Algoritmos Avanzados: Crear contraseñas de 8 caracteres mediante *Fuerza Bruta*

FELIPE IGNACIO MATURANA GUERRA[1]

[1] *Universidad de Santiago de Chile, Santiago, 2017.*
[*] *Correspondencia del autor: felipe.maturana.g@usach.cl*

Compilado March 26, 2017

Existe un sistema de seguridad el cual posee implementado la generación de contraseñas de acuerdo a combinaciones de letras y números de un largo de 8 caracteres en total, sin embargo, existen ciertos criterios que deben cumplir las contraseñas. La primera condición consiste en que dentro de una combinación no pueden existir 3 letras o 3 números de forma consecutiva, en segundo lugar si la contraseña comienza con un número, ésta no puede terminar con un número, por último, la contraseña no puede iniciar con una vocal. En esta experiencia se realizarán todas las combinaciones posibles (válidas e inválidas según los criterios anteriormente expuestos) para luego ser filtradas. Los caracteres a utilizar son el alfabeto, excluyendo la ñ, y los números del 0 al 9. Los códigos están implementados en Lenguaje C desarrollado en ambiente Linux, respetando el estándar ANSI C.

URL Github: El código fuente de la publicación y los códigos del programa se encuentran alojados en Git-Hub.

https://github.com/Felipez-Maturana/AlgoritmosAvanzados_L1_FuerzaBruta

1. INTRODUCCIÓN

En el mundo de las contraseñas, los niveles de seguridad cobran cada vez más importancia ya que nuestras distintas credenciales en las páginas web, redes sociales y nubes virtuales poseen información sensible de nuestras vidas privadas las cuales en las manos equivocadas pueden destruir la vida de una persona. Es por esto, que es necesario aumentar los niveles de seguridad de nuestras contraseñas, incluyendo una mayor cantidad de caracteres distintos o simplemente haciendo las contraseñas más extensas. En la criptografía y el mundo del hacking existen distintos métodos de generar y descifrar contraseñas mediante distintos tipos de ataques, uno de ellos es el **Ataque de Fuerza bruta**, el cual se estudiará en el presente informe, el cual consiste en crear todas y cada una de las combinaciones posibles, dada una lista de ***x caracteres*** y de un ***largo n***, originando a x^n combinaciones totales. Es por esto, que contraseñas que sólo utilicen dígitos numéricos ($x = 10$) son más fáciles de descifrar que otras que incluyan el alfabeto ($x = 25$). Sin embargo, **Fuerza bruta** al ser un método de generar todas las combinaciones posibles, además de ser de ensayo y error, es muy costoso en tiempo computacional. A lo largo de esta experiencia se detalla la solución de la implementación, además de comprobar si efectivamente el costo computacional es tan alto como señala la literatura.

2. DESCRIPCIÓN DEL PROBLEMA

La compañía Cuídate S.A crea sistemas de seguridad basadas en combinaciones de Letras y Números de un tamaño fijo de 8 caracteres los cuales deben cumplir ciertos criterios.

1. Las contraseñas generadas no pueden contener 3 letras consecutivas o 3 números consecutivos.

2. Si la contraseña termina con un número no puede terminar con un número, mientras que si empieza con letra puede terminar en número o letra.

3. La combinación no puede empezar por una vocal.

Además, se debe considerar que las letras a utilizar son todas minúsculas y los números corresponden a los dígitos, es decir, del 0 al 9. Para resolver este problema se debe utilizar el **lenguaje de programación C**, mediante la utilización de la técnica **Fuerza Bruta**.

A. Entrada

En un archivo de texto titulado "Entrada.in" se encuentran listados todas las letras y números a utilizar para formar las contraseñas.

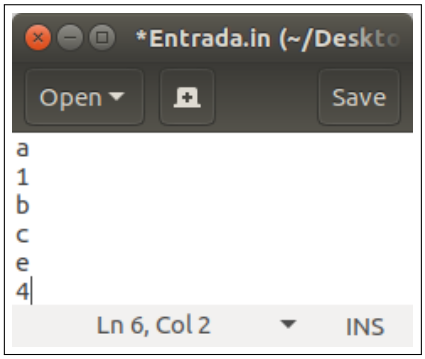


Fig. 1. Ejemplo del alfabeto ingresado en el archivo de entrada.

B. Salida

Al compilar el programa y ejecutarlo se debe generar un archivo de salida "Salida.out" el cual contiene las combinaciones válidas generadas (luego de aplicar los filtros mencionados anteriormente) enumeradas.

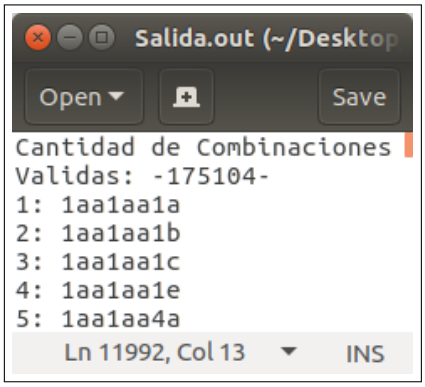


Fig. 2. Ejemplo del alfabeto ingresado en el archivo de entrada.

3. MARCO TEÓRICO

A. Algoritmo de Fuerza Bruta

En informática la búsqueda por **Fuerza Bruta**, también conocida como **Enumeración Explícita**, es una técnica de fácil implementación, que consiste en enumerar sistemáticamente[3] todos los posibles candidatos para la solución de cierto problema, para luego chequear si dicho candidato satisface las condiciones de dicha solución al problema, a este proceso se le llama filtro. Alguna de sus características principales son:

- El algoritmo es de fácil implementación.
- Es uno de los algoritmos más costosos en términos de cómputo a implementar.
- Consiste en probar todas las combinaciones posibles, de forma sistemática.
- Mientras más grande es el problema, más ineficiente se vuelve esta implementación.

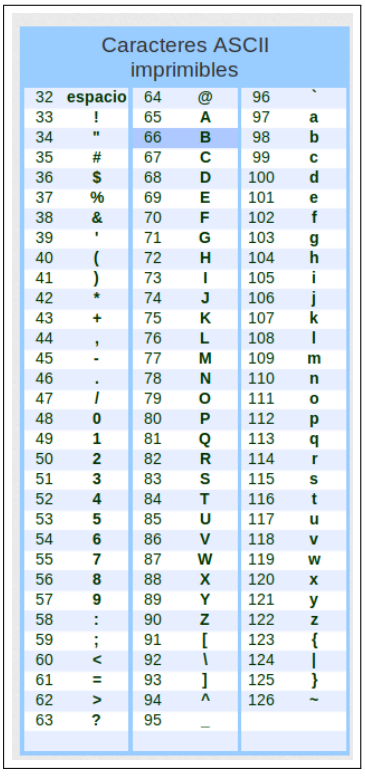
4. DESCRIPCIÓN DE LA SOLUCIÓN

La solución del problema se puede enumerar de la siguiente manera.

1. Se lee el archivo de entrada: ***Entrada.in*** y se guardan los caracteres en ***lista caracteres***. $O(n) = n$.
2. Se realizan todas las las combinaciones de contraseñas de un largo de 8 caracteres que se pueden realizar con la lista de caracteres y cada combinación se guarda en una ***lista de lista combinaciones*** (en la implementación resulta ser un puntero a lista de tamaño dinámico). $O(n) = n^8$.
3. Se recorre cada una de las combinaciones en la lista de combinaciones y son sometidos a un filtro según las 3 condiciones especificadas en el problema, cada combinación que cumpla de forma satisfactoria la función filtro es agregada a otra ***lista de listas*** llamada ***Combinaciones Válidas***.$O(n) = n^2$.
4. Finalmente se recorre cada una de las combinaciones en la ***lista de listas Combinaciones Válidas*** y se imprimen en el archivo de salida: ***Salida.out***. $O(n) = n^2$.

A. Diseño de la solución

Para efectos de este laboratorio el enunciado solicita que los caracteres permitidos a utilizar sean letras minúsculas y números del 0 al 9 (dígitos) es por esto que se elige utilizar la representación ASCII de los caracteres en la implementación en el lenguaje de programación C, donde un caracter es interpretado internamente como un entero según su representación en el *código estándar Estadounidense para el intercambio de información*, el cual posee una serie de caracteres imprimibles según la figura 3.



Caracteres ASCII imprimibles

32	espacio	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Fig. 3. Tabla de Caracteres imprimibles ASCII.

En la imagen 3 se aprecia que los caracteres en su representación ASCII son los *elementos* mayores que 48 y menores que 57 (dígitos) y por otro lado los mayores a 97 y menores que 122 (letras minúsculas). Es importante aclarar esto para entender de mejor forma el código fuente donde se encuentra implementada la solución de este problema.

B. TDA Utilizado: Listas

El primer problema que surge es obtener cada uno de los caracteres a ocupar en la generación de contraseñas y ocupar un TDA (Tipo de Dato Abstracto) adecuado en la implementación del código en C, cabe destacar que la implementación del código se respeta el estándar ANSI C y no se utiliza ninguna librería distinta a *stdio.h* y *stdlib.h* correspondientes al manejo de entrada-salida y a la librería de un correcto manejo de memoria respectivamente, ésta última cobra una gran importancia ya que es gracias a ella que es posible implementar una programación dinámica respecto a la asignación de memoria ya que resulta imposible calcular a *priori* la cantidad de combinaciones válidas que resultarán, por lo cual es necesario reasignar memoria antes de agregar cada lista a nuestra *listas de listas* de esta forma se optimiza el uso de memoria. Se utiliza una lista doblemente enlazada implementada mediante cursor, la implementación del TDA se encuentra en el archivo de código fuente *2EnlazadasCursor.c* adjunto a este informe (también disponible en github) ya que éste TDA permite un manejo adecuado para los *strings*, los cuales son manejados como una lista de caracteres de un tamaño fijo igual a 8, donde cada elemento contenido en la lista representa a un caracter partiendo desde la posición 0 hasta la última posición correspondiente a la número 7, es decir, la *lista combinación* es una lista de caracteres que si se imprime de forma concatenada, da orígen a una combinación generada por el algoritmo que se explicará en la sección D. A su vez, el TDA permite mantener una *lista de listas* (en la implementación de C corresponde a un puntero a lista) la cual contendrá la lista de combinaciones creadas (totales) así como las que son válidas. Permitiendo posteriormente, agilizar y automatizar la escritura en archivos de un conjunto de listas, ya que poseemos un puntero a la dirección de memoria donde se encuentran todas las combinaciones que son válidas las cuales deben ser escritas en el archivo de salida.

C. Lectura de datos de entrada

Los caracteres a utilizar se encuentran enlistados uno por cada línea como se puede apreciar en 1 lo cual facilita captar cada uno de los caracteres, en primer lugar se lee y se analiza si éste es una letra minúscula o si es un dígito, de no ser ninguno de los casos anteriormente mencionados el programa finaliza e imprime en el archivo de salida (Salida.out) que no se puede utilizar el alfabeto ingresado en el archivo de entrada, en caso de que sí sea un caracter permitido se comprueba que éste no se haya agregado previamente a la lista con los caracteres a utilizar, si ya se encuentra el programa finaliza e imprime el mismo mensaje anteriormente mencionado. Si cumple con las dos condiciones anteriormente mencionadas (que sea un caracter permitido y que éste no se encuentre ya en la lista de caracteres a utilizar) es agregado a la lista de caracteres a utilizar y procede a preguntar por el siguiente caracter en la lista.

D. Algoritmo de Fuerza Bruta

El algoritmo consiste en 8 iteraciones anidadas de forma que cada iterador corresponde a una posición del largo de la combinación a formar y la lista a iterar es la que posee los caracteres a utilizar (***lista caracteres***). A continuación se detalla la estructura del algoritmo, se utilizan los iteradores *a,b,c,d,e,f,g,h*, sin embargo por efectos de espacio el algoritmo queda de la siguiente forma.

Algorithm 1. Algoritmo Fuerza Bruta

```
1: procedure FUERZABRUTA(lista Caracteres)                    ▷
   largoCaractes = n
2:     a, b, c, d, e, f, g, h = 0
3:     while a < n do ▷ Contador aumenta al finalizar el ciclo
4:         while b < n do
5:             ⋮
6:             while h < n do
7:                 Combinaciones ← Caracteres[a] +
   Caracteres[b] + Caracteres[c] + ... + Caracteres[h]▷ Append
8:     return Combinaciones
```

5. ANÁLISIS DE RESULTADOS

En el siguiente capítulo se pondrá a prueba si efectivamente el código funciona, además de la robustez del código respecto a la respuesta a ***eventos inesperados***, como que el alfabeto a utilizar esté vacío, contenga un caracter repetido o contenga un caracter inválido como una letra mayúscula o la letra ñ. En cada subcapítulo se mostrarán las imágenes correspondientes a la consola luego de compilar y ejecutar el programa además de la entrada y salida de los archivos.

A. Camino Feliz

Se realiza una prueba con 6 caracteres = {a,1,b,c,e,4}. Por lo tanto se esperan $6^8 = 1.679.616$ combinaciones totales.

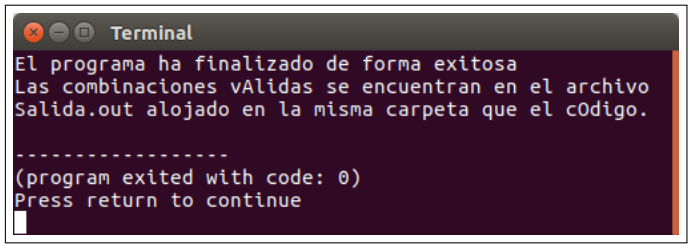


Fig. 4. Consola para el caso del camino feliz

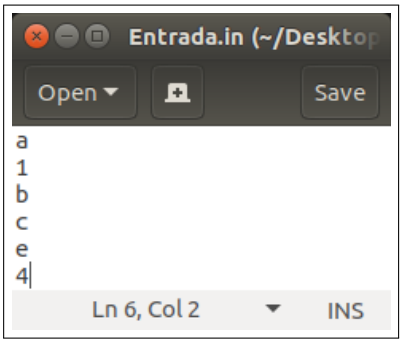


Fig. 5. Archivo de entrada para el caso del camino feliz

La prueba finaliza con éxito, el número teórico de combinaciones totales coincide con el obtenido en la práctica, es decir, en el código. Además, mediante una simple inspección visual se

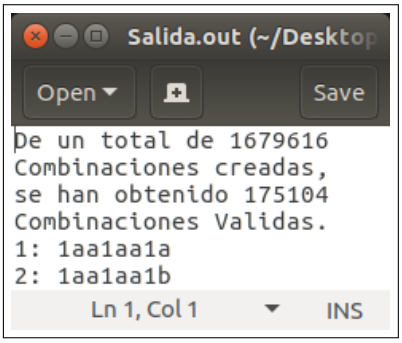


Fig. 6. Archivo de salida para el caso del camino feliz

puede apreciar que ninguna de las 3 condiciones impuestas por el enunciada son violadas dentro de las combinaciones válidas impresas en el archivo *Salida.out*.

B. Caracter no permitido en el archivo de entrada.

En esta subsección se mostrará el comportamiento del programa cuando encuentra un caracter que no es válido según las condiciones que se dieron al problema, en este caso para cualquier caracter distinto a una letra minúscula o a un número distinto a un dígito (0-9).

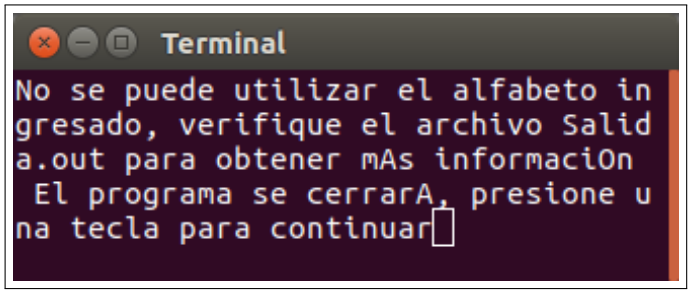


Fig. 7. Consola cuando existe un caracter inválido en el archivo de entrada

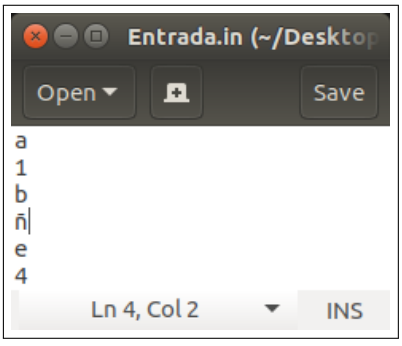


Fig. 8. Archivo de entrada cuando existe un caracter inválido en el archivo de entrada

Como se puede apreciar en la figura 7 y 9 el código responde de buena forma, terminando de forma inmediata al encontrar el primer elemento que no corresponda a los caracteres permitidos, sin embargo en la figura 9 se aprecia que al imprimir el caracter en el archivo de salida no lo hace de forma correcta ya que estamos trabajando en ASCII y el caracter ñ corresponde al

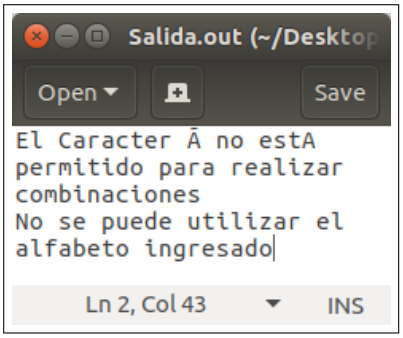


Fig. 9. Archivo de salida cuando existe un caracter inválido en el archivo de entrada

caracter número 164 por lo tanto corresponde a ASCII Extendido, es por esto que no es posible imprimirlo de forma correcta.

C. Caracter duplicado en el archivo de entrada

En esta subsección se pondrá a prueba el código para ver su comportamiento en el caso que exista un caracter duplicado en el alfabeto, ya que si éste lo permite se formarán combinaciones repetidas lo cual perjudicaría la calidad de nuestra solución.

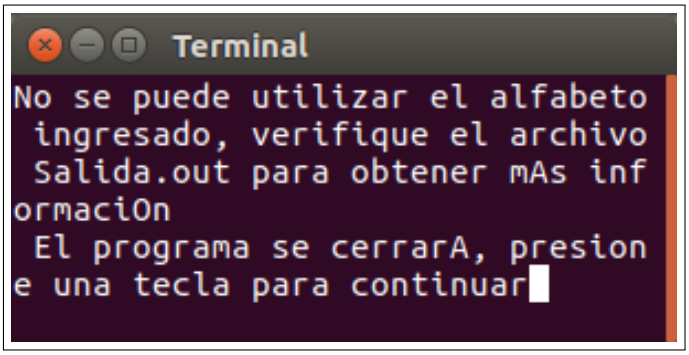


Fig. 10. Consola cuando existe un caracter duplicado en el archivo de entrada

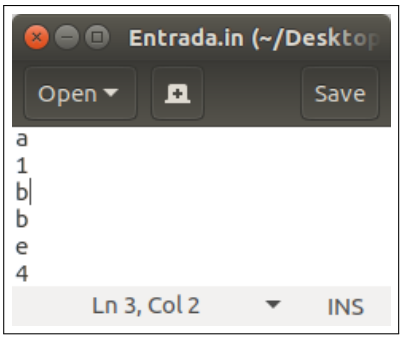


Fig. 11. Archivo de entrada cuando existe un caracter duplicado en el archivo de entrada

En la figura 12 se aprecia que el caracter si se puede imprimir de forma correcta ya que éste si corresponde a el código ASCII normal (y no al extendido) como en el caso de la figura 9 donde el caracter ñ corresponde a ASCII Extendido.

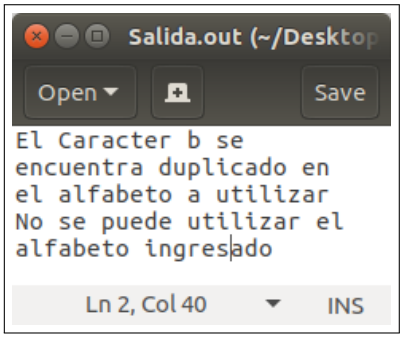


Fig. 12. Archivo de salida cuando existe un caracter duplicado en el archivo de entrada

6. TRAZA DE LA SOLUCIÓN

En esta sección se mostrará la traza de la solución generada por el archivo de entrada que se muestra en la figura 13, es necesario acotar el problema ya que la cantidad de combinaciones totales crece de forma exponencial como se ha visto durante el desarrollo de la experiencia, y 3 caracteres da un número de combinaciones viables para analizar y visualizar en la terminal de nuestro computador. De acuerdo al algoritmo 1 (Fuerza Bruta)

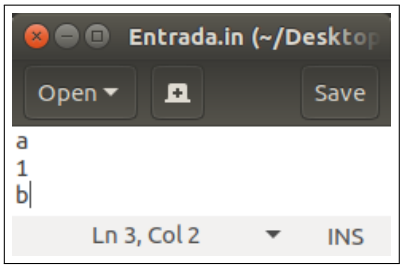


Fig. 13. Archivo de Entrada para análisis de traza

el primer elemento a iterar será el de la posición n-ésima y luego pasará a la (n-1)-ésima y así sucesivamente hasta que llegue a la primera posición de la combinación y si ésta aún posee caracteres a iterar, cambiará y se reiniciará el proceso de iteración para el resto de los caracteres. Se puede apreciar como en la primera

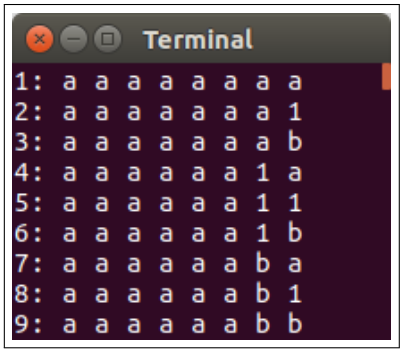


Fig. 14. Combinaciones Posibles sin filtrar

segunda y tercera iteración varía el elemento en la posición 8, y como son sólo 3 caracteres luego varía el de la posición séptima, y luego la octava hasta que no queden caracteres por iterar en la séptima posición y así consecutivamente hasta que se acaben los caracteres a iterar en la primera posición. Cabe destacar que estas son el total de las combinaciones, luego en el código fuente se aplica un filtro con las 3 condiciones descritas al comienzo del informe y da orígen a las ***combinaciones válidas***, las cuales se muestran en la figura 15.

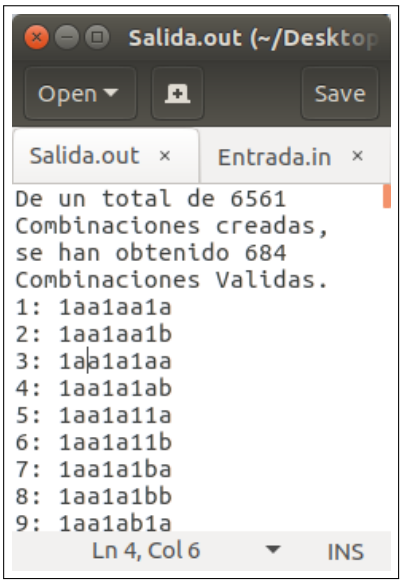


Fig. 15. Combinaciones Válidas, luego de aplicar el filtro

En la figura 15 se puede ver con claridad que no hay ninguna combinación que posea 3 letras o 3 números de forma consecutiva, o alguna combinación que comience con vocal o que comience con un número y termine con un número, es decir, ninguna combinación inválida.

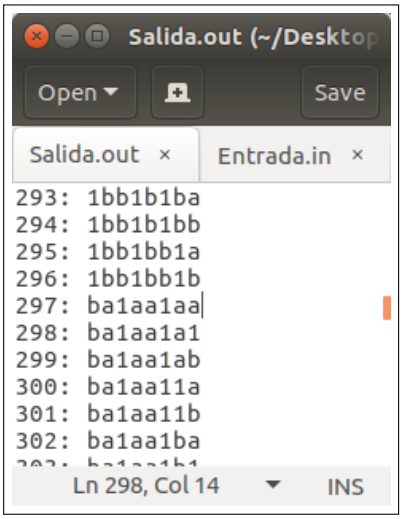


Fig. 16. Combinaciones Válidas, luego de aplicar el filtro

entre la combinación 296 y 297 se puede apreciar la transición entre la iteración del primer caracter en la combinación, dado esto, se reinicia la iteración, todas comienzan en el primer caracter (a, en este caso). La combinación 297 es la primera combinación válida luego de comenzar con la combinación 'baaaaaaa' la cual será la primera combinación que no posea 3 letras ni números de forma consecutiva, como va iterando desde la posición 8va, cada dos letras deberá ir un número, justamente lo que ocurre en esta combinación en la 6ta posición.

7. CONCLUSIONES

Si recordamos la descripción de la solución, en el capítulo 4, se calcula cada uno de los órdenes de las etapas principales para el desarrollo de la experiencia, es decir, $O(n) = n + n^8 + n^2 + n^2$. Sin embargo, el orden que predomina es el del paso correspondiente a la generación de contraseñas, ya que el orden de complejidad mayor predomina sobre los menores. En el algoritmo fuerza bruta se encuentran 8 for anidados, además, se agrega la combinación a una lista que la contendrá. Las listas no son un TDA propio del lenguaje de programación C, sin embargo, el insertar o borrar elementos de una lista son considerados de orden uno ya que son operaciones del TDA. El orden de $O(n) = n^8$ es muy alto, esto significa que mientras más caracteres posea nuestro archivo de entrada el tiempo de ejecución también aumentará de forma exponencial lo cual siempre se intenta evitar ya que el tiempo es un recurso limitado y esto interviene con la calidad de nuestra solución ya que frente a un número muy grande de caracteres como ocurre generalmente ya que las contraseñas pueden ser creadas a partir de 25 caracteres, además, de los 10 dígitos obtendremos un total de $35^8 = 2,251 \cdot 10^{12}$ combinaciones de 8 caracteres, en otras palabras, más de 2,25 trillones de combinaciones totales a analizar y posteriormente filtrar, lo cual resulta poco práctico y hasta contraproducente en términos de optimización de recursos de cómputo y tiempo. Sin embargo, de este método se puede rescatar que se ha comprobado que su implementación resulta muy sencilla en comparación a otros métodos como el de backtracking. Por otro lado, queda demostrado que si mejoramos nuestros niveles de seguridad en nuestras contraseñas (aumentando el largo de las contraseñas, o la cantidad de caracteres distintos, alfa-numérico, etc) dificultando la posibilidad de obtener nuestras credenciales por medio de uno de estos ataques debido a que la complejidad de estos problemas depende de n, donde n es la cantidad de caracteres, en teoría una contraseña de 6 dígitos formada por números y letras podría tardar hasta 20 días, mientras que una de largo 10 podría tardar hasta siglos en romperla [1]. A pesar de la facilidad en su implementación no se recomienda utilizar es método si $\boldsymbol{n}$ es muy grande, como en el caso estudiado, por las razones anteriormente expuestas, resulta ser el peor método en términos de cómputo y tiempo.

REFERENCES

1. P. Gutiérrez, "¿Cuanto tardaría un hacker en reventar nuestra contraseña?" Opt. Express **22**, (2012).
2. J. Mañas, "Análisis de Algoritmos: Complejidad" Departamento de Ingeniería en Sistemas Telemáticos **1**, (1997).
3. M. Villanueva, "Algoritmos: Teoría y aplicaciones" Departamento de Ingeniería Informática, Universidad de Santiago de Chile.**42-43**, (2012).