Strings

Leopoldo Taravilse

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp 2013

Contenidos

Contenidos

Que es String Matching?

Definición del problema

El problema de String Matching consiste en, dados dos strings S y T, con $|S| < |T|$, 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 comparar caracter por caracter con S.

Solución trivial

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

Solución trivial

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

Detección de bordes

- Un problema muy común es querer encontrar el borde más largo de un string.
- 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.
- 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.
- 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 máximo 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

```
1  int i=1,j=0;
2  bordes[0] = 0;
3  while(i<n)
4  {
5      while(j>0&&st[i]!=st[j])
6          j = bordes[j-1];
7      if(st[i]==st[j])
8          j++;
9      bordes[i++] = j;
10 }
```

Este es el código del algoritmo de detección de bordes siendo *st* el string y *n* su longitud.

Algoritmo de detección de bordes

```
1  int i=1,j=0;
2  bordes[0] = 0;
3  while(i<n)
4  {
5      while(j>0&&st[i]!=st[j])
6          j = bordes[j-1];
7      if(st[i]==st[j])
8          j++;
9      bordes[i++] = j;
10 }
```

Este es el código del algoritmo de detección de bordes siendo *st* el string y *n* su longitud.
En bordes[*i*] queda guardada la longitud del máximo borde del prefijo de *st* de longitud *i*. Luego en bordes[$n-1$] queda guardada la longitud del máximo borde de *st*.

Correctitud del algoritmo

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

En estas dos líneas comparamos el borde con el mayor borde (j) del prefijo de longitud $i-1$. Si dicho borde no se puede extender, entonces probamos con el mayor borde de ese borde (por Lema 3 esto es correcto).

```
1 if(st[i]==st[j])
2       j++;
3 bordes[i++] = j;
```

En estas líneas comparamos para ver si el borde efectivamente se puede extender (o es el borde vacío y no se puede extender) y guardamos el borde en el arreglo bordes.

Complejidad del algoritmo

En la línea

```
1 | j = bordes[j-1];
```

decrementamos *j*, y como lo aumentamos luego a lo sumo *n* veces, porque el ciclo externo se ejecuta a lo sumo *n* veces, luego esa línea se ejecuta a lo sumo *n* veces (ya que *j* nunca es menor que cero). Luego la complejidad del algoritmo es lineal en la longitud del string.

Contenidos

String Matching

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

String Matching

- Habíamos visto que existían 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 existían 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 que pasa matcheando con el *abra* del principio, ya que es un borde.

Código de KMP

```
1  string s,t;
2  void fill_table()
3  {
4      int pos = 2, cnd = 0;
5      kmp_table[0] = -1;
6      kmp_table[1] = 0;
7      while(pos<s.size())
8      {
9          if(s[pos-1]==s[cnd])
10             kmp_table[pos++] = ++cnd;
11         else if(cnd>0)
12             cnd = kmp_table[cnd];
13         else
14             kmp_table[pos++] = 0;
15     }
16 }
```

Así llenamos la tabla de KMP.

Código de KMP

```
1  int kmp(){
2      fill_table();
3      int m=0, i=0;
4      while(m+i<t.size()){
5          if(s[i] == t[m+i]){
6              if(i==s.size()-1)
7                  return m;
8              i++;
9          }
10         else{
11             m = m+i-kmp_table[i];
12             if(kmp_table[i]>-1)
13                 i = kmp_table[i];
14             else
15                 i = 0;
16         }
17     }
18     return -1;
19 }
```

Contenidos

Sufijos de un string

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

Sufijos de un string

- Muchas veces puede interesarnos ordenar los sufijos de un string lexicogáficamente.
- En principio, un string de longitud n tiene $n + 1$ sufijos (contando el string completo y el prefijo 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 tardaría al menos $o(n^2)$.

Sufijos de un string

- Muchas veces puede interesarnos ordenar los sufijos de un string lexicogáficamente.
- En principio, un string de longitud n tiene $n + 1$ sufijos (contando el string completo y el prefijo 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 tardaría al menos $o(n^2)$.
- Nuevamente, al igual que con KMP, podemos aprovechar información previamente calculada para no leer muchas veces un mismo caracter

Suffix Array

Que 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 donde empiezan los sufijos, ordenados de manera lexicográfica.

Suffix Array

Por ejemplo, para el string *abracadabra* el Suffix Array es:

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 necesitmaos 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 necesitmaos comparar *pibe* con *pepe*.
- Para saber que *chay* viene antes que *hola*, no tuvimos que comparar todo el string *chau* con el string *hola*, sólo comparamos *ch* con *ho* y sabemos que *chau* viene antes que *hola*, y para saber esto 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 necesitmaos comparar *pibe* con *pepe*.
- Para saber que *chay* viene antes que *hola*, no tuvimos que comparar todo el string *chau* con el string *hola*, sólo comparamos *ch* con *ho* y sabemos que *chau* viene antes que *hola*, y para saber esto comparamos *c* con *h*.
- La idea de Suffix Array pasa por ir comparando prefijos de los sufijos de longitud 2^t, e ir ordenando para cada t hasta que t se pase de la longitud del string.

Suffix Array

```
1  string st;
2  vector<int> sa[18];
3  vector<int> bucket[18];
4  int t,n;
5  void init(){
6      n = st.size();
7      for(int i=0;(1<<i)<2*n;i++){
8          sa[i].resize(n);
9          bucket[i].resize(n);
10         for(int j=0;j<n;j++)
11             sa[i][j] = j;
12     }
13     sort(sa[0].begin(),sa[0].end(),comp1);
14     initBuckets();
15     for(t=0;(1<<t)<n;t++){
16         sort(sa[t+1].begin(),sa[t+1].end(),comp2);
17         sortBuckets();
18     }
19 }
```

Suffix Array

```
1  bool comp1(int a, int b)
2  {
3      if(st[a]!=st[b])
4          return st[a]<st[b];
5      return a<b;
6  }
7  bool comp2(int a, int b)
8  {
9      if(bucket[t][a]!=bucket[t][b])
10         return bucket[t][a] < bucket[t][b];
11     int d = (1<<t);
12     if(a+d>=n)
13         return true;
14     if(b+d>=n)
15         return false;
16     if(bucket[t][a+d]!=bucket[t][b+d])
17         return bucket[t][a+d]<bucket[t][b+d];
18     return a<b;
19 }
```

Suffix Array

```
1  void initBuckets(){
2      bucket[0][sa[0][0]] = 0;
3      for(int i=1;i<n;i++)
4      if(st[sa[0][i]] == st[sa[0][i-1]])
5          bucket[0][sa[0][i]] = bucket[0][sa[0][i-1]];
6      else
7          bucket[0][sa[0][i]] = i;
8  }
9  void sortBuckets(){
10     bucket[t+1][sa[t+1][0]] = 0;
11     int d = (1<<t);
12     for(int i=1;i<n;i++)
13     if(bucket[t][sa[t+1][i]] == bucket[t][sa[t+1][i-1]]
14       && sa[t+1][i]+d < n && sa[t+1][i-1] < n
15       && bucket[t][sa[t+1][i]+d] == bucket[t][sa[t+1][i-1]+d])
16         bucket[t+1][sa[t+1][i]] = bucket[t+1][sa[t+1][i-1]];
17     else
18         bucket[t+1][sa[t+1][i]] = i;
19 }
```

Suffix Array

- Lo primero que hacemos es ordenar *sa*[0] segun el criterio de comparación comp1. Esto ordena a los sufijos por sus prefijos de longitud 1 (es decir por el caracter con el que empiezan).

Suffix Array

- Lo primero que hacemos es ordenar *sa*[0] segun el criterio de comparación comp1. Esto ordena a los sufijos por sus prefijos de longitud 1 (es decir por el caracter con el que empiezan).
- Luego acomodamos los buckets. Un bucket es como un paquete, sería en este caso un paquete de sufijos que hasta el momento son indistinguibles.

Suffix Array

- Lo primero que hacemos es ordenar $sa[0]$ segun el criterio de comparación comp1. Esto ordena a los sufijos por sus prefijos de longitud 1 (es decir por el caracter con el que empiezan).
- Luego acomodamos los buckets. Un bucket es como un paquete, sería en este caso un paquete de sufijos que hasta el momento son indistinguibles.
- Luego vamos ordenando los sufijos comparando sus prefijos de longitud 4, 8, 16... Notemos que cada uno de estos pasos es $n \log n$ ya que cada comparación entre dos sufijos es $o(1)$ y acomodar los buckets es lineal.

Suffix Array

- Lo primero que hacemos es ordenar $sa[0]$ segun el criterio de comparación comp1. Esto ordena a los sufijos por sus prefijos de longitud 1 (es decir por el caracter con el que empiezan).
- Luego acomodamos los buckets. Un bucket es como un paquete, sería en este caso un paquete de sufijos que hasta el momento son indistinguibles.
- Luego vamos ordenando los sufijos comparando sus prefijos de longitud 4, 8, 16... Notemos que cada uno de estos pasos es $n \log n$ ya que cada comparación entre dos sufijos es $o(1)$ y acomodar los buckets es lineal.
- Entonces podemos afirmar que el algoritmo tiene una complejidad de $o(n \log^2 n)$

Contenidos

LCP

Longest Common Prefix (LCP)

El Longest Common Prefix (LCP) entre dos strings es el prefijo común más largo entre ellos. Comunmente se conoce como LCP al problema que consiste en obtener los prefijos comunes más largos entre pares de sufijos consecutivos del Suffix Array.

LCP

Longest Common Prefix (LCP)

El Longest Common Prefix (LCP) entre dos strings es el prefijo común más largo entre ellos. Comunmente se conoce como LCP al problema que consiste en obtener los prefijos comunes más largos entre pares de sufijos consecutivos del Suffix Array.

Este problema puede ser resuelto en tiempo lineal.

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

LCP

```
1  vector<int> lcp;
2  void llenarLCP()
3  {
4      int n = st.size();
5      lcp.resize(n-1);
6      int q=0,j;
7      for(int i=0;i<n;i++)
8      if(bucket[t][i]!=0)
9      {
10         j = sa[t][bucket[t][i]-1];
11         while(q+max(i,j)<n && st[i+q]==st[j+q])
12             q++;
13         lcp[bucket[t][i]-1] = q;
14         if(q>0)
15             q--;
16     }
17 }
```

LCP

En bucket vamos a tener la posicion del i-ésimo sufijo en el Suffix Array, es decir $bucket[t][sa[t][i]] = i$ (el t es solo una cuestión de implementación), luego comenzamos por el primer sufijo. Si el sufijo que estamos viendo no es el primero lexicográficamente, nos fijamos cual es el anterior y vamos comparando caracter por caracter contando cuantos tienen en común. Si por ejemplo *abracadabra* tiene 4 caracteres en común con *abra*, entonces sabemos qeu *bracadabra* va a tener al menos 3 (4-1, o sea los 4 menos la *a* inicial) caracteres en comun con el anterior, ya que *bra* es anterior a *bracadabra*, de ahí viene la parte de

```
1 | if(q>0) q--;
```

y seguimos en el próximo paso con ese q, comparando de ahí en adelante.

Complejidad del LCP

El for corre n veces, y el while interno corre a lo sumo $2n$ veces, ya que el $q--$ se ejecuta a lo sumo n veces y q nunca es mayor a n. Luego el algorimo es lineal.