Rappel : si on utilise la librairie `math.h`, il faut utiliser `gcc` avec l'option `-lm`.

Exemple

```
eleve@mp2i:~$ gcc exemple.c -o exemple -lm
```

1 Types structurés

Exercice 1 (les complexes) : Soit $z \in \mathbb{C}$. On rappelle que z peut s'écrire de manière unique $z = x + iy$ avec $(x, y) \in \mathbb{R}^2$. x (resp. y) s'appelle la partie réelle (resp. imaginaire) de z.

1. Créer un type `complexe` ayant deux champs de type `float` : `reel` et `imag`.
2. Écrire une fonction `void print_complexe(complexe z)` qui prend un complexe $z = x + iy$ et affiche `x + i*y` à l'écran.
3. Écrire une fonction `complexe add(complexe z1, complexe z2)` qui renvoie le complexe z correspondant à $z_1 + z_2$.
4. Écrire une fonction `complexe m_const(complexe z, float a)` qui renvoie le complexe z correspondant à $a * z$.
5. En déduire une fonction `complexe soustract(complexe z1, complexe z2)` qui renvoie le complexe z correspondant à $z_1 - z_2$.
6. Écrire une fonction `complexe prod(complexe z1, complexe z2)` qui renvoie le complexe z correspondant à $z_1 * z_2$.
7. Écrire une fonction `float module(complexe z)` qui renvoie $|z|$.
8. Écrire une fonction `complexe conjug(complexe z)` qui renvoie $\bar{z}$.
9. En déduire une fonction `complexe div(complexe z1, complexe z2)` qui renvoie le complexe z correspondant à z_1/z_2.

2 Analyse d'algorithmes

On peut mesurer le temps d'exécution (en microsecondes) d'une fonction l'aide du programme suivant :

```c
#include <sys/time.h>
#include <stdio.h>

int main()
{
  struct timeval t1, t2;
  gettimeofday(&t1, NULL);

  /* fonction à tester ici */

  gettimeofday(&t2, NULL);
  printf("Temps d'exécution : %lu microsecondes\n",t2.tv_usec - t1.tv_usec);
}
```

Exercice 2 (Statistiques) : On rappelle la définition de la moyenne et de la variance d'un tableaux de valeurs t de taille n :

$$\bar{t} = \frac{1}{n}\sum_{i=0}^{n-1} t[i] \quad ; \quad \sigma^2 = \frac{1}{n}\sum_{i=0}^{n-1}(t[i] - \bar{t})^2$$

1. Écrire une fonction `float moyenne(float *tab, int n)` qui calcule la moyenne d'un tableau de flottants de taille n.
2. Écrire une fonction `float variance(float *tab, int n)` qui calcule la variance σ^2 d'un tableau de flottants de taille n.
3. Quelle est la complexité de ces programmes ?
   Mesurer leur temps d'exécution sur des tableaux de taille 1000.

Exercice 3 (tri à bulles) : Le **tri à bulles** est un algorithme de tri qui repose sur le principe suivant :

(i) On parcours le tableau que l'on veut trier de gauche à droite, en comparant chaque case avec celle à sa droite : si ces deux cases ne sont pas bien ordonnées, on les échange (sinon on ne fait rien).

(ii) (a) Si l'on n'a jamais fait d'échange durant l'étape (i), c'est que le tableau est trié : on s'arrête ;

(b) sinon (on a fait au moins un échange), on recommence l'étape (i).

1. Comment faire pour savoir si on a effectué au moins un échange à l'étape (i) ?
2. Effectuer à la main l'étape (i) sur un exemple.
   En déduire une optimisation de la prochaine exécution de l'étape (i).
3. Écrire une fonction `void echange(int* tab, int i, int j)` qui échange les valeurs de `tab[i]` et `tab[j]`.
4. Écrire une fonction `void tri_bulle(int *tab, int n)` qui trie un tableau `tab` de longueur n sur le principe du tri à bulles.
5. Montrer que votre fonction termine.
6. Quelle est la complexité de votre fonction dans le pire des cas et dans le meilleur des cas ?
7. (a) Pour chaque boucle de votre algorithme, proposer un invariant de boucle.
   (b) Montrer que votre algorithme est correct.

Exercice 4 (tri par insertion) : Le **tri par insertion** est un algorithme de tri qui repose sur le principe suivant :

- on considère les uns après les autres les éléments du tableau :
  - au moment où l'on considère l'élément d'indice i du tableau, les éléments qui le précèdent sont déjà triés ;
  - on cherche alors la position où insérer cet élément parmi les précédents, pour que les $i+1$ premiers éléments du tableau soient triés.
    ↪ Pour cela, il faut comparer la valeur `x` à insérer à la valeur en $i-1$, puis $i-2$, ..., tant que `x` est plus grande que les valeurs trouvées à ces indices, et décaler au fur et à mesure ces valeurs vers la droite ;
    ↪ lorsqu'on trouve une valeur plus petite, c'est en cette position qu'il faut insérer `x`.

1. Écrire une fonction `void tri_insertion(int *tab, int n)` qui trie un tableau `tab` de longueur n sur le principe du tri par insertion.
2. Montrer que votre fonction termine.
3. Quelle est la complexité de votre fonction dans le pire des cas et dans le meilleur des cas ?
4. (a) Pour chaque boucle de votre algorithme, proposer un invariant de boucle.
   (b) Montrer que votre algorithme est correct.

Exercice 5 (Trouver la star) : On considère un groupe de n individus, numérotés de 0 à $n-1$. On appelle **star** une personne connue de tous mais qui ne connaît personne. On dispose d'un tableau `bool **know` à n lignes et n colonnes tel que `know[i][j]` indique si l'individu i connait l'individu j.

Pour tester vos programmes, vous trouverez sur la page web du cours un fichier `star42.c` contenant une fonction `bool **star42(int n)` : cette fonction renvoie un tableau `know` de taille $n \times n$ généré au hasard et dont la star est l'individu 42.

1. Combien peut-il y avoir de stars dans le groupe d'individus ?
2. Écrire une fonction `int star(bool **know, int n)` qui renvoie la star d'un groupe de n personnes (on renverra -1 s'il n'y a pas de star).
3. Quelle est la complexité de votre algorithme (dans le pire des cas) ?
4. Il est possible de résoudre ce problème en temps linéaire. Si ce n'est pas déjà le cas de votre algorithme, écrire un algorithme ayant une complexité temporelle en $O(n)$.