Informatique
MP2I 2022/2023 — DS4 — CIV

Durée totale 2h30.
Tout document interdit. Tout dispositif électronique interdit.
On vous demande d'écrire lisiblement et de bien présenter. Les programmes en particulier doivent être bien indentés et lisibles.

1 C et mémoire

On considèrera dans ce sujet que tous les en-têtes ont déjà été inclus en C.

Q.1

Expliquer brièvement l'organisation de la mémoire d'un programme compilé.
On tachera de faire un diagramme indiquant les différentes zones mémoire ainsi que leur rôle.

R.1

voir cours.

Q.2

Si `i` est une valeur du type `int` et que `p` et `q` sont des pointeurs du type `int *`, indiquez les instructions d'affectation licites :
a) `p = i;` b) `*p = &i;` c) `&p = q;`
d) `p = &q;` e) `p = *&q;` f) `p = q;`
g) `p = *i;` h) `*p = q;` i) `*p = *q;`

R.2

Les instructions e, f et i sont licites.
Pour s'en convaincre, on pouvait faire un tableau en indiquant les types à gauche et à droite du symbole `=` :

a	`int *`	=	`int`
b	`int`	=	`int *`
c	`int **`	=	`int *`
d	`int *`	=	`int **`
e	`int *`	=	`int *`
f	`int *`	=	`int *`
g	`int *`	=	`invalide`
h	`int`	=	`int *`
i	`int`	=	`int`

Q.3

— Écrire une fonction de prototype

```
void max_min(int a[], int n, int *min, int *max);
```

qui prend en entrée un tableau `t` de `n` entiers, supposé non vide, et place la plus petite valeur dans `min` passé par pointeur, et sa plus grande valeur dans `max`.
— Écrire une fonction `main` typique qui testerait cette fonction sur le tableau `{1, 5, -1, 3}` en affichant la réponse sous la forme

```
min=-1 max=5
```

R.3

—
```
void max_min(int a[], int n, int *min, int *max)
{
    *min = a[0];
    *max = a[0];
    for(int i = 1; i < n; i++)
```

```
        {
            if(*min > a[i])
                *min = a[i];
            if(*max < a[i])
                *max = a[i];
        }
    }
— int main(void) {
        int t[] = { 1, 5, -1, 3 };
        int max, min;
        max_min(t, 4, &min, &max);
        printf("min=%d max=%d\n", min, max);
    }
```

Q.4

Écrire une fonction de prototype

```
void find_two_largest(int a[], int n,
    int *largest, int *second_largest);
```

qui prend en entrée un tableau `t` de `n` entiers, en supposant $n > 1$, et place la plus grande valeur présente dans `largest` et la seconde plus grande dans `second_largest`.
Exemple : pour `{ 1, 4, 5, 2, 4 }` on va donc avoir 5 dans `largest` et 4 dans `second_largest`.

R.4

Un classique des fausses questions faciles. Il y a deux pièges :
- on ne peut pas initialiser aux deux premières valeurs parce qu'elles peuvent être égales. Il faut donc chercher les deux premières valeurs **différentes**.
- il faut bien faire attention à ne pas faire deux if successifs dans la boucle car le premier if perturbe le second.

```
void find_two_largest(int a[], int n,
    int *largest, int *second_largest)
{
    *largest = a[0];
    int i = 1;
    while(a[i] == *largest)
        i++;
    if(a[i] > *largest) {
        *second_largest = *largest;
        *largest = a[i];
    } else {
        *second_largest = a[i];
    }
    for(int j = i+1; j < n; j++) {
        if (a[i] > *largest) {
            *second_largest = *largest;
            *largest = a[i];
        } else if (a[i] > *second_largest)
            *second_largest = a[i];
    }
}
```

Q.5

On considère les structures suivantes :

```
struct point {
    int x;
    int y;
};
```

```
struct rectangle {
    struct point upper_left;
    struct point lower_right;
};
```

Écrire une fonction de prototype

```
rectangle *mon_rectangle();
```

qui **alloue** et renvoie un `rectangle` de coin supérieur gauche de coordonnées $(10, 25)$ et de coin inférieur bas $(20, 15)$.

R.5

```
rectangle *mon_rectangle() {
    rectangle *r = malloc(sizeof(struct rectangle));
    r->upper_left.x = 10;
    r->upper_left.y = 25;
    r->lower_right.x = 20;
    r->lower_right.y = 15;
    return r;
}
```

Q.6

Écrire une fonction de prototype

```
int *create_array(int n, int initial_value);
```

qui alloue un tableau de n entiers et initialise chaque case avec la valeur `initial_value`.

R.6

```
int *create_array(int n, int initial_value)
{
    int *t = malloc(n * sizeof(int));
    for(int i = 0; i < n; i++)
        t[i] = initial_value;
    return t;
}
```

2 OCaml et complexité

C'est le grand jour où vous fêtez avec vos amis et votre famille votre réussite aux concours. Vous organisez un grand repas et vous souhaitez, par commodités, placer le maximum de convives sur une même table.
Problème : certains de vos convives ne s'entendent pas [1] et ne souhaitent pas être placés à la même table. Comment allouer des tables aux convives en essayant d'en avoir le moins possibles ?
On considère que chacun de vos n convives est identifié de manière unique par un numéro dans $[\![0, n-1]\!]$ et que l'on dispose d'une matrice :

```
val ami : bool matrix matrix
```

de dimension (n, n) telle que `ami.(i).(j)` soit vrai lorsque `i` et `j` sont amis. **Cette relation étant considéré symétrique** on a toujours `ami.(i).(j) = ami.(j).(i)`.
Un tableau `table` de longueur n qui associe à chaque convive un numéro dans $[\![0, p-1]\!]$ est appelée une affectation des convives à p tables. Ainsi `table.(i)` vaut `k` quand on place le convive `i` à la table `k`.

Q.7

Écrire une fonction de prototype

```
val valide : int array -> bool
```

1. sûrement une dispute sur leur langage de programmation préféré

qui s'assure que l'on n'a jamais placé à une même table deux personnes qui ne sont pas amies.

R.7

```
let valide table =
    let n = Array.length table in
    let is_valide = ref true in
    for i = 0 to n-1 do
        for j = i+1 to n-1 do
            (* pas amis et à la même table *)
            if not ami.(i).(j) && table.(i) = table.(j)
            then is_valide := false
        done
    done;
    !is_valide
```

Q.8

Quelle est la complexité de la fonction précédente ?

R.8

On parcourt chaque case de la moitié de la matrice, donc $O(n^2)$ cases et pour chaque case on effectue un traitement en $O(1)$, donc la complexité est en $O(n^2)$.
Note : si les personnes sont presque toutes amies, il serait plus raisonnable de travailler sur la liste des inimitiés.

On va essayer de construire une affectation valide de manière incrémentale ainsi :
- au départ on initialise le tableau `table` avec des valeurs quelconques (prenez `-1` par exemple)
- Pour i allant de 0 à n-1
  - on suppose qu'il y a un numéro valide dans `table` pour tous les convives de numéro `< i` et qu'il n'existe pas de couple (j, k) où $j, k < i$ qui ne soient pas amis et tels que `table.(j) = table.(k)`.
  - on calcule la liste des numéros de tables de tous les convives de numéro $< i$ qui ne sont pas amis avec i.
  - on en déduit le plus petit numéro ≥ 0 de table qui ne soit pas dans cette liste
  - on affecte ce numéro à i en le plaçant dans `table.(i)`.

Q.9

On suppose qu'il y a 6 convives, de 0 à 5, et que les inimitiés sont les suivantes :
- 0 et 2 sont ennemis
- 2 et 3 sont ennemis
- 2 et 4 sont ennemis
- 3 et 4 sont ennemis
- 1 et 5 sont ennemis

1. Dérouler l'algorithme précédent avec précision et en déduire une affectation en trois tables.
2. Peut-on faire mieux ?

R.9

1. On va construire `i -> table.(i)` de manière incrémentale :
   - $0 \mapsto 0$
   - $1 \mapsto 0$
   - $2 \mapsto 1$ car il a 0 comme ennemi < 2 et qu'il est déjà à la table 0.
   - $3 \mapsto 0$ car il a 2 comme ennemi < 3.
   - $4 \mapsto 2$ car il a 2 et 3 comme ennemi < 4.
   - $5 \mapsto 1$ car il a 1 comme ennemi < 5.
2. On ne peut pas faire mieux que trois tables, car comme 2, 3 et 4 ne s'apprécient pas deux à deux, il faut les placer sur trois tables différentes. La solution précédente est donc optimale.

Q.10

Écrire une fonction de prototype

```
val tables_ennemis : int array -> int -> int list
```

tel que l'appel à `table_ennemis table i` renvoie la liste des numéros de table (avec possiblement des répétitions) des ennemis du convive i dont le numéro est $< i$.

R.10

```
let tables_ennemis table i =
    let rec cherche j =
        if j < 0
        then []
        else let l = cherche (j-1) in
             if not ami.(i).(j)
             then table.(j) :: l
             else l
    in cherche (i-1)
```

Q.11

Quelle est la complexité de cette fonction ?

R.11

C'est la boucle induite par la fonction récursive qui compte car on n'y effectue que des opérations en temps constant, donc la complexité est en $O(i)$.

Q.12

Écrire une fonction de prototype

```
val plus_petit_absent : int list -> int
```

qui prend une liste d'entiers et renvoie le plus petit entier naturel qui n'est pas présent dans la liste.
Votre réponse devra être linéaire pour avoir tous les points, mais un algorithme non linéaire aura au moins la moitié des points.

R.12

Voir réponse dans la correction du TP.

Q.13

En déduire une fonction de prototype

```
val affecte : unit -> int array
```

qui réalise l'algorithme d'affectation présenté ci-dessus.

R.13

```
let affecte () =
    let n = Array.length ami in
    let table = Array.make n 0 in
    for i=1 to n-1 do
        let l = table_ennemis table i in
        table.(i) <- plus_petit_absent l
    done;
    table
```

Q.14

Quelle est sa complexité en supposant que `plus_petit_absent` est linéaire ?

R.14

Les deux appels dans la boucle sont en $O(i)$, donc, comme $i \leq n$, la complexité totale est en $O(n^2)$.

Q.15

Établir un petit contre-exemple démontrant que cette fonction ne réalise pas le plus petit nombre de table.

R.15

- — 0 est ennemi avec 1
  - — 1 est ennemi avec 0
  - — 2 est ennemi avec 3
  - — 3 est ennemi avec 1 et 2

  On va dérouler l'algorithme :
  - — $0 \mapsto 0$
  - — $1 \mapsto 1$ car évite $\{0\}$
  - — $2 \mapsto 0$ car évite $\emptyset$
  - — $3 \mapsto 2$ car évite $\{0, 1\}$

  On obtient donc $(0, 1, 0, 2)$ en 3 tables alors que $(0, 1, 1, 0)$ en deux tables est optimal.
- — Un contre-exemple classique :
  - — 0 est ennemi avec 1, 2 et 3
  - — 1 est ennemi avec 0, 2 et 4
  - — 2 est ennemi avec 0, 1 et 5
  - — 3 est ennemi avec 0, 4 et 5
  - — 4 est ennemi avec 1, 3 et 5
  - — 5 est ennemi avec 2, 3 et 4

  On va dérouler l'algorithme :
  - — $0 \mapsto 0$
  - — $1 \mapsto 1$ car évite $\{0\}$
  - — $2 \mapsto 2$ car évite $\{0, 1\}$
  - — $3 \mapsto 1$ car évite $\{0\}$
  - — $4 \mapsto 0$ car évite $\{1\}$
  - — $5 \mapsto 3$ car évite $\{0, 1, 2\}$

  Or l'affectation $(0, 1, 2, 1, 2, 0)$ est valide et utilise une table de moins.
  Le contre-exemple qu'on vient de voir en reste un en renumérotant les convives.

