CS-119(a) – ICC-C Série 12

2024-05-14

Rappel Pour faire lire des valeurs d'un fichier texte à votre programme il suffit d'utiliser la redirection de l'entrée :

```
./exo < fichier.txt
```

Exo1 Euclide

Implémentez une fonction récursive qui calcule le PGCD de deux entiers en utilisant l'algorithme d'Euclide comme vu en cours. Testez-la sur quelques exemples.

Solution de l'exercice 1

```c
int euclide(int a, int b)
{
    if (b == 0)
    {
        return a;
    }
    return euclide(b, a % b);
}
```

Exo2 Afficher sans boucle

On lit un entier `N` suivi de `N` entiers qu'on stocke dans un tableau. Écrivez une fonction récursive qui affiche ce tableau.

Dans un premier temps écrivez la fonction

```c
void afficher_indice(int *tableau, int i, int taille);
```

qui affiche l'élément du tableau qui se trouve à l'indice `i` et ensuite s'appelle elle-même pour l'indice suivant. Quel est le cas de base ?

Ensuite écrivez la fonction

```
void afficher_ptr(int *tableau, int taille);
```

qui fait la même chose, mais n'utilise pas de paramètre "indice", seulement un pointeur vers le début du tableau...

Solution de l'exercice 2

```
void afficher_indice(int *tableau, int i, int taille)
{
    if (i < taille)
    {
        printf("%d ", tableau[i]);
        afficher_indice(tableau, i+1, taille);
    }
    else
    {
        printf("\n");
    }
}

void afficher_ptr(int *tableau, int taille)
{
    if (taille > 0)
    {
        printf("%d ", tableau[0]);
        afficher_ptr(tableau + 1, taille - 1);
    }
    else
    {
        printf("\n");
    }
}
```

Exo3 Stop chrono !

Implémentez une fonction itérative `long fib_it(int n)` qui calcule le `n`-ième nombre de Fibonacci $F(n)$. Les nombres de Fibonnaci sont définis par la relation de récurrence

$$F(n) = F(n-1) + F(n-2), \tag{1}$$

avec $F(0) = 0$ et $F(1) = 1$.

Implémentez ensuite une fonction récursive "naïve" `long fib_rec(int n)` qui calcule $F(n)$ en utilisant directement la relation de récurrence. Enfin, implémentez une fonction récursive `long fib_mem(int n)` qui utilise la technique de mémoïsation. `fib_mem` utilise une variable globale `long mem[100]` avec tous les

éléments initialisés à 0, pour y stocker les valeurs de $F(n)$ la toute première fois quand ils sont calculés, comme vu en cours.

Comparez le temps d'exécution de ces trois fonctions pour la valeur de `n=42`.

Pour mesurer le temps d'exécution d'une fonction, il faut utiliser la fonction `clock` de `<time.h>`. On définit deux variables de type `clock_t` appelées `clock_start, clock_end` qui permettent d'enregistrer le temps au début et à la fin de l'exécution de la fonction. Par exemple :

```
    clock_start = clock();
    long fib42 = fib_rec(42);
    clock_end = clock();
    printf(
        "Solution naive (%ld): %.2lf ms\n",
        fib42,
        (clock_end - clock_start) * 1000.0 / CLOCKS_PER_SEC);
```

Solution de l'exercice 3

On remarque que la fonction `fib_rec` prend quelques secondes pour calculer $F(42)$, alors que les deux autres fonctions s'exécutent quasi-instantanément.

```
long fib_it(int n)
{
    long n_minus_2 = 0, n_minus_1 = 1;
    long fib_n;
    for (int k = 2; k <= n; k++)
    {
        fib_n = n_minus_1 + n_minus_2;
        n_minus_2 = n_minus_1;
        n_minus_1 = fib_n;
    }

    return fib_n;
}

long fib_rec(int n)
{
    if (n == 0)
    {
        return 0;
    }
    if (n == 1)
    {
        return 1;
    }
```

```
    return fib_rec(n - 1) + fib_rec(n - 2);
}

#define MAX_FIB 100
long mem[MAX_FIB]; // initialisé à zéro avant utilisation
long fib_mem(int n)
{
    if (n == 0)
    {
        return 0;
    }
    if (n == 1)
    {
        return 1;
    }

    if (mem[n] == 0)
    {
        mem[n] = fib_mem(n - 1) + fib_mem(n - 2);
    }
    return mem[n];
}
```

Exo4 Palindrome

On lit une chaîne de caractères depuis l'entrée standard. Écrivez une fonction récursive qui retourne `1` si cette chaîne est un palindrome et `0` sinon.

Un palindrome se lit pareil à l'endroit et à l'envers. Voici des exemples de palindromes : `abccba`, `bob`, `esoperesteicietserepose` ("Ésope reste ici et se repose").

Pour obtenir la longueur de la chaîne de caractères, utilisez la fonction que vous avez écrite pendant la série 4, ou alors la fonction `strlen` de `<string.h>`.

Votre fonction récursive peut utiliser des indices :

```
int palindrome_indice(char* chaine, int debut, int fin);
```

ou alors des pointeurs :

```
int palindrome_ptr(char* debut, char* fin);
```

où `debut` pointe vers le début de la chaîne et `fin` vers le dernier caractère.

Solution de l'exercice 4

```

```

```c
int palindrome_indice(char *chaine, int debut, int fin)
{
    if (debut >= fin)
    {
        // les indices se sont croisés - ok!
        return 1;
    }
    if (chaine[debut] != chaine[fin])
    {
        // ce n'est pas un palindrome
        return 0;
    }

    return palindrome_indice(chaine, debut + 1, fin - 1);
}

int palindrome_ptr(char *debut, char *fin)
{
    if (debut >= fin)
    {
        // les pointeurs se sont croisés - ok!
        return 1;
    }
    if (*debut != *fin)
    {
        // ce n'est pas un palindrome

        return 0;
    }

    return palindrome_ptr(debut + 1, fin - 1);
}
```

Exo5 (*) Bix AI

Sûrement vous vous souvenez du jeu vidéo avec Bix le lapin. Non ? Voici un petit rappel. Bix se trouve sur un rocher et peut sauter sur les rochers voisins (gauche, droite, haut, bas). Tant que Bix reste sur les rochers il est en sécurité. Par contre s'il fait un faux pas et qu'il tombe dans la lave, le jeu est terminé.

On nous donne une matrice binaire de taille $M \times N$ qui représente le plan du champ. Les rochers sont représentés par des `1` et la lave par des `0`. On lit d'abord `M` et `N` et ensuite les `M` lignes contenant `N` valeurs binaires.

Bix se trouve à la position `(0, 0)`. Il doit arriver à la sortie qui se trouve à la

position `(M-1, N-1)`. On aimerait produire une chaîne de caractères (sans espace vide) représentant la suite d'instructions qui mène Bix à la sortie. Les caractères dans cette chaîne peuvent être `'h'` (pour "haut"), `'b'` (pour "bas"), `'g'` (pour "gauche"), ou `'d'` (pour "droite"). Attention, Bix ne doit pas visiter une case plus qu'une fois - quand il quitte un rocher, le rocher s'effondre dans la lave !

Si plusieurs solutions sont possibles, affichez une d'entre elles. Si aucune solution n'existe, on affichera le mot "Impossible" à la sortie standard.

Par exemple, pour l'entrée

```
5 6
1 0 0 0 0 0
1 0 0 1 0 1
1 0 1 1 0 1
1 1 1 0 1 0
0 0 1 1 1 1
```

la seule solution possible est la chaîne bbbddbddd.

Pour l'entrée

```
6 6
1 0 0 0 0 0
1 0 0 1 0 1
1 0 1 1 1 1
1 1 1 0 1 0
0 0 1 0 1 0
0 0 1 1 1 1
```

il y a deux solutions possibles : bbbddbbddd ou bbbddhddbbbd.

Enfin, pour l'entrée

```
6 6
1 0 0 0 0 0
1 0 0 1 0 1
1 0 1 1 0 1
1 1 1 0 1 0
0 0 0 0 1 0
0 0 1 1 1 1
```

il n'y a pas de solution.

Indices et suggestions

Nous voulons explorer la matrice pour découvrir le (les) chemins possibles. Lors de l'exploration, il faut se souvenir des endroits par où nous sommes déjà

passés afin d'éviter les boucles infinies. Pour ce faire, il est convenable de définir une deuxième matrice (variable globale) `int vu[][]` de la même taille `M x N` qu'on initialise à `0`. Chaque fois qu'on visite une case `(i, j)`, on mettra la case `vu[i][j]` à `1`.

Pour trouver le chemin il est conseillé d'utiliser une fonction récursive. Cette fonction prend des coordonnées `(i, j)` d'une case de la matrice et retourne `1` si c'est possible d'atteindre la destination depuis cette case, ou `0` sinon.

Les cas de base à traiter sont les suivants :

— Si la case est en dehors de la matrice, alors la fonction retourne `0`.

— Si la case contient de la lave, donc `map[i][j] == 0`, alors la fonction retourne aussi `0`.

— Si la case a déjà été visitée, donc `vu[i][j] == 1`, alors de nouveau la fonction retourne `0`.

— Enfin, si tout va bien et en plus `i == M-1` et `j == N-1` alors nous avons atteint la destination et la fonction retourne `1`.

La relation récursive est donnée par les déplacements possibles – il y a quatre possibilités pour atteindre la destination depuis la case `(i, j)` : soit on saute vers le haut et on essaye de continuer depuis `(i-1, j)` ; si cela ne marche pas, alors on saute vers le bas et on essaye de continuer depuis `(i+1, j)` ; sinon on saute vers la gauche et on essaye de continuer depuis `(i, j-1)` ; la dernière chance : on saute vers la droite et on essaye de continuer depuis `(i, j+1)`. Attention de marquer la case `(i, j)` comme visitée avant tous ces essais ! Si n'importe lequel des essais précédents réussit, alors on retourne `1`, pas besoin de tous les explorer dans ce cas. Si par contre aucun des essais ne fonctionne, alors cela veut dire que ce n'est pas possible d'atteindre la destination depuis cette case et on retourne `0`.

Il faut aussi enregistrer les instructions à utiliser afin d'atteindre la destination. Il faudrait donc les accumuler dans un tableau ou dans une liste qu'on passe en paramètre de notre fonction récursive ! Lors d'un échec il faut se souvenir d'enlever la dernière instruction qui n'avait pas fonctionné. Une signature possible de la fonction récursive pourrait être

```
    int visiter_liste(int i, int j, list_t* instructions);
```

ou alors

```
    int visiter_tableau(int i, int j,
                        char* instructions, int taille);
```

Solution de l'exercice 5

```
int M, N, map[100][100];
int vu[100][100];
```

```
int visiter(int r, int c,
            char *instructions, int taille)
{
    if (r < 0 || c < 0 || r >= M || c >= N)
    {
        // Hors du tableau
        return 0;
    }

    if (vu[r][c])
    {
        // Nous sommes déjà passés par là
        return 0;
    }

    if (map[r][c] == 0)
    {
        // On tombe dans la lave...
        return 0;
    }

    if (r == M-1 && c == N-1)
    {
        // Marquer la fin de la chaine
        instructions[taille] = 0;
        // Nous avons trouvé la destination!
        return 1;
    }

    // Marquer comme vu
    vu[r][c] = 1;

    // On essaye de sauter vers le haut
    instructions[taille] = 'h';
    if (visiter(r-1, c, instructions, taille+1))
    {
        return 1;
    }

    // Sinon on essaye de sauter vers le bas
    instructions[taille] = 'b';
    if (visiter(r+1, c, instructions, taille+1))
    {
```

```
        return 1;
    }

    // Sinon on essaye de sauter vers la droite
    instructions[taille] = 'd';
    if (visiter(r, c+1, instructions, taille+1))
    {
        return 1;
    }

    // Sinon on essaye de sauter vers la gauche
    instructions[taille] = 'g';
    if (visiter(r, c-1, instructions, taille+1))
    {
        return 1;
    }

    // Pas trouvé...
    return 0;
}
```