Correction de TP
Programation Impérative I

Avec le peuple, par le peuple, pour le peuple

Table des matières

1 Premiers programmes

1.1 Première compilation

On a ici le programme :

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

int main(){

   int a, b, c;
   a= 2;
   b= 3 * a;
   c= a + b;
   printf ("bonjour: %d + %d = %d\n", a, b, c);
   exit(0);
}
```

1. Je pense qu'il serait succin de vous apprendre à créer ainsi qu'à ouvrir un dossier ou un fichier sur Linux.
   Il en est de même pour **emacs**, bien que je préfère personnellement le saint **Atom**.
2. Le programme est juste, on exécute le programme.
3. Rien à faire, mais retenez bien qu'il faut savoir un minimum ce que toutes les options de `gcc -W -Wall -std=c89 -pedantic -O3 bonjour.c -o bonjour` font.

1.2 Affectations

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

int main(){
  int a, b, c, d;
  a = 5;
  b = -4;
  c = 3;
  d = a+b;
  b = d;
  d = a+b;
  printf("a = %d, b = %d, c = %d, d = %d\n", a, b, d, c);
  exit(0);
}
```

tableau demandé :

a	b	c	d
∅	∅	∅	∅
5	∅	∅	∅
5	-4	∅	∅
5	-4	3	∅
5	-4	3	1
5	1	3	1
5	1	3	6

1.3 Calculs simples

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

int main(){
  int a, b;
  while(scanf("%d %d", &a, &b) != 2);

  printf("%d x %d = %d\n", a, b, a*b);
  if(b != 0)
    printf("%d / %d = %d, reste = %d\n", a, b, a/b, a%b);
  exit(0);
}
```

Ici, on utilise pour la première fois la fonction scanf(), scanf peut permettre de saisir des informations au clavier.
scanf retourne un entier équivalent au nombre de valeur qu'il a saisis.

Ici, l'idéal serait qu'il retourne deux, mais il est **impératif** de toujours le vérifier avec un `while` : `while(scanf("...", ...) != nb)` (où nb est le nombre de valeur voulant être saisi).

1.4 Permutation

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

int main(){
  float x, y, z;
  while (scanf("%f %f %f", &x, &y, &z) != 3);
  printf("x = %f, y = %f, z = %f\n\n", x, y, z);

  x += y;
  y = x-y;
  x -= y;
  printf("permutation : x = %f, y = %f\n\n", x, y);

  x += z;
  z = x-z;
  x -= z;
  y += z;
  z = y-z;
  y -= z;
  printf("permutation circulaire : x = %f, y = %f, z = %f\n", x, y, z);
  exit(0);
}
```

On utilise ici deux techniques qui sont celle de la permutation et de la permutation circulaire, toutes deux **sans variable intermédiaire**.

1.5 Un simple tirage aléatoire (entiers)

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

int main(){
  int x, y, z;
  srand(time(NULL));

  x = rand()%100;
  y = rand()%100;
  z = rand()%100;
  printf("x = %d, y = %d, z = %d\n", x, y, z);

  x += z;
  z = x-z;
  x -= z;
  y += z;
  z = y-z;
  y -= z;
  printf("permutation circulaire : x = %d, y = %d, z = %d\n", x, y, z);
  exit(0);
}
```

On utilise ici la bibliothèque `time.h` pour générer de l'aléatoire ou en tout cas ce qui s'en rapproche. On met écrit donc la ligne `srand(time(NULL))` pour pouvoir utiliser la fonction `rand()`.

On génére un entier random modulo 100 pour obtenir un entier dans l'intervalle $[0;100[$.

On réutilise aussi la permutation circulaire sans variable intermédiaire vue dans la question d'avant.

1.6 Pair ou Impair ?

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

int main(){
  int a;
  print("Veuillez entrer un entier\n");
  if(scanf("%d",&a)==1){
    if(a%2==0){
       printf("%d est pair\n",a);
    }
    else{
      printf("%d est impair\n",a);
    }
  }
  else{
    printf("Erreur, vous n'avez pas entrez un entier\n");
  }
  exit(0);
}
```

Ici on veut regarder si un nombre est pair, il suffit de voir si ce nombre modulo 2 est égal à 0 pour voir qu'il est, sinon, c'est qu'il est impair.

On vérifie bien entendu que ce qude nous traitons est bien un nombre en vérifiant si le scanf à retourné 1.

1.7 Expressions logiques

Pas de code ici mais amis mais des expressions logiques dont le résultat est à determiner
En considérant que a = 2, b = 13, c = 3,

1. `(a + b)!= c` : `(2 + 13) != 3` -> 1 car 15 n'est pas égal à 3
2. `!(a == c)` : `!(2 == 3)` -> 1 car 2 n'est pas égal à 3
3. `((b - a) >= c) && (a <= (c - b))` :
   `((13 - 2) >= 3) && (2 <=(3 - 13))` :
   `1 && 0` -> 0 car 15 est supérieur ou égal à 3 mais que 2 est supérieur à -10
4. `((6 * a + 1) >= b) && (c == (b%5))` :
   `((6 * 2 + 1) >= 13) && (3 == (13 % 5))` :
   `1 && 1` -> 1 car 13 est supérieur ou égal à 13 et que 3 est égal à 13 modulo 5
5. `(b/c) < (b/3.0)` :
   `(13/3) < (13/3.0)` -> 1 car 4 < 4.333333
6. `(((b + c + 3)/9) == (a + 1)) || (((b - 5)% a) > 0)` :
   `(((13 + 3 + 3)/9) == (2 + 1)) || (((13 - 5)% 2) > 0)` :
   `0 || 0` -> 0 car 2 n'est pas égal à 3 et car 0 n'est pas supérieur à 0

Rappellons qu'en C, la valeur **1** est renvoyée par une expression logique vraie, sinon c'est la valeur **0** qui sera renvoyée.

1.8 Années bissextiles

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

int main(){
  int n;
  float r = -5;

  printf("Un réel ? : ");
  if(scanf("%f", &r) != 1){
    printf("Pas un réel\n");
    exit(-1);
  }

  if((-5 < r && r <= 2) || (5 <= r && r <= 10))
    printf("Vérifié\n");

  printf("Un entier ? : ");
  if(scanf("%d", &n) != 1){
    printf("Pas un entier\n");
    exit(-1);
  }

  if ((n%4 == 0 && n%100 != 0) || (n%400 == 0))
    printf("L'année rentrée (%d) est bisextile\n", n);
  else
    printf("L'année rentrée (%d) n'est pas bisextile\n", n);
}
```

Léger rappel sur la fonction `exit()` fournie par la bibliothèque `stdlib.h` :
l'argument de exit est purement arbitraire, il correspond en réalité à des conventions
dont je vous laisse prendre connaissance par le biais d'internet, A.K.A. **le meilleur ami de l'informaticien.**

1.9 Un peu de maths

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.14159265359

int main(){
  int x, y;
  float f_x, g_xy, h_x;

  printf("Rentrez deux entiers x et y : ");

  if (scanf("%d %d", &x, &y) != 2){
    printf("scanf déféctueux \n");
    exit(0);
  }

  f_x = (x-1) * log(x);
  g_xy = sqrt((1-x*x)) / (x+y);
  h_x = sin((x + PI/3));

  printf("%f, %f, %f\n", f_x, g_xy, h_x);

  exit(0);
}
```

Rappellons avant toute chose ici que pour utiliser la bibliothèque `math.h`, il faut nécessairement compiler avec l'option `-lm`, sachant qu'elle ne se place pas à n'importe quel endroit dans la commande, comme ceci : `gcc correction.c -lm -o correction`.

Pas de panique, le retour de la commande devrait afficher `-nan` ou quelque chose comme `None` entre les deux résultats que sont f_x et h_x pour la simple et bonne raison que racine de 1-x*x n'a pas de résultat pour x > 1 étant donné que x est un entier.

2 Les boucles

2.1 Équivalence des instructions répétitives

On a le programme suivant :

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

int main (void){
  int i ;
  for (i = 0 ; i < 10 ; i = i + 1)
  printf("le carre de %d est : %d \n", i , i*i);
  exit(0);
}
```

1. Ce programme affiche les carrés de 0 à 9.
2.
```
int main (void){
  int i = 0;
  while(i < 10){
    printf("le carre de %d est : %d \n", i , i*i);
    i++;
  }
  exit(0);
}
```

3.
```
int main (void){
  int i = 0;
  do{
    printf("le carre de %d est : %d \n", i , i*i);
    i++;
  }while(i < 10);
  exit(0);
}
```

Rappellons toutefois que `i=i+1`, `i+=1` et `i++` sont équivalents.

2.2 Retour sur le Cours - Équation du 2nd degré

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

int main(){
  char reponse = 'y';

  while (reponse == 'y'){
    int a, b, c, delta, x1, x2;

    printf("Rentrez trois entiers a, b et c : ");

    if (scanf("%d%d%d", &a, &b, &c) != 3){
      printf("scanf déféctueux\n");
      exit(0);
    }

    delta = b*b - 4*a*c;

    if (delta>0){
      x1 = (-b - sqrt(delta)) / 2*a;
      x2 = (-b + sqrt(delta)) / 2*a;
      printf("Racines de %dx²+%db+%d : x1 = %d et x2 = %d\n", a, b, c, x1, x2);
    }

    if (delta == 0){
      x1 = (-b - sqrt(delta)) / 2*a;
      x2 = 0;
      printf("Racine de %dx²+%db+%d : x = %d\n", a, b, c, x1);
    }
    if (delta < 0)
      printf("Delta négatif, pas de racine réelle\n");

    printf("Nouveau calcul ? (y = oui) : \n");
    if (scanf(" %c", &reponse) != 1){
      printf("scanf déféctueux\n");
      exit(0);
  }
  exit(0);
}
```

On utilise ici evidemment math.h avec l'option `-lm` car il est nécessaire d'appliquer la fonction `sqrt()` afin d'obtenir des racines carrées.

Je pense que ce programme se passe d'expliquatiion mais venez me poser toute question nécessaire sur discord si besoin.

2.3 Min-Max

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

int main(){
  int mini=INT_MAX, maxi=INT_MIN, n;
  prinft("Veuillez entrer une suite d'entiers séparés par un saut de ligne\nPour a
  while(scanf("%d",&n)==1){
      if( n > maxi ) maxi=n;
      if( n < mini ) mini=n;
  }
  printf("max : %d   -   min : %d\n",maxi,mini);
  exit(0);
}
```

Ici on utilise la bibliothèque `limits.h` qui permet d'utiliser INT_MAX ainsi qu'INT_MIN qui sont respectivement le plus grand et le plus petit entier (`int`) avant de déborder sur le grand entier (`long int`).

2.4 Table de multiplication

```
int main(){
  int i, j;

  for(i = 1; i < 11; i++)
    printf("%d x 5 = %d\n", i, i*5);

  for(i = 2; i < 10; i++){
    for(j = 1; j < 11; j++)
      printf("%d x %d = %d\n", y, x, y*x);
    printf("\n");
  }
  exit(0);
}
```

2.5 Série harmonique

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

void usage(char *s){
  printf("Usage : %s <entier>, <entier> positif\n", s);
}

int main(int argc, char* argv[]){
  int i, n;
  float somme;

  if(argc < 2 || atoi(argv[1]) < 0){
    usage(argv[0]);
    exit(-1);
  }

  n = atoi(argv[1]);
  somme = 0;

  for(i = 1; i < n+1; i++)
    somme += 1.0 / (i*1.0);

  printf("%f\n", somme);
  exit(0);
}
```

Ici on se déclare une fonction `usage(char *s)` qui servira à stopper l'execution de la fonction si la pile de donnée ne nous convient pas.

Quand je parle de piles de données, je parle du nombre d'argument(`argc`) et des arguments eux mêmes(`argv[]`). Sachant que `argv[0]` est la commande et que les arguments suivent ensuite.

2.6 Puissance

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

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

void usage(char * s){
  printf("usage : %s <nombre reel>\n", s);
}

int main (int argc, char ** argv){
  float x, res;
  int n;
  if (argc < 2){
    usage(argv[0]);
    exit(-1);
  }
  x = atof(argv[1]);
  res = x;
  for(n = 2; n <= 10; n++)
    {
      res = res*x;
      printf("%f puissance %d est : %f\n", x, n , res);
    }
  exit(0);
}
```

Rappellons tout d'abord que `res = res*x` et `res *= x` sont équivalent. Rien de bien particulier sur cette fonction permettant de calculer les puissances de 2 à 10.

Si on voulait calculer x puissance n en partant de ce code, il suffirait d'enlever le print, et de remplacer 10 par n+1. Enfin il faudrait afficher uniquement le résultat final qui serait donc x puissance n.

2.7 Factorielle

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

void usage(char *s){
  printf("Usage : %s <entier>, <entier> positif\n",s);
}

int main(int argc, char **argv){
  int n, i, fac;
  if ( argc < 2 || (n=atoi(argv[1])) <0 ) {
      usage(argv[0]);
      exit(-1);
  }
  fac = 1 ;
  for(i=1 ; i <= n ; i++){
    if( INT_MAX/i < fac ){
        printf("DEPASSEMENT DE LA MEMOIRE\nABANDON DU CALCUL\n");
        exit(-1);
    }
    fac*=i;
  }
  printf("%d! = %d\n",n,fac);
  exit(0);
}
```

Dans cette fonction factorielle, on est confronté potentiellement à un probleme qui peut être le dépassement de mémoire, on contre ce problème par la vérification que INT_MAX divisé par i est inférieur au résultat actuel avant de continuer.

Rappellons toutefois que n! = 1 x 2 x ... x n-1 x n.

2.8 PGCD

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

void usage(char *s){
  printf("Usage : %s <entier1> <entier2>, <entier1> <entier2> positifs\n",s);
}

int main(int argc, char* argv[]){
  int x, y, pgcd, min, i;
  pgcd = 1;

  if(argc < 3 || atoi(argv[1]) < 0 || atoi(argv[2]) < 0){
    usage(argv[0]);
    exit(-1);
  }

  x = atoi(argv[1]);
  y = atoi(argv[2]);

  (x <= y)?(min=x):(min=y);

  for(i = 1; i < min+1; i++)
    if ((x%i == 0) && (y%i == 0))
      pgcd = i;

  printf("PGCD(%d, %d) = %d\n", x, y, pgcd);
  exit(0);
}
```

Nouvelle notion ici, celui du if contracté.
La ligne `(x <= y)?(min=x):(min=y);` veut tout simplement dire :

- si x inférieur ou égal à y alors min = x
- sinon min = y

On calcule donc le PGCD de x et de y, on determine le minimum entre x et y de facon a faire le moins de vérification possible car x <= PGCD(x, y) <= y.

Les vérifications qui précèdent y sont donc vaines, l'optimisation d'un programme est très importante, après le fait que le programme fonctionne certes, mais c'est très important.

Surtout quand on attaque de grandes fonctions ou des projets plus amples qu'un simple calcul de PGCD.

2.9 Étoiles

```
/*Programme ligne*/
#include <stdlib.h>
#include <stdio.h>

void usage(char *s){
  printf("Usage : %s <entier>, <entier> positif\n",s);
}

int main(int argc, char* argv[]){
  int i, n;
  if (argc < 2 || atoi(argv[1]) < 0){
    usage(argv[0]);
    exit(-1);
  }
  n = atoi(argv[1]);

  for(i = 0; i < n; i++)
    printf("*");

  printf("\n");

  exit(0);(n =
}

/*Programme rectangle*/

void usage(char *s){
  printf("Usage : %s <entier1> <entier2>\n <entier1> positif, <entier2> positif\n",s);
}

int main(int argc, char* argv[]){

  int n, m, x, y;
  if (argc < 3  || atoi(argv[1]) < 0 || atoi(argv[2]) < 0 ){
    usage(argv[0]);
    exit(-1);
  }
  n = atoi(argv[1]);
  m = atoi(argv[2]);

  for(x = 0; x < m; x++){
    for(y = 0; y < n; y++)
      printf("*");
    printf("\n");
  }
  exit(0);
}
```

```
/*Programme triangle*/

void usage(char *s){
  printf("Usage : %s <entier>, <entier> positif\n",s);
}

int main(int argc, char* argv[]){
  int h, x, y;
  if (argc < 2 || atoi(argv[1]) < 0){
    usage(argv[0]);
    exit(-1);
  }
  h = atoi(argv[1]);

  for(x = 0; x < h; x++){
    for(y = h-x; y < h+1; y++)
      printf("*");
  printf("\n");
  }
  exit(0);
}
```

Voila les trois programmes que nous devons faire pour la 2.9, j'ai mis les trois programmes, donc les trois main, sans rabacher les include, je pense que vous comprenez pourquoi.

Aucune nouvelle notion apparente, juste de l'algorithmie ici.

2.10 Décodons, Décodons

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

void usage(char * nom){
  printf("usage : %s <entier min> <entier max> <entier n> <liste de n entiers entr
}

int main(){
  int mini, maxi, n, n_bis, i, tmp;

  if (argc < 4 || atoi(argv[3]) < 1 ){
    usage(argv[0]);
    exit(-1);
  }
  n = atoi(argv[3]);

  if (atoi(argv[1]) > atoi(argv[2])){
    usage(argv[0]);
    exit(-1);
  }
  mini = atoi(argv[1]);
  maxi = atoi(argv[2]);

  n_bis = n+4;
  if (argc < n_bis){
    usage(argv[0]);
    exit(-1);
  }

  for( i=4 ; i < n_bis ; i++){
    if (atoi(argv[i]) < mini || tmp > maxi){
      usage(argv[0]);
      exit(-1);
    }
    tmp=atoi(argv[i])
  }
  printf("TOUT EST OK\n");
  exit(0);
}
```

Cet exercice est pour le moins le plus compliqué à comprendre de la partie deux, je ne détaillerais pas la méthode utilisée ici mais je vous invite à venir me poser des questions le concernant.

3 Le nombre mystère

3.1 À vous de trouver le nombre...

Nous entrons dans des exercices un peu plus complexes demandant un effort algorithmique plus conséquent.
Voici donc ma correction du nombre mystère numéro 1 :

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

int main(void){
  int nb, prop, rep;
  rep = 1;
  srand(time(NULL));

  while(rep == 1){
    printf("Un nombre ?");
    prop = -1;
    nb = rand()%100;

    while(prop != nb){
      scanf("%d", &prop);
      if (prop<nb)
        printf("La réponse est supérieure à %d, votre proposition ? : \n", prop);
      if (prop>nb)
        printf("La réponse est inférieure à %d, votre proposition ? : \n", prop);
    }
    printf("bien joué,voulez vous rejouer ?(1 = oui) : ");
    scanf("%d", &rep);
  }
  exit(0);
}
```

Description de l'algorithme :

- Variables :
  - `nb` qui sera le nombre a deviné, il est initialisé par la fonction rand, il est compris entre 0 et 100 et reinitialisé a chaque fois que le joueur relance une partie.
  - `prop` qui sera la proposition du joueur, elle est initialisée à -1 pour ne pas qu'elle soit égale à nb sans que le joueur ai pu faire une proposition.
  - `rep` qui sera la reponse du joueur au fait de rejouer ou non, elle est initialisée à 1 pour que la partie démarre.
- Algorithme : si la proposition est supérieur ou inférieur au nombre à deviner, la machine le fait remarquer au joueur et lui fait rentrer une autre proposition, si elle est égale, on sort de la boucle while, le joueur à gagner et il peut décider de rejouer ou non.

3.2 Au tour de la machine

Nombre mystere numéro 2 : Je vous propose cet algorithme, j'ai choisi qu'il ne se repete pas car ici il n'y a pas d'interet a le faire répéter, j'ai plutot choisis d'axer mon algorithme sur la vitesse de recherche du nombre par la machine.

J'utilise la dichotomie, je laisse les gens que ce sujet laisse perplexe aller le chercher sur internet.
Mon programme affichera donc le nombre de proposition de la machine quand elle aura trouvé le nombre.

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

void usage(char * s){
  printf("usage : %s <entier>, 0 <= <entier> <= 100\n", s);
}

int main(int argc, char *argv[]){
  int nb, prop, compt, ecart;
  ecart = 100;
  compt = 1;
  prop = 50;

  if(argc < 2 || atoi(argv[1]) < 0 ||  atoi(argv[1]) > 100){
    usage(argv[0]);
    exit(-1);
  }
  nb = atoi(argv[1]);

  while (nb != prop){
    compt ++;
    if (prop<nb){
      ecart = ecart/2;
      if (ecart == 0){
        ecart = 1;
      }
      prop += ecart;
    }

    if (prop>nb){
      ecart = ecart/2;
      if (ecart == 0){
        ecart = 1;
      }
      prop -= ecart;
    }
  }
  printf("trouvé, %d propositions par la machine\n", compt);
  exit(0);
}
```

La recherche dichotomique peut être pratique pour un ordinateur comme pour les humains, bien que cette recherche ne soit ici pas infaillible.
Si on prend l'exemple 51 par exemple, l'ordinateur fera 8 propositions avec cette algorithme et seulement 2 si on lui donne 75.
plus le nombre varie des écarts dichotomiques, plus la machine aura a faire propositions, mais on sait qu'elle en fera 1 minimum (en l'occurence si le nombre est 50 car on initie la recherche à 50), et 8 maximum.

4 Tableaux et fonctions

4.1 Manipulation de tableaux avec des fonctions

Ici on aborde une nouvelle notion, celle des tableaux, quelques rappels :

- `*tab` = `tab[]`
- J'évite parfois de mettre les accolades quand le bloc d'instruction ne comporte qu'une seule ligne, cette technique marche, elle fait gagner de la lisibilité et de la place dans votre code mais je vous conseille grandement de mettre tout de meme les accolades dans ce genre de situation car au partiel ou dans un projet assez ample, c'est un gage de sureté.

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define N 25

void remplir_tab_alea(int *tab, int n){
  int i;
  for(i = 0; i < n; i++)
    tab[i] = 15-rand()%31;
}

void afficher_tab(int *tab, int n){
  int i;
  for(i = 0; i < n; i++)
    printf("%d ", tab[i]);
  printf("\n");
}

int min_tab(int *tab, int n){
  int i, min;
  min = tab[0];
  for(i = 1; i < n; i++)
    if(min > tab[i])
      min = tab[i];
  return min;
}

int max_tab(int *tab, int n){
  int i, max;
  max = tab[0];
  for(i = 1; i < n; i++)
    if(max < tab[i])
      max = tab[i];
  return max;
}

double moy_tab(int *tab, int n){
  int i;
```

```
  double moy = 0;
  for(i = 0; i < n; i++)
    moy += (tab[i]*1.0)/n;
  return moy;
}

void reverse(int *tab1, int *tab2, int n){
  int i;
  for(i = 0; i < n; i++)
    tab2[n-i-1] = tab1[i];
}

void mod_4(int *tab1, int *tab2, int n){
  int i;
  for(i = 0; i < n; i++)
    tab2[i] = tab1[i]%4;
}

void valeurs_pair(int *tab1, int *tab2, int n){
  int i;
  for(i = 0; i < n; i++)
    if(!(tab1[i] % 2))
      tab2[i] = tab1[i];
    else
      tab2[i] = 0;
}

int main(){
  int t1[N], t2[N], t3[N], t_pair[N];
  srand(time(NULL));

  remplir_tab_alea(t1, N);
  afficher_tab(t1, N);

  printf("max = %d, min = %d, moy = %f\n", max_tab(t1, N), min_tab(t1, N), moy_tab

  reverse(t1, t2, N);
  afficher_tab(t2, N);

  mod_4(t2, t3, N);
  afficher_tab(t3, N);

  valeurs_pair(t3, t_pair, N);
  afficher_tab(t_pair, N);

  exit(0);
}
```

Cet exercice est un fondamental, si vous ne le réussissez pas, il faut le retravailler impérativement car les tableaux sont une constituante principale du programme du semestre 3. Nous verrons par la suite les matrices.

Il faut comprendre que le tableau ne contiennent pas vraiment des valeurs, contrairement a des variables qui, elles, en contiennent bel et bien.
Un tableau est une boite remplie de pointeurs, ces pointeurs dirigent la machine vers la valeur qu'on a assigné a la case, comme un panneau. Les pointeurs sont des adresses mémoires, il ne faut donc pas appeller ces adresses si on ne les a pas encore assignés a quelque chose. On obtient en faisant ca la fameuse erreur de segmentation ou des valeurs improbables.

Par le meme principe, les matrices sont des pointeurs de tableaux, en somme, des pointeurs de pointeurs si vous préférez. Mais nous verrons au semestre 4 qu'on peut se passer des tableaux et créer en jouant avec la mémoire, les pointeurs ainsi qu'avec les structures, des listes dynamiques.

4.2 Déplacements de valeurs

Nous allons voir ici comment déplacer des valeurs dans une liste, bien qu'on ne déplace pas les valeurs en elle meme mais qu'on en donne l'impression.

Voici donc les fonctions de l'exercice 2 de la partie 4, je vous laisse le soin de les tester vous meme, je ne ferais pas un main ici ca il n'est pas spécialement demandé. Pour tester les fonctions, je vous conseille de reprendre les fonctions `remplir_tab_alea` et `afficher_tab` de l'exercice précédent.

```
void reverse(int *tab, int n){
  int i, tmp;
  for(i = 0; i < n/2; i++){
    tmp = tab[i];
    tab[i]= tab[n-i-1];
    tab[n-i-1] = tmp;
  }
}

void decalage(int *tab, int n, char dec){
  int i, tmp;
  if(dec == 'g'){
    tmp = tab[0];
    for(i = 0; i < n-1; i++)
      tab[i] = tab[i+1];
    tab[n-1] = tmp;
  }
  if(dec == 'd'){
    tmp = tab[n-1];
    for(i = n-1; i > 0; i--)
      tab[i] = tab[i-1];
    tab[0] = tmp;
  }
}

void permutCirculaire(int *tab, int n, int k){
  int i;
  for(i = 0; i < k; i++)
    decalage(tab, n, 'd');
}
```

4.3 Tri à la volée

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#define NB_MAX 300

void afficher_tab(int *tab, int n){
  int i;
  for(i = 0; i < n; i++)
    printf("%d ", tab[i]);
  printf("\n");
}

int tri_volee(int *tab, int n){
  int i, j, taille, nb;
  taille = 0;
  nb = 0;

  while(nb > -1){
    printf("un nombre ? : ");
    if(scanf("%d", &nb) != 1){
      printf("Mauvaise entrée\n");
      exit(-1);
    }
    if(nb > -1){
      taille++;
      i = 0;
      while(nb < tab[i] && tab[i] > 0)
        i++;
      for(j = n; j > i; j--)
        tab[j] = tab[j-1];
      tab[j] = nb;
    }
  }
  return taille;
}

int main(){
  int taille, tab[NB_MAX];
  taille = tri_volee(tab, NB_MAX);
  afficher_tab(tab, taille);
  exit(0);
}
```

Ici nous abordons la premiere fonction de tri, le tri à la volée

Le probleme est que la taille du tableau final est variable, c'est pour ca que la fonction `tri_volee` retourne la taille du tableau créé pour ne pas avoir d'erreur de segmentation à l'affichage.

Je pense honnetement que beaucoup ont du ramer sur le tri à la volée, si vous ne comprenez pas l'algorithme, venez poser toutes vos questions sur discord, car cet algorithme est très important et centralise beaucoup de connaissances liés aux tableaux, comme fait de gérér une taille de tableau de manière pseudo-dynamique, le déplacement de valeur ou encore le tri d'un tableau en soi.

4.4 Le Crible d'Ératosthène

Ici la fonction qui permet de réaliser le crible d'érathostene de 2 à n :

```
#include <stdlib.h>
#include <stdio.h>
#define NB_MAX 300

void eratosthene(int n){
  int tab[NB_MAX];
  int i,j;

  for (i = 2 ; i <= n ; i++){
    tab[i] = 1 ;
  }
  for (i = 2 ; i <= n ; i++){
    if(tab[i]){
      for (j = 2*i j <= n ; j+=i ){
        tab[j] = 0;
      }
     }
   }
  printf("les nombres premiers jusqu'à %d sont : \n",n);

  for (i = 2 ; i <= n ; i++){
     if(tab[i])
      printf("%d\n",i);
  }
}
```

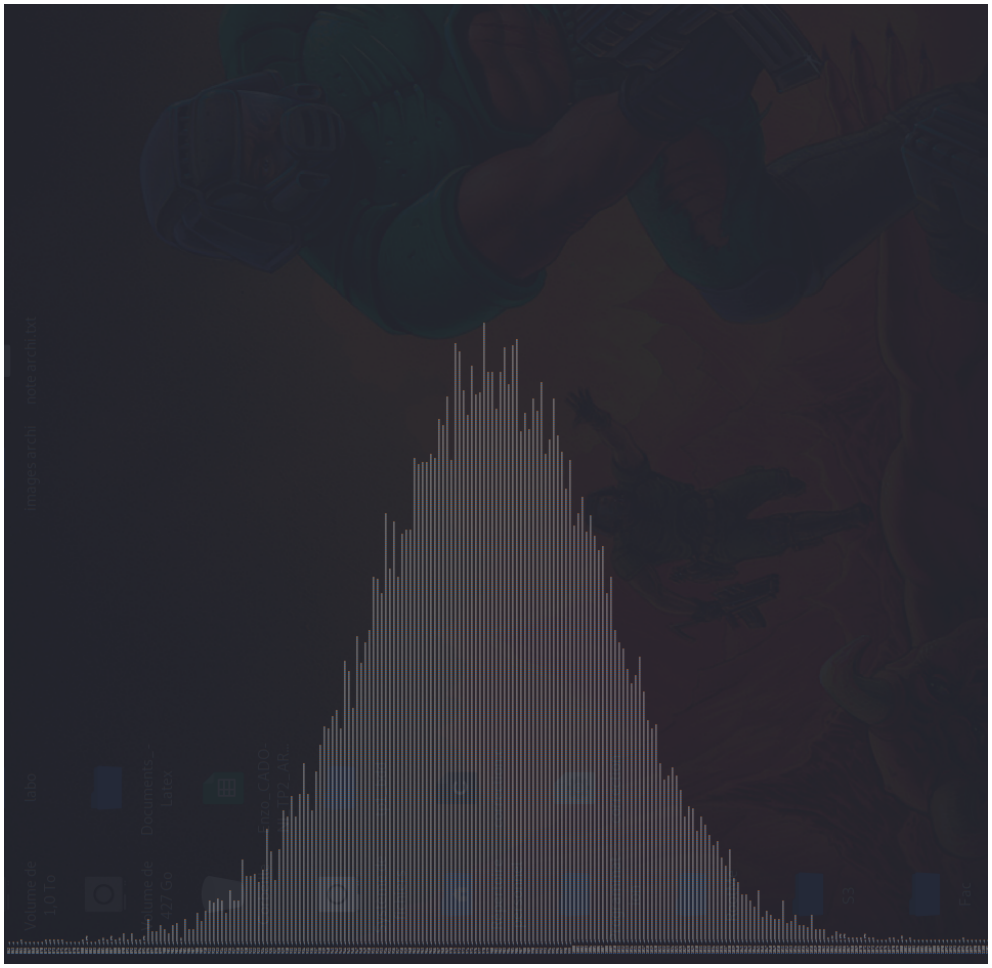
Je pense que cette fonction se passe d'expliquation, venez posez vos questions sur discord, si toutefois vous en avez.

4.5 Jeu de dés

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define NB_MAX 300

void usage(char *s){
  printf("usage : %s <entier1> <entier2>, <entier1> et <entier2> positifs\n", s);
}

void lancer(int nb_de, int lances, int *tab){
  int x, y;
  srand(time(NULL));
  for(x = 0; x <= nb_de*6; x += 1)
    tab[x] = 0;

  for(x = 0; x < lances; x += 1){
    int compt = 0;
    for(y = 0; y < nb_de; y += 1)
      compt += (rand()%6) + 1;
    tab[compt] ++;
  }
}

void affichage(int *tab, int nb_de){
  int x, y;
  for(x = nb_de; x <= 6*nb_de; x += 1){
    printf("%d ->",x);
    for(y = 0; y < tab[x]; y += 1)
        printf("*");
    printf("\n");
  }
}

int main(int argc, char *argv[]){
  int nb_de, lances, tab_occ[NB_MAX];
  if(argc < 3 || atoi(argv[1]) < 0 || atoi(argv[2]) < 0){
    usage(argv[0]);
    exit(0);
  }
  nb_de = atoi(argv[1]);
  lances = atoi(argv[2]);

  lancer(nb_de, lances, tab_occ);
  affichage(tab_occ, nb_de);
  exit(0);
}
```

Le lancé de dé est un algorithme qui peut paraitre assez complexe dans sa compréhension, on veut simuler n lancer de m dés.

On simule un dé qu'on lance, on accumule les lancés autant de fois qu'on a de dés, et on incrémente l'occurence de ce lancé. Un exemple, si on a 6 20 en argument, l'algorithme va accumuler dans une variable 6 fois le lancé random d'un dé à 6 faces. Le résultat de cette accumulation servira ensuite d'index pour incrémenter l'occurence du tableau, l'affichage consistera a afficher autant d'étoiles par lignes que vaut la valeur du tableau à l'occurence du résultat.
Si votre algorithme réussi, le résultat doit produire une courbe de gauss, comme ceci :



Cette courbe fut obtenue en executant le prgramme avec 280 et 19750 en arguments.

4.6 Conversion de bases

Le but ici sera de pouvoir convertir un nombre en base 10 à un nombre en base b comprise entre 1 et 10 contenu dans un tableau, ainsi que de pouvoir transformer un tableau representant un nombre en base b en un nombre en base 10.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#define NB_MAX 100

void encode(int nb, int base){
  int resultat[NB_MAX], i, taille;
  taille = 0;
  i = 0;
  while(nb != 0){
    resultat[i] = nb%base;
    nb /= base;
    i++;
    taille++;
  }
  for(i = taille-1; i >= 0; i--)
    printf("%d ", resultat[i]);
  printf("\n");
}

void decode(){
  int nb[8], base, resultat, i;
  srand(time(NULL));
  resultat = 0;
  base = -1;
  while(1 > base || base > 10){
    printf("la base ?\n");
    scanf("%d", &base);
  }

  for(i = 0; i < 8; i++)
    nb[i] = rand()%base;

  for(i = 0; i < 8; i++)
    resultat += nb[i]*pow(base, 7-i);

  printf("%d\n", resultat);
}

int main(){
  encode(89, 2);
  decode();
  exit(0);
}
```

Je ne vous expliquerais pas ici comment convertir un nombre d'une base à une autre, car c'est l'objet du cours d'architecture des ordinateurs, en revanche, si vous avez des soucis avec les fonctions présentés ici, venez poser vos questions.

Il s'agit pour la première fonction de prendre un nombre, et de le diviser par la base jusqu'à ce qu'il soit égal à 0. à chaque fois qu'on divise le nombre par la base, qui sont donc les deux arguments de la fonction `encode`, on ajoute a une case du tableau le nombre modulo la base. On incrémente la taille pour savoir quelle taille fera notre nombre tranformé a la fin.
La taille servira dans une boucle for à afficher notre nombre final.

Quand a la deuxieme fonction, il s'agit du principe inverse, on a un talbleau representant un nombre en base déterminée, on connais sa taille, qui est de 8 et que nous a d'ailleurs imposé l'énoncé. On ajoute donc progressivement à une variable `resultat` chaque case du nombre mis à une puissance donnée en augmentant proportionnellement la puissance à l'aide de i.
Rappellons que la premiere case du tableau sera puissance 7 et que la dernière sera donc puissance 0, peut importe la base choisie.
On rappelle aussi que le nombre est généré de manière random, comme préscrit dans l'énoncé.

N'oubliez pas l'option `-lm` pour utiliser la bibliothèque `math.h` et donc la fonction `pow()`, ou faite simplement une puissance avec un `for`, mais j'avoue que pour le coup, je ne me suis pas cassé le cul.

4.7 Mes premières matrices

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 5

void remplissage(int mat[][N], int n){
  int i, j;
  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      mat[i][j] = 15-rand()%25;;
}

void affichage(int mat[][N], int n){
  int i, j;
  for(i = 0; i < n; i++){
    for(j = 0; j < n; j++)
      printf("%d ", mat[i][j]);
    printf("\n");
  }
  printf("\n");
}

int trisup(int mat[][N], int n){
  int i, j;
  for(i = 1; i < n; i++)
    for(j = 0; j < i; j++)
      if(mat[i][j] != 0)
        return 0;
  return 1;
}

int triinf(int mat[][N], int n){
  int i, j;
  for(i = 0; i < n-1; i++)
    for(j = i+1; j < n; j++)
      if(mat[i][j] != 0)
        return 0;
  return 1;
}
```

Voici ici les fonctions concernant le premier exercice sur les matrices, je vous laisse evidemment le soin de faire un main de votre côté si vous voulez les tester, je gagne ici de la place en ne le mettant pas.
Les matrices peuvent être complexes à abordes, faisons quelques rappels sur leur utilisation.

Dans les paramètres d'une fonctions, les tableaux de n'importe quelle dimension et de n'importe quel type répondent à une regle simple et importante, ils doivent

connaitre au moins toutes leurs dimensions sauf la première, je m'explique. Si jamais on a un tableau a trois dimensions tal qu'on ai `int tab[x][y][z]`.
On écrira non pas `fonction(int ***tab)` ou `fonction(int **tab[])` ou autre mais bel et bien `fonction(int tab[][y][z])` ou bien `fonction(int tab[x][y][z])` mais cette dernière écriture est inutile.

4.8 D'autres matrices

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 5
#define M 5

void affichage_matrice(int mat[][N], int n, int m){
  int i, j;
  for(i = 0; i < n; i++){
    for(j = 0; j < m; j++)
      printf("%d ", mat[i][j]);
    printf("\n");
  }
  printf("\n");
}

void remplissage(int mat[][N], int n, int m){
  int i, j;
  for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
      mat[i][j] = 100-rand()%201;
}

void addition(int mat1[][N], int mat2[][N], int mat3[][N], int n, int m){
  int i, j;
  for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
      mat1[i][j] = mat2[i][j] + mat3[i][j];
}

void multiplication(int mat1[][N], int mat2[][N], int mat3[][N], int n, int m){
  int i, j, x;
  for(i = 0; i < n; i++){
    for(j = 0; j < m; j++){
      mat1[i][j] = 0;
      for(x = 0; x < n; x++)
        mat1[i][j] += mat2[i][x]*mat3[x][j];
    }
  }
}
```

```
void transpose(int mat_tr[][N], int mat[][N], int n, int m){
  int i, j;
  for(i = 0; i < n; i++)
    for(j = 0; j < m; j++)
      mat_tr[i][j] = mat[j][i];
}
```

Voici donc d'autres fonctions sur les matrices, je ne détaillerais pas les méthodes de calcul matriciel utilisés dans ces fonctions.

Je vous laisse comme pour l'exercice d'avant créer un main si besoin pour étudier leur fonctionnement.

4.9 Le triangle de Pascal

```
#include <stdlib.h>
#include <stdio.h>
#define MAX 500

void usage(char *s){
  printf("%s <entier>, <entier> positif\n", s);
}

void pascal(int tab[][MAX], int n){
  int x, y;
  tab[0][0] = 1;
  tab[1][0] = 1;
  tab[1][1] = 1;

  for(x = 2; x < n; x++){
    tab[x][0] = 1;
    for(y = 1; y < x+1; y++)
      tab[x][y] = tab[x-1][y-1]+tab[x-1][y];
  }
}

void affichage(int tab[][MAX], int n){
  int x, y;
  for(x = 0; x < n; x++){
    for(y = 0; y < x+1; y++)
      printf("%d ", tab[x][y]);
    printf("\n");
  }
}

int main(int argc, char* argv[]){
  int n, tab[MAX][MAX];

  if(argc < 2 || atoi(argv[1]) < 0){
    usage(argv[0]);
    exit(-1);
  }

  n = atoi(argv[1]);
  pascal(tab, n);
  affichage(tab, n);
  exit(0);
}
```

Ici j'ai préféré pour le triangle de pascal ne pas me concentrer sur le coefficient binomial mais simplement comment décrire le triangle de pascal en algorithme, pour ceux que le coefficient binomial laisserais dans le flou.

Il n'est donc pas nécessaire d'avoir de grandes connaissances en mathématique pour le faire, je vous laisse analyser cet algorithme fait par mes soins, je pense qu'il est assez court pour se passer de détails

5 Chaînes de caractères

5.1 Je crée mes propres fonctions sur les chaînes

1. `mystrlen()` :

```
int mystrlen(char *str){
  int i = 0;
  while(!(str[i] == '\0'))
    i++;
  return i;
}
```

   Ici on sait que la fin d'un tableau de caractère est `'\0'` donc on parcours le tableau jusqu'à le trouver et on retourne `i` quand on l'a trouvé, qui s'est incrementé a chaque fois que `tab[i]` n'était pas `'\0'`.

2. `mystrcmp()` :

```
int strcmp(char *str1, char *str2){
  int i = 0;
  while ((str1[i] == str2[i]) && (str1[i] != '\0')){
    i++;
  }

  if ((str1[i] == '\0') && (str2[i] == '\0'))
    return 0;
  if (str1[i] < str2[i])
    return -1;
  if (str1[i] > str2[i])
    return 1;
}
```

   On souhaite que la fonction strcmp() retourne -1 ou 1 si les deux chaines sont differentes, ici, dès que la premiere difference entre les deux chaines, si le caractère de la premiere chaine est supérieur(code ASCII) à celui de la deuxieme au même `i`, alors on retourne 1, si c'est l'inverse, on retourne -1, sinon, c'est qu'il n'y a pas de différence, donc on retourne 0.

3. `concat()` :

```
void concat(char *str1, char *str2, char *str3){
```

```
  int i, j;
  i = 0;
  j = 0;
  while(str1[i] != '\0'){
    str3[i] = str1[i];
    i++;
  }
  while(str2[j] != '\0'){
    str3[i] = str2[j];
    i++;
    j++;
  }
  str3[i] = '\0';
}
```

Le but ici est de concaténer deux chaines et de stocker le résultat dans une autre chaine.
On utilise donc deux index, `i` et `j` qui serviront a parcourir les chaines.
On parcours donc `str1` et `str3` avec `i` en associant `str1[i]` à `str3[i]` puis on fait de même avec `str2` en associant `str2[j]` à `str3[i]`, on utilise j pour débuter le parcours de `str2` du début sans retourner au début de `str3`.

4. `delspace()` :

```
void delspace(char *str){
  int i, j;
  i = 0;
  while(str[i] != '\0'){
    if(str[i] == ' '){
      j = i;
      while(str[j] != '\0'){
        str[j] = str[j+1];
        j++;
      }
    }
    i++;
  }
}
```

Dans le cas de `delspace()`, on décale tout à droite à partir `i` jusqu'à `'\0'` chaque fois que `str[i]` est un espace.

5. `strsubs()` :

```
void strsubs(char *str1, char *modif, int lg, int pos){
  int i = pos-1;
  while(i-pos+1 != lg){
    str1[i] = modif[i-pos+1];
```

```
      i++;
    }
  }
```

Le principe de `strsubs()` est de remplacer dans une chaine, à un index donnée, et sur une distance donnée, une partie d'une autre chaine équivalente à la-dite distance.
On initialise un index `i` à `pos-1`, pos étant l'index de départ auqeul on soustrait 1 puisque les index de tableaux commencent à 0.
On remplace sur la distance donnée en argument(`lg`) `str[i]` par `modif[i-pos]` puisqu'on veut partir de `pos` dans `str` mais de `0` dans `modif`.
On s'arrete dès qu'on arrive à `'\0'` ou qu'on atteint la distance(`lg`) voulue.

6. `min2maj()` :

```
void min2maj(char *str){
  int i = 0;
  while(str[i] != '\0'){
    if('a' <= str[i] && str[i] <= 'z'){
      str[i] = str[i]-'a'+'A';
    }
    i++;
  }
}
```

Pour `min2maj()`, on pourrait croire qu'il faut jouer avec la table ASCII et la connaitre. J'avoue que connaitre par coeur la table ASCII serait un moyen assez efficace pour faire ce genre d'algorithme, mais il existe une parade.
prenant en compte que les majuscules sont avant les minuscules dans la table ASCII, on vérifie bien si `str[i]` est bien une lettre minuscule donc si elle se situe bien entre `'a'` et `'z'` compris, puis on lui soustrait `'a'` et on lui ajoute `'A'`.
Tout simplement car -'a'+'A' est égal à l'écart entre les minuscules et les majuscules dans la table ASCII.
C'est comme si on faisait `str[i] + 32` (car oui c'est la valeur de l'écart) mais sans le connaitre.
Je vous invite à aller vérifier ca sur la table ASCII mais en somme on peut jouer avec sans vraiment connaitre ses valeurs.

7. `maj2min()` :

```
void maj2min(char *str){
  int i = 0;
  while(str[i] != '\0'){
    if('A' <= str[i] && str[i] <= 'Z'){
      str[i] = str[i]-'A'+'a';
    }
    i++;
```

```
    }
  }
```

Ici, j'ai réalisé la fonction avec le même trucage que dans l'exercice précédent, sans regarder la table ASCII, à une chose près, c'est que j'ai mis `-'A'+'a'` au lieu de `-'a'+'A'` pour la simple et bonne raison que cela va donner l'écart entre les majuscules et les minuscules certes mais en négatif(`-32` en somme). On aurait aussi pu faire `str[i]+'a'-'A'` ou bien `str[i]+(-'A'+'a')` mais vous comprenez l'idée.

5.2 Analyse d'un fichier texte

Pour résoudre cet exercice, j'ai décidé de modéliser le probleme sous la forme d'une fonction d'analyse complete pour limiter les ouvertures et fermetures intempestives du fichier texte.
On aura donc tout d'abord une fonction `usage()` dont je vous passe l'utilité, et est là car nous rentrerons le nom du fichier texte en argument de la commande.
Ensuite la (fastidieuse) fonction `est_separateur` qui renvoie `1` si le caractère mis en argument fait partie des séparateurs.
Nous aurons la fonction `analyse` qui prend en argument un fichier(`FILE *fichier`) et qui affiche combien il comporte de caractère, de chacune des lettres de l'alphabet, de mots ainsi que de paragraphes. Enfin, la fonction `main` qui ici est importante car elle est la ou l'on ouvre le fichier, et il est important de savoir comment utiliser le type `FILE` ainsi que les fonctions `fopen` et `fclose`.

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

void usage(char *s){
  printf("%s <string>\n", s);
}

int est_separateur(char c){
  if(c == '\n'|| c == ' ' || c == ':' || c == '.' || c == ';' ||
     c == ',' || c == '?' || c == '!' || c == '(' || c == ')' ||
     c == '"' || c == '\'')
    return 1;
  else
    return 0;
}

void analyse(FILE *fichier, int *srepertoire){
  int i;
  char c = ' ';
  for(i = 0; i < 29; i++)
    repertoire[i] = 0;

  while(c != EOF){
```

```
    c = fgetc(fichier);

    if(c != EOF && 'A' <= c && c <= 'Z'){
        repertoire[c-'A'+1]++;
        repertoire[0]++;
    }
    if(c != EOF && 'a' <= c && c <= 'z'){
        repertoire[c-'a'+1]++;
        repertoire[0]++;
    }
    else{
      while(c != EOF && est_separateur(c)){
        if(c == '\n')
          repertoire[28]++;
        while(c == '\n'){
          c = fgetc(fichier);
        }
        c = fgetc(fichier);
        repertoire[0]++;
      }
      repertoire[27]++;
    }
  }
  repertoire[0]--;
  printf("Dans ce fichier, il y a :\n\n");
  printf("%d caractères\n", repertoir
  e[0]);
  printf("Lettres :\n");
  for(i = 1; i < 27; i++)
    printf("    %d occurence de %c\n", repertoire[i], 'a'+i-1);
  printf("%d mots\n", repertoire[27]);
  printf("%d paragraphes\n", repertoire[28]);
}

int main(int argc, char *argv[]){
  int repertoire[29];
  FILE *fichier = NULL;

  if(argc < 2){
    usage(argv[0]);
    exit(0);
  }

  fichier = fopen(argv[1], "r+");
  analyse(fichier, repertoire);
  fclose(fichier);
  exit(0);
}
```

Je vous passerais ici les explication des fonctions `usage` et `est_separateur`.

Description de la fonction `analyse` :

On prend un tableau de 29 occurence appellé `repertoire`, vous allez me demander pourquoi 29, et bien parce que dans ce tableau, il y aura le nombre de caractère, le nombre d'occurence de chaque lettre, le nombre de mots et le nombre de paragraphes, ce qui fait en tout 29.

Le tableau est passé en parametre de la fonction, il est initialisé dans la fonction `main` et est mis a zero au début de la fonction `analyse`.

On commence par débuter une boucle `while` avec comme condition d'arret que `c` soit égal à `EOF`, soit **E**nd**O**f**F**ile, en gros, quand on fait `c = fgetc(fichier)` alors qu'on est a la fin du fichier, `c` prendra la valeur de `EOF` et si on retente de faire `c = fgetc(fichier)`, on aura une erreur de segmentation ou alors c prendra une valeur improbable, ce qui a mon sens est pire.

Dans le while, on assigne un caractère du fichier à c, puis on vérifie :

- Si c'est une lettre majuscule, auquel cas on incrémente la valeur `c-'A'+1` de `repertoire`, par le meme principe que dans l'exercice précédent, si on a `'A'`, alors `'A'-'A' = 0`, et `0+1 = 10`, et on sait que les occurences de lettres dans notre tableau vont de 1 à 26.
- Si c'est une lettre majuscule, auquel cas on incrémente la valeur `c-'a'+1` de `repertoire`, par le meme principe que dans l'exercice précédent, si on a `'d'`, alors `'d'-'a' = 3`, et `3+1 = 4`, qui est bien l'index reservé dans notre tableau pour les ocurences de `'d'`.
- Sinon
  - Tant que `c` n'est pas `EOF` et est un séparateur :
    - Si c est un saugt de ligne, alors on incrémente la case comptant les paragraphes dans `repertoire`, et tant que `c` est un saut de ligne alors on recupere le caractère suivant, ce qui évite de compter deux paragraphes pour deux sauts de lignes consécutifs mais bel et bien un seul.
    - Tout séparateur qui n'est pas un saut de ligne ou qui précède ou suit un saut de ligne est bien entendu compté comme un caractère par l'incrémentation de la premiere case de notre tableau `repertoire`, l'avantage de cette boucle `while` est bien entendu de compter un mot de plus, meme si il y a plusieurs séparateurs consécutifs.
  - On sort de la boucle des séparateurs, on compte donc un mot de plus en incrémentant la case comptant les mots dans notre tableau `repertoire`.

On sort de notre première boucle `while` et on décremente les caractères car on a compté `EOF`, c'est le seul defaut de mon algorithme que je n'ai pas corrigé, même si il est plutôt optimisé à mon sens.

Place à l'affichage, que je vous pouvez bien evidemment modifier car j'ai fait au plus simple, je me contente d'afficher chaque occurence de chaque chose en rappellant leur nom, simple, mais néanmois efficace.

6 Le jeu du pendu

Concernant le jeu du pendu, je ne me suis pas trop cassé le cul j'ai simplement fait les fonction demandées en exercice. Ensuite, j'ai tout assemblé en utilisant bien sur son fichier de mort et en faisant un menu bouclant au préalable, dans le fichier que j'ai épinglé soigneusement dans le serveur et qui s'appelle correction.c vous trouverez donc les deux fonctions d'enregistrement de mot, je vous laisse le soin de décrypter ce programme pas aussi long et chiant qu'il en à l'air. Nous en outre déjà étudié la gestion de fichier dans ce pdf, inutile donc de faire des rappels.

7 Un aparté : switch

Je ne vais pas m'étendre sur comment l'on utilise le switch et sur pourquoi cet exercice est plus fastidieux et chiant qu'interressant, en voici simplement une correction fonctionelle...

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

int main(void){
  int nombre;

  printf("Le nombre ? : \n");

  if(fscanf(stdin, "%d", &nombre) != 1 || !(-1 < nombre && nombre < 10)){
    printf("Erreur de saisie\n");
    exit(-1);
  }
  switch(nombre){
    case 0:
      printf("ZERO\n");
      break;
    case 1:
      printf("UN\n");
      break;
    case 2:
      printf("DEUX\n");
      break;
    case 3:
      printf("TROIS\n");
      break;
    case 4:
      printf("QUATRE\n");
      break;
    case 5:
      printf("CINQ\n");
      break;
    case 6:
      printf("SIX\n");
```

```
        break;
      case 7:
        printf("SEPT\n");
        break;
      case 8:
        printf("HUIT\n");
        break;
      case 9:
        printf("NEUF\n");
        break;
      default:
        break;
    }

    exit(0);
  }
```

8 Structures de données simples

À partir de cette partie je ne donnerais aucun code dans le pdf ou alors très peu car les codes deviennent long, je les donnerais en fichier annexe dans le serveur, dans le channel de programmation impérative I.

Je vous laisse le soin d'analyser l'archive portant le nom original de archive$_p$8$contenantlestroiscoderepresen$
$partiesdutp$8.$Posez toutes vos questions dans le channel dédié$.

9 Modules et tris

L'archive regroupant la correction complète de la partie 9 se trouve sur le serveur dans le channel dédié et soigneusement épinglé, je vous y renvoie.
Posez toutes les questions necessaire dans ce même channel.

10 Découverte de MLV

Pour toute subtilité liée à MLV, je vous renvoie au projet de jeu d'échec que j'ai réalisé avec M.Le Dortz plutôt qu'aux tps de morvants, nous y avons utilisé a peu près tout ce que MLV propose sauf les animations, y compris les musiques, les bruits et la gestion du clic/clavier (fluide ou statique).
Le github du projet.