1 Instructions séquences et conditionnelles

1. Ajoutez au fichier `definitionsFonctions.cpp` le bout de code ci–dessous que vous devrez compléter :

```
int  mystere(int n)
{   int v;
   v = 53 + n;
   v = 2 * v;
   v = ...
   ...
   return v;}
```

   Cette fonction, étant donné un entier `n`, effectue en utilisant la variable `v` la séquence de calculs suivante :
   — `v` est initialisé 53 + `n`
   — puis on double la somme précédente
   — à la valeur précédente on retranche 99
   — puis on soustrait `n` à la moitié de la valeur obtenue à l'étape précédente
   — le résultat de `mystere(n)` est alors la valeur finale de la variable `v`.

   Après avoir complété la fonction `mystere`, vous l'exécuterez avec plusieurs valeurs d'argument. Pour cela ajoutez au fichier `programmePrincipal.cpp` les évaluations :

```
EVAL(mystere(23));  et  EVAL(mystere(39));
```

   Compilez. Exécutez. Que constatez–vous ?

2. Traduisez en C/C++, l'algorithme `nbRacines` vu en TD. Les paramètres `a, b, c` sont des nombres réels, le résultat est un nombre entier. Quel est le type des variables `discr` et `nbr`.

   Algorithme : nbRacines
   Données : a : Nombre, b : Nombre, c :Nombre ; `a` et `b` ne sont pas nuls tous les 2.
   Résultat : Nombre, nombre de solutions réelles de l'équation $a.X^2 + b.X + c = 0$
   Variable : discr : Nombre, nbr : Nombre
   discr := b*b - 4*a*c ;
   si `a=0` **alors** nbr := 1;
   sinon
   si `discr < 0` **alors** nbr := 0;
   sinon
   si `discr = 0` **alors** nbr := 1;
   sinon nbr := 2;
   fin si
   fin si
   fin si;
   Le résultat est : `nbr`
   fin algorithme

   Complétez le fichier `programmePrincipal.cpp` en utilisant la fonction `nbRacines` pour calculer le nombre de solutions réelles de $0.3X^2 + X - 3.5 = 0$.

3. (**) Écrivez à présent une fonction C/C++ qui calcule les solutions réelles d'une équation du second degré. Puisque le nombre de solutions peut être 0, 1 ou 2, l'ensemble de ces solutions sera représenté par une liste. Le résultat est donc une liste de nombres réels (de longueur 0, 1 ou 2). Si vous avez oublié les formules permettant d'obtenir les solutions d'une équation du second degré à partir de son discriminant, consultez WIKIPÉDIA.

2 Itérations

L'algorithme utilisant l'itération pour calculer la somme des n premiers entiers s'écrit en C/C++ :

```
int somme(int n )
{
    int som;
    som = 0;
    for(int i=1; i <=  n; i++)
       som = som + i;
    return som;
}
```

1. Inspirez–vous de cette fonction pour écrire une fonction C/C++ `somPair` qui étant donné un entier n calcule la somme des n premiers entiers pairs. Par exemple `somPair(5)` vaut 2+4+6+8+10=30. Testez votre fonction, par exemple avec `EVAL(somPair(5));`.
2. Écrivez une fonction `somImpair` qui étant donné un entier n renvoie la somme des n premiers entiers impairs. Par exemple `somImpair(5)` vaut 1+3+5+7+9=25. Évaluez les expressions :

   ```
   EVAL( somImpair(2) );
   EVAL( somImpair(3) );
   EVAL( somImpair(4) );
   ```

   Que constatez–vous ?
3. Un *nombre parfait* est un entier dont la somme des diviseurs est égale au double de ce nombre. Par exemple 6 est un nombre parfait car la somme de ses diviseurs est 12.
   (a) En TD vous avez écrit un algorithme `somDiv` calculant à l'aide d'une itération la somme des diviseurs d'un nombre entier non nul. Traduisez cet algorithme en C/C++. Testez votre fonction. Par exemple la valeur de `somDiv(6)` doit être 12.
   (b) Écrivez une fonction `estParfait` vérifiant si un entier non nul est un nombre parfait : le résultat de cette fonction est `true` si le nombre est parfait et `false` sinon.
   Est-ce que 28 est un nombre parfait ? Est-ce que 66 est un nombre parfait ?
   (c) Soit `existeParfait` la fonction qui étant donné 2 entiers a et b vérifie si l'intervalle $[a; b]$ contient au moins un nombre parfait. Vous devez écrire 2 versions pour cette fonction :
   — la première version que vous appellerez `existeParfait1` utilise l'itération `Pour`. Testez votre fonction en évaluant `existeParfait1(1,30)`. La réponse doit être `true`.
   — la deuxième version, `existeParfait2`, utilise l'itération `Tant que` (`while`). Dans cette version l'itération s'arrête dès qu'un nombre parfait a été trouvé.
   Évaluez `existeParfait2(1,30)`.
   La deuxième version devrait donc s'exécuter plus rapidement que la première. Pour vous en convaincre évaluez `existeParfait1(10,20000)` puis `existeParfait2(10,20000)`.
   (d) *À la recherche des nombres parfaits.*
   Écrivez une fonction `parfaitSuiv` qui étant donné un entier non nul `n` calcule le plus petit nombre parfait supérieur ou égal à `n`. Cette fonction utilise l'itération `Tant que`.
   Le premier nombre parfait est 6. Pour connaître le deuxième évaluez `parfaitSuiv(7)`. Quel est le troisième nombre parfait ? le quatrième ? N'essayez pas de calculer le cinquième nombre parfait car le calcul risque d'être très long avec cet algorithme (le cinquième nombre parfait est 33 550 336) [1] [2].

1. Pour ceux qui auraient lancé la recherche du 5^e nombre parfait : la commande pour interrompre une exécution est CTRL-G.

2. Pour ceux qui seraient intéressés par les nombres parfaits : à ce jour on ne connaît que 48 nombres parfaits et ils sont tous pairs. Pour en savoir davantage consultez le site WIKIPÉDIA.

(e) Écrivez la fonction C/C++

$$\begin{array}{rccl} \textbf{emeParfait} : & \mathbb{N} & \longrightarrow & \mathbb{N} \\ & n & \longmapsto & \text{le } n^{ème} \text{ nombre parfait} \end{array}$$

Inspirez–vous de l'algorithme vu en TD calculant le $n^{ème}$ nombre multiple soit de 2 soit de 3. Utilisez cette fonction pour calculer le quatrième nombre parfait. N'essayez pas de rechercher des nombres parfaits plus grands pour la raison évoquée précédemment.

4. Deux nombres entiers a et b sont *des nombres amis* si la somme des diviseurs de a est égale à $a + b$, tout comme la somme des diviseurs de b. Par exemple 220 et 284 sont des nombres amis. En effet :
   — la somme des diviseurs de 220 est
   $$1 + 2 + 4 + 5 + 10 + 11 + 20 + 22 + 44 + 55 + 110 + 220 = 504 = 220 + 284.$$
   — la somme des diviseurs de 284 est $1 + 2 + 4 + 71 + 142 + 284 = 504 = 220 + 284$.

   (a) Écrivez la fonction `sontAmis` qui étant donné 2 nombres entiers non nuls vérifie si ces nombres sont amis.

   (b) Un nombre peut être ami avec au plus un nombre. Si n possède un ami, quel est ce nombre ? Définissez sa valeur en utilisant n et `somDiv`. Écrivez une fonction `aUnAmi` qui étant donné un entier vérifie si cet entier possède un ami (il suffit d'utiliser `sontAmis` et `somDiv` avec les bons paramètres).

   (c) Écrivez une fonction `liAmicaux` qui étant donné un entier n calcule la liste des nombres inférieurs à n qui possèdent un ami.
   Quels sont les nombres amicaux inférieurs à 10000 ?

   (d) (***) Écrivez la fonction `liAmis` qui étant donné un entier n calcule la liste des couples de nombres amis inférieurs à n que l'on représentera par une liste de listes d'entiers (`list<list<int>>` en C/C++). Par exemple la liste des couples d'amis inférieurs à 2000 est ((1184 1210) (496 496) (220 284) (28 28) (6 6)).
   On constate que certains entiers sont amis avec eux-mêmes. Quels sont ces nombres ?

5. Il faut écrire une série de fonctions dont le résultat est une liste de couples d'entiers. Comme à la question précédente on représentera un couple d'entiers par une liste de 2 entiers. Le résultat est donc une liste de listes d'entiers (`list<list<int>>` en C/C++). L'ordre des éléments dans cette liste est quelconque.

   (a) Écrivez une fonction `liCouples1` qui étant donné un entier $n \in \mathbb{N}^*$ calcule la liste de tous les couples d'entiers strictement positifs de la forme [i ; j] avec $i + j = n$.
   Ex : `liCouples1(6)=((5 1) (4 2) (3 3) (2 4) (1 5))`.

   (b) Écrivez une fonction `liCouples2` qui étant donné un entier $n \in \mathbb{N}^*$ calcule la liste de tous les couples d'entiers strictement positifs inférieurs à n. Pour traiter cette question vous devrez utiliser 2 itérations imbriquées.
   Ex : `liCouples2(3)=((1 1) (1 2) (1 3) (2 1) (2 2) (2 3) (3 1) (3 2) (3 3))`

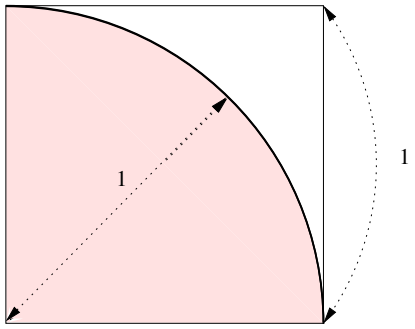
   (c) Écrivez une fonction `liCouples3` qui étant donné un entier $n \in \mathbb{N}^*$ calcule la liste de tous les couples d'entiers de la forme (i j) avec $1 \leqslant i < j \leqslant n$.
   Ex : `liCouples3(3)=((1 2) (1 3) (2 3))`

   (d) Écrivez une fonction `liCouples4` qui étant donné un entier $n \in \mathbb{N}^*$ calcule la liste de tous les couples d'entiers de la forme (i j) avec $1 \leqslant i < j \leqslant n$ et j est multiple de i.
   Ex : `liCouples4(6)=((1 2) (1 3) (1 4) (1 5) (1 6) (2 4) (2 6) (3 6))`

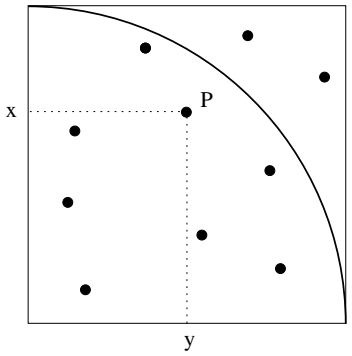
   (e) (***) Écrivez une fonction `liCouples5` qui étant donné un entier $n \in \mathbb{N}^*$ calcule la liste de tous les couples d'entiers de la forme (i j) avec $1 \leqslant i \leqslant n$ et j est le plus grand multiple de i inférieur ou égal à n.
   Ex : `liCouples5(8)=((1 8) (2 8) (3 6) (4 8) (5 5) (6 6) (7 7) (8 8))`

3 Méthode Monte–Carlo

Les méthodes Monte–Carlo consistent en des simulations de problèmes mathématiques ou physiques basées sur la génération aléatoire de nombres. Ce principe peut être utilisé pour trouver une approximation du nombre Π.
Soit un carré de coté 1 et un quart de cercle, de rayon 1, inscrit dans ce carré comme le montre la figure *a* ci-après. Le rapport entre la surface du quart de cercle et la surface du carré est $\frac{\Pi}{4}$. On peut estimer cette valeur en tirant uniformément $nTir$ points à l'intérieur du carré et en comptant le nombre $nInt$ de points situés à l'intérieur du quart de cercle : $\frac{nInt}{nTir}$ est une approximation de $\frac{\Pi}{4}$ d'autant meilleure que $nTir$ est grand (figure *b*).



a



b

Pour tirer aléatoirement un point à l'intérieur du carré il suffit de choisir aléatoirement deux coordonnées x et y dans l'intervalle réel $[0, 1]$.

1. L'expression `(float) rand()/ RAND_MAX` a pour valeur un réel aléatoire de l'intervalle $[0, 1]$. Testez cette expression en évaluant dans programmePrincipal.cpp :

   ```
   EVAL((float) rand()/ RAND_MAX);
   EVAL((float) rand()/ RAND_MAX);
   EVAL((float) rand()/ RAND_MAX);
   ```

2. Écrivez une fonction `dansCercle` qui étant donné 2 réels x et y pris dans l'intervalle réel $[0, 1]$ teste si le point de coordonnées (x, y) est à l'intérieur du quart de cercle de rayon 1.
3. En utilisant l'expression `(float) rand()/ RAND_MAX` et la fonction `dansCercle`, écrivez la fonction `monteCarlo` de paramètre $nTir$ calculant une approximation de Π, en complétant le texte ci-dessous.

   ```
   float monteCarlo(int nTir)
   {
       int nInt;  float x, y;
       ...
       return 4 * (float) nInt / nTir;
   }
   ```

4. Testez `monteCarlo` :

   ```
   EVAL( monteCarlo(1000) );
   EVAL( monteCarlo(1000) );
   EVAL( monteCarlo(5000) );
   EVAL( monteCarlo(10000) );
   EVAL( monteCarlo(20000) );
   EVAL( monteCarlo(100000) );
   ```

4 Tableaux

Placez vous dans le fichier `programmePrincipal.cpp` et ajoutez les lignes suivantes. Après chaque ajout, vous compilerez, puis exécuterez le nouveau programme.
Ajoutez la ligne suivante qui déclare la variable `ta` comme étant un tableau de 3 entiers et l'initialise au tableau [1 2 6] :

```
vector<int> ta (3); ta.at(0) = 1; ta.at(1) = 2; ta.at(2) = 6;
```

Ajoutez les lignes suivantes :
`EVAL( ta );` *(donne la valeur du tableau)*
`EVAL( ta.at(0) );` *(donne la valeur de l'élément d'indice 0)*
`EVAL( taille(ta) );` *(donne la taille du tableau)*
`EVAL( ta.at(2) );` *(donne la valeur de l'élément d'indice 2)*
On rappelle que les éléments d'un tableau sont indicés entre 0 et la taille du tableau moins 1. Le dernier élément de `ta` a pour indice 2. L'élément d'indice 3 n'existe pas.
Testez ce dernier point en ajoutant la ligne :
`EVAL( ta.at(3) );`
Lors de l'exécution une erreur est signalée indiquant que 3 n'est pas un indice. Supprimez la ligne précédente.
Les instructions suivantes affectent de nouvelles valeurs à certains éléments de `ta`.

```
ta.at(2) = 9 ;
ta.at(1) = ta.at(0) + ta.at(2) ;
```

Quelle est la nouvelle valeur de `ta` ?
La fonction `tableau` a pour résultat un tableau d'entiers dont les éléments sont passés en paramètre sous forme d'un ensemble d'entiers.
Ajoutez la ligne :
`EVAL( tableau({4,2,7}) );`

5 Algorithmes sur les tableaux

Vous ajouterez au fichier `definitionsFonctions.cpp` les définitions des fonctions suivantes :

1. Écrivez la fonction `somTab` calculant la somme des éléments d'un tableau d'entiers

   ```
   int somTab(vector<int> T)
   {
     int som;
     ...
     return som;
   }
   ```

   Testez votre fonction en évaluant (dans `programmePrincipal.cpp`)

   ```
   EVAL( somTab(ta) );
   EVAL( somTab(tableau({3,1,7,2,1})) );
   ```

2. Écrivez une fonction testant si 2 tableaux sont égaux, c'est à dire s'ils ont même taille et mêmes éléments :

   ```
   bool tabEgaux(vector<int> t1, vector<int> t2)
   {
   ...
   ```

Testez votre code :

```
EVAL( tabEgaux(ta, tableau({1,2,3})) );
EVAL( tabEgaux(ta, tableau({1,10, 9})) );
EVAL( tabEgaux(ta, tableau({1,10})) );
```

3. Écrivez la fonction `tabCarres` dont la donnée est un entier n et le résultat le tableau d'entiers tr de taille n tel que $\forall\ i \in\ [0..n-1], tr[i] = i^2$.
   Pour cela complétez ce qui suit :

```
vector<int> tabCarres(int n)
{
  vector<int> ta(n);
  ...
  return ta;
}
```

   Testez votre fonction.

4. De la même façon écrivez la fonction `tabSuite` qui étant donné un entier n donne en résultat le tableau de taille n contenant les n premiers termes de la suite $(U_k)_{k\ \mathbb{N}}$.
   $\begin{cases} U_0 &= 0 \\ U_k &= U_{k-1} + 2 \times (k-1) + 1 \text{ pour } k \in \mathbb{N}^* \end{cases}$
   Testez votre fonction et comparez le résultat avec celui de `tabCarres` :

```
EVAL( tabSuite(6) );
EVAL( tabEgaux(tabCarres(8),tabSuite(8)) );
```

5. On souhaite calculer la racine carrée entière d'un entier positif n, c'est à dire l'entier i tel que $i^2 \leqslant n < (i+1)^2$. Par exemple la racine carrée entière de 12 est 3. La racine carrée entière d'un nombre n peut être calculée à partir du tableau des carrés calculé par la fonction `tabCarres`. Écrivez la fonction qui étant donné un entier n inférieur à 10000 calcule la racine entière de n en construisant la tableau des 100 premiers carrés.

```
int racEnt(int n)
{
  vector<int> ta=tabCarres(100);
  ...
```

   Utilisez votre fonction `racEnt` pour calculer les racines entières de 25, 30 et 7543 ?

6. Écrivez une fonction `nbMaxLoc` qui, étant donné un tableau d'entiers ta, calcule son nombre de "maximum local". Un élément $ta[i]$ est un maximum local si les éléments d'indice $i-1$ et $i+1$, s'ils existent, sont inférieurs à $ta[i]$.
   Par exemple les "maximum local" du tableau [2 6 1 3 4 8 12 7 9 11 3 2 7 21] sont 6,12,11,21. Pour ce tableau le résultat de la fonction est donc 4.
   Testez en évaluant `EVAL( nbMaxLoc(tableau( {1,3,8,7,5,2,9,5,11})));`.

7. Soit ta un tableau d'entiers. On cherche à calculer `difMin(ta)`, la plus petite différence entre les valeurs des éléments de ta.
   Exemples :
   la plus petite différence du tableau `[ 9 2 4 7 3 ]` est 1 ; cette différence est atteinte pour les couples de valeurs (2,3) ou (3,4).
   la plus petite différence du tableau `[ 9 4 9 3 ]` vaut 0 ; cette différence est atteinte pour les éléments d'indice 0 et 2, deux éléments différents de même valeur (9).

(a) Écrivez `difMinTrie`, une première version de `difMin` en supposant que la donnée est un tableau trié d'entiers. Dans ce cas la plus petite différence est atteinte entre des éléments consécutifs du tableau. Testez :

```
EVAL( difMinTrie(tableau( {2,5,7,9,12,18,21,22,27,30})) );
```

(b) Écrivez une deuxième version de `difMin` qui opère pour un tableau d'entiers quelconque, pas nécessairement trié. Dans ce cas il faut utiliser 2 itérations `Pour` imbriquées pour tester tous les couples d'éléments du tableau (pas uniquement les couples d'éléments consécutifs).

```
EVAL( difMin(tableau({9,3,11,8,13,20,18,33,1})) );
```

5.1 Représentation et manipulation de polynômes

On représente un polynôme par un tableau de nombres, l'élément d'indice i étant le coefficient du monôme de degré i. Ainsi le polynôme $3 + 5.x + 4.x^2 + 2.x^3$ sera représenté par le tableau

0	*1*	*2*	*3*
3	5	4	2

Inversement un tableau t de n nombres représente le polynôme $\sum_{i=0}^{n-1} t[i].x^i$.

0	*1*	*2*	*3*	*4*
0	5	0	1	0

représente le polynôme $0 + 5.x + 0.x^2 + 1.x^3 + 0.x^4$, c'est à dire $x^3 + 5x$.
En C/C++ on représentera un polynôme par un tableau d'entiers. Les polynômes représentés seront donc des polynômes à coefficients entiers.
Dans la suite on vous demande d'écrire des fonctions C/C++ correspondant à des opérations classiques sur les polynômes (degré, valeur en un point, somme, polynôme dérivé, ...). Pour visualiser le résultat de vos fonctions nous avons écrit une fonction appelée `ecrPoly` affichant sous une forme plus lisible le polynôme représenté par un tableau d'entiers.
Placez–vous dans la fenêtre de l'interpréteur et évaluez :

```
vector<int> poly1 = tableau( {3,0,2,0,1,0});
vector<int> poly2 = tableau( {0,1,2,3,-1});
ecrPoly(poly1);
ecrPoly(poly2);
```

1. Écrivez la fonction `degre` qui étant donné un tableau d'entiers, calcule le degré du polynôme représenté par ce tableau, c'est à dire le degré max des monômes à coefficient non nul. Par exemple le degré de `poly1` et `poly2` est 4. Testez votre fonction.
2. Écrivez la fonction `valPoly` qui étant donné un entier `v` et un tableau d'entiers représentant un polynôme `p`, donne comme résultat la valeur du polynôme `p` pour la valeur `v`.
   Par exemple `valPoly(1,poly1)` vaut 6 et `valPoly(2,poly1)` vaut 27.
3. Écrivez la fonction `somPoly` qui étant donné 2 polynômes calcule la somme des 2 polynômes.
   Par exemple `somPoly(poly1,poly2)` a pour résultat le tableau [3 1 4 3 0], c'est à dire le polynôme $3 + x + 4.x^2 + 3.x^3$.
4. Écrivez la fonction `derivePoly` qui étant donné 1 polynôme calcule son polynôme dérivé.
   On rappelle que la dérivée du polynôme $\sum_{i=0}^{n-1} a_i.x^i$ est le polynôme $\sum_{i=1}^{n-1} i.a_{i-1}.x^{i-1}$.
   Par exemple `derivePoly(poly1)` a pour résultat le tableau [0 4 0 4], c'est à dire le polynôme $4.x + 4.x^3$.

5. (*) Écrivez la fonction `mulPoly` qui étant donné 2 polynômes calcule le produit des 2 polynômes.

   Par exemple `ecrPoly(mulPoly( tableau(2,1), tableau(2,1)) );` affiche $4 + 4.x + x^2$.

6 Algorithmes de tri

1. Programmez l'algorithme de tri à bulles étudié en cours.
2. Un algorithme de tri de notes. On considère ici un tableau de notes, c'est à dire un tableau dont les éléments sont des entiers compris entre 0 et 20.

   Voici un exemple de tableau `tExNotes` de notes

12	6	14	9	9	14	17	9	12	12	10

   (a) Écrivez une fonction `tabOccur` qui a pour donnée un tableau de notes `tNotes` et pour résultat le tableau `tNbOcc` de taille 21 et dont le $i^{ème}$ élément est le nombre d'éléments de `tNotes` dont la valeur est la note i.

   Par exemple si la donnée est le tableau `tExNotes`, le résultat sera le tableau :

0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*14*	*15*	*16*	*17*	*18*	*19*	*20*
0	0	0	0	0	0	1	0	0	3	1	0	3	0	2	0	0	1	0	0	0

   En effet le tableau `tExNotes` contient :

   0 note égale à 0
   0 note égale à 1
   ...
   0 note égale à 5
   1 note égale à 6
   0 note égale à 7
   0 note égale à 8
   3 notes égales à 9
   1 note égale à 10
   ...

   (b) Pour trier un tableau de notes `tNotes` l'algorithme de *tri par comptage* procède en 2 étapes :

   i. utilisation de la fonction `tabOcur` pour construire `tOcc` le tableau d'occurrences de `tNotes`.
   ii. utilisation du tableau d'occurrences pour obtenir le tableau `tNotes` trié ; le tableau `tNotes` trié contient `tOcc[0]` 0 puis `tOcc[1]` 1, puis ... `tOcc[20]` 20.

   Écrivez une fonction qui trie un tableau de notes en appliquant l'algorithme de tri par comptage.

   Le résultat pour la donnée `tExNotes` est

6	9	9	9	10	12	12	12	14	14	17

3. Programmez l'algorithme de tri par sélection étudié en TD.