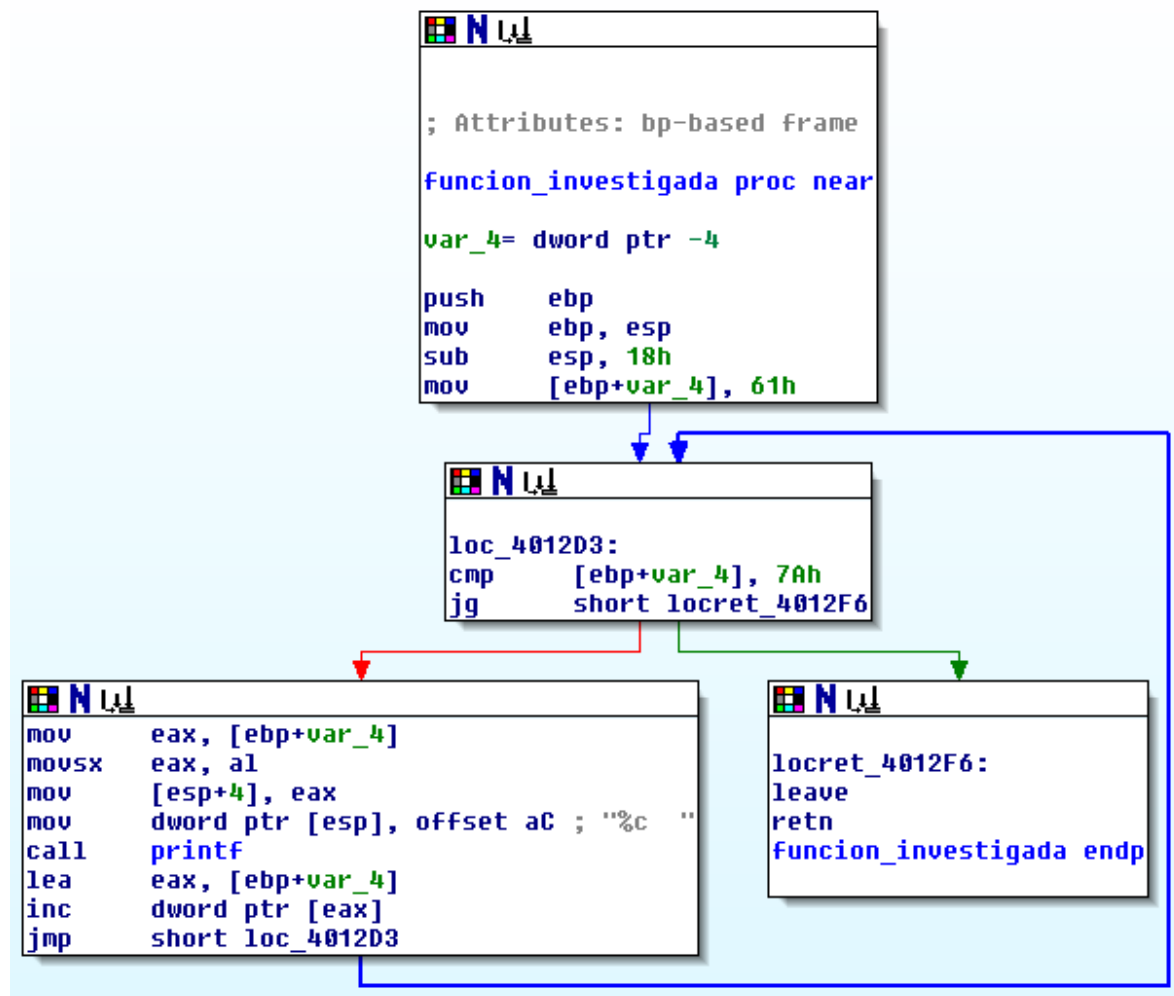
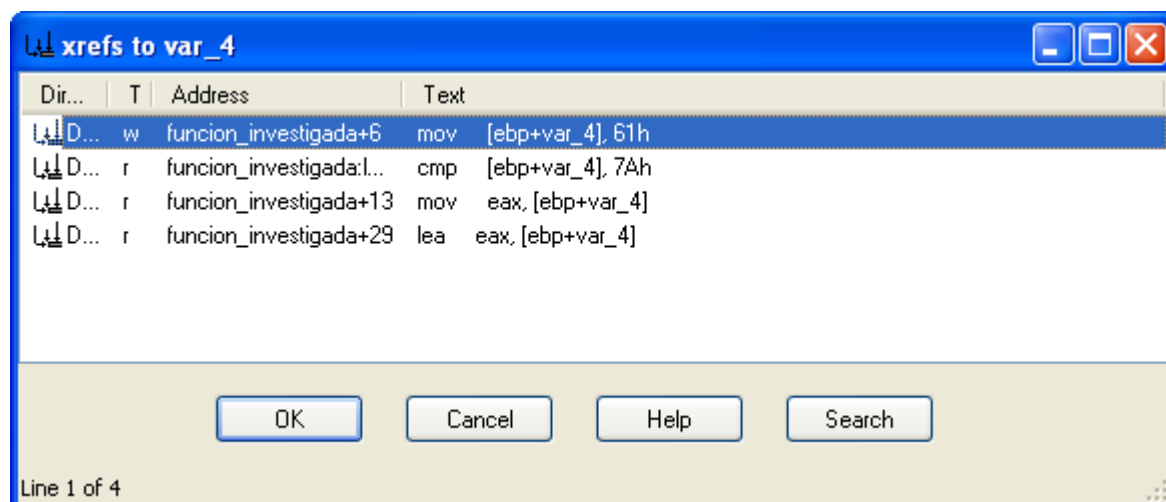


Buenas, en estas poquitas líneas voy a exponer como resolví el ejercicio 6 y 7 que ricnar no quiso resolver. No voy a explayarme mucho así que manos a la obra



Veamos donde se usa esa variable que tenemos



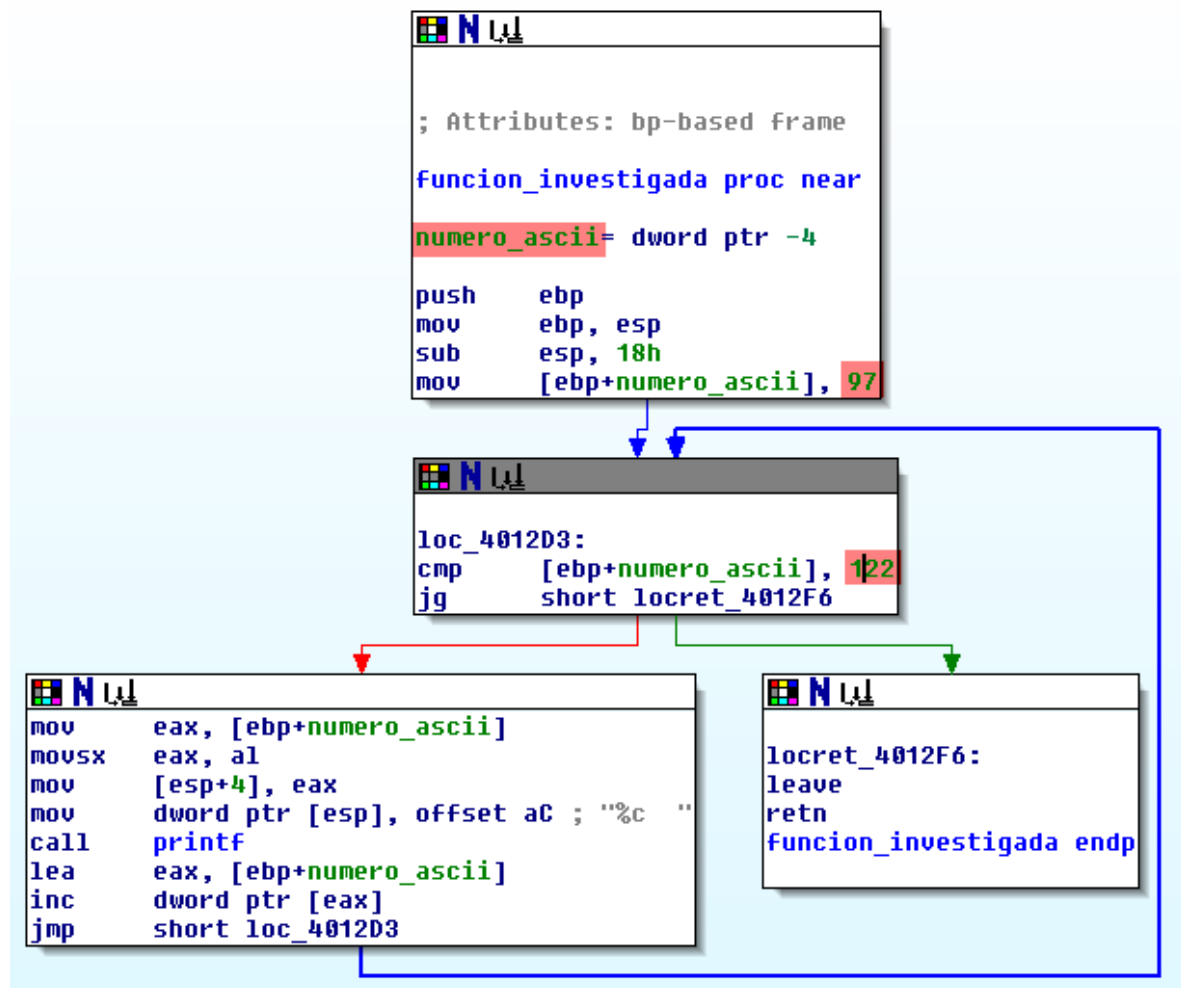
Vemos que se le asigna el valor 61h que corresponde a una “a” y después hay una comparación con 7Ah que es una “z”. Lo primero que podemos pensar es ponerle el nombre carácter a la variable, pero estaríamos en un error.

Los caracteres se almacenan como char y ocupan un byte, y si vemos la definición de esta variable tiene

```
-00000004 var_4          dd ?
+00000000 s              db 4 dup(?)
+00000004 r              db 4 dup(?)
+00000008
+00000008 ; end of stack variables
```

El tamaño es DWORD, así que tiene que ser un entero

Entonces nos tenemos que percatar al vuelo que se trata de un entero representando el valor ascii de los char, la nombramos convenientemente



También le cambie los números a decimal, ya que sabemos que no son caracteres ;)

Ahora toca entender que hace el código.

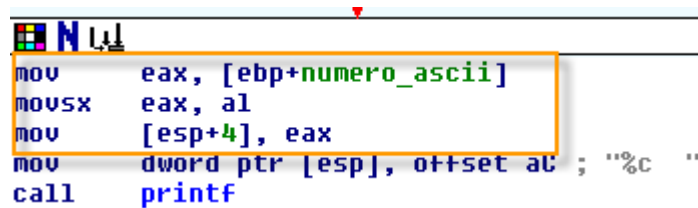
Podemos ver que se le asigna un valor a "numero\_ascii" y después es incrementada aca.

```
lea    eax, [ebp+numero_ascii]
inc    dword ptr [eax]
```

Guarda que aquí se esta incrementando el valor del entero, no vallan a pensar que incrementan el puntero o algo raro

Ya estamos en condiciones de pensar que se trata de un for.

Lo que haría sería pasar por los valores desde 97 a 122 de uno en uno e imprimiendo el char que le corresponde al numero.



```
mov    eax, [ebp+numero_ascii]
movsx  eax, al
mov     [esp+4], eax
mov     dword ptr [esp], offset ac; \"%c \"
call   printf
```

Aca se ve el casting a char. Noten que es "on the fly" porque no se guarda en ningún lado el resultado, directamente se pasa como argumento a printf.

Con la info obtenida, procedo a reescribir la función

```
#include <stdio.h>
```

```
void funcion() {
    int numero_ascii;
    for(numero_ascii = 97; numero_ascii <= 122; numero_ascii++) {
        printf("%c ", (char)numero_ascii);
    }
}
```

```
main() {
    funcion();
    getchar();
}
```

Compilamos y comparamos nuestra función con la de ricnar con ayuda del turbotdiff

identical	4018b0	ExitProcess	4018b0	ExitProcess
identical	4018f0	__sjli_init_ctor	4018f0	__sjli_init_ctor
identical	4012c6	funcion_investigada	401290	sub_401290
suspicious ++	401290	_main	4012c2	_main
unmatched 1	401840	getchar	-	-

Logramos un resultado idéntico. Vamos por el otro ejercicio

```

funcion_investigada proc near

var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_4], 7
mov     [ebp+var_8], 5
fild    [ebp+var_4]
fild    [ebp+var_8]
fdivp   st(1), st
fstp    [ebp+var_C]
fld     [ebp+var_C]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset asc_403000 ; "%f\n"
call    printf
mov     edx, [ebp+var_4]
lea     eax, [ebp+var_8]
mov     [ebp+var_10], eax
mov     eax, edx
mov     ecx, [ebp+var_10]
cdq
idiv    dword ptr [ecx]
mov     [ebp+var_10], eax
fild    [ebp+var_10]
fstp    [ebp+var_C]
fld     [ebp+var_C]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset asc_403000 ; "%f\n"
call    printf
leave
retn
funcion_investigada endp

```

Vemos un par de variables y ninguna estructura de control o bucles como nos tenia acostumbrados ricnar. Vamos a reversearlo haber que es esto.

```

var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_4], 7
mov     [ebp+var_8], 5
fild    [ebp+var_4]

```

Aca tenemos los puntos donde se usa la primer variable, ya notamos que es un entero (tamaño 4 bytes y se le asigna un 7 jejeje), después vemos que se la pasa al FPU así que podemos imaginarnos que el casting es a float

Veamos la otra

```
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_4], 7
mov     [ebp+var_8], 5
fild    [ebp+var_4]
fild    [ebp+var_8]
```

A esta le asigna un 5 y también lo pasa a la FPU, si no me creen vamos a correrlo y veamos los registros del FPU

FPU registers	
ST0	5.0
ST1	7.0
ST2	0.0
ST3	0.0
ST4	0.0
ST5	0.0
ST6	0.0
ST7	0.0
CTRL	
STAT	
TAGS	

Echemos un ojo para ver que hace con estos números

```
var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_4], 7
mov     [ebp+var_8], 5
fild    [ebp+var_4]
fild    [ebp+var_8]
fdivp   st(1), st
fstp    [ebp+var_8]
fld     [ebp+var_C]
```

Los divide y guarda el resultado en var\_C, veamos que hace var\_10 y ya reescribimos los nombres de las variables

```

var_10= dword ptr -10h
var_C= dword ptr -0Ch
var_8= dword ptr -8
var_4= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+var_4], 7
mov     [ebp+var_8], 5
fild    [ebp+var_4]
fild    [ebp+var_8]
fdivp   st(1), st
fstp    [ebp+var_C]
fld     [ebp+var_C]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], of
call    printf
mov     edx, [ebp+var_4]
lea     eax, [ebp+var_8]
mov     [ebp+var_10], eax
mov     eax, edx
mov     ecx, [ebp+var_10]
cdq
idiv    dword ptr [ecx]
mov     [ebp+var_10], eax
fild    [ebp+var_10]

```

Por todos los usos raros que tiene, me atrevo a decir que esta variable fue puesta por el compilador, así que por ahora la obviaremos

```

funcion_investigada proc near

basura_compilador= dword ptr -10h
resultado_flotante= dword ptr -0Ch
entero_1= dword ptr -8
entero_2= dword ptr -4

push    ebp
mov     ebp, esp
sub     esp, 28h
mov     [ebp+entero_2], 7
mov     [ebp+entero_1], 5
fild    [ebp+entero_2]
fild    [ebp+entero_1]
fdivp   st(1), st
fstp    [ebp+resultado_flotante]
fld     [ebp+resultado_flotante]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset asc_403000 ; "%f\n"
call    printf
mov     edx, [ebp+entero_2]
lea     eax, [ebp+entero_1]
mov     [ebp+basura_compilador], eax
mov     eax, edx
mov     ecx, [ebp+basura_compilador]
cdq
idiv    dword ptr [ecx]
mov     [ebp+basura_compilador], eax
fild    [ebp+basura_compilador]
fstp    [ebp+resultado_flotante]
fld     [ebp+resultado_flotante]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset asc_403000 ; "%f\n"
call    printf
leave
retn
funcion_investigada endp

```

Ahí la tenemos renombrada, ahora a reversearla completa. La primera parte antes del printf hace lo que ya dijimos, pasa los valores enteros 7 y 5 a la FPU y los divide entre si, poniendo el resultado en una variable float que luego es imprimida con “%f\n”.

Pero pensemos, en que caso se pasa un entero al FPU, fácil, cuando se castea a float ☺

Así que el código del primer printf es este

```

int primer_num = 7;
int segundo_num = 5;
float division_FPU;
division_FPU = (float)primer_num/(float)segundo_num;
printf("%f\n", division_FPU);

```

Ahora la segunda parte

```

mov     edx, [ebp+entero_2]
lea     eax, [ebp+entero_1]
mov     [ebp+basura_compilador], eax
mov     eax, edx
mov     ecx, [ebp+basura_compilador]
cdq


idiv     dword ptr [ecx]


mov     [ebp+basura_compilador], eax
fild    [ebp+basura_compilador]
fstp    [ebp+resultado_flotante]
fld     [ebp+resultado_flotante]
fstp    qword ptr [esp+4]
mov     dword ptr [esp], offset asc_403000 ; "%f\n"
call    printf

```

Los dos enteros son divididos devuelta, pero con idiv (maneja los enteros directamente), como ya sabemos previamente a esta tenemos un cdq que quiere decir que guardara el resto de la division en edx y el resultado el eax. Este resultado lo manda a la FPU y lo poepa a nuestra variable resultado\_flotante.

Es lo mismo que antes, pero la división la hace con los enteros sin castearlos y lo guarda en la misma variable que antes (luego de enviar la variable basura al FPU lo pushea a resultado\_flotante que es la usamos antes. Entonces el casting ahora lo tenemos en el resultado y no en los operandos de la división.

Se pushea directamente al printf y sale. Reescribamos todo y verifiquemos con el turבודiff

```
#include <stdio.h>
```

```

void funcion() {
    int primer_num = 7;
    int segundo_num = 5;
    float division_FPU;
    division_FPU = (float)primer_num / (float)segundo_num;
    printf("%f\n", division_FPU);

    division_FPU = (float)(primer_num/segundo_num);
    printf("%f\n", division_FPU);
}

main() {
    funcion();
    getchar();
}

```

Lo compilamos y comparamos con el que ya tenemos de Ricardo

identical	401920	__sji_init_ctor	401920	__sji_init_ctor
identical	401290	sub_401290	4012c6	sub_4012C6
identical	4012f2	main	401290	main

Costo, pero ahí estamos devuelta, a partir del binario obtuvimos el source code original sin ningún cambio (tal vez léxico, pero nada raro).

Espero que les haya gustado las soluciones.