université de BORDEAUX	**ANNEE UNIVERSITAIRE 2015 / 2016** **S1 D'AUTOMNE** **PARCOURS / ETAPE:** Informatique **Code UE :** J1INAW02 **Epreuve :** Programmation 1 **Date :** 05/01/2015 **Heure :** 8H30 **Durée :** 1H30 Documents : autorisés (**uniquement une feuille A4 Recto/Verso manuscrite**) Epreuve de M : Blin Guillaume	**Collège Sciences et technologies**

Indiquez votre code **d'anonymat** : N°

La notation tiendra compte de la clarté de l'écriture des réponses ainsi que de leur précision. Par la suite, vous devez répondre directement sur l'énoncé à toute question étant suivie d'un cadre prévu à cet effet.

Exercice 1. (5 points)

- **Q1 (1 point) :** Quelles sont les valeurs des entiers `a`, `b` et `c` affichés à la fin de l'exécution du programme suivant?

```
1.   #include <stdio.h>
2.   #include <stdlib.h>
3.    int a=5;
4.    int b=6;
5.
6.   int g(int a){
7.     int b;
8.     b=3*a;
9.     return b;
10.  }
11.
12.  int f(int * p){
13.   *(p)=10*a;
14.   return (*(p))*g(a);
15.  }
16.
17.
18.  int main(void){
19.   int b=2;
20.   int c=3;
21.   c=f(&b);
22.   printf("%d %d %d\n",a,b,c);
23.   return EXIT_SUCCESS;
24.  }
```

Valeur de a	Valeur de b	Valeur de c
5	50	750

- **Q2 (1 point)** : Expliquer de manière synthétique et avec vos mots les différents modes d'allocation et de stockage mémoire des variables et les conséquences (e.g., différence de localisation, de durée de vie, de visibilité, …).

(c.f. Cours 11)
Il existe 3 modes d'allocations mémoires

- Statique (dans les segments .data et .bss) qui correspond aux variables globales et statiques qui ont des adresses fixes et valides du lancement jusqu'à la fin du processus
- Dynamique (dans le tas). Le tas est un espace mémoire allouable dynamiquement (e.g. malloc) par le programmeur. Les variables stockées dans le tas sont accessibles partout dans le programme, par l'intermédiaire des pointeurs.
- Automatique (dans la pile), lors d'un appel de fonction (l'espace nécessaire est automatiquement réservé et libéré). La pile permet de stocker, entres autres, les variables locales de la fonction, une copie des arguments passés à la fonction, une valeur de retour. La durée de vie d'une variable locale à une fonction est donc bien limitée à celle de la fonction

- **Q3 (1 point) :** Expliquez avec vos mots pourquoi en C le seul mode de passage de paramètre d'une fonction est celui par copie (*aide*: pensez à la pile d'appel de fonction).

En C, lors de l'appel de fonction, les arguments de la fonction sont, dans un premier temps, évalués (i.e., remplacer par une valeur) avant d'être transmis. Cette copie des évaluations des arguments est perdue lors du retour de la fonction. Afin de modifier une variable non locale à la fonction, une copie de l'adresse de cette dernière est transmise à la fonction via la pile. A l'aide de cette adresse et de l'opérateur de déréférencement « * », il est possible de modifier la variable.

- **Q4 (1 point) :** Expliquer le résultat inattendu de ce programme qui affiche `sum=32768.000000` (*aide*: on ne demande pas d'expliquer le résultat précisément mais pourquoi on obtient un résultat très éloigné de celui attendu – i.e., `999999.997`)

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.  int main(void) {
4.    int i;
5.    float sum=0.f;
6.    for (i=0;i<999999997;i++) {
7.      sum=sum+0.001f;
8.    }
9.    printf("sum=%f\n",sum);
10.   return EXIT_SUCCESS;
11. }
```

La norme IEEE754 est basé sur une approximation des nombres représentables dans une machine. Le problème ici n'est pas dû à une somme d'approximation. Il résulte du fait que lors de l'addition de 32768 et de 0.001, le résultat (i.e. 32768,001) ne peut pas être représenté de manière exacte et la partie la moins significative est tronquée (ici, le 0,001). Dès lors, sum n'évolue plus et répète le problème jusqu'à la fin de la boucle.

- **Q5 (1 point) :** Quelles sont, pour vous, les règles principales à respecter pour garantir une maintenabilité, une réutilisation et une possibilité d'évolution les plus élevées possibles ?

  - Découpage du code en plusieurs fichiers (compilation séparée) correspondant chacun à un ensemble de services portant sur une thématique précise
  - Découplage des choix d'implémentation de la définition globales des services (implémentation/interface - encapsulation)
  - Respects de conventions
  - Commentaires
  - …

Exercice 2. (10 points)

- Soit la fonction **`foo`** suivante:

```
1.   #include <stdio.h>
2.   #include <stdlib.h>
3.
4.   char * foo(char * start, char * end){
5.     if(start > end){
6.       fprintf(stderr, "'end' should not appear before 'start'");
7.       return NULL;
8.     }
9.     char * res = start;
10.    char val = *(start);
11.    for(char * tmp = start+1; tmp <= end; tmp++){
12.      if(*(tmp) < val) {
13.        res = tmp;
14.        val = *(tmp);
15.      }
16.    }
17.    return res;
18.  }
19.
20.  int main(void) {
21.    char s[] = {'O','F','F','E','R','T'};
22.    char * m = foo(s,s+5);
23.    if(m != NULL){
24.      printf("foo=%c\n",*(m));
25.    }
26.    return EXIT_SUCCESS;
27.  }
```

- **Q1 (2 point) :** Expliquer ce que vérifie le premier test (lignes 5 à 8) au début de la fonction. De façon générale, indiquer quelles sont les diverses manières de traiter un cas d'erreur.

Le premier test vérifie que l'adresse contenue dans le pointeur « start » est inférieure à celle contenue dans le pointeur « end ». Cette vérification est nécessaire pour garantir la bonne exécution de la suite de la fonction.
Un cas d'erreur

Il est toujours bon d'afficher un message explicatif sur la sortie standard d'erreur (stderr). L'erreur peut être signalée à la fonction appelante à l'aide d'une valeur spécifique de la valeur de retour ou, lorsque cela n'est pas possible, en transmettant l'adresse de la valeur de retour en paramètre de la fonction.

En cas d'erreur insurmontable, il est possible d'interrompre le programme en appelant l'instruction exit définie dans stdlib.h qui prend en paramètre EXIT_FAILURE ou EXIT_SUCCESS

- **Q2 (2 points) :** Expliquer ce que fait la fonction **`foo`**.

La fonction foo renvoie **l'adresse** du plus petit caractère dans l'ordre alphabétique se trouvant entre les adresses « start » et « end ».

- **Q3 (2 points) :** En utilisant la fonction **`foo`** définie précédemment, proposer une fonction **`charsort`** dont le prototype est le suivant :

```
char * charsort(char * w, int len);
```

Le paramètre **`w`** représente le tableau de caractères à trier et **`len`** le nombre de caractères dans le tableau. La fonction retourne **une copie** du tableau **`w`** dont les éléments sont triés du plus petit au plus grand (*aide*: L'évaluation ne tiendra pas compte de la complexité de la solution proposée).

CORRECTION

```
1.  char * charsort(char * w, int len){
2.    if(len < 0){
3.      fprintf(stderr, "nothing to sort !");
4.      return NULL;
5.    }
6.    char * res = (char *) malloc(sizeof(char)*len);
7.    if(res == NULL){
8.      fprintf(stderr, "memory problem!");
9.      exit(EXIT_FAILURE);
10.   }
11.   for(int i = 0; i < len; i++){
12.     res[i] = w[i] ;
13.   }
14.   for(int i = 0; i < len-1; i++){
15.     char * smallest = foo(res+i,res+len-1) ;
16.     char tmp = res[i];
17.     res[i] = *(smallest);
18.     *(smallest) = tmp;
19.   }
20.   return res;
21. }
```

- **Q4 (2 points) :** Nous souhaitons offrir la possibilité de baser le tri sur une fonction quelconque (i.e., pas forcément l'ordre lexicographique). Nous proposons donc de changer la signature de la fonction **charsort** en :

```
char * charsort (char * w, int len, bool (*comp) (char, char) );
```

Le pointeur de fonction **comp**, représente la fonction de comparaison entre deux caractères. Quelles sont les modifications à apporter au code et à la signature de la fonction **foo** ainsi qu'au code de la fonction **charsort** pour pouvoir trier un tableau de caractères en utilisant la fonction de comparaison pointée par **comp** ?
CONSIGNES : Après avoir numéroté les lignes du code source de votre fonction **charsort** de la question **Q3**, pour répondre à la question précédente, préciser **uniquement** les changements à apporter en indiquant les lignes où se produisent ces derniers.
CORRECTION
Pour charsort
Ligne 4 : modification de la signature de la fonction charsort et ajout d'un test sur comp

```
char * charsort(char * w, int len, bool (*comp) (char, char)){
  if(comp == NULL){
     fprintf(stderr, "comp is not valid!");
     return NULL;
  }
  ...
```

Ligne 19 : modification de l'appel de la fonction foo pour lui transmettre la fonction de comparaison

```
    char * smallest = foo(res+i,res+len-1,comp) ;
```

Pour foo
Ligne 4 : modification de la signature de la fonction foo et ajout d'un test sur comp

```
char * foo(char * start, char * end, bool (*comp) (char, char)){
  if(comp == NULL){
    fprintf(stderr, "comp is not valid!");
    return NULL;
  }
  ...
```

Ligne 5 : modification du test entre start et end en utilisant la fonction de comparaison

```
    if(comp(end, start)){
```

Ligne 12 : modification du test entre *tmp et val en utilisant la fonction de comparaison

```
    if(comp(*(tmp), val)) {
```

- **Q5 (2 point) :** En utilisant la dernière version de la fonction **charsort**, écrire les deux fonctions

char * increasingSorting(char * w, int len);
et **char * decreasingSorting(char * w, int len);**

qui retournent respectivement une copie de **w** trié dans l'ordre alphabétique et dans l'ordre inverse (*aide* : vous avez toute liberté de créer d'autres fonctions si nécessaire. Ces dernières ne devront pas être visibles en dehors de votre fichier source).
CORRECTION

```
1.  #include <stdio.h>
2.  #include <stdlib.h>
3.
4.  static bool inc(char * a, char * b){
5.      if((a == NULL)||(b == NULL)){
6.          fprintf(stderr, "invalid arguments!");
```

```
7.              exit(EXIT_FAILURE);
8.          }
9.          return a<b;
10.     }
11.     static bool dec(char * a, char * b){
12.       return !(inc(a,b));
13.     }
14.     char * increasingSorting(char * w, int len){
15.       return charsort(w,len,inc) ;
16.     }
17.     char * decreasingSorting(char * w, int len){
18.       return charsort(w,len,dec) ;
19.     }
20.
```

Exercice 3. (5 points)
BF est un langage de programmation minimaliste, inventé par Urban Müller en 1993. Il suit un modèle de machine simple, consistant en **un tableau d'octets initialisés à 0**, **d'une tête de lecture/écriture unique** (que l'on peut voir comme un simple pointeur sur le tableau) qui est positionnée sur le premier octet du tableau au démarrage et de **deux files d'octets** pour les entrées et sorties.
Les huit instructions du langage, chacune codée par un seul caractère, sont les suivantes :

Caract.	*Signification*
>	*incrémente (augmente de 1) le pointeur.*
<	*décrémente (diminue de 1) le pointeur.*
+	*incrémente l'octet du tableau sur lequel est positionné le pointeur (l'octet pointé).*
−	*décrémente l'octet pointé.*
.	*sortie de l'octet pointé (valeur ASCII).*
,	*entrée d'un octet dans le tableau à l'endroit où est positionné le pointeur (valeur ASCII).*
[	*saute à l'instruction après le] correspondant si l'octet pointé vaut 0.*
]	*retourne à l'instruction après le [si l'octet pointé ne vaut pas 0.*

Le programme suivant affiche le traditionnel "Hello World! " suivi d'un saut de ligne à l'écran. Noter que, par souci de lisibilité, le code a été divisé en plusieurs lignes et des commentaires ont été ajoutés. *BF* considère comme étant des commentaires tous les caractères ne correspondant pas à une instruction (c.f. tableau plus haut). **Il ne s'agit pas de comprendre le code suivant. Il est donné à titre d'exemple.**

```
++++++++++
[   Boucle initiale qui affecte des valeurs utiles au tableau
   >+++++++>++++++++++>+++>+<<<<-
]
    à la sortie de la boucle le tableau contient: 0 70 100 300 10 0 0 0 0 0 etc
>++.                    'H'    = 72  (70  plus 2)
>+.                     'e'    = 101 (100 plus 1)
+++++++.                'l'    = 108 (101 plus 7)
.                       'l'    = 108
+++.                    'o'    = 111 (108 plus 3)
>++.                    espace = 32  (30  plus 2)
<<+++++++++++++++.      'W'    = 87  (72  plus 15)
>.                      'o'    = 111
+++.                    'r'    = 114 (111 plus  3)
------.                 'l'    = 108 (114 moins 6)
--------.               'd'    = 100 (108 moins 8)
>+.                     '!'    = 33  (32  plus  1)
>.                      nouvelle ligne = 10
```

Nous souhaitons proposer un programme permettant d'interpréter et exécuter un programme décrit en ***BF* simplifié où une seule boucle peut exister dans le code** (comme dans l'exemple Hello World). **En d'autres termes, on ne peut pas avoir plus d'un caractère '[' ni plus d'un caractère ']' dans le code**.
Pour ce faire, nous considérerons que

- Le tableau d'octets est représenté par une variable globale définie comme suit en utilisant la constante `MEM_SIZE` (que vous n'avez pas à définir)

```
static unsigned char memory[MEM_SIZE] ;
```

- La tête de lecture/écriture est représentée par une variable `ptr` pointant initialement sur la première case de `memory`

```
static unsigned char* ptr = memory;
```

- Et que les files d'octets pour les entrées et sorties se feront sur les entrées/sorties standard (`stdin` et `stdout`) par des lectures et écritures de caractères.

▪ **Q1 (2 point) :** Proposer une traduction en langage C de chacune des instructions suivantes du langage BF.

Caractère	Taille possible de la réponse	Traduction en C
>	moins de 20 caractères	ptr = ptr +1 ;
<	moins de 20 caractères	ptr = ptr -1 ;
+	moins de 20 caractères	*(ptr) = *(ptr) + 1 ;
–	moins de 20 caractères	*(ptr) = *(ptr) - 1 ;
.	moins de 25 caractères	printf(« %c »,*(ptr)) ;
,	moins de 50 caractères	if(fgets(ptr,1,stdin)==NULL){exit(EXIT_FAILURE);}

- **Q2 (2 points)** : Proposer une implémentation de la fonction suivante

```
void bf_eval(char* code);
```

qui, considérant la chaîne de caractères `code` comme un programme *BF simplifié*, l'exécute. Les réponses à la précédente question peuvent vous aider ! **On considérera que le code fourni est valide. Vous n'avez donc pas à gérer les cas de code mal formé.**

CORRECTION

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

void bf_eval(char * code){
    if(code != NULL)){
       char *  starposloop=NULL ;
       char *  endposloop=NULL ;
       while((*code) != '\0'){
         switch(*code){
           case '>' : ptr = ptr + 1 ; break ;
           case '<' : ptr = ptr - 1 ; break ;
           case '+' : (*ptr) = (*ptr) + 1 ; break ;
           case '-' : (*ptr) = (*ptr) - 1 ; break ;
           case '.' : printf( « %c »,*(ptr)) ; break ;
           case ',' : if(fgets(ptr,1,stdin)==NULL){
                        exit(EXIT_FAILURE);
                      }
                      break ;
           case '[' : starposloop = code ;
                      endposloop = code ;
                      while(*(endposloop) != ']'){
                        endposloop++ ;
                      }
                      if(*(ptr) == '\0'){
                        code = endposloop ;
                      }
                      break ;
           case ']' : if(*(ptr) != '\0'){
                        code = startposloop ;
                      }
                      break ;
         }
         code++ ;
       }
    }
}
```

- **Q3 (1 points) :** Proposer une implémentation de la fonction suivante

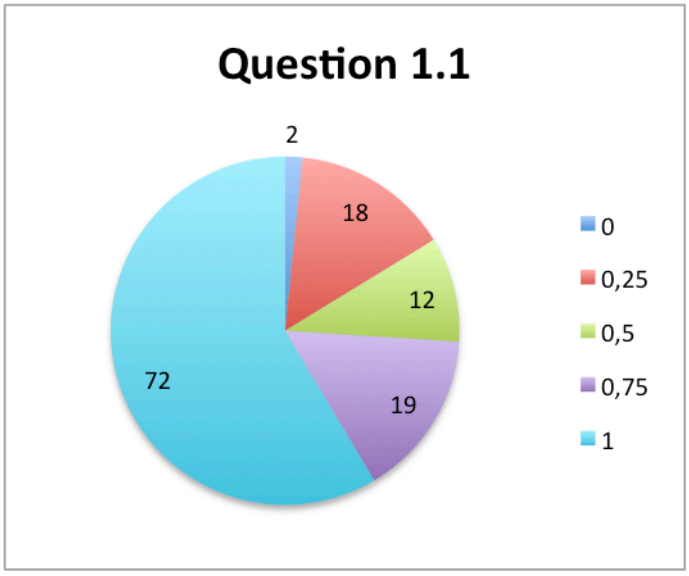
```
void bf_file(char* filename);
```

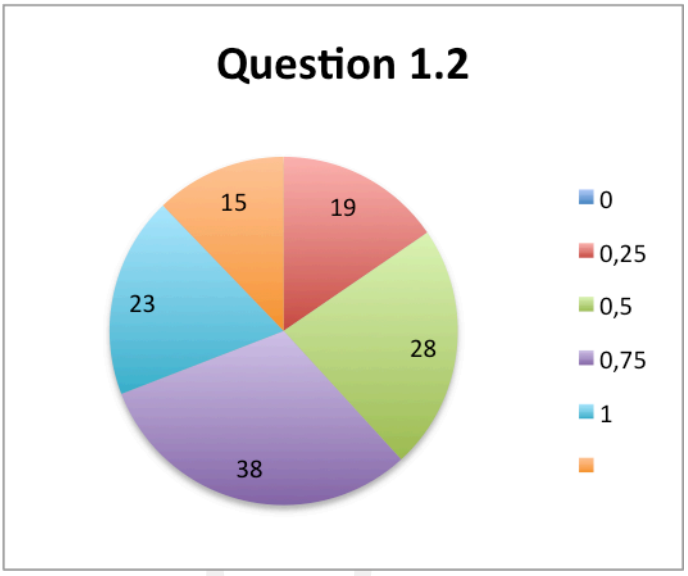
qui lit un programme *BF simplifié* depuis un fichier dont le chemin d'accès est représenté par la variable `filename`. Par simplicité, on considérera que le code *BF simplifié* est fourni dans la première ligne du fichier uniquement. **Rien ne vous empêche de répondre à cette question sans fournir une réponse à la précédente et en considérant la question 2 comme résolue.**

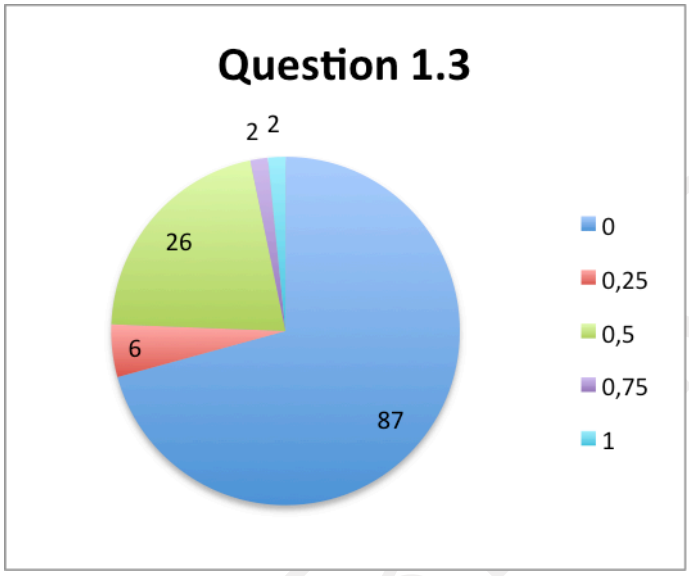
CORRECTION

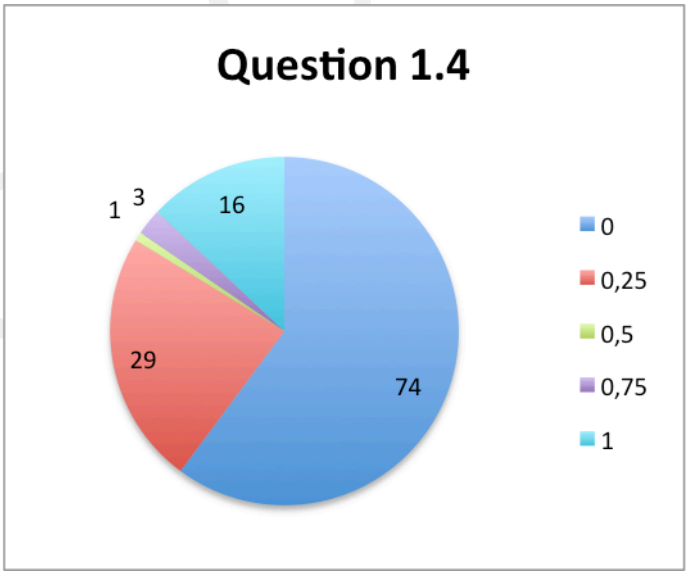
```
1.      #include <stdio.h>
37.     #include <stdlib.h>
38.
39.     void bf_file(char * filename){
40.         if(filename != NULL)){
41.            FILE * f = fopen(filename, « r ») ;
42.            if(f == NULL){
43.              fprintf(stderr, "invalid filename!");
44.              exit(EXIT_FAILURE);
45.            }
46.            fseek(f,0,SEEK_END);
47.            long size = 1+ftell(f);
48.            fseek(f,0,SEEK_SET);
49.            char * code = (char *) malloc(sizeof(char)*size);
50.            if(code == NULL){
51.              fprintf(stderr, "memory problem!");
52.              exit(EXIT_FAILURE);
53.            }
54.            if(fgets(code,size-1,f) == NULL){
55.              fprintf(stderr, "file reading problem!");
56.              exit(EXIT_FAILURE);
57.            }
58.            bf_eval(code) ;
59.          }
60.       }
```

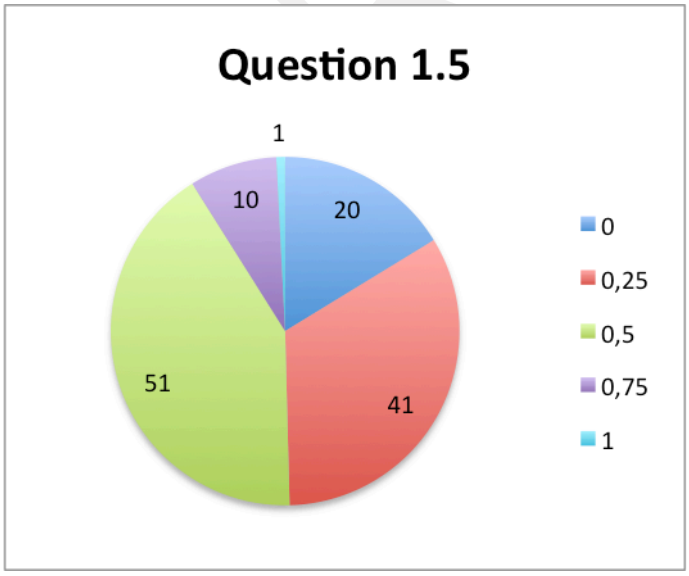
Répartition des points par question

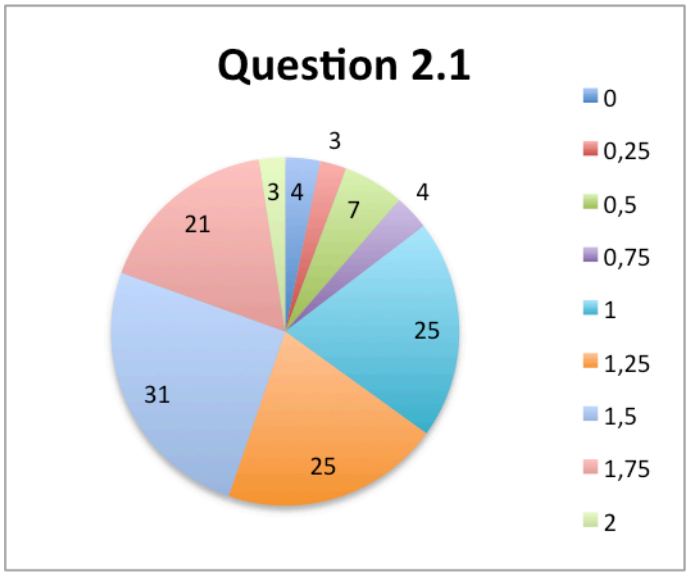


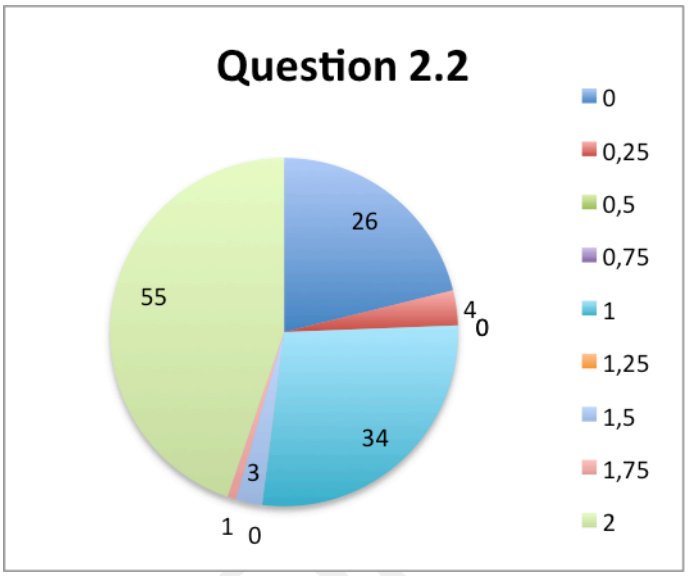


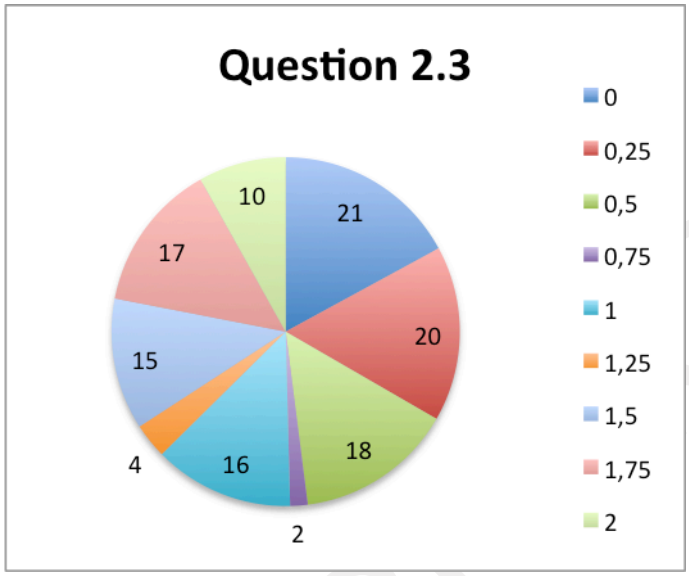


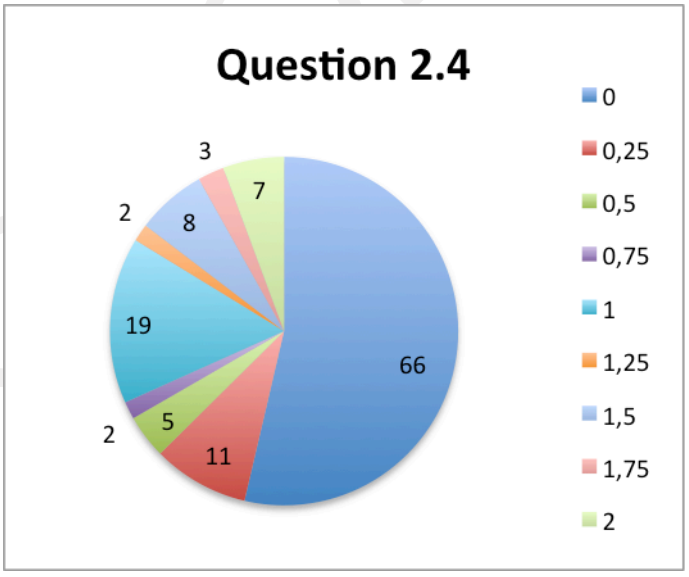


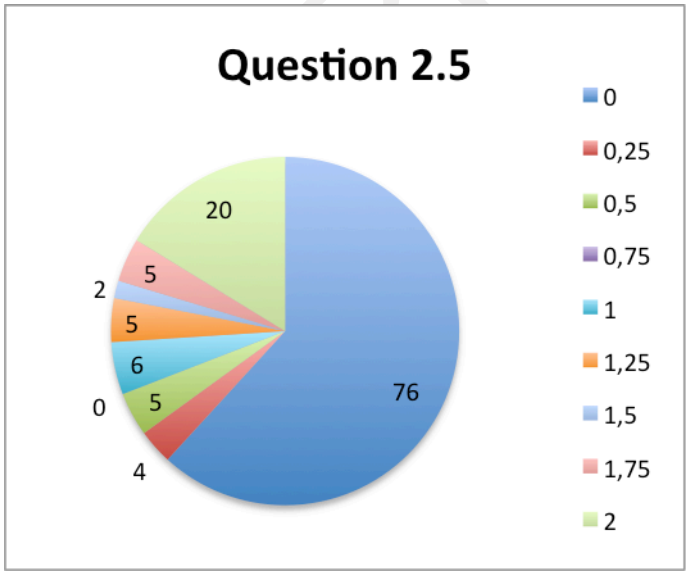


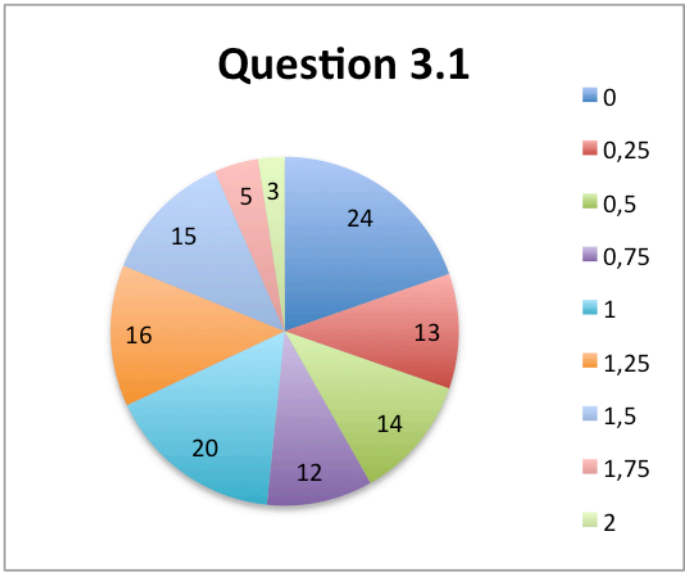

Question 2.1
3
4
7
4
25
25
31
21
3
0
0,25
0,5
0,75
1
1,25
1,5
1,75
2

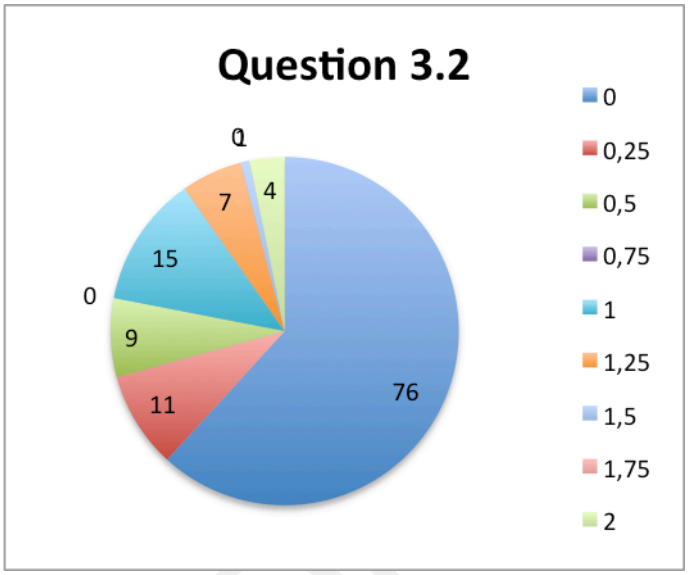

Question 2.2
26
4
0
34
3
1
0
55
0
0,25
0,5
0,75
1
1,25
1,5
1,75
2

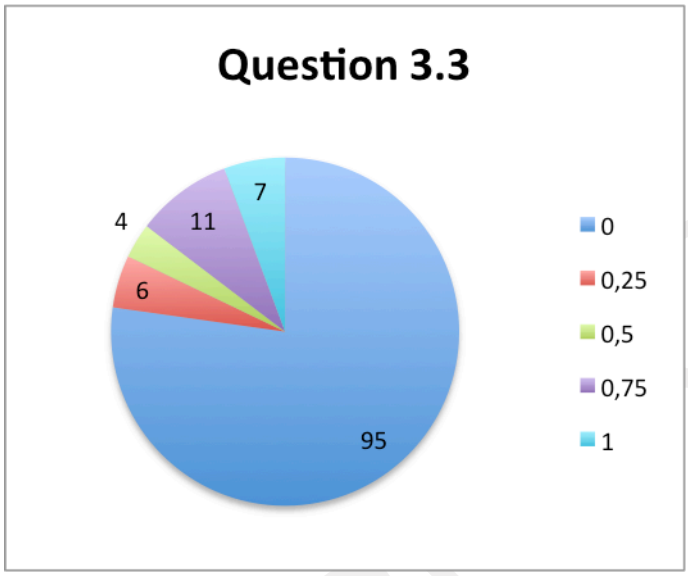

Question 2.3
21
20
18
2
16
4
15
17
10
0
0,25
0,5
0,75
1
1,25
1,5
1,75
2


Question 2.4
66
11
5
2
19
2
8
3
7
0
0,25
0,5
0,75
1
1,25
1,5
1,75
2


Question 2.5
76
4
5
0
6
5
2
5
20
0
0,25
0,5
0,75
1
1,25
1,5
1,75
2


Question 3.1
24
13
14
12
20
16
15
5
3
0
0,25
0,5
0,75
1
1,25
1,5
1,75
2


Question 3.2
76
11
9
0
15
7
0
4
0
0,25
0,5
0,75
1
1,25
1,5
1,75
2


Question 3.3
95
6
4
11
7
0
0,25
0,5
0,75
1