

# TP3: Corrigé

MP2I Lycée Pierre de Fermat  
guillaume.rousseau@ac-toulouse.fr

## Exercice 1

- Q1.** A l'exécution, le programme affiche deux adresses mémoires (qui ne sont pas les mêmes à chaque exécution). Ces deux adresses, affichées en hexadécimal, diffèrent systématiquement de deux octets, par exemple:

L'adresse de `x` est `0x7ffc13963228`,  
celle de `y` est `0x7ffc1396322c`

C'est logique étant donné que le type `int` prend 32 bits, soit 4 octets: en mémoire, `x` prend les octets `0x7ffc13963228` à `0x7ffc1396322b`, et `y` est stockée juste après, dans les octets `0x7ffc1396322c` à `0x7ffc1396322f`.

- Q2.** En remplaçant le type `int` par `long int`, on voit que les variables prennent maintenant 8 octets (car les adresses diffèrent de 8), ce qui correspond bien au type `long int` (64 bits, soit 8 octets).

## Exercice 2

- Q1.** Oui.

- Q2.** Même en modifiant `x`, la variable `y` n'est pas affectée. C'est logique: les deux variables sont indépendantes, elles correspondent à des zones mémoires différentes, donc en modifier une n'a aucun effet sur l'autre. L'instruction `int y = *px;` ne fait que recopier la valeur de `x` dans la case mémoire `y`.

## Exercice 3

- Q1.** Le programme affiche "Segmentation fault", ou "Erreur de segmentation": ce message indique que l'on n'a pas le droit de lire à l'adresse `NULL`.

- Q2.** En demandant au programme de lire l'adresse 25:

```
1 #include <stdio.h>
2
3 int main(){
4     int* p = (int*) 25;
5     printf("L'entier stocké à l'adresse numéro 25 est %d\n", *p);
6     return 0;
7 }
```

On obtient la même erreur: interdit de lire à l'adresse numéro 25 ! De manière générale, on ne peut pas **choisir** les adresses où l'on stocke les données (en tout cas pas à notre niveau). Lorsque l'on crée une variable `int x` ou `float y`, le programme leur réserve le bon nombre d'octets dans un emplacement arbitraire que l'on ne peut pas (a priori) prédire.

## Exercice 4

Q1. Oui.

Q2. Une méthode simple d'échange en 3 temps est assez classique: on utilise une variable pour stocker temporairement la valeur pointée par `pa` afin de pouvoir l'écraser:

```
1 /* Échange le contenu des cases mémoires pointées par pa et pb */
2 void échange(float* pa, float* pb){
3     float temporaire = *pa;
4     *pa = *pb;
5     *pb = temporaire;
6 }
```

Q3. Cette fonction doit à la fois renvoyer le nombre de solutions ET utiliser les pointeurs donnés en entrée pour "renvoyer" également les solutions:

```
1 /* Résout l'équation quadratique aX^2 + bX + c = 0
2    et renvoie le nombre de solutions réelles (0, 1 ou 2).
3    Stocke également la ou les racines réelles dans les zones
4    pointées par x1 et x2. */
5 int quad_solve(float a, float b, float c, float* x1, float* x2){
6     float delta = b*b - 4*a*c;
7     if (delta < 0){
8         return 0;
9     } else if (delta == 0){
10        *x1 = -b / (2*a);
11        return 1;
12    } else { // delta négatif
13        *x1 = (-b + sqrt(delta))/(2*a);
14        *x2 = (-b - sqrt(delta))/(2*a);
15        return 2;
16    }
17 }
```

En C, on ne peut pas renvoyer plusieurs valeurs à la fois, contrairement à d'autres langages comme Python ou OCaml, où l'on pourrait renvoyer des couples, triplets, etc... En revanche, les fonctions peuvent prendre en entrée des pointeurs qui leurs permettent de modifier la mémoire, et donc de simuler le renvoi de plusieurs valeurs, comme c'est le cas dans la fonction `quad_solve`.

Voici un exemple de main utilisant les trois fonctions de cet exercice. On remarque que pour la fonction `quad_solve`, on doit utiliser la valeur renvoyée (0, 1 ou 2) pour savoir si l'on a le droit d'utiliser les variables `x1, x2`. Par exemple, si la fonction renvoie `0`, alors `x1, x2` ne contiennent pas d'informations intéressantes.

```

1  int main(){
2      // test incrémenter
3      int x = 3;
4      printf("Avant: %d\n", x);
5      incrementer(&x);
6      printf("Après: %d\n", x);
7
8      // test échange
9      float a = 3.3, b = 9.47;
10     printf("Avant échange: a = %f, b = %f\n", a, b);
11     echange(&a, &b);
12     printf("Après échange: a = %f, b = %f\n", a, b);
13
14
15     // test quad_solve
16     float x1, x2;
17
18     // X^2 + 2X + 1: une solution
19     int n1 = quad_solve(1, 2, 1, &x1, &x2);
20     assert(n1 == 1);
21     printf("Solution de X^2 + 2X + 1 = 0: %f\n", x1);
22
23     // X^2 + 2X - 3 = (X - 1)(X+3)
24     int n2 = quad_solve(1, 2, -3, &x1, &x2);
25     assert(n2 == 2);
26     printf("Solutions de X^2 + 2X - 3 = 0: %f et %f\n", x1, x2);
27
28     // X^2 - 5 = 0: pas de solution réelle
29     int n3 = quad_solve(1, 0, 5, &x1, &x2);
30     assert(n3 == 0);
31
32     return 0;
33 }
```

## Exercice 5

```

1  /* Lit un entier dans le terminal, et renvoie true en cas de succès,
2     false en cas d'échec.*/
3  bool read_int_and_flush(int* p){
4      int nb_lus = scanf("%d", p);
5      bool reussite = (nb_lus == 1);
6      return reussite;
7  }
8
9
10 /* Lit des entiers en boucle jusqu'à rencontrer une erreur de lecture. */
11 int main(){
12     int x = 1;
```

## Exercice 6

**Q1.** On voit dans le code que la fonction `somme` prend en entrée un **pointeur**. On peut donc manipuler un tableau comme si c'était un pointeur: lorsque l'on a un tableau `int t[6]`, écrire `t` tout seul donne l'adresse du tableau (c'est à dire l'adresse de sa première case).

**Q2.** On itère sur toutes les cases en suivant le même schéma que la fonction `somme`:

```
1 /* Affiche les n premiers éléments de tab sur une ligne */
2 void afficher(int* tab, int n){
3     for (int i = 0; i < n; ++i){
4         printf("%d ", tab[i]);
5     }
6     printf("\n");
7 }
```

Notons que l'ordinateur n'a aucun moyen de vérifier que la taille que l'on donne en entrée correspond à la taille réelle réservée pour le tableau (d'ailleurs, la fonction n'a aucun moyen de savoir si l'on a donné un tableau ou un pointeur simple). On peut donc mentir à la fonction et lui donner une taille trop grande. Faites l'expérience: utilisez la fonction `afficher` avec un tableau de taille 10, mais en donnant  $n = 10000$ : vous verrez la fonction continuer à lire en mémoire au delà des cases du tableau et interpréter les octets lus 4 par 4 comme des entiers ! Si cette lecture dépasse la mémoire allouée pour le programme, le programme lève une erreur de segmentation.

**Q3.** Une fonction prenant en entrée un tableau (c'est à dire un pointeur vers le début du tableau) peut le modifier. Il reste juste à voir que pour générer un nombre entre -10 et 10, on peut faire `rand()%21 - 10`: on génère d'abord un nombre entre 0 et 20, puis on le ramène dans l'intervalle  $[-10, 10]$ :

```
1 /* Remplit les n premières cases de t avec n valeurs aléatoires
2    entre -10 et 10 */
3 void tab_random(int* t, int n){
4     for (int i = 0; i < n; ++i){
5         t[i] = rand()%21 - 10;
6     }
7 }
```

On n'oubliera pas d'initialiser le générateur d'aléatoire dans le main avec `srand(time(NULL));` !

**Q4.** On peut être tenté de réutiliser la fonction `somme` pour cette question, pour remplir directement chaque case  $U[i]$  avec `somme(T, i)`. Cependant ce n'est pas très efficace: pour remplir  $U[99]$ , on devra calculer  $T[0] + T[1] + \dots + T[98]$ , et au tour d'après, pour remplir  $U[100]$ , on devra calculer  $T[0] + T[1] + \dots + T[98] + T[99]$  !!

En fait, on remarque que  $U[i+1] = U[i] + T[i]$ , et que  $U[0] = 0$ : on peut remplir  $U$  en propageant de proche en proche sans refaire tous les calculs à chaque étape:

```
1 /* Remplit le tableau U de telle sorte que U[i] est la somme
2    des i premières cases de T, pour i de 0 à n inclus */
3 void sommes_partielles(int* T, int* U, int n){
4     U[0] = 0;
```

```

5   for (int i = 0; i < n; ++i){
6       U[i+1] = U[i] + T[i];
7   }
8   }

```

## Exercice 7

On peut commencer par réfléchir à la structure du programme à haut niveau. Le jeu proposé doit tourner tant que l'utilisateur n'a pas réussi à trier le tableau en entrée, et compte les coups. On peut donc imaginer que, dans le main, on aura un tableau `t`, une variable `n` pour la taille, une variable `coups` pour compter le nombre de coups. On peut aussi introduire des variables `i, j` qui serviront à lire les entrées du/de la joueur/joueuse au cours de la partie. Ainsi, la structure du main sera:

- 1 Générer  $T$  tableau aléatoire;
- 2 **tant que  $T$  n'est pas trié faire**
- 3     Récupérer le coup  $i, j$  de l'utilisateur ;
- 4     Échanger  $T[i]$  et  $T[j]$  si le coup est valide;
- 5     Incrémenter le compteur de coups;

On voit qu'on peut déjà proposer des petites fonctions utiles:

```

1  /* Renvoie true si t, tableau de taille n, est trié, false sinon */
2  bool est_trie(int* t, int n);
3
4  /* Affiche le tableau t de taille n */
5  void print_tab(int* t, int n);
6
7  /* Remplit les n premières cases de t
8   avec des entiers aléatoires entre 0 et 50 */
9  void random_tab(int* t, int n);
10
11 /* Échange les cases i et j de t un tableau supposé de taille n, s'il
12 est valide, i.e. si i et j sont adjacentes.
13 Renvoie un booléen indiquant si l'échange était valide ou non. */
14 bool echange(int* t, int i, int j, int n);

```

Avant même de les implémenter, on peut en déduire le main:

```

1  int main(){
2      srand(time(NULL));
3      int n;
4      int t[N];
5      int coups = 0;
6      int i, j; // pour la saisie de l'utilisateur
7
8      printf("Bienvenue au jeu du tri. Choisissez la taille du tableau: ");
9      scanf("%d", &n);
10     assert(n <= N);
11
12     // generation du tableau de jeu
13     random_tab(t, n);
14     printf("Voici votre tableau:\n");
15     print_tab(t, n);
16
17     while (!est_trie(t, n)){

```

```

18     printf("Rentrez les indices à échanger: ");
19     scanf("%d %d", &i, &j);
20     if (echange(t, i, j, n)){
21         coups++;
22     } else {
23         printf("Coup invalide: les cases doivent être adjacentes\n");
24     }
25     print_tab(t, n);
26 }
27 printf("Bravo !! vous avez gagné en %d coup(s)\n", coups);
28
29 return 0;
30 }

```

Puis, l'implémentation de chacune des fonctions est relativement simple. Pour la fonction d'affichage, la version proposée ici affiche de manière un peu jolie avec des crochets et des virgules, mais rien n'empêchait de juste afficher les nombres séparés par des espaces:

```

1  /* Renvoie true si t est trié, false sinon.
2     n est la taille de t */
3  bool est_trie(int* t, int n){
4      assert(t != NULL);
5      for (int i = 1; i < n; i++){
6          if (t[i-1] > t[i]){
7              return false;
8          }
9      }
10     return true;
11 }
12
13 /* Affiche le tableau t de taille n */
14 void print_tab(int* t, int n){
15     assert(t != NULL);
16     printf("[");
17     if (n > 0){
18         printf("%d", t[0]);
19     }
20     for (int i = 1; i < n; i++){
21         printf(", %d", t[i]);
22     }
23     printf("]\n");
24 }
25
26 /* Remplit les n premières cases de t
27     avec des entiers aléatoires entre 0 et 50 */
28 void random_tab(int* t, int n){
29     assert(t != NULL);
30     for (int i = 0; i < n; i++){
31         t[i] = rand()%51;
32     }
33 }
34
35 /* Échange les cases i et j de t un tableau supposé de taille n, si
36     l'échange est autorisé, i.e. si i et j sont adjacentes.
37     Renvoie un booléen indiquant si l'échange était autorisé ou non. */
38 bool échange(int* t, int i, int j, int n){
39     assert(0 <= i && i < n && 0 <= j && j < n);
40     if ((i != j+1) && (j != i+1)){

```

```
41     return false;
42 }
43 int tmp = t[i];
44 t[i] = t[j];
45 t[j] = tmp;
46 return true;
47 }
```

## Exercice 8

## Exercice 9

**Q1.** Les lettres majuscules correspondent aux entiers 65 (0x41) à 90 (0x5a), elles sont stockées dans l'ordre alphabétique. Les chiffres 0-9 correspondent aux entiers 48 (0x30) à 57 (0x39).

Le retour à la ligne, que l'on note habituellement '\n', est bien un seul caractère, et pas une suite de deux caractères ! Son code est 10 (0x0a). Les parenthèses ont pour codes respectifs 0x et 0x

**Q2.** Les codes 0x61 à 0x7a sont les lettres de minuscules.

On voit que les minuscules sont stockées exactement 32 codes après les majuscules (par exemple, 'a' correspond à 0x61 et 'A' à 0x41). Donc, si une variable `char c` contient une majuscule, faire `c = c + 32` la transforme en la minuscule correspondante !

## Exercice 10

Le tableau est rempli dans le désordre, pour déterminer la chaîne de caractère qui s'y trouve il faut:

- noter les caractères dans l'ordre;
- s'arrêter au premier caractère nul trouvé.

On voit alors que le tableau contient la chaîne de caractères "SAULT", et le caractère nul de la case 5 bloque les caractères suivants.

## Exercice 11

**Q1.** Pour déterminer la longueur d'une chaîne, il suffit de la lire jusqu'à tomber sur le caractère de fin de chaîne, en incrémentant un compteur:

```
1 /* Renvoie la longueur du string s */
2 int str_len(char* s){
3     int len = 0;
4     while (s[len] != '\0'){
5         len = len + 1;
6     }
7     // en sortie: len est l'indice du premier '\0' de s
8     // donc s contient s[0], s[1], ... s[len-1], '\0'
```

```
9 // len est bien la taille cherchée
10 return len;
11 }
```

**Q2.** Pour compter le nombre d'occurrences d'un caractère, on reprend le schéma de la question précédente, en utilisant un compteur pour le nombre d'occurrences:

```
1 /* Renvoie le nombre d'occurrences de c dans s */
2 int occurrences(char* s, char c){
3     int i = 0;
4     int cpt = 0; // compteur
5     while (s[i] != '\0'){
6         if (s[i] == c){
7             cpt = cpt + 1;
8         }
9     }
10    return cpt;
11 }
```

**Q3.** Un point **essentiel** lorsque l'on manipule des chaînes de caractères: il ne faut jamais oublier le caractère nul à la fin de la chaîne. Ici, une fois que l'on a recopié les vrais caractères de src dans dst, il faut aussi remettre un caractère nul pour terminer la chaîne:

```
1 /* Recopie le string src dans dst */
2 void str_cpy(char* dst, char* src){
3     int i = 0;
4     while (src[i] != '\0'){
5         dst[i] = src[i];
6         i++;
7     }
8     // ne pas oublier le caractère nul
9     dst[i] = '\0';
10 }
```

**Q4.** Même remarque qu'au dessus. Notons que si une chaîne  $s$  a pour longueur  $l$ , alors  $s[l]$  est le caractère nul. Si l'on veut agrandir  $s$ , on peut donc écrire à partir de la case  $s[l]$ , et il faut bien remettre un nouveau caractère nul à la fin de la chaîne résultante:

```
1 /* Concatène le string src à la fin de dst */
2 void str_cat(char* dst, char* src){
3     // trouver la fin de dst
4     int fin_dst = str_len(dst);
5     // rajouter les caractères de src à la suite de dst
6     int i = 0;
7     while (src[i] != '\0'){
8         dst[fin_dst + i] = src[i];
9         i++;
10    }
11    // ne pas oublier le caractère nul
12    dst[fin_dst + i] = '\0';
13 }
```

**Q5.** Pour comparer deux chaînes, on procède comme pour comparer deux mots français: on regarde la première lettre de chaque mot, puis la deuxième, etc... jusqu'à trou-

ver la première position où les deux mots diffèrent.

Il y a *a priori* des cas particuliers à gérer si l'on atteint la fin d'un mot, mais en réalité puisque les chaînes en C ont systématiquement un caractère nul à la fin, le code est simplifié:

```

1  /* Renvoie un entier comparant s1 et s2 dans l'ordre alphabétique:
2     0 si égalité
3     un entier < 0 si s1 < s2
4     un entier > 0 si s1 > s2 */
5  int str_cmp(char* s1, char* s2){
6      int i = 0;
7      // lire jusqu'à sortir d'une des deux chaînes ou jusqu'à
8      // trouver une position où les mots diffèrent
9      while (s1[i] != '\0' && s2[i] != '\0' && s1[i] == s2[i]){
10         i++;
11     }
12     // en sortie:
13     // (A) toutes les lettres de s1 et s2 entre 0 et i-1
14     //      sont identiques
15     // (B) soit s1[i] = 0 = s2[i], soit s1[i] != s2[i]
16     // Finalement, s1[i] - s2[i] nous donne exactement
17     // ce que l'on cherche: 0 si les chaînes sont égales,
18     // un nombre positif si s1 est plus grand, un nombre
19     // négatif sinon !
20     return s1[i] - s2[i];
21 }
```

## Exercice 12

**Q1.** Le programme lit dans le terminal, mot par mot (c'est à dire en s'arrêtant à chaque espace, retour ligne ou tabulation). En théorie, chaque mot lu étant stocké dans le tableau `buf`, on ne devrait pas avoir le droit d'écrire de mots de plus de 19 lettres (puisque'il faut aussi une case pour le caractère nul).

Pourtant, si l'on essaie de rentrer un mot de 20 lettres ou plus, le programme continue à fonctionner, apparemment sans problème: il écrit dans `buf` puis est capable d'y lire plus de 20 caractères. Cela signifie que le programme **dépasse** de la mémoire allouée pour `buf` et peut écrire dans le reste de la mémoire. C'est un problème car un utilisateur peut alors **écraser** le contenu de la mémoire (par exemple les valeurs des variables), et donc altérer l'exécution du programme ! C'est un erreur que l'on appelle **buffer overflow**. Ce type de bug peut être exploité par des utilisateurs malveillants: en choisissant bien le texte rentré, il est possible de manipuler la structure de la mémoire pour y inscrire et exécuter des instructions arbitraires.

**Q2.** A l'exécution, on remarque que l'un des deux `printf` semble afficher les **deux** chaînes à la suite (soit "BONJOUR COUCOU" soit "COUCOU BONJOUR"). De plus, l'affichage des chaînes peut se finir par des caractères étranges. Ceci est dû au fait que l'on a supprimé le caractère nul à la fin de chacune des chaînes: l'ordinateur n'a donc plus aucun moyen de savoir où terminent les chaînes !

En mémoire, les variables et tableaux réservés sont généralement stockés les uns à côté des autres. Autrement, en mémoire, on risque d'avoir `str2` qui est stockée directement après `str1`:

|     |   |   |     |   |   |      |   |   |     |   |   |      |     |
|-----|---|---|-----|---|---|------|---|---|-----|---|---|------|-----|
| ... | B | O | ... | U | R | '\0' | C | O | ... | O | U | '\0' | ... |
|-----|---|---|-----|---|---|------|---|---|-----|---|---|------|-----|

Ainsi, après avoir remplacé les caractères nuls par des espaces, on a:

|     |   |   |     |   |   |     |   |   |     |   |   |     |     |
|-----|---|---|-----|---|---|-----|---|---|-----|---|---|-----|-----|
| ... | B | O | ... | U | R | ' ' | C | O | ... | O | U | ' ' | ... |
|-----|---|---|-----|---|---|-----|---|---|-----|---|---|-----|-----|

On n'a plus aucun moyen de savoir où la chaîne `str1` s'arrête. Lorsqu'on l'affiche, `scanf` lit chaque caractère un par un jusqu'à tomber sur un caractère nul: on lit donc "BONJOUR COUCOU", puis éventuellement les octets suivants de la mémoire s'ils ne sont pas nuls.

Ce programme vous permet d'ailleurs de voir dans quel ordre sont stockées les deux chaînes, et ce n'est pas forcément l'ordre dans lequel on les a déclarées !

## Exercice 13

Q1. OK

Q2.

```
1 /* Renvoie true si la case (i, j) est valide pour un grille n x n,
2    false sinon. */
3 bool valide(int n, int i, int j){
4     return (0 <= i && i < n && 0 <= j && j < n);
5 }
```

Q3.

```
1 /* Modifie (i, j) selon la direction (H, B, G, D) donnée par c. */
2 void deplacer(char c, int* i, int* j){
3     // de combien se déplacer dans les directions i/j
4     int di = 0,
5         dj = 0;
6     if (c == 'H'){
7         di = -1;
8     } else if (c == 'B'){
9         di = 1;
10    } else if (c == 'G'){
11        dj = -1;
12    } else {
13        assert (c == 'D');
14        dj = 1;
15    }
16    *i = *i + di;
17    *j = *j + dj;
18 }
```

Q4. On peut ajouter une fonction auxiliaire permettant de tester si un caractère est une direction valide:

```
1 /* Renvoie true si c est une direction valide, false sinon. */
2 bool est_direction(char c){
3     char directions[4] = {'H', 'B', 'D', 'G'};
4     for (int i = 0; i < 4; ++i){
5         if (c == directions[i]){
6             return true;
7         }
8     }
9     return false;
}
```

10 } }

La fonction `valider_chemin` consiste alors à lire et appliquer les directions du chemin, en vérifiant à chaque fois que la nouvelle case est valide. En fin de lecture, on doit se trouver en  $(n-1, n-1)$ :

```

1  /* Renvoie true si le chemin `chemin` est valide dans la grille
2     n x n stockée dans G, false sinon.
3     Un chemin valide doit:
4     - atteindre la case (n-1, n-1) en partant de (0, 0)
5     - ne pas monter ou descendre de plus d'une unité à la fois
6     - ne pas sortir de la grille
7     - être constitué de H,B,G ou D uniquement */
8  bool valider_chemin(char* chemin, int G[N_MAX][N_MAX], int n){
9      int i = 0;
10     int j = 0;
11     int k = 0; // indice de lecture de `chemin`
12     while (chemin[k] != '\0'){
13         int h = G[i][j];
14         if (!est_direction(chemin[k])){
15             return false;
16         }
17         deplacer(chemin[k], &i, &j);
18         if (!valide(n, i, j)){
19             return false;
20         }
21         // vérification du saut de hauteur
22         int new_h = G[i][j];
23         if (new_h < h-1 || new_h > h+1){
24             return false;
25         }
26         k++;
27     }

```

**Q5.** On propose de passer par les deux fonctions suivantes:

```

1  /* Renvoie true si la case (i, j) n'a aucune case voisine accessible
2     dans la grille n x n stockée dans G, false sinon. */
3  bool est_isolee(int G[N_MAX][N_MAX], int i, int j, int n);
4
5  /* Renvoie une direction aléatoire accessible depuis (i, j) dans la grille
6     de taille n x n stockée dans G */
7  char direction_aleatoire(int G[N_MAX][N_MAX], int i, int j, int n);

```

La fonction `recherche_chemin` s'écrit alors:

```

1  /* Cherche un chemin valide de 0 à (n-1, n-1) dans G.
2     Si un chemin est trouvé, renvoie true et stocke le chemin dans
3     `chemin`, sinon renvoie false. */
4  bool recherche_chemin(int G[N_MAX][N_MAX], char* chemin, int n){
5      int k = 0; // nombre d'étapes
6      int i = 0;
7      int j = 0;
8      while (k < 5*n*n && !(i == n-1 && j == n-1)){
9          if (est_isolee(G, i, j, n)){
10             // case sans voisines
11             return false;
12         }
13         // choisir une direction aléatoire:

```

```

14     chemin[k] = direction_aleatoire(G, i, j, n);
15     deplacer(chemin[k], &i, &j);
16     k++;
17 }
18 // fermer la chaîne de caractères
19 chemin[k] = '\0';
20 return (i == n-1 && j == n-1);
21 }

```

Il reste alors à implémenter les deux fonctions auxiliaires. Pour `est_isolee`, on va essayer d'avancer dans chacune des quatre directions, et vérifier si la case d'arrivée est valide, et ni trop haute ni trop basse:

```

1  /* Renvoie true si la case (i, j) n'a aucune case voisine accessible
2     dans la grille n x n stockée dans G, false sinon. */
3  bool est_isolee(int G[N_MAX][N_MAX], int i, int j, int n){
4      int h = G[i][j]; // hauteur de la case actuelle
5      char directions[4] = {'H', 'B', 'D', 'G'};
6      for (int k = 0; k < 4; ++k){
7          char c = directions[k];
8          int new_i = i;
9          int new_j = j;
10         deplacer(c, &new_i, &new_j);
11         int new_h = G[new_i][new_j];
12         // vérifier que la nouvelle case est valide ET ni trop haute
13         // ni trop basse
14         if (valide(n, new_i, new_j) &&
15             !(new_h < h-1 || new_h > h+1)){
16             return false;
17         }
18     }
19     return true;
20 }

```

Pour la fonction `direction_aleatoire`, on commence par construire un tableau de taille 4, dans lequel on stocke les directions accessibles depuis  $(i, j)$  en utilisant la même logique que dans `est_isolee`, puis on tire au sort une de ces directions:

```

1  /* Renvoie une direction aléatoire accessible depuis (i, j) dans la
2     grille de taille n x n stockée dans G */
3  char direction_aleatoire(int G[N_MAX][N_MAX], int i, int j, int n){
4      int h = G[i][j];
5      char directions[4] = {'H', 'B', 'D', 'G'};
6      // construire un tableau contenant les directions accessibles
7      // depuis (i, j)
8      char directions_accessibles[4];
9      int nb_dir = 0; // nombre de directions accessibles trouvées
10     for (int k = 0; k < 4; ++k){
11         char c = directions[k];
12         int new_i = i;
13         int new_j = j;
14         deplacer(c, &new_i, &new_j);
15         if (valide(n, new_i, new_j) &&
16             !(G[new_i][new_j] < h-1 || G[new_i][new_j] > h+1)){
17             directions_accessibles[nb_dir] = c;
18             nb_dir++;
19         }
20     }
21     // choix d'une direction aléatoire;

```

```
22 |     int k = rand()%nb_dir;  
23 |     return directions_accessibles[k];  
24 | }
```

Notons que les deux fonctions sont un peu redondantes, on aurait pu faire les deux opérations en même temps, ici on les fait l'une après l'autre pour rendre le code plus simple à lire.

Le `main` dans l'archive du corrigé cherche un chemin dans  $G_1$ , il prend entre 1 et 7 essais et trouve des chemins assez longs.