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);
}
```

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

.....

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

.....

.....

.....

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.

.....

.....

.....

.....

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`.

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

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"`.

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.

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`.

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

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.

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.

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.

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

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

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

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.

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.

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

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.

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.