1 Partitions d'entiers

Exercice 1 (Définition et représentation) : On appelle **partition** d'un entier n une suite $u = (u_0, u_1, \dots, u_k, 0)$ finie et décroissante d'entiers se terminant par 0, et telle que $u_0 + u_1 + \cdots + u_k = n$. La valeur $k+1$ représente la **longueur** de cette partition (le zéro final n'est pas compté).

Exemple. Quelques partitions de l'entier 12 :

- $u = (4, 4, 3, 1, 0)$ est une partition de longueur 4, puisque $12 = 4 + 4 + 3 + 1$;
- $u = (5, 4, 3, 0)$ est une partition de longueur 3, puisque $12 = 5 + 4 + 3$;
- $u = (12, 0)$ est une partition de longueur 1.

En langage C, nous allons représenter une partition par une structure contenant deux champs :

- l'entier n concerné ;
- la suite u correspondant à sa partition, représentée sous forme d'un tableau d'entiers se terminant par un zéro.

1. Définir une constante `L_MAX` correspondant à la longueur maximale d'une partition (on utilisera un `#define`).
2. Définir un type structuré `t_partition` contenant les champs listés ci-dessus, en utilisant `L_MAX`.

Exemple. Une variable de type `t_partition` pourra être initialisée à la déclaration. Par exemple, pour les exemples précédents :

Exemple

```
t_partition p1 = {12, {4,4,3,1,0}};
t_partition p2 = {12, {5,4,3,0}};
t_partition p3 = {12, {12,0}};
```

3. Écrire une fonction `void affiche_partition(t_partition p)` qui affiche la partition `p` passée en paramètre. Vous devez obtenir un affichage de la forme suivante :

Output

```
12 = 4 + 4 + 3 + 1
12 = 5 + 4 + 3
12 = 12
```

4. Écrire une fonction `void copie_partition(t_partition p, t_partition *copie)` qui copie la partition `p` dans `copie`. Attention à bien gérer la copie du tableau.

Exercice 2 (Comparaison) : On peut définir un ordre appelé **ordre lexicographique** sur les partitions d'un entier n. Pour comparer les deux partitions $u = (u_0, u_1, \dots, u_p, 0)$ et $v = (v_0, v_1, \dots, v_q, 0)$, on applique la règle suivante :

- s'il existe un entier j tel que : $u_0 = v_0$, $u_1 = v_1$, …, $u_{j-1} = v_{j-1}$, et $u_j \neq v_j$, alors :
  - si $u_j > v_j$, on a $u > v$;
  - si $u_j < v_j$, on a $u < v$;
- sinon, c'est-à-dire si $p = q$ et $\forall i \in [\![0, p]\!], u_i = v_i$, alors on a $u = v$.

Exemple. Comparaison de quelques partitions de l'entier $n = 12$:

$$(12, 0) > (5, 4, 3, 0) > (5, 4, 2, 1, 1, 0) > (4, 4, 4, 0) > (4, 4, 3, 1, 0) > (4, 4, 2, 1, 1, 0)$$

1. Écrire une fonction `int compare_partitions(t_partition p1, t_partition p2)` qui utilise la règle expliquée ci-dessus pour comparer les partitions `p1` et `p2`. La fonction doit renvoyer :
   - Une valeur > 0 si $p_1 > p_2$;
   - 0 si $p_1 = p_2$;
   - Une valeur < 0 si $p_1 < p_2$.

   Remarque : on supposera que les deux partitions passées en paramètres concernent toutes les deux le même entier n.

Exemple.

- Pour $(12, 0)$ et $(4, 4, 4, 0)$, la fonction renvoie une valeur strictement positive.
- Pour $(5, 4, 2, 1, 0)$ et $(5, 4, 2, 1, 0)$, la fonction renvoie zéro.
- Pour $(4, 4, 4, 0)$ et $(5, 4, 2, 1, 0)$, la fonction renvoie une valeur strictement négative.

Exercice 3 (Partitions triviales) : On peut distinguer deux partitions **triviales** pour tout entier $n > 1$:

- la partition triviale **courte**, que l'on notera $\bar{u}$, est l'unique partition de longueur 1 de cet entier, i.e. $\bar{u} = (n, 0)$;
- la partition triviale **longue**, que l'on notera $\underline{u}$, est l'unique partition de longueur n de cet entier, i.e. $\underline{u} = (1, \ldots, 1, 0)$.

Exemple. Pour $n = 12$, on a $\bar{u} = (12, 0)$ et $\underline{u} = (1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0)$.
Remarque : pour toute partition u d'un entier n, on a $\underline{u} \leq u \leq \bar{u}$.

1. Écrire une fonction `void calcule_partition_courte(int n, t_partition *p)` qui calcule la partition triviale **courte** de n et place le résultat dans le paramètre `p` passé par adresse.
2. Écrire une fonction `void calcule_partition_longue(int n, t_partition *p)` qui calcule la partition triviale **longue** de n et place le résultat dans le paramètre `p` passé par adresse.

Exercice 4 (Génération) : Si u est la partition d'un entier n, on peut calculer la partition v qui précède u dans l'ordre lexicographique à l'aide de l'algorithme suivant :

- Si $u_0 > 1$:
  - on pose j le plus grand entier tel que $u_j > 1$;
  - on pose :
    - $v_0 = u_0$, …, $v_{j-1} = u_{j-1}$
    - $v_j = u_j - 1$
  - on calcule l'entier a égal au nombre de 1 dans la partition u, i.e. l'entier a tel que $u_{j+1} = u_{j+2} = \cdots = u_{j+a} = 1$;
  - on calcule le quotient q et le reste r de la division euclidienne de $a + 1$ par v_j, et on pose :
    - $v_{j+1} = v_{j+2} = \cdots = v_{j+q} = v_j$;
    - $v_{j+q+1} = r$ (si $r \neq 0$) ;
  - on ajoute un 0 comme dernier terme dans v.

Exemple. On considère deux partitions de $n = 12$:

- $u = (5, 4, \mathbf{3}, 0)$
  - alors : $j = 2$, $v_j = 2$, $a = 0$, $q = 0$ et $r = 1$;
  - donc : $v = (5, 4, \mathbf{2}, 1, 0)$.
- $u = (4, 3, \mathbf{2}, 1, 1, 1, 0)$
  - alors : $j = 2$, $v_j = 1$, $a = 3$, $q = 4$ et $r = 0$;
  - donc : $v = (4, 3, \mathbf{1}, 1, 1, 1, 1, 0)$.

1. En appliquant *manuellement* cet algorithme, calculer les partitions de $n = 12$ qui précèdent les partitions :
   - $(12, 0)$;
   - $(5, 5, 1, 1, 0)$.
2. Écrire une fonction `void calcule_partition_precedente(t_partition p0, t_partition *p1)` qui calcule la partition `p1` qui précède `p0` dans l'ordre lexicographique, en utilisant l'algorithme précédent.
3. Écrire une fonction `void affiche_toutes_partitions(int n)` qui calcule et affiche toutes les partitions de l'entier n dans l'ordre lexicographique.

Exemple. Pour n = 9 :

Output (1/2)

```
9 = 9
9 = 8 + 1
9 = 7 + 2
9 = 7 + 1 + 1
9 = 6 + 3
9 = 6 + 2 + 1
9 = 6 + 1 + 1 + 1
9 = 5 + 4
9 = 5 + 3 + 1
9 = 5 + 2 + 2
9 = 5 + 2 + 1 + 1
9 = 5 + 1 + 1 + 1 + 1
9 = 4 + 4 + 1
9 = 4 + 3 + 2
9 = 4 + 3 + 1 + 1
```

Output (2/2)

```
9 = 4 + 2 + 2 + 1
9 = 4 + 2 + 1 + 1 + 1
9 = 4 + 1 + 1 + 1 + 1 + 1
9 = 3 + 3 + 3
9 = 3 + 3 + 2 + 1
9 = 3 + 3 + 1 + 1 + 1
9 = 3 + 2 + 2 + 2
9 = 3 + 2 + 2 + 1 + 1
9 = 3 + 2 + 1 + 1 + 1 + 1
9 = 3 + 1 + 1 + 1 + 1 + 1 + 1
9 = 2 + 2 + 2 + 2 + 1
9 = 2 + 2 + 2 + 1 + 1 + 1
9 = 2 + 2 + 1 + 1 + 1 + 1 + 1
9 = 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1
9 = 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 + 1
```

2 Conjectures de Goldbach

Exercice 5 : La conjecture de Goldbach est un des problèmes les plus fameux des mathématiques. Malgré la simplicité de son énoncé, elle résiste à tous les efforts des mathématiciens depuis près de 400 ans. Elle s'énonce comme suit :

> "Tout nombre entier pair supérieur à 3 peut s'écrire comme la somme de deux nombres premiers."

On appelle décomposition de Goldbach une telle décomposition d'un nombre entier pair.

1. Écrire une fonction `bool estPremier(int n)` qui renvoie si un entier n est premier ou non.
2. Écrire une fonction `int *listePremiers(int n)` qui renvoie un tableau de taille n contenant les nombres premiers plus petits que n (la fin tu tableau contiendra des 0).
3. Écrire une fonction `void goldbach(int n, int *p1, int *p2)` qui prend en entrée un entier n et qui trouve une décomposition de Goldbach de $2n$. On écrira les valeurs des deux nombres premiers de cette décomposition aux adresses fournies par les pointeurs `p1` et `p2`.

Exercice 6 : Une autre conjecture de Goldbach était la suivante :

> "Tout nombre impair strictement supérieur à 3 peut s'écrire sous la forme $p + 2k^2$,
> où p est un nombre premier et k est un entier non nul."

Il se trouve que cette conjecture est fausse.

1. Trouver le premier entier ne satisfaisant pas cette conjecture.
2. Trouver tous les entiers inférieurs à 10^4 ne satisfaisant pas cette conjecture.
3. Quel est le premier entier supérieur à 10^4 ne satisfaisant pas cette conjecture ?