



1. Búsqueda y Ordenamiento

Para todos los ejercicios de esta guía, calcular el tiempo de ejecución en el peor caso de los algoritmos implementados.

Ejercicio 1. *Anagramas*

Una palabra es anagrama de otra si las dos tienen las mismas letras, con el mismo número de apariciones. Por ejemplo la palabra "delira" es anagrama de la palabra "lidera". Se pide implementar un programa que tome dos palabras con letras minúsculas de la 'a' a la 'z' y calcule si son anagramas o no siguiendo las siguientes técnicas:

- Usando ordenamiento** para ver si p_1 y p_2 son anagramas basta con ordenar los caracteres de cada una y comparar si el string resultante es el mismo.
- Usando números primos:**
 - Asignamos a cada letra del alfabeto un número primo distinto.
 - Convertimos a la palabra en número a través de multiplicar los valores asignados a cada letra.
 - Por el teorema fundamental del álgebra cada combinación de letras tiene un número que identifica unívocamente a todos los anagramas. Entonces alcanza con comparar si los números obtenidos son iguales.
- Pensar otro** método que garantice tiempo de ejecución de peor caso sea $O(|p_1| + |p_2|)$.

Ejercicio 2. Dada una secuencia de palabras (strings) de longitud n en donde cada palabra tiene a lo sumo 40 caracteres, escribir un programa que devuelva la secuencia ordenada según la longitud de cada string y ante una misma longitud, en el orden en que aparecía originalmente en la secuencia. El programa debe garantizar tiempo de ejecución de peor caso perteneciente a $O(n)$ recorriendo a lo sumo 1 vez a la secuencia original. Por ejemplo, `ordenarPorFrec({"hola", "esto", "es", "una", "prueba"})` debe devolver `{"es", "una", "hola", "esto", "prueba"}`













Ejercicio 3. Dados dos secuencias de enteros de tamaño n en donde cada una de ellas está ordenada. Escribir un programa que encuentre la mediana¹ de la secuencia combinada.

- Usando ordenamiento** luego de concatenar las dos secuencias. Por último calcular la mediana.
- Usando apareamiento** para combinar las dos secuencias. Por último calcular la mediana.
- Sin ordenar**, pensando otra técnica que garantice tiempo de ejecución de peor caso perteneciente a $O(\log(n))$.

¹[https://es.wikipedia.org/wiki/mediana_\(estadistica\)](https://es.wikipedia.org/wiki/mediana_(estadistica))

Ejercicio 4. El “ordenamiento natural” de strings es una forma de ordenar de manera alfabética pero considerando **números enteros** entre las palabras. Por ejemplo, si tenemos: `{'version19', 'version20', 'version111', 'version110'}`, ordenar de manera convencional devuelve `{'version110', 'version111', 'version19', 'version20'}`, en cambio, uno esperaría un orden que considere que la versión 19 es anterior a la versión 111.² En este caso solo vamos a considerar strings compuestos de letras de la 'a' a la 'z' y números. Para ver más ejemplos sobre ordenamiento natural, crear archivos en un mismo directorio y ver cómo los ordena sistema operativo como se muestra en la figura. Se pide entonces:

- a. Escribir un programa que determine cuando un `string` es menor a otro según el orden natural. Utilizar si es necesario el valor `ascii`³ de los caracteres del string convirtiendolos a int. Por ejemplo si queremos saber cuál es el valor `ascii` de la letra a hacemos `int('a')`. Además podemos convertir `string` a `int` usando la función `stoi`. Por ejemplo si tenemos el string `"45"` podemos transformarlo a int con `stoi("45")`.
- b. Escribir un programa `natSorted` que ordene una secuencia de `string` de manera natural.

Nombre
 Ejemplo00
 Ejemplo000
 Ejemplo0000
 Ejemplo1
 Ejemplo2
 Ejemplo12
 Ejemplo12a
 Ejemplo12a2
 Ejemplo12a12
 Ejemplo12a123
 Ejemplo124
 Ejemplos0000

²en este caso, ya que `1 = 1`, el algoritmo compara 9 contra 11.

³<https://es.wikipedia.org/wiki/ASCII>