Soluciones Ampliadas

En el presente documento se amplía la solución a dos de los ejercicios (1 y 3) para brindar más detalles sin afectar al documento del examen. La solución a todos los ejercicios del examen se subieron a la cuenta TirsoGit en github.com. Los doce códigos están listos para compilar y ejecutar permitiendo su fácil comprobación.

Ejercicio 1. Rewrite the code without recursion.	`unsigned fn(unsigned k)`
	`{`
	`  if (k == 0 || k == 1)`
	`  {`
	`  return 1;`
	`  }`
	`  else`
	`  {`
	`  return fn(k - 1) + fn(k - 2);`
	`  }`
	`}`

Solución.	`unsigned fnIterative(unsigned k)`
	`{`
	`  unsigned varfnk_1=1, varfnk_2=1;`
	`  for (unsigned i=2; i<=k; ++i)`
	`  {`
	`  unsigned result = varfnk_1 + varfnk_2;`
	`  varfnk_1 = varfnk_2;`
	`  varfnk_2 = result;`
	`  }`
	`  return varfnk_2;`
	`}`

El código fuente se subió a Github y entrega independiente.	

Método genérico propuesto para el ejercicio 1.

Se formula este método genérico simple, en cuatro pasos, para realizar la conversión de algoritmos recursivos a iterativos. Es aplicable a otros casos de uso de naturaleza similar, por ejemplo, la conversión de la función factorial.

Paso 1	Sustituir las llamadas a funciones presentes en la versión recursiva por variables en la versión iterativa.
Paso 2	Identificar el caso base donde se detiene la recursión para definir donde comenzaría la iteración.
Paso 3	Identificar el comportamiento que siguen las variables seleccionadas en el paso 1. Esto se realiza manualmente realizando un número de iteraciones representativas, menor que 10 es suficiente. En cada iteración se analizan los valores generados por la función recursiva y se asignan a las variables correspondientes de la función iterativa.
Paso 4	Crear el algoritmo en correspondencia con el comportamiento observado en las variables.

Aplicación del método propuesto al ejercicio 1.

Paso 1: Sustituir las llamadas a funciones presentes en la versión recursiva por variables en la versión iterativa.

Versión recursiva	**Versión iterativa**
return fn(k-1) + fn(k-2)	result = varfnk_1 + varfnk_2

Paso 2. Identificar el caso base donde se detiene la recursión para definir donde comenzaría la iteración.

El caso base es: k=1 y k=0

k	**Versión recursiva**	**Versión iterativa**
0	return 1	result = 1
1	return 1	result = 1

Paso 3. Identificar el comportamiento que siguen las variables seleccionadas en el paso 1. Esto se realiza manualmente realizando un número de iteraciones representativas, por ejemplo 10. En cada iteración se analizan los valores generados por la función recursiva y se asignan a las variables correspondientes de la función iterativa.

k	**Versión recursiva** **return fn(k-1) + fn(k-2)**	**Versión iterativa** **result = varfnk_1 + varfnk_2**	**Resultado**
0	return 1	result = 1	1
1	return 1	result = 1	1
2	return fn(1)+fn(0)	result = 1 + 1	2
3	return fn(2)+fn(1)	result = 2 + 1	3
4	return fn(3)+fn(2)	result = 3 + 2	5
5	return fn(4)+fn(3)	result = 5 + 3	8
6	return fn(5)+fn(4)	result = 8 + 5	13
7	return fn(6)+fn(5)	result = 13 + 8	21
8	return fn(7)+fn(6)	result = 21 + 13	34
9	return fn(8)+fn(7)	result = 34 + 21	55
10	return fn(9)+fn(8)	result = 55 + 34	89

Comportamiento observado: En cada nueva iteración las variables **varfnk_1** y **varfnk_2** toman valores de la iteración anterior. **varfnk_1** toma el valor del resultado anterior, y **varfnk_2** toma el valor que tenía **varfnk_1**.

Paso 4. Crear el algoritmo en correspondencia con el comportamiento observado en las variables.

a) Inicializar las variables (varfnk_1) y (varfnk_2) con valor igual a 1 (caso base).
b) Iterar a partir de k=2 (saltar el caso base).
c) Iterar desde 2 hasta el valor (k):
  result = varfnk_1 + varfnk_2
  varfnk_2 = varfnk_1
  varfnk_1 = result
d) Retornar el valor de (result).

Optimización: Dado que **varfnk_2** toma el valor del resultado, y el resultado se inicializa igual a 1 (caso base), se puede retornar **varfnk_2**, eliminando la necesidad de chequear con un **if** el caso base ya que éste no cumple la condición del **for** y se retornaría 1.

Método genérico (aplicado al caso de uso de la función Factorial).

```
unsigned factorial(unsigned n)
{
    if (n == 0)
    {
      return 1;
    }
    else
    {
      return n * factorial(n - 1);
    }
}
```

Solución.

```
unsigned factorialIterative(unsigned k)
{
    unsigned varfactorialn_1=1;
    for(unsigned i=1;i<=n;i++){
        unsigned result=i*varfactorialn_1;
        varfactorialn_1=result;
    }
    return varfactorialn_1;
}
```

Aplicación del método para la función factorial.

Paso 1: Sustituir las llamadas a funciones presentes en la versión recursiva por variables en la versión iterativa.

Versión recursiva	Versión iterativa
return n*factorial(n-1)	result = var_n * varFactorialn_1

Paso 2. Identificar el caso base donde se detiene la recursión para definir donde comenzaría la iteración.

El caso base es: n=0

k	Versión recursiva	Versión iterativa
0	return 1	result = 1

Paso 3. Identificar el comportamiento que siguen las variables seleccionadas en el paso 1. Esto se realiza manualmente realizando un número de iteraciones representativas, por ejemplo 10. En cada iteración se analizan los valores generados por la función recursiva y se asignan a las variables correspondientes de la función iterativa.

n	Versión recursiva return n * factotial(n-1)	Versión iterativa result = var_n * varFactorialn_1	Resultado
0	return 1	result = 1	1
1	return 1*factotial(0)	result = 1 * 1	1
2	return 2*factotial(1)	result = 2 * 1	2
3	return 3*factotial(2)	result = 3 * 2	6
4	return 4*factotial(3)	result = 4 * 6	24
5	return 5*factotial(4)	result = 5 * 24	120
6	return 6*factotial(5)	result = 6 * 120	720
7	return 7*factotial(6)	result = 7 * 720	5040
8	return 8*factotial(7)	result = 8 * 5040	40320
9	return 9*factotial(8)	result = 9 * 40320	362880
10	return 10*factotial(9)	result = 10 * 362880	3628800

Comportamiento observado: En cada nueva iteración la variable **varFactorialn_1** toma el valor del resultado de la iteración anterior. **var_n** toma simplemente el valor de **n** (el iterador).

Paso 4. Crear el algoritmo en correspondencia con el comportamiento observado en las variables.

a) Inicializar las variables **var_n** y **result** con valor igual a 1 (caso base).
b) Iterar a partir de n=1 (saltar el caso base).
c) Iterar desde 1 hasta el valor **n**:
  result = var_n * varFactorialn_1
  var_n = valor del iterador
  varfactorialn_1 = result
d) Retornar el valor de (result).

Optimización: Dado que **varFactorialn_1** toma el valor del resultado, y el resultado se inicializa igual a 1 (caso base), se puede retornar **varFactorialn_1**, eliminando la necesidad de chequear con un **if** el caso base ya que éste no cumple la condición del **for** y se retornaría 1. Dado que **var_n** toma siempre el valor del iterador, se puede simplificar **var_n**.

Ejercicio 1. Programa en lenguaje C para comprobar los resultados de las versiones iterativas.

```c
#include <stdio.h>
unsigned factorial(unsigned n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
unsigned factorialIterative(unsigned n) {
    unsigned varfactorialn_1=1;
    for(unsigned i=1;i<=n;i++){
        unsigned result=i*varfactorialn_1;
        varfactorialn_1=result;
    }
    return varfactorialn_1;
}
unsigned fn(unsigned k)
```

```
{
    if (k == 0 || k == 1)
    {
       return 1;
    }
    else
    {
       return fn(k - 1) + fn(k - 2);
    }
}
unsigned fnIterative(unsigned k){
  unsigned varfnk_1=1, varfnk_2=1;
  for (unsigned i=2; i<=k; ++i)
  {
    unsigned result=varfnk_1+varfnk_2;
    varfnk_1=varfnk_2;
    varfnk_2=result;
  }
  return varfnk_2;
}
// Place the number  of iterations to test.
#define TEST_ITERATIONS     20
// Place an exclusive 1 on test you want to perform.
#define TEST_SPT_EXERCISE   1
#define TEST_FACTORIAL      0
// Define a name to the test.
#define TEST_NAME_SPT       "SPT TEST"
#define TEST_NAME_FACTORIAL "FACTORIAL"
// Associate a name to a test.
#if TEST_SPT_EXERCISE
    #define TEST_NAME TEST_NAME_SPT
#elif TEST_FACTORIAL
    #define TEST_NAME TEST_NAME_FACTORIAL
#endif
int main(void){
    // Your code here!
    printf("\nTESTING RECURSIVE VS. ITERATIVE [ALGORITHM=%s]\n",TEST_NAME);
    unsigned errorCnt=0;
    for(unsigned i=0;i<TEST_ITERATIONS;i++)
    {
        #if TEST_SPT_EXERCISE
            unsigned rtmp=fn(i);
            unsigned itmp=fnIterative(i);
        #elif TEST_FACTORIAL
            unsigned rtmp=factorial(i);
            unsigned itmp=factorialIterative(i);
        #endif
        if(rtmp==itmp)
            printf("For k=%u [result=%u] => MATCH.\n",i,rtmp);
        else{
            printf("For k=%u [recursive=%u vs iterative=%u] => DO NOT
                    MATCH!\n",i,rtmp,itmp);
            ++errorCnt;
        }
    }
    printf("\nTESTINGS %s [ERRORS=%u].\n",(!errorCnt)?"PASSED,":"NOT
             PASSED!",errorCnt);
}
```

Ejercicio 3. Implement the median filter.
(What it is, you don't need to know, we'll discuss.)

This is expected to be a reusable object. That is, a class in C++ or functions and a structure with data in C, without global variables.

It is desirable that the complexity of the function of adding a new element and getting the result does not exceed O(n), n is the size of the filter window.

Functions are like this:

```
struct
void filter_init(int n, int init_val);
      n - Filter window size.
      init_val - Value for initial window filling.

int filter_add_val(int val_new);
      val_new - The arrival of a new element in the filter.
      Returns the result of the filter.

void filter_destroy();
      Removal after work, if necessary.
```

Ejercicio 3. **Solución.** Se presentan tres variantes de solución.

La primera es la versión clásica que organiza primeramente todas las muestras y posteriormente elige el valor central (cantidad impar de muestras) o el promedio de los dos valores centrales (cantidad par de muestras). Es conocido que este método no es eficiente, su complejidad está determinada por el algoritmo de ordenamiento qsort() que es O(nlog(n)).
La segunda, es una versión propia que quise experimentar para familiarizarme aún más con este tipo de algoritmo, no encontré ninguna propuesta similar en la web. Considero es una versión más óptima que la clásica pero que mantiene simplicidad. A continuación de la tabla se explica la propuesta.
La tercera versión **es la más eficiente** porque recurre al algoritmo Quick Select. Se obtiene una complejidad O(n).
El código fuente de las tres versiones se subió a Github y entrega independiente.

El funcionamiento del algoritmo *MEDIAN_PROPOSAL* propuesto se basa en utilizar un arreglo *S* para memorizar la secuencia real con que entran los valores *v* al filtro, y un arreglo *D* que contiene los mismos valores que *S* pero en orden ascendente. Se asume que previamente sendos arreglos ha sido inicializados, con un valor inicial, o *S* con las muestras iniciales y *D* con dichas muestras ordenadas ascendentemente.

Posteriormente, durante la ejecución del algoritmo cada nuevo valor *v* que entra al filtro se memoriza en *S* y se inserta en *D* atendiendo a su valor, previamente a esto se debe extraer de *D* el valor más antiguo o, con lo cual *D* se mantiene ordenado con una complejidad inferior a la de utilizar un algoritmo de ordenamiento, por ejemplo *O(nlog(n))* de qsort() , el caso más crítico corresponde al ingreso de un valor *v* mayor que todos los restantes por lo que se debe recorrer toda la ventana *D* para ubicarlo al final.

Si el tamaño seleccionado para la ventana *w* es impar existe un valor central que se regresa como resultado, de lo contrario es necesario regresar el promedio de los dos valores centrales.

```
MEDIAN_PROPOSAL ( v )
1       SHIFT ( v )
2       DELETE ( o )
3       for i ← 1 to w - 1
4               if v <= D[i] then
5                       INSERT ( i – 1, v )
6                       break
7               else if i = w-1 then
8                       INSERT ( i, v )
9       if w and 1 then
10              result ← D[w/2]
11      else
12              result ← ( D[(w/2)-1] + D[w/2] ) / 2
13

SHIFT ( v )
1       o ← S[0]
2       for i ← 0 to w -1
3               S[i] ← S[i+1]
4               i ← i + 1
5       S[w -1] ← v

INSERT ( pos, v )
1       for i ← 0 to i < pos
2               D[i] ← D[i+1]
3               i ← i + 1
4       D[pos] ← v

DELETE ( v )
1       found ← false
2       for i ← w-1 to i > 0
3               if not found then
4                       if D[i] = v then
5                               found ← true
6               if found then
7                       D[i] ← D[i-1]
8       i ← i - 1
```

Nombre.	*Tirso Ramón Bello Ramos.*