Strings

Leopoldo Taravilse
~~Gunther Frager~~

[1] Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp 2016

Contenidos

Tal vez me recuerden...

Tal vez me recuerden por clases como la del lunes pasado o la del viernes pasado.

Contenidos

Qué es String Matching?

Definición del problema

El problema de String Matching consiste en, dados dos strings S y T, con $|S| \leq |T|$, decidir si S es un substring de T, es decir, si existe un índice i tal que

$$S[0] = T[i], S[1] = T[i+1], \ldots, S[|S|-1] = T[i+|S|-1]$$

Solución Trivial

- Existe una solución $O(|S||T|)$ que consiste en evaluar cada substring de T de longitud $|S|$ y compararlo con S caracter por caracter.

Solución Trivial

- Existe una solución $O(|S||T|)$ que consiste en evaluar cada substring de T de longitud $|S|$ y compararlo con S caracter por caracter.
- Esta solución no reutiliza ningún tipo de información sobre S o sobre T.

Solución Trivial

- Existe una solución $O(|S||T|)$ que consiste en evaluar cada substring de T de longitud $|S|$ y compararlo con S caracter por caracter.
- Esta solución no reutiliza ningún tipo de información sobre S o sobre T.
- Existen soluciones que reutilizan información y así nos evitan tener que hacer $O(|S||T|)$ comparaciones.

Contenidos

Bordes de un String

Definición de borde

Un borde de un string S es un string B ($|B| < |S|$) que es a su vez prefijo y sufijo de S.

Bordes de un String

Definición de borde

Un borde de un string S es un string B ($|B| < |S|$) que es a su vez prefijo y sufijo de S.

Por ejemplo, *a* y *abra* son bordes de *abracadabra*.

Detección de bordes

- Un problema muy común es querer encontrar el borde más largo de un string (en realidad no es taaaan común que venga así solo, sino que se use para otras cosas, ya lo veremos más adelante).

Detección de bordes

- Un problema muy común es querer encontrar el borde más largo de un string (en realidad no es taaaan común que venga así solo, sino que se use para otras cosas, ya lo veremos más adelante).
- Nuevamente podríamos comparar cada prefijo con el sufijo correspondiente, lo que nos llevaría a una solución cuadrática.

Detección de bordes

- Un problema muy común es querer encontrar el borde más largo de un string (en realidad no es taaaan común que venga así solo, sino que se use para otras cosas, ya lo veremos más adelante).
- Nuevamente podríamos comparar cada prefijo con el sufijo correspondiente, lo que nos llevaría a una solución cuadrática.
- Existe una solución lineal para el cálculo del máximo borde de un string.

Detección de bordes

- Un problema muy común es querer encontrar el borde más largo de un string (en realidad no es taaaan común que venga así solo, sino que se use para otras cosas, ya lo veremos más adelante).
- Nuevamente podríamos comparar cada prefijo con el sufijo correspondiente, lo que nos llevaría a una solución cuadrática.
- Existe una solución lineal para el cálculo del máximo borde de un string.
- Esta solución se basa en encontrar el mayor borde de todos los prefijos del string uno por uno.

Detección de bordes

Lema 1

Si S' es borde de S y S'' es borde de S' entonces S'' es borde de S.
Al ser S'' prefijo de S' y S' prefijo de S, entonces S'' es prefijo de S, y análogamente es sufijo de S.

Detección de bordes

Lema 1

Si S' es borde de S y S'' es borde de S' entonces S'' es borde de S. Al ser S'' prefijo de S' y S' prefijo de S, entonces S'' es prefijo de S, y análogamente es sufijo de S.

Lema 2

Si S' y S'' son bordes de S y $|S''| < |S'|$, entonces S'' es borde de S'. Como S'' es prefijo de S y S' también, entonces S'' es prefijo de S'. Análogamente S'' es sufijo de S'.

Detección de bordes

Lema 1

Si S' es borde de S y S'' es borde de S' entonces S'' es borde de S. Al ser S'' prefijo de S' y S' prefijo de S, entonces S'' es prefijo de S, y análogamente es sufijo de S.

Lema 2

Si S' y S'' son bordes de S y $|S''| < |S'|$, entonces S'' es borde de S'. Como S'' es prefijo de S y S' también, entonces S'' es prefijo de S'. Análogamente S'' es sufijo de S'.

Lema 3

Si S' y S'' son bordes de S y el mayor borde de S' es S'', entonces S'' es el mayor borde de S de longitud menor a $|S'|$.

Solución lineal al problema de detección de bordes

- Empezamos con el prefijo de longitud 1. Su mayor borde tiene longitud 0. (Recordemos que no consideramos al string entero como su propio borde).

Solución lineal al problema de detección de bordes

- Empezamos con el prefijo de longitud 1. Su mayor borde tiene longitud 0. (Recordemos que no consideramos al string entero como su propio borde).
- A partir del prefijo de longitud 1, si al borde más largo del prefijo de longitud i le sacamos el último caracter, nos queda un borde del prefijo de longitud $i - 1$.

Solución lineal al problema de detección de bordes

- Empezamos con el prefijo de longitud 1. Su mayor borde tiene longitud 0. (Recordemos que no consideramos al string entero como su propio borde).
- A partir del prefijo de longitud 1, si al borde más largo del prefijo de longitud i le sacamos el último caracter, nos queda un borde del prefijo de longitud $i - 1$.
- Luego probamos con todos los bordes del prefijo de longitud $i - 1$ de mayor a menor, hasta que uno de esos bordes se pueda extender a un borde del prefijo de longitud i. Si ninguno se puede extender a un borde del prefijo de longitud i (ni siquiera el borde vacío), entonces el borde de dicho prefijo es vacío.

Algoritmo de detección de bordes

```
bordes[0] = -1 // El prefijo vacio no tiene borde
for i: 1 -> longitud(st)
    j = bordes[i-1]
    while(j>=0 and st[i-1] != st[j])
        j = bordes[j]
    bordes[i] = j+1
```

Este es el código del algoritmo de detección de bordes siendo *st* el string.

Algoritmo de detección de bordes

```
bordes[0] = -1 // El prefijo vacio no tiene borde
for i: 1 -> longitud(st)
    j = bordes[i-1]
    while(j>=0 and st[i-1] != st[j])
        j = bordes[j]
    bordes[i] = j+1
```

Este es el código del algoritmo de detección de bordes siendo *st* el string.
En bordes[*i*] queda guardada la longitud del máximo borde del prefijo de *st* de longitud *i*. Luego en bordes[*n*] queda guardada la longitud del máximo borde de *st* siendo *n* la longitud del string.

Correctitud del Algoritmo

```
while(j>=0 and st[i-1] != st[j])
    j = bordes[j]
```

En estas dos líneas comparamos el mayor borde del prefijo de longitud i con el mayor borde del prefijo de longitud $i - 1$. Si dicho borde no se puede extender, entonces probamos con el mayor borde de ese borde, y así sucesivamente.

```
bordes[i] = j+1
```

En esta línea extendemos el borde (si $j = -1$ es porque ni siquiera pudimos extender el prefijo vacío entonces tiene que quedar en 0, sino es porque podemos extenderlo y por eso el $+1$) y guardamos el borde en el arreglo bordes.

Complejidad del Algoritmo

Cada paso del for externo toma $O(p_i)$ donde p_i es la cantidad de operaciones del paso i, que es a lo sumo lo que se decrementa j más $O(1)$, y en la iteración i se decrementa a lo sumo $bordes[i-1] - bordes[i] + 1$ porque dentro del while cada paso decrementa j y si hacemos la suma telescópica (ahora explico en pizarrón lo que es una suma telescópica) la complejidad es lineal. La cuenta en pizarrón.

Marge

Homero, conozco un hombre que resolvió string matching en tiempo lineal.

Homero

¿Quién? ¿Dijkstra?

Marge

No! Es un científico.

Homero

Dijkstra es un científico!

Marge

Que no es Dijkstra!

Edsger Dijkstra

Este es Dijkstra.

Edsger Dijkstra

Este es Dijkstra.
Pero los que resolvieron String Matching en tiempo lineal fueron Knuth, Morris y Pratt.

Contenidos

String Matching

- Habíamos visto que existen soluciones más eficientes que $O(|S||T|)$ para el problema de String Matching.

String Matching

- Habíamos visto que existen soluciones más eficientes que $O(|S||T|)$ para el problema de String Matching.
- Knuth-Morris-Pratt (también conocido como KMP) es una de ellas y su complejidad es $O(|T|)$

String Matching

- Habíamos visto que existen soluciones más eficientes que $O(|S||T|)$ para el problema de String Matching.
- Knuth-Morris-Pratt (también conocido como KMP) es una de ellas y su complejidad es $O(|T|)$
- KMP se basa en una tabla muy parecida a la de bordes. La idea es que si el string viene matcheando y de repente no matchea, no empezamos de cero sino que empezamos del borde. Por ejemplo, si matcheó hasta *abracadabra* y luego no matchea, podemos ver qué pasa matcheando con el borde *abra*.

String matching (versión fuerza bruta)

```
string_matching():

    i = 0, j = 0
    while (i <= longitud(t) - longitud(s))
        if (j == longitud(s))
            reportar i
            i = i+1
            j = 0
        else if(s[j] != t[i+j])
            i++
            j=0
        else
            j++
```

String matching (versión KMP)

```
kmp():
    llenar_tablita_de_bordes()
    i = 0, j = 0
    while (i <= longitud(t) - longitud(s))
        if (j == longitud(s))
            reportar i
            i = i + j - bordes[j]
            j = bordes[j]
        else if(s[j] != t[i+j])
            i = i + max(1,j-bordes[j])
            j = max(0,bordes[j])
        else
            j++
```

String matching con bordes

Acabamos de ver que el problema de string matching se puede resolver con KMP.
Otra forma de resolver el problema de string matching en tiempo lineal es concatenando los dos strings $T + S$ y calculando los bordes.
Siempre que el borde sea mayor a $|S|$ y menor a $|T|$ quiere decir que hay un sufijo de $T + S$ (que tiene como sufijo a S) que también es prefijo de $T + S$, y por lo tanto es prefijo de T y tiene como sufijo a S, luego S es substring de T.

Contenidos

Qué es un Trie?

Definición de Trie

Los tries sirven para representar diccionarios de palabras. Un trie es un árbol de caracteres en el que cada camino de la raiz a un nodo final (no necesariamente una hoja) es una palabra de dicho diccionario.

Qué es un Trie?

Definición de Trie

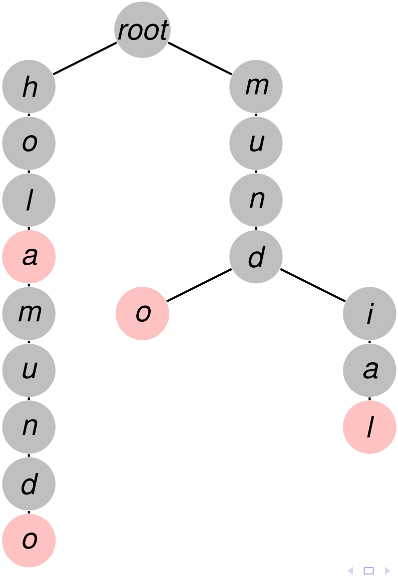
Los tries sirven para representar diccionarios de palabras. Un trie es un árbol de caracteres en el que cada camino de la raiz a un nodo final (no necesariamente una hoja) es una palabra de dicho diccionario.

Veamos un ejemplo de un Trie con las palabras *hola*, *holamundo*, *mundo* y *mundial*.

Edsger Dijkstra

Los tries no se los debemos a Dijkstra, pero antes del ejemplo siempre es bueno recordar a nuestro prócer.

Ejemplo de un Trie



Código del Trie

```
estructura trie
    diccionario
      (elementos del alfabeto -> enteros) siguientes
    bool final

trie t[mucho]
int n
inicializar:
    n = 1
    vaciar(siguientes(t[0]))
    final(t[0]) = false
```

mucho en este caso es una constante que determina el máximo tamaño del trie.

Pseudocódigo del Trie

```
insertar(string st):
    pos = 0
    for i: 0 -> longitud(st)-1
        if(noEstaDefinido(siguientes(pos),st[i]))
            siguientes(trie[pos],st[i]) = n
            vaciar(siguientes(trie[n]))
            final(t[n]) = false
            n++
        pos = siguiente(trie[pos],st[i])
    final(trie[pos]) = true
```

Problemas

- http://goo.gl/gQOSG
- http://goo.gl/KTVKd

Contenidos

Motivación

Problema

Dado un string calcular la cantidad de substrings distintos que tiene dicho string.

Motivación

Problema

Dado un string calcular la cantidad de substrings distintos que tiene dicho string.

Veremos a continuación dos algoritmos que nos sirven para resolver este problema eficientemente.

Sufijos de un string

- Muchas veces puede interesarnos ordenar los sufijos de un string lexicográficamente.

Sufijos de un string

- Muchas veces puede interesarnos ordenar los sufijos de un string lexicográficamente.
- En principio un string de longitud n tiene $n+1$ sufijos (contando el string completo y el sufijo vacío), y la suma de la cantidad de caracteres de todos esos sufijos es $O(n^2)$, por lo que tan sólo leer los sufijos para compararlos tomaría una cantidad de tiempo cuadrática en la cantidad de caracteres del string.

Sufijos de un string

- Muchas veces puede interesarnos ordenar los sufijos de un string lexicográficamente.
- En principio un string de longitud n tiene $n+1$ sufijos (contando el string completo y el sufijo vacío), y la suma de la cantidad de caracteres de todos esos sufijos es $O(n^2)$, por lo que tan sólo leer los sufijos para compararlos tomaría una cantidad de tiempo cuadrática en la cantidad de caracteres del string.
- Una forma de implementar Suffix Array en $O(n^2)$ es con un Trie. Insertando todos los sufijos y luego recorriendo el trie en orden lexicográfico podemos listar los sufijos en dicho orden.

Suffix Array

Qué es un Suffix Array?

A veces queremos tener ordenados lexicográficamente los sufijos de un string. Un Suffix Array es un arreglo que tiene los índices de las posiciones del string donde empiezan los sufijos, ordenados lexicográficamente.

Ejemplo de Suffix Array

Por ejemplo, para el string *abracadabra* el Suffix Array se obtiene de:

a	b	r	a	c	a	d	a	b	r	a
2	6	10	3	7	4	8	1	5	9	0

Y los sufijos ordenados son

```
a
abra
abracadabra
acadabra
adabra
bra
bracadabra
cadabra
dabra
ra
racadabra
```

Suffix Array

- Al igual que con KMP, el algoritmo que calcula el Suffix Array reutiliza información para ahorrar tiempo.

Suffix Array

- Al igual que con KMP, el algoritmo que calcula el Suffix Array reutiliza información para ahorrar tiempo.
- Si sabemos que *chau* viene antes que *hola*, entonces sabemos que *chaupibe* viene antes que *holapepe* y no necesitamos comparar *pibe* con *pepe*.

Suffix Array

- Al igual que con KMP, el algoritmo que calcula el Suffix Array reutiliza información para ahorrar tiempo.
- Si sabemos que *chau* viene antes que *hola*, entonces sabemos que *chaupibe* viene antes que *holapepe* y no necesitamos comparar *pibe* con *pepe*.
- Para saber que *chau* viene antes que *hola*, no tuvimos que comparar todo el string *chau* con el string *hola*, sino que sólo comparamos *ch* con *ho*, y para saber que *ch* viene antes que *ho* comparamos *c* con *h*.

Suffix Array

- Al igual que con KMP, el algoritmo que calcula el Suffix Array reutiliza información para ahorrar tiempo.
- Si sabemos que *chau* viene antes que *hola*, entonces sabemos que *chaupibe* viene antes que *holapepe* y no necesitamos comparar *pibe* con *pepe*.
- Para saber que *chau* viene antes que *hola*, no tuvimos que comparar todo el string *chau* con el string *hola*, sino que sólo comparamos *ch* con *ho*, y para saber que *ch* viene antes que *ho* comparamos *c* con *h*.
- La idea del Suffix Array pasa por ir comparando prefijos de los sufijos de longitud 2^t, e ir ordenando para cada t hasta que t sea mayor o igual que la longitud del string.

Suffix Array

- No vamos a dar código ni pseudocódigo

Suffix Array

- No vamos a dar código ni pseudocódigo
- Si queda tiempo vamos a contar un poco en pizarrón, sino queda como ejercicio!

Suffix Array

- No vamos a dar código ni pseudocódigo
- Si queda tiempo vamos a contar un poco en pizarrón, sino queda como ejercicio!
- Se puede armar un Suffix Array en $O(n)$ si el alfabeto es finito, o en $O(n \log n)$ en general, pero estas implementaciones son muy complicadas.

Suffix Array

- No vamos a dar código ni pseudocódigo
- Si queda tiempo vamos a contar un poco en pizarrón, sino queda como ejercicio!
- Se puede armar un Suffix Array en $O(n)$ si el alfabeto es finito, o en $O(n \log n)$ en general, pero estas implementaciones son muy complicadas.
- Las más standard son en $O(n \log^2 n)$ y sería muy raro que se requiera mejorarlas a una mejor complejidad en un problema de competencias.

Contenidos

LCP

Longest Common Prefix (LCP)

El Longest Common Prefix (LCP) entre dos strings es el prefijo común más largo que comparten ambos strings. Comunmente se conoce como LCP al problema que consiste en obtener el prefijo común más largo entre los pares de sufijos consecutivos lexicográficamente de un string. Para poder obtener los pares de sufijos consecutivos es necesario primero calcular el Suffix Array.

LCP

Longest Common Prefix (LCP)

El Longest Common Prefix (LCP) entre dos strings es el prefijo común más largo que comparten ambos strings. Comunmente se conoce como LCP al problema que consiste en obtener el prefijo común más largo entre los pares de sufijos consecutivos lexicográficamente de un string. Para poder obtener los pares de sufijos consecutivos es necesario primero calcular el Suffix Array.

Este problema puede ser resuelto en tiempo lineal.

Ejemplo de LCP

SA	S = abracadabra	LCP	
0	a	0	
1	abra	1	a
2	abracadabra	4	abra
3	acadabra	1	a
4	adabra	1	a
5	bra	0	
6	bracadabra	3	bra
7	cadabra	0	
8	dabra	0	
9	ra	0	
10	racadabra	2	ra

Cómo resolver LCP

Para resolver LCP la idea es que si por ejemplo el LCP de *hola* y *hongo* es 2, entonces para calcular el LCP de *chola* y *chongo* es 3, y sólo hace falta mirar un caracter.

Cómo resolver LCP

Para resolver LCP la idea es que si por ejemplo el LCP de *hola* y *hongo* es 2, entonces para calcular el LCP de *chola* y *chongo* es 3, y sólo hace falta mirar un caracter.

LA PRÓXIMA DIAPOSITIVA CONTIENE PSEUDOCÓDIGO QUE SOLO ESTÁ POR SI LO QUIEREN VER DESPUÉS, NO LO VAMOS A MOSTRAR EN CLASE

Pseudocódigo del LCP

```
llenar_tablita_del_LCP():
  n = longitud(s)
  tablita = int[n-1]
  q = 0
  for i: 0 ->n
    if bucket[i] != 0
    j = SuffixArray[bucket[i]-1]
      while(q+max(i,j) < n and st[i+q] == st[j+q])
        q++
      lcp[bucket[i]-1] = q
      if(q>0)
        q--
  return tablita
```

Pseudocódigo del LCP

bucket es el inverso del suffix array, lo que quiere decir que bucket[SuffixArray[i]] = i para todo i, y además el algoritmo que construye el Suffix Array utiliza los buckets para construirlo por lo que este arreglo viene gratis ya con el Suffix Array (y sino se construye en tiempo lineal).

Correctitud y complejidad del algoritmo de LCP

- Correctitud:
- Complejidad:

Correctitud y complejidad del algoritmo de LCP

- Correctitud: Queda como ejercicio.
- Complejidad: Queda como ejercicio.

Cantidad de substrings distintos

Recuerdan el problema que habíamos visto al principio de esta sección? La solución a este problema es con Suffix Array y LCP. La cantidad de substrings distintos es $\frac{n(n+1)}{2}$ menos la suma de los valores del LCP. ¿Porqué?

Cantidad de substrings distintos

Recuerdan el problema que habíamos visto al principio de esta sección? La solución a este problema es con Suffix Array y LCP. La cantidad de substrings distintos es $\frac{n(n+1)}{2}$ menos la suma de los valores del LCP. ¿Porqué? Queda como ejercicio

Contenidos

Lisa

"Ya sabemos resolver string matching"

Lisa Simpson

Lisa

"Pero ahora tenemos muchos strings!!"

Lisa Simpson

Buscando muchas agujas en un pajar

Generalizando String Matching

El problema de String Matching es dado un string S y un string T encontrar si S es substring de T y dónde aparece como substring. Este problema lo resolvemos con KMP construyendo una tablita para S en $O(|S|)$ y recorriendo T en $O(|T|)$. Si queremos buscar S en T_1 y T_2 corremos dos veces KMP y cuesta $O(|T_1| + |T_2|)$ ya que asumimos que $|S| \leq |T|$. Pero podemos hacer algo más inteligente que es construir una sóla vez la tablita, aunque la complejidad sigue siendo $O(|T_1| + |T_2|)$. Y lo mismo si en vez de 2 son muchos strings T.

Buscando muchas agujas en un pajar

String Matching con muchos strings

Si ahora los que son muchos son los strings $S_1, \ldots, S_n$ correr n veces KMP cuesta $O(n|T|)$, pero podemos bajarlo a $O(\sum_{i=1}^{n} |S_i| + |T|)$ gracias a Aho Corasick. La idea de Aho Corasick consiste en construir algo parecido a la tablita de KMP pero en vez de una tablita será un árbol en el cual se puede volver para un ancestro para no recorrer de nuevo desde la raiz para cada string.

"Cuando uno incluye muchas referencias a los Simpsons en sus diapositivas, se suele quedar sin tiempo, por lo que acá se termina la clase"

Roger Federer

Todo esto es...

Esto es...

Todo esto es...

Todo esto es...

Todo esto es...

Todo esto...

Todo esto es...

Todo esto es...

Todo esto es...

Todo es...

Todo esto es...

Esto es, todo...

Todo esto es...

Todo, esto, ese, todo eso es.

Todo esto es...

Éste todo

Todo esto es...

¡Oh!, ¿qué es esto?

Todo esto es...

Éste se, éste se, todo eso se, eso se tostó, se...

Todo esto es...

Ese seto es dos, dos tes, dos, eso es sed, esto es tos

Todo esto es...

Tose tose toto, o se destetó teté o es...

Todo esto es...

¡Ahh! ¡Esto es todo!

Todo esto es...