UNIVERSITÉ CÔTE D'AZUR

Partiel de Programmation C

L'usage ou la possession de tout appareil électronique sera considéré comme de la fraude. Documents interdits.

Tous les codes devront être écrits en **Langage C ANSI**.

La notation est donnée à titre indicatif.

DATE : 19/03/2026 DURÉE : 1h30 NOTE : ...../20

NOM : .....

PRÉNOM : .....

NUMÉRO D'ÉTUDIANT : .....

Exercice 1 : Syntaxe (3 points)

1. On considère la fonction ci-dessous. Réécrire cette fonction, en langage C, de manière plus lisible. Votre code devra pouvoir être transposé rapidement en Python et n'utilisera donc que des structures (`while`, `if`) communes aux langages de programmation traditionnels.

```c
void salvador(int i) {
    for (; 1-i; i=(i%2 ? 3*i+1 : i/2)) printf("%d -> ",i);
    printf("%d\n",i);
}
```

```c
void salvador(int i) {
    while (i != 1) {
        printf("%d -> ",i);
        if (i%2 == 0) {
            i = i/2;
        } else {
            i = 3*i+1;
        }
    }
    printf("%d\n",i);
}
```

2. Qu'affiche l'appel de `salvador(12)` ?

```
//
12 -> 6 -> 3 -> 10 -> 5 -> 16 -> 8 -> 4 -> 2 -> 1
```

3. Bonus : démontrer que la fonction termine toujours. La question étant sensiblement plus compliquée que les autres questions, nous vous conseillons de tenter la démonstration une fois les autres questions terminées.

```
//
C'est la célèbre conjecture de Syracuse. Personne n'a encore trouvé la démonstration…
```

Exercice 2 : Des tranches de chaînes (7 points)

1. Écrire une fonction `int longueur(char *chaine)` renvoyant, comme son nom l'indique, la longueur d'une chaîne. Il est évidement interdit d'utiliser la fonction `strlen`.

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

2. Écrire une fonction `char *copier(char *ch)` qui construit et renvoie une copie, allouée sur le tas, de la chaîne donnée en paramètre.

```c
char* copier(char *ch) {
    int n = longueur(ch);
    char* res = malloc((n+1)*sizeof(char));
    int i = 0;
    for (i=0; i<=n; i++) {
        res[i] = ch[i];
    }
    return res;
}
```

3. Écrire une fonction `char *extraire(char *ch, int a ,int b)` qui copie, sur le tas là encore, une sous-chaîne comprise entre les indices `a` (inclus) et `b` (exclu). Par exemple, `extraire("0123456", 2, 5)` renverra la chaîne `"234"`.

```c
char * extraire(char *ch, int a ,int b) {
    int taille = b - a + 2;
    char* res = malloc(taille * sizeof(char));
    int i;
    for (i=0 ; i+a<b ; i++) {
        res[i] = ch[i+a];
    }
    res[i]= 0;
    return res;
}
```

4. Afin de pouvoir travailler sur des sous-chaînes sans avoir besoin d'allouer sur le tas, on souhaite créer un type `slice` construit à partir d'une structure contenant trois champs : une chaîne de caractères `chaine`, un indice entier de début `deb` et un autre indice entier `fin` indiquant l'indice de fin (exclu). Définissez un tel type.

```c
typedef struct slice { /* 1 point */
    char* chaine;
    int deb;
    int fin;
} slice;
```

5. Écrire alors une fonction `slice extraire_slice(char *ch, int a, int b)` qui renvoie la *slice* associée à la sous-chaîne de `ch` entre `a` et `b`.

```c
slice extraire_slice(char* ch, int a, int b) { /* 0,5 points */
    slice s;
    int n = longueur(ch);
    s.chaine = ch;
    s.deb = a;
    s.fin = b;
    return s;
}
```

6. Écrire la fonction `void afficher_slice(slice s)` qui affiche la sous-chaîne correspondante à la *slice*.

```c
/* 1 points */
void afficher_slice(slice s) {
    int i;
    for (i=s.deb; i<s.fin; i++) {
        printf("%c", s.chaine[i]);
    }
}
```

7. Écrire la fonction `char *exporter_slice(slice s)` qui construit sur le tas et renvoie la sous-chaîne associée à la *slice* donnée en paramètre.

```c
/* 0,5 points */
char *exporter_slice(slice s) {
    return extraire(s.chaine, s.deb, s.fin);
}
```

Exercice 3 : Liste chaînée à deux pointeurs (7 points)

Traditionnellement, une liste chaînée permet l'accès rapide et efficace à la première valeur de la liste. Pour ajouter un élément en fin, il est alors nécessaire de la parcourir en intégralité. On souhaite, dans cet exercice, améliorer le type liste chaînée pour pouvoir ajouter des éléments en début et en fin de liste en temps constant (sans parcours).

1. Commencer par créer un type `struct cellule` contenant un champ `contenu` de type « entier » et un champ `suivant` du type « pointeur vers `struct cellule` ». Ce type correspond à une liste chaînée traditionnelle.

```c
struct cellule { /* 0.5 points */
    int contenu;
    struct cellule * suivant;
};
```

2. Créer maintenant un type `struct liste` contenant deux pointeurs. Un pointeur `deb` vers la première cellule de la liste et un pointeur `fin` vers la dernière cellule (non `NULL`) de la liste.

```c
struct liste { /* 0.5 points */
    struct cellule * deb;
    struct cellule * fin;
};
```

3. Renommer le type `struct liste` en `liste`

```c
/* 0.5 points */
typedef struct liste liste;
```

4. Écrire une fonction `liste nouvelle_liste(void)` qui crée une liste vide. Les deux pointeurs `deb` et `fin` seront initialisés à `NULL`.

```c
liste nouvelle_liste() { /* 0.5 point */
    liste m;
    m.deb = NULL;
    m.fin = NULL;
    return m;
}
```

5. Écrire une fonction `void init_liste(liste *m, int valeur)` qui modifie une liste vide `m` pour lui ajouter une unique cellule.

```c
void init_liste(liste* m, int valeur) { /* 1 points */
    struct cellule * c = malloc(sizeof(struct cellule));
    c->contenu = valeur;
    c->suivant = NULL;
    m->deb = c;
    m->fin = c;
}
```

6. Écrire une fonction `void ajout_debut(liste *m, int valeur)` qui ajoute un élément en tête de liste. On fera bien attention à gérer le cas où la liste est vide.

```c
void ajout_debut(liste* m, int valeur) { /* 1.5 points */
    struct cellule * c;
    if (m->fin == NULL) {
        init_liste(m, valeur);
    } else {
        c = malloc(sizeof(struct cellule));
        c->contenu = valeur;
        c->suivant = m->deb;
        m->deb = c;
    }
}
```

7. Écrire une fonction `void ajout_fin(liste *m, int valeur)` qui ajoute un élément en fin de liste. Il faudra ajouter une cellule à la fin de la liste chaînée (sans la parcourir) et modifier les pointeurs de `m` en conséquent. On fera attention à bien gérer le cas où la liste est vide.

```c
void ajout_fin(liste* m, int valeur) { /* 1.5 points */
    struct cellule * c;
    if (m->fin == NULL) {
        init_liste(m, valeur);
    } else {
        c = malloc(sizeof(struct cellule));
        c->contenu = valeur;
        c->suivant = NULL;
        m->fin->suivant = c;
        m->fin = c;
    }
}
```

8. Écrire la fonction `void liberer(liste m)` qui libère tous les éléments alloués sur le tas de la liste `m` donnée en paramètre.

```c
/* 1 points */
void liberer(liste m) {
    struct cellule * c;
    struct cellule * suivant;
    for (c=m.deb; c!=NULL; c=suivant) {
        suivant = c->suivant;
        free(c);
    }
}
```

Exercice 4 : Calcul de nombres premiers (3 points)

Cet exercice utilise le type de liste construit à l'exercice précédent.

1. On se donne en paramètre une liste `m` contenant un ensemble d'entiers premiers et un nombre `k`. Écrire la fonction `int multiple(liste m, int k)` qui renvoie *vrai* si k possède un diviseur dans la liste `m` et *faux* sinon. Les booléens vrais et faux devront être codés selon l'usage en C.

```c
/* 1,5 points */
int multiple(liste m, int k) {
    struct cellule * c;
    for (c=m.deb; c!=NULL; c=c->suivant) {
        if (0 == (k % (c->contenu))) {
            return 1;
        }
        if (c->contenu*c->contenu > k) return 0;
    }
    return 0;
}
```

2. À partir de la fonction précédente, écrire une fonction `liste calcul_premiers(int limite)` qui commence à créer une liste ne contenant que le nombre premier 2, puis teste tous les nombres impaires plus grands que 2 en les ajoutant à la fin de la liste s'ils sont premiers (c'est à dire s'il ne possède pas de diviseurs premiers plus petit qu'eux). Votre liste devra contenir tous les nombres premiers jusqu'à `limite` compris.

```c
/* 1,5 points */
liste calcul_premiers(int limite) {
    int i;
    liste p;
    init_liste(&p, 2);
    for (i=3; i<=limite; i+=2) {
        if (!multiple(p, i)) {
            ajout_fin(&p, i);
        }
    }
    return p;

}
```