

TP 1: Bases du langage C

Lionel Lacassagne

Introduction

Le but de ce TP est, à travers des exercices de complexités croissantes de mettre en oeuvre les concepts de base du C. Pour ce premier TP, un squelette de programme est disponible à l'adresse www.ief.u-psud.fr/~lacas/Teaching.

1 Sommes

On souhaite calculer la somme des n premiers entiers. Seuls des entiers de type `int` seront utilisés. La fonction de test utilisée pour valider les fonctions de calcul est `test_boucle()`.

1. Coder le calcul de cette somme avec une boucle `for` dans la fonction `somme_for()`.
2. Coder le calcul de cette somme avec une boucle `do` dans la fonction `somme_do()`.
3. Coder le calcul de cette somme avec une boucle `while` dans la fonction `somme_while()`.
4. Coder le calcul direct de cette somme (Eq. 1) dans la fonction `somme_math()` et comparer les précédents résultat pour que le programme affiche "calculs corrects" si tous les résultats sont égaux et corrects et "calculs incorrects" sinon.

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad (1)$$

2 Nombres premiers

Dans cet exercice on souhaite tester (plus ou moins rapidement) si un nombre est premier et trouver le prochain nombre premier. La fonction de test utilisée pour valider les fonctions de calcul est `test_prime()`.

1. Pour savoir si un nombre n est premier ou composite, on peut tester s'il existe un nombre entre 2 et n qui divise exactement (le reste de la division est nul) n . Coder la fonction `is_prime()` qui réalise cet ensemble de tests et qui renvoie 1 si n est premier et 0 autrement. Valider la fonction avec $n = 21$, $n = 25$ et $n = 101$.
2. Afin d'aller plus vite, il n'est pas nécessaire de tester les nombres pairs à part 2. De plus on peut s'arrêter à \sqrt{n} . Coder la fonction `is_prime_fast()` qui est l'adaptation de la fonction précédente. Valider la fonction avec les mêmes valeurs.

- Coder la fonction `next_prime()` qui renvoie le prochain nombre premier strictement supérieur à n (ie que n soit premier ou non) et utilisant la fonction rapide `is_prime_fast()`. Afin d'optimiser l'ensemble on ne testera que les nombres impairs, donc prévoir un cas particulier si n est pair.
- Coder la fonction `display_primes()` qui affiche tous les nombres premiers entre 2 et n .
- Modifier la fonction `display_primes()` afin de réaliser un affichage formaté de 10 nombres par ligne avec un alignement par colonne. Enfin, afficher le nombre de nombres premiers calculés. Il y a ainsi 25 nombres premiers compris entre 1 et 100 (Eq. 2)

$$\begin{array}{cccccccccc}
 2 & 3 & 5 & 7 & 11 & 13 & 17 & 19 & 23 & 29 \\
 31 & 37 & 41 & 43 & 47 & 53 & 59 & 61 & 67 & 71 \\
 73 & 79 & 83 & 89 & 97 & & & & &
 \end{array} \tag{2}$$

3 Fonctions récursives

Dans cet exercice on souhaite mettre en oeuvre des fonctions récursives ainsi que leur équivalent non-récursif. On souhaite aussi mettre en évidence qu'il faut parfois ré-écrire des fonctions pour les adapter à différents usages.

- Coder la version récursive (`fact_rec()`) et non récursive (`fact()`) qui calcule la factorielle d'un nombre entier. Dans les deux cas, traiter les cas particuliers: $0! = 1$ et $1! = 1$. La fonction de test à utiliser est `test_factorielle()`.
- Coder la fonction `cnp` qui calcule le nombre de combinaisons possibles C_n^p (Eq. 3).

$$C_n^p = \frac{n!}{p!(n-p)!} \tag{3}$$

- Coder la fonction `display_pascal()` qui affiche une ligne du triangle de pascal et qui sera appelée depuis la fonction `test_pascal()`. Cette dernière fonction réalisera une boucle pour afficher toutes les lignes depuis 1 jusqu'à n . Un exemple d'affichage formaté est donné ci-après (Eq. 4), pour l'affichage des lignes de 1 à 5.
- Que se passe-t-il pour $n \geq 13$?
- Cette fonction est simplifiable en divisant le numérateur et le dénominateur par $p!$. Il reste alors au numérateur le produit des nombres de $p + 1$ à n . Coder la fonction `cnp_fast()` qui sera appelée par `display_pascal_fast()`. Que peut on observer ?
- Proposer une solution pour que le calcul des lignes ≥ 13 soit correct. Coder les fonctions `cnp_smart()` qui sera appelée par `display_pascal_smart()`

$$\begin{array}{cccccc}
 1 & 1 & & & & \\
 1 & 2 & 1 & & & \\
 1 & 3 & 3 & 1 & & \\
 1 & 4 & 6 & 4 & 1 & \\
 1 & 5 & 10 & 10 & 5 & 1
 \end{array} \tag{4}$$