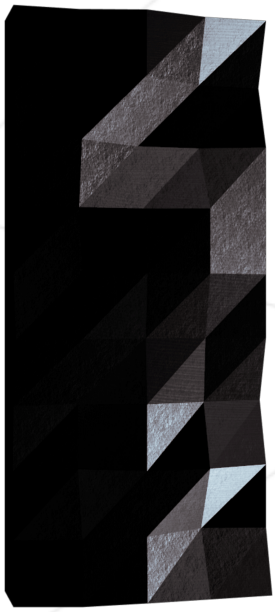


Starfleet - Day 04

Bit manipulation

Staff 42 pedago@42.fr

Summary: This document is the day04's subject for the Starfleet Piscine.

Contents

Chapter I

General rules

- Every instructions goes here regarding your piscine
- Turn-in directories are `ex00/`, `ex01/`, ..., `exn/`.
- You must read the examples thoroughly. They can contain requirements that are not obvious in the exercise's description.
- The exercises must be done in order. The evaluation will stop at the first failed exercise. Yes, the old school way.
- Read each exercise FULLY before starting it! Really, do it.
- The subject can be modified up to 4 hours before the final turn-in time.
- You will NOT be graded by a program, unless explictly stated in the subject. Therefore, you are given a certain amount of freedom in how you choose to do the exercises. However, some piscine day might explicitly cancel this rule, and you will have to respect directions and outputs perfectly.
- Only the requested files must be turned in and thus present on the repository during the peer-evaluation.
- Even if the subject of an exercise is short, it's worth spending some time on it to be absolutely sure you understand what's expected of you, and that you did it in the best possible way.
- By Odin, by Thor! Use your brain!!!

Chapter II

Day-specific rules

- If asked, you must turn-in a file named `bigo` describing the time and space complexity of your algorithm as below. You can add to it any additional explanations that you will find useful.

```
$> cat bigo
O(n) time , with n the number of elements in the array.
O(1) space
$>
```

- Your work must be written in `C`. You are allowed to use all functions from standard libraries.

- For each exercise, you must provide a file named main.c with all the tests required to attest that your functions are working as expected.

Chapter III

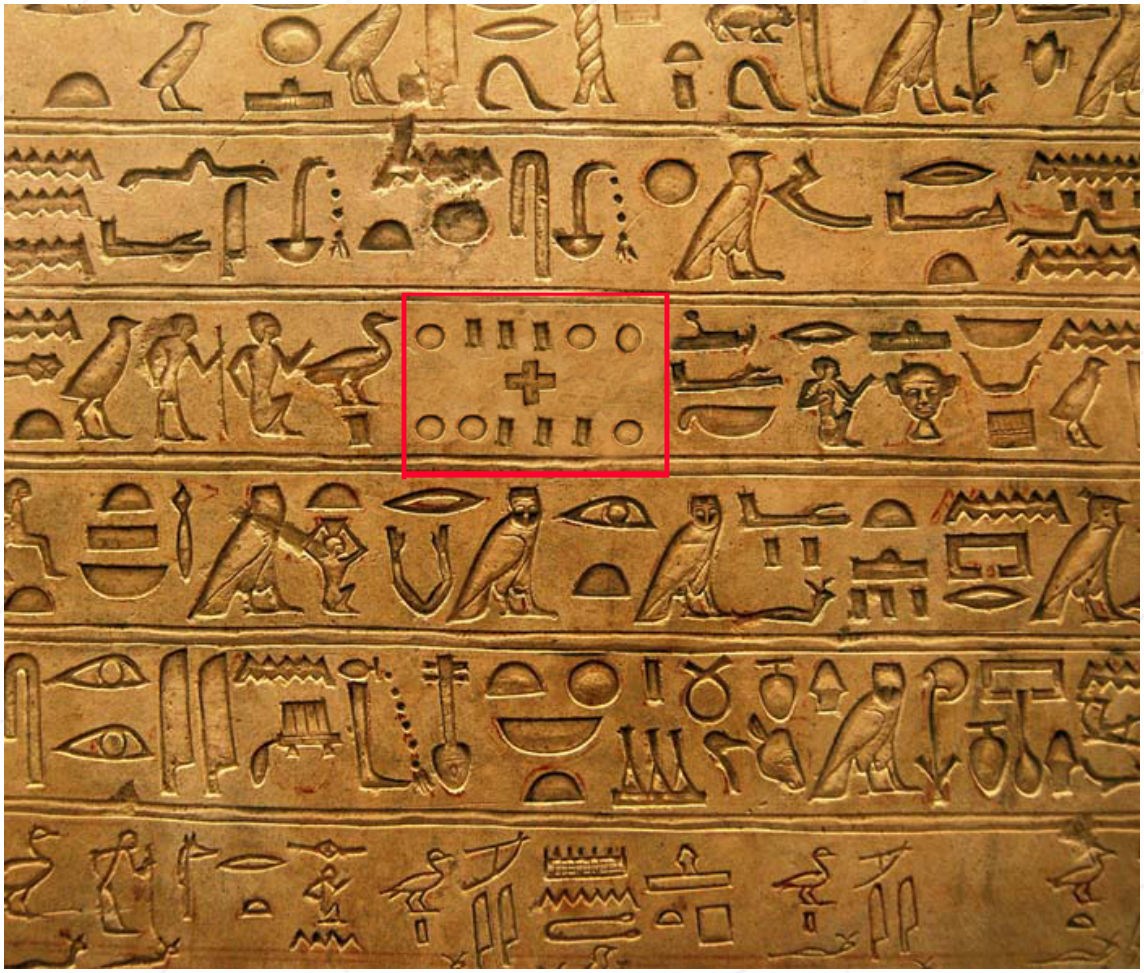
Exercise 00: What are these symbols?

	Exercise 00
Exercise 00: What are these symbols?	
Turn-in directory : $ex00/$	
Files to turn in : `decrypt.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

One day, a friend contacts you, a tremendous `archaeological discovery` is being made in `Egypt`, at the `Valley of the kings` and archaeologists need your help.

Once you are on the spot, you find yourself in a huge camp, researchers invite you to enter a cave. It is dark and the heat is stifling.

In the light of the `candle`, the researchers present you a dusty stone wall, at first normal, but when you get closer, hieroglyphics are inscribed there. Among all these `strange signs`, there are a series of symbols that you notice, which are quite familiar to you:



Researchers do not really know what it is. So, they would like you to help them solve this equation.

Your mission is to create a `program` able to return the `sum of the two binary representation numbers`.

```
$> compile decrypt.c
$> ./decrypt 000010 + 000001
000011 (3)
$> ./decrypt 000010 + 000010
000100 (4)
```

Info for resolving it:

- The binary representation will always have a length of 6 (000000 for the min and 111111 for the max).
- It will always represent a positive number.
- The sum will never be above 111111.
- you can modify the string passed as parameter, basically write in `a` or `b` the sum, then return it.

You have to create 2 functions:

The first one gets the sum of two binary strings passed as parameters (`a` and `b`):

```
char *getSum(char *a, char *b);
```

The second one converts a binary string given as parameter to an integer:

```
int toInt(char *bits);
```

You are not allowed to convert the binary strings to numbers (except for the toInt function, which purpose is only to display the number in the main)

If you are beginning with binary number, take time to understand it!

Chapter IV

Exercise 01: this is how works bit manipulation

	Exercise 01
Exercise 01: this is how works bit manipulation	
Turn-in directory : *ex*01/	
Files to turn in : `bit.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

Archaeologists still need you. They discovered new symbols. These symbols are '&', '|' and '~'.

Kindly, you decide to make them a small program able to give the result.

Examples with the symbol '&':

```
$> ./bit 0010 '&' 0000
0000 (0)
$> ./bit 0010 '&' 1111
0010 (2)
$> ./bit 0010 '&' 0010
0010 (2)
$> ./bit 0010 '&' 1101
0000 (0)
$> ./bit 0010 '&' '~1101'
0010 (2)
```

Examples with the symbol '|':

```
$> ./bit 0010 '|' 0000
0010 (2)
$> ./bit 0010 '|' 1111
1111 (15)
$> ./bit 0010 '|' 0010
0010 (2)
$> ./bit 0010 '|' 1101
1111 (15)
$> ./bit '~0010' '|' 1101
```

```
1101 (13)
```

Info for resolving it:

- The binary representation will always have a length of 4 (0000 for the min and 1111 for the max).
- It will always represent a `positive number`.
- a character '~' which is a binary negation, may be present at the begin of a binary string, you have to handle it.

You are not allowed to convert the binary strings to numbers (except for the toInt function, which purpose is only to display the number in the main)

You have to create 3 functions:

The first one returns a new string which is the product of an `and` operation ('&') between two strings:

```
char *getAnd(char *a, char *b);
```

The second one returns a new string which is the product of an `or` operation ('|') between two strings:

```
char *getOr(char *a, char *b);
```

Finally, the last function is the same as the previous exercise: it converts a binary string given as parameter to an integer:

```
int toInt(char *bits);
```

Take time to deeply understand these operations! Try these on paper! You should know it as well as '+' or '-' operations :)

Chapter V

Exercise 02: Archaeological shift

	Exercise 02
Exercise 02: Archaeological shift	
Turn-in directory : *ex*02/	
Files to turn in : `shift.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

`Archaeologists` are really grateful for your help and ask for your business card if they ever need your assistance in the future.

You feel `flaterred`, but you cannot decently leave them like this : they are so `ignorant about bit manipulation` !

You insist on staying and show them symbols they haven't even paid attention to : '»' and '«'.

As you feel they are not `ready` to understand the concept of `arithmetic shift`, you decide to create a little program for them.

Examples:

```
$> compile shift.c
$> ./shift 000011 '<<' 2
001100 (12)
$> ./shift 101000 '>>' 3
111101 (-3)
$> ./shift 110100 '>>' 5
111111 (-1)
```

Info for resolving it:

- The binary representation will always have a length of 6 (`000000` to `111111`).
- It can represent a `positive` or `negative` number (the leftmost bit is the sign bit).

- Unlike what the name of the exercise may imply, you have to implement an `arithmetic shift` !

You are not allowed to convert the binary strings to numbers (except for the toInt function, which purpose is only to display the number in the main)

You have to create 3 functions:

The first function applies an arithmetical `right` shift given as parameter a binary string `bin` and an integer `k` which is the number of shifts.

```
char *rightShift(char *bin, int k);
```

The second function applies an arithmetical `left` shift given as parameter a binary string `bin` and an integer `k` which is the number of shifts.

```
char *leftShift(char *bin, int k);
```

The last function is the same as the previous exercise, cast a binary string to `int`, except that this time it has to handle negative numbers:

```
int toInt(char *bits);
```

Chapter VI

Exercise 03: Is it a hat ?

	Exercise 03
Exercise 03: Is it a hat ?	
Turn-in directory : *ex*03/	
Files to turn in : `xor.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

As the `archaeologists` are getting more and more annoyed by your expertise, you take it upon yourself to create a little program to help them understand one of the symbols they will never notice : '^'.

Examples:

```
$> compile xor.c main.c
$> ./xor 111111 ^ 010101
101010 (42)
$> ./xor 101010 ^ 000000
101010 (42)
$> ./xor 101010 ^ 101010
000000 (0)
$>
```

Info for resolving it:

- The binary representation will always have a length of 6 (000000 for the min and 111111 for the max).
- It will always represent a `positive` number.

`You are not allowed to convert the binary strings to numbers (except for the toInt function, which purpose is only to display the number in the main)`

You have to create 2 functions:

The first function returns a `xor` product of two binary strings given as parameters:

```
char *getXor(char *a, char *b);
```

The second is the same as the previous exercise, cast a binary string to `int`:

```
int toInt(char *bits);
```

Chapter VII

Exercise 04: Vacancy

	Exercise 04
Exercise 04: Vacancy	
Turn-in directory : $ex04/$	
Files to turn in : `getPlace.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

After you finish helping the archaeologists at the `Valley of the kings`, you end up in charge of a car parking (long story about unpaid parking bills).

You find that today is very quiet, no client is coming, so you decide to create a software which can get information about the parking.

The parking is organized in rows of `20 parking spaces`. You think that will be a great idea to install some little embedded machine near each row able to tell whether the parking spaces are occupied or not!

Make a function that takes as parameter a row and say if, at a specified position, the place is taken or not. (1 = place taken, 0 = vacant place) The position index starts at 0 and ends at 19.

Each row is represented by an integer. If a place is occupied the corresponding bit is set. The position with index 0 corresponds to the `rightmost bit`.

```
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1
```

In the above example, at the place 0 and 5 there are some cars!

You have to implement the following function, that takes as parameters the parking row, and the position where we want to know :

```
int getPlace(unsigned int parkingRow, int pos);
```

You are not allowed to use any other operators than the bitwise operators. AND loops are forbidden!

Examples:

```
$> compile getPlace.c
$> #here 42 is '0000000000000101010'
$> ./getPlace 42 0
Parking place 0: vacant
$> ./getPlace 42 1
Parking place 0: occupied
$> ./getPlace 42 2
Parking place 0: vacant
$> ./getPlace 42 3
Parking place 0: occupied
$>
```

Chapter VIII

Exercise 05: The way out

	Exercise 05
Exercise 05: The way out	
Turn-in directory : $ex05/$	
Files to turn in : `clearPlace.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

Your little embedded machine works fine, but some features are missing...

Oh, a customer is just leaving his place!

That's the perfect time to implement a new functionnality!

Given a parking row organized in bits, and a position (`pos`) in this row, unset the corresponding bit:

```
unsigned int clearPlace(unsigned int parkingRow, int pos);
```

You are not allowed to use any other operators than the bitwise operators. AND loops are forbidden!

```
$> compile clearPlace.c
$> ./clearPlace 42 3
New parking row: 34
$> ./clearPlace 42 5
New parking row: 10
$>
```

Chapter IX

Exercise 06: The way in

	Exercise 06
Exercise 06: The way in	
Turn-in directory : $ex06/$	
Files to turn in : `setPlace.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

5 days before your debt is done! Sitting at the reception, you're now seeing a beautiful old car from the 60's.

Oh, wait! The car just parked at one of the parking places, and the embedded machine didn't notice it!

It's because another functionnality is missing: `setPlace` !

Given the parking row, implement a function that sets at a given position the bit at 1:

```
unsigned int setPlace(unsigned int parkingRow, int pos);
```

You are not allowed to use any other operators than the bitwise operators. AND loops are forbidden!

```
$> compile setPlace.c
$> ./setPlace 42 0
New parking row: 44
$> ./setPlace 42 2
New parking row: 46
$>
```

Chapter X

Exercise 07: We can do better

	Exercise 07
Exercise 07: We can do better	
Turn-in directory : $ex07/$	
Files to turn in : `updatePlace.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

Ok, now, everytime a car comes, the `setPlace` function is called, and everytime a car leaves its place, the `clearPlace` function is called.

You wonder if you can do better: do it in one function, with the `value` to assign to a bit, given as parameter.

Implement the following function:

```
unsigned int updatePlace(unsigned int parkingRow, int pos, int value);
```

You are not allowed to use any other operators than the bitwise operators. AND loops are forbidden!

```
$> compile updatePlace.c
$> ./updatePlace 42 0 1
Updated parking row: 43
$> ./updatePlace 42 2 1
Updated parking row: 46
$>
```

Chapter XI

Exercise 08: Consecutive places

	Exercise 08
Exercise 08: Consecutive places	
Turn-in directory : *ex*08/	
Files to turn in : `isFilled.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

When a new car arrives, it has to park at the rightmost available place of the row (if the row is empty it is place 0).

As cars come and go as they please, the rows are not always filled from right lo left.

Implement a function to check if a row is filled from right to left :

```
int isFilled(unsigned int parkingRow); // 1 = TRUE, 0 = FALSE
```

You are not allowed to use any other operators than the bitwise operators. AND loops are forbidden!

```
$> compile isFilled.c
$> ./isFilled 42
Parking row 42 is not filled from right to left
$> ./isFilled 15
Parking row 15 is filled from right to left
$> ./isFilled 1024
Parking row 1024 is not filled from right to left
$> ./isFilled 1023
Parking row 1023 is filled from right to left
$>
```

EXAMPLE in binary representation:

```
Parking row filled from right to left :       000111111
Parking row NOT filled from right to left :   000111011
```

Chapter XII

Exercise 09: Land use

	Exercise 09
Exercise 09: Land use	
Turn-in directory : $ex09/$	
Files to turn in : `occupiedPlaces.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

You now want to put a sign that says the number of remaining spaces for each row. But before you do that, you have to know the number of occupied spaces !

Make a program able to count the number of occupied spaces.

You tell yourself : "It's easy! I'll go through every place and see if the place is occupied or not!"

However, the small controller does not appreciate this kind of expenses. Your algorithm must be in O(m) time complexity, where m is the number of occupied places (active bits).

Implement the following function:

```
int occupiedPlaces(unsigned int parkingRow);
```

You are not allowed to use any other operators than the bitwise operators. (Yeah, we only like bitwise operators here :))

```
$> compile occupiedPlaces.c
$> ./occupiedPlaces 42
```

```
Parking row 42 has 3 occupied places
$> ./occupiedPlaces 15
Parking row 15 has 4 occupied places
$> ./occupiedPlaces 1024
Parking row 1024 has 1 occupied places
$> ./occupiedPlaces 1023
Parking row 1023 has 10 occupied places
$>
```

Chapter XIII

Exercise 10: Dude, what is this car?

	Exercise 10
Exercise 10: Dude, what is this car?	
Turn-in directory : $ex10/$	
Files to turn in : `carPosition.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

Before you leave work, you see that there is only one car left in the parking lot.

You want to know some information about this car, your goal is to find the position of this car.

Implement the following function:

```
int carPosition(unsigned int parkingRow);
```

If the row has more or less than 1 car, the function returns -1.

Again, you are not allowed to use any other operators than the bitwise operators.

```
$> compile occupiedPlaces.c
$> ./occupiedPlaces 0
Parking row 0 has 1 car at position -1
$> ./occupiedPlaces 1
Parking row 1 has 1 car at position 0
$> ./occupiedPlaces 16
Parking row 16 has 1 car at position 4
$> ./occupiedPlaces 1024
Parking row 1024 has 1 car at position 10
```

```
$>
```

Chapter XIV

Exercise 11: Dude, what is this car? 2

	Exercise 11
Exercise 11: Dude, what is this car? 2	
Turn-in directory : $ex11/$	
Files to turn in : `carPosition.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

You have found the car position, well done, but the embedded machine prefers that you use a `mathematical function`.

```
int carPosition(unsigned int parkingRow);
```

```
$> compile occupiedPlaces.c
$> ./occupiedPlaces 0
Parking row 0 has 1 car at position -1
$> ./occupiedPlaces 1
Parking row 1 has 1 car at position 0
$> ./occupiedPlaces 16
Parking row 16 has 1 car at position 4
$> ./occupiedPlaces 1024
Parking row 1024 has 1 car at position 10
$>
```

Chapter XV

Exercise 12: Make room

	Exercise 12
Exercise 12: Make room	
Turn-in directory : $ex12/$	
Files to turn in : `clearBits.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

The cleaning team will now clear some part of the parking row, so they put some cars out during the night.

Your embedded machine didn't notice a thing. You have to correct that!

Given as parameters an unsigned integer, which represents the parking row, and `n` a position, clear the sequence of bits from 0 to n where `n` is the position of the last bit to be cleared.

Implement the following function:

```
unsigned int clearBits(unsigned int parkingRow, int n);
```

EXAMPLE :

```
Input  : parkingRow = 1101, n = 3
Output : parkingRow = 1000
```

```
$> compile clearBits.c
$> ./clearBits 42 4
Cleared parking row: 32
$> ./clearBits 1023 5
Cleared parking row: 992
$> ./clearBits 1023 9
Cleared parking row: 512
$> ./clearBits 367 6
Cleared parking row: 320
$>
```

Chapter XVI

Exercise 13: Criminal in his car

	Exercise 13
Exercise 13: Criminal in his car	
Turn-in directory : $ex13/$	
Files to turn in : `leftmostCar.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

One day the police calls you, they just discovered that there is a criminal inside one of the cars at the parking and he is about to leave! They want your help to catch him.

They just told you that his car is the leftmost car of one parking row. Oh no, that just gives you an idea about bit manipulation...

Implement a function which returns the position of the leftmost car (don't worry about the criminal, they will catch him) :

```
int leftmostCar(unsigned int parkingRow);
```

If the parking row is empty, the function returns -1.

EXAMPLE :

```
Input  : 00101010
Output : 5
```

Again, again, You are not allowed to use any other operators than the bitwise operators.

```
$> compile leftmostCar.c
$> ./leftmostCar 42
Parking row 42: the leftmost car is at position 5
$> ./leftmostCar 15
Parking row 15: the leftmost car is at position 3
$> ./leftmostCar 1024
Parking row 1024: the leftmost car is at position 10
$> ./leftmostCar 1023
Parking row 1023: the leftmost car is at position 9
$> ./leftmostCar 1022
Parking row 1022: the leftmost car is at position 9
$> ./leftmostCar 31
Parking row 31: the leftmost car is at position 4
$>
```

Chapter XVII

Exercise 14: Criminal in his car 2

	Exercise 14
Exercise 14: Criminal in his car 2	
Turn-in directory : $ex14/$	
Files to turn in : `rightmostCar.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

In fact the police was wrong, the criminal's car is not the leftmost car of the parking row, but the rightmost.

They are really counting on you this time, don't disapoint them and catch him!

Unfortunately, we all know that you prefer doing some bit manipulation right now...

Implement a function which returns the position of the rightmost car.

```
int rightmostCar(unsigned int parkingRow);
```

If the parking row is empty, the function returns -1.

You are not allowed to use any other operators than the bitwise operators.

```
$> compile rightmostCar.c
$> ./rightmostCar 42
Parking row 42: the rightmost car is at position 1
$> ./rightmostCar 15
Parking row 15: the rightmost car is at position 0
$> ./rightmostCar 1024
Parking row 1024: the rightmost car is at position 10
```

```
$> ./rightmostCar 1023
Parking row 1023: the rightmost car is at position 0
$> ./rightmostCar 1022
Parking row 1022: the rightmost car is at position 1
$> ./rightmostCar 352
Parking row 352: the rightmost car is at position 5
$>
```

The function ffs() is forbidden !

Chapter XVIII

Exercise 15: Longest sequence

	Exercise 15
Exercise 15: Longest sequence	
Turn-in directory : $ex15/$	
Files to turn in : `longestSequence.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

Without telling you, a group of people organized a car contest on your parking! They put on a row a `sequence of cars` and are now having a party! These hooligans, you need to know how many cars are present!

Implement a function which returns the number of cars in the longest sequence of consecutive cars in a row.

```
int longestSequence(unsigned int parkingRow);
```

You are not allowed to use any other operators than the bitwise operators.

You have to do it in O(s) time, where s is the length of the longest consecutive sequence of bits.

EXAMPLE in binary representation:

```
Parking row : 100111011000
The number of cars in the longest sequence of consecutive cars : 3
```

```
$> compile longestSequence.c
$> ./longestSequence 42
Parking row 42: the longest sequence has 1 car(s)
$> ./longestSequence 15
Parking row 15: the longest sequence has 4 car(s)
$> ./longestSequence 1024
Parking row 1024: the longest sequence has 1 car(s)
```

```
$> ./longestSequence 1023
Parking row 1023: the longest sequence has 10 car(s)
$> ./longestSequence 1022
Parking row 1022: the longest sequence has 9 car(s)
$> ./longestSequence 352
Parking row 352: the longest sequence has 2 car(s)
$>
```

Chapter XIX

Exercise 16: Piano

	Exercise 16
Exercise 16: Piano	
Turn-in directory : $ex16/$	
Files to turn in : `piano.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

You paid your debt, you can finally leave the parking!

So after all you have lived these last few days, today, you want to relax, doing nothing...

Unfortunately, you have an Interview Piscine... Hum I mean, your `nephew` has no one to take care of him this afternoon, so you have to babysit him!

To get some peace, you decide to make him a little program to make piano song!

You decide to create a format file using a bit array: but you forgot to decompress it!

Now you have to decompress it I guess...

Given the following structure:

```
struct s_bit {
        int *arr;
        int n;
};
```

Create a function that reads a bit array, given as parameters a bit array structure (which holds the array `arr`, and its number of elements `n`) and the length of a row, and returns an integer matrix.

```
int         **pianoDecompress(struct s_bit *bit, int l);
```

```
$> compile decompress.c
$> cat song1.piano
32
1 1 1 1 2 2 2 2 2 4 4 4 4 8 8 8 8 16 16 16 16
$> ./decompress song1.piano
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$> ./decompress song1.piano | piano/play
```

Chapter XX

Exercise 17: Mozart Nephew

	Exercise 17
Exercise 17: Mozart Nephew	
Turn-in directory : $ex17/$	
Files to turn in : `correctSong.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

You have created your little program, your nephew is now creating his song and takes a lot of pleasure!

It turns out that your nephew is a little Mozart, and has created an amazing song!

He now wants to send it to granny, but he forgot one `note` on his masterpiece.

Given a bit array with its length (`n`), the length of a row, the position of the beginning of the note (the `row` and the `column`), and the length of the note,

implement a function able to complete your nephew masterpiece:

```
void        correctSong(struct s_bit *bit, int l, int row, int col, int dist);
```

```
$> compile correctSong.c
$> cat empty.piano
32
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
$> ./correctSong empty.piano 0 1 3
32
2 2 2 0 0 0 0 0 0 0 0 0 0 0 0 0
$> ./correctSong empty.piano 3 3 6
32
0 0 0 8 8 8 8 8 0 0 0 0 0 0 0 0
$> ./correctSong nephewMasterPiece.piano
```

Chapter XXI

Exercise 18: Why don't my cans have the same weight?

	Exercise 18
Exercise 18: Why don't my cans have the same weight?	
Turn-in directory : *ex*18/	
Files to turn in : `isEqual.c main.c header.h`	
Allowed functions : `all`	
Notes : `n/a`	

One day, you buy from the supermarket 2 cans of soup. You have the impression that they don't have the same weight, so you decide to weight them to check!

After you weight them, it's the stupor! They don't have the same weight! You decide to go the factory to complain, but before you wonder:

Using bitwise manipulation, how do I know that these two weights are equal?

Given as parameters two integers `a` and `b`, return `0` if they are equal, otherwise return a number different than 0 :

```
int        isEqual(int a, int b);
```

You can use only bitwise operators, no arithmetic operator ('+', '-', ...), no branching ('if' , 'else', ...), no loop, no relational operators ('==', '!=', ...), no logical operators ('&&', '||', ...)!

Examples :

```
$> compile isEqual.c
$> ./isEqual 11 11
0
$> ./isEqual 10 12
6
```

Chapter XXII

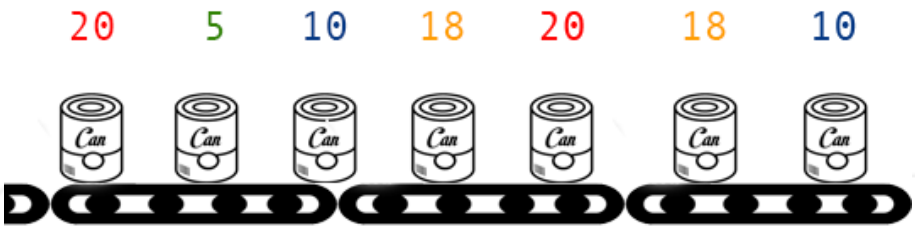
Exercise 19: The can factory!

	Exercise 19
Exercise 19: The can factory!	
Turn-in directory : $ex19/$	
Files to turn in : `aloneCan.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

You decide to go to the factory to complain about the 2 cans that you bought!

At the factory, they are totally sorry, and explain to you what happened :

During the production, cans are produced by `two` : in a production chain you will always find two cans with the same weight. Recently, the system has been failing : there is now a unique can in each production chain.



In the above example, the unique can is the can with weight `'5'`.

Now you are wondering: how will I find the `unique can` using bit manipulation?

Given as parameters an array of integers, where all the elements are repeating twice except one, print the unique number, followed by a new line.

```
void        aloneCan(int *arr, int n);
```

The input will always be valid, you don't have to worry about invalid input.

Your function has to run in O(n) time, and O(1) space, where n is the length of the array.

Example using the main:

```
$> compile aloneCan.c
$> ./aloneCan 20 5 10 18 20 18 10
5
$> ./aloneCan 10 3 10
3
```

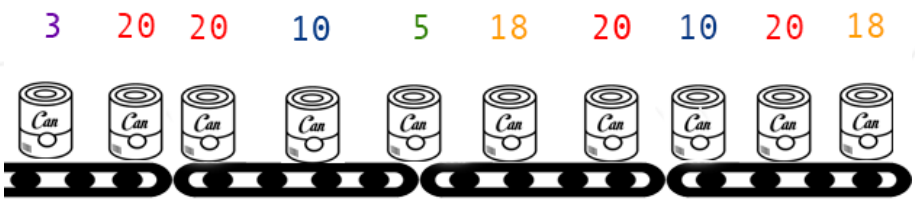
Chapter XXIII

Exercise 20: Big problem at the can factory!

	Exercise 20
Exercise 20: Big problem at the can factory!	
Turn-in directory : *ex*20/	
Files to turn in : `aloneCans.c main.c header.h bigo`	
Allowed functions : `all`	
Notes : `n/a`	

Oh no! The factory had a big trouble, and now the production of cans is messed up:

There is now a chain of cans where there is two cans which have a unique weight, and all the other cans are present an `even` number of times.



In the examples above, there is only one can of weight '5' and one can of weight '3'.

Your mission is to find the two cans which are unique on the chain!

Given as parameters `'arr'` an array of integers, and `'n'` its length, print the two unique integers present in the array.

```
void aloneCans(int *arr, int n);
```

The input will always be valid (array of integers where all the elements are repeating an even number of times except two), you don't have to worry about invalid input.

Your function has to run in O(n) time and O(1) space.

Example using the main:

```
$> compile aloneCans.c
$> ./aloneCans 3 20 20 10 5 18 20 10 20 18
5
3
$> ./aloneCans 1 2 2 3
1
3
$> ./aloneCans 1 2 2 3 2 2 2 2
1
3
```

The order of printing is not important.