Chapitre 10

Série d'exercices n°10 - Théorie chapitre IV

10.1 Correctifs des exercices

10.1.1 Exercice 4.4.7

Énoncé :

(Examen de seconde session, 2018)
On souhaite programmer une fonction `facteur_deux(n)` chargée de calculer le plus grand entier k tel que 2^k divise n, où n est un nombre de 64 bits supposé strictement positif. Par exemple, `facteur_deux(96)` doit retourner 5, car $96 = 2^5\,3$. Lorsque `n` est impair, `facteur_deux(n)` doit retourner 0.

1. Écrire, en pseudocode ou en langage C (au choix), un algorithme permettant de résoudre ce problème.
   Suggestion : En exploitant les instructions logiques, compter le nombre de bits nuls situés à la fin de la représentation binaire de n.
2. Traduire cet algorithme en assembleur x86-64, en veillant à respecter la convention d'appel de fonctions des systèmes *Unix*.

Solution :

Pour comprendre cet énoncé, on détaille d'abord les instructions qui nous sont données. On nous demande de calculer le plus grand entier k tel que 2^k divise un nombre n à passer en argument d'entrée. Il faut comprendre que cet énoncé parle de division entière, au sens où la puissance 2^k doit diviser le nombre n sans qu'il n'y ait de reste après l'unité ; le résultat de la division doit être lui-même un entier pour qu'on puisse dire, dans le langage commun, que "2^k divise n".

Par exemple, la puissance $k = 3$ divise (entre autres) 8 et 24 puisque $\frac{8}{2^3} = 1$ et $\frac{24}{2^3} = 3$.

Avec la suggestion proposée dans l'énoncé, on peut assez facilement atteindre la solution. On préferera ici une solution un peu plus compliquée et "naïve", comme si l'on avait pas eu cette suggestion, afin de développer un programme un peu plus intéressant.

On implémente d'abord la logique du problème en langage C en parcourant l'énoncé point par point.

```
unsigned long facteur_deux(unsigned long n)
{
}
```

Cette déclaration correspond aux indications de l'énoncé : on a une fonction `facteur_deux` qui prend en entrée le nombre n de 64 bits strictement positif (donc en C de type unsigned long, par exemple) et qui renvoie en argument de sortie la puissance k dont l'énoncé ne précise pas la nature, et qu'on va également supposer être un entier positif sur 64 bits.

```
unsigned long facteur_deux(unsigned long n)
{
  unsigned long k;

  for( k = 0 ; (n % 2) != 0 ; k++ )
    n /= 2 ;

  return k;
}
```

Cette boucle implémente le fonctionnement décrit : on déclare une variable k, puis on l'utilise dans la boucle tel qu'on continue à itérer en incrémentant k tant que $(n\%2) == 0$, c'est-à-dire tant que le modulo de $\frac{n}{2}$ est nul. Dans cette boucle, on divise à chaque itération n par 2. Ce test revient exactement à déterminer à chaque fois si le nombre n est divisible par 2, puis si le restant de cette division est divisible par 2, etc. Lorsqu'on aura $(n\%2) \neq 0$, c'est que le reste de la dernière division n'est plus lui-même divisible par 2 et qu'on a donc atteint le k maximal tel que par exemple $95 = 2^5 3$ où $\frac{3}{2}$ n'est plus un entier et $3\%2 = 1 \neq 0$.

On traduit ensuite ce programme rédigé en langage C en instructions assembleur, comme suit :

```
facteur_deux:
                # RDI : contient le nombre n
```

On sait par la convention d'appel que l'unique argument d'entrée du programme assembleur se trouve dans le registre RDI.

```
        MOV RAX, 0 # RAX contiendra la puissance k
        MOV R9, 0 # R9 contiendra le quotient des divisions par 2
        MOV R10, RDI # R10 contiendra le reste des divisions par 2

divide:
        CMP R10, 1
        JLE end_divide

        SUB R10, 2
        INC R9
        JMP divide
```

On initialise d'abord le registre RAX ; on va l'utiliser directement dans le programme comme registre de travail, et il contiendra à la fin de son exécution la valeur de l'argument de sortie, comme exigé par la convention d'appel. Puisqu'on va déterminer la valeur de k par une boucle qui va l'incrémenter progressivement, on doit l'initialiser à 0. On initialise ensuite R9 et R10 qui contiendront respectivement le quotient et le reste des divisions par 2 que l'on va réaliser dans la boucle. On initialise R9 à zéro pour pouvoir l'incrémenter au cours de la division, et R10 à la valeur de n mise à disposition dans RDI pour pouvoir l'utiliser dès la première itération comme base de la première division par 2.

Pour effectuer la division à l'intitulé "divide", on emploie la solution "naïve" qui consiste à utiliser SUB pour soustraire 2 au nombre à diviser autant de fois qu'on peut le faire jusqu'à ce que le reste de la division soit lui-même inférieur à 2. À chaque fois que l'on compte une fois le nombre 2 dans R10, c'est-à-dire à chaque soustraction effectuée avant que le gardien de boucle ne nous arrête, on incrémente R9. De cette manière, lorsque R10 ne contiendra plus une valeur égale ou supérieure à 2, on pourra sortir de la boucle "divide" et R9 contiendra le nombre de fois que 2 est présent dans la valeur originelle de R10, soit littéralement le quotient de cette valeur divisée par 2. De son côté, R10 contient bien le reste de la division de sa valeur originelle par 2, qui sera égal à 0 si cette valeur est divisible par 2.

Pour ce faire, on insère donc un gardien de boucle qui utilise CMP pour comparer le reste de la division R10 à 1. Si le reste est inférieur ou égal à 1, il n'est strictement plus possible de diviser le nombre d'origine par 2 ; on doit donc s'arrêter de le diviser par soustraction. Dans ce cas, on réalise la condition de JLE qui nous envoie à la fin de la division qu'on associera à un intitulé "end_divide".

```
end_divide:
            CMP R10, 0
            JNE end
            CMP R9, 0
            JE end

            MOV R10, R9
            MOV R9, 0
            INC RAX
            JMP divide
```

À la fin de la division de la valeur originellement placée dans `R10`, on doit évaluer le reste et le quotient de la division par 2. Comme mentionné précédemment, si le reste est nul, c'est que la valeur était divisible par 2. Dans ce cas, on s'attend à pouvoir procéder à la prochaine division par 2 du quotient de la division actuelle. Par exemple, $\frac{12}{2} = 6$, et il reste 0 : on sait que 12 est divisible au moins une fois par 2, et on peut encore diviser 6 pour voir si 12 et divisible par 4 (c'est bien le cas). Cependant, on doit aussi observer le quotient : s'il vaut 0, c'est que la valeur d'entrée était elle-même 0, un cas trivial techniquement divisible par n'importe quel nombre mais ne correspondant pas aux valeurs intéressantes pour notre problème.

Pour implémenter ce comportement, on teste donc la valeur du reste au moyen d'un CMP entre `R10` et 0. Si `R10` est différent de 0, on saute à la fin du programme au moyen de JNE puisque dans ce cas la valeur n d'origine n'est plus divisible par une puissance additionnelle de 2. On teste également le quotient, et s'il est nul, on saute également à la fin du programme avec JE.

Si ces clauses ne sont pas vérifiées, c'est qu'il reste au moins une puissance de 2 à extraire de n, et pour le déterminer, on place le quotient actuel disponible dans `R9` dans le registre `R10` qui sera à nouveau divisé à l'intitulé "divide". On réinitialise `R9` à 0 pour pouvoir compter de manière incrémentale le quotient de la prochaine division et on ne doit pas oublier d'incrémenter la puissance k dans `RAX`, puisque c'est ce compte du nombre de puissance de 2 présentes dans n qui nous intéresse, puis on saute à la section "divide" par un saut inconditionnel JMP.

```
                .intel_syntax noprefix
                .text
                .global facteur_deux
                .type facteur_deux, @function

facteur_deux:
                # RDI : contient le nombre n

                PUSH RBP
                MOV RBP, RSP

                MOV RAX, 0 # RAX contiendra la puissance k
                MOV R9, 0 # R9 contiendra le quotient des divisions par 2
                MOV R10, RDI # R10 contiendra le reste des divisions par 2
```

On doit ajouter les clauses habituelles de déclaration de syntaxe et de la fonction `facteur_deux` avec son attribut "global". On doit aussi sauver le contexte précédent et définir le nouveau contexte sur la pile à l'aide des instructions d'usage PUSH RBP et MOV RBP, RSP.

```
end:
        POP RBP
        RET
```

Enfin, on doit rédiger la section de fin du programme. Celle-ci se chargera de restaurer le contexte appellant en dépilant `RBP` et enfin de mettre fin à la fonction avec RET. La valeur de sortie se trouvant déjà dans le registre `RAX` suite l'exécution du programme, il n'y a rien de plus à faire.

Si on assemble les blocs rédigés jusqu'ici, on a la solution finale suivante :

```
                .intel_syntax noprefix
                .text
                .global facteur_deux
                .type facteur_deux, @function

facteur_deux:
                # RDI : contient le nombre n

                PUSH RBP
                MOV RBP, RSP

                MOV RAX, 0 # RAX contiendra la puissance k
                MOV R9, 0 # R9 contiendra le quotient des divisions par 2
                MOV R10, RDI # R10 contiendra le reste des divisions par 2

divide:
                CMP R10, 1
                JLE end_divide

                SUB R10, 2
                INC R9
                JMP divide
end_divide:
                CMP R10, 0
                JNE end
                CMP R9, 0
                JE end

                MOV R10, R9
                MOV R9, 0
                INC RAX
                JMP divide
end:
                POP RBP
                RET
```

10.1.2 Exercice 1 (troisième partie)

Énoncé : Pour chacun des exercices de la section 3.5.2, traduire votre solution en un programme assembleur x86-64 complet, accompagné si nécessaire d'un programme C permettant de le tester.

Solution :

(7) On repart de la solution établie dans les séries d'exercices précédents en rappelant l'énoncé auquel le programme rédigé doit répondre : convertir, dans une chaîne de caractères ASCII terminée par un zéro située à une adresse donnée, toutes les lettres minuscules en majuscules. Les autres caractères ne doivent pas être modifiés.

```
Hypothèses  R_str : contient l'adresse de la chaîne

init :      R_currentAdr ← R_str
            R_currentVal ← 0
            R_offset ← 32
            R_inc ← 1

loop :      R_currentVal ← ptr R_currentAdr
            JMP "end" if R_currentVal == 0

            ;être dans [97,122] == ne pas être dans ]97,122[

            JMP "next" if (R_currentVal < 97 || R_currentVal > 122)
            ptr R_currentAdr ← R_currentVal - R_offset

next :      R_currentAdr ← R_currentAdr + R_inc
            JMP "loop"

end :
```

La traduction de l'algorithme rédigé en pseudo-code en séquence d'instructions assembleur peut se faire presque immédiatement ; on la détaille ci-dessous :

```
to_upper :  # RDI : contient l'adresse de la chaîne
```

Comme d'habitude, les hypothèses sur la position des arguments d'entrée de la fonction sont résolues par la convention d'appel. En l'occurence, l'unique argument correspondant à l'adresse de la chaîne à évaluer est par convention placé dans le registre `RDI`. On remarque également que toutes les clauses d'initialisation de registres en pseudo-code sont superflues en assembleur ; en

effet, on a pas besoin de déréférencer la chaîne de caractères et on peut directement utiliser RDI dans notre gardien de boucle ; de plus, on peut utiliser des littéraux plutôt que de devoir placer des constantes dans des registres.

```
loop :  MOV R9B, byte ptr[RDI]
        CMP R9B, 0
        JE end
```

Comme mentionné ci-avant, on pourrait utiliser directement le registre RDI dans le gardien de boucle grâce à l'opérateur CMP qui accepte un adressage mémoire en seconde opérande. Cependant, on va explicitement le déréférencer dans le registre de travail arbitraire R9B pour l'unique raison qu'il va être ensuite être sujet à de nombreuses opérations qui se termineront par un MOV dont les deux opérandes ne peuvent être des accès mémoire. On remarque qu'on manipule bien la sous-partie R9B du registre R9 puisqu'on opère uniquement sur des octets.

On développe ensuite la clause de saut conditionnel en pseudo-code en une comparaison et un saut conditionnel aux moyens de CMP et JE, de telle manière qu'on sautera à la fin du programme dès que l'on rencontrera le zéro de terminaison de la chaîne de caractères.

```
CMP R9B, 0x61
JL next
CMP R9B, 0x7A
JG next
```

Ces clauses de comparaisons et de sauts conditionnels correspondent à celles rédigées en pseudo-code en une ligne : on vérifie si la valeur courante se trouve entre les valeurs 97 et 122 déterminant l'espace des miniscules. Si c'est le cas, on passe les deux sauts conditionnels et on arrive à la section suivante du code ; sinon, on va à la section associée à l'intitulé "next" qui renverra à la prochaine itération du programme.

```
SUB R9B, 0x20
MOV byte ptr[RDI], R9B
```

Si on arrive à cette section, c'est que le caractère courant est une lettre minuscule. Dans ce cas, on effectue la soustraction qui la transforme en majuscule, et on la replace à l'adresse indiquée par le pointeur sur le tableau. C'est ce MOV qui ne peut prendre deux opérandes d'accès mémoire qui nous a contraint à déréférencer RDI dans R9B ci-avant. On aurait pu attendre cette section pour effectuer ce déréférencement juste avant la soustraction, mais le code gagne en lisibilité à le faire au début de la boucle.

```
next :  INC RDI
        JMP loop

end :   RET
```

Enfin, on rédige la section "next" qui fait simplement avancer le pointeur sur le tableau que l'on évalue puis sauter de manière inconditionnelle à l'intitulé associé au début de la boucle, pour effectuer la prochaine itération ou atteindre la fin du programme. On indique également la section "end" qui effectue le RET de circonstance. Dans le cas de ce programme, il n'y a pas de valeur de sortie à placer dans le registre RAX puisqu'on place le résultat de l'opération effectuée par la fonction dans l'argument d'entrée lui-même.

Au total, avec les clauses habituelles déclarant la syntaxe de programmation et la fonction avec son attribut "global", on a le programme suivant :

```
            .intel_syntax noprefix
            .text
            .global to_upper
            .type to_upper, @function

to_upper :  # RDI : contient l'adresse de la chaîne

loop :      MOV R9B, byte ptr[RDI]
            CMP R9B, 0
            JE end

            CMP R9B, 0x61
            JL next
            CMP R9B, 0x7A
            JG next

            SUB R9B, 0x20
            MOV byte ptr[RDI], R9B

next :      INC RDI
            JMP loop

end :       RET
```

(8) On repart de la solution établie dans les séries d'exercices précédents en rappelant l'énoncé auquel le programme rédigé doit répondre : dans un tableau d'octet d'adresse et de taille données, calculer la longueur de la plus longue suite d'octets consécutifs identiques.

```
Hypothèses   R_array : contient l'adresse du tableau
             R_size : contient la taille du tableau

-------------------------------------------------------
init :       R_max ← 0
             R_currentMax ← 0
             R_inc ← 1
             ; Affectation de l'adresse du tableau :
             R_lastAdr ← R_array

loop :       JMP "end" if R_size == 0

             R_lastVal ← ptr R_lastAdr
             R_currentVal ← ptr R_array
             JMP "nextVal" if R_lastVal != R_currentVal

             R_currentMax ← R_currentMax + R_inc
             JMP "updateMax"

nextVal :    R_currentMax ← 1
             R_lastAdr ← R_array

updateMax :  JMP "nextIter" if R_max > R_currentMax
             R_max ← R_currentMax

nextIter :   R_array ← R_array + R_inc
             R_size ← R_size - R_inc
             JMP "loop"
end :
```

En se basant sur la discussion proposée lors de la rédaction de la solution proposée en pseudo-code, nous pouvons immédiatement traduire celle-ci en directives assembleur comme suit :

```
longest_seq:
              # RDI contient l'adresse du tableau
              # ESI contient la taille du tableau

              MOV RAX, 0
              MOV R12, 0

              CMP ESI, 0
              JLE end

              MOV R13B, byte ptr[RDI]
```

Les hypothèses considérées dans le pseudo-code sont résolues par la convention d'appel des fonctions : l'adresse du tableau est mis à disposition dans le registre de 64 bits `RDI`, et sa longueur dans le registre de 32 bits `ESI`. La convention d'appel nous indique aussi que la valeur de l'argument de sortie de la fonction doit se trouver dans le registre `RAX` ; on utilise donc ce registre dans notre programme pour contenir la longueur de la plus longue séquence trouvée dans le tableau analysé, de telle manière qu'à la fin du programme il n'y ait rien de plus à faire pour renvoyer l'argument de sortie.

Nous initialisons également les registres `R12` et `R13B` qui seront utilisés pour contenir respectivement la longueur de la séquence en cours d'évaluation et la valeur de l'octet précédent. Le registres qui est destiné à contenir un octet est lui-même la sous-parties d'un octet de son registre complet de 64 bit. On initialise tous ces registres à zéro sauf `R13B` ; celui-ci est fixé à la valeur de la première cellule du tableau afin d'amorcer le comptage de la première séquence. En effet, puisque les compteurs de longueur de séquence démarrent à 0, à la première itération, le programme se comportera comme si le premier octet poursuivait une séquence existante de taille nulle.

Puisqu'on déréférence un tableau, on doit d'abord vérifier qu'il n'est pas entièrement vide : c'est le but des opérations CMP et JLE qui comparent la taille du tableau passée en argument dans `ESI` à 0, et font sauter le programme à la fin de son exécution si c'est le cas. Si cela arrive, le programme renverra 0, ce qui correspond à un tableau dans lequel aucune séquence n'est trouvée.

```
iterate:
          CMP ESI, 0
          JLE end

          CMP R13B, byte ptr[RDI]
          JNE next_val

          INC R12
          JMP update_max
```

Comme dans la plupart des boucles réalisées jusqu'ici dans les exercices précédents, les opérateurs CMP et JLE (ou un autre type de saut conditionnel) sont utilisés comme gardiens de boucle. Dans ce cas-ci, on compare la longueur du tableau passée en argument dans ESI à 0, et si elle est nulle, on saute à la fin du programme. On comprend que dans ce cas-ci, ESI devra être décrémenté à chaque tour de boucle.

Le bloc d'instructions suivantes comprenant un CMP et un JNE réalisent l'opération de comparaison de la valeur courante avec la valeur précédente. Le CMP déréférence la valeur actuelle par adressage indirect utilisant le pointeur RDI puis la compare à la valeur précédente préalablement stockée dans le registre de travail R13B choisi arbitrairement. Dans le cas de la première itération, cette valeur précédente est artificiellement fixée à la valeur du premier caractère pour amorcer l'algorithme. Par la suite, elle devra être mise à jour pour que la comparaison puisse identifier les séquences se continuant et les interruptions entre deux séquences. Deux séquences sont définies par deux octets successifs différents ; dans ce cas, la condition correspondant au JNE est remplie et on saute à l'intitulé "next_val" qui accomplira les actions nécéssaires.

Si on ne trouve pas deux valeurs successives différentes, on évite le saut et on continue vers les deux dernières instructions. Celles-ci incrémentent le compteur de séquence courante incarné par le registre R12 choisi arbitrairement, puis sautent de manière inconditionnelle à la section "update_max" qui se chargera de mettre à jour le compteur de la séquence la plus longue si nécéssaire.

```
next_val:
            MOV R12, 1
            MOV R13B, byte ptr[RDI]
```

Si on arrive à cette section, c'est que le programme a trouvé deux octets successifs différents. Dans ce cas, on passe à une nouvelle séquence courante. Pour ce faire, on réinitialise le compteur de séquence courante R12 à 1 avec MOV et un littéral, puisqu'il y aura toujours au moins un octet dans une séquence, et on met à jour le octet courant en déréférençant la valeur actuelle par adressage indirect utilisant le pointeur RDI et en la placant dans R13B, puisqu'une nouvelle séquence correspond à un nouveau octet.

```
update_max:
                CMP RAX, R12
                JGE next
                MOV RAX, R12
```

On traverse cette section dans tous les cas d'exécution. On y compare d'abord avec CMP le compteur de la plus longue séquence dans RAX au compteur de la séquence courante dans R12. Si la séquence courante s'avère plus longue que la séquence originellement identifiée comme la plus longue, alors on doit la mettre à jour ; dans ce cas, on évite le saut JGE à la prochaine section du code, et on applique le MOV qui met à jour le compteur de séquence la plus longue en

transférant la valeur de R12 vers RAX. De cette manière, on s'assure que RAX contiendra toujours la bonne valeur de l'argument de sortie lorsqu'on atteindra la fin du programme.

```
next:
        INC RDI
        DEC ESI
        JMP iterate
```

Enfin, on rédige la fin de la boucle : on incrémente le pointeur sur le tableau dans RDI, on décrémente le registre ESI tel qu'il indiquera au cours du programme la longueur de tableau restante à analyser, et on saute de manière inconditionnelle à la section "iterate" où le gardien de boucle nous emmènera à la prochaine itération, ou a la fin du programme.

```
longest_seq:
              # RDI contient l'adresse du tableau
              # ESI contient la taille du tableau

              PUSH RBP
              MOV RBP, RSP
              PUSH R12
              PUSH R13
```

Puisqu'on utilise les registres R12 et R13, on les sauve sur la pile au début de l'exécution du programme afin de pouvoir les restaurer intact à la fin de l'appel à la fonction. On sauve également le contexte appellant et on définit le nouveau contexte avec les clauses consacrées impliquant RBP. Ce n'est pas explicitement nécéssaire dans le cas de ce programme, mais il s'agit d'une bonne habitude à prendre pour ne pas avoir de surprises avec des programmes plus complexes.

```
end:
       POP R13
       POP R12
       POP RBP

       RET
```

De manière correspondante, on dépile tous les registres empilés dans l'ordre inverse, et on achève enfin l'exécution de la fonction avec l'instruction RET. Comme on l'a expliqué dans les sections précédentes, le programme aura déjà placé la bonne valeur de l'argument de sortie dans RAX.

Au total, avec les clauses habituelles déclarant la syntaxe de programmation et la fonction avec son attribut "global", on a le programme suivant :

```
                .intel_syntax noprefix
                .text
                .global longest_seq
                .type longest_seq, @function

longest_seq:
                # RDI contient l'adresse du tableau
                # ESI contient la taille du tableau

                PUSH RBP
                MOV RBP, RSP
                PUSH R12
                PUSH R13

                MOV RAX, 0
                MOV R12, 0
                CMP ESI, 0
                JLE end
                MOV R13B, byte ptr[RDI]
iterate:
                CMP ESI, 0
                JLE end

                CMP R13B, byte ptr[RDI]
                JNE next_val

                INC R12
                JMP update_max
next_val:
                MOV R12, 1
                MOV R13B, byte ptr[RDI]
update_max:
                CMP RAX, R12
                JGE next
                MOV RAX, R12
next:
                INC RDI
                DEC ESI
                JMP iterate
end:
                POP R13
                POP R12
                POP RBP

                RET
```

(9) On repart de la solution établie dans les séries d'exercices précédents en rappelant l'énoncé auquel le programme rédigé doit répondre : déterminer si deux chaînes de caractères terminées par un zéro, d'adresses données, sont égales.

```
Hypothèses  R_str1 : adresse de la 1e chaine
            R_str2 : adresse de la 2e chaine

-----------------------------------------------
init :      R_inc ← 1
            R_eq ← 1

loop :      R_val1 ← ptr R_str1
            R_val2 ← ptr R_str2
            JMP "diff" if R_val1 != R_val2

            JMP "ret" if R_val1 == 0

            R_str1 ← R_str1 + R_inc
            R_str2 ← R_str2 + R_inc
            JMP "loop"
diff :      R_eq = 0

ret :
```

En se basant sur la discussion proposée lors de la rédaction de la solution proposée en pseudo-code, nous pouvons immédiatement traduire celle-ci en directives assembleur comme suit :

```
cmp_str :  # RDI : adresse de la 1e chaine
           # RSI : adresse de la 2e chaine

           MOV RAX, 1
```

Comme d'habitude, nos hypothèses sur la position des arguments d'entrée du programme sont résolues par la convention d'appel. Les deux arguments indiquant les adresses des deux chaînes à comparer doivent par convention être mis à disposition dans les registres `RDI` et `RSI`. Il s'agit bien de deux registres de 64 bits vu qu'on considère des adresses dans notre architecture x86_64. On intialise ensuite `RAX` à 1 comme on l'a fait pour *R_eq* dans le pseudo-code ; on choisit `RAX` car c'est aussi le registre qui par convention devra contenir l'argument de sortie de notre fonction. En l'occurence, il s'agit d'une valeur de vérité qui ne devra rester vraie au terme de l'exécution de la fonction que si les deux chaînes considérées sont parfaitement égales.

```
loop :  MOV R9B, byte ptr[RDI]
        CMP R9B, byte ptr[RSI]
        JNE diff
```

Si l'opérateur CMP accepte un adressage mémoire qui nous économise une des deux lignes de déréférencement obligatoire dans le pseudo-code, il n'en accepte pas deux ; on doit donc déréférencer RDI séparément par un MOV avant d'effectuer la comparaison entre la valeur déréférencée placée dans le registre de travail R9B choisi arbitrairement, et la valeur en mémoire à l'adresse pointée par la valeur contenue dans le registre RSI. Pour effectuer le saut conditionnel, on se base sur le CMP et on emploie JNE qui saute à l'adresse associée au label choisi si l'opérande de gauche de la comparaison est différente de l'opérande de droite.

On remarque qu'on a économisé une ligne de déréférencement, mais qu'on est contraint d'en ajouter une pour construire le saut conditionnel comme on l'a fait dans tous les exercices précédents, et qu'au total on en conserve donc le même nombre.

```
CMP R9B, 0
JE end

INC RDI
INC RSI
JMP loop
```

Comme pour le premier saut conditionnel, on doit développer l'unique ligne exprimée dans le pseudo-code en une opération CMP et un saut conditionnel JE. Grâce à ces deux opérations, on saute bien à la section associée au label "end" si on trouve que le contenu du registre R9B est égal à 0. On rappelle que ce registre a été pour la dernière fois modifié pour prendre la valeur pointée à l'adresse contenue dans RDI, soit l'adresse de la cellule courante du premier tableau évaluée pendant cette itération du programme, et que cette section n'est lue que si les deux caractères courants des deux chaînes ont préalablement été déterminés égaux. De cette manière, on garantit que le programme n'arrive à sa fin en ayant conservé une valeur de 1 dans RAX que si les deux chaînes sont égales en tous points et arrivent à leur fin signalée par le zéro de terminaison.

Si on ne saute pas à la fin du programme parce que les chaînes se terminent, on incrémente simplement les deux pointeurs sur les deux tableaux avec l'opérateur INC et on saute de manière inconditionnelle au label associé au début de la boucle pour entamer la prochaine itération de celle-ci avec JMP.

```
diff :  MOV RAX, 0

end :   RET
```

Enfin, on rédige la section "diff" qui fixe la valeur de RAX à 0 si l'on rencontre la moindre différence dans les deux chaînes que l'on évalue, et la section "end" qui sera atteinte dans tous les cas pour mettre fin au programme. On se rappelle que RAX n'aura conservé sa valeur de 1 que si aucune différence n'est trouvée dans les deux chaînes, de telle manière qu'au RET la bonne valeur de l'argument de sortie est bien mis à disposition dans le registre RAX.

Au total, avec les clauses habituelles déclarant la syntaxe de programmation et la fonction avec son attribut "global", on a le programme suivant :

```
            .intel_syntax noprefix
            .text
            .global cmp_str
            .type cmp_str, @function

cmp_str :   # RDI : adresse de la 1e chaine
            # RSI : adresse de la 2e chaine

            MOV RAX, 1

loop :      MOV R9B, byte ptr[RDI]
            CMP R9B, byte ptr[RSI]
            JNE diff

            CMP R9B, 0
            JE end

            INC RDI
            INC RSI
            JMP loop

diff :      MOV RAX, 0

end :       RET
```