On testera toutes les fonctions avec des valeurs au choix. On utilisera les bibliothèques suivantes : `stdio.h`, `math.h`, `stdlib.h`, `stdbool.h`, `time.h` et `assert.h`.

Exercice 0

1. Créer un fichier d'extension `.c`.
2. Inclure les bibliothèques citées ci-dessus.
3. Écrire une fonction `main` qui affiche « Hello World ! ».
4. Compiler puis exécuter le fichier. On pourra utiliser les options de compilation suivantes :
   - `-o` *`nom_exec`* : pour nommer le fichier exécutable (remplacer *`nom_exec`* par le nom choisi) ;
   - `-Wall -Wextra` : pour que le compilateur affiche plus d'avertissements que par défaut ;
   - `-fsanitize=address` : pour avoir des alertes lors de problèmes de gestion de la mémoire lors de l'exécution, par exemple une fuite mémoire.

1 Arithmétique et tableaux

Exercice 1

1. Écrire une fonction `int pgcd(int a, int b)` qui renvoie le pgcd de deux entiers naturels.
2. Écrire une fonction `float somme(int n)` qui calcule et renvoie $\sum_{k=1}^{n} \frac{1}{k^2}$.
3. Écrire une fonction `int seuil(int n)` qui renvoie le plus petit entier k tel que $2^k \geqslant n$.

Exercice 2

1. Écrire une fonction `int maximum(int tab[], int len)` qui prend en argument un pointeur de tableau d'entiers *tab* et un entier correspondant à la taille du tableau et renvoie la valeur maximale de t.
2. Modifier la fonction précédente pour qu'elle renvoie l'indice de la valeur maximale.
3. Écrire une fonction `void afficher_tableau(int tab[], int len)` qui prend en argument un pointeur de tableau d'entiers *tab* et un entier correspondant à la taille du tableau et affiche le tableau t en terminant par un retour à la ligne.
4. Écrire une fonction `int* tableau_alea(int len, int m)` qui prend en argument deux entiers `len` et m et renvoie un pointeur vers un tableau d'entier de taille `len` contenant des entiers aléatoires compris entre 0 et $m-1$.

   La fonction *`int rand(void)`* *renvoie un entier choisi aléatoirement et uniformément entre* 0 *et* *`INT_MAX`*. *On supposera* $m \ll$ *`INT_MAX`*.
5. Faire un test de création et d'affichage d'un tableau de taille 20 contenant des entiers inférieurs à 30. Que remarque-t-on si on exécute plusieurs fois le fichier ? Pour éviter ce problème, on pourra modifier la graine aléatoire par la commande (à n'exécuter qu'une seule fois dans le `main`) `srand(time(0));`.

Exercice 3

1. Écrire une fonction `bool premier(int n)` qui détermine si un entier naturel n est premier ou non. La fonction devra avoir une complexité temporelle en $\mathcal{O}(\sqrt{n})$.
2. Écrire une fonction `int* crible(int n, int* len)` qui renvoie un pointeur vers un tableau d'entiers contenant tous les nombres premiers inférieurs ou égaux à un entier naturel n donné en argument, et modifie l'entier pointé par `len` pour y indiquer la taille du tableau créé. L'algorithme devra utiliser le principe du crible d'Érathosthène.

Exercice 4

1. Écrire une fonction `void tri_selection(int tab[], int len)` qui trie en place un tableau d'entier donné en argument selon le principe du tri par sélection, l'argument `len` correspondant à sa taille.
2. Écrire une fonction `bool appartient(int x, int tab[], int len)` qui teste l'appartenance d'un entier x à un tableau *tab* de taille `len`. La fonction devra avoir une complexité linéaire en `len`.
3. Écrire une fonction `bool dichotomie(int x, int tab[], int len)` qui teste l'appartenance d'un entier x à un tableau *tab* **supposé trié par ordre croissant** de taille `len`. La fonction devra avoir une complexité logarithmique en `len`.

2 Structures chaînées et pointeurs

Exercice 5

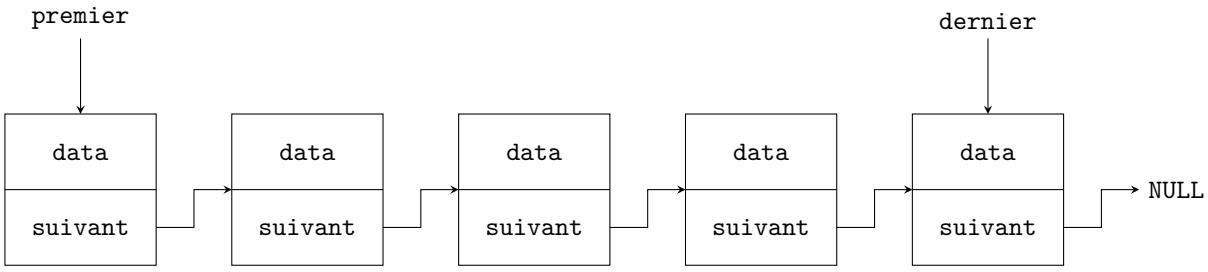
On souhaite créer une structure de liste simplement chaînée avec quelques opérations associées et l'utiliser pour implémenter une file. Un maillon de la liste chaînée contient une valeur numérique et un pointeur vers le prochain maillon. On utilise le type suivant :

```
struct Liste {
    int data;
    struct Liste* suivant;
};

typedef struct Liste liste;
```

1. Écrire une fonction `liste* creer_liste(int x)` qui crée un maillon simple contenant une valeur x donnée en argument et renvoie un pointeur vers ce maillon.
2. Écrire une fonction `liste* convertir_tab(int tab[], int len)` qui convertit un tableau en une liste chaînée et renvoie un pointeur vers le début de la liste.
3. Écrire une fonction `void afficher_liste(liste* lst)` qui affiche une liste chaînée.
4. Écrire une fonction `void liberer_liste(liste* lst)` qui libère l'espace mémoire occupé par une liste chaînée.

Pour implémenter une file, on utilise une liste simplement chaînée avec un pointeur vers le début de la liste et un pointeur vers la fin de la liste :



La file sera implémentée avec le type suivant :

```c
struct File {
    liste* premier;
    liste* dernier;
};

typedef struct File file;
```

5. Écrire une fonction `file* creer_vide(void)` qui crée une file vide.
6. Écrire une fonction `bool est_vide(file* f)` qui teste si une file est vide.
7. Écrire une fonction `void liberer_file(file* f)` qui libère l'espace mémoire occupé par une file.

Pour enfiler un élément, on l'ajoute **à la fin** de la liste chaînée. Pour défiler un élément, on le retire **au début** de la liste chaînée.

8. Écrire une fonction `void enfiler(file* f, int x)` qui enfile un élément x dans une file f.
9. Écrire une fonction `int defiler(file* f)` qui défile un élément d'une file f et renvoie sa valeur. On vérifiera (par `assert` par exemple) que la file est non vide.

3 Pour aller plus loin

Exercice 6

On définit le nombre d'**inversions** $inv(tab)$ d'un tableau $tab = [x_0, \ldots, x_{n-1}]$ comme le nombre de couples (i, j) tels que $0 \leqslant i < j < n$ et $x_i > x_j$.

1. Quelle est la valeur maximale de $inv(tab)$ pour un tableau de taille n ?
2. Écrire une fonction naïve `int inversions_naif(int tab[], int n)` qui calcule et renvoie $inv(tab)$. Déterminer sa complexité temporelle.
3. Déterminer un algorithme récursif de type « diviser pour régner » permettant de calculer $inv(tab)$ en complexité $\mathcal{O}(n \log n)$.

   Indication : il est intéressant de trier le tableau tab, en utilisant le principe du tri fusion, au fur et à mesure qu'on compte les inversions.
4. Écrire une fonction `int inversions_DPR(int tab[], int n)` qui applique cet algorithme.

Exercice 7

On souhaite créer une structure de tableau dynamique (au sens de taille dynamique, pas d'allocation dynamique) en C, comme celle de `list` en Python. Le principe est le suivant : on utilise un tableau dont une partie correspond aux valeurs effectivement présente, et le reste est un tampon inutilisé tant qu'on n'essaie pas de rajouter des éléments. Le type sera le suivant :

```c
struct Tab_dyna {
    int taille;
    int capacite;
    int* data;
};

typedef struct Tab_dyna tab_dyna;
```

– le champ `taille` correspond au nombre de valeurs effectivement présentes ;

- le champ `capacite` correspond à la capacité du tableau, c'est-à-dire l'espace mémoire occupé par le tableau `data` ;
- le champ `data` est un pointeur vers le tableau contenant les données.

1. Écrire une fonction `tab_dyna* creer_tab_dyna(void)` qui crée un tableau dynamique de taille 1.
2. Écrire une fonction `void liberer_tab_dyna(tab_dyna* td)` qui libère l'espace mémoire occupé par un tableau dynamique.
3. Écrire une fonction `int evaluer(tab_dyna* td, int i)` qui récupère la valeur à l'indice i. On vérifiera que l'entier i est un indice valide.
4. Écrire une fonction `void modifier(tab_dyna* td, int i, int x)` qui modifie la valeur à l'indice i en lui donnant la valeur x. On vérifiera que l'entier i est un indice valide.

On peut être amené à modifier la capacité du tableau, par exemple lorsqu'on doit ajouter un élément dans un tableau où la taille est égale à la capacité. Lorsque c'est le cas, on crée un nouveau tableau avec une nouvelle capacité, on recopie les valeurs de l'ancien tableau, qu'on libère, et on modifie le pointeur du tableau dynamique vers ce nouveau tableau. On peut alors ajouter le nouvel élément.

5. Écrire une fonction `void redimensionner(tab_dyna* td, int capa)` qui modifie la capacité d'un tableau dynamique. On vérifiera que la nouvelle capacité est supérieure à la taille du tableau dynamique.

Lors de l'ajout d'un élément, si le tableau est plein, on choisit de **doubler** la capacité avant l'ajout. Cela permet d'avoir une bonne complexité amortie.

6. Écrire une fonction `void ajouter(tab_dyna* td, int x)` qui ajoute un élément (à droite) dans un tableau dynamique.

Lorsqu'on retire un élément, pour éviter d'occuper inutilement de l'espace mémoire, on choisit de redimensionner le tableau si la capacité est plus du double de la taille. Dans ce cas on divise par deux la capacité du tableau.

7. Écrire une fonction `int retirer(tab_dyna* td)` qui retire l'élément de droite d'un tableau dynamique et renvoie sa valeur.
8. A-t-on la garantie d'une complexité amortie en $\mathcal{O}(1)$ en redimensionnant lorsque la capacité est plus du double de la taille ? Quel facteur devrait-on choisir pour améliorer cela ?