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);
	}
}
```

SOLUTION (SOLUCIÓN)	
Source code in file.	recursion.c
See the document.	Soluciones ampliadas del Examen SPT.docx

Code without recursion. See the document (Soluciones ampliadas del Examen SPT.docx) where I proposed a simple method to perform this type of conversion in 4 steps, it is demonstrated by applying it to the solution of the factorial (N) algorithm.

```
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;
}
```

2. Find problems in the code.

```
class Foo
{
public:
   Foo(int j) { i = new int[j]; }
   ~Foo() { delete i; }

private:
   int* i;
};

class Bar : Foo
{
public:
   Bar(int j) { i = new char[j]; }
   ~Bar() { delete i; }
```

```
private:
   char* i;
};

void main()
{
   Foo* f = new Foo(100);
   Foo* b = new Bar(200);

   *f = *b;

   delete f;
   delete b;
}
```

SOLUTION (SOLUCIÓN)
Never, we can't convert a derived class (e.g. Bar) to a base class (e.g Foo) because derived classes have members that the base class doesn't. The Bar class constructor doesn't call the Foo base class constructor and it neither has a default constructor to create some integers by default, so you can't create integers data from the Bar derived class at all. Additionally, the design itself doesn't have sense because we don't have here a generalization/specialization relationship as expected.
Nunca, podemos convertir una clase derivada (por ejemplo, Bar) en una clase base (por ejemplo, Foo) porque las clases derivadas tienen miembros que la clase base no tiene. El constructor de la clase Bar no llama al constructor de la clase base Foo y tampoco tiene un constructor predeterminado (default) para crear algunos números enteros de forma predeterminada, por lo que no puede crear datos enteros desde la clase derivada de Bar en absoluto. Además, el diseño en sí no tiene sentido porque no tenemos aquí una relación de generalización/especialización como se esperaba.

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.

SOLUTION (SOLUCIÓN)	
Source code in files.	• median_classic.c (using qsort() worst solution) • median_classic_optim.c (a personal proposal) • median_quick_select.c (**optimal solution**)
See the document.	Soluciones ampliadas del Examen SPT.docx

4. Give your description.

- Contact chatter. *Contact chatter is extended contact bounce effect that is not an inherent parameter of the relay. Contact chatter usually occurs because of either shock or vibration to the relay or an improper relay control signal. A control voltage is applied to the coil of the relay in order for the relay to operate. The relay has a minimum voltage that provides the necessary current for the relay contacts to actuate. If the control voltage drops below the specified minimum operating voltage, the relay may chatter. This rapid on- and off-cycling of the contacts occurs continuously for several seconds, causing excessive contact heating and results in severe damage to the contacts. Also, chatter is caused by the characteristics of a machining system under the continuous action of aperiodic external exciting force. It usually shows the strong relative vibration between a tool and the "load" of the process (e.g. a workpiece in a metal cutting process, target of a drilling operation). Solutions: When dealing with low-voltage chatter, the designer needs to confirm that the specified relay operating characteristics are met. The specified minimum must-operate voltage for the relay must be known. To avoid the external effects of contact chatter control/suppression techniques can be applied, these are classified broadly into passive and active techniques. Passive control technology aims to improve the stability of machining processes by improving machine design or using equipment that can absorb extra energy or get rid of the regeneration effect, thereby changing or modifying a system's behaviors. Active control technology for chatter suppression determines the dynamic behaviors of machine tools by detecting their states, and then adjusts their working statuses by active execution of a decision. Active vibration damping systems are usually composed of monitoring, diagnosis, and execution elements. This kind of control technology is becoming more and more important because of progresses in the fields of computing, sensing, and actuation. Active control technology can bring about notable improvements in the stability of processes. A great deal of research has been dedicated to the development of technologies that are able to predict and detect chatter. The purpose of these technologies is to facilitate the avoidance of chatter during different processes affected by this effect, which leads to better surface precision, higher productivity, and longer tool life.*

- Compare Flash and EEPROM. *Both are non-volatile memories but they have different uses. Flash is used to store bigger quantities of data as demanded by microcontrollers (to store the firmware, internal predetermined configurations, etc.), demanded by external flash memories, and so on. EEPROM is used to store data in less quantities compared with flash memories, they exist both internally to microcontroller (to store user dependent data, mostly these data are configuration parameters) and as standalone components. Flash is fast than EEPROM, the former generally use a parallel interface and the latter use a serial interface, like I2C, SPI, etc. EEPROM are byte-oriented while Flash is block-*

oriented, that's why is easier to erase/read/write an EEPROM. We can store data on both, on flash using a special keyword (e.g., rom). Flash memories are cheaper.

- Compare ARM Cortex-m and AVR. *The ARM Cortex-M family of microcontrollers are 32-bits while AVR are 8-bits, Cortex-m typically has more MHz and also can include an FPU on some devices, memory capacity is greater in Cortex-m, tools & compiler can be used for m0´s up to m7´s its toolchain and workflow are very similar, Cortex-m is modern and it has slowly been chipping away at the 8-bit market as the prices of low-end Cortex-m chips have moved downward, they are low power and minimal code footprint, enabling developers to achieve 32-bit performance at an 8-bit price point. There are more but just the mentioned features make this more powerful and attractive for new designs.*

- Virtual destructor. *When objects are destructed dynamically with a delete operator a problem can arise. If delete is applied to a base class pointer the base class destructor is called even if the pointer is on derived class. The solution is to declare it as virtual in the base class then when delete is applied to a base class pointer, destructors on derived classes are called.*

- Implement the addition operation. First a simple C function to add two integer numbers. Later a C++ class to implement a BCD addition operator using an operator overloading.

SOLUTION (SOLUCIÓN)	
Source code in file.	bcd_addition_operator.cpp

```
int additionOperation(int a, int b){ return a+b;} // Simple C function to add two numbers
```

```
#include <iostream>
using namespace std;

class Bcd {
   typedef uint32_t bcdDigits_t; // Holds up to 8-BCD digits
private:
   bcdDigits_t v;
public:
   Bcd(bcdDigits_t i = 0){
      v = i;
   }
   // The global operator function
   Bcd operator+(Bcd const& bcd)
   {
      bcdDigits_t t1 = v + 0x06666666;
      bcdDigits_t t2 = t1 + bcd.v;
      bcdDigits_t t3 = t1 ^ bcd.v;
      bcdDigits_t t4 = t2 ^ t3;
      bcdDigits_t t5 = ~t4 & 0x11111110;
      bcdDigits_t t6 = (t5 >> 2) | (t5 >> 3);
      Bcd res(t2-t6);
      return res;
   }
   void show() { cout << "bcd=" << hex << static_cast<int>(v); }
};

int main()
```

```
{
    Bcd c1(0x10), c2(0x20); // Remember to enter decimals coded in binary
    c1.show();
    cout << " + ";
    c2.show();
    cout << " = ";
    Bcd c3 = c1 + c2; // An example call to "operator+"
    c3.show(); // The expected result is 0x30
    cout << endl;
    return 0;
}
```

- Check if the number is a power of 2.

SOLUTION (SOLUCIÓN)	
Source code in file.	ispower2.c

```
return ((number & (number-1)) == 0); // Best solution.
```

Below other solution by code, where x is the integer number to test:

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

/*
unsigned testPower2(unsigned x);
return:
   0   when x is a power of 2.
   1   when x isn't a power of 2.
*/
unsigned testPower2(unsigned x){
if(!x) return 1; // 0 is not a power of 2 (return avoids an endless loop)
while(!(x&1))
        x>>=1;
return ((x==1) ? 0 : 1);
}

#define POWER2_ITERATION_TEST (1025)
int main()
{
   // Variable to test the swap functionality
   int result;
   // Excecuting testPower2 function
   for(unsigned i=1;i< POWER2_ITERATION_TEST;i++){
      result=testPower2(i);
      if(!result) // Printing results
         printf("The number %u is a power of two.\n",i);
   }
       return 0;
}
```

- Return a larger integer without using comparison and if.

SOLUTION (SOLUCIÓN)	
Source code in file.	larger_integer.c

```
#include <stdbool.h>

/* Returns the larger integer (does not work when (a == b), (a==0), or (b==0) */

return a*(bool)(a/b) + b*(bool)(b/a); // (bool)(x/y) is 1 when x is larger and 0 if not.

/* Returns the larger integer (improve to accept a==b)  */

return (a - b) ? (a*(bool)(a/b) + b*(bool)b/a) : a;  // but use the comparison operator ?:
```

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

/* Returns the larger integer (does not work when (a==0), (b==0) or (a==b)) */
int larger_int(int a, int b){
  return a*(bool)(a/b) + b*(bool)(b/a);
}

/* Returns the larger integer (also works when (a==b) but not when (a==0) or (b==0)) */
int larger_int1(int a, int b){
  return (a - b) ? a*(bool)(a/b) + b*(bool)(b/a) : a;
}

// Numbers to test (excluding 0)
int TEST_INTEGERS=20;

int main() {
    printf("TESTING LARGER INTEGER FUNCTION\n");

    for(int i=1;i<=TEST_INTEGERS-1;i++){
        int big=larger_int(i,TEST_INTEGERS-i);
        printf("Larger integer between [%d] vs. [%d] is [%d] <- [%s].\n",
                i,TEST_INTEGERS-i,big,
                ((big==i)||(big==(TEST_INTEGERS-i))) ? "PASSED":"DO NOT PASSED!");
                big=larger_int1(i,TEST_INTEGERS-i);
        printf("Larger integer between [%d] vs. [%d] is [%d] <- [%s].\n",i,TEST_INTEGERS-
                 i,big, ((big==i)||(big==(TEST_INTEGERS-i)))?"PASSED":"DO NOT PASSED!");
    }
    return 0;
}
```

- Implement a stack with the storage of the minimum element.

SOLUTION (SOLUCIÓN)	
Source code in file.	stack_min_storage.c

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

#define TEST_STACK_ITERATIONS (10)

#define stackCapacity  (100000L)
#define stackEmpty     (-1L)
#define stackIsEmpty() (p_stack->top==stackEmpty)
#define stackIsFull()  (p_stack->top>=stackCapacity)

// Stack type definition.
typedef struct {
   long top;
   int *data;
} t_stack;

// Initializes stack (reserving the storage of the minimum element).
bool init(t_stack* p_stack){
   if(p_stack==NULL) return false; // invalid stack pointer!
   p_stack->top = stackEmpty;
   p_stack->data = (int*)malloc(sizeof(int)); // mallocs just one element for the stack.
   return (p_stack->data!=NULL); // true if sucess, false if no enough memory.
}

// Frees stack (function renamed to sfree to avoid name conflicts with standard C function free).
void sfree(t_stack* p_stack){
   if(p_stack==NULL) return; // invalid stack pointer!
   if(p_stack->data!=NULL)
      free((void*)p_stack->data);
}

// Pushs a new value to the top of stack.
bool push(t_stack* p_stack, int val) {
   if(p_stack==NULL) return false; // invalid stack pointer!
   if(stackIsFull()) return false; // stack is full!
   p_stack->data[++p_stack->top] = val; // save value on the previous created space
   // Reallocates data to a new space reserving more space to allocate the next pushed element
   p_stack->data=realloc(p_stack->data,(p_stack->top+2)*sizeof(int));
   return true;
}

// Pops the value on top of stack.
int pop(t_stack* p_stack, int* p_val) {
   if(p_stack==NULL) return false; // invalid stack pointer!
   if(stackIsEmpty()) return false; // stack is empty!
   *p_val = p_stack->data[p_stack->top--];
   // Reallocates data to a new space releasing the space of the popped element
   p_stack->data=realloc(p_stack->data,(p_stack->top+1)*sizeof(int));
   return true;
}

// Gets the maximum value stored in the stack.
```

```
int get_max(t_stack* p_stack, int* p_val) {
   if(p_stack==NULL) return false; // invalid stack pointer!
   if(stackIsEmpty()) return false; // stack is empty!
   int max=p_stack->data[p_stack->top];
   for(int i=p_stack->top;i>=0;i--)
      if(p_stack->data[i]>max)
         max=p_stack->data[i];
   *p_val=max;
   return true;
}

// Array for testing purpose only!
int TEST_STACK_ARRAY[TEST_STACK_ITERATIONS*2];

int main() {
   t_stack oneStack;

   printf("TESTING MIN. STORAGE STACK FUNCTIONS [ITERATIONS=%d]\n",
            TEST_STACK_ITERATIONS);
   /* init */
   printf("Testing stack [init() function] %s.\n",init(&oneStack)?"PASSED":"NOT PASSED!");
   /* push */
   unsigned i;
   for(i=0;i<TEST_STACK_ITERATIONS;i++){
      int tmp=rand();
      push(&oneStack,tmp);
      printf("Pushing [%d] : [value=%d].\n",i,tmp);
      TEST_STACK_ARRAY[i]=tmp; // <<== Test line.
   }
   /* get_max */
   int max;
   get_max(&oneStack,&max);
   printf("Testing stack [get_max() function] detected maximum is [%d].\n",max);
   /* pop */
   int val;
   i=TEST_STACK_ITERATIONS;
   while(pop(&oneStack,&val)){
      printf("Poping [%d] : [value=%d].\n",TEST_STACK_ITERATIONS*2-(i+1),val);
      TEST_STACK_ARRAY[i++]=val; // <<== Test line.
   }
   /* free */
   sfree(&oneStack);

   /* testing pushing-poping order */
   for(i=0;i<TEST_STACK_ITERATIONS;i++){
      printf("Testing ");
      if(TEST_STACK_ARRAY[i]==TEST_STACK_ARRAY[(TEST_STACK_ITERATIONS*2-1)-
i])
         printf("PUSH [%d] vs POP [%d] with value [%d] ==> PASSED.\n", i,
i,TEST_STACK_ARRAY[i]);
      else
         printf("PUSH [%d] vs POP [%d] with value [%d] ==> NOT PASSED.\n", i,
               i,TEST_STACK_ARRAY[i]);
   }
   return 0;
}
```

- Discuss the UART driver. *UART drivers are hardware components used to empower microcontroller UART signals when they need to get out to the outside world, and travel some distances, in order to communicate with other external systems. Inside the board for internal UART serial communications instead of drivers sometimes voltage level translators are needed if the participants operate at different voltages.*

a. Start initialization. *There exist different types of initializations, but when an electronic device system starts for the first time or after a power interruption, its common to classify the startup, in a simple approach, as cold start or hot start. Cold starts occurs technically when the system start for the first time (here there is not a previous history) so additionally to the known initializations the variables takes its default values. Hot starts occurs when the system start after some interrupt (here there exist a history) so additionally to the known initializations some variables most retains its previous values (e.g. previous configured values to setpoints, calibrations results, parameters of control functions like PIDs, and so on).*

b. Last. interfaces, flash / eeprom. *Both types of non-volatile memory technologies (flash/eeprom) are commonly used in electronic devices for data storage. Flash memories uses a parallel interface to connect to the microprocessor/microcontroller, it is erased block-wise (uses a more complex process called "erase-before-write." To write data, a block of Flash memory must be erased first. This involves clearing a large number of cells simultaneously, which makes writing slower compared to EEPROM. Flash cells have higher density but are slower to erase and write. Flash memory offers higher endurance than EEPROM due to wear-leveling algorithms that distribute write and erase cycles across different memory blocks. It's widely used in applications that require frequent data updates. In this way they last over 10 years or more. They are cheaper then EEPROM. EEPROM uses I2C and SPI Interface, it is erased byte-wise (can be electrically programmed and erased allowing individual bytes of data to be written, erased, and re-written). It has a limited write-erase cycle endurance compared to Flash memory. It is suitable for applications where frequent updates are not required. In this way they last over 10,000 to 100,000 write cycles.*

c. RS-485 pauses. *Well, as I understand, RS485 is half duplex, it xmits and receive using the same pair of cables so we need to change the direction switching drivers. For this reason, when a master send a message to the bus, it needs to put driver direction to xmit, xmit all data, and finally it need to switch to receive the answer from slave. During switching the bus is idle, and we could say it is paused.*

d. GPIO atomicity: *When a GPIO is shared between tasks sometimes is necessary to avoid simultaneous operations on it.*

e. boot stm32. *STM32 microcontrollers have three boot options: 1. Boot from user Flash (usually where your firmware is located), 2. Boot from system memory (where the ST embedded bootloader is located), 3. Boot from embedded SRAM (usually used for debugging purpose, or a specific action). The target option must be selected configuring a combination of internal bits and external pins as specified in the reference manual of the chosen STM32 device. The bootloader is one of the preferred options, it is stored in the internal boot ROM (system memory) of STM32 devices, and is programmed by ST during production. Its main task, as most bootloader, is to download the application program to the internal flash memory through one of the available serial peripherals. USART, CAN, USB, I2C, and SPI interfaces are suppported but depending on the STM32*

device used, the bootloader may support one or more built-in serial peripheral to download code to the internal flash memory.

f. I2C in AVR and STM32: *Most AVR has a TWI module that can operate in both Master and Slave mode up to 400kHz with a maximum of 128 devices. Almost all STM32 supports 1 or more I2C interfaces in three different modes (Standard up to 100kHz, Fast up to 400kHz and Fast Plus up to 1MHz). Based on STM32Cube HAL functions, I2C data transfer can be performed in 3 modes: Blocking Mode, Interrupt Mode or DMA Mode. Depending on specific STM32 device implementation DMA capability can be available for reduced CPU overload.*

g. Signal sampling frequency: *The number of samples of a signal taken in one second. The inverse of this value (1/ Signal sampling frequency) is the sampling period, i.e. the time between samples. Must be selected taking into account the nature of the signal, usually at three times its máximum frecuency.*

h. How to run a stepper motor in continuous rotation mode. *It requires energizing and de-energizing all the coils it has within the motor, one coil in opposite pair at a time, in a certain sequence in order to achive the full rotation (360 degrees), and after that, advance one step more, to reach the first step again, repeating the sequence over and over for the time needed. For this purpose an special driver is needed or a motor control circuit based on a microcontroller.*

i. Organization of communication with PC, MCU, how the protocol works. encrypted protocol. *To establish a communication between a PC and a MCU first we need to identify the physical media. Generally, we need to use an intermediary adapter to connect them, the most popular options are USB-RS232, USB-CAN, Ethernet, USB alone, or some other. Protocols allow a common language with the rules to follow during conversation. Well designed protocols are composed by different layers (e.g. TCP/IP use the OSI model), each layer have well defined functionalities and it exchange information with the pair layers on the other devices. Physical layer must be present on all protocols because this is related with the hardware components, cables, topologies, etc. Data are exchanged between connected devices attending to the protocol in charge, in order to protect the information data aren´t moved in plain format but encripting it using different levels of complexity. Eventhough, some protocols don´t need to encrypt information, for example: NMEA protocol, MODBUS, MAP27, and many other, these are protocol to exchange local information.*

j. Simple multiplication by a fractional number in AVR. *The Atmel megaAVR (8-bit uC serie) of AVR RISC Microcontroller family included a hardware multiplier enhancement capable of multiplying integer and fractional numbers (unsigned/signed). In just two clock cycles it can multiply two 8-bit numbers giving a 16-bit result. There exist three basic types of multiplication operations (MUL, MULS and MULSU) for integers that must be preceded with the letter F to apply to fractional numbers (i.e. FMUL, FMULS and FMULSU). MUL/FMUL operate on unsigned numbers (integer/fractional), MULS/FMULS operate on signed numbers (integer/fractional), and MULSU/FMULSU operate with a mix of one signed and other unsigned operands. MULX operations are idem to FMULX operations with the exception that the latter do a shift left operation to the 2-byte result so that its high byte holds the same 1.7 bit format (bit 7 represents the integer part 0 or 1, and bits 0-6 the fractional part with weights 2-1, 2-2 and so on). These fractional numbers are in the range [-1, 1] with the negative values represented in two's complement.*

k. Features of std containers. Storing values with a counter and fast lookup. *The containers library is a collection of templates and algorithms that implement the common data structures that we work with as programmers. A container is an object that stores a collection of elements (i.e. other objects). Each of these containers manages the storage space for their elements and provides access to each element through iterators and/or member functions.*

The standard containers implement structures that are commonly used in our programs, such as:

- *dynamic arrays*
- *queues*
- *stacks*
- *linked lists*
- *trees*
- *associative sets*

The C++ container library categorizes containers into four types:

- *Sequence containers*
- *Sequence containers adapters*
- *Associative containers*
- *Unordered associative containers*

SOLUTION (SOLUCIÓN)

We can use a map (std::map) it is a sorted associative container that contains key-value pairs with unique keys. Keys are sorted by using the comparison function Compare. Search, removal, and insertion operations have logarithmic complexity.

Each value (the key) will have an associate counter (the value). Values are unique they can increase or decrease its quantities (counter), new values can be registered on the map or deleted from it.

```
#include <iostream>
#include <map>
#include <string>
#include <string_view>
using namespace std;

void print_map(string_view comment, const map<int, int>& m)
{
   cout << comment;
   // iterates using C++17 facilities
   for (const auto& [key, value] : m)
      cout << "[val=" << key << "] : Count=" << value << "; ";

   cout << '\n';
}

int main()
{
   // Create a map to store values with an associate counter that represents the quantity of this value.
   map<int, int> m{
      {0/*value 0*/, 10/*qty 10*/},
      {1/*value 1*/, 100/*qty 100*/},
```

```
        {5/*value 5*/, 2/*qty 2*/}
    };

    print_map("1) Initial map: ", m);

    m[0] = 200; // updates an existing value
    m[4] = 300; // inserts a new value
    print_map("2) Updated map: ", m);

    // Using operator[] with non-existent key always performs an insert
    cout << "3) m[val=15] => Count= " << m[15] << '\n';
    print_map("4) Updated map: ", m); // A new value 15 is inserted with counter in 0

    m.erase(0); // Erase value 0
    print_map("5) After erase(0): ", m);

    erase_if(m, [](const auto& pair){ return pair.second > 100; }); // Erase values gt 100
    print_map("6) After erase: ", m);
    cout << "7) m.size() = " << m.size() << '\n';

    m.clear();  // Clears the map
    cout << boolalpha << "8) Map is empty: " << m.empty() << '\n';
}
```

l. Complexity of sorting. *Generally sorting algorithms are used to sort groups of data in an increasing or decreasing manner. There are different sorting algorithms with an inherent time complexity, also a mixture of some of them to improve results. The time and space complexity of most of them are:*

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Selection Sort	*Ω(n^2)*	*θ(n^2)*	*O(n^2)*	*O(1)*
Bubble Sort	*Ω(n)*	*θ(n^2)*	*O(n^2)*	*O(1)*
Insertion Sort	*Ω(n)*	*θ(n^2)*	*O(n^2)*	*O(1)*
Heap Sort	*Ω(n log(n))*	*θ(n log(n))*	*O(n log(n))*	*O(1)*
Quick Sort	*Ω(n log(n))*	*θ(n log(n))*	*O(n^2)*	*O(n)*
Merge Sort	*Ω(n log(n))*	*θ(n log(n))*	*O(n log(n))*	*O(n)*
Bucket Sort	*Ω(n+k)*	*θ(n+k)*	*O(n^2)*	*O(n)*
Radix Sort	*Ω(nk)*	*θ(nk)*	*O(nk)*	*O(n+k)*
Count Sort	*Ω(n+k)*	*θ(n+k)*	*O(n+k)*	*O(k)*
Shell Sort	*Ω(n log(n))*	*θ(n log(n))*	*O(n^2)*	*O(n)*
Tim Sort	*Ω(n)*	*θ(n log(n))*	*O(n log(n))*	*O(n)*
Tree Sort	*Ω(n log(n))*	*θ(n log(n))*	*O(n^2)*	*O(n)*
Cube Sort	*Ω(n)*	*θ(n log(n))*	*O(n log(n))*	*O(n)*

m. Why you need a binary tree. *It is the data structure with more uses compared with the other types. It is an ideal form to store data in a hierarchical way, reflects structural relationships in the given data set, make insertion and deletion faster than array and linked lists, very flexible to hold and move data, it is faster than linked list but slower than arrays when accessing elements.*

5. Resolve the problem.

Given numbers:
uint32_t a, b;

Swap k bits from number a, starting from bit ia, to number b, starting from position ib. The remaining bits of b remain as they were.

Example:
a: AAA0101AAAAAA
b: BBBBBBBBBBBB

k = 4, ia = 6, ib = 3;

Result:
b: BBBBBB0101BBB

SOLUTION (SOLUCIÓN)	
Source code in file.	swap_bits.c

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

/*
int swapFunc(uint32_t* a, uint32_t* b, uint8_t k, uint8_t ia, uint8_t ib);
This version tries to swap ‘k’ bits from a source number at ‘a’ position ‘ia’ to a destination number
‘b’ at position ‘ib’, the operation success if, and only if, it can get all ‘k’ bits from source and put
all of them on destination.
Returns:
 0 success
    -1 invalid bit position in source
    -2 invalid bit position in destination
    -3 invalid number of bits to swap
    -4 invalid arguments to get all bits from source
    -5 invalid arguments to put all bit on destination
    -6 invalid address of source, destination or both
*/

int swapFunc(uint32_t* a, uint32_t* b, uint8_t k, uint8_t ia, uint8_t ib){

  // validates input arguments...
  if(ia>sizeof(uint32_t)*8-1) return -1;
  if(ib>sizeof(uint32_t)*8-1) return -2;
  if(k>sizeof(uint32_t)*8)    return -3;
  if(ia+k>sizeof(uint32_t)*8) return -4;
  if(ib+k>sizeof(uint32_t)*8) return -5;
  if(!a||!b)  return -6;
  // code
  uint32_t aaux, mask0, mask1;
  mask1=(1<<k)-1;  // creates a mask with k number of 1s
  mask1<<=ib;                  // prepares mask0 to clean bits at recipient b
```

```
    mask0=~mask1;
    *b&=mask0;        // clears destination bits
                  // shifts source bits to destination location
    if(ib<ia)   // shift right just the distance in bits
       aaux=*a>>(ia-ib);
    else   // shift left just the distance in bits, or no shift at all
       aaux=*a<<(ib-ia);
    aaux&=mask1;  // masks source bits to swap
    *b&=mask0;    // masks destination bits to preserve
    *b|=aaux;     // swaps bits
    return 0;
}

/* main function */
int main()
{
   // Variables to test the swap functionality
   uint32_t a, b, c;
   uint8_t k, ia, ib;
   int result;
   // Preparing arguments
   a=(uint32_t)0x56AAAAAA;   // Places A number
   c=b=(uint32_t)0xBBBBBBBB; // Places B number
   k=8; ia=24; ib=0;         // Places remaining arguments
   // Excecuting swap function
   result=swapFunc(&a,&b,k,ia,ib);
   // Printing results
   printf("Swap return=%d\nA=%x\nB=%x\nSwap result=%x\n", result,a,c,b);
}
```

6. Resolve the problem.

Implement a stack that stores integers.

- It must be able to return the maximum stored number.
- All functions run in constant time (independent of the size of the stored data).
- Maximum stack capacity: 100,000 items.

Functions

```
typedef struct {
   ...
} t_stack;

bool init(t_stack* p_stack);
void free(t_stack* p_stack);
void push(t_stack* p_stack, int val);
bool pop(t_stack* p_stack, int* p_val);  // False if stack is empty.
```

```
bool get_max(t_stack* p_stack, int* p_val);  // False if stack is empty.
```

SOLUTION (SOLUCIÓN)

Source code in file.	integers_stack.c

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

#define TEST_STACK_ITERATIONS (5)

#define stackCapacity  (100000L)
#define stackEmpty     (-1L)
#define stackIsEmpty() (p_stack->top==stackEmpty)
#define stackIsFull()  (p_stack->top>=stackCapacity)

// Stack type definition.
typedef struct {
   long top;
   int *data;
} t_stack;

// Initializes stack.
bool init(t_stack* p_stack){
   if(p_stack==NULL) return false; // invalid stack pointer!
   p_stack->top = stackEmpty;
   p_stack->data = (int*)malloc(stackCapacity*sizeof(int));
   return (p_stack->data!=NULL); // true if sucess, false if no enough memory.
}

// Frees stack (function renamed to sfree to avoid name conflicts with standard C function free).
void sfree(t_stack* p_stack){
   if(p_stack==NULL) return; // invalid stack pointer!
   if(p_stack->data!=NULL)
      free((void*)p_stack->data);
}

// Pushs a new value to the top of stack.
bool push(t_stack* p_stack, int val) {
   if(p_stack==NULL) return false; // invalid stack pointer!
   if(stackIsFull()) return false; // stack is full!
   p_stack->data[++p_stack->top] = val;
   return true;
}

// Pops the value on top of stack.
int pop(t_stack* p_stack, int* p_val) {
   if(p_stack==NULL) return false; // invalid stack pointer!
   if(stackIsEmpty()) return false; // stack is empty!
   *p_val = p_stack->data[p_stack->top--];
   return true;
}

// Gets the maximum value stored in the stack.
int get_max(t_stack* p_stack, int* p_val) {
   if(p_stack==NULL) return false; // invalid stack pointer!
   if(stackIsEmpty()) return false; // stack is empty!
   int max=p_stack->data[p_stack->top];
```

```
    for(int i=p_stack->top;i>=0;i--)
        if(p_stack->data[i]>max)
            max=p_stack->data[i];
    *p_val=max;
    return true;
}

// Array for testing purpose only array!
int TEST_STACK_ARRAY[TEST_STACK_ITERATIONS*2];

int main() {
    t_stack oneStack;

    printf("TESTING STACK FUNCTIONS [ITERATIONS=%d]\n",
            TEST_STACK_ITERATIONS);
    /* init */
    printf("Testing stack [init() function] %s.\n",init(&oneStack)?"PASSED":"NOT PASSED!");
    /* push */
    unsigned i;
    for(i=0;i<TEST_STACK_ITERATIONS;i++){
        int tmp=rand();
        push(&oneStack,tmp);
        printf("Pushing [%d] : [value=%d].\n",i,tmp);
        TEST_STACK_ARRAY[i]=tmp; // <<== Test line.
    }
    /* get_max */
    int max;
    get_max(&oneStack,&max);
    printf("Testing stack [get_max() function] detected maximum is [%d].\n",max);
    /* pop */
    int val;
    i=TEST_STACK_ITERATIONS;
    while(pop(&oneStack,&val)){
        printf("Poping [%d] : [value=%d].\n",TEST_STACK_ITERATIONS*2-(i+1),val);
        TEST_STACK_ARRAY[i++]=val; // <<== Test line.
    }
    /* free */
    sfree(&oneStack);

    /* testing pushing-poping order */
    for(i=0;i<TEST_STACK_ITERATIONS;i++){
        printf("Testing ");
         if(TEST_STACK_ARRAY[i]==TEST_STACK_ARRAY
            [(TEST_STACK_ITERATIONS*2-1)-i])
           printf("PUSH [%d] vs POP [%d] with value [%d] ==> PASSED.\n", i,
                   i,TEST_STACK_ARRAY[i]);
        else
           printf("PUSH [%d] vs POP [%d] with value [%d] ==> NOT PASSED.\n", i,
                   i,TEST_STACK_ARRAY[i]);
    }

    return 0;
}
```

7. Resolve the problem.

Implement dynamic memory allocation functions, for a layer between the standard memory manager (malloc or new), and applied task.
The task requires that the allocated memory start at the address multiple of 128 bytes. malloc or new are known to return memory, starting with an arbitrary address. We consider that the addresses in the system 64-bit.

The functions are:

void* my_malloc(uint32_t size_bytes);
Allocates memory of size size_bytes that can be used by applied task. The returned pointer is a multiple of 128 bytes.

void my_free(void* ptr);
Frees up memory. Accepts a pointer that was allocated via my_malloc().

It is advisable to make a test that demonstrates the correct operation.

SOLUTION (SOLUCIÓN)	
Source code in file.	malloc_layer.c

```
#include <stdio.h>
#include <stddef.h>
#include <stdlib.h>
#include <stdint.h>

#define myMallocDebug // Comment it on release version!
#define myMallocBytes 10
#define myMallocIter  myMallocBytes

typedef uint8_t myMallocOffset_t; // 8-bits is enough to hold 128 bytes boundaries

#define myMallocAligment (128)
#define myMallocAlign(p) (((p)+(myMallocAligment-1)) & ~(myMallocAligment-1))
#define myMallocOffsetSize (sizeof(myMallocOffset_t))
#define myMallocHeaderSize (myMallocOffsetSize+(myMallocAligment-1)) // Header size is
for the worst case (malloc address aligned)

void* my_malloc(uint32_t size_bytes){
   void* ptr=NULL;

   if(size_bytes){
      // Allocates header size + requested bytes
      void * p = malloc(myMallocHeaderSize+size_bytes);
      #ifdef myMallocDebug
         printf("malloc [non aligned address=%p] : [requested bytes=%d].\n",(uintptr_t)p,
                  (int32_t)(myMallocHeaderSize+size_bytes));
      #endif
      if(p){
         /* If malloc address is not aligned the user pointer will be aligned to next boundary
            If malloc address is aligned (there is no room space to store offset behind it) the user
             pointer will   skip current boundary to the next one (worst case). */
         ptr = (void *) myMallocAlign(((uintptr_t)p + myMallocOffsetSize));
         // Calculates offset to the aligned (up) boundary
         ptrdiff_t offset = (uintptr_t)ptr - (uintptr_t)p;
```

```
            #ifdef myMallocDebug
               printf("aligned address after offset [%p].\n",(uintptr_t)ptr);
               printf("offset=%d, address to store offset = %p.\n", offset,
                        (uintptr_t)((myMallocOffset_t*)ptr - myMallocOffsetSize));
            #endif
            // Stores it always before the aligned pointer
            *((myMallocOffset_t*)ptr - myMallocOffsetSize) = offset;
         }
      }
      return ptr;
}

void my_free(void* ptr){
      if(ptr==NULL) return;
      // Recovers the stored offset
      myMallocOffset_t offset = *((myMallocOffset_t*)ptr-1);
      // Restores the malloc´s address pointer
      void* p=((myMallocOffset_t*)ptr) - offset;
      #ifdef myMallocDebug
         printf("free [offset=%d] : [address to free=%p].\n",offset,(uintptr_t)p);
      #endif
      // Frees allocated memory with a valid pointer
      free(p);
}

int main() {
      int8_t* p[myMallocIter];
      for(int i=0;i<myMallocIter;i++){
         p[i]=my_malloc(myMallocBytes);
         printf("Testing aligment : address[%d]=%p ==>
%s.\n",i,(uintptr_t)p[i],((uintptr_t)p[i]%myMallocAligment)?"NOT PASSED":"PASSED");
      }
      for(int i=0;i<10;i++){
         my_free((void*)p[i]);
      }
      return 0;
}
```

8. Resolve the problem.

Consider an interesting arithmetic feature. Let's take for example 4-digit numbers (in the decimal system), in which not all digits are the same.

There is the following transformation: take 4 decimal digits of the original number, and arrange in ascending order, this will be the number A. Next, arrange the numbers in descending order, we get the number B. Subtract the smaller one from the larger one, we get the number C. This will be the result of our transformation.

It is argued that if such a transformation is performed sequentially, then as a result, in no more than 7 iterations, such a number will be obtained, which, when transformed, turns into itself. We call such a number a fixed number.

This must be done for 3 and 4 digit numbers (which do not have all the same digits), all 4 digit numbers converge to a single fixed number. 3-digit ones, too, for a maximum of 6 iterations, to their fixed number.

Task:
Make a program that checks this assumption.
We set the capacity of the numbers, and check all the numbers.

Result:
Output whether there is a fixed number, what it is, or report that the test failed.

Desirable:
Print how many numbers converge to a fixed number, for 0, 1, 2, etc. iterations of our transformation.

It will be very good if the program finds several fixed numbers, if any, and reports if not all of the original numbers converge to fixed ones. This works, for example, for 6-digit numbers: there are several variants of fixed numbers to which the transformation can converge. And for many other original 6-digit numbers, the transformation does not converge.

SOLUTION (SOLUCIÓN)	
Source code in file.	interesting_arithmetic.c

Nombre	*Tirso Ramón Bello Ramos.*
Especialidad	*Ingeniero en Electrónica.*
Grado Científico	*Master en Ciencias en Automatización*
Publicación de Maestría	"Computador de a bordo: Una solución nacional" Vol, 42 No.1 del año 2021. http://scielo.sld.cu/scielo.php?script=sci_issuetoc&pid=1815-592820210001&lng=en&nrm=i Detalles de autor/a | Revista Ingeniería Electrónica, Automática y Comunicaciones ISSN: 1815-5928 (cujae.edu.cu)
No. ORCID	0000-0003-3947-3529
Linked-In	*www.linkedin.com/in/tirso-bello-5b3813257*
Correo Electrónico	tbr1968g@gmail.com