Ordenación de una pila recursivamente (*mergesort*)

Se desea crear un algoritmo *stackOrder* y su correspondiente traducción al lenguaje C que reciba como parámetro de entrada una pila de enteros y devuelva otra pila con sus elementos ordenados en orden ascendente. La pila original no debe modificarse.

Para ello se dispone de la interfaz del TAD Pila y de las siguientes funciones derivadas (ver los problemas del Tema 2)

```
/* Divide los elementos de una pila ("split") `so` en dos pilas 's1' y 's2'. El orden de los
elementos no se modifica.
Esta función no reserva memoria.
*/
Status stackSplit (Stack *so, Stack *s1, Stack *s2);
/* Devuelve una pila ordenada a partir de dos pilas ordenadas. Es decir, combina ("merge") las dos
pilas pasadas como parámetros.
Esta función reserva memoria.
*/
Stack * stackMerge (Stack *, Stack *);
// Devuelve la copia de una pila. Reserva memoria
Stack *stackCopy (Stack *s);
```

Algoritmo:

Muchos algoritmos tienen una estructura recursiva. Una gran parte de ellos se construyen utilizando la estrategia *divide-y-vencerás*. Para ello se divide el problema en varios sub-problemas similares al problema original pero de menor tamaño. Se resuelven los sub-problemas de forma recursiva y luego se combinan sus soluciones para crear la solución al problema original.

Los algoritmos *divide-y-vencerás* tienen, por tanto, tres etapas:

- **Dividir:** El problema original se divide en un número de problemas disjuntos similares pero de menor tamaño.
- **Conquistar:** Los sub-problemas se resuelven de forma recursiva (*caso general*). Si el tamaño del problema es suficientemente pequeño la solución se obtiene de forma directa (*caso base*).
- **Combinar:** La solución de los sub-problemas se combinan para formar la solución del problema original. Notad como *dividir* y *combinar* son operaciones complementarias.

En este problema concreto ¿Cómo ordenar una pila de n elementos? La respuesta a esta pregunta constituirá el *caso general* del algoritmo recursivo.

.

```
CG1:    Dividir la pila original en dos pilas. Cada una tendrá n/2 elementos.
CG2:    Conquistar: Ordenar cada una de las pilas.
CG3:    Combinar ("merge") las dos pilas ordenadas.
```

- ¿Como dividr la pila original en dos pilas cada una con $n/2$ elementos? Llamando a la función *stackSplit*.
- ¿Como ordenar las pilas creadas en el paso anterior? Llamando recursivamente al algoritmo *stackOrder*.
- ¿Como combinar las dos pilas ordenadas? Llamando a la función *stackMerge*. Esta función crea una nueva pila ordenada con los elementos de las pilas ordenadas que recibe como parámetros.

¿Que se hace en el caso base? Ordenar la pila en el sub-problema mas sencillo: cuando la pila tiene un único elemento. Una pila con un único elemento es *necesariamente* una pila ordenada.

```
CB1:    Si la pila tiene un único elemento devolver una copia de la pila
```

Pseudocódigo:

```
Stack stackOrder (Stack s)  :
------- Base Case ------
CB1:    if stack_size (s) == 1 :
        return stack_copy (s)
------  General Case --------
CG1:    stackSplit (s, s1, s2)
CG2:    sor1 = stackOrder (s1)
      sor2 = stackOrder (s2)
CG3:     sorder = stackMerge (sor1, sor2)
CG4:    return sorder
```

Código C:

La traducción del pseudocódigo al lenguaje C es inmediata. La mayor dificultad radica en la gestión la memoria. Por ejemplo, la función *stackSplit* necesita que se haya reservado previamente memoria para los dos pilas. Por tanto antes de llamarse a *stackSplit* las pilas deberán inicializarse llamando a *stack_init*.

Recordad, además, que todos aquellos objetos que se creen dinámicamente en la función *stackOrder* y que no sean devueltos en su retorno, deberán eliminarse antes de salir de la función.

Por tanto deberán eliminarse las pilas correspondientes a los dos sub-problemas (`s1` y `s2`) y las pilas devueltas en las llamadas recursivas `sor1` y `sor2` . En nuestra implementación, la función recursiva siempre nos devolverá memoria dinámica: tanto cuando retornemos desde el caso base (*stack_copy*) como desde el caso general (*stackMerge*).

Por claridad se proporciona el código C sin control de errores:

```
Stack *stackOrder (Stack *s) {
    Stack *s1 = NULL, *s2 = NULL;     // split matrix
    Stack *sorder = NULL, *sor1 = NULL, *sor2 = NULL;  // order matrix

    if  (stack_size (s) == 1) {
        return stackCopy (s);    // CdE
    }

    s1 = stack_init (int_free, int_copy, int_print);   // CdE
    s2 = stack_init (int_free, int_copy, int_print);   // CdE

    stackSplit (s, s1, s2);
    sor1 = stackOrder (s1);
    sor2 = stackOrder (s2);
    sorder = stackMerge (sor1, sor2);

    stack_free (s1);
    stack_free (s2);
    stack_free (sor1);
    stack_free (sor2);

    return sorder;
}
```

Otros ejemplos de algoritmos *divide y vencerás*

A lo largo del curso se han visto muchos algoritmos *"divide y vencerás"*. Algunos ejemplos:

- Hallar el número total de nodos en un árbol binario
- Hacer una copia de un árbol binario
- Las torres de Hanoi.

BST Equilibrado:

Proporcione un algoritmo y su correspondiente traducción al lenguaje C que cree un árbol binario de búsqueda (BST) óptimo a partir array ordenado de enteros. Se considerará un BST óptimo cuando el coste de la búsqueda de un elemento en él sea, en promedio, $O(\log n)$.

```
Input: a = { 0, 1, 2, 3, 4, 5, 6}
output:                    3
                         /   \
                        1     5
                       / \   / \
                      0   2 4   6
```

Algoritmo:

Para que la búsqueda en BST se eficiente, el árbol debe ser equilibrado. Para ello en la raíz de todos los subárboles deben encontrarse aquellos elementos que dividen el array en partes iguales. Si nos fijamos en el diagrama del árbol binario (output), el 3 divide el array en dos partes de igual tamaño (por un lado la secuencia [0,1,2] que será el subárbol izquierdo y por otro la secuencia [4,5,6] como subárbol derecho), el 5 divide, a su vez, la mitad superior del array en dos partes iguales ([4] y [5]).

¿Como elegir los elementos que debn insertarse en raíces de los subárboles? Con el algoritmo de búsqueda binaria. El algoritmo recursivo que se propone es una variación *directa* del algoritmo de búsqueda binaria.

Pseudocódigo:

```
aOrdBSTt (Integer a[], Integer first, Integer last, BST T)  :
-------  Base case -----
E1:    if first > last :
        return
-------  General case ----
E2:    m = (first + last) / 2
E3:    tree_insert (T, a[m])
E4:    aOrdBSTt (a, first, m-1, T)
E5:    aOrdBSTt (a, m+1, last, T)
```

Código C:

Sin control de errores:

```c
void aOrdBSTt (int a[], int first, int last, BSTree *T) {
    int m;
    //---- Base case ----
    if (first > last) return;

    //---- General case ----
    m = (first + last)/2;
    tree_insert (T, a+m);
    aOrdBSTt (a, first, m-1, T);
    aOrdBSTt (a, m+1, last, T);
}
```

Piensa como sería el algoritmo anterior incorporando el oportuno control de errores.

Coste del algoritmo:

¿Cual sería el coste del algoritmo?

Mínimo de los elementos de una pila.

Proporcione el pseudocódigo de un algoritmo recursivo *stackMin* y la función C correspondiente (esta última con control de errores) para obtener el mínimo elemento de una pila de enteros. El elemento no ha de extraerse de la pila. No está

permitido utilizar mas espacio auxiliar que el proporcionado por la pila del sistema y el necesario para almacenar un único elemento.

```
Status stackMin (Stack *s, int *min);
```

Algoritmo:

En la solución de este problema explotamos la naturaleza recursiva de una pila. Podemos pensar en una pila como una *estructura recursiva:* una pila es una pila vacía o bien un elemento (el tope de la pila) y una pila (que contiene al resto de elementos).

```
-------- Caso General
GM1:    Extraer el tope de la pila
GM2:    Si el elemento extraido de la pila es menor que el valor almacenado en min :
        actualizar el valor de min
GM3:    ejecutar el algoritmo recursivo stackMin sobre la pila
GM4:    Insertar el tope en la pila previamente extraído
------- Caso Base
BM1:    Si la pila está vacía retornar
```

Antes de cada llamada recursiva se extrae el elemento que ocupa el tope de la pila. De esta forma, el algoritmo converge, ya que cada vez que se ejecuta la llamada recursiva, se hace sobre una pila de un tamaño menor. La pila original no se modifica: a la vuelta de la recursión se re-incorpora el elemento extraído previamente. Los elementos se reincorporan a la pila en orden inverso a como fueron extraídos de la pila. El elemento que se extrajo en la recursión mas profunda, el que ocupaba el fondo de la pila, es el primero que se re-incorpora.

¿Tendría algún efecto sobre el resultado del algoritmo alternar el orden de las acciones GM2 y GM3?

Es interesante comparar este algoritmo con la implementación no recursiva que utiliza una pila auxiliar que vimos en el tema 2. En aquel caso, en la pila auxiliar se almacenaban temporalmente los elementos extraídos de la pila. Aquí es la "la pila del sistema" la que se utiliza como contenedor temporal de los elementos extraídos.

Pseudocódigo:

```
stackMin (Stack s, Integer min) :
------- Base case
BM1:    if stack_isEmpty (s) == TRUE :
         return OK
------ General case
GM1:    ele = stack_pop (s)
GM2:    if ele  <  min :
        min = ele
GM3:    stackMin (s, min)
GM4:    stack_push (s, ele)
```

Notad como la variable `min` es externa al procedimiento recursivo. Se utiliza para guardar el valor mínimo. En cada llamada recursiva se comprueba si el valor almacenado en ella es menor que el elemento leído, para, en ese caso, actualizar su valor.

Código C:

```
Status stackMin (Stack *s, int *min) {
    int *ele;
    Status st, st_r;

    // ----- Base case
    if (stack_isEmpty (s) == TRUE) return OK;

    // ---- General case
```

```
    ele = stack_pop (s);
    if (*ele  <  *min)
        *min = *ele;
    st_r = stackMin (s, min);
    st = stack_push (s, ele);
    int_free (ele);

    return (st_r && st);
}

/****** Ejemplo de función main para probar el algoritmo *******/

#include <limits.h>
#include "stacks.h"

int main() {
    Stack *s = NULL;
    int p[] = {1, 10, 0, 6, 9, 8, 7};
        int i, min, n;

    s = stack_init (int_free, int_copy, int_print);
    if (!s) return EXIT_FAILURE;

    n = sizeof (p) / sizeof(p[0]) ;
    for (i=0; i < n ; i++) {
            stack_push (s, (p+i));
    }
    // print the stack before stackmin
        fprintf (stdout, "Stack Before function \n");
        stack_print (stdout, s);

    min = INT_MAX;
    stackMin (s, &min);
    // print the min and the stack
    fprintf (stdout, "Stack After function \n");
    stack_print (stdout, s);
    fprintf (stdout, "El minimo: %d", min);

    stack_free (s);
    return EXIT_SUCCESS;
}
```

En la implementación en lenguaje C de la función recursiva es importante liberar la memoria dinámica.

Obtener el valor mínimo y su posición en una pila

Proporcione un algoritmo que devuelva (sin extraer) el valor mínimo de los elementos almacenados en una pila, la posicion que este ocupa en la pila (contando desde la base de la misma) y el tamaño de la pila.

Código C:

```
int stackMinPos (Stack *s, int *min, int *pos) {
    int *ele;
    int count;

    //------ Base case
    if (stack_isEmpty (s) == TRUE) {
        *pos = 0;
        return 0;
    }
    // ---- General case
    ele = stack_pop (s);
    if (*ele  <  *min)
```

```
        *min = *ele;

    count = stackMinPos (s, min, pos);
    if (*ele == *min)
        *pos = count;

    stack_push (s, ele);
    int_free (ele);
    count ++;
    return count;
}
```

Secuencia creciente-decreciente

Suponga una secuencia de números enteros creciente que, a partir de una determinada posición, se convierte en decreciente. Por ejemplo, la secuencia $\{0, 4, 6, 8, 7, 5, 1\}$ es una secuencia creciente-decreciente. El pivote será el elemnto del array en la que cambia el sentido de la secuencia. En el ejemplo anterior el índice del pivote será 3.

Proporcione un algoritmo que reciba un array de núemros enteros y devuelva el índice del pivote

```
Input:  {0, 4, 6, 8, 7, 5, 1}
Output: 3

Input:  {0, 4, 6, 8, 7}
Output: -1
```

Algoritmo:

Este problema se puede resolver mediante la búsqueda lineal: recorrer la secuencia leyendo elemento a elemento hasta que se encuentre un elemento cuyo valor sea inferior al anterior. El coste de este algoritmo será $O(n)$. Una implementación mas eficiente utilizará la búsqueda binaria.

Si $a[m]$ es el valor medio del array:

Caso General:

- Si $a[m-1] < a[m] > a[m+1]$ entonces m es el índice del pivote
- Sino si $a[m-1] < a[m]$ el pivote se debe buscar en la parte derecha del array y si a[m] > a[m+1] se halla en la parte izquierda.

Caso base:

El mismo que en la búsqueda binaria.

Código C:

```
int bitonic_rec (int a[], int first, int last) {
      int m;

    if (first > last) return -1;

    m = (first + last)/2;


    if (a[m] > a[m+1] && a[m] > a[m-1])
        return m;

    if (a[m] < a[m+1])
        return bitonic_rec (a, m+1, last);

    else
```

```
        return bitonic_rec (a, first, m-1);
}
```

Control de errores en algoritmos recursivos:

Durante el curso se ha insitido en la necesidad de hacer un adecuado control de errores (CdE). El control de los prámetros de entrada forma parte del control de errores. En la función anterior parece evidente que deberían controlarse los parámetros de entrada: ni `a` debería ser NULL ni `first` debería ser negativo ni `last` debería exceder el tamaño del array. Sin embargo esa comprobación solo tendría sentido la primera vez que se llama a la función, no en las llamadas recursivas sucesivas. Para evitar hacer el CdE en las llamadas recursivas, una alternativa muy común consiste en crear una función no recursiva *que envuelva* a la función recursiva. La función envolvente (o **wrapper**) será la encrgada de hacer el oportuno control de los parámetros de entrada:

```
int bitonic (int a[]) {
    int n;

    if (!a) return -1;

    n = sizeof (a) / sizeof (a[0]);
    return bitonic_rec (a, 0, n);
}
```

Secuencia de Fibonacci.

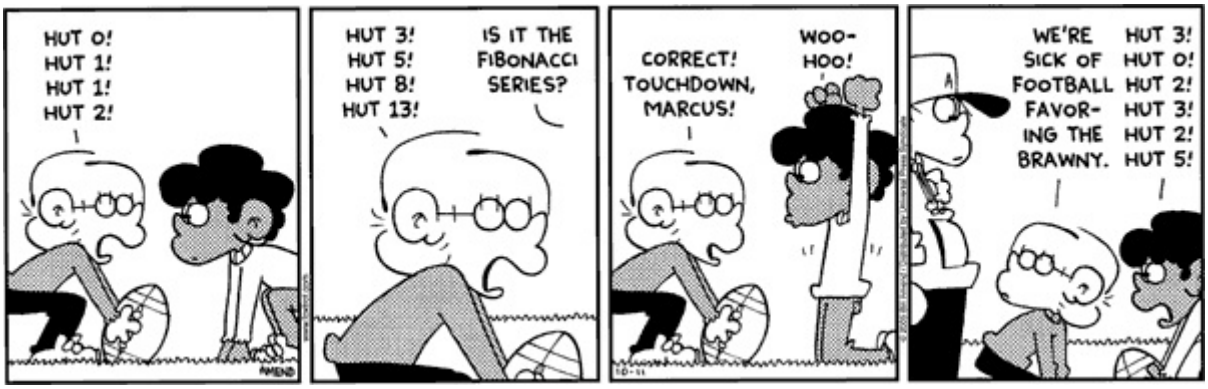
Se denominan números de Fibonacci es la secuencia de números $\{F_n\}_{n=1}^{\infty}$ definidos por la ley de recurrencia:

$$F_n = F_{n-1} + F_{n-2}$$

siendo $F_1 = 1$ y $F_0 = 0$. Notad como el cociente de dos números de Fibonacci sucesivos converge a la proporción áurea ϕ

$$\lim_{n\to\infty} \frac{F_n}{F_{n-1}} = 1.61803\ldots$$

Proporcione el código C de una función que recibe como paraémetro de entrada un entero n y devueleva el número de Fibonacci F_n



En la última viñeta se muestra la secuencia de Parrin

Código C (ineficiente):

```
int fibonacci_ineficiente (int n) {

    // Base case
    if (n == 0) return 0;
    if (n == 1) return 1;

    // General case
```

```
    return fibonacci (n-1) + fibonacci (n-2);
}
```

Notad como en esta implementación recursiva se incluyen múltiples *caso bases*: $n == 0$ y $n == 1$.

Algoritmo:

Es necesario destacar que el algoritmo recursivo anterior es **completamente inadecuado** para calcular el número de Fibonacci F_n.

En el primer problema de esta separata, *ordenar los elementos de una pila (mergesort)*, comentamos brevemente las tres etapas involucradas en la metodología *divide-y-vencerás*: (i) dividir el problema original en subproblemas, (ii) resolver cada uno de ellos recursivamente y (iii) combinar la solución de los mismos para obtener la solución del problema original.

Al igual que en el algoritmo de *mergesort*, el cálculo del número de Fibonacci F_n, se ha resuelto dividiendo el problema en dos subproblemas, F_{n-1} y F_{n-2} y combinado, conforme la ley de recurrencia, sus soluciones. Entonces ¿Por que este algoritmo es ineficiente?¿Cuanto de *ineficiente* es? Es ineficiente porqué los subproblemas en los que se ha divido el problema no son disjuntos. Para resolver F_{n-2} se necesitó calcular de nuevo F_{n-1}. El que en el algoritmo no se aprovechen los resultados obtenidos previamente es lo que lo convierte en muy ineficiente. De hecho el número de llamadas recursivas para calcular F_n es exáctamente F_{n-1}, lo que hace que el coste de este algoritmo sea exponencial $O(\phi^n)$.

- Programación dinámica

La *programación dinámica*, como el método de *divide y vencerás*, resuelve también un problemas mediante la combinación de las soluciones de sus subproblemas. ("Programación" en este contexto se refiere a un método tabular, no a escribir código). Como hemos visto, los algoritmos de *divide y vencerás* dividen el problema en subproblemas disjuntos, resuelven los subproblemas de forma recursiva y luego combinan sus soluciones para resolver el problema original. En contraste, la programación dinámica se aplica cuando los subproblemas solapan, es decir, cuando los subproblemas comparten, a su vez, subproblemas. En esta situación, un algoritmo de *divide y vencerás* realiza más trabajo del necesario: resuelve repetidamente subproblemas comunes. Un algoritmo de *programación dinámica* resuelve cada subproblema solo una vez, guardando la solución en una tabla (array), evitando así el trabajo de volver a calcular la respuesta cada vez que se necesita un resultado ya obtenido.

- *Fibonacci:* Algoritmo basado en programación dinámica

La secuencia de números de Fibonacci $\{F_i\}_{i=1}^{i=n}$ puede obtenerse iterativamente con coste lineal, $O(n)$:

```
F[0] = 0
F[1] = 1

for i = 2 to n :
    F[i] =  F[i-1] + F[i-2]
```

Esta implementación iterativa nos muestra como resolver de forma sencilla cualquier relación de recurrencia con coste lineal. En el caso concreto de de Fibonacci, el cálculo de F_n tendría coste $O(1)$ si se conociesen los números previos de Fibonacci F_{n-1} y F_{n-2}.

En el siguiente algoritmo se utiliza un array externo (`memo`) al procedimiento recursivo para evitar volver a "re-calcular" los números de Fibonacci. El *caso base* del algoritmo devuelve el valor de F_n si el número ya ha sido calculado y, por tanto, se encuentra en el array. El *caso general* resuelve recursivamente los dos subproblemas F_{n-1} y F_{n-2}, combina sus soluciones y, muy importante, **actualiza el array de valores para utilizar en sucesivas llamadas recursivas**. En esta implementación el array externo (`memo`) debe inicializarse a 0.

```
D1: Mantain an array memo[] with all Fibonacci numbers known
D2: If the value of the Fibonacci number is known, just return it
D3: Otherwise, compute it, storage it, and then return it
```

El coste de este algoritmo en tiempo de ejecución es, en el peor caso, $O(n)$ en contraste con el coste $O(\phi^n)$ del algoritmo ineficiente. El coste del almacenamiento externo necesario es lineal con n.

```
#define MAXFIB 46
int fibonacci (int , int *);


int fibonacci (int i) {
    int memo[MAXFIB] = {0} ;

    if (i < 0 || i > MAXFIB-1) return -1;

    return fibonacci_rec (i, memo);
}


int fibonacci_rec (int i, int memo[]) {
    int t;

    // Base case
    if (i == 0) return 0;
    if (i == 1) return 1;
    if (memo[i] != 0)             // If value is known, just return it (D2)
        return memo[i];

    // General case (D3)
    t = fibonacci_rec (i-1, memo) + fibonacci_rec (i-2, memo); //compute the value
    memo[i] = t;            // reemmenber the value

    return t;                // return the value
}
```

Nota: En este implementación hemos elegido el tamaño del array 46. ¿Por que?

Hemos visto que los números de Fibonacci crecen exponencialmente. $F_{46} = 1836311903$ es el mayor número de Fibonacci que puede ser representado con un entero de 32-bit. Por tanto si utilizamos un array de enteros (asumiendo `sizeof (int) == 4`) el tamaño 46 sería suficiente para almacenar los 45 primeros números de Fibonacci.

¿Se te ocurre algún procedimiento para implementar un programa capaz de trabajar con números de Fibonacci de un tamaño superior a *long int*?

Traducción decimal a binario

Proporcione el código C de una función que traduzca un número decimal positivo en su representación binaria como una cadena de caracteres.

```
Input: num = 9  (Integer)
Output: 1001    (String)
```

Código C:

```
int decimal2bin (int num, char *str, size_t n) {
    int i;

    if  (num == 0) {
        return 0;
    }

    i = decimal2bin (num / 2, str, n);
    if (i < n)
        str[i] = num % 2 + '0';  // convert int to char
```

```
    return i+1;
}

#define MAXCHAR 64
int main () {

    int num = 9;
    str[MAXCHAR] = { '\0' }; // La cadena debe finalizar con '\0'

    decimal2bin (num, str, MAXCHAR-1);
    fprintf (stdout, "%s", str);

    return EXIT_SUCCESS;
}
```

Compara esta implementación con la implementaciń no recursiva que utiliza una pila (ver Tema 2).

Palíndromo

Proporcione el código C de una función recursiva para determinar si una cadena de caracteres es un palíndromo

```
Input:"abba"
Outut: TRUE

INPUT:"ab"
OUTPUT: FALSE
```

Código C:

```
Bool isPalindrome (char *str, int first, int last) {

    // Base condition (include the case when strlen(str) is even)
    if (first >= last) {
        return TRUE;
    }

    if (str[first] != str[last])
        return FALSE ;

     // compare the next pair
     return (isPalindrome (str, first+1, last-1));
}
```

Encontrar el número total de *unos* en un array ordenado binario

Proporcione el código C de un algoritmo eficiente que encuentre el número de 1 en un array ordenado binario

```
Input: A[] = [0, 0, 0, 1, 1]
Output: El número total es: 2


Input: A[] = [0, 1, 1, 1, 1]
Output: El número total es: 4
```

Algoritmo:

La solución simple consiste en recorrer el array (búsqueda lineal) hasta encontrar el primer uno del array. El número total de unos sera el número total de elementos n menos el índice de la ocurrencia del primer uno. El coste de este algoritmo sera, en el peor caso, $O(n)$

Una solución recursiva que utilice la estrategia "*divide y vencerás*" es más eficiente. El número total de unos en un array (*caso general*) será el número total de unos que hay en la primera mitad del array mas el número total de unos que hay en su segunda mitad. ¿Cual es el *caso base*? Si la última posición del array es 0 no habrá ningún *uno* en el array y si la primera posición es uno habrá n *unos* en el array.

¿Cual es el coste de esta solución?

Código C:

```
int countOnes (int a[], int n){

    // Base case
    if (a[n-1] == 0)
        return 0;
    if (a[0] == 1)
        return n;

    // General case
    return  (countOnes (a, n/2) + countOnes (a+ n/2,   n-n/2)) ;

int main () {
    int n;
    int p[] = = {0, 0, 1, 1 , 1, 1};

     n = sizeof (p) / sizeof (p[0]);
    fprintf(stdout, "El numero total es: %d ", countOnes (p, n));

    return EXIT_SUCCESS;
}
```

Evaluar una expresión prefijo

Proporcione el código C de una función que evalúe una expresión prefijo de enteros positivos. Asuma que la expresión es siempre una expresión bien formada de números de un solo dígito y en la que los únicos operadores son +, - y *

```
Input: a[]="*+52+32"
Output: El valor de la expresion es 35
```

Código C:

```
#include <stdio.h>
int evalPrefix (char *a) {

    // Base case
    if  ( *a >= '0' && *a <= '9')
        return  *a -'0';  // convert char to integer


    // General case
    if (*a == '+' ) {
        return  evalPrefix (++a) + evalPrefix (++a) ;
    }

    else if (*a == '-') {
        return  evalPrefix (++a) - evalPrefix (++a) ;
```

```
    }

    else if (*a == '*') {
        return  evalPrefix (++a) * evalPrefix (++a) ;
    }

    else
        return -1;
}

int main () {
    char *a;

    a = "*+52+32";
    fprintf (stdout, "El valor de la expresion es %d", evalPrefix (a));
}
```

Rotaciones en un array ordenado

Proporcione el código C de un algoritmo eficiente que encuentre el número de rotaciones en un array circular ordenado. Asumir que no hay elementos repetidos y las rotaciones se realizan el el sentido del reloj

```
Input: A[] = [8, 9, 10, 2, 5, 6]
Output: El array está rotado 3 veces

Input: A[] = [2, 5, 6, 8, 9, 10]
Output: El array está rotado 0 veces
```