Strings

Leopoldo Taravilse[1]

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

Training Camp 2014

Contenidos

Contenidos

Qué es String Matching?

Definición del problema

El problema de String Matching consiste en, dados dos strigns 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 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.

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 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

```
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 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.

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

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

Complejidad del Algoritmo

El único lugar donde decrementamos *j* es en la línea

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

Además, *j*, que empieza inicializado en 0, se incrementa a lo sumo *n* veces, y nunca es menor que 0, por lo que decrementamos *j* a lo sumo *n* veces, luego la complejidad del algoritmo es lineal en el tamaño del string.

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*.

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 }
```

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.

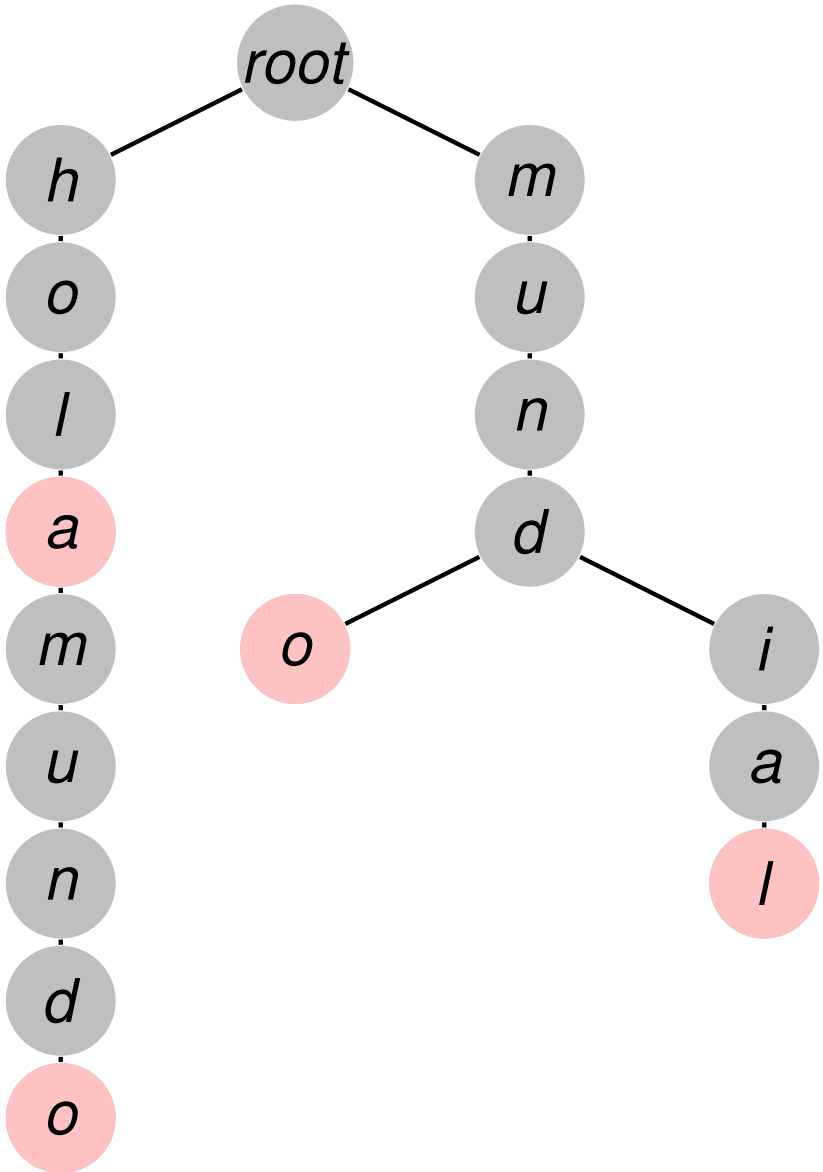
Qué es un Trie?

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*.

Ejemplo de un Trie



Código del Trie

```
1  struct trie{
2      map<char,int> sig;
3      bool final;
4  //puede ser map<int,int> si el alfabeto son enteros
5  };
6  trie t[MAXN];
7  int n;
8  void init()
9  {
10     n = 1;
11 t[0].sig.clear();
12 t[0].final = false;
13 }
```

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

Código del Trie

```
1  void insertar(string st){
2      int pos = 0;
3      for(int i=0;i<st.size();i++){
4          if(trie[pos].sig.find(st[i])==trie[pos].sig.end()){
5              trie[pos].sig[st[i]] = n;
6              trie[n].sig.clear();
7              trie[n].final = false;
8              n++;
9          }
10         pos = trie[pos].sig[st[i]];
11     }
12     trie[pos].final = true;
13 }
```

Ejemplo

Diseño de Camisetas

Dados dos equipos de rugby con n jugadores cada uno, quieren compartir las camisetas (un jugador de cada equipo por cada camiseta) de modo tal que cada camiseta tenga un prefijo común entre los apellidos de los dos jugadores que la usan (es válido el prefijo vacío), y entre todas las camisetas usen la mayor cantidad de letras posibles.

Problema D - TAP 2012. Link a la prueba: http://goo.gl/ypdYS

Diseño de Camisetas

La solución al problema consiste en:

- Probar que los dos jugadores (de distintos equipos) con el prefijo común más largo usan la misma camiseta. (Esta parte queda como ejercicio).

Diseño de Camisetas

La solución al problema consiste en:

- Probar que los dos jugadores (de distintos equipos) con el prefijo común más largo usan la misma camiseta. (Esta parte queda como ejercicio).
- Insertar todos los apellidos en un trie.

Diseño de Camisetas

La solución al problema consiste en:

- Probar que los dos jugadores (de distintos equipos) con el prefijo común más largo usan la misma camiseta. (Esta parte queda como ejercicio).
- Insertar todos los apellidos en un trie.
- Cada nodo del trie que es parte de a apellidos del equipo 1 y b apellidos del equipo 2 aporta $min(a, b)$ letras a las camisetas.

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 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 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.

Código del 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 }
```

Código del 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 }
```

Código del 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 }
```

Correctitud y Complejidad de Suffix Array

- Lo primero que hacemos es ordenar $sa[0]$ según el criterio de comparación $comp1$. Esto ordena a los sufijos según sus prefijos de longitud 1, es decir, el caracter con el que empiezan.

Correctitud y Complejidad de Suffix Array

- Lo primero que hacemos es ordenar $sa[0]$ según el criterio de comparación $comp1$. Esto ordena a los sufijos según sus prefijos de longitud 1, es decir, el caracter con el que empiezan.
- Luego dividimos a los sufijos en buckets. Un bucket es como un paquete. En este caso los buckets contienen los sufijos que hasta el momento son indistinguibles porque los prefijos por los cuales los comparamos son iguales.

Correctitud y Complejidad de Suffix Array

- Lo primero que hacemos es ordenar $sa[0]$ según el criterio de comparación $comp1$. Esto ordena a los sufijos según sus prefijos de longitud 1, es decir, el caracter con el que empiezan.
- Luego dividimos a los sufijos en buckets. Un bucket es como un paquete. En este caso los buckets contienen los sufijos que hasta el momento son indistinguibles porque los prefijos por los cuales los comparamos son iguales.
- Luego vamos ordenando los sufijos según sus prefijos de longitud 2, 4, 8,... y los ponemos en buckets según estos prefijos. Notemos que este paso es $O(n \log n)$ ya que la comparación entre dos sufijos según el prefijo que corresponda en este caso es $O(1)$.

Correctitud y Complejidad de Suffix Array

- Lo primero que hacemos es ordenar $sa[0]$ según el criterio de comparación $comp1$. Esto ordena a los sufijos según sus prefijos de longitud 1, es decir, el caracter con el que empiezan.
- Luego dividimos a los sufijos en buckets. Un bucket es como un paquete. En este caso los buckets contienen los sufijos que hasta el momento son indistinguibles porque los prefijos por los cuales los comparamos son iguales.
- Luego vamos ordenando los sufijos según sus prefijos de longitud 2, 4, 8,... y los ponemos en buckets según estos prefijos. Notemos que este paso es $O(n \log n)$ ya que la comparación entre dos sufijos según el prefijo que corresponda en este caso es $O(1)$.
- Podemos afirmar entonces que el algoritmo tiene una complejidad de $O(n \log^2 n)$.

Versiones más eficientes de Suffix Array

- Existen versiones más eficientes de Suffix Array, que se puede hacer en $O(n \log n)$.

Versiones más eficientes de Suffix Array

- Existen versiones más eficientes de Suffix Array, que se puede hacer en $O(n \log n)$.
- Para calcular el Suffix Array en $O(n \log n)$ se ordenan los caracteres del string en el primer paso usando counting sort. Luego ordenamos cada bucket por separado haciendo también counting sort.

Versiones más eficientes de Suffix Array

- Existen versiones más eficientes de Suffix Array, que se puede hacer en $O(n \log n)$.
- Para calcular el Suffix Array en $O(n \log n)$ se ordenan los caracteres del string en el primer paso usando counting sort. Luego ordenamos cada bucket por separado haciendo también counting sort.
- Incluso si el alfabeto no es acotado se puede mapear a un alfabeto acotado por la longitud del string utilizando sólo los caracteres que aparecen en el string para que el counting sort inicial sea $O(n)$. Luego los counting sort se pueden hacer también porque hay a lo sumo n buckets.

Versiones más eficientes de Suffix Array

- Existen versiones más eficientes de Suffix Array, que se puede hacer en $O(n \log n)$.
- Para calcular el Suffix Array en $O(n \log n)$ se ordenan los caracteres del string en el primer paso usando counting sort. Luego ordenamos cada bucket por separado haciendo también counting sort.
- Incluso si el alfabeto no es acotado se puede mapear a un alfabeto acotado por la longitud del string utilizando sólo los caracteres que aparecen en el string para que el counting sort inicial sea $O(n)$. Luego los counting sort se pueden hacer también porque hay a lo sumo n buckets.
- Esta optimización es bastante complicada y suele ser suficiente con la versión $O(n \log^2 n)$.

Contenidos

LCP

Longest Common Prefix (LCP)

El Longest Common Prefix (LCP) entre dos strings es el prefijo común más largo 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

Código del 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 }
```

Correctitud del algoritmo de LCP

En bucket vamos a tener la posición del i-ésimo sufijo en el Suffix Array, es decir $bucket[t][sa[t][i]] = i$ (el t viene de la implementación que dimos antes del Suffix Array). Para calcular el LCP comenzamos por el primer sufijo. Si el sufijo que estamos analizando no es el primero lexicográficamente, nos fijamos cuál es el sufijo anterior y vamos comparando caracter por caracter contando cuántos caracteres hay en común.

Correctitud del algoritmo de LCP

Si por ejemplo *abracadabra* tiene 4 caracteres en común con *abra*, entonces sabemos que *bracadabra* va a tener 3 caracteres en común con *bra*. Esto se ve reflejado en la línea

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

que decrementa q sólo en 1 en lugar de resetearlo a 0.
En la siguiente iteración arrancamos ya con q en 3 por lo que a partir de la segunda iteración del for ya no comparamos siempre todos los caracteres. Esto hace que el algoritmo sea más eficiente y una forma de convencerse de la correctitud de no resetear q a 0 es con el ejemplo de LCP(*abracadabra*,*abra*) = 4 $\Rightarrow$ LCP(*bracadabra*,*bra*) $\geq$ 3.

Complejidad del LCP

El for corre n veces, y el while interno corre a lo sumo $2n$ veces en total (entre todas las iteraciones del for), ya que q empieza en 0, se decrementa a lo sumo una vez por cada iteración del for, y nunca es mayor a n, luego el algoritmo es lineal.

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?

Contenidos

Qué es un palíndromo

Definición

Un palíndromo es un string que se lee igual de atrás para adelante y de adelante para atrás. Algunos ejemplos de palíndromos son Neuquen, Anitalavalatina o el apellido de un famoso ex presidente argentino.

Longest Palindromic Substring

Problema

Un problema muy común es el de buscar el palíndromo más largo que es substring de un string dado. Este problema tiene una solución lineal y es con el algoritmo de Manacher.

Algoritmo de Manacher

El algoritmo consiste en los siguientes pasos:

- Primero transformamos el string de la siguiente manera: Por ejemplo si el string es abracadabra lo transformamos en %#a#b#r#a#c#a#d#a#b#r#a#$. Es decir, le agregamos un # antes y después de cada letra, y dos caracteres distintos que no aparecen en el string uno al principio y otro al final.

Algoritmo de Manacher

El algoritmo consiste en los siguientes pasos:

- Primero transformamos el string de la siguiente manera: Por ejemplo si el string es abracadabra lo transformamos en %#a#b#r#a#c#a#d#a#b#r#a#$. Es decir, le agregamos un # antes y después de cada letra, y dos caracteres distintos que no aparecen en el string uno al principio y otro al final.
- En un arreglo P guardamos el mayor palíndromo de este nuevo string transformado centrado en cada posición del string.

Algoritmo de Manacher

El algoritmo consiste en los siguientes pasos:

- Primero transformamos el string de la siguiente manera: Por ejemplo si el string es abracadabra lo transformamos en %#a#b#r#a#c#a#d#a#b#r#a#$. Es decir, le agregamos un # antes y después de cada letra, y dos caracteres distintos que no aparecen en el string uno al principio y otro al final.
- En un arreglo P guardamos el mayor palíndromo de este nuevo string transformado centrado en cada posición del string.
- En una variable R guardamos donde termina el palíndromo que más adelante termina, y en una variable C guardamos el centro de dicho palíndromo.

Algoritmo de Manacher

```
1  vector<int> manacher(string S) {
2      int n = S.size();
3      vector<int> P(n,0);
4      int C = 0, R = 0;
5      for (int i = 1; i < n-1; i++) {
6          int j = C - (i-C);
7          if(R>i)
8              P[i] = min(R-i, P[j]);
9          while (S[i + 1 + P[i]] == S[i - 1 - P[i]])
10             P[i]++;
11         if (i + P[i] > R) {
12             C = i;
13             R = i + P[i];
14         }
15     }
16 }
```

Algoritmo de Manacher

```
1  vector<int> manacher(string S) {
2      int n = S.size();
3      vector<int> P(n,0);
4      int C = 0, R = 0;
5      for (int i = 1; i < n-1; i++) {
6          int j = C - (i-C);
7          if(R>i)
8              P[i] = min(R-i, P[j]);
9          while (S[i + 1 + P[i]] == S[i - 1 - P[i]])
10             P[i]++;
11         if (i + P[i] > R) {
12             C = i;
13             R = i + P[i];
14         }
15     }
16 }
```

Recordemos que S es el string transformado con la transformación previamente definida.

Complejidad del algoritmo de Manacher

- La complejidad del algoritmo es lineal en la longitud del string.

Complejidad del algoritmo de Manacher

- La complejidad del algoritmo es lineal en la longitud del string.
- Probar esto queda como ejercicio.