Corrigé du devoir en temps limité n°1

1 Un peu de C

1. *Rappeler les deux lignes permettant la compilation et l'exécution d'un code C. On supposera que notre code est dans un fichier nommé* ***fichier.c****.*

   `gcc -o fichier.out fichier.c` et **`./fichier.out`**.

2. *Écrire une fonction* ***`float maximum(float* tab, int n)`*** *qui calcule le maximum du tableau* ***`tab`****.*

   C'est une fonction qu'on a écrit en cours, en TP et revu en TD.

```c
float maximum(float* tab, int n){
  float max = tab[0];
  for(int i=0; i<n; i+=1){
    if (tab[i]>max){max=tab[i];}
  }
  return max;
}
```

3. *Pour chacune des versions suivantes de la fonction f, calculer la valeur renvoyée.*

```c
int f1 () {
   int res = 0;
   for(int i=0;i<6;i+=1){
     res+=i;
   }
   return res;
}
```

```c
int f2 () {
   int res = 0;
   for(int i=0;i<=6;i+=1){
     res+=i;
   }
   return res;
}
```

```c
int f3 () {
   int res = 0;
   for(int i=6;i>0;i-=1){
     res+=i;
   }
   return res;
}
```

```c
int f4 () {
   int res = 0;
   for(int i=6; i>=0; i+=1){
     res+=i;
   }
   return res;
}
```

   `f1` renvoie 15, **`f2`** renvoie 21, **`f3`** renvoie 21 et **`f4`** contient une boucle infinie et ne renvoie pas.

4. *Écrire une fonction* ***`int somme(int n)`*** *qui calcule la somme* $\sum_{k=0}^{n} k^2 + 3$.

```c
int somme(int n){
  int res = 0;
  for(int k=0;k<n;k+=1){
    res += k*k+3;
  }
  return res;
}
```

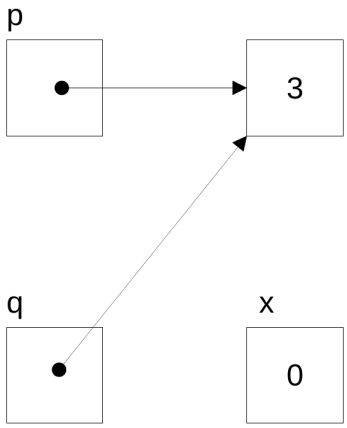
5. *Écrire une fonction* ***`int somme_puissances(int n, int x)`*** *qui calcule la somme* $\sum_{k=0}^{n} x^k$.

```c
int somme_puissances(int n,int x){
  int res = 0; //la somme
  int puissance_x = 1; //la puissance actuelle de x (pour éviter de calculer chaque puissance séparément.

  for(int k=0; k<=n; k+=1){
    res+=puissance_x;
    puissance_x *= x;
  }

  return res;
}
```

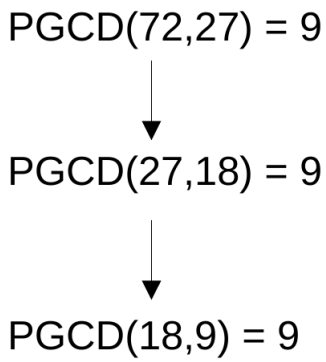
6. *On définit la suite $(u_n)_{n\in\mathbb{N}}$ suivante : $u_0 = 0$, $u_1 = 3$ et $\forall n \geq 2$, $u_n = 3 * u_{n-1} + 10 * u_{n-2} + 2$. Écrire une fonction* ***récursive*** ***`int calcul_u(int n)`*** *qui calcule le n-ième terme de la suite.*

```c
int calcul_u(int n){
    if(n==0){return 0;} //Cas de base
    else if (n==1) {return 3;}
    else{return 3*calcul_u(n-1)+10*calcul_u(n-2)+2;}
}
```

7. *Dessiner l'état de la mémoire après les lignes de code suivant (on supposera être dans le `main`) :*



8. *Dessiner une représentation des différents appels récursifs réalisés par la fonction suivante lors de l'appel `pgcd(72,27)`. On précisera la valeur de retour de chaque appel.*



9. *Écrire une fonction `float moyenne_sans_0(float* notes, int n)` qui prend en entrée un tableau de n notes et fait la moyenne des notes, en ignorant les notes valant 0 (absence au devoir).*

```c
float moyenne_sans_0(float* notes, int n){
  float res = 0;
  int nb_notes_non_nulles = 0; //Nombre de notes non nulles
  for(int i=0; i<n; i+=1){
    if(notes[i]!=0){
      res += notes[i];
      nb_notes_non_nulles+=1;
    }
  }
  return res/nb_notes_non_nulles;
}
```

2 Un peu de binaire

10. *Écrire les nombres suivants en binaire : 12, 142, 269.* 12 s'écrit 1100. 142 s'écrit 10001110. 269 s'écrit 100001101.
11. *Que représentent les écritures binaires suivantes : 1011, 1001010 ?*
    1011 représente 11. 1001010 représente 74.
12. *Effectuer les calculs suivants en binaire (en écrivant les étapes suivies et en posant les calculs) : 30-23, 100+12. On se limitera à des calculs sur un octet.*
    Pour faire $30-23$, on traduit 30 et 23 en binaire sur un octet : 00011110 et 00010111. On prend le complément à 2 de 23 : 11101001.
    Puis on pose l'addition.

(1)	1	1	1	1				
	0	0	0	1	1	1	1	0
+	1	1	1	0	1	0	0	1
(1)	0	0	0	0	0	1	1	1

Le résultat est 00000111 puisqu'on ignore la retenue qui dépasse. Ce nombre s'écrit 7 en décimal.

Pour faire 100 + 12 on traduit les nombres en binaire et on pose l'addition.

				1	1			
	0	1	1	0	0	1	0	0
+	0	0	0	0	1	1	0	0
	0	1	1	1	0	0	0	0

Le résultat est 112 (logique).

13. *Quel est le plus grand nombre qu'on peut représenter en binaire sur 2 octets ? Sur 4 octets ? (ici on parle bien de binaire classique, pas de la représentation des entiers relatifs)*
    Le plus grand nombre représentable sur n octets, n quelconque, est celui dont l'écriture comprend $8n$ bits 1. C'est aussi $2^n - 1$ car 2^n s'écrit avec un 1 suivi de n bits 0.
14. *Soit $n \in \mathbb{N}$. Combien de chiffres en binaire (ou bits) faut-il pour représenter 2^n ? Pour représenter $2^n - 1$?*
    Pour représenter 2^n il en faut $n + 1$. Pour $2^n - 1$ il en faut n, sauf si $n = 0$, auquel cas il faut 1 bit pour représenter 0.

3 Un peu de bash

Le but de cet exercice est d'utiliser quelques commandes de bash. On se place à la racine `/`.

15. *Se ramener dans votre répertoire personnel.* **`cd`**
16. *Créer un nouveau dossier* ***`tp`***.
    `mkdir tp`
17. *Aller dans le dossier* ***`tp`***. **`cd tp`**
18. *Quelle commande permet d'afficher le contenu du dossier ?* **`ls`**
19. *Créer un nouveau fichier* ***`tp.c`***. **`touch tp.c`**

On suppose que vous avez écrit du code dans le fichier et l'avez compilé en un fichier **`tp.out`**.

Lorsque vous essayez d'exécuter, le terminal vous réponds que le fichier **`tp.out`** n'est pas exécutable.

20. *Quelle commande faut il utiliser pour vérifier que notre fichier a bien les droits d'exécution ?* **`ls -l`**

Le résultat est le suivant :

```
-rw-r--r-- 1 vous users 1690 4 octobre 2025 tp.c
-rw-r--r-- 1 vous users 1690 4 octobre 2025 tp.out
```

21. *Changer les permissions pour que vous puissiez exécuter* ***`tp.out`***. **`chmod u+x tp.out`**
22. *Modifier les permissions pour que tout le groupe* ***`users`*** *(votre groupe) puisse écrire dans le fichier* ***`tp.c`***. **`chmod g+w tp.c`**

4 Un peu de preuves de programmes

23. *Qu'est-ce qu'un variant de boucle ?* Un variant de boucle est une quantité entière, fonction des variables du programme, qui reste positive est qui décroit strictement à chaque itération de la boucle.
24. *Qu'est-ce qu'un invariant ?* Un invariant est une propriété qui, si elle est vraie au début d'une itération de la boucle, est préservée par cette itération.

On considère le code suivant, qui prend en entrée un tableau d'entiers, sa taille et un entier x :

```
int mystere(int* tab, int n, int x){
 int res = 0;
 int i = 0;
 while(i<n){
   if(tab[i]==x){res+=1;}
```

```
    i+=1;
  }

  return res;
}
```

25. *Soit* **`tab`** $= [1, 2, 1, 1, 3, 7, 1, 1]$, $n = 8$ *et* $x = 1$. *Que renvoie la fonction pour ces entrées ?*
    La fonction renvoie 5.
26. *Écrire sa spécification.*
    Précondition : n est bien la taille du tableau (et est ≥ 0).
    Postocndition : la fonction renvoie le nombre de fois où la valeur x est présente dans le tableau. Si la valeur n'est jamais présente, 0 est renvoyé.
27. *Déterminer et prouver un variant de la boucle while.*
    Un variant est $n - i$. En effet c'est une quantité entière, qui est positive au début (quand $i = 0$) et qui le reste puisque la boucle s'arête dès que $n - i <= 0$.
    À chaque itération de la boucle, i est augmenté de 1, donc $n - i$ diminue strictement.
28. *Qu'est-ce que la question précédente permet de conclure ?*
    Avoir un variant nous permet d'affirmer que la boucle (et donc la fonction) termine sur toutes ses entrées admissibles.
29. *Quel invariant permettrait de prouver la correction de la fonction ? Aucune preuve n'est attendue.*
    Un invariant utile est "**`res`** est le nombre d'apparitions de x dans le tableau jusqu'à l'indice i".

5 Un peu de tri

Dans cette section on va étudier un algorithme de tri de tableaux qu'on appellera tri par positions.

On considère un tableau **`t`** de taille **`n`**. Le principe est le suivant :

- On crée un deuxième tableau **`t2`** de même taille que **`t`**.
- Pour chaque élément **`t[i]`** du tableau, on détermine son rang **`r`** dans le tableau trié en comptant le nombre d'éléments qui lui sont strictement inférieurs.
- Puis pour chaque élément, on le place dans la case **`r`** de **`t2`**.
- Le nouveau tableau est alors une version triée du tableau initial.

Dans un premier temps, on suppose qu'aucun élément n'est un doublon, c'est à dire que pour $i \neq j$ des indices du tableau, **`t[i]`**$\neq$**`t[j]`**.

30. *Écrire une fonction* **`int`** `compte_plus_petit(int* t, int n, int i)` *qui prend en entrée un tableau, sa taille et un indice i du tableau et renvoie le nombre de valeurs dans le tableau qui sont strictement inférieures à* **`t[i]`**.

```
int compte_plus_petit(int* t, int n, int i){
  int res = 0;
  for(int j=0; j<n; j+=1){
    if(t[i]>t[j]){ //On a trouvé un élément strictement plus petit, on ajoute 1 à res
      res+=1;
    }
  }
  return res;
}
```

31. *Écrire une fonction* **`int`**`* tri_position(int* t, int n)` *qui renvoie un nouveau tableau qui contient les mêmes valeurs que t, mais triées dans l'ordre croissant. On utilisera la méthode décrite.*

```
int* tri_position(int* t, int n){
  int* new_t = malloc(n*sizeof(int));
  for(int i=0; i<n;i+=1){
    int r = compte_plus_petit(t, n, i); //Trouver le rang dans new_t
    new_t[r]=t[i]; //Copier la valeur au bon rang
  }
  return new_t;
}
```

Dans un deuxième temps, on va généraliser au cas où le tableau peut contenir des doublons.

32. *Montrer sur un exemple pourquoi le fait d'avoir un doublon pose problème avec notre méthode.*
Pour le tableau suivant : $t = [1, 4, 2, 3, 1, 5, 0]$, le nombre d'éléments strictement plus petits que 1 est 0, donc le rang de 1 dans **new_t** est 0. Cependant il y a deux 1, et les deux vont être mis dans la case 0. La case 1, elle, n'aura pas été modifiée.

33. *En utilisant des phrases et des schémas, expliquer une manière de résoudre le problème.*
On peut commencer par remplir **new_t** de 0 pour éviter des suprises : on rapelle que malloc fournit un tableau dont les valeurs des cases peuvent être n'importe quoi.
Ensuite, pour chaque i, si **t[i]** n'est pas 0, on calcule le nombre d'éléments strictement inférieurs r, puis on regarde **new_t[r]**. Si **new_t[r]** contient **t[i]**, alors on va à la case $r+1$. Si cette case contient aussi déjà **t[i]**, on va à la case suivante, etc jusqu'à trouver une case qui contient un 0, dans laquelle on peut mettre **t[i]**.
On remarquera que cette méthode met gratuitement les 0 à leur place.
Un exemple pour $t = [1, 4, 2, 3, 1, 5, 0]$, **new_t** évolue de la manière suivante :
$[0, 0, 0, 0, 0, 0, 0] \to [0, 1, 0, 0, 0, 0, 0] \to [0, 1, 0, 0, 0, 4, 0] \to [0, 1, 0, 2, 0, 4, 0] \to [0, 1, 0, 2, 3, 4, 0] \to [0, 1, 1, 2, 3, 4, 0] \to [0, 1, 1, 2, 3, 4, 5]$

34. *Programmer votre méthode. On peut réécrire entièrement le fonction `tri_position` ou écrire un bout de code à rajouter à la fonction (en précisant où on le rajoute).*
On réécrit intégralement `tri_position`.

```c
int* tri_position(int* t, int n){
  int* new_t = malloc(n*sizeof(int));

  //Remplir de 0
  for(int i=0; i<n; i+=1){
    new_t[i]=0;
  }

  for(int i=0; i<n;i+=1){
    if(t[i]!=0){
      int r = compte_plus_petit(t, n, i); //Trouver le rang dans new_t

      while(new_t[r]==t[i]){ //Tant que les cases sont occupées par un doublon, on se décale d'une case
        r+=1;
      }
      new_t[r]=t[i]; //Copier la valeur au bon rang
    }
  }
  return new_t;
}
```

Dans un troisième temps, on veut modifier notre tri pour qu'il soit **en place**. Cela signifie qu'on a plus le droit d'utiliser un deuxième tableau et on doit modifier le tableau initial.

35. *(Difficile) Écrire une fonction `void tri_position_en_place(int* t, int n)` qui trie le tableau par positions, mais sans utiliser un nouveau tableau.*
On procède de la même manière : pour un indice `pos` du tableau, on calcule r sa position théorique dans le tableau trié. Ensuite, plutôt qu'essayer de mettre notre valeur à la case r dans un nouveau tableau, on la met à la case r dans le même tableau.
Cependant la case r n'est à priori pas vide, donc on se retrouve avec une nouvelle valeur `t[r]` dont il faut trouver la place.
En itérant ainsi, on finit par tomber sur la valeur qui est censée aller à la poisition `pos`.
Une fois le raisonnement fini à la position `pos`, le tableau peut être trié, mais la plupart du temps il ne le sera pas. Il faut donc nous refaire le raisonnement pour `pos+1` et tester ainsi toutes les positions possibles.
On remarquera qu'au cours de l'algorithme il n'est pas possible de mettre un élément à une position plus petite. Le fait que $pos \leq r$ est un invariant (qui pourrait être utile pour prouver la correction). C'est ce fait qui garantit la terminaison de la fonction.
Un dernier détail à gérer est l'ajout des doublons dans le raisonnement, qui se fait similairement aux questions précédentes.

```c
void tri_position_en_place(int* t, int n){
 int pos = 0; //on commence avec la case 0

  while(pos<n){
      int r = compte_plus_petit(t, n, pos); //position triée de t[pos] (sans compter les doublons)

      // Gestion des doublons : si la case r est déjà occupée t[pos] et que ce n'est PAS la case pos.
```

```
        while(r!=pos && t[r]==t[pos]){ //On cherche la prochaine case qui ne contient pas t[pos]
          //Cas particulier : si on tombe sur r==pos durant cette recherche, t[pos] est déjà à
sa place
          r+=1;
        }
        //r est maintenant forcément la bonne position de t[pos]

        if (r!=pos){ //Si la nouvelle position n'est pas l'ancienne
            //On échange les valeurs aux cases r et pos
                int sto = t[r];
                t[r] = t[pos];
                t[pos] = sto;

                //La nouvelle valeur de la case pos n'est pas forcément à sa place,
                //donc on reste à la position pos
        }
        else{ //La nouvelle position est l'ancienne. La position pos est résolue, on va voir la suite
            pos=pos+1;
        }
    }
  }
```