Olimpiada Peruana de Informática 2025 Fase 1 - Solucionario

Víctor Racsó Galván Oyola

18 de mayo de 2025

Índice

Problema 1: Inecuación no cuadrática	2
Problema 2: Segundo Problema	3
Problema 3: Cota Superior	9
Problema 4: Función de Bases	12

Tutorial: “Inecuación no cuadrática”

Author(s): Racsó Galván

Editorial by: Racsó Galván

Grupo 1 (Complejidad Esperada: $O(n \log n)$)

Para resolver este grupo nos basta con iterar sobre todos los $i = 1, \dots, n$, calcular en $O(\log n)$ la suma de dígitos de cada valor y contar cuántos de estos cumplen con la condición.

Grupo 2 (Complejidad Esperada: $O(m \log m)$)

Para resolver este grupo debemos notar la siguiente observación:

Observación 1.1. *La función $f(i)$ es no decreciente.*

Demostración. Considerando que $f(i) = i^2 - i \cdot s(i) = i(i - s(i))$, probaremos que $g(i) = i - s(i)$ es no decreciente.

Supongamos que los últimos $k \geq 0$ dígitos de i son 9, entonces se cumpliría que

$$s(i + 1) = s(i) - 9k + 1$$

Por lo tanto,

$$\begin{aligned} ((i + 1) - s(i + 1)) - (i - s(i)) &= (i + 1 - (s(i) - 9k + 1)) - (i - s(i)) \\ &= i + 1 - s(i) + 9k - 1 - i + s(i) \\ &= 9k \\ &\geq 0 \end{aligned}$$

Así que $g(i)$ es no decreciente y la función identidad es estrictamente creciente, así que la función $f(i) = i \cdot g(i)$ es no decreciente. $\square$

Gracias a lo anterior, podemos iterar hasta encontrar la primera posición x tal que $f(x) \geq m$ y la respuesta será $n - x + 1$.

Esto nos tomará aproximadamente $O(m \log m)$.

Grupo 2 (Complejidad Esperada: $O(\log^2 n)$)

Aprovechando 1.1, podemos usar búsqueda binaria sobre los valores de x para encontrar el menor x tal que $f(x) \geq m$. Para establecer límites que no requieran usar `__int128`, podemos notar que el máximo valor x tal que $f(x) \leq 10^{18}$ no excede a $10^9 + 3$, así que podemos hacer una búsqueda binaria sobre el rango $[1, \min\{n, 10^9 + 3\}]$ y obtendremos la respuesta en $O(\log^2 n)$.

Tutorial: “Segundo Problema”

Author(s): Racsó Galván

Editorial by: Racsó Galván

Grupo 1 (Complejidad Esperada: $O(nq)$)

Para resolver este grupo nos basta aplicar cada una de las operaciones en tiempo lineal, lo cual es trivial para el problema.

Grupo 2 (Complejidad Esperada: $O(q \log n)$ u $O(q\sqrt{n})$)

Para resolver este grupo debemos soportar solo las operaciones de consulta de máximo y adición en rango, así que podemos usar SQRT Decomposition o Segment Tree con Lazy propagation para obtener este puntaje en $O(q\sqrt{n})$ u $O(q \log n)$.

Idea con Segment Tree y Lazy propagation

Para cada nodo del Segment Tree de máximos mantendremos un entero extra que nos señale cuánto queda pendiente por modificar en los nodos hijos, de esta manera podemos cortar la recursión de actualización apenas el intervalo del nodo actual esté completamente incluido en el rango de modificación. Cuando tengamos que hacer recursión en cualquier función, tendremos que propagar la información pendiente de cada nodo a sus hijos previamente.

Ya que tanto la modificación como la consulta tomarían $O(\log n)$, la complejidad sería de $O(q \log n)$.

Idea con SQRT Decomposition y Lazy propagation

Vamos a particionar el arreglo en bloques de tamaño B de manera que cada bloque mantiene la información del máximo y un valor extra D tal que cada elemento x dentro del bloque representa al valor $x + D$.

Cuando tengamos que modificar un rango $[l, r]$, para un bloque incompleto podemos simplemente reconstruirlo en $O(B)$, mientras que para los bloques completamente dentro del rango nos bastará con sumar 1 a su D . Si usamos $B \approx \sqrt{n}$, tendremos una complejidad de $O(q\sqrt{n})$.

Grupo 3 (Complejidad Esperada: $O(n \log n)$ u $O(\sqrt{q} \cdot n \log n)$)

Para resolver este grupo debemos soportar solo las operaciones de consulta de máximo, adición en rango y shift cíclico, así que podemos usar SQRT Decomposition o Treap con Lazy propagation para obtener este puntaje en $O(q\sqrt{n})$ u $O(q \log n)$.

Podemos visualizar el shift cíclico como quitar el valor en la posición r e insertarlo en la posición l del arreglo.

Idea con Treap y Lazy propagation

Es sencillo notar que usando las operaciones típicas del Treap implícito $\text{SPLIT}(t, k, add)$ (devuelve un par de raíces de treap) y $\text{MERGE}(l, r)$ (devuelve una raíz de treap) podemos aplicar el siguiente algoritmo para el shift cíclico:

Algoritmo 1: SHIFT-CICLICO(T, l, r)

```
( $L_1, R_1$ )  $\leftarrow$  SPLIT( $T, r, 0$ ) ;  
( $L_2, R_2$ )  $\leftarrow$  SPLIT( $L_1, r - 1, 0$ ) ;  
( $L_3, R_3$ )  $\leftarrow$  SPLIT( $L_2, l, 0$ ) ;  
( $L_4, R_4$ )  $\leftarrow$  SPLIT( $L_3, l - 1, 0$ ) ;  
 $T \leftarrow$  MERGE( $L_4, R_2$ ) ;  
 $T \leftarrow$  MERGE( $T, R_3$ ) ;  
 $T \leftarrow$  MERGE( $T, R_4$ ) ;  
 $T \leftarrow$  MERGE( $T, R_1$ ) ;
```

devolver T

donde T es el nodo raíz del treap.

Además de ello, podemos usar una idea similar a la de segment tree sobre las funciones SPLIT(t, k, add) y MERGE(l, r) para sumar 1 y consultar en rango en $O(\log n)$ esperado.

Finalmente, la complejidad total sería de $O(q \log n)$.

Idea con SQRT Decomposition y Lazy propagation

Podemos considerar como una base la idea de SQRT Decomposition para el grupo 2 pero tendremos que considerar bloques de tamaño variable, así que mantendremos como información extra el tamaño del bloque para poder definir bien aquellos bloques completos e incompletos para cada consulta.

Ya que el shift cíclico solo afectará a lo mucho a dos bloques (el bloque al cual le quitamos la posición r y al cual le insertamos la posición l), uno podría pensar que tendría una complejidad natural de $O(B)$ para esta operación, lo cual es cierto para las primeras veces. A pesar de ello, si realizamos múltiples inserciones en un mismo bloque y luego realizamos consultas en las cuales este está incompleto, tendremos una complejidad muy grande, así que plantearemos que la estructura de bloques para todo el arreglo se reconstruirá cada $O(\sqrt{q})$ consultas, obteniendo una complejidad de $O(\sqrt{q} \cdot n \log n)$

Grupo 4 (Complejidad Esperada: $O(q\sqrt{n \log n})$)

Vamos a plantear la solución del problema usando SQRT Decomposition, iremos paso por paso definiendo el bloque y cómo soportar las operaciones de consulta de máximo, adición en rango y reseteo en rango.

Estructura interna de cada bloque

Ya que todos los elementos dentro de un bloque se pueden manejar como un todo, para resolver la operación del tipo 2 vamos a usar un entero extra add que implica que todos los valores x almacenados en el bloque en realidad representan a $x + add$. Así podemos aplicar la operación 2 en $O(1)$ si esta se debe aplicar a todo el bloque o sino en $O(build)$, donde $build$ es lo que nos tomará construir la información de un bloque.

Como tendremos a lo mucho B elementos en cada bloque, esto implica que tendremos a lo mucho B valores diferentes X_j , así que realizaremos un mapeo de cada uno de estos valores hacia un entero entre 0 y $B - 1$ (compresión de coordenadas), de manera que todas las posiciones que tengan dicho valor estén asociadas a su posición en X_j (0-indexed).

Además, como en la operación del tercer tipo solo modificaríamos el valor del máximo dentro del bloque, podemos mantener los valores X_j en una cola doble, para poder quitar el máximo del

final de la cola y (posiblemente) reinsertarlo al inicio con el valor de 0 luego de hacer los ajustes correspondientes en los valores asociados. Como resultado de este reajuste, podría suceder que nuevas posiciones estén asociadas al mismo valor, así que podríamos usar un DSU para mantener un representante de estas posiciones.

Notemos que la compresión de coordenadas de X_j nos toma al menos $O(B \log B)$.

Construcción del bloque

Considerando que la información de los bloques están en las siguientes variables globales. Usaremos $S = \lceil \frac{n}{B} \rceil$:

- Arreglo de S colas dobles *sorted_ids* — Para mantener los identificadores de los valores ordenados.
- Arreglo de S enteros *add* — Para la operación de tipo 2. Todo valor x almacenado en un bloque representa a $x + add[id]$.
- Arreglo de enteros de $S \times B$ *value_by_id* — Valores originales con los que fueron construidos los mapeos.
- Arreglo de enteros de $S \times B$ *rep* — Representante de clase de cada identificador de valor.
- Arreglo de n enteros *par* — Para el DSU interno de cada bloque. Las funciones definidas son *get(i)* para el representante del conjunto de i y *join(a, b)* para unir los conjuntos que contienen a a y b .
- Arreglo de n enteros *internal_id* — Identificador interno de valor asociado a cada posición.

Para construir un bloque, comprimiremos los valores diferentes para realizar el mapeo en $O(B \log B)$ por bloque (durante la asignación de identificadores podemos insertarlos en el *sorted_ids* del bloque), luego iremos asignando el identificador interno correspondiente a cada posición y uniendo en el DSU (previamente deberemos inicializarlo) las posiciones que corresponden al mismo, para ello podemos simplemente mantener la última posición que fue asociada al identificador actual y unir las posiciones. Esto tomaría $O(B\alpha(B))$ u $O(B \log B)$ según la implementación del DSU.

Resolviendo las operaciones

Vamos a denotar la función $id(i) = \lfloor \frac{i}{B} \rfloor$ para simplificar las expresiones usadas a partir de ahora.

Para resolver las operaciones, debemos plantear las siguientes propiedades. :

- El valor actual de la posición i es $add[id(i)] + value_by_id[id(i)][internal_id[get(i)]]$. Denotaremos la función *get_value(i)* como la que devuelve la expresión anterior.
- El máximo de un bloque b es igual a $add[b] + value_by_id[b][sorted_ids[b].back()]$. Denotaremos la función *get_block_max(b)* como la que devuelve la expresión anterior.

Consulta del máximo en rango

Podemos aplicar el clásico algoritmo de SQRT Decomposition y trabajar según corresponda.

Algoritmo 2: QUERY-MAX(l, r)

```
res ← 0 ;
mientras  $l \not\equiv 0 \pmod{B}$  AND  $l \leq r$  hacer
  | res ←  $\max\{res, get\_value(l)\}$  ;
  |  $l \leftarrow l + 1$  ;
mientras  $l + B - 1 \leq r$  hacer
  | res ←  $\max\{res, get\_block\_max(id(l))\}$   $l \leftarrow l + B$  ;
mientras  $l \leq r$  hacer
  | res ←  $\max\{res, get\_value(l)\}$  ;
  |  $l \leftarrow l + 1$  ;
devolver res
```

Así resolveremos la operación en $O(S)$.

Sumar 1 a un rango

Podemos aplicar el clásico algoritmo de SQRT Decomposition y trabajar según corresponda pero con los rangos que no son bloques completos simplemente reconstruiremos el bloque.

Algoritmo 3: ADD-ONE(l, r)

```
si  $id(l) = id(r)$  entonces
  | REBUILD-ADD( $id(l), l, r$ );
en otro caso
  |  $to \leftarrow l$ ;
  | mientras  $to \pmod{B} \neq 0$  hacer
  | |  $to \leftarrow to + 1$ ;
  | REBUILD-ADD( $id(l), l, to - 1$ );
  |  $l \leftarrow to$ ;
  | mientras  $l + B - 1 \leq r$  hacer
  | |  $add[id(l)] \leftarrow add[id(l)] + 1$ ;
  | |  $l \leftarrow l + B$ ;
  | si  $l \leq r$  entonces
  | | REBUILD-ADD( $id(l), l, r$ );
```

Donde REBUILD-ADD simplemente obtiene todos los valores del bloque usando $get_value(i)$, le agrega 1 a los $i \in [l, r]$ y construye el bloque completo como está descrito en la sección de construcción.

Reasignación de los máximos globales en un rango

Podemos aplicar el clásico algoritmo de SQRT Decomposition y trabajar según corresponda pero con los rangos que no son bloques completos simplemente reconstruiremos el bloque.

Algoritmo 4: RESET-RANGE(l, r, cur_max)

```
si  $id(l) = id(r)$  entonces
  si  $get\_block\_max(id(l)) = cur\_max$  entonces
    REBUILD-RESET( $id(l), l, r, cur\_max$ )
en otro caso
   $to \leftarrow l$ ;
  mientras  $to \bmod B \neq 0$  hacer
     $to \leftarrow to + 1$ ;
  si  $get\_block\_max(id(l)) = cur\_max$  entonces
    REBUILD-RESET( $id(l), l, to - 1, cur\_max$ )
   $l \leftarrow to$ ;
  mientras  $l + B - 1 \leq r$  hacer
    si  $get\_block\_max(id(l)) = cur\_max$  entonces
      SET-MAX-TO-ZERO( $id(l)$ )
     $l \leftarrow l + B$ ;
  si  $l \leq r$  and  $get\_block\_max(id(l)) = cur\_max$  entonces
    REBUILD-RESET( $id(l), l, r, cur\_max$ )
```

Donde REBUILD-RESET simplemente obtiene todos los valores del bloque usando $get_value(i)$, le asigna 0 a los $i \in [l, r]$ tales que su valor actual es igual a cur_max y construye el bloque completo como está descrito en la sección de construcción. Por otro lado, la función SET-MAX-TO-ZERO se vería de la siguiente forma:

Algoritmo 5: SET-MAX-TO-ZERO(id)

```
// Extraer el ID con el valor máximo
 $max\_value\_id \leftarrow BACK(sorted\_ids[id]);$ 
 $max\_value \leftarrow add[id] + value\_by\_id[id][max\_value\_id];$ 
POP-BACK( $sorted\_ids[id]$ );
si  $sorted\_ids[id]$  no está vacío entonces
  // Mirar el ID con el valor mínimo restante
   $min\_value\_id \leftarrow FRONT(sorted\_ids[id]);$ 
   $min\_value \leftarrow add[id] + value\_by\_id[id][min\_value\_id];$ 
  si  $min\_value = 0$  entonces
    // Unir representantes de min y max y redirigir a min_value_id
     $min\_rep \leftarrow rep[id][min\_value\_id];$ 
     $max\_rep \leftarrow rep[id][max\_value\_id];$ 
    JOIN( $min\_rep, max\_rep$ );
     $min\_rep \leftarrow GET(min\_rep);$ 
     $internal\_id[min\_rep] \leftarrow min\_value\_id;$ 
     $rep[id][min\_value\_id] \leftarrow min\_rep;$ 
    devolver;
// Reinsertar el ID máximo para que su valor aparente sea 0
PUSH-FRONT( $sorted\_ids[id], max\_value\_id$ );
 $value\_by\_id[id][max\_value\_id] \leftarrow -add[id];$ 
```

Notaremos que la complejidad por operación es:

1. $O(S)$.
2. $O(S + B \log B)$.
3. $O(S + B \log B)$.

Ya que la notación asintótica ignora techos o pisos, nuestra complejidad por consulta sería

$$T(\text{query}) = O(S + B \log B) = O\left(\frac{n}{B} + B \log B\right)$$

Si elegimos $B \approx \sqrt{\frac{n}{\log n}}$, tendremos una complejidad por consulta de

$$T(\text{query}) = O\left(\frac{n}{\sqrt{\frac{n}{\log n}}} + \sqrt{\frac{n}{\log n}} \log\left(\sqrt{\frac{n}{\log n}}\right)\right) \approx O(\sqrt{n \log n})$$

Lo cual es suficientemente rápido para el problema. Elegir $B \approx 131$ de manera fija también permite un rendimiento bueno en promedio.

Grupo 5 (Complejidad Esperada: $O(q\sqrt{n \log n})$)

Para resolver este grupo nos basta con combinar la idea de shift cíclico del grupo 3 usando SQRT Decomposition con la del grupo 4, tendríamos que manejar adecuadamente el tamaño del bloque para obtener una complejidad de $O(q\sqrt{n \log n})$.

Tutorial: “Cota Superior”

Author(s): Racsó Galván

Editorial by: Racsó Galván

Grupo 1 (Complejidad Esperada: $O(n^2)$ u $O(n\sqrt{n})$)

Para resolver este grupo nos basta con iterar sobre todos los $i = 1, \dots, n$, calcular sus divisores y obtener el menor con la máxima cantidad de divisores. Podemos calcular los divisores en $O(n)$ u $O(\sqrt{n})$, cualquiera puede pasar el tiempo límite.

Grupo 2 (Complejidad Esperada: $O(n \log n)$)

Para resolver este grupo podemos usar una criba de divisores para calcular cuántos divisores tiene cada número en el rango $[1, n]$. Luego podemos obtener el menor con máxima cantidad de divisores en $O(n)$.

Usando el siguiente algoritmo de procesamiento previo:

Algoritmo 6: CRIBA-DIVISORES(n)

Sea $cnt[1 \dots n]$ un arreglo de enteros inicializados en 0 ;

para $i \leftarrow 1$ **a** n **hacer**

```
     $j \leftarrow i$  ;  
    mientras  $j \leq n$  hacer  
         $cnt[j] \leftarrow cnt[j] + 1$  ;  
         $j \leftarrow j + i$ 
```

Obtendremos la respuesta en $O(n \log n)$.

Grupo 3 (Complejidad Esperada: $O(n)$)

Para resolver este grupo podemos modificar ligeramente la implementación de la criba lineal para que también calcule la cantidad de divisores de cada $i \in [1, n]$. Para ello, tendremos que mantener dos arreglos extra e y pot , donde e_i será el exponente del menor factor primo de i y pot_i será la potencia de dicho factor.

Podemos usar el siguiente algoritmo:

Algoritmo 7: CRIBA-ERATOSTENES(n)

```

Sea  $cnt[1 \dots n]$  un arreglo de enteros inicializados en 0 ;
Sea  $e[1 \dots n]$  un arreglo de enteros inicializados en 0 ;
Sea  $composite[1 \dots n]$  un arreglo de bits inicializados en 0 ;
Sea  $pot[1 \dots n]$  un arreglo de enteros ;
 $primos \leftarrow \emptyset$  ;
 $cnt[1] \leftarrow 1$  ;
para  $i \leftarrow 2$  a  $n$  hacer
    si  $composite[i] = 0$  entonces
         $cnt[i] \leftarrow 2$  ;
         $primos \leftarrow primos \cup \{i\}$  ;
         $pot[i] \leftarrow i$  ;
         $e[i] \leftarrow 1$ 
    para  $p \in primos$  hacer
        si  $i \cdot p > n$  entonces
            break ;
         $composite[i \cdot p] \leftarrow 1$  ;
         $cnt[i \cdot p] \leftarrow 2 \cdot cnt[i]$  ;
         $pot[i \cdot p] \leftarrow p$  ;
         $e[i \cdot p] \leftarrow 1$  ;
        si  $i \equiv 0 \pmod p$  entonces
             $e[i \cdot p] \leftarrow e[i] + 1$  ;
             $cnt[i \cdot p] \leftarrow (e[i \cdot p] + 1) \cdot cnt \left[ \frac{i}{pot[i]} \right]$  ;
             $pot[i \cdot p] \leftarrow p \cdot pot[i]$  ;
            break ;

```

Y obtener la respuesta en $O(n)$.

Grupo 4 (Complejidad Esperada: $O\left(\binom{74}{15}\right)$)

Para obtener el puntaje completo, debemos plantear algunas observaciones importantes:

Observación 3.1. Si x es la respuesta óptima y su factorización es de la forma $x = \prod_{i=1}^k p_i^{\alpha_i}$ con p_i ordenado de manera creciente, entonces se cumple que

$$\alpha_i \geq \alpha_{i+1}, \forall i = 1, \dots, k-1$$

Demostración. Supongamos que hay una respuesta óptima $x' = \prod_{i=1}^k p_i^{\alpha_i}$ y además que existe un par de posiciones $a < b$ tal que $\alpha_a < \alpha_b$.

Ya que $p_a < p_b$, esto implica que:

$$p_a^{\alpha_b - \alpha_a} < p_b^{\alpha_b - \alpha_a} \rightarrow \left(\frac{p_a}{p_b}\right)^{\alpha_b - \alpha_a} < 1$$

Si multiplicamos esta inecuación por n :

$$n \cdot \left(\frac{p_a}{p_b}\right)^{\alpha_b - \alpha_a} < n$$

$$m < n \text{ con } m = p_1^{\alpha_1} \dots p_a^{\alpha_a} \dots p_b^{\alpha_b} \dots p_k^{\alpha_k}$$

Y es sencillo notar que la cantidad de divisores de n y m son $\prod_{i=1}^k (\alpha_i + 1)$.

□

Además, notemos que siempre es óptimo usar los primeros X primos, donde X es la máxima cantidad de factores primos diferentes de cualquier entero menor o igual que n . Para los límites establecidos, $X = 15$.

Si usamos backtracking para generar todas las posibles configuraciones de α_i válidas (que no excedan a n en el producto acumulado), podremos obtener la respuesta con una complejidad de $O\left(\binom{74}{15}\right)$, ya que tendremos que elegir 15 valores posiblemente repetidos de un rango $[0, 59]$ (ya que $2^{59} \leq 10^{18}$ es el caso con mayor exponente posible).

A pesar de la complejidad tan grande, el rendimiento práctico debido al backtracking que ignora los casos inválidos, termina visitando a lo mucho 415878 estados.

Tutorial: “Función de Bases”

Author(s): Racsó Galván

Editorial by: Racsó Galván

Grupo 1 (Complejidad Esperada: $O(n^2 \log n)$ por caso)

Para resolver este grupo podemos iterar sobre todos los subarreglos de s , asociar cada base a una máscara de bits y usar un `std::map` para mantener la longitud más grande asociada a cada base. Esto tomará $O(n^2 \log n^2) = O(n^2 \log n)$ por cada caso.

Grupo 2 (Complejidad Esperada: $O(n|\Sigma| \log n)$ por caso)

Para resolver este grupo podemos usar el planteamiento de asociar cada base a una máscara de bits y construir un *Sparse Table* en $O(n \log n)$ de todas las bases cuyas longitudes sean potencias de 2, de manera que cuando consultemos un rango $[l, r]$ podemos simplemente tomar la unión de las bases de los rangos $[l, l + p - 1]$ y $[r - p + 1, r]$ donde p es la máxima potencia de 2 menor o igual a $r - l + 1$.

De esta manera podríamos consultar la base de cualquier rango en $O(1)$. Denotemos dicha función como $\text{QUERY}(l, r)$.

Ahora, una observación importante es que cuando fijamos el extremo izquierdo de un rango con l y variamos el extremo derecho r , podremos notar que los valores de las bases de $[l, r]$ a medida que r aumenta **solo pueden cambiar a lo mucho $|\Sigma|$ veces, donde Σ es el alfabeto**, esto debido a que los tamaños de dichas bases estarán ordenados de forma no decreciente, así que el peor caso es recorrer todos los tamaños desde 1 hasta $|\Sigma|$ usando búsqueda binaria para hallar cada cambio.

Podemos usar el siguiente algoritmo como subrutina de una iteración de $l = 1, \dots, |s|$:

Algoritmo 8: PROCESAR-BASES(l)

```
at ← l ;
mientras at ≤ |s| hacer
  lo ← at ;
  hi ← n ;
  cur_base ← QUERY(l, at) ;
  mientras lo < hi hacer
    mi ← lo + ⌈ $\frac{hi-lo}{2}$ ⌉ ;
    si QUERY(at, mi) = cur_base entonces
      lo ← mi
    en otro caso
      hi ← mi - 1
  REGISTRAR-LONGITUD(cur_base, lo - at + 1) ;
  at ← lo + 1 ;
```

donde $\text{REGISTRAR-LONGITUD}(base, len)$ maximiza la longitud actual almacenada para la base $base$ con len .

Esta idea toma $O(n|\Sigma| \log n)$ de complejidad temporal y $O(n \log n + \min\{2^{|\Sigma|}, n|\Sigma|\})$ de memoria, lo cual es suficiente para pasar el ML.

Grupo 3 (Complejidad Esperada: $O(n|\Sigma| \log |\Sigma|)$ por caso)

Para resolver este grupo podemos notar que los $O(|\Sigma|)$ cambios se pueden obtener calculando la siguiente ocurrencia de cada caracter con respecto a una posición l , de manera que si ordenamos dichas ocurrencias, podremos iterar sobre todas las bases en $O(|\Sigma|)$, así que es posible calcular las bases en $O(n|\Sigma| \log |\Sigma|)$. Sin embargo, esto requeriría un arreglo de tamaño $O(n|\Sigma|)$ para esa información; sin embargo, podemos procesar esos datos desde el final de la cadena hacia el inicio y así reduciremos la memoria a solo $O(|\Sigma|)$ para los cambios de base.

Algoritmo 9: PROCESAR-TODAS-LAS-BASES()

Sea $next[1 \dots |\Sigma|]$ un arreglo de enteros inicializado en $|s| + 1$;

Sea $pos[1 \dots |\Sigma|]$ un arreglo de enteros ;

para $i \leftarrow 1$ **a** $|\Sigma|$ **hacer**

$pos[i] \leftarrow i$;

para $i \leftarrow |s|$ **a** 1 **hacer**

$next[s[i]] \leftarrow i$;

 Ordenar pos de acuerdo a $next[pos[j]]$ no decreciente ;

$base \leftarrow \emptyset$;

para $j \leftarrow 1$ **a** $|\Sigma|$ **hacer**

$base \leftarrow base \cup \{pos[j]\}$ **si** $j = |\Sigma|$ **entonces**

$len \leftarrow n + 1 - i$;

en otro caso

$len \leftarrow next[pos[j + 1]] - i$;

si $j = |\Sigma|$ **o** $next[pos[j]] \neq next[pos[j + 1]]$ **entonces**

si $len > 0$ **entonces**

 REGISTRAR-LONGITUD($base, len$) ;

La complejidad temporal es suficiente para pasar en tiempo y si manejamos la memoria como se menciona podremos obtener el puntaje completo.