

Obteniendo todas las posibles permutaciones
únicas de 7 letras o menos.

por Mauricio Cerón Medina

Introducción

La repetición se puede lograr escribiendo ciclos o bucles, como por ejemplo con `for` o con `while`. Otra forma de lograr la repetición es usando recursividad, la cual ocurre cuando una función se llama a si misma.

Podemos definir la recursividad de la siguiente forma:

Un problema que pueda ser definido en función de su tamaño, sea este N , pueda ser dividido en instancias más pequeñas ($< N$) del mismo problema y se conozca la solución explícita a las instancias más simples, lo que se conoce como casos base, se puede aplicar inducción sobre las llamadas más pequeñas y suponer que estas quedan resueltas.

Para que se entienda mejor a continuación se exponen algunos ejemplos:

- **Factorial(x : Entero):** Sea $N := x$ el tamaño del problema, podemos definir el problema de forma recurrente como $x * \text{Factorial}(x - 1)$; como el tamaño de $\text{Factorial}(x - 1)$ es menor que N podemos aplicar inducción por lo que disponemos del resultado. El caso base es el $\text{Factorial}(0)$ que es 1.
- **Ordenación por fusión(v : vector):** Sea $N := \text{tamaño}(v)$, podemos separar el vector en dos mitades. Estas dos mitades tienen tamaño $N/2$ por lo que por inducción podemos aplicar la ordenación en estos dos subproblemas. Una vez tenemos ambas mitades ordenadas simplemente debemos fusionarlas. El caso base es ordenar un vector de 0 elementos, que está trivialmente ordenado y no hay que hacer nada.

En estos ejemplos podemos observar como un problema se divide en varias (≥ 1) instancias del mismo problema, pero de tamaño menor gracias a lo cual se puede aplicar inducción, llegando a un punto donde se conoce el resultado (el caso base).

Definición del algoritmo

Tomamos una palabra de 3 letras, por ejemplo "ola". ¿Qué es lo que espero obtener? La lista de posibles permutaciones con esas 3 letras, es decir: ola, oal, lao, loa, aol, alo.

Como podemos darnos cuenta existe un patrón: Tomamos una letra de la palabra, la ponemos en la primera posición, y simplemente tomamos las dos letras restantes y les cambiamos el orden.

Siguiendo nuestro ejemplo quedaría de la siguiente forma tomamos la "o" como letra principal quedando "la", cuyas únicas combinaciones posibles son "la" y "al", después se agrega la letra principal. Realizamos la misma acción con las letras "l" y "a". Juntando todos los resultados parciales obtendremos la lista de palabras que buscamos.

En base a la definición anterior podemos analizar palabras de cualquier longitud haciendo uso de la recursividad.

¿Cuántas posibles combinaciones podemos obtener de un mayor número de letras posibles?

Una palabra de tres letras resultó en una lista de seis "palabras". ¿Cuántas resultaran de una de 4? Dado que lo que buscamos son permutaciones, la fórmula para saberlo es sencilla:

$$n!$$

Revisemos:

$$3! = 3 \times 2 \times 1 = 6$$

El resultado cuadra con la cantidad de combinaciones que obtuvimos. Por lo que con una extensión de 4 letras tendremos:

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

Aún no son demasiadas, pero ¿qué sucede si queremos obtener las posibles permutaciones de 7 letras, que es el número de fichas que se utiliza en el videojuego Apalabrados o en el juego de mesa Scrabble. Pues:

$$7! = 7 \times 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 5040$$

Es una cantidad bastante grande como para revisar cada una.

¿Pero qué pasa si tenemos letras repetidas? Obviamente habrá palabras repetidas, lo cual reduce nuestra lista. ¿A cuánto?

$$\frac{n!}{(a!b!c!\dots)}$$

En donde a es el número de veces que se repite una letra, b es el número de veces que se repite otra, c el número de veces que se repite otra y así sucesivamente.

Supongamos que tenemos las letras "atajada". La letra "a" se repite 4 veces, entonces, tendremos:

$$\frac{7!}{4!} = 7 \times 6 \times 5 = 210$$

Código Fuente

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-

def shuffle(source, partial, out):
    newPartial = partial
    newSource = ""
    if len(source) == 0:
        out.append(newPartial)
        return
    for i in range(len(source)):
        newPartial = partial + source[i:i + 1]
        newSource = source[0:i]
        if i < (len(source)+1):
            newSource = newSource + source[i+1:len(source)]
        shuffle(newSource, newPartial, out)

def generate(source):
    out = []
    partial = ""
    shuffle(source, partial, out)
    return out

word = raw_input("Letras a permutar: ")
mix = generate(word)

words = []
for i in range(len(mix)):
    if mix[i] not in words:
        words.append(mix[i])

for i in range(len(words)):
    print words[i]
```

Repasemos el código por partes para comprender la forma en que funciona.

```
word = raw_input("Letras a permutar: ")
mix = generate(word)
```

La parte inicial del programa y que es bastante simple. Solicita al usuario que ingrese las letras de las cuales desea obtener las permutaciones y las guarda en una variable del tipo string llamada word.

La siguiente línea llama a la función generate y almacena en una lista lo que esta devuelve. La función generate comprende las siguientes líneas:

```
def generate(source):  
    out = []  
    partial = ""
```

En la primera línea solo se define la función y los requerimientos que esta tiene, en este caso las letras iniciales.

Posteriormente generamos dos variables en blanco, la primera una lista llamada out que almacenará todas las permutaciones que se generarán y la segunda una variable de tipo string que formará las combinaciones parciales mientras se termina de obtener los resultados posibles.

```
    shuffle(source, partial, out)  
    return out
```

En esta línea llamamos a la función shuffle la cual realizará todas las permutaciones y nos devolverá la lista out con todas ellas. Finalmente devolvemos el contenido de la lista out como resultado de la función.

```
def shuffle(source, partial, out):  
    newPartial = partial  
    newSource = ""  
    if len(source) == 0:  
        out.append(newPartial)  
        return  
    for i in range(len(source)):  
        newPartial = partial + source[i:i + 1]  
        newSource = source[0:i]  
        if i < (len(source)-1):  
            newSource = newSource + source[i+1:len(source)]  
        shuffle(newSource, newPartial, out)
```

Es en esta función donde se realiza todo el proceso de las permutaciones haciendo uso de la recursividad. En las primeras líneas además de definir la función asignamos algunas variables de tipo string que utilizaremos de forma interna en la recursión.

```
if len(source) == 0:  
    out.append(newPartial)  
    return
```

Revisamos si la longitud de la cadena original (source) que estamos utilizando es igual a cero, de ser así agregamos el contenido del string newPartial a la lista out puesto que se trata de una de las permutaciones resultantes. Y devolvemos los valores que hemos obtenido. Será más fácil comprender esta parte una vez que veamos las siguientes líneas.

```
for i in range(len(source)):  
    newPartial = partial + source[i:i +1]  
    newSource = source[0:i]
```

Aquí iniciamos un ciclo for usando como rango la longitud de la cadena original que ingreso el usuario.

Concatenamos los valores de los string partial y del substring correspondiente a la posición [i] a la cadena newPartial. Posteriormente creamos una nueva cadena de origen (newSource) formada por el substring de source hasta la posición [i].

```
if i < (len(source)+1):  
    newSource = newSource + source[i+1:len(source)]
```

Si el valor de [i] es menor a la longitud de nuestra cadena source mas uno entonces concatena el valor que tenemos en newSource y el substring de nuestra cadena original a partir de la posición [i +1] (las letras restantes).

```
shuffle(newSource, newPartial, out)
```

En esta línea es donde se aplica la recursividad, al llamar a la función shuffle dentro de si misma, esto nos permite ir ejecutando el ciclo las veces que sea necesaria hasta que todas las permutaciones hayan sido agregadas a la lista de salida (out).

Como ya se mencionó, la función shuffle cambia los valores en la lista out al ser llamada desde la función generate la cual la devuelve finalmente al programa principal, almacenándola en la lista llamada mix.

```
mix = generate(word)
```

Ahora la lista `mix` contiene todas las permutaciones posibles, ahora solo resta asegurarnos de que, en caso de que hubiera letras repetidas, todas las permutaciones sean únicas y eliminar aquellas que se repiten.

```
words = []
for i in range(len(mix)):
    if mix[i] not in words:
        words.append(mix[i])
```

Para verificar esto creamos una nueva lista en blanco y haciendo uso de un ciclo revisamos si nuestra permutación se encuentra en la nueva lista, de no estarlo la agrega a la misma, así evitamos cualquier permutación repetida.

```
for i in range(len(words)):
    print words[i]
```

Finalmente imprimimos cada uno de los elementos de la lista con las permutaciones únicas a través de una iteración.

Las únicas líneas que nos faltan explicar son las de cabecera:

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

Estas líneas están contempladas dentro de las buenas prácticas de programación en python. La primera nos indica donde se encuentra instalado el interprete de python para que el archivo pueda ser ejecutable en cualquier sistema con base UNIX (Linux, Mac OS-X) sin necesidad de generar un binario como tal.

La segunda línea muestra la codificación en que debe ser leído el código del programa lo cual nos permite utilizar dentro del mismo algunos caracteres especiales como acentos y la letra ñ.

Pruebas del programa

Una vez revisado el código, podemos realizar algunos ejemplos para verificar el funcionamiento del mismo.

Comencemos por la palabra "ola", la cual sabemos nos debe generar un total de 6 permutaciones:

```
[Maw@Shigure ~]$ ./shuffle.py
Letras a permutar: ola
ola
oal
loa
lao
aol
alo
```

Esperábamos 6 combinaciones en específico y las obtuvimos correctamente. Ahora probemos con "hola"

```
[Maw@Shigure ~]$ ./shuffle.py
Letras a permutar: hola
hola
hoal
hloa
hlao
haol
halo
ohla
ohal
olha
olah
oahl
oalh
lhoa
lhao
loha
loah
laho
laoh
ahol
ahlo
aohl
aolh
alho
aloh
```


Deberíamos haber obtenido 24 resultados, veamos si es cierto, pero como contar una por una sería un poco tardado utilizaré una característica de la terminal bash.

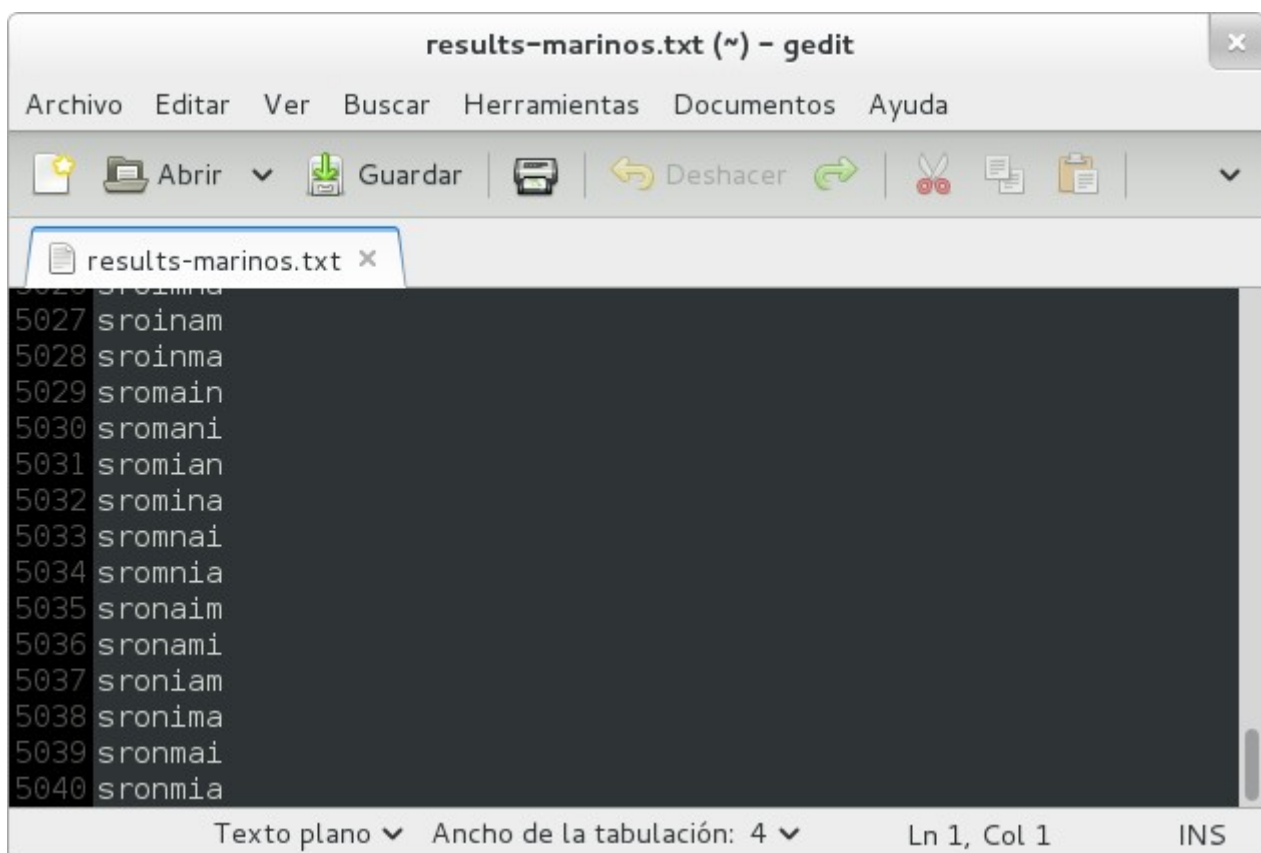
```
[Maw@Shigure ~]$ ./shuffle.py | wc -l  
hola  
24
```

Todo correcto hasta aquí, por lo tanto si ponemos algo como "marinos" que consta de 7 letras sin haber ninguna repetida deberíamos obtener 5040 posibles permutaciones.

```
[Maw@Shigure ~]$ ./shuffle.py | wc -l  
marinos  
5040
```

```
[Maw@Shigure ~]$ ./shuffle.py | sort > results-marinos.txt  
marinos
```

Y hemos obtenido todos los resultados en un archivo de texto que como podemos ver contiene 5040 permutaciones únicas.



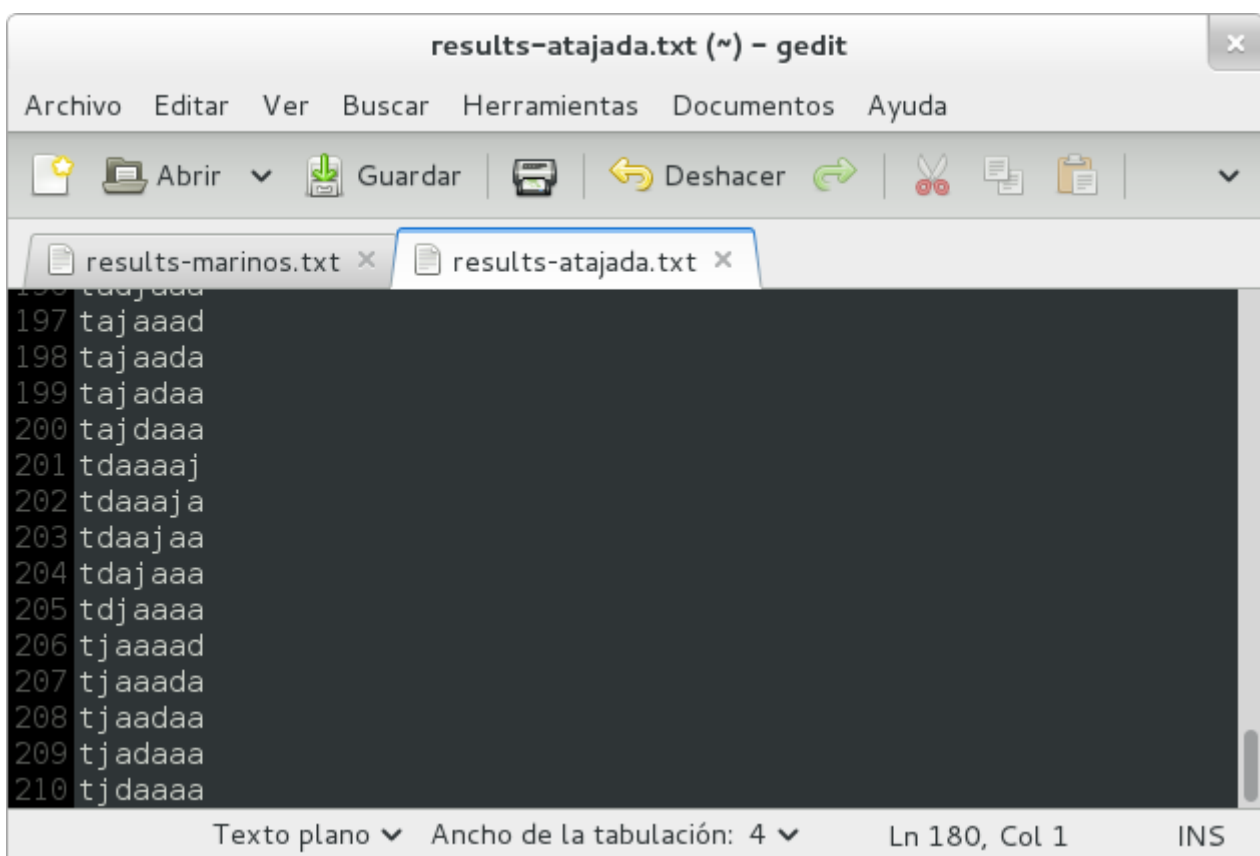
```
results-marinos.txt (~) - gedit  
Archivo  Editar  Ver  Buscar  Herramientas  Documentos  Ayuda  
Abrir  Guardar  Deshacer  
results-marinos.txt x  
5026 sroimna  
5027 sroinam  
5028 sroinma  
5029 sromain  
5030 sromani  
5031 sromian  
5032 sromina  
5033 sromnai  
5034 sromnia  
5035 sronaim  
5036 sronami  
5037 sroniam  
5038 sronima  
5039 sronmai  
5040 sronmia  
Texto plano  Ancho de la tabulación: 4  Ln 1, Col 1  INS
```

Por último utilicemos el ejemplo de “atajada” formada por 7 letras con una de ellas repetida 4 veces, por lo que deberíamos obtener 210 permutaciones.

```
[Maw@Shigure ~]$ ./shuffle.py | wc -l  
atajada  
210
```

```
[Maw@Shigure ~]$ ./shuffle.py | sort > results-atajada.txt  
atajada
```

Y una vez más se obtiene un archivo de texto, esta vez con 210 permutaciones únicas.



Notas finales:

- El lenguaje de programación utilizado para desarrollar este proyecto fue Python 2.7.3
- Esta documentación, así como el programa mismo se elaboraron en el sistema operativo Linux utilizando la distribución Fedora 17 "Beefy Miracle" distribuida por el Fedora Project.
- Se han anexado los archivos de texto que se generaron en las pruebas mencionadas ambos pueden encontrarse en la carpeta Anexos en la misma ubicación que este archivo.
- Se ha generado un archivo ejecutable para Windows, desgraciadamente con este no se pueden generar los archivos de texto o el conteo de resultados ya que son características propias de la terminal Linux.
- Este documento está liberado bajo una Licencia Creative Commons Attribution-Share Alike, por lo que puede copiarse, distribuirse y modificarse libremente siempre que se reconozca y de crédito al autor original "Mauricio Cerón Medina" y los trabajos derivados de este sean compartidos con las mismas libertades.
- La licencia del código fuente puede encontrarse en un archivo llamado LICENSE.txt en la carpeta "src"