

Ce TP est à rendre, car il compte pour une note de travaux pratiques. Il faudra donc déposer votre travail sur Moodle. C'est simple, il suffira de copier-coller les différents programmes demandés dans une feuille de réponse, puis ensuite de déposer cette feuille sur Moodle.

Le travail est personnel, la note est individuelle. Toute tentative pour copier les réponses de quelqu'un d'autre sera sanctionnée par un zéro (Discord et autres). Et le TP doit être commencé et fini uniquement pendant les séances de TP, pas à la maison. Par contre, vous pouvez communiquer verbalement pour vous aider, pendant les séances. Le deuxième DS portera sur le contenu des TP.

👉 Téléchargez [ReponsesTP1.txt](#) et mettez votre nom/prénom et groupe de TP aux endroits indiqués. Ne modifiez surtout pas la structure de ce fichier.

On va reprendre et appliquer tous les concepts vus en TD jusqu'à présent et on va les appliquer à des calculs arithmétiques.

👉 Ouvrez l'URL <https://perso.univ-rennes1.fr/pierre.nerzic/IntroArchi/CimPU/>.

👉 Agrandissez la fenêtre (ou zoomez) pour travailler confortablement, et configurez CimPU ainsi :

- Mettez en niveau 1.
- Affichez la mémoire centrale en mode *Contexte d'exécution*.
- Désactivez les animations.

## 1. Instructions simples

### 1.1. Rappels

On va réviser les instructions d'affectation vues dans les TD5 et 6.


Déjà, pour commencer, une unité centrale comme CimPU ne gère que des octets, donc des nombres compris entre 0 et 255, ou entre -128 et +127 s'ils sont en convention  $C2^8$ . On verra dans la deuxième partie du TP comment faire avec des nombres plus grands que ça. Les octets codent à la fois des entiers, des textes et aussi les instructions des programmes.

Ensuite, il n'y a que deux types d'emplacements pour des nombres, la mémoire centrale et les registres. Les registres, R0 et R1 servent uniquement pour les calculs. La mémoire centrale stocke les nombres de manière permanente. La mémoire de CimPU contient 256 octets en tout. Chaque octet possède une adresse entre 0 et 255. Les premiers octets sont réservés pour les programmes, et le reste de la mémoire est pour les données.

Il y a deux instructions à connaître :

- LD *reg*, *valeur* : elle place la valeur dans le registre. Cette valeur peut prendre plusieurs formes :
  - une constante (appelée *immédiate*) comme dans LD R1, 9 qui met 9 dans R1.
  - un autre registre comme dans LD R0, R1. Les combinaisons comme LD R0, R0 sont interdites.
  - une adresse mémoire mise entre crochets comme dans LD R1, [34], qui met le contenu de la mémoire à l'adresse 34 dans le registre R1.
- ST *reg*, [*adresse*] : elle recopie le registre dans la mémoire.

Les instructions sont limitées. CimPU ne peut faire qu'un seul accès mémoire dans une instruction, donc on ne peut pas faire `ST [adresse1], [adresse2]`. On est systématiquement obligé de passer par un registre.

Au lieu de mettre des adresses fixes, on utilise des *labels*. Ce sont des noms donnés à des cellules mémoire, comme dans le programme suivant : 

```
LD R0, [V1]
LD R1, [V2]
ST R0, [V2]
ST R1, [V1]
HLT

V1:    DB 12
V2:    DB 98
```

☛ Assemblez et exécutez-le. Que fait-il ? Inspectez la mémoire avant et après exécution...


☛ Le programme suivant fait quelque chose de bizarre. Avec quoi affecte-t-il la cellule mémoire V2 et pourquoi ? 

```
LD R0, V1
ST R0, [V2]
HLT

V1:    DB 12
V2:    DB 98
```

Donc attention à ne pas confondre un accès mémoire noté entre crochets, avec une constante sans crochets. La mémoire est comme une sorte de tableau – les adresses mémoire sont les indices dans ce tableau. Ainsi, l'écriture `[adresse]` ressemble à l'accès à un tableau en langage C `tab[i]`, sauf qu'on ne met pas le nom du tableau.

## 1.2. Exercice

☛ Complétez ce programme qui doit mettre à zéro les trois variables V1, V2 et V3. Ça écrase donc les anciennes valeurs : 


```
    ; TODO mettre V1, V2 et V3 à zéro
    HLT

V1:    DB 12
V2:    DB 98
V3:    DB $C3    ; nombre en base 16
```

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP1.txt`.

Vérifiez maintenant si votre `ReponsesTP1.txt` est bien enregistré. Avec des quotas disque épuisés, les nouveaux fichiers sont vides ! Utilisez du `-hs ~` pour savoir combien de place vous occupez et `rm -fr .cache` pour récupérer un peu de place.

### 1.3. Exercice


☛ Complétez ce programme qui doit recopier ce qu'il y a dans V1 (pas l'adresse V1 mais le contenu de la mémoire en V1) dans V2 et V3. Ça écrase donc les anciennes valeurs : 

```
        ; TODO lire ce qu'il y a en V1 et le mettre dans V2 et dans V3
        HLT

V1:     DB 12
V2:     DB 98
V3:     DB $C3        ; nombre en base 16
```

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP1.txt`.

### 1.4. Exercice

☛ Complétez ce programme qui doit faire une circulation des valeurs (on dit *permutation circulaire*) : V1 doit aller dans V2, pendant que V2 va dans V3 et V3 va dans V1. Il est autorisé d'utiliser une case mémoire supplémentaire : 

```
        ; TODO ce qui est dans V1 est à mettre dans V2,
        ; ce qui était dans V2 doit être dans V3
        ; et l'ancien contenu de V3 doit aller dans V1
        HLT

V1:     DB 1
V2:     DB 2
V3:     DB 3
```

Il faut penser à utiliser les deux registres, l'un mémorise une valeur pendant que l'autre réaffecte les autres cellules.

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP1.txt`.

## 2. Entrées/sorties

Un ordinateur n'a d'intérêt que s'il peut communiquer avec l'extérieur. Pour cela, il est entouré de périphériques (*devices*) : écran, clavier, carte réseau, etc. Dans CimPU, c'est très simplifié<sup>1</sup>. On communique avec un périphérique par un *port*. Un port peut être lu ou écrit. C'est comme une sorte de boîte aux lettres. On peut y prendre du courrier, ou en mettre. Par exemple sur CimPU, si on écrit un octet sur le port 0, ça allume certaines leds. Si on lit un octet sur le port 10, ça demande de taper un entier (comme un `scanf("%d", &R0)`).

Le panneau *DSKY* rassemble les périphériques. Faute de place, il ne peut en afficher qu'un seul à la fois. Celui qui est affiché est défini par le *mode*, mais il ne faut pas confondre le mode et les ports. C'est à dire qu'on peut toujours écrire sur n'importe quel port, mais seul le périphérique

---

<sup>1</sup>Dans un vrai ordinateur, c'est le même principe, mais les ports sont beaucoup plus complexes et plus nombreux. En général, il faut écrire plusieurs ports à la suite pour faire quelque chose. Voir par exemple [cette page](#) sur l'ATmega328 utilisé dans les Arduino.

correspondant au mode sera visible à un moment donné. C'est comme avoir plusieurs fenêtres ouvertes sur votre écran. Vous voyez celle qui est en plein écran devant, et ça n'empêche pas les autres de fonctionner derrière.

Si on écrit un octet sur le port 255, ça change le mode du DSKY. C'est comme le sélectionner à la souris.

Deux instructions sont à connaître :

- `IN reg, port` : elle lit le port indiqué et place la valeur dans le registre. Dans certains cas, c'est bloquant : IP n'est pas incrémenté tant que vous ne saisissez pas une valeur dans le DSKY.
- `OUT reg, port` : elle écrit le port avec ce qu'il y a dans le registre.

Il n'est pas demandé de connaître les ports. Ils seront indiqués selon les besoins des exercices. Voici ceux utilisés dans ce TP :

- port 0 en écriture : allume des leds dans le DSKY en mode 0,
- port 8 en écriture : allume des pixels à droite dans l'afficheur LCD du DSKY en mode 8,
- port 10 en écriture : affiche un entier de différentes manières (décimal, convention  $C2^8$ , hexadécimal, binaire, ASCII)
- port 255 en écriture : change le mode du DSKY.

Pour se rafraîchir les souvenirs du TD5, essayez ce programme :



```
; mettre le DSKY en mode 0 = leds
LD R0, 0           ; mode demandé = les leds
OUT R0, 255        ; port 255 = choix du mode du DSKY
; allumer des leds
LD R0, %11001001
OUT R0, 0
HLT
```

## 2.1. Exercice

👉 Étudiez ce programme. Il envoie des octets sur le port 8, celui de l'afficheur LCD :



```
; mettre le DSKY en mode 8 = LCD matriciel
LD R0, 8
OUT R0, 255        ; port 255 = choix du mode du DSKY = 8
; recopier les données sur le port 8
LD R0, [donnees+0]
OUT R0, 8
LD R0, [donnees+1]
OUT R0, 8
LD R0, [donnees+2]
OUT R0, 8
LD R0, [donnees+3]
OUT R0, 8
LD R0, [donnees+4]
OUT R0, 8
LD R0, [donnees+5]
```

```

OUT R0, 8
LD R0, [donnees+6]
OUT R0, 8
LD R0, [donnees+7]
OUT R0, 8
HLT
    
```

```

donnees:
DB %00111100
DB %01000010
DB %10010001
DB %10101101
DB %10100001
DB %10101101
DB %01000010
DB %00111100
    
```

Les octets des données semblent étranges. Chaque bit à 1 affiche un point sur l'écran. Ces données font afficher un petit dessin. On appelle ça un *bitmap*. Voici comment ça fonctionne. Chaque octet qu'on écrit sur le port 8 s'affiche verticalement à droite, en décalant tout vers la gauche.

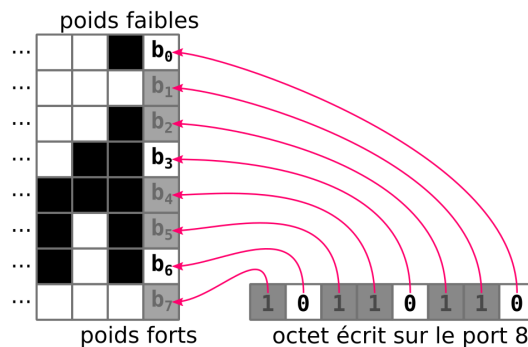


Figure 1: Écriture sur le port 8

☛ Modifiez le programme pour qu'il affiche l'alien de la figure 2.

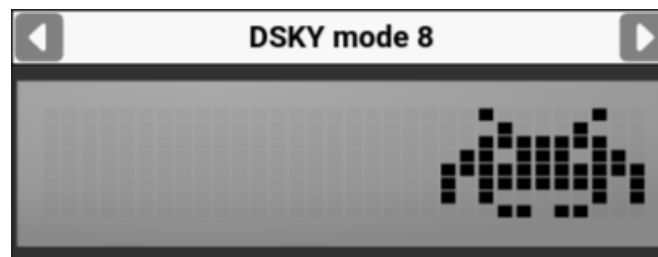


Figure 2: Alien

☛ Une fois qu'il marche, copiez-collez ce programme dans ReponsesTP1.txt.

## 2.2. Amélioration

Le programme précédent est vraiment lourd. Il y a plus d'instructions de copie que de données elles-mêmes. Heureusement, il est possible d'écrire des instructions comme `LD R0, [R1+donnees]`, avec `R1` allant de 0 à 7. Ça fonctionne comme ça :

- Le CPU additionne `R1` à la valeur de `donnees`. Cette dernière est fixe et c'est l'adresse du début des octets à envoyer sur le port 8, mais `R1` est variable, donc l'adresse finale change.
- Le CPU va chercher l'octet qui est à cette adresse et le met dans `R0`.

Cette manière d'écrire les programmes est parfaitement possible. Cela s'appelle l'*adressage indirect avec décalage*. Le mot *indirect* signifie qu'on désigne un octet en mémoire par une adresse située dans un registre. Ici, en plus, on rajoute un décalage. Ce qui fait que l'adresse de la donnée, c'est la valeur du registre `R1` + le décalage.

L'adressage direct consiste à mettre seulement une adresse, comme dans `LD R0, [donnees]`. Le fait d'écrire `LD R0, [donnees+1]` ne change rien, parce que l'addition `donnees+1` reste une constante.

NB: on ne peut pas écrire `LD R0, [donnees+R1]`, on doit toujours mettre le registre en premier.

Le programme se simplifie énormément en utilisant une boucle :



```
; mettre le DSKY en mode 8 = LCD matriciel
LD R0, 8
OUT R0, 255 ; port 255 = choix du mode du DSKY = 8
; recopier les données sur le port 8
LD R1, 0
repete:
LD R0, [R1+donnees]
OUT R0, 8
INC R1 ; R1++
jusqua:
CMP R1, 7
BLS repete
HLT

donnees:
DB %00111100
DB %01000010
DB %10010001
DB %10101101
DB %10100001
DB %10101101
DB %01000010
DB %00111100
```

👉 Mettez l'unité centrale au niveau 3 pour pouvoir utiliser ce mode d'adressage.

👉 Assemblez-le et testez-le.

Vous remarquerez qu'il est un peu plus lent, parce qu'il y a davantage d'instructions à exécuter. Avec `gcc`, il y a le même dilemme quand on veut optimiser (option `-O`). On peut demander à ce

qu'un programme soit plus rapide, alors en général il est plus grand, parce que certaines boucles sont transformées en une répétition des instructions. Et on peut demander à ce qu'un programme soit plus petit, mais alors il est un peu plus lent.

Retenir que ce programme considère les données comme un tableau et R1 est l'indice de la case à envoyer sur le port 8.

☛ Adaptez ce programme pour qu'il dessine l'alien.

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP1.txt`.

### 3. Arithmétique sur 16 bits

Le processeur simulé par CimPU est qualifié de « 8 bits ». Dans les PC actuels, on a des processeurs 64 bits. Dans les deux cas, ça définit la taille des entiers que l'unité centrale manipule directement, mais ça n'empêche pas du tout de calculer avec des entiers plus grands.

On va voir comment faire des additions et soustractions de grands nombres avec CimPU.

#### 3.1. Principe

On va commencer par la base 10. Vous-même êtes un processeur « 1 chiffre ». C'est à dire que vous pouvez calculer directement la somme de chiffres isolés. Par exemple  $2+5$  ou  $7+8$ . Quand vous devez calculer la somme de nombres de plusieurs chiffres, comme  $789+654$ , vous ne pouvez normalement pas le faire directement. Vous devez poser le calcul en alignant bien les unités à gauche et vous devez additionner successivement les paires de chiffres de chaque rang, en commençant à gauche, et en reportant les retenues d'un rang à l'autre. Et il ne faut pas oublier de reporter la toute dernière retenue de gauche.

Arrêtons-nous un instant sur cette dernière retenue. Soit 789 à additionner avec 654 :

$$\begin{array}{r} \phantom{+} 0789 \\ + 0654 \\ \hline 1443 \end{array}$$

En fait, le dernier chiffre calculé tout à gauche résulte d'une addition  $1 + 0 + 0$  esquissée en gris clair tout à gauche. C'est une dernière addition implicite pour nous, mais dans un programme, il faudra la faire explicitement, parce que la retenue n'est pas un nombre comme un autre, mais un indicateur dans l'UAL.

C'est le même algorithme avec des entiers faisant plusieurs octets sur CimPU. On doit additionner les octets de chaque rang en commençant par les « unités », et en reportant bien les retenues. En fait, on considère les octets comme des « chiffres » de la base 256.

Question : quelle est la valeur maximale d'une retenue dans l'addition de deux chiffres ? En base 10, la retenue maximale est atteinte quand on additionne 9 avec 9, ça fait 8 avec une retenue de 1. Et quand il y a déjà une retenue de 1 en amont,  $1 + 9 + 9$  font 9 avec une retenue de 1. Donc la plus grande retenue possible est 1.

Quand on calcule avec des octets, quelle est la retenue maximale ?  $255 + 255 = ?$  et ça fait quel nombre à mettre en dessous et quelle retenue ? Et si on a cette retenue qu'on additionne à  $255 + 255$ , combien ça fait ? Donc quelle est la plus grande retenue possible ?

✎ Écrire les réponses à ces deux questions principales (retenue max et nombre max en bas) dans `ReponsesTP1.txt`.

Dans CimPU, la retenue est mémorisée dans l'indicateur  $C$  et il y a deux instructions pour faire des additions :

- `ADD reg, valeur` : elle effectue  $reg = reg + valeur$ ,
- `ADC reg, valeur` : elle calcule  $reg = reg + valeur + C$ .

Dans ces deux instructions, la valeur peut être écrite comme pour l'instruction `LD` : avec une constante, un registre ou la mémoire. L'indicateur  $C$  passe à 1 si  $reg + valeur > 255$  et revient à zéro s'il n'y a pas de retenue.

Donc le principe est le suivant : on additionne les « unités » avec `ADD` puis on additionne les octets suivants avec `ADC`.

## 3.2. Expérimentations

✎ Assemblez et exécutez ce programme.



```
LD R0, 12      ; nombre 1
LD R1, 24      ; nombre 2

ADD R0, R1     ; R0 = nombre1 + nombre2

OUT R0, 10     ; afficher la somme
HLT
```

✎ Remplacez les deux nombres par 250. Le total est 500, mais on ne voit affiché que 244. C'est parce qu'il y a une retenue qui représente 256, car  $244 + 256 = 500$ . La retenue est visible dans l'UAL, indicateur  $C$ . Accessoirement, on voit aussi  $N$  à 1, mais on ne s'en occupe pas.

## 3.3. Application au calcul 16 bits

Qu'est-ce qu'un nombre 16 bits ? C'est un nombre qui s'écrit sur deux octets, par exemple  $(1001\ 0101\ 1011\ 0111)_2$ . On l'écrit plus facilement en base 16 : `95B7`. Il est composé de deux octets, `95` qui est le poids fort, et `B7`, le poids faible. Quand on additionne deux nombres 16 bits, on doit d'abord additionner les poids faibles, puis les poids forts en tenant compte des retenues.

✎ Lancez une console Python et tapez `f"{0x95B7 + 0x4D6A:X}"` ou `hex(0x95B7 + 0x4D6A)`, la réponse est `'E321'`<sup>2</sup>. C'est la somme des deux nombres. Tapez `f"{0x95B7 + 0x8D6A:X}"` et analysez le résultat (retenue ?).

Voici la même addition posée pour des octets. Vous vérifierez avec Python que  $B7 + 6A = 121$  et que  $1 + 95 + 8D = 123$  (tout en hexadécimal) : `f"{0xB7 + 0x6A:X}"` et `f"{1 + 0x95 + 0x8D:X}"`.

---

<sup>2</sup>En Python, une chaîne `f"...` est appelée *chaîne formatée* (f-string), voir [la doc](#). Les parties entre accolades `{expression}` sont remplacées par la valeur de l'expression. On peut ajouter un format, `{expression:X}`, par exemple pour dire qu'on veut le résultat en hexadécimal.



$$\begin{array}{r}
 \phantom{+} \phantom{00} \phantom{8D} \phantom{6A} \\
 \phantom{+} \phantom{00} \phantom{8D} \phantom{6A} \\
 + \phantom{00} \phantom{8D} \phantom{6A} \\
 \hline
 01 \phantom{23} \phantom{21}
 \end{array}$$

Voici un point de départ. On a ces deux nombres 16 bits stockés dans 4 octets, N1 et N2. Le but est de les additionner dans S (sur 3 octets). 

```

; addition des poids faibles
LD R0, [N1+0] ; unités du premier nombre
ADD R0, [N2+0] ; unités du second nombre
ST R0, [S+0]
; TODO addition des poids forts
HLT

N1:  DB $B7      ; poids faible
     DB $95      ; poids fort
N2:  DB $6A      ; poids faible
     DB $8D      ; poids fort
S:   DB 0        ; poids faible
     DB 0        ; poids intermédiaire
     DB 0        ; poids fort
    
```

☛ Complétez ce programme pour qu'il finisse l'addition dans S. N'oubliez pas la troisième addition pour que la retenue finale soit mise dans le 3e octet de S. Cette dernière addition se fait avec des zéros.

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP1.txt`.

### 3.4. Soustraction 16 bits

C'est le même principe. On soustrait les poids faibles puis les poids forts, en tenant compte des retenues. On va se poser une petite contrainte, c'est que le premier nombre soit plus grand que le second, par exemple  $(B567)_{16} - (8DA6)_{16} = (27C1)_{16}$ , pour ne pas avoir de retenue finale. On les soustrait *octet par octet* :

$$\begin{array}{r}
 \phantom{-} \phantom{8D} \phantom{+1} \phantom{A6} \\
 \phantom{-} \phantom{8D} \phantom{+1} \phantom{A6} \\
 - \phantom{8D} \phantom{+1} \phantom{A6} \\
 \hline
 27 \phantom{C1}
 \end{array}$$

Vous vérifierez avec Python que  $(167)_{16} - (A6)_{16} = (C1)_{16}$  et que  $(B5)_{16} - (8D)_{16} - 1 = (27)_{16}$  (tout en hexadécimal) : `f"{0x167 - 0xA6:X}"` et `f"{0xB5 - 0x8D - 1:X}"`.

Il y a deux instructions :

- `SUB reg, valeur` : elle effectue  $reg = reg - valeur$ ,
- `SBC reg, valeur` : elle calcule  $reg = reg - valeur - C$ .

☛ Reprendre le programme précédent pour qu'il calcule  $(B567)_{16} - (8DA6)_{16}$ . Attention, ce ne sont pas les mêmes valeurs que pour l'addition.

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP1.txt`.

### 3.5. Décalage à gauche 16 bits non signés

On a vu l'importance de cette opération pour faire des multiplications. Voir figure 3 le principe avec des nombres sur 2 octets considérés comme non signés.

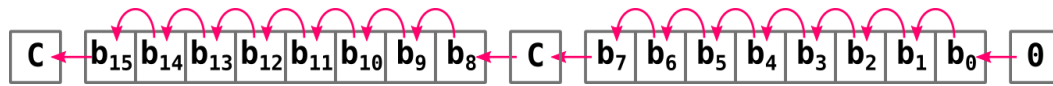


Figure 3: Décalages 16 bits

Comment faire ? On dispose de deux instructions, LSL et ROL illustrées figure 4.

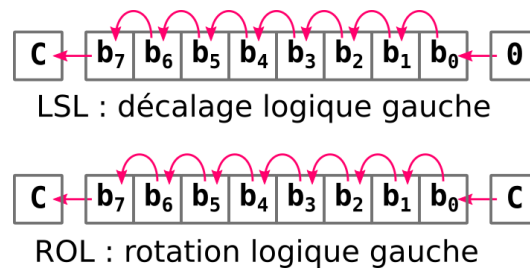



Figure 4: Décalages et rotations 8 bits

Les deux instructions concernées sont :

- LSL `reg` : elle décale le registre à gauche en faisant rentrer un zéro à droite. Le bit de poids fort sortant va dans la retenue C,
- ROL `reg` : elle décale le registre à gauche en faisant rentrer la retenue à droite. Le bit de poids fort sortant va dans la retenue C.

Voici un point de départ. On a un nombre 16 bits,  $N = (1001\ 1110\ 1011\ 0111)_2$  qu'on doit décaler à gauche *sur place* c'est à dire que  $N$  est modifié après l'opération, car les instructions écrasent son ancienne valeur (comme quand on fait `i=i+1`; en langage C). 

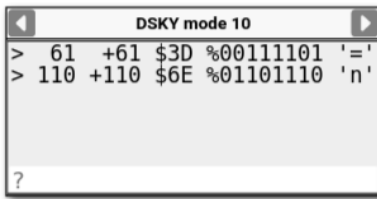
```

; TODO décalage à gauche de N
; ...ici...

; affichage de N sur le terminal 10 dans l'ordre poids fort, poids faible
LD R0, [N+1] ; poids fort
OUT R0, 10
LD R0, [N+0] ; poids faible
OUT R0, 10
HLT

N:   DB %10110111 ; poids faible
     DB %10011110 ; poids fort
    
```

Le résultat attendu est  $(0011\ 1101\ 0110\ 1110)_2$  avec C valant 1 et ces affichages :



```
DSKY mode 10
> 61 +61 $3D %00111101 '='
> 110 +110 $6E %01101110 'n'
?
```

☛ Une fois qu'il marche, copiez-collez ce programme dans `ReponsesTP1.txt`.

Pensez à déposer votre `ReponsesTP1.txt` dans la zone de dépôt sur Moodle. Rappel : c'est un travail personnel, individuel qui est soumis à une notation. Les tricheries seront sanctionnées.