Boucles et tableaux

1 Boucles

1.1 Quelques sommes

Écrire des fonctions calculant les sommes suivantes :

1. $S_1(n) = \sum_{k=1}^{n}(2k+1)$
2. $S_2(n) = \sum_{k=1}^{n}\frac{1}{k^2}$
   Attention, si on écrit `n / p` *avec* `n` *et* `p` *entiers, on aura une division entière (quotient de la division euclidienne), comme en* OCaml*. Pour forcer une division flottante, il faut que l'un des opérandes soit un flottant : une manière de forcer cela est d'écrire* `1.0 * n / p`.
3. $S_3(n) = \sum_{i=1}^{n}\sum_{j=1}^{n}(i+j)$
4. $S_4(n) = \sum_{i=1}^{n}\sum_{j=i}^{n}(i+j)$
5. $S_5(n) = \sum_{i=1}^{n}\sum_{j=1}^{i-1}(i+j)$

On écrira ensuite une fonction `main` permettant d'obtenir l'affichage suivant :

```
$ ./sommes
S1(0) = 0
S2(0) = 0.000000
S3(0) = 0
S4(0) = 0
S5(0) = 0
S1(1) = 3
S2(1) = 1.000000
S3(1) = 2
S4(1) = 2
S5(1) = 0
S1(2) = 8
S2(2) = 1.250000
S3(2) = 12
S4(2) = 9
S5(2) = 3
S1(3) = 15
S2(3) = 1.361111
S3(3) = 36
S4(3) = 24
S5(3) = 12
```

1.2 Logarithme entier

Écrire une fonction prenant en entrée un entier n, et renvoyant le plus petit $k \geqslant 0$ tel que $2^k \geqslant n$: on fera une version utilisant une boucle `while` et une utilisant une boucle `for`.

2 Tableaux statiques

On peut déclarer un tableau de la manière suivante :

```c
int t[10]; // tableau de 10 entiers, non initialisé
double u[3] = {1.2, 3.4, 2.5} // tableau de 3 doubles, initialisé
int v[] = {1, 2, 7, 8} // la taille est déduite de l'initialisation
```

— Les éléments d'un tableau sont indicés à partir de 0, et on accède à un élément par `t[indice]`. Les éléments du tableau `u` défini ci-dessus sont donc `u[0]`, `u[1]` et `u[2]`.
— Si le tableau n'est pas initialisé explicitement (cas du tableau `t` ci-dessus) :

  - si c'est une variable globale, les éléments sont implicitement initialisés à zéro ;
  - si c'est une variable locale, les éléments ne sont pas initialisés, et ont au départ une valeur quelconque.
- La taille d'un tel tableau doit *absolument être connue statiquement* (c'est-à-dire à la compilation). La taille doit être une *constante littérale* :

```
int t[10]; // OK

int n = 10;
int u[n]; // NON

const int p = 5;
int v[p]; // NON PLUS
```

2.1 Tri

On part du squelette suivant avec un tableau défini comme une variable globale :

```
#include <stdio.h>
#include <stdlib.h>

const int taille = 10;
int tab[10];
```

Noter que si l'on veut modifier le code pour que le tableau soit de taille 10, il faut modifier à la fois la constante `taille` que l'on utilisera pour les boucles dans le programme et le `10` qui apparaît dans la définition de `tab` (ce n'est évidemment pas satisfaisant, et nous verrons ultérieurement comment améliorer cela)..

1. La fonction `int rand(void)`, déclarée dans `stdlib.h`, renvoie un entier positif aléatoire. Écrire une fonction `void remplire(void)` qui remplit le tableau global `tab` avec des entiers tirés aléatoirement.
2. Écrire une fonction `void affiche(void)` qui affiche le contenu du tableau `tab` (en séparant les entiers par une espace et en revenant à la ligne à la fin).
3. Écrire une fonction `int min(void)` qui renvoie le minimum du tableau `tab`.
4. Écrire une fonction `int indice_min(void)` qui renvoie l'indice de la première occurrence du minimum dans `tab`.
5. Écrire une fonction `void tri_insertion(void)` qui trie le tableau `tab` en utilisant l'algorithme du tri par insertion.
6. Écrire un programme qui effectue les tâches suivantes :
   - initialiser le tableau `tab` avec des valeurs tirées au hasard ;
   - l'afficher ;
   - calculer et afficher le minimum de ce tableau, ainsi que l'indice de sa première occurrence ;
   - trier ce tableau ;
   - afficher ce tableau trié.

3 Arguments en ligne de commande

Pour l'instant, notre fonction `main` avait systématiquement le prototype suivant :

```
int main()
```

Un autre prototype est possible pour cette fonction :

```
int main(int argc, char* argv[])
```

Le fonctionnement est essentiellement le même qu'en OCaml (disons plutôt qu'OCaml a repris le fonctionnement du C) :

- `argc` est le nombre d'arguments passés au programme, sachant que le premier argument est toujours la commande utilisée pour le lancer ;
- `argv` est un tableau de chaînes de caractères, de longueur `argc`. Le premier élément de ce tableau contient le nom de la commande, les autres contiennent les différents arguments.

En C, une chaîne de caractères est simplement un tableau de caractères, avec un dernier caractère nul. Cependant, pour gérer les arguments en ligne de commande, il n'est pas forcément nécessaire de les voir comme telles : on dispose de fonctions pour convertir une chaîne de caractères en entier ou en flottant.

— La fonction `int atoi(char* s)` convertit la chaîne de caractères `s` en entier. Elle saute les éventuels caractères d'espacement trouvés en début de chaîne (et les erreurs ne sont essentiellement pas gérées, donc mieux vaut être sûr que la chaîne correspond bien à un entier !).

— La fonction `double atof(char* s)` fait le même travail, mais lit un nombre flottant, qu'elle renvoie sous la forme d'un `double`.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[]) {
    int n;
    float x;
    if (argc != 3) {
        printf("Il faut donner un entier et un flottant\n");
        return 1;
    }
    n = atoi(argv[1]);
    x = atof(argv[2]);
    for (int i = 0; i < n; i = i + 1) {
        printf("%f\n", x);
    }
    return 0;
}
```

Le programme ci-dessus attend deux arguments en ligne de commande :

— un entier n ;

— un flottant x.

Elle affiche ensuite n fois le flottant x. Si le nombre d'arguments n'est pas exactement 2, elle affiche un message d'erreur et termine avec une valeur non nulle pour signaler le problème.

3.1 Exponentiation rapide

Écrire un programme qui accepte deux arguments en ligne de commande, un flottant `x` et un entier `n` et renvoie la valeur de x^n en effectuant ce calcul par une exponentiation rapide. On écrira la version itérative pour se rafraîchir la mémoire.