
















IN 301 Langage C – TD

Version du 08 décembre 2020

Pierre COUCHENEY – pierre.coucheney@uvsq.fr
Coline GIANFROTTA – coline.gianfrotta@ens.uvsq.fr
Maël GUIRAUD – mael.guiraud@uvsq.fr
Franck QUESSETTE – franck.quessette@uvsq.fr
Yann STROZECKI – yann.strozecki@uvsq.fr
Sandrine VIAL – sandrine.vial@uvsq.fr

Table des matières

1	BLOC 1 : Fichiers – Environnement de programmation	2
1.1	 Lecture et écriture dans les fichiers	2
1.2	 Environnement de programmation	4
2	BLOC 2 : Structures de données, tri et complexité	8
2.1	 Tri d'entiers	8
2.2	 Comparaison de tris	9
2.3	 Tri de chaînes de caractères	10
3	BLOC 3 : Chaînes de caractères	12
3.1	 Exercices de base	12
3.2	 Exercices de avancés	13
4	BLOC 4 : Arbres et expressions arithmétiques	14
4.1	 Travail préparatoire à faire à la maison avant le TD	14
4.2	 Manipulation d'arbres binaires pour les expressions arithmétiques	14
5	BLOC 5 et 6 : Chaînes de caractères et expressions arithmétiques	16
5.1	 Liste simplement chaînée	16
5.2	 Liste doublement chaînée	16
5.3	 Tokens	16
6	BLOC 7 : Petits exos 1/2	18
6.1	 Petits exercices 1/2	18
7	BLOC 8 : Petits exos 2/2	19
7.1	 Petits exercices 2/2	19
A	 Révisions du L1	21

Durant ce semestre vous travaillerez sous l'environnement Linux de la machine virtuelle (ou mac OS) que ce soit pour les TDs ou pour le projet.

Dans cet environnement, vous utiliserez le terminal afin de vous déplacer dans l'arborescence des fichiers, d'éditer, compiler, débbugger votre programme, et de mettre vos fichiers sous contrôle de version sur un répertoire distant.

Pour cela, vous utiliserez les programmes suivants :

- **geany** pour l'édition,
- **gcc** pour la compilation,
- **gdb** pour le débbugage,
- **git** pour le contrôle de version.

Biblio

- **Le langage C, 2ème édition** – Brian W. KERNIGHAN, Dennis M. RITCHIE
- **Programmer en langage C - 5ème édition** – Claude DELANNOY

Plan du Cours

Dans toute la suite les sections ou exercices sont notés \mathcal{P} résentiel ou \mathcal{D} istanciel.

BLOC 1 : Fichiers

- Environnement de programmation, éditeur, compilateur, arborescence des fichiers, fenêtre shell, ligne de commande, commandes shell de base, commande de compilation.
- Entrées sorties dans les fichiers, sur l'écran, principe.
- Makefile.
- Tableaux statiques.

BLOC 2 : Compilation séparée malloc , free de base

Malloc, free, int *. Compilation séparée.

1 BLOC 1 : Fichiers – Environnement de programmation

1.1 P Lecture et écriture dans les fichiers

Dans toute cette section, comme dans la suite du poly, vous écrirez un **Makefile** permettant de compiler et d'exécuter les programmes.

P Exercice 1. Premier fichier .h

Créer un fichier `constantes.h` qui contient les lignes suivantes qui définissent trois constantes :

```
#define N 10000
#define MAX 1000000
#define NOMFIC "nombres.data"
```

P Exercice 2. Écriture dans un fichier

a. Écrire un programme `genere.c` qui génère N entiers positifs aléatoires compris entre zéro et `MAX`. Ces nombres seront écrits dans le fichier `NOMFIC`. Afin de pouvoir utiliser les constantes dans ce programme, ajouter au début du programme, la ligne :

```
#include "constantes.h"
```

L'ouverture, l'écriture et la fermeture du fichier se feront dans une fonction :

```
void ecrire ()
```

Dans le `main` l'appel se fera par :

```
ecrire ();
```

Formatez l'affichage des nombres avec `%6d` pour que le fichier généré soit plus lisible.

P Exercice 3. Lecture depuis un fichier

a. Écrire un programme `algos.c` qui déclare en variable globale un tableau de taille N :

```
int T[N];
```

et qui lit les nombres qui sont dans le fichier `NOMFIC` généré à l'exercice précédent et les stocke dans les cases du tableau de 0 à $N-1$.

Vous devez faire une fonction :

```
void lecture ()
```

qui lit dans le fichier `NOMFIC` fichier et remplit le tableau `T`. Vous devez comme dans l'exercice précédent écrire :

```
#include "constantes.h"
```

au début de votre programme.

b. Programmer une fonction qui écrit les nombres qui sont dans le tableau dans un fichier `nombres-verif.data`

c. Utiliser la commande `diff` pour vérifier que les fichiers sont identiques :

```
diff nombres.data nombres-verif.data
```

`diff` affiche les différences, donc si `diff` n'affiche rien c'est que les deux fichiers sont identiques.

d. Modifier le code en supposant que l'on ne connaît pas le nombre de valeurs dans le fichier `NOMFIC`. On peut alors utiliser la valeur de retour de `fscanf` pour savoir «quand s'arrêter». Tapez

```
man 3 fscanf
```

et lisez la section `valeur de retour`.

Le `Makefile` sera commun avec celui de l'exercice précédent.

\mathcal{P} Exercice 4. premier algo

a. Dans le programme `algos.c` écrire une fonction :

```
int recherche (int x)
```

qui renvoie 1 si le nombre `x` est dans le tableau et 0 sinon.

b. Dans la fonction `recherche` ajouter un compteur pour compter le nombre de comparaisons que fait la fonction. Afficher ce nombre de comparaisons avant le retour de la fonction.

c. Écrire une fonction `stat_recherche(int x)` qui fait comme la fonction `recherche` mais renvoie la valeur du compteur du nombre de comparaisons.

d. Dans la fonction `main` appeler dix mille fois la fonction `stat_recherche` en lui donnant en argument une valeur aléatoire entre zéro et $5 \times \text{MAX}$. Calculer le nombre comparaisons en moyenne. Ce nombre était-il prévisible ?

\mathcal{P} Exercice 5. Commande `wc`

Programmer l'équivalent de la commande du terminal `wc -c` qui donne le nombre de caractères d'un fichier texte passé en argument. Le programme sera appelé de la manière suivante :

```
./monwc fichier
```

1.2 Environnement de programmation

Dans un premier temps, vous allez apprendre les fonctionnalités de base du terminal. Les commandes que vous utiliserez sont :

```
man, ls, cd, pwd, mkdir, mv, rm, wc, cat
```

leur description peut être obtenue avec la commande `man commande`. Deux fonctionnalités du terminal qui s'avèrent très utiles à l'usage, et que vous vous efforcerez d'utiliser dès le début, sont la complétion automatique avec la touche tabulation, et le parcours de l'historique des commandes avec les flèches haut et bas.

Exercice 6. Terminal et compilation

a. Ouvrez un terminal. Dans quel répertoire vous trouvez-vous ? Que se passe-t'il si vous appuyer sur la touche `w` puis deux fois sur `TAB` dans le terminal ? Puis `wh` et deux fois sur `TAB` ? Puis `who` et une fois sur `TAB` ? Que fait la commande que vous venez de taper ? Essayer `whereis ls`.

b. Déplacez vous vers le répertoire `Bureau`. Revenez dans le répertoire initial, puis encore dans le répertoire `Bureau` sans taper complètement au clavier `Bureau`.

c. Affichez ce que contient le répertoire `Bureau`. Créez un répertoire `essai`, et vérifiez que celui-ci a bien été créé.

d. Déplacez vous dans le répertoire `essai`, puis éditez un fichier vide `fic1` avec `geany` en tapant la commande :

```
geany fic1 &
```

puis fermez `geany`. Que contient le répertoire maintenant ?

e. Copiez le fichier vers un fichier `fic2`. Changez le nom de `fic2` en `fic3`. Supprimez le fichier initial `fic1`.

f. Supprimez tous les fichiers et ajoutez le programme `debug.c` qui est sur `moodle`. Affichez le programme dans le terminal. Combien de lignes et de caractères contient-il ?

g. Nous allons maintenant compiler le programme. Taper la commande

```
gcc debug.c
```

Vérifiez qu'un fichier `a.out` a été créé. Exécutez le programme en tapant la commande `./a.out`. Manifestement ce programme contient un bug... mais vous ne le corrigerez pas pour l'instant.

h. Pour choisir un nom à votre exécutable, tapez la commande `gcc -o nom debug.c`. Vérifiez qu'un exécutable ayant le nom que vous avez choisi a bien été créé, et exécutez le.

i. Compilez en ajoutant les warnings de compilation avec la commande

```
gcc -o nom -Wall debug.c
```

Que constatez-vous ?

Notez bien que pour compiler un programme qui utilise la librairie mathématique, il faut ajouter l'option `-lm`. Au final, cela donne la commande suivante :

```
gcc -o nom -Wall debug.c -lm
```

D Exercice 7.

Organisation de vos dossiers Afin de stocker correctement vos fichiers vous devez créer les dossiers suivants :

- un dossier **IN301**
- dans ce dossier un dossier par TD, commencez donc par un dossier **TD 01 - Environnement de programmation** correspondant au TD de cette section
- Vous pouvez également créer un dossier par séance de TD avec la commande :

```
for x in $(seq 1 12); do mkdir "td_$x"; done
```

- Vous pouvez créer un dossier **Notes de Cours** et un dossier **Projet**.
- Pour cette séance vous travaillerez dans le dossier **TD 01 - Environnement de programmation**.
- Vous sauvegarderez tous les fichiers de ce TD dans ce dossier.

D Exercice 8.

Utilisation de commandes shell

- a. Écrire un programme C qui génère dix milles nombres aléatoires entre 0 et 1 000. Utiliser la commande :

```
man 3 rand
```

pour voir la syntaxe des fonctions **rand()** et **srand()**. Dans votre programme pour afficher à l'écran la syntaxe est :

```
int a;
a = rand();
printf("%d\n",a);
```

- b. Utiliser **srand()** avec comme argument **getpid()** pour changer l'initialisation des nombres aléatoires. Faites **man 2 getpid()**

- c. Quelle est la commande de compilation correcte ?
- d. Utiliser une redirection pour rediriger le résultat dans un fichier **nombres.data**.
- e. Utiliser la commande **wc** pour compter le nombre de nombres générés
- f. Utiliser la commande **sort -n** (faites **man sort** pour voir la doc) pour trier le fichier et rediriger le tri vers un autre fichier **nombres_tries.data**
- g. Utiliser la commande **uniq** pour supprimer les doublons et mettre le résultat dans **nombres_uniq.data**. Combien y a-t-il de nombres différents ?
- h. mettre toutes ces commandes dans un fichier, le rendre exécutable et exécutez le.
- i. Enchaîner toutes les commandes en une seule ligne pour compter le nombre de nombres différents.

D Exercice 9. Contrôle de version : git

Vous allez utiliser un gestionnaire de version qui s'appelle **git**, et un dépôt distant qui s'appelle **github**. Il y a (au moins) 2 intérêts à cela par rapport à une sauvegarde classique : d'une part vous conservez l'historique de toutes les versions et pouvez revenir en arrière en cas d'erreur ou de suppression inopinée de votre travail, et d'autre part, vous faites des sauvegardes sur un dépôt distant qui vous prémunit des accidents que pourrait toucher votre machine personnelle. L'objectif de cet exercice est de vous familiariser avec l'utilisation de cet outil que vous utiliserez tout au long du semestre (et au-delà je l'espère!).

- a. Ouvrez un navigateur et allez sur la page web : <https://github.com/>
- b. Créez un compte en mémorisant bien votre nom d'utilisateur et votre mot de passe.

c. Une fois que vous êtes connecté, créez un dossier qui s’appelle **IN301** en cliquant sur la croix en haut à droite de l’écran (create new repository). Votre dépôt distant sous gestionnaire de version git est créé, vous allez maintenant le récupérer sur votre machine.

d. Dans votre terminal, déplacez vous à l’endroit où vous voulez ajouter le dossier **IN301**. Créez une copie locale en tapant la commande suivante dans le terminal :

```
git clone https://github.com/moi/IN301
```

et en remplaçant **moi** par votre nom d’utilisateur. Un répertoire **IN301** a normalement été créé, ce que vous pouvez vérifier en tapant la commande **ls**.

e. Déplacez vous dans le répertoire **IN301**. Créez le dossier **td0** (commande **mkdir**)¹. Allez dans le dossier **td0** et créez le fichier vide **essai** avec la commande **touch essai**. Nous allons maintenant ajouter ce fichier au répertoire distant.

f. Pour mettre un nouvel élément en contrôle de version, il faut utiliser la commande **git add fichier**, donc, ici, **git add essai**. Pour valider cet ajout, utiliser la commande **git commit essai**, ou la commande **git commit -a** qui validera toutes les modifications faites sur des fichiers qui sont sous contrôle de version. Une fenêtre s’ouvre pour que vous renseigniez une description de la modification. Ecrivez, par exemple, “ajout fichier td0/essai”, cliquez ensuite sur **CTRL+x**, puis sur **o** (pour OUI) et **ENTREE**. Pour propager cela au répertoire distant, il reste à taper la commande **git push**. Vérifiez ensuite sur votre compte **github** que le dossier a bien été ajouté.

g. Ajoutez le fichier **debug.c** que vous avez manipulé dans l’exercice précédent dans le répertoire **td0**. En suivant les mêmes instructions que pour **td0** faites en sorte de mettre ce fichier sous gestion de version et de l’ajouter dans **github**.

h. Une fois que cela a bien été réalisé, supprimez le répertoire **IN301** de votre machine (commande **rm -rf IN301**) et vérifiez que cela a bien fonctionné. Faites maintenant un clône de votre dépôt distant **github**, et vérifiez que vous avez bien récupéré le dossier **td0** et qu’il contient le fichier **debug.c**.

i. Ajoutez du texte dans le fichier **essai** et sauvegardez. Testez alors la commande **git status**. Commitez le changement. Que renvoie la commande **git status**? Poussez votre changement sur **github** et testez de nouveau **git status**.

Dorénavant, pour ceux qui travaillent avec une machine en prêt, vous pourrez récupérer vos travaux en début de séance en clonant votre répertoire distant.

A tout moment, vous pouvez “commiter” vos changements (ne pas oublier de faire **git add** pour les dossiers et fichiers nouvellement créés), et les propager avec la commande **git push**. Localement vous pouvez vérifier si vos fichiers sont bien enregistré avec la commande **git status**.

Enfin, notez qu’il ne faut commiter que les fichiers textes (typiquement les programmes terminant par l’extension **.c**) ainsi que les répertoires qui les contiennent, et non les exécutables.

D Exercice 10. Compilation et debug

Récupérez le fichier **debug.c** de l’exercice précédent. Nous allons utiliser le logiciel **gdb** afin de déboguer ce programme.

a. Sans éditer le programme **debug.c**, compilez et exécutez le (voir exercice 1).

Les commandes de base du debugger **gdb** sont

```
run, quit, break, bt, print, step, next
```

b. Pour exécuter le programme dans le debugger, lancer la commande **gdb ./progDebug** dans le terminal, puis la commande **run**. Que se passe-t’il ?

c. Essayez d’ajouter un point d’arrêt à la ligne 11 (commande **break 11**). Cela ne doit pas être possible car l’exécutable ne sait pas faire référence au fichier source. Il faut pour cela ajouter l’option **-g** à la compilation. Pour cela quittez **gdb** (commande **quit**) et recompilez en ajoutant l’option.

1. Dorénavant, pour la feuille de **td i**, vous créez un dossier **tdi** à cet emplacement

- d. Retournez dans `gdb` et ajoutez un point d'arrêt à la ligne 11 puis lancez l'exécution du programme. Afficher la pile des appels (commande `bt`), et la valeur des variables dans la fonction courante. Continuez l'exécution du programme pas à pas en affichant les valeurs des variables régulièrement.
- e. Corrigez la fonction `factorielle`, puis faites de même pour les fonctions `somme` et `maximum` en vous aidant si nécessaire de `gdb`.

2 BLOC 2 : Structures de données, tri et complexité

le but de cette partie est de comprendre les mécanismes de la compilation séparée et d'effectuer des comparaisons de temps de calcul de différents tris.

2.1 P Tri d'entiers

Dans cette première partie, l'ensemble des fichiers nécessaires vous est fourni sur moodle uvsq dans un fichier IN301-303_TD_TRI_INT.zip. Récupérez ce fichier et unzippez le et dans le terminal allez dans le dossier IN301-303_TD_TRI_INT

P Exercice 11. Compréhension de l'ensemble des fichiers

- Listez les fichiers du dossier IN301-303_TD_TRI_INT et ouvrez le Makefile dans un éditeur.
- Faites un schéma présentant tous les fichiers et les relations entre fichiers. les relations sont de deux types : “est utilisé par” “est créé à partir de”.
- Indiquez la cible du Makefile qui correspond à chacune des relations entre fichiers.

P Exercice 12. Bibliothèque de manipulation de tableau

Le but de cet exercice est de faire une librairie de manipulation de tableaux d'entiers.

a. Dans le fichier `tabint.h` est défini le type :

```
struct tabint {
    int N; // Taille du tableau
    int *val; // Pointeur vers le tableau
};
typedef struct tabint TABINT;
```

Expliquez cette définition.

b. Programmez dans le fichier `tabint.c` une fonction qui alloue un tableau :

```
TABINT alloue_tabint (int N)
```

c. Programmez dans le fichier `tabint.c` une fonction qui alloue et remplit un tableau de N cases avec les valeurs aléatoires comprises dans l'intervalle $[0..K]$:

```
TABINT gen_alea_tabint (int N, int K)
```

d. Programmez dans le fichier `tabint.c` une fonction qui désalloue un tableau :

```
void desalloue_tabint (TABINT T)
```

e. Programmez dans le fichier `tabint.c` une fonction qui affiche un tableau :

```
void aff_tabint (TABINT T)
```

f. Programmez dans le fichier `tabint.c` une fonction `verif_si_tableau_croissant(TABINT T)` qui renvoie 1 si le tableau en argument est en ordre croissant et 0 sinon.

Testez cette fonction en modifiant provisoirement la fonction `gen_alea_tabint` pour initialiser le tableau avec des valeurs croissantes.

P Exercice 13. Tri à bulle et tri fusion

a. Programmez dans le fichier `tri_bulle.c` une fonction qui implémente le tri à bulle :

```
??? tri_bulle_tabint (TABINT T)
```

A-t-on besoin de retourner le tableau pour qu'il soit modifié ? Vérifiez que votre tri est correct en affichant les valeurs du tableau avant et après le tri.

b. Testez votre code avec un main de la forme :

```
printf("Tri bulle avec N = %d\n",N);
printf(" Trié avant : %d\n",verif_si_tableau_croissant(T));
aff_tabint(T);
tri_bulle(T);
aff_tabint(T);
printf(" Trié après : %d\n",verif_si_tableau_croissant(T));
```

c. Programmez dans le fichier `tri_fusion.c` une fonction qui implémente le tri fusion :

```
void tri_fusion_tabint (TABINT T)
```

d. Modifiez le main des deux tris pour prendre un argument qui sera la taille du tableau.

e. Modifiez le `Makefile` pour comparer les temps d'exécution des deux tris avec la commande `time` :

```
run: tri_bulle tri_fusion
time ./tri_bulle 10000
time ./tri_fusion 10000
time ./tri_bulle 100000
time ./tri_fusion 100000
time ./tri_bulle 200000
time ./tri_fusion 200000
```

Commentez les affichages des main pour plus de lisibilité. Modifiez les tailles du tableau pour que les temps d'exécution soient ni trop longs trop courts.

f. Quand la taille du tableau est multipliée par 10 par combien est multiplié le temps d'exécution de chacun des tris.

2.2 D Comparaison de tris

D Exercice 14. Analyse statistique

a. Dans le fichier `stat.h` est défini une structure permettant de stocker des stats :

```
struct stat {
    float nb_moy_comp;
    float nb_moy_ech;
};
```

b. Programmer dans le fichier `tri_bulle.c` une fonction qui prend en argument la taille du tableau `N` et un nombre de répétitions `NbFois` et qui :

- génère et trie `NbFois` fois un tableau de taille `N`
- renvoie une `struct stat` qui contient le nombre moyen de comparaisons et d'échanges effectués :

```
struct stat stat_moy (int N, int NbFois)
```

c. Écrire un programme qui fait varier `N` entre 10 et 10^3 en le multipliant par 1.2, avec `NbFois` fixé à 10^4 et qui calcule la moyenne du nombre de comparaisons et d'échanges.

Ce programme doit écrire les résultats dans un fichier `test_tri_bulle.data` en mettant sur chaque ligne, la valeur de `N` suivi de la moyenne du nombre de comparaisons et de la valeur du nombre d'échanges.

d. Utilisez `gnuplot` pour afficher les courbes et générer un pdf. Un fichier de commandes `gnuplot` vous est fourni : `stat.gplt`. Pour lancer `gnuplot` sur ce fichier de commandes, il suffit de taper :

```
gnuplot stat.gplt
```

D Exercice 15. Comparaisons statistiques de tris

- Programmer la même chose que pour le tri à bulle pour faire des stats dans un fichier `test_tri_fusion.data`
- Modifier le fichier `stat.gplt` pour afficher les stats du tri à bulle et du tri fusion.

2.3 **D** Tri de chaînes de caractères

D Exercice 16. Trier des mots 1 : Quicksort

Dans cet exercice, nous allons voir comment trier des tableaux de chaînes de caractères selon l'ordre alphabétique.

- Récupérer le programme source dans l'archive compressée `tri_mot.zip` et le décompresser.
- Éditer les fichiers `tableau.h` et `tableau.c` qui contiennent la structure à trier et des fonctions de manipulation qui sont fournies. Par défaut, le programme lit le fichier texte `romeoetjuliette` pour initialiser le tableau. Ce fichier peut être modifié dans le fichier `Makefile`.
- Éditer le fichier `Makefile` et dessiner les liens entre les fichiers.
- Dans le fichier `tri.c`, implémenter le tri rapide en utilisant la fonction `strcmp` de la librairie `string.h`.

D Exercice 17. Trier des mots 2 : Tri par base

Nous allons maintenant implémenter le tri par base qui est une variante du tri par dénombrement adapté au tri de chaînes de caractères.

Le principe de ce tri est le suivant : on trie l'ensemble des mots selon leur dernier caractère en utilisant un tri par dénombrement. On fait de même sur l'avant dernier caractère et ainsi de suite jusqu'au premier caractère. Attention, à chaque fois, le tri utilisé doit être un tri stable.

- Rappeler ce qu'est un tri stable.
- Exécuter alors les étapes du tri par base sur les chaînes suivantes : `ca, cb, aa, ba, bc, bb, ac, aa, cb`.
- Montrer que cet algorithme est correct.
- Rappeler le principe du tri par dénombrement lorsque l'on trie des entiers entre 0 et K . Quelle est sa complexité? En déduire la complexité du tri par base.
- Afin d'adapter le tri par dénombrement pour trier des caractères, suivez les indications suivantes (on suppose que l'on est en train de trier selon le i ème caractère) :
 - Indication 1 : Créer un tableau qui contient les fréquences d'apparition des 26 lettres de l'alphabet en position i dans l'ensemble des mots.
 - Indication 2 : Si le 'a' apparaît n_a fois (n_a est donc la fréquence du 'a'), et le 'b' n_b fois, où doit-on "ranger" tous les mots ayant un 'b' en i ème position et dans quel ordre.
 - Même question pour n'importe quelle lettre.
 - Indication 3 : En supposant que l'on a déjà trié les mots selon les positions de `taille` (taille max des chaînes de caractère) jusqu'à $i + 1$, écrire l'algo pour trier jusqu'à la position i .
- Écrire la fonction

```
int indice(char c)
```

qui convertit des caractères en indice dans le fichier `tri.c`.

- Écrire la fonction

```
void tri_base_indice(Tableau t, int i)
```

qui implémente un tri par dénombrement stable des mots du tableau `t` selon le caractère `i` de chaque mot. Pour implémenter une version stable de ce tri, vous devrez utiliser un tableau auxiliaire (on dit que le tri n'est pas en place).

- h. Comparer les temps d'exécution des deux algorithmes de tri (quicksort et tri par base).

D Exercice 18. Trier des mots 3 : MSD

Le tri par base de l'exercice précédent est en fait la variante LSD (Least Significant Digit !). Il existe une autre variante appelée MSD (Most Significant Digit) qui peut être vue comme une adaptation du tri rapide (ou plus précisément du tri par paquets). Celle-ci consiste à trier les mots selon le premier caractère puis à trier récursivement l'ensemble des mots qui commencent par le plus petit caractère (c'est-à-dire le caractère `0`), puis le deuxième plus petit (le caractère `'a'`), etc...

- a. Programmer cet algorithme et comparer son temps d'exécution aux deux autres.
- b. Écrire une fonction qui supprime les doublons.

3 BLOC 3 : Chaînes de caractères

3.1 P Exercices de base

En C, il est possible d'initialiser un tableau de caractères par une chaîne de caractère de la manière suivante :

```
char *c = "salut"
```

Le tableau se termine alors par le caractère de fin de chaîne `'\0'`. Sur l'exemple, il s'agit donc un tableau de taille six. Cela permet l'usage de fonctions adaptées aux chaînes de caractères, fournies dans la librairie `string.h`, telle que la fonction `strlen()` qui donne la taille du tableau sans le caractère de fin de chaîne.

Pour avoir la liste des différentes fonctions de la librairie `string.h` taper :

```
man 3 string
```

Pour avoir le manuel d'une fonction en particulier, taper :

```
man 3 strlen
```

P Exercice 19. Arguments de la ligne de commande

Une utilisation des chaînes de caractères est le passage d'argument au programme dans la ligne de commande. Pour cela, le prototype de la fonction `main()` doit être adapté de la manière suivante :

```
int main(int argc, char** argv)
```

où `argv` est un tableau de chaînes de caractères, et `argc` est la taille du tableau.

a. Tester et comprendre le programme `testarg.c` suivant :

```
#include <stdio.h>
#include <string.h>
int main( int argc, char** argv){
    int i;
    printf("argc = %d\n", argc);
    for(i=0 ; i<argc ; i++)
        printf("%s %d\n", argv[i],strlen(argv[i]));
    return 0;
}
```

en l'appelant avec une ligne de commandes qui contient n'importe quel nombre d'arguments, par exemple :

```
./testarg 6 toto salut2016 ?!;;
```

b. Pourquoi la variable `argv` est-elle de type `char**` ?

c. En utilisant la fonction de conversion `atof()` (voir le manuel), modifier le programme pour qu'il affiche la somme des nombres passés en arguments. Par exemple l'exécution

```
./testarg 6.2 -8 2.0
```

doit afficher 0.2.

d. Que se passe-t-il si les arguments ne sont pas des nombres flottants ?

e. En utilisant la fonction `isdigit()` (voir le manuel), vérifier au préalable que les arguments passés sont bien des nombres flottants éventuellement négatifs.

P Exercice 20. Tableau et liste de caractères

L'objectif est d'écrire des fonctions simples de manipulation des chaînes de caractères. Nous travaillerons avec deux chaînes de caractères qui seront passées en argument de l'appel au programme de la manière suivante :

```
./programme chaine1 chaine2
```

Vous devez avoir dans la **Makefile** une cible **test** qui appelle plusieurs fois le programme avec à chaque fois des arguments pertinents pour tester ce programme.

a. Écrire une fonction **itérative** qui calcule la longueur d’une chaîne de caractère. Afficher la longueur de chacune des chaînes passées en argument.

b. Écrire une fonction **récurive** qui calcule la longueur d’une chaîne de caractère. Afficher la longueur de chacune des chaînes passées en argument.

c. Écrire une fonction **itérative** qui calcule la différence entre les deux chaînes comme le fait **strcmp()**. Comparer les deux chaînes passées en argument.

d. Écrire une fonction **récurive** qui calcule la différence entre les deux chaînes comme le fait **strcmp()**. Comparer les deux chaînes passées en argument.

e. Écrire une fonction qui renvoie vrai si les chaînes sont miroir l’une de l’autre. Comparer les deux chaînes passées en argument.

f. Écrire une fonction qui renvoie vrai si la chaîne 2 est une sous-chaîne de la chaîne 1. Tester sur les deux chaînes passées en argument.

g. Écrire une fonction qui renvoie 1 si une chaîne est un palindrome et 0 sinon. Tester sur les deux chaînes passées en argument.

3.2 D Exercices de avancés

D Exercice 21. Manipulations de chaînes de caractères

Vous devez écrire et tester les fonctions suivantes :

- **void majuscule(char *s)** qui met en majuscule les caractères minuscules de **s**.
- **void minuscule(char *s)** qui met en minuscule les caractères majuscules de **s**.
- **char *duplique(char *s)** qui duplique **s** et renvoie la nouvelle chaîne.
- **char *cherche(char *s, char c)** qui renvoi un pointeur vers la première occurrence de **c** dans **s** et renvoie **NULL** si **c** est absent.
- **char *concatene(char *s1, char *s2)** qui renvoie une nouvelle chaîne qui contient la concaténation de **s1** et **s2**.
- **int difference(char *s1, char *s2)** qui renvoie la différence de code ASCII entre les premiers caractères de **s1** et **s2** qui sont différents.
- **char *souschaine(char *s1, char *s2)** qui renvoie un pointeur vers la première occurrence de **s2** dans **s1**.¹

D Exercice 22. Matrices

Vous devez écrire une bibliothèque (**matrice.c**, **matrice.h**) de manipulation de matrices d’entier. Vous devez avoir une fonction d’allocation de mémoire de matrice, de remplissage aléatoire de matrice, de désallocation de mémoire, d’addition, multiplication.

4 BLOC 4 : Arbres et expressions arithmétiques

4.1 D Travail préparatoire à faire à la maison avant le TD

D Exercice 23. mise en place du canevas de base

Avant d'arriver en TD vous devez :

1. Créer un dossier TD-EA dans votre dossier de travail habituel.
2. Dans le dossier TD-EA créer les quatre fichiers suivants : `Makefile`, `ea.h`, `ea.c`, `principal.c`.

Le fichier `ea.h` doit contenir :

- la définition d'un type `struct noeud` contenant :
 - un champ `op_ou_val` de type `int`;
 - un champ `op` de type `char`;
 - un champ `val` de type `float`;
 - deux champs `opg` et `opd` de type `struct noeud *`.
- l'alias de type : `typedef struct noeud* EA;`
- les signatures des trois fonctions suivantes :
 - `EA ea_creer_valeur (float val)`
 - `EA ea_creer_operation (char op, EA opg, EA opd)`
 - `float ea_evaluer (EA e)`

Le fichier `ea.c` doit contenir les bons `#include` et les trois fonctions dont les signatures sont dans `ea.h`. Le corps des fonctions est pour le moment vides avec éventuellement un `return` pour compiler sans warning.

Le fichier `principal.c` doit contenir les bons `#include`, une fonction `main` qui doit contenir le code suivant :

```
EA e,e1,e2;
e1 = ea_creer_valeur(12.3);
e2 = ea_creer_valeur(4.56);
e = ea_creer_operation('+',e1,e2);
printf("%f\n",ea_evaluer(e));
```

Dans le fichier `Makefile`, il doit y avoir les cibles pour :

- compiler `ea.c` en `ea.o`;
- compiler `principal.c` en `principal.o`;
- faire l'édition de liens de `ea.o` et `principal.o` pour créer l'exécutable `principal`;
- effacer les fichiers produits par la compilation et l'édition de liens;
- exécuter.

Vous devez tester pour vérifier que tout fonctionne correctement.

4.2 P Manipulation d'arbres binaires pour les expressions arithmétiques

P Exercice 24. Dessins d'arbres

Dessinez les arbres binaires correspondant aux expressions arithmétiques suivantes :

- $12.3 + 4.56$
- $(1 + 2)$
- $1 + (2 \times 3)$
- $1 + 2 \times 3$, en supposant les priorités habituelles des opérateurs
- $(1 + 2) \times (5 - (4 - 3))$
- $(1 + 2) \times (((3 - 4) \times (5 + (6 + 7))) + (8 - 9))$

P Exercice 25. Codes des EA

Pour chacune des expressions arithmétiques de l'exercice précédent :

- dans le `main` taper le code permettant de créer en mémoire les EA;
- compiler;
- donner la représentation mémoire sous forme de diagrammes avec des rectangles et des flèches.

 \mathcal{P} Exercice 26. `ea_creer_valeur` (float val))

- Donnez en français les différentes étapes que doit contenir cette fonction (allocation, affectation, retour).
- Quel est le type de la valeur de retour de cette fonction.
- Programmez la fonction, compiler.

 \mathcal{P} Exercice 27. `ea_creer_operation` (char op, EA opg, EA opd))

- Donnez en français les différentes étapes que doit contenir cette fonction (allocation, affectation, retour).
- Quel est le type de la valeur de retour de cette fonction.
- Programmez la fonction, compiler.

 \mathcal{P} Exercice 28. `ea_evaluer` (EA e))

On considère la dernière expression arithmétique de l'exercice 24 :

- dessinez son arbre ;
- expliquez en quoi cette expression peut-être évaluée récursivement ;
- donnez la suite des appels à la fonction `ea_evaluer` et leur valeur de retour ;
- combien y a-t-il d'appels à la fonction `ea_evaluer` ?
- quel lien y a-t-il entre le nombre d'appels et le nombre d'opérateurs ?
- Programmez la fonction, compiler, exécuter.

5 BLOC 5 et 6 : Chaînes de caractères et expressions arithmétiques

5.1 P Liste simplement chaînée

D Exercice 29. Mise en place du canevas de base

Avant d'arriver en TD vous devez :

1. Dans le dossier TD-EA utilisé au BLOC 4, créer les deux fichiers suivants : `token.h`, `token.c`.

Le fichier `token.h` doit contenir :

- la définition d'un type `struct token` contenant :
 - un champ `val` de type `float`;
 - un champ `suiv` de type `struct token*`.
- l'alias de type : `typedef struct token* TOKEN`;
- les signatures des deux fonctions suivantes :
 - `TOKEN token_ajouter_fin_liste (TOKEN t, float val)`
 - `void token_afficher (TOKEN t)`

Le fichier `token.c` doit contenir les bons `#include` et les deux fonctions dont les signatures sont dans `token.h`. Le corps des fonctions est pour le moment vides avec éventuellement un `return` pour compiler sans warning.

Le fichier `principal.c` doit contenir les bons `#include`, une fonction `main` qui permet de tester ces fonctions

Modifier le fichier `Makefile`, pour compiler `token.c` et faites toutes les modifications nécessaires.

Vous devez tester pour vérifier que tout fonctionne correctement.

P Exercice 30. Programmation

Avant de programmer les fonctions `token_ajouter_fin_liste` et `token_afficher`, on suppose que l'on a dans le `main` :

```
TOKEN t = NULL;
t = token_ajouter_fin_liste (t, 2.3);
t = token_ajouter_fin_liste (t, 8);
t = token_ajouter_fin_liste (t, 5.12);
token_afficher(t);
```

L'exécution doit afficher : 2.30 8.00 5.12. Donnez la représentation en mémoire après chaque instruction. Programmez les deux fonctions et le `main`.

5.2 P Liste doublement chaînée

P Exercice 31.

Ajouter à la structure `struct token` un champ `prec` de type `struct token*` qui doit pointer vers le `struct token` précédent.

Donnez la représentation en mémoire après chacune des instructions du code de l'exercice précédent.

Modifier la fonction `token_ajouter_fin_liste` pour que le champ `prec` contienne la bonne valeur.

5.3 P Tokens

Un token est un élément lexical qui a du sens du point de vue du texte que l'on considère.

Par exemple l'expression arithmétique : `"(2* (3+4))"` comprend la liste des tokens suivants :

(2 * (3 + 4))

Il y a quatre types de tokens :

- Parenthèse ouvrante : (
- Parenthèse fermante :)

- Opérateur : `+`, `*`
- Valeur : `2`, `3`, `4`

\mathcal{P} Exercice 32.

Ajoutez au type `struct token` les champs :

- un champ `type_token` de type `int` ;
- un champ `op` de type `char` ;.

\mathcal{P} Exercice 33. Liste de tokens

Donnez la liste des tokens correspondant aux expressions arithmétiques suivantes :

- $12.3 + 4.56$
- $(1 + 2)$
- $1 + (2 \times 3)$
- $1 + 2 \times 3$, en supposant les priorités habituelles des opérateurs

Donnez la représentation en mémoire de cette liste de token en utilisant le type `TOKEN`.

\mathcal{P} Exercice 34. Chaîne vers float

Rappelez les valeurs des codes ascii des caractères représentant les chiffres de 0 à 9.

On suppose que l'on a une chaîne de caractère commençant par un chiffre et contenant la représentation d'un float, comment calculer la valeur du float contenu dans cette chaîne ?

\mathcal{P} Exercice 35. Ajout de tokens de types différents

Modifiez la signature et le code de la fonction `token_ajouter_fin_liste` pour pouvoir ajouter les différents types de tokens.

\mathcal{P} Exercice 36. `TOKEN token_creer_liste (char *s)`

Le but est de créer une liste de tokens à partir d'une chaîne de caractère

- Donnez en français les différentes étapes permettant de transformer la chaîne de caractère `"(12.3 + 3.54)"` dans une liste de token.
- Idem pour `"(2* (3+4))"`.
- Idem pour une chaîne de la forme `"((s1) * (s2))"` où `s1` et `s2` sont des chaînes d'expressions arithmétiques.
- Programmez la fonction, compiler.

\mathcal{P} Exercice 37. `EA token_to_EA (token t)`

- Expliquer comment construire l'arbre binaire `EA` à partir d'une liste de token comme :

`(2 * (3 + 4))`

- Donner un algo pour construire l'arbre binaire `EA` à partir d'une liste de token.

6 BLOC 7 : Petits exos 1/2

6.1 P Petits exercices 1/2

Pour chacun des exercices vous devez faire un fichier `exo_xx.c` et un fichier `exo_xx.h`. Pour l'ensemble des exercices de ce bloc, vous devez avoir un fichier `principal.c` qui contient le `main` permettant de tester les fonctions demandées. Vous devez bien sûr faire un `Makefile`.

Dans tous ces exercices vous veillerez à ce que les affichages soient propres et lisibles.

D Exercice 38. Mise en place du canevas

Mettez en place le canevas demandé ci-dessus avec tous les fichiers créés, avec les fonctions vides.

P Exercice 39. Affichage

- Écrire une fonction `affn` qui prend en argument un entier n et affiche sur une ligne avec un retour à la ligne à la fin les entiers de $-n$ à n .
- Écrire une fonction réursive `affnrec` qui prend en argument un entier n et affiche sur une ligne avec un retour à la ligne à la fin les entiers de $-n$ à n .

P Exercice 40. `sizeof`

Écrire une fonction `afftailletype` qui affiche de façon claire, le nombre d'octets des différents types suivants :

- `char`, `unsigned char`
- `int`, `unsigned int`
- `short`, `short int`
- `long`, `long int`, `long long`, `long long int`
- que se passe-t-il pour les types : `short short int` et `long long long int` ?

Rajoutez les types

- `float`, `double`, `long double`
- que se passe-t-il pour les types : `long float` et `long long double` ?

P Exercice 41. Complément à deux

Écrire une fonction `compadeux` :

- déclarer une variable de type `int` et donnez lui la valeur `-1` ;
- afficher la taille d'un `int` ;
- afficher cette valeur avec " `%d`" ;
- afficher cette valeur avec " `%u`".

À quoi correspond la valeur affichée avec " `%u`" ?

À la suite du code précédent :

- déclarer une variable de type `short` et donnez lui la valeur `-1` ;
- afficher la taille d'un `short` ;
- afficher cette valeur avec " `%hd`" ;
- afficher cette valeur avec " `%hu`".

À quoi correspond la valeur affichée avec " `%hu`" ?

7 BLOC 8 : Petits exos 2/2

7.1 P Petits exercices 2/2

P Exercice 42. Complément à deux 2

Écrire une fonction `compadeux2` :

- Faites une boucle sur un entier `i` qui affiche proprement et lisiblement les entiers de 1 à 300 en retournant à la ligne tous les 10.
- Modifier la boucle pour ne plus afficher `i` mais un `char c`; avec `"%03d %02x"` en commençant à 0 et que vous augmenterez de 1 à chaque tour de boucle. La boucle doit toujours se faire sur `i`.
- Dupliquer la boucle pour afficher un `unsigned char c`; avec `"%03u %02x"`.

Expliquez l’affichage qui se produit.

P Exercice 43. Décalage

Écrire une fonction `decalage` :

Tous les affichages se font avec `printf("%03u %02x\n",x,x);`

- Déclarer un `unsigned char x` et initialisez le à 1, afficher sa valeur.
- Ajoutez l’instruction : `x = x<<1` et afficher `x`. Expliquez ?
- Ajoutez l’instruction : `x = x<<5` et afficher `x`. Expliquez ?
- Ajoutez l’instruction : `x = x>>3` et afficher `x`. Expliquez ?
- Que se passe-t-il si on décale 1 à droite ?
- Que se passe-t-il si on décale un 1 vers la gauche de plus que la taille de la variable ?
- Affectez -1 à `x` et afficher `x`. Expliquez ?
- Ajoutez l’instruction : `x = x<<1` et afficher `x`. Expliquez ?
- Ajoutez l’instruction : `x = x>>1` et afficher `x`. Expliquez ?

P Exercice 44. ET OU NON bit à bit

Écrire une fonction `bitabit` :

Tous les affichages se font avec `printf("%03u %02x\n",x,x);`

- Déclarez deux variables `unsigned char` que vous initialisez à 13 et à 3.
- Mettez le résultat de l’opérateur binaire `&` entre ces deux variable dans une troisième et affichez la.
- Mettez le résultat de l’opérateur binaire `|` entre ces deux variable dans une troisième et affichez la.
- Mettez le résultat de l’opérateur binaire `^` entre ces deux variable dans une troisième et affichez la.
- Mettez le résultat de l’opérateur unaire `~` d’une deux variable dans une troisième et affichez la.

P Exercice 45. Bits d’un char

Écrire une fonction `affbitschar` qui prend en argument un `unsigned char` et affiche ses bits.

P Exercice 46. Bits d’un float

Écrire une fonction `affbitsfloat` qui prend en argument un `float` et affiche ses bits.

P Exercice 47. Précision des float

- Avant de tester ce code, que devrait-il afficher ?

```
float a,b,c,d;
a = 1e-8;
b = a + 1.0;
c = b - 1.0;
d = (a + 1.0) - 1.0;
if (c==a) printf("ok c=a\n"); else printf("pb c!=a\n");
if (d==a) printf("ok d=a\n"); else printf("pb d!=a\n");
```

```
if (c==d) printf("ok c=d\n"); else printf("pb c!=d\n");
```

- Écrire une fonction `affbits` qui contient ce code, qu’affiche-t-il ?
- Modifier la valeur de `a` pour voir la limite de la précision.
- Si vous avez fait l’exercice précédent appeler `affbits` sur les quatre variables `a`, `b`, `c` et `d`.
- Remplacer `float` par `double`, que se passe-t-il ?
- trouver la plus grande valeur de `a` qui met en valeur la précision des `double`

\mathcal{P} Exercice 48. Macros

Faites des macros `#define nom(x)` qui permettent de :

- Qui renvoie le carré de l’argument. Attention au parenthésage.
- Qui renvoie le max des deux arguments en utilisant l’opérateur ternaire `? :`.
- Qui renvoie le min des deux arguments.
- qui renvoie le max des trois arguments.
- qui inverse le contenu des deux arguments.

\mathcal{P} Exercice 49. Makefile

Taper le Makefile avec cibles génériques vu en cours et tester le.

A D Révisions du L1

Les exercices de cette section sont faits pour se mettre à jour du niveau de début de ce cours.

D Exercice 50. Etoiles

Écrire un programme qui affiche à l'écran 10 étoiles sous la forme suivante :

```

      *
     *
    *
   *
  *
 *
*

```

D Exercice 51. Conversions

Écrire un programme qui convertit un temps donné en secondes en heures, minutes et secondes (avec l'accord des pluriels).

Exemple d'exécution :

3620 secondes correspond à 1 heure 0 minute 20 secondes

D Exercice 52. Multiplication Égyptienne

Pour multiplier deux nombres, les anciens égyptiens se servaient uniquement de l'addition, la soustraction, la multiplication par deux et la division par deux. Ils utilisaient le fait que, si X et Y sont deux entiers strictement positifs, alors :

$$X \times Y = \begin{cases} (X/2) \times (2Y) & \text{pour } X \text{ pair} \\ (X-1) \times Y + Y & \text{pour } X \text{ impair} \end{cases}$$

Écrire un programme qui, étant donnée deux nombres (dans l'exemple 23 et 87), effectue la multiplication égyptienne, en affichant chaque étape de la façon suivante :

```

23 x 87
= 22 x 87 + 87
= 11 x 174 + 87
= 10 x 174 + 261
= 5 x 348 + 261
= 4 x 348 + 609
= 2 x 696 + 609
= 1 x 1392 + 609
= 2001

```

D Exercice 53. Limites

Calculez la limite de la suite

$$S_n = \sum_{i=1}^{i=n} \frac{1}{i^2}$$

en sachant que l'on arrête l'exécution lorsque $|S_{n+1} - S_n| < \epsilon$, ϵ étant la précision fixée à l'avance par une constante.

D Exercice 54. Nombres premiers

Écrire un programme qui teste si un nombre est premier ou pas.

D Exercice 55. Nombres amis

Soit n et m , deux entiers positifs. n et m sont dits *amis* si la somme de tous les diviseurs de n (sauf n lui-même) est égale à m et si la somme de tous les diviseurs de m (sauf m lui-même) est égale à n .

Écrire une fonction qui teste si deux entiers sont des nombres amis ou non.

Écrire une fonction qui, étant donné un entier positif $nmax$ affiche tous les couples de nombres amis (n, m) tels que $n \leq m \leq nmax$.

Aide : 220 et 284 sont amis.

D Exercice 56. Racines

Ecrivez un programme qui calcule la racine d'un nombre à une erreur ε fixée par une méthode de dichotomie.

D Exercice 57. Les suites de Syracuse

On se propose de construire un petit programme qui permet d'étudier les suites dites de Syracuse :

$$u_{n+1} = \begin{cases} \frac{u_n}{2} & \text{si } u_n \text{ est pair} \\ 3u_n + 1 & \text{si } u_n \text{ est impair} \end{cases}$$

La conjecture de Syracuse dit que quelle que soit la valeur de départ, la suite finit par boucler sur les valeurs 4,2,1,4,2,1,...

a. Construire un programme qui, à partir d'une valeur de départ u_0 , affiche les valeurs successives jusqu'à tomber sur la valeur 1 ;

b. modifier le programme pour qu'il compte le nombre d'itérations, sans affichage intermédiaire ;

c. modifier le programme pour qu'il affiche le nombre d'itérations pour toutes les valeurs de départ entre 1 et une valeur fixée.

D Exercice 58. Factorielle

Pourquoi la fonction suivante peut donner des résultats faux ?

```
int factorielle (int n)
{
    int f;
    if (n <= 1)
        f = 1;
    else
        f = n * factorielle(--n);
    return f;
}
```

D Exercice 59. Calcul de suite

Calculer les valeurs successives de la suite :

$$u_n = \sqrt{1 + \sqrt{2 + \sqrt{\dots + \sqrt{n}}}}, \text{ pour } 1 \leq n \leq N.$$

D Exercice 60. Table ASCII

Le code ASCII permet de coder chaque caractère (imprimable par une machine) à l'aide d'un octet. Ecrire un programme qui affiche le code ASCII. Par exemple, 65 code pour A, 48 code pour 0, etc

D Exercice 61. Tableau

Dans la suite vous utiliserez un tableau d'entiers comportant N cases où N est une constante.

- a. Ecrire une fonction qui initialise toutes les cases du tableau à 1.
- b. Ecrire une fonction qui affiche le produit des éléments d'un tableau.
- c. Ecrire une fonction qui retourne le minimum d'un tableau
- d. Ecrire une fonction qui effectue un décalage de 1 case à droite de tous les éléments d'un tableau. La case à gauche est affectée à 0. Le dernier élément est supprimé du tableau.
- e. Ecrire une fonction qui insère une valeur dans un tableau trié. Après l'insertion, le tableau est toujours trié. Le dernier élément du tableau est supprimé.
- f. Ecrire une fonction qui inverse les éléments d'un tableau. Cette inversion s'effectue sur le tableau lui-même (n'utilisez pas de tableau intermédiaire).
- g. Ecrire une fonction qui élimine les valeurs en double (ou plus) d'un tableau d'entiers positifs en remplaçant ces valeurs en double par leur valeur négative. La première apparition de la valeur reste inchangée.
- h. On suppose que le tableau est découpé en section de nombres significatifs. Chaque section est séparée par un ou plusieurs 0. Ecrire une fonction qui calcule **la moyenne des produits** de chaque section. Par exemple, si l'on dispose du tableau suivant :

1	2	3	0	0	5	4	0	0	8	0	10	11
---	---	---	---	---	---	---	---	---	---	---	----	----

. Il y a 4 sections. La moyenne des produits correspond à

$$\frac{(1.2.3) + (5.4) + 8 + (10.11)}{4} = 36,0$$

D Exercice 62. Tri

- a. Remplir un tableau de valeurs aléatoires comprises entre 0 et 99.
- b. Calculer le nombre de valeurs différentes dans le tableau.
- c. Calculer le tableau d'entiers de taille 10 dont l'élément indicé par i contient le nombre de valeurs aléatoires dont la division par 10 vaut i .