Proyecto Final

Julián Andrés Bermúdez Valderrama, Juan Camilo Lara Navarro
{ja.bermudez10, jc.lara10}@uniandes.edu.co
{201519648, }
Universidad de los Andes, Bogotá, Colombia

Problema B

1. Algoritmo de solución.

El problema se abordó mediante el análisis de la posible permutación de un número. Si una secuencia numérica está ordenada descendentemente de izquierda a derecha (orden lexicográfico descendente), no es posible obtener una permutación que produzca otra secuencia numérica mayor. Por ejemplo, con base en lo antes mencionado, no hay una permutación para la secuencia $[7, 6, 5, 4, 3, 2, 1]$.

De este modo, se busca un elemento perteneciente a la secuencia que no cumpla con la condición de orden lexicográfico descendente (OLD). Se realiza una comparación entre dos elementos consecutivos desde el final de la secuencia hasta su inicio, con el objetivo de encontrar el elemento que no cumple OLD. Entonces, teniendo a r como la secuencia (o el arreglo) de números, el cumplimiento de OLD se determina mediante el cumplimiento de $r[i] \geq r[i+1]$ para $0 \leq i \leq (|r| - 2)$; si $r[i] < r[i+1]$, el elemento $r[i]$ no cumple con el OLD, y hay posibilidad de permutar la secuencia para obtener una secuencia mayor.

Ahora se sabe qué, $r[i]$ da un indicio del elemento que debería permutarse. La permutación se realiza nuevamente recorriendo la secuencia desde el último elemento hasta el primero, y ahora se busca un elemento mayor a $r[i]$. Es decir, $r[j] > r[i]$ para $i < j \leq (|r| - 1)$. Una vez ubicados $r[i]$ y $r[j]$, se intercambia la posición de sus valores correspondientes, $r[i], r[j] \coloneqq r[j], r[i]$.

Aun así, el intercambio realizado no da como resultado la secuencia deseada. Se deben organizar los valores desde la posición $i + 1$ (inclusive) en adelante, de tal modo que la secuencia resultante sea exactamente la siguiente secuencia mayor respecto a la que se tiene por entrada.

- Explicación gráfica:

Si se tiene una entrada: $1\ 123542$, con $k = 1$ y $r = [1, 2, 3, 5, 4, 2]$, la k-ésima permutación es computada así:

Indice	0	1	2	3	4	5
Valor	1	2	3	5	4	2
						^

Indice	0	1	2	3	4	5
Valor	1	2	3	5	4	2
					^	

Indice	0	1	2	3	4	5
Valor	1	2	3	5	4	2
				^		

Indice	0	1	2	3	4	5
Valor	1	2	3	5	4	2
			^			

Indice	0	1	2	3	4	5
Valor	1	2	3	5	4	2
			^		^	

Indice	0	1	2	3	4	5
Valor	1	2	3	5	4	2
			^		^	

Indice	0	1	2	3	4	5
Valor	1	2	4	5	3	2
				^		^

Indice	0	1	2	3	4	5
Valor	1	2	4	2	3	5
					^^	

Indice	0	1	2	3	4	5
Valor	1	2	4	2	3	5
					^^	

Indice	0	1	2	3	4	5
Valor	1	2	4	2	3	5

Es así como la k-ésima $(k = 1)$ permutación mayor de $r = [1, 2, 3, 5, 4, 2]$, es $r_1 = [1, 2, 4, 2, 3, 5]$.

- Métodos:
    - Principales:

m1. Permutar

Ctx: $nums[0..n)$
Pre: $0 < |nums| < 10^{1000}$
Post: $permutar(nums)$

```
fun permutar(nums array [0, n) of String) ret nums : array [0, n) of String
      i := |nums| - 2;

      while (i >= 0 ^ int(nums[i]) >= int(nums[i + 1])) →
            i := i-1;
      elihw

      if (i >= 0) →
            int j := |nums| - 1;

            while (j >= 0 ^ int(nums[j]) <= int(nums[i])) →
                  j := j-1;
            elihw

            intercambiar(nums, i, j);
            reverso(nums, i + 1);
      fi

      ret nums;
```

Nota: Se asume la función $int()$ como una función propia del lenguaje de programación, la cual obtiene el valor numérico de cualquier valor perteneciente a $nums$.

m2. Intercambiar

Ctx: $nums[0..n)$
Pre: $true$

Post: $intercambiar(nums, i, j)$

```
fun intercambiar(nums array [0, n) of String, i: nat, j: nat) ret : void
      nums[i], nums[j] := nums[j], nums[i];

      ret;
```

m3. Reverso

Ctx: $nums[0..n)$
Pre: $i < inicio < (|nums| - 1)$
Post: $reverso(nums, inicio)$

```
fun reverso(nums array [0, n) of String, inicio: nat) ret : void
      i, j = inicio, |nums| - 1;

      while (i < j) →
            intercambiar(nums, i, j);
            i := i+1;
            j := j-1;
      elihw

      ret;
```

- Auxiliares:

m1. EsPermutable

Este método ahorra el llevar a cabo el cálculo hasta la k-ésima permutación, cuando en cierto punto de las n-ésima permutación ($n \leq k$) ya no es posible permutar la secuencia.

Ctx: $nums[0..n)$
Pre: $0 < |nums| < 10^{1000}$
Post: $esPermutable(nums)$

```
fun esPermutable(nums array [0, n) of String) ret esPermutable : bool
      i := |num|- 2;

      while (i >= 0 ^ int(nums[i]) >= int(nums[i + 1])) →
            i := i-1;
      elihw

      if (i < 0) →
            ret false;
      fi

      ret true;
```

Nota: Se asume la función $int()$ como una función propia del lenguaje de programación, la cual obtiene el valor numérico de cualquier valor perteneciente a $nums$.

- Alternativas de implementación:

Se pensó en usar programación dinámica, usando un arreglo como estructura contenedora de la respuesta. Esta implementación se descartó dado que se consideró que la creación de un arreglo de tamaño k que almacenara las k permutaciones, sería un mal gasto de espacio en el caso en que la secuencia de entrada tuviera h permutaciones en donde $h < k$.

Se pensó en usar un entero int/Integer/Long/BigInteger para almacenar la secuencia de entrada, pero dado el tamaño del problema el tipo de dato y los objetos no soportan la precondición ($0 < r < 10^{1000}$). Además, en el caso de BigInteger, puesto que es una estructura inmutable, la modificación de una instancia de este tipo implica el recálculo de ésta, en general los métodos de manipulación son algo engorrosos y no muy prácticos para el contexto del problema.

Finalmente, se optó por guardar la secuencia de entrada en forma de String, lo cual permitió fácil manipulación y cumplimiento de la restricción del tamaño del problema.

2. Análisis de complejidades espacial y temporal.

- Complejidad temporal

Para el cálculo de $O(x)$ se considera como constante: el acceso a arreglo y la función $int()$.

- Intercambiar

En el peor caso, se realiza una asignación.

Operación	Veces
Asignación (:=)	1

Complejidad: $O(1)$.

- Reverso

En el peor caso, con $i = 1$, se realizan $(n-1)+1$ asignaciones y $(n-1)$ intercambios, entonces se tiene $2(n-1)+1$.

Operación	Veces
Asignación (:=)	$(n-i)+1$
Intercambiar (fun)	$(n-i)$

Complejidad: $O(n)$.

- Permutar

En el peor caso, se puede realizar k permutaciones.

Operación	Veces
Asignación (:=)	$2n+2$
Comparación (>=, <=)	$4n+1$

Intercambiar (fun)	1
Reverso (fun)	$2(n-1)+1$

Complejidad: $O(n) = k(2n + 2 + 4n + 1 + 1 + 2(n-1) + 1)$

- Complejidad espacial

Dado que el intercambio y reorganización de la secuencia se hace directamente al arreglo k veces, $S(n) = O(n)$.

3. Comentarios finales.

Utilizando los casos de prueba suministrados en el documento y otros casos planteados por nosotros, nuestra solución demora tiempos aceptables en resolver el problema.

Una de las decisiones importantes fue definir la secuencia de entrada como una cadena de caracteres, puesto que de esta manera se facilitan cálculos y manipulación, además se cumple con la condición de tamaño de la entrada. También, el ahorro cálculos en permutaciones mediante el método $esPermutable$, el cual nosotros consideramos que reduce tiempo de ejecución.