**K - Missing LDAP**

**Reducción del Problema**

**Descripción en palabras:**  
Dado un nombre completo de estudiante con tres o cuatro palabras (nombres y apellidos), se debe encontrar el primer LDAP disponible que no haya sido usado, siguiendo un patrón de generación específico.

**Descripción en lenguaje matemático:**  
Sea el nombre N compuesto por n1, n2, ..., nk, con k igual a 3 o 4. Los apellidos son A1 igual a n(k-1) y A2 igual a nk. Los nombres son G igual a n1 hasta n(k-2). El LDAP se genera combinando letras de los nombres con los apellidos según reglas definidas, y se busca el menor Li tal que Li no pertenece al conjunto U, donde U es el conjunto de LDAP ya usados.

**Reducción de la Solución**  
**Descripción en palabras:**  
Construir sucesivos candidatos de LDAP a partir del nombre completo siguiendo las reglas dadas. Consultar si cada candidato ya está en uso, hasta encontrar uno que no lo esté.

**Descripción en lenguaje matemático:**  
 Generar los posibles LDAP en orden: L1, L2, ..., Ln. Para cada Li, consultar si Li pertenece al conjunto U. Retornar el primer Lj tal que Lj no pertenece a U, e imprimir "! Lj".

**3. Código Realizado y Análisis**

* Enlace(s) Código: <https://vjudge.net/solution/61083813/u9ELvlIEsTa5XKBttINm>

**4. Casos de Prueba**

**Caso 1:  
Entrada: ana pérez gómez  
 Justificación:** Solo un nombre y dos apellidos. La generación del LDAP es directa y rápida.

**Caso 2:**  
 **Entrada: juan carlos pérez lópez**  
 **Justificación**: Dos nombres y dos apellidos. Se prueba la inclusión de la primera letra del segundo nombre en el LDAP.

**Caso 3:**  
 **Entrada:** maria fernanda gonzález rodríguez  
 **Justificación**: Nombres más largos. Se generan múltiples combinaciones de LDAP hasta encontrar uno libre.

**Caso 4:**  
**Entrada**: jose andres camilo ramírez castro  
**Justificación**: Nombre completo con cuatro palabras. Se pone a prueba la generación avanzada y la lógica de expansión del LDAP.

**Caso 5:**  
 **Entrada**: nombre ya utilizado en casi todas sus combinaciones posibles, como "james david rodriguez rubiano"  
 **Justificación**: Se fuerza al algoritmo a generar muchas variaciones hasta hallar una libre, evaluando su eficiencia en el peor caso.

**5. Iteración en Caso de Solución Incorrecta (o explicación Solución Correcta)**

**Código incorrecto:** [**https://vjudge.net/solution/61031901/gGwYVR279XlaqdaHWIf5**](https://vjudge.net/solution/61031901/gGwYVR279XlaqdaHWIf5)

* **Descripción del error:**

En el código es que nunca se generan los casos en los que el prefijo del segundo apellido tenga cero letras en las etapas intermedias, y además el orden de generación no se ajusta al algoritmo descrito.

Esto es lo que debería pasar:

1. 1 letra de primer nombre + 1 letra de segundo nombre (si hay) + primer apellido completo + 1 letra de segundo apellido.
2. 2 letras de primer nombre + 1 letra de segundo nombre + primer apellido completo + **cero** letras del segundo apellido, antes de empezar a añadir de nuevo letras del segundo apellido

Esto es lo hace mi codigo:

Arranca siempre el bucle for letras\_segundo\_apellido in range(1, …) desde 1, por lo que:

1. **Nunca** prueba la variante con 0 letras de segundo apellido.
2. Genera primero la de 1 letra, luego la de 2 letras, cuando debería ser primero 0, luego 1, 2, … para cada tamaño de prefijo de primer nombre.

* **Proceso de depuración:**

Primero corrí el programa con un ejemplo sencillo y vi que nunca encontraba un LDAP libre aunque sabía que debería llegar al caso base. En ese momento me dije: ‘Algo pasa en la generación de candidatos’.

Entonces metí unos print justo antes de cada llamada a consultar\_ldap, mostrando cuántas letras del primer nombre y cuántas del segundo apellido estaba usando. Al ejecutar de nuevo, vi que el segundo apellido siempre empezaba tomando al menos una letra y jamás probaba la opción de cero letras, cuando en realidad el algoritmo pide que, al aumentar el prefijo del nombre, pruebes primero con cero letras de segundo apellido y después con uno, dos, etc.

Con esa pista, ajusté los bucles: para cada longitud de prefijo del nombre, primero probar con cero letras de segundo apellido, y luego iterar de 1 hasta el mínimo entre la longitud del nombre y del apellido.

Ejecuté todo, incluyendo casos límite (sin segundo nombre, apellidos muy cortos, nombres largos), y confirmé que ahora sí aparece el LDAP correcto. Por último, quité los prints de depuración para dejar el programa limpio y listo.

* **Solución corregida:**

<https://vjudge.net/solution/61045093/skkGiPzERQQoZjyuEVSp>

**6. Preguntas de Aprendizaje**

* **Temas aplicados:**
* Manipulación de cadenas
* Búsqueda binaria
* Las típicas: for, arrays, condicionales y así.
* **Dificultad de la implementación**: Aunque no lo hice a la primera fue bastante fácil, quizá lo hubiera hecho a la primera si hubiera entendido bien el enunciado desde el principio
* **Recursos utilizados**: Ninguno

**7. Feedback LLM**

* **Envío código LLM**: <https://vjudge.net/solution/61083870/3Yo7RV4iUGGh4yeDhFUV>
* **Comparación de su códigos:**

La diferencia clave es que el de LLM, al construir para cada “i” (número de letras del primer nombre) recorría siempre j desde 0 hasta el final del segundo apellido, sin respetar la regla de que cuando i = 1, hay que empezar con al menos una letra de apellido, y tampoco separaba el momento en que una vez usado todo el primer nombre el sufijo de apellido debe seguir creciendo más allá de i. Eso hace que probé combinaciones fuera de orden y nunca llegue al caso correcto. En cambio, el mio define para cada i un inicio\_j que vale 1 si i = 1 o 0 si i ≥ 2, genera primero hasta min(i, len(apellido2)), y luego, en una segunda fase, extiende el sufijo más allá de i cuando el apellido es más largo. Así consigue el orden exacto de candidatos que pide el enunciado y por eso funciona.   
  
Y obviamente como el mio funciona seria mejor.

* **Feedback de LLM**: Pida al LLM que evalúe y corrija su código:





