Un fichier `fenelon_tp10.zip` est à télécharger sur la page web du cours, et à extraire dans votre dossier de travail.

Exercice 1 (trouver la star) : Pour cet exercice, ouvrir le fichier `star.c`.
On considère une groupe de n individus, numérotés de 0 à $n-1$. On appelle **star** une personne connue de tous mais qui ne connait personne. On modélise un tel groupe par un tableau `bool** connait` à n lignes et n colonnes, tel que `connait[i][j]` vaut `true` si l'individu i connait l'individu j (et `false` sinon).

1. Combien peut-il y avoir de stars dans un groupe d'individus ?
2. Écrire une fonction `bool** creer_matrice(int n);` renvoyant une matrice de booléens allouée sur le tas de taille $n \times n$ dont toutes les cases sont initialisées à `false`.
3. Écrire une fonction `void liberer_matrice(bool** M, int n);` libérant toute la mémoire allouée par un matrice `M` renvoyée par la fonction précédente.

On fournit une fonction `bool** creer_connait(int n, int star);` qui génère une matrice aléatoirement représentant n individus dont la star est l'individu numéro `star`.

4. Écrire une fonction `int trouver_star(bool** connait, int n);` qui cherche s'il y a une star dans un groupe de n individus représentés par la matrice `connait` prise en argument. On renverra le numéro de l'individu qui est la star si un tel individu existe, et -1 sinon.

```
$ gcc star.c -o test_star; ./test_star
La star est : 42
```

5. Quelle est la complexité temporelle de votre fonction dans le pire des cas ?
6. Il est possible de trouver la star avec une complexité temporelle linéaire. Si ce n'est pas déjà le cas pour votre fonction, écrire une fonction qui trouve la star avec une complexité temporelle en $\mathcal{O}(n)$.

Exercice 2 (structure de pile) : Pour cet exercice, ouvrir le fichier `pile.c`.
On se propose d'implémenter une structure de pile via des listes chainées. On se donne les types suivants :

```
typedef struct maillon maillon;
struct maillon {
  maillon* suivant;
  valeur_t val;
};

typedef struct pile pile;
struct pile {
  maillon* sommet;
};
```

1. Écrire une fonction `pile* creer_pile(void);` renvoyant une novelle pile vide allouée sur le tas.
2. Écrire une fonction `bool pile_est_vide(pile* p);` prenant en argument un pointeur vers une pile et renvoyant `true` si la pile est vide (et `false` sinon).
3. Écrire une fonction `void empiler(pile* p, valeur_t x);` prenant en arguments un pointeur vers une pile et une valeur x et modifiant la pile pour y empiler x au sommet.
4. Écrire une fonction `valeur_t sommet(pile* p);` prenant en arguments un pointeur vers une pile supposée non vide et renvoyant la valeur stockée au sommet de la pile (sans modifier la pile).
5. Écrire une fonction `valeur_t depiler(pile* p);` prenant en arguments un pointeur vers une pile supposée non vide et renvoyant la valeur stockée au sommet de la pile (cette fois-ci, on modifiera la pile pour lui retirer son sommet).
6. Écrire une fonction `void liberer_pile(pile* p);` prenant en arguments un pointeur vers une pile et libérant toute la mémoire allouée sur le tas par cette pile.

Exercice 3 (mots bien parenthésés) : Pour cet exercice, ouvrir le fichier `parentheses.c`.
On souhaite vérifier qu'une chaine de caractères est bien parenthésée. Pour cela, on utilise la structure de pile codée dans l'exercice 2. On rappelle une convention des chaines de caractères en C : si la chaine est de longueur n, on a en mémoire un tableau d'au moins $n+1$ caractères, dont les cases d'indices 0 à $n-1$ contiennent les caractères de la chaine, et dont le caractère d'indice n contient le caractère spécial `'\0'` (s'il y a d'autres cases après ce caractère, elles sont ignorées).

1. Écrire une fonction `int longueur(char* s);` prenant en argument une chaine de caractères `s` respectant cette convention, et renvoyant la longueur de la chaine.
   (il est interdit d'utiliser la fonction `strlen` de la librairie `string.h`)

Pour la prochaine question, on suppose que nos chaines de caractères ne contiennent que les caractères `'('`,`')'` (et le caractère spécial `'\0'`), et qu'elle est bien parenthésée (c'est-à-dire que chaque parenthèse ouvrante est correctement fermée, et que chaque parenthèse fermante est également correctement ouverte). On souhaite, pour une telle chaine de taille n, calculer un tableau d'entiers `par` de longueur n tel que $\forall i \in [\![0, n-1]\!]$, `par[i]` contienne :

- l'indice i si `s[i]` est une parenthèse ouvrante ;
- l'indice de la parenthèse ouvrante correspondante si `s[i]` est une parenthèse fermante.

2. Écrire une fonction `int* parentheses(char* s);` prenant en argument une telle chaine et renvoyant le tableau `par` correspondant. Votre fonction devra avoir une complexité en $\mathcal{O}(n)$, et devra échouer au moyen d'un `assert` si la chaine `s` ne vérifie pas les hypothèses.
   Indication : on utilisera une structure de pile.

Tests question 2

```
$ gcc pile.c parentheses.c -o testQ2; ./testQ2
(()())() : 0 1 1 3 3 0 6 6
$ gcc pile.c parentheses.c -o testQ2; ./testQ2
((()))() : testQ2 pile.c:36: depiler: Assertion `!pile_est_vide(p)' failed.
Aborted (core dumped)
$ gcc pile.c parentheses.c -o testQ2; ./testQ2
(()())( : testQ2 parentheses.c:27: parentheses: Assertion `pile_est_vide(p)' failed.
Aborted (core dumped)
```

On revient maintenant au cas général : pour la prochaine question, nos chaines peuvent contenir n'importe quel caractère. On souhaite vérifier que toutes les parenthèses ouvrantes sont bien correctement fermées et que toutes les parenthèses fermantes sont bien correctement ouvertes (on traite les parenthèses suivantes : `"([{<>}])"`, les autres caractères sont ignorés).

3. Écrire une fonction `bool bien_parenthesee(char* s);` prenant en argument une chaine de caractères `s` et renvoyant `true` si c'est une chaine bien parenthésée (et `false` sinon). Votre fonction devra avoir une complexité en $\mathcal{O}(n)$.

Tests question 3

```
$ gcc pile.c parentheses.c -o testQ3; ./testQ3
f(g[x],{y})(<>) est bien prenthésée
$ gcc pile.c parentheses.c -o testQ3; ./testQ3
f(g[x],{y})(<>)) n'est pas bien prenthésée
$ gcc pile.c parentheses.c -o testQ3; ./testQ3
f(g[x],{y})(<> n'est pas bien prenthésée
$ gcc pile.c parentheses.c -o testQ3; ./testQ3
f(g[x],{y))(<>) n'est pas bien prenthésée
```

Exercice 4 (valeurs inférieures les plus proches) : Pour cet exercice, ouvrir le fichier `valeurs.c`.
On s'intéresse au problème suivant : étant donné un tableau `tab` de n entiers, on souhaite calculer un tableau `vipp` de n entiers tel que :

$$\forall i \in [\![0, n-1]\!],\ \texttt{vipp[i]} = \texttt{tab[j\_m]} \text{ avec } \texttt{j\_m} = \max_{0 \leq j < i} \{j \mid \texttt{tab[j]} < \texttt{tab[i]}\}$$

Par convention, `vipp[i]` contient -1 si l'ensemble ci-dessus est vide.

1. Écrire une fonction `int* calcul_vipp(int* tab, int n);` prenant en arguments un tableau d'entiers `tab` et sa longueur n, et renvoyant une tableau contenant les valeurs inférieures les plus proches du tableau `tab`.
2. Quelle est la complexité de votre fonction ?
3. Si ce n'est pas déjà le cas, améliorer votre fonction pour qu'elle fonctionne en $\mathcal{O}(n)$.
   Indication : on pourra utiliser la structure de pile codée à l'exercice 2.

```
$ gcc pile.c valeurs.c -o test_vvip; ./test_vvip
Tableau initial :   0  8  4 12  2 10  6 14  1  9  5 13  3 11  7 15
Tableau résultat : -1  0  0  4  0  2  2  6  0  1  1  5  1  3  3  7
```

Exercice 5 (circulation automobile) : Pour cet exercice, ouvrir le fichier `voiture.c`.
Nous souhaitons simuler (de façon très simplifiée) la circulation automobile sur un route à une seule voie (et dans un seul sens, de droite à gauche). L'idée est la suivante : on discrétise la route qui est donc assimilée à un tableau d'entiers de longueur n (une case contient 0 si elle est libre, est un entier strictement positif représentant un modèle de véhicule sinon). Si au temps t, un véhicule v se trouve dans la case d'indice i, alors :

- si la case d'indice $i-1$ est libre au temps t, le véhicule v se retrouve dans la case d'indice $i-1$ au temps $t+1$;
- si la case d'indice $i-1$ est libre au temps t, le véhicule v reste dans la case d'indice i au temps $t+1$.

Pour les cases extrémales, on impose le comportement suivant :

- un véhicule situé dans la case d'indice 0 au temps t disparait au temps $t+1$;
- si la case d'indice $n-1$ est libre au temps t, alors un véhicule peut y apparaitre de façon aléatoire au temps $t+1$;
- si la case d'indice $n-1$ est occupée au temps t, alors aucun véhicule ne peut y apparaitre au temps $t+1$.

Nous allons implémenter un modèle ayant 4 types de véhicules ; une case de la route contiendra donc une valeur dans $[\![0, 4]\!]$, 0 désignant une case libre, et les autres valeurs représentant ces 4 types de véhicules.

1. Écrire une fonction `int nouveau_vehicule(int f);` prenant en argument un entier f (f est la fréquence d'apparition), et renvoyant :
   - 0 avec probabilité $\frac{1}{f}$;
   - un entier représentant un véhicule sinon (chaque véhicule avec une probabilité $\frac{1}{4}$).
2. Écrire une fonction `int* init(int f, int n);` qui crée un tableau de taille n (alloué sur le tas) et initialise toutes ses cases à l'aide de la fonction `nouveau_vehicule`, puis renvoie ce tableau.
3. Écrire une fonction `int* copie(int* tab, int n);` prenant en arguments un tableau d'entiers `tab` et sa longueur n, et qui effectue une copie de `tab` (allouée sur le tas) et la renvoie.
4. Écrire une fonction `int* next(int* route, int n, int f);` prenant en arguments un tableau de taille n représentant l'état de la route et une fréquence d'apparition, et renvoyant le prochain état de la route selon les règles de modélisation spécifiées plus haut.
5. Tester votre programme avec une route de taille 60, une fréquence $f = 2$, et les temps $0 \leq t \leq 30$.
   Remarque : si vous avez des problèmes d'affichage des voitures dans votre terminal, essayer le terminal de VSCode.
6. Votre programme a-t-il des fuites mémoire (des tableaux créés par `malloc` mais non libérés par `free`) ? Si oui, corriger ces fuites mémoire.