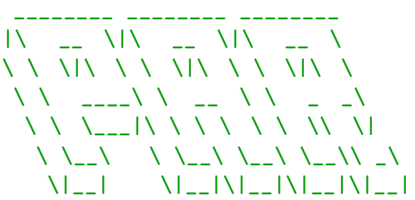
# DM 2

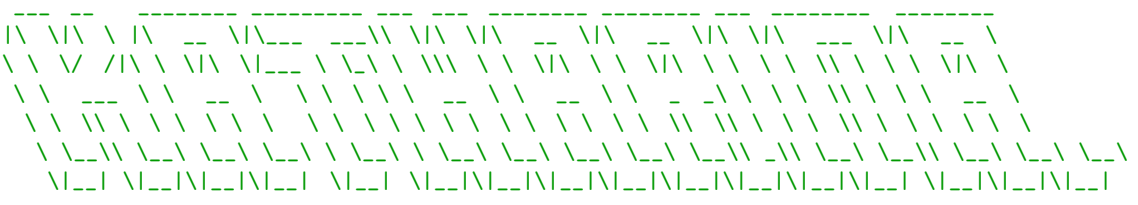
Katharina, Olympe, Gabriel et Lukas

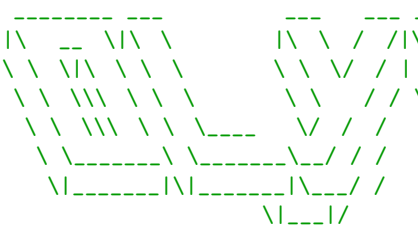
5 novembre 2024

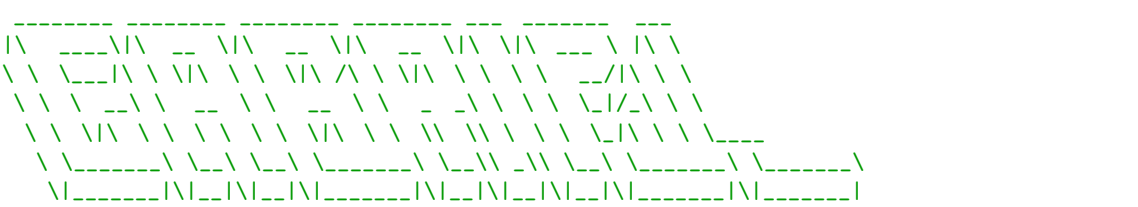
## 1 Code

```
/*
```









```
 ___      ___ ___ ___ __    ________ ________
|\  \    |\  \|\  \|\  \|\  \ |\  __  \|\   ____\
\ \  \   \ \  \\\  \ \  \/  /|\ \  \|\  \ \  \___|_
 \ \  \   \ \  \\\  \ \   ___  \ \   __  \ \_____  \
  \ \  \___\ \  \\\  \ \  \\ \  \ \  \ \  \|____|\  \
   \ \_______\ \_______\ \__\\ \__\ \__\ \__\____\_\  \
    \|_______|\|_______|\|__| \|__|\|__|\|__|\_________\
                                          \|_________|


DM - 2
Par Katharina, Olympe, Gabriel et Lukas

Code source, progression et autre: https://github.com/gnaboo/DM_REICHERT/DM2
Mention speciale a Ezequiel et Cyprien, avec qui nous avons pas mal discute de la
    formulation
(parfois un peu alembique) des enonces.

Fait avec amour en C (bien meilleur que le OCaml)

*/
```

```c
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
#include <assert.h>
#include <stdlib.h>
#include <string.h>


// Exo 20
int spitze(bool* tab, int size) {
    assert(size > 0);

    bool first = tab[0];
    for(int i = 1; i<size; i += 1) {
        if(tab[i] != first) return i-1;
    }
    return 1; // s'il n'y a pas d'elements differents
}

// Exo 21
bool nul(bool *t, int size)
{
    int tr=0;
    int fa=0;

    int last_tr= 0;
    for (int i=0; i<size; i+=1)
    {
        if (t[i]) last_tr=i;
    }

    int j=0;
    while (j<last_tr)
    {
        j+=1;
        if(t[j]) tr+=1;
        else fa +=1;

        if (fa>tr) return false;
    }
    return true;
}
```

```c
// Exo 22
double c_abs(double a) {
    if(a < 0) return -a;
    return a;
}

double modfloat(double a, double b) {
    if(b == 0) exit(133);
    int sgn = b > 0 ? 1 : -1;

    if(c_abs(b) > c_abs(a)) return sgn*a; // fonctionnement similaire au a%b sur python

    int c = (int) a/b;
    return a - (c*b);
}


// Exo 23
int* minmax(int* tab, int size) {
    assert(size > 0);

    int max = tab[0];
    int min = max; // tab[0];
    for(int i = 0; i<size; i += 1) {
        if(tab[i] > max) max = tab[i];
        if(tab[i] < min) min = tab[i];
    }
    int* ret = malloc(sizeof(int)*2);
    ret[0] = min;
    ret[1] = max;
    return ret;
}

// Exo 24
int medianemax(int *t, int size)
{
    int max=t[0];
    for (int i=1; i<size;i+=1)
    {
        if (t[i]>max) max=t[i];
    }

    int *indices = malloc(size*sizeof(int));
    int count = 0;
    for (int i = 0; i<size; i+=1)
    {
        if (t[i]==max)
        {
            indices[count]=i;
            count+=1;
        }
    }

    int mediane;
    if (count%2 == 0) {
        mediane=indices[(count/2)-1];
    }
    else {
        mediane=indices[count/2];
    }
    free(indices);
    return mediane;
}
```

```c
// Exo 25
void derive(double* tab, int size) {
    for(int i = 1; i < size-1; i += 1) {
        tab[i - 1] = i * tab[i];
    }
    if(size > 0) tab[size - 1] = 0;
}

// Exo 26
double* convolve(double* a, int size_a, double* b, int size_b) { // technically not a
     convolution but a cross-correlation
    int result_size = size_a + size_b - 1;
    double* result = malloc(result_size * sizeof(double));
    for(int i = 0; i < result_size; i += 1) {
        result[i] = 0.0;
    }

    for(int i = 0; i < size_a; i += 1) {
        for(int j = 0; j < size_b; j += 1) {
            result[i+j] += a[i]*b[j];
        }
    }
    return result;
}

double* multpol(double* tab, double* tab2, int size, int size2) {
    return convolve(tab, size, tab2, size2); // multiplying a polynomial is equivalent to
         the convolution of the coefficiants
}



bool is_lowercase(char car)
{
    if (car >= 'a' && car <= 'z')
        return true;
    return false;
}

bool is_uppercase(char car)
{
    if (car >= 'A' && car <= 'Z')
        return true;
    return false;
}

bool is_letter(char car)
{
    return (is_lowercase(car) || is_uppercase(car));
}


// from TP1
int PGCD(int a, int b) {
    int c = a;
    if(a == b) return abs(a);
    if(a < b) {
        c = b;
        b = a;
    }
    while (a != 0) {
        a = c % b; // r = a, soit c = b * x + a
        c = b;
        b = a;
    }
    return c;
```

```
}

// Exo 27
char* cesar(char* string, int a, int b) {
    assert(a != 0 && PGCD(a, 26) == 1); // la fonction est bijective
    // verifie que a et 26 sont premiers entre eux (parce que pourquoi pas)

    int size = (int) strlen(string);
    char* new_string = malloc(sizeof(char) * (size+1));

    for(int i = 0; i<size; i += 1) {
        if(!is_letter(string[i])) new_string[i] = string[i];
        else if(is_lowercase(string[i])) new_string[i] = 'a'-1 + ((a * (string[i]+1-'a') +
            b) % 26 + 26) % 26;
        else new_string[i] = 64+((a * (string[i]+1 - 'A') + b) % 26 + 26) % 26;
    }
    new_string[size] = '\0';
    return new_string;
}



// Exo 28
char* auguste(int nombre) {
    assert((nombre >= 1) && (nombre <= 3999));

    int value[] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};
    char* symbols[] = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV",
        "I"};

    char* result = malloc(sizeof(char) * 20);
    for(int i = 0; i<sizeof(char)*19; i += 1) { result[i] = '\0'; }

    for(int i = 0; i<13; i += 1) {
        while(nombre >= value[i]) {
            strcat(result, symbols[i]);
            nombre -= value[i];
        }
    }
    //strcat(result, '\0'); with `strcat()`, the resulting string in dest is always
        null-terminated. source: https://linux.die.net/man/3/strcat

    return result;
}



// Exo 29
bool in_list(int* tab, int size, int element) {
    for(int i = 0; i<size; i += 1) {
        if(tab[i] == element) return true;
    }
    return false;
}

int first_iteration(int* tab, int size, int element) {
    for(int i = 0; i<size; i += 1) {
        if(tab[i] == element) return i;
    }
    return -1;
}

int min(int* tab, int size) {
    int _min = tab[0];
    for(int i = 0; i<size; i += 1) {
```

```
        if(tab[i] < _min) {
            _min = tab[i];
        }
    }
    return _min;
}

int* ecart(int* tab, int size) {
    int* repertoire = malloc(sizeof(int) * size);
    for(int i = 0; i<size; i += 1) { repertoire[i] = 0; }

    for(int i = 0; i<size; i += 1) {
        int indice = first_iteration(tab, size, tab[i]);
        repertoire[indice] += 1;
    }

    for(int i = 0; i<size; i += 1) {
        if(repertoire[i] == 0) repertoire[i] = size + 1; // exclure les 0 places par la
            fonction precedente
    }

    int min_1 = min(repertoire, size);

    int min_1_indice = first_iteration(repertoire, size, min_1);
    repertoire[min_1_indice] = size + 2; // new max

    int min_2 = min(repertoire, size);
    int min_2_indice = first_iteration(repertoire, size, min_2);

    free(repertoire);

    int* return_value = malloc(sizeof(int) * 2);

    if(min_1 > 1 && min_2 > 1) min_2_indice = min_1_indice;

    return_value[0] = tab[min_1_indice];
    return_value[1] = tab[min_2_indice];

    return return_value;
}




int main() {

    // moins il y a de code, moins il y a de risques de perdre des points
    // (surtout en oubliant de free des resultats de fonctions...)

    return 0;
}
```