**Algoritmo Knuth-Morris-Pratt’s**

**Explicación, eficiencia y ejemplos de uso**

Materia: TTPS Opción C

Alumno: Gómez Lacchini, Lihuel Joaquín

Profesora: Schiavoni, María Alejandra

Diciembre 2022

Introducción:

En esta monografía se explica el funcionamiento del algoritmo Knuth-Morris-Pratt’s (KMP), las ventajas que presenta en términos de eficiencia y se muestran ejemplos de uso para resolver problemas.

Objetivo:

KMP es un algoritmo diseñado para realizar búsquedas de coincidencias de un texto dentro de otro (String Matching). Sirve para encontrar uno o todos los índices en los que un string o patrón P, se encuentra en un texto T.



Ejemplo: Buscar P = “na” en T = “Una manada de lobos”. En este caso se encuentra P comenzando en los índices 1 y 6 (Con indexación base 0).

Otro caso de String Matching en el que podría usarse KMP, es el clásico shortcut de búsqueda en Windows Ctrl+F para encontrar coincidencias en editores de texto, navegadores web, etc.

Algoritmo básico:

Antes de ver cómo funciona KMP, podemos revisar el funcionamiento de un algoritmo para String Matching básico. Si hubiese que desarrollar algo rápido y no necesariamente eficiente, lo más intuitivo es comenzar en cada caracter del texto, e ir chequeando de a uno que sea igual que P. Si se llega al final de la comparación sin encontrar diferencias entonces significa que se halló un índice. Luego se comienza por el siguiente carácter de T.

Sería algo así:

Texto

Descripción generada automáticamente

cod 1

Output:

Icono

Descripción generada automáticamente

Si tomamos m como el largo de T, y n como el largo P, entonces el tiempo de ejecución de este algoritmo es de O(m\*n), porque realiza un chequeo por cada posible índice de T, multiplicado por la cantidad de caracteres de P.

Una manera rápida de mejorar esto sería cortar la ejecución del segundo for (línea 5 cod 1) una vez que se encuentra una diferencia, pues no tiene sentido seguir comparando para ese índice.

Quedando algo así:

Imagen que contiene Texto

Descripción generada automáticamente

cod 2

Esto volvería el tiempo de ejecución variable según el caso, desde O(m) en el mejor caso, donde se encuentra una diferencia en la primera comparación de cada posible índice, hasta O(m\*n) en el peor caso, donde todas las comparaciones avanzan hasta el final.

Ejemplo: Con P = ‘XXXXY’,

1. Tener T= ‘QWERTYUIOP’, sería el mejor caso (O(m)), pues ‘X’, primer caracter de P siempre es diferente del primero de cada posible índice de T.
2. Con un T= ‘XXXXXXXXXY’ sería el peor caso (O(m\*n)), pues todos los posibles índices de T coinciden en ‘XXXX’, por ende, siempre avanzan en su comparación hasta el final.

**Knuth-Morris-Pratt’s (KMP)**

El problema con la solución anterior es que uno avanza sobre el string T, y puede (ej 2 del punto anterior, peor caso) estar chequeando el índice T[0], e ir comparando hasta encontrar diferencias recién en T[4], y ahí se vuelve a comparar desde T[1] en adelante. Esto haría que las posiciones T[1..4] se chequeen cada una varias (hasta un máximo de n) veces, contra distintos caracteres de P.

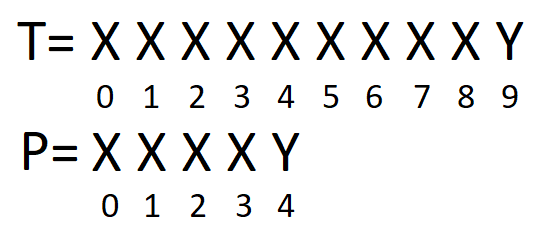
Lo que hace KMP es chequear cada carácter T[x] una sola vez, y sea que se encuentra una diferencia o una coincidencia, continuar comparando T[x+1]. De esta manera se podría mantener el tiempo de ejecución en O(m) en todos los casos.

Para lograr esto propone que, al momento de encontrar una diferencia, según el carácter de P que estoy comparando, saber de antemano cuántos de los últimos que revisé en T me sirven para la próxima posible coincidencia.

*Ejemplo 1:*

P = ‘XXXXY’,

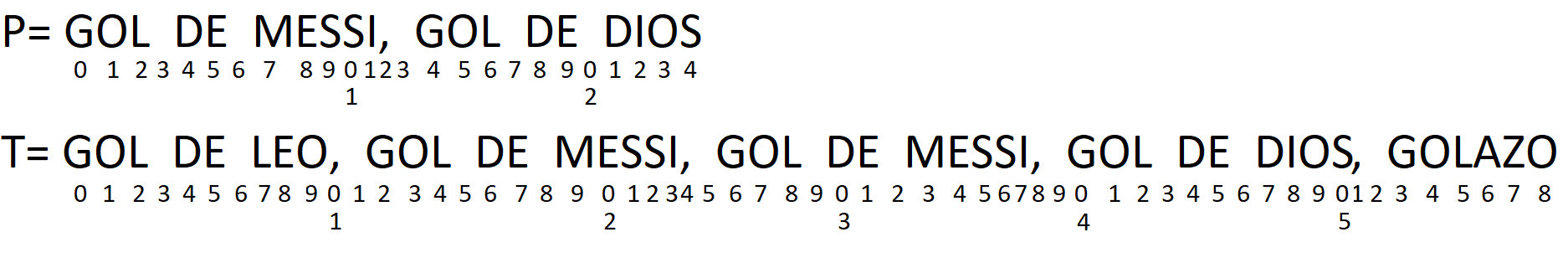
T= ‘XXXXXXXXXY’



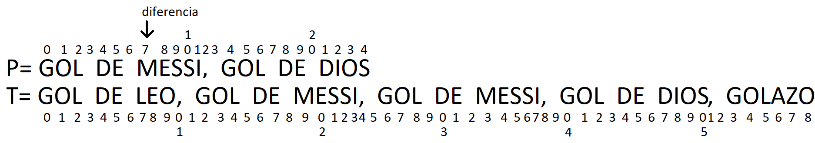
Inicio chequeando P[0] con T[0], hasta P[3] y T[3] que coinciden, pero P[4] y T[4] son diferentes. Aquí entonces lo que yo sé es que los 4 últimos que comparé son iguales, pero ¿Me sirven para algo?

En este caso sí, porque teniendo en cuenta que desde P[1] hasta P[3] (el lugar anterior a donde encontré desigualdad), el substring sería “XXX”, que coincide con el substring de P[0] a P[2], también “XXX”, lo que me permite seguir comparando desde P[3] con la posición en la que estoy actualmente en T: T[4]

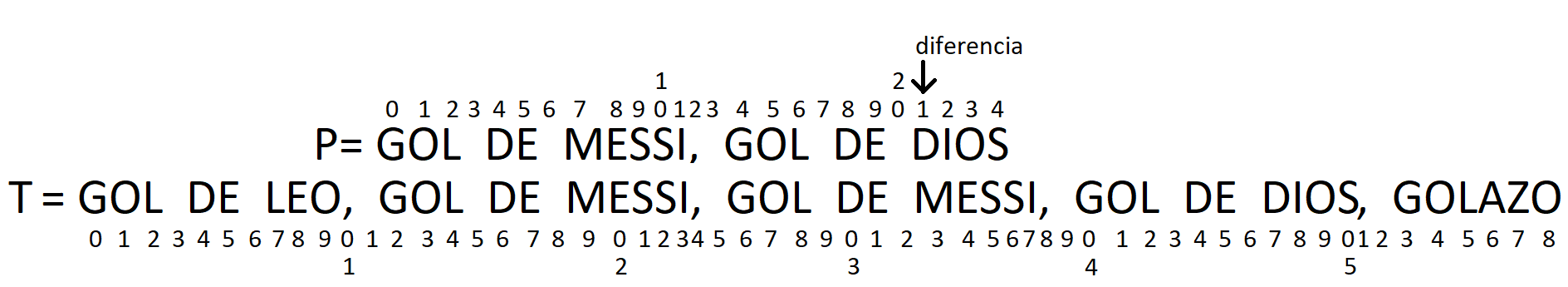
*Ejemplo 2:*



Inicio chequeando P[0] con T[0], hasta P[6] y T[6] que coinciden, pero P[7] y T[7] son diferentes. Aquí entonces lo que se sabe es que los 7 últimos que comparé son iguales, pero ¿Sirven para algo?

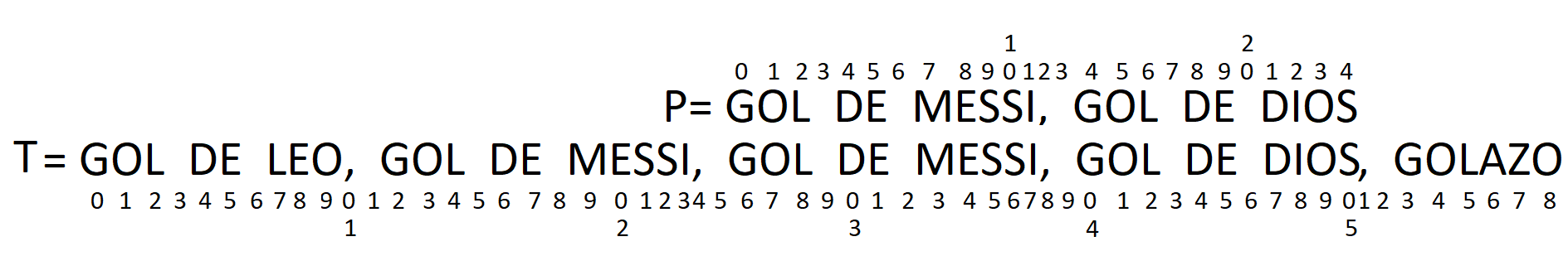


En este caso no, porque cualquier substring de inicio P[1..5] hasta P[6] son diferentes del inicio de P. A partir de aquí, T[7..11] difieren con P[0]. En T[12] se vuelve a encontrar coincidencia, que continua hasta llegar a la comparación de T[33] con P[21] donde hay una diferencia.



Aquí es donde se puede continuar verificando desde más adelante que el inicio de P, pues el substring de P[14] a P[20] “GOL DE “ (inmediatamente anterior a donde encontre la diferencia), es igual al inicio de P (P[0] a P[6]).

Entonces, la comparación continua con T[33] contra P[7], que son iguales, y en este caso llega hasta la comparación de P[24] con T[50] encontrando una igualdad.

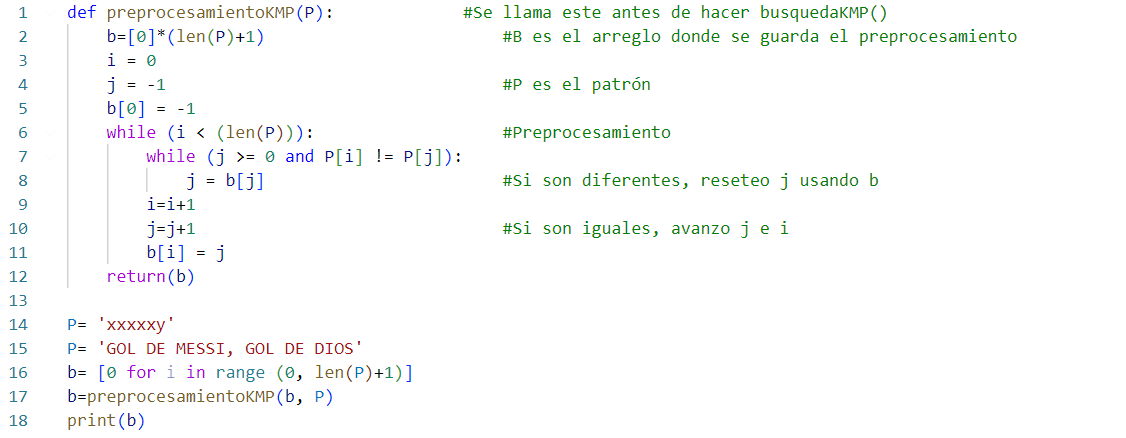


Luego de esto, se continúa comparando T[51] con P[0], desde el inicio de P pues el final de este no coincide con su inicio.

*Contraejemplo:*

Lo contrario sería, por ejemplo, teniendo P = “ABCDEF” y encontrando diferencia de P[4] = “E” con el texto al que estoy comparando, quiere decir que los últimos 4 fueron “ABCD”. Las ultimas cadenas que podrían servirme desde aquí son “BCD”, “CD”, y “D”, pero ninguna coincide con el principio de P. Así, se puede comenzar en el índice de T que estoy actualmente, verificando contra P[0], sin miedo de estar salteándome ninguna posible coincidencia.

Entonces, lo que hace KMP es un preprocesamiento del patrón P para guardar en un arreglo b, por cada uno de sus caracteres, cuántos (el máximo posible) de los últimos chequeados me sirven para continuar desde ese punto.

 cod 3

Este sería la función que cumple con ese preprocesamiento. Por ejemplo, enviándole el P del ejemplo anterior,

Texto

Descripción generada automáticamente

Nos devolvería:



De esta manera, ya habiendo visto la función de preprocesamiento, e incluyéndole la parte de búsqueda, el algoritmo completo quedaría así:

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

cod 4

El output de este código es:



Podemos ver que *busquedaKMP* usa dos punteros, i para avanzar sobre T, y j para avanzar en P. Luego de inicializarlos, tiene un primer while (línea 21) que recorre todo T para encontrar posibles coincidencias.

Una vez dentro, chequea si el T[i] es igual a T[j]. De no serlo, reinicia j usando el arreglo b con la información del procesamiento previo. Esta verificación también se realiza en un while, para que inmediatamente vuelva a hacer la misma comparación con el nuevo j.

Ejemplo: Si se entra a una iteración del primer while con j= 15, eso quiere decir que ya van 15 caracteres que coincidieron. Si el actual es diferente, se fija en b con que índice actualizar j. Supongamos que b[15]= 7. Entonces aquí vuelve a verificar T[i] en la iteración en la cual estaba, pero con P[7]. Si son iguales entonces continúa verificando normalmente (desde el 8vo carácter de P). Si no lo son, entonces vuelve a actualizar j. Si en este caso b[7]=0 (o sea que se debe empezar a comparar desde el inicio de P), entonces se actualiza j con b[0] que es igual a -1. Este -1 hace que salga del while de verificación, y que luego en el avance de j antes de la siguiente iteración, el puntero quede en 0 apuntando al comienzo de P.

Una vez sale del segundo while se avanzan ambos punteros, y luego verifica si j es igual al largo de P, que caso de ser verdadero indica que se encontró un índice. En este caso lo imprime, luego prepara j para el siguiente posible match, y continúa buscando coincidencias.

Ejemplos de uso:

En primer lugar, voy a desarrollar como es que utilicé el algoritmo para resolver el problema UVA-736: Lost in Space.

Puede encontrarse la consigna, input y output de ejemplo en <https://vjudge.net/problem/UVA-736>.

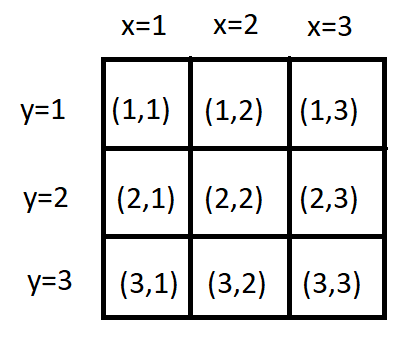
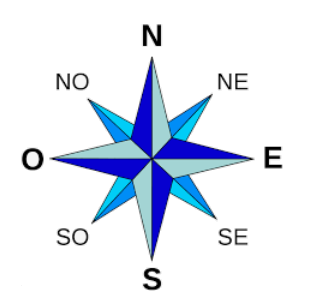
Resumidamente, lo que debe hacer el programa es buscar strings en un arreglo de N por N, en todas las direcciones. Con la dificultad de que puede haber “espacio” entre un carácter y el siguiente. Es decir, al encontrar un caracter ‘ ‘, no debe tenerse en cuenta en la comparación.

Con respecto a los strings que se deben buscar, no es necesario realizar ninguna modificación en el algoritmo de preprocesamiento. Por ende, la función preprocesamientoKMP() pude usarla tal cual se mostró en la explicación anterior. La función de busquedaKMP() si que necesitó algunos cambios.

Lo primero que busque fue una manera de reescribir la búsqueda de manera que pueda realizarse en cualquier dirección. Para esto, el índice i ya no sirve pues solo avanza en una dimensión. En este caso, es necesario tener dos variables: *x* e *y,* que servirán de puntero al carácter de la matriz que se está comparando.

En segundo lugar, es necesario modificar la manera en que estas dos variables avanzan, pues si para avanzar siempre sumo 1 a ambas, se movería siempre en la misma dirección.

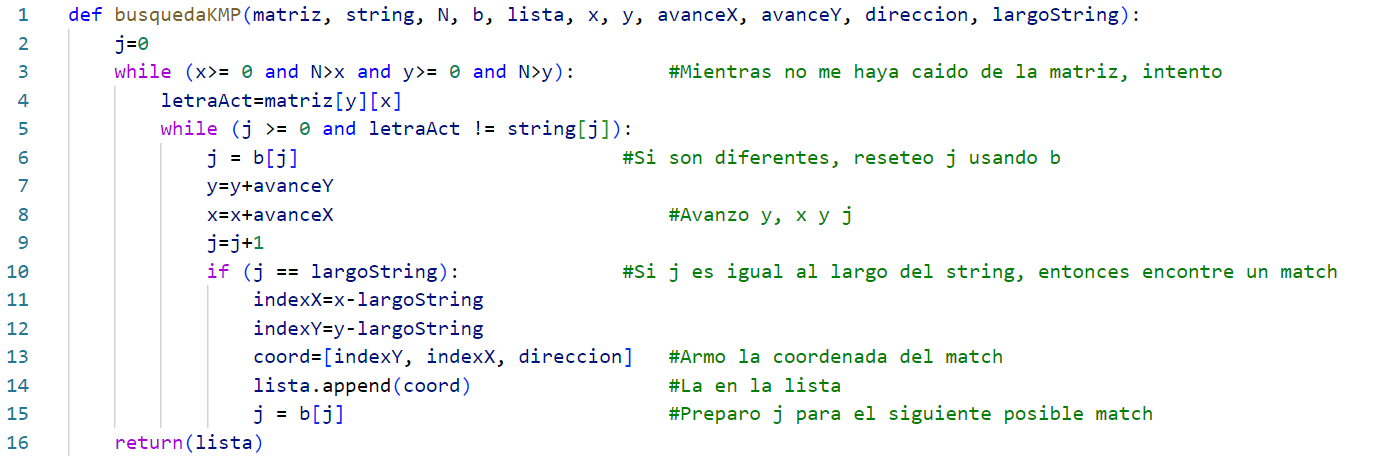
Ejemplo: Si estoy parado en *x*=2 e *y*=2, y sumo 1 a ambos punteros quedando *x*=3, *y*=3, el avance sería en dirección Sudeste. Pero para hacer un movimiento en dirección Oeste, la posición del siguiente carácter es *x*=1 e *y*=2, por lo que debería restarse 1 a *x*, y no modificar *y*. Para resolver esto, incluí dos variables *avanceX* y *avanceY,* indicadas por parámetro cuyo valor puede ser +1, 0 o -1, que representan el desplazamiento que se debe hacer a *x,* e *y* para ir en la dirección que se necesita.

img 1  img 2

Por último, como el recorrido de la matriz debe ser en todas las direcciones, y desde todos los lugares en que pueda iniciarse esa dirección, las posiciones de comienzo de *x* e *y* también deben venir dadas por parámetro.

Ejemplo: En la img 1, una búsqueda de determinado string en dirección Este se debe iniciar desde (1,1), (2,1) y (3,1). En cambio, una búsqueda en dirección Noroeste se debe comenzar en las coordenadas (1,3), (2,3), (3,3), (3,2) y (3,1).

Incluyendo estas variables, la modificación de busquedaKMP() quedaría así:



Ahora, falta tener en cuenta algunas cosas. Una de ellas es que la impresión de las coordenadas debe ser impresa en determinado orden, para esto utilice el modulo “bisect”, que permite realizar inserciones de manera ordenada.

El orden debe darse de menor a mayor, primero por numero de fila (indexY), luego por numero de columna (indexX), que resultan bien ordenados por el módulo. El último factor para el orden es la dirección, primero el Norte, luego NorEste y continuando en sentido horario. Como esto no corresponde con su orden alfabético, sume otra variable *codigoDireccion,* dada por parámetro para realizar este ordenamiento. Norte tendría código ‘A’, NorEste código ‘B’ y así.

Otra cosa para tener en cuenta es el “vacío” que puede haber entre caracteres. Para esto debí agregar un if que haga esta diferenciación, y en caso de encontrarse un espacio vacío solamente avance los punteros *x* e *y,* sin modificar j. Esto también cambia la manera en que se debe calcular *indexX* e *indexY,* ya que realizar la resta de la posición actual menos el largo del string(patrón) no sería correcto. La manera en que lo calculé fue con un while que retroceda en la matriz el largo del string, pero sin tener en cuenta los vacíos para obtener las coordenadas.

Texto

Descripción generada automáticamente

Este algoritmo ya devuelve todos los índices correctos y de manera ordenada en la dirección solicitada, por lo que resuelve las condiciones planteadas en el problema, pero hay algo más que se puede agregar para llegar a una solución más eficiente.

Teniendo en cuenta que, si yo estoy realizando una búsqueda en determinada dirección, y luego voy a hacer otra en la dirección opuesta, se puede invertir el string de la segunda y realizar la comparación de ambas a la vez.

Ejemplo: Buscando el patrón ‘SOL’ en el string ‘ESTAMOS SOLOS AQUÍ’, puedo buscar a la vez el patrón ‘SOL’ en dirección Este, y su inverso ‘LOS’ que, al encontrarlo, significaría una coincidencia para la dirección Oeste. En este caso hallando ‘SOL’ en la posición 9, y ‘LOS’ en la posición 11, lo que significa que ‘SOL’ fue ubicado en dirección Oeste en la posición 13.

Para el caso de los strings invertidos, ya no sería necesario retroceder en la matriz para encontrar el inicio, si no que la posición en la que se finaliza la comparación y se halla la coincidencia, es la de inicio en el sentido opuesto.

Incluyendo esto, el algoritmo queda así:

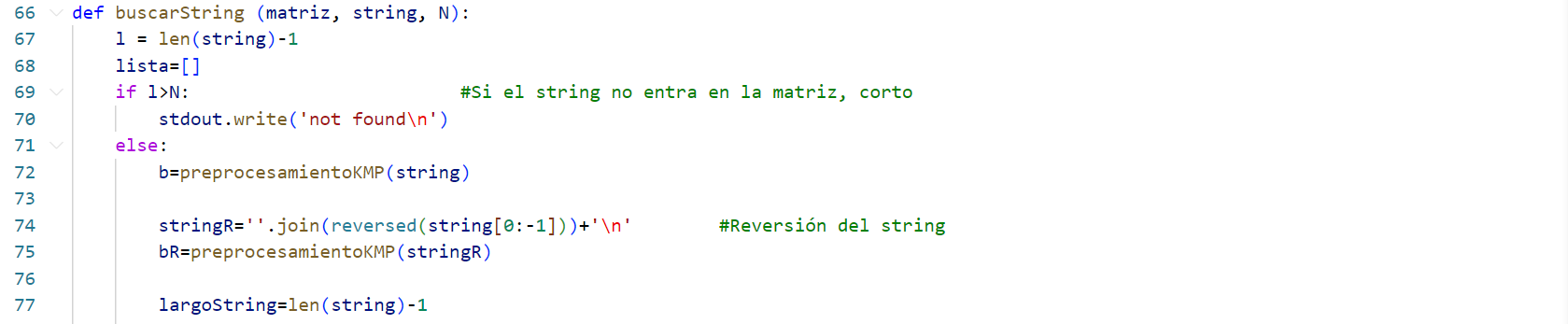
Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Texto

Descripción generada automáticamente

A partir de aquí, teniendo listo el algoritmo de búsqueda modificado, hace falta definir una función que al recibir una matriz y un string a buscar, solicite a busquedaKMP() la lista de índices en todos los sentidos y desde todos los puntos de inicio posible por cada una de las direcciones, pasándole en cada caso todos los parámetros necesarios:



Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente con confianza media

El parámetro N que llega a buscarString() es el largo (y ancho) de la matriz.

Por último, queda definir el cuerpo del programa, que lee la cantidad de casos, sus matrices, los patrones a buscar, realiza esta búsqueda por cada uno e imprime los resultados:

Texto

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Con esto, queda presentado y explicado todo el código con el que resolví el problema UVA-736: Lost in Space.

A continuación, la explicación del algoritmo que resuelve el problema COLWORD: Word Collection.

Puede encontrarse la consigna, input y output de ejemplo en <https://vjudge.net/problem/CodeChef-COLWORD>.

Lo que pide el ejercicio el calcular el número total de formas en que se puede tomar al menos una aparición de una palabra dentro de un string. No se debe tener en cuenta superposiciones, por ejemplo, en el string ‘ababa’ buscando el patrón ‘aba’, para calcular una nueva forma sólo se debe tener en cuenta la primera aparición o la segunda, no ambas a la vez.

En este caso, la función que cumple KMP en la solución es bastante clara, pues sirve para identificar todas las apariciones de W en N de manera eficiente. Una vez que tengamos todos los índices en los que W se encuentra en N, podemos calcular el total de combinaciones en que se puede ‘recortar’ W.

Para hacer el cálculo de posibilidades utilicé programación dinámica, que es un método que se puede utilizar para reducir los tiempos de ejecución al resolver problemas de ciertas características. Para poder utilizar esta estrategia se deben cumplir dos condiciones: ‘Overlapping subproblems’ y ‘Optimal substructure’.

‘Overlapping subproblems’ refiere a cuando se utiliza un subproblema para resolver problemas mayores, y estos se repiten. Por ejemplo: en la sucesión de Fibonacci (F3 = F1 + F2, y F4 = F2 + F3), calcular cada término de F3 en adelante supone calcular F2. Entonces si se está calculando F5, se necesitan F3 y F4, que ambos necesitan conocer F2. Si para esto la implementación calcula 2 veces F2 se estaría perdiendo eficiencia, lo que hace programación dinámica es realizar ese cálculo una sola vez. En este caso, F2 sería el subproblema superpuesto.

‘Optimal substructure’ refiere a cuando una solución óptima puede ser construida eficientemente a partir de las soluciones óptimas de sus subproblemas. En el ejemplo anterior, para calcular de manera óptima F5, es necesario calcular de manera óptima F3 y F4.

En el caso de Word Collection se cumplen ambas condiciones. Puede pensarse de la misma manera que Fibonacci. Dado un string de largo x, si yo sé el número total de formas en que se puede tomar al menos una aparición de W en el string, y le adjunto a este W al final, el nuevo número total sería el dato que ya tenía del primer string, más las nuevas combinaciones que suma la nueva aparición.

Ejemplo: String= ‘abcabc’ y W=’a’. W aparece 2 veces sin superposiciones en el String, por lo que el total de formas es 3 (sólo la primera, sólo la segunda, o ambas). Si el string pasa a ser ‘abcabca’, el nuevo total sería 7, que se puede separar en las 3 opciones que ya tenía antes de tener en cuenta la última aparición, las 3 nuevas que son las que ya se tenía más esta (sólo la primera más la última, sólo la segunda más la última, y ambas más la última), y una posibilidad extra que es la nueva aparición por sí sola.

Pasándolo a código, sería: dp[j]= dp[j-1]+dp[j-largoW]+1 , siendo j el índice, dp[j-1] las opciones que ya se tenían antes de tener en cuenta la última aparición, dp[j-largoW] las posibilidades a las que se puede sumar esta nueva (restarle largo W es para excluir aquellas en las que hay superposición), y +1 de la nueva aparición por sí sola. Esta es la suma en el caso de que j si sea una coincidencia con W, en el caso de no serlo se mantendrían las combinaciones de la posición anterior: dp[j]= dp[j-1].

Entonces, la parte de programación dinámica quedaría así:

Texto

Descripción generada automáticamente con confianza baja

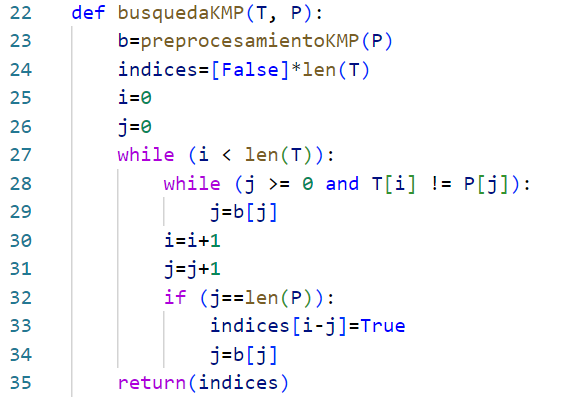
Para tener en cuenta: El caso de j=0 debe hacerse aparte, pues dp[j-1] daría error.

En los casos en que j es menor que el largo de W, dp[j-largoW] también estaría fuera del arreglo, por lo que esta no debe sumarse. Tampoco es necesario, pues en esos casos siempre habría superposición con una aparición que inicie antes.

Las sumas aplicándole %MOD al final, son con MOD = 1000000007, pues el ejercicio indica imprimir la respuesta módulo este número para achicarla. En vez de hacerlo solamente al final, se puede ir haciendo en cada suma para mantener el número chico durante la ejecución.

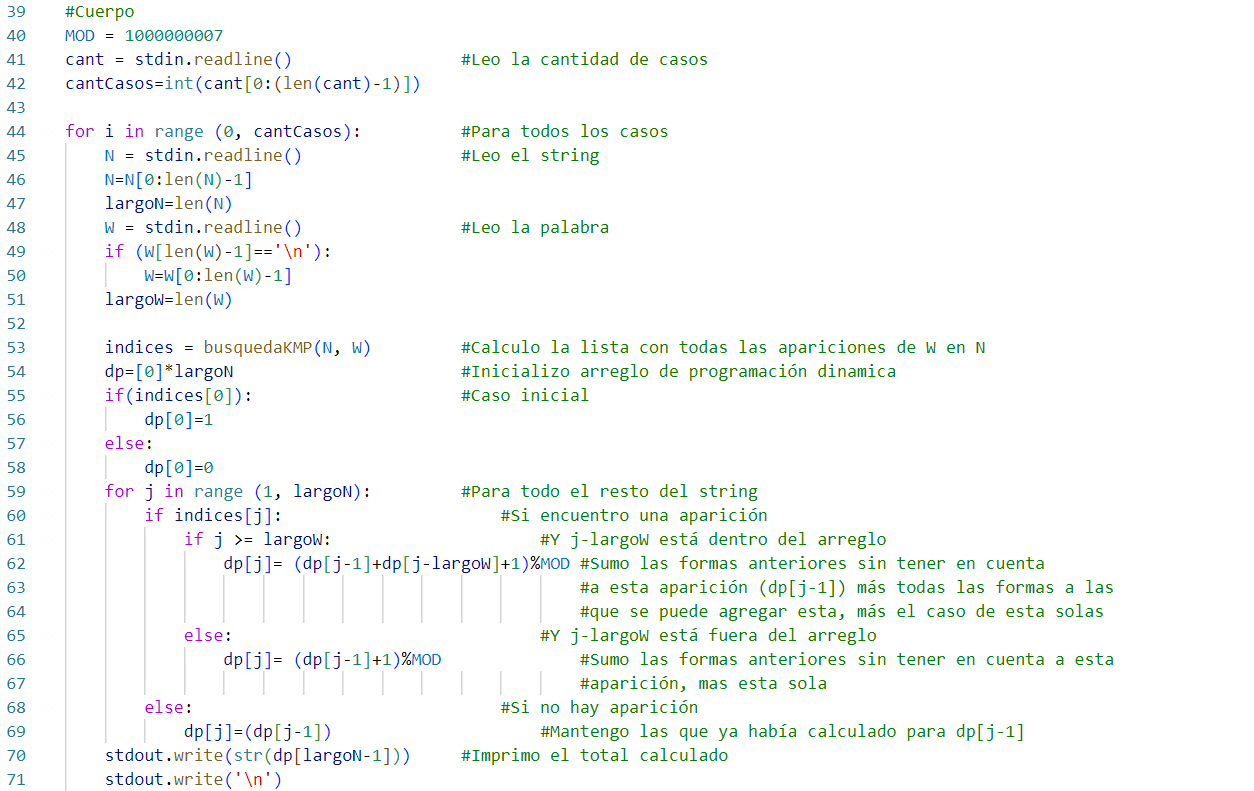
Incluso aplicando programación dinámica de esta manera la solución no es lo suficientemente eficiente, pues en los casos en que la lista de índices que devuelva busquedaKMP() sea muy extensa, la condición if j in indices: de las líneas 56 y 61 tomaría mucho tiempo de ejecución. Para resolver esto, se puede realizar una pequeña modificación en busquedaKMP() para que, en vez de devolver una lista con los índices, devuelva una con la información que indique si cada posición es un índice en el que aparece W o no, de manera que if j in indices: pueda convertirse en if indices[j]:, mejorando mucho la eficiencia.

Así, la función busquedaKMP() queda:



Con la función preprocesamientoKMP() definida de la misma manera que se desarrolló en la primera parte de la monografía.

Agregando este cambio, e incluyendo la lógica de lectura de input e impresión de output, el cuerpo de la solución final es:



Conclusión:

Como hemos visto, el algoritmo Knuth-Morris-Pratt’s presenta una eficiencia notable para la búsqueda de coincidencias de texto, y puede adaptarse de manera exitosa para resultar beneficiosa a la resolución de variedad de problemas de esta índole.