String Pattern Matching
INFO088 - Taller de Estructuras de Datos y Algoritmos
Tarea 1

Académicos: Mauricio Ruíz-Tagle, Héctor Ferrada.
Ayudante: Tomás Herceg.
Instituto de Informática, Universidad Austral de Chile.

Abril 29, 2022

Resumen

El objetivo de este trabajo es que usted adquiera un mayor dominio en la implementación de algunas estructuras de datos y algoritmos estudiados en clases, y resuelva de diversas maneras un problema algorítmico interesante. El problema a abordar es el llamado *String Pattern Matching Problem*. Para esto, diseñará tres soluciones al mismo problema, propondrá algunas hipótesis iniciales acerca del funcionamiento de cada solución, luego implementará cada solución para ejecutar experimentos sobre sus códigos, visualizando correctamente sus resultados con buenos gráficos. La primera solución que propondrá será un algoritmo *ingenuo*, llamado también de búsqueda exhaustiva. Luego construirá estructuras de datos adicionales y diseñará nuevos algoritmos de búsqueda que se espera que sean más eficientes que la solución preliminar. Finalizará su trabajo testeando experimentalmente el performance de las tres soluciones a fin de inferir buenas conclusiones en base a sus resultados que le permitan o no validar sus hipótesis iniciales.

Entrega. El trabajo tendrá una evaluación incremental compuesta de 4 entregables. En cada entrega debe de incluir el informe y solo en las entregas 3 y 4 debe adjuntar los códigos de sus implementaciones, además de su informe. La notas de las 4 entregas ponderan: 10 %, 20 %, 30 % y 40 % respectivamente, donde se espera que cada nueva entrega contenga, además del nuevo material, mejoras de lo entregado previamente.

Informe. El informe debe ser claro, objetivo y detallado, este debe contemplar: Resumen, Introducción, metodología, experimentación y conclusiones. Se debe introducir adecuadamente el contexto del problema, explicar la lógica que usó en sus implementaciones y la justificación de ello (p. ej. explicar por qué no uso otra alternativa más eficiente si la hubiese), detallar la experimentación realizada, incluir gráficas de sus resultados y dar buenas conclusiones de su trabajo; con todo esto debe validar o rechazar sus hipótesis iniciales justificando detalla y objetivamente.

Código Fuente. Todas las implementaciones deben ser en C++. Se exige un código ordenado, que este correctamente tabulado, que use variables con nombres nemotécnicos (que describan su uso), que sea modular e incluya los comentarios que sean necesarios para un buen entendimiento del código. No puede utilizar funciones para strings, los algoritmos de búsqueda y ordenamiento, o cualquier otro método que pueda necesitar para trabajar con secuencias de símbolos, los debe implementar usted mismo, y siempre trabajar con arreglos de char; por ejemplo, si necesita ordenar un conjunto de strings, puede adaptar un buen algoritmo de ordenamiento. Los códigos deben ir dentro de una carpeta .zip que incluya un Makefile. En la cabecera del fuente principal (donde está su rutina main) debe explicar como se ejecuta su código.

Etapa 1 (una semana de plazo). En esta entrega debe incluir: *i*- un primer **Resumen**, el cual debiese ir mejorando entrega a entrega; *ii*- la **Introducción**, donde se describe en detalle el problema a resolver y se entregan las **hipótesis iniciales** del trabajo que va a desarrollar, considerando los resultados que se esperan a priori en términos del performance empírico de cada solución. En este punto, considere que lo más importante es estimar a priori el tiempo de respuesta y el espacio requerido por cada solución.

Etapa 2. (una semana de plazo). Iteración de lo anterior más la **Metodología**. En su metodología debe entregar los pseudocódigos de los algoritmos que resuelven el problema. Se pide el procedimiento principal en cada caso, el que invocará para encontrar las ocurrencias del patrón que se está buscando en el texto (ver descripción del problema más abajo). Por tanto, para cada una de las tres soluciones pedidas, debe de describir una rutina que al menos reciba $T[0 \dots n-1]$ y $P[0 \dots m-1]$ como argumentos de entrada y retorne las ocurrencias de P en T. Explique cada algoritmo en su informe, analizando a priori, mediante nociones básicas de notación O (ver Appendice A), la eficiencia de los algoritmos descritos en sus pseudocódigos; de tal manera que le permitan **redefinir (de ser necesario) sus hipótesis iniciales**. Además, describa la traza o tabla de ejecución detallada para cada pseudocódigo tomando como ejemplo el texto y patrón de la Figura 1.

Etapa 3. (una semana de plazo). Iteración de lo anterior más una primera versión del código. En este debe de incluir: *i*- declaración de los procedimientos de búsqueda a utilizar; *ii*- la rutina *main*, bien ordenada; *iii*- la implementación completa de su solución de búsqueda exhaustiva —que ha propuesto con su propio algoritmo— y *iv*- experimentos representativos del performance de esta primera solución, incluya los gráficos que requiera para esto.

Etapa 4. (dos semanas de plazo). Entrega final del trabajo que agrega las implementaciones de las soluciones que utilizan estructuras de datos adicionales; se esperan buenos gráficos de resultados de las tres soluciones y mejores conclusiones aún, las cuales le permitirán validar o refutar sus hipótesis de investigación.

Envío del Trabajo. Debe enviar todo en un archivo, t1Alumnos.zip (grupos de 4 integrantes), con su implementación (que incluya el Makefile) y el informe final. Enviar al correo de su ayudante a cargo, tomas.herceg@alumnos.uach.cl, con asunto INFO088 TAREA 1 2022 —toda consulta debe canalizarla con el ayudante.

Descripción del Problema - String Pattern Matching

El problema a tratar en este trabajo es el de búsquedas de patrones en un texto fijo, conocido como String Pattern Matching [3, 2], el cual se define de la siguiente manera:
Dado un texto de entrada $T = \langle t_0 t_1 t_2 \dots t_{n-1} \rangle$ y un patrón de búsqueda $P = \langle p_0 p_1 p_2 \dots p_{m-1} \rangle$, con $m \leq n$, se desea encontrar todas las posiciones de T en donde comienza una ocurrencia de P. La figura 1 ilustra este proceso.

Como podrá imaginar hay muchas formas de buscar P en T, siendo la más básica la llamada **Búsqueda Exhaustiva**. Este método, para cada posición i de T, con $0 \leq i < n$, verifica si hay o no una ocurrencia de $P[0 \dots m-1]$ en $T[i \dots i+m-1]$; es decir, chequea todas las posibilidades.
Otra forma de buscar P es construir previamente alguna estructura de datos adicional. Luego, en tiempo de consulta, se utiliza la nueva estructura a fin de acelerar el tiempo de respuesta de una consulta. Algunas de estas estructuras se construyen en base a ordenar lexicográficamente los n sufijos del texto. La definición de un sufijo es la siguiente:

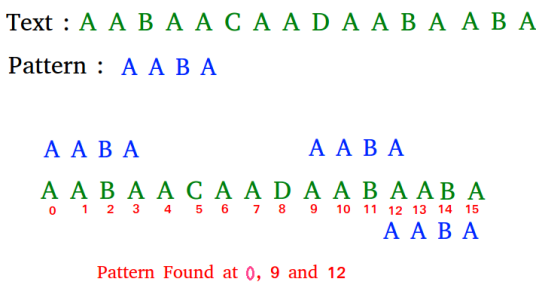

Figura 1: Ilustración del proceso de búsqueda del patrón $AABA$ en un texto de largo 16, arrojando como resultado las posiciones: $\{0, 9, 12\}$, que es donde encontramos el patrón.

Dado el texto de entrada $T = \langle t_0 t_1 \dots t_{n-1} \rangle$, **se define al sufijo de posición** i **de** T, con $0 \leq i < n$, como al string $S_i = t_i t_{i+1} \dots t_n$. Por ejemplo, para la imagen de la figura 1, tenemos que: $S_{10} = ABAABA$ y $S_5 = CAADAABAABA$. Como puede apreciar cada sufijo S_i es único en el texto, ya que todos son de largos diferentes.
Cuando hablamos de orden lexicográfico, nos referimos a una comparación carácter a carácter entre dos strings, la cual se hace de izquierda a derecha hasta encontrar un par de símbolos distintos y así determinar su orden. El orden de los símbolos es el dado por el típico código ASCII de 256 símbolos. Así, por ejemplo: *banana* $\prec$ *bandera* ya que la diferencia se produce en el cuarto carácter, donde $a \prec d$. Cuando un string es prefijo de otro, entonces el string de menor largo se considera lexicográficamente menor; por ejemplo: *casa* $\prec$ *casado*.
Imagine ahora que dispone de una estructura de datos adicional que almacena los índices de los n sufijos de T en orden lexicográfico; es decir, de menor a mayor. Por ejemplo, la siguiente tabla da el orden lexicográfico de los $n = 16$ sufijos del ejemplo dado en la figura 1:

S_i	String
S_{15}	A
S_{12}	$AABA$
S_9	$AABAABA$
S_0	$AABAACAADAABAABA$
S_3	$AACAADAABAABA$
S_6	$AADAABAABA$
S_{13}	ABA
S_{10}	$ABAABA$
S_1	$ABAACAADAABAABA$
S_4	$ACAADAABAABA$
S_{14}	BA
S_{11}	$BAABA$
S_2	$BAACAADAABAABA$
S_5	$CAADAABAABA$
S_8	$DAABAABA$

Cuadro 1: Los $n = 16$ sufijos del texto $T[0 \dots n-1] = AABAACAADAABAABA$ ordenados lexicográficamente de menor a mayor (de arriba a abajo en la tabla); el menor de ellos es S_{15} mientras que el mayor es S_8. Observe como S_0 corresponde al texto mismo, y que por ejemplo, S_{13} es prefijo de S_{10} y por tanto $S_{13} \prec S_{10}$.

De este modo, si usted dispone de alguna estructura de datos que almacene la permutación dada por la primera columna de la tabla 1: $\langle 15, 12, 9, 0, 3, 6, 13, 10, 1, 4, 14, 11, 2, 5, 8 \rangle$, dispone entonces del orden lexicográfico que existe entre los n sufijos del texto; y por tanto, puede utilizar esta información para

acelerar de alguna manera sus búsquedas.

Diseño de Soluciones

A continuación se describen las tres soluciones que se exigen para resolver el problema planteado. Para cada solución, proponga un algoritmo que encuentre todas las ocurrencias de P en T. Para cada propuesta, entregue el pseudocódigo de cada rutina de búsqueda y realice un análisis asintótico mediante notación O. De este modo, determine el tiempo asintótico de búsqueda en términos de los largos del texto y del patrón a buscar. Quizá desee añadir, como una cuarta solución, a algún algoritmo del estado del arte.
Se pide también que incluya algún análisis del espacio requerido por cada una de las soluciones pedidas. En cuanto a sus implementaciones, se espera que utilice los contenedores disponibles en la stl, como *vector* y *list*; pero cada vez que necesite resolver una tarea sobre los datos almacenados en los contenedores, debe de resolverlo algorítmicamente y no mediante las funciones que ya están disponibles en la stl; por ejemplo *sort* lo puede utilizar solo como medio de validación, para comprobar si su algoritmo de ordenamiento está bien o no, pero no puede ser parte de su desarrollo.

Solución 1. Búsqueda Exhaustiva

Proponga un algoritmo que analice todas las posibilidades de ocurrencia de P en T, y que no se apoye en ninguna estructura de datos adicional maś que texto mismo y el patrón.

Solución 2. Búsqueda Mediante la Lista Enlazada de Sufijos

En esta solución, debe construir una lista doblemente enlazada que contenga el orden lexicográfico de los n sufijos del texto, como se explicó con la tabla 1. En su algoritmo, debe utilizar la lista para acelerar de algún modo las búsquedas de patrones. Se espera que proponga un método que de algún modo aproveche el doble enlace de su lista.

Solución 3. Búsqueda Mediante el Arreglo de Sufijos

Construya ahora un arreglo que contenga los índices de los n sufijos del texto ordenados lexicográficamente. Con este arreglo, diseñe un algoritmo que busque eficientemente las ocurrencias del patrón.

Experimentación

Descargue los siguientes archivos para ejecutar sus experimentos:

- english.50MB.
- dna.50MB.

Utilice estos archivos para ejecutar experimentos con texto en inglés y con secuencias de ADN. Cada uno de estos archivos hace el papel del texto $T[0 \ldots n-1]$ de entrada al programa. Para cada T, construya sus estructuras, la lista doble y el arreglo, luego realice al menos los siguientes experimentos —desde luego puede agregar más experimentos si estima que es necesario para su análisis y presentación clara de los resultados.
Considere un parámetro REP, que equivale a la cantidad de repeticiones del experimento.

1. **Tiempo de búsqueda de patrones**. Para diferentes largos del patrón, $m \in \{2, 4, 8, 16, 32, 64, 128\}$, genere enteros aleatorios k que correspondan a posiciones aleatorias del texto, con $0 \leq k < n - m$. Para cada patrón $P[0 \ldots m-1] = T[k \ldots k+m-1]$ (para los REP k generados) busque con sus tres soluciones las ocurrencias de P en T. Valide que los resultados de las tres soluciones entregan los mismos índices y son correctos.
   Obtenga el tiempo promedio de CPU que tarda cada solución en encontrar un patrón de largo m en T.

2. **Espacio utilizado**. Determine el adicional utilizado por sus implementaciones, considerando la cantidad de memoria que requieran las estructuras auxiliares de las que dependan sus algoritmos.

Grafique adecuadamente los resultados de sus experimentos para luego **analizar y concluir sobre los eventos.** Comente también sobre la forma y las condiciones en las cuales llevó a cabo la experimentación, incluyendo detalles que permitan reproducir sus resultados.

Referencias

[1] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Data Structures and Algorithms*. Computer Science and Information Processing. Addison-Wesley, Reading, Massachusetts, 1st edition, January 1983.

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009.

[3] Rolf Fagerberg. *String Sorting*, pages 2117–2121. Springer New York, New York, NY, 2016.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[5] Robert Sedgewick. *Algorithms in C++, Parts 1-4: Fundamentals, Data Structure, Sorting, Searching, Third Edition*. Addison-Wesley Professional, third edition, 1998.

[6] Steven S. Skiena. *Sorting and Searching*, pages 103–144. Springer London, London, 2008.

A. Notación O grande

- ¿Sabe cuál es la complejidad del algoritmo quicksort o del de búsqueda binaria?

Si no sabe la respuesta, o si la sabe pero no ha comprendido como se obtiene dicho resultado, entremos en materia.

Cuando nos referimos al performance de un algoritmo se hace muy necesario tener alguna herramienta que nos permita medir que tan bueno es una solución algorítmica a un problema bien definido. La notación O[1] —que en ingles es llamada *Big O Notation* [1, 2, 4, 5, 6]— permite establecer una medida que da cuenta del rendimiento de un algoritmo en términos asintóticos. La notación O nos permite dar una nomenclatura o simbología a la complejidad de los algoritmos para medir el consumo de un recurso en particular; siendo el recurso más importante y estudiado, el tiempo de ejecución del algoritmo. Sin embargo, no estime como menos importantes otros recursos como el espacio o el consumo energético —más

[1] Un sencillo vídeo para una primera introducción https://www.youtube.com/watch?v=dyw0SohyEkw

aún hoy en día con todo el tema del *Big Data*. Para estos otros recursos también es posible determinar su complejidad con la notación O si ajustamos adecuamos el modelo de cómputo.

Tenga en cuenta que no nos estamos refiriendo al tiempo exacto de tardará un algoritmo en ser ejecutado. La notación O nos da las herramientas para un análisis teórico de un algoritmo y que no depende de la implementación en particular de este. El tiempo exacto depende de factores como: el hardware de la máquina en la que se ejecuten los experimentos, la dedicación exclusiva o no de esta en el momento de ejecución, del tamaño de los datos de entrada e incluso de la forma o característica de los datos de entrada en cada ejecución. Note la importancia de que esta medida es solo en términos asintóticos y no absolutos; es decir, nos dice acerca del compartimiento con entradas significativamente grandes. Una justificación de esto es que para escenarios pequeños, muchas veces, no tiene sentido o importancia saber cuál es el costo computacional de la solución. Si, por ejemplo, evaluamos a los algoritmos de ordenamiento, hemos visto en clases que *insertionSort*, que no tiene un tiempo óptimo asintóticamente, es de los mejores para entradas relativamente pequeñas o semi-ordenadas; y para entradas muy grandes, *quickSort* es de los más utilizados, incluso a pesar de ofrecer tiempo cuadrático (al igual que *insertionSort*) en su peor caso —resultado que se aleja muchísimo del óptimo $O(n \log n)$ para algoritmos de ordenamientos basados en comparaciones de sus elementos.
En resumen, *big O notation* es utilizado en ciencias computacionales para describir la complejidad del rendimiento de un algoritmo. Generalmente describe el peor escenario —que es lo que (al menos) se pide en este trabajo; es decir, el máximo trabajo que es proporcional al tiempo que tardará su ejecución en el peor de los casos posibles —esto es cuando la forma como viene la entrada obliga a nuestro algoritmo a trabajar y tardar el mayor tiempo que este puede alcanzar.

Ejemplos clásicos del uso de la notación O

Sea la entrada de un algoritmo de tamaño n elementos; por ejemplo, para un algoritmo de ordenamiento serán los n números a ordenar. Describimos algunos casos populares de Notación O grande:

- **Orden Constante:** $O(1)$.
  Esta expresión indica tiempo asintóticamente constante; es decir, **No depende del tamaño de la entrada** y su ejecución siempre tarda un tiempo que es prácticamente idéntico. Visto desde otro modo, la ejecución del algoritmo no se verá afectado por la cantidad de datos de entrada.
  Por ejemplo, almacenar el entero x en la posición i del arreglo de enteros $A[0 \ldots n-1]$, con $0 \leq i < n$; lo cual se realiza simplemente haciendo $A[i] = x$. Así, el tiempo que tarda no depende del tamaño del arreglo A y para cualquier valor de n tomará un pequeño tiempo constante muy parecido.

- **Orden Logarítmico:** $O(\log n)$.
  Esta expresión indica tiempo asintóticamente logarítmico; es decir, la ejecución siempre tarda un tiempo proporcional a $\log_b(n)$, para alguna base cualquiera b y para el n de entrada. En términos sencillos el tiempo que tarda el algoritmo se puede establecer como $T(n) = c \cdot \log_b n$, con $c > 0$ una constante y b una base válida. Visto de otro modo, indica que el tiempo $T(n)$ aumenta linealmente, mientras que n sube exponencialmente (con una base $b = 10$); lo que quiere decir que si se tarda 1 unidad de tiempo (t) con una entrada de 10 elementos, se necesitarán 2 t para 100, 3 t para 1000 y así sucesivamente.
  Por ejemplo, la típica búsqueda binaria es de tiempo logarítmico; ya que en el peor de los casos se realizarán $\log_2 n$ saltos en el arreglo para determinar si el dato buscado está o no en el arreglo ordenado (aquí $b = 2$).

- **Orden Lineal:** $O(n)$.
  Esta expresión indica tiempo asintóticamente lineal; es decir, la ejecución siempre tarda un tiempo proporcional al tamaño n de entrada. En términos sencillos el tiempo que tarda el algoritmo se puede establecer como $T(n) = c \cdot n$, para alguna constante $c > 0$. Por ejemplo, buscar el mínimo en un arreglo $A[0 \ldots n-1]$ se puede hacer con un simple recorrido de A, o más preciso, con un sencillo recorrido lineal del arreglo; ya que basta atravesarlo una sola vez de extremo a extremo.

- **Orden Cuadrático:** $O(n^2)$.
  Esta expresión indica tiempo asintóticamente cuadrático; es decir, la ejecución siempre tarda un tiempo proporcional al valor n^2 . En términos sencillos el tiempo que tarda el algoritmo se puede establecer como $T(n) = c \cdot n^2$, para alguna constante $c > 0$. Por ejemplo, si deseamos buscar cual es el valor más repetido en un arreglo $A[0 \ldots n-1]$, podríamos diseñar una solución que para cada valor $A[i]$, con $0 \leq i < n$, realice un recorrido lineal del arreglo para contar cuantas ocurrencias de $A[i]$ existen en A, y, para cada $x = A[i]$, nos vamos quedando siempre con el x para el cual encontremos una mayor cantidad de ocurrencias. A continuación el pseudocódigo de esta solución:

Input: un arreglo de n enteros $A[0 \ldots n-1]$
Output: u, el elemento más repetido en A

```
mas Repetido(A, n){
  u = A[0]
  occ = 0
  for i = 0 to n − 1 do{
    count = 0
    x = A[i]
    for j = 0 to n − 1 do
      if (A[j] == x) then
        count = count + 1
    if (count > occ) then{
        occ = count
        u = x
    }
  }
  return u
}
```