TD 1 – Structures de contrôle

Conditionnelles

1) Valeur absolue

Écrire une fonction prenant en argument un réel et renvoyant sa valeur absolue.

2) Recherche d'extremum

a) Écrire une fonction `max2` renvoyant le maximum de deux entiers passés en paramètres.

b) Écrire une fonction `max3` renvoyant le maximum de trois entiers passés en paramètres.
Expliquer le déroulement de l'appel `max3(-5,6,8)` puis de l'appel `max3(7,-3,-9)`.

c) À l'aide d'un arbre, expliquer comment on peut concevoir une fonction `max4` renvoyant le maximum de quatre nombres. Combien de comparaisons sont-elles nécessaires ?

d) Écrire une fonction renvoyant le maximum *et* le minimum de trois entiers.
Il serait bon de faire un petit dessin expliquant ce qu'on sait après chaque comparaison.
Recommencer avec quatre entiers. Le petit dessin est encore plus nécessaire !

3) Valeurs distinctes

a) Écrire une fonction `compare` prenant en argument deux entiers `x` et `y`, et renvoyant `1` si `x>y`, `0` si `x=y` et `-1` si `x<y`.

b) Écrire une fonction `distincts` prenant en argument trois entiers, et renvoyant le nombre d'arguments distincts. Par exemple, `distincts(1,2,3)` doit renvoyer `3`, et `distincts(1,2,1)` doit renvoyer `2`.

c) Reprendre la question précédente avec quatre arguments au lieu de trois.

4) Tri d'un petit nombre de valeurs

5) Écrire une fonction `tri3` prenant en argument trois entiers, et écrivant à l'écran ces trois entiers dans l'ordre croissant.

6) Reprendre la question précédente avec quatre entiers au lieu de trois.

7) Équation du second degré

On cherche à écrire un programme résolvant les équations du second degré, de la forme $ax^2 + bx + c = 0$.

On rappelle que cette équation se résout en calculant son *discriminant* $\beta = b^2 - 4ac$.

- Si $\Delta > 0$, alors l'équation a deux solutions, $x_1 = \dfrac{-b - \sqrt{\Delta}}{2a}$ et $x_2 = \dfrac{-b + \sqrt{\Delta}}{2a}$,
- si $\Delta = 0$, l'équation n'a qu'une seule solution, $x_0 = -\dfrac{b}{2a}$,
- enfin, si $\Delta < 0$, l'équation n'a pas de solution.

a) Écrire un premier programme ne tenant compte que du cas particulier où $a \neq 0$.

b) Corriger votre programme de manière à ce qu'il traite aussi le cas particulier $a = 0$.

Boucles

1) Compte à rebours

Écrire un programme demandant un entier positif n, et écrivant à l'écran un compte à rebours de n à 0. Par exemple, si l'on tape "3", le programme doit écrire :

Plus que 3 secondes...
Plus que 2 secondes...
Plus qu'1 seconde...
Booooooooommmmm !!!!!

- à l'aide d'une boucle indexée,
- à l'aide d'une boucle indexée *dont l'indice croit de* 0 *à* n,
- à l'aide d'une boucle `tant que`,
- à l'aide d'une boucle `répéter`.

2) Factorielle

Écrire trois programmes demandant à l'utilisateur un entier n, et calculant la factorielle $n!$ de n, définie par :

$$0! = 1, \quad \text{et} \quad n! = 1 \times 2 \times \cdots \times n \text{ pour } n \geqslant 1$$

- Le premier programme devra utiliser une boucle indexée,
- le deuxième une boucle `while`,
- le troisième une boucle `repeat`.

3) Somme des entiers impairs

a) Écrire un programme demandant à l'utilisateur un entier n, et calculant la somme des n premiers entiers impairs.

b) Lorsqu'on exécute ce programme pour différentes valeurs de n, on constate que pour certaines valeurs, le résultat est négatif. Quelle est la raison de cette bizarrerie ? Comment modifier votre programme pour calculer la somme des 1000 premiers entiers impairs ?

c) Modifier votre programme pour calculer la somme des n premiers carrés : $1^2 + 2^2 + \cdots + n^2$.

4) Choix d'une boucle

Dans les exercices précédent, une boucle semble-t-elle plus adaptée que les autres ?

5) Un milliard, c'est long ?

Implémentez une boucle de manière à faire compter votre ordinateur jusqu'à 10^6, puis 10^9 (n'affichez surtout pas chaque nombre, cela augmenterait singulièrement le temps d'exécution !). Attention à la représentation des entiers en machine.

Le temps nécessaire pour ces calculs vous semble-t-il conforme à ce que vous attendiez ?

6) Étude de suites

a) Écrire une fonction calculant la somme $\displaystyle\sum_{i=1}^{n} \frac{1}{1+i^2}$ en fonction de n.

b) On définit u_n par la formule : $u_n = \sqrt{1 + \sqrt{1 + \cdots + \sqrt{1+\sqrt{1}}}}$, la formule contenant n radicaux.

Trouver une relation entre u_{n-1} et u_n, et en déduire une fonction calculant u_n en fonction de n.

7) Suites divergentes

a) Écrire une fonction prenant en argument un réel A, et calculant le premier entier N à partir duquel $\displaystyle\sum_{i=1}^{N} \frac{1}{i}$ devient strictement supérieur à A.

b) Écrire une fonction prenant en argument un entier k, et calculant le premier entier N à partir duquel $n!$ devient strictement supérieur à n^k.

c) Écrire une fonction prenant en argument un taux t, et calculant le temps au bout duquel un placement à intérêt composé de taux t aura doublé.

8) Racine carrée entière

Soit n un entier naturel. On cherche la *racine carrée entière* de n, c'est à dire le plus grand entier p tel que $p^2 \leqslant n$. On notera $\mathrm{r}(n)$ cet entier.

Par exemple, la racine carrée entière de 16 est 4, puisque $4^2 \leqslant 16$, et $5^2 > 16$. On retrouve ainsi dans le cas des carrés parfaits la racine carrée classique.

De même, $\mathrm{r}(233) = 15$, car $15^2 = 225 \leqslant 233 < 256 = 16^2$.

81) À l'aide de la seule définition, écrire une fonction `r` associant à un entier naturel n sa racine carrée entière.

Combien de multiplications et de comparaisons sont-elles nécessaires pour calculer la racine carrée d'un entier n donné ?

82) Proposer un algorithme basé sur une recherche dichotomique. Cet algorithme est-il plus ou moins efficace que l'algorithme "naïf" ?

83) Héron, mathématicien grec du premier siècle après JC, a proposé l'algorithme suivant : partant de $x_0 = n$, on calcule les termes successifs d'une suite (x_n) par la relation de récurrence :

$$x_{n+1} = \frac{1}{2}\left(x_n + \frac{n}{x_n}\right)$$

Bien entendu, tous les calculs se font dans l'ensemble des entiers.

Tel quel, cette relation ne permet pas de définir un algorithme, puisqu'il manque un critère d'arrêt. On arrêtera les calculs dès que deux termes successifs de la suite seront identiques.

i. Expliquer pourquoi il est inutile de poursuivre les calculs après avoir obtenu deux valeurs identiques.

ii. Implémenter cet algorithme sous la forme d'une fonction.

iii. Tester sur quelques valeurs de n cet algorithme, et comparer son efficacité à celle de l'algorithme "naïf" et de l'algorithme dichotomique.

9) Logarithme entier

On souhaite programmer une fonction calculant $L_2(n)$, logarithme entier de base 2 de l'entier n. C'est le plus grand entier k tel que $2^k \leqslant n$. Ainsi, par définition :

$$2^{L_2(n)} \leqslant n < 2^{L_2(n)+1}$$

91) Écrire une version de cette fonction utilisant une boucle "tant que" implémentant mot pour mot la définition.

92) Écrire une version de cette fonction utilisant cette définition récursive de $L_2(n)$:

$$L_2(1) = 0, \quad \text{et} \quad L_2(n) = L_2(n/2) + 1 \text{ si } n > 1$$

où bien entendu $N/2$ est le quotient *entier* de la division de n par 2.

93) On rappelle qu'en mathématiques, on définit $\log_2(n)$ sous la forme :

$$\log_2(n) = \frac{\ln n}{\ln 2}$$

Utiliser cette définition, et le fait que $L_2(n)$ est la partie entière (*floor*, en anglais) de $\log_2(n)$, pour écrire une troisième version de cette fonction. Comparer les mérites respectifs de ces trois implémentations.

10) Écriture de tables de Pythagore

Le but de cet exercice et de faire un programme imprimant une table d'addition ou de multiplication

a) Écrire une fonction traçant à l'écran une table d'addition. On prendra bien garde à aligner correctement les résultats, et à les séparer avec des caractères `"|"` et `"-"`.

Indication : pour l'écartement des colonnes, utiliser les options de formatage de write et writeln, ou bien utiliser une fonction insérant un nombre correct d'espace avant d'afficher le résultat (il est bien entendu conseillé d'explorer les deux solutions).

b) Recommencer avec une table de multiplication.

11) Conjecture de Syracuse

On considère la suite (u_n) définie par la donnée de u_0 entier strictement positif, et de la relation de récurrence :

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{sinon} \end{cases}$$

Par exemple, pour $u_0 = 7$, les termes successifs de la suite sont :

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	...
u_n	7	22	11	34	17	52	26	13	40	20	10	5	16	8	4	2	1	...

et à partir de cet instant, la suite ne prend plus que les trois valeurs 4, 2 et 1. On dit qu'elle est *ultimement périodique*.

La conjecture de Syracuse (ou conjecture de Collatz) affirme que quel que soit l'entier u_0, il existe un rang n tel que $u_n = 1$ (autrement dit, toute suite construite sur ce modèle est ultimement périodique sur le cycle $4 - 2 - 1$).

a) Écrire une fonction prenant en argument un entier a et affichant les termes successifs de la suite démarrant à $u_0 = a$ jusqu'à atteindre 1.

   L'utiliser pour vérifier la conjecture de Syracuse pour les valeurs de u_0 comprises entre 0 et N, N étant une borne donnant un temps de vérification raisonnable.

b) Le *temps de vol* de la suite est le plus petit entier n tel que $u_n = 1$. Pour notre exemple initial, il est de 16. Écrire une fonction prenant en argument le terme initial de la suite et renvoyant le temps de vol.

   Dresser une table de valeurs de ce temps de vol pour les valeurs de u_0 comprises entre 1 et N.

c) On s'intéresse aux *records de temps de vol*. Un entier a établit un record si la suite de terme initial a a un temps de vol plus long que toute suite de départ $b < a$.

   Écrire un programme recherchant les records de temps de vol jusqu'à une borne fixée à l'avance.

d) Le *temps de vol en altitude* est le plus petit entier n tel que $u_{n+1} < u_0$. Ainsi, le temps de vol en altitude de notre exemple est 10 puisque u_{11} est la première valeur de u_n inférieure à u_0.

   Écrire une fonction prenant en argument la valeur initiale de la suite, et renvoyant son temps de vol en altitude.

   Y a-t-il une corrélation entre le temps de vol et le temps de vol en altitude ?

e) *L'altitude maximale* est la plus grande valeur que prend la suite. Ainsi, l'altitude maximale de notre exemple est 52. Écrire une fonction calculant l'altitude maximale d'une suite connaissant sa valeur initiale.

   Écrire ensuite un programme calculant l'altitude maximale de toutes les suites telles que u_0 est compris entre 1 et N. Essayer d'optimiser ce programme de façon à ne pas refaire des calculs déjà effectués. Par exemple, si une suite de valeur initiale supérieure à 7 passe par la valeur 7, on peut facilement calculer son altitude maximale sans calculer d'autres termes de cette suite.