7 siete

CORR: DATO

```c
#include <stdlib.h>

//Alumno: AgustÃ-n Gabrielli , legajo 104954

//----------------------------------------------------------------------

//ejercicio 2
void** pila_multifondo(pila_t* pila, size_t k){
    void** arreglo = calloc(k, sizeof(void*)); //para inicializar todos con NULL por si la pila tiene menos de
k elem.
    if(! arreglo) return NULL;

    //usamos una pila auxiliar
    pila_t* pila_aux = pila_crear();
    if(! pila_aux){
        free(arreglo);
        return NULL;
    }

    //pasamos los elem. de la pila original a la aux
    size_t num_elem = 0;
    while(!pila_esta_vacia(pila)){
        pila_apilar(pila_aux, pila_desapilar(pila));
        num_elem++;
    }

    void* dato;
    for(size_t i = 0; !pila_esta_vacia(pila_aux); i++){
        dato = pila_desapilar(pila_aux);
        pila_apilar(pila, dato);
        if(i < k && k < num_elem){
            arreglo[k-1-i] = dato;
        }
        else if(num_elem < k){
            arreglo[k-1-i-num_elem] = dato;
        }
    }
```

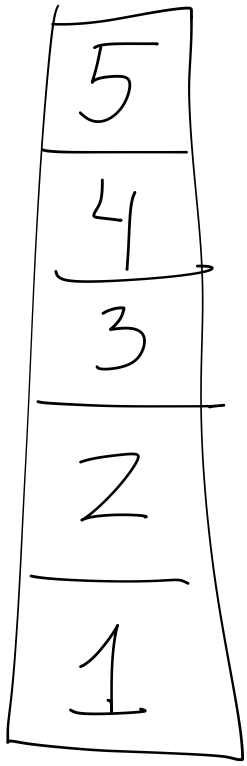
B

i=2 i=1 i=0

3	2	1

n elem - 1 - i

k=3



La consigna pide los NULL al final del arreglo.

```c
    return arreglo;
}

/*ORDEN. T(n) = O(n) siendo n el nÃºmero de elementos apilados en la pila. Esto se debe a que
 tenemos dos ciclos que iteran por todos los elementos de la pila original y aux (desapilando hasta que estÃ©n
vacÃ-as)
 y dentro de esos ciclos las operaciones son todas O(1). Lo que estÃ¡ fuera de los ciclos es O(1), salvo el
 calloc que es O(k). Con lo cual se tendrÃ-a algo asÃ- como 2.O(n) + O(k), o sea O(n+k), aunque si k es desprec
iable
 serÃ-a O(n).*/

//----------------------------------------------------------------------

//ej 6

int _elemento_faltante(int* arreglo, size_t ini, size_t fin, int dif){
    if(ini == fin) return arreglo[ini];

    size_t medio = (ini+fin) / 2;
    suma_izq = arreglo[ini] + (medio-ini) * dif;
    suma_der = arreglo[medio+1] + (fin-medio-1) * dif;

    if(suma_izq == arreglo[medio]) return _elemento_faltante(arreglo, medio+1, fin, dif);
    if(suna_der == arreglo[fin]) return _elemento_faltante(arreglo, ini, medio, dif);

}

int elemento_faltante(int* arreglo, size_t largo){ //wrapper
    //calculo diferencia. La funcion tiene la precondicion de que largo >=4.
    int dif1 = arreglo[1] - arreglo[0];
    int dif2 = arreglo[2] - arreglo[1];
    int dif3 = arreglo[3] - arrelgo[2];

    if(dif1 == dif2) dif = dif1;
    else if(dif1 == dif3) dif = dif1;
    else dif = dif2;

    return _elemento_faltante(arreglo, 0, largo-1, dif);
}

/*Complejidad: es de division y conquista, asÃ- que uso el Teorema Maestro. Hago 1 llamado recursivo
(son 2 pero solo 1 se ejecuta) --> A = 1. B = 2 pues llamo para la mitad del tamaÃ±o del arreglo. C = 0
pues todas las otras operaciones son de tiempo constante. Como logB(A) = 0 = C --> nos queda O(logn).*/

//----------------------------------------------------------------------

//ej 9
```

B

+1

else return arreglo[0] + dif2;

Se pidió PSEUDOCÓDIGO

Pseudocódigo

```
/*Uso Radix Sort. Los alumnos que tienen padron < 100000, que son muy pocos, los ordeno aparte con algun otro a
lgoritmo,
por ejemplo mergesort. Esto es para hacer lo mÃ¡s rÃ¡pido posible este algoritmo. Luego, como no hay padrones d
e 6 cifras que empiecen
con un numero distinto a 1, esa cifra no la cuento. Tampoco hay padrones con la segunda cifra distinta de 0 (po
r ej, no hay 1x0000
con x distinto de 0), asÃ- que eso tampoco lo tengo en cuenta. Nos quedan entonces solamente 4 cifras por anali
zar:
a, b, c y d siendo el padrÃ³n 10abcd; a es la mÃ¡s significativa, d es la menos.
Usamos como algo. de ordenamiento interno counting sort versiÃ³n simplificada.
*/

/*Seguimiento. Agarro 6 padrones de la lista: [104954, 104668, 103448, 105103, 101713, 101713]
Empecemos por la cifra menos sign., la Ã°ltima. Como dije antes, usamos counting sort v. simplificada. La tabla
 de listas
tiene 10 posiciones (del 0 al 9). 104954 --> 4, asÃ- que inserto en la lista del 4. 104668 --> 8, inserto en la
 del 8. 103448 -->8
inserto a continuacion del 104668 en la lista del 8. AsÃ- con cada uno. Luego junto las listas en el arreglo fi
nal.
Me queda [105103, 101713, 101713, 104954, 104668, 103448].
Luego pasamos a las decenas. Haciendo el mismo proceso que antes obtenemos
[105103, 101713, 101713, 103448, 104954, 104968].
Aplicamos devuelta counting sort v. simplificada con las centenas. Me queda:
[105103, 103448, 101713, 101713, 104954, 104968].
Finalmente ordenamos la Ã°ltima cifra, la maÅ\233 significativa. Nos queda:
[101713, 101713, 103448, 104954, 104968, 105103].
Gracias a la estabilidad de counting sort se logra esto.
El orden de Radix Sort es O(d* (n+k)) siendo d la cant. de dÃ-gitos, n el total de elementos, y k el rango del
counting sort.
En nuestro caso, d=4 (es bastante bajo, si n es muy grande termina siendo menor a log(n) seguro), por eso fue i
mportante
reducirlo de 6 a 4 y ver los otros pocos casos aparte. k en este caso es 10 (numeros del 0 al 9) con lo cual es
 despreciable.
Como d y k son despreciables, nos queda que T(n) = O(n)
*/
```

p.g.

sea un hipotético counting sort nuestro ordenamiento auxiliar:

```
fun sort_padrones(lista):
    n = max(padrones)
    dim = 1
    while n >= dim:
        counting_sort(l, clave = λx→(x/dim)%10)
        dim *= 10
```