GUÍA DE APRENDIZAJE PARA CONCURSANTES ICPC Y IOI: ALGORITMO DE MANACHER

1. Introducción

Digamos que contamos con una cadena dada s con longitud n. Se nos pide encontrar todas las parejas (i, j) tal que subcadena $s[i \ldots j]$ es un palíndromo. Como podemos resolver este problema ?.

2. Conocimientos previos

2.1. Palíndromo

Un palíndromo (del griego palin dromos, *volver a ir atrás*), también llamado palíndroma o palindroma, es una palabra o frase que se lee igual en un sentido que en otro (por ejemplo; Ana, Anna, Otto). Si se trata de números en lugar de letras, se llama capicúa.

3. Desarrollo

En el peor de los casos, la cadena podría tener hasta $O(n^2)$ subcadenas palindrómicas, y a primera vista parece que no existe un algoritmo lineal para este problema.

Pero la información sobre los palíndromos se puede mantener de forma compacta : para cada posición i encontraremos el número de palíndromos no vacíos centrados en esta posición.

Los palíndromos con un centro común forman una cadena contigua, es decir si tenemos un palíndromo de longitud l centrado en i, también disponemos de palíndromos de longitudes $l-2$, $l-4$ y así sucesivamente también centrados en i. Por lo tanto, recopilaremos la información sobre todas las subcadenas palindrómicas de esta manera.

Los palíndromos de longitudes pares e impares se contabilizan por separado como $d_{impar}[i]$ y $d_{par}[i]$. Para los palíndromos de igual longitud asumimos que están centrados en la posición i si sus dos carecteres centrales son $s[i]$ y $s[i-1]$.

Por ejemplo, cadena $s = ababab c$ tiene tres palíndromos de longitud impar con centros en la posición $s[3] = b$, es decir $d_{impar}[3] = 3$:

$$a \overbrace{b\,a\,\underbrace{b}_{s_3}\,a\,b}^{d_{odd}[3]=3} c$$

y la cadena $s = cbaabd$ tiene dos palíndromos de igual longitud con centros en la posición $s[3] = a$, es decir $d_{par}[3] = 2$:

$$c \overbrace{b\,a\,\underbrace{a}_{s_3}\,b}^{d_{even}[3]=2} d$$

Es un hecho sorprendente que exista un algoritmo, que es bastante simple, que calcula estas *matrices de palindromidad* $d_{impar}[]$ y $d_{par}[]$ en tiempo lineal. El algoritmo se describe en este artículo.

3.1. Algoritmo trivial

Es el algoritmo que hace lo siguiente. Para cada posición central i trata de aumentar la respuesta en uno tanto como sea posible, comparando un par de caracteres correspondientes cada vez.

3.2. Algoritmo Manacher

Este algoritmo fue descubierto por Glenn K. Manacher en 1975. Describimos el algoritmo para encontrar todos los subpalíndromos con longitud impar, es decir, para calcular $d_{impar}[]$.

Para un cálculo rápido, mantendremos los bordes (l, r) del (sub) palíndromo más a la derecha encontrado (es decir, el (sub) palíndromo más a la derecha actual es $s[l+1]s[l+2]\dots s[r-1]$). Inicialmente establecemos $l = 0, r = 1$, que corresponde a la cadena vacía.

Entonces, queremos calcular $d_{impar}[i]$ para el siguiente i, y todos los valores anteriores en $d_{impar}[]$ ya han sido calculados. Hacemos lo siguiente:

- Si i está fuera del subpalíndromo actual, es decir $i \geq r$, simplemente lanzaremos el algoritmo trivial.

  Así que aumentaremos $d_{impar}[i]$ consecutivamente y verifique cada vez si la subcadena más a la derecha actual $[i - d_{impar}[i] \dots i + d_{impar}[i]]$ es un palíndromo. Cuando encontramos el primer desajuste o llegamos a los límites de s, pararemos. En este caso finalmente hemos calculado $d_{impar}[i]$. Después de esto, no debemos olvidar actualizar (l, r). r debe actualizarse de tal manera que represente el último índice del subpalíndromo más a la derecha actual.

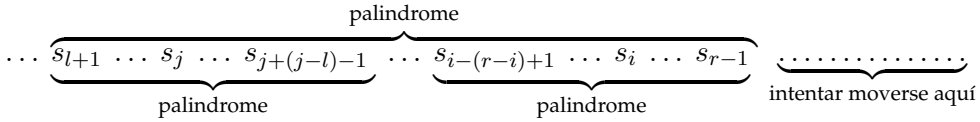
- Ahora considere el caso cuando $i \leq r$. Intentaremos extraer alguna información de los valores ya calculados en $d_{impar}[]$. Entonces, encontremos la posición de *espejo* de i en el subpalíndromo (l, r), es decir, obtendremos la posición $j = l + (r - i)$, y comprobamos el valor de $d_{impar}[j]$. Porque j es la posición simétrica a i con respecto a $(l + r)/2$, casi **siempre podemos** asignar $d_{impar}[i] = d_{impar}[j]$. Ilustración de esto (palíndromo alrededor j es en realidad *copiado* en el palíndromo alrededor i):

$$\dots \overbrace{s_{l+1} \dots \underbrace{s_{j-d_{odd}[j]+1} \dots s_j \dots s_{j+d_{odd}[j]-1}}_{\text{palindrome}} \dots \underbrace{s_{i-d_{odd}[j]+1} \dots s_i \dots s_{i+d_{odd}[j]-1}}_{\text{palindrome}} \dots s_{r-1}}^{\text{palindrome}} \dots$$

  Pero hay un caso difícil de manejar correctamente: cuando el palíndromo interior alcanza los bordes del exterior, es decir, $j - d_{impar}[j] \leq l$ (o, lo que es lo mismo, $i + d_{impar}[j] \geq r$). Debido a que la simetría fuera del palíndromo *exterior* no está garantizada, simplemente asignando $d_{impar}[i] = d_{impar}[j]$ será incorrecta: no disponemos de datos suficientes para afirmar que el palíndromo en la posición i tiene la misma longitud.

En realidad, deberíamos restringir la longitud de nuestro palíndromo por ahora, es decir, asignar $d_{impar}[i] = r - i$, para manejar tales situaciones correctamente. Después de esto, ejecutaremos el algoritmo trivial que intentará aumentar $d_{impar}[i]$ mientras sea posible.

Ilustración de este caso (el palíndromo con centro j está restringida para adaptarse al palíndromo exterior):



Se muestra en la ilustración que aunque el palíndromo con centro j podría ser más grande y salir del palíndromo exterior, pero con i como centro podemos usar solo la parte que encaja completamente en el palíndromo exterior. Pero la respuesta para la posición i ($d_{impar}[i]$) puede ser mucho más grande que esta parte, por lo que a continuación ejecutaremos nuestro algoritmo trivial que intentará hacer que crezca fuera de nuestro palíndromo externo, es decir, a la región *intenta moverte aquí*.

Nuevamente, no debemos olvidar actualizar los valores (l, r) después de calcular cada $d_{impar}[i]$.

Para calcular $d_{impar}[]$, obtenemos el siguiente código. Cosas a tener en cuenta:

- i es el índice de la letra central del palíndromo actual.
- Si i excede r, $d_{impar}[i]$ se inicializa a 0.
- Si i no excede r, $d_{impar}[i]$ se inicializa en el $d_{impar}[j]$, dónde j es la posición del espejo de i en (l, r), o $d_{impar}[i]$ está restringida al tamaño del palíndromo exterior.
- El ciclo while denota el algoritmo trivial. Lo lanzamos independientemente del valor de k.
- Si el tamaño del palíndromo centrado en i es x, entonces $d_{impar}[i]$ almacena $\frac{x+1}{2}$.

Aunque es posible implementar el algoritmo de Manacher para longitudes pares e impares por separado, la implementación de la versión para longitudes pares a menudo se considera más difícil, ya que es menos natural y conduce fácilmente a errores de uno por uno.

Para mitigar esto, es posible reducir todo el problema al caso cuando solo tratamos con los palíndromos de longitud impar. Para ello, podemos poner un # carácter adicional entre cada letra de la cadena y también al principio y al final de la cadena:

$$abcbcba \rightarrow \#a\#b\#c\#b\#c\#b\#a\#,$$

$$d = [1, 2, 1, 2, 1, 4, 1, 8, 1, 4, 1, 2, 1, 2, 1].$$

Como se puede ver, $d[2i] = 2d_{par}[i] + 1$ y $d[2i+1] = 2d_{impar}[i]$ dónde d denota la matriz de Manacher para palíndromos de longitud impar en # cadenas unidas, mientras que d_{impar} y d_{par} corresponden a las matrices definidas anteriormente en la cadena inicial.

De hecho, # los caracteres no afectan a los palíndromos de longitud impar, que todavía están centrados en los caracteres de la cadena inicial, pero ahora los palíndromos de longitud par de la cadena inicial son palíndromos de longitud impar de la nueva cadena centrada en caracteres #.

Tenga en cuenta que $d[2i]$ y $d[2i+1]$ son esencialmente los aumentados por 1 longitudes de los palíndromos de longitud par e impar más grandes centrados en i correspondientemente.

4. Implementación

4.1. C++

4.1.1. Algoritmo trivial

```
vector<int> manacher_odd(string s) {
  int n = s.size();
  s = "$" + s + "^";
  vector<int> p(n + 2);
  for(int i = 1; i <= n; i++) {
     while(s[i-p[i]]==s[i+p[i]])p[i]++;
  }
  return vector<int>(begin(p) + 1, end(p) - 1);
}
```

Los caracteres terminales \$ y $\wedge$ se usaron para evitar tratar los extremos de la cadena por separado.

4.1.2. Algoritmo Manacher

```
int manacher(string _text){
   int n=_text.size();
   int rad[2*n];
   int i,j,k;
   for(i=0,j=0;i<2*n;i+=k, j=max(j-k,0)){
      while(i-j>=0 && i+j+1<2*n && _text[(i-j)/2]==_text[(i+j+1)/2])
        ++j;
      rad[i]=j;
      for(k=1; i-k>=0 && rad[i]-k>=0 && rad[i-k]!=rad[i]-k ;++k)
         rad[i+k]=min(rad[i-k],rad[i]-k);
    }
    return *max_element(rad,rad+(2*n));
}
```

4.2. Java

4.2.1. Algoritmo Manacher

```
public int manacher(String _text){
  int n=_text.length();
  int [] rad=new int [2*n];
  int i,j,k;
  for(i=0,j=0;i<2*n;i+=k, j=Math.max(j-k,0)){
     while(i-j>=0 && i+j+1<2*n && _text.charAt((i-j)/2)==_text.charAt((i+j+1)
         /2))
       ++j;
     rad[i]=j;
     for(k=1; i-k>=0 && rad[i]-k>=0 && rad[i-k]!=rad[i]-k ;++k)
     rad[i+k]=Math.min(rad[i-k],rad[i]-k);
   }
   return Arrays.stream(rad).max().getAsInt();
}
```

5. Complejidad

Lo interesante del algoritmo Manacher es que lo realiza en un tiempo O(n) lo cual los hace muy eficiente. Mientras el mismo problema se puede resolver con Hashing se puede resolver en $O(n \cdot \log n)$, y con Suffix Trees y LCA rápido, este problema se puede resolver en $O(n)$. Pero el método descrito aquí es suficientemente más simple y tiene menos constante oculta en el tiempo y la complejidad de la memoria. En el caso del algoritmo trivial es lento, puede calcular la respuesta solo en $O(n^2)$.

A primera vista, no es obvio que el Manacher tenga una complejidad de tiempo lineal, porque a menudo ejecutamos el algoritmo ingenuo mientras buscamos la respuesta para una posición en particular. Sin embargo, un análisis más cuidadoso muestra que el algoritmo es lineal. Podemos notar que cada iteración del algoritmo trivial aumenta r por uno. También r no se puede disminuir durante el algoritmo. Entonces, el algoritmo trivial hará $O(n)$ iteraciones en total. Otras partes del algoritmo de Manacher funcionan obviamente en tiempo lineal. Así, obtenemos $O(n)$ complejidad del tiempo.

6. Aplicaciones

El algoritmo Manacher permite dentro de una cadena de carácteres hallar la longitud de la subsecuencia máxima concecutiva que es palíndrome. En caso de que queramos extraer la cadena se hace el siguiente el procedimiento:

- Se tiene rad[i] donde rad[i] es la longitud del palíndrome encontrado en la cadena y i la posición en el arreglo *rad*.

- Si i es par entonces el principio de la cadena palíndrome empieza en i/2 - rad[i]/2 y termina en la posición i/2 + rad[i]/2 de la cadena original.
- Si i es impar entonces el principio de la cadena palíndrome empieza en i/2 - rad[i]/2 +1 y termina en la posición i/2 + rad[i]/2 de la cadena original.

7. Ejercicios propuestos

A continuacion una lista de ejercicios que se pueden resolver aplicando este algoritmo:

- DMOJ - Palíndromo Máximo
- DMOJ - Haciendo Palíndromos
- DMOJ - Contando prefijos palíndromos