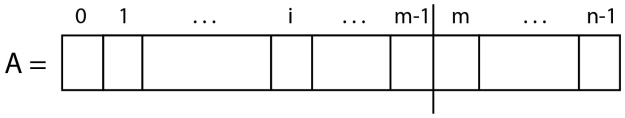

Figura 3:

Además se cumple, para $i \geq m$, que $f(i,i) = 2$, puesto que poner cualquier caracter en posiciones de A mayores que el prefijo T (4) origina una cadena de longitud 1 válida para la definición de la función f (iii) y el valor es 2, puesto que de igual modo este caracter se pudo colocar por delante o por atrás de la cadena vacía.

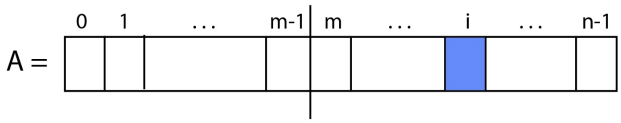

Figura 4:

A modo general $f(i,j)$ se puede construir recursivamente como:

$$f(i,j) = f(i+1,j)\mathbb{I}_{\{T_i = S_{j-i+1} \vee i \geq m\}} + f(i,j-1)\mathbb{I}_{\{T_j = S_{j-i+1} \vee j \geq m\}}$$

Ya que S_{j-i+1} se puede agregar por delante a las cadenas $A_{i+1}...A_j$ (5), o por atrás a las cadenas $A_i...A_{j-1}$ (6), cuando S_{j-i+1} coincide con T_i o T_j respectivamente, formando así cadenas $A_i...A_j$.

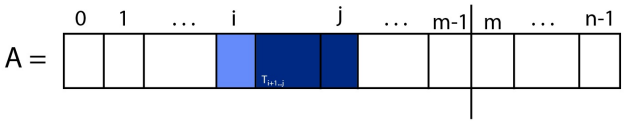

Figura 5:

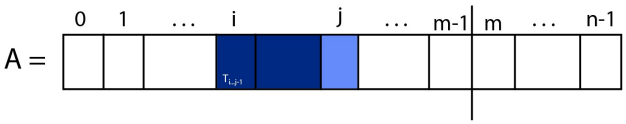

Figura 6:

En los casos (ii), a la subcadena $A_{i...j-1}$ se le puede agregar el caracter S_{j-i+1} por atrás sin afectar el prefijo (7), generando la cantidad de cadenas que había en $f(i, j-1)$, pero ahora de la forma $A_{i...j}$. Note que esta condición no implica que no se pueda cumplir que $T_i = S_{j-i+1}$ por lo que en este caso podría ponerse por delante sin problema (8).

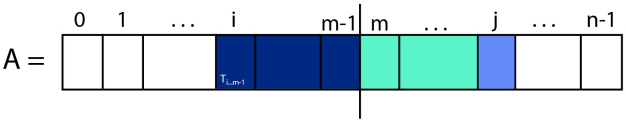

Figura 7:

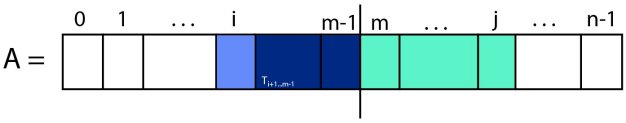

Figura 8:

Por último cuando $i \geq m$ entonces $j \geq m$ y se puede agregar S_{j-i+1} como prefijo de $A_{i+1...j}$ (9) y como sufijo de $A_{i...j-1}$ (10) cayendo en el caso (iii), por lo tanto tendríamos $f(i+1, j) + f(i, j-1)$ cadenas de la forma $A_{i...j}$.

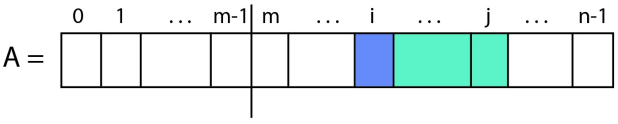

Figura 9:

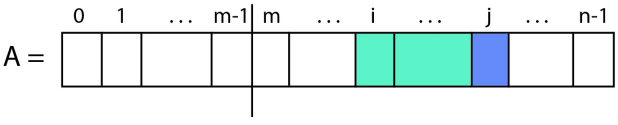

Figura 10:

Implementación de la dinámica

```
def solve_dp(S,T):
    n = len(S)
    m = len(T)

    if m > n:
        return 0

    dp = [[0 for j in range(n)] for i in range(n)]

    for i in range(n):
        if i >= m or T[i] == S[0]:
            dp[i][i] = 2

    for k in range(1, n):
        c = S[k]

        i = 0
        for j in range(k, n):
            if i >= m or c == T[i]:
                dp[i][j] += dp[i+1][j]
            if j >= m or c == T[j]:
                dp[i][j] += dp[i][j-1]
            i += 1

    return sum(dp[0][m-1:])
```

Correctitud del algoritmo

Como nos queda claro que $\sum_{j=m-1}^{n-1} f(0,j)$ es la respuesta a nuestro problema, demostrar que $dp[i][j] = f(i,j)$, sería suficiente para probar la correctitud del algoritmo. Por lo tanto demostremos que en el momento que se actualiza el valor de $dp[i][j]$ este coincidirá con $f(i,j)$.

Hagamos inducción en la longitud de la cadena que estamos formando $A_{i\ldots j}$, lo que es equivalente a hacer la inducción sobre el ciclo en que itera k.

- Caso base: $|A_{i\ldots j}| = 1$

  Como por definición, $i \leq j$ entonces $|A_{i\ldots j}| = 1$ si y solo si $i = j$, ya que $|A_{i\ldots j}| = j - i + 1 = 1$. Luego todas las cadenas que puedan pertenecer a nuestro caso base están en la diagonal de dp, y se inicializan atendiendo a la definición de $f(i,i)$ dada en (1), entonces tenemos que:

  $$dp[i][i] = f(i,i)$$

  . Note que se inicializan así en el ciclo que itera por i del código.

- Caso Hipótesis: Supongamos que para toda cadena $A_{i'\ldots j'}$ de tamaño menor que p se cumple que $dp[i'][j'] = f(i',j')$ con $j' - i' + 1 < p$. Esto es equivalente a que hasta la iteración $p-1$ del ciclo que itera sobre k, se cumple que lo planteado anteriormente[1].

[1] Observe que, para que la fórmula tenga sentido, $i' \leq j'$

Sea $A_{i...j}$ de tamaño p. Podemos asumir $i < j$, porque el caso base ya está tratado a parte. Luego como

$$f(i,j) = f(i+1,j)\mathbb{I}_{\{T_i=S_{j-i+1}\vee i\geq m\}} + f(i,j-1)\mathbb{I}_{\{T_j=S_{j-i+1}\vee j\geq m\}}$$

se aprecia que $f(i,j)$ depende de los valores de $f(i+1,j)$ y $f(i,j-1)$. Luego como $|A_{i+1...j}| = j-(i+1)+1 = j-i$, y $j-i < p$, entonces por hipótesis $f(i+1,j) = dp[i+1][j]$. De manera similar como $|A_{i...j-1}| = j-1-i+1 = j-i < p$, aplicamos la hipótesis pero en este caso $f(i,j-1) = dp[i][j-1]$. Luego

$$f(i,j) = dp[i+1][j]\mathbb{I}_{\{T_i=S_{j-i+1}\vee i\geq m\}} + dp[i][j-1]\mathbb{I}_{\{T_j=S_{j-i+1}\vee j\geq m\}}$$

y finalmente $dp[i][j]$ al actualizarse con este cálculo nos quedará:

$$dp[i][j] = f(i,j)$$

Luego por Principio de Inducción Matemática demostramos que al concluir el algoritmo queda computado para cada $dp[i][j]$, con $0 \leq i \leq j < n$, la cantidad de cadenas $A_{i...j}$ que se pueden *construir por aburrimiento* con los primeros $|A_{i...j}| = j-i+1$ caracteres de S, es decir, queda guardado en $dp[i][j]$ el valor de $f(i,j)$.

Complejidad Temporal

Definir e inicializar la matriz dp se hace en $2n^2$ operaciones ya que esta tiene dimensión $n \times n$.

El ciclo que itera por k ejecuta el ciclo que itera por j, unas $n-1$ veces. Y este ciclo se ejecuta $n-k$ veces para la iteración k. Es decir, ambos ciclos ejecutan un total de veces igual a:

$$\sum_{k=1}^{n-1}(n-k) = (n-1)+(n-2)+...+1 = \frac{n(n-1)}{2}$$

Finalmente cuando se calcula la suma de los $dp[0][i]$ para $i \geq m$, se hacen $n-m+1$ iteraciones.

Por lo tanto al sumar cada uno de estos costos considerando también, el tiempo constante que requieren las operaciones intermedias, nos quedaría algo así:

$$T(n,m) = c_1 + 2n^2 + c_2\frac{n(n-1)}{2} + n - m + 1$$

Demostremos que $T(n,m)$ es $O(n^2)$:

Para esto debemos encontrar una constante C tal que a partir de cierto n, se cumpla $T(n,m) \leq Cn^2$.

$$T(n,m) \leq c_1 + 2n^2 + c_2\frac{n(n-1)}{2} + n + 1$$

Calculemos el límite:

$$\lim_n \left(\frac{c_1 + 2n^2 + c_2\frac{n(n-1)}{2} + n + 1}{n^2}\right)$$

$$= \lim_n \left(\frac{2c_1 + 4n^2 + c_2 n(n-1) + 2n + 2}{2n^2} \right)$$

$$= \frac{4 + c_2}{2}$$

Como este límite es finito y mayor que 0, existirá una constante C tal que :

$$c_1 + 2n^2 + c_2 \frac{n(n-1)}{2} + n + 1 \leq Cn^2$$

y por transitividad, $T(n,m) \leq Cn^2$. Luego $T(n,m)$ es $O(n^2)$.

Incluso si queremos ser más exquisitos podemos demostrar que $T(n,m) = \theta(n^2)$.

Como:

$$c_1 + 2n^2 + c_2 \frac{n(n-1)}{2} \leq T(n,m)$$

Calculando el límite siguiente:

$$\lim_n \left(\frac{c_1 + 2n^2 + c_2 \frac{n(n-1)}{2}}{n^2} \right)$$

$$= \lim_n \left(\frac{2c_1 + 4n^2 + c_2 n(n-1)}{2n^2} \right)$$

$$= \frac{4 + c_2}{2}$$

De forma similar, existirá una constante C tal que a partir de cierto n, se cumple que

$$Cn^2 \leq c_1 + 2n^2 + c_2 \frac{n(n-1)}{2}$$

y por transitividad se cumple que $Cn^2 \leq T(n,m)$. Luego $T(n,m) = \Omega(n^2)$. Y finalmente $T(n,m)$ es $\theta(n^2)$.

Y si te dijera que esto se puede mejorar??

Al analizar numerosos casos, pudimos comprobar que en muchos de ellos, el primer caracter de S, S_0 no estaba presente en la cadena T, por tanto luego de poner este caracter en A había solo una de las dos opraciones posibles para formar el prefijo T en A y es poner todo caracter de T en A con movimientos hacia alante.

(Dem) Sea una cadena A, con prefijo T,creada a partir de una cadena S, cuyo primer caracter S_0 no pertenezca al alfabeto de T, supongamos que en el prefijo T de A, algún caracter se colocó con un movimiento hacia atrás. Sea S_k ese caracter, entonces como se puso con un movimiento hacia atrás, la posición j de S_k en A es mayor que la posición de S_0 en A. Como asumimos que S_k está en el prefijo T de A, entonces todo otro caracter anterior a S_k en A tiene que estar en el prefijo T, por lo tanto S_0 tendría que estar en el prefijo, lo cual es una contradicción. Así queda demostrado que si S_0 no pertenece a las letras

que forman a T, todas las cadenas A que tengan a T como prefijo construidas por aburimiento, contruyen al prefijo solo com movimientos hacia adelante.

De la demostración anterior se deduce que como solo se usan movimientos hacia alante, reverso del prefijo T ha de ser subsecuencia de S.

De esta observación percibimos que era posible obtener una solución al problema separando los modos de construcción de prefijos en una partición:

- Con cadenas cuyo prefijo T se contruyó solo con movimientos hacia alante.
- Con cadenas cuyo prefijo T se contruyó con al menos un movimiento hacia atrás.

Prefijo T solo con movimientos hacia alante

Definamos la función $prefix(i, j)$ como la cantidad de prefijos de T de longitud $j + 1$ que se pueden formar utilizando los últimos $n - i$ caracteres de S con movimientos hacia alante.

Si hacemos un primer análisis, nos podemos dar cuenta que:

$$prefix(i,0) = \begin{cases} 0, & i = n \\ prefix(i+1,0) + n - i, & i < n \text{ y } T_0 = S_{n-i-1} \\ prefix(i+1,0), & eoc \end{cases}$$

En el caso $i = n$ la cantidad de prefijos de longitud 1 que se pueden hacer con 0 caracters es evidentemente 0.

En el caso $i < n$ y $T_0 = S_{n-i-1}$, como el caracter S_{n-i-1} puede construir una cadena de longitud 1 que sea prefijo de T al ser agregado con un movimiento hacia alante a la cadena en formación; y los $n - i - 1$ caracteres finales de S se pueden (o no) insertar hacia atrás (uno a uno) en la cadena en formación, se tendrían $1 + (n - i - 1) = n - i$ nuevas cadenas con prefijo de longitud 1 de T. Además se tienen las formas que ya existían de construir cadenas con $n-(i+1)$ caracteres, es decir $prefix(i + 1, 0)$.

En el tercer caso, al no haber coincidencia de S_{n-i-1} con T_0, la inclusión de S_{n-i-1} al ponerlo por delante, no genera nuevas cadenas con prefijos de longitud 1 de T. Note que no tiene sentido poner estos caracteres por detrás hasta que no se tenga la seguridad de que exista un caracter que pueda formar el prefijo de longitud 1 de T, y si este caracter existe, en alguna posición $k < n - i - 1$ de S, más adelante estos caracteres se incluirían en $prefix(k, 0)$. Por tanto, hasta este punto se mantiene que la cantidad de prefijos formados es la misma a la que se tenía sin contar con S_{n-i-1}, es decir $prefix(i + 1, 0)$.

Esto se puede representar como:

$$prefix(n,0) = 0$$
$$prefix(i,0) = prefix(i+1,0) + (n-i)\mathbb{I}_{\{T_0 = S_{n-i-1}\}}$$

Luego con un análisis similar se tiene que para $j > 0$:

$$prefix(i,j) = \begin{cases} 0, & i = n \\ prefix(i+1,j) + prefix(i+1,j-1), & i < n \text{ y } T_0 = S_{n-i-1} \\ prefix(i+1,j), & eoc \end{cases}$$

Trivialmente cuando $i = n$, la cantidad de prefijos de longitud $j+1$ que se pueden hacer con los últimos 0 caracteres de S es 0.

Cuando $i < n$ y $T_i = S_{n-i-1}$, se puede colocar por delante S_{n-i-1} faltaría poner las letras correspondientes al prefijo de T de longitud j para obtener el prefijo T de longitud $j+1$, como en la cadena en formación hasta el momento se tienen que haber puesto con los movimientos permitidos todos los caracteres de $S_{0\ldots n-i-1}$, el prefijo de longitud j que falta se debe pode hacer con los restantes $n-i$ caracteres finales, es decir que se pueden obtener tantos como $prefix(i+1, j-1)$. Además se conservan las formas de hacer el prefijo de longitud $j+1$ sin utilizar el caracter S_{n-i-1} como parte del prefijo, o sea $prefix(i+1, j)$.

En otro caso, solo se puede construir T con lo que ya existía de la formación, es decir, sin utilizar S_{n-i-1}, se conservan las formas de hacer el prefijo de longitud $j+1$ ($prefix(i+1, j)$).

Correctitud de la función `compute_prefix`

Para obtener una representación computacional de la función $prefix$, rellenamos una matriz dp de $(n+1) \times m$ tal que $dp[i][j] = prefix(i, j)$. Utilicemos Inducción Matemática para demostrar la correctitud del algoritmo. Primero demostremos que $dp[i][0] = prefix(i, 0)\ \forall\ n \leq i \leq 0$.

- Caso Base: $i = n$ Cuando $i = n$, $dp[n][0] = 0$ ya que la inicialización de la matríz rellena dp con 0 por defecto (esto se cumple además para toda j). y $prefix(n, 0) = 0$ (en general $prefix(n, j) = 0$), de done $prefix(n, 0) = dp[n][0]$.
- Caso Hipótesis: Supongamos que para $i = k$ se cumple que $dp[k][0] = prefix(k, 0)$.

  Sea $i = k - 1$, se tiene que

  $$prefix(k-1, 0) = prefix(k, 0) + (n-k+1)\mathbb{I}_{\{T_0 = S_{n-k}\}}$$

  . Luego como $dp[k][0] = prefix(k, 0)$ por hipótesis, entonces

  $$prefix(k-1, 0) = dp[k][0] + (n-k+1)\mathbb{I}_{\{T_0 = S_{n-k}\}}$$

  y precisamente la expresión $dp[k][0] + (n-k+1)\mathbb{I}_{\{T_0 = S_{n-k}\}}$ es lo que se le asigna a $dp[k-1][0]$ en el código, en el ciclo que inicializa la matriz dp. Por tanto se tiene que $dp[k-1][0] = prefix(k-1, 0)$.
- Caso Base: $j = 0$

  Como se demostró en la inducción anterior para todo $n \leq i \leq 0$ se cumple que $dp[i][0] = prefix(i, 0)$.

- Caso Hipótesis: Supongamos que para $j = k$, para todos los valores $n \leq i \leq 0$, se cumple que $dp[i][k] = prefix(i, k)$.

  Sea $j = k + 1$, sabemos que:

  $$prefix(i, k+1) = prefix(i+1, k+1) + prefix(i+1, k)\mathbb{I}_{\{T_{k+1}=S_{n-i-1}\}}$$

  . Por hipótesis sabemos que $prefix(i+1, k) = dp[i+1][k]$, faltaría demostrar que $prefix(i+1, k+1) = dp[i+1][k+1]$. Hagamos una inducción sobre i, manteniendo la columna actual para demostrarlo.

  - Caso Base: $j = k + 1$, $i = n$
    Como planteamos con anterioridad la inicialización de la matríz rellena dp con 0 por defecto, por tanto y $prefix(n, k+1) = dp[n][k+1] = 0$.
  - Caso Hipótesis: Supongamos que $prefix(l, k + 1) = dp[l][k + 1]$.
    Sea $i = l - 1$, como

    $$prefix(l-1, k+1) = prefix(l, k+1) + prefix(l, k)\mathbb{I}_{\{T_{k+1}=S_{n-l}\}}$$

    Por la hipótesis interna tenemos que $prefix(l, k) = dp[l][k]$, y por la hipótesis externa tenemos que $dp[l][k + 1] = prefix(l, k + 1)$
    Luego

    $$prefix(l-1, k+1) = dp[l][k+1] + dp[l][k]\mathbb{I}_{\{T_{k+1}=S_{n-l}\}}$$

    y como $dp[l - 1][k + 1]$ se computa de esta forma, tenemos que:

    $$dp[l-1][k+1] = prefix(l-1, k+1)$$

  Luego por Principio de Inducción Matemática queda demostrado que $prefix(i + 1, k + 1) = dp[i + 1][k + 1]$. De esta manera ahora se tiene que

  $$prefix(i, k+1) = dp[i+1][k+1] + dp[i+1][k]\mathbb{I}_{\{T_{k+1}=S_{n-i-1}\}}$$

  Y finalmente como $dp[i][k + 1]$ se calcula con este valor, entonces concluimos que

  $$prefix(i, k+1) = dp[i][k+1]$$

Luego por Principio de Inducción Matemática queda demostrado que $dp[i][j] = prefix(i, j)$.

Entonces la cantidad de cadenas que se pueden formar con prefijo T construido solo con movimientos hacia alante, con los caracteres de S sería:

$$2 \cdot \sum_{\forall i, T_{m-1}=S_i} (prefix(i, m-1) - prefix(i+1, m-1)) \cdot 2^{\text{máx}(i-1,0)}$$

Note que $prefix(i, m - 1) - prefix(i + 1, m - 1)$ representa del total de cadenas posibles, con prefijo T construido solo con movimientos hacia adelante, aquellas que tienen como último caracter de T el caracter S_i. Observe que antes

de S_i podemos formar de cualquier forma (por aburrimiento) cadenas antes de empezar a formar el prefijo. La cantidad de formas para construir cadenas con los primeros $i-1$ caracteres sería colocar una a una las letras de 2 formas: por delante o por detrás. Esto nos dar'ia un árbol binario de profundidad $i-1$, en el cual la cantidad de cadenas sería 2^{i-1}. Note que cuando no hay caracteres, es decir, cuando la primera letra forma parte de T, partimos con solo una forma de generar T (con movimientos hacia adelante). De esta forma se justifica la parte de la sumatoria de la fórmula anterior. Ahora, como el primer caracter se asume puede ser colocado por delante y por detrás, y esto no afecta la contrucción de T, y solo estamos contando la cantidad de cadenas en las que el primer caracter de S se pone en solo uno de estos sentidos, faltaría duplicar las cantidades obtenidas de cadenas, por esto el resultado final es el doble de la sumatoria.

Prefijo T con al menos un movimiento hacia atrás

Definamos la función $sufix(i,j)$ como la cantidad de cadenas $A_{i\ldots j}$ *construidas por aburrimiento* con los primeros m caracteres de S (es decir con los caracteres de $S_{0\ldots m-1}$), que tengan al menos un caracter colocado con un movimiento hacia atrás en la subcadena de T que forman, donde para:

i $j < m, \quad A_{i\ldots j} = T_{i\ldots j}$

ii $j \geq m$ y $i < m, \quad A_{i\ldots j} = T_{i\ldots m-1} A_{m\ldots j}$

Observe que solo interesan los primeros m caracteres de S, ya que a partir de los $m+1$, hacer cualquier movimiento hacia atrás se sale del rango del prefijo de longitud m de la cadena que se está formando, y por lo tanto este caracter no formaría parte de un futuro prefijo T.

Lema 1
Sea A una cadena construida por aburrimiento, a partir de S y prefijo T. Si para construir A se utilizó al menos un movimiento hacia atrás para formar a T, entonces $A_{m-1} = T_{m-1} = S_l$ fue colocado con un movimiento hacia atrás.

(Dem) Supongamos que $A_{m-1} = T_{m-1} = S_l$ fue colocado con un movimiento hacia adelante. Como sabemos que al menos se realizó un movimiento hacia atrás durante la construcción del prefijo T, entonces existe algún S_k que con un movimiento hacia atr'as fue colocado en A en una de las primeras $m-1$ posiciones. Si $k < l$, entonces S_l al colocarse por delante en A, este ya tendrá colocado S_k, en alguna posición a su derecha, y por lo tanto como S_l es el útlimo caracter del prefijo, S_k quedar'ia fuera de este (contradicción). Para $l < k$, el caracter S_l fue colocado primeramente por delante, y luego S_k por detrás, donde nuevamente este caracter estaría en alguna posición a la derecha de S_l en A quedando nuevamente fuera del prefijo (contradicción). Por último no tiene sentido tomar $k = l$, porque este se pone por alante bajo la suposición y entramos en otra contradicci'on.

Del **Lema 1** y del principio que todo movimiento hacia atrás presente en el prefijo T que se forme en la cadena A se realiza en los primeros m caracteres de S, se deduce que las cadenas que podamos formar que tengan prefijo T con al menos un movimiento hacia atrás se originan solo si m letras de S se generan

sufijos de T que tengan al menos un movimiento hacia atrás.

De ahí que de la función $sufix(i, j)$ sean de interés los valores

$$sufix(m - 1 + i, i) \ \forall \ 0 \leq i \leq m$$

puesto que estos valores representan la cantidad de cadenas que se pueden constuir con prefijo igual al sufijo de T de longitud $m - i$, a partir de los primeros m caracteres de S habiendo colocado algún caracter con un movimiento por detrás.

Correctitud de la función `compute_sufix`

Para obtener una representación computacional de la función $sufix$, rellenamos una matriz dp de $m \times m$ tal que $dp[i][j] = sufix(i, j)$.

Análisis de complejidad temporal

Como el algoritmo final contempla el llamado a las funciones `compute_prefix` y `compute_sufix` es necesario conocer el costo de estas funciones.

Sea el código de la función `compute_prefix`:

```
def compute_prefix(S,T):
    m = len(T)
    n = len(S)

    dp = [[0 for j in range(m)] for i in range(n +1)]


    for i in range(n-1,-1,-1):
        dp[i][0] = (n - i)*(S[i] == T[0])  + dp[(i+1)%n][0]

    for i in range(1,m):
        for j in range(n-i -1,-1,-1):
            dp[j][i] = dp[(j+1)%n][i]
                + dp[(j+1)%n][i-1] * (T[i] == S[j])

    return dp
```

Se tiene que el costo de calcular m (n) es constante c_1 (c_2), y la construcción de la matriz dp tiene un costo de $m(n + 1)$. Luego tenemos un primer ciclo que itera en la longitud de S y que ejecuta en su interior operaciones de costo constante, por lo que su costo es de $c_3 n$. Finalmente se tiene un ciclo que recorre la longitud de T excepto un caracter y que en su interior ejecuta otro ciclo tal que en la i-ésima iteración del ciclo externo, el ciclo interno realiza $n - i$ iteraciones con operaciones constantes, esto se puede expresar como:

$$\sum_{i=1}^{m-1} c_4(n - i) = c_4 \sum_{i=1}^{m-1} (n - i) = c_4[(n - 1) + (n - 2) + ... + (n - m + 1)]$$

$$= c_4 \left(\frac{n(n-1)}{2} - \frac{(n-m)(n-m+1)}{2} \right) = c_4 \frac{n(n-1) - (n-m)(n-m+1)}{2}$$

$$c_4 \frac{n^2 - n - n^2 + 2nm - n - m^2 + m}{2} = c_4 \frac{2nm - m^2 - 2n + m}{2}$$

De lo que en total resulta que:

$$P(n,m) = c_1 + c_2 + c_3 n + c_4 \frac{2nm - m^2 - 2n + m}{2}$$

Como esta función solo se ejecuta para $1 \leq m \leq n$ entonces:

$$P(n,m) \leq c_1 + c_2 + c_3 n + c_4 \frac{2nm + m}{2}$$

$$\leq c_1 + c_2 + c_3 n + 3c_4 nm \leq nm(c_1 + c_2 + c_3 + 3c_4) = Cnm$$

Sea el código de la función `compute_sufix`:

```
def compute_sufix(S,T):
    m = len(T)
    dp = [[0 for j in range(2*m)] for i in range(2*m)]
    tmdp = [[0 for j in range(2*m)] for i in range(2*m)]

    for i in range(2*m):
        if i >= m or T[i] == S[0]:
            tmdp[i][i] = 2

    for k in range(1, m):
        c = S[k]
        i = 0

        for j in range(k, m + k):

            if j >= m:
                dp[i][j] += dp[i][j-1]
                tmdp[i][j] += tmdp[i][j-1]

            if j< m and c==T[j]:
                dp[i][j] += tmdp[i][j-1]
                tmdp[i][j] += tmdp[i][j-1]

            if i >= m or c == T[i]:
                dp[i][j] += dp[i+1][j]
                tmdp[i][j] += tmdp[i+1][j]

            i += 1
    return dp
```

La obtención de m tiene costo constante c_1, la creación de las matrices dp y $tmdp$ es de $4m^2$ cada una. El ciclo que pone los valores iniciales en la matriz $tmdp$ tiene un costo de $2c_2 m$ y finalmente se tiene un ciclo que itera $m-1$ veces y en cada iteración ejecuta otro ciclo de $m-1$ iteraciones en el que a su vez se teien que en el peor caso se ejecutan operaciones de costo constante c_3. Finalmente se tiene que:

$$S(n,m) = c_1 + 4m^2 + 4m^2 + 2c_2m + c_3(m-1)^2$$

Como esta función solo se ejecuta si $1 \leq m$

$$S(n,m) \leq c_1 + 8m^2 + 2c_2m + c_3m^2 \leq m^2(c_1 + 8 + 2c_2 + c_3) = Cm^2$$

Finalmente el código principal `ppp` (**P**rocastinación**P**lus**P**lus) presenta el siguiente código:

```
def ppp(S,T):
    n = len(S)
    m = len(T)
    if n> m: return 0

    prefix = compute_prefix(S,T)
    count = 0

    if T.__contains__(S[0]):
        sufix = compute_sufix(S, T)
        count= sufix[0][m-1] * (n - m + 1)

        for i in range(1,m):
            count += sufix[i][m-1+i]* prefix[m][i-1]

    return count + int(sum((prefix[i][-1]
                    - prefix[i+1][-1])* (2**(max(i-1,0)))
                        *(T[-1]==S[i]) for i in range(n))*2)
```

Se tiene que el costo de calcular m (n) es constante c_1 (c_2), en el peor caso se tiene que $m \leq n$ por lo que se ejecuta la función `compute_prefix` de costo $P(n,m)$ y se inicializa el contador que es una operación de costo constante c_3. Comprobar si el primer caracter de S está en T tiene un costo de m y para el peor caso se ejecuta el código dentro de la condicional, es decir, se llama a la función `compute_sufix` que tiene costo $S(n,m)$, se actualiza el contador con una operación de orden constante c_4 y se continúa incrementando en contador en un ciclo que se ejecuta $m-1$ veces. Además siempre se ejecuta un ciclo final de n iteraciones para computar la cantidad de cadenas de prefijo T donde el prefijo se obtiene solo con mivimientos hacia adelante. De modo que se obtiene:

$$T(n,m) = c_1 + c_2 + P(m,n) + m + S(n,m) + c_4 + c_5(m-1) + c_6n$$

$$T(n,m) \leq c_1 + c_2 + C_1nm + m + C_2m^2 + c_4 + c_5(m-1) + c_6n$$

Como $1 \leq m \leq n$

$$\leq mn(c_1 + c_2 + C_1 + 1 + C_2 + c_4 + c_5 + c_6) = Cmn$$

De donde existe $C' \geq C$ tal que para todo m, n se cumple que $T(n,m) \leq C'nm$, por tanto $T(n,m) = O(nm)$.

Generador y Probador de casos

Para comprobar la correctitud y eficiencia de los algoritmos creamos un generador y probador de casos pruebas. Para la generación de casos definimos 3 alfabetos:

- $A_1 = \{\texttt{a}, \texttt{b}, \texttt{c}\}$
- $A_2 = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}\}$
- $A_3 = \{\texttt{a}, \texttt{b}, \texttt{c}, \texttt{d}, \texttt{e}\}$

Generamos 6 cadenas T, con longitud entre 2 y 19. Y por cada T se generaron cadenas S de tamaño desde $|T|$ hasta $10|T| - 1$, de 3 tipos, mezclando aleatoriamente las letras de T con las que se pueden añadir según el alfabeto del tipo seleccionado:

- Tipo 1: Añadiendo caracteres del Alfabeto 1
- Tipo 2: Añadiendo caracteres del Alfabeto 2
- Tipo 3: Añadiendo caracteres del Alfabeto 3

La selección aleatoria de los caracteres es uniforme, es decir, cada letra tiene la misma probabilidad de ser seleccionada.

De esta forma se generaron 22680 casos de pruebas de los que 7404 resultaron con valor de 0 al ser evaluadas por el algoritmo de programación dinámica, para representar el 32,6455 % de los casos totales. Para comparar los resultados del backtrack con la dinámica $O(n^2)$, se utilizaron 630 casos de pruebas de los que el 92 de ellos resultaron en 0, representando el 14,6031 % de estos. Además para los 630 casos seleccionados, se obtuvo el mismo valor al evaluar ambos algoritmos. La diferencia significativa de ambos algoritmos es en la complejidad temporal, donde analíticamente son muy diferentes. Y en la práctica esto se comprobó donde, los 630 casos corrieron en menos de 1 minuto con el algoritmo de programación dinámica. Mientras que con el algoritmo de backtrack, se demoró más de 12 horas.

Luego con el código de la dinámica $O(n^2)$ se ejecutaron los 22680 casos de prueba para comparar los resultados con los de la dinámica $O(nm)$ en los que se obtuvo el mismo resultado para todos los casos prueba.

El generador, el tester, y el comparador se encuentran en los archivos `generator.py`, `tester.py` y `check.py`, respectivamente.